THÈSE DE DOCTORAT DE
L'UNIVERSITÉ PIERRE ET MARIE CURIE
Spécialité Informatique
(École doctorale informatique, télécommunications et électronique)

Présentée par
M. DARRASSE Alexis

Pour obtenir le grade de
DOCTEUR DE L'UNIVERSITÉ PIERRE ET MARIE CURIE

Structures arborescentes complexes :
analyse combinatoire, génération aléatoire et applications

Soutenue le 26 janvier 2010
devant le jury composé de :

M. CHYZAK Frédéric
M. DUCHON Philippe
M. FLAJOLET Philippe
Mme KANG Mihyun
M. LATAPY Matthieu
M. QUEINNEC Christian
M. SCHAEFFER Gilles (rapporteur)
Mme SORIA Michèle (directrice)

et après avis de :

M. HWANG Hsien-Kuei
M. SCHAEFFER Gilles

À Cyrille

Ευχαριστίες

Κλείνοντας αυτό το μεγάλο κεφάλαιο που λέγεται σπουδές θα ήθελα να ευχαριστίσω την οικογένειά μου που με άφησε να χαράξω μόνος μου την πορεία μου και που μου έδωσε όλα τα εφόδια για να την πραγματοποιήσω.

Ο μακρύς αυτός δρόμος ήταν γεμάτος περιπέτειες, γεμάτος γνώσεις, χάρη στους δασκάλους, συμμαθητές, καθηγητές, συμφοιτητές και συναδέλφους που είχα την τύχη να συναντήσω. Τους ευχαριστώ όλους και ιδια΄τερα την Michèle Soria που με καθοδήγησε χωρίς οικονομία δυνάμεων και με απίστευτη κατανόηση.

Τέλος, θα ήθελα να κάνω ιδιαίτερη μνεία στην υπέροχη σύζηγό μου που έχει βρει τον τρόπο για να με επαναφέρει στην πραγματικότητα όταν χρειάζεται.

Résumé

Ce travail a pour but l'application des techniques d'analyse et de génération aléatoire de la combinatoire analytique à des problèmes issus de domaines variés où apparaissent des structures arborescentes.

Dans un premier temps, nous étudions les propriétés des *k*-arbres, une famille de graphes qui généralise celle des arbres. Les *k*-arbres jouent un rôle central en algorithmique des graphes et apparaissent aussi (sous le nom de réseaux apolloniens aléatoires ou triangulations en pile) comme modèle pour les graphes de terrain. Notre contribution consiste en une bijection entre *k*-arbres et une famille simple d'arbres et son utilisation pour analyser certaines propriétés des *k*-arbres sous la distribution uniforme : distribution des degrés en loi de puissance avec chute exponentielle, distance moyenne en racine carrée de la taille et profil qui suit une loi de Rayleigh.

La seconde partie a pour objet la génération aléatoire de structures arborescentes avec la méthode de Boltzmann. Nous avons mis en place un cadre générique et efficace que nous avons appliqué à la génération de données issues de plusieurs domaines logiciels : instances de types de données algébriques, instances de méta-modèles et de documents XML selon une grammaire. La complexité linéaire des algorithmes de génération rend ces outils bien adaptés aux tests de robustesse et de performance.

Mots clés : combinatoire analytique, k-arbres, génération de Boltzmann, test logiciel, types de données algébriques, méta-modèles, XML.

Ces travaux ont été menés au laboratoire d'informatique de Paris 6 (LIP6) — UMR 7606, 104 avenue du président Kennedy, 75016 Paris.

Complex tree-like structures:
combinatorial analysis, random sampling and applications

Abstract

The goal of this thesis is the application of the analysis and random sampling techniques of analytic combinatorics on problems concerning tree-like structures and coming from a wide range of domains.

First we study the properties of k-trees, a class of graphs that extends the class of trees. The class of k-trees is central in the domain of graph algorithms and also appears (under the name of random Apollonian networks or stack triangulations) as a model for "real-world" graphs. Our contribution is a bijection between k-trees and a simple family of trees and its use for describing some properties of k-trees under the uniform distribution: the degree distribution follows a power law with an exponential cutoff, the mean distance is proportional to the square root of the size and the profile follows a Rayleigh distribution.

The second part focuses on random sampling of tree-like structures using the Boltzmann method. We developped a generic and efficient framework and we used it for sampling data respecting an algebraic data type, instances of meta-models and XML documents following a grammar. Thanks to the linear complexity of the sampling algorithms, these tools are well suited for robustness testing and benchmarking.

Keywords: analytic combinatorics, k-trees, Boltzmann sampling, software testing, algebraic data types, meta-models, XML.

Table des matières

1 Présentation générale **11**

1.1 Contexte . . . 11
1.2 Les k-arbres . . . 12
1.3 Génération aléatoire d'arbres de terrain . . . 15
1.4 Plan détaillé . . . 18

2 Les méthodes **19**

2.1 Combinatoire analytique . . . 19
2.2 Arbres aléatoires . . . 22
2.3 Génération de Boltzmann d'arbres . . . 25

I Analyse des k-arbres **31**

3 Introduction **33**

3.1 Les modèles de graphes aléatoires . . . 33
3.2 Les propriétés étudiées . . . 34
3.3 Les k-arbres . . . 37
3.4 Nos résultats : la bijection . . . 41
3.5 Estimation des paramètres . . . 45

4 Degree distribution of RANS and Boltzmann sampling **51**

4.1 Introduction . . . 51
4.2 Random Apollonian network structures . . . 53
4.3 Boltzmann sampling . . . 55
4.4 Degree distribution in RANS . . . 58
4.5 Extensions . . . 61

5 Distances in random Apollonian network structures **63**

5.1 Introduction . . . 63
5.2 Random Apollonian network structures . . . 64
5.3 Distance from an outermost vertex . . . 65
5.4 Total distance between pairs of vertices . . . 69

6 Limiting Distribution for Distances in k-trees 75
6.1 Introduction . . . 75
6.2 Structures: k-trees and class $\mathcal{K}$ . . . 76
6.3 Distance to the root . . . 79
6.4 Distances to a random vertex . . . 83

7 Analysis of Parameters in k-trees 87
7.1 Introduction . . . 87
7.2 The bijection . . . 88
7.3 Algorithm for distances . . . 90
7.4 Degree distribution . . . 95
7.5 Calculation of the profile . . . 102

II Applications de la génération d'arbres 107

8 Introduction 109
8.1 Domaines d'application . . . 110
8.2 Algorithmes . . . 113
8.3 Réalisation . . . 117
8.4 Traduction . . . 118

9 Random generation for automated testing in O'Caml 121
9.1 Introduction . . . 121
9.2 Context . . . 123
9.3 Underlying theory : the Boltzmann model . . . 125
9.4 Application to O'Caml . . . 130
9.5 Demonstration . . . 132
9.6 Performance . . . 136
9.7 Future works and other applications . . . 137
9.8 Conclusion . . . 138

10 Random generation of metamodel instances 139
10.1 Introduction . . . 139
10.2 Boltzmann random generation of trees . . . 140
10.3 Model generation based on meta model specification . . . 146
10.4 Validation . . . 148
10.5 Related works . . . 152
10.6 Conclusion . . . 153

11 Random XML sampling the Boltzmann way 155
11.1 Introduction . . . 155
11.2 Translating the RELAX NG . . . 156
11.3 Solving the system of equations . . . 157
11.4 Generating XML documents . . . 158

11.5 Related work and work in progress 159

Conclusion **161**

Bibliographie **163**

Chapitre 1

Présentation générale

Cette thèse porte sur l'étude de structures arborescentes complexes, issues de domaines applicatifs réels, tels que les graphes de terrain ou le test logiciel. Les méthodes utilisées reposent sur des bijections combinatoires et de l'analyse complexe. La combinatoire analytique permet non seulement d'analyser des algorithmes en moyenne et en distribution mais aussi, via la méthode de Boltzmann de générer aléatoirement des structures de façon très efficace.

1.1 Contexte

Que ce soit pour classifier les livres d'une bibliothèque, représenter l'évolution des espèces ou décomposer la structure d'une phrase, on modélise couramment ces structures hiérarchiques par des arbres. Un arbre peut être vu comme un nœud, la racine, auquel sont accrochés ses fils, qui sont eux-mêmes des arbres. Certaines familles d'arbres sont spécifiées par des grammaires qui contraignent cette construction récursive. Par exemple, l'ensemble des phrases syntaxiquement correctes en français est une famille respectant certaines règles.

Étant donnée une famille d'arbres, nous sommes intéressés à estimer les propriétés des représentants typiques de cette famille, pour les comparer aux propriétés des objets que ces arbres sont censés modéliser. Par exemple, la hauteur d'un arbre peut varier beaucoup d'une famille à l'autre ; un arbre binaire de recherche est très tassé tandis qu'un arbre de termes est plus filiforme.

Dans cette thèse, nous utilisons des arbres pour modéliser certaines familles de graphes, ce qui permet d'étudier leur propriétés de ces graphes dans le contexte de l'analyse des arbres. Les méthodes que nous utilisons sont essentiellement issues de la combinatoire analytique : outil séries génératrices dans le contexte algébrique de la méthode symbolique et dans le contexte analytique de l'analyse complexe et des théorèmes limites des probabilités.

La génération aléatoire de structures combinatoires est un domaine très riche et ancien de la combinatoire algorithmique. L'introduction récente d'approches méthodologiques de portée générale qui permettent d'engendrer en temps linéaire des objets complexes lui donnent un nouvel essor. Ceci donne la possibilité d'aborder des domaines d'applications où le passage

à l'échelle est indispensable pour une génération automatique intensive (graphes de plusieurs millions de sommets par exemple).

Nous utilisons le modèle de Boltzmann, issu de la combinatoire analytique, pour générer des structures arborescentes qui simulent des structures plus complexes de l'ingénierie des modèles, des différents domaines de l'informatique qui utilisent le format XML pour représenter leurs données et de la programmation fonctionnelle.

1.2 Les *k*-arbres

La classe des k-arbres est une famille de graphes largement étudiée d'un point de vue à la fois combinatoire et algorithmique. Notre intérêt pour cette classe vient de la modélisation de graphes de terrain, avec l'objectif d'analyser des propriétés telles que le degré d'un sommet ou la distance entre deux sommets. Le point d'entrée pour cette étude est une bijection qui transforme les k-arbres en structures arborescentes, sur lesquelles les propriétés intéressantes peuvent être analysées par les méthodes de combinatoire analytique.

1.2.1 Famille de graphes représentables par des arbres

Les k-arbres et les autres classes de graphes que nous considérons ont une définition récursive simple, ce qui permet de les décomposer et donc de les représenter par des arbres. Cette simplicité structurelle a fait que les k-arbres apparaissent de manière indépendante dans différents domaines.

L'instance que nous avons rencontrée en premier lieu s'appelle *réseaux apolloniens aléatoires*, ou RAN pour *random Apollonian network*. Un RAN est soit le triangle (1,2,3), soit un RAN à n sommets dans lequel on ajoute un nouveau sommet étiqueté par $n+1$ et relié aux trois sommets d'un triangle choisi au hasard parmi les $3(n-3)+1$ triangles du RAN initial (voir figure 1.1).

Ces triangulations particulières ont été proposées récemment par Zhou et al. [ZYW05] comme un modèle intéressant pour simuler des graphes de terrain parce qu'ils ont les mêmes propriétés que ces derniers : distance moyenne logarithmique, distribution de degrés en loi de puissance, grand coefficient de clustering et clustering inversement proportionnel au degré.

Les 2-arbres, classe dont la définition est similaire à celle des RAN, ont été introduits en combinatoire et énumérés par Harary et Palmer [HP68]. Leur généralisation aux dimensions supérieures, les k-arbres, ont fait l'objet de nombreux travaux aux alentours de 1970.

La différence la plus importante entre k-arbres et RAN est que les étiquettes des sommets d'un RAN retracent l'historique de la construction itérative du graphe, ce qui n'est pas le cas dans un k-arbre. Nous avons aussi considéré des RAN sans contrainte sur les étiquettes, que nous avons appelé *structures de réseaux apolloniens aléatoires* (RANS).

Les k-arbres jouent un rôle important dans l'algorithmique des graphes, depuis que Arnborg et Proskurowski [AP89] ont montré qu'un grand nombre d'algorithmes NP-complets de-

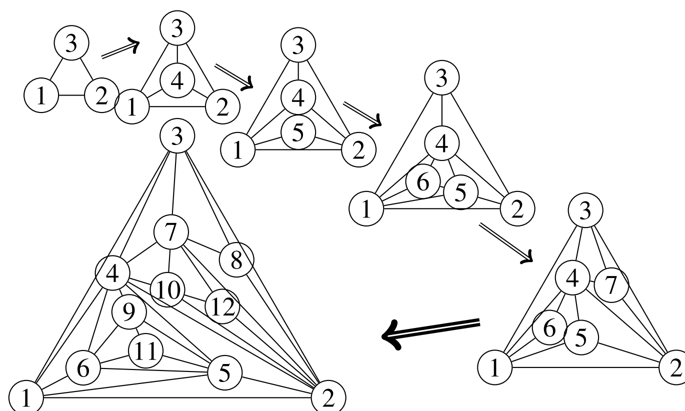


FIGURE 1.1 – Un réseau apollonien aléatoire et les premières étapes de sa construction.

viennent polynomiaux si on se restreint aux k-arbres partiels.

1.2.2 Représentation arborescente des k-arbres

Le point clé de notre étude est d'étudier une classe d'arbres qui représente ces graphes, plutôt que les graphes eux-mêmes. C'est la nature récursive des graphes qui rend possible cette représentation, qui elle-même permet l'application de méthodes d'analyse bien établies et très efficaces. Notons que d'autres classes de graphes sont en bijection avec des structures arborescentes, comme les cartes planaires [Sch98] ; la particularité des graphes que nous étudions ici est qu'ils ont des propriétés qui sont plus proches de celles des arbres que de celles des graphes.

Il est facile de créer des arbres qui codent les k-arbres. Il faut faire un peu plus attention si l'on veut que cette classe d'arbres soit *facile à décrire* et *en bijection* avec la classe des k-arbres. De plus, comme notre but est d'étudier les propriétés des k-arbres, il faut que ces propriétés soit lisibles sur les arbres qui les représentent.

Un autre avantage de la représentation arborescente est qu'il est facile de les générer aléatoirement. La simulation a joué un rôle primordial dans nos études. Elle a permis à la fois de préparer l'étude théorique en observant le comportement des différents paramètres et de donner une assurance supplémentaire pour la validité des calculs.

Notre contribution combinatoire est un algorithme qui construit, à partir d'un k-arbre, un arbre sur lequel il est facile de voir les voisins de chaque sommet du graphe. En étudiant la classe d'arbres produits par cet algorithme, nous avons ensuite réalisé l'étude analytique de certains paramètres des k-arbres : distribution de degrés, distance moyenne entre sommets, distribution des distances entre sommets.

1.2.3 Deux types de comportement

Les différentes classes de graphes que nous avons étudiées sont en fait toutes des sous-classes des k-arbres. Elle présentent deux types de comportement vis-à-vis des propriétés qui

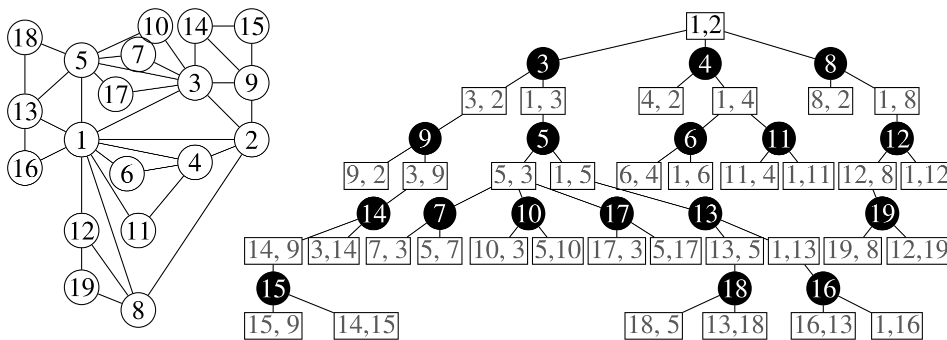


FIGURE 1.2 – Un 2-arbre et l'arbre correspondant.

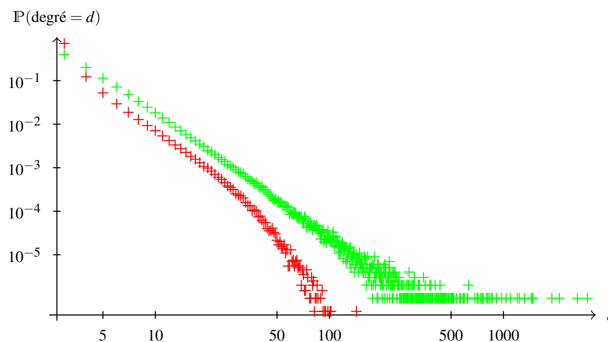


FIGURE 1.3 – La distribution de degrés dans un 3-arbre de taille 2 millions et d'un RAN de taille 1 million.

nous ont intéressées. Ces différents comportements reflètent la nature différente des arbres qui sont mis en bijection avec les deux classes : arbres uniformes pour les k-arbres et arbres croissants pour les RAN. Les autres classes issues des k-arbres ont des propriétés de même nature que l'une ou l'autre de ces deux classes types ; par exemple les RANS se comportent comme des k-arbres.

Pour les RAN on connaissait déjà la distance moyenne logarithmique et la distribution de degrés avec une chute polynomiale. Notre étude met en évidence un comportement tout à fait différent pour la classe générale des k-arbres : une distance moyenne en racine carrée et une distribution de degrés avec une chute exponentielle. Plus précisément nous avons obtenu les résultats suivants :

- la probabilité que le degré d'un sommet soit égal à $d+k$ est proportionnelle à $c_k^d d^{-\frac{3}{2}}$, avec $c_k = e^{\frac{1}{k}} \frac{k-1}{k}$;
- si on choisit deux sommets au hasard, la probabilité que leur distance soit égale à $\lceil x\sqrt{n} \rceil$ est proportionnelle à $xe^{-\frac{(h_k x)^2}{2}}$, où n est la taille du k-arbre et h_k le k-ième nombre harmonique.

1.2.4 Méthodes

Notre étude des k-arbres consiste en deux étapes successives utilisant chacune des méthodes différentes : la combinatoire bijective puis la combinatoire analytique.

Tout d'abord, nous mettons en place une bijection entre k-arbres et une famille $\mathcal{T}$ d'arbres. La particularité de cette bijection est qu'elle permet de traduire la notion de plus court chemin dans le graphe en notion de chemin dans l'arbre. Plus précisément, fixant un sommet v du graphe, on construit un arbre qui met en évidence les plus courts chemins de v aux autres sommets du graphe.

Nous utilisons ensuite la méthode symbolique pour transformer les descriptions combinatoires des arbres obtenus en spécifications algébriques, définissant des séries génératrices multivariées ; la combinatoire analytique vient enfin pour donner les outils d'analyse de ces séries. Notre étude, bien que s'apparentant à celle des arbres aléatoires, présente une difficulté supplémentaire provenant de la complexité des spécifications de nos structures combinatoires. Nous illustrons ainsi l'applicabilité de ces méthodes à des structures plus complexes, venant des graphes de terrain ou de l'algorithmique des graphes.

Ce travail s'est accompagné de simulations, à la fois pour créer des intuitions sur les objets et pour valider les résultats sur les propriétés. Pour cette étape de simulation, il s'agit de réaliser rapidement des implantations qui soient des prototypes, contrairement à ce qui sera fait dans la seconde partie où il s'agit de développement classique. Pour obtenir cette agilité dans l'implantation, le point clé est le découpage du problème en unités qui sont chacune traitée par les outils les mieux adaptés : système de calcul formel pour les précalculs, générateurs développés à dessein pour la génération aléatoire, logiciels de statistique pour la manipulation des données et la visualisation des propriétés.

1.3 Génération aléatoire d'arbres de terrain

L'objectif premier de la génération aléatoire est de créer de manière efficace des ensembles de données avec des aspects variés. Pour pouvoir travailler rigoureusement avec ces données il faut de plus connaître la distribution de probabilités sous-jacente à la génération, c'est à dire savoir, pour chaque objet quelle est sa probabilité d'apparaître. Lorsque tous les objets ont la même probabilité d'apparaître on parle de distribution uniforme et c'est dans ce cadre que nous nous plaçons pour générer les structures arborescentes qui approximent les structures des domaines applicatifs considérés.

Nous utilisons la méthode de Boltzmann qui permet de générer en temps linéaire des arbres dans une fenêtre de taille. Nous proposons trois applications pour lesquelles nous avons développé et appliqué un cadre unifié d'implantation de la génération aléatoire efficace de structures arborescentes complexes.

1.3.1 Modèle de Boltzmann

Le modèle de Boltzmann nous permet, étant donnée la spécification combinatoire d'une classe d'arbres d'obtenir automatiquement un générateur aléatoire de structures de cette classe.

Le générateur a la propriété d'uniformité : deux structures de même taille ont la même probabilité d'être générées. De plus, le temps nécessaire à générer un objet est proportionnel à sa taille. Nous avons donc une méthode automatique, rigoureuse et efficace.

Une particularité de la génération de Boltzmann est que la taille des objets produits est aléatoire, avec une distribution qui dépend non seulement de la spécification mais aussi d'un paramètre donné en entrée. On peut choisir ce paramètre de manière à obtenir une structure d'une taille donnée en un temps proportionnel à cette taille.

Le modèle de Boltzmann est aussi caractérisé par la simplicité des algorithmes de génération ; tous les calculs non-triviaux dépendent de la spécification et non de la taille des objets générés. Ce sont des précalculs qui sont effectués dans un programme auxiliaire, l'oracle. Le générateur fait des tirages aléatoires qui orientent la production de la structure aléatoire alors que l'oracle calcule les paramètres des tirages aléatoires.

Dans cette thèse nous nous intéressons au cadre restreint des arbres, dont les spécifications utilisent seulement le produit, la somme et la récursion. Dans ce cas, les précalculs sont simplifiés, il s'agit de trouver une solution particulière (la seule qui a un sens combinatoire) d'un système algébrique. L'implémentation de l'oracle devient alors possible sans avoir recours à un système de calcul formel et on peut, sous certaines conditions, prévoir tous les appels à l'oracle en avance et ainsi faire un seul précalcul par spécification combinatoire, indépendamment de la taille visée.

1.3.2 Implémentation

Pour rendre l'utilisation de la génération de Boltzmann plus facile dans les applications qui utilisent des structures arborescentes nous avons mis en place un cadre générique pour produire automatiquement, rigoureusement et efficacement des arbres. La figure 1.4 montre la structure de ce cadre.

- L'oracle est un programme qui prend en entrée une spécification combinatoire et une taille visée, et produit en sortie les paramètres nécessaires pour la génération (valeur du paramètre de Boltzmann et valeur de séries génératrices en ce point).
- Le compilateur prend une spécification combinatoire et produit un générateur qui lui même utilise le résultat de l'oracle pour effectuer la génération.

Pour utiliser ce cadre dans une application donnée, il suffit de donner deux traducteurs :

- un traducteur pour aller des spécifications de l'application vers les spécifications combinatoires et
- un traducteur pour aller des objets combinatoires générés vers les objets de l'application.

Nous avons réalisé un oracle sous la forme d'une librairie en C, ainsi que des interfaces O'Caml et Haskell pour cette librairie. La réalisation d'un compilateur générique est en cours, mais dans le cadre de nos applications nous avons préféré, pour plus d'efficacité, réaliser des compilateurs génériques qui se passent de traducteurs.

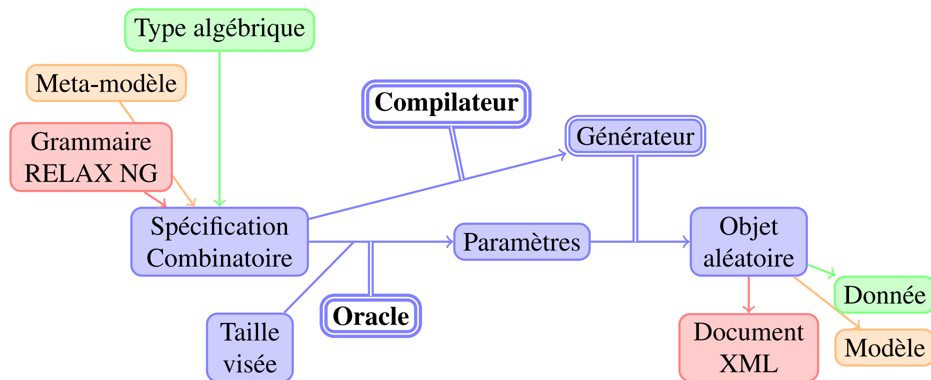


FIGURE 1.4 – Articulation de notre cadre de génération aléatoire de structures arborescentes.

1.3.3 Applications

On a utilisé ce cadre générique dans trois applications différentes : types de données algébriques, méta-modèles et grammaires XML. Il s'agit de domaines qui ne font pas partie des domaines d'application habituels de la combinatoire, mais dans lesquels des structures arborescentes définies par une grammaire jouent un rôle important. Le travail à accomplir consiste en une traduction de la grammaire du domaine vers les spécifications combinatoires d'arbres, ce qui soulève à chaque fois des difficultés différentes.

Types de données algébriques. Les types de données algébriques sont au cœur de tout langage fonctionnel fortement typé, comme O'Caml ou Haskell. Notre méthode s'applique particulièrement bien dans ce contexte car la traduction vers les spécifications combinatoires est naturelle. Nous pensons que ce type de génération peut compléter les outils de test automatique et être utile pour l'automatisation des tests de performance.

Méta-modèles. L'ingénierie dirigée par les modèles est basée sur des outils qui manipulent les modèles. Cependant les outils actuels ne passent souvent pas à l'échelle, et en même temps il n'existe de méthode établie pour tester leur robustesse. Avec notre méthode on a créé un prototype qui génère très rapidement des modèles excédant les capacités des outils actuels.

Un modèle n'est cependant pas un arbre mais un graphe. Pourtant, dans beaucoup de cas, notamment celui des diagrammes UML, le modèle est composé d'une structure arborescente enrichie de quelques liens. On fait donc une génération aléatoire uniforme de la structure arborescente et on utilise des méthodes *ad hoc* proches de celles déjà existantes pour compléter le modèle.

Grammaires XML. Un document XML est un arbre avec des données potentiellement complexes sur les noeuds. Notre application traduit la grammaire définissant une famille de documents XML en une spécification combinatoire qui tient compte de la structure arborescente. L'approche est suffisamment efficace pour traiter en quelques minutes des grammaires de très grandes tailles, telle que OpenDocument qui contient 500 règles. En couplant ce générateur de

squelettes à un générateur de valeurs pour les feuilles on obtient un générateur très efficace de documents XML.

1.4 Plan détaillé

Le chapitre 2 donne un aperçu de differents aspects de la théorie sur lesquels sont basés nos travaux : la combinatoire analytique, les arbres aléatoires et la génération de Boltzmann.

Le reste de ce mémoire est separé en deux parties, la première (chapitres 3-7) dediée à l'étude combinatoire des paramètres des k-arbres et la seconde (chapitres 8-11) aux applications de la génération aléatoire d'arbres. Dans chacune de ces deux parties on a présenté dans un chapitre introductif le domaine les méthodes et les résultats, puis on a inséré les différentes communications liées à ces parties.

Première partie. Dans le chapitre 3 nous présentons le domaine des graphes aléatoires avant d'exposer nos résultats : la bijection entre k-arbres et une famille simple d'arbres et son utilisation pour estimer différents paramètres des k-arbres.

Les autres chapitres de cette partie sont des reproductions de nos communications introduisant ces résultats. Le chapitre 4 [DS07] montre la bijection entre RANS et arbres ternaires et l'estimation de la distribution de degrés des RANS. Le chapitre 5 [BDS08] s'intéresse à la distance moyenne entre deux sommets quelconques et la distribution des distances à un sommet particulier d'un RANS. Dans le chapitre 6 [DS09] on étend la bijection aux k-arbres et on étudie la distribution des distances entre deux sommets quelconques. Le chapitre 7 [DS] reprend l'analyse des degrés et des distances dans les k-arbres avec une approche combinatoire unifiant les modèles uniforme et croissant.

Deuxième partie. Le chapitre 8 présente brièvement les domaines pour lesquels nous avons mis en place la génération aléatoire d'arbres, et détaille la partie générique de notre méthode.

Les derniers chapitres sont des reproductions de nos communications destinées à présenter nos résultats aux communautés des différents domaines d'application ; les types de données algébriques dans le langage O'Caml dans le chapitre 9 [CD09], les meta-modèles dans le chapitre 10 [MDBS09] et enfin les documents XML pour le chapitre 11 [Dar08].

Chapitre 2

Les méthodes

Ce chapitre présente une synthèse rapide des différentes méthodes et modèles sur lesquels notre travail est fondé. La combinatoire analytique, qui permet de traduire les spécifications combinatoires en équations fonctionnelles sur les séries génératrices et d'obtenir une estimation asymptotique de ces séries. Ensuite, les différents modèles d'arbres aléatoires étiquetés, croissant ou uniforme. Et enfin la méthode de Boltzmann pour générer des arbres aléatoires.

2.1 Combinatoire analytique

La combinatoire analytique s'intéresse à l'étude des propriétés asymptotiques des objets combinatoires et utilise pour ce faire des outils puissants de l'analyse complexe. Cette section a pour objet la présentation succincte de quelques concepts fondamentaux de la combinatoire analytique qui seront extensivement utilisés dans cette thèse. Ces concepts sont brillamment exposés dans le livre de référence du domaine, « Analytic Combinatorics » de Philippe Flajolet et Robert Sedgewick [FS09].

2.1.1 Classes combinatoires

Au cœur de la combinatoire est la question de denombrement : combien y a-t-il de façons de combiner n atomes en respectant certaines règles. Par exemple, de combien de manières peut-on ordonner les entiers de 1 à n ? Ou combien existe-t-il de façons pour couper un n-gône en triangles ? [1]

Pour procéder à une étude systématique on a d'abord besoin d'une définition de l'objet de notre étude, c'est à dire les classes combinatoires. Nous utiliserons ici celle donnée par Flajolet et Sedgewick [FS09, p.16].

Définition Une classe combinatoire est un ensemble fini ou dénombrable sur lequel une fonction de taille est définie et qui satisfait les deux conditions suivantes :

1. La réponse à la première question, $n!$, est connue depuis des millénaires. La deuxième question, pour laquelle Euler a trouvé la réponse $\frac{(2n)!}{(n+1)!n!}$ aujourd'hui connue sous le nom de nombres de Catalan, reviendra dans le chapitre 3.

1. la taille d'un élément est un entier positif,
2. le nombre d'éléments d'une taille donnée est fini.

Cette définition laisse certains degrés de libertés, elle ne dit par exemple rien sur la nature des atomes qui composent les éléments d'une classe. Dans le cadre de cette thèse nous nous intéressons uniquement aux classes combinatoires étiquetées, c'est à dire construites à partir d'atomes identifiés de manière unique par une étiquette (on utilise une permutations des entiers de 1 à n pour étiqueter les atomes d'un objet de taille n). Pour modéliser des objets issus de l'informatique il est naturel d'utiliser des classes étiquetées car les adresses mémoire peuvent être interprétées comme des étiquettes uniques on peut aussi, d'un point de vue combinatoire, considérer des classes non-étiquetées nécessitent la prise en compte des symétries des structures, augmentant ainsi la difficulté de leur étude.

Étant donnée une classe combinatoire $\mathcal{C}$ on cherche d'abord à connaître la suite C_n du nombre d'éléments de taille n, soit de façon exacte, soit de façon asymptotique ce qui qui permet de donner la vitesse de croissance de cette suite. Pour étudier la suite C_n dans le cadre des objets étiquetés on utilise sa série génératrice exponentielle $C(z) = \sum_{n=0}^{\infty} C_n z^n/n!$.

La puissance de la combinatoire analytique pour estimer la suite C_n a deux sources, l'une algébrique et l'autre analytique, qui font l'objet des deux sections suivantes.

Premièrement, il est facile de se donner un ensemble de règles pour définir des classes combinatoires et traduire facilement ces règles en équations sur les séries génératrices : c'est la *méthode symbolique*.

Deuxièmement, les méthodes de l'analyse complexe permettent d'obtenir des résultats d'estimation très précis sur la suite C_n en étudiant les singularités de la fonction $C(z)$: c'est l'*asymptotique des coefficients*.

2.1.2 Méthode symbolique

La méthode symbolique [FS09, partie A] décrit des constructions combinatoires qui se traduisent systématiquement en fonctions sur les séries génératrices de dénombrement. On a ainsi un langage qui permet de raisonner de manière formelle sur les structures combinatoires et, quand il faut quantifier les propriétés des classes combinatoires, on travaille directement sur les séries génératrices correspondantes.

Pour les classes que nous allons étudier (arbres étiquetés), on n'utilise qu'une partie du langage de la méthode symbolique : on autorise des spécifications éventuellement récursives, qui sont définies à partir de deux atomes et des quatre opérateurs union, produit étiqueté, sequence et ensemble.

Les atomes sont $\mathcal{E}$, qui a taille zéro, et $\mathcal{Z}$, qui est l'unité de taille. Les opérateurs binaires sont l'union notée $\cup$ et le produit étiqueté noté $\star$; la taille d'un objet dans $\mathcal{A} \cup \mathcal{B}$ est héritée de l'ensemble d'origine de l'objet, et la taille d'un couple de $(a,b) \in \mathcal{A} \star \mathcal{B}$ est la somme des tailles de ses composantes a et b. On a aussi deux opérateurs unaires : la séquence (suite), notée Seq, et l'ensemble, noté Set, spécifiant des ensembles respectivement ordonnés ou non-ordonnés d'objets ; la taille d'un tel ensemble est la somme des tailles de ses composantes.

Par exemple les mots sur un alphabet binaire $(0,1)$ sont décrits par la spécification Seq$(\mathcal{Z} \cup \mathcal{Z})$ qui correspond à la série génératrice $\frac{1}{1-2z}$.

Le tableau suivant présente la traduction des spécifications en équations sur les séries génératrices.

Spécification	$\mathcal{E}$	$\mathcal{Z}$	$\mathcal{A} \cup \mathcal{B}$	$\mathcal{A} \star \mathcal{B}$	Seq$(\mathcal{A})$	Set$(\mathcal{A})$
Série génératrice	1	z	$A(z)+B(z)$	$A(z) \cdot B(z)$	$\frac{1}{1-A(z)}$	$\exp(A(z))$

La méthode symbolique contient d'autres opérateurs sur les classes combinatoires. Citons par exemple la construction $\mathcal{Z}^{\square} \star \mathcal{C}$ qui décrit le produit de l'atome portant l'étiquette 1 avec un objet de la classe $\mathcal{C}$ et que l'on utilise pour spécifier les structures croissantes. Il s'agit d'un cas particulier d'utilisation de l'opérateur *boîte* ($^{\square}$) [Gre91] qui permet d'exprimer des contraintes d'ordre dans les étiquettes. La classe $\mathcal{Z}^{\square} \star \mathcal{C}$ a comme série génératrice $\int_0^z C(t)dt$.

La méthode symbolique aide aussi à estimer, à partir de la spécification combinatoire, la proportion d'atomes vérifiant certaines propriétés dans les objets d'une classe. Pour ce faire on traduit la spécification de la classe, augmentée de « marqueurs » indiquant les atomes à compter, en séries bivariées. On étudie alors la série $C(z,u) = \sum_{n=0}^{\infty} C_{n,k} u^k z^n / n!$, avec $C_{n,k}$ le nombre d'objets de taille n avec k atomes marqués. Par exemple la série génératrice bivariée où on marque le nombre de 0 dans un mot binaire est $\frac{1}{1-z-uz}$.

2.1.3 Asymptotique des coefficients

La série génératrice $C(z)$ d'une classe combinatoire $\mathcal{C}$ est une série entière qui converge à l'intérieur d'un disque de rayon $R \geq 0$ centré à l'origine. Dans le cadre de la méthode symbolique, les séries obtenues ont un *rayon de convergence* R strictement positif qui donne le taux de croissance exponentiel de la suite C_n :

$$C_n = R^{-n} \theta(n) \text{ avec } \theta(n) = o((1+\eta)^n), \forall \eta > 0.$$

La nature du facteur sous-exponentiel $\theta(n)$ dépend de la nature du développement singulier de la fonction $C(z)$ au voisinage de sa singularité dominante (qui est toujours sur l'axe réel positif pour des séries à coefficients positifs, par le théorème de Pringsheim [FS09, théorème VI.6]).

Si $C(z)$ est une fraction rationnelle alors $\theta(n)$ est un polynôme. La décomposition en éléments simples fait apparaître un pôle dominant de multiplicité q en R et le degré de $\theta(n)$ est $q-1$ (théorème des résidus de Cauchy [FS09, théorème IV.3]).

Si $C(z)$ est équivalente autour de sa singularité dominante à $(1-z)^{-\alpha}$ avec α strictement positif ou non entier, alors $\theta(n)$ est de l'ordre de $n^{\alpha-1}$. Plus précisément :

Théorème 2.1.1 (Analyse de singularités). *[FS09, Theorem VI.4] Sous des conditions de prolongement analytique dans un Δ-domaine à la singularité, une approximation de la fonction C*

$$C(z) \sim_{z \to R} (1-z)^{-\alpha}, \qquad \alpha \in \mathbb{R} \setminus (\{0\} \cup \mathbb{Z}^-)$$

se « transfère » en une estimation de la croissance sous-exponentielle des coefficients

$$\theta(n) \sim_{n\to\infty} \frac{n^{\alpha-1}}{\Gamma(\alpha)}.$$

Dans le cas où la fonction $C(z)$ a un rayon de convergence infini, ou que la croissance de $C(z)$ quand elle s'approche de sa singularité dominante est exponentielle, et en particulier pour les fonctions H-admissibles [FS09, section VIII.5], la méthode qui s'applique est celle du point col ; les structures étudiées ici ne rentrent pas dans ce cadre.

Diverses méthodes ont été développées pour extraire des estimations sur le comportement asymptotique des paramètres des objets d'une classe à partir des séries bivariées qui les comptent. Tout d'abord, en extrayant le coefficient de z^n de la série $\frac{\partial}{\partial u}C(z,u)\Big|_{u=1}$ on obtient le nombre total de marques dans tous les objets de taille n, ce qui permet de connaître le nombre moyen de marques dans un objet de taille n :

$$\frac{[z^n]\ \frac{\partial}{\partial u}C(z,u)\Big|_{u=1}}{[z^n]\ C(z,u)|_{u=1}}.$$

Une autre possibilité est de calculer la limite, quand n devient grand, de la série $n![z^n]C(z,u)$. Cette série en u satisfait les conditions de l'analyse de singularités, ce qui permet d'en déduire des lois limites pour le paramètre.

2.2 Arbres aléatoires

La notion d'arbre, qui décrit une structure hiérarchisée, correspond à un grand nombre de concepts différents issus de domaines divers et variés. On s'intéresse uniquement aux arbres enracinés et étiquetés de la combinatoire, qui ont en commun une définition récursive : un arbre est soit l'arbre vide, soit un nœud contenant une étiquette, avec des fils qui sont eux-mêmes des arbres. Les différentes possibilités pour rendre cette définition précise donnent naissance à un grand nombre de classes différentes. De plus, pour étudier les propriétés en moyenne des arbres d'une classe donnée il faut spécifier la distribution de probabilité qu'on associe à la classe. Nous présentons ici une sélection représentative des modèles les plus connus ; pour un traitement complet, on peut voir le livre récent de Drmota [Drm09].

2.2.1 Classes d'arbres

Les différentes classe d'arbres correspondent à différentes spécifications : on peut voir les fils de chaque nœud comme une séquence (ordonnée) ou comme un ensemble (non-ordonné) ; on peut spécifier le nombre de fils de chaque nœud, p.e. 2 fils correspond aux arbres binaires, et pas de contrainte aux arbres généraux. Ces règles peuvent varier parmi les nœuds, p.e. on peut avoir des nœuds noirs avec chacun trois fils rouges ordonnés qui ont eux-mêmes deux fils noirs non ordonnés.

Cependant, si on essaye de décrire les arbres aléatoires de grande taille, tous ces détails de la spécification jouent un rôle moins important que le choix de la distribution de probabilité.

Exemple : les arbres binaires. Nous allons illustrer ce dernier point sur les arbres binaires planaires (les fils sont ordonnés), qui sont définis comme suit : un arbre binaire est

- soit l'arbre vide (feuille)
- soit un nœud avec un fils gauche et un fils droit, tous deux des arbres binaires.

La spécification combinatoire correspondante est $\mathcal{T} = \mathcal{E} \cup \mathcal{Z} \star \mathcal{T} \star \mathcal{T}$. On considère deux façons pour générer aléatoirement un arbre binaire, la façon uniforme et la façon croissante.

Pour générer un arbre binaire aléatoire croissant on commence par une feuille et puis de manière itérative on remplace une des feuilles par un nouveau nœud dont les fils sont des feuilles. Le nom « croissant » vient du fait que si on met l'étiquette i au nœud qui a été créé à la i-ème étape de la génération, alors l'arbre généré vérifie la propriété de croissance : chaque nœud a une étiquette plus petite que tous les nœuds qui se trouvent dans sa descendance. On a alors une sous-classe spécifiée par $\hat{\mathcal{T}} = \mathcal{E} \cup \mathcal{Z}^{\square} \star \hat{\mathcal{T}} \star \hat{\mathcal{T}}$ pour laquelle chaque arbre est généré avec probabilité $1/n!$, tandis que les arbres de $\mathcal{T}$ qui ne sont pas dans $\hat{\mathcal{T}}$ ne sont jamais générés.

La génération uniforme est facile à énoncer mais plus difficile à mettre en œuvre : on veut générer un arbre de taille n de sorte que la probabilité d'obtenir un arbre donné est $1/C_n$, où $C_n = \frac{(2n)!}{n!(n+1)!}$ est le n-ème nombre de Catalan, qui compte (entre autres) les arbres binaires planaires de taille n. Un des algorithmes pour générer des arbres binaires aléatoires uniformes est celui de Rémy [Ré85] :

- on commence par une feuille puis de manière à chaque étape itérative on choisit une feuille ou un nœud ;
- si c'est une feuille alors on la remplace avec un nouveau nœud dont les fils sont des feuilles ;
- si c'est un nœud n alors on met à sa place un nouveau nœud dont un fils est une feuille et l'autre le nœud n.

La figure 2.1 montre un arbre binaire aléatoire croissant et un arbre binaire aléatoire uniforme. Quelques différences sont visibles à l'œil nu et resteront valables avec probabilité 1 si on génère de nouveau nos arbres. L'arbre croissant est très tassé, ses premiers niveaux semblent complets et il a une forme assez régulière. L'arbre uniforme s'étale autant en largeur qu'en profondeur et sa forme semble plus irrégulière.

2.2.2 Quelques propriétés étudiées

Ces observations, qui différencient les arbres binaires croissants des arbres binaires uniformes, resteraient vraies si on remplaçait les arbres binaires par une autre famille d'arbres. Le modèle croissant et le modèle uniforme donnent lieu à deux types de comportement pour les arbres générés, qui peuvent être quantifiées par plusieures propriétés que nous allons voir maintenant. Il est à noter qu'on retrouvera dans le chapitre 3 des modèles de graphes, croissant ou uniforme, qui présentent aussi deux types de comportements similaires à ces différents modèles d'arbres.

Profondeur. Un paramètre facile à observer expérimentalement est la *profondeur* d'un arbre, c'est à dire le maximum des distances des nœuds à la racine. Dans le cas des arbres croissants la profondeur est proportionnelle au logarithme de la taille [Dev84 ; Pit94], tandis que dans les

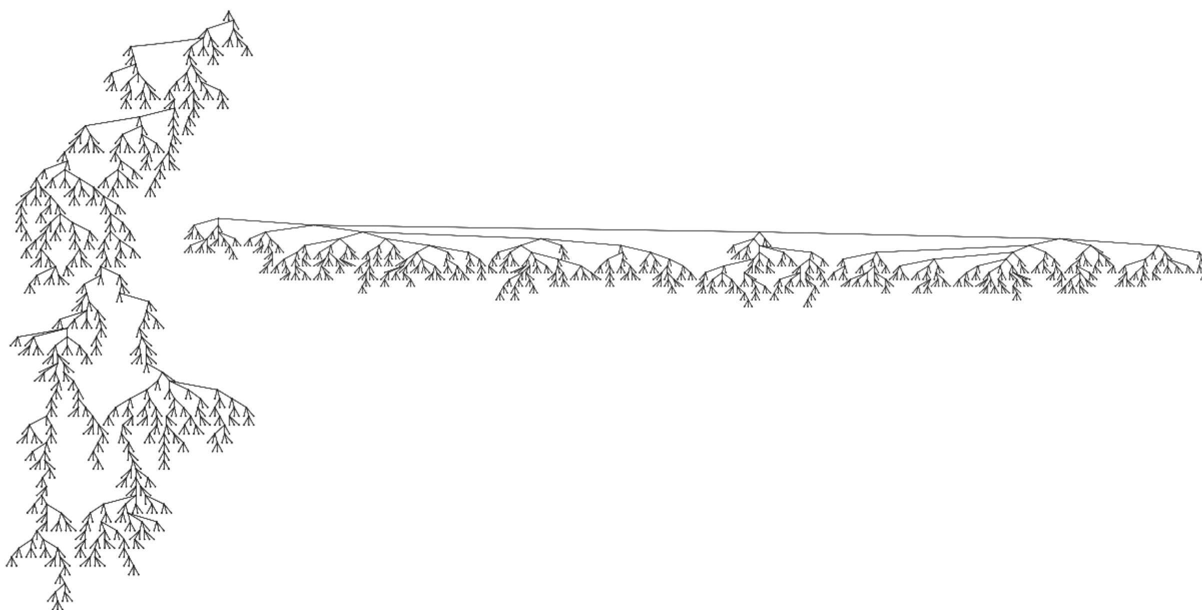


FIGURE 2.1 – Un arbre binaire aléatoire croissant (à gauche) et un arbre binaire aléatoire uniforme (à droite).

arbres uniformes elle est proportionnelle à la racine carrée de la taille [FO82].

Une propriété similaire est la largeur, qui correspond au nombre maximal de nœuds à une distance donnée de la racine.

Longueur de cheminement. La profondeur est difficile à estimer avec les méthodes de la combinatoire analytique, il est plus facile d'étudier la *longueur de cheminement*, qui est définie comme la somme des distances des nœuds de l'arbre à la racine. En divisant par la taille de l'arbre on obtient bien sûr la profondeur moyenne des nœuds, mais on préfère parler de longueur de cheminement pour éviter la confusion avec « profondeur moyenne » qui désigne la moyenne de la profondeur des arbres. Le comportement asymptotique de la longueur de cheminement est équivalent, à une constante près, à celui de la profondeur [MM78].

Profil moyen. Il est intéressant d'avoir une définition rigoureuse de la « forme » d'un arbre. Une possibilité est le *profil* d'un arbre, qui est une fonction qui à une profondeur d associe le nombre de nœuds à distance d de la racine. Les points de la figure 2.2 montrent les profils de nos deux arbres aléatoires, normalisés par la profondeur moyenne. Les courbes de cette même figure montrent le profil moyen des deux classes, c'est à dire la fonction qui à chaque profondeur associe le nombre moyen de nœuds à distance d. Les courbes correspondent donc à la hauteur moyenne des points sur l'ensemble des arbres d'une taille donnée.

Dans le cas des arbres croissants le profil moyen est une gaussienne [Lou87] ; dans les arbres uniformes une loi de Rayleigh [MM78].

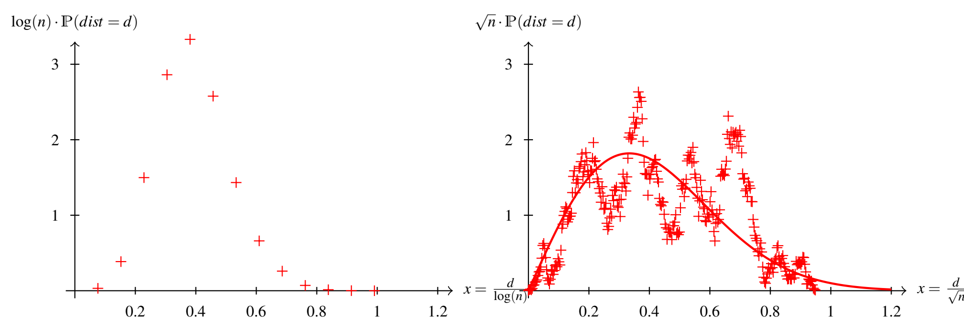


FIGURE 2.2 – Le profil d'un arbre binaire aléatoire croissant (à gauche) et d'un arbre binaire aléatoire uniforme (à droite).

Moments d'ordre supérieur du profil. La différence la plus visible entre les deux profils de la figure 2.2 est la régularité du profil des arbres croissants comparée à l'irrégularité de celui des arbres uniformes. Mais comment quantifier cette irrégularité ? Une mesure est la variance du nombre de nœuds à une distance donnée, qui est beaucoup plus grande dans le cas uniforme [MM78] que dans le cas croissant [CDJ01 ; DH05a].

Cependant la variance ne mesure pas vraiment l'irrégularité de la courbe, mais juste sa distance de la courbe moyenne. On remarque par exemple que la variance du profil des arbres croissants est plus grande sur les deux flancs de la cloche que à son milieu. Cela veut juste dire que la hauteur de ces points est moins prévisible.

Corrélation entre niveaux du profil. Une meilleure quantification de la régularité des profils est la mesure de la corrélation entre les niveaux. Dans le cas des arbres croissants cette corrélation est très forte [DH05b], alors que le profile d'un arbre uniforme se comporte localement comme un mouvement brownien [DG97] et donc la corrélation est faible.

L'étude de ces corrélations entre niveaux permet aussi d'étudier le comportement de la largeur des arbres croissants, qui est en $O(n/\sqrt{\log n})$ (à comparer avec une largeur en $O(\sqrt{n})$ dans le cas des arbres uniformes).

2.3 Génération de Boltzmann d'arbres

Le problème de génération des arbres aléatoires uniformes est un problème ancien, pour lequel on a d'abord développé des méthodes *ad hoc*, comme celle de Rémy pour les arbres binaires, mais aussi des méthodes bijectives ou des méthodes par rejet [AS95]. Sont venues ensuite les méthodes de génération utilisant la structure récursive des arbres, qui ont pu être étendues à des méthodes génériques sur des structures récursives plus complexes [FZC94]. Pour passer d'une complexité quadratique à une complexité linéaire, la méthode de Boltzmann autorise de générer des objets de taille approchée, avec de plus des algorithmes simples à implémenter. Avant de présenter en détail la méthode de Boltzmann nous allons l'illustrer sur deux classes simples d'arbres.

2.3.1 Arbres binaires

Nous avons vu comment générer un arbre binaire uniforme avec la méthode de Rémy. Oublions cette méthode pour l'instant et demandons-nous quelle serait la première intuition pour un générateur ? Utiliser la définition récursive, qui nous donne deux choix : soit on a l'arbre vide, soit on a un nœud avec deux fils. Comment choisir ? En tirant à pile ou face.

Ce procédé de génération connu sous le nom de procédé de Galton-Watson [WG75]

- génère des arbres des taille aléatoire, mais on connaît la distribution des tailles : l'arbre généré a taille n avec probabilité $\frac{1}{\zeta(\frac{3}{2})}n^{-\frac{3}{2}}$;
- la génération est uniforme pour une taille donnée : deux arbres de taille n ont la même probabilité d'apparaître ;
- la génération se termine toujours (probabilité 1 que la taille soit finie), même si la moyenne de la taille des arbres générés est infinie.

Les processus de Galton-Watson étudient aussi les pièces biaisées : si la probabilité p de ne pas s'arrêter est supérieure à un demi alors la plupart du temps le générateur ne termine pas. Si p est inférieur ou égal à un demi, alors la probabilité de générer un arbre de taille n est de l'ordre de $n^{-\frac{3}{2}}e^{\frac{1}{2}-p}$.

Ce type de génération ne se généralise pas tel quel, comme le montre le paragraphe suivant, mais son principe de fonctionnement peut s'étendre à n'importe quelle classe d'arbres spécifiée par une grammaire dans le cadre de la méthode symbolique.

2.3.2 Arbres 1-2

Considérons maintenant la classe des arbres 1-2 définie comme suit : un arbre 1-2 est soit l'arbre vide, soit un nœud avec un fils, soit un nœud avec deux fils. Si on veut appliquer la même approche intuitive que pour les arbres binaires on est tentés de générer avec la même probabilité $1/3$ soit un arbre vide, soit un nœud avec un fils, soit un nœud avec deux fils.

Or un tel générateur n'est pas uniforme : les arbres ⫯ et ⋏ ont tous les deux taille 1, mais le premier est généré avec probabilité $1/9$ et le deuxième avec probabilité $1/27$. En fait, du moment que la probabilité de générer un nœud unaire p_1 est la même que la probabilité de générer un nœud binaire p_2 ces deux arbres ne sont pas équiprobables. Pour qu'ils le soient il faut que $p_1 = \frac{1-p_2}{1+p_2}p_2$ et dans ce cas tout arbre de taille n est généré avec probabilité $p_1^n(1-p_1-p_2) = p_1^{n-k}p_2^k(1-p_1-p_2)^{k+1}$. Il reste à savoir quelle est la valeur maximale que peut prendre p_2 en conservant la propriété de terminaison du générateur. Pour cela la probabilité de s'arrêter doit être égale à la probabilité de générer un nœud binaire, c'est à dire $p_2 = \sqrt{2}-1$.

2.3.3 La méthode de Boltzmann

Un générateur est dit de Boltzmann [DFLS04] si la probabilité de générer un objet étiqueté γ est

$$\mathbb{P}_x(\gamma) = \frac{x^{|\gamma|}}{|\gamma|!C(x)}$$

où $C(z)$ est la série génératrice de la classe considérée et x un réel entre 0 et ρ_C.

L'intérêt majeur de ce modèle est qu'on peut traduire automatiquement une spécification combinatoire en un générateur de Boltzmann par composition d'algorithmes élémentaires correspondant aux constructions de la méthode symbolique.

La traduction est la suivante :

- pour générer un objet de la classe $\mathcal{A}+\mathcal{B}$ on génère avec probabilité $\frac{A(x)}{A(x)+B(x)}$ un objet de $\mathcal{A}$, sinon un objet de $\mathcal{B}$;
- pour générer un objet de la classe $\mathcal{A}\star\mathcal{B}$ on génère **indépendamment** un objet de $\mathcal{A}$ et un objet de $\mathcal{B}$.

Lorsque la classe $\mathcal{C}$ qui nous intéresse est définie par $\mathrm{Seq}(\mathcal{A})$ elle est aussi définie récursivement par $\mathcal{C}=\mathcal{E}+\mathcal{A}\star\mathcal{C}$. On va donc générer une séquence d'éléments indépendants de $\mathcal{A}$ dont la longueur correspond à la première apparition d'un 1 dans une suite de tirages Bernoulli de paramètre $A(x)$. Autrement dit, la longueur de la séquence est une variable aléatoire qui suit une loi géométrique de paramètre $A(x)$. Et de manière similaire, pour générer un objet de la classe $\mathrm{Set}(\mathcal{A})$ il suffit de tirer aléatoirement une longueur avec une loi de Poisson de paramètre $A(x)$ et générer autant d'objets de $\mathcal{A}$.

L'avantage crucial de cette méthode est que les générateurs produits sont très efficaces : leur complexité est linéaire si on accepte que la taille de l'objet produit soit dans une fenêtre autour de la taille visée. Si on veut générer un objet de taille proche de n, on utilise le rejet : on relance le générateur tant que la taille de l'objet produit n'est pas d'un l'intervalle choisi $[n(1-\varepsilon),n(1+\varepsilon)]$.

Au niveau de l'implantation de la méthode, il faut traiter différemment le rejet des objets trop petits de celui des objets trop grands. Pour les petits objets, leur génération n'est pas coûteuse et par ailleurs la génération d'objets trop grands peut être arrêtée aussitôt que le dépassement de la taille a été détecté, cela doit donc avoir un coût proportionnel à n.

2.3.4 Distribution de la taille du résultat

L'efficacité du modèle repose sur le bon choix du paramètre x en fonction de la taille visée n et de la classe considérée $\mathcal{C}$. En particulier, la taille moyenne des objets générés est $x\frac{C'(x)}{C(x)}$ et elle croît de 0 à l'infini quand x varie de 0 à $\rho_{\mathcal{C}}$.

Il est aussi important de connaître la distribution des tailles des objets produits par le générateur. Pour les constructions combinatoires étudiées ici il existe seulement trois comportements possibles, correspondant à différentes natures des singularités de la série $C(z)$ [DFLS04] : distribution centrée, distribution plate et distribution piquée.

Les figures 2.3, 2.4 et 2.5 montrent des exemples représentatifs des trois distributions sous deux formes : traditionnelle et log-log. Pour la deuxième, au lieu de tracer une courbe $(x,f(x))$, on trace une courbe $(\log(x),\log(f(x))$. Cela permet de mieux voir la nature des queues de distribution et de visualiser des distributions qui décroissent très vite.

Distribution centrée. Lorsque la singularité dominante de $C(z)$ est exponentielle, les tailles sont normalement distribuées autour de la moyenne.

En choisissant donc le paramètre x comme la solution de l'équation $n=x\frac{C'(x)}{C(x)}$ on obtient un générateur très efficace, qui réussit sans rejet avec forte probabilité.

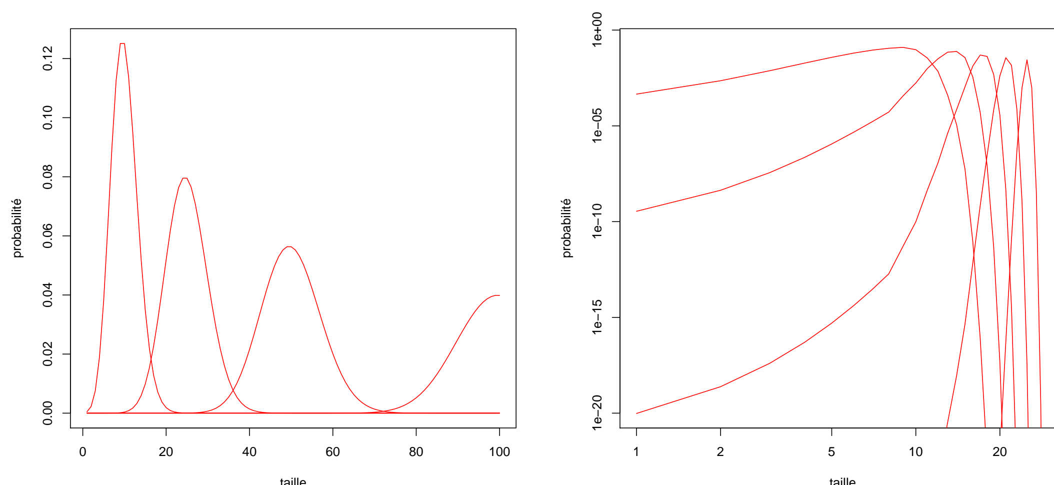


FIGURE 2.3 – La distribution centrée.

Distribution plate. Lorsque $C(z)$ a un exposant singulier α négatif, la distribution a deux régimes ; elle commence comme une loi de puissance pour finir comme une exponentielle : $\mathbb{P}(|\gamma| = n) \sim x^n n^\alpha$. Le changement de régime intervient de plus en plus tard quand x s'approche de ρ_C et toujours après la taille moyenne $x\frac{C'(x)}{C(x)}$.

Ainsi on peut encore une fois choisir la solution de l'équation $n = x\frac{C'(x)}{C(x)}$ comme valeur du paramètre x et obtenir un générateur qui retourne un objet de taille proche de n après un nombre de rejets constant.

Distribution piquée. Lorsque l'exposant singulier est positif et non entier alors on retrouve les mêmes deux régimes dans la distribution des tailles mais la loi de puissance qui correspond au premier régime décroît plus vite. La fréquence d'apparition des petits objets est trop grande quelle que soit la valeur du paramètre x choisi. Mais la taille totale des petits objets rejetés est toujours proportionnelle à la taille visée.

Dans le cas de la génération d'arbres l'exposant singulier de la série génératrice est toujours $\frac{1}{2}$. Nous pouvons alors transformer la grammaire par le procedé de « pointage » qui ramène a une distribution plate (exposant singulier $-\frac{1}{2}$) avec en contrepartie une grammaire généralement un peu plus complexe.

Lorsque la série génératrice de dénombrement a une valeur finie en sa singularité (ce qui est le cas pour les arbres), on peut faire de la génération singulière, c'est à dire choisir $x = \rho_C$. La probabilité pour obtenir un objet de taille n est strictement positive quel que soit n et de plus le coût de la génération est linéaire pour toutes les tailles visées.

L'implantation de la génération de Boltzmann pour les arbres est décrite plus en détail dans la seconde partie de ce mémoire.

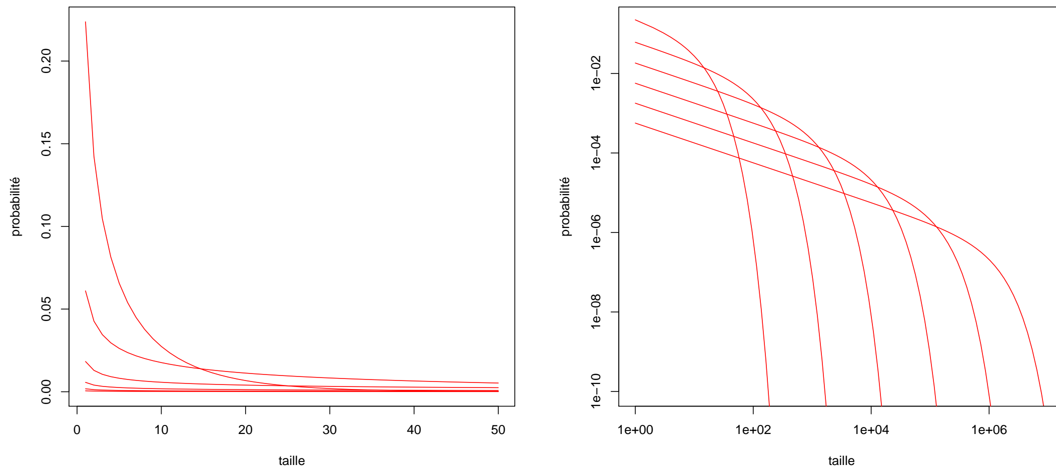


FIGURE 2.4 – La distribution plate.

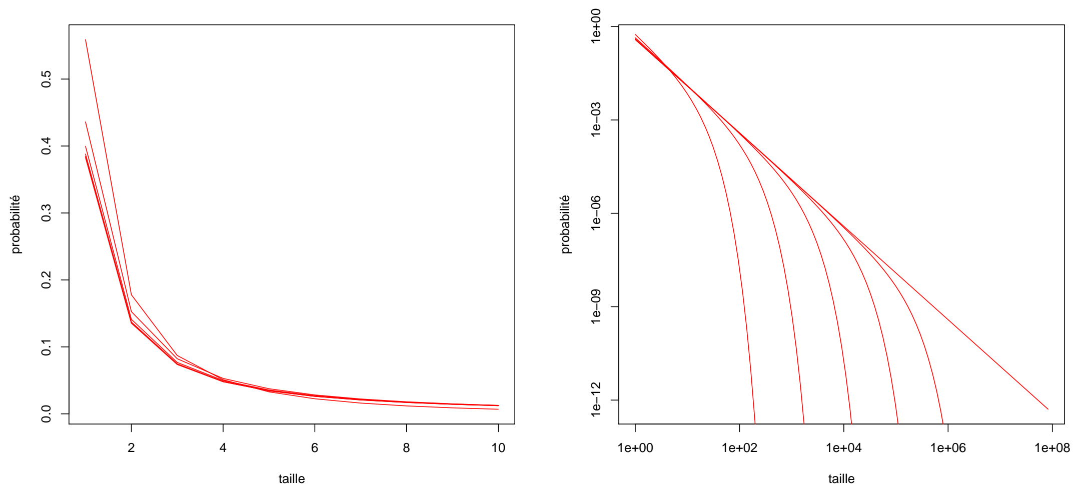


FIGURE 2.5 – La distribution piquée.

Première partie

Analyse des *k*-arbres

Chapitre 3

Introduction

Ce chapitre introduit nos travaux sur l'étude des paramètres de la classe de graphes des k-arbres. Cela commence par une présentation du domaine de la modélisation des graphes de terrain, qui était notre point de départ. Ensuite nous faisons un survol de l'histoire des k-arbres. Enfin nous énonçons nos résultats, qui sont exposés en détail dans les chapitres suivants.

3.1 Les modèles de graphes aléatoires

La généralisation de l'utilisation de l'informatique dans l'ensemble des disciplines scientifiques a permis le traitement de masses de données ingérables jusque là. Pour modéliser les relations la structure de données de choix est le graphe. Nous avons donc une profusion de graphes de terrain de tailles de plus en plus grandes. Le résultat majeur de la fin du 20ème siècle est la découverte d'une uniformité dans les propriétés des graphes issus de domaines très variés, qui au même temps sont différentes des propriétés des graphes aléatoires étudiés en mathématiques.

Dans cette thèse nous considérons uniquement des graphes étiquetés, simples, non orientés. Formellement, un graphe G de taille n est défini comme un couple (V,E) où V est l'ensemble des entiers entre 1 et n, représentant les sommets (vertices en anglais), et E un ensemble de paires de sommets, les arêtes (edges en anglais).

Définition G est un graphe étiqueté, simple et non orienté de taille n si et seulement si

$$G = (V,E), \quad V = \{i \mid 1 \leq i \leq n\}, \quad E \subset \{\{i,j\} \mid i,j \in V\}.$$

Au milieu du 20ème siècle de nombreux travaux ont eu comme sujet les propriétés des graphes aléatoires. Ces travaux partageaient une hypothèse importante : deux graphes avec le même nombre de sommets et d'arêtes sont équiprobables. Ce modèle est connu sous le nom de *modèle de Erdős-Rényi* [ER59 ; Gil59 ; Bol01] et une des définitions est la suivante : étant donnés un entier n et une probabilité p, on construit un graphe à n sommets où chacune des $\binom{n}{2}$ arêtes possibles est construite avec probabilité p.

Quand les graphes issus des domaines applicatifs ont commencé à être abondamment étudiés, on s'est aperçu qu'ils avaient certaines propriétés en commun : forte connexité, faible

densité, petite distance moyenne, une distribution de degrés particulière, une certaine organisation hiérarchique, etc. Certaines de ces propriétés n'apparaissant pas dans les graphes produits par le modèle de Erdős-Rényi [NBW06], il est très intéressant de comprendre les mécanismes qui les font apparaître. Une manière de le faire est de trouver un modèle simple de graphes aléatoires qui a ces mêmes propriétés. Aujourd'hui plusieurs modèles simples coexistent, chacun possédant certaines des propriétés mais pas d'autres. Au même temps, les applications qui ont besoin de simulations réalistes utilisent des modèles *ad hoc*, très difficiles à étudier. Les deux modèles simples les plus connus, le modèle de Watts et Strogatz [WS98] et le modèle Barabàsi-Albert [BA99], ont été proposés à la fin des années 1990.

Watts et Strogatz ont voulu proposer le modèle le plus simple possible qui illustre l'effet « petit monde » : un regroupement des sommets en « communautés » et une distance moyenne faible. Dans leur construction on commence avec N sommets placés sur un cercle et on relie chaque sommet aux K sommets les plus proches. Ensuite pour chaque arrête du graphe on décide avec probabilité β de remplacer une extrémité par un sommet choisi au hasard.

Barabàsi et Albert ont popularisé la notion d'attachement préférentiel : ils ont proposé de construire des graphes aléatoires par un processus itératif où chaque nouveau sommet se connecte à m sommets existants, choisis avec une probabilité proportionnelle à leur degré. C'est une transposition dans le monde des graphes de l'idée que « l'argent va à l'argent ». Ce processus semble apparaître naturellement dans la construction des graphes de terrain et expliquerait la différence de leur distribution de degrés par rapport aux graphes d'Erdős-Rényi.

En plus des modèles de graphes aléatoires il existe un grand nombre de modèles déterministes, visant le plus souvent à illustrer une propriété caractéristique des graphes de terrain. Par exemple, les graphes hiérarchiques [BRV01] ont une structure en modules hiérarchisés qu'on retrouve dans les graphes de terrain mais pas dans les modèles aléatoire actuels.

3.2 Les propriétés étudiées

Regardons plus en détail les propriétés les plus étudiées dans les graphes de terrain.

3.2.1 Nombre d'arêtes

Une des premières observations sur les graphes de terrain est qu'ils sont peu denses : leur nombre d'arêtes est proportionnel au nombre de sommets ($|E| \propto |V|$). Les modèles plus récents sont tous construits de manière a respecter cette propriété. Le modèle d'Erdős-Rényi prend un paramètre p qui influe directement sur la proportion sommets/arêtes, le nombre moyen d'arêtes étant $\binom{n}{2}p$. Il peut donc respecter la propriété si on choisit une valeur de p proportionnelle à $1/n$, et c'est ce que nous ferons par la suite.

3.2.2 Connexité

Les graphes de terrain ont une composante connexe regroupant l'écrasante majorité des sommets. La connexité des graphes issus du modèle Erdős-Rényi a été abondamment étudiée, notamment par Erdős et Rényi eux-mêmes. Elle dépend du paramètre p choisi, et pour notre

cas où $p = \alpha/n$ avec $\alpha > 1$, on a l'apparition d'une composante géante [ER60]. La plupart des études et des modèles récents se concentrent sur cette composante connexe géante en négligeant le reste du graphe.

3.2.3 Distance

Historiquement l'étude des graphes de terrain a commencé par la sociologie. Un des concepts les plus médiatisés est le « phénomène du petit monde », c'est à dire l'hypothèse que deux personnes prises au hasard sont reliées par une courte chaîne de connaissances. Ce phénomène a été mis en évidence par une série d'expériences très connues et controversées faites par Stanley Milgram, qui ont donné naissance au concept de « six degrés de séparation ».

Nous savons depuis que pour tous les graphes de terrain la distance moyenne est très petite, mais il est difficile de quantifier, justement parce que la croissance est lente. Le consensus actuel est que la distance moyenne dans les graphes de terrain est du même ordre que le logarithme du nombre de sommets. Mais comment distinguer la différence entre un log, un log log et une racine, sachant que le rôle du coefficient peut alors être très important, que les graphes considérés ont souvent des tailles qui dépassent pas le million de sommets, ou encore pire, et que parfois on étudie un seul graphe (p.e. internet).

3.2.4 Degré

Le degré des sommets d'un graphe est à la fois facile à mesurer et met en évidence la nature particulière des graphes de terrain.

Définition Le degré d'un sommet v, noté $d(v)$, est son nombre de voisins :

$$d^\circ(v) = |N(v)| = |\{w|\{v,w\} \in E\}|$$

On s'intéresse à la distribution de degrés des sommets d'un graphe, c'est à dire, le nombre de sommets qui ont degré 1, combien ont degré 2, etc. Dans un graphe aléatoire produit par le modèle Erdős-Rényi la distribution suit une loi binomiale : $\mathbb{P}(d^\circ(v) = d) = \binom{n-1}{d} p^d (1-p)^{n-1-d}$, qui tend vers une loi de Poisson quand n devient grand : $\mathbb{P}(d^\circ(v) = d) \to e^{-(n-1)p}(n-1)^d p^d / d!$.

Mais dans les graphes de terrain la très grande majorité des sommets a un très petit degré, et il existe aussi des sommets avec des degrés énormes ; le type de distribution est caractéristique d'une loi de puissance : $\mathbb{P}(d^\circ(v) = d) \sim d^{-\alpha}$. Les modèles qui ont une distribution de degrés en loi de puissance sont ceux qui suivent un attachement préférentiel, comme Barabàsi-Albert.

Cependant Newman [New05] a mis en évidence l'existence des familles de graphes de terrain avec des distributions équivalentes pour les petits degrés mais avec des degrés maximaux plus petits ; ce type de distributions s'apparente au produit d'une loi de puissance et d'une exponentielle, qu'on appellera une loi de puissance avec une chute exponentielle : $\mathbb{P}(d^\circ(v) = d) \sim d^{-\alpha}\beta^d$.

Comme on peut le voir dans la figure 3.1, la meilleure façon pour visualiser une loi de puissance est en log-log. Quand $f(x) = x^{-\alpha}$, on aura une courbe $(\log(x), -\alpha\log(x)) = (x', -\alpha x')$,

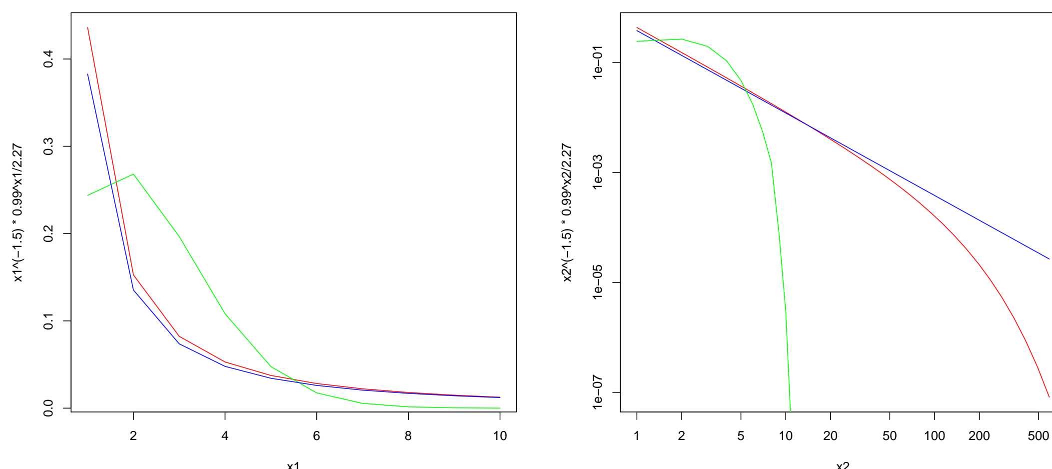


FIGURE 3.1 – Les trois types de distribution de degrés observés dans les graphes de terrain : loi de puissance, loi de puissance avec chute exponentielle et poisson.

c'est à dire une ligne droite de pente $-\alpha$. De manière équivalente, une loi de puissance avec une chute exponentielle commencera comme une ligne droite avant de chuter.

Molloy et Reed [MR95] étudient les graphes dont la distribution de degrés est une séquence donnée. Les outils de simulation de graphes aléatoires les plus utilisés actuellement (voir p.e. [VL05]) sont inspirés de ce modèle. L'utilisateur donne une séquence de degrés, qui est elle-même le résultat d'un tirage aléatoire suivant la loi de distribution voulue, comme la loi de puissance, et le générateur produit un graphe simple connexe avec la bonne distribution de degrés.

3.2.5 Clustering

Une des particularités des graphes de terrain est la tendance des sommets à se regrouper en « paquets » ou clusters avec une connexité forte à l'intérieur des clusters et une connexité faible entre clusters. Il est très important de prendre les clusters en compte, puisque ils sont en général liés à des informations pertinentes du domaine initial. L'identification des clusters dans un graphe donnée est un problème très important et largement étudié.

Pour vérifier l'adéquation des modèles vis à vis de la propriété de clustering on a besoin d'une quantité mesurable. La plus utilisée est le coefficient de clustering : pour un sommet donné on calcule la proportion de ses voisins qui sont voisins entre eux.

Définition Le coefficient de clustering d'un sommet v, $C(v)$, est défini par :

$$C(v) = \frac{|\{\{w,w'\} \in E \,|\, w,w' \in N(v)\}|}{d(v)(d(v)-1)}.$$

Le coefficient de clustering d'un graphe est la moyenne du coefficient de clustering de ses sommets.

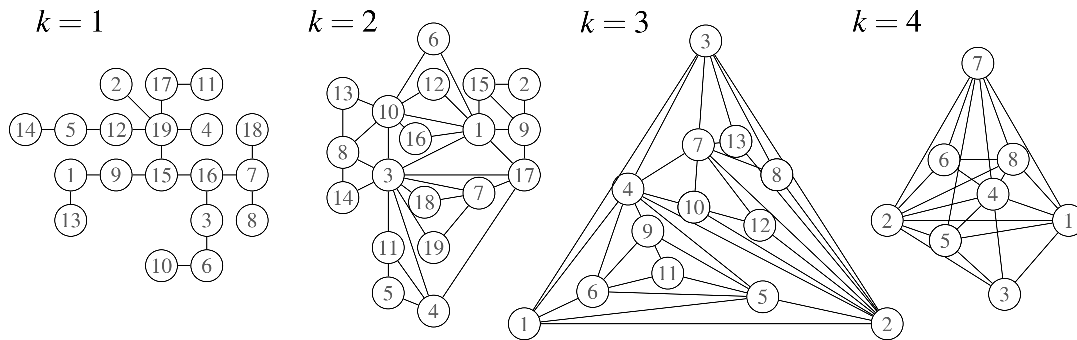


FIGURE 3.2 – Quelques exemples de k-arbres

Les graphes de terrain semblent avoir un coefficient de clustering très grand, mais aussi le coefficient de clustering de leurs sommets est inversement proportionnel à leur degré.

Le graphes produits par le modèle Erdős-Rényi ont un coefficient de clustering faible. Le modèle de Barabàsi-Albert a un coefficient de clustering plus fort que Erdős-Rényi, mais qui décroît quand la taille des graphes augmente. Le modèle de Watts et Strogatz a été conçu pour être très simple et avoir à la fois une distance moyenne très faible et un coefficient de clustering fort.

3.3 Les *k*-arbres

Les k-arbres et une multitude de classes de graphes très proches apparaissent indépendamment, depuis le 18ème siècle et jusqu'à nos jours, dans différents domaines comme la géodésie ou la physique statistique. Cette section est consacrée à un tour d'horizon de ces occurrences.

Définition [BP69] Un k-arbre à n sommets est soit une k-clique (le graphe complet à k sommets), soit un k-arbre à $n-1$ sommets dans lequel on a choisit une k-clique et on ajoute un nouveau sommet adjacent à tous les sommets de la clique.

La figure 3.2 illustre cette définition par quelques exemples. On remarque que quand $k = 1$ on retrouve une définition des arbres. Les 2-arbres étaient utilisés en géodésie [Cla58 ; McK97] pendant un siècle avant d'être découverts indépendamment en combinatoire [HP68].

Il existe un très grand nombre de classes de graphes liés aux k-arbres parmi lesquelles nous considérons uniquement deux d'entre elles, les k-arbres planaires qui sont une sous-classe des k-arbres et les graphes cordaux qui les contiennent.

Définition Un k-arbre est dit planaire si et seulement si on peut placer ses sommets sur une boule de dimension k de manière à éviter toute intersection entre les k-simplexes définis par les k-cliques du graphe.

De manière équivalente, un k-arbre est planaire si et seulement si aucune de ses k-cliques n'a plus de 2 sommets voisins. Cette notion de planarité est celle des complexes simpliciaux ;

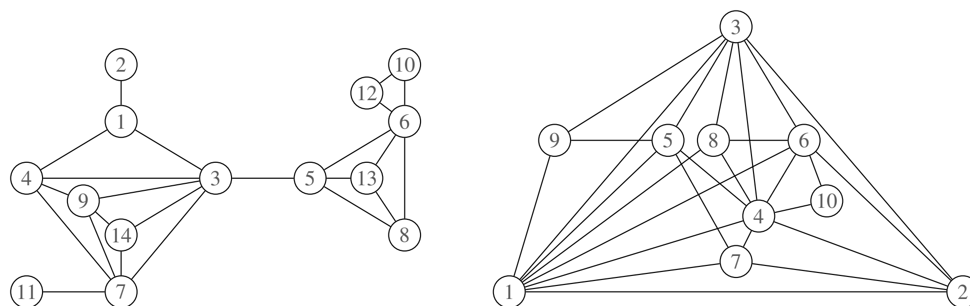


FIGURE 3.3 – Deux exemples de graphes cordaux.

elle se confond avec la notion de planarité de la théorie des graphes dans le cas $k = 3$ et avec la notion de « outerplanar » dans le cas $k = 2$.

Les 2-arbres planaires sont plus connus sous le nom de triangulations d'un polygone, ou encore graphes « outerplanar » maximaux. Les triangulations des polygones ont été l'objet de nombreux travaux et leur énumération remonte à Euler [Eul43], donnant ainsi la première occurrence des nombres de Catalan, qui comptent aussi, parmi beaucoup d'autres structures, les arbres planaires enracinés.

La classe des graphes cordaux, ou graphes triangulés, englobe celle des k-arbres. On peut les définir de manière similaire aux k-arbres :

Définition Le graphe réduit à un sommet est un graphe cordal et pour obtenir un graphe cordal de taille n on prend un graphe cordal de taille $n-1$ dans lequel on choisit une clique et on relie tous les sommets de cette clique à un nouveau sommet.

Les graphes cordaux sont apparus en théorie des graphes à la même période que les k-arbres et jouent un rôle central en algorithmique des graphes, notamment en tant que structure intermédiaire pour des algorithmes efficaces.

On présente un bref état de l'art sur les études qui ont été faites des k-arbres d'un point de vue combinatoire et algorithmique et d'un point de vue de modélisation.

3.3.1 Les k-arbres en combinatoire

Les 2-arbres ont été introduits en combinatoire par Harary et Palmer [HP68] dans le contexte des complexes simpliciaux, objet de base de la topologie algébrique et étudiés en combinatoire [Sta93]. Ces auteurs donnent une équation vérifiée par la série génératrice qui compte les 2-arbres non-étiquetés et l'utilisent pour compter les triangulation du polygone non-étiquetés. La même année Beineke et Pippert [BP68] énumèrent les 2-arbres étiquetés, et ils introduisent et comptent les k-arbres étiquetés en 1969 [BP69] : le nombre de k-arbres de taille n est $n!\binom{n+k}{k}(kn+1)^{n-2}$.

Des preuves alternatives de ce résultat sont proposées par d'autres auteurs [Moo69 ; Foa71]. Palmer et Read s'intéressent au cas des 2-arbres plans en 1973 [PR73].

En 1985, Wormald [Wor85] étudie la série génératrice comptant les graphes cordaux étiquetés et en calcule les premiers termes.

Les 2-arbres refont surface dans les années 2000 au Québec, avec l'énumération des 2-arbres non-étiquetés [FGLL02] et l'introduction d'une autre extension, les 2-arbres k-gonaux [LLL04]. Les 2-arbres k-gonaux étiquetés sont énumérés par la même série génératrice que les k-arbres, mais n'ont pas les mêmes symétries ; les séries ne sont donc pas les mêmes pour les structures non-étiquetés correspondantes.

3.3.2 Les k-arbres en théorie des graphes

Beineke et Pippert [BP71] se sont intéressés aussi aux propriétés caractéristiques des k-arbres, suivis par Dewdney [Dew74] qui introduit également la classe de (m,n)-arbres. Indépendamment de ces travaux, Rose présente les k-arbres comme des cas particuliers de graphes triangulés et s'intéresse également à leur propriétés caractéristiques.

Dans ces deux cas il s'agit de trouver des propriétés caractéristiques des arbres qui s'étendent aux dimensions supérieures. Par exemple un graphe est un arbre si et seulement si il est connexe, sans triangles et tous ses séparateurs minimaux sont des sommets, alors qu'il est un k-arbre si et seulement si il est connexe, contient une k-clique mais pas de $(k+2)$-clique et et tous ses séparateurs minimaux sont des k-cliques.

Au début des années 1980, Proskurowski [Pro80] met en place des algorithmes efficaces sur les k-arbres, notamment pour calculer la distance entre sommets. Ces algorithmes se basent sur une représentation arborescente des k-arbres. Proskurowski avec ses coauteurs [KCP82] utilisera cette représentation pour mettre en place un algorithme efficace d'isomorphisme pour une classe de graphes englobant les k-arbres. Plusieurs auteurs se sont intéressés à l'amélioration de cet algorithme [Cha90 ; ADK07 ; GSS08].

Un résultat très important et qui a beaucoup contribué à la notoriété des k arbres concerne les k-arbres partiels. Un k-arbre partiel est un graphe à partir duquel on peut obtenir un k-arbre en rajoutant des arêtes. Beaucoup d'algorithmes NP-complets deviennent « faciles » si on se restreint à la classe des k-arbres partiels, pour un k donné [AP89]. Bien sûr la constante croit extrêmement vite en fonction de k.

Graphes planaires uniquement 4-colorables. Une propriété triviale des k-arbres est qu'il sont uniquement $(k+1)$-colorables. Jensen et Toft [JT94] conjecturent que les 3-arbres planaires (qu'ils appellent « recursive maximum planar graphs ») sont les seuls graphes planaires uniquement 4-colorables. Cette conjecture a été démontrée par Fowler [Fow98 ; Tho98] en adaptant la preuve du théorème des 4 couleurs de Robertson et al. [RSST97]. Xu [Xu09] a très récemment proposé une preuve non assistée par ordinateur de cette conjecture et du théorème des 4 couleurs. Sa preuve n'a pas encore été vérifiée mais son papier donne au moins un bon aperçu de l'histoire du problème. Il est à noter que les travaux autour de ce problème sont sans aucun lien avec les autres travaux sur les k-arbres.

3.3.3 Les k-arbres en tant que modèle de graphes aléatoires

Récemment certains modèles de graphes aléatoires, qui se trouvent être des k-arbres, ont été proposés en tant que modèles pour les graphes de terrain. La manière la plus simple pour

construire des k-arbres aléatoires est de transformer la définition récursive en un algorithme itératif :

- on commence avec une k-clique
- à chaque étape on choisit aléatoirement une k-clique et on relie tous ces sommets à un nouveau sommet.

Si on étiquette les sommets dans leur ordre d'apparition seuls $(n-2)!/(k-1)!$ parmi les $n!\binom{n+k}{k}(kn+1)^{n-2}$ k-arbres étiquetés de taille n peuvent être construits (dans la figure 3.2 seul le 3-arbre est planaire). On les appelle k-arbres croissants.

Ces deux définitions, k-arbres et k-arbres croissants, amènent à deux modèles probabilistes différents pour l'étude statistique d'objets générés uniformément dans chacune des deux classes.

Comme nous avons vu dans la section 2.2, dans le cadre des arbres un grand nombre de variantes sont étudiées, notamment les arbres croissants. Il a notamment été observé que les arbres aléatoires uniformes ont une profondeur moyenne proportionnelle à la racine carrée de leur taille, tandis que les arbres aléatoires croissants sont beaucoup plus tassés avec une profondeur logarithmique. Nous verrons qu'on observe des phénomènes de même nature dans le cas des k-arbres aléatoires.

Réseaux apolloniens. Un nombre important de travaux récents en physique [AHAS05 ; LGH04 ; DM05 ; PAHP07] s'intéresse à une sous-classe très particulière des 3-arbres planaires, les réseaux apolloniens. [1] Il s'agit de partir d'un triangle et à chaque étape itérative prendre tous les triangles et dans chacun d'entre eux poser un nouveau sommet relié aux trois sommets du triangle, créant ainsi trois nouveaux triangles à la place de chaque ancien. L'intérêt de ce modèle est que malgré sa simplicité il présente une très grande similitude avec les graphes de terrain au niveau des propriétés significatives.

Réseaux apolloniens aléatoires. Les réseaux apolloniens ont aussi une variante aléatoire [ZYW05], qui présente les mêmes propriétés intéressantes de distribution de degrés et de distance moyenne. Ces objets correspondent encore une fois aux 3-arbres planaires croissants. Une généralisation à des dimension supérieures a aussi été introduite [ZCFR06], et il s'agit des k-arbres planaires croissants.

Triangulation en pile. Différents travaux de combinatoire [BB07 ; FZ08] introduisent les triangulations en pile comme objet intermédiaire. Cette classe est en fait celle des 3-arbres planaires. Marckert et Albenque [MA08] étudient les propriétés des triangulations en pile et des réseaux apolloniens aléatoires avec des méthodes probabilistes de convergence de l'arbre ternaire associé vers un arbre continu.

Un travail récent de Gao [Gao09] utilise des méthodes de martingale pour étudier la distribution de degrés dans un k-arbre aléatoire croissant.

1. Les réseaux apolloniens prennent leur nom d'Apollonius de Perga, qui avait énoncé le problème des contacts [VS46] : étant donnés trois cercles trouver un quatrième qui soit tangent à tous les trois.

3.4 Nos résultats : la bijection

La construction des k-arbres par greffes successives de nouveaux sommets suggère une structure arborescente sous-jacente, qui peut être confondue avec l'historique de cette construction. C'est cette idée qui permet la mise en place d'algorithmes efficaces sur ces structures. Il faut cependant un effort supplémentaire pour décrire une classe d'arbres facile à énumérer et en bijection avec les k-arbres. Les constructions classiques soit contiennent beaucoup d'informations redondantes soit ne sont pas des bijections.

L'objectif de cette section est de présenter notre bijection et de montrer comment se transportent les paramètres.

Nous considérons des k-arbres enracinés : une k-clique est désignée *clique racine* et les sommets de cette clique sont ordonnés. Le premier d'entre eux est appelé le *sommet racine*.

Théorème 3.4.1. *La classe des k-arbres enracinés est en bijection avec la classe $\mathcal{K}$ des arbres définie par :*

$$\mathcal{K} = \mathcal{Z}^k \star Set(\mathcal{T}), \quad \mathcal{T} = \mathcal{Z} \star Set(\mathcal{T}^k).$$

Dans cette construction il est facile de voir la distance d'un sommet au sommet racine. Il est donc possible de l'utiliser pour estimer le degré de la racine ou la distance d'un sommet aléatoire au sommet racine.

Pour mieux expliquer notre bijection, commençons par le cas plus simple des k-arbres planaires.

3.4.1 Les 2-arbres planaires

Quand $k = 2$, il s'agit de la classe des triangulations des polygones, dont la bijection avec les arbres binaires est un résultat classique : on représente chaque triangle par un nœud et chaque arête par une arête. Un nœud est adjacent à une arête si le triangle qu'il représente la contient. Deux nœuds sont donc reliés par une arête si les triangles respectifs partagent une arête, et on obtient un arbre non-enraciné dont tous les nœuds ont degré 3. On conserve le positionnement relatif des triangles dans le plan, cet arbre est alors planaire. On désigne comme racine une des arêtes du polygone, qui correspond sur l'arbre à une arête reliée à un seul nœud qui devient la racine de l'arbre. Nous avons alors un arbre binaire planaire.

Nous varions un peu cette bijection parce que nous voulons que chaque nœud représente un sommet du graphe et pas un triangle. Il y a deux sommets de plus que de triangles, on ajoute donc deux nouveaux nœuds en dehors de l'arbre, représentant les deux sommets de l'arête (clique) racine. Le nœud correspondant au seul triangle contenant la clique racine sera la racine de l'arbre et représentera le seul sommet du triangle non inclus dans la clique racine. Ensuite on visite tous les nœuds de l'arbre en partant de la racine et en y associant à chaque fois le seul sommet du triangle correspondant qui n'apparaît pas encore dans l'arbre.

Théorème 3.4.2. [2] *La classe des 2-arbres planaires enracinés est en bijection avec la classe*

2. Le fait que les triangulations d'un polygone étaient énumérés par la même série que des structures arborescentes (en l'occurence des produits de nombres) était connu par Catalan [FS09, p.20]. La notion d'arbre est cépendant plus récente et la bijection sous cette forme date donc du 20ème siècle.

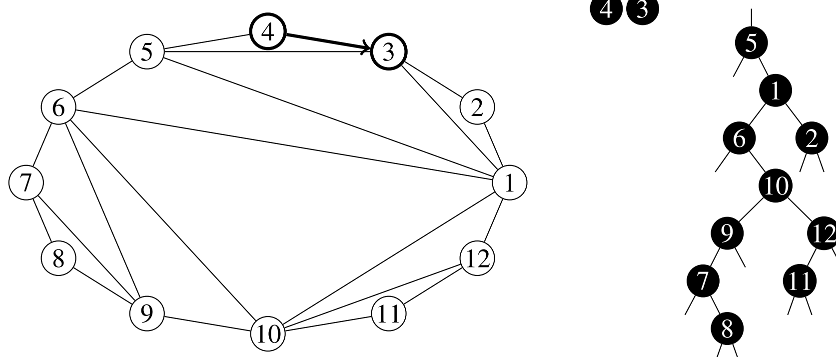


FIGURE 3.4 – Un 2-arbre planaire (triangulation d'un polygone) enraciné et l'arbre correspondant.

$\mathcal{Z}^2 \star \mathcal{B}$, avec $\mathcal{B}$ les arbres binaires planaires enracinés définis par :

$$\mathcal{B} = \mathcal{E} \cup \mathcal{Z} \star \mathcal{B}^2.$$

3.4.2 Les k-arbres planaires

Pour k quelconque, cette construction se généralise facilement si on enracine les k-arbres de la manière suivante : on choisit une k-clique *feuille*, c'est à dire une k-clique incluse dans une seule $(k+1)$-clique, qu'on appelle la *clique racine*, et on impose un ordre sur ses k sommets. On la représente dans l'arbre correspondant par une séquence de k nœuds placés à la racine. On représente le seul sommet du graphe voisin de tous les sommets de la clique racine par un nœud au bout de la racine. Ce sommet compose avec la clique racine une $(k+1)$-clique qui elle-même contient, en plus de la clique racine, k k-cliques.

L'ordre des sommets de la racine induit un ordre sur ces k-cliques voisines de la clique racine ($k-1$ sommets de la racine plus le nouveau sommet), qui vont correspondre à k arrêtes partant de la racine. Si un sommet est voisin de tous les sommets d'une de ces k-cliques, il sera représenté par un nœud attaché à l'arrête correspondante. Comme le k-arbre est planaire, il ne peut y avoir qu'un seul sommet à la fois.

La k-clique en question est un séparateur du graphe et on peut traiter la partie non encore explorée de la même manière, en prenant cette k-clique comme racine avec le même ordre de sommet que la racine originelle, le nouveau sommet prenant la place du seul manquant. On obtient ainsi un arbre k-aire qui représente le k-arbre.

Corollaire 3.4.3. *La classe des k-arbres planaires enracinés est en bijection avec la classe $\mathcal{K}_p$ des arbres k-aires planaires enracinés définie par :*

$$\mathcal{K}_p = \mathcal{Z}^k \star \mathcal{T}_p, \text{ avec } \mathcal{T}_p = \mathcal{E} \cup \mathcal{Z} \star \mathcal{T}_p^k.$$

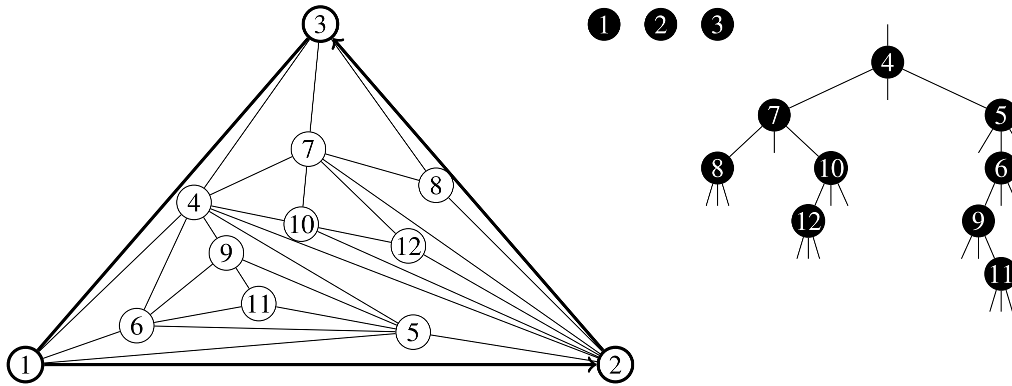


FIGURE 3.5 – Un 3-arbre planaire (RANS) et l'arbre correspondant.

3.4.3 Les k-arbres

Intéressons-nous maintenant au cas général des k-arbres. Comme une k-clique peut avoir un nombre quelconque de voisins, et pas uniquement 0 ou 1 comme c'est le cas pour les k-arbres planaires, nous avons potentiellement plusieurs nœuds au bout de chaque arête. On parlera alors de *groupes de fils* d'un nœud. Un groupe de fils est un ensemble de nœuds non ordonnés, tandis que la séquence des k groupes qui se trouve sous chaque fils est ordonnée. On peut interpréter chaque groupe comme une multi-arête, que nous pouvons représenter p.e. avec un nœud intermédiaire, dont les fils ne sont pas ordonnés, ou alors annoter chaque arête par l'indice du groupe.

La seule autre différence avec les k-arbres planaires est à la clique racine, qui ne sera plus une k-clique feuille, car contrairement aux k-arbres planaires, le nombre de k-cliques feuilles n'est pas déterminé par le nombre de sommets du graphe. La racine de la représentation arborescente ne sera pas un nœud qui correspond à un sommet du graphe, mais une multi-arête. On retrouve la spécification du théorème 3.4.1.

La série génératrice $K(z) = \sum_{n=0}^{\infty} K_n z^n/n!$, où K_n est le nombre d'arbres de taille n dans la classe $\mathcal{K}$, peut être obtenue à partir de la spécification de $\mathcal{K}$ grâce à la méthode symbolique :

$$K(z) = z^k \exp(T(z)), \quad T(z) = z\exp(T^k(z)).$$

Cette série nous permet de retrouver le résultat sur le nombre de k-arbres de Beineke et Pippert :

Théorème 3.4.4. *[BP69] Le nombre de k-arbres étiquetés de taille $n+k$ est*

$$n!\binom{n+k}{k}(kn+1)^{n-2}.$$

3.4.4 Réenracinement

La clique racine est introduite pour pouvoir réaliser la bijection mais n'a de statut particulier dans le graphe de départ que pour le cas croissant, où elle correspond toujours à la clique

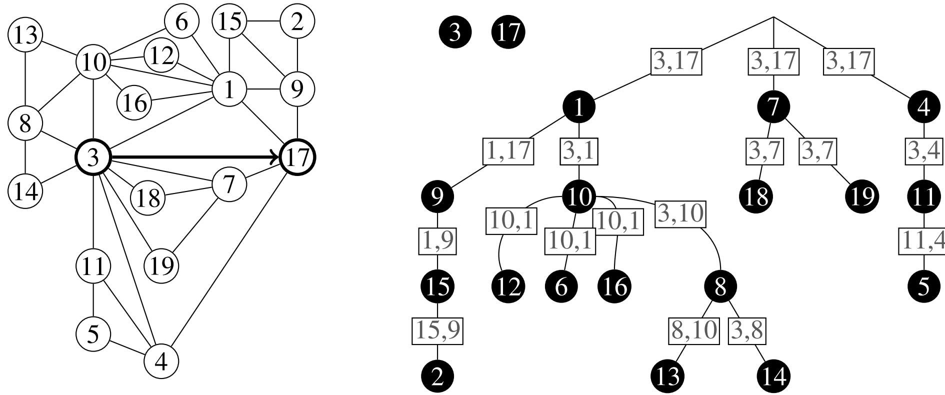


FIGURE 3.6 – Un 2-arbre enraciné et l'arbre correspondant. Les groupes de fils vides ne sont pas représentés.

$(1,\ldots,k)$. Dans le cas non croissant elle est choisie uniformément parmi l'ensemble des cliques du graphe. On s'attend alors à ce que un sommet choisi au hasard ait des propriétés équivalentes à celle du sommet racine. Ce résultat n'est pourtant pas immédiat, parce que tous les sommets d'un graphe n'ont pas la même probabilité d'être le sommet racine : certains sont présents dans plus de cliques que d'autres.

Pour étudier les propriétés d'un sommet choisi au hasard on transforme l'arbre correspondant au k-arbre pour mettre le sommet en question à la position du sommet racine. Proskurowski [Pro80] a utilisé avant nous la même approche pour son algorithme de calcul des distances dans les k-arbres. L'arbre qu'on obtient après le retournement fait partie d'une nouvelle classe, car il garde la trace de son ancienne racine. La différence avec la classe de départ est cependant négligeable, ce qui confirme que dans les k-arbres aléatoires uniformes, tout comme dans les arbres uniformes, un sommet aléatoire a des propriétés très similaires à la racine.

Théorème 3.4.5. *La classe des k-arbres enracinés avec un sommet marqué est en bijection avec la classe $\mathcal{K}^\circ$ des arbres définie par :*

$$\mathcal{K}^\circ = k\mathcal{Z}^k \star Set(\mathcal{T}) \cup \mathcal{Z}^k \star Set(\mathcal{T}) \star \mathcal{T}^\circ$$
$$\mathcal{T}^\circ = \mathcal{Z} \star Set(\mathcal{T}^k) \star (\mathcal{T}^k \cup k\mathcal{T}^{k-1} \star \mathcal{T}^\circ).$$

3.4.5 Restriction au cas croissant

Pour étudier les propriétés des k-arbres aléatoires croissants nous aurons besoin d'identifier la sous-classe de $\mathcal{K}$ correspondant aux seuls k-arbres dont les étiquettes respectent la définition itérative. Cela est équivalent à dire que soit le graphe est une k-clique, soit le voisinage du sommet avec la plus grande étiquette est une k-clique et le graphe sans ce sommet vérifie cette définition récursive. La classe $\mathcal{K}_c$ correspondante contient tous les arbres de $\mathcal{K}$ où la clique racine est la séquence $\{1,\ldots,k\}$ et l'étiquette de chaque nœud est plus grande que toutes les étiquettes dans le sous-arbre au dessous.

Corollaire 3.4.6. *La classe des k-arbres croissants est en bijection avec la classe $\mathcal{K}_c$ des arbres définie par :*

$$\mathcal{K}_c = \mathcal{Z}^\square \star \cdots \star \mathcal{Z}^\square \star Set(\mathcal{T}_c), \quad \mathcal{T}_c = \mathcal{Z}^\square \star Set(\mathcal{T}_c^k).$$

Corollaire 3.4.7. *Le nombre de k-arbres croissants à $n > k$ sommets est $(n-2)!/(k-1)!$.*

3.4.6 Extension aux graphes cordaux

Les graphes cordaux peuvent être vus comme une généralisation des k-arbres, dans le sens où la taille des cliques utilisées à chaque étape itérative n'est pas fixée. Il est donc naturel d'essayer d'étendre la bijection à cette classe. La difficulté vient du fait que le nombre de cliques de chaque taille est variable dans l'ensemble des graphes cordaux d'une taille donnée, nous avons donc besoin de garder cette information quand on enracine un graphe cordal. Wormald [Wor85] l'a fait pour compter le nombre de graphes cordaux étiquetés, même si sa présentation ne fait pas apparaître explicitement la structure arborescente sous-jacente. Une construction très proche est le graphe clique-séparateur de Ibarra [Iba04], qui est un DAG représentant les graphes cordaux et qui devient un arbre très proche du notre si on se restreint aux k-arbres.

La bijection peut s'appliquer aux graphes cordaux croissants, pour lesquels la racine est imposée par les étiquettes. On part du sommet avec l'étiquette 1 et chaque nouveau sommet v qui s'attache à une clique c est représenté par un nœud appartenant au groupe de fils correspondant à c. Sous le nœud représentant v on trouve $2^{|c|}$ groupes de fils, correspondant aux nouvelles cliques créés par l'ajout de v.

La classe d'arbres qu'on obtient est $\mathcal{C} = \mathcal{T}_0$ où

$$\mathcal{T}_k = \mathcal{Z}^\square \star \prod_{i=1}^{k+1} \mathrm{Set}^{\binom{k+1}{i}}(\mathcal{T}_i)$$

ce qui met en évidence le point faible de cette bijection : la classe est spécifiée par un système infini. Pour pouvoir le manipuler il faut borner la taille des cliques.

3.5 Estimation des paramètres

Nous utilisons la bijection pour étudier les propriétés des k-arbres aléatoires uniformes. En effet, la classe d'arbres $\mathcal{K}$ est construite de manière à rendre facilement calculable la distance de chaque sommet au sommet racine. Ce calcul est proche de l'algorithme de Proskurowski [Pro80] qui calcule aussi la distance dans les k-arbres, mais le fait de se baser sur une bijection nous permet en plus de calculer le nombre de sommets à une distance donnée du sommet racine.

3.5.1 La distance entre sommets du graphe vue sur l'arbre

L'idée de Proskurowski est de créer un arbre des distances du k-arbre, ce qui n'est rien de plus que l'arbre résultant d'un parcours en largeur (BFS) de ce graphe. Dans cet arbre la

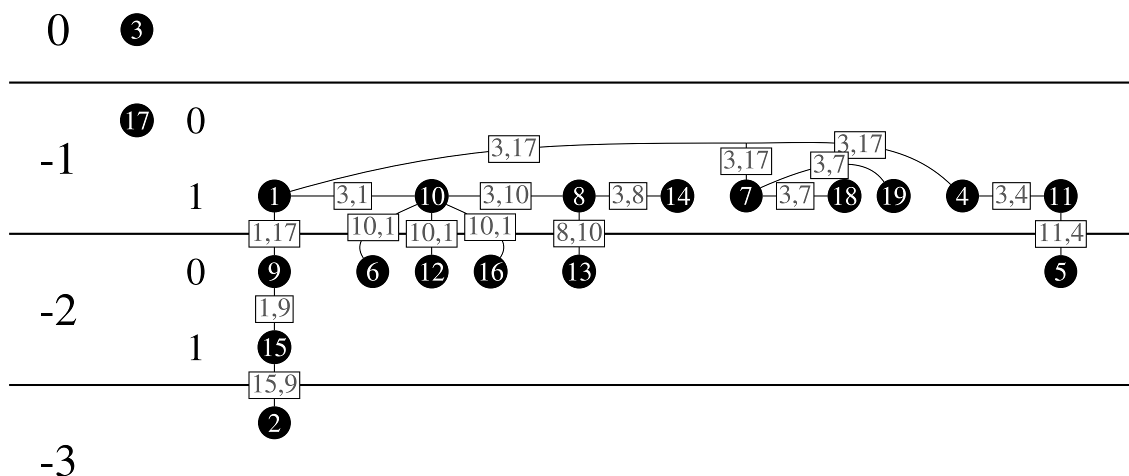


FIGURE 3.7 – L'arbre de la figure 3.6 avec les niveaux et sous-niveaux mis en évidence.

profondeur d'un nœud est égale à la distance du sommet correspondant au sommet racine du k-arbre.

Notre construction arborescente a une propriété plus faible :

Propriété Soit v un sommet d'un k-arbre G, et d_v la distance de v au sommet racine de G ; nommons v_p le sommet qui se trouve juste au-dessus de v dans la représentation arborescente de G et d_p la distance de v_p au sommet racine. Alors $d_v - d_p \in \{0, 1\}$.

Autrement dit, quand on descend d'un niveau dans cet arbre, la distance correspondante dans le graphe soit reste constante, soit croît de un.

Pour visualiser cette propriété nous pouvons dessiner l'arbre en mettant à la même hauteur les sommets qui se trouvent à la même distance du sommet racine (voir figure 3.7). On voit alors que les fils d'un nœud sont soit au même niveau que lui, comme 1 et 10, soit au niveau inférieur comme 1 et 9. Il y a donc deux types de relation père-fils. [3]

Il reste à savoir comment distinguer les deux types de relation en utilisant uniquement l'information contenue dans l'arbre. Pour cela nous devons affiner le découpage de l'arbre en niveaux, car nous avons besoin de savoir un peu plus sur un nœud pour comprendre ses relations avec ses fils. On remarque que chaque sommet v est voisin de exactement k sommets qui apparaissent au dessus de lui dans l'arbre et ces sommets forment une k-clique, la *clique père* de v. Si v est à distance d de la racine, alors cette clique va contenir $i \in \{0, \ldots, k-1\}$ sommets à distance d de la racine et $k-i$ sommets à distance $d-1$. On mettra alors v au niveau $(-d)_i$ de l'arbre, et en faisant cela pour chaque nœud on obtient une nouvelle disposition de l'arbre. Cette disposition vérifie toujours la propriété qu'un nœud se trouve soit au même niveau que son père, soit au niveau inférieur ; mais en plus, le nombre de groupes de fils au dessous d'un nœud au niveau $(-d)_i$ est toujours égal à $k-i$.

3. La même situation apparaît si on regarde la représentation en arbre binaire d'un arbre général : le fils gauche d'un nœud correspond à un fils dans l'arbre général, tandis que le fils droit correspond à un frère.

Propriété La racine de l'arbre est au niveau $(-1)_{k-1}$ puisque la clique racine du graphe contient $k-1$ sommets à distance 1 du sommet racine. Soit v un nœud au niveau $(-d)_i$ de la représentation arborescente d'un k-arbre, alors parmi ses k groupes de fils il y a i qui sont au même niveau, $(-d)_i$ et $k-i$ qui sont au niveau suivant : niveau $(-d)_{i+1}$ si $i < k-1$, niveau $(-(d+1))_0$ sinon.

Pour savoir quels sont les $k-i$ groupes de fils qui se trouvent au niveau suivant il faut voir la position des $k-i$ sommets à distance $d-1$ du sommet racine dans la clique père de v. Si le j-ème sommet de la clique père de v est à distance $d-1$ du sommet racine, alors le j-ème groupe mène au niveau suivant. En particulier, les voisins du sommet racine sont les nœuds accessibles depuis la racine sans jamais aller dans le premier groupe de fils d'un nœud.

3.5.2 Séries bivariées

Pour estimer les valeurs des paramètres des k-arbres nous avons besoin de la série bivariée qui compte le nombre de sommets à distance d du sommet racine. Pour trouver la récurrence qui décrit cette série nous utilisons le découpage en niveaux que nous venons de présenter. On opère un décalage sur les niveaux, en plaçant la racine de l'arbre au niveau $(d-1)_{k-1}$, ainsi les sommets que nous voulons compter sont tous ceux qui se trouvent au niveau 0.

Nous allons utiliser la famille de séries bivariées $T_{d,i}(z,u) = \sum T_{d,i,n,p} z^n u^p / n!$ où $T_{d,i,n,p}$ compte le nombre d'arbres de $\mathcal{T}$ de taille n avec leur racine au niveau d_i et p nœuds au niveau 0. En particulier $T_{d,k-1}(z,u)$ le nombre de sommets à distance $d+1$ du sommet racine.

Les différents comportements des nœuds en fonction de leur niveau dans l'arbre se traduisent directement en système d'équations sur les séries $T_{d,i}(z,u)$. Pour d positif on a

$$\begin{aligned} T_{d,i}(z,u) &= zu^{\delta_0(i)} \exp(iT_{d,i}(z,u) + (k-i)T_{d,i+1}(z,u)) && \text{si } i < k \\ T_{d,k}(z,u) &= T_{d-1,0}(z,u) && \text{sinon} \end{aligned}$$

tandis que pour d négatif il reste plus de sommets à compter et donc $T_{d,i}(z,u) = T(z)$.

La série $T_{0,k-1}(z,u)$ qui compte le nombre de voisins du sommet racine est donc définie par $T_{0,k-1}(z,u) = zu\exp((k-1)T_{0,k-1}(z,u) + T(z))$.

3.5.3 Résultats

À partir de ces séries bivariées nous pouvons estimer plusieurs paramètres. La probabilité que le sommet racine d'un k-arbre de taille n ait degré d est donnée par

$$\frac{[z^n u^d] z^k u^{k-1} \exp(T_{0,k-1}(z,u))}{[z^n]K(z)}.$$

On obtient ainsi la distribution du degré du sommet racine, mais aussi la distribution de degrés dans tout le graphe, en utilisant le fait que tous les voisins d'un sommet sauf k sont parmi ses descendants dans l'arbre.

Théorème 3.5.1. *La distribution de degrés dans les k-arbres suit une loi de puissance avec une chute exponentielle : pour d assez grand*

$$\mathbb{P}(d^\circ(v) = d + k) = \frac{k}{(k-1)e\sqrt{\pi}} \left(e^{\frac{1}{k}} \frac{k-1}{k} \right)^d d^{-\frac{3}{2}}.$$

Nous nous intéressons ensuite au nombre moyen de sommets à distance d du sommet racine, qui est donné par

$$\frac{[z^n] \frac{\partial}{\partial u} z^k \exp(T_{d,k-1}(z,u)) \Big|_{u=1}}{[z^n]K(z)} = \frac{[z^{n-k}] \exp(T(z)) \frac{\partial}{\partial u} T_{d,k-1}(z,u) \Big|_{u=1}}{[z^n]K(z)}.$$

La relation de récurrence qui lie les $\frac{\partial}{\partial u} T_{d,k-1}(z,u)$, pour $d > 1$ est une simple multiplication par une fraction rationnelle $H(z,T(z))$, que nous pouvons interpréter combinatoirement comme ce qu'il reste dans un arbre de $\mathcal{T}$ si on enlève un de ses sous-arbres commençant un (grand) niveau au-dessous de la racine.

Nous avons donc besoin de calculer le nombre de d-uplets de taille n d'arbres comptés par $H(z,T(z))$. Le fait que $H(z,T(z))$ compte une famille simple d'arbres fait que la « bonne » normalisation, c'est à dire la fonction avec laquelle il faut diviser d pour avoir un comportement indépendant de n est $\sqrt{n}$.

Théorème 3.5.2. *Le profil d'un k-arbre de taille n, c'est à dire le nombre moyen de sommets à une distance donnée du sommet racine suit une loi de Rayleigh dans une échelle $\sqrt{n}$:*

$$\lim_{n\to\infty} \sqrt{n}\mathbb{P}(d(v,r) = \lceil x\sqrt{n} \rceil) = h_k^2 x e^{-\frac{(h_k x)^2}{2}}, \text{ où } h_k = k \textstyle\sum_{i=1}^k \frac{1}{i}.$$

Nous nous attendons à ce que les distances à un sommet quelconque ne soient pas vraiment différentes de celles du sommet racine. Pour le confirmer nous utilisons la bijection avec les arbres réenracinés qui ont une spécification très proche de $\mathcal{K}$ mais placent à la racine de l'arbre un sommet quelconque. Les séries bivariées sont construites de la même manière et la différence de leur comportement asymptotique est négligeable.

Théorème 3.5.3. *La distance moyenne entre deux sommets dans un k-arbre de taille n est de l'ordre de $\sqrt{n}$.*

La construction des séries bivariées s'applique de la même manière au cas croissant, la différence étant la nature du système résultant, qui est différentiel plutôt que algébrique. L'étude [DHBS10] des propriétés des k-arbres aléatoires croissants est postérieure aux travaux présentés ici.

Le tableau 3.1 résume les résultats qu'on a obtenus pour les k-arbres aléatoires uniformes et les compare à ceux connus pour les RAN. Les k-arbres planaires aléatoires uniformes se comportent comme les k-arbres aléatoires uniformes et les k-arbres aléatoires croissants comme les RAN. Le profil et la distribution du degré à la racine des RAN n'a pas été étudié mais il est certainement de même nature que celui des k-arbres aléatoires croissants [DHBS10].

	k-arbres uniformes	RAN
Classe d'arbres	$\mathcal{T} = \mathcal{Z} \star \mathrm{Set}^k(\mathcal{T})$	$\hat{\mathcal{T}} = \mathcal{Z}^\square \star (\mathcal{I} \cup \hat{\mathcal{T}})^3$
Série génératrice	$T(z) = z\exp(kT(z))$	$\hat{T}'(z) = (1+\hat{T}(z))^3$
Développement sing.	$T(z) = \tau - h\sqrt{1-z/\rho} + \dots$	$\hat{T}(z) = (1-3z)^{-1/3}$
Distance moyenne	$O(\sqrt{n})$	$O(\log n)$
Distrib. de degrés	Loi de puissance avec chute exp.	Loi de puissance
Degré à la racine	Loi de puissance avec chute exp.	Loi stable
Profil moyen	Loi limite Rayleigh	Loi limite Gaussienne

TABLE 3.1 – Propriétés des k-arbres aléatoires uniformes et des réseaux apolloniens aléatoires.

Cet article est tiré des actes de la conférence AofA07 [DS07].

Chapter 4

Degree distribution of random Apollonian network structures and Boltzmann sampling

Random Apollonian networks have been recently introduced for representing real graphs. In this paper we study a modified version: random Apollonian network structures (RANS), which preserve the interesting properties of real graphs and can be handled with powerful tools of random generation. We exhibit a bijection between RANS and ternary trees, that transforms the degree of nodes in a RANS into the size of particular subtrees. The distribution of degrees in RANS can thus be analysed within a bivariate Boltzmann model for the generation of random trees, and we show that it has a Catalan form which reduces to a power law with an exponential cutoff: $\alpha^k k^{-3/2}$, with $\alpha = 8/9$. We also show analogous distributions for the degree in RANS of higher dimension, related to trees of higher arity.

4.1 Introduction

The introduction of computers as a tool in every scientific domain has allowed the analysis of an ever increasing volume of data. It became possible to represent, in every detail, complex networks that emerge from the interaction of different entities. But traditional graph theory is not sufficient for the study of these real-life networks, in particular random graphs have very different properties than the networks under consideration. This created an interest for a formal description of the characteristic properties of real-life networks and the development of random network models respecting these properties. The most important results were the description of two sets of properties, accompanied by models respecting them. The first is the small world model, characterized by a small mean distance and a large clustering coefficient. The second is scale free networks, characterized by the fact that the vertex degrees follow a power-law distribution.

Current research is trying to provide a single model having all the significant properties at the same time. Ravasz et al. [RSMOB02] introduced the hierarchical networks, a model

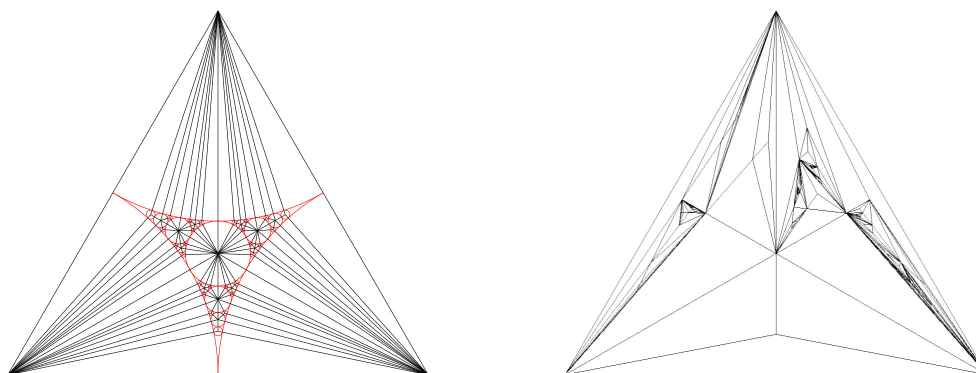


Figure 4.1: An Apollonian network and a random Apollonian network

that satisfies these properties but is deterministic; followed by a similar model, the Apollonian networks by Andrade et al. [AHAS05]. That in turn inspired the random Apollonian networks (RAN) proposed by Zhou et al. [ZYW05]. The RAN model provides a simple random model with very interesting properties, particularly a large clustering coefficient and power-law degree distribution. For more information on these notions, Newman et al. [NBW06] give a compilation of recent developments and results on the structure and dynamics of networks.

In this paper we propose a variation of RAN that we call *random Apollonian network structures* (RANS). Our study of the RANS model relies on a bijection with unlabeled ternary trees. This bijection can be extended to the RAN model, which is isomorphic to increasing ternary trees. Indeed RANS and RAN are the same objects but generated with different distributions: a RANS corresponds to many different RAN when underlying labelings are taken into account.

In the RANS model, generated networks have a distribution for degrees which has the form of a power law with an exponential cutoff, a distribution appearing in real graphs, as pointed out by Amaral et al. [ASBS00]. Large clustering is also preserved due to its direct relation to degree (a relation which is established for RAN and is carried over to RANS).

As for the RAN model (see [ZRC06]), the basic RANS model can be extended to higher dimensions and the same properties still hold. The key point for RANS is that they are isomorphic to families of trees that can be efficiently generated within a generic model of random sampling for combinatorial structures: Boltzmann model. This model does not reduce to efficient sampling but is also used to show properties of random objects.

This paper is divided in five parts. After this introduction, we present the random Apollonian network structures together with the bijection with ternary trees, and show the correspondence between the degree of a vertex in a RANS and the size of a partial subtree of the ternary tree. Part 3 is dedicated to Boltzmann sampling of trees and the distribution of underlying parameters. We exhibit non critical substitution schemas that lead to discrete distributions. Part 4 deals with the degree distribution in RANS. Using bivariate analysis we show that it reduces to a non critical substitution of trees within trees and the distribution has a Catalan form that expresses as a power law with an exponential cutoff. In part 5 the RANS model is extended to higher dimensions with analog distributions.

4.2 Random Apollonian network structures

Original RAN can be generated using a simple iterative algorithm. Starting with an empty triangle, repeat the following step until the network reaches the desired size: choose randomly an empty triangle, place a new vertex in it and connect the new vertex to the three vertices of the triangle.

In this generation the order of creation of the vertices is relevant whereas for RANS we consider the resulting structure, independently of generation order. This leads to the following definition.

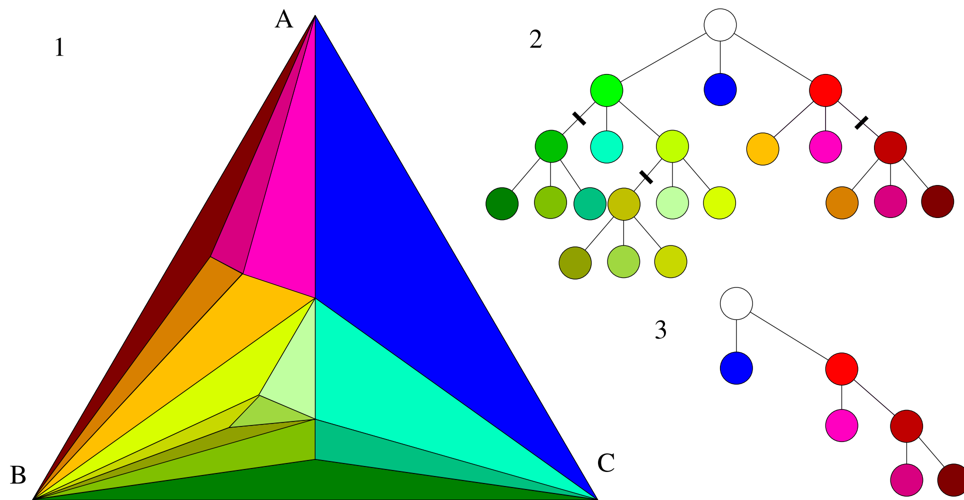


Figure 4.2: A random Apollonian network structure of order 7 (1), its corresponding tree T (with the neighborhood of the center delimited by cuts on the branches) (2) and the $\bar{1}$-subtree of T (3).

Definition A random Apollonian network structure (RANS) is recursively defined as:

- an empty triangle (corresponding to order 0) or
- a triangle T split in three parts, by placing a vertex v connected to the three vertices of the triangle; each sub-triangle being a RANS (see figure 4.2).
  Triangle T will be called the *outermost triangle* of the RANS, its vertices the *outermost vertices*; and vertex v will be called the *center* of the RANS.
  The order of the RANS is one plus the sum of the orders of the three sub-RANS.

Proposition 4.2.1. *There is a bijection between random Apollonian networks structures of order N and rooted plane ternary trees of size N (counting only internal nodes).*

Proof. A ternary tree is recursively defined as either an empty tree or an internal node with three ternary trees as children. In the bijection the root of the ternary tree corresponds to the center of the RANS and each internal node corresponds to a vertex of the RANS. At the same time, every node of the ternary tree (including leaves) corresponds to a triangle of the RANS.

Ternary trees are planar trees, the linear ordering of siblings being relevant. This order is carried over to triangles: naming A, B, C the outermost vertices of the RANS, imposes a linear ordering on the sub-RANS: the first (resp. second, third) one will be the one not containing A (resp. B, C). Recursively replacing the missing outermost vertex by the center of the RANS preserves the order in sub-RANS. □

Remark 4.2.2. In the original RAN the order of creation of the vertices is relevant (note that this order has nothing to do with the linear ordering of siblings). Labeling the vertices by their time of creation leads to a labeled network with a corresponding tree in which the labels increase on every branch, from the root to the leaves. Extended in this way, the bijection of proposition 4.2.1 transforms RAN into increasing ternary trees.

We show that in this bijection, the degree of the center of a RANS can be read on a triplet of binary subtree at the root of the corresponding tree. The important point for constructing these binary trees is the operation of recursively cutting a specified son in the tree.

Definition The binary tree that remains after cutting the first son of all nodes in a ternary tree will be called $\bar{1}$-subtree, and more generally we call $\bar{n}$-subtree the binary tree that remains after cutting each n-th son of all nodes in a ternary tree.

Let's call *neighborhood of an outermost vertex in a RANS* the neighbors of the vertex in the RANS seen as a graph, excluding the two other outermost vertices.

Lemma 4.2.3. *The size of the neighborhood of outermost vertex A (resp. B, C) in a RANS is equal to the size (counting only internal nodes) of the* $\bar{1}$*-subtree (resp.* $\bar{2}$, $\bar{3}$*) of the corresponding ternary tree.*

Proof. This is a consequence of the linear ordering of the triangles, recursively applied. □

Let's call *neighborhood of the center of a RANS* the neighbors of the vertex in the RANS seen as a graph, excluding the outermost vertices of the RANS. This corresponds to the neighborhood of outermost vertex A in the first sub-RANS, plus the neighborhood of outermost vertex B in the second sub-RANS, plus the neighborhood of outermost vertex C in the third sub-RANS.

Lemma 4.2.4. *The degree of the center of a RANS is equal to three plus the size of its neighborhood.*

Proof. The center v of a RANS of non-zero order is always linked with the three vertices of the outermost triangle. The other links come from the sub-RANS where v becomes an outermost vertex. □

Terminology We shall call *R-degree* of a ternary tree $T = \text{Node}(T_1, T_2, T_3)$ the degree of the center of the corresponding RANS; that is equal to three plus the size of the $\bar{1}$-subtree of T_1 plus the size of the $\bar{2}$-subtree of T_2 plus the size of the $\bar{3}$-subtree of T_3. Thus the R-degree of a ternary tree is the size of a triplet of binary trees.

Example Let's illustrate the preceding notions on the example of figure 4.2. The outermost vertex A has a neighborhood made of three vertices (its neighbors in the graph excluding B

and C), B has a neighborhood of seven vertices and C of three vertices. The neighborhood of outermost vertex A corresponds to the $\bar{1} - substree$ of ternary tree T (figure 4.2.3). The neighborhood of the center of the RANS is made of three vertices, as also seen on tree T when cutting as indicated on figure 4.2.2. Two vertices come from the (internal) nodes of the $\bar{1}$-subtree of the first subtree at the root of T, and one vertex comes from the internal node of the $\bar{3}$-subtree of the third subtree at the root of T. Thus the degree of the center is six, and this is the R-degree of T.

It is also important to notice that every vertex can be seen as the center of the smallest RANS containing it; all its neighbors are exclusively included in this sub-RANS. Section 4.4 will use this bijection to study the degree distribution of the RANS by analysing the different tree classes.

4.3 Boltzmann sampling

Boltzmann model for random generation by Duchon et al. [DFLS04] relies on the idea that an object receives a probability proportional to an exponential of its size $\Pr(\gamma) \asymp x^{|\gamma|}$, where x is a tuning parameter.

This model well combines with analytic combinatorics to produce efficient samplers for structures described in terms of combinatorial constructors, with composition of generators designed according to composition of structures.

Boltzmann sampling is not only of interest for it's efficiency but it also provides a way of proving properties of random objects by studying the structure of the underlying sampler.

This section concentrates on bivariate Boltzmann sampling especially in the case of trees. Section 4.4 relies on properties of Boltzmann model to deduce the degree distribution in RANS.

4.3.1 Distribution of a parameter under Boltzmann sampling

In the case of unlabeled structures, the normalizing factor is given by the value at x of the ordinary generating function $C(z) = \sum C_n z^n$, where C_n is the number of objects of size n: $\Pr(\gamma) = \frac{x^{|\gamma|}}{C(x)}$, with x in the range $]0, \rho[$, where ρ is the radius of convergence of $C(z)$.

Boltzmann sampling is uniform (all objects of a given size have the same probability to be generated), but the size N of the generated object is a random variable, with probability distribution depending on the tuning parameter: $\Pr(N = n) = \frac{C_n x^n}{C(x)}$.

Bivariate generating functions classically take into account the enumeration of structures according both the size and another parameter Ω: $C(z,u) = \sum_{n,k} C_{n,k} u^k z^n$, where $C_{n,k}$ is the number of objects γ of size n such that $\Omega(\gamma) = k$ (see eg. [SF95]). And $C_{n,k}/C_n$ represents the probability distribution of parameter Ω, conditioned on structures with fixed size n.

In the usual exact model, one considers the probability generating function of Ω over objects of size n:

$$p_n(u) = \sum_k \frac{C_{n,k}}{C_n} u^k = \frac{[z^n]C(z,u)}{[z^n]C(z)}. \tag{4.1}$$

In Boltzmann model, the distribution of Ω under a Boltzmann sampling of the objects is obtained by summing over all possible sizes:

$$\begin{aligned}\Pr(\Omega = k) &= \sum_n \Pr(\Omega = k/N = n) \times \Pr(N = n) \\ &= \sum_n \frac{C_{n,k}}{C_n} \times \frac{C_n x^n}{C(x)} = \frac{\sum_n C_{n,k} x^n}{C(x)} = \frac{[u^k]C(x,u)}{C(x,1)}.\end{aligned}$$

Thus the probability generating function of parameter Ω on Boltzmann generated objects is given by

$$p(u) \equiv \sum_k \Pr(\Omega = k)\, u^k = \frac{C(x,u)}{C(x,1)}. \tag{4.2}$$

4.3.2 Tree sampling

Tree structures (simple families of trees as in [MM78]) constitutes a basic family of objects, constructed on the sole operations of sum, product and recursion. Their generating functions satisfy equations of the form $T(z) = z\phi(T(z))$, where operator ϕ encodes the possible degrees of nodes in trees of the given family (the case $T(z) = 1 + \phi(T(z))$ can also be treated in the same way). This recursive schema leads to square root singularity for the corresponding generating function. The series has radius of convergence ρ and a singular extension of the form

$$T(z) = \tau - h\sqrt{1 - z/\rho} + O(1 - z/\rho).$$

There are several efficient methods to generate random trees [Lot05]. The seminal paper by Duchon et al. [DFLS04] presents singular Boltzmann samplers (*i.e.* tuned on $x = \rho$) for trees, that lead to size distribution according to a power law with parameter $3/2$: for large enough n,

$$\Pr(|T| = n) = \frac{T_n \rho^n}{\tau} \sim \frac{n^{-3/2}}{2\tau\sqrt{(\pi)}}.$$

Figure 4.3 shows the sizes of 100,000 ternary trees generated using a singular Boltzmann sampler (the generating function is $T(z) = 1 + zT^3(z)$; with radius of convergence $\rho = 4/27$ and singular value $\tau = 3/2$). Both axes are in a logarithmic scale: the points follow a line of slope $-3/2$, which confirms the fact that the distribution follows a power law with parameter $3/2$.

To be able to analyse the generated trees we need to assure that they can be contained in the memory of our computer. Most generated trees are of manageable sizes, and the probability of generating an infinite tree is null, but for any given size we will obtain a tree bigger than that with probability one if we wait long enough. We will thus reject all trees exceeding a fixed size. Since the rejection is based solely on the size of the trees, we preserve the uniformity of the sampling on any given size class.

4.3.3 Non critical substitution schemas

In families of trees, the square root nature of the singularity implies a finite value τ for the series $T(z)$ at its singularity ρ. Substituting $T(z)$ in a construction with radius of convergence

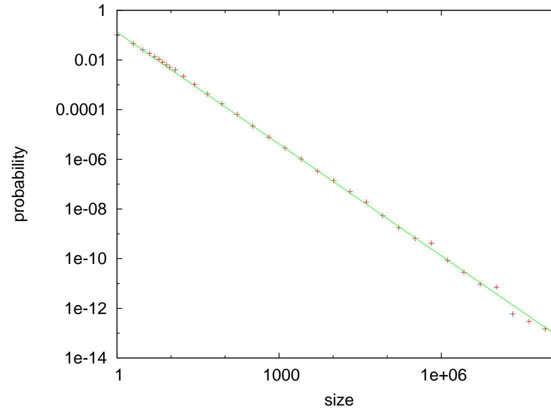


Figure 4.3: Distribution of sizes of ternary trees generated using a singular Boltzmann sampler.

bigger than τ gives a so called *non critical substitution schema*. In this context, Boltzmann sampling comes to discrete distributions for parameters. Non critical substitution schema is a keystone in our analysis of degree distribution in (regular or higher dimensional) RANS.

As examples of non critical substitution schemas, one can cite the distribution of the root degree in general trees randomly produced by singular Boltzmann samplers. In the case of planar non labeled trees the distribution is geometric: and in the case of Cayley trees (non planar labeled trees) it is Poisson. Moreover, by proposition 4.4.1 these results hold for the distribution of degrees in random trees.

Proposition 4.3.1. *Let $F(t) = \sum f_n t^n$ be a series with non negative coefficients and radius of convergence $r > 0$, associated to $\mathcal{F}$ structures; and let $T(z)$ be the generating function of a simple family of trees $\mathcal{T}$:*

$$T(z) = \tau - c\sqrt{1 - z/\rho} + O(1 - z/\rho), \qquad \text{with} \qquad c > 0.$$

Consider the bivariate generating function $C(z,u) = F(uT(z))$, with u marking substitution of $\mathcal{T}$ in $\mathcal{F}$. If $\tau < r$, then the distribution for the number of $\mathcal{T}$-structures in a random $\mathcal{F}$-structure generated by a Boltzmann sampler tuned at ρ follows a discrete law

$$p(u) = \frac{F(u\tau)}{F(\tau)}.$$

Proof. The proof uses equation (4.2), and relies on the hypothesis that F is regular at the singular value of T, so that for $z = \rho, F(T(z)) = F(\tau)$. □

Remark 4.3.2. There exists a non critical substitution schema in the exact model: under the same hypothesis, the distribution for the number of $\mathcal{T}$-structures in a random $\mathcal{F}$-structure of size n asymptotically follows a discrete law, which is the derivative of the corresponding law in proposition 4.3.1. Using equation (4.1), and expanding $F(T(z))$ around $z = \rho$, one gets

$$p_n(u) \sim_{n\to\infty} \frac{uF'(u\tau)}{F'(\tau)}.$$

4.4 Degree distribution in RANS

The results of section 4.2 show that the degree distribution in RANS is the same as the distribution of the sizes of $\bar{1}$-subtrees in triplets of ternary trees. The analysis relies on two points. First notice that, for additive parameters in trees, Boltzmann sampling emphasizes the fact that the distribution at the root of a tree is the same as the distribution at any node of the tree. Second express the parameter in terms of subcritical substitution of trees within trees and thus conclude with a Catalan law.

4.4.1 Degree distribution and tree parameters

Our goal is to study the distribution of degrees for the set of vertices in a RANS. This resumes to studying the distribution of the R-degree for the set of subtrees in the random ternary tree corresponding to the RANS. Each vertex in the RANS is the center of a sub-RANS and it's degree is the R-degree of the ternary subtree corresponding to this sub-RANS.

Instead of analysing the properties of the set of subtrees in a single random ternary tree, we will work on a set of Boltzmann generated random ternary trees and use the fact that both sets have the same statistical properties.

Proposition 4.4.1. *The statistical properties of the set of all subtrees of a random tree are the same as the statistical properties of a set of random trees independently generated with a Boltzmann sampler.*

Proof. The proof simply relies on the principle of Boltzmann sampling: every subtree of a random tree is generated by an independent call to the Boltzmann sampler, with the same parameter. □

Corollary 4.4.2. *The degree distribution in a RANS is the same as the distribution of the R-degree in a set of ternary trees using a Boltzmann sampler.*

Proof. As noted at the end of Section 4.2, the degree distribution in a RANS can be seen as the distribution of the center of each of its sub-RANS. This in turn, by Lemmas 4.2.3 and 4.2.4 is related to properties of the subtrees of the corresponding ternary tree and by proposition 4.4.1 it suffices to study a set of Boltzmann randomly generated ternary trees. □

The analysis of the degree distribution in RANS, which is done in subsection 4.4.2, reduces to studying the size distribution of triplets of binary trees and leads to a power law of parameter $3/2$ with an exponential cutoff $\alpha^k, \alpha < 1$. More precisely the result is stated in the proposition below, which will be proved by bivariate analysis in next subsection.

Theorem 4.4.3. *The degree distribution in random Apollonian network structures has mean value 6, and a Catalan form for the probability generating function, which reduces to a power law with an exponential cutoff:*

$$\Pr(D=3+k) = \frac{8}{9}\frac{1}{k+3}\binom{2k+2}{k}\frac{2^k}{9^k} \qquad \textit{that is} \qquad \Pr(D=3+k) \sim C\left(\frac{8}{9}\right)^k (k+3)^{-3/2}.$$

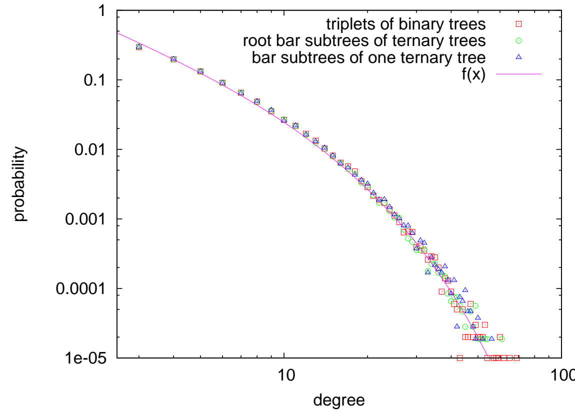


Figure 4.4: Degree distribution in RANS: theory, simulation and counting

Figure 4.4 shows the degree distribution in RANS of size 100,000 using four overlayed plots. The squares show the distribution of sizes in 100,000 triplets of binary trees generated using a Boltzmann sampler tuned at the singularity of ternary trees. The circles show the degree distribution of vertices in one big RANS of size 100,000. The triangles show the distribution of the degree of the center in 100,000 RANS. The line is the theoretical distribution of the size of triplets of binary trees, as given by theorem 4.4.3.

4.4.2 Bivariate analysis

Following Lemma 4.2.4 the R-degree D of a ternary tree $T = \text{Node}(T_1, T_2, T_3)$ is:

$$D(T) = 3 + |\bar{N}_1(T_1)| + |\bar{N}_2(T_2)| + |\bar{N}_3(T_3)|$$

where $\bar{N}_i(T_i)$ denotes the size of the $\bar{i}-$ subtree of T_i.

Clearly the distribution of the size of $\bar{n}$-subtrees is the same for any n in $\{1,2,3\}$. So that if $D(z,u)$ represents the bivariate generating function marking the R-degree of a ternary tree, one has

$$D(z,u) = \sum_T z^{|T|} u^{D(T)} = zu^3 T^3(z,u), \tag{4.3}$$

where $T(z,u)$ is the bivariate generating function for ternary trees with u marking the size of the $\bar{1}$-subtree. In the following, the distributions in both $T(z,u)$ and $D(z,u)$ are shown to be of Catalan type.

Recursively marking only two over the three subtrees at each node one gets:

$$T(z,u) = 1 + uzT(z)T^2(z,u). \tag{4.4}$$

One can read on this equation that $T(z,u)$ is obtained by substituting a ternary tree at the internal nodes of a binary tree, with u counting the number of ternary trees. Moreover this is a non critical substitution schema, and by proposition 4.3.1 the distribution is discrete.

Proposition 4.4.4. *In random ternary trees generated by a singular Boltzmann sampler, the size of the $\bar{1}$-subtree has a Catalan distribution, with mean value 1 and probability generating function*

$$p(u) = \frac{1}{\tau}\sum_k B_k \rho^k \tau^k u^k,$$

where ρ and τ are respectively the singularity and the singular value of the generating function of ternary trees, and B_k the kth Catalan number. For large values of k, the probability is asymptotic to

$$Pr(|\bar{1}\text{-subtree}| = k) \sim C\alpha^k \frac{k^{-3/2}}{\sqrt{\pi}} \qquad \text{with} \qquad \alpha = 8/9 \qquad \text{and} \qquad C = 2/3\sqrt{\pi}.$$

Proof. The substitution induced by equation (4.4) is $T(z,u) = B(zuT(z))$, where $B(t) = \sum B_n t^n$ is the generating function for binary trees: $B(t) = 1 + tB^2(t)$, and $B_n = \frac{1}{n+1}\binom{2n}{n}$. This substitution is non-critical since $\rho\tau < 1/4$, the radius of convergence of $B(t)$.

Hence the probability generating function for the size of the $\bar{1}$-subtree of a ternary tree generated with a singular Boltzmann sampler is

$$p(u) \equiv \frac{T(\rho,u)}{T(\rho,1)} = \frac{1}{\tau}\sum_k B_k \rho^k \tau^k u^k \tag{4.5}$$

and the probability evaluates to

$$\Pr(|\bar{1}\text{-subtree}| = k) = \frac{2}{3(k+1)}\binom{2k}{k}(2/9)^k. \tag{4.6}$$

Deriving equation (4.5), an easy computation shows that the mean value of the parameter under singular Boltzmann sampling is constant: $\mu \equiv p'(1) = \frac{(\tau-1)}{\tau(1-2\rho\tau^2)} = 1$.

Using Stirling approximation, the probability of equation (4.6) can be transformed as stated in the text of the proposition. □

Proof of theorem 4.4.3 By (4.3), the bivariate generating function for the degree at the center of a RANS is $D(z,u) = zu^3T^3(z,u)$, so that the probability for having R-degree k in a random ternary tree generated by a singular Boltzmann sampler, is

$$\Pr(D = k) = \frac{1}{\rho\tau^3}[u^k]\rho u^3 T^3(\rho,u) = \frac{1}{\tau^3}[u^k]u^3 B^3(u\rho\tau).$$

By Bürmann-Lagrange theorem, $[t^k]B^3(t) = \frac{3}{k}\binom{2k+2}{k-1}$, thus

$$\Pr(D = 3+k) = \frac{8}{9}\frac{1}{k+3}\binom{2k+2}{k}\frac{2^k}{9^k}.$$

And the power law with exponential cutoff comes from Stirling approximation:

$$\Pr(D = 3+k) \sim C\left(\frac{8}{9}\right)^k (k+3)^{-3/2}.$$

The mean value is computed by derivation: for ternary trees, the average number of nodes in the $\bar{1}$-subtree is $\rho B'(\rho\tau) = 1$. The mean degree of six for the center corresponds to the three edges from the center to the outermost vertices, plus the edges in each of the three sub-RANS.

This completes the proof for the degree distribution of the center in a RANS. Theorem 4.4.3 states that these statistics apply to the degree distribution of vertices in a RANS. And this result comes from corollary 4.4.2.

4.5 Extensions

The RAN model is very easy to extend, and so is the RANS model. High dimensional random Apollonian networks have been proposed and analysed by Zhange et al. [ZRC06], showing that increasing the dimension gives a wider degree distribution. Here we show that random Apollonian network structures can be extended to higher dimensions and have a similar behavior for large degrees. These extensions rely on using d-ary trees (d is the dimension) instead of ternary trees, and the generation and analysis develop along the same lines as for regular RANS.

The first step is to extend the bijection between RANS and ternary trees: RANS of dimension d (d-RANS) correspond to d-tuples of d-ary trees, and the degree of the center of the d-RANS still expresses in terms of the size of a d-tuple of $\bar{n}$-subtrees, which are (d-1)-ary subtrees. In the bijection, the trees under consideration are d-ary trees, with generating function $T_d(z) = 1 + zT_d^d(z)$, radius of convergence ρ_d and singular value τ_d.

Marking $\bar{1}$-subtrees gives the bivariate generating function $T_d(z,u)$, that satisfies

$$T_d(z,u) = 1 + uzT_d^{d-1}(z,u)T_d(z) = T_{d-1}(uzT_d(z))$$

which is still a non critical substitution schema, since $\rho_d\tau_d < \rho_{d-1}$. By proposition 4.3.1 the distribution is of Catalan type. Finally the bivariate function marking R-degree in d-ary trees is $D_d(z,u) = zu^dT_d^d(z,u)$, and the result stated below, follows from Bürmann-Lagrange theorem.

Proposition 4.5.1. *The degree distribution in random Apollonian network structures of dimension $d \geq 3$ has a Catalan form*

$$\Pr(D_d = d+k) = \frac{d}{\tau_d}\frac{1}{k(d-2)+d}\binom{k(d-1)+d-1}{k}\rho_d^k\tau_d^k$$

that is

$$\Pr(D_d = d+k) \sim C\,\alpha^k\left(k+\frac{d}{d-2}\right)^{-\frac{3}{2}} \quad \text{with} \quad C = \frac{d}{\sqrt{2\pi}}\left(\frac{d-1}{d-2}\right)^{\frac{3}{2}} \quad \text{and} \quad \alpha = \frac{(d-1)^{2d-3}}{d^{d-1}(d-2)^{d-2}}.$$

Remark 4.5.2. In the case $d = 2$ the subcritical schema substitutes trees in a list, leading to a geometric distribution.

Figure 4.5 shows the distribution of degrees in d-RANS for $d \in \{2,3,4,5,7,10,25\}$. Apart from the special case $d = 2$, every law is of the form $\alpha^k \times (k+\delta)^{-\frac{3}{2}}$ where α and δ tend to

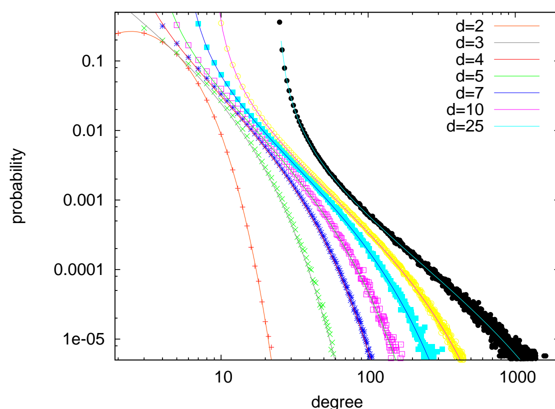


Figure 4.5: Degree distribution in high dimensional random Apollonian network structures

1 when d becomes large, so that the limiting law is a power law with parameter $3/2$. The adequacy between the theoretical distributions (solid lines) and the clouds of points resulting from Boltzmann sampling is almost perfect.

There are other interesting extensions, each one associated with a particular tree class. For example triangles can be reused, creating multiple layers, or different dimensions can be combined. These various refinements may be studied with the same tools and combine to conform to the behavior of real networks as far as degree is concerned. Moreover other relevant parameters of networks, such as mean distance, can also be dealt with in the same analytic framework; for example we show in a forthcoming work that the mean distance between vertices in a RANS is of order $\sqrt{n}$.

Cet article est tiré des actes de la conférence FPSAC08 [BDS08].

Chapter 5

Distances in random Apollonian network structures

In this paper, we study the distribution of distances in random Apollonian network structures (RANS), a family of graphs which has a one-to-one correspondence with planar ternary trees. Using multivariate generating functions that express all information on distances, and singularity analysis for evaluating the coefficients of these functions, we prove a Rayleigh limit distribution for distances to an outermost vertex, and show that the average value of the distance between any pair of vertices in a RANS of order n is asymptotically $\sqrt{n}$.

5.1 Introduction

Many graph models have been recently introduced for representing the structure and dynamics of real-life networks (see e.g. [NBW06]). Their adequacy to data can be measured by comparing some properties of graphs, especially the degree distribution of the vertices, which is related to scale-free properties, and properties related to the "small world" effect, such as distance between pairs of vertices and grouping in clusters.

The random Apollonian networks (RAN) proposed by [ZYW05] provides an interesting model, with a power-law degree distribution, a mean distance of logarithmic order and a large clustering coefficient. We introduced in [DS07] a modified version, random Apollonian network structures (RANS), which preserves the interesting properties of real graphs concerning degree distribution (a power-law with an exponential cut-off) and large clustering. This paper is devoted to the analysis of distances in RANS, which is showed to be of square root order. This result may seem disappointing in the scope of "small world" networks, however one only knows that the distance in real networks is small, not necessarily logarithmic.

Anyhow, there is no surprise with the square root distance since our underlying model is simple families of trees, whereas the model underlying RAN is increasing trees (that lead to a logarithmic distance [Zha+08]). Relying on these different tree models and using probabilistic methods, a unified study of various parameters in RAN and RANS was concurrently done [MA08].

A RANS can indeed be seen as a certain type of triangulation of a triangle, and the study of RANS relies on the bijection with planar ternary trees (see figure 5.1). From this bijection we can express the enumerative generating function for RANS, and use multivariate functions for marking several distance parameters. Moreover the asymptotic values of the quantities under consideration can be dealt with using singularity analysis (according to methods developed in [FS09]).

We are interested in two types of parameters measuring distance, and develop two methods to handle them. We first attack the distances between a special vertex (an outermost vertex A of the RANS) and all the other vertices. The method is built on computing a series of generating functions with increasingly many variables, that contains all informations concerning distances from A to the other vertices. Distribution analysis is based on the study of partial derivatives of this multivariate series, which correspond to the series counting the number of vertices at a certain distance from A. These series all express in terms of the generating function for RANS and asymptotic analysis gives a Rayleigh limit distribution with a mean value of order $\sqrt{3\pi n}/11$.

The second study addresses the total distance between all pairs of vertices. We exhibit a generating function in four variables that expresses simultaneously distances from one, two or three outermost vertices. This generating function has a nice recursive definition, due to the symmetries of the problem. It contains all information to compute the total distance between pairs of vertices. Geometrical considerations splits this total distance in two parts, depending on whether a path between two vertices spans over disjoints sub-RANS or not. The resulting mean distance between two vertices is of order $\sqrt{3\pi n}/11$.

This paper divides in four sections: this introduction, followed by a section that recalls the definition of random Apollonian network structures, the bijection with ternary trees, and the result for degree distribution. Section 3 describes the distribution of distances from an outermost vertex and section 4 is dedicated to the study of the total distance between all pairs of vertices.

5.2 Random Apollonian network structures

The recursive definition of RANS shows a one-to-one correspondence with ternary trees. The degree distribution, which is a power law with an exponential cut-off, was studied in [DS07] by considering bivariate series marking the corresponding parameter in trees.

5.2.1 Bijection with ternary trees

A random Apollonian network structure (RANS) R is recursively defined as: either an empty triangle or a triangle T split in three parts, by placing a vertex v inside T and connect it to the three vertices of the triangle; each sub-triangle being substituted by a RANS (see figure 5.1).

The vertices of T will be called the *outermost vertices* of R (noted $\mathcal{O}(R)$); and vertex v will be called the *center* of R. We will note $\mathcal{R}$ the class of all RANS.

The *order* of the empty RANS is zero and the order of a non-empty RANS is one plus the sum of the orders of the three sub-RANS.

Proposition 5.2.1. *[DS07] There is a bijection between random Apollonian network structures of order N and rooted plane ternary trees of size N (with N internal nodes).*

In planar ternary trees, the linear ordering of siblings is relevant. This order is carried over to triangles: naming $\{O_1, O_2, O_3\}$ the vertices of $\mathcal{O}(R)$, imposes a linear ordering on the sub-RANS ($\{S_1, S_2, S_3\} = \mathcal{S}(R)$): S_i will be the one not containing O_i. Recursively replacing the missing outermost vertex by the center of R preserves the order in sub-RANS.

The generating function for ternary trees $T(z) = \sum T_N z^N$, where T_N is the number of trees with N internal nodes, satisfies the functional equation $T(z) = 1 + zT^3(z)$. $T(z)$ has radius of convergence $\rho = 4/27$ and singular value $\tau = 3/2$; and the singular expansion of $T(z)$ near ρ is

$$T(z) = \frac{3}{2} - \frac{\sqrt{3}}{2}\sqrt{1 - z/\rho} + \frac{2}{3}(1 - z/\rho) - \frac{35\sqrt{3}}{108}(1 - z/\rho)^{3/2} + O\left((1 - z/\rho)^{5/2}\right). \quad (5.1)$$

Thus the asymptotic form of the coefficients: $T_N \sim c\rho^{-N} N^{-3/2}$, with $c = \sqrt{3}/4\sqrt{\pi}$.

The derivative $T'(z) = \frac{T^3(z)}{1-3zT^2(z)}$ will also appear in the computations below. The leading term in its singular expansion is $\frac{\sqrt{3}}{4\rho}(1 - z/\rho)^{-1/2}$, thus a coefficient T'_N of asymptotic order $\frac{27\sqrt{3}}{16\sqrt{\pi}}\rho^{-N} N^{-1/2}$.

5.2.2 Degree distribution

The degree distribution in random Apollonian network structures follows a power law with an exponential cutoff. This is obtained [DS07] by analysing the degree of the center of a RANS, and propagating this study to the whole of the sub-RANS.

The bivariate generating function marking the degree of the center is $D_g(z,u) = zu^3 T^3(z,u)$, where $T(z,u)$ is the bivariate generating function for ternary trees with u marking the degree of an outermost vertex in the corresponding RANS:

$$T(z,u) = 1 + uzT(z)T^2(z,u). \quad (5.2)$$

The degree of a vertex is exactly the number of vertices at distance 1 from this vertex, the generating functions used in the following section will be generalizations of $T(z,u)$.

5.3 Distance from an outermost vertex

This section is devoted to computing the distribution of the distances from a fixed specific outermost vertex. We introduce a generating function with infinitely many variables, each variable u_i marking the number of vertices at distance i from the outermost vertex. Relying on the symmetries of the problem and the recursive nature of RANS, we are able to express and study this generating function.

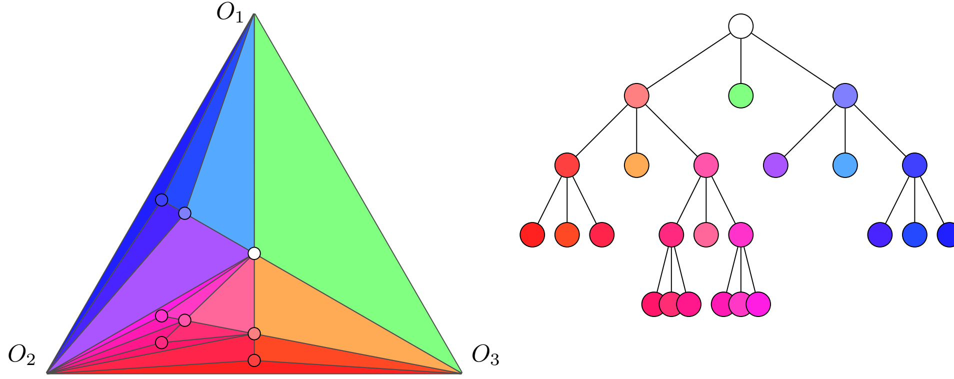


Figure 5.1: A random Apollonian network and its corresponding ternary tree

The interest of this analysis is not only to find, with a different method, some results of the following section; but moreover this study can be adapted to compute the distribution of distances to *any distinguished vertex* in a RANS, which may be considered as a more realistic parameter.

5.3.1 Multivariate generating function

Due to the recursive nature of RANS, we often have to consider a RANS R as having an *environment*, that is a bigger RANS containing R. Given a RANS R, the distance of any of its vertices to a vertex v in the environment of R is determined by the three distances of the elements of $\mathcal{O}(R)$ to v.

Since the outermost vertices of R form a clique, their distances to any vertex cannot differ by more than one. This observation allows us to reduce our study to a few cases. First we work modulo a translation and restrict ourselves to the case when the three distances to $\mathcal{O}(R)$ are either 1 or 0. Second we can work modulo a permutation and restrict ourselves to only three cases: $(0,1,1)$, $(0,0,1)$ and $(0,0,0)$, illustrated in figure 5.2. These three cases actually correspond to labelling the internal vertices of R by their distances either to one (out of three), or to two (out of three) or to all three outermost vertices.

Definition The $\delta_{①}$-*labeling* of $R \in \mathcal{R}$ consists in putting on each vertex a label corresponding to its distance from $O_1(R)$ (or equivalently to one of any $O_i(R)$):

- the outermost vertices O_1, O_2, O_3 respectively receive labels 0, 1 and 1;
- the center of $R' \in \mathcal{S}(R)$ is labeled by 1 plus the minimum of the labels of $\mathcal{O}(R')$.

Definition Let us define the *type of* $R \in \mathcal{R}$ as the 3-tuple of labels of $\mathcal{O}(R)$. We say that

- two RANS types are *equivalent* iff they have the same type up to a permutation.
- two RANS types are *translated* by θ iff their labellings are the same up to a translation θ.

In a RANS R of type $(0,1,1)$ the center gets label 1. Thus R is equivalent to $S_2(R)$ and $S_3(R)$, but it is not equivalent to $S_1(R)$, which is of type $(1,1,1)$: if a vertex is at distance d from $O_1(S_1(R))$, its distance from $O_1(R)$ is $d+1$.

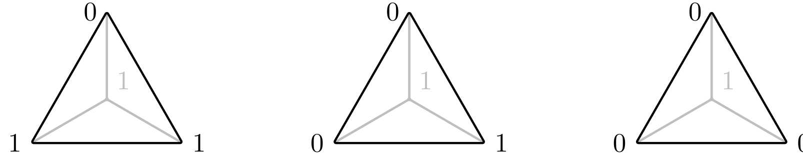


Figure 5.2: RANS of type $(0,1,1),(0,0,1),(0,0,0)$ and their sub-RANS.

This remark leads to a recursive definition of $T_1(z,u_1) = \sum T_{n,k} u_1^k z^n$, where $T_{n,k}$ is the number of RANS of order n with k vertices at distance 1 from $O_1(R)$:

$$T_1(z,u_1) = 1 + zu_1 T_1^2(z,u_1) T(z). \tag{5.3}$$

This equation follows the recursive definition of RANS, noticing first that the center is at distance 1 from $O_1(R)$ and second that the configuration is the same in $S_2(R)$ and $S_3(R)$, whereas in $S_1(R)$ there is no vertex at distance 1 from $O_1(R)$.

The problem of marking *both* vertices at distance 1 and 2 from $O_1(R)$ is treated in the same way: the configuration of R, of type $(0,1,1)$, recursively occurs in $S_2(R)$ and $S_3(R)$, which are of the same type. But the case of $S_1(R)$, of type $(1,1,1)$, is a little more tricky and requires a deeper decomposition. If $S_1(R)$ is not empty, its center is at distance 2 from $O_1(R)$, and its three sub-RANS $\mathcal{S}(S_1(R))$ are equivalent, of type $(1,1,2)$: either they are empty, or their center is at distance 2 from $O_1(R)$, one sub-RANS is of type $(1,1,2)$, equivalent to $\mathcal{S}(S_1(R))$ and the two others are of type $(1,2,2)$, translated by 1 with R, which means that the number of their vertices at distance 2 from $O_1(R)$ in is the same as the number of vertices at distance 1 from $O_1(R)$ in R.

This decomposition leads to the following functional equations, with u_j marking vertices at distance j from $O_1(R)$:

$$\begin{aligned} T_2(z,u_1,u_2) &= 1 + zu_1 T_2^2(z,u_1,u_2) F(z,u_2) \\ F(z,u_2) &= 1 + zu_2 G^3(z,u_2), \\ G(z,u_2) &= 1 + zu_2 G(z,u_2) T_1^2(z,u_2). \end{aligned}$$

We go from d to $d+1$ the same way as from 1 to 2 and the preceding equations hold for $d \geq 3$, when considering multivariate generating functions $T(z,u_1,u_2,\ldots,u_d)$, with u_j marking vertices at distance j from $O_1(R)$, and we get the following result.

Proposition 5.3.1. *Let $r_{n,k_1,\ldots,k_d}$ be the number of RANS of order n with k_j vertices at distance j from O_1. Then $r_{n,k_1,\ldots,k_d}$ is the coefficient of $u_1^{k_1} u_2^{k_2} \ldots u_d^{k_d} z^n$ in the multivariate series $T_d(z,u_1,\ldots,u_d)$, where the series T_d satisfy the recurrence relations:*

$$T_1(z,u_1) = 1 + zu_1 T_1^2(z,u_1) T_0(z) \qquad \textit{with} \qquad T_0(z) = T(z)$$

and for $d \geq 2$,

$$T_d(z,u_1,\ldots,u_d) = 1 + zu_1 T_d^2(z,u_1,\ldots,u_d) \left(1 + zu_2 \frac{1}{(1 - zu_2 T_{d-1}^2(z,u_2,\ldots,u_d))^3} \right).$$

All information concerning distances from vertex O_1 can be retrieved from proposition 5.3.1:

- The enumerative series for the number of vertices at distance i from O_1, over all RANS, is
$$D_i(z) = \frac{\partial}{\partial u_i} T_i(z, u_1, \ldots, u_i)\bigg|_{u_j=1,\forall j} = \sum_n k_i r_{n,k_i} z^n.$$
- The total distance from O_1 expresses as
$$\frac{\partial}{\partial u} D(z,u)\bigg|_{u=1}, \qquad \text{where } D(z,u) = \sum_{i=1}^{\infty} D_i(z) u^i.$$

The aim of the next paragraph is to evaluate these quantities.

5.3.2 Distribution analysis

Generating functions counting the number of vertices at distance i from O_1 express as rational functions in z and $T(z)$, and have a singular behaviour similar to $T(z)$: radius of convergence ρ, and singular expansion of the square-root type.

Lemma 5.3.2. *The sequence of enumerative series for the number of vertices at distance i from O_1 is:*

$$\begin{aligned} & D_1(z) = zT^3(z)/(1-2zT^2(z)), \\ & D_2(z) = H(z) \times (1+2z^2T^4(z))/(6zT(z)(1-2zT^2(z))) \\ \textit{and for } i \geq 2 \quad & D_{i+1}(z) = H^{i-1}(z) \times D_2(z), \end{aligned}$$

where $H(z) = 6z^2(T(z)-1)T(z)/(1-3z-zT(z)-zT^2(z)+2z^2T^2(z))$ [1]*. $H(z)$ has a singular expansion $H(z) = 1 - \frac{11}{\sqrt{3}}\sqrt{1-z/\rho} + \frac{2}{3}(1-z/\rho) + (1-z/\rho)^{3/2} + O((1-z/\rho)^2)$, with radius of convergence $\rho = 4/27$.*

Proof. From proposition 5.3.1, it is easy to compute the expressions of $D_1(z)$ and $D_2(z)$, and show that $D_{i+1}(z) = H(z, T(z)) \times D_i(z)$. The singular expansion comes from expressing z as $(T(z)-1)/T^3(z)$ and plugging in H the singular expansion (5.1) of $T(z)$. A full expansion of $T(z)$ yields a full expansion for $H(z)$, the first terms of which are given in the lemma. □

Theorem 5.3.3. *Given R a RANS of order n and v a random internal vertex of R, the distance from v to $O_1(R)$ has a Rayleigh limit distribution:*

$$\Pr(\mathrm{dist}(v, O_1(R)) = x\sqrt{n}) = c\frac{x}{\sqrt{n}} e^{-c^2\frac{x^2}{4}}.$$

1. Since $\mathbb{Q}[z,T(z)]/\langle T(z)-1-zT^3(z)\rangle$ is a $\mathbb{Q}(z)$-vector space with dimension three, all rational functions in z and $T(z)$ that appear in this paper can be expressed in a canonical form. However we didn't use it since it usually hides the combinatorial interpretation of the generating functions under consideration.

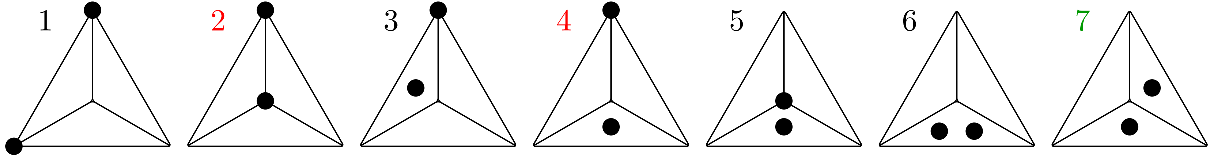


Figure 5.3: All possible configurations for pairs of vertices in $R \in \mathcal{R}$. For each pair in $\mathcal{C}(R)$ we can always find a unique sub-RANS of R such that the two vertices are in one of configurations 2, 4 or 7. The other cases reduce to one of these: case 1 reduces to 2 by looking at the RANS containing R, case 3 leads to 2, 3 or 4 by looking in the sub-RANS containing the two vertices, and such is also the case for case 5. Case 6 is amenable to any of 5, 6 or 7 by looking at the sub-RANS containing the two vertices.

Proof. The full singular expansion of $D_i(z)$ can be derived from its expression in terms of H and D_2. Thus the proportion of vertices at distance i from O_1, that is $\frac{1}{nT_n}[z^n]D_i(z)$ can be evaluated:

$$\Pr(\mathrm{dist}(v, O_1(R)) = i) = \frac{1}{nT_n}[z^n]D_i(z) = \frac{1}{nT_n}[z^n]H^{i-2}(z)D_2(z).$$

The result follows from theorem IX.16 (Semi-large powers) of [FS09]: the singular exponent $1/2$ for $H(z)$ implies a Rayleigh distribution for $k = x\sqrt{n}$. □

The average distance from O_1 is also obtainable by differentiation of $D(z,u) = \sum D_i(z)u^i$. From lemma 5.3.2 there is a closed form for $D(z,u)$, and differentiation leads (fortunately) to the same series as the one obtained for $\Delta_{①}(z)$, in section 5.4.1.

This series has a singular expansion around ρ with first term $\frac{3}{44}(1-z/\rho)^{-2}$, so for the mean distance

$$\frac{1}{nT_n}[z^n]\left.\frac{\partial}{\partial u}D(z,u)\right|_{u=1} = \frac{\sqrt{3\pi n}}{11}\left(1 + O(\frac{1}{n})\right).$$

We thus conclude this section with the following proposition:

Proposition 5.3.4. *In a RANS of order n, the average distance from O_1 is of order $c\sqrt{n}$, with $c = \sqrt{3\pi}/11$.*

5.4 Total distance between pairs of vertices

In this section, we are interested in computing the total distance of every pair of vertices in a RANS of order n, and will show that the mean value of this quantity is still of order $\sqrt{n}$.

We call $\mathcal{C}(R)$ the set of pairs of vertices (we call pair a set of size two) in $R \in \mathcal{R}$, excluding pairs where both vertices are in $\mathcal{O}(R)$.

The enumerative generating function for the total distance between pairs in $\mathcal{C}(R)$ is

$$G(z) = \sum_{R \in \mathcal{R}} \sum_{(v,w) \in \mathcal{C}(R)} \mathrm{dist}(v,w)\, z^{|R|}.$$

$\mathcal{C}(R)$ splits into two parts

- the pairs (v,w) such that they are both *internal* vertices of the smallest sub-RANS of R that contains both of them, corresponding to case 7 in figure 5.3.
  We will note them $\mathrm{Inter}(R)$ and their contribution to the total distance will be called *interdistance*.
- the others, which can also be defined as the pairs (v,w) such that there exists a sub-RANS S of R with v an outermost vertex of S and w an internal vertex of S, corresponding to cases 2 and 4 in figure 5.3.
  We will note them $\mathrm{Intra}(R)$ and their contribution to the total distance will be called *intradistance*.

Remark: $\mathcal{C}(R)$ has $n(n-1)/2+3n$ elements: each pair of internal vertices is counted only once, and the $3n$ term takes into account all pairs made of one internal vertex and one outermost vertex. Among all these pairs, an amount of order $n\sqrt{n}$ belongs to $\mathrm{Intra}(R)$ and the rest is in $\mathrm{Inter}(R)$. As we will show, the total distance of pairs in $\mathrm{Intra}(R)$ is of order n^2 and the total distance of pairs in $\mathrm{Inter}(R)$ is of order $n^2\sqrt{n}$. We can thus say that the interdistance gives the dominant term of the total distance in RANS, which is of order $\sqrt{n}$.

We introduce in the following subsection a new generating function which serves as a basis for the computations of all the quantities that are needed. Then we calculate the intradistance followed by the interdistance. Putting everything together gives the following result.

Theorem 5.4.1. *The mean distance between two vertices in a RANS of order n is asymptotically equivalent to $C\sqrt{n}$, with $C=\sqrt{3\pi}/11$.*

5.4.1 Cumulated weights generating function

Given $R \in \mathcal{R}$, the distances of inner vertices to $\mathcal{O}(R)$ are denoted by the three following parameters:

$$\Delta_{①}(R)=\sum_{x\in R} d(x,O_1(R)),\ \Delta_{②}(R)=\sum_{x\in R} d(x,\{O_1(R),O_2(R)\}) \text{ and } \Delta_{③}(R)=\sum_{x\in R} d(x,\mathcal{O}(R)).$$

Notice that $\Delta_{①}(R)$ is the sum of all the labels in the $\delta_{①}$-labeling of R, based on RANS of type $(0,1,1)$ —see (5.3.1). Similarly, $\Delta_{②}(R)$ is the sum of all the labels in the $\delta_{②}$-labeling of R, that starts with a RANS of type $(0,0,1)$; and $\Delta_{③}(R)$ is the sum of all the labels in the $\delta_{③}$-labeling of R, starting with a RANS of type $(0,0,0)$.

In the following generating function, parameter $\Delta_{ⓘ}$ is marked by variable $d_{ⓘ}$:

$$\Delta(z,d_{①},d_{②},d_{③})=\sum_{R\in\mathcal{R}} d_{①}^{\Delta_{①}(R)} d_{②}^{\Delta_{②}(R)} d_{③}^{\Delta_{③}(R)} z^{|R|}=\sum_{n,i,j,k=0}^{\infty} \alpha_{n,i,j,k}\, d_{①}^{i} d_{②}^{j} d_{③}^{k} z^{n},$$

where $\alpha_{n,i,j,k}$ is the number of RANS of order n (*i.e.* with n internal points), and respective values i,j,k for parameters $\Delta_{①},\Delta_{②},\Delta_{③}$. This generating function is called the *cumulated weights generating function* since it expresses the distances according to the three different types of RANS.

Proposition 5.4.2. *The cumulated weights generating function satisfies the recursive equation*

$$\begin{aligned}\Delta(z,d_①,d_②,d_③) = 1 + zd_①d_②d_③ &\times \Delta(zd_①,d_②,d_③,d_①)\\ &\times \Delta(z,d_①,d_②d_③,1)\\ &\times \Delta(z,d_①d_②,d_③,1).\end{aligned}$$

Proof. Let's follow the recursive definition of RANS R. If R is empty the contribution to the series is 1. Otherwise it has a center, which is at distance 1 from each of the outermost vertices (hence the factor $zd_①d_②d_③$) and the contributions come from the 3 sub-RANS.

Factor $\Delta(zd_①,d_②,d_③,d_①)$ comes from $S_1(R)$. Suppose $S_1(R)$ has, by itself, a generating series $\Delta(z,d_①,d_②,d_③)$, that corresponds to the three different labellings, with types $(0,1,1)$, $(0,0,1)$ and $(0,0,0)$. When it is considered as embedded as the first sub-RANS of R, the top most vertex has label 1 instead of 0, so that the three different labellings now start with types $(1,1,1)$, $(1,0,1)$ and $(1,0,0)$. Thus variable $d_①$ transforms into $d_②$, variable $d_②$ transforms into $d_③$, and variable $d_③$ transforms into $d_①$ with a 1-translation.

Factor $\Delta(z,d_①,d_②d_③,1)$ comes from $S_2(R)$. Suppose $S_2(R)$ had, by itself, a generating series $\Delta(z,d_①,d_②,d_③)$, corresponding to the three different types $(0,1,1)$, $(0,0,1)$ and $(0,0,0)$. When it is considered as embedded as the second sub-RANS of R, $O_2(S_2(R))$ has to have label 1, so that the three different labellings now start with types $(0,1,1)$, $(0,1,1)$ and $(0,1,0)$. Thus variables $d_②$ and $d_③$ transform into $d_②$, variable $d_①$ stays $d_①$, and nothing transforms to $d_③$.

Factor $\Delta(z,d_①d_②,d_③,1)$ comes from $S_3(R)$, and the proof is equivalent. □

The series of cumulated distances from A is obtained by differentiation

$$\Delta_①(z) = \sum_{R\in\mathcal{R}} \Delta_①(R)z^{|R|} = \frac{\partial}{\partial d_①}\Delta(z,d_①,1,1)\bigg|_{d_①=1}$$

and the same holds for the two other cases.

Proposition 5.4.3. *The distance generating functions $\Delta_{ⓘ}(z)$ have the following expressions:*

$$\begin{aligned}\Delta_①(z) &= zT^3(z)(1-2zT^2(z)+z^2T^4(z)-6z^3T^6(z))/Q(z,T(z))\\ \Delta_②(z) &= zT^3(z)(1-3zT^2(z)+4z^2T^4(z)-6z^3T^6(z))/Q(z,T(z))\\ \Delta_③(z) &= zT^3(z)(1-3zT^2(z)+2z^2T^4(z))/Q(z,T(z)),\\ &\textit{where } Q(z,T(z)) = (1+2z^2T^4(z))(1-3zT^2(z))^2.\end{aligned}$$

Each $\Delta_{ⓘ}(z)$ has radius of convergence ρ and a singular expansion of the form:

$$\Delta_{ⓘ}(z) = 3/(44(1-z/\rho)) + O((1-z/\rho)^{-1/2}).$$

Proof. The distance generating functions $\Delta_{ⓘ}(z)$ satisfy the system of equations:

$$\begin{cases}\Delta_①(z) &= zT^3(z)+2z\Delta_①(z)T^2(z)+zT^2(z)(zT'(z)+\Delta_①(z))\\ \Delta_②(z) &= zT^3(z)+2z\Delta_①(z)T^2(z)+z\Delta_③(z)T^2(z)\\ \Delta_③(z) &= 3z\Delta_②(z)T^2(z)+zT^3(z)\end{cases}$$

where $T'(z) = T^3(z)/(1-3zT^2(z))$.

The resolution of the system shows that each $\Delta_{ⓘ}(z)$ has a dominant term that expresses as $T'^2(z)$ with the same constant factor, thus a pole in $z=\rho$. The singular expansions only differ on their second term. □

5.4.2 Intradistance

We first consider the pairs (v,w) such that there exists a sub-RANS S of R with v an outermost and w an internal vertex of S. There may be many embedded sub-RANS S and we focus on the smallest one, S_0. In S_0, vertex v is outermost (e.g. O_1) and w is either the center of S_0 or in the sub-RANS opposite to v (e.g. $S_1(S_0)$) (cf. cases 2 and 4 in figure 5.3).

We will first study the pairs $\mathrm{Intra}_1(R)$, for which $S_0 = R$, and then recursively extend the computation to the rest of the intradistance.

Lemma 5.4.4. *The generating function for the total distance of pairs in* $\mathrm{Intra}_1(R)$*, satisfies*

$$\delta(z) = 3T(z) + 3zT^2(z)\Delta_{③}(z) + 3z^2T^2(z)T'(z).$$

Proof. The distance of $\mathrm{Intra}_1(R)$ is made of two categories of distances:

- from the center of R to each of the outermost vertices $\mathcal{O}(R)$,
- from each outermost vertex $O_i(R)$ to all the internal vertices of its opposed sub-RANS, $S_i(R)$.

The distance from an outermost vertex $O_i(R)$ to an internal vertex w of $S_i(R)$ is $1+d(w,\mathcal{O}(S_i(R)))$. Thus, the distance from an outermost vertex $O_i(R)$ to all the internal vertices of $S_i(R)$ is $|S_i(R)| + \Delta_{③}(S_i(R))$. Taking into account all three sub-RANS of R we have

$$\delta(z) = \sum_{R\in\mathcal{R}} \left(3 + \sum_{S\in\mathcal{S}(R)} (\Delta_{③}(S) + |S|)\right) z^{|R|},$$

thus the expression of the generating function as stated in the lemma. □

Theorem 5.4.5. *The generating function for intradistances in a RANS is* $\mathrm{Intra}(z) = \delta(z)/(1-3zT^2(z))$ *and the total distance between pairs of vertices in* $\mathrm{Intra}(R)$*, for* $R \in \mathcal{R}_n$*, is asymptotically* $\frac{3}{11}n^2$.

Proof. The total intradistance is obtained by recursively computing intradistances at any level of the RANS. The effect of this recursion process, akin to recursive descent in subtrees of ternary trees, is to multiply the generating function by $T'(z)/T^3(z)$, that is $1/(1-3zT^2(z))$. The dominant term in the singular expansion of $\mathrm{Intra}(z)$ thus is $3z\Delta_{③}(z)T'(z)/T(z)$. The total distance in $\mathrm{Intra}(R)$ is obtained by evaluating $[z^n]\mathrm{Intra}(z)/T_n$. □

5.4.3 Interdistance

We now consider the pairs (v,w) such that they are both *internal* vertices of the smallest sub-RANS S of R that contains both of them.

Since S is minimal by inclusion, v and w are in different sub-RANS of S, which we will call S_v and S_w. The shortest path from v to w can be decomposed in three sub-paths: from v to $\mathcal{O}(S_v)$, from $\mathcal{O}(S_w)$ to w and, if these two sub-paths are disjoint, an one-edge path (on the border of S_v, S_w or S). We will call this last part the *f-edge*. This decomposition is illustrated in figure 5.4.

We will first compute a lower bound $\mathrm{Inter}^-(R)$ of the interdistance by neglecting the f-edges. This lower bound gives a total distance on pairs of $\mathrm{Inter}(R)$, with $R \in \mathcal{R}_n$ order $n^2\sqrt{n}$.

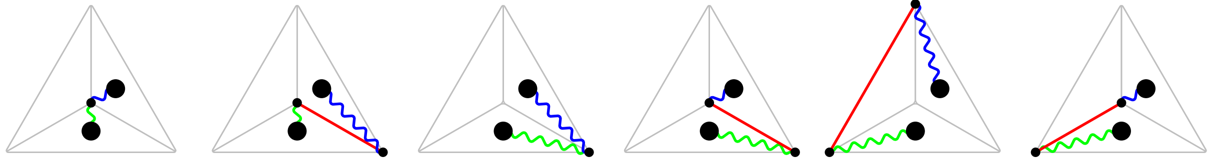


Figure 5.4: The different scenarios for paths between pairs of vertices in Inter(R). The three colors correspond to the three sub-paths, the red one being the *f-edge*.

We can also take an upper bound $\text{Inter}^+(R)$ by forcing every path to pass from the center of R and this still gives a contribution of order $n^2\sqrt{n}$ with the same factor. Counting the exact number of f-edges allows us to compute the following terms of the interdistance.

Lower bound and upper bound

As for the intradistance we first compute a lower (resp. upper) bound on interdistances at the topmost level $\text{Inter}_1(R)$ (i.e. $S_v, S_w \in \mathcal{S}(R)$) and extend it recursively for the whole RANS.

Lemma 5.4.6. *The generating function for lower bound (resp. upper bound) of the total distance of pairs in* $\text{Inter}_1(R)$, $\gamma^-(z)$ *(resp.* $\gamma^+(z)$*), satisfies*

$$\gamma^-(z) = 6z^2T(z)T'(z)\Delta_{③}(z) \qquad \gamma^+(z) = 6z^2T(z)T'(z)\Delta_{①}(z).$$

Proof. At level one, for each sub-RANS, the contribution to the interdistance is the total length of the sub-paths contained in this sub-RANS. Thus for each vertex v, situated in a sub-RANS $S_1(R)$, we will count its distance to $\mathcal{O}(S_1(R))$, multiplied by the number of vertices in $S_2(R)$ and $S_3(R)$. The lower bound is obtained by adding all these values (parameter $\Delta_{③}$), and for the upper bound we consider that the frontier is reduced to only one of its two points (parameter $\Delta_{①}$). Thus the expression of the generating function $\gamma^-(z)$ (and γ^+ is obtained by replacing $\Delta_{③}$ by $\Delta_{①}$):

$$\gamma^-(z) = 3\sum_{R\in\mathcal{R}} \Delta_{③}(S_1(R)) \times (|S_2(R)| + |S_3(R)|)\, z^{|T|}.$$ □

Theorem 5.4.7. *The generating function for the lower bound (resp. upper bound) of interdistances in a RANS is*

$$\text{Inter}^-(z) = \frac{\gamma^-(z)}{1-3zT^2(z)} \qquad \text{Inter}^+(z) = \frac{\gamma^+(z)}{1-3zT^2(z)}$$

and in both cases the total distance between pairs of vertices in Inter(R)*, for* $R \in \mathcal{R}_n$*, is asymptotically* $Cn^2\sqrt{n}$ *with* $C = \sqrt{3\pi}/22$.

Proof. The proof is similar to the proof of theorem 5.4.5. The generating functions $\text{Inter}^-(z)$ and $\text{Inter}^+(z)$ have both a dominant term in $6\Delta_{ⓘ}(z)z^2T'^2(z)/T^2(z)$. □

Exact computation

To count f-edges we consider the shortest way, for a vertex v in S, to reach the outermost vertices of S. Let us denote by $\mathcal{X}(S,v)$ the subset of $\mathcal{O}(S)$ of exit points for v out of S. Let $(v,w) \in \text{Inter}(R)$, the shortest path from v to w will contain a f-edge iff $\mathcal{X}(S_v,v) \cap \mathcal{X}(S_w,w) = \emptyset$.

Given a RANS R, we note by $\mathcal{E}_i(R), i \in \{1,2,3\}$ the number of vertices inside R with i exit points out of R, divided by $\binom{3}{i}$ (to take symmetries into account). The corresponding generating functions are $E_i(z) = \sum_{R \in \mathcal{R}} |\mathcal{E}_i(R)| z^{|R|}$.

Lemma 5.4.8. *The generating function for the number of f-edges in the total distance of* $\text{Inter}_1(R)$ *is*

$$\phi(z) = 3zT(z)\left(7E_1^2(z) + 10E_1(z)E_2(z) + 2E_2^2(z) + 2E_2(z)E_3(z)\right).$$

Proof. In order to count the number of pairs in $\text{Inter}_1(R)$ for which $\mathcal{X}(S_v,v) \cap \mathcal{X}(S_w,w) = \emptyset$, we need to enumerate all the possible values of $\mathcal{X}(S_v,v)$ and $\mathcal{X}(S_w,w)$. There are seven possibilities for each of them, thus forty-nine combinations from which twenty-one produce a f-edge. The number of times each of the twenty-one combinations arises is $E_i(S_v)E_j(S_w)(|S| - |S_v| - |S_w|)$, if we note $i = |\mathcal{X}(S_v,v)|$ and $j = |\mathcal{X}(S_w,w)|$. □

Calculating $E_i(z)$. We use a multivariate generating function $T_e(z,u_1,u_2,u_3)$ for RANS marked with vertices in $\mathcal{E}_i$. This will be defined by a system of thirty three equations in the same spirit as in section 5.3.1. The analysis of this system is too long to be included in this abstract, but it leads to rational functions in terms of T(z) and singular expansions around ρ which are respectively equivalent to $\frac{3\sqrt{3}}{22}(1-z/\rho)^{-1/2}$, $\frac{3\sqrt{3}}{44}(1-z/\rho)^{-1/2}$, and $\frac{\sqrt{3}}{22}(1-z/\rho)^{-1/2}$.

Theorem 5.4.9. *The generating function for the number of f-edges in a RANS is* $F(z) = \phi(z)/(1-3zT^2(z))$ *and the total number of f-edges in R, for* $R \in \mathcal{R}_n$, *is asymptotically* $\frac{29}{121}n^2$.

Proof. The proof is similar to theorem 5.4.5. But in this case, each term of $\phi(z)$ gives a part of the dominant contribution. □

5.4.4 Conclusion

Summing all contributions, the enumerating generating function for the total distance between pairs of vertices $G(z) = \text{Intra}(z) + \text{Inter}^-(z) + F(z)$ expresses (with $t = T(z)$) as:

$$G(z) = \frac{(180 - 828t + 1428t^2 - 488t^3 - 2029t^4 + 3649t^5 - 2826t^6 + 1086t^7 - 168t^8)t}{(3t^2 - 4t + 2)^2(2t-3)^4}.$$

We made an exhaustive study of the different parts of $G(z)$. The contribution coming from intradistances happens to be of smaller order (n^2) than the contribution coming from interdistances ($n^2\sqrt{n}$). In the computation of interdistances, we first considered approximations that give a lower and an upper bound with the same dominant term ($\frac{\sqrt{3\pi}}{22}n^2\sqrt{n}$) that is a mean distance in $\frac{\sqrt{3\pi}}{11}\sqrt{n}$. The study of f-edges provides an exact computation of the total distance. With this contribution of f-edges it is possible to express the second term in the asymptotic expression of the total distance. Moreover, relying on full singular expansion of all series under consideration, it is possible to give a full asymptotic expansion of the total distance.

Cet article est tiré des actes de la conférence IWOCA09 [DS09].

Chapter 6

Limiting Distribution for Distances in *k*-trees

This paper examines the distances between vertices in a rooted k-tree, for a fixed k, by exhibiting a correspondence with a variety of trees that can be specified in terms of combinatorial specifications. Studying these trees via generating functions, we show a Rayleigh limiting distribution for distances to a random vertex in a random k-tree: in a k-tree on n vertices, the proportion of vertices at distance $d = x\sqrt{n}$ from a random vertex is asymptotic to $\frac{c_k^2 x}{\sqrt{n}} \exp(-\frac{c_k^2 x^2}{2})$, where $c_k = kH_k$.

6.1 Introduction

This work takes place within the general framework of analyzing statistical properties of combinatorial structures: we evaluate distances between vertices in a graph structure named k-tree.

In graph theory, very important research is being done on k-trees, for their characterization [Ros74; MJP06] and from an algorithmic viewpoint, since many NP-complete problems can be solved linearly on k-trees [AP89]. The class of k-trees, together with many close families, have also been extensively studied as combinatorial structures for their enumeration [BP69; Moo69; FGLL02; LLL04].

Our interest in k-trees focuses on their graph structure and the behavior of parameters such as degree or distance. This work is a generalization of our study of planar 3-trees [BDS08], and the results presented here can be easily extended to planar k-trees.

Here we are interested in quantifying the distribution of distances between two random vertices in a random k-tree, for a fixed k. Our main advantage for addressing this problem is a bijection between k-trees and a class $\mathcal{K}$ which is specifiable in terms of combinatorial constructions and thus amenable to the powerful tool of generating functions which combines algebraic methods for constructing relevant power series and analytic methods for evaluating parameters of interest.

Our bijection extends works by Klawe et al. [KCP82] and Ibarra [Iba04] providing a one

to one correspondence that exploits the recursive structure of k-cliques to transform the k-tree graph into a labeled tree structure. Moreover the parameter "k-tree distance to the root" transfers to a clearly identifiable parameter on subtrees of the tree structure, that can be precisely analysed by bivariate generating functions and leads to an asymptotic distribution that obeys a Rayleigh law. In order to deal with distances between random pairs of vertices we use a tree "rerooting" process that is also expressible in terms of generating functions and the same result of a Rayleigh asymptotic law still holds.

Though the tree parameter that we are studying is not a profile, its behavior is similar to profiles in simple varieties of trees, where Rayleigh distributions were first shown by Meir and Moon [MM78], and further investigated by Drmota and Gittenberger [DG97].

While the class $\mathcal{K}$ that we consider here is a class of labeled trees corresponding to all k-trees on n vertices, the class of label-increasing trees of $\mathcal{K}$ would also be of interest: it would correspond, with the same bijection, to k-trees whose labeling is constrained by their recursive construction. The corresponding generating functions satisfy differential rather than algebraic equations and we expect, as with binary search trees [CDJ01], an asymptotic normal law centered on $O(\log(n))$.

In section 6.2, we set the bijective algorithm between k-trees and class $\mathcal{K}$, and derive the enumerative generating function for k-trees. Section 6.3 explains the algorithm to calculate distances (in the graph) to the root of a k-tree, on the corresponding tree-structure in $\mathcal{K}$. The corresponding equations on bivariate generating functions are established by a careful analysis, marking vertices at distance d to the root and working on the cumulative generating functions to get the proportion of vertices at distance d. Finally, evaluating coefficients by means of complex analysis, we obtain the limiting distribution of distances to the root. In section 6.4 we extend this result to distances between a random pair of vertices in a random k-tree: we give an algorithm to "reroot" a k-tree, and show, via the equality of their generating functions, that "rerooted" k-trees are in bijection with k-trees with a pointed vertex. Using the same technique as in section 6.3, we finally obtain a Rayleigh limiting distribution.

6.2 Structures: k-trees and class $\mathcal{K}$

In this section we show a bijection between rooted k-trees and a simple variety of labeled trees, named $\mathcal{K}$, which is specifiable in terms of combinatorial constructions: $\mathcal{K} = \mathcal{Z}^k\mathcal{T}$, and $\mathcal{T} = \mathrm{Set}(\mathcal{Z} \times \mathcal{T}^k)$. This representation of k-trees, which highlights the cliques and their relationships, will be helpful to study the distance between vertices.

Inductive definition of k-trees. A k-tree on n vertices is a graph defined inductively as follows: the complete graph on k vertices (a k-clique) is a k-tree (with $n = k$), and if G is k-tree on $n-1$ vertices, then the graph resulting from adding a new vertex adjacent to the vertices of a k-clique of G is also a k-tree. This definition corresponds to graph theory trees when $k = 1$.

A *rooted k-tree* is a k-tree with a distinguished k-clique, together with a given permutation of the k vertices.

The $\mathcal{K}$-representation of a k-tree is a tree with black and white nodes: black nodes correspond to k+1-cliques and white nodes to k-cliques. Each black node is adjacent to the $k+1$ white nodes representing the k-cliques it contains. The size of a tree in $\mathcal{K}$ is the number of

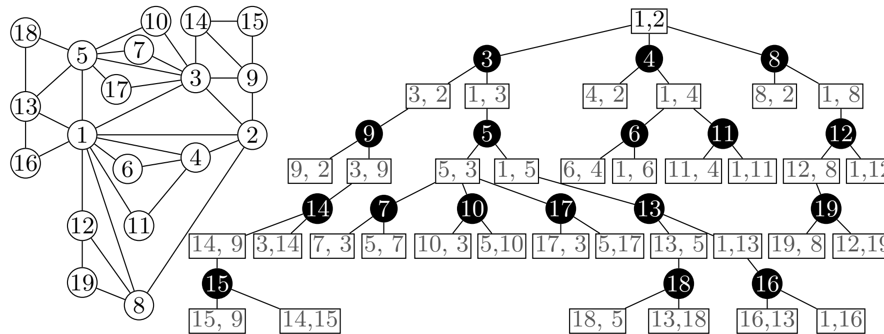


Figure 6.1: A 2-tree and the corresponding tree in $\mathcal{K}$

black nodes it contains plus k.

Remark 6.2.1. In this paper we shall consistently use the term *vertex* to denote “points” of the graph (k-tree), and *node* to denote “points” of the tree (in $\mathcal{K}$).

Inductive definition of $\mathcal{K}$. A tree $T \in \mathcal{K}$ of size n is either reduced to its root, a white node with k vertices (size k), or a tree $T' \in \mathcal{K}$ of size $n-1$ in which we add a new black node adjacent to a white node of T and its k white sons.

6.2.1 Bijection

The transformation of a rooted k-tree G into a tree $T \in \mathcal{K}$ is a two step process. First, create a tree called the completed clique-separator tree of G, whose nodes are cliques of G; moreover the root of the k-tree is the root of the corresponding tree. Second, simplify the labels of the tree by encoding most of the information in the tree structure.

The completed clique-separator tree of a k-tree G is a bipartite graph, whose black nodes are the k+1-cliques of G (which are also the maximal cliques of G), and whose white nodes are the k-cliques of G (which include the minimal separators of G). A black node is adjacent to the $k+1$ white nodes it contains. This structure is very similar to the clique-separator graph Ibarra [Iba04] defined for the larger class of chordal graphs.

Proposition 6.2.2. *The completed clique-separator tree of G is a tree.*

Proof. Following the inductive definition of k-trees it is easy to see that each new vertex adds one black node connected to one existing and k new white nodes. □

Proposition 6.2.3. *There is a bijection between the class of rooted k-trees and* $\mathcal{K}$.

Sketch of proof. For a rooted k-tree, the completed clique-separator tree is a rooted tree; its root is the root of the k-tree (a k-clique presented as an ordered list of vertices).

We see that the only information carried by a black node is the label of the vertex not included in its father. We can thus simplify the labeling of black nodes by retaining only this one vertex.

The one-to-one correspondence between k-trees and trees in $\mathcal{K}$ relies on an ordering of the sons of the black nodes. We proceed recursively from the root: consider a white node w with

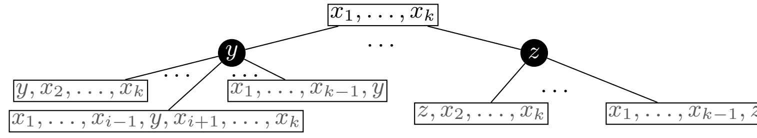


Figure 6.2: The ordering of sons of black nodes.

its ordered list of vertices $(x_1,\dots,x_k)$; this node is adjacent to a set of black nodes and there is a natural order between the k sons of any of these black nodes. Let b be a black node: the i-th son s_i is the k-clique that does not contain x_i. The list of vertices in s_i is the same as in w except that x_i is missing and is replaced by b: $(x_1,\dots,x_{i-1},b,x_{i+1},\dots,x_k)$.

Now we only need n labels in the representation of a k-tree on n vertices; the list of vertices in each white node can be completely determined by the position of the node in the tree, the vertices in the black nodes and the list of vertices at the root (i.e.: the first white node). We do however keep the labels on the white nodes for clarity, in figure 6.1 for instance, they still appear, in grey color. □

Algorithm 6.1 Tree representation

Input: a rooted k-tree G on n vertices
Output: a tree T, with a list of k vertices at the root

1: Create one white node for each k-clique of G
2: Create one black node for each k+1-clique of G
3: Put an edge between each black node and all k+1 k-cliques it contains
{At this point we have the completed clique-separator tree of G}
4: Remove all vertices from each b. node except for the one not included in its father
5: Propagate the order of the vertices in white nodes starting from the root
6: Order the sons of b. nodes: the i-th does not containing the i-th vertex of the father
7: Remove all vertices from each white node {The resulting tree is in $\mathcal{K}$}

Reversing this algorithm is easy. The labels removed in steps 4 and 7 can be determined by the position of the nodes in the tree, and the list of k+1-cliques created in step 2 suffices to reconstruct the k-tree; no edge is missing: each one of them is between two vertices belonging to the same k+1-clique.

6.2.2 Generating function for $\mathcal{K}$

Algorithm 6.1 transforms a rooted k-tree into a $\mathcal{K}$ structure, composed of a tree-structure $\mathcal{T}$ (that we call the *proper tree*), and a list of k vertices. A rooted k-tree on n vertices leads to a proper tree with $n-k$ black nodes. The proper tree is made of a white root, from which stems a set of black nodes; and each black node has a list of children which are k subtrees of the same type as the proper tree. Thus we get the following specification, with $\mathcal{E}$ denoting white nodes and $\mathcal{Z}$ denoting vertices of the original k-cliques:

$$\mathcal{K} = \mathcal{Z}^k\mathcal{T}, \qquad \mathcal{T} = \mathcal{E}\times \mathrm{Set}(\mathcal{Z}\times\mathcal{T}^k).$$

Using the symbolic method, see e.g. Flajolet and Sedgewick [FS09], we derive from this specification the functional equations on exponential generating functions, $K(z) = \sum_n K_n \frac{z^n}{n!}$ and $T(z) = \sum_n T_n \frac{z^n}{n!}$, where K_n (resp. T_n) is the number of trees of size n in $\mathcal{K}$ (resp. $\mathcal{T}$):

$$K(z) = z^k T(z), \qquad T(z) = \exp(zT^k(z)). \tag{6.1}$$

These generating functions are extensively used in the rest of this paper, especially in a bivariate form. We first use them to compute the number of k-trees on n vertices (this is another proof of a well known result [BP69; Moo69, ...]).

Theorem 6.2.4. *The number of k-trees on $n+k$ vertices is $\binom{n+k}{k}(kn+1)^{n-2}$.*

Proof. Let $C_k(z) = kzT^k(z)$. Then equation (6.1) becomes $C_k(z) = kze^{C_k(z)}$ and $T(z) = \exp(\frac{C_k(z)}{k})$. Using the Lagrange-Bürmann theorem we thus get $[z^n]T(z) = \frac{1}{n!}(kn+1)^{n-1} = \frac{K_{n+k}}{(n+k)!}$. Since $\mathcal{K}_n$ is in bijection with the class of rooted k-trees on n vertices, we need to divide K_{n+k} by the number of possible roots, $k!(kn+1)$, to obtain the number of k-trees. □

6.3 Distance to the root

This section deals with distances to the first vertex of the root of a k-tree. These distances can be easily marked on the corresponding tree structure, and by studying the resulting family of bivariate generating functions, we show that the proportion of vertices at distance d from the first vertex of the root is asymptotically Rayleigh distributed.

6.3.1 Using the tree to calculate the distance in the graph

We use an algorithm similar to that of Proskurowski [Pro80] to decorate each vertex with its (graph) distance to the root. Given a k-tree G and the corresponding tree T we start by assigning the distance 0 to the vertices of the root. Then, given a white node w, each of its black sons corresponds to a vertex at distance 1 plus the minimum of the distances of w's vertices.

Note that this process can be applied recursively starting from the root since each vertex in a white node w is either a part of the root or in a black node on the path from w to the root.

Algorithm 6.2 Distances to the first vertex of the root in $\mathcal{K}$

Input: a tree $K \in \mathcal{K}$ (with root r)

Output: an association table (vertex, distance)

1: Create an empty table A
2: **for all** vertices v in r **do**
3: **if** v is the first vertex of r **then** Add $(v, 0)$ to A **else** Add $(v, 1)$ to A
4: **return** $A \cup$ the result of Algorithm 6.3 on the proper tree of K and $(0, 1, \dots, 1)$

Remark 6.3.1. The same process can be used to calculate the distance to any subset $s = (x_{j_1}, \dots, x_{j_i})$ of i vertices in the root: instead of assigning distance 0 to all the vertices of the

root, we set to 0 the distance of the i vertices in s and set to 1 the distance of the other $k-i$ vertices of the root. Notice that the resulting decorated tree could have been obtained as a subtree of the distance tree in the original process.

Algorithm 6.3 Distances in $\mathcal{T}$

Input: a tree $T \in \mathcal{T}$ and k integers $(a_i)_{i\in\{1,\ldots,k\}}$
Output: an association table (vertex, distance)

1: Let d be $1+min(a_1,\ldots,a_k)$ and A an empty table
2: **for all** sons v of the root of T **do**
3: Add (v,d) to A
4: **for all** $i \in \{1,\ldots,k\}$ **do**
5: $A \leftarrow A \cup$ recursive call on the i-th son of v and $(a_1,\ldots,a_{i-1},d,a_{i+1},\ldots,a_k)$
6: **return** A

Remark 6.3.2. It is clear that if we shift, by a value d, all distances of a white node w's vertices, then all the distances in the subtree under w will be consistently shifted by d.

6.3.2 Bivariate generating functions

In this section, we are interested in estimating $K_{d,n,p}$, the number of trees of size n in $\mathcal{K}$ with p vertices at distance d from the first vertex of the root. Though it is possible to directly work on the combinatorial objects, we use a generating function framework that makes presentation easier. We thus define the bivariate generating functions $K_d(z,u) = \sum_{n,p} K_{d,n,p} u^p \frac{z^n}{n!}$, with $K_d(z,1) = K(z)$, for all d. Differentiating $K_d(z,u)$ with respect to u and setting $u=1$, provides function

$$K'_d(z) \equiv \frac{\partial}{\partial u} K_d(z,u)\Big|_{u=1},$$

where the coefficient of $\frac{z^n}{n!}$ represents the total number of vertices at distance d from the first vertex of the root in all trees of size n in $\mathcal{K}$. So that, in a random k-tree of size n the proportion of vertices at distance d from the vertex of the root is $\frac{1}{nK_n} K'_d(z)$. The aim of this section is to give a closed form expression for $K'_d(z)$ (see proposition 6.3.13).

We mostly argument and calculate on the proper subtree T rather than on the whole structure K and the generating functions are very similar.

Lemma 6.3.3. *Let $T_d(z,u)$ be the bivariate generating function with u marking vertices at distance d from the first vertex of the root,*

$$K_1(z,u) = z^k u^{k-1} T_1(z,u), \text{ and } \forall d \geq 2, K_d(z,u) = z^k T_d(z,u).$$

Proof. In the case $d=1$, each of the $k-1$ other vertices of the root are at distance one from the first vertex, since they are in the same clique. Whereas for $d \geq 2$ vertices in the root do not interfere. □

Case $d = 1$. We begin with vertices at distance one to the first vertex of the root. In a second paragraph we shall be interested in vertices at distance one to a subset of i vertices in the root, in order to prepare the general study of vertices at distance d.

Lemma 6.3.4. $T_1(z,u) = \exp(zuT(z)T_1^{k-1}(z,u))$.

Proof. All sons of the root are at distance one, and for each of these black nodes, all but the first of its white children contain the first vertex of the root, so that the black nodes immediately below them are also at distance one. Hence the result for generating functions. □

By differentiation we thus obtain the generating function for the total number of vertices at distance one, (also using the fact that $T_1(z,1) = T(z)$).

Lemma 6.3.5. $T_1'(z) \equiv \frac{\partial}{\partial u} T_1(z,u)\Big|_{u=1} = \frac{zT^{k+1}(z)}{1-(k-1)zT^k(z)}$.

Remark 6.3.6. If we want to count the distances to some other vertex in the root, the whole computation is the same: permuting the root's vertices brings us back to the initial problem.

The next problem is to count the number of vertices at distance one from a subset s of i vertices in the root. Extending the preceding notation ($T_1(z,u) = T_{1,1}(z,u)$), let $T_{1,i}(z,u)$ be the bivariate generating function for the number of vertices at distance one from any vertex in s.

Lemma 6.3.7. $T_{1,i}(z,u) = \exp(zuT_{1,i-1}^i(z,u)T_{1,i}^{k-i}(z,u))$.

Proof. The idea of the proof is the same as when $i = 1$: for each black node at distance one, $k-i$ of its white children contain all the i vertices of the subset and the remaining i contain all but one of them. By symmetry, the corresponding bivariate generating functions do not depend on the position of the i vertices in the root. □

Differentiating leads to the generating function for the number of vertices at distance one from i vertices of the root, (notice that for all i, $T_{1,i}(z,1) = T(z)$).

Lemma 6.3.8. $T_{1,i}'(z) \equiv \frac{\partial}{\partial u} T_{1,i}(z,u)\Big|_{u=1} = \frac{zT^k(z)}{1-(k-i)zT^k(z)} \Big(T(z) + iT_{1,i-1}'(z)\Big)$.

This recurrence can be solved and $T_{1,i}'(z)$ is a rational function in z and $T(z)$.

General case. By remark 6.3.2, in order to calculate the number of vertices at distance two from the first vertex of the root, it suffices to find all the white nodes containing only vertices at distance one (from the first vertex of the root) and apply the previous process to count the vertices at distance one from any of the vertices of these white nodes. In terms of generating functions, this means that $T_{2,1}(z,u) = \exp(zT_{1,k}(z,u)T_{2,1}^{k-1}(z,u))$ (notice that there is no occurrence of u outside $T_{1,k}(z,u)$).

Applying the same argument recursively, we can treat the case of any distance $d \geq 2$, and obtain the following lemma.

Lemma 6.3.9. *The bivariate generating function $T_{d,i}(z,u)$, with u marking the vertices at distance d from a subset of i vertices of the root, satisfies, for $d \geq 2$:*

$$T_{d,i}(z,u) = \exp(zT^{i}_{d,i-1}(z,u)T^{k-i}_{d,i}(z,u)), \quad \textit{for} \quad i \geq 2, \quad \textit{and}$$
$$T_{d,1}(z,u) = \exp(zT_{d-1,k}(z,u)T^{k-1}_{d,1}(z,u)).$$

By differentiating we obtain:

Lemma 6.3.10. $T'_{d,i}(z) \equiv \frac{\partial}{\partial u}T_{d,i}(z,u)\Big|_{u=1} = \frac{izT^k(z)}{1-(k-i)zT^k(z)}T'_{d,i-1}(z)$, *for $i \geq 1$ and $d \geq 2$, setting $T'_{d,0}(z) = T'_{d-1,k}(z)$.*

Lemma 6.3.11. $T'_{d,i}(z) = H(z)T'_{d-1,i}(z)$, *for $d \geq 2$, where $H(z) = k!(zT^k(z))^k \prod_{i=1}^{k-1} \frac{1}{1-izT^k(z)}$.*

This recurrence is easy to expand, the only difficulty is that it does not extend to when $d = 1$. We can however calculate $T'_{2,1}(z)$ which has the form of a rational function in z and $T(z)$.

Lemma 6.3.12. $T'_{d,1}(z) \equiv \frac{\partial}{\partial u}T_{d,1}(z,u)\Big|_{u=1} = H^{d-2}(z)T'_{2,1}(z)$.

Back to the whole structure $\mathcal{K}$, we have $K_d(z,u) = z^k T_{d,1}(z,u)$, so that

Proposition 6.3.13. *The exponential generating function counting the total number of vertices at distance d from the first vertex of the root in a rooted k-tree satisfies*

$$K'_d(z) \equiv \frac{\partial}{\partial u}K_d(z,u)\Big|_{u=1} = H^{d-2}(z)K'_2(z),$$

with $K'_2(z) = z^k T'_{2,1}(z)$ and $H(z) = k!(zT^k(z))^k \prod_{i=1}^{k-1} \frac{1}{1-izT^k(z)}$.

6.3.3 Limiting distribution

The number of vertices at distance d from the first vertex of the root in a k-tree of size n is obtained by estimating the coefficient of $\frac{z^n}{n!}$ in $K'_d(z)$ and normalizing by nK_n. We study the asymptotic of this quantity when n becomes large.

We first turn to the asymptotic estimation of coefficients of $T(z)$, $H(z)$ and $H^d(z)$, which relies on complex analysis.

Proposition 6.3.14. *Function $T(z)$ is analytic at the origin, with radius of convergence $\rho = \frac{1}{ke}$, singular value $\tau = e^{\frac{1}{k}}$, and a square-root singular expansion:*

$$T(z) = \tau - \frac{\tau}{k}\sqrt{2(1-z/\rho)} + O(1-z/\rho).$$

Proof. Class $\mathcal{T}$ is a simple variety of trees [MM78], and the result follows from the implicit function theorem. Moreover by singularity analysis $T_n \sim \frac{\tau}{k\sqrt{2\pi}}\rho^{-n}n^{-\frac{3}{2}}$. □

Lemma 6.3.15. *$H(z)$ is singular in ρ, with a square-root singular expansion*

$$H(z) = 1 - kH_k\sqrt{2(1-z/\rho)} + O(1-z/\rho), \text{ where } H_k = \sum_{i=1}^{k} \frac{1}{i}.$$

Proof. As seen before, $H(z) = k!(zT^k(z))^k \prod_{i=1}^{k-1} \frac{1}{1-izT^k(z)}$. For all $k \in \mathbb{N}^*$ and $i \in \{1,\ldots,k-1\}$, the term izT^k is singular in ρ and asymptotically equivalent to $i\rho\tau^k = \frac{i}{k} < 1$, so that no singularity comes from the cancellation of the denominators in the product. This product is shown to be equivalent, around the singularity ρ, to $G(z) = \frac{k^{k-1}}{(k-1)!}\prod_{i=1}^{k-1}\left(1 - \frac{i\sqrt{2(1-z/\rho)}}{k-i}\right)$. $H(z)$ is equivalent to $(zT^k(z))^k G(z)$, and the combination of square-root terms brings up a factor involving the k-th harmonic number H_k. □

Evaluating the coefficient of z^n in $H^d(z)$ depends on the values of d. The region where an interesting renormalization takes place is for $d = x\sqrt{n}$, as shown in the semi-large power theorem [FS09, Theorem IX.16].

Proposition 6.3.16. *For $d = x\sqrt{n}$, with x in any compact of $\mathbb{R}_+^*$,*

$$[z^n]H^d(z) \sim \frac{kH_k x}{n\sqrt{2\pi}} e^{-\frac{k^2 H_k^2 x^2}{4}}.$$

Proof. This result can be obtained by using the saddle-point method or singularity analysis [FS09]. □

Back to the estimation of distances, we have $K_d'(z) = H^{d-2}(z)K_2'(z)$, where $K_2'(z)$ is a rational function of z and $T(z)$ with no pole in $[0,\rho]$, so that it contributes for a constant in the coefficient of z^n. Finally, to get the proportion of vertices at distance d, we normalize by nK_n, and obtain the following theorem.

Theorem 6.3.17. *In a k-tree on n vertices, the probability that a random vertex r is at distance $d = x\sqrt{n}$ (with x in a compact of $\mathbb{R}_+^*$) from the first vertex v of the root, satisfies, as $n \to \infty$, a local law of the Rayleigh type:*

$$\lim_{n\to\infty} \sqrt{n}\mathbb{P}(D(v,r) = \lceil x\sqrt{n}\rceil) = c_k^2 x e^{-\frac{(c_k x)^2}{2}}, \text{ with } c_k = k\sum_{i=1}^{k}\frac{1}{i}.$$

In parallel to proving the limiting distribution, we made a series of measures on random k-trees generated with a purpose-built Boltzmann sampler [1], and the experimental curves perfectly fit the theoretical results, as shown in figure 6.3.

6.4 Distances to a random vertex

To estimate the distance to a random vertex of a k-tree we use the same idea as Proskurowski [Pro81]: "rerooting" the corresponding tree in order to move the vertex we are interested in to the root.

1. Available at http://www-apr.lip6.fr/~darrasse/ktrees

In our case, we find an expression $K^{\circ}(z)$ of the rerooted k-trees in terms of generating functions. This is proved by exhibiting a bijection between rerooted k-trees and pointed k-trees (that is k-trees with one pointed vertex, counted by $K^{\bullet}(z)$).

The limiting distribution is obtained by using the same analytic tools as before and we finally prove that the distances to a random vertex exactly obey the same distribution as the previous case.

6.4.1 Rerooting process

Given a distinguished vertex v in a rooted k-tree G with root r, we want to associate another rooted k-tree G' having the same underlying k-tree as G and where v is the first vertex of the new root r'. Two cases appear: either v belongs to the list of vertices of r, and exchanging v with r's first vertex suffices, or we need to find another k-clique of G to be the root of G'.

It is easier to work on the corresponding tree $T \in \mathcal{K}$. In T, we want to find a white node containing v as first vertex. There are many choices, but only one of them is always the closest to the root: the first son of v. This white node, named r', will be the root of G'. The corresponding tree T' can be either constructed using the method of section 6.2, or obtained directly from T by pulling on r'.

Algorithm 6.4 Rerooting

Input: a tree $T \in \mathcal{K}$ (with root r) and a vertex v
Output: a tree in $\mathcal{K}^{\circ}$, with v its first root vertex

1: **if** v is in r **then**
2: Put a mark on the first vertex of r
3: Exchange the first vertex in r with v
4: Apply steps 5 and 6 of Algorithm 6.1 to reorder the sons of black nodes
5: **return** the resulting tree
6: **else** {in this case v is a black node}
7: Let r' be the first son of v {r' is a white node and v its first vertex}
8: Get the k-tree G corresponding to T by the inverse of Algorithm 6.1
9: Let G' be the same graph as G, with a new root: r'
10: Apply Algorithm 6.1 to G' to obtain a new tree T'
11: **return** T' with a mark on the white node corresponding to r

Note that this process is reversible, since the mark allows to find the old root r.

Remark 6.4.1. The mark of the old root is always contained in the first subtree of one of the sons of the new root.

Theorem 6.4.2. *The class of rerooted k-trees is counted by the generating function* $K^{\circ}(z) = kz^kT(z) + z^{k+1}T^k(z)T^{\circ}(z)$, *where* $T^{\circ}(z)$ *counts the trees in* $\mathcal{T}$ *with a mark on a white node.*

Proof. The two terms of the sum correspond to the two cases of the rerooting process. For the first one, we have k possibilities. For the second, remark 6.4.1 implies that the black son of the

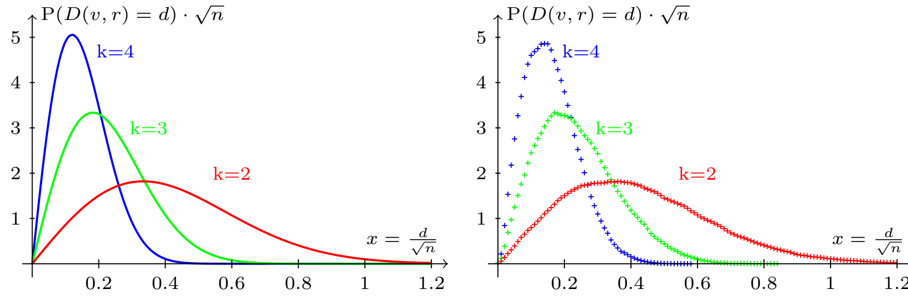


Figure 6.3: Theoretical (on the left) and experimental (on the right) distributions of distances in k-trees, for $k = 2,3$ and 4. The experimental curves come from measures on 10^3 random k-trees of size $10^4 \pm 10\%$.

root containing the old root is described by $zT^\circ(z)T^{k-1}(z)$ and the other black sons give the factor $\exp(zT^k(z))$:

$$K^\circ(z) = kz^kT(z) + z^k\exp(zT^k(z))\, zT^\circ(z)T^{k-1}(z) = kz^kT(z) + z^{k+1}T^k(z)T^\circ(z).$$

□

Theorem 6.4.3. *There is a bijection between rerooted k-trees and pointed k-trees.*

Proof. We show that the generating functions for both classes are the same. We first need to express $T^\circ(z)$: adding a variable x to count white nodes for trees in $\mathcal{T}$, deriving with respect to x and setting $x = 1$, we get

$$T^\circ(z) = T(z) + k\exp(zT^k(z))zT^\circ(z)T^{k-1}(z) = T(z) + zkT^k(z)T^\circ(z),$$

which can also be obtained with a combinatorial argument: marking a white node consists either in marking the root or in choosing one of the sets of black nodes below it and marking in one of the k subtrees below this black node.

We now need to show the equality of generating functions $T^\bullet(z) = zT^k(z)T^\circ(z)$, where $T^\bullet(z)$ counts trees in $\mathcal{T}$ with a mark on a black node.

For that, we use the fact that a tree in $\mathcal{T}$ with n black nodes contains $kn+1$ white nodes and that $[z^n]T^\bullet(z) = n[z^n]T(z)$. We thus have

$$\begin{aligned}[z^n]zT^k(z)T^\circ(z) &= \frac{1}{k}[z^n](T^\circ(z) - T(z)) = \frac{kn+1}{k}[z^n]T(z) - \frac{1}{k}[z^n]T(z)\\ &= n[z^n]T(z) = [z^n]T^\bullet(z).\end{aligned}$$

Hence $K^\bullet(z) = kz^kT(z) + z^kT^\bullet(z) = kz^kT(z) + z^{k+1}T^k(z)T^\circ(z) = K^\circ(z)$. □

6.4.2 Limiting distribution

We now come to study the distances to the first vertex of the root in a tree of type $\mathcal{K}^\circ$, which are exactly the distances to the marked vertex of a tree of type $\mathcal{K}^\bullet$. This allows to show that the distance between two random points in a random k-tree follows a Rayleigh distribution.

With similar notations as in section 6.3 and a similar analysis of the recursive structure of the trees, but more involved computations, we obtain:

Lemma 6.4.4. *The b.g.f. of vertices at distance d in K° can be expressed as*

$$K_1^\circ(z,u) = kz^k u^{k-1} T_{1,1}(z,u) + z^{k+1} u^k T_{1,1}^k(z,u) T^\circ(z),$$
$$K_d^\circ(z,u) = kz^k T_{d,1}(z,u) + z^{k+1} T_{d,1}^k(z,u) T_{d-1,k}^\circ(z,u).$$

Differentiating with respect to u gives

$$K_d^{\circ\prime}(z) = kz^k T^{k-1}(z) T_{d,1}'(z) + kz^{k+1} T^{k-1}(z) T_{d,1}'(z) T^\circ(z) + z^{k+1} T^k(z) T_{d-1,k}^{\circ\prime}(z).$$

In this function the dominant term is $z^{k+1}T^k(z)T_{d-1,k}^{\circ\prime}(z)$ and has the same singular behavior as $K_d'(z)$, up to a factor n, corresponding to the choice of a random vertex.

This result agrees with the general fact that in a very large random tree the root tends to have the same properties as any random vertex. We thus get exactly the same asymptotic distribution, as in section 6.3.

Theorem 6.4.5. *Given a random k-tree G over n vertices, the distance between two random vertices v, w of G has mean value of order $\sqrt{n}$ and is asymptotically Rayleigh distributed in the range $x\sqrt{n}$:*

$$\lim_{n\to\infty} \sqrt{n}\mathbb{P}(D(v,w) = \lceil x\sqrt{n} \rceil) = c_k^2 x e^{-\frac{(c_k x)^2}{2}}, \text{ with } c_k = k\sum_{i=1}^k \frac{1}{i}.$$

Cet article est destiné à être soumis à une revue.

Chapter 7

A Unifying Structural Approach to the Analysis of Parameters in k-trees

7.1 Introduction

The classes of k-trees are families of graphs that have been largely studied, both in combinatorics (for enumeration and characteristic properties [BP69; Ros74]) and in graph algorithms (since many NP-complete problems on graphs can be solved in polynomial time on k-trees [AP89]). They stand out by their underlying tree structure, related to their recursive definition, which makes easier both the analysis of the properties and the exploration of the structure. Indeed, for $k = 1$, k-trees are just trees, and for $k \geq 2$ a bijection [DS09] can be explicitly defined between k-trees and a non trivial simple family of trees.

A k-tree on n vertices is recursively defined as either a k-clique (if $n = k$), or a k-tree on $n-1$ vertices with an additional new vertex linked with every vertex of a clique in the original graph.

We are interested in the study of two parameters, the degree of a vertex and the distance between two vertices, in a k-tree chosen uniformly at random between all k-trees of a given size. These two parameters are widely studied in relation with complex networks. Indeed, the class of k-trees is related to random Apollonian networks [ZYW05], a class which has been investigated as a model in statistical physics (a random Apollonian network is a planar 3-tree with a non-uniform probability model).

The degree of a vertex is the number of its neighbors, and the distance between two vertices is the length of the shortest path between them. The profile is a more precise shape measure: in a rooted graph, the profile counts the number of vertices on the different levels at distance 1, 2, etc. from the root. In this paper, we study the asymptotic distribution of the vertex degree in a k-tree, and we estimate the mean distance and the asymptotic expected profile (proportion of vertices on each level).

More precisely, the results of this paper are the following. We first present in Section 7.2 the definition and combinatorial specification of k-trees along with the central bijection to a

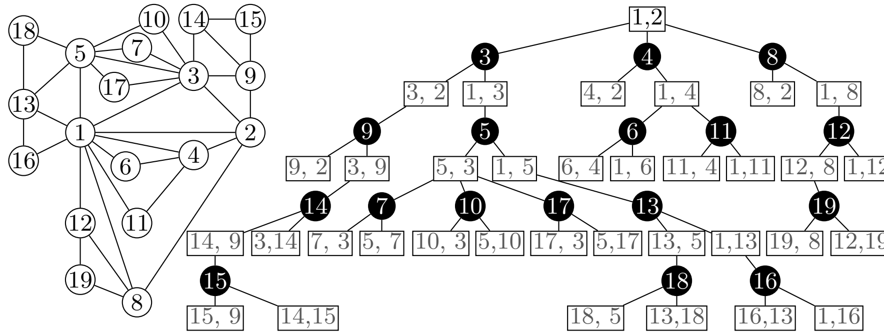


Figure 7.1: A 2-tree and the corresponding tree in $\mathcal{K}$.

simple family of trees. This bijection inspires an algorithm for counting distances in k-trees: this is the subject of section 7.3, where we give a combinatorial specification for k-trees with a marked vertex at a given distance from the root. This specification directly translates to bivariate generating functions that we use to analyze the parameters under consideration. The degree distribution is shown to be a power law with an exponential cutoff in Section 7.4. The profile follows a Rayleigh distribution as shown in Section 7.5, which gives as a byproduct that the mean distance between two random vertices is proportional to the square root of the size of the graph.

The advantage of this combinatorial approach is that it also applies to different subclasses of k-trees, in particular increasing k-trees [DHBS10] that correspond to the probability model of random Apollonian networks.

7.2 The bijection

Definition A k-tree on n vertices is either a k-clique (complete graph on k vertices) or a k-tree on $n-1$ vertices where a k-clique is chosen and a new vertex is added, adjacent to all the vertices of the k-clique.

We consider labelled k-trees, with vertices labelled with integers from 1 to n without any constraint. Figure 7.1 illustrates this definition. It is notable that for $k=1$ this definition comes to the classical definition of trees.

This recursive construction suggests an underlying tree structure. It is possible to carefully define this structure, in particular the way to chose its labels, and obtain a bijection. We consider *rooted k-trees*: the root will be a distinguished k-clique with an order on its vertices and the main result of this section will be the following.

Theorem 7.2.1. *The class of labelled rooted k-trees is in one-to-one correspondence with the class $\mathcal{K}$ of trees defined by:*

$$\mathcal{K} = \mathcal{Z}^k \star Set(\mathcal{T}), \quad \mathcal{T} = \mathcal{Z} \star Set(\mathcal{T}^k).$$

The proof of this theorem is made by first considering planar k-trees. We define *planar k-trees* as k-trees where each k-clique has a neighborhood of size at most 2. (We call *neighborhood* of a clique the set of vertices adjacent to all the vertices of the clique.) Remark that in the case $k = 2$ this corresponds to triangulations of a polygon and for $k = 3$ to RANS.

Instead of rooted planar k-trees we consider leaf-rooted planar k-trees: the root is a k-clique $v_1 \dots v_k$ with a neighborhood containing a single vertex v_{k+1}.

Lemma 7.2.2. *For each planar k-tree on $n+k$ vertices there correspond $(n-k)(k-1)+2$ leaf-rooted planar k-trees, when $n > 1$.*

Proof. The k-tree on $k+1$ vertices is a $(k+1)$-clique with k vertices, each with a single vertex in their neighborhood. Every new vertex added in an existing planar k-tree adds exactly k new cliques whose neighborhood is a single vertex, but also adds a vertex in the neighborhood of an existing k-clique. □

The algorithm for constructing a tree representation of a leaf-rooted plane k-tree is as follows:

- We start the construction of the tree by its root, which will be an edge labelled by the vertices $v_1 \dots v_k$.
- This edge ends to a node ν, labelled by the vertex v_{k+1}.
- The $(k+1)$-clique $v_1 \dots v_{k+1}$ contains k new k-cliques $(v_1 \dots v_{i-1} v_{k+1} v_{i+1} \dots v_k)_{i=1 \dots k}$ that will label a sequence of k edges $(e_i)_{i=1 \dots k}$ hanging from node ν.
- The clique labelling edge e_i has either zero or one new vertex neighbour. In the first case the edge leads to an empty tree. In the second case it is the root of a tree constructed recursively.

Lemma 7.2.3. *The class of rooted planar k-trees is in a one-to-one correspondence with class $\mathcal{K}_p$ of rooted planar k-ary trees defined by:*

$$\mathcal{K}_p = \mathcal{Z}^k \star \mathcal{T}_p, \qquad \mathcal{T}_p = \mathcal{E} \cup \mathcal{Z} \star \mathcal{T}_p^k.$$

Proof. This bijection comes directly from the tree construction. The only important detail is that every edge label can be deduced from the labels of the root and the vertices on its path from the root. Thus the number of necessary labels on the tree is equal to the number of vertices on the k-tree. □

This construction extends to general k-trees: in this case a k-clique can have an arbitrary number of neighbours so that each edge is a multi-edge that can lead to an unconstrained set of nodes. Moreover, any k-clique can be at the root of the k-tree, and thus the root of the tree also leads to a set of nodes.

We can now prove theorem 7.2.1 by taking $\mathcal{T} = \mathcal{Z} \star \mathrm{Set}(\mathcal{T}^k)$ replacing $\mathcal{T}_p$ and $\mathcal{K} = \mathcal{Z}^k \star \mathrm{Set}(\mathcal{T})$ replacing $\mathcal{K}_p$.

Remark 7.2.4. This construction is identical in the case of increasing structures, for example RAN. The corresponding class of trees is the class of increasing trees obtained by replacing all the products $\mathcal{Z} \star \mathcal{X}$ by the boxed product $\mathcal{Z}^{\square} \star \mathcal{X}$.

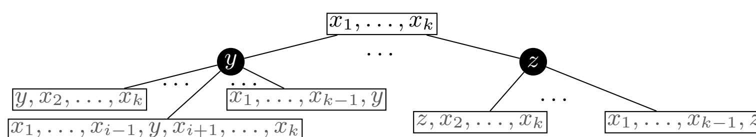


Figure 7.2: The ordering of sons of black nodes.

7.3 Algorithm for distances

The goal of this section is to describe a combinatorial class of trees, which is in bijection with *k*-trees marked on a vertex at a given distance from the first vertex of the root. We first present a modified version of an existing algorithm for counting distances in *k*-trees. Then, this algorithm is translated into combinatorial equations that define the desired class of trees.

For brevity, we will call f-distance the distance to the first vertex of the root and r-distance the distance to any of the k-vertices of the root.

7.3.1 Description of the algorithm

In this section we present an algorithm to calculate the f-distance of a given vertex in a given k-tree. This algorithm is a variant of Proskurowski's algorithm [Pro80] for calculating all f-distances in a k-tree and our own version of that algorithm [DS09].

The algorithm works on the tree representation of the k-tree by analyzing the path from the root to a given vertex v. If v is in the root, its f-distance is 1. Else, we use the fact that the vertex contained in a black node has f-distance one plus the minimum f-distance of the k vertices in the white node above it. Given that two successive generations of white nodes only differ by one vertex, we need to replace one by one all the vertices with f-distance d to attain a vertex with f-distance $d+2$. Considering that the root has its first vertex with f-distance 0 and all others with f-distance 1, if the path to v never goes to the first son of a black node then v has f-distance 1, else we study the path starting at the first "left" step. The white node there is composed exclusively by vertices with f-distance 1. Every time we take a step in a new direction down of a black node, one of these vertices is replaced by a vertex with f-distance 2. If all k possible directions are taken, we have reached f-distance 3 and start over, until we reach v.

Algorithm 7.1 Path from the root in $\mathcal{T}$

Input: a tree $T \in \mathcal{T}$ and a vertex v

Output: a stack containing the sequence of the ranks of white nodes in the path from the root of T to v

```
1: s ← ∅
2: w ← father(v)
3: while w ≠ root(T) do
4:    push(s, rank(w)) {rank(w) = i means that w is the i-th son of its father}
5:    w ← father(father(w))
6: return s
```

Algorithm 7.2 Distance to the first vertex of the root (f-distance) in $\mathcal{K}$

Input: a tree $K \in \mathcal{K}$ (with root r) and a vertex v

Output: the distance from v to the first vertex of r

```
1: p ← path(v)
2: d ← 1
3: if p ≠ ∅ then {v is not in the root}
4:    repeat
5:       t ← pop(p)
6:    until p ≠ ∅ and t ≠ 1
7:    if p ≠ ∅ then {v is has f-distance more than 1}
8:       repeat
9:          D⃗ ← 0⃗ {a k-dimensional bit vector to remember the directions that have already
            been seen in the path}
10:         while p ≠ ∅ and D⃗ ≠ 1⃗ do
11:            t ← pop(p)
12:            D⃗_t ← 1
13:         d ← d + 1
14:      until p ≠ ∅
15: return d
```

7.3.2 A combinatorial expression of the algorithm

We introduce a set of recursive definitions of combinatorial classes related to this algorithm. All these classes correspond to trees in $\mathcal{T}$ with distinguished black nodes. We first define the class $\mathcal{T}_{1,1}$ of trees with a mark on all vertices with f-distance 1. This is the base of our recursion and only depends on class $\mathcal{T}$. Then the class $\mathcal{T}_{1,i}$ of trees with a mark on all vertices at distance 1 from any of the i first vertices of the root is recursively defined using $\mathcal{T}_{1,i-1}$. Finally, the class $\mathcal{T}_{d,1}$ of trees with a mark on all vertices with f-distance d is also recursively defined.

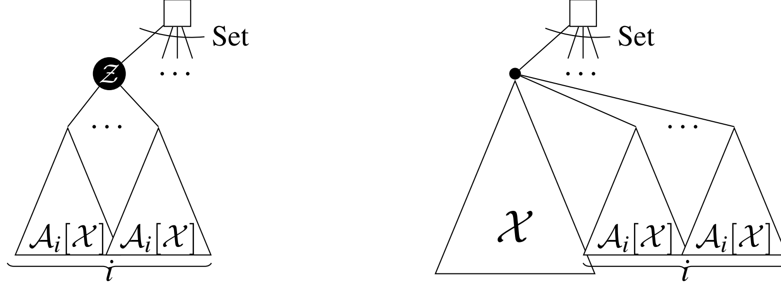


Figure 7.3: A schematic representation of $\mathcal{A}_i$ and $\mathcal{A}_i[\mathcal{X}]$.

Basic schema and operations

We introduce a family of classes of trees $\mathcal{A}_i$ which is the basic schema underlying the tree representation of k-trees: a $\mathcal{T}$ tree can always be obtained by uniformly substituting $\mathcal{T}$ trees at the nodes of $\mathcal{A}_i$ trees. It will allow: first, to add a layer of abstraction that will make easier the passage from the general class of k-trees to one of the interesting sub-classes. Second, the expression of the combinatorial operations that will be shown in what follows will be simplified by the use of this class.

Definition The class $\mathcal{A}_i$ is defined by

$$\forall 0 \leq i \leq k, \quad \mathcal{A}_i = \mathrm{Set}(\mathcal{Z}\mathcal{A}_i^i)$$

Note that the class $\mathcal{T}$ can be expressed in terms of $\mathcal{A}_i$: $\mathcal{T} = \mathcal{A}_k$.

Furthermore different sub-classes of k-trees can be obtained by choosing different definitions for $\mathcal{A}_i$.

For RANS the set construction reduces to sets of 0 or 1 elements $\mathcal{A}_i = \mathrm{Set}_{\leq 1}(\mathcal{Z}\mathcal{A}_i^i) = 1 + \mathcal{Z}\mathcal{A}_i^i$.

For increasing k-trees, to take into account the order constraints the Cartesian product is replaced by the boxed product [Gre91] $\mathcal{A}_i = \mathrm{Set}(\mathcal{Z}^{\square} \star \mathcal{A}_i^i)$. A simple bijection shows that this equivalent to $1 + \mathcal{Z}^{\square} \star \mathcal{A}_i^{i+1}$. And for RAN (increasing RANS) we combine the two restrictions and finally get $\mathcal{A}_i = \mathrm{Set}_{\leq 1}(\mathcal{Z}^{\square} \star \mathcal{A}_i^i) = 1 + \mathcal{Z}^{\square} \star \mathcal{A}_i^i$.

Substitution. Combinatorial operations will be considered via substitution.

Definition We use the notation $\mathcal{A}_i[\mathcal{X}]$ to represent $\mathcal{A}_i$ structures in which the atoms $\mathcal{Z}$ are substituted by elements of $\mathcal{X}$.

Since we will consistently want $\mathcal{A}_i[\mathcal{X}]$ to represent trees in $\mathcal{T}$ with some of its black nodes marked, we only allow substitutions with classes of a certain form. Thus $\mathcal{X}$ must either be equal to $\mathcal{Z}\mathcal{A}_j^{k-i}[\mathcal{Y}]$, with the possibility for $\mathcal{Z}$ to be marked (see below), with $j \leq k$ and $\mathcal{Y}$ satisfying this same constraint.

For increasing trees, the substitution must satisfy the increasing nature the structure. Thus $\mathcal{X}$ must be of the form $\mathcal{Z}^{\square} \star \mathcal{A}_j^{k-i}[\mathcal{Y}]$.

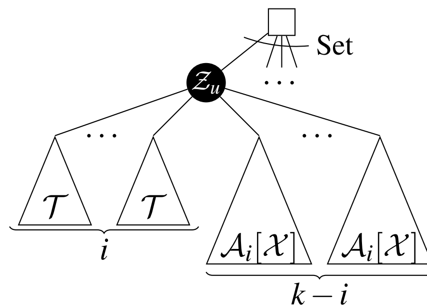


Figure 7.4: A schematic representation of $\mathcal{A}_{k-i}[\mathcal{Z}_u\mathcal{T}^i]$.

Marking. In what follows, we want to distinguish some of the atoms in a structure. For this, we will replace these atoms, by "marked" atoms $\mathcal{Z}_u$.

As an illustration of marking, consider the class of trees $\mathcal{A}_{k-i}[\mathcal{Z}_u\mathcal{T}^i]$, the corner stone of marked substitution. It can be seen as a tree of $\mathcal{T}$ with a mark on every vertex that is accessible from the root without ever going to one of the i first subtrees of any black node. Equivalently, this can be seen as a $(k-i)$-ary tree with all nodes marked and on which we adjoin, at each node, i (non-marked) trees of $\mathcal{T}$. It is important to note that by removing the marks we get back to class $\mathcal{T}$: $\mathcal{A}_{k-i}[\mathcal{Z}\mathcal{T}^i] = \mathcal{T}$.

Pointing. We will use the pointing operation, that is classical in combinatorics, which consists in distinguishing one particular atom in a structure. Thus $\mathcal{A}_i^\bullet$ will be the class of structures of $\mathcal{A}_i$ with one of its $\mathcal{Z}$ atoms replaced by atom $\mathcal{Z}^\bullet$.

Definition By extending the pointing operation to marked atoms $\mathcal{Z}_u$, we denote by $\mathcal{X}^{(u)}$ the class of $\mathcal{X}$ structures (containing both $\mathcal{Z}$ and $\mathcal{Z}_u$ atoms) where all $\mathcal{Z}_u$ atoms are replaced by $\mathcal{Z}$ atoms except one which is replaced by $\mathcal{Z}^\bullet$. Note that $\mathcal{X}^{(u)}$ does not contain any u marks.

Note that this is akin to differentiation in the sense of species theory [BLL98]. It is also the combinatorial equivalent to the differentiation with respect to variable u marking a parameter in bivariate generating functions [FS09].

Lemma 7.3.1. *If we note $\mathcal{A}_i^\bullet[\mathcal{X};\mathcal{Y}]$ the substitution of a structure in $\mathcal{A}_i^\bullet$ where all $\mathcal{Z}$ are substituted by elements of $\mathcal{X}$ and the one $\mathcal{Z}^\bullet$ by an element of $\mathcal{Y}$, the pointing with respect to $\mathcal{Z}_u$ atoms of a $\mathcal{A}_i[\mathcal{X}]$ structure will lead to*

$$(\mathcal{A}_i[\mathcal{X}])^{(u)} = \mathcal{A}_i^\bullet[\mathcal{X};\mathcal{X}^{(u)}].$$

Proof. Pointing a structure of $\mathcal{A}_i[\mathcal{X}]$ can be seen as a two steep process. First find the instance of $\mathcal{X}$ that will contain the mark. This gives a $\mathcal{A}_i^\bullet[\mathcal{X},\mathcal{X}]$ structure. Now a mark should be placed in the singled-out $\mathcal{X}$ structure, which means to replace it by a $\mathcal{X}^{(u)}$ structure to get $\mathcal{A}_i^\bullet[\mathcal{X},\mathcal{X}^{(u)}]$. □

Distance 1.

Let $\mathcal{T}_{1,1}$ be the class of trees in $\mathcal{T}$ where each vertex with f-distance 1 receives a mark u and the corresponding atom is $\mathcal{Z}_u$ instead of $\mathcal{Z}$.

Lemma 7.3.2. *$\mathcal{T}_{1,1}$ can be seen as the class $\mathcal{A}_{k-1}$ in which each black node is substituted by both a marked atom $\mathcal{Z}_u$ and a tree of $\mathcal{T}$.*

$$\mathcal{T}_{1,1} = \mathcal{A}_{k-1}[\mathcal{Z}_u\mathcal{T}].$$

Proof. Starting from the root and going down, all the vertices have f-distance 1 as long as we do not take the first subtree of any black node. As we have seen, these vertices are the vertices marked by u in $\mathcal{A}_{k-1}[\mathcal{Z}_u\mathcal{T}]$. □

We now turn to class $\mathcal{T}_{1,i}$, where each vertex at distance 1 to the i first vertices of the root receive a mark u. $\mathcal{T}_{1,i}$ is recursively defined with a marked substitution.

Lemma 7.3.3.

$$\mathcal{T}_{1,i} \equiv \mathcal{A}_{k-i}[\mathcal{Z}_u\mathcal{T}_{1,i-1}^i]$$

Proof. Consider a tree $T \in \mathcal{T}$ on which we want to mark with u all the neighbors of the i first vertices of the root. Either T is a leaf, in which case there are no vertices to mark, or there is a certain number of black nodes under the root, and each of them is treated the way we now describe.

The vertex in the black node is in the same $(k+1)$-clique as the root's vertices and will thus be marked.

The black node has k sons which gather in two groups:

1. the last $k-i$ sons all contain the i first vertices of the root, they are thus equivalent to the root as far as the marking is concerned. If we ignore the rest of the tree, what remains is a tree in $\mathcal{A}_{k-i}[\mathcal{Z}_u]$.
2. The i first sons of the black node contain all but one of the first i vertices of the root and we will mark (modulo a permutation of the root's vertices) the neighbors of the $i-1$ first vertices of the root of these subtrees. It then remains to add to each black node in the $\mathcal{A}_{k-i}[\mathcal{Z}_u]$ tree i trees of $\mathcal{T}_{1,i-1}$ to obtain a tree in $\mathcal{A}_{k-i}[\mathcal{Z}_u\mathcal{T}_{1,i-1}^i]$.

□

Our goal being the estimation of the mean number of vertices at a given distance, the classes we just defined contain more information than needed. Hence, we will derive from them the simpler family of classes of trees containing a single marked atom (of class $\mathcal{Z}^\bullet$) representing a vertex at the given distance. To get there, we use the extension of the pointing operation to marked atoms $\mathcal{Z}_u$.

Lemma 7.3.4.

$$\mathcal{T}_{1,1}^{(u)} \equiv \mathcal{A}_{k-1}^\bullet[\mathcal{Z}\mathcal{T};\mathcal{Z}^\bullet\mathcal{T}]$$

$$\mathcal{T}_{1,i}^{(u)} \equiv \mathcal{A}_{k-i}^\bullet[\mathcal{Z}\mathcal{T}^i;\mathcal{Z}^\bullet\mathcal{T}^i + i\mathcal{Z}\mathcal{T}^{i-1}\mathcal{T}_{1,i-1}^{(u)}]$$

Proof. The pointing is easy using the lemma 7.3.1 and the fact that $\mathcal{T}$ does not contain any $\mathcal{Z}_u$ atom. □

Distance $d \geq 2$.

Finding a vertex with r-distance d is equivalent to finding a white node w whose k vertices all have r-distance 1 and then, in the subtree under w, a vertex at distance $d-1$ from the k vertices of w. To reach w, starting from the root, we replace one by one the vertices of the root by new vertices, which have r-distance 1. Taking the j-th branch out of a black node replaces the j-th vertex.

Lemma 7.3.5. *For $d \geq 2$, with the convention that $\mathcal{T}_{d,0} = \mathcal{T}_{d-1,k}$, the class of trees with a mark on each vertex at distance d from the i first vertices of the root satisfies*

$$\mathcal{T}_{d,i} \equiv \mathcal{A}_{k-i}[\mathcal{Z}\mathcal{T}_{d,i-1}^{i}].$$

Proof. We are in the same situation as that of lemma 7.3.3, with the only difference that since we want to mark vertices at a distance greater than 1, the black nodes under the root contain vertices that should not be marked. □

Lemma 7.3.6. *For $d \geq 2$, the class of trees with a mark on one of its vertices at distance d from the i first vertices of the root satisfies*

$$\mathcal{T}_{d,i}^{(u)} \equiv \mathcal{A}_{k-i}^{\bullet}[\mathcal{Z}\mathcal{T}^{i}; i\mathcal{Z}\mathcal{T}^{i-1}\mathcal{T}_{d,i-1}^{(u)}]. \tag{7.1}$$

Remark 7.3.7. Note that the above recurrence relation is linear, in the sense that there is only one occurrence of $\mathcal{T}_{d,i-1}^{(u)}$ in the definition of $\mathcal{T}_{d,i}^{(u)}$, as there is only one mark in the tree.

Proof. This result is obtained directly by pointing class $\mathcal{T}_{d,i}$ on a $\mathcal{Z}_u$ atom. □

Combinatorial interpretation of the recurrence relation (7.1). The recurrence relation (7.1) allows to interpret $\mathcal{T}_{d,i}^{(u)}$, for all $d \geq 2$ and all $i \leq k$, as a class of trees $\mathcal{H}_i$, with a hole to be filled by a tree of $\mathcal{T}_{d,i-1}^{(u)}$ ($\mathcal{H}_i$ depends only on the value of i and not d). Thus, the k recursive steps that lead from the definition of $\mathcal{T}_{d,k}^{(u)}$ to that of $\mathcal{T}_{d-1,k}^{(u)}$ are always the same, for all $d \geq 2$, and correspond to the stacking of k trees coming successively from classes $\mathcal{H}_k$ through $\mathcal{H}_1$ (see figure 7.5.1). We will call the class of these stacks of trees $\mathcal{H}$. Finally, we can see a tree in $\mathcal{T}_{d,k}^{(u)}$ as a stack of $d-1$ trees in $\mathcal{H}$ on top of a tree in $\mathcal{T}_{1,k}$ (see figure 7.5.2).

Thus, the combinatorial interpretation of the algorithm leads to an arborescent vision of the recurrence relation and gives a combinatorial description of this recurrence's solution. In the general case of k-trees this combinatorial description directly translates into generating functions for counting k-trees with a marked vertex with a given r-distance. For increasing k-trees, it only translates to a differential equation whose solution is the generating function.

7.4 Degree distribution

This section is devoted to studying the degree of a random vertex in a random k-tree: we show a limiting distribution which is a power law with an exponential cut-off. The goal of this section is to prove the following theorem:

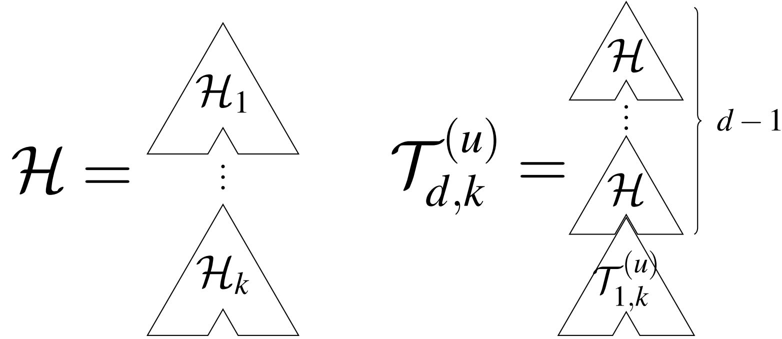


Figure 7.5: A schematic representation of class $\mathcal{H}$.

Theorem 7.4.1. *The degree of a random vertex v in a random k-tree of size n+k, has a limiting distribution which is a power law with an exponential cut-off: for large enough values of p,*

$$\lim_{n\to\infty} \Pr(d^o(v) = k+p) = C_k \alpha_k^p p^{-3/2}, \quad \textit{with } \alpha_k, C_k \in \mathbb{R}^+ \textit{ and } \alpha_k < 1.$$

This result is an extension of the analog proposition concerning the degree distribution of vertices in RANS, that appears in our previous work [DS07], and was also proved with probabilistic methods by Albenque and Marckert [MA08].

According to the tree representation of k-trees defined in section 7.3, the degree of a vertex v (that is the number of vertices at distance 1 from v) in the k-trees, translates into a subtree-size parameter in the tree representation (the size of the partial subtree of v obtained by recursively keeping all but one of the black nodes subtrees under v): this is expressed by the substitution $\mathcal{T}_{1,1} = \mathcal{A}_{k-1}[\mathcal{Z}_u\mathcal{T}]$ of Lemma 7.3.2.

The analytic part of the work consists in studying the associated bivariate generating function, that satisfies: $T(z,u) = \exp(zuT(z)T^{k-1}(z,u))$. The key point is the subcritical nature of the substitution, that reduces the degree enumeration problem to a size enumeration problem for another family of trees; hence the asymptotic distribution with exponential tails.

7.4.1 Statistics on subtrees

In this section we repeatedly use results of combinatorial analysis, that can be easily found in [FS09], to compute the generating functions related to the parameters under consideration.

Families of trees

The specification of the family $\mathcal{K}$ of rooted k-trees expresses as

$$\mathcal{K} = \mathcal{Z}^k \star \mathcal{T}, \quad \mathcal{T} = \mathrm{Set}(\mathcal{Y}) \quad \text{and} \quad \mathcal{Y} = \mathcal{Z} \star \mathcal{T}^k,$$

where $\mathcal{T}$ represents the family of trees rooted at a white node (k-clique), and $\mathcal{Y}$ represents the family of trees rooted at a black node (vertex). Denoting by $K(z) = \sum K_n \frac{z^n}{n!}$, $T(z) = \sum T_n \frac{z^n}{n!}$ and

$Y(z) = \sum Y_n \frac{z^n}{n!}$ the corresponding exponential generating functions, where K_n, T_n and Y_n are the number of trees of size n in each family, the specification translates into the following equations

$$K(z) = z^k T(z), \quad T(z) = \exp(Y(z)) \quad \text{and} \quad Y(z) = zT(z)^k.$$

Using the Implicit Function Theorem to solve $Y(z)$, it is easy to show that the three generating functions all have a dominant singularity at $\rho = 1/ke$, and a singular expansion of the square-root type: $f(z) = \tau - h\sqrt{1 - z/\rho} + \ldots$, with different singular values, for instance $\tau_Y = 1/k$ and $\tau_T = \exp(1/k)$.

The use of Lagrange and Lagrange-Burmann inversion theorem also gives the exact value of the enumeration sequences:

$$Y_n = (kn)^{n-1} \quad \text{and} \quad T_n = (kn+1)^{n-1},$$

and finally

$$\frac{K_{n+k}}{(n+k)!} = \frac{T_n}{n!}.$$

Bivariate generating functions

Bivariate generating functions (BGF) enumerate objects both according to their size and another parameter of interest. We shall be interested in the tree parameter corresponding to the degree of a vertex in the graph. With a stretch of language, we shall speak about the degree of a tree:

Definition A *black rooted subtree is of degree* p if its root-vertex has degree p in the corresponding k-graph, and a *white rooted subtree is of degree* p if the first vertex of the root-clique has degree p in the k-graph.

From the substitution $\mathcal{T}_{1,1} = \mathcal{A}_{k-1}[\mathcal{Z}_u \mathcal{T}]$ of Lemma 7.3.2, we get the equation satisfied by the bivariate generating function $T(z,u) = \sum T_{n,p} u^p \frac{z^n}{n!}$, where $T_{n,p}$ is the number of $\mathcal{T}$-trees with size n and degree p (for the first vertex of the root-clique):

$$T(z,u) = \exp(zuT(z)T^{k-1}(z,u))$$

For black-rooted trees, the degree of the root-vertex comes from the degrees of its white subtrees at the root:

$$Y(z,u) = zT^k(z,u),$$

with $Y(z,u) = \sum Y_{n,p} u^p \frac{z^n}{n!}$, and $Y_{n,p}$ is the number of $\mathcal{Y}$-trees with size n and degree p (for the root vertex).

In the following, we shall be interested in counting the number of black vertices with degree p in $\mathcal{T}$-trees. This parameter recursively decomposes into subtrees, and we first study how to express this kind of additive parameters.

Additive parameters

Let $\mathcal{A}$ be a simple family of trees, with generating function $A(z) = z\phi(A(z))$. A parameter $\omega : \mathcal{A} \to \mathbb{N}$ is said to be *additive* if it can be expressed as the sum of the values of ω on all the root subtrees, plus the value of a simpler root-parameter η: for any tree $A =< z, A_1, \ldots, A_r >$,

$$\omega(A) = \eta(A) + \sum_{i=1\ldots r} \omega(A_i), \tag{7.2}$$

and unwinding the recursion leads to $\omega(A) = \sum \eta(s)$, where the sum is taken on *all subtrees* of A.

For example, in a black-rooted $\mathcal{Y}$-tree, if ω counts the total number of black vertices with degree p, the corresponding root-parameter η has value 1 if the root has degree p, and 0 otherwise.

Let $A(z,u) = \sum_{A\in\mathcal{A}} u^{\eta(A)} z^{|A|}$ be the bivariate generating function for the root-parameter η: $A(z,u) = \sum h_{n,p} u^p z^n$, where $h_{n,p}$ is the number of $\mathcal{A}$-trees of size n and value p for parameter η. Since we are interesting in evaluating the proportion of nodes with value p for parameter η, in all $\mathcal{A}$-trees of size n, we consider the BGF $\Omega^{(p)}(z,u) = \sum \omega^{(p)}_{n,m} u^m z^n$, where $\omega^{(p)}_{n,m}$ is the number of $\mathcal{A}$-trees of size n with m subtrees having value p for parameter η. Relation (7.2) rewrites as

$$\Omega^{(p)}(z,u) = (u-1)[u^p]A(z,u) + z\phi(\Omega^{(p)}(z,u)). \tag{7.3}$$

Thus $m.\omega^{(p)}_{n,m}$, which is the total number of subtrees with parameter η equal to p, in all $\mathcal{A}$-trees of size n is is obtained by differentiating relation (7.3) with respect to u, and set $u = 1$. Noticing that $\frac{1}{1-z\phi'(A(z))} = z\frac{A'(z)}{A(z)}$, we have shown the following proposition:

Proposition 7.4.2. *Let $\mathcal{A}$ be a simple family of trees, with generating function satisfying $A(z) = z\phi(A(z))$.*

Let $\Omega : \mathcal{A} \to \mathbb{N}$ be an additive parameter, and η its associated root-parameter. Then the probability that a random subtree has parameter η equal to p, in a random $\mathcal{A}$-trees of size n is

$$\frac{m.\omega^{(p)}_{n,m}}{nA_n} = \frac{1}{nA_n}[z^n]\left(z\frac{A'(z)}{A(z)}[u^p]A(z,u)\right). \tag{7.4}$$

N.B. This result concerns ordinary generating function, in the case of exponential generating function, an extra factor $n!$ is needed.

Corollary 7.4.3. *Let $\mathcal{A}$ be a simple family of trees satisfying the hypothesis of proposition 7.4.2. If $\mathcal{B} = \mathcal{A}^r$ then the probability that a random subtree has parameter η equal to p, in a random $\mathcal{B}$-trees of size n is*

$$\frac{1}{nB_n}[z^n]\left(zrA(z)^{r-2}A'(z)[u^p]A(z,u)]\right).$$

Proof. For $\mathcal{B} = \mathcal{A}^r$, we have $B(z) = A^r(z)$. The bivariate generating function counting the number of subtrees with a given value p for parameter η is $\left(\Omega^{(p)}(z,u)\right)^r$, and the result follows by differentiation.

Furthermore, by linearity, we also get that for any function ψ such that $B(z) = \psi(A(z))$, the proportion of subtrees with value p for η, in all $\mathcal{B}$-trees of size n is

$$\frac{1}{nB_n}[z^n]\left(z\psi'(A(z))\frac{A'(z)}{A(z)}[u^p]A(z,u)\right). \tag{7.5}$$

□

Singular Boltzmann generation.

Proposition (7.4.2) can also be interpreted in terms of random generation with a singular Boltzmann sampler. Let ρ be the dominant singularity of the generating function $A(z)$, which has a singular expansion of the square root type since $\mathcal{A}$ is a simple family of trees: when $z \to_{-} \rho$

$$A(z) = \tau - h\sqrt{1 - z/\rho} + \dots$$

Provided that $A_p(z) \equiv [u^p]A(z,u)]$ tends to a finite value $A_p(\rho)$ when $z \to_{-} \rho$, we can estimate the asymptotic probability that a random subtree, in a large random tree, has a given value p for parameter η, which is given by (7.4):

$$\lim_{n\to\infty} \frac{1}{nA_n}[z^n]\left(z\frac{A'(z)}{A(z)}[u^p]A(z,u)\right) = \frac{A_p(\rho)}{A(\rho)}$$

This result can easily be obtained by Singularity Analysis: the singular expansion of $A(z)$ implies that $[z^n]z\frac{A'(z)}{A(z)} \sim \frac{nA_n}{A(\rho)}$.

It is noteworthy that this asymptotic probability is equal to the probability of obtaining a tree with a singular Boltzmann generation. In a Boltzmann sampler with tuning value x, the probability of generating a given tree A is equal to $\frac{x^{|A|}}{A(x)}$, for a tuning value $x < \rho$ (and for generating trees it is even possible to take $x = \rho$). Thus proposition (7.4) can be rephrased as follows (in [DS07], we used this proposition to work on the degree of RANS).

Proposition 7.4.4. *Given a simple family of trees $\mathcal{A}$, and A a random tree of size n, the set of subtrees of A tends (when $n \to \infty$) to be identical to the set of trees of $\mathcal{A}$ generated with a singular Boltzmann sampler. So that, under very large analytic conditions, the statistics on these two sets tends to be the same.*

7.4.2 Degree distribution

Expression

We are looking for the proportion of vertices with degree $p+k$ in all rooted k-trees of size $n+k$. This quantity expresses as $\frac{R_{n,p}}{(n+k)K_{n+k}}$, where $R_{n,p}$ is the total number of vertices of degree $p+k$ in all rooted k-trees of size $n+k$. Since there are $\frac{(n+k)!}{n!}$ rooted k-trees of size $n+k$ corresponding to each $\mathcal{T}$-tree of size n, we get $R_{n,p} = \frac{(n+k)!}{n!}\Theta_{n,p}$, with $\Theta_{n,p}$ being the total

number of vertices of degree $p+k$ in all rooted $\mathcal{T}$-trees of size n. So that finally, the proportion of vertices with degree $p+k$ is

$$\Pr(d^\circ = p+k) = \frac{1}{(n+k)T_n}\Theta_{n,p}.$$

There are two kinds of vertices with degree $p+k$ in a $\mathcal{T}$-tree:
- a vertex of the root-clique with $p+1$ black vertices at distance 1 from the first vertex of the root has degree $p+k$ since we also have to count the $k-1$ other links to the vertices of the root-clique.
- a black node whose subtree contains p vertices at distance 1 from the root has degree $p+k$ since we also have to count the k links that were added when adding this node.

Taking into account that the k vertices of the root-clique play the same role, we have now proved the following proposition.

Proposition 7.4.5. *In a rooted ktrees of size $n+k$, the proportion of vertices with degree $p+k$ expresses as*

$$\Pr(d^\circ = p+k) = \frac{1}{(n+k)T_n}(kT_{n,(p+1)} + S_{n,p})$$

where $T_{n,p}$ is the number of white-rooted trees of degree p, and $S_{n,p}$ is the total number of black-rooted subtrees of degree p in all $\mathcal{T}$-trees of size n.

We now have to evaluate $T_{n,p}$ and $S_{n,p}$: by definition

$$T_{n,p} = [\frac{z^n}{n!}u^p]T(z,u),$$

and by the Corollary of Proposition (7.4.2) the second quantity is

$$S_{n,p} = [\frac{z^n}{n!}u^p]z\frac{Y'(z)}{Y(z)}e^{Y(z)}Y(z,u). \tag{7.6}$$

Subcriticity

The combinatorial substitution $\mathcal{T}_{1,1} = \mathcal{A}_{k-1}[\mathcal{Z}_u\mathcal{T}]$, translates into a substitution of generating functions

$$T(z,u) = \exp(zuT(z)T^{k-1}(z,u)) = A_{k-1}[uzT(z)],$$

where $A_{k-1}(t)$ satisfies the equation $A_{k-1}(t) = \exp(tA_{k-1}^{k-1}(t))$ and has radius of convergence $1/(k-1)$.

It is easy to see that the singular value of $zT(z)$, $\rho\tau = \frac{e^{1/k}}{ke}$, is smaller than the radius of convergence of A_{k-1}; thus the substitution $A_{k-1}[uzT(z)]$ is *subcritical*, and we can use the Taylor expansion of $A_{k-1}(t)$ to express $T_{n,p}$. The same property holds for expressing $S_{n,p}$ in terms of the Taylor coefficients of $A_{k-1}^k(t)$.

Lemma 7.4.6. *The coefficients $T_{n,p}$ and $S_{n,p}$ satisfy*

$$\begin{aligned} T_{n,p} &= \frac{a_p}{p!} n! [z^n] z^p T^p(z), \quad \textit{with} \quad a_p = \big(p(k-1)+1\big)^{p-1} \\ S_{n,p} &= \frac{a_p^{(k)}}{p!} n! \left[z^{n-p}\right] \left(zT^{p+1}(z) + kz^2 T^p(z) T'(z)\right), \textit{ with } a_p^{(k)} = k\big((k-1)p+k\big)^{p-1} \end{aligned}$$

Proof. By Lagrange Inversion Theorem,

$$A_{k-1}(t) = \sum a_p \frac{t^p}{p!}, \qquad \text{with} \qquad a_p = \big((k-1)p+1\big)^{p-1}.$$

Since $T(z,u) = A_{k-1}[uzT(z)]$, and the substitution is subcritical,

$$T(z,u) = \sum T_{n,p} \frac{z^n}{n!} u^p = \sum \frac{a_p}{p!} T^p(z) u^p z^p.$$

For $S_{n,p}$, we need to express $A_{k-1}^k(t)$, and by Lagrange Burmann Inversion:

$$A_{k-1}^k(t) = \sum a_p^{(k)} \frac{t^p}{p!}, \qquad \text{with} \qquad a_p^{(k)} = k\big((k-1)p+k\big)^{p-1}.$$

By equation (7.6), and expressing Y in terms of T, we have:

$$z\frac{Y'(z)}{Y(z)} e^{Y(z)} Y(z,u) = zA_{k-1}^k[uzT(z)]\Big(T(z) + kzT'(z)\Big). \tag{7.7}$$

The substitution being subcritical, we can use the Taylor expansion of $A_{k-1}^k(t)$, and this gives the expression of $S_{n,p}$. □

Estimations

We now use Singularity Analysis to give an estimation, when $n \to \infty$, of the probability

$$\Pr(d^o = p+k) = \frac{1}{(n+k)T_n}(kT_{n,(p+1)} + S_{n,p}).$$

Since $T(z)$ has a singular expansion of the square-root type,

$$T(z) = \tau - h\sqrt{1 - z/\rho} + \dots,$$

we have $T_n \sim n! \frac{h}{2\sqrt{pi}} \rho^{-n} n^{-3/2}$, and for $p > 1$: $[z^n]T^p(z) \sim p\tau^{p-1} T_n$, and for the derivative: $[z^n]T'(z) \sim nT_n/\rho$.

Thus the leading term in the estimation of the degree probability, comes from the coefficient of the term containing $T'(z)$ in $S_{n,p}$:

$$\begin{aligned} \Pr(d^o = p+k) &= \frac{1}{nT_n} \frac{a_p^{(k)}}{p!} \left[z^{n-p}\right] \left(kz^2 T^p(z) T'(z)\right) \left(1 + O\left(\frac{1}{n}\right)\right) \\ &= \frac{a_p^{(k)}}{p!} \frac{1}{e} (\rho\tau)^p \left(1 + O\left(\frac{1}{n}\right)\right). \end{aligned}$$

Using Stirling formula for evaluating $p! \sim p^{p+1/2}\sqrt{2\pi}/e^p$, we get

$$\frac{a_p^{(k)}}{p!} \sim \frac{k}{\sqrt{2\pi}} p^{-3/2} e^p (k-1)^{p-1} \left(1 + \frac{k}{p(k-1)}\right)^{p-1}$$

Since the last factor tends to 1 for large enough p, we finally proved that the degree distribution eventually has a limiting distribution which is a power law in $-3/2$, with an exponential cut-off.

Theorem 7.4.7. *The degree of a random vertex v in a random k-tree of size $n+k$, has a limiting distribution which behaves as a power law with an exponential cut-off: for large enough values of p,*

$$\lim_{n\to\infty} \Pr(d^\circ(v) = k+p) = C_k \alpha_k^p p^{-3/2},$$

with $\alpha_k = e^{1/k}(k-1)/k < 1$ and $C_k = k/\big((k-1)e\sqrt{\pi}\big)$.

7.5 Calculation of the profile

This section deals with the estimation of the mean number of vertices at a given distance from the first vertex of the root in a random k-tree. We first show how one can see $\mathcal{H}$ as a multiplicative operator. Then we translate the combinatorial classes to generating functions and complete the calculation by means of singularity analysis.

Pointing of non-increasing trees. A pointed tree can always be seen as the product of the subtree rooted at the marked node and a sequence of trees with "holes". The first such tree is the subtree of the original subtree rooted at the father of the marked node, with a hole replacing the marked node's subtree. Remark that this transformation is not possible in the case of increasing trees since the constraints on labels don't allow to extract a subtree from its context.

Lemma 7.5.1.

$$\mathcal{A}_{k-i}^\bullet = \mathcal{Z}^\bullet \mathcal{A}_{k-i}^{k-i+1} Seq((k-i)\mathcal{Z}\mathcal{A}_{k-i}^{k-i})$$

Proof. By differentiating the definition of $\mathcal{A}_i = \mathrm{Set}(\mathcal{Z}\mathcal{A}_i^i)$ we obtain

$$\mathcal{A}_i^\bullet = \mathcal{A}_i(\mathcal{Z}^\bullet \mathcal{A}_i^i + i\mathcal{Z}\mathcal{A}_i^{i-1}\mathcal{A}_i^\bullet) = \mathcal{Z}^\bullet \mathcal{A}_i^{i+1} + i\mathcal{Z}\mathcal{A}_i^i\mathcal{A}_i^\bullet.$$

Using the recursive definition of $\mathcal{X} = \mathcal{Y}_1\mathrm{Seq}(\mathcal{Y}_2)$: $\mathcal{X} = \mathcal{Y}_1 + \mathcal{Y}_2\mathcal{X}$ we finally get $\mathcal{A}_i^\bullet = \mathcal{Z}^\bullet \mathcal{A}_i^{i+1}\mathrm{Seq}(i\mathcal{Z}\mathcal{A}_i^i)$. □

We will use this result to express the trees with a marked vertex at a given distance from the root as a product of trees. Since each of the k steps for going from the class that counts vertices with f-distance d to the one counting vertices with f-distance $d+1$ consists in attaching a tree of the previous class on the marked node of a pointed tree, this operation can be seen as a product too. And given that the passage from d to $d+1$ is the same for all values of d greater than 1, knowing the classes $\mathcal{T}_{1,1}$ and $\mathcal{T}_{2,1}$ counting trees at distance 1 and 2 respectively as well as class $\mathcal{H}$, the multiplicative operator of the passage from distance d to $d+1$ gives the classes counting the nodes having any given f-distance.

Lemma 7.5.2. *The class $\mathcal{H}_i$ that governs the recurrence relation between $\mathcal{T}_{d,i}^{(u)}$ and $\mathcal{T}_{d,i-1}^{(u)}$ can be expressed as a simple substitution:*

$$\mathcal{H}_i \equiv \mathcal{A}_{k-i}^{\bullet}[\mathcal{Z}\mathcal{T}^i; i\mathcal{Z}\mathcal{T}^{i-1}].$$

Proof. Using equation 7.1 and remark 7.3.7, we know that

$$\begin{aligned}\mathcal{T}_{d,i}^{(u)} &\equiv \mathcal{A}_{k-i}^{\bullet}[\mathcal{Z}\mathcal{T}^i; i\mathcal{Z}\mathcal{T}^{i-1}\mathcal{T}_{d,i-1}^{(u)}] \\ &\equiv \mathcal{T}_{d,i-1}^{(u)} \star \mathcal{A}_{k-i}^{\bullet}[\mathcal{Z}\mathcal{T}^i; i\mathcal{Z}\mathcal{T}^{i-1}].\end{aligned}$$

We thus don't need to mark with a hole the position of the recursive call. □

Lemma 7.5.3. *The class $\mathcal{H}_i$ can be seen as a product of trees:*

$$\mathcal{H}_i \equiv i\mathcal{Z}\mathcal{T}^k Seq((k-i)\mathcal{Z}\mathcal{T}^k)$$

Proof. By substituting $\mathcal{A}_{k-i}^{\bullet}$ by its value given in lemma 7.5.1 in the preceding lemma we get

$$\begin{aligned}\mathcal{H}_i &\equiv i\mathcal{Z}\mathcal{T}^{i-1}\mathcal{T}^{k-i+1}\mathrm{Seq}((k-i)\mathcal{Z}\mathcal{T}^i\mathcal{T}^{k-i}) \\ &\equiv i\mathcal{Z}\mathcal{T}^k\mathrm{Seq}((k-i)\mathcal{Z}\mathcal{T}^k).\end{aligned}$$

□

Finally the class $\mathcal{H}$ that expresses the recursive relation between $\mathcal{T}_{d,k}^{(u)}$ and $\mathcal{T}_{d-1,k}^{(u)}$ can be explicitely defined.

Proposition 7.5.4. *$\mathcal{H}$ is a class whose elements are products of black vertex rooted trees ($\mathcal{Z}\mathcal{T}^k$) with some "colors" and a hole, to be filled by a tree:*

$$\mathcal{H} = \mathcal{H}_k\mathcal{H}_{k-1}\cdots\mathcal{H}_1 \equiv k!(\mathcal{Z}\mathcal{T}^k)^k \prod_{i=1}^{k-1} Seq(i\mathcal{Z}\mathcal{T}^k)$$

Proof. The linearity mentioned in remark 7.3.7 allows us to see the decomposition of $\mathcal{H}$ into $\mathcal{H}_i$ as a product and using lemma 7.5.3 leads to the result. □

The combinatorial expression of $\mathcal{H}$ can be directly translated into a generating function: $H(z) = k!(zT(z))^k \prod_{i=1}^{k-1} \frac{1}{1-izT^k(z)}$. The fact that the denominator of this expression is never null on the domain of definition of $T(z)$ means that $H(z)$ behaves as a finite product of trees.

Thus we can evaluate the coefficients in $H^d(z)$ using the semi-large power theorem [FS09, Theorem IX.16].

Proposition 7.5.5. *For $d = x\sqrt{n}$, with x in any compact of $\mathbb{R}_+^*$,*

$$[z^n]H^d(z) \sim \frac{kh_kx}{n\sqrt{2\pi}}e^{-\frac{k^2h_k^2x^2}{4}} \text{ where } h_k = \sum_{i=1}^k \frac{1}{i}.$$

Proof. For all $k \in \mathbb{N}^*$ and $i \in \{1,\ldots,k-1\}$, the term izT^k is singular in ρ and asymptotically equivalent to $i\rho\tau^k = \frac{i}{k} < 1$, so that no singularity comes from the cancellation of the denominators in the product. The product $\prod_{i=1}^{k-1} \frac{1}{1-izT^k(z)}$ is equivalent, around the singularity ρ, to $G(z) = \frac{k^{k-1}}{(k-1)!} \prod_{i=1}^{k-1} \left(1 - \frac{i\sqrt{2(1-z/\rho)}}{k-i}\right)$. $H(z)$ is equivalent to $(zT^k(z))^k G(z)$, and the combination of square-root terms brings up a factor involving the k-th harmonic number H_k.

We thus get that $H(z)$ is singular in ρ, with a square-root singular expansion

$$H(z) = 1 - kh_k\sqrt{2(1-z/\rho)} + O(1-z/\rho).$$

Using this fact and the semi-large power theorem we obtain the evaluation of the coefficient of z^n in $H^d(z)$. □

We now have all the elements to prove the main result of this section.

Theorem 7.5.6. *[DS09] In a k-tree on n vertices, the probability that a random vertex r is at distance $d = x\sqrt{n}$ (with x in a compact of $\mathbb{R}_+^*$) from the first vertex v of the root, satisfies, as $n \to \infty$, a local law of the Rayleigh type:*

$$\lim_{n\to\infty} \sqrt{n}\mathbb{P}(D(v,r) = \lceil x\sqrt{n}\rceil) = h_k^2 x e^{-\frac{(h_k x)^2}{2}}, \text{ with } h_k = k\sum_{i=1}^{k} \frac{1}{i}.$$

Proof. The number of k-trees with a distinguished vertex at f-distance d are counted by the generating function $z^k \exp(T_{d,1}^{(u)}(z))$. Thus the quantity we are trying to evaluate is

$$\mathbb{P}(D(v,r) = d) = \frac{[z^n]z^k \exp(T_{d,1}^{(u)}(z))}{n[z^n]z^k \exp(T(z))}.$$

Given that $T_{d,1}^{(u)}(z)$ is equal to $H_1(z)T_{d-1,k}^{(u)}(z) = H_1(z)H^{d-2}(z)T_{1,k}^{(u)}(z)$ and since the singular expansions of $H_1(z)$ and $T_{1,k}^{(u)}(z)$ are similar to that of $T(z)$, the dominant contribution for the coefficient comes from the factor $H^{d-2}(z)$. □

7.5.1 Distances to a random vertex

To calculate the distance between two random vertices, we first distinguish a random vertex, calculate the total distances to all other vertices and divide by the size of the graph. Since we know how to calculate the distance to the first vertex of the root, the combinatorial idea [Pro80] is to “reroot” the k-tree in order to place the distinguished vertex in the position of first vertex of the root. Figure 7.6 illustrates the rerooting process.

Only in the case of non-increasing k-trees, this transformation can be directly operated on the corresponding trees and gives a family of simple trees $\mathcal{K}^\circ$ [DS09]. The generating function of this family is

$$K^\circ(z) = kz^k \exp T(z) + z^k \exp(T(z))T^\circ(z),$$

where $T^\circ(z)$ counts the trees in $\mathcal{T}$ with a mark on a white node:

$$T^\circ(z) = z\exp(T^k(z))(T^k(z) + kT^{k-1}(z)T^\circ(z)).$$

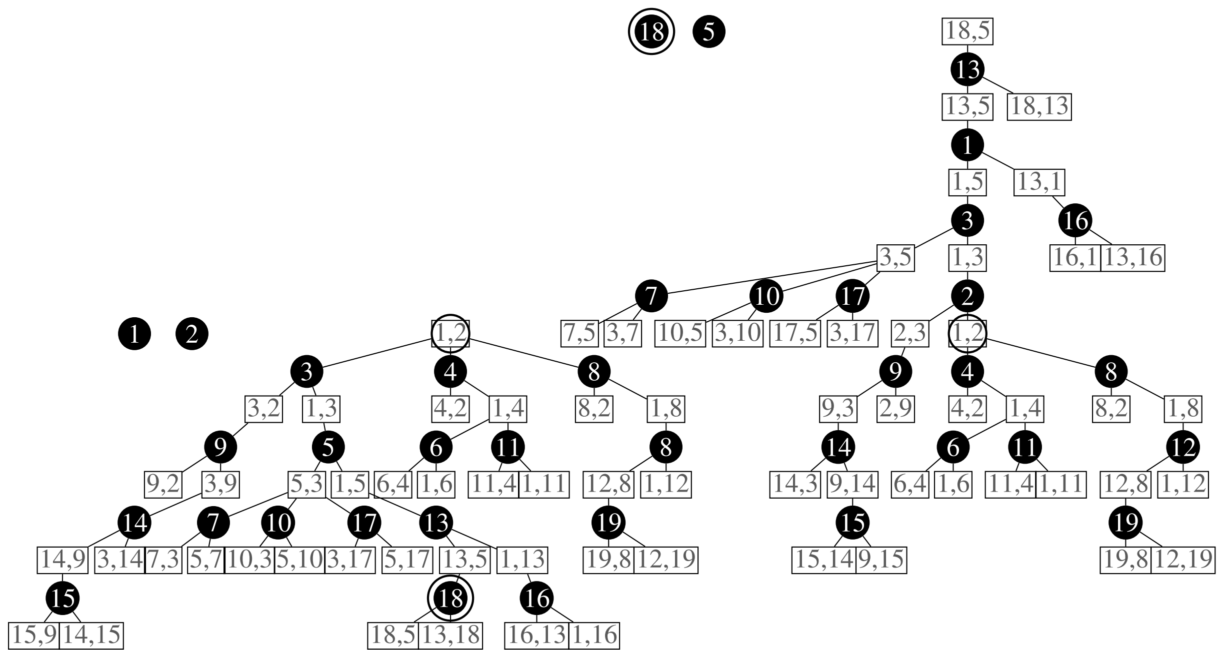


Figure 7.6: The tree representing a 2-tree with a distinguished vertex (18) and the corresponding rerooted tree.

These generating functions are very close to the initial generating function $K(z)$, thus confirming that the properties of the distance to a random vertex are similar to the properties of the distance to the first vertex of the root.

The rerooting algorithm and the proof of the bijection between class $\mathcal{K}^\circ$ and k-trees with a distinguished vertex are detailed in [DS09].

Figure 7.7 shows the distribution of distances from a random vertex, first as obtained from simulations on one thousand random k-trees and second as given by the following theorem.

Theorem 7.5.7. *Given a random k-tree G over n vertices, the distance between two random vertices v, w of G has mean value of order $\sqrt{n}$ and is asymptotically Rayleigh distributed in the range $x\sqrt{n}$:*

$$\lim_{n\to\infty} \sqrt{n}\mathbb{P}(D(v,w) = \lceil x\sqrt{n}\rceil) = h_k^2 x e^{-\frac{(h_k x)^2}{2}}, \text{ with } h_k = k\sum_{i=1}^{k}\frac{1}{i}.$$

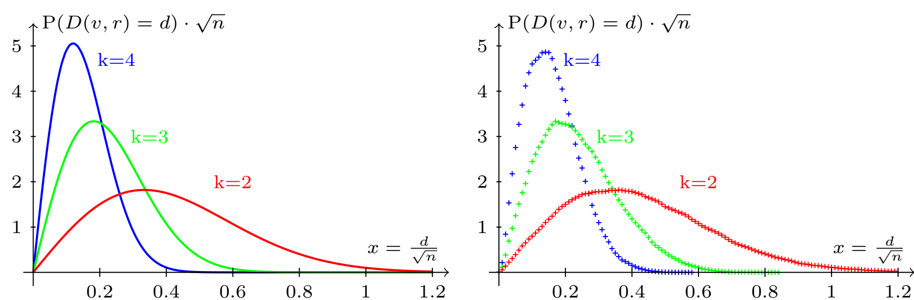


Figure 7.7: Theoretical (on the left) and experimental (on the right) distributions of distances in k-trees, for $k = 2, 3$ and 4. The experimental curves come from measures on 10^3 random k-trees of size $10^4 \pm 10\%$.

Deuxième partie

Applications de la génération d’arbres

Chapitre 8

Introduction

Une des problématiques de la combinatoire est d'estimer les propriétés d'une classe de structures, avec comme motivation principale l'analyse de la complexité des algorithmes. Par exemple, face à un algorithme sur les arbres binaires dont le temps d'execution est proportionnel à la hauteur de l'arbre donné en entrée, la propriété qui va permettre de donner une estimation du temps d'execution de l'algorithme est la hauteur de l'arbre.

Ce qui est vraiment intéressant n'est pas la hauteur d'un arbre particulier, mais la hauteur d'un arbre « typique ». Il faut donc d'abord quantifier la notion de « typique » et ensuite calculer la valeur de la propriété. La quantification la plus simple est la mesure uniforme, où chaque arbre a le même poids et à partir de l'uniforme on peut faire des modulations pour approcher d'autres mesures.

L'analyse mathématique étant souvent difficile, il est d'une grande utilité de pouvoir générer aléatoirement des structures pour faire de la simulation. Comme les estimations des paramètres ne sont valables que quand les tailles des objets générés sont suffisamment grandes, il faut des générateurs efficaces.

Les approches les plus répandues actuellement sont soit d'utiliser des générateurs *ad hoc* spécifiques à la classe étudiée soit d'utiliser la méthode récursive [FZC94]. Cette dernière propose une approche générique, transformant une spécification combinatoire en un générateur aléatoire uniforme. Les générateurs ainsi produits permettent de construire des structures avec quelques milliers d'éléments mais sont très gourmands en mémoire. L'introduction récente du modèle de Boltzmann, que nous avons présenté dans la section 2.3 permet d'atteindre des tailles de plusieurs millions en utilisant peu de mémoire tout en restant générique.

Ce chapitre présente l'utilisation du modèle de Boltzmann pour faire de la génération d'arbres. Cette description commence par une présentation rapide des trois contextes d'application sur lesquels cette méthode a été mise en œuvre. On présente ensuite l'algorithmique et l'implantation de l'oracle. Enfin on s'intéresse aux problèmes liées à la traduction des langages de spécification applicatifs vers les spécifications combinatoires.

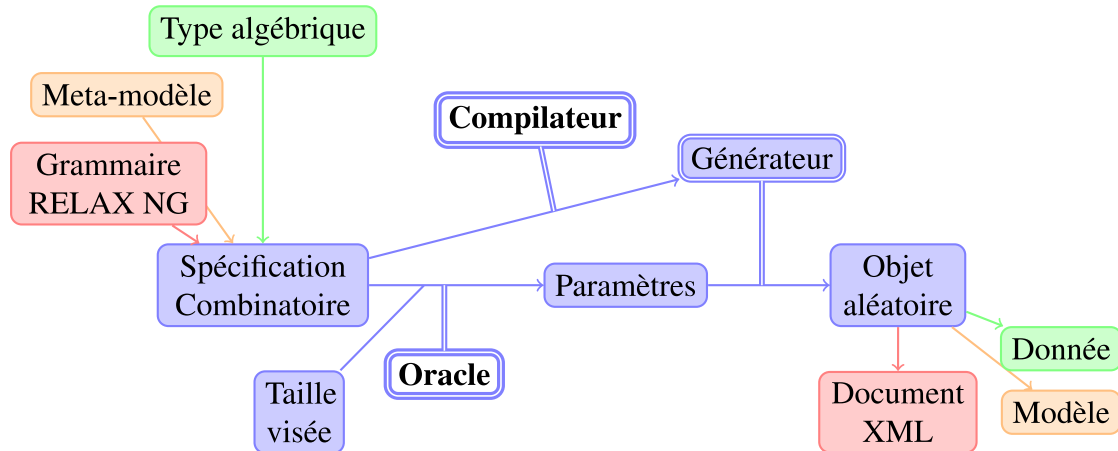


FIGURE 8.1 – Articulation de notre cadre de génération aléatoire de structures arborescentes.

8.1 Domaines d'application

Nous présentons dans cette séction les trois domaines sur lesquels nous avons appliqué la génération aléatoire d'arbre.

8.1.1 Types de données algébriques dans les langages fonctionnels

La récurrence joue un rôle central dans la programmation fonctionnelle et les structures arborescentes y sont donc omniprésentes. Par conséquent, la plupart des langages fonctionnels proposent des types de données algébriques pour structurer ces arbres. De plus le paradigme fonctionnel commence à s'introduire dans les langages de programmation les plus courants comme C♯ et Java.

Par ailleurs, le test des logiciels étant toujours une activité aussi importante que fastidieuse, son automatisation est un champ de recherche actif.

Test logiciel. Les pratiques du bon testeur de logiciels ont récemment été enrichies avec le test unitaire, qui consiste à donner une liste d'entrées et les sorties attendues correspondantes pour chaque fonction qu'on veut tester. On passe de l'approche traditionnelle globale où on teste toute l'application comme un ensemble à une approche locale, où on teste chaque unité fonctionnelle séparément.

Le test unitaire peut être couplé au test basé sur les spécifications, où plutôt que donner la liste des sorties attendues on spécifie des propriétés que les sorties doivent respecter, ce qui évite d'avoir à les écrire manuellement. Les entrées peuvent elles-mêmes être soit écrites manuellement, soit générées à partir d'une fonction dédiée, ou alors, dans les langages statiquement typés, le type de la fonction peut être utilisé pour créer automatiquement un générateur d'entrées.

Une utilisation complètement différente de l'aléa pour le test des logiciels est la technique du « fuzzing » [SGA07]. Cette approche consiste à partir d'une donnée valide et de faire des

petites modifications locales. Par exemple prendre un fichier vidéo et changer la valeur de quelques bits pris au hasard. Cela a permis de mettre en évidence un grand nombre de failles sur des logiciels de lecture de vidéo [Hoc].

On peut aussi utiliser la génération aléatoire pour générer des chemins dans le flot de contrôle d'un programme [DGGLP06].

Le travail présenté en chapitre 9 se place dans le cadre du test automatique basé sur des spécifications. Nous faisons de la génération aléatoire de données correspondant à un type de données algébrique et la mise en oeuvre est simplifiée par le fait que les spécifications des types algébriques sont très proches de celles des classes combinatoires d'arbres. Les outils actuels sont plutôt conçus pour traiter les types simples. Dans le cas des types de données algébriques, soit il est demandé à l'utilisateur de fournir lui-même les générateurs [CH00], soit les outils utilisent des générateurs exhaustifs ou résolvant des contraintes [RNL08], ce qui limite la taille maximale des données générées.

Le fait de générer des données de très grande taille est surtout utile pour tester la robustesse des systèmes. Cette génération aléatoire peut aussi sérvir à éstimer la complexité moyenne en temps des algorithmes. Dans la plupart des cas le calcul théorique est difficile et la simulation sur un ensemble de données aléatoires peut permettre d'éstimer la complexité. Pour que l'éstimation soit vraiment utilisable, il faut connaître la distribition sous-jacente à la génération et on donne alors au développeur un outil de benchmark appréciable.

Il faut cependant noter que beaucoup d'algorithmes prennent en entrée des données qui doivent vérifier des contraintes qui ne sont pas exprimables par types algébriques (p.e. arbres rouge-noir). Les seules solutions adéquates dans ce cas sont le rejet, qui peut être très couteux ; sauf dans certains cas où on sait traduire dans les types (p.e. arbres rouge-noir avec une hauteur bornée).

8.1.2 Modèles

L'ingénierie du logiciel a comme but de rendre possible le passage à l'échelle du développement logiciel. L'ingénierie dirigée par les modèles (Model Driven Engineering, MDE) utilise les modèles pour y parvenir.

Il est difficile pour un utilisateur non-informaticien et un développeur de communiquer sur le fonctionnement d'un logiciel. Il est important d'avoir une bonne adéquation entre le monde réel et sa représentation informatique et donc de développer un langage commun suffisament proche de la pensée de l'utilisateur et suffisament précis pour les traitements informatiques. Par ailleurs, lorqu'il s'agit de projets trop grands pour qu'une seule personne puisse maîtriser l'ensemble des détails, on perd la très grande efficacité observée dans les petits projets. L'utilisation des modèles permet de s'abstraire des détails techniques sans perdre des informations. Ainsi il est possible d'avoir une vision globale d'une complexité raisonnable qui garde un lien traçable avec le code qui l'implémente, mais aussi d'avoir un langage non-technique et précis.

Les modèles sont ainsi utilisés pour décrire les systèmes informatiques dans un langage graphique, comme par exemple UML. Le modèle produit peut être transformé pour arriver jusqu'au code source. L'automatisation de ces transformations est un des buts de l'ingénierie des modèles.

Les outils actuels de l'ingénierie des modèles ne passent souvent pas à l'échelle. Ils sont développés et testés en prenant en compte uniquement des modèles de petite taille. Des grands modèles sont souvent présents dans l'industrie, mais ne sont pas disponibles aux développeurs d'outils car les industriels ne veulent pas les diffuser. Il n'existe donc pas de jeu de tests standard pour comparer entre eux les différents outils.

L'outil de génération aléatoire que nous avons développé et que nous présentons dans le chapitre 10 permet d'engendrer des modèles de plusieures centaines de milliers d'élements et aura certainement sa place dans la panoplie d'un développeur d'outils pour l'ingénierie des modèles.

8.1.3 Documents XML

Le format XML est aujourd'hui omniprésent dans la sérialisation des données, que ce soit pour les stocker ou pour les échanger entre programmes. Il est la base d'une grande partie des formats les plus utilisés, comme XHTML, OpenDocument et XMPP.

Plusieurs langages de spécifications permettent d'écrire des grammaires pour valider des documents XML. Les plus utilisés sont DTD, qui est inclus dans la définition officiele de XML [W3C08], XMLSchema [W3C04] et Relax NG [OAS01]. Nous utiliserons ce dernier, car il est spécifié de manière rigoureuse et propose une sémantique proche de la combinatoire, tout en détournant son objectif initial, puisque il s'agit pour nous de générer des documents et non de les valider.

Savoir générer des documents XML aléatoires peut servir à tester les applications qui les utilisent. Notre approche, permettant de générer avec une complexité linéaire des documents XML arbitrairement grands est bien adaptée à vérifier la robustesse des ces applications, notamment des applications web qui doivent se prémunir des utilisateurs malicieux. En particulier les web services devraient arrêter rapidement et sans vagues le traitement de documents invalides, car lancer des traitements coûteux à la réception de très grands documents peut mettre l'application, ou même la machine qui l'héberge hors service.

La génération aléatoire des documents permet aussi de voir des coquilles dans les grammaires, en créant des constructions qui ne devraient pas apparaître.

Il existe plusieurs logiciels propriétaires qui contiennent des générateurs de documents XML à partir de grammaires Relax NG ou autres [Kaw ; Oxy ; Sty], mais tous utilisent des méthodes de génération *ad hoc* qui ne permettent pas de connaître la distribution de probabilités sous-jacente. Dans une approche complémentaire, le logiciel Fuzzware [Fuz] permet de randomiser un document existant.

La particularité de notre travail sur XML, présenté au chapitre 11, par rapport aux deux autres applications que nous avons considérées est la très grande taille de certaines grammaires. Par exemple la spécification des documents MathML est composée de près de 200 règles et notre précalcul de la génération se fait en moins de 20 secondes sur un ordinateur de bureau standard.

8.2 Algorithmes

Comme nous avons vu dans la section 2.3, un générateur de Boltzmann a besoin d'un paramètre qu'il faut calculer en fonction de la nature des objets qu'on veut générer et de la taille visée. Pour calculer ce paramètre nous avons besoin de savoir calculer la veur des séries génératrices associées à une classe combinatoire en un point donné.

8.2.1 Calcul des séries

Les séries sont définies dans notre cas par un système algébrique. Évaluer ces séries pour une valeur du paramètre donnée revient à trouver l'unique solution de ce système qui a un sens combinatoire. Résoudre un système algébrique est dans le cas général un problème difficile, trouver dans l'ensemble des solutions quelle est la bonne l'est tout autant.

La solution est de ne manipuler que des objets qui ont un sens combinatoire. Il s'agira dans un premier temps d'une méthode itérative pour construire tous les objets de la classe. Cette itération sur les objets peut s'appliquer directement sur les séries et même sur les valeur des séries en un point donné. On va donc obtenir une itération numérique qui a un sens combinatoire.

Parmi les méthodes possibles nous en considérons deux : l'itération simple, facile à comprendre mais avec une vitesse de convergence insuffisante ; l'itération de Newton [PSS08], adaptation de la méthode de Newton classique dans le cadre des séries issues de la combinatoire, avec une vitesse de convergence quadratique.

Itération simple. La forme du système, composé uniquement d'équations $\mathcal{T}_i = \phi(\mathcal{T})$ permet d'exécuter une itération simple, qui est détaillée dans l'algorithme 8.1. Prenant la classe vide comme point de départ, on applique les équations du système comme règles de construction d'objets. Si le système est bien fondé, on obtient tous les objets d'une taille n dans un temps $O(n)$. On peut faire cette même itération en remplaçant les classes d'objets par les séries génératrices, en partant de la série nulle et en itérant sur le système traduit en équations sur les séries génératrices, on convergera vers la série correspondant à la classe définie par le système. Enfin, en donnant une valeur au paramètre z et en partant de la valeur zéro, nous avons une itération numérique qui converge vers la valeur de la série au point z.

Algorithm 8.1 Itération-simple

Entrées : Une fonction $\vec{\phi}(z,\vec{y})$, une valeur du paramètre z et une précision ε
Sortie : Une approximation du vecteur $\vec{y}$, solution de $\vec{y} = \vec{\phi}(z,\vec{y})$

```
y⃗ ← 0⃗
y⃗' ← 0⃗
while y⃗' − y⃗ > ε do
    y⃗ ← y⃗'
    y⃗' ← φ⃗(z, y⃗)
return y⃗'
```

Newton combinatoire. La méthode de Newton nous permet de transformer une itération à convergence linéaire en une itération à convergence quadratique. Le fait que nos séries sont issues de la combinatoire nous donne un avantage important par rapport au cas général : nous avons la garantie que si nous partons du vecteur nul, alors l'itération va toujours converger vers la bonne solution, si le paramètre est dans le disque de convergence de la série, et elle va diverger sinon. L'algorithme 8.2 est presque le même que celui de l'itération simple, sauf que l'appel à la fonction $\vec{\phi}$ est remplacé par un appel à l'algorithme 8.3.

Algorithm 8.2 Newton-combinatoire

Entrées : Une fonction $\vec{\phi}(z,\vec{y})$, une valeur du paramètre z et une précision ε
Sortie : Une approximation du vecteur $\vec{y}$, solution de $\vec{y} = \vec{\phi}(z,\vec{y})$

```
  ȳ ← 0̄
  ȳ' ← 0̄
  U ← I
  while ȳ' − ȳ > ε do
    ȳ ← ȳ'
    ȳ', U ← Pas-itération-Newton(φ̄, z, U, ȳ)
  return ȳ'
```

$\vec{y} \leftarrow \vec{0}$
$\vec{y}' \leftarrow \vec{0}$
$U \leftarrow I$
while $\vec{y}' - \vec{y} > \varepsilon$ **do**
  $\vec{y} \leftarrow \vec{y}'$
  $\vec{y}', U \leftarrow$ Pas-itération-Newton$(\vec{\phi}, z, U, \vec{y})$
return $\vec{y}'$

Algorithm 8.3 Pas-itération-Newton

Entrées : Une fonction $\vec{\phi}(z,\vec{y})$, un vecteur $\vec{y}$, une matrice U et une valeur du paramètre z
Sortie : La valeur, après une itération de l'algorithme de Newton, du vecteur $\vec{y}$ et de la matrice U

$U \leftarrow U + U(\frac{\partial}{\partial \vec{y}}\vec{\phi}(z,y)U - (U - I))$
return $\vec{y} + U(\vec{\phi}(z,\vec{y}), U$

8.2.2 Calcul du paramètre

Comme nous avons vu dans la section 2.3 l'efficacité d'un générateur de Boltzmann repose sur le bon choix du paramètre. La nature de la classe qu'on veut générer influe sur la manière de choisir le paramètre. Pour cette application nous distinguons deux cas : celui des arbres, dont la série génératrice $T(z)$ a une singularité en $\rho \in \mathbb{R}^+$ et une valeur $T(\rho) \in \mathbb{R}^+$ et les autres, qui incluent les listes dont la valeur de la série génératrice diverge en la singularité et les classes finies qui ont une série génératrice définie sur l'ensemble des réels.

Si notre classe n'est pas un arbre alors il faut choisir le paramètre x en fonction de la taille moyenne n visée :

$$n = x\frac{T'(x)}{T(x)}.$$

Si notre classe est un arbre, deux stratégies sont possibles, toutes deux basées sur la génération singulière avec rejet anticipé :

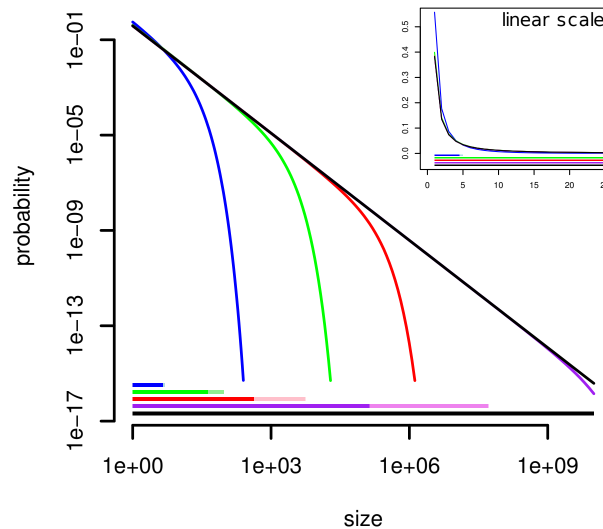


FIGURE 8.2 – La distribution théorique des tailles des arbres générés par la méthode de Boltzmann, avec un paramètre $x = 0.9\rho, 0.999\rho, 0.99999\rho, 0.9999999999\rho$, et ρ. La figure principale est en échelle log-log, tandis que la figure encadrée a une échelle linéaire.

– soit on essaye de calculer exactement la position de la singularité, ce qui demande une implémentation sophistiquée pour affiner la précision du calcul au fur et à mesure de la génération ;
– soit on calcule la singularité avec une précision fixée, ce qui revient à se placer un peu avant la singularité et donc introduire un seuil sur les tailles des objets produits par le générateur.

La deuxième stratégie est beaucoup plus facile à mettre en œuvre et marche bien en pratique. La taille maximale atteinte en calculant la singularité avec une double précision est de l'ordre de la mémoire vive des machines actuelles.

La figure 8.2 nous montre la distribution des tailles des objets produits par un générateur de Boltzmann pour une classe d'arbres et pour différentes valeurs de précision du calcul de la singularité. Il faut noter que la queue de la distribution est la même pour presque toutes les classes d'arbres, ce qui permet de choisir la précision indépendamment de la définition de la classe.

Recherche dichotomique. Quelle que soit la nature de la classe, on utilise une recherche dichotomique pour calculer la valeur du paramètre, comme illustré dans l'algorithme 8.4. Cela est possible car l'algorithme de Newton diverge si on choisit un paramètre plus grand que la singularité et converge si on choisit un paramètre plus petit que la singularité. On commence en calculant la valeur de la série quand le paramètre vaut 1.

Si l'algorithme de Newton retourne une valeur, cela veut dire que le rayon de convergence de la série génératrice de notre classe est infini. Il faut donc chercher le paramètre qui donne la bonne taille moyenne, en utilisant le fait que cette moyenne est une fonction strictement croissante du paramètre, ce qui est fait par l'algorithme 8.6.

Si l'algorithme de Newton diverge pour une valeur du paramètre égal à 1 on cherche la

valeur de la singularité entre 0 et 1 en utilisant la convergence de l'algorithme de Newton comme test (algorithme 8.5).

Si la valeur de la série diverge quand la valeur du paramètre tend vers la singularité par la gauche, notre classe n'est pas une classe d'arbres. Il faut donc encore une fois chercher le paramètre qui donne la bonne taille moyenne.

Si la valeur de la série converge sur la singularité nous avons une classe d'arbres et nous allons utiliser la valeur approchée de la singularité comme paramètre.

Algorithm 8.4 Chercher-paramètre

Entrées : Une série génératrice C, une taille moyenne visée n et une précision ε
Sortie : La valeur du paramètre de Boltzmann

```
if C(1) < ∞ then
  x ← 1
  while x C'(x)/C(x) < n do
    x ← 2x
  return Chercher-moyenne(C, n, 0, x, ε)
ρ ← Chercher-singularité(C, 0, 1, ε)
if C(ρ) < ∞ then
  return ρ
else
  return Chercher-moyenne(C, n, ε)
```

Algorithm 8.5 Chercher-singularité

Entrée : Une série génératrice C, une valeur minimale $x_{\min}$, une valeur maximale $x_{\max}$ et une précision ε
Sortie : Une valeur dans l'intervalle $[\rho_C - \varepsilon, \rho]$

```
if x_max − x_min < ε then
  return x_min
x ← (x_max + x_min)/2
if C(x) < ∞ then
  return Chercher-singularité(C, x, x_max, ε)
else
  return Chercher-singularité(C, x_min, x, ε)
```

Algorithm 8.6 Chercher-moyenne

Entrée : Une série génératrice C, une taille moyenne visée n, une valeur minimale $x_{\min}$, une valeur maximale $x_{\max}$ et une précision ε

Sortie : La valeur du paramètre de Boltzmann

if $x_{\max} - x_{\min} < \varepsilon$ **then**
  return $x_{\min}$
$x \leftarrow \frac{x_{\max}+x_{\min}}{2}$
if $x\frac{C'(x)}{C(x)} < n$ **then**
  return Chercher-moyenne($C, n, x_{\min}, x, \varepsilon$)
else
  return Chercher-moyenne($C, n, x, x_{\max}, \varepsilon$)

8.3 Réalisation

Nous avons implémenté ces algorithmes avec les objectifs suivants :

- l'interoperabilité avec un maximum de langages de programmation, pour ne pas se limiter lors du développement des applications spécifiques,
- l'efficacité, pour traiter des spécifications de très grande taille et
- la possibilité de faire les calculs avec une précision arbitraire.

Le choix c'est donc porté naturellement à une librairie écrite dans le langage C.

Cette réalisation est complémentaire de l'implémentation originale de la méthode de Newton combinatoire en Maple par Carine Pivoteau. Cette dernière traite beaucoup plus de constructions combinatoires, et notamment les structures non-étiquetées pour lesquelles l'utilisation d'un système de calcul formel est quasi-inévitable.

L'implémentation directe des algorithmes présentés dans la section précédente est facile à réaliser et permet de traiter de manière satisfaisante tous les cas où la grammaire combinatoire est petite. C'est le cas, par exemple, de la très grande majorité des types de données algébriques.

Si nous avons à traiter une grammaire avec un grand nombre d'équations N, la complexité de l'itération a deux sources. La première est la multiplication de matrices $N \times N$, où N est le nombre d'équations. La deuxième, l'évaluation d'un ensemble fixé de N^2 expressions arithmétiques pour différentes valeurs de leurs variables. Notre implémentation essaye de diminuer le coût de ces opérations : les précalculs non-optimisés de la génération de documents MathML sur un ordinateur de bureau ordinaire durent 1m33. Avec une multiplication de matrice rapide on passe à 54s, avec une précompilation des expressions arithmétiques à 57s et avec les deux optimisations combinées à 18s.

Pour accélérer les multiplications des matrices, on utilise la librairie BLAS [Bla+02] qui fourni des implémentations optimisées des opérations de base de l'algèbre linéaire. Il existe plusieurs implémentations de cette librairie, certaines génériques et d'autres optimisées pour un type de machines particulier. Dans tous les cas on gagne au moins un ordre de grandeur par rapport à une implémentation naïve de la multiplication de matrices.

Pour accélérer l'évaluation des expressions arithmétiques, on fait de la compilation à la volée. Dans une implémentation naïve, la représentation symbolique d'une expression est interprétée chaque fois que nous voulons l'évaluer avec une valuation donnée de ces variables. La solution est de transformer la représentation symbolique en code machine, dont l'évaluation est un ordre de grandeur plus rapide. Pour ce faire on utilise la librairie GNU lightning [Foua] qui réalise cette optimisation sans perdre la portabilité du code. Elle permet d'écrire dans un langage d'assembleur agréable qu'elle traduit vers le langage assembleur de la machine visée.

Une optimisation présente dans l'implémentation en Maple utilise le fait que très souvent la grammaire peut être décomposée en composantes fortement connexes de taille beaucoup plus petite. Dans ce cas nous pouvons appliquer l'itération de Newton séparemment sur chacune des composantes, et la complexité est quadratique en la taille de la plus grande composante connexe. La mise en place de cette optimisation dans notre prototype est en cours.

Notre implémentation utilise pour l'instant des nombres flottants de double précision. Il faudrait cependant une précision arbitraire pour atteindre des tailles extrêmement grandes. L'utilisation de la précision arbitraire augmenterai le temps de calcul même dans les cas où elle n'est pas nécessaire et elle demande des versions spécialisées des librairies d'algèbre linéaire. Cela se fera donc dans une réalisation parallèle, quand les utilisateurs des générateurs auront besoin d'objets plus grands que ceux atteignables avec la précision actuelle.

8.4 Traduction

Pour pouvoir utiliser la génération de Boltzmann dans un domaine d'application on a besoin de décrire les objets de ce domaine dans le langage de la méthode symbolique. Pour les trois applications qu'on a mis en place ce procédé est détaillé dans les chapitres qui suivent. L'objet de cette section est d'en extraire quelques principes généraux.

Classe combinatoire. Dans une classe combinatoire tous les objets sont de taille finie et pour chaque taille il existe un nombre fini d'objets différents. Au niveau des objets d'un domaine applicatif nous devons définir la taille d'un objet et la notion d'égalité entre deux objets pour pouvoir décrire exactement l'ensemble des objets d'une taille donnée et en faire une classe combinatoire.

Considérons un exemple simple, une application dont les objets sont des arbres binaires avec des entiers positifs sur les nœuds. Si on dit que la taille d'un arbre est son nombre de noeuds et que deux arbres sont égaux si ils sont identiques à la fois pour leurs structures et pour les entiers stockés dans le nœuds, alors nous n'avons pas une classe combinatoire. Il existe une infinité d'arbres réduits à une feuille, un pour chaque entier. On peut résoudre ce problème en changeant la notion d'égalité, par exemple en regardant uniquement la structure arborescente et pas les entiers stockés dedans. On peut aussi changer la notion de taille, dire par exemple que la taille d'un arbre est la somme des entiers qu'il contient. La pertinence des solutions dépend bien-sûr de l'application.

Grammaire non-contextuelle. Pour pouvoir utiliser les constructeurs de la méthode symbolique il est nécessaire de disposer d'une grammaire non-contextuelle pour décrire la classe des objets à générer. Certains domaines applicatifs, comme XML, fournissent des grammaires que nous pouvons directement traduire, alors que dans le cas où la spécification est contextuelle, voir même implicite, il faut créer la grammaire correspondante.

Parfois la classe qui nous intéresse ne peut pas être décrite dans le cadre de la méthode symbolique, mais on peut en décrire une approximation arborescente. Il peut s'agir de structures qui ne sont pas des arbres, parce qu'elles contiennent des liens transversaux, comme par exemple dans les méta-modèles. On peut aussi rencontrer des structures avec un squelette arborescent mais des feuilles arbitrairement complexes et structurées, comme c'est le cas dans les documents XML. Notre approche pour générer des objets de ces classes est d'utiliser la méthode de Boltzmann pour générer la structure arborescente sous-jacente et une méthode *ad hoc* pour compléter l'objet, en essayant de contrôler le biais engendré.

Grammaires avec poids. Nous avons souvent besoin de générateurs qui suivent une distribution autre que l'uniforme, car les propriétés des objets aléatoires uniformes ne sont pas toujours les mêmes que celles rencontrés dans le terrain. Il faut que les objets générés ressemblent à ceux de la « vraie vie ».

Il est facile de biaiser la génération en ajoutant des poids dans la grammaire [DRT00]. La manière dont nous avons procédé dans les applications que nous avons traitées, est d'essayer différentes pondérations jusqu'à ce que les objets générés « ressemblent » à des « vrais » objets, aux yeux d'un spécialiste du domaine. L'avantage de cette approche est qu'il n'est pas difficile ensuite de calculer la probabilité pour un objet donné d'être généré.

L'autre approche est d'essayer de calculer les pondérations pour que les objets générés aient des valeurs données pour certaines paramètres, le but à terme étant de permettre à l'utilisateur de donner des valeurs de certains paramètres et d'avoir un générateur qui produit des objets avec ces paramètres. C'est ce type d'approche qui est utilisé dans le cadre de la génération de structures génomiques [PTD06 ; Pon08] avec une méthode récursive, qui va toujours générer des objets d'une taille donnée et avec les paramètres donnés. Avec la méthode de Boltzmann la taille et les paramètres seraient aléatoires avec des moyennes égales aux valeurs fournies. La distribution des paramètres dépend de la nature du paramètre : elle peut être concentrée, auquel cas on aura des valeurs du paramètre proches de celle voulue, ou alors étendue avec des valeurs très variables. C'est donc une approche non-triviale mais des travaux prometteurs [PB09] sont en cours.

Cet article est tiré des actes du ML Workshop 09 [CD09].

Chapter 9

Fast and sound random generation for automated testing and benchmarking in Objective Caml

Numerous software testing methods involve random generation of data structures. However, random sampling methods currently in use by testing frameworks are not satisfactory: often manually written by the programmer or at best extracted in an ad-hoc way relying on no theoretical background. On the other end, random sampling methods with good theoretical properties exist but have a too high cost to be used in testing, in particular when large inputs are needed.

In this paper we describe how we applied the recently developed Boltzmann model of random generation to algebraic data types. We obtain a fully automatic way to derive random generators from Objective Caml type definitions. These generators have linear complexity and, the generation method being uniform, can also be used as a sound sampling back-end for benchmarking tools.

As a result, we provide testing and benchmarking frameworks with a sound and fast generation basis. We also provide a testing and benchmarking library, available for download , showing the viability of this experiment.

9.1 Introduction

Testing has always been a major part of software development but also one of the most tedious. In particular, testing algorithms requires the programmer to write a lot of input data to cover a significantly wide amount of cases. The same input generation problem appears in benchmarking: when one wants to compute a practical average efficiency. Moreover, in this case, one also wants the generated input to uniformly cover the set of all possible inputs of a given size in order to soundly obtain statistical properties.

In our context of statically typed languages with algebraic data-types, we have an exact and

complete formal definition of the shape of the values (types such as integers, strings, etc. will be treated as leaves throughout this paper). It is then straightforward that values of such a type can be automatically generated by extracting random generators from type definitions. Moreover, functional languages appear to be a very good environment for fully automatic specification-based testing techniques, as shown by QuickCheck [CH00] [1].

However even if there have been numerous experiments on fully automatic testing, the random generation methods currently in use are not satisfactory enough. Input data is generated

- by handcrafted generators,
- by automatically extracted generators with no theoretical background to ensure probabilistic properties of the generation, or
- by a constraint-based method which is better on a theoretical point of view but too expensive and restricts it to small sizes.

For all of these methods, it is usually extremely difficult to calculate the probability for a given value to appear and thus to estimate the bias of the generation.

Creating random generators with good properties is one of the topics in the field of combinatorics. The methods it proposed until recently were however not applicable in this context. They were either specific to some structure and thus not automatically applicable to large enough classes of objects or in the case of the recursive method [FZC94] too costly (with a simple implementation) or too complicated.

In this paper, we propose a solution to solve this problem by using the recently developed Boltzmann model of random generation from the combinatorics research field. The Boltzmann model provides uniform random sampling, meaning that each object in the selected class has the exact same probability to be produced. Moreover, the generators it produces are very simple and have a linear-time complexity. We designed a translation from Objective Caml [LD-GRV08] algebraic data-types to combinatorics specifications, and use it to automatically extract generators from type definitions.

Practically, we provide testing framework designers with a random sampling core for generating (possibly very large) objects that respect a given algebraic data-type. The probability for an object to appear is known and the cost of the generation is linear to the size of the object. We integrated this generation into Objective Caml via a syntax extension and tested its viability on a testing and benchmarking library prototype.

This work can be directly adapted to any functional language. It can also be used to generate tree-like structures in any programming language.

In section 9.2, we give an overview of software testing methods, explain how our work can interact with them and present some related work in typed functional languages. Section 9.3 first gives the theoretical background and section 9.4 explains the application to random sampling of algebraic data-types in Objective Caml. Then, section 9.5 demonstrates the practical applications of our random generation core and our associated testing and benchmarking library

1. We use the notation (*n*) to reference external links; see section Links at the end of the document for full URLs.

prototype in the form of a tutorial. We then give some performance results on the generation speed in section 9.6. Finally, we present our future work on this project as well as other ongoing experiments involving the Boltzmann model and programming languages.

9.2 Context

In this section, we describe the many sorts of source code level program testing frameworks. We then explain how these framework can integrate our work, and why they should.

Program testing The most widespread software testing technique in use in software development industry is called *unit testing*. In a regular industrial development environment, the work is organized around a well defined methodology called a *development model*. The most venerable is the *V-Model*, which splits the software development work in layers of abstraction, going from global conception of the system to detailed source code level development. To each conception task, this model associates a verification one. Unit testing consists in finding tests cases and procedures for each unit of code (methods, functions, procedure, etc. depending on the programming language). It is therefore the counterpart of detailed software conception in this model.

Independently from, and for best results in addition to, unit testing, there exists another method usually called *specification-based testing*. This approach tests a function by applying a validation predicate over pairs of values from its domain and the associated results. Whereas the goal of unit testing was to check that the function performs correctly on hand-written typical or pathological input values, specification-based testing tries to find bugs by checking the results of the function over a significant amount of inputs. The problem, which we are not addressing, is then how to define which inputs are *significant* and how to obtain them.

If the domain of the function to test is finite and small enough, the programmer can write every case and then obtain an exhaustive test. Since we focus on a functional language with well defined algebraic data-types, the procedure to obtain all the values of a given finite type can be automatically derived from its definition. By defining a size operator over a non-finite type, exhaustive testing can also be used to test the function for inputs up to a given size. This approach has already been experimented in [RNL08] for the Haskell language.

The problem is then how to obtain *meaningful* sample inputs of large size. There are several methods for generating random values of a given type. In most test frameworks, these values are created by hand-crafted generators. QuickCheck, for example, defines several random generators for basic types and combinators to build from them generators for more complex types. With such a manual approach, it is very hard not to bias the form of generated values, and thus to unknowingly concentrate the domain of tested values to an arbitrary subset of values. A solution to control the shape of generated values is to use constraints-based random generation. However, it is not a viable solution in terms of generation time for large values.

Our proposition The solution we propose is to use uniform random generation. In other words, a generator of values of a given size gives each value of this size the exact same probability to appear. With such an approach, we know that we are concentrating on the subset of

the most representative values. To achieve this property, generators are not hand-written but automatically derived from a type definition as we explain in the following section. Moreover, the derived generators are capable of producing values of a given size in linear complexity. Furthermore, we provide a mechanism of adding a measurable bias on the distribution, in order to target different value subsets.

Applications Our work can thus provide any kind of testing framework based on random generation with a sound and fast generation basis, in particular for inputs of large size.

Moreover, as we have just said, we use uniform (unbiased) random generation. From a statistical point of view, this means that if we run a test over a great number of generated values, we can obtain statistical properties over the values of given type and size.

In particular, we can obtain a good evaluation of average performance of programs. Of course, this technique is not suited to obtain worst/best case complexity. This can be used to obtain the practical performance of a program, since obtaining a theoretical average complexity is often tedious, and many algorithms with bad worst-case complexity can be very efficient in practice because of their average behaviour. Moreover, it can reveal errors like complexity miscalculations due to the use of an unsuited underlying primitive data structure. Our work can thus be used use as well by mean-time complexity checking or benchmarking tools.

Related works In this paper, we argue that the most widesreapd testing frameworks lack a sound generation basis. We propose the use of uniform generation and present some advantages of this method. For instance, uniformity of generators prevent them from systematically missing subsets of input values because of a bias. Other research works share the same goal to add a theoretical background to test input generation. We can cite [SMA05], focusing on the notion of coverage thus generating inputs to exercise the maximum number of control paths. In a nearer field, we can also mention [FK08] in which the authors define a notion of data-flow coverage of declarative programs and produce generators accordingly.

On the benchmarking side, we use uniformity to give practical information on the performance of the program, as well as a sound statistical measure of average complexity. Other works focus on producing inputs to show worst case complexity [BJS09] and could be used to obtain complementary complexity information.

Related OCaml projects Research experiments on automatic code generation from Objective Caml type definitions have already been leaded in the past, including random generation (without theoretical background). We can cite for example OCaml Templates [Mau04] or multi-stage [Tah04] programming with MetaOcaml . Nowadays, efforts in the community are done to work with types. We use type-conv for type-definition preprocessing code and dyn for run-time type information. Other research solutions exists, one can cite for example [HMC07] which modifies the compiler to add first-class run-time type representation to the language.

On the side of testing frameworks, a few tools exist and are maintained for the Objective Caml language, namely oUnit and mlquickcheck which have recently been unified by the Kaputt project. Other functional languages have more widely used tools like Haskell with QuickCheck [CH00].

9.3 Underlying theory: the Boltzmann model

Our approach is based on the random sampling of combinatorial structures, within the frame of the Boltzmann model, as introduced by [DFLS04]. The main feature of this model is uniform generation with linear complexity, thus allowing for generation of much larger objects than was possible before.

Given a finite class of objects $\mathcal{C}$, the generation is uniform, meaning that any object of $\mathcal{C}$ is produced with equal probability $1/||\mathcal{C}||$, where $||\mathcal{C}||$ is the number of objects in $\mathcal{C}$. In the Boltzmann model, each object γ, with size $|\gamma|$, is generated with probability proportional to $x^{|\gamma|}$, where x is a control parameter, thus the generation is uniform for a sub-class of objects of the same size.

Though uniform for a given size, the size distribution of Boltzmann sampling is spread over the whole class $\mathcal{C}$ (this is different from most random generators, that, given a size n, output a random object of size exactly n). However the expected size of generated objects can be tuned by the choice of parameter x, thus giving approximate size sampling as detailed in section 9.3.3. In the case of testing, this feature is not a restriction: when generating a large set of very large objects, say with one million elements, the exact size of the objects is not relevant up to a few percent. And the important result is that relaxation of exact size from a small percentage is rewarded by a linear time complexity of generation.

The Boltzmann method is generic and can be applied to classes described by specifications based on a rich set of constructors, such as disjoint union, Cartesian product, sequences, sets, cycles, ... (as illustrated in figure 9.1 by some tree class specifications). The method relies on transforming a system of specifications into a system of functional equations involving generating functions, and working on these functions with analytical techniques (this is the domain of analytical combinatorics, described in [FS09]). By these means, sampling can be automatically compiled from specifications.

The generation algorithms are very simple: for instance generating a couple of objects (a,b) in class $\mathcal{A} \times \mathcal{B}$, simply reduces in independently generating $a \in \mathcal{A}$ and $b \in \mathcal{B}$, and the probability is correct; for disjoint union, generating an object in $\mathcal{A} \cup \mathcal{B}$ is obtained by throwing a biased coin and derive either a generation in $\mathcal{A}$ or a generation in $\mathcal{B}$. In this case, the bias of the coin is computed by evaluating generating functions at parameter x.

Boltzmann samplers are particularly efficient if we accept some variability in the size of the generated structures: fixing a target size n and a margin of error ε, generating random structures untill we get one of size belonging to $[(1-\varepsilon)n, (1+\varepsilon)n]$ can be completed in mean time $O(n)$ (whereas exact size average complexity can be up to quadratic). This *relaxed size* generator has a measurable bias towards smaller sized trees which should not be a problem for practical applications.

This section is continued with a presentation of Boltzmann generation of trees. We first explain how to compute the parameters for the generator in order to obtain a linear complexity generation. Then we present the notion of *tree specification* which will be used for specifying the structure of the trees to be generated. Finally, we describe how to automatically derive a

Tree type	A corresponding grammar
Ternary trees:	$\mathcal{T} = \mathcal{Z} + \mathcal{T} \times \mathcal{T} \times \mathcal{T}$
One-two trees:	$\mathcal{T} = \mathcal{Z} + \mathcal{U} + \mathcal{B}$, $\mathcal{U} = \mathcal{Z} \times \mathcal{T}$, $\mathcal{B} = \mathcal{Z} \times \mathcal{T} \times \mathcal{T}$
General trees:	$\mathcal{T} = \mathcal{Z} \times \mathcal{F}$, $\mathcal{F} = \text{Seq}(\mathcal{T})$

Figure 9.1: Examples of tree specifications.

parametrized uniform tree generator that follows such a specification.

9.3.1 Generating functions

The Boltzmann method applies to the generation of structured objects, using the powerful tool of *generating functions*. Given a class $\mathcal{C}$, we consistently denote by $C(z)$ its generating function, which is the series $C(z) = \sum_{\gamma \in \mathcal{C}} z^{|\gamma|} = \sum_n c_n z^n$, where c_n is the number of objects of size n in $\mathcal{C}$.

The symbolic method [FS09] provides a dictionary for translating structural constructions into operators on generating functions: concerning tree constructions, the dictionary reduces to

$$\begin{aligned} \mathcal{C} = \mathcal{A} + \mathcal{B} &\quad\rightarrow\quad C(z) = A(z) + B(z), \\ \mathcal{C} = \mathcal{A} \times \mathcal{B} &\quad\rightarrow\quad C(z) = A(z) \cdot B(z), \\ \mathcal{C} = Seq(\mathcal{A}) &\quad\rightarrow\quad C(z) = \frac{z}{1 - A(z)}. \end{aligned}$$

Thus in a tree specification, each production rule (non-terminal elements) transforms into a corresponding *generating function* equation, and a grammar transforms into a polynomial system of equations.

To generate trees from these specifications, we need to evaluate these functions for a given value x of variable z by solving such systems of equations. The resolution is analytically coherent for $0 < x \leq \rho$, where ρ is a special value, called the *singularity* of the system.

Solving polynomial systems of equations is a very complex problem in general, but systems corresponding to specifications do have a structure, that can be exploited in the computations. In our implementation we use a combinatorial newton method that gives a very efficient solver [PSS08], that can also be used to calculate an approximation of the singularity ρ.

In figure 9.2, we show the generating functions for the previously introduced tree specifications and the value ρ for each of these systems.

In each case, using the values of these functions at $x = \rho$, the Boltzmann algorithms of section 9.3.2 derive a linear time generator with the property of *uniformity*: given a size n, two trees of that size have exactly the same probability of being generated. These generators however have the particularity that the generated trees are not all of size n, but have a random size, with a mean value depending on parameter x. We show in section 9.3.3 how to deal with this aspect, using ρ as the value for x.

Tree type	Generating functions	ρ
Ternary trees:	$T(z) = z + T^3(z)$	$\rho = 2\sqrt{3}/9$
One-two trees:	$T(z) = z + U(z) + B(z)$ $U(z) = z \cdot T(z)$ $B(z) = z \cdot T^2(z)$	$\rho = 1/3$
General trees:	$T(z) = z \cdot F(z)$ $F(z) = \frac{1}{1-T(z)}$	$\rho = 1/4$

Figure 9.2: The generating functions and radius of convergence ρ of the example grammars.

9.3.2 Generation algorithms

In this paper, a *tree specification* will be a unambiguous context-free grammar with one terminal, $\mathcal{Z}$ and three operators: a unary operator to create a *Sequence* (denoted by Seq), where a sequence is made of and arbitrary number $k \geq 0$ of trees (an empty sequence is $\mathcal{Z}$); and two binary operators to compose trees: *Union* (denoted by $+$) and *Product* (denoted by $\times$).

The *size* of a tree T, denoted by $|T|$, will be the number of $\mathcal{Z}$ it contains. This means that the size of a tree is its number of nodes. Other choices for the size are of course possible, as long as there is always a finite number of trees of a given size.

A tree specification can be automatically transformed to a random generation algorithm. Let's see how to do it.

Each production rule of the specification describes a non-terminal. We deduce from it an algorithm to generate objects defined by this non-terminal and the corresponding generating function (we note $A(z)$ the generating function of $\mathcal{A}$). We detail here how to interpret the terminals and operators in the tree specification to obtain these.

$\mathcal{Z}$: the $\mathcal{Z}$ element in the tree specification corresponds to one size unit. Wherever there is a $\mathcal{Z}$ in the specification an object of size one is to be generated. The generating function of $\mathcal{Z}$ is z.

$\mathcal{B} \times \mathcal{C}$: first generate independently both an element $b \in \mathcal{B}$ and an element $c \in \mathcal{C}$. The result will be the couple (b, c). The corresponding function is $B(z) \cdot C(z)$.

$\mathcal{B} + \mathcal{C}$: with this construction, either an element in $\mathcal{B}$ or in $\mathcal{C}$ will be generated. The probability of generating in $\mathcal{B}$ is $B(z)/(B(z)+C(z))$, and the probability of generating in $\mathcal{C}$ is symmetric. A pseudo-random number is used to determine which element should be generated, with respect to the given probability. The generating function is $B(z) + C(z)$.

Seq($\mathcal{B}$): first the number k of components in the sequence is drawn, following a geometric law with parameter $B(z)$, and then k elements of type $\mathcal{B}$ are independently generated and returned as a sequence. The corresponding function is $\frac{z}{1-B(z)}$.

Figure 9.3 shows the generation algorithms for the specifications given in figure 9.1.

Ternary trees:
TTree () = **if** random 0 1 $< x/T(x)$ **then** Leaf ()
else Node (TTree ()) (TTree ()) (TTree ())

One-two trees:
OTTree () = **let** r = random 0 1 **in**
if $r < x/T(x)$ **then** Leaf ()
elsif $r < (x+U(x))/T(x)$ **then** UTree ()
else BTree ()
UTree () = UNode (OTTree ())
BTree () = BNode (OTTree ()) (OTTree ())

General trees:
GTree () = GNode (Forest ())
Forest () = **let** k = geom $T(z)$ **and** res = ref [] **in**
for i = 1 **to** k **do** res := GTree ()::!res **done;**
!res

Figure 9.3: The generation algorithms for the example grammars.

9.3.3 Parameter tuning and complexity

With the Boltzmann method, the size of generated trees is random, with a distribution that depends on the specification $\mathcal{C}$ and a mean value that goes from 0 to infinity when parameter x goes from 0 to ρ, and is equal to $xC'(x)/C(x)$. The probability for the result to be of size n is $c_n x^n \rho^{-n}$, which for most tree specifications and for large n is proportional to $n^{-\frac{3}{2}} x^n \rho^{-n}$. In all cases, the closest is x to the value of ρ, the biggest is the probability of generating large size trees.

The precise size distribution of the generator depends on the nature of the tree structure. We will identify three cases: finite classes, lists and trees [2], each case demanding a different strategy for the choice of parameter x in order to generate objects of size approximately n with a linear complexity (including the cost to generate trees outside the size target, that will be thrown away). The precise definition of the classes depends on the corresponding generating functions, allowing for an automatic classification of specifications. Finite classes have a value of $\rho = \infty$. The generating function of lists tends to infinity when z tends to ρ, while that of trees tends to a finite value.

For lists and finite classes, we have to choose x such that the mean size of the generated objects equals n. In that case, the theory [DFLS04] guarantees that there will be a constant number of rejects before generating an object of size approximately n. Therefore the mean time complexity is linear.

In the case of trees, linear time complexity can be achieved by using either *pointing* or

2. some structures may look like trees but are actually lists

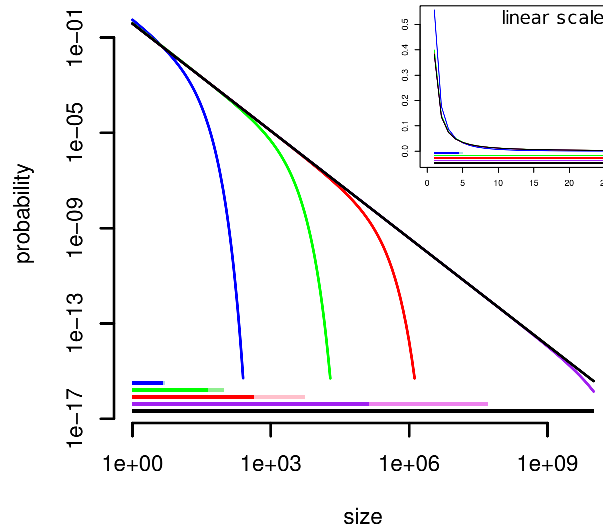


Figure 9.4: Probability distribution of sizes for trees generated with the Boltzmann method, with a parameter $x = 0.9\rho, 0.999\rho, 0.99999\rho, 0.9999999999\rho$, and ρ. The solid color bars show the range inside which the generators have a guaranteed linear complexity, which in practice extends to the whole colored range. In the main plot, both axes are in logarithmic scale, while the miniature plot is the same one in linear scale.

singular sampling. Pointing consists of differentiating the tree specification, akin to Huet's 'zipper' [Hue97]. The result of this transformation is a specification of the list class. For our implementation, we chose the second approach, consisting in taking ρ as the value of x. In the case of singular sampling, the mean size of the generated structures is infinite. We will never generate an infinite object, but there is a non-trivial probability of generating objects of sizes that we cannot handle. The solution to this problem is simple and consists in aborting the generating process as soon as we pass the upper bound of our target size.

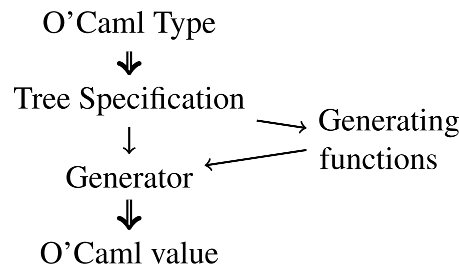
The only remaining problem is to calculate the value of ρ (for trees) or of x given a target size n (for lists and finite classes). The details of this calculation out of scope for this paper; it uses the Newton algorithm of [PSS08] that evaluates the generating function of a given structure on a value x together with a dichotomy heuristic.

The value ρ is an algebraic real number for which we calculate a numeric approximation. The sizes up to which the generation is efficient depends on the precision of this approximation. Adding one digit allows to reach trees of one more order of magnitude. As an illustration, the probability plot given in figure 9.4 shows the probability of producing a tree of size n as a function of n, with different values of x. It is quasi-impossible to obtain a tree of size one hundred with a precision of $1/10$ for ρ, whereas it is likely to produce a tree of size ten million when ρ is approximated with a precision of $1/10^{10}$.

Our implementation uses double precision floating point numbers, which allows the calculation of ρ to a precision of at least $1/10^{12}$. Thus, if tree size corresponds to bytes, the biggest trees we are expecting to generate will be of some gigabytes, the size of the main memory of a current desktop computer.

9.4 Application to O'Caml

The following schema summarizes the steps necessary to get from an algebraic data type to a random value respecting it.



The simple arrows represent transformations that were explained in the previous section, and this section deals with the double arrows. First we explain how to relate algebraic data types to tree specifications. Then, how to create a generator for O'Caml values rather than abstract trees.

9.4.1 O'Caml types to tree specifications

The size function has to be defined over O'Caml values. One may want to count the number of bytes in the memory representation of the value, or, in the contrary, only count "units" of information carried by the value. We made the choice of an intermediate path, knowing that it is very easy to change the implementation to adapt to a different one.

Base types (unit, int, float, bool, string) will have size one. A constructor (including the empty list, list constructor, tuple and record) add one to the size, so `[1]` and `(True,False)` are both of size 3.

We need a type to express the tree specifications, as presented in last section:

```
1  type grammar = (string * def) list
2  and  def     = Sum of def list
3               | Prod of def list
4               | Seq of def
5               | Ref of string
6               | Z
```

Listing 9.1: Type of tree specifications

where the `strings` correspond to type names.

A run-time representation of O'Caml types is needed. Many are available and we chose DynLib , which is itself based on Type-conv .

The translation itself is quite natural, as O'Caml lists correspond to sequences, constructions with a fixed number of arguments to products and variants to sums. Formally, for a type t in a set D of mutually recursive type definitions, we define the transformation as the recursive

function $\mathcal{T}$ such as

$$
\begin{aligned}
\mathcal{T}(\text{int}, \text{bool}, \ldots) &= \text{Z} \\
\mathcal{T}(t \in D) &= \text{Ref}(``t\,") \\
\mathcal{T}(t \notin D) &= \text{Z} \\
\mathcal{T}(t \text{ list }) &= \text{Prod } [\text{Z}; \\
&\qquad\qquad \text{Seq } (\text{Prod } [\text{Z}; \mathcal{T}(t)\,])] \\
\mathcal{T}(t_1 * \ldots * t_n) &= \text{Prod } [\text{Z}; \mathcal{T}(t_1);\ \ldots;\ \mathcal{T}(t_n)] \\
\mathcal{T}(\{n_1 = t_1; \ldots; n_n = t_n\}) &= \text{Prod } [\text{Z}; \mathcal{T}(t_1);\ \ldots;\ \mathcal{T}(t_n)] \\
\mathcal{T}(\text{A1 } \mathbf{of}\ t_1 \mid \ldots \mid \text{An } \mathbf{of}\ t_n) &= \text{Sum } [\text{Prod } [\text{Z}; \mathcal{T}(t_1)\,]; \ldots; \\
&\qquad\qquad \text{Prod } [\text{Z}; \mathcal{T}(t_n)\,]]
\end{aligned}
$$

9.4.2 Tree sampler to O'Caml value sampler

Once we obtain a tree specification corresponding to the types, we can apply the methods of section 9.3 to obtain a tree sampler. It would then suffice to transform these trees back to O'Caml values and be done. It is however more efficient and as simple to create directly a generator for O'Caml values and use the tree specification only to calculate the generating function values, necessary for the generation.

We need to decide how to generate values for the leaves of our trees, which are either primitive types or types external to the generator. All testing frameworks include generators for primitive types and one generator will not fit all use cases, it is thus important to use a callback mechanism to allow for some flexibility. Our prototype provides some simple generators and allows for the user to name a generator for each type.

We will once again need the definition of the O'Caml type we are using, and this time we will translate it to a generator. Let's see in detail the algorithm to generate a value given its type:

- unit, int, float, bool and string: call the corresponding generator.
- $t \in D$: call the generator created by us.
- $t \notin D$: call the generator provided by the user.
- $(t_1 * \ldots * t_n)$: generate independently each element of the tuple.
- $\{n_1 = t_1; \ldots; n_n = t_n\}$: generate independently each element of the record.
- t list : first draw the list length using a geometric law with the parameter calculated using the generating function corresponding to t. Then generate independently each element of the list.
- A1 **of** t_1 | ... | An **of** t_n: choose randomly which constructor to use with the probabilities of drawing each one being calculated using the corresponding generating functions. Then generate the value of the constructor's argument.

9.4.3 Modifying the generation result

While we argue that it is important to know the exact probability distribution of the generator, we are aware that the only uniform distribution is not sufficient. For this reason, we propose the user a simple means of modifying the probability distribution, while still being able to easily calculate it. Each type and its variant constructor can be associated to a real number that will be its coefficient.

Adapting our framework to deal with coefficients is fairly straightforward. The generation algorithms are not modified, only the generating functions change. The new values affect the probabilities of choosing a constructor in a variant type or the length of a list. The default coefficient value of 1 changes nothing to the distribution, increasing it will make the corresponding type or constructor appear more often.

Tuning the coefficients is still a manual task though. Given a function on trees (e.g. depth, root degree, number of nodes at a given level), we would like the framework to calculate the coefficients that will make the generator produce trees with a given mean value for the function. The extension of the underlying theory to allow for this is work in progress.

9.5 Demonstration

In this section, we give the reader a practical overview of the different applications of our random generation core. In this regard, we developed a library handling specification-based testing and benchmarking based on it.

The implementation is available as a CamlP4[3] syntax extension along with a core library to be linked with the executable. To parse type definitions and obtain a run-time representation from them, we respectively rely on and .

Downloading, building and launching a toplevel The tester must have at least version 3.11.0 of the Objective Caml compiler and tools. The aforementioned dependencies also have to be available. Our library can then be installed with the usual `make install` command. The package provides a top-level pre-loaded with the library and the syntax extension; let's start by loading it with `genadttop`.

An algebraic data type We need a data type to work on, let's choose a simplified representation of text documents. It shows products (tuples), sum types and mutually recursive types.

```
1  type document = string * block list
2  and block =
3    | Paragraph of text
4    | Section of (string * block list)
5  and text =
6    | Text of string * text
7    | Attribs of attribs * text
8    | Empty
9  and attribs =
10   | Bold | Italic | Underlined
```

3. The Objective Caml preprocessor included in the standard distribution

```
11  with sample
```

Listing 9.2: Our example data type

The ⌜**with** sample⌝[4] keyword is preprocessed by our CamlP4 syntax extension to extract its combinatorics grammar.

$$
\begin{aligned}
\mathcal{D} &= \mathcal{Z} \times (\mathcal{Z} \times \mathrm{Seq}(\mathcal{B})) \\
\mathcal{B} &= \mathcal{Z} \times \mathcal{T} + \mathcal{Z} \times (\mathcal{Z} \times (\mathcal{Z} \times (\mathcal{Z} \times \mathrm{Seq}(\mathcal{B})))) \\
\mathcal{T} &= \mathcal{Z} \times (\mathcal{Z} \times \mathcal{T}) + \mathcal{Z} \times (\mathcal{A} \times \mathcal{T}) + \mathcal{Z} \\
\mathcal{A} &= \mathcal{Z} + \mathcal{Z} + \mathcal{Z}
\end{aligned}
$$

A note on size In this section, we shall talk about document size, let's make this notion clear. As explained in section 9.4, each constructor in a sum type, each product type as well as each base type has size one. For example, the following value has a size of 8.

```
1  ("book", Paragraph (Text ("text", Empty)) :: [])
2  1  1          1        1       1        1      1  1
```

Listing 9.3: Size count example

The syntax extension produces the function to build a generator of documents of a given maximum size (the minimum size can be adjusted with another optional argument).

```
1  val sample_document
2    : ?min:int max:int -> (unit -> document)
```

Listing 9.4: Generated sampler signature

We also provide a syntax to define specialized samplers in which the programmer can give coefficients to constructors or types, as well as specify the samplers for external types appearing in the definition.

```
1  let my_sample_document =
2    <:sampler< document
3                coeff 10 for Paragraph
4                sample string with my_string_sampler >>
5    : ?min:int max:int -> (unit -> document)
```

Listing 9.5: Tweaked sampler

In this example, we ask for values in which the ⌜Paragraph⌝ constructor appear much more often than in a uniform random generation by assigning a coefficient of 10 to it (by default, each constructor is assigned a coefficient of 1). We also require strings to be sampled by our own function rather than by the default string sampler.

Testing We shall now demonstrate how to use these samplers in specification-based testing. We provide several functions for this task. For example, the ⌜run_test⌝ one which runs an interactive loop, testing a function with a predicate and showing the problematic inputs/outputs on demand.

4. We use the notation ⌜code⌝ for code extracts in the text

```
1  run_tests :
2     int -> (unit -> 'a) ->
3     ('a -> 'b) ->
4     ('a -> 'b -> bool) ->
5     ('a -> string) ->
6     ('b -> string) ->
```

Listing 9.6: Interactive testing function signature

Here is an example testing the validity of our pretty printing function for this document format.

```
1    run_tests
2      10 (sample_document ~min:1 ~max:10_000)
3      print_document
4      print_endline
5      to_string
6      (fun d r -> d = of_string r)
```

Listing 9.7: Launching an interactive testing

This example reveals simple errors. For example, it seems that we did not make a difference between sections containing only the empty string and empty sections in our printer:

```
Test 5/100 failed.
Show input (size 11) (y/n) ? y
("x", [
  Paragraph (Text ("y",
               Attribs (Underlined,
                         Attribs (Bold, Empty))))
])
Show output (y/_) ? y
"x"  ( p "y" ( u ( b "" ) ) )
```

Listing 9.8: Interactive testing session

But our ability to easily generate large size values can also make other classes of errors show up, like the fact that one of our function is not as tail recursive as we thought:

```
Test 1/100 raised Stack overflow.
Show input (size 1000000) (y/n) ? n
```

Listing 9.9: Interactive testing session (2)

Also, checkers also have to be written correctly to handle checking of large size values:

```
Checker failed on test 13/100 with Stack overflow.
```

Listing 9.10: Interactive testing session (3)

Observing properties As explained in section 9.2, since we use uniform random generation, we can soundly obtain statistical properties about the values of a given size. For example, it is theoretically hard to obtain an average height for a non-trivial tree structure. Even if it quite meaningless in this example, let's see how to obtain the average number of sections in a random document of size ranging between 10,000 and 20,000.

```
1  range
2    sample_document     (* generator *)
3    10_000 20_000       (* size *)
4    100                 (* 100 samples *)
5    average             (* combinator *)
6    ["height", height]  (* property *)
```

Listing 9.11: Simple statistical property computing example

This example showed how to obtain a single value by applying the ⌜average⌝ function to all the results. We provide several functions to select the inputs (range, exact size, histogram, etc.). We also provide several others to combine the results (identity, average, min_max, etc.).

The next example show how we can obtain an histogram of all documents of size between 1000 and 100,000 with 10 subdivisions with the ⌜histogram⌝ selector and obtain the average of each subset with the ⌜average⌝ combinator. The samples can be used simultaneously on several properties, here on the height and the maximum length of paragraphs.

```
1  histogram
2    sample_document
3    1000 100_000      (* sizes from 1000 to 100,000 *)
4    10                (* ten subdivisions *)
5    10                (* 10 samples per subdivision *)
6    average
7    ["height", height ;
8     "paragraph␣length", parlen ]
```

Listing 9.12: Histogram with multiple properties

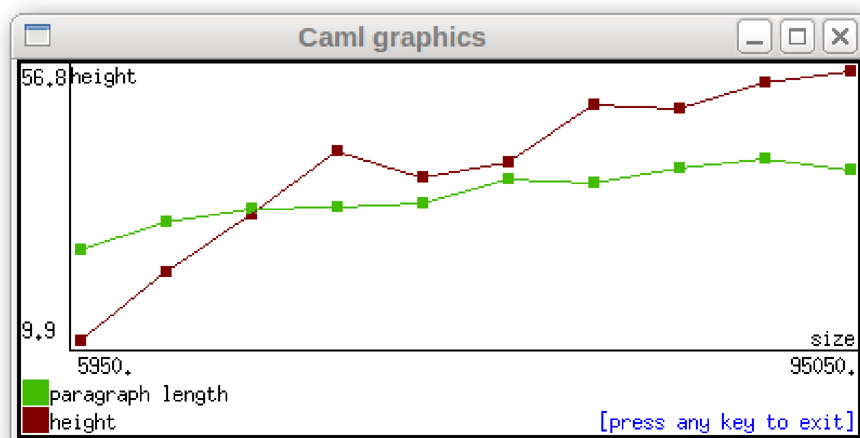


Figure 9.5: Integrated Graphics display

We provide an integrated display written with the standard ⌜Graphics⌝ module, in order to be able to easily see the results from the top-level in a single line. Figure 9.5 shows this display applied to the result of the previous example.

Benchmarking One big difficulty when writing complex programs is to figure out how the complexity of the implementation of some data structure can affect the global performance. As we can for any property, we can obtain statistical properties about complexity by running an algorithm over a number of samples.

We provide a benchmarking function taking a list of implementation and running them on the same samples. Figure 9.6 shows for example the performance of the ⌜to_string⌝ function with two different string implementations (standard strings and ropes).

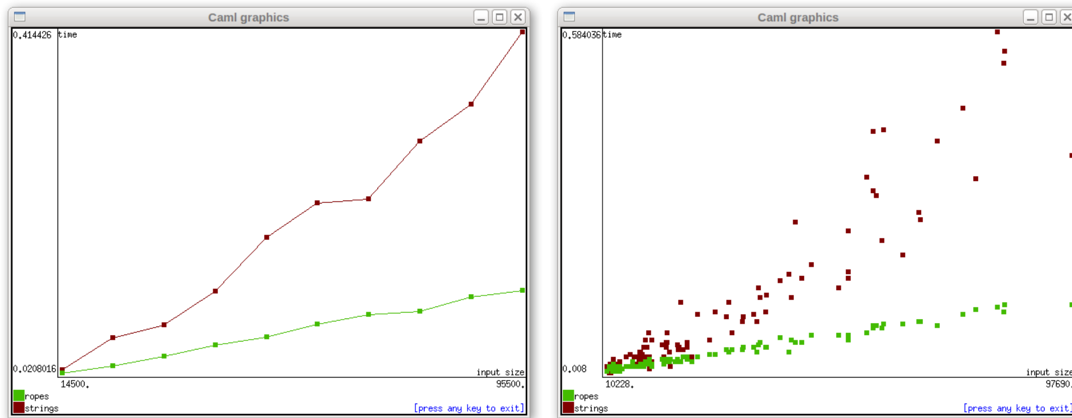


Figure 9.6: Display of benchmarking results for to_string implementations with strings and ropes. The top image has been obtained with the histogram function, the bottom one with range.

9.6 Performance

In order to demonstrate the efficiency of the Boltzmann model, we measured the time[5] needed to generate values of the `document` type with our prototype implementation. The measures were made on a recent desktop computer (Intel Core 2 2.4GHz with 3GB of RAM). Since the running time of a generation depends on the number of rejections which is random, the numbers shown below are averages over 100 runs.

We compared these timings to the recursive generation implemented in the combstruct module of the comuter algebra system Maple version 10. This module has not been designed for generating very large objects, sizes over 2000 result in a stack overflow.

size	recursive	Boltzmann	
		exact	$\varepsilon = 0.1$
100	0.5s	0.07s	-
200	1.5s	0.25s	-
500	9.4s	1.6s	-
1000	42s	8.1s	0.037s
1500	1m46s	13.5s	0.062s
10 K	*fail*	-	0.4s
100 K	*fail*	-	4.2s
1 M	*fail*	-	50s

This experiment confirms that exact-size generation of trees using the Boltzmann method is quadratic, while approximate-size generation is linear. This includes the pre-processing time that is dependant on the specification size. O'Caml types will typically produce small grammars, but our prototype is capable of dealing with grammars containing several hundred rules

5. processor time in user mode

in minutes.

The O'Caml code needed for this prototype is just a few hundred lines long. The generating function values are calculated by a seperate C library, shared with other projects, that is itself less than a thousand lines long.

9.7 Future works and other applications

Plans for this work The current implementation is still in prototype stage. We are willing to help its integration and adaptation by testing or benchmarking framework designers. If the community shows interest in this project and our library prototype, we shall be ready to develop and maintain it, but in this case we shall probably choose a better run-time type representation. Of course, this does not apply only to Objective Caml but also to other statically typed functional languages like SML or Haskell for which the adaptation of the model is straightforward.

One of the limitations is that we don't handle type parameters yet. The easy solution with the current tools would be to add an argument to the generator for each type parameter so the programmer can provide the sub-generator by hand. This means that type parameters are considered as leaves in the combinatorics specification. We would however like the combinatorics specification of each type parameter to be inlined into the main type so that the generation is uniform over the whole grammar. In practice, we want to obtain the instantiation of type parameters when building the generator. If we stay at the syntax extension level, this would require the programmer to manually explicit the instances along with some tricky type operations. A better solution could be to use the integrated run-time type representation, as proposed by [HMC07] which automatically handles instantiation but requires a modified compiler.

Another limitation is that the generated values do not contain cycles or sharing. This is due to the fact that the theory only handles trees. However, generating graphs, or at least directed acyclic ones, could be very interesting for testing purposes. Work is done on the theoretical side, but until a satisfactory solution is found, we have practical ideas for ad-hoc methods. For example, a method already experimented in other applications is to tweak the generated values.

Work in progress about programming languages Any programming language with algebraic data types is amenable to the techniques presented in this paper. Haskell is particularly interesting, since the QuickCheck specification-based test library is widely adopted by this language's community. A small library extending QuickCheck in order to automatically generate random instances of algebraic data types is almost ready, and it's functioning principles are identical to those showed here.

Other tree-like data structures can also be generated with these methods, sometimes with a more involved translation process. XML documents respecting a given RelaxNG grammar is an example we have treated [Dar08]. Possible applications include stress-testing of web services and this can be easily adapted to related languages, like CDuce or OCamlDuce.

It might even be interesting to generate the tree skeleton of more complicated data structures, in cases where there is no efficient random generation yet. In a recent work [MDBS09],

this method has been applied to generate randoms models that respect a given meta-model. The cross-references that appear in these structures are treated in an ad-hoc manner.

Work in progress about combinatorics As we just said, we cannot handle sharing and cycles in the generated values. A rigorous treatment of structures with cross-references is being worked on, but the complexity rises with the number of shared objects (e.g. variables in λ-terms). Generating DAGs or graphs in general (other than specific classes that are in bijection with some kind of trees) is out of reach for the moment.

Boltzmann sampling theory is much more wide than the part that we present here. More constructions are available, especially to take symmetries into account. These could be easily integrated to this work, but these constructions are out of the range of expressiveness of ML's type system.

We believe however that there is room for more collaboration between the fields of combinatorial structures and functional languages. The theory of species [BLL98], that treats combinatorial structures from a category theory perspective is central to such efforts, as can be seen for instance in the work on derivatives of [AAGM03].

9.8 Conclusion

We applied the recent develoment of the Bolzmann random generation model to algebraic data types by defining a translation from type definitions to combinatorics specifications. We developed a syntax extension for Objective Caml generating combinatorics grammars from type definitions, and a random generation core generating values respecting such a grammar. In the end, we extract in a fully automated way random generators from Objective Caml type definitions.

The theoretical results on these generators ensure soundness and efficiency properties that had never been reached simultaneously by existing generation techniques.

1. The generation is uniform: the generator for a given type and size gives the same probability to be produced to each possible value. In a testing context; this property ensures that no subclass will be missed because the generator is biased. Moreover, it also allows the method to be used as a sound input generator for benchmarking.
2. If we authorize a little approximation on the size, which is fine for testing purposes, the time and space complexity is linear. The method is thus able to produce very large objects.

Other generation techniques providing the first property have super-linear space and/or time complexity while techniques providing the second produce biased generators.

To demonstrate the method, we developed a small testing and benchmarking open source library for Objective Caml. We are ready and willing to maintain and extend it if the community is enthusiast. We are as well available to help testing and benchmarking frameworks integrate this method.

Cet article est tiré des actes de la conférence ECMDA09 [MDBS09].

Chapter 10

Uniform random generation of huge metamodel instances

The size and the number of models is drastically increasing, preventing organizations from fully exploiting Model Driven Engineering benefits. Regarding this problem of scalability, some approaches claim to provide mechanisms that are adapted to numerous and huge models. The problem is that those approaches cannot be validated as it is not possible to obtain numerous and huge models and then to stress test them.

In this paper, we face this problem by proposing a uniform generator of huge models. Our approach is based on the Boltzmann method, whose two main advantages are its linear complexity which makes it possible to generate huge models, and its uniformity, which guarantees that the generation has no bias.

10.1 Introduction

The size and the number of models is drastically increasing, preventing organizations from fully exploiting MDE (Model Driven Engineering) benefits [MCF03]. Today systems are already composed of hundreds of models whose size is quite often close to the thousand of model elements [Sel03]. Regarding the evolution of system complexity [Ult], one can easily observe that scalability is the most critical of today's (and tomorrow's) challenges.

Approaches that address scalability issues claim to propose adapted mechanisms that deal with numerous and huge models. However, they lack of a complete large-scale validation. Indeed, as it is very difficult to obtain numerous and huge models, it is not possible to really stress test them.

To face this problem, one possible approach is to gather numerous and huge models into open repositories [LMFW08]. The idea is to ask large organizations to populate the repositories by providing their larger models. This approach is however not really convincing because, as said by the authors themselves, "one of the main challenges was to find a good quantity of models". Indeed, only 150 models have been stored in the Moogle repository [LMFW08], which correspond to 80 thousands model elements, and is therefore not sufficient to realize

large-scale stress test.

In this paper, we propose another approach that consists in a uniform generator of huge models. Indeed, we argue that (1) the generator should be uniform [1] in order to be used to validate existing approach without introducing any bias and that (2) the generated models should be huge (millions of model elements) in order to measure the scalability of the approaches.

As we will detail in section 10.5, most of existing approaches that provide generators of models aim at generating constrained models. Their objective is to find, if possible, models that are consistent regarding a set of constraints. Those approaches are based on constraint solvers and hence have difficulties in generating huge models.

Our approach is based on the Boltzmann method [DFLS04] whose two main advantages are its linear complexity which makes it possible to generate huge models, and its uniformity, which guarantees that the generation has no bias.

This article is structured as follows. Section 10.2 presents the Boltzmann method. Section 10.3 presents our contribution that is to exploit the Boltzmann method to generate huge models. Section 10.4 then presents our realization, then section 10.5 presents works related to the problem of models generation and section 10.6 presents our conclusion.

10.2 Boltzmann random generation of trees

Our approach is based on the random sampling of combinatorial structures, within the frame of Boltzmann method, as introduced in [DFLS04]. The main feature of this method is uniform generation with linear complexity, thus allowing for generation of much larger objects than was possible before.

Most random generation methods deal with finite classes of objects, usually objects with a given size, for example binary trees of size one thousand. In the case of Boltzmann method, the notion of uniformity is extended to classes of objects for which the cardinality is infinite, like binary trees of any size. The Boltzmann method only guarantees uniformity for structures of the same size, with the constraint that there is a finite number of elements having the same size. For instance, the number of possible binary trees is infinite, but there is a finite number of binary trees for a given size.

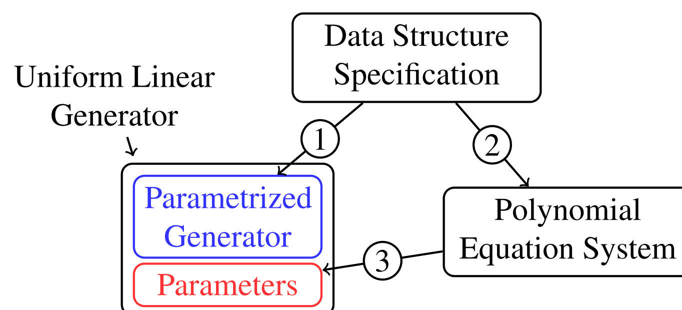


Figure 10.1: Boltzmann Method Process Overview

1. By uniform we mean that for a finite class of objects $\mathcal{C}$, any object of $\mathcal{C}$ is produced with equal probability $1/\text{card}(\mathcal{C})$.

Boltzmann method is generic and can be applied to data whose structure specifications are based on a rich set of constructors, such as disjoint union, Cartesian product, sequences, sets, cycles, etc. It relies on three steps. The first one is the transformation of a `data structure specification` (1) into a `parameterized generator`. The second and third step aim at computing the right parameters for this generator. Step two is the production of a `polynomial equations system` (2) from the `data structure specification` and step three consists in working on the `polynomial equations system` with analytical techniques (these are the domain of analytical combinatorics, described in [FS09]) in order to compute the actual parameters of the `parameterized generator`, which will make the generator uniform with a linear complexity. By these means, a generator can be automatically compiled from a data structure specification (see figure 10.1).

This section is continued with a presentation of Boltzmann generation of trees. Section 10.2.1 presents the notion of *tree specification* which will be used for specifying the structure of the trees to be generated. Section 10.2.2 describes how to automatically derive the corresponding parametrized generator. Section 10.2.3 then presents how to compute the actual parameter for making the generator uniform with a linear complexity. Finally, section 10.2.4 explains why the uniform generator has a linear complexity.

10.2.1 Tree specifications

In this paper we use the Boltzmann method to generate trees. A *tree specification* in this context will be a context-free grammar with two terminals (Z and ε) and three operators (Seq, $|$ and $*$). Z represents one instancible element (either a leaf or any node) whereas ε is the empty element. The unary operator (Seq) is used to specify sequences of an arbitrary size $k \geq 0$. The binary operators ($|$) and ($*$) are used to specify respectively union and product.

The size of a tree T, denoted by $|T|$, will be the number of Z it contains. Remember that for a grammar to be admissible by the Boltzmann method, it must only allow for a finite number of trees of a given size, therefore constructions such as $\text{Seq}(\varepsilon)$, which creates an infinity of zero-sized objects, are not allowed. Figure 10.2 shows three classical examples of tree specifications. First, a binary tree ($\mathcal{T}_1$) which is either a leaf ($\mathcal{L}_1$) or a node ($\mathcal{N}$), with leaves being of one size unit (Z) and nodes being of one size unit that aggregate two binary trees ($Z * \mathcal{T}_1 * \mathcal{T}_1$). Then, one-two tree and a general tree specifications are also given as example.

Tree type	A	corresponding grammar
Binary trees	$\mathcal{T}_1$	$= \mathcal{L}_1 \mid \mathcal{N}$
	$\mathcal{L}_1$	$= Z$
	$\mathcal{N}$	$= Z * \mathcal{T}_1 * \mathcal{T}_1$
One-two trees	$\mathcal{T}_2$	$= \mathcal{L}_2 \mid \mathcal{U} \mid \mathcal{B}$
	$\mathcal{L}_2$	$= Z$
	$\mathcal{U}$	$= Z * \mathcal{T}_2$
	$\mathcal{B}$	$= Z * \mathcal{T}_2 * \mathcal{T}_2$
General trees	$\mathcal{T}_3$	$= Z * \text{Seq}(\mathcal{T}_3)$

Figure 10.2: Classical examples of tree specifications.

10.2.2 General generator automatic construction

In this section we present the transformation that inputs a tree structure specification and returns a corresponding parameterized generator. A parameterized generator is a set of procedures that correspond to each non terminal of the tree structure specification (by convention, we name genT() the procedure that corresponds to the generation of the non terminal $\mathcal{T}$). The parameterized generator can be used to generate any tree that conforms to the input structure specification.

When there is a choice point, for union or sequence in the case of tree specifications, the generator uses its parameters to determine either which element of the union should be generated, and or the length of the sequence to generate. Each choice point must respect a particular *choice probability* in order for the global generator to be uniform. These probabilities are driven by a weight operator, noted w, that will ensure that the generation is uniform. The actual parameters of the generator are the weights of each non-terminal in the specification that, if set correctly, will guarantee the uniformity of the generation and the weight of the terminal Z that, if set correctly, will ensure the linear complexity.

The following rules are then used to build the generation procedures, in addition to each rule we give the corresponding weight operator equation for each construction:

- $A = B$: This construct means that the element A is in fact specified by B. Any generation of A is substituted by the generation of B. $w(A)$ is a generator parameter, and $w(A) = w(B)$.

- $B \mid C$: with this construction, either B or C will be generated. The weights of the elements are used here to control the probability to generate either one or the other. The probability of generating B is $w(B)/(w(B)+w(C))$, and the probability of generating C is symmetric. A pseudo-random number can be used to determine which element should be generated with respect to the given probability. The corresponding weight is $w(B \mid C) = w(B) + w(C)$.

- $\mathrm{Seq}(B)$: this construction independently generates a sequence of B. First the number k of components in the sequence is drawn, following a geometric law ($k = \mathrm{geom}(w(B)) = \lfloor \frac{\ln(random([0,1[))}{\ln(w(B))} \rfloor$), and then k elements of type B are independently generated and returned as a sequence. The weight of such a construction is $w(\mathrm{Seq}(B)) = \frac{1}{1-w(B)}$.

- $B * C$: this construction independently generates both an element B and an element C, and the weight is $w(B * C) = w(B) \cdot w(C)$.

- Z: the Z element in the tree specification corresponds to one tree size unit, which very often corresponds to one node. Wherever there is a Z in the specification, a terminal element is generated. The generation of terminals is not handled by the Boltzmann method, it therefore needs to be provided. The weight $w(Z)$ is a special parameter of the generator given by the solver (see details below).

- ε: ε is the empty element, thus nothing will be generated, and $w(\varepsilon) = 1$.

Figure 10.3 shows one generation algorithm for each specification given in figure 10.2. In this example we name Ext() the constructor of terminals which must be provided by an external source. The explicit values of the weight will be given in section 10.2.3.

For better understanding of the implementation of such generator, we detail here the construction of the binary tree's generator, the two other ones are given as illustrations.

The specification of binary trees is $\mathcal{T}_1 = \mathcal{L}_1 \mid \mathcal{N}$, $\mathcal{L}_1 = Z$, $\mathcal{N} = Z * \mathcal{T}_1 * \mathcal{T}_1$. This recursive specification states that a binary tree is either a leaf ($\mathcal{L}_1$) or a node ($\mathcal{N}$) aggregating two binary trees. The binary tree random generator (noted genT_1) will have to respect the $|$ specification; the probability to generate a leaf must be $w(Z)/(w(\mathcal{L}_1)+w(\mathcal{N}))$ (note that as $w(\mathcal{L}_1)+w(\mathcal{N}) = w(\mathcal{T}_1)$, the probability to generate a leaf is simplified as $w(Z)/w(\mathcal{T}_1)$). To generate elements with the right probability we use a pseudo-random generator for real numbers in $]0,1]$, if the value produced by the pseudo-random generator is smaller than the leaf probability ($w(Z)/w(\mathcal{T}_1)$) a leaf will be produced, otherwise a node is produced using the external binary tree node constructor that inputs two binary trees generated using recursive calls.

Binary trees:	genT_1() =	**if** random() $< w(Z)/w(\mathcal{T}_1)$
		then return genL_1() **else return** gen$N$()
	genL_1() =	**return** Ext()
	gen$N$() =	**return** Ext(GenT_1(),GenT_1())
One-two trees:	genT_2() =	r := random**;**
		if $r < w(Z)/w(\mathcal{T}_2)$ **then return** genL_2()
		elsif $r < w(\mathcal{U})/w(\mathcal{T}_2)$ **then return** gen$U$()
		else return gen$B$()
	genL_2() =	**return** Ext()
	gen$U$() =	**return** Ext(genT_2())
	gen$B$() =	**return** Ext(genT_2(),genT_2())
General trees:	genT_3() =	k := geom($w(\mathcal{T}_3)$)**;** res := []**;**
		for i **from** 1 **to** k **do** res := genT_3()::res **done;**
		return Ext(res)

Figure 10.3: The generation algorithms for the example grammars.

Remark 10.2.1. If the tree specification has more than one equation, we obtain one generator for each non-terminal, with possible calls to the other non-terminals generators. We thus need to specify one non-terminal as the "root" of the grammar, in order to provide an entry point for the generation.

The goal of the second step of the Boltzmann method is then to compute the actual parameters in order to make the generator uniform with a linear complexity.

10.2.3 Boltzmann method

In this section we present how to calculate the weights used by the generator.

Tree type	Corresponding generating functions	Weights
Binary trees	$T_1(z) = L_1(z) + N(z)$ $L_1(z) = z$ $N(z) = z \cdot T_1(z)^2$	$w(Z) = \rho = 1/2$ $w(T_1) = 1$ $w(L_1) = 1/2$ $w(N) = 1/2$
One-two trees	$T_2(z) = L_2(z) + U(z) + B(z)$ $L_2(z) = z$ $U(z) = z \cdot T_2(z)$ $B(z) = z \cdot T_2(z)^2$	$w(Z) = \rho = 1/3$ $w(T_2) = 1$ $w(L_2) = 1/3$ $w(U) = 1/3$ $w(B) = 1/3$
General trees	$T_3(z) = z \cdot \frac{1}{1-T_3(z)}$	$w(Z) = \rho = 1/4$ $w(T_3) = 1/2$

Figure 10.4: The generating functions and calculated weights of the example grammars.

Boltzmann method applies to the generation of structured objects, using the powerful tool of *generating functions*. Given a class $\mathcal{C}$ of objects, each object γ having a size denoted by $|\gamma|$, we denote by $C(z)$ its generating function, which is the series $C(z) = \sum_{\gamma \in \mathcal{C}} z^{|\gamma|} = \sum_n c_n z^n$, where c_n is the number of objects of size n in $\mathcal{C}$. In Boltzmann method each object γ is generated with probability $z^{|\gamma|}/C(z)$.

The symbolic method [FS09] provides a dictionary for translating structural constructions into operators on generating functions: concerning tree constructions, the dictionary reduces to:

$$\begin{array}{rclcrcl} & & \mathcal{Z} \rightarrow z, & & \varepsilon & \rightarrow & 1, \\ \mathcal{C} & = & \mathcal{A} \mid \mathcal{B} & \rightarrow & C(z) & = & A(z) + B(z), \\ \mathcal{C} & = & \mathcal{A} * \mathcal{B} & \rightarrow & C(z) & = & A(z) \cdot B(z), \\ \mathcal{C} & = & \mathrm{Seq}(\mathcal{A}) & \rightarrow & C(z) & = & \frac{1}{1-A(z)}. \end{array}$$

Thus in a tree specification, each line transforms into a corresponding *generating function* equation (which also corresponds to the weight relations from section 10.2.2), and a system of specifications transforms into a polynomial system of equations.

For computing weights as described in section 10.2.2, we need to solve such systems of equations for a given value x of variable z: the weight of element Z is set to x, and the weight of a non-terminal C is the value of series $C(z)$ evaluated at $z = x$. The resolution is analytically coherent for $0 \leq x \leq \rho$, where ρ is a special value, called the *singularity* of the system.

Solving polynomial systems of equations is a very complex problem in general, but systems corresponding to specifications do have a structure that can be exploited in the computations. In our implementation we use a combinatorial newton method that gives a very efficient solver [PSS08], that can also be used to calculate an approximation of the singularity ρ.

In figure 10.4, we show the generating functions for the previously introduced tree specifications and the calculated weights for each of these systems for $z = \rho$.

In each case, using the values of these functions at $z = \rho$, the Boltzmann algorithms of 10.2.2 derive a linear time generator with the property of *uniformity*: given a size n, two

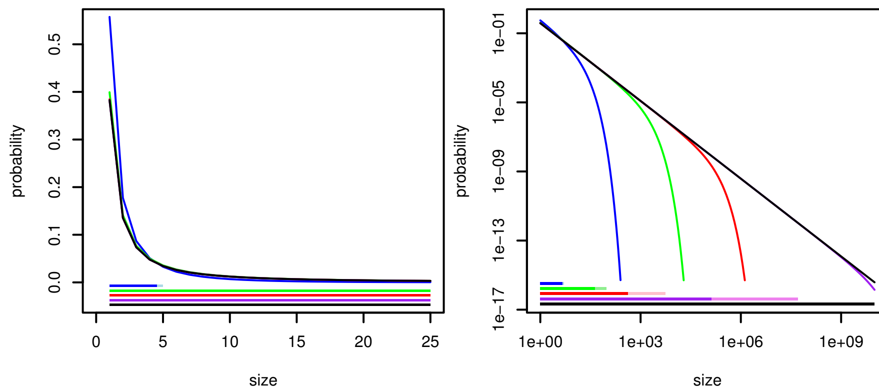


Figure 10.5: Probability distribution of sizes for trees generated with Boltzmann method, with a parameter $x = 0.9\rho, 0.999\rho, 0.99999\rho, 0.9999999999\rho$, and ρ. The solid color bars show the range inside which the generators have a guaranteed linear complexity, which in practice extends to the whole colored range. In the second plot, both axes are in logarithmic scale.

trees of that size have exactly the same probability of being generated. These generators however have the particularity that the generated trees are not all of size n, but have a random size, with a mean value depending on parameter z. We show in section 10.2.4 how to deal with this aspect, using ρ as the value for z.

10.2.4 Complexity and generation of huge trees

With Boltzmann method, the size of the generated trees is random, with a distribution that depends on the specification and a mean value that goes from 0 to infinity when parameter x goes from 0 to ρ. More precisely the probability for the result to be of size n depends on parameter x and on the singularity ρ, which is attached to the system of equations corresponding to the specification: for large n, this probability is proportional to $n^{-\frac{3}{2}} x^n \rho^{-n}$. Thus the closest is x to the value of ρ, the biggest is the probability of generating large size trees.

As an illustration, in figure 10.5 we plot the probability of producing a tree of size n in function of n, with different values of x: there are five different curves, corresponding to $x = 0.9\rho, x = 0.999\rho, x = 0.99999\rho, x = 0.9999999999\rho$ and $x = \rho$. In the right part, the curves are plotted with both axes in logarithmic scale, in order to show up the differences. It is quasi-impossible to obtain a tree of size one hundred with a precision of $1/10$ for x/ρ, whereas it is likely to produce a tree of size ten million when ρ is approximated with a precision of $1/10^{10}$.

Boltzmann samplers are particularly efficient if we accept some variability in the size of the generated structures: fixing a target size n and a margin of error δ, generating a structure of size belonging to $[(1-\delta)n, (1+\delta)n]$ can be completed in mean time $O(n)$ (whereas exact size average complexity can be up to quadratic).

In [DFLS04], it is showed that in the case of tree sampling, linear time complexity can be achieved by Boltzmann method by using either *pointing* or *singular sampling*. For our implementation, we chose the second approach, consisting in taking ρ as the value of x, and this leads to both issues of computing ρ and rejecting trees of non admissible size. Indeed in

the case of singular sampling, the mean size of the generated structures is infinite. We will never generate an infinite object, but there is however a non-trivial probability of generating objects of sizes that we cannot handle. The solution to this problem is simple and consists in aborting the generating process as soon as we pass the upper bound of our target size. As for the second point, the evaluation of ρ is non trivial and will not be detailed in this paper; it uses a dichotomy heuristic and the Newton algorithm of [PSS08].

10.3 Model generation based on meta model specification

We present in this section a scalable, uniform, random model generation process. This process can be used to generate very big models based on their metamodel specification. It makes use of the uniform random trees generation previously presented. We present here how to transform a metamodel specification into a tree specification and how to complete the generated trees to obtain the final instance.

10.3.1 Running example

In figure 10.6 is presented a simple MOF/ECORE [OMG06] metamodel inspired from the classic ECLIPSE EMF[Foub] Library example. This metamodel will be used throughout this paper to illustrate our approach. It contains four meta-classes: Library, Book, Volume and Compilation where Volume and Compilation inherit the Book abstract meta-class. This metamodel shows three containment relations, from Library to Book, fro Library to Writer and from Compilation to Book, and one relation from book to writer.

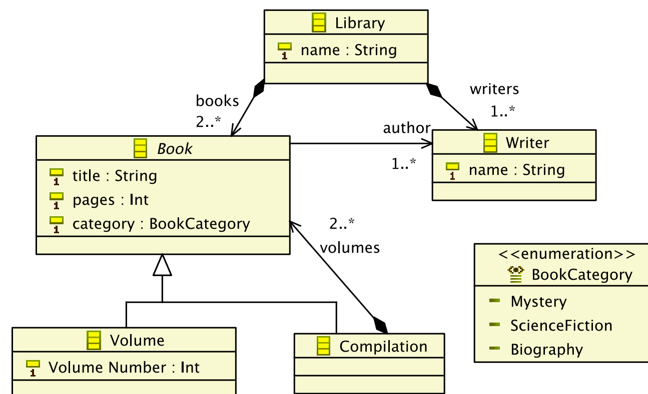


Figure 10.6: Running example metamodel diagram

10.3.2 From metamodels to tree specification

Metamodels and tree specifications are not equivalent. The metamodel language is far more expressive than tree specifications. We present here how to interpret metamodel constructions

into tree specification constructions. The transformation we propose is done in three steps where each step refines the output specification tree. Note that the model transformation we define here is not total, some elements in the metamodel will not be translated into the corresponding tree specification. Our approach only generates the core structure of the model.

Identification of base trees

The first step to build the random generator is to identify parts of the metamodel that will correspond to the randomly generated tree. A metamodel is a directed graph that is used to specify other graphs. We need to identify trees in the metamodel graph to be able to generate trees that respect the metamodel specifications. The trees used by the random generator are identified thanks to the containment relationships in the metamodel. The containment relationship offers two advantages, they allow to hierarchically generate the model and are acyclic. For each containment relationship found in the metamodel both source and target meta-classes are created in the tree specification and are equal to Z, i.e. an element with one size unit. Abstract meta-classes are created but are not equal to anything at this stage as they should not be instantiated. Finally, each of the containment relationships adds that source equals its value times the target. In the running example we identified three containment relationships. The result of the transformation on the running example is this tree specification:

$$\begin{aligned} Library &= Z * Book * Writer \\ Book &= void \\ Writer &= Z \\ Compilation &= Z * Book \end{aligned}$$

Inheritance relations

The second step to obtain the tree specification is to handle inheritance relations. The inheritance relation is interpreted as a logical or. If meta-class B inherits A, the generation of an instance of A can be replaced by the generation of an instance of B. Therefore, are added to the specification rules for each meta-class A that has daughter meta-class B the fact that A is $A \vee B$. If a new meta-class is encountered, it is equals Z. At the end of this stage all tree specification entries must have a value, all abstract meta-classes that are not inherited must be removed from the tree specification system as they can not be instantiated.

In the running example, *Compilation* and *Volume* inherit *Book*. The resulting tree specification is:

$$\begin{aligned} Library &= Z * Book * Writer \\ Book &= Volume \mid Compilation \\ Writer &= Z \\ Volume &= Z \\ Compilation &= Z * Book \end{aligned}$$

Cardinalities

The third step makes sure the cardinalities constraints are respected. To respect lower and upper bound cardinalities the target is multiplied as many times as requested in the tree specification, if the cardinality of a relation A to B is $x..y$ then A is $\vee_{i=x}^{y} B^i$. If the upper-bound is * we use the sequence concept, where $Seq(B)$ denotes an arbitrary long B sequence, note that a sequence may be empty. If the lower bound is 0, the empty element ε is used.

If we apply the cardinality constraints to our tree specification, we obtain :

$$\begin{aligned} Library &= Z * Book^2 * Seq(Book) * Writer * Seq(Writer) \\ Book &= Volume \mid Compilation \\ Writer &= Z \\ Volume &= Z \\ Compilation &= Z * Book^2 * Seq(Book) \end{aligned}$$

Unadapted metamodels

At the end of the transformation, all meta-classes in the metamodel should have a value in the tree specification. If it is not the case, this meta-classes are not accessible, meaning that they can not be randomly generated in a global uniform random generation process, i.e. the metamodel is not suited for our random generation process.

At the end of the transformation, all meta-classes should be linked directly or indirectly with the root of the metamodel. If it is not the case, the given metamodel has more than one root, our random generation process can be used with any of this roots, but only a subpart of the metamodel will be generated.

10.3.3 Model final structure generation

The tree specification corresponding to the metamodel is used to generate the skeleton of the instance generation as described in section 10.2, however the Boltzmann tree random generator only generates a model core. It needs a mechanism to generate basic relations that are not containments in order to generate instances of the metamodel. To our knowledge, there is no methodology to uniformly generate such structures and keep the overall generation process random (Boltzmann model does not apply well to graphs). Therefore, any generation process for the basic relations that respects the metamodel specification can be used to complete the skeleton. For instance, in [EKT09] is described as stage two and three such a process. In our implementation on UML Class models we implemented a generator that strictly satisfies the lower bound constraints.

10.4 Validation

In this section we present our implementation of the generator, as well as particularities of the random sampling that were verified in particle uses when generating UML 2.2 Class Models.

Figure 10.7: UML 2.2 based tree specification for class models

$$
\begin{aligned}
model &= package \\
package &= 0,01Z * Seq(packageableElement) \\
packageableElement &= package \mid class \mid association \\
class &= Z * Seq(property) * Seq(operation) * Seq(generalization) \\
generalization &= Z \\
property &= 3Z * (valueSpecification \mid \varepsilon) \\
association &= Z \\
valueSpecification &= literalBoolean \mid literalNull \mid literalInteger \mid literalString \\
literalBoolean &= Z \\
literalNull &= Z \\
literalInteger &= 2Z \\
literalString &= Z \\
operation &= 2Z * Seq(parameter) \\
parameter &= 3Z * (valueSpecification \mid \varepsilon)
\end{aligned}
$$

10.4.1 Implementation

We present in this section the application of the generation process to UML class models. From the official UML 2.2 specification we extracted a simplified class metamodel. The figure 10.7 presents the tree specification corresponding to this metamodel. Note that the generation processed has been tweaked, the probability to generate packages was reduced and the probability to generate operations, properties, literal integers and parameters augmented. This manipulation is later detailed in this section.

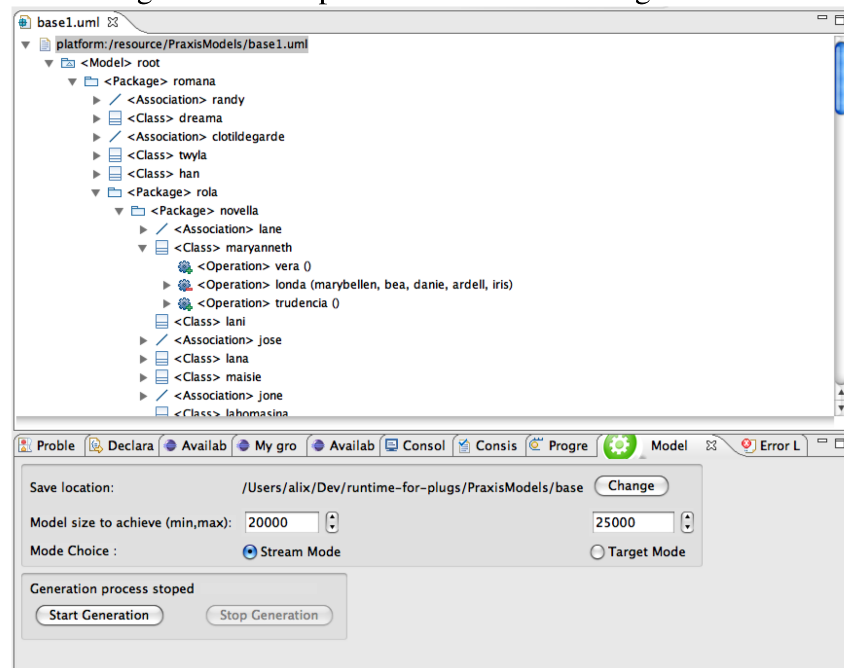
We implemented the generator as an Eclipse Plugin which is available online [2]. The plugin can generate UML files containing the class model from a graphical interface shown in figure 10.8.

We implemented a value generator for the names, literal values, visibility kinds and direction kind in order to produce a valid model. The properties value generation we implemented is constrained in order to only produce valid models. We also implemented a generator for generalization and references that randomly chose a valid target in the generated elements. However, the generation of constrained values is not in the scope of this paper.

Even with a linear complexity in random calls, the actual generation process may be long for big models. Indeed, the meta-classes instantiations and properties value generation is time consuming, in order to avoid the generation of unused elements we propose to use a simulation. When a model of a particular size is to be generated, the current random seed is saved, the algorithm to generate a random instance is run without any instantiation, and if the simu-

2. see http://meta.lip6.fr for more details.

Figure 10.8: snapshot of the class model generator



lation is successful (the size of the model is correct), the seed is reused to generate the actual model, otherwise the simulation is run again. This optimization does not change the theoretical complexity of the sampling but allows to gain a huge amount of time. In our implementation on UML Class models, there is a factor higher than 1000 between the simulation of a valid generation and the same calculation with all meta-classes instantiations.

We present in figure 10.4.1 the performances of our prototype. This chart shows that the time to obtain a seed that will produce a valid model (valid size) is very short and its average is linear. However our implementation based on EMF [Foub] is quite slow and can not reasonably produce models above a size of 250 000 model elements due to a huge memory consumption. Therefore we can not provide valid building times for the size 500 000 and one million. It is important to note that the time we provide for obtaining a valid seed is an average for one hundred runs. As the complexity is an mean time complexity, the actual time spent to find a valid seed can significantly vary.

10.4.2 Generating instances of a particular size

The presented theory states that the mean time to generate a model of a particular size, within a reasonable margin, is linear. The probability to obtain a model of the right size allows us to randomly generate models and only keep the ones with a valid size. As the complexity to generate one model of the wanted size is linear, the complexity to generate a fixed size sampling of uniform models with a size in $[n(1-\delta), n(1+\delta)]$ has a linear complexity too.

It has to be noted, however, that the Boltzmann sampler is very sensible to the value of its parameter, particularly as it approaches its maximal value ρ. Our method depends on taking

Model size (10% margin)	Average simulation time	Building time
100	6.50 ms	44.6 ms
1 000	11.2 ms	154 ms
10 000	91.1 ms	1.61 s
50 000	0.501 s	9.87 s
100 000	0.934 s	26.0 s
200 000	1.79 s	52.8 s
250 000	2.48 s	63.2 s
500 000	4.32 s	not applicable
1 000 000	8.86 s	not applicable

Figure 10.9: Prototype's performance chart ran on a MacBook Air

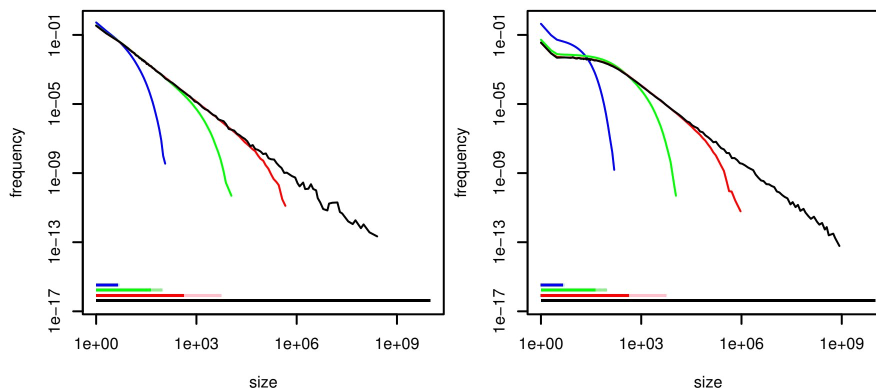


Figure 10.10: Distribution of sizes of generated class models by a Boltzmann sampler with a parameter of 0.9ρ, 0.999ρ, 0.99999ρ and ρ (ρ calculated with a 10^{-15} precision). The left plot corresponds to a grammar without coefficients, while in the right the exact grammar of figure 10.7 is used. Note that both axes are in logarithmic scale.

a parameter equal to ρ, but as ρ can be any real number between 0 and 1, it is not possible to calculate it exactly in most cases and we will use an approximation. The effect of this, illustrated in figure 10.10, is a "ceiling" in the maximal size attainable by the generator. It is therefore important to make a very precise approximation of ρ to be able to generate very large objects. In figure 10.10 is represented the distribution of class model sizes generated using different approximations of ρ, it appears that the proportion of big models is affected by the accuracy of ρ's calculation. For instance no model of a size greater than five hundred thousand model elements could be generated with a value of ρ correct up to the seventh digit, but with a precision of fifteen digits, ten million model element is possible to reach in linear time.

10.4.3 Influencing generation output

It is in our opinion very important to be able to characterize the probability distribution of the generated structures, as it allows us to control a possible bias. However, the uniform distribution provided by the Boltzmann samplers may not be the best fit to this needs. We might find, for example, that the number of operations in classes in the random class models is not sufficient. We thus need to add ponderations in our specification, in order to influence the frequency of appearance of the different elements, in a way that allows us to calculate the resulting bias.

The extension of the theory of Boltzmann sampling in this direction is work in progress, but there are some elements that we can already use. We allow the definition, for each non-terminal, of a coefficient with a default value of 1 that will influence the frequency of the corresponding element. The influence of the coefficient values to the frequencies is not trivial, as the different frequencies depend on each other, so the good choice of coefficients is done for the moment via trial and error. It is possible to calculate the frequencies given the values of the coefficients, and the complexity of the generation process is not affected. In figure 10.7 where the tree specification of simplified UML class diagrams is given, the probabilities have been modified, the probability to generate packages was reduced and the probability to generate operations, properties, literal integers and parameters augmented in order to obtain more *realistic* class models.

10.5 Related works

Alloy which is a lightweight specification language based on first-order relational logic [Jac06] can be used to generated models. Indeed, its main principle is to compute all models of a fixed size and that correspond to a particular specification. Then Alloy is able of extracting form this set of models the ones that are consistent regarding to a set of specified constraints. Alloy is based on a SAT solver (the SAT problem belongs to the NP-Hard class) and therefore is not able to produce huge models.

In [EKT09], is presented an algorithm that can generate instances of metamodels. It is based on a transformation of the metamodel structure into a set of graph specification rules. This set of rules is able to generate any skeleton of metamodel instances, and can be coupled with constraint rules in order to respect specific needs. The random process resides in the random election of generation rules. The outputted models can be biased as the choosing of the rule is constrained by the graph specification rule application formalism. Plus, this approach may not scale, the applying of each graph rule has an exponential complexity as it needs to find the existence of a subgraph in the already generated graph which limits the efficiency of this tool (the general problem belongs to the NP-Hard class).

In [BFSBT06], a formalism is presented to generate random constrained models. The approach consists in using mutations to derive, from a given instance, random other alike instances. The approach is effective and can handle very huge models since the mutation process is very effective. However, this approach is biased by definition, it needs to input one instance of the model to generate others, therefore the outputted models will have a lot of similarities.

10.6 Conclusion

In this paper we presented an adaptation of the Boltzmann random sampling theory to metamodel instance generation. The resulting generator has three interesting particularities.
First it is scalable, the complexity of the generating process is linear with the size of the generated structures. And this size is controllable.
Then it outputs uniform samplings for a given size, the probability for any structure of size n to be generated is the same.
And finally, it allows to experimentally change the form of outputted models to meet with specific requirements.

However, metamodels usually come with a set of constraints to precise the specification of its instances, in this paper we did not describe how to generate values for the properties of these instances, however our implementation on class models successfully took this challenge in consideration. In the particular purpose of generating models that satisfy important model constraints, the property generation must be carefully controlled, and possibly a random generation process may not be adapted. Further research in this direction must be done in order to exploit the high performances of the random generation of metamodel instances to constrained models.

Cet article a été mis en ligne dans le dépôt arXiv [Dar08].

Chapter 11

Random XML sampling the Boltzmann way

In this article we present the prototype of a framework capable of producing, with linear complexity, uniformly random XML documents with respect to a given RELAX NG grammar. The generation relies on powerful combinatorial methods together with numerical and symbolic resolution of polynomial systems.

11.1 Introduction

The Extensible Markup Language (XML) is extensively used today, either to encode documents (like in XHTML) or to serialize structured data. The XML standard [W3C08] only defines some basic syntax rules followed by well-formed documents. However applications often define a set of higher order syntactic [1] rules that an XML document must respect to be considered as valid for the given application. A set of such rules is called a schema and is defined in one of several languages, like DTD, XML Schema, RELAX NG and others. We chose to deal with RELAX NG for its simplicity and solid theoretical basis.

Our work is based on the observation that RELAX NG [OAS01] has essentially the same expressive power as the specifications of combinatorial structures using only union, product, sequence and allowing for recursive definitions. This means that valid XML documents are just trees and can be efficiently generated using a Boltzmann sampler [DFLS04].

In Boltzmann sampling we can easily derive a generator from the description of a combinatorial class. This generator has the following characteristics:

- it needs a precalculation that must be done once and involves finding a particular solution of a polynomial system,
- the sampling itself involves only basic mathematic operations and has a linear complexity in the size of the generated object and
- the size of the generated object is a random variable following a power law distribution.

We go from a RELAX NG grammar to a random XML document in three steps:

1. often called semantic

1. translating the grammar to a system of equations,
2. solving the system of equations and
3. sampling XML documents.

The first two steps need to be executed only once for a given grammar and their cost is only dependant on the complexity of the grammar. The final step has a complexity that is linear in the size of the result.

The current prototype includes a ruby program executing the first and last step and a maple program dealing with the second step. Future versions could consist of a single program not depending on a computer algebra system.

Even in its current form, our framework is capable of generating documents for all RELAX NG grammars that we were able to find, include XHTML, MathML, SVG, DocBook, OpenDocument and RELAX NG itself.

11.2 Translating the RELAX NG

We parse the RELAX NG document to get a combinatorial description of the grammar in the form of an Abstract Syntax Tree (AST). Every RELAX NG element is matched to a combinatorial construction, for example `choice` to union, `group` to product, `oneOrMore` to sequence etc.

The mapping from RELAX NG to combinatorial constructions is not unique, we thus have to make a few arbitrary decisions. First of all, we must decide of a way to count the size of an XML document, while satisfying the constraint that there must be a finite number of XML documents for a given size. Our choice is to count the number of elements plus the number of attributes.

Some facts on the AST: at the root we have the definition of `start` which is our entry point to the grammar and the definition of every `element` of the grammar. We can thus represent the AST as a forest, with every root being the definition of an `element`. `ref` elements found in different parts of the AST point to one of these definitions, which can lead us to see the AST also as a graph.

Note on `data`

Our framework concentrates on the tree-form structure of the XML document and treats `data` elements as simple leaves. These elements however represent data respecting arbitrarily complex datatypes which cannot be reasonably treated by a generic framework. We thus provide the possibility to call an arbitrary (written in Ruby since performed at step 3) function when the generator needs a value of a `data` element.

Note on uniformity

One of the main advantages of using Boltzmann sampling is that it guarantees the uniformity of the distribution inside each size class. The simplifications made in our prototype break this uniformity in two ways:

- sets of attributes are considered as sequences, so `<a x="foo" y="bar"/>` and `<a y="bar" x="foo"/>` count as two different XML documents, while they are the same (but two attributes of the same element will always have different names) and
- the `interleave` element, which calls for the interleaving of it's arguments, is not taken into account, so all the interleavings count as a single element.

11.3 Solving the system of equations

The AST is fed to a Maple program that solves the set of equations defining the grammar and provides us with the constants needed for sampling. The calculation time needed for this operation depends heavily on the size and structure of the grammar. The user can use this Maple program as a black box, as it is not interactive. But for the purpose of explanation we now describe it a little more precisely.

A RELAX NG grammar defines, in our point of view, a (combinatorial) family of trees. As for every other combinatorial class, a generating function $C(x) = \sum_{n=0}^{\infty} t_n x^n$ is assocatied to this family, with t_n being the number of different trees of size n. An important parameter is the radius of convergence ρ of the generating function $C(x)$. The Boltzmann sampler uses the value of $C(x)$ and some related generating functions for a given parameter x. In the case of trees, ρ is an excellent choice for x [DFLS04].

The AST of the grammar can be directly translated into an algebraic system (that can itself be simplified to a polynomial one) defining the generating function $C(x)$. To be able to sample we need to solve two problems: evaluate the radius of convergence of the system and find its only solution that corresponds to C.

A major difficulty comes from the fact that simply solving the system for a given x gives us a set of solutions with no simple way of finding the good one. Thankfully, a combinatorial interpretation of Newton's method [PSS08] gives us a very efficient and guaranteed numerical algorithm for evaluating $C(x)$.

As for the radius of convergence, for the time being, using a dichotomy approach and Newton algorithm we obtain an estimation. We can also (see section 11.5) take advantage of the particularities of the generating functions arising from grammars defining trees, in which the radius of convergence can be automatically calculated, as explained in [DFLS04].

To experiment the solving algorithm, we tried it on different RELAX NG grammars (found for most of them on the internet). Table 11.1 shows some measures on the grammars related to their complexity, together with the time needed to evaluate the corresponding generating functions. The size of the corresponding polynomial systems is also mentioned, showing that solving them with a symbolic approach is a challenge. Here is a precise description of the column headers:

sing. Lower bound on the singularity of the system of equations.
file. Size of the file containing the grammar in Simple RELAX NG.
 # el. Number of `element` definitions.
s.c.c. Size of the biggest strongly connected component of the grammar.
newton Time needed to evaluate the values of the generating functions using the newton algorithm.
 # eq. Number of equations defining the polynomial system.

grammar	file	sing.	# el.	s.c.c.	newton	# eq.	# mon.	# sol.
ternary trees	1024	0.52912	2	1	0.260s	3	10	3
RSS	9.5K	0.44721	10	1	0.320s	16	79	2
PNML	23K	0.22526	22	1	0.322s	36	193	4
ILP 1	21K	0.23696	20	6	0.416s	27	211	9
ILP 6	99K	0.13951	51	31	0.828s	72	948	6
RELAX NG	124K	0.04127	33	18	0.696s	114	3725	32
XSLT	168K	0.05283	40	17	0.469s	122	1503	10
XHTML	289K	0.02456	47	32	0.932s	134	2077	26
XML Schema	237K	0.08615	59	9	0.528s	188	143465	
XHTML basic	284K	0.03039	53	38	1.080s	96	2073	13
XHTML strict	1.2M	0.01991	80	58	2.414s	151	6445	32
XHTML	1.5M	0.01609	93	66	3.798s	172	8449	56
SVG tiny	1.6M	0.03542	49	7	0.371s	101	4390	34
SVG full	6.3M	0.01834	118	27	0.718s	232	13831	
MathML	2.2M	0.00318	182	48	2.432s	265	265159	18
OpenDocument	2.8M	0.01757	500	101	6.544s	814	8890517	
DocBook	11M	0.01627	407	295	143.411s	977	183051	

Table 11.1: Evaluating the generating functions of different RELAX NG grammars. Measures made on a 3.2GHz Xeon with 6GB of RAM, using Maple 10 in a 64bit environment.

 # mon. Number of monomials in the polynomial system.

 # sol. Number of solutions of the zero-dimensional system, given a parameter smaller than the singularity.

These results show that with the numerical method we were able to deal with all the grammars in a very reasonable time.

11.4 Generating XML documents

To generate the XML documents the Boltzmann way, we explore the AST, starting from the `start` element of the grammar and treating the different elements the following way:

- for a union we choose randomly between the two siblings,
- for a product we take both children,
- for a sequence we take the child a random number of times,
- for a `ref` we continue at the corresponding `element`,
- for a leaf produce the corresponding value.

Thus the cost to produce an XML document is proportional to the size of the document and the constant factor depends on the grammar. That size however is random and actually follows a power law distribution. This means that we generate a very large number of very small documents and from time to time a huge document, eventually bigger than we can handle.

For this reason we include a mechanism for rejecting documents whose size falls outside a given window. The cost of the rejection can be precisely estimated [DFLS04]: generating a document of size $n(1\pm\varepsilon)$, for a fixed tolerance ε, costs $O(n)$, while generating a document of size exactly n costs $O(n^2)$ (costs are average, in the worst case the generator never returns a valid document, but this happens with probability zero).

11.5 Related work and work in progress

The current state of our work shows what can be achieved by applying the Boltzmann sampling techniques in the problem of sampling random XML documents. We are in the process of completing and extending our framework in order to create tools useful as-is in a large number of use cases.

Efficient and guaranteed calculation of the radius of convergence. As explained in [DFLS04] we can calculate the position of the singularity of the generating functions arising from the combinatorial description of tree types by identifying the smallest real positive number for which the determinant of the Jacobian matrix of the system defining the generating functions is equal to 1. For this method to be applied we must make sure that our specification is irreducible and non-lattice, or modify it to be. The calculation of the determinant of the Jacobian matrix of a very big polynomial system is however not trivial.

Efficient implementation and integration of the framework. The current implementation of our framework is a prototype with a design targeting flexibility and ease of development. Consequently it lacks in efficiency and ease of deployment, which will be the goals of future versions.

Conserve the entropy, guarantee the precision. By implementing the non-deterministic parts of the samplers using bitwise comparisons, we can make sure not to use more entropy bits than necessary. At the same time we can check that the precision used for calculating the constants is sufficient, and if not, get more precision from the solver.

Create default samplers for XML Schema Datatypes. The RELAX NG language allows the use of arbitrary datatypes through the `data` element. However, the XML Schema Datatypes [W3C04] are frequently used. It will thus be useful to create samplers for each of the datatypes contained in this library, while preserving the possibility for the users to overload them.

Progressive serialisation of the generated document. To be able to sample documents of millions of elements, or more, we can no longer keep the whole document in memory. The solution is to write the data on disk as soon as we know its position in the resulting document. This should be trivial for grammars not containing `interleave` constructions and could be reasonably treated even in that case.

Assure uniformity in all cases. The bias introduced in the sampling by the simplified treatment of the `interleave` element and the attributes could be dealt with, if necessary. The set construction is already treated in the Boltzmann sampling, but leads to systems that are no longer polynomial. The treatment of the interleave construction is work in progress.

11.5.1 Random sampling of recursive data types

There are other possible applications of random sampling of trees. We have a prototype for an extension to QuickCheck [CH00], a tool for automatically testing programs, that randomly generates test cases. Quickcheck includes samplers for basic data types, but the samplers for trees and other recursive data types have to be provided by the user. Our framework easily adapts for the automatic generation of these samplers.

In addition to this Haskell prototype, we made (thanks to the participation of Benjamin Canou) an O'Caml prototype doing the same thing: given the definition of a recursive data type, we provide the user with a function producing random instances of this type.

Conclusion et perspectives

Les travaux présentés dans cette thèse appliquent les techniques de pointe de la combinatoire analytique à des problèmes issus de domaines qui n'y étaient pas encore exposés. Le double objectif est d'augmenter la visibilité de ces techniques qui méritent d'être plus largement connues, mais aussi d'apporter de nouvelles sources de motivation à la combinatoire.

Dans un premier temps, nous avons étudié des propriétés caractéristiques de certaines familles de graphes aléatoires. Notre résultat principal est l'estimation du degré et de la distance dans les k-arbres et il s'appuie sur une bijection avec une famille simple d'arbres.

Nous n'avons pas présenté dans ce mémoire l'étude de la classe des k-arbres croissants. Comme il s'agit d'une sous-classe la bijection s'applique directement mais produit un système différentiel au niveau des séries génératrices. L'analyse de ces systèmes, que nous avons menée récemment [DHBS10], s'apparente à l'étude des paramètres dans les arbres croissants.

Notre bijection peut aussi s'étendre à d'autres familles de graphes qui ont une décomposition récursive, en particulier les graphes cordaux qui constituent une classe importante pour l'algorithmique des graphes. Cette classe étant plus complexe que celle des k-arbres, sa représentation arborescente est, elle aussi, plus complexe : la spécification prend la forme d'un système infini. Pour étendre notre analyse à ces graphes il faut d'abord borner la taille de ce système, soit par une constante, ce qui donne une analyse très directe, soit par une fonction de la taille, ce qui donne un modèle plus réaliste mais plus difficile à analyser et pour lequel il faudra probablement développer des nouvelles méthodes.

Le cas des structures non-étiquetées serait également intéressant à étudier. Il faudrait d'abord adapter la bijection pour prendre en compte les symétries et ensuite analyser les systèmes complexes résultants.

Dans un second temps nous avons développé une approche générique pour la génération aléatoire de structures arborescentes, qui apparaissent dans des contextes différents de l'informatique : les types de données algébriques, les méta-modèles et les documents XML.

Par la suite, avant de considérer de nouveaux domaines applicatifs, il serait intéressant de consolider la validité de nos prototypes en les appliquant à des problèmes grandeur nature, ainsi que de mieux valoriser notre méthode en l'intégrant dans des outils largement diffusés.

Le travail de génération d'arbres s'inscrit dans un cadre beaucoup plus large de génération automatique des structures aléatoires, dans le but d'avoir des outils génériques et faciles à utiliser. L'expertise développée au cours de cette thèse permet d'envisager une systématisation des simulations pour l'étude des structures aléatoires, dans le cadre de l'encyclopédie des structures

combinatoires [Enc].

Un enjeu important est celui de la génération non-uniforme. En mettant en place nos solutions *ad hoc*, nous avons constaté l'importance pour l'utilisateur de la ressemblance entre les objets générés aléatoirement et les objets réels de son domaine. L'aboutissement des travaux de la génération aléatoire sur le contrôle de biais dans la génération est donc de première importance car il permettra de proposer des outils automatiques produisant des objets qui correspondent mieux aux objets de terrain.

Bibliographie

[AAGM03] Michael ABBOTT, Thorsten ALTENKIRCH, Neil GHANI et Conor MCBRIDE. « Derivatives of Containers ». Dans : *Typed Lambda Calculi and Applications*. 2003, p. 1086. DOI : 10.1007/3-540-44904-3_2.

[ADK07] V. ARVIND, Bireswar DAS et Johannes KÖBLER. « The Space Complexity of k -Tree Isomorphism ». Dans : *Algorithms and Computation*. 2007, p. 822–833. DOI : 10.1007/978-3-540-77120-3_71.

[AHAS05] Jr. ANDRADE, Hans J. HERRMANN, Roberto F. S. ANDRADE et Luciano R. da SILVA. « Apollonian Networks : Simultaneously Scale-Free, Small World, Euclidean, Space Filling, and with Matching Graphs ». Dans : *Physical Review Letters* 94.1 (2005), p. 018702–4. DOI : 10.1103/PhysRevLett.94.018702.

[AP89] Stefan ARNBORG et Andrzej PROSKUROWSKI. « Linear time algorithms for NP-hard problems restricted to partial k-trees ». Dans : *Discrete Applied Mathematics* 23.1 (avr. 1989), p. 11–24. DOI : 10.1016/0166-218X(89)90031-0.

[AS95] Laurent ALONSO et René SCHOTT. *Random generation of trees : random generators in computer science*. Kluwer Academic Publishers, 1995. ISBN : 9780792395287.

[ASBS00] Luis A. Nunes AMARAL, Antonia SCALA, Marc BARTHÉLÉMY et H. Eugene STANLEY. « Classes of small-world networks ». Dans : *Proceedings of the National Academy of Sciences of the United States of America* 97.21 (oct. 2000), p. 11149–11152. DOI : 10.1073/pnas.200327197.

[BA99] Albert-László BARABÁSI et Réka ALBERT. « Emergence of Scaling in Random Networks ». Dans : *Science* 286.5439 (oct. 1999), p. 509–512. DOI : 10.1126/science.286.5439.509.

[BB07] Olivier BERNARDI et Nicolas BONICHON. *Catalan's intervals and realizers of triangulations*. Avr. 2007. arXiv : 0704.3731.

[BDS08] Olivier BODINI, Alexis DARRASSE et Michèle SORIA. « Distances in random Apollonian network structures ». Dans : *DMTCS Proceedings*. 20th Annual International Conference on Formal Power Series and Algebraic Combinatorics (FPSAC 2008). DMTCS, 2008, p. 307–318. URL : http://www.dmtcs.org/dmtcs-ojs/index.php/proceedings/article/view/dmAJ0127.

[BFSBT06] Erwan BROTTIER, Franck FLEUREY, Jim STEEL, Benoit BAUDRY et Yves Le TRAON. « Metamodel-based Test Generation for Model Transformations : an Algorithm and a Tool ». Dans : *Proceedings of the 17th International Symposium on Software Reliability Engineering*. IEEE Computer Society, 2006, p. 85–94. DOI : 10.1109/ISSRE.2006.27.

[BJS09] Jacob BURNIM, Sudeep JUVEKAR et Koushik SEN. « WISE : Automated test generation for worst-case complexity ». Dans : *Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, p. 463–473. DOI : 10.1109/ICSE.2009.5070545.

[Bla+02] L. Susan BLACKFORD et al. « An updated set of basic linear algebra subprograms (BLAS) ». Dans : *ACM Transactions on Mathematical Software* 28.2 (2002), p. 135–151. ISSN : 00983500. DOI : 10.1145/567806.567807.

[BLL98] François BERGERON, Gilbert LABELLE et Pierre LEROUX. *Combinatorial Species and Tree-sike Structures*. Encyclopedia of Mathematics and its Applications 67. Cambridge University Press, 1998. ISBN : 0521573238.

[Bol01] Béla BOLLOBÁS. *Random graphs*. Cambridge University Press, 2001. ISBN : 978-0521797221.

[BP68] Lowell W. BEINEKE et Raymond E. PIPPERT. « The enumeration of labelled 2-trees ». Dans : *Notices of the American Mathematical Society*. T. 15. 1968, p. 384.

[BP69] Lowell W. BEINEKE et Raymond E. PIPPERT. « The number of labeled k-dimensional trees ». Dans : *Journal of Combinatorial Theory* 6.2 (mar. 1969), p. 200–205. DOI : 10.1016/S0021-9800(69)80120-1.

[BP71] Lowell W. BEINEKE et Raymond E. PIPPERT. « Properties and characterizations of k-trees ». Dans : *Mathematika* 18 (1971), p. 141–151.

[BRV01] Albert-László BARABÁSI, Erzsébet RAVASZ et Tamás VICSEK. « Statistical Mechanics and its Applications : Deterministic scale-free networks ». Dans : *Physica A* 299.3-4 (oct. 2001), p. 559–564. DOI : 10.1016/S0378-4371(01)00369-7.

[CD09] Benjamin CANOU et Alexis DARRASSE. « Fast and sound random generation for automated testing and benchmarking in objective Caml ». Dans : *Proceedings of the 2009 ACM SIGPLAN workshop on ML*. Edinburgh, Scotland : ACM, 2009, p. 61–70. DOI : 10.1145/1596627.1596637.

[CDJ01] Brigitte CHAUVIN, Michael DRMOTA et Jean JABBOUR-HATTAB. « The Profile of Binary Search Trees ». Dans : *The Annals of Applied Probability* 11.4 (nov. 2001), p. 1042–1062. ISSN : 10505164. URL : http://www.jstor.org/stable/2699908.

[CH00] Koen CLAESSEN et John HUGHES. « QuickCheck : a lightweight tool for random testing of Haskell programs ». Dans : *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. ACM, 2000, p. 268–279. DOI : 10.1145/351240.351266.

[Cha90] N. CHANDRASEKHARAN. « Isomorphism testing of k-trees is in NC, for fixed k ». Dans : *Information Processing Letters* 34.6 (mai 1990), p. 283–287. DOI : 10.1016/0020-0190(90)90011-L.

[Cla58] A. R. CLARKE. « Account of the Observations and Calculations of the Principal Triangulation ; and of the Figure, Dimensions and mean Specific Gravity of the Earth as derived therefrom ». Dans : *Eyre and Spottiswoode, London* (1858).

[Dar08] Alexis DARRASSE. *Random XML sampling the Boltzmann way*. Juil. 2008. arXiv : 0807.0992.

[Dev84] Luc DEVROYE. « A probabilistic analysis of the height of tries and of the complexity of triesort ». Dans : *Acta Informatica* 21.3 (oct. 1984), p. 229–237. DOI : 10.1007/BF00264248.

[Dew74] Alexander Keewatin DEWDNEY. « Higher-dimensional tree structures ». Dans : *Journal of Combinatorial Theory, Series B* 17.2 (oct. 1974), p. 160–169. DOI : 10.1016/0095-8956(74)90083-5.

[DFLS04] Philippe DUCHON, Philippe FLAJOLET, Guy LOUCHARD et Gilles SCHAEFFER. « Boltzmann Samplers for the Random Generation of Combinatorial Structures ». Dans : *Combinatorics, Probability and Computing* 13.4-5 (2004), p. 577–625. DOI : 10.1017/S0963548304006315.

[DG97] Michael DRMOTA et Bernhard GITTENBERGER. « On the profile of random trees ». Dans : *Random Structures and Algorithms* 10.4 (1997), p. 421–451. DOI : 10.1002/(SICI)1098-2418(199707)10:4<421::AID-RSA2>3.0.CO;2-W.

[DGGLP06] Alain DENISE, Marie-Claude GAUDEL, Sandrine-Dominique GOURAUD, Richard LASSAIGNE et Sylvain PEYRONNET. « Uniform random sampling of traces in very large models ». Dans : *Proceedings of the 1st international workshop on Random testing*. ACM, 2006, p. 10–19. DOI : 10.1145/1145735.1145738.

[DH05a] Michael DRMOTA et Hsien-Kuei HWANG. « Bimodality and Phase Transitions in the Profile Variance of Random Binary Search Trees ». Dans : *SIAM Journal on Discrete Mathematics* 19.1 (2005), p. 19–45. DOI : 10.1137/S0895480104440134.

[DH05b] Michael DRMOTA et Hsien-Kuei HWANG. « Profiles of random trees : correlation and width of random recursive trees and binary search trees ». Dans : *Advances in Applied Probability* 37.2 (2005), p. 321–341. DOI : 10.1239/aap/1118858628.

[DHBS10] Alexis DARRASSE, Hsien-Kuei HWANG, Olivier BODINI et Michèle SORIA. « The connectivity-profile of random increasing k-trees ». Dans : *ANALCO*. 2010.

[DM05] Jonathan P. K. DOYE et Claire P. MASSEN. « Characterizing the network topology of the energy landscapes of atomic clusters ». Dans : *The Journal of Chemical Physics* 122.8 (fév. 2005), p. 084105–13. DOI : 10.1063/1.1850468.

[Drm09] Michael DRMOTA. *Random Trees*. Springer Vienna, 2009. ISBN : 978-3-211-75355-2. DOI : 10.1007/978-3-211-75357-6.

[DRT00] Alain DENISE, Olivier ROQUES et Michel TERMIER. « Random generation of words of context-free languages according to the frequencies of letters ». Dans : *Mathematics and Computer Science : Algorithms, Trees, Combinatorics and Probabilities*. Versailles-St-Quentin : Birkhäuser, 2000, p. 113–125. URL : http://books.google.fr/books?id=b9nHPJvP7xgC&pg=PA113.

[DS] Alexis DARRASSE et Michèle SORIA. *A Unifying Structural Approach to the Analysis of Parameters in k-trees*.

[DS07] Alexis DARRASSE et Michèle SORIA. « Degree distribution of random Apollonian network structures and Boltzmann sampling ». Dans : *DMTCS Proceedings*. 2007 Conference on Analysis of Algorithms, AofA 07. DMTCS, 2007, p. 313–324. URL : http://www.dmtcs.org/dmtcs-ojs/index.php/proceedings/article/view/dmAH0124.

[DS09] Alexis DARRASSE et Michèle SORIA. « Limiting distribution for distances in k-trees ». Dans : *Combinatorial Algorithms*. 20th International Workshop, IWOCA 2009. T. 5874. Lecture Notes in Computer Science. Springer-Verlag, 2009, p. 170–182. DOI : 10.1007/978-3-642-10217-2_19.

[EKT09] Karsten EHRIG, Jochen KÜSTER et Gabriele TAENTZER. « Generating instance models from meta models ». Dans : *Software and Systems Modeling* 8.4 (2009), p. 479–500. DOI : 10.1007/s10270-008-0095-y.

[Enc] *Encyclopedia of Combinatorial Structures*. URL : http://algo.inria.fr/encyclopedia/.

[ER59] Paul ERDŐS et Alfréd RÉNYI. « On Random Graphs ». Dans : *Publicationes Mathematicae* 6 (1959), p. 290–297.

[ER60] Paul ERDŐS et Alfréd RÉNYI. « The Evolution of Random Graphs ». Dans : *Publications of the Mathematical Institute of the Hungarian Academy of Sciences* 5 (1960), p. 17–61.

[Eul43] Leonhard Paul EULER. « Lettre CXL. Euler à Goldbach. Même sujet. Recherche sur le nombre des manières dont un polygone peut être partagé en triangles par des diagonales. Berlin d. 4. September 1751. » Dans : *Correspondance Mathématique et Physique de quelques célèbres géomètres du XVIIIème siècle*. Éd. par P.-H. FUSS. T. 1. Académie impériale des sciences de Saint-Pétersbourg, 1843, p. 549–552. URL : http://books.google.com/books?id=gf1OEXIQQgsC&pg=549.

[FGLL02] Tom FOWLER, Ira GESSEL, Gilbert LABELLE et Pierre LEROUX. « The Specification of 2-trees ». Dans : *Advances in Applied Mathematics* 28.2 (fév. 2002), p. 145–168. DOI : 10.1006/aama.2001.0771.

[FK08] Sebastian FISCHER et Herbert KUCHEN. « Data-flow testing of declarative programs ». Dans : *SIGPLAN Not.* 43.9 (2008), p. 201–212. DOI : 10.1145/1411203.1411233.

[FO82] Philippe FLAJOLET et Andrew M. ODLYZKO. « The average height of binary trees and other simple trees ». Dans : *Journal of Computer and System Sciences* 25.2 (1982), p. 171–213. DOI : 10.1016/0022-0000(82)90004-6.

[Foa71] Dominique FOATA. « Enumerating k-trees ». Dans : *Discrete Mathematics* 1.2 (sept. 1971), p. 181–186. DOI : 10.1016/0012-365X(71)90023-9.

[Foua] Free Software FOUNDATION. *GNU lightning*. URL : http://www.gnu.org/software/lightning/.

[Foub] The Eclipse FOUNDATION. *EMF (Eclipse Modeling Framework)*. URL : http://www.eclipse.org/modeling/emf/.

[Fow98] Thomas George FOWLER. « Unique coloring of planar graphs ». Thèse de doct. Georgia Institute of Technology, 1998. URL : http://hdl.handle.net/1853/30358.

[FS09] Philippe FLAJOLET et Robert SEDGEWICK. *Analytic Combinatorics*. Cambridge University Press, 2009. ISBN : 9780521898065.

[Fuz] *Fuzzware*. URL : http://www.fuzzware.net/.

[FZ08] Stefan FELSNER et Florian ZICKFELD. « On the Number of Planar Orientations with Prescribed Degrees ». Dans : *The Electronic Journal of Combinatorics* 15.R77 (2008), p. 1. URL : http://www.combinatorics.org/Volume_15/Abstracts/v15i1r77.html.

[FZC94] Philippe FLAJOLET, Paul ZIMMERMAN et Bernard Van CUTSEM. « A calculus for the random generation of labelled combinatorial structures ». Dans : *Theoretical Computer Science* 132.1-2 (1994), p. 1–35. DOI : 10.1016/0304-3975(94)90226-7.

[Gao09] Yong GAO. « The degree distribution of random k-trees ». Dans : *Theoretical Computer Science* 410.8-10 (mar. 2009), p. 688–695. ISSN : 0304-3975. DOI : 10.1016/j.tcs.2008.10.015.

[Gil59] Edgar N. GILBERT. « Random Graphs ». Dans : *Annals of Mathematical Statistics* 30.4 (1959), p. 1141–1144. DOI : 10.1214/aoms/1177706098.

[Gre91] Daniel H. GREENE. « Labelled Formal Languages and Their Uses ». Thèse de doct. Stanford University, 1991. URL : http://handle.dtic.mil/100.2/ADA328418.

[GSS08] John G. Del GRECO, Chandra. N. SEKHARAN et Radhakrishnan SRIDHAR. « Fast Parallel Reordering and Isomorphism Testing of k -Trees ». Dans : *Algorithmica* 32.1 (mar. 2008), p. 61–72. DOI : 10.1007/s00453-001-0052-4.

[HMC07] Grégoire HENRY, Michel MAUNY et Emmanuel CHAILLOUX. *Typer la désérialisation sans sérialiser les types*. Journée francophone des langages applicatifs (JFLA) 2006. Mai 2007. arXiv : 0705.1452.

[Hoc] Sam HOCEVAR. *List of bugs found with zzuf*. URL : http://caca.zoy.org/wiki/zzuf/bugs.

[HP68] Frank HARARY et Edgar M. PALMER. « On acyclic simplicial complexes ». Dans : *Mathematika* 15.1 (1968), p. 115–122.

[Hue97] Gérard HUET. « The Zipper ». Dans : *Journal of Functional Programming* 7.05 (1997), p. 549–554. DOI : 10.1017/S0956796897002864.

[Iba04] Louis IBARRA. « The clique-separator graph for chordal graphs and subclasses of chordal graphs ». Dans : *Symposium on Discrete Mathematics, Nashville, TN*. 2004.

[Jac06] Daniel JACKSON. *Software Abstractions*. MIT Press, 2006. ISBN : 0-262-10114-9.

[JT94] Tommy R. JENSEN et Bjarne TOFT. *Graph Coloring Problems*. 1st. Wiley-Interscience, 1994. ISBN : 0471028657.

[Kaw] Kohsuke KAWAGUCHI. *Sun Multi-Schema Validator*. URL : https://msv.dev.java.net/.

[KCP82] Maria M. KLAWE, Derek G. CORNEIL et Andrzej PROSKUROWSKI. « Isomorphism Testing in Hookup Classes ». Dans : *SIAM Journal on Algebraic and Discrete Methods* 3.2 (juin 1982), p. 260–274. DOI : 10.1137/0603025.

[LDGRV08] Xavier LEROY, Damien DOLIGEZ, Jacques GARRIGUE, Didier RÉMY et Jérôme VOUILLON. *The Objective-Caml system, release 3.11*. Nov. 2008. URL : http://caml.inria.fr/pub/docs/manual-ocaml/index.html.

[LGH04] Pedro G. LIND, Jason A. C. GALLAS et Hans J. HERRMANN. « Coherence in scale-free networks of chaotic maps ». Dans : *Physical Review E* 70.5 (nov. 2004), p. 056207. DOI : 10.1103/PhysRevE.70.056207.

[LLL04] Gilbert LABELLE, Cédric LAMATHE et Pierre LEROUX. « Labelled and unlabelled enumeration of k-gonal 2-trees ». Dans : *Journal of Combinatorial Theory, Series A* 106.2 (mai 2004), p. 193–219. DOI : 10.1016/j.jcta.2004.01.009.

[LMFW08] Daniel LUCRÉDIO, Renata de M. FORTES et Jon WHITTLE. « MOOGLE : A Model Search Engine ». Dans : *Model Driven Engineering Languages and Systems*. 2008, p. 296–310. DOI : 10.1007/978-3-540-87875-9_22.

[Lot05] M. LOTHAIRE. *Applied Combinatorics on Words*. Encyclopedia of Mathematics and its Applications 105. Cambridge University Press, 2005. ISBN : 0521848024. URL : http://www.cambridge.org/catalogue/catalogue.asp?isbn=0521848024.

[Lou87] Guy LOUCHARD. « Exact and asymptotic distributions in digital and binary search trees ». Dans : *Informatique théorique et applications* 21.4 (1987), p. 479–495.

[MA08] Jean-François MARCKERT et Marie ALBENQUE. « Some families of increasing planar maps ». Dans : *Electronic Journal of Probability* 13 (sept. 2008), p. 1624–1671. URL : http://www.math.washington.edu/~ejpecp/viewarticle.php?id=1863.

[Mau04] François MAUREL. « Ocaml-templates, méta-programmation à partir des types ». Dans : *Journées francophones des langages applicatifs*. 2004. URL : http://hal.archives-ouvertes.fr/hal-00153820/.

[MCF03] Stephen J. MELLOR, Anthony N. CLARK et Takao FUTAGAMI. « Guest Editors' Introduction : Model-Driven Development ». Dans : *IEEE Softw.* 20.5 (2003), p. 14–18. DOI : 10.1109/MS.2003.1231145.

[McK97] Terry A. MCKEE. « 2-trees in geodesy, 1850-1950 ». Dans : *Bulletin of the Institute of Combinatorics and its Applications* 21 (1997), p. 77–82.

[MDBS09] Alix MOUGENOT, Alexis DARRASSE, Xavier BLANC et Michèle SORIA. « Uniform Random Generation of Huge Metamodel Instances ». Dans : *Model Driven Architecture - Foundations and Applications*. 5th European Conference, ECMDA-FA 2009. T. 5562. Lecture Notes in Computer Science. Springer-Verlag, 2009, p. 130–145. DOI : 10.1007/978-3-642-02674-4_10.

[MJP06] Lilian MARKENZON, Claudia Marcela JUSTEL et Newton PACIORNIK. « Subclasses of k-trees : Characterization and recognition ». Dans : *Discrete Applied Mathematics* 154.5 (avr. 2006), p. 818–825. DOI : 10.1016/j.dam.2005.05.021.

[MM78] Amram MEIR et John W. MOON. « On the altitude of nodes in random trees ». Dans : *Canadian Journal of Mathematics* 30 (1978), p. 997–1015.

[Moo69] John W. MOON. « The number of labeled k-trees ». Dans : *Journal of Combinatorial Theory* 6.2 (mar. 1969), p. 196–199. DOI : 10.1016/S0021-9800(69)80119-5.

[MR95] Michael MOLLOY et Bruce REED. « A critical point for random graphs with a given degree sequence ». Dans : *Random Structures and Algorithms* 6.2-3 (1995), p. 161–180. DOI : 10.1002/rsa.3240060204.

[NBW06] Mark E. J. NEWMAN, Albert-László BARABÁSI et Duncan J. WATTS. *The Structure and Dynamics of Networks*. Princeton Studies in Complexity. Princeton University Press, 2006. ISBN : 0691113572.

[New05] Mark E. J. NEWMAN. « Power laws, Pareto distributions and Zipf's law ». Dans : *Contemporary Physics* 46.5 (sept. 2005), p. 323–351. DOI : 10.1080/00107510500052444.

[OAS01] OASIS. *RELAX NG Specification*. Déc. 2001. URL : http://www.relaxng.org/spec-20011203.html.

[OMG06] OMG. *Meta Object Facility (MOF) 2.0 Core Specification*. 2006. URL : http://www.omg.org/mof/.

[Oxy] *<oXygen/> XML Editor*. URL : http://www.oxygenxml.com/.

[PAHP07] Gian Luca PELLEGRINI, Lucilla de ARCANGELIS, Hans J. HERRMANN et Carla PERRONE-CAPANO. « Activity-dependent neural network model on scale-free networks ». Dans : *Physical Review E (Statistical, Nonlinear, and Soft Matter Physics)* 76.1 (juil. 2007), p. 016107–9. DOI : 10.1103/PhysRevE.76.016107.

[PB09] Yann PONTY et Olivier BODINI. « Boltzmann en dimension “un et des poussières...” » Dans : *Journées ALÉA 2009*. 2009.

[Pit94] Boris PITTEL. « Note on the heights of random recursive trees and random <m>-ary search trees ». Dans : *Random Structures and Algorithms* 5.2 (1994), p. 337–347. DOI : 10.1002/rsa.3240050207.

[Pon08] Yann PONTY. « Non-redundant random generation from weighted context-free languages ». Dans : *Proceedings of GASCOM'08*. 2008. URL : http://www.lri.fr/~ponty/docs/NonRedundantContextFreeGeneration-Ponty-GASCOM08.pdf.

[PR73] Edgar M. PALMER et Ronald C. READ. « On the Number of Plane 2-Trees ». Dans : *J. London Math. Soc.* s2-6.4 (1973), p. 583–592. DOI : 10.1112/jlms/s2-6.4.583.

[Pro80] Andrzej PROSKUROWSKI. « K-trees : representation and distances ». Dans : *Congressus Numerantium*. T. 29. Utilitas Mathematica, 1980, p. 785–794.

[Pro81] Andrzej PROSKUROWSKI. « Recursive Graphs, Recursive Labelings and Shortest Paths ». Dans : *SIAM Journal on Computing* 10.2 (1981), p. 391–397. DOI : 10.1137/0210028.

[PSS08] Carine PIVOTEAU, Bruno SALVY et Michèle SORIA. « Boltzmann oracle for combinatorial systems ». Dans : *Fifth Colloquium on Mathematics and Computer Science*. DMTCS Proceedings. Discrete Mathematics et Theoretical Computer Science, 2008, p. 475–488. URL : http://www.dmtcs.org/dmtcs-ojs/index.php/proceedings/article/view/dmAI0132.

[PTD06] Yann PONTY, Michel TERMIER et Alain DENISE. « GenRGenS : software for generating random genomic sequences and structures ». Dans : *Bioinformatics* 22.12 (juin 2006), p. 1534–1535. DOI : 10.1093/bioinformatics/btl113.

[RNL08] Colin RUNCIMAN, Matthew NAYLOR et Fredrik LINDBLAD. « Smallcheck and lazy smallcheck : automatic exhaustive testing for small values ». Dans : *Proceedings of the first ACM SIGPLAN symposium on Haskell*. Victoria, BC, Canada : ACM, 2008, p. 37–48. ISBN : 978-1-60558-064-7. DOI : 10.1145/1411286.1411292.

[Ros74] Donald J. ROSE. « On simple characterizations of k-trees ». Dans : *Discrete Mathematics* 7.3-4 (1974), p. 317–322. DOI : 10.1016/0012-365X(74)90042-9.

[RSMOB02] Erzsébet RAVASZ, A. L. SOMERA, Damian A. MONGRU, Zoltán Nagy OLTVAI et Albert-László BARABÁSI. « Hierarchical Organization of Modularity in Metabolic Networks ». Dans : *Science* 297.5586 (août 2002), p. 1551–1555. DOI : 10.1126/science.1073374.

[RSST97] Neil ROBERTSON, Daniel SANDERS, Paul SEYMOUR et Robin THOMAS. « The Four-Colour Theorem ». Dans : *Journal of Combinatorial Theory, Series B* 70.1 (mai 1997), p. 2–44. DOI : 10.1006/jctb.1997.1750.

[Ré85] Jean-Luc RÉMY. « Un procédé itératif de dénombrement d'arbres binaires et son application à leur génération aléatoire ». Dans : *R.A.I.R.O. Informatique théorique* 19.2 (1985), p. 179–195.

[Sch98] Gilles SCHAEFFER. « Conjugaison d'arbres et cartes combinatoires aléatoires ». Thèse de doct. Université de Bordeaux 1, 1998.

[Sel03] Bran SELIC. « The Pragmatics of Model-Driven Development ». Dans : *IEEE Softw.* 20.5 (2003), p. 19–25. DOI : 10.1109/MS.2003.1231146.

[SF95] Robert SEDGEWICK et Philippe FLAJOLET. *An Introduction to the Analysis of Algorithms*. Addison-Wesley Professional, 1995. ISBN : 020140009X.

[SGA07] Michael SUTTON, Adam GREENE et Pedram AMINI. *Fuzzing : Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007. ISBN : 0321446119.

[SMA05] Koushik SEN, Darko MARINOV et Gul AGHA. « CUTE : a concolic unit testing engine for C ». Dans : *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*. Lisbon, Portugal : ACM, 2005, p. 263–272. DOI : 10.1145/1081706.1081750.

[Sta93] Richard P. STANLEY. « A combinatorial decomposition of acyclic simplicial complexes ». Dans : *Discrete Mathematics* 120.1-3 (sept. 1993), p. 175–182. DOI : 10.1016/0012-365X(93)90574-D.

[Sty] *Stylus Studio*. URL : http://www.stylusstudio.com/.

[Tah04] Walid TAHA. « A Gentle Introduction to Multi-stage Programming ». Dans : *Domain-Specific Program Generation*. 2004, p. 30–50. DOI : 10.1007/b98156.

[Tho98] Robin THOMAS. « An update on the four-color theorem ». Dans : *Notices of the American Mathematical Society* 45.7 (août 1998), p. 848–859.

[Ult] *Ultra-Large-Scale Systems : The Software Challenge of the Future*. Rap. tech. Software Engineering Institute, Carnegie Mellon University, 2006, p. 150.

[VL05] Fabien VIGER et Matthieu LATAPY. « Efficient and Simple Generation of Random Simple Connected Graphs with Prescribed Degree Sequence ». Dans : *Computing and Combinatorics*. 2005, p. 440–449. DOI : 10.1007/11533719_45.

[VS46] François VIÈTE et Frans Van SCHOOTEN. « Apollonius Gallus. Seu, Exsuscitata Apollonii Pergæi Περὶ Επαφῶν Geometria. » Dans : *Francisci Vietae Opera mathematica*. ex officinâ Bonaventuræ & Abrahami Elzeviriorum (Lugduni Batavorum), 1646, p. 325–346. URL : http://gallica.bnf.fr/ark:/12148/bpt6k107597d.image.f327.

[W3C04] W3C. *XML Schema Part 2 : Datatypes Second Edition*. Oct. 2004. URL : http://www.w3.org/TR/xmlschema-2/.

[W3C08] W3C. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. Nov. 2008. URL : http://www.w3.org/TR/REC-xml/.

[WG75] Henry William WATSON et Francis GALTON. « On the Probability of the Extinction of Families. » Dans : *The Journal of the Anthropological Institute of Great Britain and Ireland* 4 (1875), p. 138–144. ISSN : 09595295. URL : http://www.jstor.org/stable/2841222.

[Wor85] Nicholas WORMALD. « Counting labelled chordal graphs ». Dans : *Graphs and Combinatorics* 1.1 (déc. 1985), p. 193–200. DOI : 10.1007/BF02582944.

[WS98] Duncan J. WATTS et Steven H. STROGATZ. « Collective dynamics of 'small-world' networks ». Dans : *Nature* 393.6684 (juin 1998), p. 440–442. DOI : 10.1038/30918.

[Xu09] Jin XU. *Mathematical Proofs of Two Conjectures : The Four Color Problem and The Uniquely 4-colorable Planar Graph*. Nov. 2009. arXiv : 0911.1587.

[ZCFR06] Zhongzhi ZHANG, Francesc COMELLAS, Guillaume FERTIN et Lili RONG. « High-dimensional Apollonian networks ». Dans : *Journal of Physics A : Mathematical and General* 39.8 (2006), p. 1811–1818. DOI : 10.1088/0305-4470/39/8/003.

[Zha+08] Zhongzhi ZHANG et al. « Analytical solution of average path length for Apollonian networks ». Dans : *Physical Review E (Statistical, Nonlinear, and Soft Matter Physics)* 77.1 (2008), p. 017102–4. DOI : 10.1103/PhysRevE.77.017102.

[ZRC06] Zhongzhi ZHANG, Lili RONG et Francesc COMELLAS. « High-dimensional random Apollonian networks ». Dans : *Physica A : Statistical Mechanics and its Applications* 364 (mai 2006), p. 610–618. ISSN : 0378-4371. DOI : 10.1016/j.physa.2005.09.042.

[ZYW05] Tao ZHOU, Gang YAN et Bing-Hong WANG. « Maximal planar networks with large clustering coefficient and power-law degree distribution ». Dans : *Physical Review E (Statistical, Nonlinear, and Soft Matter Physics)* 71.4 (avr. 2005), p. 046141–12. DOI : 10.1103/PhysRevE.71.046141.