Heuristiques et conjectures à propos de la 2-dimension des ordres partiels

Kaoutar Ghazi

N° d'ordre : D. U : 810
E D S P I C :

Université Clermont Auvergne

ECOLE DOCTORALE

SCIENCES POUR L'INGENIEUR DE CLERMONT-FERRAND

Thèse

Présentée par

Kaoutar GHAZI

pour obtenir le grade de

DOCTEUR D'UNIVERSITÉ

SPECIALITE : INFORMATIQUE

Titre de la thèse

Heuristiques et conjectures à propos de la 2-dimension des ordres partiels

Soutenue publiquement le 29 Septembre 2017 devant le jury :

Mme. Marianne Huchard	Présidente
M. Christophe Crespelle	Rapporteur
M. Michel Habib	Rapporteur
Mme. Karell Bertet	Examinatrice
Mme. Fatiha Bendali	Examinatrice
M. Lhouari Nourine	Examinateur
M. Laurent Beaudou	Directeur
M. Olivier Raynaud	Directeur

Remerciement

Il m'est agréable de m'acquitter d'une dette de reconnaissance auprès de toutes les personnes dont l'intervention au cours de ma thèse a favorisé son aboutissement jusqu'à sa phase actuelle.

Ainsi, j'exprime mes sincères remerciements à mes directeurs de thèse Olivier Raynaud et Laurent Beaudou de la confiance qu'ils m'ont accordée en acceptant d'encadrer ce travail de thèse et de toutes les heures qu'ils ont consacrées à le diriger. Enfin, de leurs nombreuses relectures et corrections de mes rédactions. Je témoigne mes profondes gratitudes à mon encadrant Laurent Beaudou qui m'a fait profiter de ses compétences mathématiques et prodiguée de conseils qui m'ont permis de progresser. Je le remercie infiniment pour avoir patiemment supervisé et suivi cette thèse. Mes remerciements vont aussi aux Messieurs Arnaud Mary et Éric Thierry pour leur implication, de prés ou de loin, dans mon travail, et spécialement à madame Fatiha bendali-mailfert et monsieur Lhouari Nourine pour l'intérêt et l'attention qu'ils ont porté pour mon sujet de thèse.

Mes remerciements s'adressent également à tous les membres du jury pour avoir accepté d'examiner ce mémoire et en particulier à la présidente du jury, madame Marianne Huchard.

Je profite de l'occasion pour exprimer mes vifs remerciements à tous mes enseignants qui ont veillé à me transmettre leur savoir tout au long de mon parcours scolaire et universitaire, et également aux enseignants qui m'ont confiée ce rôle pendant ma thèse : Vincent, Herve et Alexandre que je remercie encore une fois pour ses précieux conseils. Many thanks for Mr Christian Laforest and my english professors Susan Arbon and Mohamed Chouchene. Et Merci beaucoup à Béatrice pour avoir traité administrativement toutes mes missions et pour sa gentillesse.

Je voudrais aussi remercier le directeur de l'ISIMA Mr. Vincent Barra et le directeur du laboratoire LIMOS Mr. Farouk Toumani, de m'y avoir bien accueillie et de m'avoir fournie de meilleures conditions de travail.

Je remercie tous les thésards du LIMOS, en particulier Maxime, Jan-Thierry, Abdeslem, Rafael et Matthieu avec qui j'ai eu la chance de partager le bureau, Giacomo qui m'a facilitée la tâche de génération du jeux de données, Vanel pour des

corrections de mon français et ses encouragements, Karima, Sahar et Salsabil pour l'amitié qui nous a réunis.

À Houda, Ilham, Chaima, Latifa, Ibtissem, Maimounatou, Marwa, Loubna et aussi Sahar, Karima et Salsabil, je vous dis : Milles Mercis pour tout ... !

Enfin, je ne saurais finir ces remerciements sans une pensée toute spéciale à ma famille dont l'affection, l'amour, le soutien et l'encouragement constants m'ont été d'un grand réconfort dans toute ma vie.

Que tous ceux que je n'ai pas cité, car la liste est longue, se voient ici chaleureusement remerciés.

Titre Heuristiques et conjectures à propos de la 2-dimension des ordres partiels

Résumé Dés qu'on manipule des ordres partiels (des hiérarchies), il est naturel de se demander comment les représenter dans un système informatique. Parmi les solutions proposées dans la littérature, on retrouve le codage par vecteur de bits. Dans cette thèse, nous nous intéressons au problème de calcul d'un codage des ordres par vecteur de bits de taille minimale, aussi connu par le problème de calcul de la 2-dimension des ordres, qui est $\mathcal{NP}$-complet. Nous proposons des solutions du problème de nature heuristique, pour le cas général et pour des classes d'ordres particulières.

Cette thèse présente également des résultats sur des conjectures autour de la 2-dimension des arbres. Notamment celle de Habib *et al.* à propos de la 2-approximabilité de la 2-dimension des arbres. Nous proposons quelques pistes de preuve de cette conjecture puis une reformulation, permettant d'apporter un nouveau regard sur le problème en question et d'espérer trouver des codages des ordres par vecteur de bits efficaces et de taille inférieure à leur 2-dimension. Nous apportons une réponse négative à deux autres conjectures.

Mots-clés Ordre partiel, 2-dimension, codage par vecteur de bits, heuristique, conjecture.

Title Heuristics and conjectures about the 2-dimension of partial orders

Abstract The main question asked when manipulating partial orders (hierarchies), is how to represent them in computer. Among solutions proposed in literature, there is the bit-vector encoding. In this thesis, we consider the problem of computing a bit-vector encoding of orders with minimal size, which is also known as the problem of computing the 2-dimension of orders that is $\mathcal{NP}$-complete. We propose heuristics solutions of the problem for the general case and for some particular order classes. In addition, this thesis presents some results about conjectures on the 2-dimension of trees. Especially, the conjecture of Habib *et al.* about the 2-approximability of the 2-dimension of trees. We propose some ideas of a proof of this conjecture then give a reformulation of it that brings new perspectives on the problem that are finding efficient bits-vector encodings of orders of size less than their 2-dimension. We disprove two other conjectures.

Keywords Partial order, 2-dimension, bit-vector encoding, heuristic, conjecture.

Table des matières

Liste des figures

Liste des tableaux

Liste des symboles

(X, R) Ensemble (partiellement) ordonné ou Ordre (partiel)

(x, y) Arc incident à x et à y

$(Y, Z, \leq)$ Ordre biparti

$<$ Relation d'ordre strict

$\perp$ Élément minimum

$\chi(G)$ Nombre chromatique de G

$\downarrow x$ Idéal de x

$\leq$ Relation d'ordre

$\mathcal{B}(P)$ *Blow up* de P

$\mathcal{B}_n$ Treillis booléen de dimension n

$\mathcal{O}(n)$ Complexité en ordre de grandeur

$\preceq$ Relation de couverture

$\prod_{i=1}^{k} P_i$ Composition série de $P_1, \ldots, P_k$

$\sum_{i=1}^{k} P_i$ Composition parallèle de $P_1, \ldots, P_k$

$\top$ Élément maximum

$\uparrow x$ Filtre de x

$\{x, y\}$ Arête incidente à x et à y

$d(x)$ Degré de l'élément x

$d_{in}(x)$ Degré entrant de x

$d_{out}(x)$ Degré sortant de x

$dim(P)$ Dimension de l'ordre P

$dim_2(P)$ 2-dimension de l'ordre P

$E(G)$ Arêtes du graphe G

$F(x)$ Filtre de x

$h(P)$ Hauteur de P

$I(x)$ Idéal de x

$id(X)$ Relation d'identité sur X

$ImmPred_P(x)$ Prédécesseurs immédiats de x dans P

$ImmSucc_P(x)$ Successeurs immédiats de x dans P

$J(P)$ Ensemble des sup-irréductibles

$M(P)$ Ensemble des inf-irréductibles

$P \cap Q$ Intersection de P et Q

$P \cup Q$ Union de P et Q

$P \rightsquigarrow Q$ P se plonge dans Q

$P + Q$ Composition parallèle de P et Q

$P[Y]$ Sous-ordre de P induit par Y

$P \cdot Q$ Composition série de P et Q

$P \times Q$ Produit Cartésien de P et Q

$P \uplus Q$ Union disjointe de P et Q

P^d Dual de l'ordre P

$P_{/\mathcal{M}}$ Ordre quotient de P, avec $\mathcal{M}$ sa partition modulaire

$P_{x \to Q}$ Substitution de x par Q dans P

$Pred_P(x)$ Prédécesseurs de x dans P

R^{-1} Relation inverse

S_n Couronne

$sp(n)$ Le sperner de l'entier n

$Succ_P(x)$ Successeurs de x dans P

$V(G)$ Sommets du graphe G

$w(P)$ Largeur de P

xRy x est en relation d'ordre avec y

Introduction générale

La définition d'un ordre partiel, aussi appelé ensemble partiellement ordonné, fait intervenir la notion de hiérarchie : c'est une classification dans laquelle les termes classés sont dans une relation de subordination, chaque terme dépendant du précédent et commandant le suivant [Larousse[1]]. Nous illustrons la notion d'ordre partiel par un exemple tiré de l'étude de Foutlane *et al.* portant sur la restauration de la qualité des eaux potables du Maroc [Foutlane *et al.*, 1997]. Il y a quelques années, des algues microscopiques ont été détectées dans les eaux du Maroc comme étant à l'origine de la dégradation de la qualité de ces eaux en termes de goût et d'odeur. Afin de purifier l'eau, de multiples solutions ont été proposées, entre autres l'introduction des carpes chinoises, consommatrices de ces algues, dans les retenues des barrages. Ces algues ne présentent aucun danger si elles sont consommées par l'être humain et les autres poissons comme les maquereaux. L'ensemble des êtres vivants précités, avec leur relation de consommation, constitue un exemple d'ordre partiel et permet de modéliser les résultats de cette étude : carpes $<$ algues, homme $<$ algues, maquereaux $<$ algues, homme $<$ carpes, homme $<$ maquereaux.

Les ordres partiels trouvent des applications dans divers domaines de l'informatique comme l'analyse des systèmes parallèles [Gazagnaire, 2008], la caractérisation de solutions robustes en ordonnancement [La, 2005], la modélisation des systèmes concurrents [Pratt, 1986], etc.. Ces applications font de l'étude de cette structure d'ordre le centre d'intérêt de nombreux travaux de recherche, notamment ceux de Urrutia [Urrutia, 1989], de Fernandez *et al.* [Fernandez *et al.*, 2009] et de Lam [Lam, 2015]. La manipulation des ordres partiels nécessite, dans un premier temps, l'étude de leur représentation dans un système informatique. Cette thèse s'inscrit dans le cadre de cette étude.

Contexte scientifique

La représentation des données dans un système informatique correspond in fine à une représentation binaire. Dans une telle représentation, chaque donnée est transformée en une séquence de bits. Toutefois, il existe d'autres représentations intermédiaires plus commodes, comme la représentation décimale ou hexadécimale des entiers. La transformation des données d'une représentation initiale en une autre représentation bien définie est appelée **codage** (cf. Figure 1).

L'étude de la représentation des données dans un système informatique repose sur la recherche de méthodes efficaces pour le codage de ces données. Les critères d'efficacité d'un codage dépendent du type des données à coder. Dans cette thèse, nous nous intéressons au codage des données ordonnées (codage des ordres).

1. www.larousse.fr/dictionnaires/francais/hierarchie/. Consulté en 2017

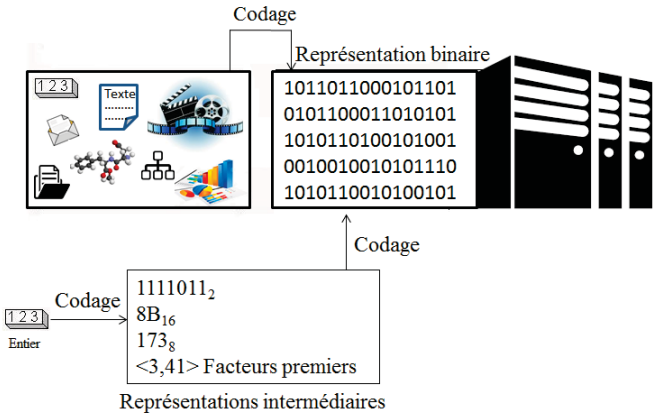


FIGURE 1: Représentation et codage des données

Le codage d'un ordre permet de déterminer l'espace mémoire dédié à son stockage ainsi que le coût d'interrogation de ses éléments stockés. Nous évaluons alors l'efficacité d'un codage d'un ordre P à partir de trois critères : (1) la taille du codage, qui correspond au nombre de bits codant chacun des éléments de P ; (2) le coût de comparaison de deux éléments de P ; (3) la complexité en temps et en espace du calcul du codage de P. En notant e la taille d'un codage de l'ordre P, le stockage de ce dernier nécessite $e \times |P|$ bits. Un codage est d'autant plus efficace que les mesures de ses critères d'évaluation sont faibles.

Il existe plusieurs méthodes de codage des ordres partiels dans la littérature, mais nous nous focalisons sur le **codage par vecteur de bits**.

Le codage par vecteur de bits d'un ordre consiste à associer à chacun de ses éléments un vecteur de bits de même taille. Le calcul de ce codage se ramène au plongement de l'ordre dans un treillis booléen (cf. Figure 2). La dimension de ce treillis est alors la taille de ce codage. La taille minimale d'un codage par vecteur de bits d'un ordre P correspond à sa 2-dimension, notée $dim_2(P)$. La 2-dimension de P se définit donc aussi par le plus petit treillis booléen dans lequel se plonge P. Cette thèse, intitulée « **heuristiques et conjectures à propos de la 2-dimension des ordres partiels** », porte sur l'étude de ce paramètre – dim_2.

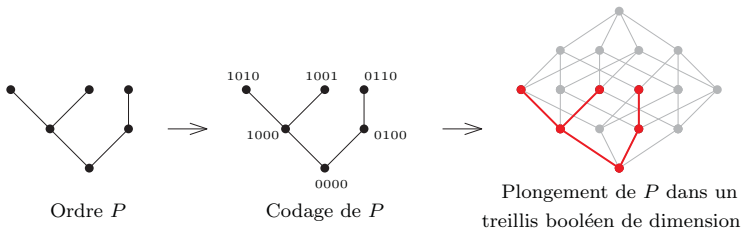


FIGURE 2: Codage par vecteur de bits et la 2-dimension

Contributions

Les travaux effectués durant cette thèse ont été réalisés sous l'encadrement de Laurent Beaudou et Olivier Raynaud. Nos contributions principales sont les suivantes :

- ☞ Le développement d'une heuristique de codage des ordres par vecteur de bits à travers la décomposition modulaire [Beaudou *et al.*, 2016] : cette heuristique est à la base d'un travail collaboratif avec Éric Thierry et Giacomo Kahn pour le cas des ordres séries-parallèles. Un ordre série-parallèle est un ordre qui ne contient pas de sous-ordre isomorphe à l'ordre  . Étant donné un ordre série-parallèle P, l'ordre P peut être vu comme une composition série ou parallèle d'ordres $P_1, \dots P_n$. Les ordres P_i, avec i entre 1 et n, sont générés par la décomposition de P en modules forts maximaux et sont aussi des ordres séries-parallèles par définition. Pour coder P, nous proposons alors un codage de la composition série (resp. parallèle) des ordres déjà codés de manière récursive. Notons que la composition série (resp. parallèle) des ordres à un seul élément est une chaîne (resp. antichaîne) dont la 2-dimension est connue. Afin de traiter le cas général des ordres, nous intégrons dans cette heuristique le codage proposé par Krall *et al.* [Krall *et al.*, 1997], dans le but de coder la composition ni série ni parallèle des ordres.
- ☞ Un codage par vecteur de bits des arbres : ce codage suit la même stratégie des trois dernières heuristiques existantes de codage des arbres [Habib *et al.*, 2004], [Filman, 2002], [Colomb *et al.*, 2008]. Ces heuristiques reposent sur le calcul d'un poids pour chaque élément d'un arbre à partir des poids de ses fils, déjà calculés. Le poids d'un élément reflète la taille du codage du sous-arbre enraciné en cet élément. Soit T un arbre et x un élément de T. Soit $s_1 \leq \dots \leq s_n$ les poids des fils de x. Pour calculer le poids de x, nous observons que les trois heuristiques procèdent d'abord par le partitionnement de la séquence $[s_1, \dots, s_n]$ en des sous-séquences $S_1, \dots, S_k$ tel que chaque sous-séquence S_i génère un poids p_i dans $\{s_n - 1, s_n\}$. Ensuite, elles calculent le poids de x à partir de la nouvelle séquence $[p_1, \dots, p_k]$. Nous proposons alors une nouvelle méthode de partitionnement de la séquence initiale $[s_1, \dots, s_n]$.
- ☞ La réfutation de deux conjectures à propos de la 2-dimension des ordres : la première conjecture, de Thierry, affirme que la 2-dimension des arbres n-complets de hauteur h vaut exactement $h.sp(n)$ [Thierry, 2001]. La seconde conjecture, attribuée à Habib *et al.* atteste que la 2-dimension d'un arbre T dont les sous-arbres $T_1, \dots, T_k$, tel que chacun est enraciné par un des k fils de la racine de T : vérifie $\sum_{i=1}^{k} 2^{dim_2(T_i)} \leq 2^{dim_2(T)}$ [Habib *et al.*, 2004].

☞ L'étude d'une conjecture de Habib *et al.* à propos de la 2-approximabilité de la 2-dimension des arbres [Habib *et al.*, 2004] : nous confirmons la conjecture dans le cas des chenilles et des arbres 2^k-complets puis nous développons le principe d'une preuve de cette conjecture par induction et par monotonie. Nous proposons également une nouvelle formulation de la conjecture.

Structure du manuscrit

Le manuscrit de la thèse comporte quatre chapitres. Dans le **premier chapitre**, nous définissons formellement la notion du codage des ordres partiels, puis nous présentons les différentes méthodes de codage et plus précisément le codage par vecteur de bits, dont la taille minimale définit la 2-dimension des ordres. Dans le même chapitre, nous citons certaines propriétés du paramètre « 2-dimension », puis nous dressons un état de l'art des principaux travaux sur le calcul de la 2-dimension pour le cas général des ordres partiels, puis en particulier pour le cas des arbres. Nous listons à la fin de ce chapitre certains problèmes ouverts autour de la 2-dimension des ordres partiels.

Le **second chapitre** de ce manuscrit décrit les solutions que nous proposons pour résoudre le problème de calcul de la 2-dimension des ordres. Ce problème se définit également par le calcul d'un codage des ordres par vecteur de bits de taille minimale. Comme il s'agit d'un problème $\mathcal{NP}$-difficile, nous proposons alors une heuristique de codage par vecteur de bits, d'abord pour les ordres séries-parallèles, en se basant sur la technique de décomposition modulaire. Le choix de cette classe d'ordres ainsi que la technique de codage est justifié dans ce chapitre. Nous combinons ce travail avec celui de Krall *et al.* [Krall *et al.*, 1997] pour concevoir ensuite une heuristique calculant un codage par vecteur de bits pour tout ordre. Dans cette thèse, nous traitons également le cas des arbres. À base d'une analyse approfondie des heuristiques existantes, nous dressons leur stratégie commune pour le codage des arbres puis, suivant cette stratégie, nous proposons une nouvelle heuristique calculant un codage par vecteur de bits des arbres de taille plus petite. Toutes les heuristiques proposées sont testées sur des hiérarchies de référence ou d'autres générées aléatoirement. Les résultats obtenus (théoriques et pratiques) sont donnés dans ce second chapitre.

Le **troisième chapitre** est dédié à l'étude de certains problèmes ouverts autour de la 2-dimension des ordres partiels énoncés sous forme de conjectures. Nous infirmons une première conjecture de Thierry à propos de la 2-dimension exacte de la classe des arbres n-complets [Thierry, 2001], puis une seconde de Habib *et al.* à propos d'une borne inférieure de la 2-dimension des arbres [Habib *et al.*, 2004]. En outre, nous confirmons la conclusion de chacune des deux conjectures sur

quelques classes d'arbres. La dernière partie de ce chapitre est consacrée à l'étude d'une conjecture de Habib *et al.* à propos de la 2-approximabilité de la 2-dimension des arbres [Habib *et al.*, 2004]. En effet, les auteurs proposent un algorithme de codage des arbres appelé « Dicho » et conjecturent que cet algorithme donne une 2-approximation de la 2-dimension des arbres. Dans ce chapitre, nous étendons les classes d'arbres validant leur conjecture, puis nous évoquons les différentes pistes abordées pour sa résolution. Enfin, nous donnons une re-formulation de la conjecture, dont l'intérêt est accentué dans le quatrième chapitre.

Au **quatrième chapitre**, nous généralisons d'abord la notion de codage par vecteur de bits puis nous proposons d'autres codages par vecteur de bits, pour le cas des arbres binaires, plus efficaces que le codage basé sur le plongement d'ordres dans un treillis booléen. Nous donnons à la fin du chapitre un nouveau codage par vecteur de bits pour tout arbre T, calculable en temps polynomial, et nous conjecturons que sa taille est plus petite que la 2-dimension de T. Notre conjecture implique que l'algorithme « Dicho » est une 2-approximation de la 2-dimension des arbres et permet également de donner un élan vers un codage par vecteur de bits des arbres plus intéressant que celui basé sur le calcul de la 2-dimension.

Nous définissons formellement les notions évoquées tout au long de ce manuscrit juste après cette introduction.

Définitions et Préliminaires

Cette section englobe les notions et notations utilisées dans ce manuscrit. Pour les définitions proposées, nous invitons le lecteur à consulter le livre de Nathalie Caspard, Bruno Leclerc et Bernard Monjardet [Caspard *et al.*, 2007] pour des définitions similaires. Le lecteur pourra également se référer aux travaux de William T. Trotter (voir chapitre 8 du livre *Handbook of combinatorics* [Trotter, 1995]) et aux livres de Brian Davey et Hilary Priestley [Davey et Priestley, 2002] et de Steven Roman [Roman, 2009] à propos des ensembles partiellement ordonnés. Pour plus de détails sur la décomposition modulaire, nous recommandons l'article de Michel Habib et Christophe Paul [Habib et Paul, 2010]. Nous recommandons également les livres de Christian Roux [Roux, 2009] et Douglas B. West [West, 2000] pour une initiation en théorie des graphes ainsi que le livre d'Ivan Lavallée [Lavallée, 2008] et celui de Michael Sipser [Sipser, 1996] à propos d'algorithmique et complexité. Le lecteur peut consulter la liste des symboles et l'index pour une recherche rapide des notions et notations abordées.

Ordre partiel : Définition et caractéristiques

Un **ordre partiel** est une relation binaire (cf. définition ci-dessous) réflexive, antisymétrique et transitive, souvent noté $\leq$. Un ordre partiel est aussi appelé une **relation d'ordre** . Si un ordre partiel est irréflexif, on dit qu'il est **strict** et on le note $<$. Un **ordre total** ou **linéaire** est un ordre partiel qui est de plus une relation binaire totale (Définition 1.1 [Caspard *et al.*, 2007]).

Définition (Relation binaire)**.** *Soit X et Y deux ensembles. Une **relation binaire** R, entre X et Y, est l'ensemble des paires (x, y) de $X \times Y$ tel que x est en relation avec y, noté xRy. Quand $X = Y$, on dit que R est une relation binaire sur X. Une relation binaire est :*

- *Réflexive, si pour tout x de X, xRx.*
- *Irréflexive, si pour tout x de X, $(x, x) \notin R$.*
- *Symétrique, si pour tout x et y de X, xRy implique yRx.*
- *Antisymétrique, si pour tout x et y de X, xRy et yRx impliquent $x = y$.*
- *Transitive, si pour tout x, y et z de X, xRy et yRz impliquent xRz.*
- *Totale, si pour tout x et y de X, xRy ou yRx.*

Un ensemble X muni d'une relation d'ordre partiel R est appelé un **ensemble partiellement ordonné** noté (X, R). Par exemple, l'ensemble $\{\{1, 2, 3\}, \{1, 2\}, \{1, 3\}\}$ muni de la relation d'ordre $\subseteq$ est un ensemble partiellement ordonné.

Au fil de ce manuscrit, nous utiliserons parfois le terme ordre (partiel) pour désigner un ensemble (partiellement) ordonné. Dans la suite, nous ne considérons que les ordres finis, i.e. les ordres définis sur un ensemble fini d'éléments.

Soit $P = (X, \leq)$ un ordre partiel et soient x et y deux éléments de X. On dit que x et y sont comparables dans P si $x \leq y$ ou $y \leq x$. Dans le cas contraire, on dit qu'ils sont incomparables. La relation $x \leq y$ exprime que x est plus petit que y, ou autrement, y est plus grand que x. Si $x \leq y$ et s'il n'existe pas un élément z de X tel que $x \leq z < y$, on dit que x est couvert par y ou encore y couvre x. Cette **relation de couverture** est notée $\preceq$.

Un ordre dont tous les éléments sont comparables (resp. incomparables) est appelé chaîne (resp. antichaîne). Une chaîne de P est dite maximale si elle n'est pas incluse dans une autre chaîne de P. Idem pour une antichaîne. Par ailleurs, la **hauteur** de P, notée $h(P)$, correspond au nombre d'éléments d'une chaîne maximale induite de P moins un, et sa **largeur**, notée $w(P)$, correspond au nombre d'éléments d'une antichaîne maximale induite de P. La **taille** de P est le cardinal de X, noté $|X|$.

Dans la suite, nous allons parfois utiliser P pour faire référence à la fois à l'ordre P et à ses éléments.

Représentation des ordres

Soit $X = \{a, b, c, d, e\}$ un ensemble d'éléments et soit $P = (X, \leq)$ un ordre partiel tel que a est plus petit que b, c, d et e, l'élément e est plus grand que a, b, c et d, et les éléments b, c et d sont incomparables. On peut visualiser graphiquement P via un **diagramme sagittal** (Figure 3) qui est un graphique comprenant deux ensembles S_1 et S_2 reliés par des flèches. Chaque ensemble contient les éléments de X et une flèche reliant un élément x de S_1 à un élément y de S_2 signifie que ces deux éléments sont en relation d'ordre dans P et $x \leq y$. Ce diagramme peut être simplifié par la superposition de S_1 et S_2.

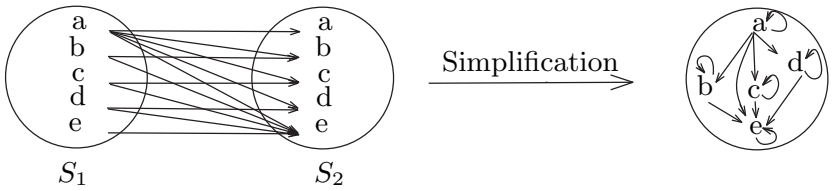


FIGURE 3: Diagramme Sagittal

Nous pouvons encore simplifier le diagramme sagittal en un **diagramme de Hasse** (Définition 1.8 [Caspard *et al.*, 2007]) par la suppression des relations réflexives et celles déduites par transitivité, puis par la transformation des flèches,

dans le graphique obtenu, en simples arêtes tracées suivant une orientation bien définie (Figure 4). Le sens de l'orientation est convenu par certains auteurs du haut vers le bas et par d'autres du bas vers le haut. Dans ce manuscrit nous considérons une orientation du bas vers le haut. Cette orientation signifie que l'élément en bas est plus petit que l'élément en haut.

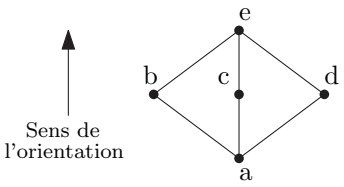


FIGURE 4: Diagramme de Hasse

Éléments (sous-structures) particuliers d'un ordre

Soit $P = (X, \leq)$ un ordre. Pour $Y \subseteq X$, la restriction de $\leq$ à Y, notée $\leq_Y$, est un ordre partiel sur Y. On omet souvent l'indice Y et on note $(Y, \leq)$ le **sous-ordre** de P induit par Y ou l'**ordre induit** de P sur Y, aussi noté $P[Y]$.

Soit x un élément de X. L'élément x est **maximal** si pour tout élément y de P, $x \leq y$ implique $x = y$, et il est **maximum**, si pour tout élément y de P on a $y \leq x$. Lorsqu'il existe, on note $\top$ le maximum de P. Idem, l'élément x est **minimal** si pour tout élément y de P, $y \leq x$ implique $x = y$, et il est **minimum**, si pour tout élément y de P on a $x \leq y$. Lorsqu'il existe, on note $\bot$ le minimum de P (Définition 1.38 [Caspard *et al.*, 2007]).

Soit $Y \subset X$ et m un élément de X. Si pour tout élément y de Y on a $y \leq m$, l'élément m est appelé **majorant** de $P[Y]$. S'il existe, le minimum de l'ensemble des majorants de $P[Y]$ est appelé **borne supérieure** ou **supremum** de $P[Y]$. De la même façon, nous définissons un minorant de $P[Y]$ comme étant l'élément m de X tel que pour tout élément y de Y on a $m \leq y$. Le maximum de l'ensemble des minorants de $P[Y]$, s'il existe, est appelé **borne inférieure** ou **infimum** de $P[Y]$ (Définition 1.39 [Caspard *et al.*, 2007]).

Tout élément x de P possède une **liste des prédécesseurs** (resp. **liste des successeurs**), potentiellement vide, notée $Pred_P(x)$ (resp. $Succ_P(x)$) et définie par l'ensemble de ses ancêtres (resp. descendants) dans P, i.e.

$$Pred_P(x) = \{y \in X | y < x\} \text{ et } Succ_P(x) = \{y \in X | x < y\}.$$

La **liste des prédécesseurs immédiats** (resp. **liste des successeurs immédiats**) de x, notée $ImmPred_P(x)$ (resp. $ImmSucc_P(x)$) est définie par l'ensemble de ses parents (resp. ses fils) dans P, i.e.

$$ImmPred_P(x) = \{y \in X | y \prec x\} \text{ et } ImmSucc_P(x) = \{y \in X | x \prec y\}.$$

On définit le **degré sortant** de x par $|ImmSucc_P(x)|$ et on le note $d_{out}(x)$. Idem, on note son **degré entrant** par $d_{in}(x)$ et il correspond à $|ImmPred_P(x)|$. Le degré de x, noté $d(x)$, n'est autre que $d_{out}(x) + d_{in}(x)$.

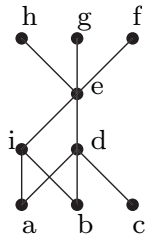


FIGURE 5: Un ordre E

Dans la Figure 5, les éléments d, e, f, g et h sont des successeurs de c et aussi des majorants du sous-ordre induit de E sur $Y = \{a, b, c\}$. L'élément d, étant le minimum de ces majorants est donc une borne inférieure de $E[Y]$. De même, les éléments a, b, c, d et e sont des prédécesseurs de f et aussi des minorants du sous-ordre de E induit sur $Z = \{h, g, f\}$. Comme e est le maximum de ces minorants, il est la borne inférieure de $E[Z]$.

Un élément x de P est dit **sup-irréductible** s'il a un unique prédécesseur immédiat ou s'il existe un autre élément y de P tel que tout prédécesseur immédiat de x est aussi un prédécesseur immédiat de y. L'élément i de l'ordre E (Figure 5) est un sup-irréductible dans E. L'ensemble des éléments sup-irréductibles de P est noté $J(P)$. Similairement, on note $M(P)$ l'ensemble des éléments **inf-irréductibles** de P, qui sont les éléments avec un unique successeur immédiat ou dont les successeurs immédiats sont aussi des successeurs immédiats d'un autre élément de P. L'élément d de l'ordre E (Figure 5) est un inf-irréductible (Définition 1.41 [Caspard *et al.*, 2007]). Formellement, on a

$$x \in J(P) \Leftrightarrow \exists y \in X \text{ tel que } ImmPred_P(x) \subseteq ImmPred_P(y) \cup \{y\}.$$

$$x \in M(P) \Leftrightarrow \exists y \in X \text{ tel que } ImmSucc_P(x) \subseteq ImmSucc_P(y) \cup \{y\}.$$

Un **idéal** de P est un sous-ensemble I de X vérifiant pour tout élément x de I et y de X, $y \leq x$ implique y est dans I. Pour tout élément x de P, on appelle idéal de x dans P, et on le note $I(x)$ ou $\downarrow x$, l'ensemble des éléments plus petit ou égaux à x dans P, i.e. $\{y \in X | y \leq x\}$. La notion duale d'un idéal est le **filtre**. Ainsi, pour tout élément x de P, on note $F(x)$ ou $\uparrow x$ son filtre et ceci correspond aux éléments de P plus grand ou égaux à x dans P, i.e. $\{y \in X | x \leq y\}$ (Définition 1.42 [Caspard *et al.*, 2007]).

Définition (Relations flèches)**.** *Soit P un ordre et x, y deux éléments de P.*
- *$x \downarrow y$ si x est un élément minimal dans $P \backslash I(y)$.*
- *$x \uparrow y$ si y est un élément maximal dans $P \backslash F(x)$.*
- *$x \updownarrow y$ si $x \downarrow y$ and $x \uparrow y$. Dans ce cas, on dit que x et y sont en relation double flèche.*

Vous trouverez dans la Définition 1.36 [Caspard *et al.*, 2007] une description équivalente des relations flèches.

Définition (Paire critique)**.** *Soit P un ordre et x, y deux éléments de P. La paire (x, y) est une* ***paire critique*** *si $x \updownarrow y$.*

Dans l'ordre E (Figure 5) nous avons $g \downarrow f$, $c \uparrow i$ et $i \updownarrow d$. Donc, la paire (i, d) est une paire critique dans E.

Classification des ordres

Nous introduisons dans la Figure 6 la hiérarchie des classes d'ordres citées dans ce manuscrit.

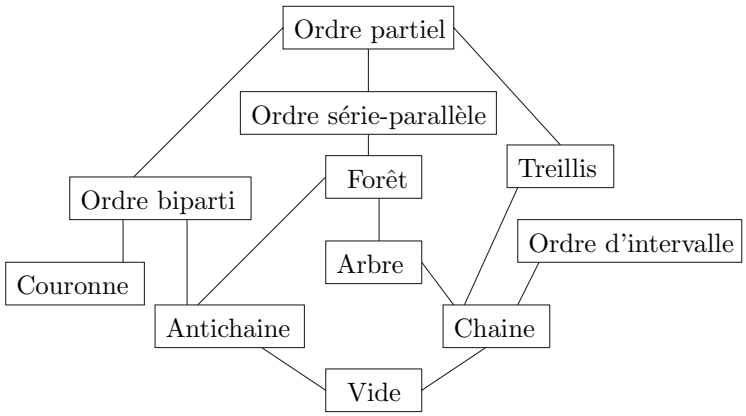


FIGURE 6: Hiérarchie de quelques classes d'ordres

Rappelons qu'une **chaîne** est un ordre total, i.e. un ordre dont tous les éléments sont comparables, par exemple, l'ordre lexicographique défini sur l'alphabet français est une chaîne. Le **degré d'une chaîne** est la somme des degrés sortants de ses éléments. Concernant l'**antichaîne**, il s'agit d'un ordre partiel dont tous les éléments sont incomparables, ainsi, les nombres premiers munis de la relation divise est une antichaîne. Un **ordre série-parallèle** est un ordre partiel dont chacun de ses sous-ordres induit par un module est soit une chaîne soit une antichaîne soit un ordre série-parallèle. La notion de module est donnée dans la section suivante.

Un **arbre** est un ordre partiel possédant un minimum appelé la racine et dont chacun de ses éléments, sauf la racine, a un parent unique. On appelle une feuille, un élément de l'arbre qui n'a aucun fils. L'arbre couvrant d'un ordre $P = (X, R)$, avec $\perp$ un élément de X, est l'arbre (X, R') avec $R' \subseteq R$. Un arbre est généralement noté par T. En biologie, un arbre phylogénétique est un ordre qui a la structure d'arbre. Une **forêt** est une union disjointe de plusieurs arbres.

Étant donné un arbre T, on dit que T est un **arbre n-aire** si chacun de ses sommets possède au plus n fils. Un **arbre binaire** est un arbre 2-aire. Lorsque chaque sommet d'un arbre n-aire T possède exactement n fils et toutes les chaînes reliant la racine de T à ses feuilles ont la même taille, on dit que T est un **arbre n-complet**. Une chaîne est un arbre à une feuille tandis qu'une **chenille** est un arbre qui, privé de ses feuilles, devient une chaîne. Une **peigne** est une chenille dont chaque sommet a au plus 2 fils. Nous parlons d'une peigne de degré n si chacun de ses sommets possède n fils.

Un **ordre biparti** P est un ordre partiel de hauteur 1 dont tout élément x est minimal ou maximal dans P. Soit Y (resp. Z) l'ensemble des éléments maximaux (resp. minimaux) de P. On note P par $(Y, Z, \leq)$. Ainsi, pour y et z deux éléments de P, $z \leq y$ implique y dans Y et z dans Z. Une **couronne** à $2n$ éléments, notée S_n, est un ordre partiel défini sur $\{1, \ldots, 2n\}$ avec $i < n + i$, $i + 1 < n + i$ pour i de 1 à $n - 1$, et $1 < 2n$, $n < 2n$.

Un **treillis** (Lattice en anglais), généralement noté par L, est un ordre partiel dont chaque paire d'éléments admet une borne supérieure et une borne inférieure dans L (Définition 2.13 [Caspard *et al.*, 2007]). Si tout sous-ensemble de L admet à la fois une borne supérieure et une borne inférieure, on parlera de **treillis complet**. Par exemple, l'ensemble des parties d'un ensemble $\{1, \ldots, n\}$ muni de la relation d'ordre inclusion est un treillis complet appelé **treillis booléen** (aussi appelé **hypercube** ou simplement **cube**) de dimension n et noté $\mathcal{B}_n$, avec $\mathcal{B}_n = (2^{\{1,\ldots,n\}}, \subseteq)$.

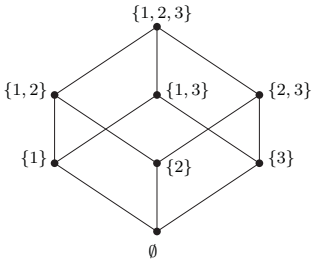


FIGURE 7: Le treillis booléen $\mathcal{B}_3 = (2^{\{1,2,3\}}, \subseteq)$

Un **ordre d'intervalle** est un ordre défini sur un ensemble d'intervalles de $\mathbb{R}$ tel que pour toute paire d'intervalles I_1 et I_2 de cet ordre, I_1 est en relation d'ordre avec I_2 si pour tout élément x de I_1 et y de I_2, $x <_{\mathbb{R}} y$.

Opérations sur les relations d'ordres

Soit X un ensemble d'éléments. La **relation d'identité** sur X, notée $id(X)$, est l'ensemble des paires (x, x) avec x un élément de X.

Soit R une relation d'ordre sur X. La **relation inverse** de R, notée R^{-1}, est la relation définie sur X telle que pour toute paire d'éléments x et y de X, xRy si et seulement si $yR^{-1}x$.

La **fermeture réflexive** de R est $R \cup id(X)$. Remarquez que R en tant que relation d'ordre non stricte est déjà fermée par réflexivité. La **réduction réflexive** de R serait $R \backslash id(X)$ et définirait une relation d'ordre stricte.

La **fermeture transitive** de R est $\bigcup_{i=1}^{\infty} R^i$ avec ($R^i = R \circ R \circ \ldots R$, i fois). Toutefois, la **réduction transitive** de R est $R \backslash \bigcup_{i=2}^{\infty} R^i$.

La **fermeture symétrique** de R est $R \cup R^{-1}$.

Définition (Extension (linéaire) d'une relation binaire)**.** *Étant donné deux relations d'ordres R_1 et R_2, on dit que R_2 est une extension de R_1 si $R_1 \subseteq R_2$. Cette extension est linéaire si R_2 est totale.*

Hormis les opérations de fermetures, la **linéarisation des ordres partiels** (cf. définition ci-dessus) est parmi les opérations fondamentales en théorie des ordres. Ainsi, la relation « divise » définie sur les entiers naturels admet comme extension linéaire l'ordre usuel sur $\mathbb{N}$.

Opérations sur les ordres

Soit $P = (X, \leq_P)$ et $Q = (Y, \leq_Q)$ deux ordres.

Le **dual** de P est l'ordre $P^d = (X, \leq_{P^d})$, vérifiant pour toute paire d'éléments x et y de X, $x \leq_{P^d} y$ si et seulement si $y \leq_P x$. Remarquez que P est le dual de son dual.

L'**intersection** de P et Q est l'ordre partiel $(X \cap Y, \leq)$, vérifiant pour toute paire d'éléments x et y de $X \cap Y$, $x \leq y$ si et seulement si $x \leq_P y$ et $x \leq_Q y$. Par ailleurs, l'**union** (resp. union disjointe) de P et Q est l'ordre partiel $(X \cup Y, \leq)$ (resp. $(X \uplus Y, \leq)$) défini sur l'union (resp. union disjointe) de X et Y et vérifiant pour toute paire d'éléments x et y de $X \cup Y$ (resp. $X \uplus Y$), $x \leq y$ si et seulement si $x \leq_P y$ ou $x \leq_Q y$. On note l'union (resp. union disjointe) de P et Q par $P \cup Q$ (resp. $P + Q$).

Un **plongement** de P dans Q est défini par une application f de X dans Y préservant la relation d'ordre : pour toute paire d'éléments x et y de X, $x \leq_P y$ si et seulement si $f(x) \leq_Q f(y)$. La fonction f est aussi appelée **morphisme d'ordre**. Quand f est bijective, on parle d'**isomorphisme d'ordre**. Si la fonction f existe, on dit que P admet un plongement dans Q et on note $P \rightsquigarrow Q$.

Le **produit Cartésien** de P et Q, noté $P \times Q$, est l'ordre partiel $(X \times Y, \leq)$, tel que pour toute paire d'éléments (x_1, y_1) et (x_2, y_2) de $X \times Y$, $(x_1, y_1) \leq (x_2, y_2)$ si et seulement si $x_1 \leq_P x_2$ et $y_1 \leq_Q y_2$.

La **composition série** de P et Q, notée $P \cdot Q$, est l'ordre partiel $(X \uplus Y, \leq)$, tel que pour toute paire d'éléments x et y de $X \uplus Y$, $x \leq y$ si et seulement si

- $(x, y) \in X^2$ et $x \leq_P y$, ou
- $(x, y) \in Y^2$ et $x \leq_Q y$, ou
- $(x, y) \in X \times Y$.

Une **composition parallèle** de P et Q n'est autre que l'ordre $P + Q$.

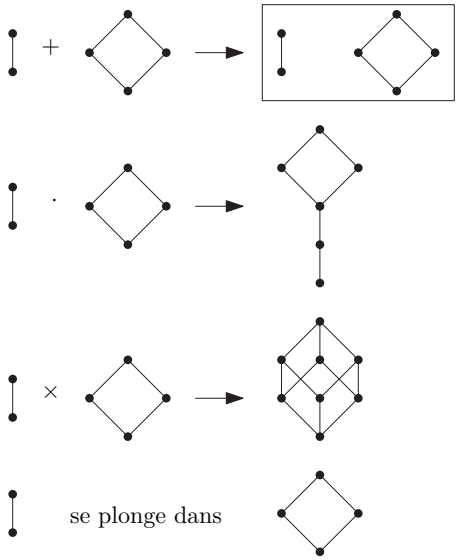


FIGURE 8: Quelques opérations sur les ordres

La **substitution** d'un élément x de P par Q, que l'on note $P_{x \to Q}$, consiste à remplacer x par Q dans P de telle sorte que

$$\forall y \in X, \quad y \leq_P x \Leftrightarrow y \leq_P q \text{ pour tout } q \text{ de } Q.$$

La Figure 9 illustre cette opération de substitution.

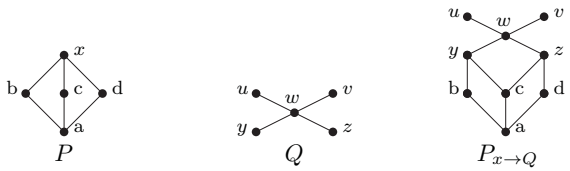


FIGURE 9: Substitution de x par Q dans P

Une **décomposition d'un ordre** P consiste à générer l'ensemble des sous-ordres $P_1, \ldots, P_n$ de P vérifiant certaines propriétés. Si pour tout élément x de P il existe un unique entier i, avec i entre 1 et n, tel que x est dans P_i, cette décomposition est appelée un **partitionnement** de P.

Il existe plusieurs formes de décomposition d'un ordre : **décomposition en chaînes**, **décomposition en antichaînes** et la **décomposition modulaire**.

Une décomposition ou un partitionnement de P en (anti)chaînes consiste à extraire un ensemble de sous-ordres de P dont chacun est une (anti)chaîne. Lorsqu'il s'agit d'une décomposition modulaire, chacun de ces sous-ordres correspondra à un module de P.

Décomposition modulaire des ordres

On appelle **module** de P, un sous-ensemble M de X dont les éléments se comportent comme un seul élément vis-à-vis de l'extérieur. Cela se traduit par,

$$\forall x \in X \backslash M, \forall (y,z) \in M^2, \quad x \leq_P y \Leftrightarrow x \leq_P z.$$

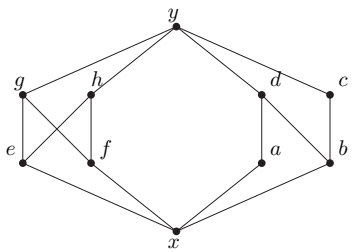


FIGURE 10: Un ensemble partiellement ordonné E

Un **module fort** est un module qui ne chevauche aucun autre module. En d'autres termes, un module M est fort si pour tout autre module M' de P, on a soit $M \subset M'$, $M' \subset M$ ou bien $M \cap M'$ est vide. Les **modules triviaux** de P sont : $\emptyset$, X et $\{x\}$ pour tout x de X. Il est clair qu'un module trivial est fort. Un **module maximal** est un module non trivial qui n'est pas inclus dans un autre module. L'ensemble $\{x, a, b, c, d, e, f, g, h\}$ est un module de l'ordre E (Figure 10) mais il n'est pas fort puisqu'il chevauche le module $\{y, a, b, c, d, e, f, g, h\}$. Dans le même ordre E, on a $\{a, b\}$ un module fort et $\{a, b, c, d, e, f, g, h\}$ un module fort maximal.

Une **partition modulaire** de P est un partitionnement de X en modules. L'ensemble $\{\{x\} : x \in P\}$ est une partition modulaire de P. Une partition modulaire est maximale si ses modules sont forts et maximaux. Tout ordre possède une partition modulaire maximale unique et dorénavant, nous ne considérerons que ce type de partition. La partition modulaire maximale de l'ordre E est $\{x\}, \{a, b, c, d, e, f, g, h\}, \{y\}$.

Finalement, une **décomposition modulaire** de P consiste à générer une partition modulaire maximale de P, puis, pour chaque module non trivial M de la partition, générer une partition modulaire maximale de $P[M]$ et ainsi de suite.

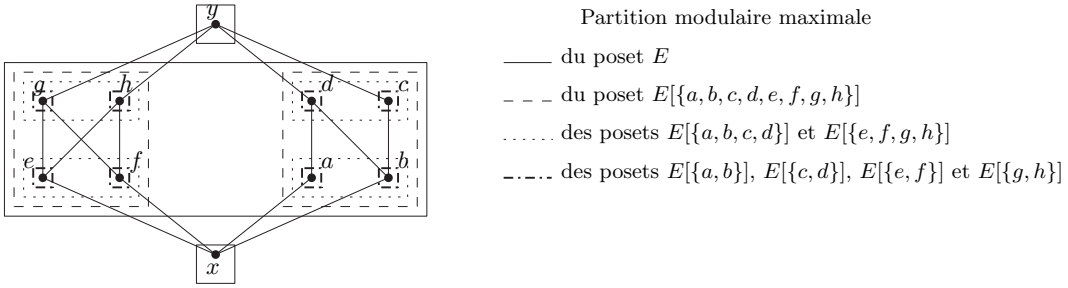


FIGURE 11: Décomposition modulaire de E

Soit $\mathcal{M}$ une partition modulaire (maximale) de P. L'**ordre quotient** de P, noté $P_{/\mathcal{M}}$, est l'ordre $(\mathcal{M}, \leq_m)$ vérifiant pour toute paire d'éléments M_1 et M_2 de $\mathcal{M}$, $M_1 \leq_m M_2$ si et seulement si il existe (x, y) dans $M_1 \times M_2$ tel que $x \leq_P y$. Il s'agit également du sous-ordre induit de P sur l'ensemble constitué d'un seul élément de chaque module. Ainsi, en substituant chaque élément de $P_{/\mathcal{M}}$, appartenant à un module M, par $P[M]$, on obtiendra l'ordre P. L'ordre quotient de l'ordre E est la chaîne définie sur $\mathcal{M} = \{M_1, M_2, M_3\}$ avec $M_1 = \{x\}$, $M_2 = \{a, b, c, d, e, f, g, h\}$ et $M_3 = \{y\}$.

L'ensemble des modules forts générés à partir de la décomposition modulaire de P peuvent être arrangés par inclusion dans un **arbre de décomposition modulaire** dont les feuilles sont les modules triviaux $\{x\}$ pour tout x dans X, et dont chaque sommet interne M, avec M un module non trivial, est étiqueté selon la nature de l'ordre quotient $P[M]$ par : **Série** (cas d'une chaîne), **Parallèle** (cas d'une antichaîne) ou **Premier** (autres cas) (Figure 12 pour une illustration).

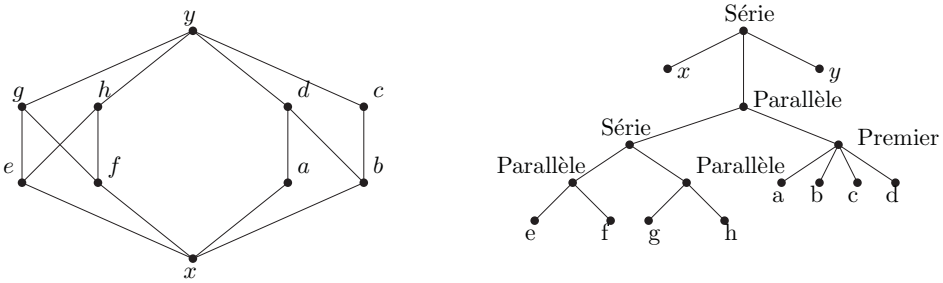


FIGURE 12: Arbre de décomposition modulaire de E

Opérateurs : Sup et Inf

Soit a et b deux éléments d'un ordre P. On a $a \vee b$ renvoie la borne supérieure de la paire (a, b) dans P et $a \wedge b$ renvoie sa borne inférieure. Les opérateurs $\vee$ et $\wedge$ sont appelés respectivement **Sup** et **Inf**. Dans certains livres, notamment le livre [Caspard *et al.*, 2007], les auteurs utilisent ces opérateurs pour introduire quelques notions et notations sur les ordres.

Quelques notions sur les graphes

Un **graphe** G non orienté (resp. orienté) est une paire (V, E), avec V un ensemble d'éléments, appelés **sommets** de G, et $E \subset V \times V$ une relation binaire symétrique (resp. antisymétrique) et irréflexive, désignant les **arêtes** (resp. **arcs**) de G. Cette relation binaire est aussi appelée **relation d'adjacence**. De ce fait, deux éléments x et y sont adjacents dans un graphe G, ou encore **voisins**, s'ils sont en relation binaire dans G. Une arête (resp. arc) reliant x à y, notée $\{x, y\}$ (resp. (x, y)) ou xy tout court, est dite **incidente** à x et à y. Le **degré d'un sommet** dans un graphe est donné par le nombre de ses voisins.

L'ensemble des sommets ainsi que l'ensemble des arêtes d'un graphe G sont généralement notés respectivement par $V(G)$ et $E(G)$.

Un **chemin** dans G est défini par une suite finie de sommets distincts de G, $v_1, \dots v_k$, telle que pour tout i entre 1 et $k-1$, v_i est en relation d'adjacence avec v_{i+1}. Si v_k est en relation d'adjacence avec v_1 et $k \geq 2$ le chemin est appelé **cycle**. Un graphe est dit **acyclique** s'il ne contient pas de cycle. Dans le cas de graphe orienté, on parle de **chemin orienté** et **cycle orienté**.

Un graphe est complet si tous ses sommets sont deux à deux adjacents. Ce graphe est aussi appelé **clique**. De façon complémentaire, on appelle **stable** un graphe dont tous les sommets sont deux à deux non adjacents.

Tout ordre partiel $P = (X, \leq)$ peut être vu comme un graphe orienté acyclique (**DAG**) dont l'ensemble des sommets est X et dont l'ensemble des arcs est formé de toutes les paires (x, y) de X^2 telles que $x \leq y$.

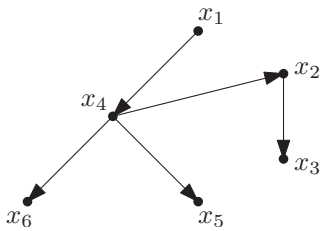


FIGURE 13: Graphe orienté acyclique (DAG)

Autres notions d'ordre général

Nous rappelons ci-après les définitions de quelques termes scientifiques utilisés dans le manuscrit.

Un **algorithme** (du nom du mathématicien al-Khuwarizmi) est une suite d'instructions dont l'exécution résout un problème donné. La **complexité** en temps d'un algorithme est le nombre de ses instructions élémentaires, par ailleurs, sa complexité en espace est la taille mémoire nécessaire pour stocker les différentes structures de données utilisées pendant son exécution. La complexité se réfère généralement au temps si aucune précision n'est indiquée.

La complexité (en temps et en espace) se calcule en fonction de la taille de la donnée en entrée et elle est généralement donnée en ordre de grandeur. Une fonction $f(n)$ a un ordre de grandeur $\mathcal{O}(g(n))$ s'il existe une constante c telle que $\mid f(n) \mid \leq c \mid g(n) \mid$. La Table 1 présente les classes de complexité par ordre de grandeur croissant avec n la taille de la donnée en entrée. Notez que les complexités en temps et en espace d'une instruction élémentaire sont constantes, ce que l'on note $\mathcal{O}(1)$. Lorsque les opérations arithmétiques sur les entiers sont considérées comme des instructions élémentaires, on parlera d'une **complexité arithmétique** [Faigle *et al.*, 2013].

Ordre de grandeur	Classe de complexité
$\mathcal{O}(1)$	Constante
$\mathcal{O}(\log(n))$	Logarithmique
$\mathcal{O}(n)$	Linéaire
$\mathcal{O}(n\log(n))$	Quasi linéaire
$\mathcal{O}(n^k)(k$ constante avec $k > 1)$	Polynomiale
$\mathcal{O}(2^n)$	Exponentielle

TABLE 1: Classes de complexité d'un problème

Un **problème de décision** est un problème dont la solution est soit 'oui' soit 'non'. Par exemple, étant donné un graphe G, répondre s'il existe un cycle dans G de taille inférieure à un certain entier k est un problème de décision.

Un **problème d'optimisation** est un problème dont l'objectif est de trouver une solution optimale parmi l'ensemble des solutions admissibles. Par exemple, trouver un cycle de taille maximale dans un graphe est un problème d'optimisation. Les problèmes d'optimisation généralisent en quelque sorte les problèmes de décision. Par conséquent, pour classifier les problèmes nous pouvons nous limiter aux problèmes de décision.

La **classe P (PTIME)** est la classe des problèmes de décision pour lesquels il existe un algorithme de résolution de complexité en temps polynomiale. Si pour un problème de décision donné il existe un algorithme de résolution de complexité en espace polynomiale, on dit qu'il appartient à la **classe PSPACE**.

Les classes **EXPTIME** et **EXPSPACE** désignent respectivement les problèmes de décision pour lesquels il existe un algorithme de résolution de complexité en temps et en espace exponentielle. Un problème est dans la **classe** $\mathcal{NP}$ si pour une solution donnée (parmi un nombre de solutions potentielles au pire exponentiel), il est possible de vérifier cette solution en temps polynomial. Un problème est **$\mathcal{NP}$-difficile** s'il est au moins aussi difficile que tout problème de $\mathcal{NP}$. Il est **$\mathcal{NP}$-complet** s'il est $\mathcal{NP}$ et $\mathcal{NP}$-difficile. Pour montrer qu'un problème B est au moins aussi difficile que tout problème de $\mathcal{NP}$, il suffit de montrer qu'il existe un problème A de la classe $\mathcal{NP}$-difficile (ou $\mathcal{NP}$-complet) pouvant être réduit en temps polynomial au problème B, i.e. il est possible de résoudre A par un algorithme de résolution de B en passant par une transformation polynomiale des instances (entrées) de A en des instances de B.

Le problème de **calcul du nombre chromatique d'un graphe** est parmi les problèmes classiques de la classe $\mathcal{NP}$-complet. Soit G un graphe, on appelle **nombre chromatique** de G, et on le note $\chi(G)$, le nombre minimum de couleurs nécessaires pour colorer les sommets de G. Une **coloration des sommets** de G consiste en l'attribution à chacun de ses sommets d'une couleur de telle sorte que deux sommets adjacents n'aient pas la même couleur.

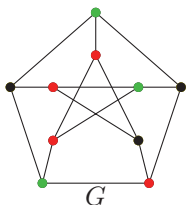


FIGURE 14: Coloration optimale du graphe de *Petersen*

À ce jour, il n'existe pas de problème $\mathcal{NP}$-complet résoluble en temps polynomial. Tous les algorithmes connus de résolution des problèmes $\mathcal{NP}$-complet ont une complexité en temps exponentiel ce qui rend la recherche d'une solution exacte pour de tels problèmes impraticable. Néanmoins, il existe des algorithmes calculant des solutions réalisables, mais pas nécessairement optimales, en un temps raisonnable. Ces algorithmes s'appellent des **heuristiques**. Soit I une instance d'un problème de minimisation et soit Opt sa solution optimale. Soit Sol une solution de I retournée par une heuristique $\mathcal{H}$. Le **facteur d'approximation** de l'heuristique $\mathcal{H}$ est l'entier k vérifiant $\frac{Sol}{Opt} \leq k$, pour toute instance I du problème. Lorsqu'il s'agit d'un problème de maximisation, on a plutôt $k \leq \frac{Sol}{Opt}$. S'il est possible d'identifier l'entier k, l'heuristique est appelée une **k-approximation du problème** ou encore une **méthode d'approximation**. Sinon, on dit que le problème est **non-approximable** .

Chapitre 1

État de l'art autour de la 2-dimension des ordres partiels

> Research is to see what everybody else has seen, and to think what nobody else has thought.
>
> Albert Szent-Gyorgyi

Introduction

La notion d'ordre intervient dans la vie quotidienne, par exemple dans le traitement de l'ordre des clients dans une file d'attente. La structure d'ordre intervient également dans la modélisation des systèmes parallèles [Gazagnaire, 2008] ou concurrents [Pratt, 1986], dans l'analyse des réseaux (sociaux, biochimiques, électriques, urbains, etc.), dans l'extraction des connaissances en fouille de données, dans la résolution des problèmes d'ordonnancement [La, 2005] ainsi que dans la représentation des données hiérarchiques (par exemple la hiérarchie des interfaces (ou des classes) d'un langage de programmation orienté objet, comme illustrée sur la Figure 1.1), etc. [Brüggemann et Patil, 2011].

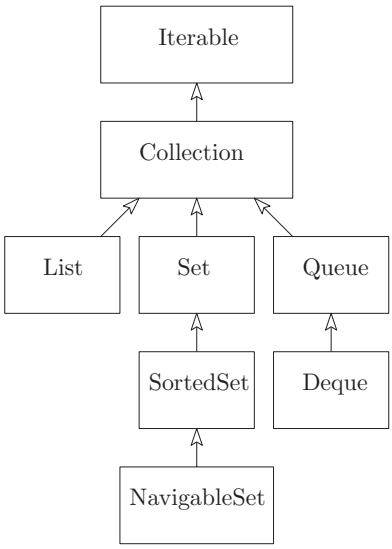


FIGURE 1.1: Hiérarchie de collection Java

Les ordres s'appliquent dans des contextes variés, ce qui montre l'importance de l'étude de leurs structures et de leurs propriétés. Cette étude est ancrée dans la **théorie des ordres**.

« *... Si l'on veut bien imaginer la théorie des ordres comme un fleuve, on lui trouve de nombreuses sources. La théorie des ordres vient en effet de l'arithmétique ordinale de* G.CANTOR *mais aussi des travaux en analyse réelle de* G.CANTOR *et* R.DEDEKIND*, et encore de la théorie des équations et des groupes avec* E.GALOIS *et* C.JORDAN *(groupes résolubles, suites de* JORDAN-HOLDER*) et encore de la théorie des anneaux (*E.NOETHER*), ceci pour ne citer que quelques exemples importants. De ces diverses origines naissent plusieurs courants tels la théorie des treillis, les algèbres de Boole, la topologie, etc., drainant les eaux vers le fleuve en formation.* » [Pouzet et Richard, 1984]

La considération des ordres se manifeste dès 1690 dans les axiomes de G. Leibniz [Leibniz, 1989]. Ce n'est toutefois qu'à partir du XIXe siècle qu'apparurent les premières discussions systématiques autour de ces structures, par C.S. Pierce en 1880, G. Cantor en 1883 puis par E. Schroder en 1890. Des années après, en 1914, F. Hausdorff introduisit la définition formelle des ensembles partiellement ordonnés [Hausdorff, 1914] (version anglaise [Hausdorff, 1957]). Le fameux théorème de Hausdorff établit le *principe de maximalité*.

Théorème 1.0.1 (Principe de maximalité de Hausdorff)**.** *Tout ensemble ordonné non vide contient une chaîne maximale.*

Théorème 1.0.2 (Konstantopoulos, 2010)**.** *Les propriétés suivantes sont équivalentes :*

- *Principe de maximalité de Hausdorff.*
- *Lemme de Tukey : Toute famille finie non vide possède un élément maximal.*
- *Lemme de Zorn : Tout ensemble non vide ordonné dans lequel toute chaîne admet une borne supérieure possède un élément maximal.*
- *Théorème de Zermelo : Tout ensemble peut être bien ordonné, i.e. muni d'un ordre tel que toute partie non vide de cet ensemble admet un plus petit élément.*
- *Principe d'induction mathématique (mis au point par le mathématicien Blaise Pascal).*

Malgré sa simplicité, le théorème de Hausdorff est d'une importance capitale, et ses formes équivalentes trouvent des applications variées. Ainsi, le lemme de Zorn s'applique en topologie, en analyse fonctionnelle, en algèbre (tout espace vectoriel admet une base) [Conrad, 2012], en politique économique [Barbie *et al.*, 2001], voire, par certains aspects, dans la religion [Meyer, 1987]. À l'heure actuelle, T. Powell a mené de nouveaux travaux sur ce lemme présentés lors du workshop international Proof, Computation, Complexity 2016.

Dans ce manuscrit, nous nous intéressons aux ordres partiels finis qui vérifient trivialement le lemme de Zorn. Une conséquence importante de ce lemme est le théorème de Szpilrajn qui montre que tout ordre partiel peut être étendu en un ordre total, appelé son extension linéaire [Szpilrajn, 1930].

Théorème 1.0.3 (Szpilrajn, 1930)**.** *Pour tout ordre partiel $(X, \leq)$, il existe un ordre total $(X, \leq')$ tel que $\leq'$ est une extension linéaire de $\leq$.*

Démonstration. Soit $POSET(X, \leq)$ l'ensemble des extensions de $\leq$ sur X. Nous appliquons le lemme de Zorn sur l'ordre $P = (POSET(X, \leq), \subseteq)$.

Montrons d'abord que toute chaîne de P admet une borne supérieure. Soit C une chaîne de P et $Q = \bigcup_{x \in C} x$. Comme tout élément de C est une relation d'ordre, il est facile de montrer que Q est également une relation d'ordre par héritage des

propriétés de réflexivité, d'antisymétrie et de transitivité. Ceci implique que Q est un élément de P et constitue la borne supérieure de C.

Selon le lemme de Zorn, l'ordre P possède un élément maximal, que l'on note $\leq'$. Supposons que $\leq'$ n'est pas une relation d'ordre totale. Alors, il existe une paire d'éléments (a, b) qui n'appartient pas à $\leq'$, i.e. $a \not\leq' b$ et $b \not\leq' a$.

Considérons la relation binaire $\leq''$ sur X telle que

$$\leq'' = \{(x, y) \in X^2 : x \leq' y\} \cup \{(a, b)\} \cup \{(a, t) : t \in F(b)\} \cup \{(t, b) : t \in I(a)\}.$$

Comme $(X, \leq'') = (X, \leq') \cup ((I(a), \leq').(F(b), \leq'))$ (voir les opérations prédéfinies sur les posets), $\leq''$ est une relation d'ordre sur X et $\leq' \subset \leq''$ ce qui contredit le fait que $\leq'$ soit maximale. Par conséquent, $\leq'$ est une relation d'ordre totale. □

Théorème 1.0.4 (Dushnik et Miller, 1941)**.** *Tout ordre partiel sur un ensemble X est l'intersection d'un ensemble d'ordres linéaires sur X.*

En 1941, Dushnik et Miller formulent le Théorème 1.0.4 qui découle immédiatement du Théorème 1.0.3. La Figure 1.2, qui illustre leur théorème, présente un premier diagramme de Hasse d'un ordre donné $(X, \leq)$ et un second de l'ordre $(POSET(X, \leq), \subseteq)$ défini sur l'ensemble des extensions de $\leq$ muni de la relation d'ordre inclusion. L'ordre $(POSET(X, \leq), \subseteq)$ possède au moins un élément maximal (Lemme de Tukey) qui forme une extension linéaire de $\leq$ (Théorème 1.0.3). Dushnik et Miller constatent que $\leq$ est l'intersection de ses extensions linéaires. Ils définissent alors la *dimension* d'un ordre comme le nombre minimum de ses extensions linéaires dont il est l'intersection. Il est clair qu'un ordre total est de dimension 1, i.e. unidimensionnel.

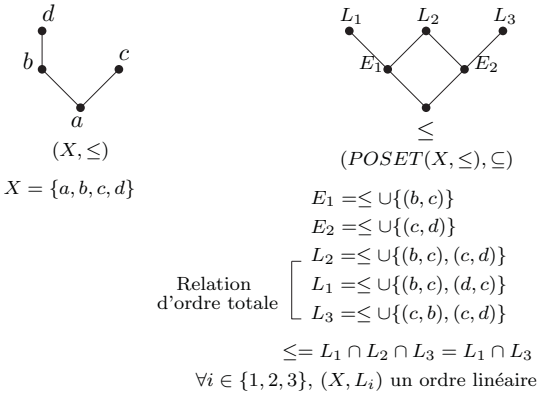


FIGURE 1.2: Illustration des Théorèmes 1.0.3 et 1.0.4

La notion de la dimension d'ordre, introduite par Dushnik et Miller, fut l'un des paramètres les plus étudiés en théorie des ordres, pour des raisons théoriques mais également pour son utilité dans certaines modélisations. Par exemple, en sciences sociales, la modélisation des préférences d'un agent économique sur un ensemble de biens exige la considération d'un certain nombre de critères dont chacun est un ordre total de ses préférences. Le nombre minimum de critères considérés pour modéliser l'ordre des préférences de l'agent constitue la dimension de cet ordre. En informatique, le calcul de la dimension permet de concevoir des méthodes de *codage* des ordres assurant leur *représentation* dans un système informatique.

1 Représentation et codage des ordres partiels

Dans cette section, nous définissons d'abord la notion de représentation et de codage des ordres, puis nous fixons les critères déterminant l'efficacité d'un codage. Nous citons par la suite différentes représentations des ordres ainsi que quelques méthodes de codage existantes permettant d'assurer ces représentations. Nous mettrons en évidence le caractère intuitif des codages cités et nous évaluerons leur efficacité.

1.1 Définitions

1.1.1 Représentation et codage des ordres

Une représentation des données permet de définir un format sous lequel ces données doivent apparaître afin d'être manipulées par un système précis. Lorsqu'il s'agit d'un système informatique, on parle généralement d'une représentation binaire. Ainsi, toute donnée (texte, image, son, vidéo, etc.) transitant par un périphérique numérique, pour être traitée ou enregistrée, doit être représentée sous forme d'une séquence de bits. Un bit – *binary digit* – désigne l'unité d'information manipulable par un composant électronique et ne peut prendre que deux valeurs, 1 ou 0. La valeur 1 signifie que la donnée correspond physiquement à un signal numérique de tension positive ; la valeur 0 indique un signal numérique de tension nulle. Il existe d'autres représentations intermédiaires plus pratiques, comme par exemple la représentation vectorielle des images.

Un codage se définit par la transformation des données d'une représentation externe (initiale) en une autre, interne (cible), suivant des règles bien précises. Dans le cas des données ordonnées, Habib *et al.* définissent formellement le concept de codage d'un ordre $P = (X, \leq_P)$, par le plongement f de X dans un ensemble Y, tel que $x \leq_P y$, x et y étant deux éléments de X, si et seulement si $\rho(f(x), f(y)) = 1$ avec $\rho : Y \times Y \to \{0, 1\}$ [Habib *et al.*, 1995].

Sans restreindre la généralité, nous admettons que les ordres sont initialement représentés par une paire d'éléments (X, R), avec X l'ensemble des entiers $\{1, \ldots, n\}$, n étant la taille de l'ordre, et R l'ensemble des couples (i, j) de X^2 tels que i est en relation d'ordre avec j. Dans la suite, nous parlons également de méthode de codage pour désigner l'ensemble des instructions et des structures de données nécessaires pour calculer un codage.

1.1.2 Critères d'évaluation d'un codage

L'efficacité d'un codage est évaluée à partir de plusieurs critères : (1) la taille du codage, définie par le nombre de bits codant les données, ce qui permet de quantifier l'espace mémoire dédié à leur stockage ; (2) les complexités en temps et en espace de la méthode de codage ; (3) le coût d'interrogation des données après leur codage ; (4) le coût de l'adaptation du codage aux différentes modifications potentielles apportées aux données ; et d'autres critères pouvant se rajouter selon le type des données traitées. Dans ce manuscrit, nous ne considérons que les trois premiers critères. Comme nous nous intéressons au codage des ordres, nous limitons les opérations d'interrogation à la requête « $x \leq y$? », pour x et y deux éléments d'un ordre $P = (X, \leq)$; d'autre part, la taille du codage est limitée au nombre de bits maximal codant un élément de P. Ainsi, s'il existe un codage de P de taille e alors le stockage de P nécessite au plus $e|X|$ bits.

1.1.3 Codage efficace des ordres

Un codage est jugé d'autant plus intéressant que les mesures de ses critères d'évaluation sont petites comparées aux mesures évaluant les autres codages existants. Nous considérons qu'il est efficace pour le cas d'un ordre $P = (X, R)$ si : (1) sa taille est en $\mathcal{O}(\log_2(|X|))$; (2) il teste la relation d'ordre entre deux éléments de P en temps linéaire par rapport à la taille du codage ; (3) il existe un algorithme de complexité arithmétique polynomial (en temps et en espace) qui le calcule. Dans la suite, nous exprimons la taille du codage en nombre de bits plutôt qu'en ordre de grandeur lorsqu'il s'agit de mesurer l'efficacité du codage d'un point de vue pratique.

Une étude du IDC (*International Data Corporation*) en 2014 estime que le volume des données numériques sera multiplié par 10 à l'horizon 2020, ce qui engendrera des besoins importants en terme d'efficacité du traitement et du stockage des données. On peut se demander si cela mettra fin à la conjecture de Moore, parue à la revue *Electronics Magazine* en 1965, qui affirme que la capacité de calcul et de stockage d'information double tous les deux ans. Dernièrement, Brian Krzanich, le patron actuel d'Intel, a lancé un projet nommé "More than Moore" s'inscrivant dans les tendances actuelles du Big Data face à la croissance exponentielle des

données [Loukil, 2016]. La problématique discutée dans ce manuscrit pourra également s'inscrire dans ce cadre. En effet, nous nous intéressons aux méthodes de codage des ordres minimisant à la fois leur espace de stockage et le coût de comparaison de leurs objets. Avant d'exposer nos contributions à ce sujet, citons certaines méthodes de codage connues dans la littérature.

1.2 Représentations classiques

Les premières méthodes de codage des ordres s'appuient sur des représentations connues en théorie des graphes, celles des graphes orientés acycliques (DAG), vu que tout ordre partiel strict est isomorphe à un DAG. Il s'agit d'une représentation par *matrice d'adjacence* ou par *liste d'adjacence*.

1.2.1 Matrice d'adjacence

La représentation d'un ordre partiel P de taille n par une *matrice d'adjacence* consiste à projeter sa relation d'ordre dans une matrice binaire de taille n^2, de telle sorte que l'élément à la case (i, j) de la matrice vaut 1 si l'élément i est en relation avec l'élément j dans P, et 0 sinon. Ainsi, déterminer si un élément x est en relation d'ordre avec un élément y nécessite un accès direct à la case (x, y) en temps constant. La taille de ce codage vaut n. L'algorithme 11 (cf. Appendice A) calcule une représentation de P par une matrice binaire en temps et en espace $\mathcal{O}(n^2)$. Cette représentation est utilisée pour le codage de la hiérarchie des interfaces JAVA dans CACAO 64-bits JIT Compiler [Krall et Grafl, 1997].

1.2.2 Liste d'adjacence

Une représentation par *liste d'adjacence*, aussi appelée représentation par *liste des successeurs*, consiste à associer à chaque élément d'un ordre une liste de ses successeurs. Soit $P = (X, R)$ un ordre de taille n avec m le nombre maximum des successeurs d'un élément de P. La taille de ce codage vaut alors $m\lceil\log_2(n)\rceil$. Pour x et y deux éléments de X, tester si $x \ R \ y$ revient à tester si y appartient à la liste des successeurs de x en temps $\mathcal{O}(m\lceil\log_2(n)\rceil)$. La complexité en temps du test de comparabilité peut être réduite en $\mathcal{O}(\log_2(m)\lceil\log_2(n)\rceil)$, en utilisant la structure d'arbre équilibré pour stocker les listes des successeurs ; elle peut être réduite davantage en $\mathcal{O}(\sqrt{\log_2(m)}\lceil\log_2(n)\rceil)$ *via* les Q-fast tries proposés par Willard [Willard, 1984] ; et encore plus en $\mathcal{O}(\log_2(\log_2(m))\lceil\log_2(n)\rceil)$ grâce aux arbres stratifiés de van Emde Boas [Van Emde Boas, 1977]. L'algorithme 12 (cf. Appendice A) effectue une représentation par *liste d'adjacence* de P en temps et en espace $\mathcal{O}(|R|)$, avec $|R| \leq n^2$. De façon duale, nous pourrons aussi définir une représentation par *liste des prédécesseurs*.

Notons que la représentation par *liste d'adjacence* offre un codage plus compact des ordres creux, dont les éléments sont peu en relation d'ordre, contrairement à une représentation par *matrice d'adjacence* qui est surtout privilégiée pour les ordres denses.

1.2.3 Liste des successeurs immédiats

Le codage d'un ordre par *liste des successeurs immédiats* (dualement par *liste des prédécesseurs immédiats*) consiste à associer à chaque élément de l'ordre la liste de ses successeurs immédiats qui sont ses successeurs non déduits par transitivité. La taille de ce codage est alors inférieure à celle du codage par liste des successeurs. Soit $P = (X, R)$ un ordre de taille n et soit x et y deux éléments de X. Tester si x est en relation d'ordre avec y implique de parcourir la liste des successeurs immédiats de x, puis leurs propres successeurs immédiats, et ainsi de suite jusqu'à rencontrer l'élément y ou visiter tous les successeurs de x. Ce test s'effectue en temps $\mathcal{O}(nm'\lceil\log_2(n)\rceil)$ avec $m' = \max\{d_{out}(x)|x \in X\}$. Le codage par *liste des successeurs immédiats* est souvent utilisé lorsque l'ordre donné en entrée est réduit par transitivité. L'algorithme 13 (cf. Appendice A) calcule un codage par *liste des successeurs immédiats* de P en temps et en espace $\mathcal{O}(|R|)$ pour le cas où P est réduit par transitivité. Sinon, une étape de réduction transitive devrait se rajouter à l'algorithme, ce qui augmenterait sa complexité en temps et espace en $\mathcal{O}(n^3)$.

Bien que le codage d'un ordre par une matrice binaire soit optimal (minimal) pour la comparaison de ses objets, il est en contrepartie de taille maximale. Le codage par *liste des successeurs* (*immédiats*), lui, améliore la taille du codage des ordres au profit du temps de comparaison de leurs objets. Le but est de trouver un compromis entre ces deux paramètres.

1.3 Représentation intervallaire

Nous présentons dans cette sous-section des méthodes de codage des ordres plus efficaces que celles basées sur les représentations classiques. Nous montrons d'abord comment déduire, à partir d'une représentation classique des arbres, une nouvelle représentation plus compact pour cette classe d'ordre. Il s'agit de la représentation intervallaire, donnant lieu à un codage efficace des arbres ainsi que des ordres d'intervalles. Nous citons par la suite quelques méthodes de codage par intervalle(s) permettant d'assurer une représentation intervallaire de tout ordre.

1.3.1 Définition

Une représentation intervallaire d'un ordre indique que chacun de ses éléments doit être représenté par des intervalles de $\mathbb{R}$. Afin d'assurer cette représentation pour un ordre P, il faut trouver une méthode de codage qui associe à chacun de ses éléments x un code I_x, sous forme d'un ou plusieurs (liste ou union) intervalle(s) de $\mathbb{R}$, tel(s) qu'il existe une relation d'ordre R vérifiant, pour tous x et y deux éléments de P, $x \leq_P y$ si et seulement si $I_x \; R \; I_y$.

1.3.2 Origine de la représentation intervallaire

La représentation intervallaire est directement issue d'une représentation classique des arbres. Soit T un arbre de taille n tel que son élément i coïncide avec le $i^{\text{ème}}$ élément visité suivant un parcours préfixe de T (Figure 1.3). Rappelons que le parcours préfixe d'un arbre consiste à visiter sa racine puis à parcourir récursivement tous ses sous-arbres, enracinés par un successeur immédiat de la racine, de gauche à droite. Il s'agit d'un parcours en profondeur lorsque l'arbre est représenté graphiquement par un diagramme de Hasse orienté du haut vers le bas.

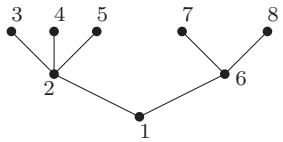
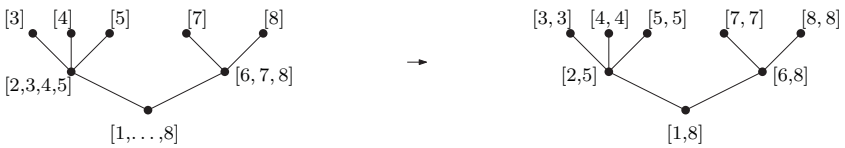


FIGURE 1.3: Les éléments d'un arbre T en ordre préfixe

Afin de calculer une représentation de T par *liste des successeurs*, nous optons pour la récupération de la liste des successeurs de chaque élément suivant un parcours préfixe de T. En procédant ainsi, nous remarquons que la liste des successeurs de chaque élément i de l'arbre forme un ensemble d'entiers consécutifs $i, i+1, \ldots k$, qui peut être représenté efficacement par l'intervalle d'entiers $[i, k]$. En affectant à chaque élément de T un intervalle d'entiers englobant la liste de ses successeurs, nous obtenons une représentation intervallaire de l'arbre, illustrée par la Figure 1.4.



Codage de T par liste des successeurs — Codage intervallaire de T

FIGURE 1.4: D'une représentation classique à une représentation intervallaire

En calculant une représentation par *matrice d'adjacence* de T, nous obtenons une matrice binaire particulière dont chaque ligne l_x est un vecteur de la forme $[0]^r.[1]^s.[0]^t$, avec $r + s + t = n$ et s le nombre des successeurs de l'élément x, plus x. Nous en déduisons que les successeurs de x sont les éléments de l'intervalle $[r + 1, r + s]$. Cela mènera également à une représentation intervallaire de T (Figure 1.5).

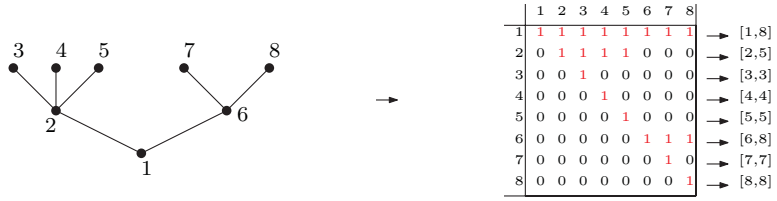


FIGURE 1.5: *Matrice d'adjacence* d'un arbre dont les éléments sont en ordre préfixe

1.3.3 Codage par intervalle unique

Le codage d'un ordre par *intervalle unique* consiste à associer à chacun de ses éléments un intervalle de $\mathbb{R}$ tel qu'il existe une relation d'ordre entre l'ensemble des intervalles codant les éléments de l'ordre qui préserve la relation entre ces éléments dans l'ordre. Nous montrons que ce codage est bien défini dans le cas des arbres et des ordres d'intervalles, puis nous mettons en évidence la difficulté d'assurer un tel codage pour l'ordre  .

▶ Cas des arbres

Soit $T = (X, R)$ un arbre de taille n. Nous admettons que l'élément i de T coïncide avec le $i^{\text{ème}}$ élément visité suivant un parcours préfixe de T (cf. Figure 1.3). Notons qu'il est possible de renommer les éléments de T afin de vérifier cette condition. Le codage MPTT (*Modified Preorder Tree Traversal*) de T consiste à associer à chacun de ses éléments x un intervalle $[a_x, b_x]$ tel que a_x (resp. b_x) représente le plus petit (resp. le plus grand) entier référant à un successeur de x.

Théorème 1.1.5. *Le codage par intervalle unique des arbres est efficace.*

Le codage MPTT de T est calculable en temps et en espace polynomiaux, et sa taille vaut exactement $2\lceil \log_2(n) \rceil$. Ainsi, afin de tester la relation d'ordre entre deux éléments x et y de X, il suffit de tester si $a_y \leq a_x$ et $b_x \leq b_y$. Ce test est vérifié en temps $\mathcal{O}(\lceil \log_2(n) \rceil)$. Nous en déduisons que le codage des arbres par intervalle est efficace (Théorème 1.1.5).

Avantage

La représentation intervallaire est une technique très utilisée dans les bases de données relationnelles pour représenter les hiérarchies de type arborescence – arbre tout simplement – dans le but de simplifier les requêtes de recherche et d'améliorer leurs performances. Pour illustrer l'avantage de cette représentation, nous traitons l'exemple de stockage de la hiérarchie des formations à l'Université Blaise Pascal, donnée dans la Figure 1.6.

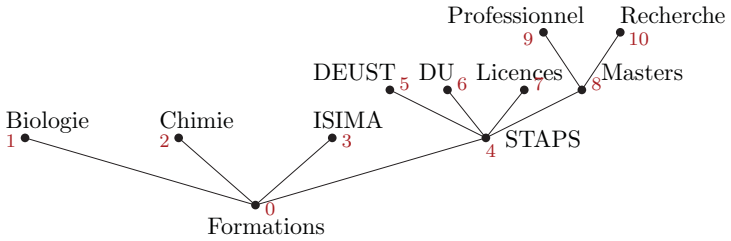


FIGURE 1.6: Hiérarchie des formations à l'UBP en 2016

Le stockage classique de cette hiérarchie dans une base de données relationnelle, comme MySQL, fait appel à une représentation par *liste des prédécesseurs immédiats* (Table 1.1).

Id	Nom	Parent_id
0	Formations	NULL
1	Biologie	0
2	Chimie	0
3	ISIMA	0
4	STAPS	0
5	DEUST	4
6	DU	4
7	Licences	4
8	Masters	4
9	Professionnel	8
10	Recherche	8

TABLE 1.1: Stockage classique de la hiérarchie des formations

Cette méthode de stockage n'est toutefois pas pratique lorsqu'il s'agit de récupérer une sous-arborescence de la hiérarchie. Prenons l'exemple de l'extraction de toutes les formations du STAPS. Deux manières de procéder sont possibles :

1. Requêtes récursives :

   `Select Nom from TableFormations where Parent_id` $= 4$

   `Select Nom from TableFormations where Parent_id IN (Select Id from TableFormations where Parent_id` $= 4$`)`

2. Requête par auto-jointures :

   `Select` F_1`.Nom from TableFormations as` F_1 `join TableFormations as` F_2 `on` F_1`.Parent_id =` F_2`.Id join TableFormations as` F_3 `on` F_2`.Parent_id =` F_3`.Id where` F_2`.Nom = 'STAPS' or` F_3`.Nom = 'STAPS'`

Les deux requêtes d'extraction d'une sous-hiérarchie de l'arborescence initiale sont coûteuses en temps et en espace de mémoire vive. Ces problèmes de performances sont réglés par la représentation intervallaire. La Table 1.2 stocke la hiérarchie des formations après son codage MPTT. Notons que les successeurs d'un élément de la hiérarchie sont compris entre `Left` et `Right`.

Id	Nom	Left	Right
0	Formations	0	10
1	Biologie	1	1
2	Chimie	2	2
3	ISIMA	3	3
4	STAPS	4	10
5	DEUST	5	5
6	DU	6	6
7	Licences	7	7
8	Masters	8	10
9	Professionnel	9	9
10	Recherche	10	10

TABLE 1.2: Stockage de la hiérarchie des formations après son codage MPTT

Grâce à la représentation intervallaire de la hiérarchie des formations, il est possible, par une simple requête, d'extraire toutes les formations du STAPS :

`Select Nom from TableRIFormations where` $Left \geq 4$ `and` $Right \leq 10$`.`

La représentation intervallaire des arbres permet non seulement d'accélérer la recherche dans des bases de données relationnelles, mais également de faciliter la création de menus dynamiques dans des pages web, des applications mobiles ou des logiciels.

▶ Cas des ordres d'intervalles

Un ordre d'intervalle $P = (X, <)$ est un ordre tel que, x,y,z et w étant des éléments de X, $x < y$ et $z < w$ impliquent $x < w$ ou $z < y$.

Théorème 1.1.6 (Fishburn, 1970). *Un ordre P est un ordre d'intervalle si et seulement si il ne contient pas un sous-ordre isomorphe à l'ordre* :: .

Trotter définit P comme un ordre d'intervalle s'il existe une fonction f associant à chacun de ses éléments x un intervalle $I_x = [l_x, r_x]$ de $\mathbb{R}$ tel que pour toute paire d'éléments x et y de P, $x < y$ si et seulement si $r_x <_{\mathbb{R}} l_y$, i.e. pour tout élément a de I_x et b de I_y, $a <_{\mathbb{R}} b$ [Trotter, 1997]. Plusieurs approches sont présentes dans la littérature pour le calcul de la fonction f [Baldy et Morvan, 1993], [Greenough, 1976]. Nous choisissons l'approche fondée sur l'algorithme de Greenough qui se base sur l'équivalence de ces trois propriétés [Rabinovich, 2005] :

1. $P = (X, <)$ est un ordre d'intervalle ;
2. $U = (\{Pred_P(x) | x \in X\}, \subset)$ est un ordre total ;
3. $S = (\{Succ_P(x) | x \in X\}, \subset)$ est un ordre total.

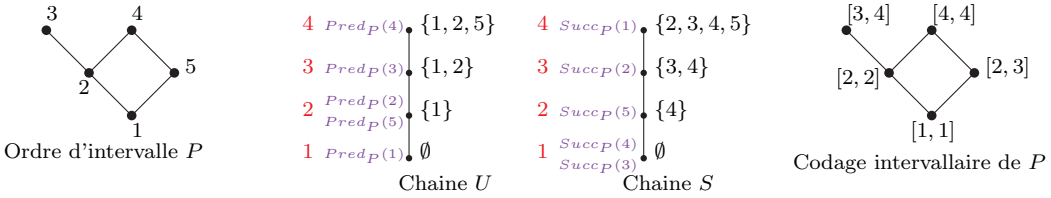


FIGURE 1.7: Codage intervallaire des ordres d'intervalles

L'algorithme de Greenough calcule d'abord les ordres U et S puis associe à chaque élément x de P l'intervalle $[U[x], |S| - S[x] + 1]$, $U[x]$ étant l'indice de l'élément $Pred_P(x)$ dans la chaîne U. De même, $S[x]$ est l'indice de $Succ_P(x)$ dans S. Nous supposons que les indices commencent au rang 1.

Théorème 1.1.7. *Le codage des ordres d'intervalles par intervalle unique est efficace.*

Nous concluons que l'approche de Greenough permet de calculer un codage de P par *intervalle unique* (cf. Figure 1.7) en temps et en espace polynomiaux. La taille de ce codage vaut exactement $2\log_2(|X|)$. Enfin, pour tester si $x < y$, x et y étant deux éléments de P, il suffit de tester si $r_x <_{\mathbb{R}} l_y$ en temps $\mathcal{O}(\log_2(|X|))$. Par conséquent, le codage des ordres d'intervalles par *intervalle unique* est efficace.

Avantage

Les ordres d'intervalles permettent de modéliser des problèmes d'ordonnancement et leur codage permet non seulement de mieux les stocker et les exploiter, mais également de répondre à certaines questions logistiques, comme le nombre de ressources nécessaires pour une planification précise. Considérons par exemple le problème de l'ordonnancement des présentations lors d'une journée scientifique. Supposons que certaines présentations ne peuvent démarrer qu'à la condition que d'autres sont bien terminées, ainsi qu'il existe des présentations pouvant avoir lieu en même temps. L'ordre modélisant la relation temporelle entre les présentations de cette journée est un ordre d'intervalle. Ainsi, le codage de cet ordre par *intervalle unique* permet d'analyser certaines contraintes, comme le nombre d'amphithéâtres nécessaires pour réaliser l'ensemble de ces présentations. En effet, toutes les présentations ayant un code intervallaire non chevauchant peuvent se dérouler dans le même lieu. Le lecteur pourra se référer au [Yannakakis et Papadimitriou, 1979] pour plus de détails sur l'intérêt de la structure d'ordre d'intervalle en ordonnancement. Cette structure a également des applications en archéologie [Kendall, 1969], en diagnostic médical [Nökel, 1991], en génétique [Benzer, 1959], en biologie [Karp, 1993], en psychologie [Coombs et Smith, 1973], etc..

▶ Cas de l'ordre 

Nous avons évoqué, dans ce qui précède, deux méthodes de codage des ordres par *intervalle unique* : le codage MPTT et le codage de Greenough. Rappelons que ces deux méthodes consistent à associer à chaque élément x d'un ordre P un intervalle I_x de $\mathbb{R}$ tel que $x \leq_P y$ correspond soit à $I_y \subseteq I_x$, dans le cas d'un codage MPTT, soit à $a <_{\mathbb{R}} b$ pour toute paire d'éléments a de I_x et b de I_y, dans le cas du codage de Greenough. Toutefois, ces deux méthodes ne codent pas l'ordre  car il n'est ni un arbre ni un ordre d'intervalle. De plus, il n'existe aucune affectation possible d'intervalles aux éléments de cet ordre telle que deux éléments soient en relation si et seulement si leurs intervalles s'intersectent (partiellement ou totalement) ou ne chevauchent pas. La Figure 1.8 illustre différentes tentatives de codage de l'ordre  par *intervalle unique*. Nous présentons dans la suite deux méthodes de codage des ordres, inspirées du codage MPTT et de celui de Greenough, permettant de coder l'ordre  .

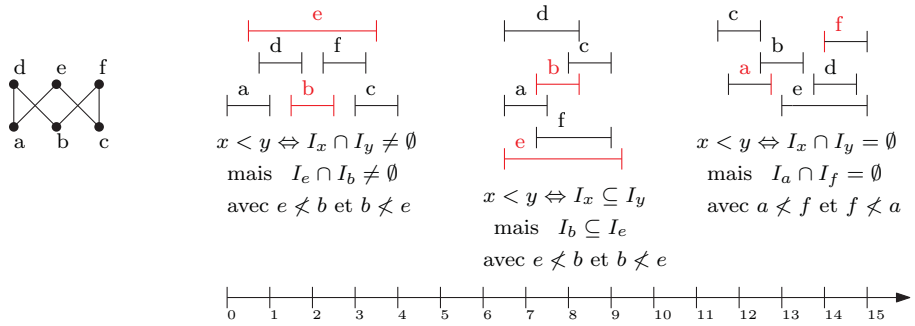


FIGURE 1.8: Tentatives de codage de l'ordre  par *intervalle unique*

1.3.4 Codage par compression de fermeture transitive

Nous avons montré précédemment que le codage par intervalle MPTT offre un codage efficace pour la classe des arbres. Nous présentons ici une méthode de codage, inspirée de la technique du codage MPTT, permettant d'assurer une représentation intervallaire pour tout ordre. Il s'agit du codage par *compression de fermeture transitive* de Agrawal *et al.* qui repose sur un codage MPTT d'un arbre couvrant d'un ordre [Agrawal *et al.*, 1989].

Soit P un ordre de taille n. Nous supposons que P contient un élément minimum. Le codage de P par *compression de fermeture transitive* se déroule en trois étapes. D'abord, il s'agit de calculer un arbre couvrant de P, puis de le coder par la méthode MPTT, et enfin, suivant un ordre topologique inverse des éléments de P, d'associer à chacun de ses éléments x l'ensemble $\cup_{x \leq y} I_y$, I_y étant l'intervalle attribué à y suite au codage de l'arbre couvrant de P. Notons que l'ensemble d'intervalles associé à x pourra être réduit en supprimant tout intervalle subsumé par un autre dans l'ensemble.

Le codage par *compression de fermeture transitive* assure une représentation intervallaire de P telle que, x et y étant deux éléments de P, x est en relation d'ordre avec y si et seulement si l'union des intervalles associée à y est incluse dans celle associée à x. La taille de ce codage dépend de l'arbre couvrant considéré. Sa valeur la moins satisfaisante est $2n\lceil \log_2(n) \rceil$. Agrawal *et al.* proposent un algorithme calculant un arbre couvrant optimum, dans le sens où le nombre total d'intervalles associés à chaque élément de l'arbre soit minimum, en temps $\mathcal{O}(n^2)$ [Agrawal *et al.*, 1989]. Nous considérons alors cet arbre couvrant optimum pour calculer un codage de tout ordre par *compression de fermeture transitive* en temps et en espace $\mathcal{O}(n^2)$ (cf. l'Algorithme 14 en Appendice A). La Figure 1.9 illustre le codage de l'ordre  par *compression de fermeture transitive*.

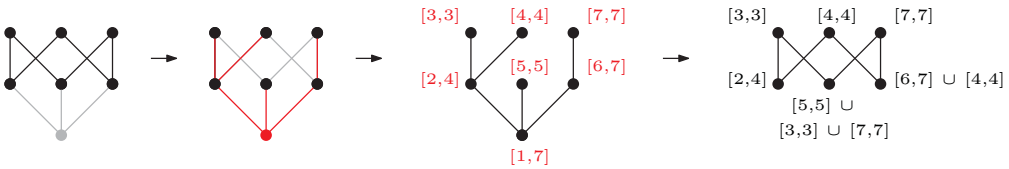


FIGURE 1.9: *Codage par compression de fermeture transitive* de l'ordre

1.3.5 Codage par union des ordres d'intervalles

Le codage par *intervalle unique* offre un codage efficace, non seulement pour les arbres, mais aussi pour la classe des ordres d'intervalles. Nous présentons ici une méthode de codage utilisant le codage par intervalle de cette classe d'ordres (codage de Greenough), afin d'assurer une représentation intervallaire pour tout ordre. Il s'agit du codage par *union des ordres d'intervalles* proposé par Capelle [Capelle, 1994].

Soit $P = (X, R)$ un ordre de taille n, R étant une relation d'ordre fermée par transitivité. L'approche de Capelle consiste à partitionner R en $R_1, \dots R_k$ tel que l'ordre (X, R_i), pour i entre 1 et k, est un ordre d'intervalle. Ensuite, elle associe à chaque élément x de l'ordre une liste l_x de triplets (i, l, r), $[l, r]$ étant le code de x dans une représentation intervallaire de (X, R_i) assurée par l'algorithme de Greenough. Cette liste est ordonnée suivant le premier paramètre du triplet.

Le codage par *union des ordres d'intervalles* assure une représentation intervallaire de P telle que, étant donnés deux éléments x et y de P, $x\ R\ y$ si et seulement si il existe un entier i tel que les triplets (i, a_x, b_x) de l_x et (i, a_y, b_y) de l_y vérifient $b_x < a_y$. Ainsi, tester si $x\ R\ y$ revient à vérifier l'existence de cet entier i en temps $\mathcal{O}(n\lceil\log_2(n)\rceil)$. Notons que la taille de ce codage vaut au plus $3n\lceil\log_2(n)\rceil$. L'algorithme 15 (cf. Appendice A) effectue un codage par *union des ordres d'intervalles* en temps et en espace $\mathcal{O}(|X|(|X| + |R|))$ (voir la version détaillée de l'algorithme dans [Capelle, 1994]). La Figure 1.10 illustre le codage de l'ordre par *union des ordres d'intervalles*.

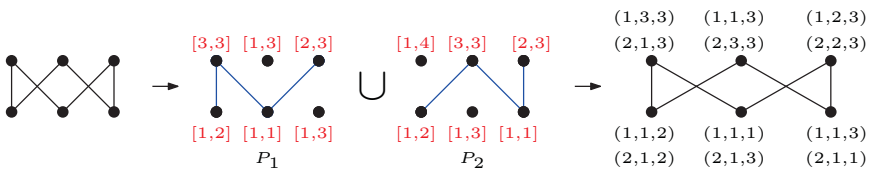


FIGURE 1.10: Codage par *union des ordres d'intervalles* de l'ordre

1.3.6 Codage par intervalles lié au calcul de $i(P)$

Un codage par *intervalle unique* n'étant pas évident à définir pour tout ordre, nous nous orientons vers des codages par liste d'intervalles ou par union d'intervalles afin d'assurer une représentation intervallaire de tout ordre. Les deux dernières méthodes de codage des ordres que nous venons de citer reposent précisément sur de tels codages. Nous citons dans la suite d'autres méthodes de codage par union d'intervalles.

Le paramètre $i(P)$ [Madej et West, 1991] Madej et West définissent le nombre d'inclusions d'intervalles d'un ordre P – ou, tout simplement, le nombre d'intervalles – comme le plus petit entier k tel qu'il est possible de coder chaque élément x de P par l'union d'au plus k intervalles de l'ensemble $\mathbb{R}$, notée I_x, avec $x \leq_P y$ correspond à $I_x \subseteq I_y$. Cet entier, noté $i(P)$, est bien défini pour tout ordre. Il suffit de coder chaque élément x de P par $\cup\{[u,u]|u \leq x\}$.

Étant donné un entier k, décider si $i(P) \leq k$ est un problème $\mathcal{NP}$-complet [Madej et West, 1991]. Néanmoins, Madej et West établissent la valeur exacte de ce paramètre pour certaines classes d'ordres comme les treillis booléens et les couronnes. Ainsi, $i(\mathcal{B}_n) = \lceil \frac{n}{2} \rceil$ pour tout entier n et $i(S_n) = 2$ pour tout entier $n \geq 3$.

Le *codage par intervalles lié au calcul de* $i(P)$ consiste à coder chaque élément x d'un ordre P par I_x (union d'intervalles de $\mathbb{R}$) telle que, x et y étant deux éléments de P, $x \leq_P y$ si et seulement si $I_x \subseteq I_y$. Le nombre maximum d'intervalles dont l'union code un élément de P établit une borne supérieure de $i(P)$.

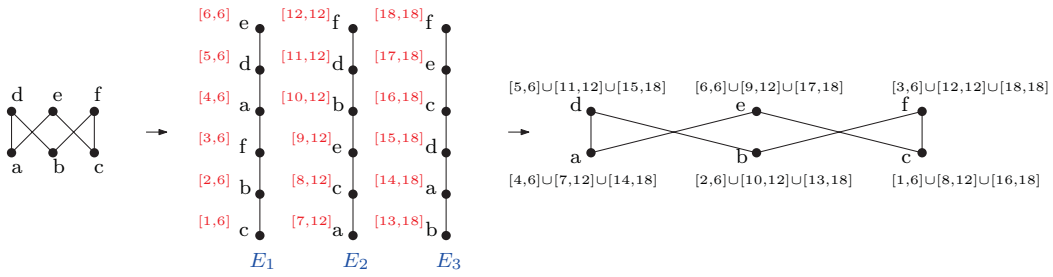
Nous citons dans la suite des méthodes de codage par *intervalles liées au calcul de* $i(P)$ permettant d'assurer une représentation intervallaire de P.

▶ Codage par intersection d'extensions linéaires

Soit $P = (X, R)$ un ordre partiel de taille n. D'après le Théorème 1.0.4 cité précédemment, il est possible de définir P comme l'intersection d'un ensemble d'ordres linéaires sur X. La taille minimale de cet ensemble correspond à la dimension de P.

La dimension d'un ordre [Dushnik et Miller, 1941] Introduite par Dushnik et Miller, la dimension d'un ordre P est le nombre minimum de ses extensions linéaires $E_1, \ldots E_k$ tel que $P = \cap_{i=1}^{k} E_i$. La dimension d'un ordre P est notée $dim(P)$. Pour tout entier k, décider si $dim(P) \leq k$ est un problème $\mathcal{NP}$-complet [Yannakakis, 1982].

Description du codage Soient $E_1, \ldots, E_k$ des extensions linéaires de P, avec $k = dim(P)$ et $P = \cap_{i=1}^{k} E_i$. Notons que les ordres linéaires sont à la fois des arbres et des ordres d'intervalles. Ainsi, en codant chacune des extensions linéaires de P par *intervalle unique* (codage MPTT), il est possible de déduire un codage de P par *union d'intervalles* qui, à chaque élément x de P, associe l'union des intervalles $V_x^1, \ldots V_x^k$, V_x^i étant l'intervalle attribué à x lors d'un codage de E_i par *intervalle unique*. Ainsi, $x \leq_P y$ si et seulement si $\cup_{i=1}^{k} V_y^i$ est incluse dans $\cup_{i=1}^{k} V_x^i$. Par conséquent, $i(P) \leq dim(P)$. La taille de ce codage, calculable en temps polynomial, est de $2dim(P)\lceil \log_2(k.n) \rceil$. La Figure 1.11 illustre ce type de codage sur l'ordre dont la dimension est 3.



Application du codage par intervalle unique
(Algorithme de Greenough)

FIGURE 1.11: Codage par *intersection d'extensions linéaires* de l'ordre

▶ Codage par intersection d'extensions intervallaires

Soit $P = (X, R)$ un ordre partiel de taille n. Une extension intervallaire de P est l'ordre d'intervalle (X, R'), R' étant une extension de la relation d'ordre R. Tout ordre linéaire étant un ordre d'intervalle, nous pouvons déduire du Théorème 1.0.4 que tout ordre s'exprime par l'intersection de ses extensions intervallaires.

La dimension intervallaire d'un ordre Soit P un ordre et $I_1, \ldots I_k$ ses extensions intervallaires. La dimension intervallaire de P, notée $idim(P)$, se définit en fonction du plus petit entier k tel que $P = \cap_{i=1}^{k} I_i$. Comme toute extension linéaire d'un ordre est une extension intervallaire de ce dernier, $idim(P) \leq dim(P)$. Yannakakis prouve que décider si $idim(P) \leq 3$ est un problème $\mathcal{NP}$-complet [Yannakakis, 1982]. Cela justifie la difficulté du calcul de $idim(P)$. La Figure 1.12 illustre toutes les extensions de l'ordre , et en particulier ses extensions intervallaires non linéaires (indiquées en rouge).

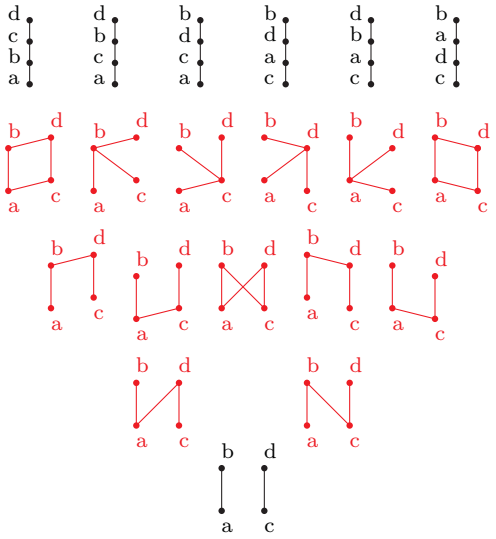


FIGURE 1.12: Les extensions de d'ordre 

Description du codage Soient $I_1, \ldots, I_k$ des extensions intervallaires de P, avec $k = idim(P)$ et $P = \cap_{i=1}^{k} I_i$. Ayant montré que pour tout ordre d'intervalle I, $i(I) \leq 2$, Madej et West en déduisent que si $idim(P) \leq 2$ alors $i(P) \leq 3$ [Madej et West, 1991]. Ces deux inégalités sont illustrées par la Figure 1.13 qui présente un codage de l'ordre  déduit du codage de ses deux extensions intervallaires. Madej et West constatent que tout ordre P peut être exprimé par l'intersection des ordres de dimension intervallaire au plus 2. En effet, $P = \cap_{i=1}^{k} I_i = (I_1 \cap I_{\lceil \frac{k}{2} \rceil + 1}) \cap (I_2 \cap I_{\lceil \frac{k}{2} \rceil + 2}) \cap \cdots \cap (I_{\lceil \frac{k}{2} \rceil} \cap I_k)$. Ainsi, en codant chacun des ordres $I_i \cap I_{\lceil \frac{k}{2} \rceil + i}$, pour i entre 1 et $\lceil \frac{idim(P)}{2} \rceil$, par l'union d'au plus trois intervalles (comme illustré sur la Figure 1.13), il est possible d'obtenir un codage de P par l'union d'au plus $\lceil 3\frac{idim(P)}{2} \rceil$ intervalles tel que $x \leq_P y$, x et y étant deux éléments de P, si et seulement si l'union des intervalles codant x est incluse dans celle codant y. C'est ainsi que Madej et West prouvent que $i(P) \leq \lceil 3\frac{idim(P)}{2} \rceil$. Le codage de l'ordre  par *intersection d'extensions intervallaires* utilise au plus 5 intervalles pour coder chacun des éléments de cet ordre.

▶ Codage par intersection d'ordres 2-dimensionnels

Dans cette partie 1.3.6, nous avons étudié des codages des ordres par *union d'intervalles liés au calcul du paramètre* $i(P)$, permettant de borner la valeur de $i(P)$. L'idéal serait de trouver un codage vérifiant $i(P) = 1$, pour tout ordre P.

Proposition 1.1.1 (Dushnik et Miller, 1941)**.** *Pour tout ordre* P, $i(P) = 1$ *si et seulement si* $dim(P) \leq 2$.

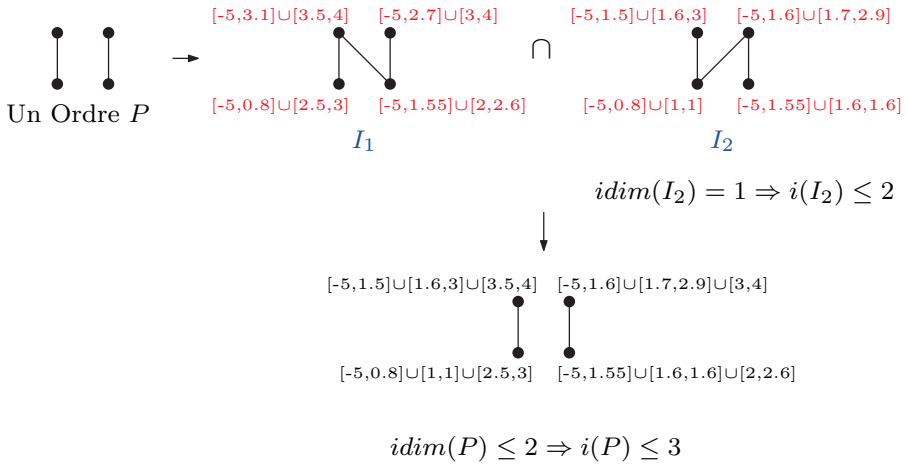


FIGURE 1.13: Codage par *intersection d'extensions intervallaires*

Dushnik et Miller prouvent que seuls les ordres 2-dimensionnels se caractérisent par un nombre d'intervalles unitaire (Proposition 1.1.1). Leur preuve est illustrée par la Figure 1.14, qui propose un codage d'un ordre 2-dimensionnel P vérifiant $i(P) = 1$. Ainsi, il n'existe pas de codage intervallaire pour lequel $i(P) = 1$ pour tout ordre P.

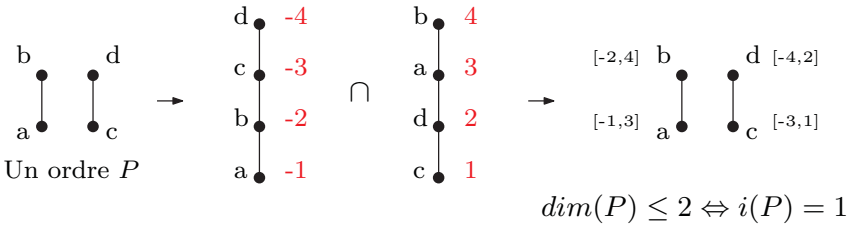


FIGURE 1.14: Codage d'un ordre 2-dimensionnel par *intervalle unique*

Description du codage Soit P un ordre et soient $E_1, \ldots, E_k$ des extensions linéaires de P dont l'intersection est P, avec $k = dim(P)$. Comme $P = \cap_{i=1}^{k} E_i = (E_1 \cap E_{\lceil \frac{k}{2} \rceil + 1}) \cap (E_2 \cap I_{\lceil \frac{k}{2} \rceil + 2}) \cdots \cap (E_{\lceil \frac{k}{2} \rceil} \cap E_k)$, l'ordre P constitue l'intersection d'un ensemble d'ordres 2-dimensionnel. Ainsi, en codant chacun des ordres $E_i \cap E_{\lceil \frac{k}{2} \rceil + i}$ par *intervalle unique*, avec i entre 1 et $\lceil \frac{dim(P)}{2} \rceil$, il est possible d'obtenir un codage de P par l'union d'au plus $\lceil \frac{dim(P)}{2} \rceil$ intervalles, tel que pour tous x et y de P, $x \leq_P y$ si et seulement si l'union des intervalles codant x est incluse dans celle codant y. C'est ainsi que Dushnik et Miller prouvent que $i(P) \leq \lceil \frac{dim(P)}{2} \rceil$. Le codage de l'ordre  par *intersection d'ordres 2-dimensionnels* utilise au plus 2 intervalles pour coder chacun de ses éléments.

1.4 Représentation vectorielle

Cette sous-section expose la représentation vectorielle des ordres. Nous montrons d'abord comment l'amélioration du codage par *intersection d'extensions linéaires*, décrit dans 1.3.6, permet d'aboutir à une telle représentation puis nous citons quelques méthodes de codage assurant cette représentation pour tout ordre.

1.4.1 Définition

Une représentation vectorielle d'un ordre indique que chacun de ses éléments doit être représenté par des vecteurs de $\mathbb{R}^n$. Le codage permettant d'assurer une telle représentation pour un ordre P consiste à associer à chacun des éléments x et y de P un ou plusieurs intervalle(s) de $\mathbb{R}^n$, tel(s) qu'il existe une relation d'ordre entre les vecteurs associés à x et ceux associés à y, qui préserve la relation d'ordre entre x et y dans P.

1.4.2 Origine de la représentation vectorielle

D'après le Théorème 1.1.7, le codage par *intervalle unique* offre un codage efficace pour les chaînes (ordres linéaires). Néanmoins, il est possible de coder ces ordres plus efficacement d'un point de vue pratique. Soit une représentation intervallaire d'une chaîne à n éléments telle que son $i^{\text{ème}}$ élément x est codé par l'intervalle $[i, i]$ (ou bien $[i, n]$ *via* le codage MPTT). En compactant cet intervalle en l'unique entier i, et en faisant de même pour les autres éléments, nous obtenons un nouveau codage de la chaîne tel que la relation d'ordre entre ses éléments est reflétée par l'ordre usuel sur $\mathbb{N}$ de leurs nouveaux codes. De taille $\lceil \log_2(n) \rceil$, ce codage améliore celui basé sur la représentation intervallaire, puisque sa taille est $2\lceil \log_2(n) \rceil$. Nous faisons appel à ce codage pour améliorer le codage intervallaire des d extensions linéaires $E_1, \ldots, E_d$ intervenant dans le codage d'un ordre P par *intersection d'extensions linéaires*. Nous en déduisons un codage de P affectant à chacun de ses éléments x un vecteur de d entiers, $< x_1, \ldots, x_d >$, tel que $[x_i, x_i]$ pour i entre 1 et d, correspond au code attribué à x lors d'un codage de l'extension linéaire E_i par *intervalle unique*. Nous obtenons alors une représentation vectorielle de P que nous illustrons par la Figure 1.15.

1.4.3 Codage par vecteur d'entiers lié au calcul de la dimension

Nous avons montré dans 1.4.2 que tout ordre P admet une représentation vectorielle qui repose sur une amélioration du codage intervallaire de P par *intersection d'extensions linéaires*. Nous introduisons ici une méthode de codage par *vecteur d'entiers lié au calcul de la dimension* permettant d'assurer une représentation vectorielle plus compacte pour tout ordre.

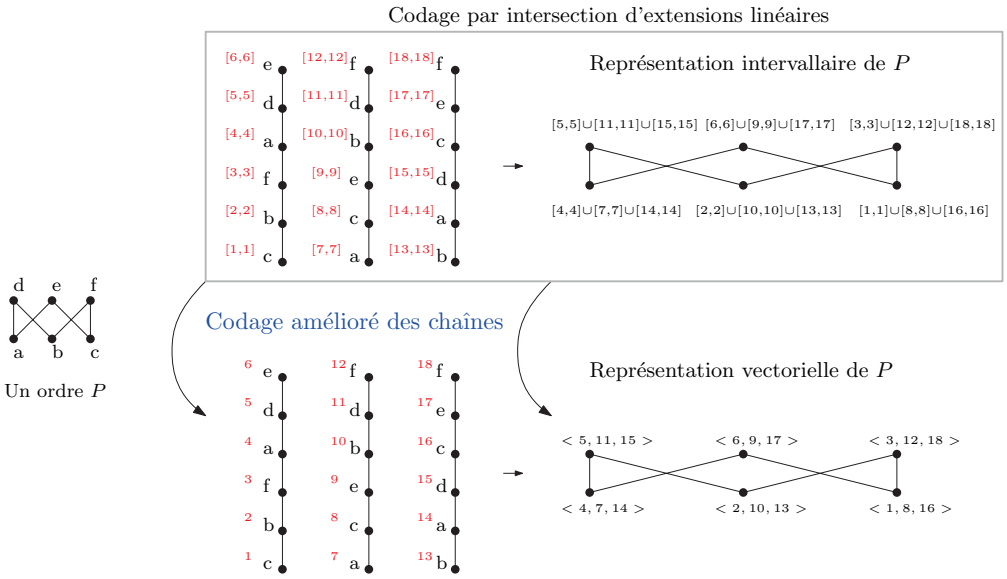


FIGURE 1.15: D'une représentation intervallaire à une représentation vectorielle

Définition 1.1.1 (Dimension). *La dimension d'un ordre P, notée $dim(P)$, est le plus petit entier d tel que $P \rightsquigarrow E_1 \times \cdots \times E_d$, ou encore $P = \cap_{i=1}^{d} E_i$, avec E_i une extension linéaire de P pour i de 1 à d.*

Soit $P = (X, R)$ un ordre de taille n et soient $E_1, \ldots, E_d$ des extensions linéaires de P avec $d = dim(P)$ et $P = \cap_{i=1}^{d} E_i$. Soit le codage associant au $j^{\text{ème}}$ élément de chaque extension linéaire E_i l'entier $j-1$. La relation d'ordre entre deux éléments de E_i est alors vérifiée par la comparaison sur $\mathbb{N}$ de leurs codes. L'application de ce codage aux $dim(P)$ extensions linéaires de l'ordre P, permet d'obtenir un codage de P qui, à son élément x, associe le vecteur $< x_1, \ldots, x_d >$, x_i étant l'entier attribué à x lors du codage de E_i. Ainsi, tester si $x \; R \; y$, x et y étant deux éléments de P, revient à tester si $x_i \leq y_i$, pour i de 1 à d. La taille de ce codage, illustré par la Figure 1.16, est $dim(P)\lceil \log_2(n) \rceil$.

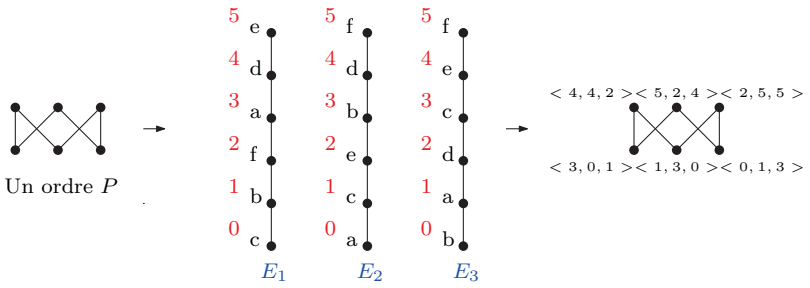


FIGURE 1.16: Codage par *vecteur d'entiers lié au calcul de la dimension*

Le codage de P par *vecteur d'entiers lié au calcul de la dimension* décrit un plongement de P dans un espace euclidien d-dimensionnel – $\mathbb{R}^d$ – ainsi que dans le produit cartésien de $dim(P)$ chaînes de taille n (voir Figure 1.17).

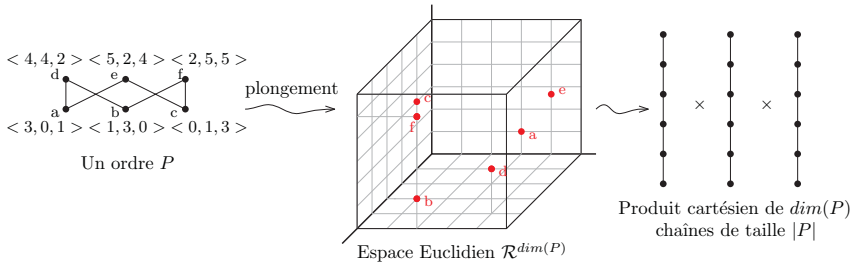


FIGURE 1.17: Interprétations de codage vectoriel lié au calcul de la dimension

1.4.4 Codage par vecteur de bits lié au calcul de la 2-dimension

La notion de dimension d'ordres est généralisée par Novak en notion de k-dimension, notée dim_k, définissant le nombre minimum de chaînes de taille k dans le produit desquelles un ordre est plongeable [Novak, 1963]. Notons que la dimension d'un ordre P de taille n correspond à sa n-dimension. Ainsi, $dim_n(P) \leq dim_k(P)$ pour tout entier $k \leq n$. Nous pouvons alors généraliser le codage présenté dans la sous-section 1.4.3 en un codage par *vecteur d'entiers lié au calcul de la k-dimension.*

Soit P un ordre et $C_1, \ldots, C_d$ des chaînes de taille k telles que $P \rightsquigarrow C_1 \times \cdots \times C_d$ et $d = dim_k(P)$. Un codage de P par *vecteur d'entiers lié au calcul de la k-dimension* consiste à associer à chacun de ses éléments x un vecteur de $dim_k(P)$ entiers, noté $< x_1, \ldots, x_d >$, tel que, pour i entre 1 et d, $< x_i >$ correspond au code attribué à x suite au codage de la chaîne C_i par *vecteur d'entiers lié au calcul de la dimension.* Notons que $dim(C_i) = 1$ et x_i appartient à $[0, k-1]$. La taille de ce codage est $dim_k(P)\lceil \log_2(k) \rceil$. Dans cette thèse, nous nous intéressons au cas $k = 2$. Le vecteur codant chaque élément de P est alors un vecteur de bits. Ainsi, nous parlons également de codage des ordres par vecteur de bits. La Figure 1.18 présente le codage par vecteur de bits de l'ordre  . La taille de ce codage vaut 3 et il constitue la plus petite taille d'un codage de cet ordre. En effet, comme la taille de l'ordre vaut 6, il faut au moins $\lceil \log_2(6) \rceil = 3$ bits pour coder ses éléments.

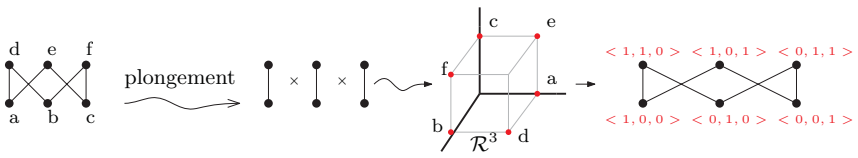


FIGURE 1.18: Codage par *vecteur d'entiers lié au calcul de la* 2*-dimension*

Dans les sections qui suivent, nous présentons un état de l'art autour du codage par *vecteur de bits lié au calcul de la 2-dimension*.

Notons que d'autres méthodes de codage des ordres se présentent dans la littérature comme le PQ-Encoding [Zibin et Gil, 2001], la décomposition en chaînes [Bouchet, 1971] et la décomposition en antichaînes [Fall, 1998], le plongement d'un ordre vers d'autres structures ou sa description en fonction de certains ordres particuliers [Ceroi, 2000], [Aït-Kaci *et al.*, 1989]. Cependant, les diverses interprétations de la dimension des ordres garantissent des pistes de recherches intéressantes autour des codages liés au calcul de ce paramètre.

2 Codage par vecteur de bits et la 2-dimension

Dans cette section, nous reprenons la définition du codage des ordres par vecteur de bits ainsi que la notion de la 2-dimension d'ordres. Nous donnons également quelques propriétés du paramètre dim_2 ainsi que des éléments de preuve justifiant la complexité de calcul de la 2-dimension des ordres.

2.1 Définitions

En 1963, V. Novak introduit la notion de la 2-dimension d'un ordre définie comme le nombre minimum de chaînes de taille 2 tel qu'il existe un plongement de l'ordre dans leur produit [Novak, 1963]. Trotter l'introduit différemment : il s'agit de la dimension du plus petit treillis booléen dans lequel il est possible de plonger l'ordre [Trotter, 1975]. La 2-dimension d'un ordre P, aussi appelée sa *dimension booléenne*, est notée $dim_2(P)$.

Définition 1.2.2 (2-dimension [Trotter, 1975])**.** *Pour tout ordre P, nous avons $dim_2(P) = n$ si $P \rightsquigarrow \mathcal{B}_n$ et $P \not\rightsquigarrow \mathcal{B}_{n-1}$.*

Treillis booléen Soit $\mathcal{B}_n = (2^{\{1,\dots,n\}}, \subseteq)$ un treillis booléen de dimension n. Ce treillis est également défini par $(\{0,1\}^n, \leq)$. En effet, il suffit de faire correspondre tout élément x de $2^{\{1,\dots,n\}}$ à un vecteur de bits V_x de taille n dont le $i^{\text{ème}}$ bit est 1 si l'élément i appartient à x, et 0 sinon (Figure 1.19). Trivialement, x et y étant deux éléments de $2^{\{1,\dots,n\}}$, $x \subseteq y$ si et seulement si $V_x \leq V_y$. La relation d'ordre entre V_x et V_y est également obtenue par l'opération $\texttt{OR}(V_x, V_y) = V_y$, avec $\texttt{OR}$ l'opérateur booléen « OU logique ».

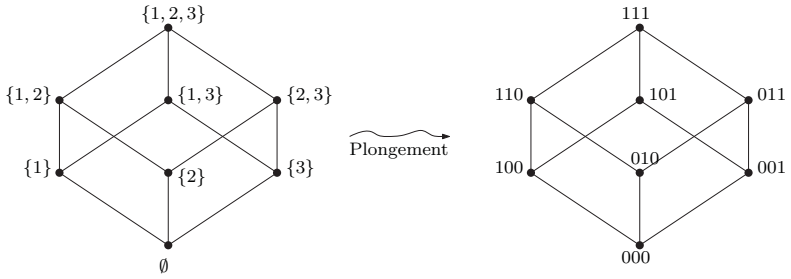


FIGURE 1.19: À gauche $\mathcal{B}_3 = (2^{\{1,2,3\}}, \subseteq)$ et à droite $\mathcal{B}_3 = (\{0,1\}^3, \leq)$

Codage par vecteur de bits Soit P un ordre et n un entier. Le plongement de P dans le treillis booléen $(\{0,1\}^n, \leq)$ définit un *codage par vecteur de bits* de P de taille n où à chaque élément de P est attribué un vecteur de n bits constituant son code. La taille minimale d'un tel codage vaut exactement $dim_2(P)$.

Le codage par vecteur de bits d'un ordre permet de déterminer l'espace mémoire nécessaire pour son stockage et d'accélérer la recherche de la relation d'ordre entre chaque paire de ses éléments. Ainsi, si P admet un codage par vecteur de bits de taille e, alors il faut au moins $e.|P|$ bits pour le stocker en mémoire. De plus, tester si $x \leq_P y$, x et y étant une paire d'éléments de P, revient à tester si $V_x \leq V_y$ en temps $\mathcal{O}(e)$. Notons que dans un modèle Word-RAM, où la taille d'un mot machine vaut w, ce test de comparabilité s'effectue plus précisément en $\mathcal{O}(\frac{e}{w})$. Rappelons qu'il s'agit d'un modèle de machine dans lequel la mémoire peut stocker une séquence de bits – un mot – dans une seule case et effectuer les opérations arithmétiques et logiques sur ces mots en une unité de temps.

Codage par vecteur de bits (version ensembliste) Soit P un ordre admettant un plongement dans le treillis booléen $(2^{[\![n]\!]}, \subseteq)$, avec n un entier. Ce plongement permet de définir un codage de P qui à chacun de ses éléments associe un sous-ensemble de $S = \{1, \ldots, n\}$ tel que la relation d'ordre entre deux éléments coïncide avec l'inclusion de leurs sous-ensembles. Les éléments de S sont appelés **couleurs** et l'ensemble des couleurs associé à un élément de P constitue son code. Nous pouvons en déduire un codage par vecteur de bits de P par la transformation de chaque sous-ensemble codant un élément de P en un vecteur de bits de taille n dont le $i^{\text{ème}}$ bit vaut 1 si l'élément i appartient à x, et 0 sinon. Nous admettons alors que ce codage décrit également (et implicitement) un codage par vecteur de bits de P. Nous utilisons dans la suite cette représentation ensembliste du codage par vecteur de bits dans la mesure où elle est plus simple à manipuler.

Codage réduit d'un ordre Soit $P = (X, \leq)$ un ordre et n un entier. Soit ϕ un plongement de P dans $(2^{[\![n]\!]}, \subseteq)$. Un codage réduit de P est le codage associant à chacun de ses éléments x l'ensemble $\phi(x) \setminus \bigcup_{u<x} \phi(u)$. Cet ensemble constitue le **code propre** de x ; c'est également l'**ensemble de ses couleurs propres**. Il est possible de déduire, d'un codage réduit de P, un codage de P grâce à un processus d'héritage de couleurs : chaque élément de l'ordre reçoit l'union de ses couleurs propres et des couleurs propres de tous ses prédécesseurs (il s'agit des couleurs héritées). Notons que, dans un codage réduit optimal de P, seuls ses éléments sup-irréductibles ont un ensemble de couleurs propres non vide.

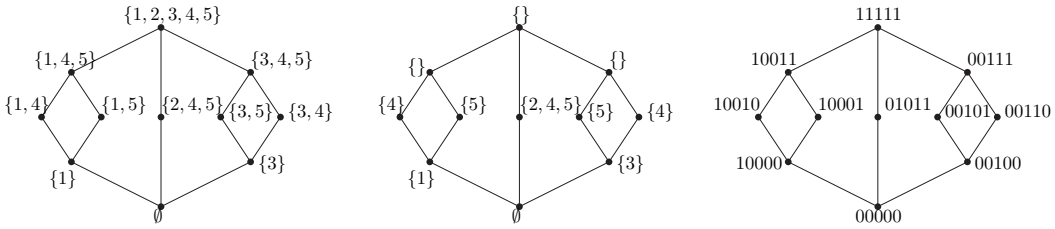


FIGURE 1.20: Des descriptions du codage par vecteur de bits

Calcul d'un codage par vecteur de bits Soit P un ordre et n un entier. Le calcul d'un codage de P par vecteur de bits de taille n implique de vérifier si $P \rightsquigarrow \mathcal{B}_n$, ce qui revient à décider si $dim_2(P) \leq n$. Ainsi, le calcul d'un codage par vecteur de bits d'un ordre est lié au calcul de sa 2-dimension. Toutefois, calculer $dim_2(P)$, et même simplement décider si $dim_2(P) \leq k$, quelle que soit la valeur de l'entier k, sont des problèmes difficiles à résoudre. Nous donnons dans la suite des éléments de preuve de la complexité de ces problèmes ainsi que des heuristiques envisagées pour les résoudre.

2.2 Propriétés de la 2-dimension

Nous citons dans cette partie des propriétés élémentaires du paramètre dim_2.

Proposition 1.2.2 (Folklore)**.** *Soit P un ordre.*
- *Pour P^d le dual de P, $dim_2(P) = dim_2(P^d)$ (**Dualité**).*
- *Pour Q un sous-ordre de P, $dim_2(Q) \leq dim_2(P)$ (**Monotonie**).*
- *Si $Q \rightsquigarrow P$ alors $dim_2(Q) \leq dim_2(P)$ (**Monotonie**).*
- *Pour x un élément de P, $dim_2(P) \leq dim_2(P \setminus \{x\}) + 2$ (**Continuité**).*

Nous donnons dans la Proposition 1.2.3 des bornes triviales de la 2-dimension des ordres.

Proposition 1.2.3 (Folklore). *Pour tout ordre P, nous avons*

$$\lceil \log_2(|P|) \rceil \leq dim_2(P) \leq |P|.$$

Démonstration. Notons n la 2-dimension de P. Soit ϕ un plongement de P dans le treillis booléen $\mathcal{B}_n$. Comme $|P| \leq |\mathcal{B}_n|$ et $|\mathcal{B}_n| = 2^n$, $\lceil \log_2(|P|) \rceil \leq n$. On en déduit que la borne inférieure est vérifiée : $\lceil \log_2(|P|) \rceil \leq dim_2(P)$.

Soit $\{x_1, \ldots, x_m\}$ l'ensemble des éléments de P. Pour déterminer la borne supérieure, on considère la fonction qui à chaque élément x_i de P associe l'ensemble $\{j | x_j \leq x_i\}$. Il est facile de vérifier que cette fonction définit un codage par vecteur de bits de P de taille $|P|$. Par conséquent, $dim_2(P) \leq |P|$. □

Remarquons que la taille de la plus longue chaîne du treillis booléen $\mathcal{B}_n$ est $n+1$ ainsi que la taille de sa plus large antichaîne est $\binom{n}{\lfloor \frac{n}{2} \rfloor}$. Nous pouvons en déduire la 2-dimension des chaînes et des antichaînes (Proposition 1.2.4).

Proposition 1.2.4. *Soit P un ordre à n éléments.*

- *Pour P une chaîne, on a $dim_2(P) = n - 1$ (Folklore).*
- *Pour P une antichaîne, on a $dim_2(P) = sp(n)$ avec $sp(n)$ est égale à $\min\{k | \binom{k}{\lfloor \frac{k}{2} \rfloor} \geq n\}$ [Sperner, 1928].*

La fonction $sp(n)$ (cf. Appendice B), introduite dans la Proposition 1.2.4, est à l'origine des travaux de Sperner et permet de déterminer un codage optimal par vecteur de bits des antichaînes. Ce codage consiste à associer à chaque élément d'une antichaîne de taille n un sous-ensemble de taille $\lfloor \frac{sp(n)}{2} \rfloor$ de l'ensemble $\{1, \ldots, sp(n)\}$ (Figure 1.21). Habib *et al.* bornent la valeur de $sp(n)$ par $\lfloor \log_2(n) + \frac{\log_2(\log_2(n))}{2} + 1 \rfloor$ et $\lfloor \log_2(n) + \frac{\log_2(\log_2(n))}{2} + 2 \rfloor$ [Habib *et al.*, 2004].

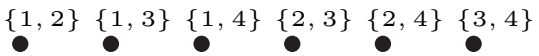


FIGURE 1.21: Codage optimal par vecteur de bits d'une antichaîne

Soit P un ordre. Suite à la monotonie de la 2-dimension des ordres et d'après la Proposition 1.2.4, il est clair que $\max(h(P), sp(w(P))) \leq dim_2(P)$. En effet, la hauteur de P ($h(P)$) correspond à la taille d'une chaîne maximale induite de P moins un, et sa largeur ($w(p)$) correspond à la taille d'une antichaîne maximale induite de P. La Proposition 1.2.5 donne quelques expressions de la 2-dimension d'un ordre en fonction de la 2-dimension de ses sous-ordres.

Proposition 1.2.5. *Soit P et Q deux ordres. Alors,*

- *Si P ne possède pas d'élément maximum ou Q ne possède pas d'élément minimum, alors*
$$dim_2(P.Q) = dim_2(P) + dim_2(Q).$$
Sinon
$$dim_2(P.Q) = dim_2(P) + dim_2(Q) + 1 \text{ [Trotter, 1975].}$$
- *Si P possède un maximum et Q un minimum ou vice versa, donc*
$$dim_2(P \times Q) = dim_2(P) + dim_2(Q).$$
Sinon
$$dim_2(P \times Q) \leq dim_2(P) + dim_2(Q) \text{ [Trotter, 1975].}$$
- *Soit $\mathcal{M}$ une partition modulaire de P, alors*
$$dim_2(P) \leq dim_2(P_{/\mathcal{M}}) + \sum_{M \in \mathcal{M}} dim_2(P[M]) \text{ [Thierry, 2001].}$$

2.3 Complexité de la 2-dimension

Stahl et Wille, [Stahl et Wille, 1986] puis Habib *et al.* [Habib *et al.*, 2004], montrent que pour tout ordre P, décider si $dim_2(P) \leq k$ est un problème $\mathcal{NP}$-complet. Nous présentons dans le Théorème 1.2.8 les éléments clés de ce résultat.

Théorème 1.2.8 (Habib *et al.*, 2004)**.** *Soit $P = (X, \leq)$ un ordre et k un entier. Les trois propositions suivantes sont équivalentes :*

1. *Il existe un plongement de P dans $\mathcal{B}_k$.*
2. *Il existe un recouvrement de $B(P) = (X, X, \not\leq)$ par k bipartis complets.*
3. *Le graphe $G(P) = (V, E)$ avec $V = \{xy | (x, y)$ dans X^2 tel que $x \not\leq y\}$ et $E = \{(xy, st) | x \leq t$ ou $s \leq y\}$ est k-colorable.*

Démonstration. La preuve complète peut être retrouvée dans [Thierry, 2001]. Néanmoins, nous en présentons la partie qui rejoint le contexte de ce manuscrit.

(1) $\Rightarrow$ (2) : soit ϕ un plongement de P dans $\mathcal{B}_k$. Nous affectons chaque arête xy de $B(P)$ au biparti B_i tel que $i \in \phi(x) \backslash \phi(y)$. Notons que $x \not\leq y$ implique $y \leq x$ ou $y \not\leq x$, et dans les deux cas, on a bien $\phi(x) \backslash \phi(y) \neq \emptyset$. Supposons qu'il existe un biparti B_j qui n'est pas complet. Il existe alors deux éléments a et b de B_j tel que ab n'est pas une arête de B_j, ce qui signifie $a \leq b$. Cela implique $\phi(a) \subseteq \phi(b)$. Soient x et y deux éléments de B_j tels que ax et yb sont des arêtes de B_j.

Comme $\phi(a)\backslash\phi(x) \subseteq \phi(b)$ et $\phi(y)\backslash\phi(b) \cap \phi(b) = \emptyset$, ax et yb ne peuvent pas être dans le même biparti. Cela contredit le fait que ces deux arêtes sont dans B_j. Donc, les k bipartis sont complets et forment un recouvrement de $B(P)$.

(2) $\Rightarrow$ (1) : la preuve fait intervenir une implication intermédiaire détaillée dans [Thierry, 2001].

(2) $\Rightarrow$ (3) : soient $B_1, \ldots, B_k$ des bipartis complets recouvrant $B(P)$. Nous colorons chaque sommet xy de $G(P)$ avec la couleur i tel que xy est une arête dans B_i. Soient xy et st deux sommets adjacents dans $G(P)$. Cela signifie que $x \leq t$ ou $s \leq y$. Ainsi, les arêtes xy et st ne peuvent pas appartenir au même biparti B_i puisqu'il est complet. Comme xy et st sont dans deux bipartis différents, des couleurs différentes sont attribuées aux sommets xy et st, permettant de définir une coloration du graphe de conflit $G(P)$ par k couleurs.

(3) $\Rightarrow$ (2) : soit une coloration des sommets de $G(P)$ par k couleurs. Nous construisons le biparti B_i à partir de toutes les arêtes xy telles que le sommet xy soit coloré par i dans $G(P)$. Montrons que le biparti B_i est complet. Soient xy et st deux arêtes de B_i. Supposons que xt n'est pas une arête de B_i. Alors $x \leq t$. Par conséquent, les sommets xy et st sont adjacents dans $G(P)$. Ils sont donc colorés différemment, ce qui contredit le fait que xy et st sont dans le même biparti. Finalement, nous obtenons un recouvrement de $B(P)$ par k bipartis complets. □

Le graphe $G(P)$ considéré dans le Théorème 1.2.8 comprend le graphe de conflit critique de l'ordre P, noté $G_{cr}(P)$ et défini sur les sommets xy de $G(P)$ tels que $x \updownarrow y$ (voir définition des relations flèches à la page 11). Skorsky formalise le problème de décision associé au problème du calcul de la 2-dimension des ordres en terme de problème de coloration du graphe de conflit.

Théorème 1.2.9 (Skorsky, 1992)**.** *Pour tout ordre P, $dim_2(P) = \chi(G_{cr}(P))$.*

Le Théorème 1.2.9 ainsi que les équivalences prouvées dans le Théorème 1.2.8 permettent de conclure que le problème de calcul de la 2-dimension d'un ordre est équivalent d'une part au problème de calcul d'un recouvrement minimal d'un biparti par des bicliques (bipartis complets), d'autre part au problème de calcul du nombre chromatique d'un graphe. Ces deux problèmes, auxquels se rattache le calcul de la 2-dimension sont $\mathcal{NP}$-complets et non-approximables. Il en résulte que le calcul de la 2-dimension est aussi non-approximable.

Théorème 1.2.10 (Habib *et al.*, 2004)**.** *Pour tout $\epsilon > 0$, il n'existe pas d'algorithme polynomial approchant la 2-dimension d'un ordre de taille n en $\mathcal{O}(n^{\frac{1}{21}-\epsilon})$ sauf si $\mathcal{P}$ est égal à $\mathcal{NP}$.*

3 Calcul de la 2-dimension

Face à la difficulté intrinsèque du problème de calcul de la 2-dimension des ordres, les travaux de recherche menés pour le résoudre s'orientent vers la conception des heuristiques de calcul des codages des ordres par vecteur de bits. Rappelons que le calcul d'un codage par vecteur de bits d'un ordre permet de borner supérieurement sa 2-dimension, c'est-à-dire de se rapprocher de sa valeur exacte. Nous présentons dans cette section les principales heuristiques calculant un codage par vecteur de bits des ordres, et en particulier des arbres.

3.1 Cas des ordres partiels généraux : Heuristiques

Nous avons montré dans 2.3 que le problème de calcul de la 2-dimension des ordres est lié à un problème central de l'optimisation combinatoire : *la coloration de graphes* [Skorsky, 1992]. Il n'est donc pas surprenant que l'heuristique principale de calcul d'un codage des ordres par vecteur de bits soit une heuristique de coloration de graphes. Nous décrivons dans cette sous-section le principe de cette heuristique. Après en avoir expliqué le désavantage, nous en proposons une amélioration. Nous présentons enfin la meilleure solution existante pour calculer un codage des ordres par vecteur de bits.

3.1.1 La Simple Coloration

En 1993, Caseau introduisit l'heuristique de *Simple Coloration* permettant de calculer un codage des ordres par vecteur de bits [Caseau, 1993] à travers une coloration de graphe. Le principe de cette heuristique est d'assurer un codage réduit d'un ordre dans lequel à chaque élément est attribuée une couleur simple, à savoir l'ensemble vide ou un singleton. Nous constatons qu'un codage réduit optimal d'une chaîne décrit bien le principe de cette heuristique (Figure 1.22).

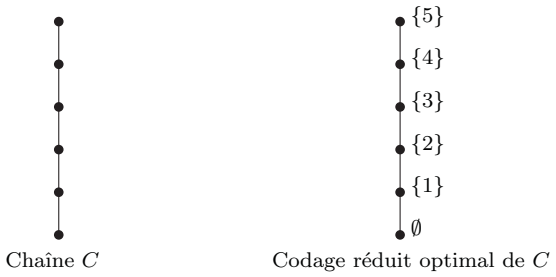


FIGURE 1.22: Codage réduit optimal d'une chaîne

Le graphe considéré par l'heuristique de *Simple Coloration* a été discuté à plusieurs reprises et correspond actuellement au graphe de conflit défini par Caseau *et al.* [Caseau *et al.*, 1999] (Définition 1.3.3). Ce graphe a pour sommets les éléments d'un ordre devant avoir un code propre non vide dans un codage réduit optimal de l'ordre. Ces éléments coïncident avec les sup-irréductibles de l'ordre. Ainsi, une arête reliant deux sommets du graphe de conflit exprime que les éléments sup-irréductibles qui représentent ces deux sommets n'admettent pas une couleur simple identique. Le calcul d'un codage d'un ordre P par l'heuristique de *Simple Coloration* se résume dans la coloration de $G_c(P)$ (Figure 1.23) : la couleur d'un élément de P dans $G_c(P)$ définit sa couleur propre dans un codage réduit de P (cf. Algorithme 16 en Appendice A). Nous illustrons ce codage par la Figure 1.23. Caseau *et al.* montrent que le calcul d'un codage par la *Simple Coloration* de taille optimale – $\chi(G_c(P))$ – est un problème $\mathcal{NP}$-complet [Caseau *et al.*, 1999].

Définition 1.3.3 (Caseau *et al.*, 1999)**.** *Le graphe de conflit d'un ordre P est $G_c(P) = (J(P), E)$ avec E l'ensemble des paires (j, j') de $J(P)^2$ tel qu'il existe un élément m de $M(P)$ vérifiant l'une des conditions suivantes :*

- *(j, m) est une paire critique et $j' \leq m$.*
- *(j', m) est une paire critique et $j \leq m$.*

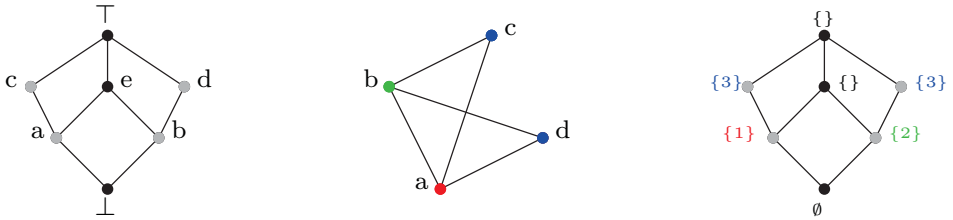


FIGURE 1.23: À gauche, un ordre dont les éléments sup-irréductibles sont en gris. Au centre, son graphe de conflit coloré et à droite un codage réduit de l'ordre par la *Simple Coloration*.

3.1.2 Problématique

L'heuristique de *Simple Coloration* permet de calculer un codage par vecteur de bits, d'un ordre P, de taille au moins $\chi(G_c(P))$. Néanmoins, rappelons que le codage des antichaînes par cette heuristique est loin d'être efficace (Figure 1.24). En effet, étant donnée une antichaîne A de taille n, nous avons $dim_2(A)$ vaut $sp(n)$ et s'exprime en $\mathcal{O}(\log_2(n))$ (page 105), ce qui est négligeable par rapport à la taille du codage de A par une *Simple Coloration* qui vaut n (coloration d'une clique).

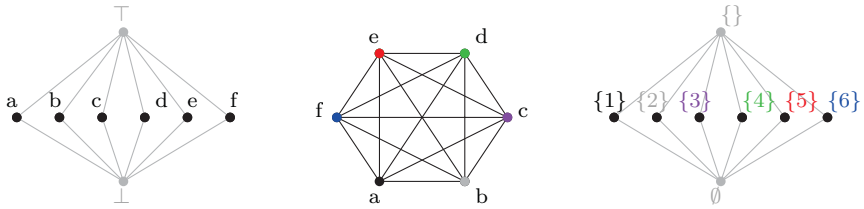


FIGURE 1.24: Une *Simple Coloration* d'une antichaîne

Rappelons que le codage réduit optimal d'une antichaîne consiste à affecter à chacun de ses éléments un code propre qui n'est pas nécessairement un singleton ni un ensemble vide. Ce codage, obéissant au principe d'une *Coloration Multiple*, permet de minimiser la taille du codage des ordres.

3.1.3 La Coloration multiple : *SBSC*

L'heuristique de *Coloration Multiple*, la *Split and Balancing Simple Coloring*, permet également d'assurer un codage des ordres par vecteur de bits. Le principe de cette heuristique est d'attribuer à chacun des éléments d'un ordre un code propre constitué de multiples couleurs. Cet ensemble de couleurs propres n'est pas nécessairement vide ni réduit à une unique couleur. Le calcul d'un tel codage fait appel à une phase du pré-traitement de l'ordre initial, appelée *Split and Balancing*, suivie du calcul d'une *Simple Coloration* de l'ordre après son pré-traitement. Ces deux étapes de l'heuristique de calcul d'une *Coloration Multiple* des ordres sont décrites dans la suite. L'acronyme *SBSC* est utilisé pour désigner cette heuristique.

Phase du Split and Balancing Krall *et al.* proposent une phase du pré-traitement *Split and Balancing* d'un ordre, qui consiste à subdiviser récursivement les successeurs immédiats d'un élément de l'ordre en des groupes, si le degré sortant de cet élément est important, et à créer un nouveau parent pour chaque groupe d'au moins deux éléments. Plusieurs critères entrent en jeu pour générer cette subdivision en maintenant la hauteur de la hiérarchie équilibrée. À titre d'exemple, les éléments ayant quelques descendants en commun doivent appartenir au même groupe. Le lecteur pourra se référer à l'article [Krall *et al.*, 1997] pour plus de détails autour de ce pré-traitement (voir la Figure 1.25 pour une illustration).

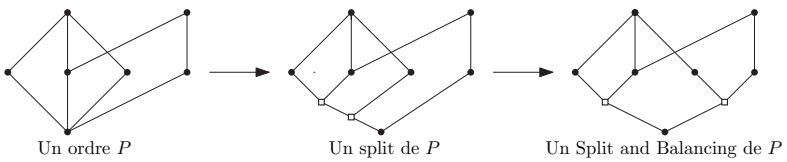


FIGURE 1.25: Le *Split and Balancing* d'un ordre

L'importance de la phase de pré-traitement est justifiée à travers des expérimentations effectuées sur des hiérarchies naturelles. Nous constatons que la taille du codage d'un ordre par une *Simple Coloration* diminue considérablement si l'ordre a subi un pré-traitement au préalable. Ce constat est illustré par la Figure 1.26. Nous cherchons à déterminer si la phase du *Split and Balancing* améliore l'heuristique de *Simple Coloration* pour tout ordre. Cette problématique est traitée dans le chapitre suivant.

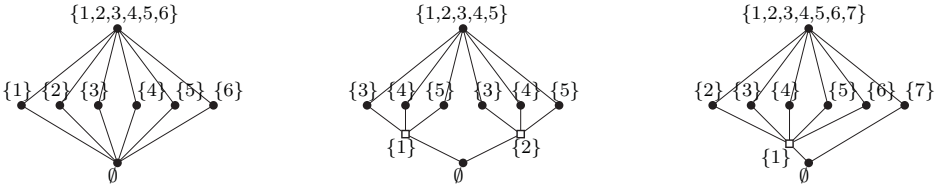


FIGURE 1.26: À gauche, une *Simple Coloration* de l'ordre. Au centre une *Simple Coloration* après un *Split and Balancing*. À droite, une *Simple Coloration* après un *Split* sans *Balancing*.

Phase d'une *Simple Coloration* Après la phase du pré-traitement, l'heuristique de *Coloration Multiple* calcule une *Simple Coloration* de l'ordre résultant du *Split and Balancing* de l'ordre initial. Enfin, elle associe à chaque élément x de l'ordre initial l'union de sa couleur simple et des couleurs simples des éléments introduits lors de la phase de pré-traitement : il s'agit des éléments qui sont à la fois des prédécesseurs de x et des successeurs de ses parents (dans l'ordre initial) dans l'ordre résultant (Figure 1.27).

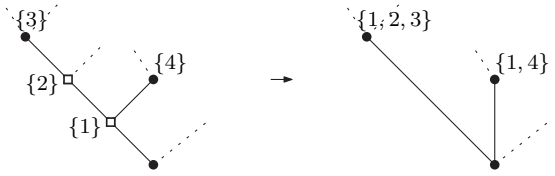


FIGURE 1.27: À gauche, une *Simple Coloration* de l'ordre pré-traité. À droite, une *Coloration Multiple* de l'ordre initial.

La *Simple Coloration* est calculée par l'heuristique présentée dans 3.1.1. En effet, l'heuristique de *Simple Coloration* d'un ordre après pré-traitement, proposée par Krall *et al.*, fait appel à un graphe de conflit qui n'est pas optimal, dans le sens où il détecte de faux conflits entre les éléments de l'ordre [Krall *et al.*, 1997]. Le graphe de conflit introduit par Caseau *et al.* (Définition 1.3.3) corrige ce problème. Rappelons que deux éléments d'un ordre sont en conflit lorsqu'ils ne doivent pas avoir la même couleur simple dans un codage réduit de l'ordre, ce qui s'exprime par une relation d'adjacence entre ces deux éléments dans le graphe de conflit. Un faux conflit est alors détecté par une arête entre deux éléments pouvant avoir une même couleur simple.

Heuristique *SBSC* : Nous admettons que la meilleure solution de codage des ordres par vecteur de bits repose sur une phase de pré-traitement *Split and Balancing* d'un ordre initial [Krall *et al.*, 1997] suivie d'une *Simple Coloration* de l'ordre pré-traité [Caseau *et al.*, 1999]. Cette combinaison n'a jamais été testée préalablement. Cependant, elle sera considérée comme l'heuristique de référence qui calcule un codage des ordres par vecteur de bits. Notons que le pré-traitement d'un ordre [Krall *et al.*, 1997], de même que la construction de son graphe de conflit par paires critiques [Caseau *et al.*, 1999], sont réalisables en temps et en espace polynomiaux. Par conséquent, l'heuristique *SBSC* calcule un codage des ordres par vecteur de bits en temps et en espace polynomiaux (cf. Algorithme 17 en Appendice A). Le processus de cette heuristique est illustré par la Figure 1.28.

Algorithme de coloration de graphes Toutes les heuristiques principales de codage des ordres par vecteur de bits font appel à un algorithme de coloration de graphe. La majorité de ces algorithmes se basent sur un parcours séquentiel des sommets du graphe en attribuant à chaque sommet visité la plus petite couleur possible. La qualité de la solution dépend de l'ordre dans lequel les sommets sont traités. Les algorithmes de coloration les plus répandus sont ceux qui considèrent un ordre des sommets par degré décroissant, *largest degree first ordering*, ou par degré partiel décroissant, *smallest degree last ordering*. L'ordre par degré partiel décroissant est obtenu en supprimant récursivement le sommet de plus petit degré avec toutes ses arêtes, puis en considérant l'ordre inverse de suppression de l'ensemble des sommets. Nous constatons que la coloration du graphe de conflit par l'algorithme *smallest degree last ordering* donne les meilleurs résultats. Nous optons donc pour cet algorithme afin de calculer une coloration du graphe de conflit par paires critiques lors d'un codage des ordres à l'aide de l'heuristique *SBSC*.

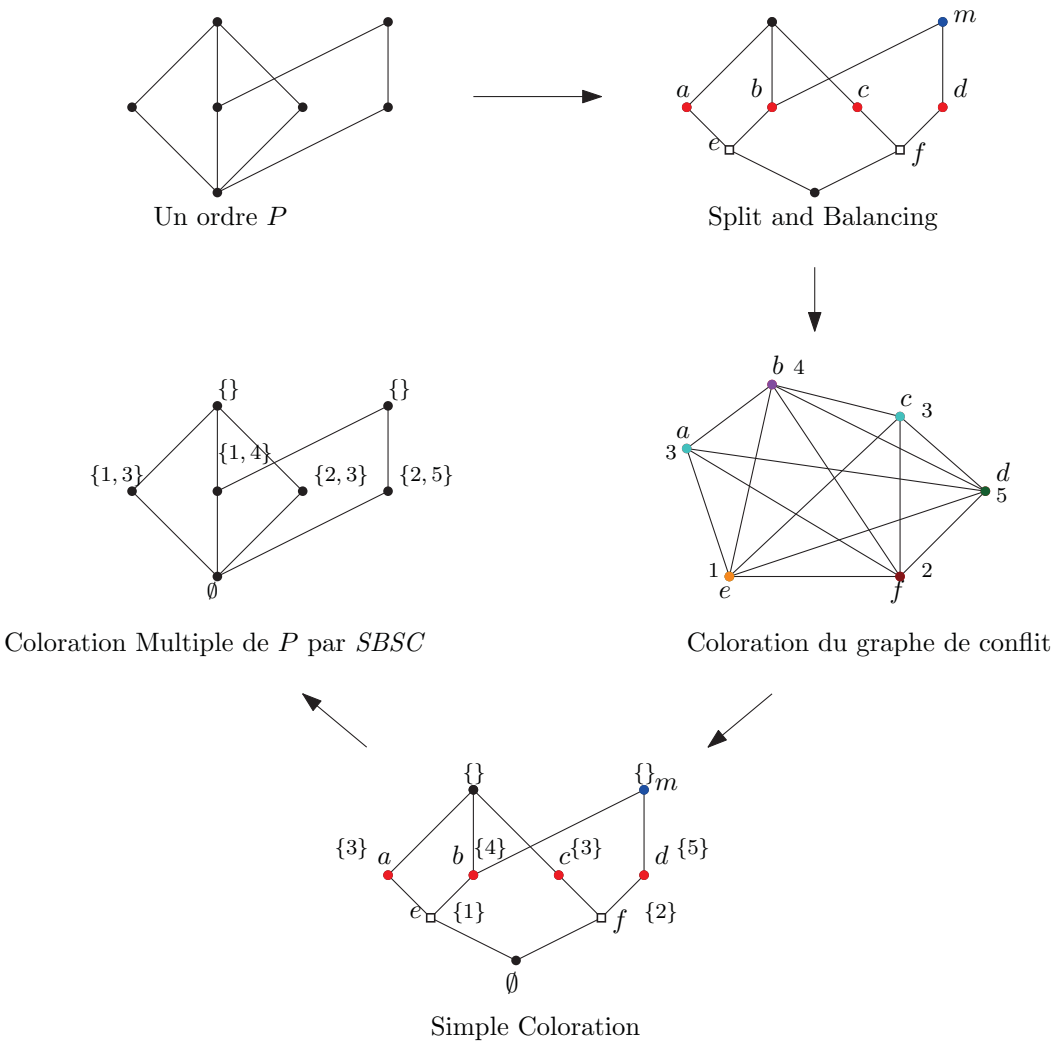


FIGURE 1.28: Le codage d'un ordre par la *SBSC*

3.2 Cas des arbres : Heuristiques

Nous avons évoqué dans la sous-section précédente deux heuristiques de calcul d'un codage des ordres par vecteur de bits. La première heuristique est basée sur une *Simple Coloration* d'un ordre et la seconde sur sa *Coloration Multiple*. Dans cette sous-section, qui traite la classe des arbres, nous présentons d'abord un algorithme calculant un codage des arbres suivant le principe d'une *Simple Coloration*, puis nous exposons d'autres algorithmes qui suivent le principe d'une *Coloration Multiple* pour les coder.

3.2.1 Algorithme de codage basé sur une *Simple Coloration*

Introduite par Caseau en 1993, l'heuristique de la *Simple Coloration* constitue la première méthode de codage des arbres par vecteur de bits. Elle fut ensuite généralisée pour le codage de tout ordre. Dans le cas des arbres, cette méthode est assurée par un algorithme nommé *Cmax* [Caseau, 1993]. Cet algorithme effectue un parcours en largeur d'un arbre, et récupère, à chaque élément x visité, la couleur maximale c_{max} (par ordre lexicographique) attribuée au fils de son prédécesseur immédiat. Ensuite, il associe à chacun des fils de l'élément x une couleur simple (non vide) de l'ensemble $\{c_{max}+1, \ldots, c_{max}+d_{out}(x)\}$, telle qu'il n'existe pas deux fils de même couleur simple. Pour le cas de la racine, $c_{max}=0$. Ce codage, illustré par la Figure 1.29, est calculable en temps et en espace polynomiaux, et sa taille coïncide avec le maximum degré d'une chaîne de l'arbre : $max\{\sum_{x\in C} d_{out}(x) \mid C$ une chaîne de l'arbre$\}$.

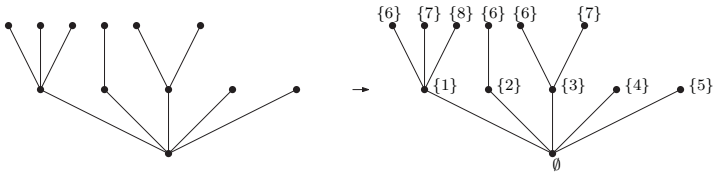


FIGURE 1.29: Codage des arbres par l'algorithme *Cmax*

Caseau montre que son algorithme *Cmax* calcule une *Simple Coloration* optimale d'un arbre, dans le sens où il fournit la plus petite taille d'un codage calculé suivant le principe d'une *Simple Coloration*. Rappelons que la *Simple Coloration* d'un arbre en assure un codage réduit et que le codage par vecteur de bits d'un arbre se déduit directement de son codage réduit.

3.2.2 Algorithmes de codage basé sur une *Coloration Multiple*

Nous remarquons que l'algorithme *Cmax*, introduit pour un codage des arbres basé sur une *Simple Coloration*, code les successeurs immédiats d'un élément x d'un arbre par $d_{out}(x)$ couleurs alors qu'il est possible de les coder seulement par $sp(d_{out}(x))$ couleurs (voir le codage optimal des antichaînes dans la Proposition 1.2.4). Ainsi, la problématique soulevée dans 3.1.2 concernant l'heuristique de *Simple Coloration* est également posée par l'algorithme *Cmax*. Nous présentons ici des améliorations apportées à cet l'algorithme, basées sur le principe de l'heuristique de *Coloration Multiple* défini dans 3.1.3.

Algorithme *CHNR* [Caseau *et al.*, 1999] Caseau *et al.* proposent l'algorithme *CHNR* qui calcule un codage des arbres basé sur une *Coloration Multiple*. Cet algorithme diffère de l'algorithme *Cmax* par l'attribution à chacun des fils d'un élément x d'un arbre un sous-ensemble de $\frac{sp(d_{out}(x))}{2}$ couleurs de l'ensemble $\{c_{max}+1, \ldots, c_{max}+sp(d_{out}(x))\}$. L'algorithme *CHNR* est calculable en temps polynomial et offre un codage d'un arbre de taille $max\{\sum_{x\in C} sp(d_{out}(x)) \mid C$ une chaîne de l'arbre$\}$. Nous illustrons ce codage sur la Figure 1.30.

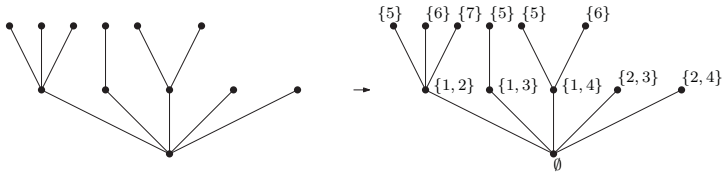


FIGURE 1.30: Codage des arbres par l'algorithme *CHNR*

Algorithme *Dicho* [Habib *et al.*, 2004] Habib *et al.* proposent une heuristique appelée *Dichotomique*, qui calcule aussi un codage des arbres basé sur une *Coloration Multiple*. Ils caractérisent ce codage par « codage *Dichotomique* ». Leur heuristique se déroule en deux étapes : (1) générer un arbre binaire à travers une phase du pré-traitement *Split and Balancing* d'un arbre initial, en divisant cet arbre en deux parties susceptibles d'être codées indépendamment, et en répétant récursivement cette opération sur chaque partie ; (2) appliquer l'algorithme *CHNR* (ou encore *Cmax*) à l'arbre binaire généré. L'heuristique est certes décrite différemment dans [Habib *et al.*, 2004], mais nous voulions donner une description dans le même contexte de l'heuristique de *Coloration Multiple* (*SBSC*). Le codage *Dichotomique* est calculable en temps polynomial et sa taille correspond au maximum degré d'une chaîne de l'arbre binaire produit. Habib *et al.* proposent alors un algorithme *Dicho* calculant un codage *Dichotomique* d'un arbre de plus petite taille [Habib *et al.*, 2004].

Théorème 1.3.11 (Habib *et al.*, 2004)**.** *L'algorithme* Dicho *calcule un codage* Dichotomique *de plus petite taille en temps* $\mathcal{O}(n.e)$*, n étant la taille de l'arbre et e celle du codage, avec $e \leq n$.*

Le but de l'algorithme *Dicho* est de plonger un arbre initial dans un arbre binaire ayant le plus petit maximum degré d'une chaîne. Le calcul d'un tel plongement repose sur l'affectation des poids aux éléments de l'arbre suivant un ordre topologique inverse de ces éléments. Soit T un arbre et x un élément de T. Il existe quatre cas de figure pour calculer le poids $w(x)$ de l'élément x :

1. Si l'élément x est une feuille, alors $w(x) = 0$.
2. Si l'élément x possède un unique fils y, alors $w(x) = w(y) + 1$.
3. Si l'élément x possède deux fils y et z, alors $w(x) = \max(w(y), w(z)) + 2$.
4. Si l'élément x possède au moins trois fils, alors un nouvel élément a est introduit dans T comme fils de x et parent de y et z, y et z étant les deux fils de x de plus petits poids. Dans ce cas, on ne peut calculer le poids de x qu'après avoir calculer celui de a.

La taille d'un codage *Dichotomique* de T, calculé par l'algorithme *Dicho*, correspond au poids attribué à sa racine. Dans ce manuscrit, nous le notons $Dicho(T)$. Nous tenons à préciser que la phase du *Split and Balancing* est traduite, dans l'algorithme *Dicho*, par l'introduction de nouveaux éléments dans l'arbre (Split), et par le choix des éléments du plus petit poids pour envisager cette introduction (Balancing). La Figure 1.31 illustre un codage *Dichotomique* d'un arbre calculé par l'algorithme *Dicho*.

Théorème 1.3.12 (Habib *et al.*, 2004)**.** *La* 2*-dimension des arbres est* 4*-approximable par l'algorithme Dicho.*

Autres algorithmes En 2002, Filman améliore le codage *Dichotomique* en *codage Polychotomique* [Filman, 2002]. Ce dernier fut encore amélioré par Colomb *et al.* en *codage Polychotomique généralisé* [Colomb *et al.*, 2008]. Nous définissons et analysons ces deux dernières heuristiques dans le chapitre suivant. Toutefois, nous pouvons d'ores et déjà affirmer que ces heuristiques diminuent faiblement la taille du codage *Dichotomique* calculé par l'algorithme *Dicho*. Cette taille est probablement proche de la 2-dimension des arbres, d'autant que le calcul de leur 2-dimension en temps polynomial constitue une question ouverte. Néanmoins, la 2-dimension des arbres est approximable (Théorème 1.3.12) et polynomialement calculable pour quelques arbres particuliers (Proposition 1.3.6).

Proposition 1.3.6.
- *La* 2*-dimension d'une peigne de hauteur* h *est* $h + 1$ *[Habib* et al.*, 2004].*
- *La* 2*-dimension d'une chenille de hauteur* h *et dont chaque élément possède* d *fils est* $h + sp(d) - 1$ *[Thierry, 2001].*
- *Pour* C_n *une chaîne de taille* n*,* $dim_2(C_n + C_1) = n + 1$ *[Trotter, 1975].*

Plusieurs autres questions sont ouvertes à propos de la 2-dimension des arbres et certains ordres particuliers. Nous en citons quelques-unes à la fin de ce chapitre.

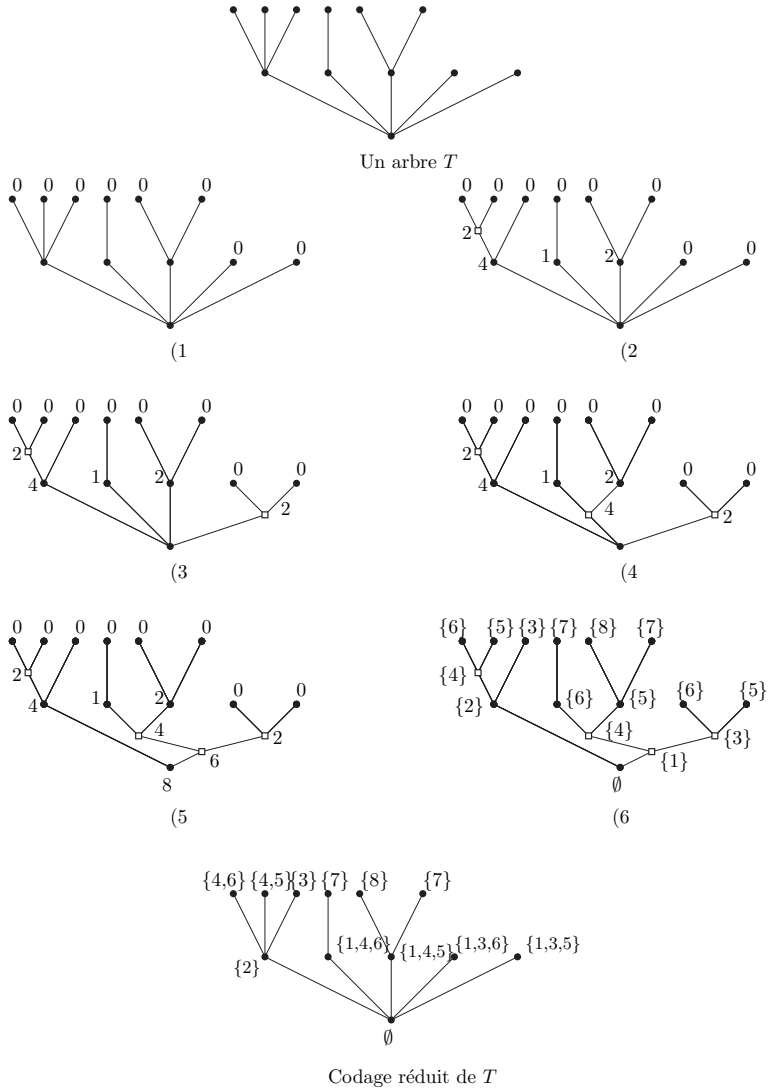


FIGURE 1.31: Étapes d'un codage *Dichotomique* calculé par l'algorithme *Dicho*

4 Problèmes ouverts

En 2004, Habib *et al.* conjecturent que la 2-dimension des arbres est 2-approximable par l'algorithme *Dicho* et qu'il existe un algorithme calculant la 2-dimension exacte de tout arbre en temps polynomial [Habib *et al.*, 2004]. Dans le même article, les auteurs donnent une troisième conjecture à propos d'une borne inférieure de la 2-dimension des arbres. En 2001, Thierry formule une conjecture [Thierry, 2001] à propos de la 2-dimension exacte des arbres n-complets. L'étude de ces conjectures, encore ouvertes, fait l'objet du troisième chapitre de ce manuscrit.

D'autres conjectures sont encore ouvertes au sujet de la 2-dimension, comme celle de Caseau sur les treillis.

Conjecture 1.4.1 (Caseau)**.** *Soit L un treillis. Alors,*

$$dim_2(L) \leq \max\{\sum_{x \in C} d_{out}(x) \mid C \text{ une chaîne dans } L\}.$$

La Conjecture 1.4.1 indique qu'il est possible de calculer un codage par vecteur de bits d'un treillis L (probablement via la *Simple Coloration*) de taille égale au maximum degré d'une chaîne de L, noté Δ_L. Nous déduisons de la Proposition 1.4.7 que, si $\max(|J(L)|, |M(L)|) \leq \Delta_L$, alors la conclusion de la conjecture est vérifiée pour L. Un autre cas particulier est étudié dans [Habib *et al.*, 1995] – les treillis coatomiques – mais le cas général est toujours ouvert.

Proposition 1.4.7 (Folklore)**.** *Soit L un treillis. Alors,*

$$dim_2(L) \leq \max(|J(L)|, |M(L)|).$$

Soit L un treillis distributif. Comme $h(L) \leq \Delta_L$, et $dim_2(L) = |J(L)| = |M(L)| = h(L)$ (propriété des treillis distributifs), la conclusion de la Conjecture 1.4.1 est alors vérifiée dans le cas des treillis distributifs. Nous déduisons aussi que la 2-dimension de cette classe d'ordres est calculable en temps polynomial.

Les classes d'ordres dont la 2-dimension est calculable en temps polynomial ne sont pas nombreuses. On se demande de quelles autres classes d'ordres il est possible de déterminer la 2-dimension exacte, ou simplement de mieux cerner sa valeur. Quel serait l'algorithme qui déterminerait les valeurs potentielles de la 2-dimension de ces classes d'ordres, et comment l'étendre pour le codage de tout ordre ? Existe-t-il des méthodes de codage par vecteur de bits non nécessairement liées au calcul de la 2-dimension ? et que peut on dire de leur performance ?

Cette thèse s'articule autour de ces questionnements que nous structurons en trois parties. La première partie (Chapitre 2) porte sur des heuristiques qui bornent au mieux la 2-dimension des ordres, et en particulier des arbres. La deuxième partie (Chapitre 3) abordent quelques conjectures autour de la 2-dimension des ordres. Enfin, la troisième partie (Chapitre 4) expose un codage des arbres par vecteur de bits plus intéressant.

« Ayant traversé le paradis infiniste où se trouvent à la fois des sommets inaccessibles (i.g. tous les problèmes se ramenant à la consistance) et des vallées fécondes (les grandes structures : treillis, algèbre de Boole,...) » [Pouzet et Richard, 1984], nous décrivons dans la suite ce que nous avons découvert dans cet univers.

Chapitre 2

La 2-dimension des ordres partiels : Heuristiques

> No problem can be solved until it is reduced to some simple form. The changing of a vague difficulty into a specific, concrete form is a very essential element in thinking.
>
> J.P. Morgan

Introduction

« *Si les problèmes d'organisation industrielle sont la plupart du temps relativement simples à énoncer, il ne faut en aucun cas sous-estimer l'effort nécessaire pour leur trouver une solution* » [Widmer, 1998], surtout lorsqu'il s'agit des problèmes d'optimisation combinatoire $\mathcal{NP}$-complets, ou encore $\mathcal{NP}$-difficiles, pour lesquels il faut compter un temps exponentiel (sauf si $\mathcal{P} = \mathcal{NP}$) pour leur trouver une solution optimum. Ces problèmes admettent pourtant des schémas de résolution permettant de fournir de bonnes solutions (proches de l'optimum) en un temps – quasi – polynomial. Ces schémas caractérisent les méthodes heuristiques.

Le problème de calcul de la 2*-dimension des ordres*, aussi connu par *le problème de calcul d'un codage des ordres par vecteur de bits de taille minimale*, fait partie des problèmes d'optimisation $\mathcal{NP}$-complets. Cela justifie la nature heuristique des méthodes existantes pour le résoudre (voir le premier chapitre d'état de l'art). Suivant ce schéma de résolution, nous traitons dans ce chapitre, de notre côté, le problème de calcul de la 2-dimension des ordres.

Dans la **première section**, nous proposons une heuristique calculant un codage par vecteur de bits des ordres séries-parallèles, à travers la technique de décomposition modulaire. En **deuxième section**, nous proposons une extension de cette heuristique permettant de calculer un codage par vecteur de bits de tout ordre. Enfin, dans la **troisième section**, nous dressons la stratégie commune des trois dernières heuristiques de codage des arbres, proposées dans la littérature, puis, suivant cette stratégie, nous concevons une nouvelle heuristique minimisant la taille du codage par vecteur de bits de cette classe d'ordres. Dans chaque section, nous donnons des résultats théoriques et expérimentaux propre à l'heuristique proposée. Nous précisons que les résultats théoriques sont majoritairement prouvés par induction mathématique et que les évaluations expérimentales sont assurées par des algorithmes, implémentés en langage C/C++, appliqués aux données de référence ou générées aléatoirement.

Notons que la conception des heuristiques développées dans ce chapitre fait appel à des outils mathématiques dont quelques-uns sont rappelés dans *Définitions et préliminaires* (pages 7-19) et d'autres sont définis dans ce chapitre.

1 Cas des ordres séries-parallèles

La notion d'ordre série-parallèle fut introduite en 1978 par Lawler [Lawler, 1978] lors de son étude d'un problème d'ordonnancement des tâches dans une machine. L'objectif de cette étude était de minimiser le temps total d'exécution de l'ensemble des tâches. Lawler montre que ce problème $\mathcal{NP}$-complet est quasi linéaire, si les contraintes de précédence entre les tâches sont modélisables par un ordre série-parallèle. Rappelons qu'un ordre est série-parallèle s'il ne contient pas de sous-ordre isomorphe à l'ordre $\mathbb{N}$. Dès lors, étudier la restriction des problèmes combinatoires $\mathcal{NP}$-complets, et même $\mathcal{NP}$-difficiles, aux ordres « séries-parallèles », a fait l'objet de plusieurs travaux de recherche. Il en résulte un nombre important de problèmes combinatoires polynomialement résolubles sur cette classe d'ordres. Citons ici le problème du nombre de sauts [Möhring, 1985] – le nombre minimum de comparabilités à rajouter à un ordre pour le rendre total –, et le problème de calcul de la dimension d'ordres [Garg, 2015] – les ordres séries-parallèles sont 2-dimensionnels.

La considération de la structure série-parallèle fut également le centre d'intérêt de plusieurs chercheurs en théorie des graphes. Nous référons le lecteur au [Takamizawa *et al.*, 1981] pour une liste des problèmes $\mathcal{NP}$-complets, qui sont résolubles en temps polynomial sur la classe des graphes séries-parallèles. Il est à noter que cette structure permet de modéliser des données (ordonnées) provenant de nombreux domaines comme l'ordonnancement, les systèmes concurrents, la fouille de données, les réseaux, etc. [Möhring, 1989].

Dans ce mémoire, nous étudions le problème de calcul de la 2-dimension des ordres qui est $\mathcal{NP}$-complet. Avant de traiter ce problème pour le cas général, étudions d'abord sa restriction aux ordres séries-parallèles.

1.1 Motivation

Nous avons évoqué dans le premier chapitre quelques classes d'ordres dont la 2-dimension est connue, à savoir les chaînes et les antichaînes. Remarquons qu'une chaîne correspond à une composition série d'un ensemble d'ordres triviaux (ordres réduits en un unique élément), alors qu'une antichaîne correspond à leur composition parallèle. En appliquant ces deux opérations de composition, un nombre fini de fois, à un ensemble d'éléments donné, nous obtenons un ordre « série-parallèle » défini sur cet ensemble. La Figure 2.1 illustre un exemple d'un ordre série-parallèle ainsi obtenu.

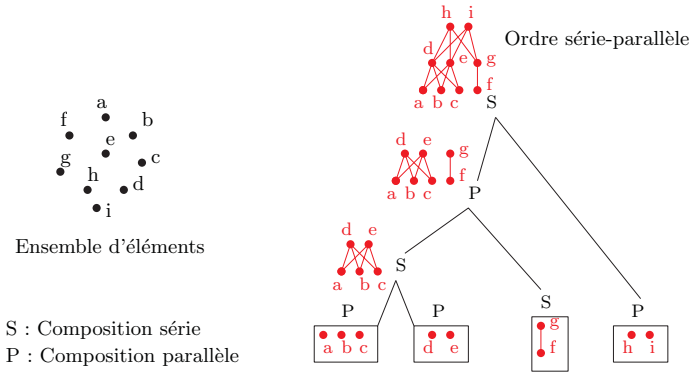


FIGURE 2.1: Construction progressive d'un ordre série-parallèle

La reconnaissance d'un ordre série-parallèle se fait grâce à son *arbre de décomposition modulaire* (définition à la page 15). Rappelons que l'arbre de décomposition modulaire d'un ordre est un arbre dont chaque nœud interne correspond à une opération sur ses sous-ordres. Ainsi, lorsqu'il s'agit d'un ordre série-parallèle, ces opérations sont limitées aux compositions série et parallèle. Observez que l'arbre de décomposition modulaire d'un ordre série-parallèle indique les étapes permettant de le générer (Figure 2.2). Il est à noter que l'ordre quotient (définition à la page 16) d'un ordre série-parallèle est soit une chaîne soit une antichaîne.

Définition 2.1.4 (Ordre série-parallèle)**.** *Un ordre est série-parallèle si son arbre de décomposition modulaire ne contient que des nœuds séries ou parallèles.*

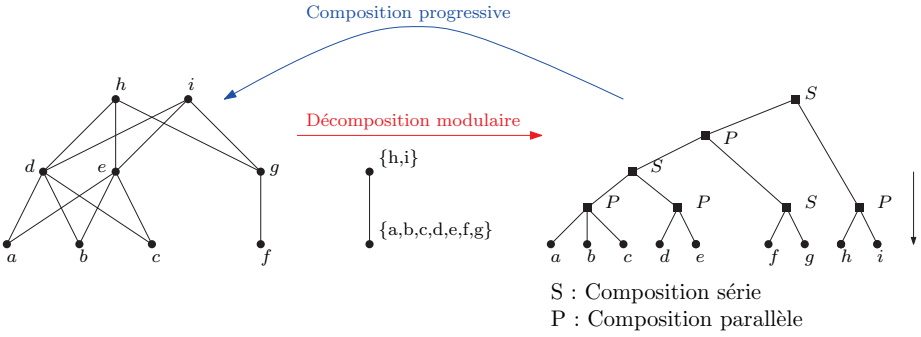


FIGURE 2.2: À gauche, un ordre série-parallèle. Au centre son ordre quotient et à droite son arbre de décomposition modulaire.

Nous dérivons de la Définition 2.1.4, une définition récursive des ordres séries-parallèles.

Définition 2.1.5 (Ordre série-parallèle)**.** *Un ordre est série-parallèle si :*
- *Il est réduit à un unique élément.*
- *Il est une composition série d'ordres séries-parallèles.*
- *Il est une composition parallèle d'ordres séries-parallèles.*

La définition récursive des ordres séries-parallèles facilite la procédure de leur codage par vecteur de bits. Il suffit de trouver un codage par vecteur de bits de la composition série, et aussi parallèle, d'un ensemble d'ordres déjà codés. Connaissant la 2-dimension de la composition série (resp. parallèle) d'ordres triviaux – cas d'une chaîne (resp. d'une antichaîne) –, nous nous demandons dans quelle mesure il est possible de calculer la 2-dimension de la composition série (resp. parallèle) d'ordres quelconques.

1.2 Stratégie

Soit SP un ordre série-parallèle et soit $M = \{M_1, \ldots, M_\ell\}$ une partition modulaire de SP. Nous rappelons qu'il s'agit de sa partition modulaire maximale. Par Définition 2.1.5, deux cas se présentent : (1) l'ordre SP correspond à une composition série des ordres $P[M_1], \ldots, P[M_\ell]$ $(\prod_{i=1}^{\ell} P[M_i])$; (2) l'ordre SP correspond à une composition parallèle de ces ordres $(\sum_{i=1}^{\ell} P[M_i])$. Nous supposons que les ordres $P[M_1], \ldots, P[M_\ell]$ sont déjà codés. Nous détaillons dans la suite la démarche que nous proposons pour le codage de l'ordre SP dans chacun des deux cas.

1.2.1 Cas d'une composition série d'ordres

Supposons que $SP = \prod_{i=1}^{\ell} P[M_i]$. La Proposition 2.1.8, illustrée par la Figure 2.3, donne la 2-dimension de SP en fonction de la 2-dimension des sous-ordres $P[M_i]$, pour 1 entre 1 et ℓ.

Proposition 2.1.8 (Composition série d'ordres [Trotter, 1975])**.** *Soit P un ordre avec $\{M_1, \ldots, M_\ell\}$ sa partition modulaire et la chaîne $\{M_1 < \cdots < M_\ell\}$ son ordre quotient. Donc,*

$$dim_2(P) = \sum_{1 \leq i \leq \ell} dim_2(P[M_i]) + |\{i \in [\![2, \ell]\!] : |M_i| = |M_{i-1}| = 1\}|$$

Démonstration. Cette preuve suit le schéma classique d'une induction sur la taille de l'ordre quotient.
Pour $\ell = 1$, nous avons $dim_2(P) = dim_2(P[M_1])$. Alors, la conclusion de la proposition est valide dans ce cas.
Pour un entier ℓ plus grand que 1, nous supposons que la propriété est vraie pour les valeurs inférieures à l. Soit P un ordre dont l'ordre quotient est la chaîne $\{M_1 < \cdots < M_{\ell-1} < M_\ell\}$. Notons P' le sous-ordre $P \backslash P[M_\ell]$.

Soit ϕ un codage par vecteur de bits de taille minimale de M_ℓ. Soit x un élément de M_ℓ et soit y un élément de P'. Alors, $y \leq_P x$. Par conséquent, un codage par vecteur de bits de P, de taille minimale, assignera à x la concaténation des vecteurs de bits $[1]^{dim_2(P')}$ et $\phi(x)$. En effet, l'ensemble des couleurs codant les éléments de P' est hérité par les éléments de M_ℓ et par conséquent ne peut pas être utilisé pour les distinguer. La taille minimale d'un codage par vecteur de bits de P vaut alors au moins $dim_2(P') + |\cup\{\phi(x) : x \in M_\ell\}|$. Nous en déduisons que $dim_2(P)$ vaut exactement :

- $dim_2(P') + dim_2(P[M_\ell])$, si $dim_2(P[M_\ell]) > 0$.
- $dim_2(P')$, si $dim_2(P[M_\ell]) = 0$ et $dim_2(P[M_{\ell-1}]) > 0$. En effet, le singleton M_ℓ est codé par $[1]^{dim_2(P')}$.
- $dim_2(P') + 1$, si $dim_2(P[M_\ell]) = dim_2(P[M_{\ell-1}]) = 0$. En effet, le singleton $M_{\ell-1}$ étant codé par $[1]^{dim_2(P')}$, il faut rajouter un bit supplémentaire pour coder l'élément de M_ℓ.

Tous les cas étant traités, nous concluons que la Proposition 2.1.8 est valide. □

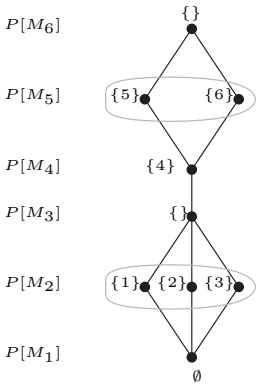


FIGURE 2.3: La 2-dimension de la composition série des ordres $P[M_i]$, pour i entre 1 et 6, est 6. La 2-dimension de $P[M_2]$ est 3, celle de $P[M_5]$ est 2 et elle est nulle pour les autres.

La Proposition 2.1.8 donne la taille d'un codage par vecteur de bits d'une composition série d'ordres déjà codés. L'Algorithme 1 calcule ce codage que nous illustrons par la Figure 2.4.

Données : Des ordres $P_1, \ldots, P_\ell$ déjà codés par l'ensemble $[\![1, s_i]\!]$ si $|P_i| > 1$, et par $\emptyset$ sinon, pour i de 1 à ℓ
Résultat : Un codage de $\prod_{i=1}^{\ell} P_i$

$E \leftarrow$ ensemble de couleurs codant P_1
pour *tout* i *allant de* 2 *à* ℓ **faire**
  pour *tout* x *dans* P_i **faire**
    $S \leftarrow \emptyset$
    `/* Recoder` x `par un ensemble de couleurs n'appartenant pas à` E `*/`
    pour *tout* s *dans* $code(x)$ **faire**
      $S \leftarrow S \cup \{s + |E|\}$
    fin
    $code(x) \leftarrow S \cup E$
    `/*` $s_i = 0$ `si` $|P_i| = 1$ `*/`
    si $s_i \neq 0$ **alors**
      `/* Union disjointe des couleurs codant les ordres` $P_1, \ldots, P_i$ `*/`
      $E \leftarrow E \cup \{1 + |E|, \ldots, s_i + |E|\}$
    fin
    sinon si $s_{i-1} = 0$ **alors**
      $code(s_i) \leftarrow code(s_i) \cup \{|E| + 1\}$
      $E \leftarrow E \cup \{|E| + 1\}$
    fin
  fin
fin

Algorithme 1 : Codage d'une composition série d'ordres

1.2.2 Cas d'une composition parallèle d'ordres

Supposons que $SP = \sum_{i=1}^{\ell} P[M_i]$. Lorsque les ordres $P[M_i]$ sont des chaînes, le codage de SP se ramène au codage de l'arbre construit à partir d'une composition série d'un élément r (racine de l'arbre) et l'ordre SP (Figure 2.5). Notons que le calcul d'un codage des arbres par vecteur de bits de taille minimale est un problème ouvert, même pour cette classe d'arbres particulière : racine reliée à une composition parallèle de chaînes. Nous utilisons alors la meilleure heuristique existante de codage des arbres par vecteur de bits, *Polychotomique Généralisé* [Colomb *et al.*, 2008], afin de calculer un codage de SP. Le principe de cette heuristique est expliqué en détail dans la dernière section de ce chapitre, mais nous le synthétisons comme suit :

Soit $s_1, \ldots, s_\ell$ des entiers. L'heuristique *Polychotomique Généralisé* peut donner la taille d'un codage par vecteur de bits d'un arbre dont les sous-arbres tels

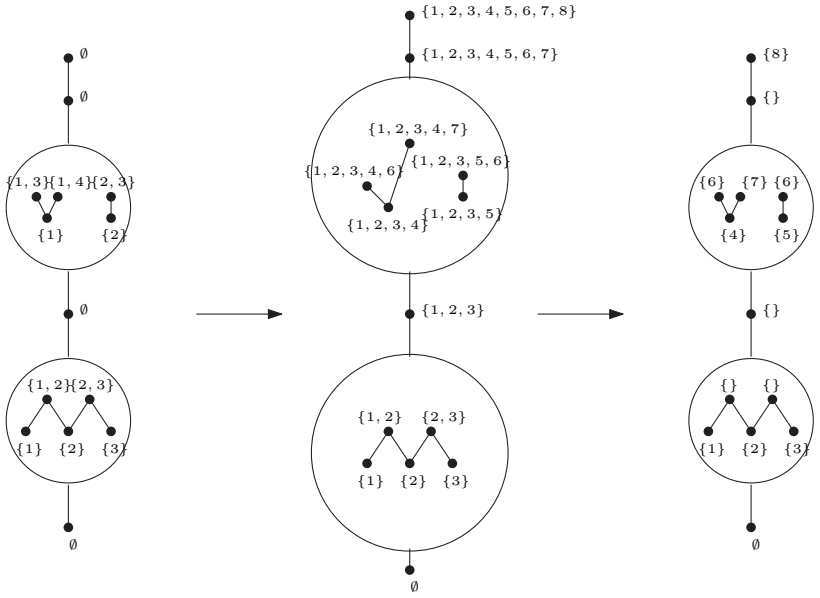


FIGURE 2.4: À gauche, une composition série d'ordres déjà codés. Au centre, le codage par vecteur de bits de cette composition et à droite son codage réduit.

que chacun est enraciné par un successeur immédiat de sa racine, sont codés respectivement par $s_1, \ldots, s_\ell$ bits. La taille de ce codage est calculée par une fonction $\mathcal{GP}$ qui prend en paramètre la séquence $s_1, \ldots, s_\ell$. La fonction $\mathcal{GP}$ calcule alors la taille d'un codage d'une composition parallèle d'arbres déjà codés sans tenir compte de leur structure.

Nous considérons maintenant que les ordres $P[M_1], \ldots, P[M_\ell]$ sont quelconques et sont déjà codés respectivement par $s_1, \ldots, s_\ell$ bits. Ainsi, la fonction $\mathcal{GP}$ peut calculer la taille d'un codage par vecteur de bits de leur composition parallèle et donc de l'ordre SP.

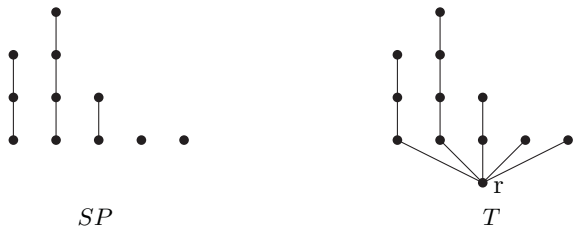


FIGURE 2.5: À gauche, une composition parallèle de chaînes SP. À droite, un arbre T avec $dim_2(SP) = dim_2(T)$.

Proposition 2.1.9 (Composition parallèle d'ordres)**.** *Soit P un ordre avec $\{M_1, \ldots, M_\ell\}$ sa partition modulaire et l'antichaîne $\{M_1, \cdots, M_\ell\}$ son ordre quotient. Soit s_i la taille d'un codage par vecteur de bits de $P[M_i]$, pour i entre 1 et ℓ. Alors,*

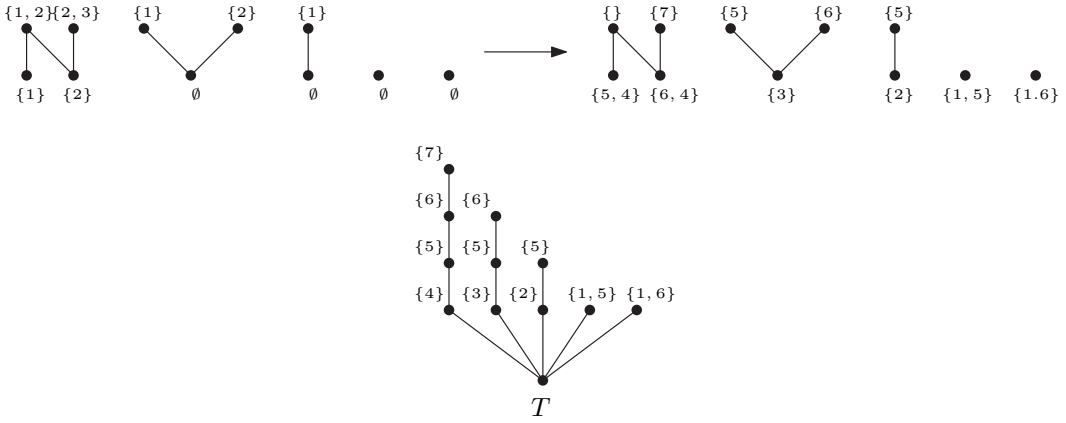
$$\dim_2(P) \leq \mathcal{GP}([s_1, \cdots, s_\ell]).$$

La Proposition 2.1.9 donne la taille d'un codage par vecteur de bits d'une composition parallèle d'ordres déjà codés. L'Algorithme 2 calcule ce codage que nous illustrons par la Figure 2.6.

Données : Des ordres $P_1, \ldots, P_\ell$ déjà codés par l'ensemble $[\![1, s_i]\!]$ si $|P_i| > 1$, et par $\emptyset$ sinon, pour i de 1 à ℓ
Résultat : Un codage de $\sum_{i=1}^{\ell} P_i$

$E \leftarrow 0$
pour *tout i allant de 1 à ℓ* **faire**
  `/* Construction d'ordre linéaire de taille` $s_i + 1$ `réduit par transitivité,` $s_i = 0$ `si` $|P_i| = 1$ `*/`
  $X \leftarrow \{1 + |E|, \ldots, s_i + 1 + |E|\}$
  $R \leftarrow \cup\{(k, k+1) : k \in [\![1 + |E|, s_i + |E|]\!]\}$
  $C_i \leftarrow (X, R)$
  `/*` $1, \ldots |E|$ `les éléments des ordres` $C_1, \ldots, C_i$ `*/`
  $E \leftarrow E + s_i + 1$
fin
Codage *Polycotomique Généralisé* de l'arbre $T = \{r\}.(\sum_{i=1}^{\ell} C_i)$
pour *tout i allant de 1 à ℓ* **faire**
  $Min \leftarrow$ code du minimum de C_i
  $Max \leftarrow$ code du maximum de C_i
  $S \leftarrow Max \backslash Min$
  pour *tout x dans P_i* **faire**
    $code \leftarrow \emptyset$
    pour *tout c dans $code(x)$* **faire**
      $code \leftarrow code \cup S[c]$
    fin
    $code(x) \leftarrow code \cup Min$
  fin
fin

Algorithme 2 : Codage d'une composition parallèle d'ordres



Codage réduit de T par
Polycotomique Généralisé

FIGURE 2.6: À gauche, des ordres déjà codés. À droite, le codage réduit par vecteur de bits de leur composition parallèle déduit du codage de l'arbre T.

1.3 Description de l'heuristique

Comme la complexité de calcul de la 2-dimension des ordres séries-parallèles est ouverte, nous proposons alors une heuristique qui calcule leur codage par vecteur de bits en se basant sur la technique de décomposition modulaire. Notons que la taille de ce codage constitue une borne supérieure de la 2-dimension de ces ordres.

Soit SP un ordre série-parallèle et soit T son arbre de décomposition modulaire. Pour tout élément x de T, notons $F(x)$ l'ensemble des feuilles appartenant à la liste des successeurs de x dans T. Nous pouvons alors associer à x le sous-ordre de SP défini sur $F(x)$. La partition modulaire maximale de ce sous-ordre est $\{F(y_1), \ldots, F(y_k)\}$, les éléments $y_1, \ldots, y_k$ étant les fils de x dans T (Figure 2.7).

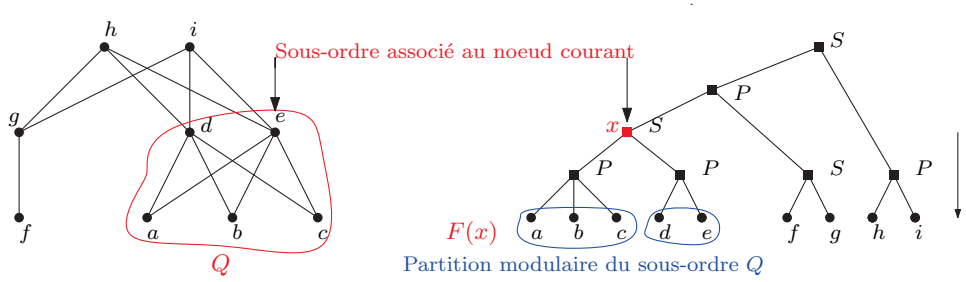


FIGURE 2.7: Le sous-ordre Q associé au nœud x et $F(x) = \{a, b, c, d, e\}$

L'heuristique que nous proposons pour calculer un codage par vecteur de bits de SP, génère d'abord l'arbre T, puis, suivant un ordre topologique inverse de ses éléments, elle calcule, pour tout élément x de T, un codage par vecteur de bits du sous-ordre associé à x. Ce codage est calculé par l'Algorithme 1 si le nœud x correspond à une composition série. Il est calculé par l'Algorithme 2, lorsque le nœud x correspond à une composition parallèle. Les feuilles sont initialement codées par l'ensemble vide. Au final, nous obtenons un codage de SP. Dans les sous-sections qui suivent, nous prouvons qu'il s'agit d'un codage par vecteur de bits. Nous donnons également les algorithmes nécessaires pour calculer ce codage et leur complexité. La Figure 2.8 synthétise le processus de l'heuristique proposée.

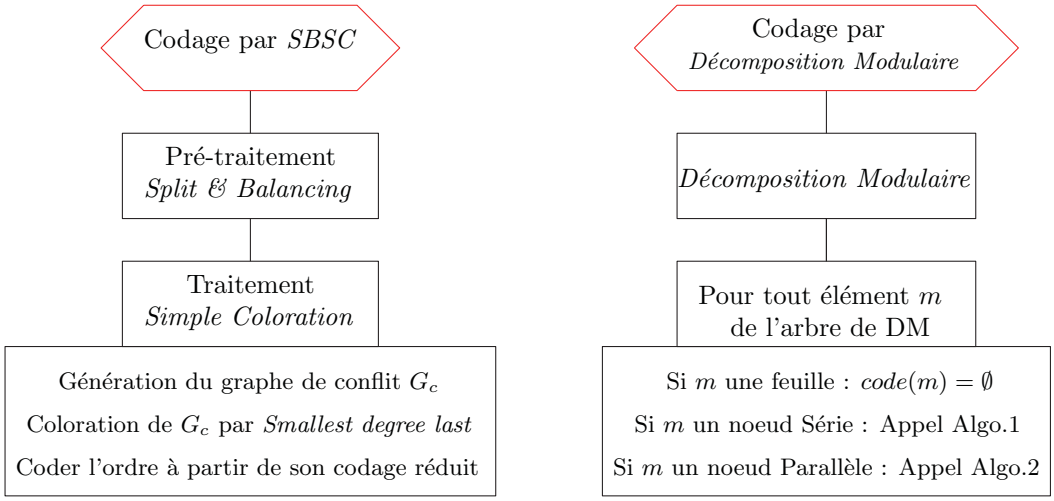


FIGURE 2.8: À gauche, le processus de l'heuristique de référence *SBSC*. À droite, le processus de la nouvelle heuristique basée sur la décomposition modulaire.

1.4 Algorithmes et complexités

Dans cette sous-section, nous donnons les instructions de calcul d'un codage des ordres séries-parallèles suivant le processus de l'heuristique proposée dans la sous-section précédente. Nous divisons l'ensemble de ces instructions en plusieurs algorithmes : algorithme de construction du graphe de comparabilité d'un ordre, algorithme de génération de son arbre de décomposition modulaire ainsi qu'un algorithme de codage de l'ordre. Nous évaluons également la complexité de chacun de ces algorithmes.

1.4.1 Construction du graphe de comparabilité

L'heuristique de codage des ordres séries-parallèles, proposée dans cette section, repose sur la technique de décomposition modulaire. Cette technique est généralement connue pour les graphes non orientés. Cependant, elle s'étend naturellement aux ordres – graphes orientés – grâce à leur graphe de comparabilité.

Un *graphe de comparabilité* est un graphe non orienté admettant une orientation transitive, ce qui peut induire un ordre sur les éléments du graphe. Réciproquement, tout ordre (X, R) est représentable par un graphe de comparabilité unique $(X, R' \cup R'^{-1})$, R' étant la fermeture transitive de R et R'^{-1} la fermeture symétrique de R'. Ainsi, la décomposition modulaire d'un ordre n'est autre que la décomposition modulaire de son graphe de comparabilité.

Données : Un ordre $P = (X, R)$
Résultat : Le graphe de comparabilité de P
`/* Algorithme trivial de fermeture transitive */`
1 $R' \leftarrow R$
2 **pour** *tout* k *dans* X **faire**
3 | **pour** *tout* i *dans* X **faire**
4 | | **pour** *tout* j *dans* X **faire**
5 | | | **si** (i, k) *dans* R *et* (k, j) *dans* R **alors**
6 | | | | $R' \leftarrow R' \cup (i, j)$
7 | | | **fin**
8 | | **fin**
9 | **fin**
10 **fin**
`/* Calcul de fermeture symétrique */`
11 $R'^{-1} \leftarrow \emptyset$
12 **pour** *tout* (x, y) *dans* R' **faire**
13 | $R'^{-1} \leftarrow R'^{-1} \cup \{(y, x)\}$
14 **fin**
`/* Le graphe de comparabilité */`
15 retourner $(X, R' \cup R'^{-1})$

Algorithme 3 : Graphe de comparabilité d'un ordre

L'Algorithme 3 construit le graphe de comparabilité d'un ordre, présenté par une matrice d'adjacence, en temps polynomial. En effet, il calcule la fermeture transitive (lignes $1 - 10$) d'un ordre de taille n en temps $\mathcal{O}(n^3)$, ainsi que sa fermeture symétrique (lignes $11-14$) en temps $\mathcal{O}(n^2)$ (réellement en temps $\mathcal{O}(|R'|)$ avec $|R'| \leq n^2$). Nous présentons dans la suite un algorithme plus efficace de fermeture transitive d'ordres.

Algorithme efficace de fermeture transitive Pour calculer la fermeture transitive d'un graphe orienté acyclique de taille n, nous disposons de l'algorithme de Goralčíková et Koubek [Goralčíková et Koubek, 1979]. En notant m_{tc} le nombre d'arcs de la fermeture transitive du graphe, l'algorithme en question a une complexité arithmétique $\mathcal{O}(m_{tc}^{\frac{3}{2}} \log_2(n))$ [Borassi *et al.*, 2016], et son pseudo code est incorporé dans l'Algorithme 4 (les lignes : 1,3,4,8,9 et 12).

Données : Un graphe orienté acyclique $D = (V, N)$, représenté par liste des successeurs immédiats $N(v)$ pour tout v dans V
Résultat : La fermeture transitive $D' = (V, N')$ de D, représentée par liste des successeurs $N'(v)$ pour tout v dans V.

```
1  Calculer un ordre topologique (v_1, ..., v_n) des sommets de V
2  Ctrl ← [0]^n
3  pour i allant de n à 1 faire
4  |  N'(v_i) ← N(v_i)
5  |  pour tout x dans N'(v_i) faire
6  |  |  Ctrl[x] ← 1
7  |  fin
8  |  pour tout w dans N(v_i) faire
9  |  |  pour tout x dans N'(w) faire
10 |  |  |  si Ctrl[x] = 0 alors
11 |  |  |  |  Ctrl[x] ← 1
12 |  |  |  |  N'(v_i) = N'(v_i) ∪ {x}
13 |  |  |  fin
14 |  |  fin
15 |  fin
16 |  pour tout x dans N'(v_i) faire
17 |  |  Ctrl[x] ← 0
18 |  fin
19 fin
```

Algorithme 4 : Algorithme efficace de fermeture transitive

L'algorithme de Goralčíková et Koubek calcule la fermeture transitive d'un graphe en récupérant, pour chacun de ses éléments, la liste de ses successeurs. Il s'agit de la liste de ses successeurs immédiats (instruction en ligne 4) à laquelle s'ajoute tout successeur déduit par transitivité (instruction en ligne 12). Afin d'éviter la duplication d'éléments dans la liste, Borassi *et al.* supposent une implémentation de cette dernière par un arbre binaire de recherche équilibré. Ainsi, il est possible, en temps $\mathcal{O}(\log_2(n))$, n étant la taille du graphe, de vérifier l'existence d'un élément y dans la liste des successeurs d'un élément x, avant de l'y ajouter.

En supposant que les éléments du graphe en entrée sont des entiers de 1 à n (sinon les numéroter de 1 à n en un seul parcours du graphe), nous proposons de contrôler la duplication d'éléments par un vecteur `Ctrl[]` de n bits (voir les instructions en lignes : 2,5,6,10,11,16 et 17). Nous vérifions alors, en temps constant, si y appartient à la liste des successeurs de x à travers un accès direct au $y^{\text{ème}}$ bit du vecteur `Ctrl[]`. Nous diminuons ainsi la complexité en temps de l'algorithme de fermeture transitive proposé par Goralčíková et Koubek [Goralčíková et Koubek, 1979]. Le Théorème 2.1.13 affirme que l'Algorithme 4 améliore effectivement cette complexité.

Théorème 2.1.13. *L'Algorithme 4 calcule une fermeture transitive d'un graphe orienté acyclique de taille n en temps $\mathcal{O}(m_{tc}^{\frac{3}{2}} + n)$, m_{tc} étant le nombre d'arc dans la fermeture transitive.*

Démonstration. La correction de l'algorithme est due à l'ordre de traitement des éléments du graphe [Borassi *et al.*, 2016]. Nous calculons maintenant sa complexité.

Les instructions élémentaires 6 et 17 sont répétées $m_{tc} + n$ fois : le nombre des successeurs d'un élément (les boucles en lignes 5 et 16), pour tout élément du graphe (la boucle en ligne 3).

L'analyse de la complexité des instructions restantes est déjà donnée dans [Borassi *et al.*, 2016]. Cette analyse considère une implémentation des listes des successeurs par des arbres binaires de recherche équilibrés. Nous reprenons cette analyse en considérant une implémentation par des listes simplement chaînées.

En notant m le nombre d'arcs du graphe en entrée, la complexité du calcul du tri topologique est $\mathcal{O}(n + m)$ (ligne 1).
L'initialisation du vecteur de contrôle `Ctrl[]` se fait en $\mathcal{O}(n)$ (ligne 2)
L'instruction en ligne 4 correspond à un parcours de la liste des successeurs immédiats de chaque élément. Sa complexité vaut alors $\mathcal{O}(n + m)$.
Notons $n_{out}(w)$ (resp. $n_{in}(w)$) le degré sortant (resp. le degré entrant) de l'élément w dans la fermeture transitive du graphe. Ainsi, les instructions élémentaires 10, 11 et 12 sont exécutées $\sum_{w \in V} n_{in}(w) n_{out}(w)$ fois. En effet, puisque w est un successeur de $n_{out}(w)$ éléments, la boucle 9 sera alors appelée $n_{out}(w)$ fois. À chaque fois, les instructions de cette boucle sont exécutées $n_{in}(w)$ fois. Notons que la somme $\sum_{w \in V} n_{in}(w) n_{out}(w)$ correspond au nombre de triangles dans la fermeture transitive du graphe. Ce nombre vaut $\mathcal{O}(m_{tc}^{\frac{3}{2}})$ [Itai et Rodeh, 1978].

La complexité en temps totale de l'Algorithme 4 est $\mathcal{O}(m_{tc}^{\frac{3}{2}} + n)$. □

Remarque Nous précisons que la complexité de l'Algorithme 4 est $\mathcal{O}(m_{tc}^{\frac{3}{2}})$, si le graphe en entrée est connexe ($n \leq m_{tc}$). Nous précisons qu'il s'agit d'une complexité arithmétique.

1.4.2 Génération de l'arbre de décomposition modulaire

Nous calculons l'arbre de décomposition modulaire d'un graphe par un algorithme linéaire, dû à Capelle *et al.* [Capelle *et al.*, 2002]. Cet algorithme utilise la notion de *permutation factorisante*. Une permutation factorisante d'un graphe est un ordre total O sur ses sommets, vérifiant pour toute paire d'éléments x et y, appartenant respectivement aux modules forts maximaux M_i et M_j, $x \prec_O y$ si et seulement si $i = j$, ou pour tout élément $z \leq_O x$ (resp. $y \leq_O z$), z n'appartient pas à M_j (resp. M_i). Nous référons le lecteur à la thèse de De Montgolfier [Montgolfier, 2003] pour plus de détails sur cette approche.

1.4.3 Codage des ordres séries-parallèles par vecteur de bits

Nous donnons ci-dessous un algorithme de codage des ordres séries-parallèles par vecteur de bits puis nous calculons sa complexité. La correction de l'algorithme est prouvée dans la sous-section suivante.

Données : Un ordre série-parallèle P
Résultat : Un codage de P *via* la décomposition modulaire
1 $G_c \leftarrow$ graphe de comparabilité de P
2 $\mathcal{T} \leftarrow$l'arbre de décomposition modulaire de P
3 $\mathcal{M} \leftarrow$ ordre topologique inverse des éléments de T
4 **pour** *tout* m *de* $\mathcal{M}$ **faire**
5 **si** *type du nœud* m *est feuille* **alors**
6 $code(m) = \emptyset$ // code de m
7 **fin**
8 **sinon si** *type du nœud* m *est Série* **alors**
9 Appel de l'algorithme 1 sur les sous-ordres associés à chaque successeur immédiat de m
10 **sinon**
 `/* type du nœud` m `est Parallèle */`
11 Appel de l'algorithme 2 sur les sous-ordres associés à chaque successeur immédiat de m
12 **fin**
13 **fin**
14 **fin**

Algorithme 5 : Algorithme de codage des ordres séries-parallèles par vecteur de bits *via* la décomposition modulaire

Le Théorème 2.1.14 prouve que l'Algorithme 5 calcule un codage des ordres séries-parallèles en temps polynomial.

Théorème 2.1.14. *L'Algorithme 5 calcule un codage par vecteur de bits d'un ordre série-parallèle de taille n en temps $\mathcal{O}(n^3)$ et en espace $\mathcal{O}(n^2)$.*

Démonstration. La correction de l'algorithme est prouvée dans la sous-section suivante (voir la preuve de la Proposition 2.1.10). Nous dédions cette preuve à l'analyse de complexité.

La construction du graphe de comparabilité (ligne 1) fait appel à l'Algorithme 3. La complexité en temps de cet algorithme est $\mathcal{O}(m_c^{\frac{3}{2}} + n)$. En effet, il s'agit de la complexité en temps de la fermeture transitive (Théorème 2.1.13) vu que la fermeture symétrique s'exécute en $\mathcal{O}(m_c)$. La complexité en espace de l'algorithme est $\mathcal{O}(m_c + n)$. Pour construire l'arbre de décomposition modulaire (ligne 2), nous faisons appel à l'algorithme implémenté par De Montgolfier[1], qui construit cet arbre sous forme d'une matrice d'adjacence. Les complexités en temps et en espace de cet algorithme sont données par l'auteur et correspondent respectivement à $\mathcal{O}(m_c \log_2(n))$ et $\mathcal{O}(m_c)$. Un tri topologique inverse des éléments d'un arbre (ligne 3) est obtenu en inversant l'ordre de visite de ses éléments lors d'un parcours en largeur de l'arbre. Ce tri s'effectue en temps linéaire en la taille de l'arbre de décomposition modulaire qui vaut au plus $2n$, n étant la taille de l'ordre et également le nombre de feuilles de l'arbre. Notons que le nombre de nœuds internes d'un arbre de décomposition modulaire ne peut pas dépasser le nombre de ses feuilles. Ainsi, les instructions en lignes 6, 9, 11 s'exécutent $\mathcal{O}(n)$ fois. L'instruction en ligne 6 étant élémentaire, sa complexité en temps est $\mathcal{O}(1)$. L'Algorithme 1 (ligne 9) met à jour l'ensemble des couleurs de chaque élément de l'ordre en entrée. Comme la taille de l'ordre ainsi que l'ensemble des couleurs est au plus n, la complexité en temps et en espace de cet algorithme est en $\mathcal{O}(n^2)$. L'Algorithme 2 (ligne 11) construit, en temps linéaire, un arbre à au plus n éléments puis le code par l'algorithme *Polychotomique Généralisé* en temps et en espace $\mathcal{O}(n^2)$ dans le pire des cas [Colomb *et al.*, 2008] – cet espace est libéré à chaque itération (ligne 4). Enfin, il met à jours l'ensemble des couleurs associé à chaque élément de l'ordre en entrée. Comme la taille de l'ordre ainsi que l'ensemble des couleurs est au plus n, la complexité en temps et en espace de cet algorithme est $\mathcal{O}(n^2)$. Nous précisons que les Algorithmes 1 et 2 réutilisent le même espace mémoire à chaque itération (ligne 4). Il s'agit de l'espace mémoire dédié au stockage du code de chaque élément de l'ordre, et mis à jour à chaque appel de ces algorithmes.

Comme $m_c \leq n^2$, la complexité en temps de l'Algorithme 5 est alors en $\mathcal{O}(n^3)$ ainsi que sa complexité en espace est en $\mathcal{O}(n^2)$.

□

1. https://searchcode.com/file/47014363/sage/graphs/modular_decomposition/src/dm.c. Consulté en 2017

1.5 Résultats théoriques

Nous montrons dans cette sous-section que l'Algorithme 5 calcule un codage par vecteur de bits des ordres séries-parallèles (Proposition 2.1.10). Comme le processus de cet algorithme peut se ramener à celui de l'heuristique *Polychotomic Généralisé*, ils donnent naturellement les mêmes résultats pour la classe des arbres (sous-classe des ordres séries-parallèles). Nous en proposons tout de même une preuve (Proposition 2.1.11), pour déduire enfin que l'Algorithme 5 est une 4-approximation de la 2-dimension des arbres (Proposition 2.1.12).

Proposition 2.1.10. *Soit P un ordre série-parallèle. L'Algorithme 5 calcule un codage par vecteur de bits de P. En notant $dm(P)$ sa taille, nous avons $dim_2(P)$ vaut au plus $dm(P)$.*

Démonstration. Nous proposons une preuve par induction sur la taille de l'ordre.

Pour P un ordre réduit en un unique élément, l'Algorithme 5 le code par l'ensemble vide. La taille de ce codage est nulle et nous avons également $dim_2(P) = 0$. La conclusion de la proposition est alors vérifiée dans ce cas.

Pour P un ordre à n éléments, nous supposons que la propriété à démontrer est vraie pour tout sous-ordre de P. Afin de la prouver pour P, nous considérons ϕ une fonction définie sur les éléments de P telle que pour tout élément x, $\phi(x)$ est l'ensemble de couleurs codant x lors du codage de P par l'Algorithme 5. Nous devons alors montrer que $x \leq_P y$ si et seulement si $\phi(x) \subseteq \phi(y)$, x et y étant deux éléments de P.

Soit $\mathcal{T}$ l'arbre de décomposition modulaire de P. Les feuilles de $\mathcal{T}$ sont alors les éléments de P. Soit z le plus petit prédécesseur commun à x et à y dans $\mathcal{T}$. Notons P_z le sous-ordre de P associé à z (Figure 2.7). Soit $\phi_z(x)$ et $\phi_z(y)$ les couleurs associées à x et à y respectivement, lors du codage de P_z.

Notons que le codage d'une composition parallèle de P_z et d'autres sous-ordres de P, qui sont tous déjà codés, consiste à ajouter aux codes des éléments de chaque sous-ordre (composante connexe) un ensemble de couleurs identique (Algorithme 2). Il en est de même pour une composition série de P_z et d'autres sous-ordres de P (Algorithme 1). Rappelons que x et y, étant dans un même module, ont les mêmes prédécesseurs dans $P \backslash P_z$. Nous en déduisons que $\phi(x) \backslash \phi_z(x) = \phi(y) \backslash \phi_z(y)$.

Si z n'est par la racine de $\mathcal{T}$, alors $x \leq_P y \Leftrightarrow \phi_z(x) \subseteq \phi_z(y)$ (par induction). Cela est équivalent à $\phi(x) \subseteq \phi(y)$.

Sinon, nous avons deux cas : (1) le nœud z est parallèle, alors x et y sont codés par le *Polychotomique Généralisé* qui assure un codage par vecteur de bits ; (2) le nœud z est série donc x et y forment une chaîne. Il est facile de constater que l'Algorithme 1 assure un codage par vecteur de bits des chaînes (lignes 13 et 14).

Nous concluons que l'Algorithme 5 calcule un codage par vecteur de bits de P de taille $dm(P)$. Comme $dim_2(P)$ est la taille minimale d'un codage par vecteur de bits de P, alors $dim_2(P) \leq dm(P)$. □

Nous venons de prouver la correction de l'Algorithme 5. La sous-section suivante présente des résultats expérimentaux montrant que cet algorithme améliore la taille du codage des ordres séries-parallèles par vecteur de bits. Cependant, pour le cas particulier des arbres, nous constatons que les résultats obtenus sont les mêmes que ceux calculés par l'heuristique *Polychotomique Généralisé*. Nous prouvons théoriquement ce constat.

Proposition 2.1.11. *L'Algorithme 5 calcule un codage des arbres de même taille que celle de leur codage par le Polychotomique Généralisé.*

Démonstration. Pour cette preuve, nous notons $\mathcal{DM}$ (resp. $\mathcal{A}_{GP}$) la fonction qui prend en paramètre un arbre et calcule la taille de son codage par l'Algorithme 5 (resp. par le *Polychotomique Généralisé*). Soit T un arbre de racine r et soit $T_1, \ldots, T_k$ ses sous-arbres dont chacun est enraciné par un successeur immédiat de r.
Remarquons que $\mathcal{A}_{GP}(T) = \mathcal{GP}([\mathcal{A}_{GP}(T_1), \ldots, \mathcal{A}_{GP}(T_k)])$ (voir la brève description de l'heuristique *Polychotomique Généralisé* donnée à la page 68).

Pour démontrer la proposition, nous suggérons une preuve par induction sur la taille de T.
Pour T réduit à un singleton, nous avons $\mathcal{DM}(T) = \mathcal{A}_{GP}(T) = 0$ (l'élément r est codé par l'ensemble vide).
Supposons que la conclusion de la proposition est vraie pour tout sous-arbre de T.
Nous distinguons deux cas :

Cas.1 $k > 1$

Nous avons d'un côté $\mathcal{A}_{GP}(T) = \mathcal{GP}([\mathcal{A}_{GP}(T_1), \ldots, \mathcal{A}_{GP}(T_k)])$.
D'un autre côté, nous avons T une composition série de r et $\sum_{i=1}^{k} T_i$. Par conséquent, le codage de T par l'Algorithme 5 code r par l'ensemble vide et chaque élément de $T \backslash \{r\}$ par le code qui lui a été attribué lors du codage de la composition parallèle de $T_1, \ldots, T_k$ par l'Algorithme 2. Par conséquent, $\mathcal{DM}(T) = \mathcal{GP}([\mathcal{DM}(T_1), \ldots, \mathcal{DM}(T_k)])$.
Par hypothèse d'induction, nous avons $\mathcal{A}_{GP}(T_i) = \mathcal{DM}(T_i)$, pour tout i entre 1 et k. Nous concluons que $\mathcal{DM}(T) = \mathcal{A}_{GP}(T)$.

Cas.2 $k = 1$

nous avons d'une part $\mathcal{A}_{GP}(T) = \mathcal{A}_{GP}(T_1) + 1$. En effet, le poids attribué à r est le poids de son fils plus un. Nous mentionnons au lecteur que l'heuristique *Polychotomique Généralisé* se comporte comme l'algorithme *Dicho*, détaillé

dans le premier chapitre à la page 57, lorsqu'un nœud a au plus deux fils. Nous donnons ultérieurement (troisième section de ce chapitre) une description détaillée de l'heuristique *Polychotomique Généralisé*.

D'autre part, nous avons T une composition série d'une chaîne à deux éléments (r et son fils) et l'arbre T_1 privé de sa racine. Comme $\mathcal{DM}(T_1)$ est la taille du codage de T_1, $\mathcal{DM}(T) = \mathcal{DM}(T_1) + 1$ selon la Proposition 2.1.8 et l'Algorithme 1.

Par hypothèse d'induction, nous avons $\mathcal{A}_{GP}(T_1) = \mathcal{DM}(T_1)$. Nous concluons que $\mathcal{DM}(T) = \mathcal{A}_{GP}(T)$.

Pour $k = 0$, nous nous ramenons au cas du singleton (le cas de base de l'induction). Ayant étudié tous les cas possibles, ceci conclut la preuve de la Proposition 2.1.11. □

Nous donnons maintenant un facteur d'approximation de l'Algorithme 5 pour le problème de calcul de la 2-dimension des arbres.

Proposition 2.1.12. *L'Algorithme 5 est une 4-approximation de la 2-dimension des arbres.*

Démonstration. Soit T un arbre et soit $Dicho(T)$ la taille du codage de T par l'algorithme *Dicho*. Selon le Théorème 1.3.12, nous avons $Dicho(T) \leq 4dim_2(T)$.

Puisque l'heuristique *Polychotomique Généralisé* améliore le codage des arbres par l'algorithme *Dicho* la taille du codage calculé par cette heuristique vaut au plus $Dicho(T)$. Cette taille correspond à la taille du codage calculé par l'Algorithme 5 (Proposition 2.1.11). Ainsi, cet algorithme calcule un codage de T de taille au plus $4dim_2(T)$, et donc il est une 4-aproximation de la 2-dimension des arbres. □

1.6 Résultats numériques

Nous présentons ici les résultats des tests effectués pour mesurer la performance de l'heuristique de codage des ordres séries-parallèles que nous avons introduit dans cette section. Nous parlons également des données de test considérées.

1.6.1 Données de test

Les données de test considérées pour évaluer la performance de notre heuristique sont des ordres séries-parallèles aléatoires. Le générateur de ces ordres est développé par Kahn [Kahn, 2015] et consiste à construire un ordre série-parallèle à partir de son arbre de décomposition modulaire qui est généré aléatoirement. Cette construction est illustrée par les Figures 2.1 et 2.2. Afin de générer un arbre de décomposition modulaire aléatoire, il suffit de produire un arbre quelconque puis d'étiqueter ses nœuds internes soit par « Série » soit par « Parallèle ».

Le jeux de données que nous avons considérés comporte trois paquets d'instances d'ordres séries-parallèles. Un premier de 500 instances de taille petite (inférieure à 100), un second de 500 instances de taille moyenne (entre 100 et 10000) et un dernier de 150 instances de grande taille (entre 1000 et 10000).

1.6.2 Expérimentations

Les expérimentations réalisées portent sur un jeu de données de tailles variées, permettant de visualiser l'écart entre la taille du codage calculé par l'heuristique de référence *SBSC* et celle du codage calculé par notre heuristique. Nous inscrivons dans la Table 2.1 la moyenne des valeurs expérimentales des tests effectués. Nous utilisons l'acronyme $\mathcal{DM}$ pour faire référence à notre heuristique de codage *via* la décomposition modulaire.

Taille des instances	*SBSC*	$\mathcal{DM}$
≤ 100	13.43	10.91
$]100, 1000]$	21.2	16.13
$]1000, 10000]$	26	18.77

TABLE 2.1: *SBSC* vs $\mathcal{DM}$ sur des ordres séries-parallèles aléatoires de tailles variées (moyenne des résultats de 500 instances de taille au plus 1000 et 150 instances de taille plus que 1000).

Nous avons observé, à partir des tests effectués, que notre heuristique diminue significativement la taille du codage des ordres séries-parallèles ayant une largeur plus importante que la hauteur. Nous illustrons ces observations par un échantillon tiré de nos expérimentations (cf. la Table 2.2). Nous pouvons expliquer cela par le fait que notre heuristique améliore le codage des ordres parallèles, surtout les antichaînes, par rapport à l'heuristique de référence *SBSC*.

1.7 Synthèse

L'heuristique que nous proposons permet de calculer un codage des ordres séries-parallèles par vecteur de bits, en se basant sur la technique de décomposition modulaire. Son processus intègre l'heuristique de codage des arbres *Polychotomique Généralisé*, pour traiter la composition parallèle des ordres. C'est la raison pour laquelle notre heuristique appliquée aux arbres donne les mêmes résultats que s'ils

Taille	Hauteur	Degré entrant Max	*SBSC*	$\mathcal{DM}$
200	13	64	49	**34**
200	29	4	49	**47**
400	11	133	56	**33**
400	43	7	64	**62**
600	19	83	79	**49**
600	14	6	30	**26**
1000	22	140	90	**56**
1000	19	32	53	**39**
10000	6	6222	32	**19**
10000	27	11	41	**32**

TABLE 2.2: *SBSC* vs $\mathcal{DM}$ sur des ordres séries-parallèles de différentes caractéristiques.

sont codés par l'heuristique *Polychotomique Généralisé*. Ces résultats restent tout de même plus intéressants que ceux obtenus en codant les arbres par l'heuristique de référence *SBSC*.

Les tests montrent que notre heuristique améliore la taille du codage des ordres séries-parallèles et nous pensons que le point fort réside dans le fait qu'un codage d'une composition parallèle des ordres par une heuristique efficace de codage des arbres est plus intéressant que son codage par une *Simple Coloration* après un pré-traitement.

2 Cas général des ordres partiels

Nous traitons dans cette section le problème de calcul de la 2-dimension pour le cas général des ordres. Comme indiqué préalablement, ce problème admet un schéma de résolution qui repose sur la conception des heuristiques de codage par vecteur de bits.

Nous proposons dans cette section une heuristique qui calcule un codage des ordres par vecteur de bits en utilisant la technique de décomposition modulaire. Cette heuristique est une extension de celle proposée dans la section précédente pour le codage des ordres séries-parallèles. Nous décrivons cette heuristique et ses différentes variantes, puis nous donnons l'algorithme qui l'implémente et sa complexité. Nous justifions la correction de cet algorithme et enfin nous exposons les résultats de l'évaluation expérimentale de l'heuristique proposée.

2.1 Motivation

L'efficacité de l'heuristique de codage des ordres séries-parallèles *via* la technique de décomposition modulaire, observée dans la section précédente, nous motive à utiliser encore cette technique pour le codage des ordres dans le cas général. Notons que l'arbre de décomposition modulaire d'un ordre non série-parallèle possède au moins un nœud ni série ni parallèle. Par conséquent, le sous-ordre associé à ce nœud (Figure 2.7) ne peut être codé par aucun des deux algorithmes proposés pour le codage d'une composition série et parallèle d'ordres (Algorithmes 1 et 2 respectivement). Néanmoins, nous pouvons assurer son codage à l'aide de l'heuristique *SBSC – Split and Balancing, Simple Coloring* – présentée dans le premier chapitre (page 52). Nous en déduisons qu'un codage *via* la technique de décomposition modulaire peut être défini pour tout ordre. Nous détaillons juste après la démarche suivie pour calculer ce codage.

2.2 Stratégie

Soit P un ordre et soit $\mathcal{T}$ son arbre de décomposition modulaire. Nous supposons que la racine de $\mathcal{T}$ est étiquetée par « Premier » (ni « Série » ni « Parallèle », cf. page 16). Soit $\mathcal{M} = \{M_1, \ldots, M_\ell\}$ la partition modulaire de P. La Proposition 1.2.5 affirme que $dim_2(P) \leq dim_2(P_{/\mathcal{M}}) + \sum_{i=1}^{\ell} dim_2(P[M_i])$.

Ainsi, il existe un codage de P de taille $dim_2(P_{/\mathcal{M}}) + \sum_{i=1}^{\ell} dim_2(P[M_i])$. Par exemple celui qui code les ordres $P_{/\mathcal{M}}$ et $P[M_i]$, pour i de 1 à ℓ, par des ensembles de couleurs disjoints. Ce codage, illustré par la Figure 2.9, assigne à tout élément x de P, appartenant au module M_i, un code propre $\phi(M_i) \uplus \phi_i(x)$ avec ϕ et ϕ_i des fonctions calculant un codage par vecteur de bits de taille optimale de $P_{/\mathcal{M}}$ et $P[M_i]$ (pour i de 1 à ℓ) respectivement. Nous proposons dans la suite des améliorations portant sur la taille de ce codage.

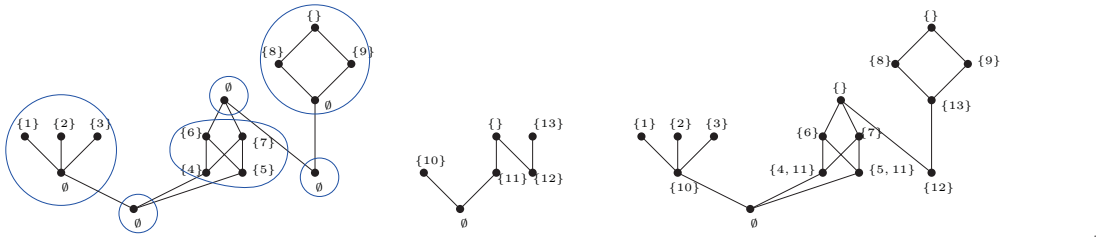


FIGURE 2.9: À gauche, un ordre P avec $\{M_1, \ldots M_6\}$ sa partition modulaire, et dont chaque sous-ordre $P[M_i]$, pour i entre 1 et 6, est doté d'un codage réduit. Au centre, le codage réduit de son ordre quotient, et à droite un codage réduit de P (Proposition 1.2.5).

2.2.1 Composition ni série ni parallèle d'ordres (nœud premier)

Soit M_i et M_j deux modules incomparables dans P. Soit x un élément de M_i et y un élément de M_j. Nous considérons le codage proposé au premier paragraphe de cette sous-section, qui code x par $\phi(M_i) \uplus \phi_i(x)$ et y par $\phi(M_j) \uplus \phi_j(y)$. Comme les ensembles $\phi(M_i)$ et $\phi(M_j)$ sont incomparables par inclusion, du fait que ϕ est un codage par vecteur de bits de $P_{/\mathcal{M}}$, nous en déduisons que $\phi(M_i) \uplus \phi_i(x)$ et $\phi(M_j) \uplus \phi_j(y)$ le sont également quelle que soit la valeur de $\phi_i(x)$ et $\phi_i(y)$. Par conséquent, nous pouvons coder les sous-ordres $P[M_i]$ et $P[M_j]$ par le même ensemble de couleurs ce qui peut réduire la taille du codage considéré. Nous définissons alors un nouveau codage de P où seuls ses sous-ordres définis sur des modules comparables sont codés par des ensembles de couleurs disjoints. La taille de ce codage, illustré par la Figure 2.10, est donnée dans la Proposition 2.2.13. Remarquez que ce codage améliore celui proposé dans la Figure 2.9.

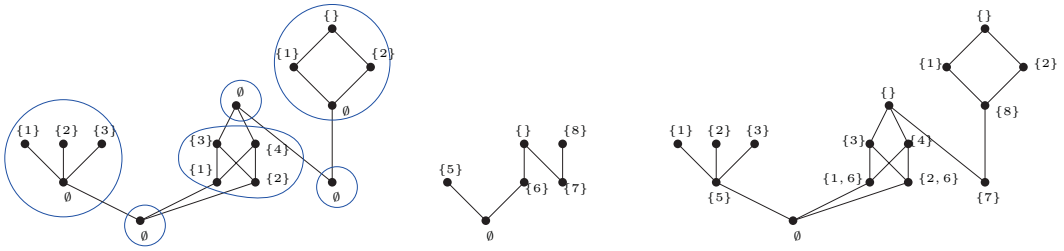


FIGURE 2.10: À gauche, un ordre P avec $\{M_1, \dots M_6\}$ sa partition modulaire, et dont chaque sous-ordre $P[M_i]$, pour i de 1 à 6, est doté d'un codage réduit. Au centre, le codage réduit de son ordre quotient, et à droite un codage réduit de P (Proposition 2.2.13).

Proposition 2.2.13. *Pour P un ordre et $\mathcal{M}$ sa partition modulaire, nous avons*

$$dim_2(P) \leq dim_2(P_{/\mathcal{M}}) + \max_{\substack{C \text{ chaîne} \\ \text{de } P_{/\mathcal{M}}}} \sum_{M \in C} dim_2(P[M]).$$

Démonstration. Notons m la valeur $\max_{\substack{C \text{ chaîne} \\ \text{de } P_{/\mathcal{M}}}} \sum_{M \in C} dim_2(P[M])$.
Afin de prouver la proposition, nous considérons R le minimum de $P_{/\mathcal{M}}$ (le rajouter s'il n'existe pas), et w une fonction de $\mathcal{M}$ vers $\mathbb{N}$ telle que $w(R)$ est nul et $w(M)$ est égal à $dim_2(P[M]) + \max\{w(N) | N \prec_{P_{/\mathcal{M}}} M\}$ pour tout M dans $\mathcal{M}$.
Prouvons par induction que $\max\{w(M) | M \in \mathcal{M}\}$ est égal à m. Pour cela, il suffit de prouver que, pour tout M dans $\mathcal{M}$, $w(M)$ est égal à

$$\max\{\sum_{N \in C} dim_2(P[N]) | C \text{ une chaîne dans } P_{/\mathcal{M}} \text{ de } R \text{ vers } M\} \quad (1).$$

La propriété (1) est vraie si M n'a aucun prédécesseur, i.e. si $M = R$ car $w(R) = 0$. Pour M différent de R, nous supposons que la propriété est vraie pour tous ses prédécesseurs. Prouvons qu'elle est vraie même pour M.
Nous avons $w(M) = dim_2(P[M]) + \max\{w(N)|N \prec_{P_{/\mathcal{M}}} M\}$. Par hypothèse d'induction, nous obtenons

$$w(M) = dim_2(P[M]) + \max\{\max\{\sum_{K\in C} dim_2(P[K])|C \text{ une chaîne dans } P_{/\mathcal{M}} \text{ de } R \text{ vers } N\}|N \prec_{P_{/\mathcal{M}}} M\}. \text{ Alors,}$$

$w(M) = \max\{\sum_{K\in C} dim_2(P[K])|C$ une chaîne dans $P_{/\mathcal{M}}$ de R vers $M\}$. La propriété (1) étant vérifiée, nous en déduisons que $\max\{w(M)|M \in \mathcal{M}\}$ vaut $\max\{\max\{\sum_{K\in C} dim_2(P[K])|C$ une chaîne dans $P_{/\mathcal{M}}$ de R vers $M\}|M \in \mathcal{M}\}$, qui vaut m.

Maintenant, prouvons que $dim_2(P) \leq dim_2(P_{/\mathcal{M}}) + m$. Soit ϕ_0 et ϕ_M des codages par vecteur de bits de taille minimale de $P_{/\mathcal{M}}$ et $P[M]$ respectivement, M étant un élément de $\mathcal{M}$. Soit ϕ une fonction qui associe à tout élément x de M la concaténation des vecteurs de bits $\phi_0(M)$, $[0]^{m-w(M)}$, $\phi_M(x)$ et $[1]^{w(M)-dim_2(P[M])}$. Prouvons que pour toute paire d'éléments x et y de P, $x \leq_P y$ si et seulement si $\phi(x) \leq \phi(y)$. Soit M_x et M_y deux éléments de $\mathcal{M}$ contenant x et y respectivement.

Supposons que $x \leq_P y$. Si $M_x = M_y$, alors $\phi_0(M_x) = \phi_0(M_y)$ et $\phi_{M_x}(x) \leq \phi_{M_y}(y)$ ce qui implique $\phi(x) \leq \phi(y)$. Sinon, nous avons $M_x <_{P_{/\mathcal{M}}} M_y$ puisque M_x et M_y sont des modules contenant x et y respectivement avec $x \leq_P y$. Cela implique $\phi_0(M_x) < \phi_0(M_y)$ et il existe des éléments de $\mathcal{M}$ tels que $M_x \prec_{P_{/\mathcal{M}}} M_1 \preceq_{P_{/\mathcal{M}}} M_2 \cdots \preceq_{P_{/\mathcal{M}}} M_y$. Alors, $w(M_x) \leq w(M_1) - dim_2(P[M_1]) \leq w(M_1) \leq \cdots \leq w(M_y) - dim_2(P[M_y])$. Comme $w(M_x) \leq w(M_y) - dim_2(P[M_y])$, on en déduit que $\phi(x) \leq \phi(y)$. En effet,

- $\phi(x) = \phi_0(M_x).[0]^{m-w(M_x)}.X$, avec X un vecteur de $w(M_x)$ bits,
- $\phi(y) = \phi_0(M_y).[0]^{m-w(M_y)}.Y.[1]^{w(M_y)-dim_2(P[M_y])}$, avec Y un vecteur de bits de taille $dim_2(P[M_y])$,
- $|\phi(x)| = |\phi(y)|$ et $\phi_0(M_x) < \phi_0(M_y)$.

Réciproquement, supposons que $\phi(x) \leq \phi(y)$. Si nous avons $\phi_0(M_x) < \phi_0(M_y)$ alors $M_x <_{P_{/\mathcal{M}}} M_y$ implique $x <_P y$. Sinon, nous avons $\phi_0(M_x) = \phi_0(M_y)$ implique $M_x = M_y$. Donc, $\phi(x) \leq \phi(y)$ implique $\phi_{M_x}(x) \leq \phi_{M_y}(y)$. Par la définition du plongement ϕ_{M_x}, nous obtenons $x \leq_P y$.

Nous concluons que ϕ est un codage de P par vecteur de bits dont la taille vaut $dim_2(P_{/\mathcal{M}}) + m$. Ainsi, $dim_2(T) \leq dim_2(P_{/\mathcal{M}}) + m$, ce qui valide La Proposition 2.2.13.

□

2.2.2 *Blow up*

Nous améliorons le codage que nous venons de décrire dans le cas où P est une composition ni série ni parallèle d'ordres (page 83).

Supposons que M_1 est l'unique module non trivial de P (non réduit à un seul élément). D'après les deux Propositions 1.2.5 et 2.2.13, nous avons $dim_2(P)$ est au plus $dim_2(P_{/\mathcal{M}}) + dim_2(P[M_1])$. Ainsi, il existe un codage de P par vecteur de bits de taille $dim_2(P_{/\mathcal{M}}) + dim_2(P[M_1])$ qui, à chaque élément x de M_i, associe le code propre $\phi(M_i) \uplus \phi_i(x)$, avec ϕ et ϕ_i des fonctions calculant un codage par vecteur de bits de taille optimale de $P_{/\mathcal{M}}$ et $P[M_i]$ (pour i de 1 à ℓ) respectivement (Figure 2.11). Notons que $\phi_i(x) = \emptyset$ pour $i > 1$.

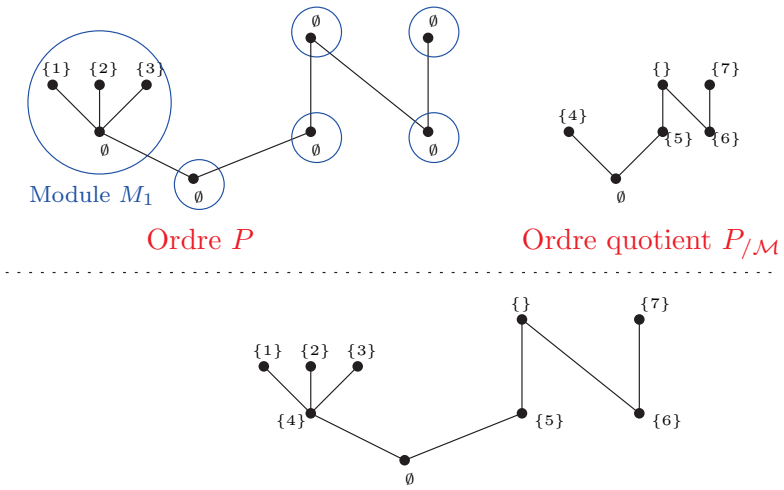


FIGURE 2.11: Un codage d'un ordre P, ayant un unique module non trivial, *via* la décomposition modulaire (Propositions 1.2.5 et 2.2.13)

Remarquons que ce codage utilise des ensembles de couleurs disjoints pour coder les ordres $P_{/\mathcal{M}}$ et $P[M_1]$. Nous proposons alors de réutiliser quelques couleurs intervenant dans le codage de $P[M_1]$ pour coder $P_{/\mathcal{M}}$. Afin de garantir cela, nous introduisons la notion du *Blow up* d'un ordre (Définition 2.2.6) que nous illustrons par la Figure 2.12. Cette solution s'applique même aux ordres ayant plusieurs modules non triviaux. Elle permet ainsi d'utiliser certaines couleurs pour coder à la fois l'ordre quotient $P_{/\mathcal{M}}$ et les sous-ordres $P[M_i]$, pour M_i non trivial avec i entre 1 et ℓ.

Définition 2.2.6 (*Blow up*)**.** *Soit P un ordre et soit $\mathcal{M}$ sa partition modulaire et $P_{/\mathcal{M}}$ son ordre quotient. Le* Blow up *de P, noté $\mathcal{B}(P)$, est l'ordre obtenu à partir de $P_{/\mathcal{M}}$ en substituant chaque module M de $\mathcal{M}$ par une chaîne de $dim_2(P[M])+1$ éléments.*

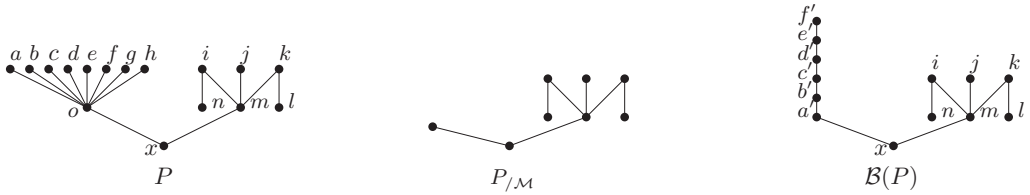


FIGURE 2.12: À gauche, un ordre P dont la partition modulaire (maximale) est $\mathcal{M} = \{\{o, a, b, c, d, e, f, g, h\}, \{x\}, \{i\}, \{j\}, \{k\}, \{l\}, \{m\}, \{n\}\}$. Au centre $P_{/\mathcal{M}}$ et à droite $\mathcal{B}(P)$.

Nous montrons comment définir un codage par vecteur de bits de P à partir du codage de son *Blow up*. Il s'agit désormais du codage de P *via* la décomposition modulaire de plus petite taille (Proposition 2.2.14). Ce codage, illustré par la Figure 2.13, est décrit par l'Algorithme 6.

Proposition 2.2.14 (Bornes du *Blow up*)**.** *Soit P un ordre et soit $\mathcal{M}$ sa partition modulaire et $\mathcal{B}(P)$ son* Blow up*. Alors*

$$dim_2(P) \leq dim_2(\mathcal{B}(P)) \leq dim_2(P_{/\mathcal{M}}) + \max_{\substack{C \text{ chaîne} \\ \text{de } P_{/\mathcal{M}}}} \sum_{M \in C} dim_2(P[M])$$

Démonstration. Pour cette preuve, nous notons C_M la chaîne par laquelle un élément M de $P_{/\mathcal{M}}$ était substitué lors de la construction du *Blow up* de P. Rappelons que $|C_M| = dim_2(P[M]) + 1$. Soit inf_M et sup_M les éléments minimum et maximum de C_M respectivement.

Prouvons d'abord la borne inférieure.
Soit S un ensemble de $dim_2(\mathcal{B}(P))$ éléments et soit ϕ un plongement de $\mathcal{B}(P)$ dans 2^S. Pour M un élément de $\mathcal{M}$, notons S_M l'ensemble $\phi(sup_M)$ privé des éléments $\phi(inf_M)$. Nous considérons ϕ_M un plongement de $P[M]$ dans 2^{S_M}, qui existe puisque $dim_2(P[M]) \leq |S_M|$. En effet, C_M étant une chaîne codée par S_M couleurs, il est évident que $dim_2(C_M) = |C_M| - 1 \leq |S_M|$.
Nous considérons ϕ', le plongement qui, à tout élément x de P appartenant à un module M, associe l'union de $\phi(inf_M)$ et $\phi_M(x)$. Remarquons que $\phi'(x)$ est inclus dans $\phi(sup_M)$.

Montrons que pour toute paire d'éléments x et y appartenant respectivement aux modules M et N de $\mathcal{M}$, $x \leq_P y$ si et seulement si $\phi'(x) \subseteq \phi'(y)$.

Supposons que $x \leq_P y$. Nous avons
- $\phi'(x) = \phi(inf_M) \uplus \phi_M(x)$,
- $\phi'(y) = \phi(inf_N) \uplus \phi_N(x)$.

Si $M = N$, alors $\phi(inf_M) = \phi(inf_N)$ et $\phi_M(x) \subseteq \phi_M(y)$ (par définition du plongement ϕ_M). Cela implique $\phi'(x) \subseteq \phi'(y)$.
Sinon, nous avons $M <_{P/\mathcal{M}} N$, car M et N sont des modules contenant respectivement x et y avec $x \leq_P y$. Cela implique $sup_M \leq_{\mathcal{B}(P)} inf_N$ ce qui implique $\phi(sup_M)$ est inclus dans $\phi(inf_N)$. Comme $\phi'(x) \subseteq \phi(sup_M)$ et $\phi(inf_N) \subseteq \phi'(y)$, nous en déduisons que $\phi'(x) \subseteq \phi'(y)$.

Réciproquement, supposons que $\phi'(x) \subseteq \phi'(y)$.
Comme $\phi(inf_M) \subseteq \phi'(x) \subseteq \phi'(y) \subseteq \phi(sup_N)$, nous avons $\phi(inf_M) \subseteq \phi(sup_N)$. Cela implique $inf_M \leq_{\mathcal{B}(P)} sup_N$. En associant inf_M et sup_N à un élément de M et N respectivement, nous obtenons $M \leq_{P/\mathcal{M}} N$. Par conséquent, $x \leq_P y$.

On conclut que ϕ' est un plongement de P dans 2^S. Ainsi, $dim_2(P) \leq |S|$. Comme $|S| = dim_2(\mathcal{B}(P))$, la borne inférieure est alors prouvée.

Passons maintenant à la borne supérieure.
Soit S un ensemble de $dim_2(P_{/\mathcal{M}})$ éléments et ϕ un plongement de $P_{/\mathcal{M}}$ dans 2^S.
Pour tout module M de $\mathcal{M}$, notons S_M l'ensemble $\{1, \ldots, dim_2(P[M])\}$.
Comme $|S_M| = dim_2(P[M])$, nous avons $|S_M| = |C_M| - 1$. Nous considérons alors ϕ_{C_M} le plongement de la chaîne C_M dans 2^{S_M}.
Soit $\mathcal{S}$ un ensemble d'éléments de taille $\max_{\substack{C \text{ chaîne} \\ \text{de } P_{/\mathcal{M}}}} \sum_{M \in C} |S_M|$.
Nous considérons ϕ' le plongement de $\mathcal{B}(P)$ dans $2^{S \uplus \mathcal{S}}$ qui, à chaque élément m de $\mathcal{B}(P)$ appartenant à C_M, associe

$$\phi_{C_M}(m) \uplus \phi(M) \uplus \{\phi'(k) : k <_{\mathcal{B}(P)} m\}.$$

Il est facile de vérifier que ϕ' est un codage par vecteur de bits de $\mathcal{B}(P)$. Cela signifie que $dim_2(\mathcal{B}(P)) \leq |S \uplus \mathcal{S}| = |S| + |\mathcal{S}|$, ce qui justifie la borne supérieure.

La Proposition 2.2.14 est maintenant prouvée.

□

L'Algorithme 6 permet de calculer un codage d'un ordre à partir du codage de son *Blow up*.

Données : Un ordre P et sa partition modulaire $\mathcal{M}$, avec $P[M]$ déjà codé par l'ensemble des entiers $[\![1, s_M]\!]$, pour tout M de $\mathcal{M}$.
L'ordre $\mathcal{B}(P)$ déjà codé avec inf_M (resp. sup_M) correspond à l'élément minimum (resp. maximum) de la chaîne de $\mathcal{B}(P)$ remplaçant le module M
Résultat : Codage de P

```
pour tout M de M faire
    pour tout x dans P[M] faire
        S_M ← code(sup_M)\code(inf_M)
        Old ← code(x)
        pour tout k dans Old faire
            code(x) ← S_M[k]
        fin
        code(x) ← code(inf_M) ⊎ code(x)
    fin
fin
```

Algorithme 6 : Codage d'un ordre à partir du codage de son *Blow up*

Nous appliquons l'Algorithme 6 à l'ordre déjà codé dans la Figure 2.11 afin d'illustrer l'intérêt du *Blow up*. Le codage obtenu est présenté par la Figure 2.13. Remarquez que la taille de ce codage est effectivement plus petite que celle du précédent.

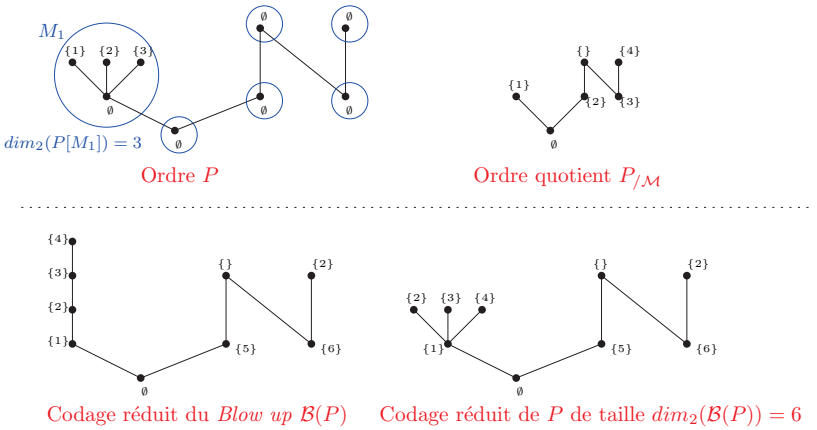


FIGURE 2.13: Une amélioration du codage présenté par la Figure 2.11 grâce à la notion du *Blow up*

Le *Blow up* d'un ordre étant d'une structure quelconque, nous proposons de le coder par l'heuristique de référence *SBSC – Split and Balancing, Simple Coloring*.

2.2.3 Quelques notes sur l'heuristique *SBSC*

Les expérimentations réalisées par Krall *et al.* [Vitek *et al.*, 1997] sur des hiérarchies connues, montrent que la phase du *Split and Balancing* réduit considérablement la taille du codage de ces hiérarchies par une *Simple Coloration*. Or, nous pouvons construire des instances *ad hoc* pour lesquelles la phase du *Split and Balancing* dégrade la taille du codage (Figure 2.15). Toutefois, cette phase de pré-traitement reste primordiale lorsqu'il s'agit de coder des instances de taille importante.

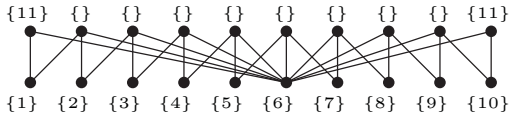


FIGURE 2.14: Un ordre dont le codage par une *Simple Coloration* sans le *Split and Balancing* nécessite 11 bits, tandis que son codage par *SBSC* nécessite 12 bits.

2.3 Description de l'heuristique

Le codage des ordres à travers la décomposition modulaire était d'abord conçu pour la classe des ordres séries-parallèles (voir la première section de ce chapitre). Nous l'étendons ici pour le codage de tout ordre. Rappelons que ce codage génère, en premier temps, l'arbre de décomposition modulaire d'un ordre. Ensuite, il calcule un codage du sous-ordre associé à chaque nœud de l'arbre (Figure 2.7), visité par ordre topologique inverse, soit par l'Algorithme 1, si le nœud visité est « Série », soit par l'Algorithme 2 s'il est « Parallèle ». Lorsqu'il s'agit d'un ordre non forcément série-parallèle, il est possible de visiter un nœud « Premier » (ni « Série » ni « Parallèle »). Dans ce cas, nous proposons de coder le *Blow up* du sous-ordre associé à ce nœud par l'heuristique de référence *SBSC – Split and Balancing, Simple Coloring* – puis d'appeler l'Algorithme 6 pour déduire un codage de ce sous-ordre.

Nous constatons que cette heuristique augmente la taille du codage des ordres ayant un sous-ordre, associé à un nœud « Premier », de taille importante. Nous supposons que cela est lié à la phase du *Split and Balancing*. En effet, nous observons que cette phase est plus significative si elle est appliquée à un ordre plutôt qu'à ses sous-ordres. Nous illustrons cette observation dans la Figure 2.15, où la phase du *Split and Balancing* du sous-ordre P' de P est sans effet.

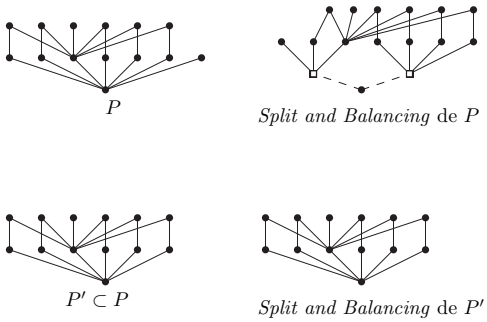


FIGURE 2.15: Le *Split and Balancing* d'un ordre et son sous-ordre

Nous proposons alors deux options de notre heuristique de codage des ordres *via* la décomposition modulaire. Une première, appelée *Global Split and Balancing* ou GSP, qui applique la phase du pré-traitement une seule fois à l'ordre initial. Puis une seconde, appelée *Local Split and Balancing* ou LSP, qui applique le pré-traitement localement à chaque sous-ordre associé à un nœud « Premier ». La Figure 2.16 synthétise le processus des deux options de l'heuristique de codage des ordres à travers la décomposition modulaire.

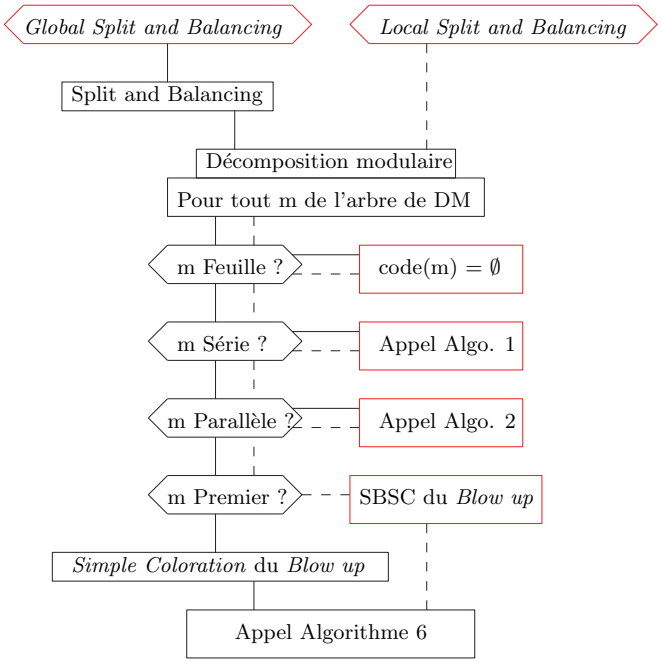


FIGURE 2.16: Processus de l'heuristique de codage des ordres *via* la décomposition modulaire

2.4 Algorithmes et complexités

Nous donnons un algorithme qui calcule un codage des ordres par vecteur de bits suivant le processus de l'heuristique tout juste décrite. Nous montrons que cet algorithme s'exécute en temps polynomial mais nous justifions sa correction dans la sous-section qui suit. Comme l'heuristique fait intervenir la notion du *Blow up* d'un ordre, nous donnons d'abord l'algorithme permettant de le construire.

2.4.1 Construction du *Blow up*

Nous présentons l'algorithme qui permet de construire le *Blow up* d'un ordre P. Nous supposons connaître déjà la partition modulaire $\{M_1, \ldots, M_\ell\}$ de l'ordre ainsi qu'un codage de ses sous-ordres $P[M_1], \ldots, P[M_\ell]$.

Données : Les ordres P et $P_{/\mathcal{M}}$, la partition $\mathcal{M} = \{M_1, \ldots, M_\ell\}$ et les ordres $P[M_1], \ldots P[M_\ell]$ déjà codés par $s_1, \ldots, s_\ell$ bits
Résultat : $\mathcal{B}(\mathcal{P})$

pour *tout* i *dans* $[\![1, \ell]\!]$ **faire**
  Remplacer le module M_i dans $P_{/\mathcal{M}}$ par une chaîne de $s_i + 1$ éléments
  Sauvegarder les éléments minimum et maximum de cette chaîne dans les variables inf_i et sup_i respectivement
fin

Algorithme 7 : Construction du *Blow up* d'un ordre

2.4.2 Codage des ordres par vecteur de bits

Nous présentons l'algorithme de codage des ordres par vecteur de bits *via* la décomposition modulaire puis nous calculons sa complexité. Nous vérifions la correction de cet algorithme dans la sous-section suivante.

Données : Un ordre P
Résultat : Un codage de P par la décomposition modulaire

```
si il s'agit de l'option GSB alors
    P ← Pré-traitement Split and Balancing [Krall et al., 1997] de P
fin
G_c ← graphe de comparabilité de P
T ←l'arbre de décomposition modulaire de P
M ← ordre topologique inverse des éléments de T
pour tout m de M faire
    si type du nœud m est feuille alors
        code(m) = ∅ // code de m
    fin
    sinon si type du nœud m est Série alors
        Appel de l'algorithme 1 sur les sous-ordres associés à chaque
         successeur immédiat de m
        sinon si type du nœud m est Parallèle alors
            Appel de l'algorithme 2 sur les sous-ordres associés à chaque
             successeur immédiat de m
            sinon
                /* type du nœud est Premier                    */
                Q ← le sous-ordre associé à m
                B ← Blow up du Q (Appel de l'Algorithme 7)
                si il s'agit de l'option LSB alors
                    B ← Pré-traitement Split and Balancing de B
                fin
                Codage de B par Simple Coloration (Appel de
                 l'Algorithme 16)
                Appel de l'Algorithme 6 pour coder Q
            fin
        fin
    fin
fin
```

Algorithme 8 : Algorithme de codage des ordres par vecteur de bits *via* la décomposition modulaire

Nous affirmons, selon le Théorème 2.2.15, que l'Algorithme 8 calcule un codage des ordres par vecteur de bits en temps et en espace polynomiaux.

Théorème 2.2.15. *L'Algorithme 8 calcule un codage par vecteur de bits d'un ordre de taille n en temps $\mathcal{O}(n^5)$ et en espace $\mathcal{O}(n^2)$.*

Démonstration. La correction de l'algorithme est prouvée dans la sous-section suivante. Nous dédions cette preuve à l'analyse de complexité.

D'abord, on a le pré-traitement de Krall *et al.* [Krall *et al.*, 1997] qui s'effectue en temps et en espace $\mathcal{O}(n^2)$ (ligne 2), puis, de la ligne 4 à la ligne 14, on a les instructions du codage des ordres séries-parallèles de complexité en temps $\mathcal{O}(n^3)$ et en espace $\mathcal{O}(n^2)$ (Théorème 2.1.14). Enfin, afin de traiter les nœuds premiers, on effectue les instructions restantes au plus $\mathcal{O}(n)$ fois. En fait, elles sont effectuées au plus $\mathcal{O}(|\mathcal{T}|)$ avec $|\mathcal{T}| \leq 2n$ car $\mathcal{T}$ est un arbre à n feuilles dont chaque nœud interne est de degré au moins 2. Ces instructions sont :

- La récupération de l'ordre Q en temps et en espace $\mathcal{O}(n^2)$ (ligne 16). Nous considérons une représentation par matrice d'adjacence des ordres.
- La construction du *Blow up* de Q en temps et en espace $\mathcal{O}(n^2)$(ligne 17). Pour cela, il faut récupérer la taille du codage s_i de chaque sous-ordre $P[M_i]$ associé à un fils du nœud courant m. On suppose que la structure de données utilisée pour stocker l'arbre de décomposition modulaire mémorise la taille du codage du sous-ordre associé à chaque nœud de l'arbre. Ainsi, on considère une représentation du *Blow up* par matrice d'adjacence de taille au plus n^2, telle que ses s_1+1 premières lignes et colonnes correspondent au module M_1 qu'on désigne par le bloc M_1. Les s_2+1 lignes et colonnes suivantes correspondent au module M_2 qu'on désigne par le bloc M_2, et ainsi de suite. Tout bloc M_i correspond à une sous-matrice dont les éléments vérifient $a_{k,l\geq k} = 1$. Il s'agit de la matrice d'adjacence d'une chaîne (celle qui substitue M_i). Ainsi, pour tout $j \neq i$, si M_i est inférieur à M_j dans l'ordre quotient, alors le bloc (M_i, M_j) est une sous-matrice dont tous les éléments sont à 1, sinon, les éléments du bloc sont à 0. Une telle construction du *Blow up* est calculée en temps et en espace $\mathcal{O}(n^2)$. On peut envisager une réduction transitive de la relation d'ordre représentée par la matrice en temps $\mathcal{O}(n^3)$.
- Le pré-traitement de Krall en temps et en espace $\mathcal{O}(n^2)$ (ligne 19).
- La *Simple coloration* de B en temps $\mathcal{O}(n^4)$ [Caseau *et al.*, 1999] et en espace $\mathcal{O}(n^2)$ (ligne 21). Notons que la taille du *Blow up* est au plus n.
- Appel à l'Algorithme 6 qui met à jour le code de chaque élément de l'ordre. Ce code étant de taille au plus n, cet algorithme s'exécute alors en temps et en espace $\mathcal{O}(n^2)$ (ligne 22). Nous utilisons ce même espace mémoire, dont la taille reste limitée à n^2, pour traiter les autres nœuds de l'arbre.

Enfin, la complexité en temps de l'Algorithme 8 est en $\mathcal{O}(n^5)$. Nous précisons que l'espace mémoire utilisé pour récupérer l'ordre Q, construire son *Blow up* ainsi que calculer la *Simple coloration*, peut être libéré à la fin du traitement de chaque nœud. Ainsi, la complexité en espace de l'algorithme est en $\mathcal{O}(n^2)$.

□

2.5 Résultat théorique

Nous montrons ici que les deux options de notre heuristique calculent un codage par vecteur de bits de tout ordre. Cela permet de prouver la correction de l'Algorithme 8.

Proposition 2.2.15. *Pour tout ordre P, l'Algorithme 8 calcule un codage par vecteur de bits de P. En notant $GSB(P)$ (resp. $LSB(P)$) la taille de ce codage avec l'option GSB (resp. LSB), nous avons $dim_2(P) \leq \min(GSB(P), LSB(P))$.*

Démonstration. Nous proposons une preuve par induction sur la taille de P.

Pour P un ordre réduit à un unique élément, l'Algorithme 8 le code par l'ensemble vide. La taille de ce codage est nulle et nous avons aussi $dim_2(P) = 0$. La conclusion de la proposition est alors vérifiée dans ce cas.

Pour P un ordre à n éléments, nous supposons que la propriété à démontrer est vraie pour tout sous-ordre de P. Soit $\mathcal{T}$ l'arbre de décomposition modulaire de P. Nous distinguons deux cas :

Cas.1 La racine de $\mathcal{T}$ est un nœud série ou parallèle. Dans ce cas, comme les sous-ordres associés aux successeurs immédiats de la racine sont codés par vecteur de bits (selon l'hypothèse d'induction), le codage de leur composition par l'Algorithme 1 ou 2 est un codage par vecteur de bits (d'après la preuve de la Proposition 2.1.10).

Cas.2 La racine de $\mathcal{T}$ est un nœud premier. Dans ce cas, le codage de P est calculé à partir d'un codage par vecteur de bits de son *Blow up via* l'heuristique *SBSC* (Algorithme 6). La preuve de la Proposition 2.2.14 montre que le calcul de ce codage, comme décrit par l'Algorithme 6, est un codage par vecteur de bits.

Nous concluons que l'Algorithme 8 calcule un codage par vecteur de bits de P. Comme $GSB(P)$ (resp. $LSB(P)$) est la taille de ce codage avec l'option GSB (resp. LSB), nous avons $dim_2(P) \leq GSB(P)$ (resp. $dim_2(P) \leq LSB(P)$). Nous en déduisons $dim_2(P) \leq \min(GSB(P), LSB(P))$. □

2.6 Résultats numériques

Nous mesurons la performance de l'heuristique de codage des ordres *via* la décomposition modulaire, proposée dans cette section, à travers des tests effectués sur des données connues. Nous décrivons ces données puis nous donnons les résultats des tests obtenus.

2.6.1 Données de test

Nous évaluons la performance de notre heuristique sur des hiérarchies issues des langages de programmation orientés objets. Ces hiérarchies sont accessibles en ligne [2] et constituent les données de test de l'heuristique de codage des ordres développée par Krall *et al.* [Krall *et al.*, 1997]. Il s'agit des hiérarchies des classes des langages de programmation orientés objets **Self**, **LOV** (similaire au langage **Eiffel**), **Laure**, **Java**, ainsi que **Unidraw** (framework C++) et **Geode** (librairie pour C++ et Python). Nous ajoutons à cette collection, la hiérarchie des classes du langage **Java8** générée par Loiseau (cf. Figure 2.17). La Table 2.3 présente quelques caractéristiques de l'ensemble de ces hiérarchies.

Hiérarchie	Taille	Hauteur	$\max(d_{in})$	$\max(d_{out})$	Taille maximale d'un SONP [3]
Unidraw	613	10	2	147	21
Self	1801	18	9	232	426
LOV	436	10	10	78	298
Laure	295	12	3	8	51
Geode	1318	14	16	323	1027
Ed	434	11	7	78	307
Java	225	7	3	112	16
Java8	17086	11	17	4621	7258

TABLE 2.3: Caractéristiques de quelques hiérarchies connues

2.6.2 Expérimentations

Nous testons ici les deux options « GSB » et « LSB » de l'heuristique que nous proposons pour le codage des ordres par vecteur de bits *via* la décomposition modulaire. La Table 2.4 présente les résultats obtenus en moyenne pour 20 exécutions de notre heuristique et l'heuristique de référence *SBSC*. Nous y ajoutons également les résultats de la dernière heuristique existante testée sur notre jeu de données, celle de Krall *et al.* que nous désignons par *KVH* [Krall *et al.*, 1997].

2. http://www.complang.tuwien.ac.at/andi/typecheck/. Consulté en 2017
3. Sous-ordre associé à un nœud « Premier »

Les résultats montrent que, sur nos données, il existe toujours une option calculant un codage de taille petite. Comme nous l'avions mentionné auparavant, notre heuristique, avec l'option *Local Split and Balancing*, se montre inefficace face aux hiérarchies ayant un sous-ordre associé à un nœud « Premier » de taille importante (plus que le $\frac{2}{3}$ de la taille de la hiérarchie). Au moyen de l'option *Global Split and Balancing*, nous retrouvons au moins la taille du codage connue pour de telles hiérarchies.

Hiérarchie	KVH	SBSC	GSB	LSB
Unidraw	30	30	30	**24**
Self	53	53	53	**52**
Love-ed	57	**54**	**54**	58
Laure	**23**	**23**	**23**	**23**
Geode	95	**89**	**89**	97
Ed	54	**50**	**50**	53
Java	19	19	19	**16**
Java8	94	84	81	**78**

TABLE 2.4: Taille de codage des ordres par *KVH* et *SBSC vs* les deux options de codage *via* la décomposition modulaire (résultats pour 20 exécutions)

2.7 Synthèse

Rappelons que l'heuristique de codage des ordres *via* la décomposition modulaire, proposée dans cette section, est une extension de celle introduite précédemment pour le codage des ordres séries-parallèles. Nous observons, dans le cas général, que cette heuristique améliore sensiblement la taille du codage des hiérarchies. En particulier, l'option LSB diminue la taille du codage des hiérarchies *Self*, *Java8*, *Java* et *Unidraw* de 1.8% à 20%, cependant, elle augmente la taille du codage des autres hiérarchies d'environ 6%. Associée à l'option GSP, les deux options de notre heuristique permettent d'offrir un codage plus compact pour toutes les hiérarchies considérées.

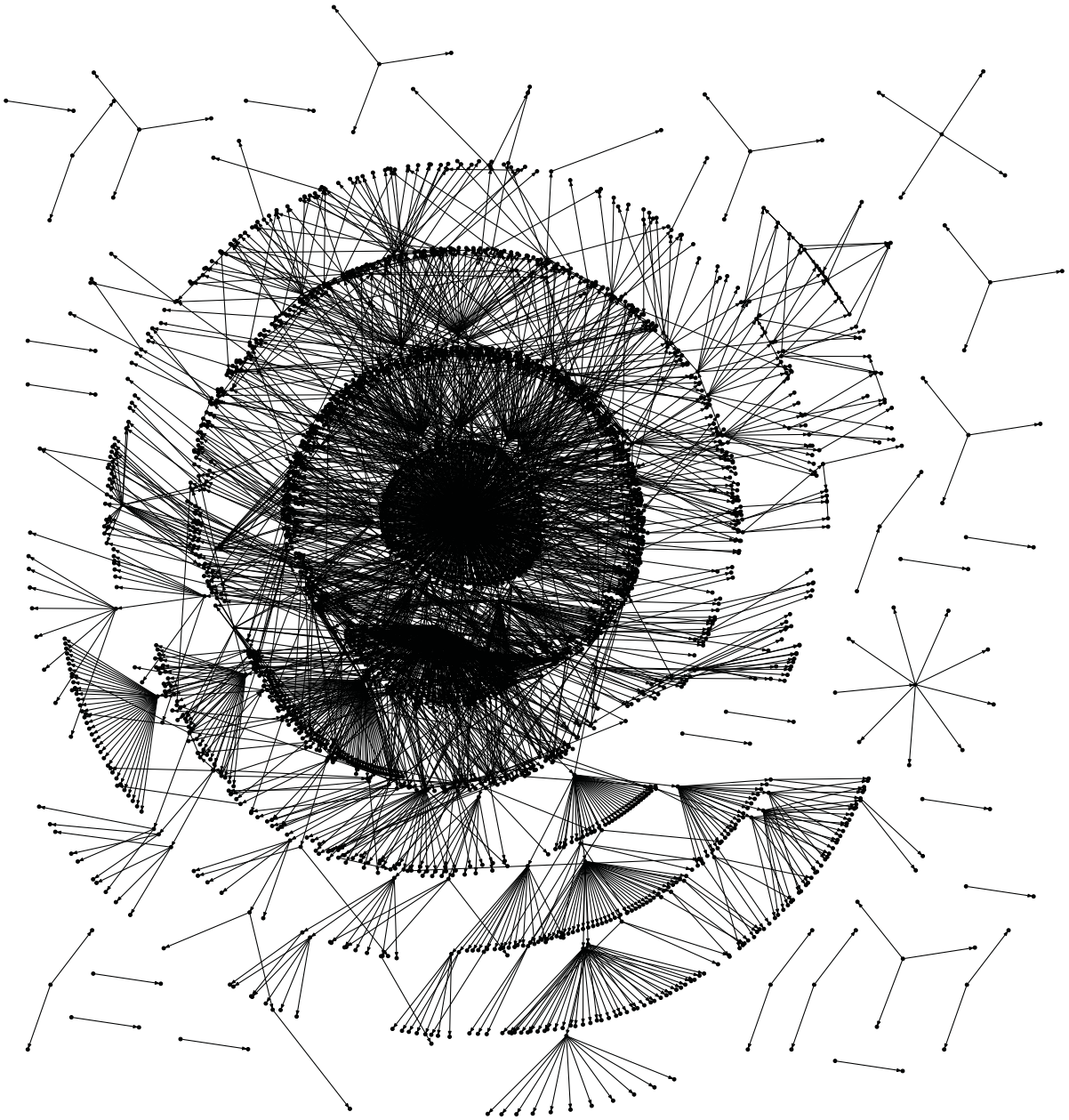


FIGURE 2.17: Diagramme de Hasse de la hiérarchie des classes du Java8

3 Cas des arbres

Nous traitons dans cette section le problème de calcul de la 2-dimension pour le cas particulier des arbres. Nous proposons ainsi une heuristique de codage des arbres par vecteur de bits basée sur la même stratégie que celle des trois dernières heuristiques existantes, à savoir : *Dichotomique*, *Polychotomique* et *Polychotomique Généralisé*. Nous définissons, dans un premier temps, cette stratégie, puis nous décrivons notre solution ainsi qu'un algorithme polynomial qui l'implémente. Nous prouvons enfin que notre heuristique améliore la taille du codage des arbres par vecteur de bits puis nous illustrons ce résultat par des tests réalisés sur des données connues.

3.1 Motivation

Si le problème de calcul de la 2-dimension des ordres est $\mathcal{NP}$-complet et non-approximable, il devient par contre approximable dans le cas des arbres. Néanmoins, la classe de complexité de ce problème dans ce même cas reste encore ouverte. Nous admettons alors qu'il est possible de concevoir des heuristiques intéressantes de codage des arbres par vecteur de bits dont nous pouvons nous inspirer pour améliorer le codage des ordres, comme cela a été le cas pour des techniques déjà citées au premier chapitre : la méthode de codage par *compression de fermeture transitive* qui se base sur un codage par intervalle unique des arbres ; l'heuristique de codage des ordres par une *Simple Coloration* qui généralise la *Simple Coloration* des arbres ; la représentation intervallaire introduite suite aux codages classiques des arbres ; et aussi la représentation vectorielle définie à partir d'un codage efficace des chaînes. Nous citons également le cas de notre heuristique de codage des ordres *via* la décomposition modulaire, qui délègue le traitement d'une composition parallèle d'ordres à une heuristique de codage des arbres. Sans négliger ces contextes d'études pour lesquels les données sont modélisables par cette structure (arbre ou arborescence), nous sommes totalement convaincus de l'intérêt de dédier cette section à l'étude du problème de calcul de la 2-dimension pour la classe des arbres.

3.2 Analyse des travaux précédents

Le codage des arbres par vecteur de bits était d'abord étudié par Caseau qui développe l'heuristique *Cmax* [Caseau, 1993] puis l'améliore avec Habib *et al.* par le codage *CHNR* [Caseau *et al.*, 1999]. En 2001, Raynaud et Thierry proposent le codage *Dichotomique* améliorant les heuristiques précédentes [Raynaud et Thierry, 2001]. Le principe de ces heuristiques est détaillé au premier chapitre (page 56, voir aussi leurs algorithmes en Appendice A). Nous reprenons celui du codage *Dichoto-*

mique, comme décrit dans [Habib *et al.*, 2004], puis nous décrivons les heuristiques proposées pour l'améliorer. Nous analysons le comportement des heuristiques décrites ici pour en tirer des propriétés puis une stratégie de codage développée dans la sous-section qui suit.

3.2.1 Codage *Dichotomique*

Un codage *Dichotomique* d'un arbre consiste à le diviser en deux parties, susceptibles d'être codées indépendamment, puis à répéter ce processus sur chaque partie (cf. Définition 2.3.7).

Définition 2.3.7 (Codage *Dichotomique* [Habib *et al.*, 2004])**.** *Un codage par vecteur de bits d'un arbre T est dit Dichotomique s'il existe deux couleurs distinctes i et j telles que :*

- *Tout code non vide d'un élément de T contient soit la couleur i soit la couleur j.*
- *Pour T_i, l'arbre composé des éléments de T tels que leurs codes contient i, la suppression de i de ces codes engendre un codage Dichotomique de T_i.*
- *Pour T_j, l'arbre composé des éléments de T tels que leurs codes contient j, la suppression de j de ces codes engendre un codage Dichotomique de T_j.*
- *Si T est un singleton, son codage par l'ensemble vide est un codage Dichotomique.*

En 2001, Raynaud et Thierry conçoivent l'algorithme *Dicho* permettant de calculer un codage *Dichotomique* pour tout arbre. Nous décrivons en détail les étapes de leur algorithme [Raynaud et Thierry, 2001].

Soit T un arbre. Afin de coder les éléments de T, l'algorithme *Dicho* procède par l'attribution, à chacun de ses éléments x suivant leur ordre topologique inverse, d'un poids $w(x)$ calculé à partir des poids de ses fils. Le calcul de ce poids permet de coder simultanément les fils de x. Nous distinguons quatre cas de figures :

1. L'élément x n'a aucun fils : $w(x) = 0$.
2. L'élément x a un seul fils y : $w(x) = w(y) + 1$ et $code(y) \leftarrow \{w(x)\}$.
3. L'élément x a deux fils y et z : $w(x) = \max(w(y) + w(z)) + 2$ et $code(y) \leftarrow \{w(x) - 1\}$, $code(z) \leftarrow \{w(x)\}$.
4. L'élément x a au moins trois fils, avec a et b ses deux fils de plus petits poids. Dans ce cas, un nouveau fils c de x est introduit dans l'arbre T comme parent de a et b. Ainsi, $w(c) = \max(w(a) + w(b)) + 2$ et $code(a) \leftarrow \{w(c) - 1\}$, $code(b) \leftarrow \{w(c)\}$. Le poids de x sera calculé par la suite à partir du poids de c et les poids de ses autres fils (a et b ne sont plus des fils de x).

L'algorithme *Dicho*, ainsi conçu, permet de générer un arbre binaire BT, dans lequel se plonge l'arbre T, et de calculer son codage réduit. Nous en déduisons un codage de T qui, à chacun de ses éléments x, associe le code $\bigcup_{u \leq_{BT} x} code(u)$. Nous illustrons ce codage par la Figure 2.18.

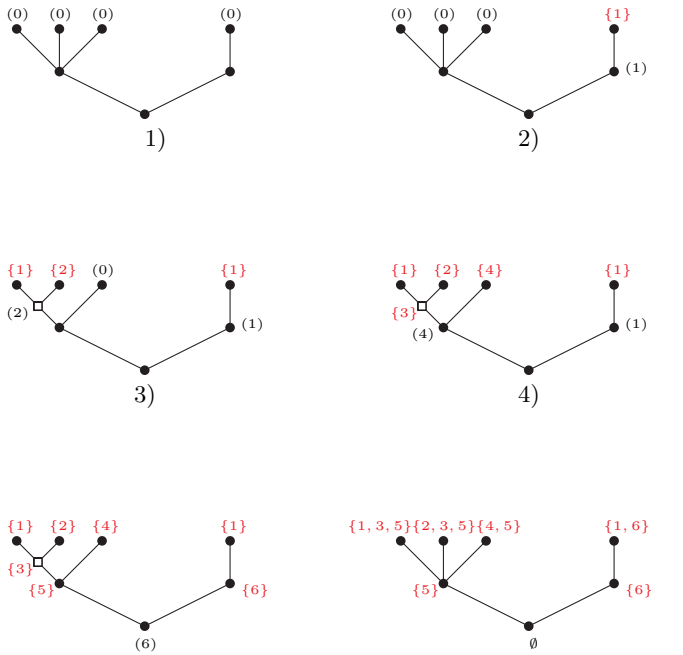


FIGURE 2.18: Codage *Dichotomique* par l'algorithme *Dicho*

Théorème 2.3.16. *[Habib et al., 2004] L'algorithme* Dicho *calcule un codage Dichotomique de plus petite taille en temps $\mathcal{O}(n.e)$, n étant la taille de l'arbre et e la taille du codage, avec $e \leq n$.*

La taille du codage de T, calculé par l'algorithme *Dicho*, correspond au poids attribué à sa racine. Elle correspond aussi au maximum degré d'une chaîne de BT [Habib *et al.*, 2004]. Notons qu'il s'agit de la plus petite taille d'un codage *Dichotomique* de T (Théorème 2.3.16) puisque l'algorithme *Dicho* génère l'arbre binaire ayant le plus petit maximum degré d'une chaîne. Afin d'améliorer la taille de ce codage, il suffit de diminuer les poids attribués aux éléments de T et surtout le poids de sa racine. Notons que le poids d'un élément de T détermine la taille du codage du sous-arbre enraciné en cet élément. Nous nous concentrons sur le calcul du poids des éléments de l'arbre.

3.2.2 Codage *Dichotomique* : nouvelle formulation

En 2002, Filman propose une re-formulation du processus de calcul du poids d'un élément d'un arbre, par l'algorithme *Dicho*, par l'introduction de la fonction de poids $\mathcal{D}$ (Équation 2.1). Cette fonction calcule le poids d'un élément à partir des poids de ses fils, passés en paramètre à la fonction sous forme d'une séquence croissante S d'entiers. Nous dénotons une séquence S de n entiers par $[s_1, \ldots, s_n]$, avec $s_1 \leq \cdots \leq s_n$. L'élément s_n correspond au plus grand élément de S, i.e. au $\max(S)$.

$$\mathcal{D}(S) = \begin{cases} 0 & \text{si } |S| = 0, \\ s_1 + 1 & \text{si } |S| = 1, \\ s_2 + 2 & \text{si } |S| = 2, \\ \mathcal{D}([s_3, \ldots, s_2 + 2, \ldots, s_n]) & \text{sinon.} \end{cases} \tag{2.1}$$

Nous justifions à travers la Proposition 2.3.16 que l'algorithme *Dicho* attribue les poids minimums aux éléments ayant au plus un fils. Nous étudions dans ce qui suit le cas des éléments ayant au moins deux fils.

Proposition 2.3.16. *Soit T un arbre enraciné en r.*
- *Si T est réduit à r, alors $dim_2(T)$ est nulle.*
- *Si r possède un fils unique, alors $dim_2(T)$ est égale à $dim_2(T\backslash\{r\}) + 1$.*

Démonstration. Pour le premier cas, nous avons T correspond à une chaîne de taille 1. Selon la Proposition 1.2.4, $dim_2(T) = 0$.
Pour le second cas, nous avons T une composition série de r, la racine de $T\backslash\{r\}$, et $T\backslash\{r\}$ privé de sa racine. Comme la 2-dimension de $T\backslash\{r\}$ reste la même si on le prive de sa racine, nous obtenons $dim_2(T) = dim_2(T\backslash\{r\}) + 1$ (cf. Proposition 2.1.8).

□

D'une manière générale, nous pouvons définir un codage *Dichotomique* d'un arbre qui consiste à calculer un poids pour chacun de ses éléments, à partir des poids de ses fils $s_1, \ldots, s_n$, passés en paramètre à la fonction de poids $\mathcal{W}$ que nous définissons comme suit :

$$\mathcal{W}(S) = \begin{cases} 0 & & \text{si } |S| = 0, \\ s_1 + \alpha \text{ avec } \alpha \geq 1 & & \text{si } |S| = 1, \\ s_2 + \beta \text{ avec } \beta \geq 2 & & \text{si } |S| = 2, \\ \mathcal{W}([s_j + \gamma] \cup S\backslash\{s_i, s_j\}) & \text{avec } 1 \leq i \leq j \leq |S| \text{ et } \gamma \geq 2 & \text{sinon.} \end{cases} \tag{2.2}$$

3.2.3 Codage *Polychotomique*

Nous venons de montrer que la taille d'un codage *Dichotomique* des arbres est calculée progressivement par une fonction de poids et correspond au poids attribué à la racine. Nous avons de plus indiqué que l'algorithme *Dicho* attribue les poids minimums aux éléments ayant au plus un fils. Nous introduisons ici le codage *Polychotomique* permettant de minimiser le poids des éléments ayant au moins deux fils.

Nous précisons que l'algorithme *Dicho*, présenté auparavant, calcule un codage des arbres suivant le principe d'une *Simple Coloration*. En effet, il suffit de remarquer que le code associé à chaque élément contient au plus une couleur propre (page 99). Comme discuté au premier chapitre, il est possible de minimiser la taille de ce codage grâce à une *Coloration Multiple*.

Le codage *Polychotomique*, proposé par Filman [Filman, 2002] (Équation 2.3), introduit le principe de *Coloration Multiple* dans le calcul du poids d'un élément x dont les poids de ses fils, $s_1, \ldots, s_n$, vérifient $s_n - s_2 < 2$ (premier cas de l'Équation 2.3). Nous expliquons cela juste après.

$$\mathcal{P}(S) = \begin{cases} s_n + sp(n) & \text{si } |S| \geq 2 \text{ } et \text{ } s_n - s_2 < 2, \\ \mathcal{P}([s_3, \ldots, s_2 + 2, \ldots, s_n]) & \text{si } |S| \geq 2 \text{ } et \text{ } s_n - s_2 \geq 2. \end{cases} \tag{2.3}$$

Soit T un arbre dont les sous-arbres $T_1, \ldots T_n$, enracinés en un fils de sa racine, sont déjà codés avec un ensemble de s_n ou $s_n - 1$ couleurs, sauf T_1 avec $s_1 \leq s_n$ couleurs (selon la condition $s_n - s_2 < 2$). Afin d'assurer le codage de T, l'idée de Filman est de générer, à partir d'un ensemble de $sp(n)$ couleurs additionnelles, n sous-ensembles incomparables, $e_1, \ldots, e_n$, de $\lceil \frac{sp(n)}{2} \rceil$ couleurs. Puis, de coder chaque élément de T_i, par l'union de son code dans T_i et le sous-ensemble e_i. Ceci donne lieu à un codage de T par vecteur de bits, ou encore de la composition parallèle de $T_1, \ldots, T_n$, suivant le principe d'une *Coloration Multiple*. La taille de ce codage, illustré par la Figure 2.19, constitue le poids attribué à la racine de T et vaut $s_n + sp(n)$.

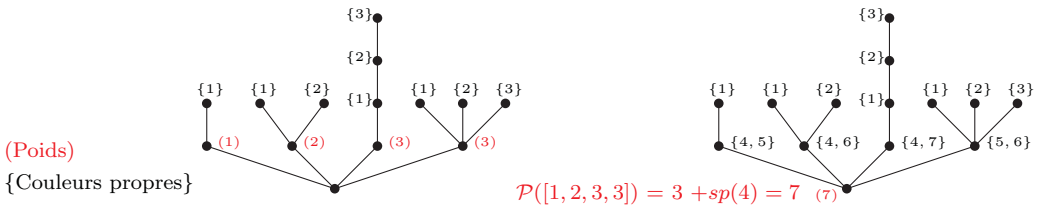


FIGURE 2.19: Codage *Polychotomique* d'un arbre

3.2.4 Notion de flat-séquence et propriétés adjointes

La notion de flat-séquence est introduite par Filman [Filman, 2002] dans le cadre d'un codage *Polychotomique* des arbres. Il s'agit d'une séquence d'entiers $[s_1, \ldots, s_n]$ dans laquelle $s_n - s_2 < 2$. Nous étendons cette définition comme suit.

Définition 2.3.8 (k-flat-séquence)**.** *Une séquence $[s_1, \ldots, s_n]$ est une* ***k-flat-séquence*** *– ou simplement une flat-séquence – si l'un de ces cas se présente :*

- $s_n - s_2 < 2$ *et* $s_n = k$.
- $s_n = s_2 = k - 1$.
- $n = 1$ *et* $s_1 \leq k$.

Nous dérivons de cette définition, celle d'un k-flat-partitionnement d'une séquence.

Définition 2.3.9 (k-flat-partitionnement)**.** *Soit $\mathcal{W}$ une fonction de poids qui, à une séquence d'entiers S, associe un entier positif.*
Un ***k-flat-partitionnement*** *de S par $\mathcal{W}$, est un partitionnement de S en $X_1, .., X_m$, tel que $[\mathcal{W}(X_i) | X_i \neq \{k\}$ et i de 1 à $m]$ est une k-flat-séquence.*

Nous discutons maintenant le poids des éléments dont le poids des fils forment une flat-séquence. Soit S une séquence d'entiers $[s_1, \ldots, s_n]$. Nous supposons que S est une s_n-flat-séquence et correspond aux poids des fils d'un élément x. Le codage *Polychotomique* associe à x le poids $s_n + sp(n)$ (Équation 2.3), alors que l'algorithme *Dicho* lui associe le poids $\mathcal{D}(S)$ tel que

$$s_n + 2\lceil \log_2(n) \rceil - 1 \leq \mathcal{D}(S) \leq s_n + 2\lceil \log_2(n) \rceil \text{ [Filman, 2002].} \tag{2.4}$$

Nous donnons, dans la Proposition 2.3.17, la valeur exacte de $\mathcal{D}(S)$ lorsque s_2 est égal à s_n.

Proposition 2.3.17. *Pour toute séquence d'entiers $S = [s_1, \ldots, s_n]$, avec $n > 1$ et $s_2 = s_n$, $\mathcal{D}(S) = s_n + 2\lceil \log_2(n) \rceil$.*

Démonstration. Nous proposons une preuve par induction sur la taille de S.
Pour $|S| = 2$, nous avons $\mathcal{D}(S) = s_2 + 2$ (Équation 2.1). La conclusion de la proposition est satisfaite dans ce cas.
Pour $|S| > 2$, nous supposons que la propriété est vraie pour les séquences de taille $|S| - 1$. Remarquons que, après $\lfloor \frac{n}{2} \rfloor$ itérations de l'algorithme *Dicho* (appels récursifs), S sera substituée par $R = [s_n + t, s_2 + 2, s_4 + 2, \ldots]$ telle que $t = 2(1 - n \mod 2)$. Comme tout élément de R (sauf possiblement le premier élément) est égal à s_n+2, alors, par hypothèse d'induction, $\mathcal{D}(R) = s_n+2+2\lceil \log_2(\frac{n}{2}) \rceil$ qui est égal à $s_n + 2 + 2\lceil \log_2(n) \rceil - 2$. Sachant que $\mathcal{D}(S) = \mathcal{D}(R)$, nous concluons que $\mathcal{D}(S) = s_n + 2\lceil \log_2(n) \rceil$. La conclusion de la proposition est satisfaite dans ce cas aussi.
Nous concluons que la Proposition 2.3.17 est valide. □

Nous montrons dans la suite que, étant donnée une séquence d'entiers S quelconque, l'algorithme *Dicho* génère toujours une flat-séquence à partir de S lors du calcul de $\mathcal{D}(S)$. Nous exprimons par la suite le poids $\mathcal{D}(S)$, calculé par l'algorithme *Dicho*, en fonction de la taille de la flat-séquence générée. Nous faisons la même étude pour le codage *Polychotomique*, pour en déduire que ce dernier améliore la taille du codage *Dichotomique*.

Soit S une séquence d'entiers $[s_1, \ldots, s_n]$. Montrons que l'algorithme *Dicho* génère une s_n-flat-séquence lors du calcul de $\mathcal{D}(S)$. Cela signifie qu'il existe toujours une s_n-flat-séquence Q telle que $\mathcal{D}(S) = \mathcal{D}(Q)$. Pour cela, nous montrons par induction que, pour tout entier i entre 1 et n, $\mathcal{D}(S) = \mathcal{D}(X_i \cup [s_{i+1}, \ldots, s_n])$, avec X_i une s_i-flat-séquence.

Nous avons déjà $S = [s_1] \cup [s_2, \ldots, s_n] = [s_1, s_2] \cup [s_3, \ldots, s_n]$, avec $[s_1]$ une s_1-flat-séquence et $[s_1, s_2]$ une s_2-flat-séquence, par Définition 2.3.8. Supposons que $\mathcal{D}(S) = \mathcal{D}(X_i \cup [s_{i+1}, \ldots, s_n])$, avec X_i une s_i-flat-séquence.
Si la sous-séquence $X_i \cup [s_{i+1}]$ est une s_{i+1}-flat-séquence, dans ce cas nous avons $\mathcal{D}(S) = \mathcal{D}(X_{i+1} \cup [s_{i+2}, \ldots, s_n])$.
Dans le cas contraire, nous exécutons quelques itérations de l'algorithme *Dicho* : remplacer les deux plus petits entiers de la séquence S par leur maximum plus 2, et recommencer le même traitement sur la séquence résultante et ainsi de suite (Équation 2.1). Au moment où les deux plus petits entiers a et b de la séquence courante satisfont $a \leq s_{i+1}$ et $s_{i+1} - 1 \leq b \leq s_{i+1}$, cette séquence est alors de la forme $[a, b, a', b', \ldots, a^\ell, b^\ell, s_{i+1}, \ldots, s_n]$ telle que $s_{i+1} - b < 2$. Dans ce cas, nous avons $\mathcal{D}(S) = \mathcal{D}(X_{i+1} \cup [s_{i+2}, \ldots, s_n])$, avec X_{i+1} la flat-séquence $[a, b, a', b', \ldots, a^\ell, b^\ell, s_{i+1}]$. Nous vérifions ainsi la propriété à démontrer.

Nous déduisons de cette propriété qu'il existe une s_n-flat-séquence X_n telle que $\mathcal{D}(S) = \mathcal{D}(X_n)$ (prendre $i = n$). La génération de cette flat-séquence donne lieu à un s_n-flat-partitionnement de S par $\mathcal{D}$ (Figure 2.20). En général, pour tout entier $k \geq s_n$, l'algorithme *Dicho* produit une k-flat-séquence à partir de S, lors du calcul de $\mathcal{D}(S)$.

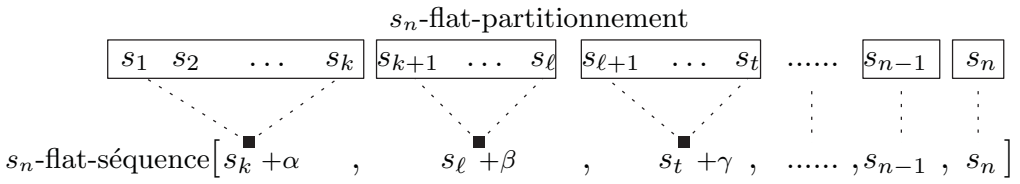


FIGURE 2.20: Génération d'un flat-partitionnement par l'algorithme *Dicho*

Nous utilisons les notations suivantes dans la suite de ce chapitre :

- F_d^S : s_n-flat-séquence générée par l'algorithme *Dicho* en calculant $\mathcal{D}(S)$
- $f_d(S)$: la taille de F_d^S
- $\mathcal{P}_d^S$: s_n-flat-Partitionnement de S par $\mathcal{D}$
- $F_d^{S_k}$: k-flat-séquence générée par l'algorithme *Dicho* à partir de S, avec $k \geq s_n$
- $f_d^k(S)$: la taille de $F_d^{S_k}$

Nous reformulons le comportement de l'algorithme *Dicho* (Équation 2.1) de la façon suivante :

$$\mathcal{D}(S) = \mathcal{D}(F_d^{[s_1,\ldots,s_i]} \cup [s_{i+1}, \ldots, s_n]) = \mathcal{D}(F_d^S) = \mathcal{D}(F_d^{S_k}) - \left\lfloor \frac{1}{|F_d^{S_k}|} \right\rfloor . \tag{2.5}$$

L'Inéquation 2.4 et l'Équation 2.5 impliquent

$$\max(S) + 2\lceil \log_2(f_d(S)) \rceil - 1 \leq \mathcal{D}(S) \leq \max(S) + 2\lceil \log_2(f_d(S)) \rceil. \tag{2.6}$$

Nous traitons maintenant le codage *Polychotomique*. Soit S une séquence d'entiers. D'après l'Équation 2.3, le codage *Polychotomique* effectue les mêmes itérations que l'algorithme *Dicho*, jusqu'à ce que la séquence courante soit une $\max(S)$-flat-séquence. Il retourne ensuite

$$\mathcal{P}(S) = \max(S) + sp(f_d(S)). \tag{2.7}$$

Filman prouve que $\mathcal{P}(S) \leq \mathcal{D}(S)$ pour toute séquence d'entiers S [Filman, 2002]. Il est également possible, d'après l'Inéquation 2.6 et l'Équation 2.7, de vérifier ce résultat en prouvant que

1. Pour $n \leq 4$, $\mathcal{P}(S) \leq \mathcal{D}(S)$. La preuve est immédiate.
2. Pour $n > 4$, $sp(n) \leq 2\lceil \log_2(n) \rceil - 1$. Cela est facilement vérifiable grâce à l'inéquation proposée par Habib *et al.* [Habib *et al.*, 2004]

$$\lfloor \log_2(n) + \frac{\log_2(\log_2(n))}{2} + 1 \rfloor \leq sp(n) \leq \lfloor \log_2(n) + \frac{\log_2(\log_2(n))}{2} + 2 \rfloor \tag{2.8}$$

Nous avons défini, dans ce qui précède, la fonction f_d^k qui retourne la taille de la k-flat-séquence générée par l'algorithme *Dicho*, à partir de la séquence d'entiers qui lui est passée en paramètre. Nous montrons qu'il s'agit de la taille minimale d'une k-flat-séquence générée par un codage *Dichotomique*. Nous parlons ici des codages dont le processus fait appel à une certaine fonction de poids $\mathcal{W}$ (cf. Équation 2.2). Par abus de langage, nous désignons par un codage *Dichotomique* d'une séquence S, le processus du calcul de $\mathcal{W}(S)$ (la taille de ce codage).

Théorème 2.3.17. *Pour toute séquence d'entiers S et tout entier $k \geq \max(S)$, $f_d^k(S)$ est la plus petite taille d'une k-flat-séquence générée, à partir de S, par un codage Dichotomique.*

Démonstration. Pour cette preuve, nous supposons le cas où S n'est pas une k-flat-séquence puisque la conclusion du théorème demeure évidente le cas échéant.

Soit $\mathbb{A}$ l'ensemble des algorithmes de codage *Dichotomique*, générant un k-flat-partitionnement de S lors du calcul de son codage. Soit d_k une fonction qui prend en paramètre une séquence d'entiers X, et retourne k si X est la séquence $[k]$ et $\mathcal{D}(X)$ sinon. Soit $\mathbb{B}$ les algorithmes de $\mathbb{A}$ qui produisent un k-flat-partitionnement de S, $\{X_1, \ldots, X_m\}$, tel que $[d_k(X_1), \ldots, d_k(X_m)]$ est une k-flat-séquence. Nous supposons que $d_k(X_1) \leq \cdots \leq d_k(X_m)$ conformément à notre définition d'une séquence d'entiers.

Soit $\mathcal{O}$ un algorithme de $\mathbb{A}\backslash\mathbb{B}$ et $P = \{X_1, \ldots, X_m\}$ un k-flat-partitionnement de S par $\mathcal{O}$. Alors $\mathcal{O}(X_i) \leq k$ pour tout $1 \leq i \leq m$, ainsi que $\mathcal{D}(X_i) \leq \mathcal{O}(X_i)$ (Théorème 2.3.16). L'algorithme $\mathcal{O}$ n'étant pas dans $\mathbb{B}$, il existe alors au moins deux entiers i et j, soit les entiers 1 et 2 sans perte de généralité, tels que $\mathcal{D}(X_1) \leq k-2$ et $\mathcal{D}(X_2) \leq k-2$. Nous définissons par la suite, un algorithme $\mathcal{O}'$ qui génère un k-flat-partitionnement de S à partir de la séquence $[d_k(X_1), d_k(X_2), d_k(X_3), \ldots, d_k(X_m)]$, en remplaçant ses deux éléments minimaux $d_k(X_1)$ et $d_k(X_2)$ par $\mathcal{D}(X_1 \cup X_2)$ puis en itérant ainsi. Notons que $\mathcal{D}(X_1 \cup X_2) \leq \max(\mathcal{D}(X_1), \mathcal{D}(X_2)) + 2 \leq k$ (Théorème 2.3.16) : il suffit de considérer le codage *Dichotomique* qui code $X_1 \cup X_2$ par $\max(\mathcal{D}(X_1), \mathcal{D}(X_2)) + 2$. L'algorithme $\mathcal{O}'$ générera donc un k-flat-partitionnement de S de taille inférieure à m. Comme cet algorithme appartient à la classe $\mathbb{B}$, nous nous focalisons sur des algorithmes de cette classe pour prouver le théorème.

Nous proposons une preuve par induction sur la taille de S.
Pour $|S| \leq 2$, nous avons S une k-flat-séquence. La conclusion du théorème est évidente dans ce cas.
Soit S une séquence d'entiers $[s_1, \ldots, s_n]$. Nous supposons que la propriété du théorème est satisfaite pour toute séquence de taille au plus $n-1$. Nous la prouvons pour S en traitant ces trois cas :

Cas.1 $f_d^k(S) = 1$
Dans ce cas, $f_d^k(S)$ est minimale.

Cas.2 $f_d^k(S) = 2$
Dans ce cas, nous avons $\mathcal{D}(S) > k$. S'il existe un codage *Dichotomique* générant une k-flat-séquence de taille 1, il fournira alors un codage *Dichotomique* de S de taille inférieure à k. Cela contredit la minimalité de la taille d'un codage *Dichotomique* de S déterminée par la fonction $\mathcal{D}$ (Théorème 2.3.16). Par conséquent, $f_d^k(S)$ est minimale.

Cas.3 $f_d^k(S) > 2$

Nous prouvons ce cas par contradiction. Soit $\mathcal{O}$ un codage *Dichotomique* (de la classe $\mathbb{B}$) générant un k-flat-partitionnement de S de taille minimale. Soit $P = \{X_1, \ldots, X_m\}$ ce partitionnement avec $m < f_d^k(S)$. Ici $m > 2$. En effet, $f_d^k(S) > 2$ implique $\mathcal{D}(S) > k + 2$. Par conséquent, si $m \leq 2$, alors la taille de codage de S peut être $k + 2$. Contradiction selon le Théorème 2.3.16.
Rappelons que, pour tout élément X_i de P, sauf au plus un élément, $\mathcal{D}(X_i)$ est dans $\{k-1, k\}$. Deux sous-cas se présentent :

(a) Il existe une sous-séquence X_i contenant s_1 et s_2. Alors, $P\backslash\{X_i\} \cup \{X_i\backslash\{s_1, s_2\} \cup \{s_2 + 2\}\}$ est un k-flat-partitionnement de la séquence $[s_3, \ldots, s_2 + 2, \ldots, s_n]$ de taille m, générée par un algorithme de $\mathbb{B}$. Notons que $\mathcal{D}(X_i\backslash\{s_1, s_2\} \cup \{s_2 + 2\}) = \mathcal{D}(X_i)$ (Équation 2.1).
Par hypothèse d'induction, on a $f_d^k([s_3, \ldots, s_2{+}2, \ldots, s_n]) \leq m$. Comme $f_d^k(S) = f_d^k([s_3, \ldots, s_2{+}2, \ldots, s_n])$, nous concluons que $f_d^k(S) \leq m$. Ceci contredit la minimalité de m.

(b) Il existe une sous-séquence X_i et une autre X_j contenant s_1 et s_2 respectivement. Soit X_{ij} l'union de X_i et X_j et soit Q un k-flat-partitionnement de X_{ij} par l'algorithme *Dicho*. Par hypothèse d'induction, nous avons $|Q| \leq 2$ puisque $\{X_i, X_j\}$ est un k-flat-partitionnement de X_{ij} de taille 2. Nous avons alors $P' = P\backslash\{X_i, X_j\} \cup Q$ un k-flat-partitionnement de S par un codage *Dichotomique*.
Si $|Q| = 1$, alors $|P'| < m$, ce qui contredit la minimalité de m.
Sinon, Q est composé de deux parties dont l'une contient s_1 et s_2. En effet, il s'agit des plus petits éléments de X_{ij}, qui n'est pas une k-flat-séquence, et dont le partitionnement était réalisé par l'algorithme *Dicho* (Équation 2.1). Dans ce cas, nous nous référons au premier sous-cas (a).

Les deux sous-cas contredisent notre hypothèse qu'il existe un codage *Dichotomique* $\mathcal{O}$ produisant un k-partitionnement de S de taille plus petite que $f_d^k(S)$. Nous concluons que $f_d^k(S)$ est la plus petite taille d'une k-flat-séquence (aussi un k-flat-partitionnement) produite par un codage *Dichotomique* à partir de S.

Les trois cas traités prouvent le Théorème 2.3.17. □

Nous montrons maintenant que la fonction f_d^k est monotone. Nous définissons d'abord les conditions de monotonie de cette fonction.

Définition 2.3.10 (Monotonie)**.** *Soit $\mathbb{S}$ l'ensemble de toutes les séquences d'entiers et soit f une fonction de $\mathbb{S}$ vers $\mathbb{N}$.*

La fonction f est monotone si pour toute paire d'éléments Q et S de $\mathbb{S}$:

1. *$|Q| = |S|$ et $Q \leq_{lexico} S$ implique $f(Q) \leq f(S)$.*
2. *$Q \subseteq S$ implique $f(Q) \leq f(S)$*

La Proposition 2.3.18 affirme que la fonction f_d^k vérifie la première condition de monotonie (cf. Définition 2.3.10).

Proposition 2.3.18. *Soit Q et S deux séquences de n entiers avec $Q \leq_{lexico} S$. Pour tout entier $k \geq \max(S)$, $f_d^k(Q) \leq f_d^k(S)$.*

Démonstration. Nous proposons une preuve par induction sur la taille de S.
Pour $|S| = 1$, nous avons $f_d^k(Q) = f_d^k(S) = 1$. Cela satisfait la conclusion de la proposition.
Soit S la séquence d'entiers $[s_1, \ldots, s_n]$ et Q la séquence $[q_1, \ldots, q_n]$. Notons $S[i]$ (resp. $Q[i]$) le i^{th} élément de S (resp. Q).
Supposons que la conclusion de la proposition est vérifiée pour toute paire de séquences d'au plus $n-1$ éléments. Nous la prouvons pour S et Q.
Notons R et R' les séquences générées après l'exécution d'une itération de l'algorithme *Dicho* sur Q et S respectivement. Nous avons alors $R = [q_3, \ldots, q_2{+}2, \ldots q_n]$ et $R' = [s_3, \ldots, s_2 + 2, \ldots s_n]$. Prouvons que $R \leq_{lexico} R'$.
Nous avons $q_2 + 2 \leq s_2 + 2$. Soit i (resp. j) le rang de $q_2 + 2$ (resp. $s_2 + 2$) dans R (resp. R'). Nous distinguons deux cas :

- Lorsque $j \leq i$, alors, pour tout entier k tel que $k < j$ ou $i < k$, nous avons $R[k] \leq_{lexico} R'[k]$. En effet, nous avons $R[k] = Q[k]$ et $R'[k] = S[k]$. Ainsi que pour k dans $[j, i]$, nous avons $R[k] \leq_{lexico} R[i] \leq_{lexico} R'[j] \leq_{lexico} R'[k]$. Par conséquent, $R \leq_{lexico} R'$.
- Maintenant, lorsque $i \leq j$, alors, pour tout entier k tel que $k < i$ ou $j < k$, nous avons $R[k] \leq_{lexico} R'[k]$. En effet, on a $R[k] = Q[k]$ et $R'[k] = S[k]$. De plus, pour k dans $[i, j-1]$, nous avons $R[k] \leq_{lexico} R[k+1] \leq_{lexico} R'[k]$ et $R[j] \leq_{lexico} R'[j-1] \leq_{lexico} R'[j]$. Par conséquent, $R \leq_{lexico} R'$.

Par hypothèse d'induction, nous avons $R \leq_{lexico} R'$ implique $f_d^k(R) \leq f_d^k(R')$.
Comme, $f_d^k(Q) = f_d^k(R)$ et $f_d^k(S) = f_d^k(R')$, nous obtenons $f_d^k(Q) \leq f_d^k(S)$.
La Proposition 2.3.18 est maintenant prouvée. □

La Proposition 2.3.19 affirme que la fonction f_d^k vérifie la deuxième condition de monotonie (cf. Définition 2.3.10). Rappelons que pour S une séquence d'entiers quelconque, $f_d^k(S)$ est définie pour $k \geq \max(S)$. Cependant, dans les Propositions 2.3.19 et 2.3.20, nous étendons cette définition pour tout entier k et considérons $f_d^k(S)$ la taille de la k-flat-séquence générée par l'algorithme *Dicho* à partir de la plus longue sous-séquence de S dont les éléments sont inférieurs ou égaux à k.

Proposition 2.3.19. *Pour toute paire de séquences Q et S et pour tout entier k,*

$$Q \subseteq S \text{ implique } f_d^k(Q) \leq f_d^k(S).$$

Démonstration. Dans la preuve, nous notons S^k la k-flat-séquence générée par l'algorithme *Dicho* à partir de la plus longue sous-séquence de S dont les éléments sont inférieurs ou égaux à k. Notons a_s^k le nombre des éléments inférieurs ou égaux à $k-1$ dans S^k et b_s^k le nombre des éléments égaux à k dans S^k. Alors, $f_d^k(S) = a_s^k + b_s^k$. Notons X_s^k le nombre des éléments égaux à k dans S. Nous utilisons les mêmes notations lorsqu'il s'agit de la séquence Q. D'après le principe de l'algorithme *Dicho*, nous avons

$$\begin{array}{lcl} (a_s^0, b_s^0) & = & (0, X_s^0) \\ (a_s^{k+1}, b_s^{k+1}) & = & (b_s^k + a_s^k \mod 2,\ X_s^{k+1} + \lfloor \frac{a_s^k}{2} \rfloor) \end{array}$$

Dans la suite, nous considérons les notations suivantes :

$$\delta a^k = a_s^k - a_q^k, \quad \delta b^k = b_s^k - b_q^k, \quad \text{et}$$

$$\delta X^k = X_s^k - X_q^k \geq 0. \text{ Notons que } X_q^k \leq X_s^k \text{ puisque } Q \subseteq S.$$

Nous exprimons alors $f_d^k(Q) \leq f_d^k(S)$ par $\delta a^k + \delta b^k \geq 0$.
Prouvons par induction sur la valeur de k que $\delta a^k + \delta b^k \geq 0$.
Pour $k = 0$, nous avons $\delta a^0 + \delta b^0 = \delta X^0 \geq 0$.
Pour $k > 0$, nous supposons que la propriété est vraie pour les entiers inférieurs ou égaux à k. Prouvons que $\delta a^{k+1} + \delta b^{k+1} \geq 0$.
Nous avons,

$$\begin{array}{rcl} \delta a^{k+1} & = & a_s^{k+1} - a_q^{k+1} \\ & = & b_s^k + a_s^k \mod 2 - b_q^k - a_q^k \mod 2 \\ & = & \delta b^k + a_s^k \mod 2 - a_q^k \mod 2 \end{array}$$

Si δa^k est pair, i.e. a_s^k et a_q^k ont la même parité, alors $\delta a^{k+1} = \delta b^k$.
Sinon, δa^k est impair, i.e. a_s^k et a_q^k n'ont pas la même parité. Deux cas se présentent :

1. La valeur a_q^k est paire, implique $\delta a^{k+1} = \delta b^k + 1$

2. La valeur a_q^k est impaire, implique $\delta a^{k+1} = \delta b^k - 1$

D'un autre côté, nous avons

$$\begin{aligned}
\delta b^{k+1} &= b_s^{k+1} - b_q^{k+1} \\
&= X_s^{k+1} + \lfloor \tfrac{a_s^k}{2} \rfloor - X_q^{k+1} - \lfloor \tfrac{a_q^k}{2} \rfloor \\
&= \delta X^{k+1} + \tfrac{a_s^k - a_s^k \mod 2}{2} - \tfrac{a_q^k - a_q^k \mod 2}{2} \\
&= \delta X^{k+1} + \tfrac{\delta a^k - (a_s^k \mod 2 - a_q^k \mod 2)}{2}
\end{aligned}$$

Si δa^k est pair, alors $\delta b^{k+1} = \delta X^{k+1} + \frac{\delta a^k}{2}$.
Sinon, nous avons deux cas :

1. La valeur a_q^k est paire, implique $\delta b^{k+1} = \delta X^{k+1} + \frac{\delta a^k - 1}{2}$
2. La valeur a_q^k est impaire, implique $\delta b^{k+1} = \delta X^{k+1} + \frac{\delta a^k + 1}{2}$

En conclusion, nous avons

- Si δa^k est pair, alors $\delta a^{k+1} + \delta b^{k+1} = \delta b^k + \delta X^{k+1} + \frac{\delta a^k}{2}$ (1)
- Si δa^k est impair et a_q^k est pair, alors

$$\delta a^{k+1} + \delta b^{k+1} = \delta b^k + 1 + \delta X^{k+1} + \frac{\delta a^k - 1}{2} \quad (2)$$

- Si δa^k est impair et a_q^k est impair, alors

$$\delta a^{k+1} + \delta b^{k+1} = \delta b^k - 1 + \delta X^{k+1} + \frac{\delta a^k + 1}{2} \quad (3)$$

Prouvons maintenant que $\delta a^{k+1} + \delta b^{k+1} \geq 0$. Nous avons quatre cas :

1. $\delta a^k \geq 0$ et $\delta b^k \geq 0$

   Dans ce cas, (1) et (2) sont positifs. Pour (3), il suffit de remarquer que $\delta a^k \geq 1$ car δa^k est impair, pour en déduire que (3) est aussi positif.

2. $\delta a^k < 0$ et $\delta b^k \geq 0$

   Par induction, nous avons $\delta a^k + \delta b^k \geq 0$, alors (1) et (2) sont positifs. Pour (3), nous avons $\delta a^k + \delta b^k \geq 0$ et $\delta a^k < 0$ impliquent $\delta b^k \geq 1$. Alors, (3) est aussi positif.

3. $\delta a^k \geq 0$ et $\delta b^k < 0$

   Pour ce cas, nous devons prouver que $\delta a^k + 2\delta b^k \geq 0$.

   D'après ce qui précède, nous avons trois cas :

   (a) Du cas (1), nous avons $\delta a^k + 2\delta b^k = \delta b^{k-1} + 2\delta X^k + \delta a^{k-1}$

   (b) Du cas (2), nous avons $\delta a^k + 2\delta b^k = \delta b^{k-1} + 1 + 2\delta X^k + \delta a^{k-1} - 1$

(c) Du cas (3), nous avons $\delta a^k + 2\delta b^k = \delta b^{k-1} - 1 + 2\delta X^k + \delta a^{k-1} + 1$

Comme $\delta a^{k-1} + \delta b^{k-1} \geq 0$ (par hypothèse d'induction) et $\delta X^k \geq 0$, alors $\delta a^k + 2\delta b^k \geq 0$ dans les trois cas.
Comme $\delta a^k + 2\delta b^k \geq 0$, nous avons (1) et (2) positifs. Pour (3), nous avons δa^k impair implique $\delta a^k + 2\delta b^k$ impair, donc $\delta a^k + 2\delta b^k \geq 1$. Il en résulte (3) est positif.

4. $\delta a^k < 0$ et $\delta b^k < 0$

   Ce cas n'est pas possible puisque $\delta a^k + \delta b^k \geq 0$.

Nous concluons que $\delta a^{k+1} + \delta b^{k+1} \geq 0$. Ainsi, pour tout entier k, $\delta a^k + \delta b^k \geq 0$ implique $f_d^k(Q) \leq f_d^k(S)$. □

Les deux Propositions 2.3.18 et 2.3.19 impliquent la monotonie de f_d^k. Nous déduisons de cette monotonie, la Proposition 2.3.20 suivante.

Proposition 2.3.20. *Pour toute séquence d'entiers S,*

$$f_d^k(S) - 1 \leq f_d^k(S \backslash \{\min(S)\}) \leq f_d^k(S).$$

Démonstration. Dans cette preuve, nous appliquons le même raisonnement prouvant la Proposition 2.3.19. Prenons $k = \min(S)$ et $Q = S\backslash\{\min(S)\}$, alors $\delta X^k = 1$. Notons que $\delta X^{k+\alpha} = 0$, pour tout $\alpha > 0$. Nous avons ces différents cas de figures :

- $\delta a^{k-1} = 0$, $\delta b^{k-1} = 0$ (1) implique
- $\delta a^k = 0$ et $\delta b^k = 1$ (2), ce qui implique
- $\delta a^{k+1} = 1$ et $\delta b^{k+1} = 0$ (3), qui implique
- $\delta a^{k+2} = 1$ et $\delta b^{k+2} = 0$ (3) ou
  $\delta a^{k+2} = -1$ et $\delta b^{k+2} = 1$ (4), ce cas implique
- $\delta a^{k+3} = 0$ et $\delta b^{k+3} = 0$ (1) ou
  $\delta a^{k+3} = 2$ et $\delta b^{k+3} = -1$ (5), ce cas implique
- $\delta a^{k+4} = -1$ et $\delta b^{k+4} = 1$ (4)

Au delà de $k + 4$, aucun cas n'est rajouté. Nous avons alors que 5 cas de figures pour les valeurs de δa^k et δb^k pour tout $k \geq \min(S)$. Dans chacun de ces 5 cas, nous avons $0 \leq \delta a^k + \delta b^k \leq 1$. Nous concluons que $0 \leq f_d^k(S) - f_d^k(Q) \leq 1$. Ainsi, $f_d^k(Q) \leq f_d^k(S)$ et $f_d^k(S) - 1 \leq f_d^k(Q)$. La Proposition 2.3.20 est alors prouvée.
□

3.2.5 Codage *Polychotomique Généralisé*

Nous avons discuté le codage *Dichotomique* des arbres puis leur codage *Polychotomique* qui fait introduire la notion de flat-séquence. Nous analysons ici le codage *Polychotomique Généralisé* [Colomb *et al.*, 2008], décrit par l'Équation 2.9, qui améliore toutes ces heuristiques de codage des arbres par vecteur de bits précédemment discutées. Notons que l'Équation 2.9 correspond à une fonction de poids, qui calcule le poids d'un élément x à partir d'une séquence d'entiers S référant aux poids de ses fils. Ainsi, le codage *Polychotomique Généralisé* améliore le poids de x, calculé par le codage *Polychotomique* (resp. *Dichotomique*). Ce poids, par abus de langage, désigne la taille du codage de la séquence S.

$$\mathcal{G}(S) = \begin{cases} s_n + sp(n) & \text{si } S \text{ une } s_n\text{-flat-séquence,} \\ \mathcal{G}([s_k + sp(k), s_{k+1}, \ldots, s_n]) & \text{si } \exists k \leq n-1 \text{ avec} \\ & \ s_k - s_2 < 2 \text{ et } s_k + sp(k) \leq s_{k+1}, \\ \mathcal{G}([s_3, \ldots, s_2 + 2, \ldots, s_n]) & \text{sinon.} \end{cases} \tag{2.9}$$

Le codage *Polychotomique Généralisé* diffère du codage *Polychotomique* par le fait qu'il puisse substituer les k plus petits éléments d'une séquence en entrée S (les k premiers éléments) par un seul élément, mais à condition que la sous-séquence composée de ces k éléments soit une flat-séquence dont la taille de son codage *Polychotomique* ne dépasse pas la valeur du $(k+1)^{\text{ème}}$ élément de la séquence S (deuxième cas de l'Équation 2.9). Si cette condition n'est pas vérifiée, il substitue, pareillement au codage *Polychotomique*, les deux plus petits éléments de la séquence par leur maximum plus deux. Il répète le même traitement sur la séquence résultante et ainsi de suite jusqu'à ce que la séquence courante soit une $\max(S)$-flat-séquence, notons $f_g(S)$ sa taille. Il retourne ensuite

$$\mathcal{G}(S) = \max(S) + sp(f_g(S)). \tag{2.10}$$

Soit k le plus grand entier tel que $\mathcal{G}(S) = \mathcal{G}([s'_1, \ldots, s'_t, s_{k+1}, \ldots, s_n])$ et $[s'_1, \ldots, s'_t]$ est une flat-séquence avec $\mathcal{G}([s'_1, \ldots, s'_t]) = s'_t + sp(t) \leq s_{k+1}$. Il est à noter que $\mathcal{G}([s_1, \ldots, s_k]) = \mathcal{G}([s'_1, \ldots, s'_t])$. Nous reformulons ainsi le codage *Polychotomique Généralisé* comme suit :

$$\mathcal{G}(S) = \begin{cases} \mathcal{P}([\mathcal{G}([s_1, \ldots, s_k]), s_{k+1}, \ldots, s_n]) & \text{si } k > 1 \text{ est le plus grand entier t.q.} \\ & \mathcal{G}([s_1, \ldots, s_k]) \leq s_{k+1}, \\ \mathcal{P}(S) & \text{sinon.} \end{cases} \tag{2.11}$$

Colomb *et al.* [Colomb *et al.*, 2008] prouve que $\mathcal{G}(S) \leq \mathcal{P}(S)$ pour toute séquence d'entiers S. Nous expliquons ce résultat par le fait que le codage *Polychotomique Généralisé* génère une $\max(S)$-flat-séquence de taille plus petite que celle générée par le codage *Polychotomique* lors du codage de S (Théorème 2.3.18).

Ainsi, $f_g(S) \leq f_d(S)$ implique $\max(S) + sp(f_g(S)) \leq \max(S) + sp(f_d(S))$. Notons que sp (fonction sperner) est une fonction croissante. Depuis les Équations 2.10 et 2.7, nous retrouvons le résultat $\mathcal{G}(S) \leq \mathcal{P}(S)$. Le Théorème 2.3.18 évalue l'écart entre $f_g(S)$ et $f_d(S)$.

Théorème 2.3.18. *Pour toute séquence d'entiers S, $f_d(S) - 1 \leq f_g(S) \leq f_d(S)$.*

Démonstration. Nous proposons une preuve par induction sur la taille de S.
Pour $|S| \leq 2$, nous avons $f_d(S) = f_g(S)$. La conclusion de la proposition est satisfaite dans ce cas.
Pour $|S| > 2$, nous supposons que la conclusion de la proposition est satisfaite pour les séquences de taille au plus $|S| - 1$.
Soit $S = [s_1, \ldots, s_n]$. Nous supposons que S n'est pas une $\max(S)$-flat-séquence vu que $f_g(S) = f_d(S)$ le cas échéant.
Nous traitons les deux cas :

1. Lorsque la première condition de l'Équation 2.11 n'est pas vérifiée, le codage *Polychotomique Généralisé* se comporte comme le codage *Polychotomique*. Dans ce cas, il est évident que $f_d(S) = f_g(S)$.
2. Lorsqu'il existe un entier k, soit le plus grand, tel que $\mathcal{G}([s_1, \ldots, s_k]) \leq s_{k+1}$, le codage *Polychotomique Généralisé* substituera S par $[a, s_{k+1}, \ldots, s_n]$, la valeur a étant $\mathcal{G}([s_1, \ldots, s_k])$.

   Puisque $\mathcal{G}([s_1, \ldots, s_k]) \leq s_{k+1}$, le codage *Polychotomique Généralisé* génère une s_{k+1}-flat-séquence à partir de $[s_1, \ldots, s_k]$ de taille 1. Par hypothèse d'induction, le codage *Polychotomique* génère alors, à partir de la même sous-séquence, une s_{k+1}-flat-séquence de taille 1 ou 2.

   Comme le codage *Polychotomique* se comporte comme l'algorithme *Dicho* lorsque S n'est pas une flat-séquence (Équation 2.3), il substituera alors S par $F_d^{[s_1,\ldots,s_{k+1}]} \cup [s_{k+2}, \ldots, s_n]$ (Équation 2.5), avec $F_d^{[s_1,\ldots,s_{k+1}]}$ une s_{k+1}-flat-séquence de taille 2 ou 3. Nous considérons la plus longue et nous la notons $[b, c, s_{k+1}]$. Par conséquent, calculer $\mathcal{P}(S)$ revient à calculer $\mathcal{P}([b, c, s_{k+1}, \ldots, s_n])$, ce qui implique $f_d(S) = f_d([b, c, s_{k+1}, \ldots, s_n])$.

   Selon l'Équation 2.11 et d'après la maximalité de l'entier k, le calcul de $\mathcal{G}(S)$ se ramène au calcul de $\mathcal{P}([a, s_{k+1}, \ldots, s_n])$, ou encore de $\mathcal{P}([c, s_{k+1}, \ldots, s_n])$. Alors, $f_g(S) = f_d([c, s_{k+1}, \ldots, s_n])$.

   Selon la Proposition 2.3.20, nous avons

   $$f_d([b, c, s_{k+1}, \ldots, s_n]) - 1 \leq f_d([c, s_{k+1}, \ldots, s_n]) \leq f_d([b, c, s_{k+1}, \ldots, s_n]).$$

   Nous concluons que

   $$f_d(S) - 1 \leq f_g(S) \leq f_d(S).$$

Les deux cas traités prouvent la Proposition 2.3.18. □

3.2.6 Conclusion

Nous rappelons que nous nous sommes focalisés sur les fonctions de poids afin de discuter les éventuelles améliorations du codage *Dichotomique*. Comme nous l'avions détaillé pour ce codage, une fonction de poids décrit implicitement un plongement d'un arbre initial dans un arbre de structure différente (arbre binaire dans le cas d'un codage *Dichotomique*). Pour expliciter cela, nous supposons un appel d'une fonction de poids sur une séquence d'entiers $S = [s_1, \ldots, s_n]$, permettant de calculer le poids d'un élément x, d'un arbre T, dont les fils ont les poids $s_1, \ldots, s_n$. Cet appel peut engendrer une substitution des entiers $s_1, \ldots, s_k$ par un entier t qui s'interprète par le rajout d'un nouvel élément de poids t comme fils de x et parent des k éléments de poids respectifs $s_1, \ldots, s_k$. Par conséquent, en calculant le poids de chaque élément de T, nous obtenons à la fin un nouvel arbre T' dans lequel se plonge T. Notons que le codage de T' par l'heuristique *CHNR* induit un codage par vecteur de bits de T dont la taille correspond au poids attribué à sa racine. Cela est justifié par le fait que l'heuristique *CHNR* code les sous-arbres de T' dont chacun est enraciné en un fils de x dans T' avec un ensemble d'au plus s_n couleurs (en raisonnant par récurrence) et code les m fils de x dans T' avec $sp(m)$ couleurs supplémentaires. Ce codage est de taille $s_n + sp(m)$ et correspond effectivement au poids attribué à x par le codage *Polychotomique* et le codage *Polychotomique Généralisé*. Il s'agit de la taille du codage d'une s_n-flat-séquence : le cas d'arrêt de leur fonctions de poids récursives.

3.3 Stratégie

Suite à l'analyse établie sur le comportement des trois dernières heuristiques de codage des arbres par vecteur de bits, nous constatons que ces heuristiques reposent sur l'attribution d'un poids à chaque élément d'un arbre suivant un ordre topologique inverse de ses éléments. De plus, elles déterminent le poids de chaque élément x, dont les n fils ont les poids respectifs $s_1 \leq \cdots \leq s_n$, à partir du codage d'une s_n-flat-séquence qu'elles tentent de générer depuis la séquence $[s_1, \ldots, s_n]$. La stratégie commune de ces heuristiques, ainsi constatée, permet de définir une classe d'algorithmes de codage des arbres par vecteur de bits. L'efficacité de ces algorithmes peut dépendre de la taille de la flat-séquence générée et de son codage. Leurs instructions principales sont :

```
S ← les poids des fils d'un élément x
F ← max(S)-flat-séquence générée à partir de S
poids(x) ← taille du codage de F
```

À présent, le meilleur codage d'une k-flat-séquence de taille n sollicite $k+sp(n)$ bits. Il s'agit du nombre de bits minimum codant des 0-flat-séquences, car cela se ramène au codage des antichaînes. Cependant, nous ne pouvons pas généraliser ce résultat d'optimalité vu qu'il existe des k-flat-séquences à n éléments admettant un codage par $k + sp(n) - 1$ bits (voir la Figure 2.21 pour une illustration). Tout de même, nous utilisons dans la suite, ce meilleur codage pour coder les flat-séquences et nous cherchons des algorithmes réduisant plutôt leur taille. Nous proposons, comme solution, le *Partitionnement Contigu Généralisé*. Nous décrivons juste après cette solution.

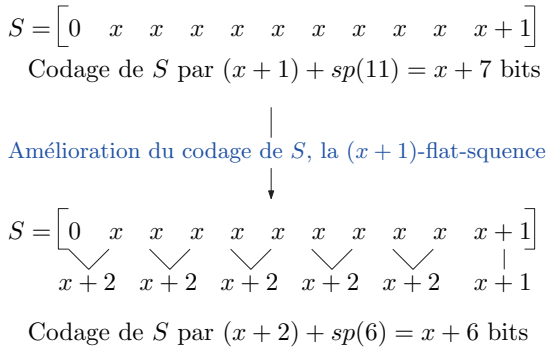


FIGURE 2.21: Codage plus efficace des flats-séquences

3.4 Description de l'heuristique

Nous proposons dans cette sous-section une heuristique de codage des arbres par vecteur de bits, suivant la même stratégie heuristique des codages *Dichotomique*, *Polychotomique* et *Polychotomique Généralisé*. Nous rappelons que cette stratégie consiste à attribuer un poids à chaque élément d'un arbre, calculé à partir des poids de ses fils $s_1, \ldots, s_n$, en s'appuyant sur le codage d'une s_n-flat-séquence générée depuis la séquence $[s_1, \ldots, s_n]$.

Définition 2.3.11 (Partitionnement contigu)**.** *Soit S une séquence d'entiers et soit $P = \{X_1, \ldots, X_k\}$ un partitionnement de S. Ce partitionnement est contigu si pour tous i et j tels que $1 \leq i \leq j \leq k$, $\max(X_i) \leq \min(X_j)$.*

Soit $S = [s_1, \ldots, s_n]$ une séquence d'entiers correspondant aux poids des fils d'un élément x. Afin de calculer le poids de x, nous proposons la fonction de poids $\mathcal{C}$, définie par l'Équation 2.12, qui génère une s_n-flat-séquence à travers un partitionnement contigu de S. Rappelons que le codage de cette flat-séquence, notons $f_c(S)$ sa taille, nécessite $s_n + sp(f_c(S))$ bits, comme nous l'avions fixé préalablement.

$$\mathcal{C}(S) = \begin{cases} s_n + sp(n) & \text{si } S \text{ est une } s_n\text{-flat-séquence,} \\ \mathcal{C}([s_{k+1}, .., \max\{\mathcal{C}([s_1, .., s_k]), s_n\}, .., s_n]) & \text{si } \exists k \text{ tel que } k < n-1 \\ & \text{et } \mathcal{C}([s_1, \ldots, s_k]) \leq s_n \\ & \text{et } \mathcal{C}([s_1, \ldots, s_k, s_{k+1}]) > s_n, \\ s_n + 2 & \text{sinon, i.e. } \mathcal{C}([s_1, \ldots, s_{n-1}]) \leq s_n \end{cases} \tag{2.12}$$

La fonction de poids $\mathcal{C}$ (Équation 2.12), calcule un flat-partitionnement de S qui est à la fois contigu et de taille minimale. En effet, toute sous-séquence $[s_i, .., s_{i+\alpha}]$, détectée à chaque itération, regroupe le maximum d'éléments successifs de S, telle qu'elle peut être codée par au plus s_n bits. Malheureusement, cette fonction est parfois moins efficace que la fonction du poids $\mathcal{G}$ (Équation 2.9) associée au codage *Polychotomique Généralisé*. Par exemple, en prenant $S = [1]^{36} \cup [2, 3, 9]$, nous avons d'une part $\mathcal{C}(S) = 12$ et d'autre part $\mathcal{G}(S) = 11$. Nous détaillons ce calcul dans la Figure 2.22.

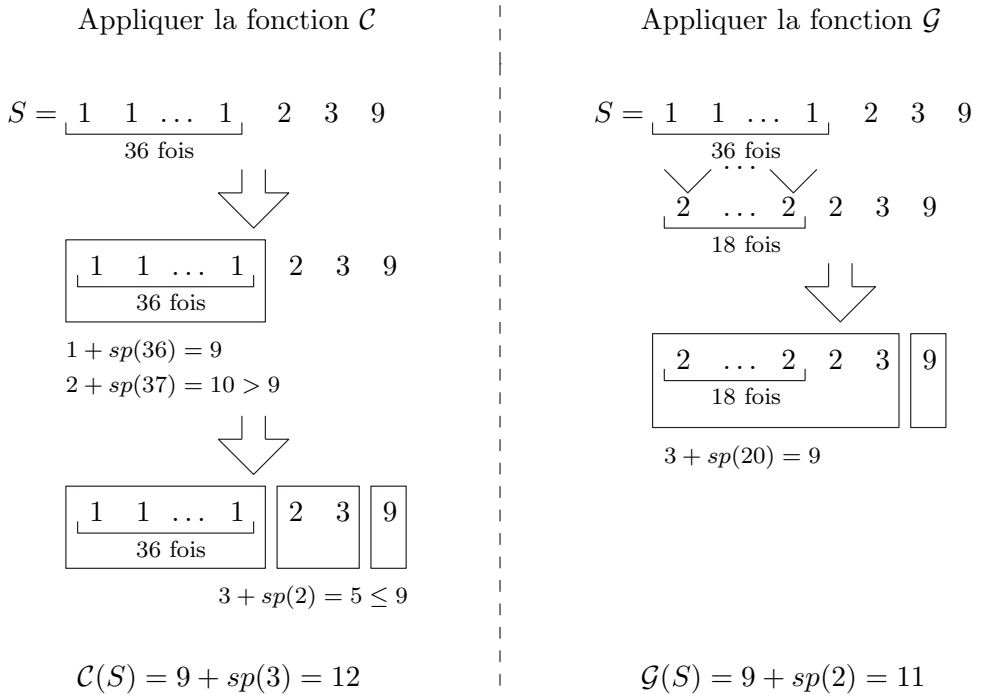


FIGURE 2.22: Codage d'une séquence par la fonction de poids $\mathcal{C}$ *vs* $\mathcal{G}$

Nous rectifions alors la fonction $\mathcal{C}$, et nous proposons à sa place la fonction de poids $\mathcal{GC}$ définie comme suit :

$$\mathcal{GC}(S) = \begin{cases} \mathcal{C}([0, s_{k+1}, \ldots, s_n]) & \text{si } k > 1 \text{ est le plus grand entier tel que} \\ & \mathcal{G}([s_1, \ldots, s_k]) \leq s_{k+1}, \\ \mathcal{C}([s_1, \ldots, s_n]) & sinon. \end{cases} \tag{2.13}$$

La fonction de poids $\mathcal{GC}$ génère une $\max(S)$-flat-séquence, grâce à un $\max(S)$-flat-partitionnement contigu de S. En notant $f_{gc}(S)$ la taille de cette flat-séquence (aussi ce flat-partitionnement), nous obtenons

$$\mathcal{GC}(S) = \max(S) + sp(f_{gc}(S)). \tag{2.14}$$

Nous proposons maintenant une heuristique de calcul d'un codage par vecteur de bits des arbres, nommée *Partitionnement Contigu Généralisé*. Le principe de cette heuristique est le suivant : étant donné un arbre T, nous calculons le poids de chacun de ses éléments, suivant leur ordre topologique inverse, au moyen de la fonction de poids $\mathcal{GC}$ (Équation 2.13). Le calcul du poids des éléments de T permet de plonger cet arbre dans un arbre T'. En effet, une substitution d'une sous-séquence $[s_1, \ldots, s_k]$ par un entier t, lors du calcul du poids d'un élément x, entraîne l'introduction d'un nouvel élément de poids t comme fils de x et parent de ses k successeurs immédiats de poids respectifs $s_1, \ldots, s_k$. Ainsi, le codage de T' par l'algorithme *CHNR* implique un codage par vecteur de bits de T via le *Partitionnement Contigu Généralisé*. Nous donnons dans ce qui suit un algorithme polynomial calculant ce codage puis nous montrons qu'il améliore le codage *Polychotomique Généralisé*.

3.5 Algorithme et Complexité

Nous présentons ici un algorithme polynomial de codage des arbres par vecteur de bits *via* le *Partitionnement Contigu Généralisé*. Nous discutons d'abord l'implémentation de la fonction de poids $\mathcal{GC}$.

Soit $S = [s_1, \ldots, s_n]$ une séquence d'entiers. Nous supposons que la valeur du plus grand entier k, telle que $\mathcal{G}([s_1, \ldots, s_k]) < s_{k+1}$, est déjà calculée en temps linéaire par le codage *Polychotomique Généralisé*. Une implémentation naïve de la fonction de poids associée à notre heuristique – $\mathcal{GC}$ – est alors basée sur la définition formelle de la fonction $\mathcal{C}$ (Équation 2.12). Nous expliquons pourquoi une telle implémentation n'est pas recommandée.

Afin de résoudre $\mathcal{C}(S)$, l'algorithme naïf procède à la recherche de la plus longue sous-séquence contiguë de S codée par au plus s_n bits. Ainsi, il commence par calculer $\mathcal{C}([s_1, s_2])$, puis $\mathcal{C}([s_1, s_2, s_3])$, etc., jusqu'à trouver $S' = [s_1, \ldots, s_k]$, avec $k < n$, telle que $\mathcal{C}(S') > s_n$. Il reprend ensuite ce traitement sur la séquence $[s_k, \ldots, s_n]$. Notons que résoudre le sous-problème $\mathcal{C}(S')$ nécessite le re-calcul de $\mathcal{C}([s_1, s_2])$,

puis $\mathcal{C}([s_1, s_2, s_3])$, etc., qui étaient déjà calculés mais ignorés par l'algorithme naïf. Pour accélérer ce processus, il faut éviter de résoudre les sous-problèmes plusieurs fois en *mémorisant* leurs résultats. Cette technique caractérise la *programmation dynamique*.

Nous considérons alors une implémentation de la fonction de poids $\mathcal{GC}$ par un algorithme de programmation dynamique. Pour cela, nous utilisons une matrice $\mathcal{M}_{n,n}$ pour mémoriser les calculs déjà effectués : l'élément $a_{i,j}$ de la matrice correspond à $\mathcal{C}([s_i, \ldots, s_j])$, si $i < j$, à s_i si $i = j$ et à l'élément nul 0 sinon, ainsi que l'élément $a_{1,n}$ correspond à $\mathcal{C}(S)$. Le calcul des éléments de la matrice s'effectue progressivement par le calcul de ceux de $\mathcal{M}_{1,1}$ puis $\mathcal{M}_{2,2}$ et ainsi de suite.

Afin de calculer $\mathcal{C}([s_i, \ldots, s_j])$, pour $i < j$, nous cherchons, à la $i^{\text{ème}}$ ligne de la matrice, le premier indice k tel que $a_{i,k} > s_j$ et $k < j$. Rappelons que les éléments de la matrice $\mathcal{M}_{j-1,j-1}$ sont déjà calculés. Puis, nous reprenons cette recherche à la $k^{\text{ème}}$ ligne. Nous continuons ainsi jusqu'à ce qu'aucune nouvelle ligne ne puisse être visitée. Ainsi, $\mathcal{C}([s_i, \ldots, s_j]) = s_j + sp(r+1)$, r étant le nombre de lignes visitées. Ce résultat est sauvegardé par l'élément $a_{i,j}$ et il est calculé en temps $\mathcal{O}(n)$.

L'Algorithme 22 détaille les instructions de calcul d'un codage des arbres *via* le *Partitionnement Contigu Généralisé*.

Données : Un arbre T
Résultat : Un codage de T *via* un *Partitionnement Contigu Généralisé*
`/* initialisation des poids des éléments de` T `*/`
pour *tout* x *dans* T **faire**
  si x *est une feuille* **alors**
    $w(x) \leftarrow 0$
  fin
  sinon
    $w(x) \leftarrow -1$
  fin
fin
`/* Plongement de` T `dans un arbre` T' `*/`
tant que *il existe un élément* x *de poids inférieur à zéro et dont tous les fils ont des poids positifs* **faire**
  $S \leftarrow$ une séquence croissante des poids des fils de x
  si S *est une flat-séquence* **alors**
    $w(x) \leftarrow \max(S) + sp(|S|)$
  fin
  sinon
    `/* Appel du codage` ***Polychotomique Généralisé*** `*/`
    $k \leftarrow$ le plus grand entier tel que $\mathcal{G}([s_1, \ldots, s_k]) \leq s_{k+1}$
    si $k > 1$ **alors**
      Introduire un nouvel élément s comme fils de x et père des éléments de poids $s_1, \ldots, s_k$
      $w(s) = 0$
      sinon
        `/* Calcul par programmation dynamique */`
        Calculer la matrice des poids $\mathcal{M}_{n,n}$
        $k \leftarrow$ le plus grand entier tel que $\mathcal{C}([s_1, \ldots, s_k]) \leq \max(S)$
        si $k < n - 1$ **alors**
          Introduire un nouvel élément s comme fils de x et père des éléments de poids $s_1, \ldots, s_k$
          $poids(s) = \max(\mathcal{C}([s_1, .., s_k]), \max(S))$
        fin
        si $k = n - 1$ **alors**
          Introduire un nouvel élément s comme fils de x et père des éléments de poids $s_1, \ldots, s_{n-1}$
          $w(s) = \mathcal{C}([s_1, .., s_{n-1}])$
          $w(x) = s_n + 2$
        fin
      fin
    fin
  fin
fin
`/* Les éléments de` T `sont codés lors du codage de` T' `*/`
Coder l'arbre T' par l'algorithme *CHNR*

Algorithme 9 : Codage des arbres par l'algorithme de *Partitionnement Contigu Généralisé*

Théorème 2.3.19. *L'Algorithme 9 calcule un codage des arbres par vecteur de bits en temps $\mathcal{O}(n^5)$ et en espace $\mathcal{O}(n^2 \cdot e)$, n et e étant respectivement la taille de l'arbre et la taille du codage.*

Démonstration. Comme l'Algorithme 22 récupère le codage de T à partir d'un codage par vecteur de bits de T' (codage par l'algorithme *CHNR*) car $T \rightsquigarrow T'$, il est normal que cet algorithme fournisse un codage par vecteur de bits de T. C'est ainsi que nous justifions la correction de l'algorithme. Nous calculons maintenant sa complexité.

Le premier bloc d'instructions, de la ligne 1 à la ligne 8, concerne l'initialisation des poids des éléments de T. Cela se fait en temps et en espace $\mathcal{O}(n)$.

Le deuxième bloc d'instructions, de la ligne 9 à la ligne 34, permet de créer le plongement de T dans T'. Ce plongement consiste à générer, pour tout élément x, une partition de ses fils en groupe puis à ajouter un nouveau parent pour chaque groupe. Cela est décrit implicitement par les Équations 2.12 et 2.13. Comme le nombre des fils d'un élément de T est au plus $n-1$, nous en déduisons que la taille de T' est en $\mathcal{O}(n^2)$. De plus, nous pouvons libérer l'espace mémoire occupé par la matrice de poids $\mathcal{M}_{n,n}$, qui est de taille n^2, à chaque itération. Par conséquent, la complexité en espace de ce bloc d'instructions est $\mathcal{O}(n^2)$. Nous calculons maintenant sa complexité en temps. Nous supposons que les opérations arithmétiques de base (addition, multiplication, division, comparaison, etc.) s'exécutent en temps constant.
La boucle à la ligne 9 indique que les instructions du bloc s'effectuent au plus $\mathcal{O}(n^2)$ fois. En effet, pour un élément x de T, il est possible de ré-exécuter le corps de la boucle à chaque fois qu'un nouvel élément est introduit dans T comme fils de x. Ainsi, le nombre total d'exécutions de la boucle est $|T'| \leq n^2$. Nous évaluons la complexité en temps des autres instructions du bloc.

- L'instruction à la ligne 10 permet de récupérer en temps $\mathcal{O}(n^2)$ les poids des fils d'un élément de l'arbre triés par ordre croissant.
- L'instruction à la ligne 12 calcule le poids d'un élément en temps $\mathcal{O}(\log_2(n))$. En effet, afin de trouver la valeur de $sp(n)$, il faut calculer $\binom{t}{\lfloor \frac{t}{2} \rfloor}$ pour tout $1 \leq t \leq sp(n)$. Rappelons que $sp(n) = \min\{t | \binom{t}{\lfloor \frac{t}{2} \rfloor} \geq n\}$. Nous supposons que ce calcul s'effectue en temps constant si nous considérons que le calcul de $\binom{t+1}{\lceil \frac{t+1}{2} \rceil}$ consiste à multiplier la valeur de $\binom{t}{\lceil \frac{t}{2} \rceil}$, déjà calculée, par $(t+1)\frac{1}{\lceil \frac{t+1}{2} \rceil}$. Ainsi, nous comptons $sp(n)$ calculs à effectuer, soit $\mathcal{O}(\log_2(n))$ calculs selon l'Inéquation 2.8.
- L'instruction à la ligne 15 calcule un codage *Polychotomique Généralisé* de la séquence S en temps $\mathcal{O}(n)$ afin de récupérer la valeur de l'entier k.

– Les instructions aux lignes 17, 23 et 27 permettent d'introduire un nouvel élément dans l'arbre en temps $\mathcal{O}(n)$. Nous supposons une représentation de l'arbre par une liste de successeurs immédiats.
– Les instructions élémentaires aux lignes 18, 24, 28 et 29 s'effectuent en temps constant.
– L'instruction à la ligne 20 calcule la matrice de poids en temps $\mathcal{O}(n^3)$. En effet, comme nous l'avons suggéré au début de cette sous-section, cette matrice permet de calculer efficacement le poids d'un élément de l'arbre par la fonction $\mathcal{C}$ (Équation 2.12). Nous avons aussi mentionné que chaque élément de cette matrice est calculé en temps $\mathcal{O}(n)$.
– L'instruction à la ligne 21 s'effectue en temps $\mathcal{O}(n)$ car il s'agit de parcourir n éléments de la matrice $\mathcal{M}_{n,n}$, comme décrit auparavant.

Enfin, la complexité en temps du second bloc d'instructions est $\mathcal{O}(n^5)$.

Concernant la dernière instruction (ligne 35), elle permet de calculer un codage de T' par l'algorithme *CHNR*. Selon la description de cet algorithme, donnée à la page 57, nous avons besoin de générer n sous-ensembles de taille k d'un ensemble X de $sp(n)$ éléments, n étant ici le nombre maximum de fils d'un élément de T' (la valeur de $sp(n)$ est déduite de l'instruction à la ligne 12). Une façon simple de le faire est d'énumérer tous les vecteurs de bits de taille $sp(n)$ et de ne garder que ceux ayant k bits à un. Nous pouvons utiliser le codage de Gray [Gray, 1953] pour énumérer ces vecteurs de bits en temps et en espace $\mathcal{O}(n \log_2(n))$ (exponentiel par rapport à $sp(n)$ dont la valeur est en $\mathcal{O}(\log_2(n))$ d'après l'Inéquation 2.8). La transformation d'un vecteur de bits V en un sous-ensemble $\{a_1, \ldots, a_k\}$ de X est immédiate : l'élément a_i, pour i de 1 à k, est le $j^{\text{ème}}$ élément de X tel que le $j^{\text{ème}}$ bit de V correspond à son $i^{\text{ème}}$ bit à un. Cette transformation se fait en $\mathcal{O}(\log_2(n))$ (linéaire par rapport à $sp(n)$: la taille de V et de X). Ainsi, nous pouvons récupérer tous les sous-ensembles de X de taille k en temps et en espace $\mathcal{O}(n \log_2(n))$.
Le codage *CHNR* consiste alors à générer, pour tout élément x de T' de degré sortant $d \leq n$, tous les sous-ensembles de taille $\lceil \frac{d}{2} \rceil$ d'un ensemble de d éléments, puis à coder chaque fils de x par l'union disjointe de $code(x)$ et l'un des sous-ensembles générés. Cette union est calculable en temps et en espace $\mathcal{O}(e)$, e étant la taille du codage avec $e \leq n$. Notons que l'espace mémoire dédié pour générer les sous-ensembles de taille $\lceil \frac{d}{2} \rceil$ peut être libéré à la fin du traitement de l'élément x. Nous précisons que la racine de T' est initialement codée par $\emptyset$.
Nous en déduisons que la dernière instruction de l'algorithme s'effectue en temps $\mathcal{O}(n^3 \log_2(n))$ et en espace $\mathcal{O}(n^2 \cdot e)$. Rappelons que $|T'| \leq n^2$.

Nous concluons que la complexité en temps de l'Algorithme 22 est $\mathcal{O}(n^5)$ ainsi que sa complexité en espace est $\mathcal{O}(n^2 \cdot e)$.

□

3.6 Résultats théoriques

Nous avons proposé dans cette section une nouvelle heuristique de codage des arbres par vecteur de bits, le *Partitionnement Contigu Généralisé*. Nous montrons ici que notre heuristique améliore la taille du codage des arbres *via* le *Polychotomique Généralisé*. Notre preuve s'appuie sur le Lemme 2.3.1.

Lemme 2.3.1. *Soit $S = [s_1, \ldots, s_n]$ une séquence d'entiers.*

1. *Pour tout entier k, $f_c(S) \leq f_c([s_1, \ldots, s_k, s_n]) - 1 + f_c([s_{k+1}, \ldots, s_n])$.*
2. *Pour k tel que $\mathcal{D}([s_1, \ldots, s_k, s_{k+1}]) > s_n$, $1 + f_d([s_{k+1}, \ldots, s_n]) \leq f_d(S)$.*

Démonstration. La première propriété est évidente puisque le partitionnement de S est contigu.

Pour prouver la seconde propriété, nous supposons qu'il existe un entier k tel que $\mathcal{D}([s_1, \ldots, s_k, s_{k+1}]) > s_n$. Notons A la sous-séquence $[s_1, \ldots, s_k]$ et B la sous-séquence $[s_{k+1}, \ldots, s_n]$. Nous devons prouver que $1 + f_d(B) \leq f_d(S)$ qui s'exprime également par $1 + f_d^{s_n}(B) \leq f_d^{s_n}(S)$.

Pour tout entier x, notons S^x la plus longue sous-séquence de S (potentiellement vide) telle que $\max(S^x) \leq x$. Nous appelons une partition de S^x *good*, s'il s'agit d'un x-flat-partitionnement (Définition 2.3.9) de S^x, par un algorithme *Dichotomique* quelconque, de taille $f_d^x(S^x)$ – la taille d'un x-flat-partitionnement généré par l'algorithme *Dicho*. Une telle *good* partition existe pour tout entier x puisque l'algorithme *Dicho* en produit une. En effet, il peut toujours générer une x-flat-séquence (donc un x-flat-partitionnement) à partir de S^x (cf. la Sous-section 3.2). Pour toute partition P de S^x, notons $M(P)$ le nombre d'éléments de P ayant une intersection non vide avec A et B.

Focalisons-nous sur les entiers x tels que $s_n - x$ est pair ou $x \geq s_n$ et tels qu'il existe une *good* partition P de S^x avec $M(P) = 0$, i.e. une partition dont les éléments sont inclus soit dans A soit dans B. De tels entiers sont bien définis puisque s_k ou $s_k + 1$ satisfait ces conditions. En effet, d'après le paragraphe précédent, pour x dans $\{s_k, s_k + 1\}$, S^x admet au moins une *good* partition P. Nous considérons celle générée par l'algorithme *Dicho*. Pour une contradiction, supposons que $M(P) \geq 1$ et soit X un élément de P ayant une intersection non vide avec A et B. Alors, il existe une sous-sequence $[y, z]$ de X telle que y dans A et z dans B, ce qui signifie $y \leq s_k \leq z$. Selon la Proposition 2.3.19, nous avons $f_d^x([y, z]) \leq f_d^x(X)$ car $[y, z] \subseteq X$. Comme $\mathcal{D}([y, z]) = z + 2 > s_k + 1 \geq x$, $f_d^x([y, z]) > 1$. Alors, $f_d^x(X) > 1$ implique $\mathcal{D}(X) > x$ et il s'agit de la plus petite valeur d'un codage *Dichotomique* de X (Théorème 2.3.16). Cela contredit le fait que P soit un x-flat-partitionnement de S^x par $\mathcal{D}$.

Nous considérons maintenant le plus grand entier x tel que S^x possède une *good* partition P avec $M(P) = 0$ et $s_n - x$ pair ou $x \geq s_n$. Nous distinguons deux cas :

Cas.1 $x \geq s_n$

Soit L un élément de P tel que $L \subseteq A$. Nous avons $P\backslash\{L\}$ un x-flat-partitionnement de $S\backslash L$, par un algorithme *Dichotomique*. Cet algorithme génère alors une x-flat-séquence de taille $|P\backslash\{L\}|$ à partir de $S\backslash L$. Selon le Théorème 2.3.17, nous avons $f_d^x(S\backslash L) \leq |P\backslash\{L\}|$. Alors $1 + f_d^x(S\backslash L)$ vaut au plus $|P|$. Comme P est une *good* partition, $|P| = f_d^x(S)$. Nous en dérivons $1 + f_d^x(S\backslash L) \leq f_d^x(S)$.

Comme $B \subseteq S\backslash L$, $f_d^x(B) \leq f_d^x(S\backslash L)$ (Proposition 2.3.19). Par conséquent, $1 + f_d^x(B) \leq f_d^x(S)$, ce qui est vrai pour $x = s_n$. La conclusion du lemme est satisfaite dans ce cas.

Cas.2 $x \leq s_n$ et $s_n - x$ pair

Prouvons d'abord que pour tout entier t, avec $1 \leq t \leq \frac{s_n - x}{2}$, il existe une *good* partition Q de S^{x+2t} telle que $M(Q) = 1$, i.e. Q possède exactement une sous-séquence ayant une intersection non vide avec A et B. Nous proposons une preuve par induction :

– Cas de base : $t = 1$

Nous essayons de construire une *good* partition Q de S^{x+2} avec $M(Q) = 1$. Nous avons P une *good* partition de S^x avec $M(P) = 0$ (définie juste avant Cas.1). Nous considérons les notations P_A et P_B, avec $P_A \cup P_B = P$, et

$$P_A = \{U_1, \ldots, U_h\} \cup \{V_1, \ldots, V_h\} \cup I$$

avec $U_i \cup V_i \subseteq A$ pour tout $1 \leq i \leq h$ et $h = \lfloor \frac{P_A}{2} \rfloor$

$$P_B = \{W_1, \ldots, W_l\} \cup \{Z_1, \ldots, Z_l\} \cup J$$

avec $W_i \cup Z_i \subseteq B$ pour tout $1 \leq i \leq l$ et $l = \lfloor \frac{P_B}{2} \rfloor$.

Soit $Q' = \{\{U_i \cup V_i\} | 1 \leq i \leq h\} \cup \{\{W_i \cup Z_i\} | 1 \leq i \leq l\} \cup \{I \cup J\}$. Il est clair que $|Q'| = \lceil \frac{|P|}{2} \rceil$. Ainsi, pour tout élément de Q' (sauf au plus un), prenons $W_1 \cup Z_1$ sans perte de généralité, nous avons $x < \mathcal{D}(W_1 \cup Z_1) \leq x + 2$. Pour prouver cette inégalité, nous avons d'une part $\{W_1, Z_1\}$ un x-flat-partitionnement de $W_1 \cup Z_1$ de taille 2, implique, d'après le Théorème 2.3.17, $f_d^x(W_1 \cup Z_1) \leq 2$. Par conséquent, $\mathcal{D}(W_1 \cup Z_1) \leq x + 2$. D'autre part, si $\mathcal{D}(W_1 \cup Z_1) \leq x$, alors, en substituant W_1 et Z_1 par $W_1 \cup Z_1$ dans P, nous obtenons un x-flat-partitionnement de S^x de taille strictement inférieure à $f_d^x(S^x)$. Comme cela contredit le Théorème 2.3.17, nous avons bien $x < \mathcal{D}(W_1 \cup Z_1)$.

Soit R la plus longue sous-séquence de S dont les éléments sont égaux à $x+1$ ou $x+2$. Alors, $S^{x+2} = S^x \cup R$ ce qui implique $f_d^{x+2}(S^{x+2})$ vaut $f_d^{x+2}(S^x) + |R|$ (R étant déjà une $(x+2)$-flat-sequence). Ainsi, nous avons $f_d^{x+2}(S^{x+2}) = \lceil \frac{f_d^x(S^x)}{2} \rceil + |R|$. En effet, à partir d'une x-flat-séquence $[a_1, \ldots, a_m]$, l'algorithme *Dicho* génère une $(x+2)$-flat-séquence en substitution a_1 et a_2 par $a_2 + 2$, puis a_3 et a_4 par $a_4 + 2$ et ainsi de suite. La taille de la flat-séquence ainsi générée est $\lceil \frac{m}{2} \rceil$.

Finalement, nous considérons $Q = Q' \cup \{\{r\} | r \in R\}$. D'après les deux paragraphes précédents, nous avons Q un $(x+2)$-flat-partitionnement de S^{x+2}. La taille de ce partitionnement est $\lceil \frac{|P|}{2} \rceil + |R|$ qui coïncide, d'après le paragraphe précédent, avec $f_d^{x+2}(S^{x+2})$. Nous en déduisons que Q est une *good* partition de S^{x+2}. Si $M(Q) = 0$, cela contredit le fait que x est le plus grand entier tel que S^x admet une *good* partition P avec $M(P) = 0$ (voir la condition énoncée juste avant Cas.1). Alors $M(Q) > 1$. Nous avons les éléments de Q sont soit des singletons soit des éléments de Q' dont seul $I \cup J$ pouvant avoir une intersection non vide avec A et B. Par conséquent, $M(Q)$ vaut exactement 1.

Nous concluons que la propriété d'inclusion est vérifiée pour le cas de base.

– Cas récursif

Supposons que la propriété d'induction est vraie pour $x + 2t$. Soit P' une *good* partition de S^{x+2t} avec $M(P') = 1$ et soit C l'élément de P' ayant une intersection non vide avec A et B. Nous construisons une partition Q' à partir de $P' \backslash C$, de la même manière que nous avons construit Q' à partir de P dans le cas de base. Nous remplaçons ensuite $I \cup J$ par les deux éléments $C \cup I$ et J. Ainsi, nous obtenons $M(Q') = 1$.

Soit R la plus longue sous-séquence de S dont les éléments sont égaux à $x + 2t + 1$ ou $x + 2t + 2$. En suivant le même raisonnement que le cas de base, nous concluons que $Q = Q' \cup \{\{r\} | r \in R\}$ est une *good* partition de S^{x+2t+2} avec $M(Q) = 1$. La propriété d'inclusion est alors vérifiée pour tout entier t entre 1 et $\frac{s_n - x}{2}$.

La propriété que nous venons de prouver par induction implique l'existence d'une *good* partition Q de S^{s_n} contenant un unique élément X ayant une intersection non vide avec A et B (prendre $t = \frac{s_n - x}{2}$).

Supposons que $A \subseteq X$. Alors, pour s_t un élément de $X \cap B$, nous avons

$$[s_1, \ldots, s_k, s_{k+1}] \leq_{lexico} [s_1, \ldots, s_k, s_t] \text{ et } [s_1, \ldots, s_k, s_t] \subseteq X.$$

Selon les Propositions 2.3.18 et 2.3.19, nous avons

$$f_d^{s_n}([s_1, \ldots s_k, s_{k+1}]) \leq f_d^{s_n}([s_1, \ldots, s_k, s_t]) \leq f_d^{s_n}(X).$$

Comme $\mathcal{D}([s_1, \ldots, s_{k+1}]) > s_n$ (énoncé du lemme), $f_d^{s_n}([s_1, \ldots, s_{k+1}]) \geq 2$. Alors $f_d^{s_n}(X) \geq 2$ implique $\mathcal{D}(X) > s_n$. Comme X est un élément de Q, cela contredit le fait que Q est une *good* partition de S^{s_n}. Notre hypothèse, $A \subseteq X$, n'est pas correcte. Par conséquent, il existe un élément L de Q tel que $L \subseteq A$. Suivant le même raisonnement de Cas.1, nous obtenons

$$1 + f_d^{s_n}(B) \leq 1 + f_d^{s_n}(S \backslash L) \leq 1 + |Q \backslash \{L\}| = f_d^{s_n}(S).$$

La conclusion du lemme est aussi satisfaite dans ce second cas.

Les deux cas traités permettent de valider le Lemme 2.3.1. □

Le Théorème suivant affirme que notre heuristique améliore la taille du codage des arbres par vecteur de bits. Rappelons que son principe ainsi que celui de la meilleure heuristique existante – *Polychotomique Généralisé* – repose sur l'affectation d'un poids, à chaque élément d'un arbre donné, désignant la taille du codage du sous-arbre enraciné en cet élément. Ainsi, il suffit de prouver que notre heuristique minimise le poids attribué aux éléments de l'arbre. Comme nous l'avions introduit dans cette section, la fonction $\mathcal{GC}$ (resp. $\mathcal{G}$) correspond à la fonction de poids associée à notre heuristique (resp. l'heuristique *Polychotomique Généralisé*). Montrons alors, pour toute séquence d'entiers S, $\mathcal{GC}(S) \leq \mathcal{G}(S)$.

Théorème 2.3.20. *Pour toute séquence d'entiers S,*

$$f_{gc}(S) \leq f_g(S) \quad \textit{implique} \quad \mathcal{GC}(S) \leq \mathcal{G}(S).$$

Démonstration. Nous supposons que la séquence S n'est pas une $\max(S)$-flat-séquence, car la conclusion du théorème demeure évidente dans ce cas. En effet, $f_{gc}(S) = f_g(S)$ implique $\mathcal{GC}(S) = \mathcal{G}(S) = \max(S) + sp(f_{gc}(S))$ (Équations 2.14 et 2.10). Prouvons le théorème par induction sur la taille de S.

Pour $|S| = 1$, nous avons $f_{gc}(S) = f_g(S) = 1$ et $\mathcal{GC}(S) = \mathcal{G}(S) = s_1$. La conclusion du théorème est vérifiée dans ce cas.

Pour $|S| = 2$, nous avons $f_{gc}(S) = f_g(S)$ et $\mathcal{GC}(S) = \mathcal{G}(S) = s_2 + 2$. La conclusion du théorème est vérifiée dans ce cas aussi.

Pour $|S| > 2$, nous supposons que la propriété est vraie pour les séquences de taille au plus $|S| - 1$.
Soit S une séquence d'entiers $[s_1, \ldots, s_n]$. Nous traitons le codage de S par le *Polychotomique Généralisé*. Selon l'Équation 2.11, deux cas se présentent :

Cas.1 Il existe un entier $k > 1$ tel que $\mathcal{G}([s_1, \ldots, s_k]) \leq s_{k+1}$. Dans ce cas, le codage *Polychotomique Généralisé* substitue S par $[\mathcal{G}([s_1, \ldots, s_k]), s_{k+1}, \ldots, s_n]$, que l'on note S'. Alors, $\mathcal{G}(S) = \mathcal{G}(S')$ et $f_g(S) = f_g(S')$.
Selon le principe du codage *Polychotomique Généralisé*, détaillé dans la sous-section 3.2, la valeur du premier élément d'une séquence (son élément minimum) n'est jamais utilisée lors de son codage. Ainsi, $f_g(S')$ est égal à $f_g([0, s_{k+1}, \ldots, s_n])$.
Comme $\mathcal{G}([s_1, \ldots, s_k]) \leq s_{k+1}$, le codage *Partitionnement Contigu Généralisé* substitue S par $[0, s_{k+1}, \ldots, s_n]$ (Équation 2.13). Nous avons alors $f_{gc}(S) = f_{gc}([0, s_{k+1}, \ldots, s_n])$.
Puisque $k > 1$, nous obtenons, par hypothèse d'induction,

$$f_{gc}([0, s_{k+1}, \ldots, s_n]) \leq f_g([0, s_{k+1}, \ldots, s_n]).$$

Nous en déduisons que $f_{gc}(S) \leq f_g(S)$. Comme la fonction sp est croissante, $\max(S) + sp(f_{gc}(S)) \leq \max(S) + sp(f_g(S))$. Cela implique, d'après les Équations 2.10 et 2.14, $\mathcal{GC}(S) \leq \mathcal{G}(S)$. Nous admettons que la conclusion du théorème est valide dans ce premier cas.

Cas.2 Le codage *Polychotomique Généralisé* se comporte comme le codage *Polychotomique* (voir le deuxième cas de l'Équation 2.11). Dans ce cas, il est évident que $f_g(S) = f_d(S)$ et $\mathcal{G}(S) = \mathcal{P}(S)$.
Comme le premier cas de l'Équation 2.13 n'est pas satisfait, le codage de S par le *Partitionnement Contigu Généralisé* vérifie $f_{gc}(S) = f_c(S)$, ce qui implique $\mathcal{GC}(S) = \mathcal{C}(S)$.
Soit k l'entier tel que $\mathcal{C}([s_1, \ldots, s_k]) \leq s_n$ et $\mathcal{C}([s_1, \ldots, s_{k+1}]) > s_n$. Ainsi, $f_c(S) = 1 + f_c([s_{k+1}, \ldots, s_n])$. Pour $k < n - 1$, nous avons, par hypothèse d'induction, $f_c([s_1, \ldots, s_{k+1}]) \leq f_d([s_1, \ldots, s_{k+1}])$ implique

$$\max(S) + sp(f_c([s_1, \ldots, s_{k+1}])) \leq \max(S) + sp(f_d([s_1, \ldots, s_{k+1}]))$$

$$\text{alors, } \mathcal{C}([s_1, \ldots, s_{k+1}]) \leq \mathcal{P}([s_1, \ldots, s_{k+1}]) \leq \mathcal{D}([s_1, \ldots, s_{k+1}]),$$

selon les Équations 2.14 et 2.7. Rappelons que le codage *Polychotomique Généralisé* minimise la taille du codage *Dichotomique* ainsi que la fonction de poids $\mathcal{G}$ (resp. $\mathcal{GC}$) se comporte comme la fonction $\mathcal{P}$ (resp. $\mathcal{C}$) dans ce Cas.2. .
Comme $s_n < \mathcal{C}([s_1, \ldots, s_{k+1}])$, alors $s_n < \mathcal{D}([s_1, \ldots, s_{k+1}])$. Selon le Lemme 2.3.1, nous obtenons $1 + f_d([s_{k+1}, \ldots, s_n]) \leq f_d(S)$.
Par hypothèse d'induction, nous avons $f_c([s_{k+1}, \ldots, s_n]) \leq f_d([s_{k+1}, \ldots, s_n])$. Ainsi,

$1 + f_c([s_{k+1}, \ldots, s_n]) \leq 1 + f_d([s_{k+1}, \ldots, s_n])$ implique $f_c(S) \leq f_d(S)$, donc $\max(S) + sp(f_c(S)) \leq \max(S) + sp(f_d(S))$ implique $\mathcal{C}(S) \leq \mathcal{G}(S)$. Nous en déduisons que, $f_{gc}(S) \leq f_g(S)$ implique $\mathcal{GC}(S) \leq \mathcal{G}(S)$. Cela est vrai même pour $k = n - 1$. En effet, on a d'une part $f_c(S) = 2$ et $\mathcal{C}(S) = s_n + 2$. D'autre part, si $\mathcal{G}([s_1, \ldots, s_{n-1}]) \leq s_n$, alors, nous avons aussi $f_g(S) = 2$ et $\mathcal{G}(S) = s_n + 2$. Sinon, $f_g^{s_n}([s_1, \ldots, s_{n-1}]) \geq 2$ implique $f_g(S) \geq 3$, ce qui implique $\mathcal{G}(S) \geq s_n + 3$. La conclusion du théorème est alors valide dans ce second cas.

Les deux cas traités permettent de prouver le Théorème 2.3.20.

□

3.7 Résultats numériques

Nous venons de prouver théoriquement que l'heuristique proposée dans cette section, le *Partitionnement Contigu Généralisé*, améliore la taille du codage des arbres par vecteur de bits. Afin de mesurer sa performance d'un point de vue pratique, nous testons l'heuristique sur des hiérarchies connues. Nous décrivons le jeu de données considéré puis nous présentons les résultats des tests réalisés.

3.7.1 Données de test

Nous évaluons la performance de notre heuristique de codage des arbres par vecteur de bits sur les hiérarchies, de type arborescence, publiées sur internet par Krall[4]. Il s'agit des hiérarchies **Visualworks2** et **Digitalk3** du langage de programmation orienté objet *Smalltalk*, de la hiérarchie **NeXTStep** du langage *Objective-C* ainsi que **ET++** du langage *C++*. Des caractéristiques de ces hiérarchies sont données par la Table 2.5.

Hiérarchie	Taille	Hauteur	$\max(d_{out})$
Visualworks2	1957	15	181
Digitalk3	1357	14	141
NeXTStep	311	8	142
ET++	371	9	87

TABLE 2.5: Caractéristiques de quelques arborescences connues

4. http://www.complang.tuwien.ac.at/andi/typecheck/. Consulté en 2017

3.7.2 Expérimentations

Nous présentons dans la Table 2.6, la taille du codage des données de test par les heuristiques étudiées dans cette section, à savoir *Dichotomique*, *Polychotomique*, *Polychotomique Généralisé* et *Partitionnement Contigu Généralisé*. Chaque heuristique est repérée dans la table par sa fonction de poids.

Hiérarchie	$\mathcal{D}$	$\mathcal{P}$	$\mathcal{G}$	$\mathcal{GC}$
VisualWorks2	32	30	29	**27**
Digitalk3	29	28	28	**27**
NeXTStep	20	19	19	**17**
ET++	20	20	19	**18**

TABLE 2.6: Taille de codage des arbres par vecteur de bits calculé par différentes heuristiques

Les résultats montrent que chaque heuristique rapporte de légères améliorations au niveau de la taille du codage des hiérarchies. Il est possible que ces heuristiques calculent un codage des arbres par vecteur de bits de taille proche de l'optimum (la 2-dimension).

3.8 Synthèse

Les heuristiques de codage des arbres par vecteur de bits, étudiées dans ce chapitre, sont une 4-approximation de la 2-dimension des arbres. Ce facteur d'approximation est prouvé par Habib *et al.* pour l'algorithme *Dicho* (Théorème 1.3.12), mais reste valable pour toutes les heuristiques qui l'améliorent.

Nous avons montré que le calcul de la taille d'un codage des arbres par vecteur de bits peut s'effectuer progressivement à l'aide d'une fonction de poids, qui prend en entrée une séquence d'entiers à partir de laquelle elle génère une flat-séquence et retourne la taille du codage de la flat-séquence générée. L'heuristique que nous proposons fait appel à une telle fonction de poids en minimisant la taille de la flat-séquence produite. Elle permet ainsi de minimiser la taille du codage des hiérarchies considérées d'environ 6%.

Les prochaines investigations au sujet de la 2-dimension des arbres peuvent porter sur l'étude d'un codage efficace des flat-séquences. Nous favorisons la recherche d'heuristiques efficaces de codage des arbres par vecteur de bits dans l'amélioration du codage des ordres en général. Dans le cas présent, notre heuristique spécifique peut être incorporée dans celle que nous proposons pour le codage des ordres *via* la décomposition modulaire. Rappelons que cette dernière délègue le traitement d'une composition parallèle d'ordres à une heuristique de codage des arbres par vecteur de bits.

Chapitre 3

La 2-dimension des ordres partiels : Conjectures

> Your heart knows things that your mind can't explain.
>
> Unknown

Introduction

Le problème de calcul de la 2-dimension des ordres n'est pas facile. Il a été démontré qu'il s'agit d'un problème $\mathcal{NP}$-complet [Stahl et Wille, 1986] et non-approximable [Habib *et al.*, 2004]. Nous avons aussi identifié peu d'instances dont la valeur exacte de leur 2-dimension est facilement calculable. Pour le cas des arbres, il existe un algorithme d'approximation de leur 2-dimension mais l'algorithme polynomial exact nous échappe. Face à la difficulté intrinsèque du problème, plusieurs conjectures ont vu le jour, notamment celle de Habib *et al.* (cf. Conjecture 3.0.2) qui affirme que la complexité du problème est polynomiale pour la classe des arbres. Une autre conjecture plus spécifique, des mêmes auteurs, affirme que l'algorithme *Dicho* est une 2-approximation de la 2-dimension des arbres. Rappelons que le facteur d'approximation 4 est déjà démontré. Deux autres conjectures sont proposées dans le cas des arbres, une première suggère une borne inférieure sur leur 2-dimension et une seconde donne la 2-dimension exacte des arbres n-complets.

Conjecture 3.0.2 (Habib *et al.*, 2004)**.** *La 2-dimension des arbres est calculable en temps polynomial.*

Dans ce chapitre, nous étudions et apportons un regard critique sur ces conjectures. Nous introduisons immédiatement un outil fondamental dans notre étude, traitant la 2-dimension de la composition parallèle de deux ordres.

Proposition 3.0.21. *Soit P et Q deux ordres avec $dim_2(Q) \leq dim_2(P)$.*

(*A*) *Si $dim_2(Q) = dim_2(P)$, alors $dim_2(P + Q)$ est bornée inférieurement par $dim_2(P)$. Cette borne n'est pas stricte.*

(*B*) *Si $dim_2(P) - 1 \leq dim_2(Q)$ et P et Q possèdent chacun un élément minimum (dualement un élément maximum), alors $dim_2(P + Q) = dim_2(P) + 2$.*

(*C*) *Si $dim_2(Q) = dim_2(P) - 1$ et P ou Q possède un élément minimum (dualement un élément maximum), alors $dim_2(P + Q)$ est bornée inférieurement par $dim_2(P) + 1$. Cette borne n'est pas stricte.*

Démonstration. Pour (A), nous avons $P \subseteq P+Q$ implique $dim_2(P) \leq dim_2(P+Q)$ (par monotonie). L'exemple de la Figure 3.1 montre que la borne peut être atteinte.

Pour (B) et (C), nous notons n la 2-dimension de $P + Q$ ainsi que m_p (resp. m_q) l'élément minimum de P (resp. Q) s'il existe. Soit ϕ un plongement de $P + Q$ dans $\mathcal{B}_n$ et soit S la fonction qui prend en paramètre un ensemble d'éléments et retourne 1 s'il est singleton et 0 sinon. Montrons d'abord que

$$n \geq \max\{dim_2(P) + |\phi(m_p)| + S(\phi(m_q)), dim_2(Q) + |\phi(m_q)| + S(\phi(m_p))\}. \quad (3.1)$$

FIGURE 3.1: $dim_2(P+Q) = dim_2(P)$

Pour tout élément x de P, nous avons $\phi(m_p) \subseteq \phi(x)$ car $m_p \leq_P x$ et ϕ est un plongement de P dans $\mathcal{B}_n$. Donc, $|\bigcup_{x\in P} \phi(x) \backslash \phi(m_p)| \geq dim_2(P)$. Si $|\phi(m_q)| = 1$, alors $\phi(x) \cap \phi(m_q)$ est vide, car dans le cas contraire, nous avons $\phi(m_q) \subseteq \phi(x)$ implique $m_q \leq_{P+Q} x$. Ceci contredit la structure de $P+Q$: x appartient à P et m_q à Q. Comme $n = |\bigcup_{x\in P+Q} \phi(x)|$, nous concluons que

$$n \geq |\cup_{x\in P} \phi(x)| + S(\phi(m_q)) \geq dim_2(P) + |\phi(m_p)| + S(\phi(m_q)).$$

De la même façon, nous prouvons que $n \geq dim_2(Q) + |\phi(m_q)| + S(\phi(m_p))$. Ainsi, l'Inéquation 3.1 est démontrée.

Pour (B), il est facile de vérifier que $n \leq dim_2(P) + 2$. Il suffit de coder P et Q par $dim_2(P)$ couleurs et rajouter deux couleurs, une pour coder m_p et une autre pour coder m_q. Prouvons maintenant que $dim_2(P) + 2 \leq n$. D'après l'Inégalité 3.1, nous avons

- $|\phi(m_p)| = 1$ et $|\phi(m_q)| = 1$ implique $dim_2(P) + 2 \leq n$.
- $|\phi(m_p)| = 1$ et $|\phi(m_q)| \geq 2$ implique $dim_2(P) + 2 \leq dim_2(Q) + 3 \leq n$.
- $|\phi(m_p)| \geq 2$ implique $dim_2(P) + 2 \leq n$.

Nous en déduisons que $n = dim_2(P) + 2$.

Pour (C), en supposant que P possède un élément minimum, nous obtenons $1 \leq |\phi(m_p)|$ implique $dim_2(P) + 1 \leq n$ (selon l'Inégalité 3.1). Supposons maintenant que Q possède un élément minimum. Selon l'Inégalité 3.1, on a :

- $|\phi(m_q)| = 1$ implique $dim_2(P) + 1 \leq n$.
- $2 \leq |\phi(m_q)|$ implique $dim_2(P) + 1 \leq dim_2(Q) + 2 \leq n$

On en déduit que $dim_2(P) + 1 \leq n$. Il est possible d'avoir l'égalité (cf. Figure 3.2).

FIGURE 3.2: $dim_2(P+Q) = dim_2(P) + 1 = dim_2(Q) + 2$

□

1 2-dimension des arbres n-complets

Soit T un arbre n-complet de hauteur h. Nous montrons que le codage de T par l'heuristique de *Partitionnement Contigus Généralisé*, proposée dans le deuxième chapitre, génère un codage par vecteur de bits de T de taille $h.sp(n)$ [1]. Thierry conjecture que la 2-dimension de T vaut exactement $h.sp(n)$.

Proposition 3.1.22. *L'heuristique de* Partitionnement Contigus Généralisé *code un arbre n-complet de hauteur h par $h.sp(n)$ bits.*

Démonstration. Nous proposons une preuve par induction sur la hauteur de l'arbre.
Soit T un arbre n-complet de hauteur h.
Pour $h = 0$, l'arbre T est un singleton codé par l'ensemble vide. Ainsi, la taille de son codage est nulle. La proposition est alors vérifiée dans ce cas.
Pour $h > 0$, nous supposons que la propriété d'induction est vraie pour tout arbre de hauteur au plus $h - 1$.
Afin de coder T, l'heuristique de *Partitionnement Contigus Généralisé* traite en premier temps les sous-arbres de T dont chacun est enraciné en un fils de sa racine, qui sont tous n-complets de hauteur $h-1$. Comme la taille du codage de ces sous-arbres est $(h-1)sp(n)$, par hypothèse d'induction, l'heuristique attribue à la racine de chacun d'eux le poids $(h-1)sp(n)$. Elle calcule ensuite le poids de la racine de T, en fonction des poids de ses fils, *via* la fonction $\mathcal{GC}$ (Équation 2.13). Ce poids correspond à $\mathcal{GC}([\underbrace{(h-1)sp(n), \ldots, (h-1)sp(n)}_{n fois}])$ et vaut $(h-1)sp(n)+sp(n)$ (cas d'une flat-séquence). Par conséquent, la taille du codage de T est $h.sp(n)$.
La Proposition 3.1.22 est maintenant prouvée. □

La conjecture de Thierry s'énonce comme suit :

Conjecture 3.1.3 (Thierry, 2001)**.** *La 2-dimension des arbres n-complets de hauteur h est $h.sp(n)$.*

Nous proposons d'étudier la conjecture pour la classe des arbres 2-complets, 3-complets et 4-complets.

1.1 Les arbres 2-complets

La 2-dimension exacte des arbres binaires complets est donnée par Kahn [Kahn, 2015] et correspond à la 2-dimension des arbres 2-complets suggérée par la Conjecture 3.1.3. Nous proposons une preuve alternative de ce résultat, permettant de valider la conclusion de la Conjecture 3.1.3 pour le cas des arbres 2-complets.

1. $sp(n) = \min\{k | \binom{k}{\lfloor \frac{k}{2} \rfloor} \geq n\}$

Proposition 3.1.23 (Kahn, 2015)**.** *La 2-dimension des arbres binaires complets de hauteur h est $2h$.*

Démonstration. Nous proposons une preuve par induction sur la hauteur de l'arbre. Soit T un arbre binaire complet de hauteur h et de racine r.
Pour $h = 0$, l'arbre T est une chaîne à un seul élément dont la 2-dimension est nulle. Ainsi, la proposition est vérifiée dans ce cas.
Pour $h > 0$, nous supposons que la propriété d'induction est vraie pour tout arbre de hauteur au plus $h - 1$.
Notons a et b les successeurs immédiats de r ainsi que T_a et T_b les sous-arbres de T enracinés respectivement en a et b. Il est clair que $T\backslash\{r\}$ est l'arbre $T_a + T_b$ ainsi que $dim_2(T\backslash\{r\}) = dim_2(T)$. Comme T_a et T_b sont des arbres binaires complets de hauteur $h - 1$, nous avons $dim_2(T_a) = dim_2(T_b) = 2(h - 1)$ (par hypothèse d'induction). Alors, d'après la Proposition 3.0.21 (cas (B)), nous avons $dim_2(T\backslash\{r\}) = dim_2(T_a) + 2 = 2(h - 1) + 2 = 2h$. Nous en déduisons que $dim_2(T) = 2h$. La conclusion de la proposition est alors satisfaite dans ce cas.
Finalement, la Proposition 3.1.23 est prouvée. □

1.2 Les arbres 3-complets

Nous venons de montrer que tous les arbres 2-complets valident la conclusion de la Conjecture 3.1.3. Nous montrons ici que ce n'est pas le cas pour les arbres 3-complets, ce qui nous permet de conclure que la conjecture en question est fausse.

Proposition 3.1.24. *La 2-dimension des arbres 3-complets de hauteur h vaut exactement $h.sp(3)$, si $h \leq 2$, et vaut au plus $h.sp(3) - 1$ sinon.*

Démonstration. Soit T un arbre 3-complet de hauteur h. Nous traitons la preuve pour les différentes valeurs de h.
Pour $h = 0$, nous avons $dim_2(T) = 0$.

Pour $h = 1$, la 2-dimension de T est égale à la 2-dimension d'une antichaîne de taille 3. Ainsi, $dim_2(T) = sp(3)$ (Proposition 1.2.4). Rappelons que la 2-dimension d'un arbre ne change pas si l'on prive de sa racine.

Pour $h = 2$, nous avons déjà $dim_2(T) \leq 2sp(3)$ (Proposition 3.1.22). Montrons que $2sp(3) \leq dim_2(T)$ (i.e. $6 \leq dim_2(T)$).
Soit r la racine de T. Notons a, b et c les successeurs immédiats de r, ainsi que T_a, T_b et T_c, les sous-arbres de T enracinés respectivement en a, b et c. Comme ces sous-arbres sont de hauteur 1, leur 2-dimension vaut 3. Soit ϕ un plongement de T dans $\mathcal{B}_d$ avec d la 2-dimension de T. Nous avons quatre cas de figures :

1. Si $|\phi(a)| = |\phi(b)| = |\phi(c)| = 1$, alors $dim_2(T_a) + 3 \leq dim_2(T)$. En effet, s'il existe un élément x de T_a tel que $\phi(b) \subseteq \phi(x)$, alors $b \leq_T x$ par la définition du plongement ϕ. Ceci contredit la structure de T. Idem pour $\phi(c)$. Comme le code de tout élément de T_a contient $\phi(a)$ par héritage, le codage de T nécessite au moins les $dim_2(T_a)$ couleurs codant T_a auxquelles s'ajoute $\phi(a)$, $\phi(b)$ et $\phi(c)$.
2. Sans perte de généralité, si $|\phi(a)| = 2$ et $|\phi(b)| = 1$, alors l'ensemble codant T doit comprendre au moins les $dim_2(T_a)$ couleurs qui codent T_a auxquelles s'ajoute $\phi(a)$ (cause d'héritage) et $\phi(b)$ (même raison discutée dans le cas précédent). Donc, $dim_2(T_a) + 3 \leq dim_2(T)$.
3. Si $|\phi(a)| = |\phi(b)| = |\phi(c)| = 2$. Nous distinguons deux cas de figures :
   (i) Les ensembles $\phi(a)$, $\phi(b)$ et $\phi(c)$ sont disjoints, donc l'ensemble qui code T contient au moins les 6 couleurs codant les éléments a, b et c. Par conséquent, $6 \leq dim_2(T)$.
   (ii) Il existe au moins deux éléments, soit a et b sans perte de généralité, tels que $\phi(a) \cap \phi(b)$ est non vide. Soit s l'élément appartenant à $\phi(a) \cap \phi(b)$ et soit $\phi(b) = \{s, t\}$. S'il existe un élément x de T_a contenant t, alors $\phi(b) \subseteq \phi(x)$. En effet, comme $\phi(a) \subseteq \phi(x)$ (par héritage), l'élément s est déjà dans $\phi(x)$. Cela implique $b \leq_T x$, ce qui contredit la structure de T. Par conséquent, l'ensemble codant T_a ne doit pas contenir t. Nous en déduisons que le codage de T nécessite au moins les couleurs codant T_a auxquelles s'ajoute les éléments de $\phi(a)$ (par héritage) et la couleur t. Par conséquent, $dim_2(T_a) + 3 \leq dim_2(T)$.
4. Sans perte de généralité, si $|\phi(a)| \geq 3$, alors l'ensemble qui code T doit comprendre au moins les $dim_2(T_a)$ couleurs codant T_a auxquelles s'ajoute $\phi(a)$ à cause de l'héritage. Cela implique $dim_2(T_a) + 3 \leq dim_2(T)$.

Toutes les possibilités du codage de T par ϕ vérifient $6 \leq dim_2(T)$.
Nous en déduisons que $dim_2(T) = 2sp(3) = 6$.

Pour $h > 2$, nous considérons le codage présenté par la Figure 3.3 pour les trois premiers niveaux de T. Nous définissons le $i^{\text{ème}}$ niveau d'un arbre par l'ensemble de ses éléments reliés à la racine par i arcs dans le diagramme de Hasse de l'arbre, avec $i > 0$. Nous codons ensuite, les sous-arbres de T dont chacun est enraciné en un élément du troisième niveau, par notre heuristique de *Partitionnement Contigus Généralisé* en utilisant le même ensemble de couleurs. Rappelons que la taille de ce codage vaut $(h-3)sp(n)$ puisque les arbres codés sont 3-complets de hauteur $h-3$ (Proposition 3.1.22). Il est facile de vérifier que le codage de T, ainsi décrit, est un codage par vecteur de bits de taille $(h-3)sp(3) + 8 = h.sp(3) - 1$. Par conséquent, $dim_2(T) < h.sp(3)$.

La Proposition 3.1.24 est maintenant prouvée.

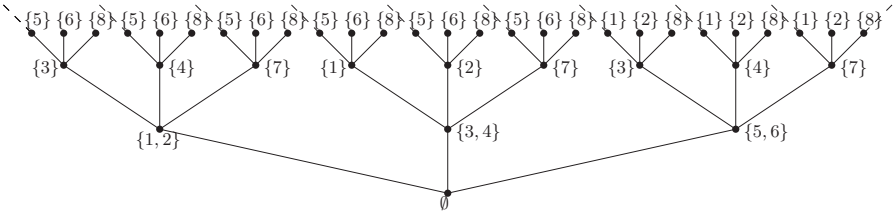


FIGURE 3.3: Codage réduit d'un arbre 3-complet

□

Nous déduisons de la Proposition 3.1.24, que la formule proposée par Thierry pour le calcul de la 2-dimension des arbres n-complets n'est pas valable pour des arbres 3-complets d'une hauteur quelconque. Il ne s'avère pas aussi évident d'exprimer la 2-dimension des arbres en fonction de leurs paramètres (hauteur, taille, etc.), ce qui peut mettre en cause la polynomialité du calcul de la 2-dimension des arbres.

Notons que la conclusion de la Conjecture 3.1.3 est trivialement vérifiée dans le cas des arbres de hauteur 1 (cela correspond au codage des antichaînes). Nous nous posons la question si cela est vrai dans le cas des arbres n-complets de hauteur 2. Pour $n \leq 3$, nous avons déjà une réponse positive. Nous traitons le cas de $n = 4$ juste après.

1.3 Les arbres 4-complets

Nous prouvons ici que les arbres 4-complets de hauteur au moins 2 infirment la Conjecture 3.1.3.

Proposition 3.1.25. *La 2-dimension des arbres 4-complets de hauteur $h > 1$ est au plus $h.sp(4) - 1$.*

Démonstration. Soit T un arbre 4-complet de hauteur h. Nous appelons le $i^{\text{ème}}$ niveau de T, et nous le notons l_i, pour i entre 1 et h, l'ensemble des éléments de T distant de i de la racine de T (dans le diagramme de Hasse de T). La distance entre deux éléments de T se mesure par la taille de la chaîne reliant ces deux éléments moins un (cette chaîne est unique dans le cas d'un arbre).
Soit ϕ'_2 la fonction qui associe à chaque élément de $l_1 \cup l_2$ un sous-ensemble de l'ensemble $\{1, \ldots, 7\}$, exactement comme décrit dans la Figure 3.4. Remarquons que ϕ'_2 est un codage réduit par vecteur de bits du sous-arbre de T défini sur $l_1 \cup l_2$. Soit ϕ'_i, pour i entre 3 et h, la fonction qui associe à chaque élément de l_i un singleton de l'ensemble $\{4(i-1), \ldots, 4(i-1)+3\}$, telle que deux éléments avec le même parent aient des singletons différents (Figure 3.4).

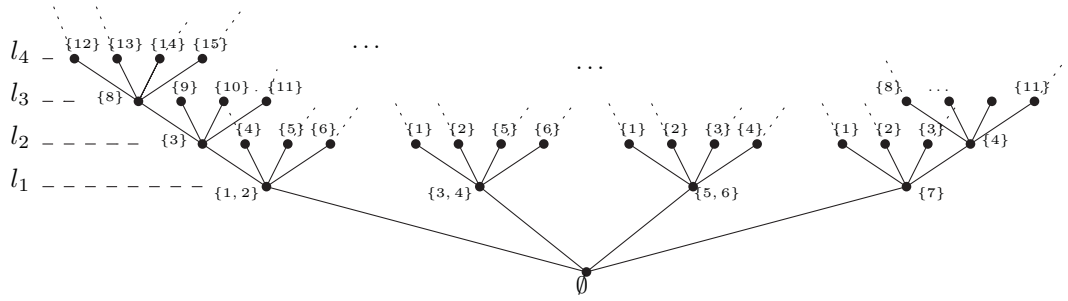


FIGURE 3.4: Codage réduit d'un arbre 4-complet

Nous considérons maintenant la fonction ϕ qui associe à chaque élément x de $l_i \subseteq T$ le code $\bigcup_{2\leq j\leq i}\bigcup_{y\leq_T x}\phi'_j(y)$. Notons que $|\bigcup_{x\in T}\phi(x)| = 4(h-2)+7 = 4h-1$. Nous prouvons que ϕ est un plongement de T dans $\mathcal{B}_{4h-1}$.
Soit x et y deux éléments de T tels que $x \leq_T y$. Alors x et y appartiennent respectivement aux niveaux i et j avec $i \leq j$. Par définition de ϕ, on a $\phi(x) \subseteq \phi(y)$. Réciproquement, nous supposons que $\phi(x) \subseteq \phi(y)$. Soit z le plus grand prédécesseur commun de x et y dans T. Nous avons trois cas :

1. Si $z = x$, alors $x \leq_T y$.
2. Si $z = y$, alors $y \leq_T x$ implique $\phi(y) \subseteq \phi(x)$ (d'après ce qui précède). Ceci contredit l'hypothèse $\phi(x) \subseteq \phi(y)$.
3. Sinon, notons z_1 et z_2 les deux éléments de T tels que $z \prec z_1 \leq x$ et $z \prec z_2 \leq y$. Comme z_1 et z_2 ont le même parent z, ils appartiennent tous les deux à un certain niveau l_i. Nous distinguons entre deux cas :
   (a) Pour $i > 1$, nous avons $\phi'_j(t)$ un singleton pour tout t de l_j avec $j \geq i$. De plus, l'élément x (resp. y) possède au plus un seul prédécesseur p dans $l_{j\geq i}$ avec $\phi'_j(p) \subseteq \phi(x)$ (resp. $\phi'_j(p) \subseteq \phi(y)$) et $\phi'_j(p)$ différent de $\phi'_i(z_2)$ (resp. $\phi'_i(z_1)$) (par la définition de la fonction ϕ). Nous concluons que $\phi(x)$ et $\phi(y)$ ne sont pas comparables par inclusion.
   (b) Pour $i = 1$, nous notons u (resp. v) le fils de z_1 (resp. z_2) qui est prédécesseur de x (resp. y). On a $\phi'_2(u)\cup\phi'_2(z_1)$ est incomparable à $\phi'_2(v)\cup\phi'_2(z_2)$ par la définition du plongement ϕ'_2. On a également, par la définition de la fonction ϕ, $\phi'_2(u) \cup \phi'_2(z_1) \subseteq \phi(x)$ et $\phi'_2(v) \cup \phi'_2(z_2) \subseteq \phi(y)$. Puisque x (resp. y) possède au plus un seul prédécesseur p dans $l_{j>2}$ avec $\phi'_j(p) \subseteq \phi(x)$ (resp. $\phi'_j(p) \subseteq \phi(y)$) un singleton n'appartenant pas à $\{1, \ldots, 7\}$, donc $\phi(x)$ et $\phi(y)$ sont incomparables par inclusion.

   Dans les deux cas, nous contredisons l'hypothèse $\phi(x) \subseteq \phi(y)$.

Nous concluons que $x \leq_T y$. Ainsi, la fonction ϕ est un plongement de T dans $\mathcal{B}_{4h-1}$ implique $dim_2(T) < 4h$. La Proposition 3.1.25 est maintenant prouvée.

□

Nous nous sommes demandés si la conclusion de la Conjecture 3.1.3 est vraie pour les arbres n-complets de hauteur 2. La réponse est non, d'après la Proposition 3.1.25. Nous laissons le calcul de la 2-dimension de ces arbres ouvert.

2 Borne inférieure sur la 2-dimension des arbres

Soit T un arbre de dimension d. Comme il est possible de définir un plongement de T dans $\mathcal{B}_d$, il est clair que la taille de T ne dépasse pas 2^d, la taille du treillis booléen $\mathcal{B}_d$. Notons $T_1, \ldots, T_k$ les sous-arbres de T dont chacun est enraciné en un successeur immédiat de sa racine. Notons d_i la 2-dimension de T_i, pour i entre 1 et k. Ainsi, nous avons $|T_i| \leq 2^{d_i}$. Nous avons également $|T| = \sum_{i=1}^{k} |T_i| + 1$. Nous en déduisons les deux inégalités suivantes :

$$|T_1| + \cdots + |T_k| \leq 2^{d_1} + \cdots + 2^{d_k}.$$

$$|T_1| + \cdots + |T_k| \leq 2^{d}.$$

La conjecture de Habib *et al.* met en relation les deux bornes supérieures sur $|T_1| + \cdots + |T_k|$. Elle s'énonce comme suit :

Conjecture 3.2.4 (Habib *et al.*, 2004)**.** *Pour T un arbre et $T_1, \ldots T_k$ ses sous-arbres dont chacun est enraciné en un successeur immédiat de sa racine, nous avons*

$$2^{dim_2(T_1)} + \cdots + 2^{dim_2(T_k)} \leq 2^{dim_2(T)}.$$

La Conjecture 3.2.4 permet de fournir une borne inférieure sur la 2-dimension de l'arbre T. Notons que si les sous-arbres $T_1, \ldots, T_k$ ont la même 2-dimension d_1, cette borne exprimerait qu'un codage optimal par vecteur de bits de leur composition parallèle (de $\sum_{i=1}^{k} T_i$) nécessite au moins $d_1 + \lceil \log_2(k) \rceil$ bits.

Dans cette section, nous validons d'abord la conclusion de la Conjecture 3.2.4 pour la classe des arbres 4-aires, puis nous la réfutons à l'aide d'un contre-exemple. Nous traitons à la fin le cas où les sous-arbres T_i, pour i de 1 à k, sont des chaînes de même taille.

2.1 Les arbres 4-aires

Nous montrons ici que les arbres 4-aires valident la conclusion de la Conjecture 3.2.4.

Proposition 3.2.26. *Pour T un arbre 4-aire et $T_1, \ldots, T_k$, avec $k \geq 1$, ses sous-arbres dont chacun est enraciné en un successeur immédiat de sa racine, nous avons*

$$2^{dim_2(T_1)} + \cdots + 2^{dim_2(T_k)} \leq 2^{dim_2(T)}.$$

Démonstration. Nous traitons toutes les valeurs possibles de k.
Pour $k = 1$, nous avons T une composition série de la racine de T, celle de T_1 ainsi que T_1 privé de sa racine. Selon la Proposition 2.1.8, on a $dim_2(T) = dim_2(T_1)+1$. Ainsi, $2^{dim_2(T_1)} \leq 2^{dim_2(T)}$.

Pour $1 < k \leq 4$, nous considérons ϕ un plongement de T dans le treillis booléen de dimension $dim_2(T)$. Nous notons d_i la 2-dimension de T_i, pour i entre 1 et k, et r_i sa racine. Sans perte de généralité, nous supposons que $d_1 \leq \cdots \leq d_k$.

Pour $k = 2$, nous avons $2^{d_1} + 2^{d_2} \leq 2^{d_2+1}$. Comme $d_2 + |\phi(r_2)| \leq dim_2(T)$ et $1 \leq |\phi(r_2)|$, nous avons $d_2 + 1 \leq dim_2(T)$ implique $2^{d_2+1} \leq 2^{dim_2(T)}$.
Nous en déduisons que $2^{d_1} + 2^{d_2} \leq 2^{dim_2(T)}$.

Pour $k = 3$, nous avons deux cas :

(i) $2^{d_1} + 2^{d_2} \leq 2^{d_3}$
D'après le raisonnement juste précédent, nous avons $2^{d_3+1} \leq 2^{dim_2(T)}$. Alors, $2^{d_1} + 2^{d_2} + 2^{d_3} \leq 2^{d_3} + 2^{d_3} \leq 2^{d_3+1} \leq 2^{dim_2(T)}$.

(ii) $2^{d_3} < 2^{d_1} + 2^{d_2}$
Comme $2^{d_1} + 2^{d_2} + 2^{d_3} \leq 3 \times 2^{d_3}$, montrons que $d_3 + 2 \leq dim_2(T)$.
On a $d_3 + |\phi(r_3)| \leq dim_2(T)$. Si $2 \leq |\phi(r_3)|$, alors $d_3 + 2 \leq dim_2(T)$. Sinon, nous avons $|\phi(r_3)| = 1$ implique $dim_2(T_1+T_2)+1 \leq dim_2(T)$. En effet, la couleur codant la racine de T_3 n'intervient pas dans le codage de $T_1 + T_2$. D'après ce qui précède, $2^{d_1} + 2^{d_2} \leq 2^{dim_2(T_1+T_2)}$. Par conséquent, $2^{d_3} < 2^{dim_2(T_1+T_2)}$ implique $d_3 + 1 \leq dim_2(T_1 + T_2)$. Nous déduisons que $d_3 + 2 \leq dim_2(T)$.
Comme $d_3 + 2 \leq dim_2(T)$, on a $2^{d_1} + 2^{d_2} + 2^{d_3} \leq 3 \times 2^{d_3} \leq 2^{d_3+2} \leq 2^{dim_2(T)}$.

Pour $k = 4$, nous traitons également deux cas :

(i) $2^{d_1} + 2^{d_2} + 2^{d_3} \leq 2^{d_4}$
Comme pour le cas précédent, $k = 3$, nous avons $2^{d_4+1} \leq 2^{dim_2(T)}$ implique $2^{d_1} + 2^{d_2} + 2^{d_3} + 2^{d_4} \leq 2^{d_4} + 2^{d_4} \leq 2^{d_4+1} \leq 2^{dim_2(T)}$.

(ii) $2^{d_4} < 2^{d_1} + 2^{d_2} + 2^{d_3}$
En prenant le sous-arbre $T\backslash T_1$, on a $2^{d_4+2} \leq 2^{dim_2(T\backslash T_1)}$ (voir (ii) pour $k = 3$). Comme $dim_2(T\backslash T_1) \leq dim_2(T)$ et $2^{d_1} + 2^{d_2} + 2^{d_3} + 2^{d_4} \leq 4 \times 2^{d_4}$, nous en déduisons $2^{d_1} + 2^{d_2} + 2^{d_3} + 2^{d_4} \leq 2^{d_4+2} \leq 2^{dim_2(T)}$.

Nous concluons enfin que la Proposition 3.2.26 est correcte. □

2.2 Les arbres 5-aires

Nous réfutons dans cette sous-section la Conjecture 3.2.4 à l'aide d'un exemple d'un arbre 5-aire. Soit T un arbre dont la racine possède 5 fils, et tel que ses sous-arbres, $T_1, \ldots, T_5$, dont chacun est enraciné en un fils de sa racine, sont des arbres binaires complets de hauteur 4. Selon la Proposition 3.1.23, la 2-dimension de chacun de ces sous-arbres vaut 8. La Conjecture 3.2.4 affirme que $5 \times 2^8 \leq 2^{dim_2(T)}$, i.e. $1280 \leq 2^{dim_2(T)}$. Nous proposons un codage de T de taille 10 (Figure 3.5) impliquant que $dim_2(T) \leq 10$ et par conséquent $2^{dim_2(T)} \leq 1024$. L'arbre T constitue alors un contre exemple de la Conjecture 3.2.4.

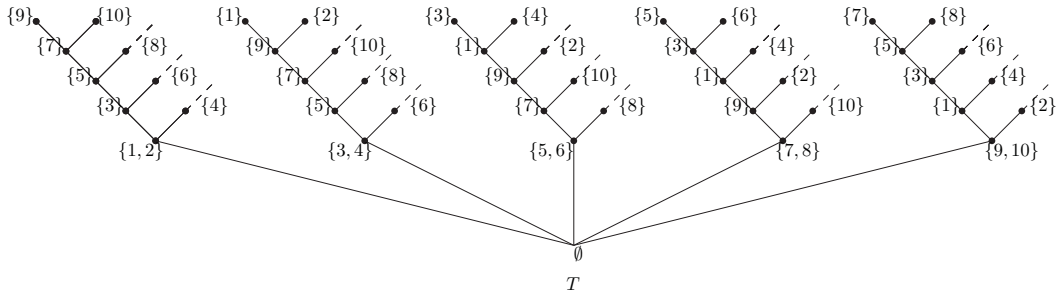


FIGURE 3.5: Codage réduit d'un arbre contredisant la Conjecture 3.2.4

Notons que la conjecture est par contre vérifiée si les sous-arbres binaires T_i de T, pour i entre 1 et 5, ont plutôt une hauteur de 3 (Figure 3.6). Nous admettons alors la difficulté de formuler même une borne inférieure intéressante sur la 2-dimension des arbres.

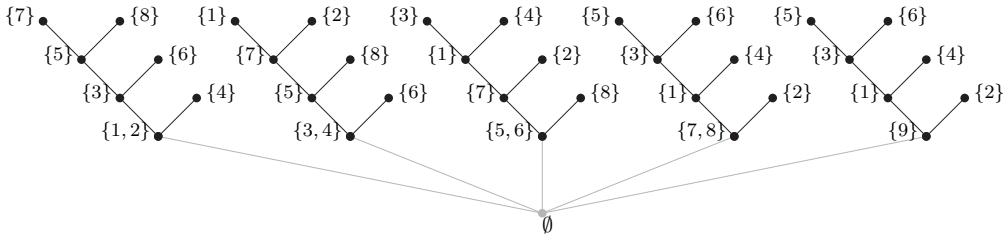


FIGURE 3.6: Codage réduit d'un arbre 5-aire validant la Conjecture 3.2.4

L'arbre T n'est pas l'unique contre exemple de la Conjecture 3.2.4. Nous proposons ici une classe d'arbres dont les instances, y compris T, réfutent la conjecture en question. Il s'agit des arbres composés d'une racine reliée à $k+1$ arbres binaires complets de hauteur k, avec $k > 3$. Rappelons que pour $k \leq 3$, la conclusion de la conjecture est vérifiée (Proposition 3.2.26).

Proposition 3.2.27. *Pour T un arbre dont la racine a $k+1$ fils, avec $k > 3$, et tel que ses sous-arbres $T_1, \ldots, T_{k+1}$, dont chacun est enraciné en un fils de la racine, sont des arbres 2-complets de hauteur k, nous avons $dim_2(T) \leq 2k+2$ implique $2^{dim_2(T)} < \sum_{i=1}^{k+1} 2^{dim_2(T_i)}$.*

Démonstration. Pour i de 1 à $k+1$, nous proposons d'attribuer à chaque paire d'éléments x et y de T_i, de hauteur h pour $1 < h \leq k+1$, les sous-ensembles $\{2(i+h-1)-1 \mod 2(k+1)\}$ et $\{2(i+h-1) \mod 2(k+1)\}$ respectivement, et à la racine de T_i, qui est de hauteur 1, le sous-ensemble $\{2i-1, 2i\}$, comme illustré par la Figure 3.5. En codant chaque élément de T par l'union du sous-ensemble qui lui est attribué et ceux attribués à ses prédécesseurs, il est facile de vérifier qu'il s'agit d'un codage par vecteur de bits de T. En d'autres termes, nous pouvons facilement vérifier que $x \leq_T y$ si et seulement si le code de x est inclus dans celui de y. La taille de ce codage est $2k+2$. Ainsi, $dim_2(T) \leq 2k+2$.

Nous avons $dim_2(T) \leq 2k+2$ implique $2^{dim_2(T)} \leq 4 \times 2^{2k}$. Comme $3 < k$, on a $4 < k+1$ implique $4 \times 2^{2k} < (k+1)2^{2k}$. Selon la Proposition 3.1.23, la 2-dimension des arbres T_i est $2k$. Ainsi, $2^{2k} = 2^{dim_2(T_i)}$, pour tout i entre 1 et $k+1$, implique $2^{dim_2(T)} < \sum_{i=1}^{k+1} 2^{dim_2(T_i)}$. La Proposition 3.2.27 est alors valide. □

2.3 Composition parallèles de chaînes

Nous validons ici la conclusion de la Conjecture 3.2.4 pour la classe des arbres définis par une composition série d'une racine et une composition parallèle de chaînes de taille identique. La Figure 3.7 illustre cette classe d'arbres.

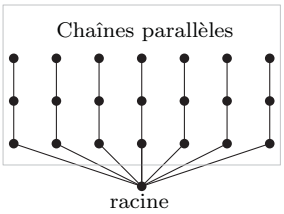


FIGURE 3.7: Composition série d'une racine et une composition parallèle de chaînes de taille identique

Nous proposons d'abord des bornes sur la 2-dimension de la composition parallèle de chaînes de même taille.

Proposition 3.2.28. *La 2-dimension de la composition parallèle de k chaînes de taille $n+1$ est entre $n + \lceil \log_2(k) \rceil$ et $n + sp(k)$.*

Démonstration. Pour i entre 1 et k, nous considérons C_i une chaîne de taille $n+1$ et B_i le treillis booléen $\mathcal{B}_n$. Prouvons d'abord que la composition parallèle des chaînes C_i ($\sum_{i=1}^{k} C_i$) a la même 2-dimension que l'ordre $\sum_{i=1}^{k} B_i$.
Notons m_i^c l'élément minimum de C_i et M_i^c son élément maximum. Idem, nous notons m_i^b l'élément minimum de B_i et M_i^b son élément maximum.
Comme C_i est un sous-ordre de B_i, on a $dim_2(\sum_{i=1}^{k} C_i) \leq dim_2(\sum_{i=1}^{k} B_i)$.

Prouvons maintenant que $dim_2(\sum_{i=1}^{k} B_i) \leq dim_2(\sum_{i=1}^{k} C_i)$.
Soit ϕ_c un plongement de $\sum_{i=1}^{k} C_i$ dans le treillis booléen de dimension $dim_2(\sum_{i=1}^{k} C_i)$.
Comme C_i est une chaîne à $n+1$ éléments, nous avons $dim_2(C_i) = n$ (Proposition 1.2.4) implique $n \leq |\phi_c(M_i^c) \backslash \phi_c(m_i^c)|$. Il est alors possible de coder B_i par $|\phi_c(M_i^c) \backslash \phi_c(m_i^c)|$ couleurs puisque $dim_2(B_i) = n$.
Soit ϕ_i^b une fonction qui utilise l'ensemble de couleurs $\phi_c(M_i^c) \backslash \phi_c(m_i^c)$ pour coder les éléments de B_i, en associant l'ensemble vide à m_i^b, l'ensemble $\phi_c(M_i^c) \backslash \phi_b(m_i^c)$ à M_i^b, et en vérifiant, pour toute paire d'éléments x et y de B_i avec $x \leq_{B_i} y$, $\phi_i^b(x) \subseteq \phi_i^b(y)$.
Soit ϕ' un plongement qui associe à chaque élément x de $\sum_{i=1}^{k} B_i$ appartenant au sous-ordre B_i, le sous-ensemble $\phi_c(m_i^c) \cup \phi_i^b(x)$. Prouvons que pour tout x et y de l'ordre $\sum_{i=1}^{k} B_i$, x est un prédécesseur de y si et seulement si $\phi'(x) \subseteq \phi'(y)$.

Soit x et y deux éléments de $\sum_{i=1}^{k} B_i$. Si x est un prédécesseur de y, alors x et y appartiennent à un certain sous-ordre B_i. Par définition de la fonction ϕ_i^b, nous avons $x \leq_{B_i} y$ implique $\phi_i^b(x) \subseteq \phi_i^b(y)$. Ceci implique $\phi'(x) \subseteq \phi'(y)$.
Réciproquement, supposons que $\phi'(x) \subseteq \phi'(y)$, tel que x appartient à B_i et y à B_j. D'après la définition de ϕ_j^b, nous avons $y \leq_{B_j} M_j^b$ implique $\phi_j^b(y) \subseteq \phi_j^b(M_j^b)$. Cela implique $\phi_c(m_j^c) \cup \phi_j^b(y) \subseteq \phi_c(m_j^c) \cup \phi_j^b(M_j^b)$. Donc, $\phi'(y) \subseteq \phi_c(M_j^c)$. Par définition du plongement ϕ', nous avons aussi $\phi_c(m_i^c) \subseteq \phi'(x)$. À partir de $\phi'(x) \subseteq \phi'(y)$, nous concluons que $\phi_c(m_i^c) \subseteq \phi_c(M_j^c)$. Comme ϕ_c est un codage par vecteur de bits de $\sum_{i=1}^{k} C_i$, $\phi_c(m_i^c) \subseteq \phi_c(M_j^c)$ implique m_i^c est un prédécesseur de M_j^c. Par conséquent, nous avons $i = j$ pour ne pas contredire la structure de l'ordre $\sum_{i=1}^{k} C_i$.
Nous en déduisons que x et y appartiennent au même sous-ordre B_i. Nous avons $\phi'(x) \subseteq \phi'(y)$ implique $\phi'(x) \backslash \phi_c(m_i^c) \subseteq \phi'(y) \backslash \phi_c(m_i^c)$. Cela implique $\phi_i^b(x) \subseteq \phi_i^b(y)$. Par définition de la fonction ϕ_i^b, nous avons $\phi_i^b(x) \subseteq \phi_i^b(y)$ implique $x \leq_{B_i} y$.
Nous concluons que ϕ' est un plongement de $\sum_{i=1}^{k} B_i$ dans le treillis booléen de dimension $|\bigcup \phi_c(M_i^c)|$. Puisque $|\bigcup \phi_c(M_i^c)| \leq dim_2(\sum_{i=1}^{k} C_i)$, nous obtenons $dim_2(\sum_{i=1}^{k} B_i) \leq dim_2(\sum_{i=1}^{k} C_i)$.

Par conséquent $dim_2(\sum_{i=1}^{k} \mathcal{B}_n) = dim_2(\sum_{i=1}^{k} C)$. Comme le treillis booléen $\mathcal{B}_n$ possèdent 2^n éléments, la taille de $\sum_{i=1}^{k} B_i$ vaut $k2^n$. Il faut alors un vecteur de bits (ou ensemble de couleurs) de taille au moins $\lceil \log_2(k2^n) \rceil$ pour coder l'ordre $\sum_{i=1}^{k} B_i$.

Nous concluons enfin que $n + \lceil \log_2(k) \rceil \leq dim_2(\sum_{i=1}^{k} B_i) = dim_2(\sum_{i=1}^{k} C_i)$. Cela démontre la borne inférieure.

Afin de démontrer la borne supérieure, nous proposons de coder l'arbre T obtenu à partir d'une composition série d'une racine r et l'ordre $\sum_{i=1}^{k} C_i$ par notre heuristique de codage des arbres : *Partitionnement Contigus Généralisé*. Rappelons que la taille de ce codage correspond au poids attribué à r, qui se calcule à partir des poids de ses fils. Notons que le poids de son $i^{\text{ème}}$ fils correspond à la taille du codage de la chaîne C_i par l'heuristique et il est facile de vérifier qu'elle vaut n. Ainsi, le poids de r est $\mathcal{GC}([\underbrace{n,\ldots,n}_{k fois}])$ qui vaut $n + sp(k)$ (cas d'une flat-séquence).
Comme la 2-dimension de T est égale à $dim_2(\sum_{i=1}^{k} C_i)$, $dim_2(\sum_{i=1}^{k} C_i) \leq n + sp(n)$. Cela démontre la borne supérieure.

La Proposition 3.2.28 est maintenant prouvée. □

Soit $C_1, \ldots, C_k$ des chaînes de taille $n+1$. Selon la Proposition 3.2.28, on a $n + \lceil \log_2(k) \rceil \leq dim_2(\sum_{i=1}^{k} C_i)$ implique $k2^n \leq 2^{dim_2(\sum_{i=1}^{k} C_i)}$. Comme la 2-dimension de C_i, pour i entre 1 et k, vaut n (Proposition 1.2.4), nous obtenons $\sum_{i=1}^{k} 2^{dim_2(C_i)} \leq 2^{dim_2(\sum_{i=1}^{k} C_i)}$. Par conséquent, nous validons la conclusion de la Conjecture 3.2.4 pour les arbres dont la racine est reliée à des chaînes de taille identique. Une généralisation de ce résultat est déjà prouvée par Thierry (cf. Proposition 3.2.29). Cependant, nous dérivons de notre preuve, que la composition parallèle d'ordres $Q_1, \ldots, Q_k$, de même hauteur $n+1$, vérifie la propriété $\sum_{i=1}^{k} 2^{dim_2(Q_i)} \leq 2^{dim_2(\sum_{i=1}^{k} Q_i)}$. Il suffit de remarquer que

$$2^{dim_2(\sum_{i=1}^{k} C_i)} \leq 2^{dim_2(\sum_{i=1}^{k} Q_i)} \leq 2^{dim_2(\sum_{i=1}^{k} B_i)}.$$

Proposition 3.2.29 (Thierry, 2001)**.** *La conclusion de la Conjecture 3.2.4 est vraie pour toute forêt composée de chaînes parallèles.*

Nous conjecturons que la 2-dimension d'une composition parallèle de chaînes de taille identique est calculable en temps polynomial, et il s'agit de la borne supérieure donnée par la Proposition 3.2.28. Si notre conjecture est vraie, nous pouvons alors calculer un codage de cet ordre par vecteur de bits de taille minimale à l'aide de notre heuristique de codage des arbres (voir la preuve de la Proposition 3.2.28).

Conjecture 3.2.5. *La 2-dimension d'une composition parallèle de $k > 1$ chaînes de même taille $n+1$ est $n + sp(k)$.*

3 2-approximation de la 2-dimension des arbres

Soit T un arbre de hauteur h et à l feuilles. Alors, T contient une chaîne de taille $h+1$ et une antichaîne de taille l. Comme la 2-dimension de ces deux sous-ordres de T est h et $sp(l)$ respectivement (Proposition 1.2.4), la 2-dimension de T est au plus $\max(h, sp(l))$.

Habib *et al.* prouvent que le codage de l'arbre T par l'algorithme *Dicho* est de taille au plus $2(h + \lceil \log_2(l) \rceil)$ [Habib *et al.*, 2004]. Sachant que $\lceil \log_2(l) \rceil \leq sp(l)$, nous avons $h + \lceil \log_2(l) \rceil \leq 2\max(h, sp(l))$. Nous en déduisons que l'algorithme *Dicho* calcule un codage de T de taille au plus $4dim_2(T)$.

Proposition 3.3.30 (Habib *et al.*, 2004)**.** *L'algorithme* Dicho *est une 4-approximation de la 2-dimension des arbres.*

Habib *et al.* conjecturent que l'algorithme *Dicho* est une 2-approximation de la 2-dimension des arbres. Leur conjecture se formule comme suit :

Conjecture 3.3.6 (Habib *et al.*, 2004)**.** *Pour tout arbre T, $Dicho(T) \leq 2dim_2(T)$.*

Dans cette section, nous donnons d'abord quelques classes d'arbres validant la conclusion de la Conjecture 3.3.6. Nous proposons ensuite quelques pistes pour étudier la conjecture en question que nous reformulons à la fin de la section.

3.1 Classes de validité

La conclusion de la Conjecture 3.3.6 est validée par Habib *et al.* pour la classe des arbres 4-aires [Habib *et al.*, 2004]. Nous validons ici la conclusion de cette conjecture pour deux autres classes d'arbres.

Proposition 3.3.31 (Habib *et al.*, 2004)**.** *Pour T un arbre 4-aire, nous avons $Dicho(T) \leq 2dim_2(T)$.*

Nous montrons que les chenilles et les arbres 2^k-complets valident la conclusion de la Conjecture 3.3.6.

Proposition 3.3.32. *Soit T un arbre de hauteur h.*

(i) Pour T une chenille, nous avons $Dicho(T) \leq 2dim_2(T)$.

(ii) Pour T un arbre 2^k-complet, nous avons $Dicho(T) \leq 2dim_2(T)$.

Démonstration. Nous proposons une preuve par induction sur la hauteur de l'arbre. Les notations et les propriétés utilisées dans cette preuve sont prédéfinies dans le chapitre précédent 3.2.

Pour $h = 0$, nous avons $Dicho(T) = dim_2(T) = 0$. La conclusion de la proposition est satisfaite dans ce cas.

Pour $h > 0$, nous supposons que les deux propriétés (i) et (ii) sont vraies pour tout arbre de hauteur au plus $h - 1$.
Notons r la racine de T et $r_1, \ldots, r_n$ les fils de r, n étant le degré sortant de r. Notons T_i le sous-arbre de T enraciné en r_i, pour i entre 1 et n. Nous traitons séparément les deux cas de la proposition :

(i) L'arbre T est une chenille

Nous posons T_n le sous-arbre non-trivial de T (différent du singleton). Nous supposons que $n > 0$, car pour $n = 0$, nous revenons au cas de base $h = 0$.
Pour $n = 1$, nous avons $Dicho(T) = Dicho(T_n) + 1$.
Pour $n = 2$, nous avons $Dicho(T) = Dicho(T_n) + 2$.
Pour $n > 2$, nous avons $Dicho(T) = \mathcal{D}([\underbrace{0, \ldots, 0}_{n-1}, Dicho(T_n)])$.

Rappelons que $Dicho(T)$ est la taille du codage de T par l'algorithme *Dicho*, qui est aussi le poids attribué à la racine de T, calculé par la fonction de poids $\mathcal{D}$. Étant donnée une séquence d'entiers $[s_1, \ldots, s_n]$, le calcul de $\mathcal{D}([s_1, \ldots, s_n])$ consiste à substituer à chaque fois les deux plus petits éléments de la séquence par leur maximum plus deux (Équation 2.1). Nous proposons une autre fonction alternative, qui ne substitue le dernier élément s_n qu'à la dernière itération (Figure 3.9). Elle permet ainsi d'attribuer le poids $\max(Dicho(T_n) + 2, \mathcal{D}([\underbrace{0, \ldots, 0}_{n-1}]) + 2)$ à l'élément r. Notons que $\mathcal{D}([\underbrace{0, \ldots, 0}_{n-1}]) = 2\lceil \log_2(n-1) \rceil$ (Proposition 2.3.17).

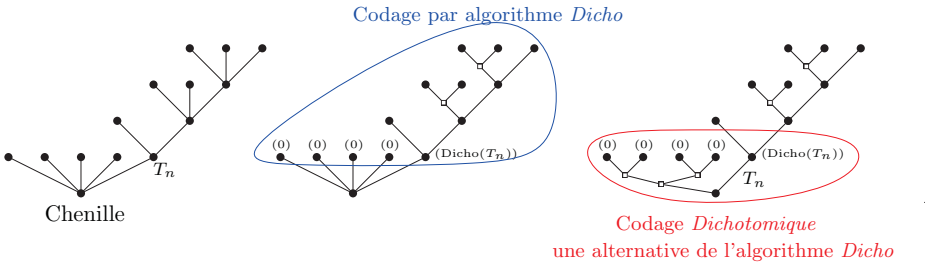


FIGURE 3.8: Codage *Dichotomique* d'une chenille

Selon la Proposition 2.3.16 et d'après ce qui précède, nous déduisons que

$$Dicho(T) \leq \max(Dicho(T_n) + 2, 2\lceil \log_2(n-1) \rceil + 2) \quad \text{pour } n > 2$$
$$Dicho(T) \leq Dicho(T_n) + 2 \quad \text{pour } n \leq 2 \quad (3.2)$$

Nous bornons maintenant la 2-dimension de T.

Notons T' le sous-arbre de T, défini par une composition série de r, r_n et T_n privé de sa racine. Ainsi, $dim_2(T') = dim_2(T_n) + 1$ (Proposition 2.1.8). Par conséquent, $dim_2(T_n) + 1 \leq dim_2(T)$.

On a également $dim_2(T \backslash T_n) \leq dim_2(T)$. Comme $T \backslash T_n$ est un arbre de hauteur 1 et à $n-1$ feuilles, $dim_2(T \backslash T_n) = sp(n-1)$. Ainsi, $sp(n-1) \leq dim_2(T)$. On prouve juste après (Proposition 3.3.33) que $\lceil \log_2(n-1) \rceil + 1 \leq sp(n-1)$, pour $n - 1 > 1$. Par conséquent, nous avons $\lceil \log_2(n-1) \rceil + 1 \leq dim_2(T)$, pour $n > 2$. Nous en déduisons

$$\begin{aligned} \max(dim_2(T_n) + 1, \lceil \log_2(n-1) \rceil + 1) \leq dim_2(T) \quad & \text{pour } n > 2 \\ dim_2(T_n) + 1 \leq dim_2(T) \quad & \text{pour } n \leq 2 \end{aligned} \tag{3.3}$$

Par hypothèse d'induction, nous avons $Dicho(T_n) \leq 2dim_2(T_n)$. Depuis les Inégalités 3.2 et 3.3, nous obtenons $Dicho(T) \leq 2dim_2(T)$ pour tout n.

(ii) L'arbre T est 2^k-complet

Prouvons d'abord, par induction sur h, que $Dicho(T) = 2hk$.

Pour $h = 0$, nous avons $Dicho(T) = 0$. La propriété est ainsi vérifiée pour le cas de base.

Pour $h > 0$, nous supposons que la propriété est vraie pour les arbres de hauteur au plus $h - 1$.

Nous avons T un arbre de hauteur h. La taille du codage de T par l'algorithme $Dicho$ vaut $\mathcal{D}([Dicho(T_1), \ldots, Dihco(T_n)])$ (Figure 3.9). Ainsi, $Dicho(T)$ vaut $Dicho(T_n) + 2\lceil \log_2(n) \rceil$ (Proposition 3.3.33), avec $n = 2^k$. Comme les sous-arbres T_i sont 2^k-complet de hauteur $h - 1$, nous avons, par hypothèse d'induction, $Dicho(T_i) = 2(h-1)k$, pour i entre 1 et n. Nous en déduisons que $Dicho(T) = 2(h-1)k + 2k = 2hk$. La propriété d'induction est ainsi démontrée.

Nous discutons maintenant la 2-dimension de T.

Comme T est un arbre 2^k-complet de hauteur h, il possède 2^{hk} feuilles qui forment une antichaîne de T. Ainsi, $sp(2^{hk}) \leq dim_2(T)$.

Nous avons $\lceil \log_2(2^{hk}) \rceil \leq sp(2^{hk})$ implique $hk \leq dim_2(T)$.

Comme $Dicho(T) = 2hk$, $Dicho(T) \leq 2dim_2(T)$.

Les deux cas récursifs que nous venons de prouver permettent de valider la Proposition 3.3.32.

□

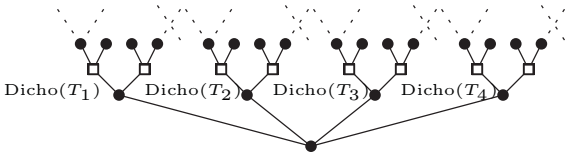


FIGURE 3.9: Codage *Dichotomique* d'un arbre 2^2-complet

Dans la preuve de la Proposition 3.3.32, nous avons utilisé cette propriété $\lceil \log_2(n) \rceil + 1 \leq sp(n)$, pour n un entier plus grand que 2. Nous la justifions ici.

Proposition 3.3.33. *Pour tout entier $n \geq 2$, nous avons $\lceil \log_2(n) \rceil + 1 \leq sp(n)$.*

Démonstration. Posons $k = sp(n)$. Par définition, nous avons $n \leq \binom{k}{\lfloor \frac{k}{2} \rfloor}$. Comme $n \geq 2$, la valeur de k vaut aussi au plus 2.
Nous avons $\binom{k}{\lfloor \frac{k}{2} \rfloor - 1} + \binom{k}{\lfloor \frac{k}{2} \rfloor + 1} = (\frac{\lfloor \frac{k}{2} \rfloor}{\lceil \frac{k}{2} \rceil + 1} + \frac{\lceil \frac{k}{2} \rceil}{\lfloor \frac{k}{2} \rfloor + 1})\binom{k}{\lfloor \frac{k}{2} \rfloor}$.
Comme $2 \leq k$, nous avons $1 \leq (\frac{\lfloor \frac{k}{2} \rfloor}{\lceil \frac{k}{2} \rceil + 1} + \frac{\lceil \frac{k}{2} \rceil}{\lfloor \frac{k}{2} \rfloor + 1})$.
Cela implique $2\binom{k}{\lfloor \frac{k}{2} \rfloor} \leq \binom{k}{\lfloor \frac{k}{2} \rfloor - 1} + \binom{k}{\lfloor \frac{k}{2} \rfloor + 1} + \binom{k}{\lfloor \frac{k}{2} \rfloor}$.
Selon la formule du binôme de Newton, on a $\binom{k}{\lfloor \frac{k}{2} \rfloor - 1} + \binom{k}{\lfloor \frac{k}{2} \rfloor + 1} + \binom{k}{\lfloor \frac{k}{2} \rfloor} \leq 2^k$.
Nous concluons que $2n \leq 2^k$ ce qui implique $\lceil \log_2(2n) \rceil \leq \lceil \log_2(2^k) \rceil$. Ainsi, $\lceil \log_2(n) \rceil + 1 \leq k = sp(n)$. La Proposition 3.3.33 est maintenant démontrée. □

3.2 Quelques pistes de preuve

Habib *et al.* [Habib *et al.*, 2004] montrent que la Conjecture 3.2.4 implique que l'algorithme *Dicho* est une 2-approximation de la 2-dimension des arbres, ce qui permet de confirmer la Conjecture 3.3.6.

Dans la section juste précédente, nous avons infirmé la Conjecture 3.2.4. Nous proposons ici quelques pistes pour étudier la Conjecture 3.3.6. Nous nous intéressons en particulier aux raisonnements par induction puis par monotonie. Nous discutons également des pistes déjà abordées.

3.2.1 Raisonnement par induction

Soit T un arbre. Afin de prouver par induction que l'algorithme *Dicho* calcule un codage de T de taille au plus $2dim_2(T)$ (Conjecture 3.3.6), nous devons d'abord le vérifier pour le cas de base : T est un singleton. Dans ce cas, on a bien $Dicho(T)$ est au plus $2dim_2(T)$, puisque $Dicho(T)$ et $dim_2(T)$ sont nuls.

Nous considérons maintenant que la propriété d'induction est vraie pour tout sous-arbre strict T' de T. Ainsi, nous avons $Dicho(T') \leq 2dim_2(T')$, par hypothèse d'induction. De plus, nous avons $dim_2(T') \leq dim_2(T)$, d'après la monotonie de la 2-dimension. Alors, si nous avons $Dicho(T) \leq Dicho(T')$, nous pouvons immédiatement déduire que $Dicho(T) \leq 2dim_2(T)$.

Pour T quelconque, nous ne pouvons pas admettre qu'il existe toujours un sous-arbre T' de T vérifiant la condition $Dicho(T) \leq Dicho(T')$. La Figure 3.10 donne un exemple d'un arbre T tel qu'aucun de ses sous-arbres ne vérifie cette condition.

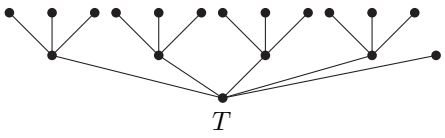


FIGURE 3.10: Pour tout $T' \subset T$, $Dicho(T') < Dicho(T)$

Nous prouvons maintenant que pour tout sous-arbre T' de T, nous avons $Dicho(T') \leq Dicho(T)$.

Proposition 3.3.34. *Pour T un arbre et T' un sous-arbre de T, nous avons $Dicho(T') \leq Dicho(T)$.*

Démonstration. Nous proposons une preuve par induction sur la taille de T.
Pour T un arbre réduit à un unique élément, nous avons $Dicho(T)$ et $Dicho(T')$ sont nuls. La propriété d'induction est satisfaite dans ce cas.
Nous supposons qu'elle est satisfaite pour tout sous-arbre strict T' de T.
Notons $T_1, \ldots, T_n$ les sous-arbres de T dont chacun est enraciné en un fils de sa racine. Idem, nous notons $T'_1, \ldots, T'_m$ les sous-arbres de T' dont chacun est enraciné en un fils de sa racine. Comme $T' \subset T$, nous supposons, sans perte de généralité, les deux cas suivant : $T' \subseteq T_n$ ou $T'_i \subseteq T_i$, pour i entre 1 et m.
Nous avons $Dicho(T) = \mathcal{D}([Dicho(T_1), \ldots, Dicho(T_n)]) \geq Dicho(T_n)$.
$\hookrightarrow$ Pour le premier cas, nous avons $Dicho(T') \leq Dicho(T_n)$ (par hypothèse d'induction) implique $Dicho(T') \leq Dicho(T)$.
$\hookrightarrow$ Pour le second cas, nous avons $Dicho(T') = \mathcal{D}([Dicho(T'_1), \ldots, Dicho(T'_m)])$.
Par hypothèse d'induction, nous avons $Dicho(T_i{}') \leq Dicho(T_i)$, pour i entre 1 et m, implique $[Dicho(T'_1), \ldots, Dicho(T'_m)] \leq_{lexico} [Dicho(T_1), \ldots, Dicho(T_m)]$.
Selon la Proposition 3.3.35, que nous prouverons juste après, nous avons

$$\mathcal{D}([Dicho(T'_1), \ldots, Dicho(T'_m)]) \leq \mathcal{D}([Dicho(T_1), \ldots, Dicho(T_n)]).$$

Par conséquent, $Dicho(T') \leq Dicho(T)$, ce qui vérifie la propriété d'induction dans ce cas. La Proposition 3.3.34 est ainsi prouvée. □

Nous prouvons maintenant le résultat énoncé par la Proposition 3.3.35 qui nous a permis de prouver que la taille du codage d'un arbre, par l'algorithme *Dicho*, est plus grande que la taille du codage de ses sous-arbres, par le même algorithme (Proposition 3.3.34).

Proposition 3.3.35. *Pour S et Q deux séquences d'entiers telles qu'il existe S' de S de même taille que Q avec $Q \leq_{lexico} S'$, nous avons $\mathcal{D}(Q) \leq \mathcal{D}(S)$.*

Démonstration. Nous proposons une preuve par induction sur la taille de S.
Pour $|S| = 0$, nous avons $\mathcal{D}(Q)$ et $\mathcal{D}(S)$ sont nuls. Ainsi, la propriété d'induction est vérifiée dans ce cas.
Pour $|S| > 0$, nous supposons que la propriété est vraie pour toute séquence d'au plus $|S| - 1$ entiers.
Posons $S = [s_1, \ldots, s_n]$, $S' = [s'_1, \ldots, s'_m]$ et $Q = [q_1, \ldots, q_m]$, avec $m \leq n$. Nous distinguons deux cas :

1. $s'_2 = s_2$
   Dans ce cas, nous avons bien $[s'_3, \ldots, s'_2 + 2, \ldots, s'_m]$ une sous-séquence de $[s_3, \ldots, s_2 + 2, \ldots, s_n]$. À partir de $Q \leq_{lexico} S'$, nous pouvons vérifier que $[q_3, \ldots, q_2 + 2, \ldots, q_m] \leq_{lexico} [s'_1, \ldots, s'_2 + 2, \ldots, s'_m]$ (voir la preuve de la Proposition 2.3.18). Par hypothèse d'induction, nous pouvons déduire que $\mathcal{D}([q_3, \ldots, q_2 + 2, \ldots, q_m]) \leq \mathcal{D}([s_3, \ldots, s_2 + 2, \ldots, s_n])$. Comme nous avons $\mathcal{D}(Q) = \mathcal{D}([q_3, \ldots, q_2 + 2, \ldots, q_m]$ et $\mathcal{D}(S) = \mathcal{D}([s_3, \ldots, s_2 + 2, \ldots, s_n])$, nous concluons que $\mathcal{D}(Q) \leq \mathcal{D}(S)$.
2. $s'_2 = s_i$, avec $i > 2$
   Dans ce cas, il est facile de trouver S'' de $[s_3, \ldots, s_2 + 2, \ldots, s_n]$, de taille m, telle que $S' \leq_{lexico} S''$ (S'' peut être la sous-séquence $[s'_2, \ldots, s_2 + 2, \ldots, s'_m]$). Ce qui implique $Q \leq_{lexico} S''$. Par hypothèse d'induction, nous obtenons $\mathcal{D}(Q) \leq \mathcal{D}([s_3, .., s_2 + 2, .., s_n])$. Comme $\mathcal{D}(S) = \mathcal{D}([s_3, \ldots, s_2 + 2, \ldots, s_n])$, nous en déduisons que $\mathcal{D}(Q) \leq \mathcal{D}(S)$.

La propriété d'induction est alors vérifiée dans les deux cas. La Proposition 3.3.35 est maintenant démontrée. □

Nous venons de montrer que pour tout sous-arbre T' de T, $Dicho(T') \leq Dicho(T)$. En supposant, par hypothèse d'induction, que $Dicho(T') \leq 2dim_2(T')$, nous cherchons comment en déduire que $Dicho(T)$ est au plus $2dim_2(T)$.

D'après la Définition 2.3.7, un codage *Dichotomique* de T consiste à le partitionner en deux forêts, susceptibles d'être codées par le même ensemble de couleurs, puis de répéter ce processus sur chacune d'elles. En considérant le codage par l'algorithme *Dicho*, il est certain qu'il subdivise T en deux forêts dont l'une admet un codage de taille $Dicho(T) - 2$ (Figure 3.11). Ainsi, si la 2-dimension de cette forêt est au plus $dim_2(T) - 1$, on conclut immédiatement que $Dicho(T) \leq 2dim_2(T)$.

Par abus de langage, nous désignons par un codage *Dichotomique* d'une forêt F, le codage de l'arbre obtenu par la composition série d'une racine et cette forêt. Nous notons $Dicho(F)$ la taille de ce codage.

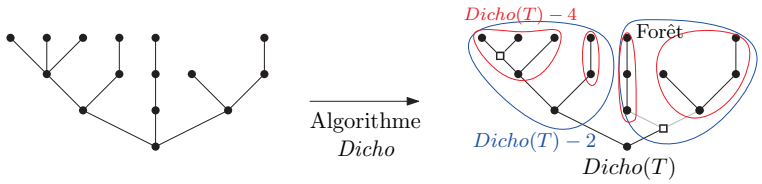


FIGURE 3.11: Partitionnement d'un arbre en forêts par l'algorithme *Dicho*

Thierry conjecture qu'il existe toujours un partitionnement de T en deux forêts dont chacune est de 2-dimension au plus $dim_2(T) - 1$ [Thierry, 2001]. Il montre aussi que cette conjecture implique $Dicho(T) \leq 2dim_2(T)$. Toutefois, il infirme sa conjecture à l'aide d'un exemple d'un arbre (cf. Figure 3.12) admettant un partitionnement par l'algorithme *Dicho* en deux forêts F_1 et F_2 avec $dim_2(F_1) = dim_2(T)$ et $dim_2(F_2) \leq dim_2(T) - 2$. La Proposition 3.3.36 affirme que ce partitionnement permet de valider la conclusion de la Conjecture 3.3.6 pour le cas récursif.

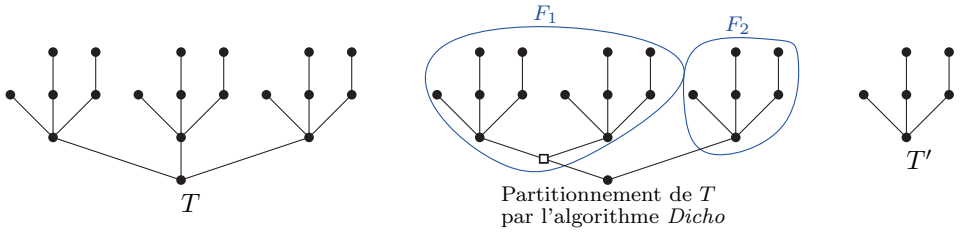


FIGURE 3.12: $dim_2(T) = dim_2(T' + T') = dim_2(T') + 2$

Proposition 3.3.36. *Soit T un arbre admettant un partitionnement par l'algorithme* Dicho *en deux forêts F_1 et F_2 de tel sorte que $dim_2(F_1) = dim_2(T)$ et $dim_2(F_2) < dim_2(T) - 1$. En supposant que la taille du codage de F_1 (resp. F_2), par l'algorithme* Dicho*, est au plus deux fois sa 2-dimension, nous pouvons dire que $Dicho(T) \leq 2dim_2(T)$.*

Démonstration. Comme l'arbre T est partitionné par l'algorithme *Dicho* en deux forêts F_1 et F_2, nous distinguons alors deux cas pour calculer $Dicho(T)$.

1. $Dicho(T) = Dicho(F_1) + 2$

   Supposons que F_1 correspond à un sous-arbre de T. Ainsi, la composition série de la racine de T, celle de F_1 et F_1 privé de sa racine est aussi un sous-arbre de T de 2-dimension $dim_2(F_1)+1$ (Proposition 2.1.8). Par conséquent, $dim_2(F_1) + 1 \leq dim_2(T)$. Comme $dim_2(F_1) = dim_2(T)$, notre hypothèse de départ est fausse. Alors F_1 est une forêt aussi partitionnée en deux sous-forêts F_{11} et F_{12}. Sans perte de généralité, nous supposons que $Dicho(F_{11})$ vaut $Dicho(F_1) - 2$. Ainsi, $Dicho(F_{11})$ vaut $Dicho(T) - 4$ ce qui implique $Dicho(T) - 4 \leq Dicho(F_2)$, car le codage est calculé par l'algorithme *Dicho* qui substitue les plus petits poids en premier ($Dicho(F_{11})$ et $Dicho(F_{12})$ déjà substitués par $Dicho(F_1)$).

   Nous avons $dim_2(F_2) \leq dim_2(T) - 2$ (selon l'énoncé de la proposition). Nous supposons que $Dicho(F_2) \leq 2dim_2(F_2)$. Comme $Dicho(T) - 4 \leq Dicho(F_2)$, nous en déduisons que $Dicho(T) \leq dim_2(T)$.

2. $Dicho(T) = Dicho(F_2) + 2$

   Supposons que $Dicho(F_2) \leq 2dim_2(F_2)$. Comme $dim_2(F_2) \leq dim_2(T) - 2$ et $Dicho(T) - 2 = Dicho(F_2)$, nous obtenons $Dicho(T) \leq 2dim_2(T) - 2$, ce qui implique $Dicho(T) \leq 2dim_2(T)$.

Les deux cas traités permettent de valider la Proposition 3.3.36.

□

Il est à noter que si le partitionnement de T est limité aux cas précédemment discutés, nous aurons ainsi tous les éléments pour rédiger une preuve par induction de la Conjecture 3.3.6. Il reste maintenant à vérifier s'il existe un partitionnement de T en deux forêts dont la 2-dimension de l'une est $dim_2(T)$ et de l'autre est au moins $dim_2(T) - 1$. En effet, l'arbre présenté dans la Figure 3.13 admet un tel partitionnement (cf. Figure 3.1 de la Proposition 3.0.21, cas (A)).

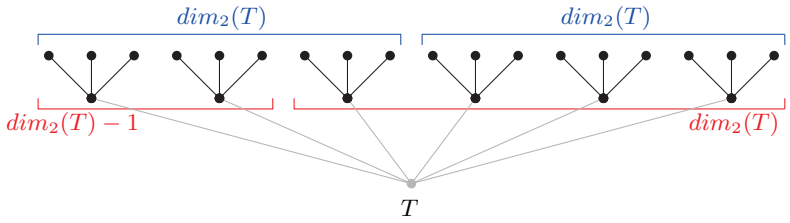


FIGURE 3.13: Différents partitionnements possibles d'un arbre

Dans la suite de notre étude du raisonnement par induction, nous nous sommes posés la question suivante :

Question : Est ce que tout arbre T contient un sous-arbre strict T' tel que l'écart entre la taille du codage de T et T' par l'algorithme *Dicho* est au plus deux fois l'écart entre leur 2-dimension ?

Notons que si un tel sous-arbre T' existe, et en supposant par hypothèse d'induction que $Dicho(T') \leq 2dim_2(T')$, nous pouvons immédiatement en déduire que $Dicho(T) \leq 2dim_2(T)$.

Pour répondre à cette question posée, nous avons examiné plusieurs cas particuliers de sous-arbre de T, mais n'avons pas eu de résultats probants. À présent, la question est ouverte et nous la laissons comme perspective.

3.2.2 Raisonnement par monotonie

Soit T un arbre non trivial (différent du singleton). Nous donnons ici quelques idées permettant de justifier que $Dicho(T) \leq 2dim_2(T)$ suivant un raisonnement par monotonie.

Soit h la hauteur d'un arbre binaire généré suite au codage de T par l'algorithme *Dicho*. En notant B_T l'arbre binaire complet de hauteur h, nous avons T se plonge dans B_T. Puisque T est non trivial, il admet alors une chaîne à 2 éléments comme sous-arbre. Ainsi nous avons $\vdots \subseteq T \subseteq B_T$. Comme nous l'avons démontré précédemment, la taille du codage des arbres par l'algorithme *Dicho* est monotone par sous-arbres : pour tout sous-arbre strict T' de T, nous avons $Dicho(T') \leq Dicho(T)$ (Proposition 3.3.34). Il en résulte de cette propriété de monotonie et aussi celle de la 2-dimension, les deux inéquations suivantes, illustrées par la Figure 3.14.

$$Dicho(\vdots) \leq Dicho(T) \leq Dicho(B_T),$$
$$2dim_2(\vdots) \leq 2dim_2(T) \leq 2dim_2(B_T). \quad (3.4)$$

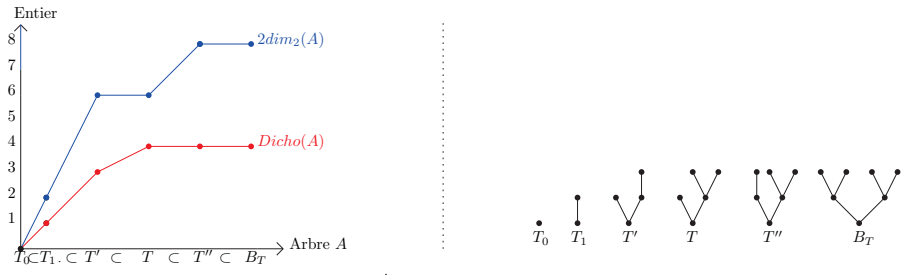


FIGURE 3.14: La 2-dimension vs la taille du codage *via Dicho*

Il est normal que la taille du codage d'un arbre binaire par l'algorithme *Dicho* corresponde au maximum degré d'une chaîne de cet arbre. Ainsi, nous avons $Dicho(\overset{\bullet}{\bullet}) = 1$ et $Dicho(B_T) = 2h$. Sachant que $dim_2(\overset{\bullet}{\bullet})$ vaut 1 (Proposition 3.1.23) et $dim_2(B_T)$ vaut $2h$ (Proposition 1.2.4), nous obtenons

$$Dicho(\overset{\bullet}{\bullet}) < 2dim_2(\overset{\bullet}{\bullet}) \text{ et } Dicho(B_T) < 2dim_2(B_T). \tag{3.5}$$

Nous nous proposons de voir dans quelle mesure l'arbre T admet un codage par l'algorithme *Dicho* de taille $2dim_2(T)$.

Dans leur étude de la même Conjecture 3.3.6, Habib *et al.* proposent une variante de l'algorithme *Dicho*, appelée *DichoEven*, permettant de calculer un codage par vecteur de bits de taille paire pour tout arbre [Habib *et al.*, 2004]. Elle se distingue de l'algorithme *Dicho* par le fait qu'elle attribue aux éléments ayant un fils unique, le poids de leurs fils plus deux (au lieu du poids de leurs fils plus un, comme le propose l'algorithme *Dicho*). Elle permet ainsi de calculer un codage de l'arbre T de taille paire, notée $DichoEven(T)$. La taille de ce codage vaut $Dicho(T)$ si $Dicho(T)$ est paire. Par conséquent, $Dicho(T) = 2dim_2(T)$ implique $\frac{DichoEven(T)}{2} = dim_2(T)$. Nous nous demandons dans quels cas $\frac{DichoEven(T)}{2}$ vaut la 2-dimension de T.

Conjecture 3.3.7. *Soit T un arbre. Nous avons $\frac{DichoEven(T)}{2} = dim_2(T)$ si et seulement si T est une chaîne.*

Nous montrons maintenant que la Conjecture 3.3.7 implique une preuve par monotonie de la Conjecture 3.3.6.

Proposition 3.3.37. *Si la Conjecture 3.3.7 est vraie, alors l'algorithme Dicho est une 2-approximation de la 2-dimension des arbres.*

Démonstration. Nous supposons que la Conjecture 3.3.7 est vraie. Étant donné un arbre T de hauteur h, montrons par monotonie que $Dicho(T)$ est au plus $2dim_2(T)$. Il est à noter que $Dicho(T) \leq DichoEven(T)$ et que la propriété de monotonie de *Dicho* s'étend naturellement au *DichoEven*.

Pour T une chaîne, on a $\frac{DichoEven(T)}{2} = dim_2(T)$ implique $Dicho(T) \leq 2dim_2(T)$.

Nous traitons maintenant le cas où l'arbre T est différent d'une chaîne.
Soit C une chaîne maximale de T. Remarquons que pour tout sous-arbre C' de C, nous avons $\frac{DichoEven(C')}{2} = dim_2(C')$ car C' est aussi une chaîne.

Soit C^2 le plus petit sous-arbre de T, différent d'une chaîne et contenant C, i.e. $C \subset C^2 \subseteq T$. L'arbre C^2 est alors une peigne de hauteur h et à 2 feuilles.

Nous avons $dim_2(C^2) = h + 1$, $DichoEven(C^2) \leq 2h + 2$ [Habib *et al.*, 2004] et $DichoEven(C^2) \neq dim_2(C^2)$ (car C^2 n'est pas une chaîne) implique

$$\frac{DichoEven(C^2)}{2} < dim_2(C^2). \tag{3.6}$$

Nous avons également $DichoEven(B_T) = 2h = Dicho(B_T)$ [Habib *et al.*, 2004], $dim_2(B_T) = 2h$ (Proposition 3.1.23) implique

$$\frac{DichoEven(B_T)}{2} < dim_2(B_T). \tag{3.7}$$

De plus, comme tout sous-arbre T' tel que $C^2 \subseteq T' \subseteq B_T$ est différent d'une chaîne, nous avons

$$\frac{DichoEven(T')}{2} \neq dim_2(T'). \tag{3.8}$$

Déduire de la propriété de monotonie des fonctions $DichoEven$ et dim_2 et des Inéquations 3.6, 3.7 et 3.8, que $\frac{DichoEven(T)}{2} < dim_2(T)$, n'est pas suffisant. Nous devons aussi vérifier si les courbes des deux fonctions se croisent en une structure qui n'est pas forcément un arbre. Pour cela, nous prenons T_1 et T_2 deux arbres tels que $|T_2| = |T_1| + 1$. Nous supposons que $\frac{DichoEven(T_1)}{2} < dim_2(T_1)$ et $dim_2(T_2) < \frac{DichoEven(T_2)}{2}$ et nous avons aussi $dim_2(T_1) \leq dim_2(T_2)$ car $T_1 \subseteq T_2$. Dans ce cas, nous obtenons $\frac{DichoEven(T_1)}{2} + \alpha = \frac{DichoEven(T_2)}{2}$ avec $\alpha \geq 2$. Ceci implique que l'écart entre $DichoEven(T_2)$ et $DichoEven(T_1)$ vaut au moins 4. Or, cela n'est pas possible car la taille du codage d'un arbre par l'algorithme $DichoEven$ peut augmenter d'au plus 2 bits si on augmente la taille de l'arbre d'un élément. Maintenant, nous pouvons conclure que $\frac{DichoEven(T)}{2} < dim_2(T)$, ce qui implique $Dicho(T) < 2dim_2(T)$ (cf. Figure 3.15 pour une illustration).

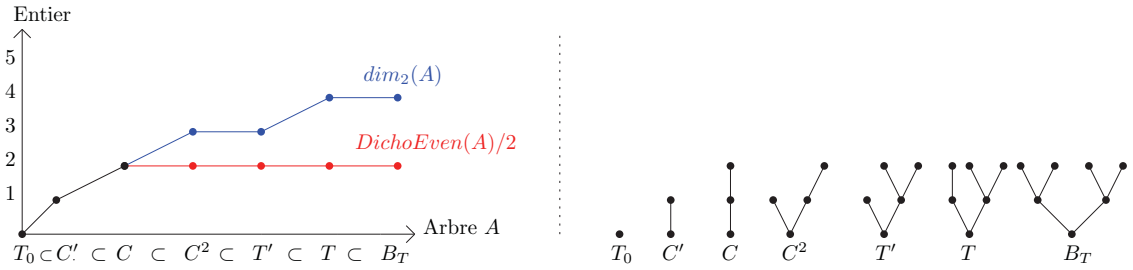


FIGURE 3.15: La 2-dimension vs la taille du codage *via* $DichoEven$

Nous avons ainsi rédigé une preuve par monotonie de la Conjecture 3.3.6 qui repose sur la validité de la Conjecture 3.3.7. Notre Proposition 3.3.37 est alors démontrée. □

3.3 Reformulation

Nous proposons ici une reformulation de la Conjecture 3.3.6 que nous avons étudié dans cette section. Nous expliquons l'intérêt de cette reformulation dans le dernier chapitre de ce manuscrit.

Soit T un arbre. Par Définition 2.3.7, un codage *Dichotomique* de T est clairement basé sur son plongement dans un arbre binaire. Soit B un arbre binaire généré par l'algorithme *Dicho* lors du codage de T, et dans lequel se plonge T. Rappelons que la taille de ce codage correspond au maximum degré d'une chaîne de B. Puisque l'algorithme *Dicho* calcule un codage *Dichotomique* de taille minimale [Habib *et al.*, 2004], il génère alors un plongement de T dans un arbre binaire ayant le plus petit maximum degré d'une chaîne.

Nous proposons ici une variante de l'algorithme *Dicho*, appelée *Unicho*, permettant de plonger T dans un arbre binaire de plus petite hauteur, notée $Unicho(T)$. Nous détaillons les instructions de cette variante dans l'Algorithme 10.

Données : Un arbre T
Résultat : Plongement de T dans un arbre binaire
$O \leftarrow$ ordre topologique inverse des éléments de T
`/* Initialisation des poids */`
pour *tout* x *dans* O **faire**
  $w(x) \leftarrow -1$
fin
pour *tout* x *dans* O *de poids négatif* **faire**
  si x *est une feuille* **alors**
    $w(x) \leftarrow 0$
  fin
  si x *possède un seul fils* u **alors**
    $w(x) \leftarrow w(u) + 1$
  fin
  si x *possède exactement deux fils* u *et* v **alors**
    $w(x) \leftarrow \max(w(u), w(v)) + 1$
  fin
  si x *possède au moins trois fils* **alors**
    Choisir les deux fils u et v de plus petits poids
    Ajouter un nouvel élément z à T comme fils de x et parent de u et v
    $w(z) \leftarrow \max(w(u), w(v)) + 1$
  fin
fin
`/* retourner` $Unicho(T)$`, la hauteur de l'arbre binaire */`
retourner $w(x)$

Algorithme 10 : Algorithme *Unicho*

Notons que les deux algorithmes *Unicho* et *Dicho* se distinguent au niveau des instructions en lignes 13 et 18. En effet, l'algorithme *Dicho* attribue le poids $\max(w(u),\ w(v)) + 2$ à chacun des éléments x (ligne 13) et z (ligne 18). Ainsi, l'Algorithme 10 s'exécute en temps polynomial.

Nous prouvons maintenant que, effectivement, l'algorithme *Unicho* plonge T dans un arbre binaire de hauteur minimale.

Théorème 3.3.21. *L'Algorithme 10 calcule un plongement d'un arbre T dans un arbre binaire de hauteur minimale.*

Démonstration. Nous proposons une preuve par induction sur la taille de T. Notons r la racine de T et $T_1, \ldots, T_n$ les sous-arbres de T dont chacun est enraciné en un fils de sa racine.

Pour T réduit à r, nous avons $Unicho(T) = 0$ et il est minimal.

Nous supposons que la propriété d'induction est satisfaite pour tout sous-arbre T_i, pour i entre 1 et n, avec $n > 0$. Afin de prouver la conclusion du théorème pour le cas récursif, il suffit de montrer que l'Algorithme 10 attribue le poids minimum à r. Pour montrer cela, nous définissons la fonction de poids $\mathcal{U}$ qui calcule le poids de r à partir des poids de ses fils ($Unicho(T_i)$, pour i entre 1 et n), passés à la fonction sous forme d'une séquence d'entiers S. Cette fonction retourne $Unicho(T)$. Notons $Unicho(T_i)$ par s_i, pour i entre 1 et n. Sans perte de généralité, nous considérons $s_1 \leq \cdots \leq s_n$.

$$\mathcal{U}(S) = \begin{cases} s_1 + 1 & \text{si } |S| = 1, \\ s_2 + 1 & \text{si } |S| = 2, \\ \mathcal{U}([s_3, \ldots, s_2 + 1, \ldots, s_k]) & \text{sinon} \end{cases} \tag{3.9}$$

Prouvons par induction sur la taille de S que $\mathcal{U}(S)$ est minimal. Nous vérifions d'abord les cas de bases.

Pour $|S| = 1$, nous avons r à un seul fils. La hauteur minimale d'un arbre binaire dans lequel se plonge T est $s_1 + 1$, et nous avons $\mathcal{U}(S) = s_1 + 1$.

Pour $|S| = 2$, nous avons r à deux fils. La hauteur minimale d'un arbre binaire dans lequel se plonge T est $s_2 + 1$, car $s_1 \leq s_2$, et nous avons $\mathcal{U}(S) = s_2 + 1$.

Pour $|S| = 3$, nous avons r à trois fils. Nous énumérons, dans la Figure 3.16, toutes les possibilités du plongement de T dans un arbre binaire. Selon la figure, la hauteur minimale d'un arbre binaire où se plonge l'arbre T est $\max(s_2 + 1, s_3) + 1$ qui correspond à $\mathcal{U}([s_1, s_2, s_3])$.

Pour $|S| > 3$, nous supposons que la propriété d'induction est vraie pour toute séquence de taille au plus $|S| - 1$.

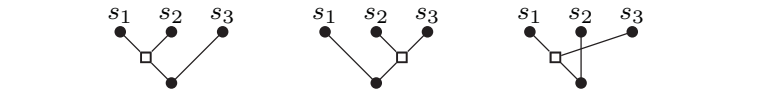


FIGURE 3.16: Plongements d'un arbre, dont la racine a trois fils, dans un arbre binaire

Dans ce cas, nous avons $\mathcal{U}(S) = \mathcal{U}([s_3, \ldots, s_2+1, \ldots, s_n])$. Par hypothèse d'induction, nous avons $\mathcal{U}([s_3, \ldots, s_2+1, \ldots, s_n])$ minimal ce qui implique la minimalité de $\mathcal{U}(S)$.

Par conséquent, l'Algorithme 10 attribue le poids minimum à r, qui correspond à la hauteur de l'arbre binaire dans lequel se plonge T. Nous concluons que l'algorithme *Unicho* plonge T dans un arbre binaire de hauteur minimale. Le Théorème 3.3.21 est ainsi prouvé.

□

Nous venons de prouver que l'algorithme *Unicho* plonge l'arbre T dans un arbre binaire de hauteur minimale. Nous conjecturons qu'il existe un plongement de T dans un arbre binaire de hauteur $dim_2(T)$. Notre conjecture s'énonce comme suit :

Conjecture 3.3.8. *Pour tout arbre T différent d'une chaîne, $Unicho(T) \leq dim_2(T)$, et plus précisément, $Unicho(T)+1 \leq dim_2(T)$.*

Nous montrons maintenant que notre Conjecture 3.3.8 implique la Conjecture 3.3.6 : la 2-approximation de l'algorithme *Dicho*.

Proposition 3.3.38. *Pour tout arbre T, si $Unicho(T)+1 \leq dim_2(T)$ alors $Dicho(T) \leq 2dim_2(T)$.*

Démonstration. Nous montrons d'abord que $Dicho(T) \leq 2Unicho(T)$.
Soit B_d et B_u des arbres binaires générés par les algorithmes *Dicho* et *Unicho* respectivement, à partir de la donnée de l'arbre T. En notant m_d (resp. m_u) le maximum degré d'une chaîne de B_d (resp. B_u), nous avons $m_d \leq m_u$. Nous rappelons que l'algorithme *Dicho* plonge T dans un arbre binaire de plus petit maximum degré d'une chaîne.
Notons h la hauteur de l'arbre binaire B_u. Ainsi, $m_u \leq 2h$ ce qui implique $m_d \leq 2h$. Comme $Dicho(T) = m_d$ et $Unicho(T) = h$, nous en déduisons que $Dicho(T) \leq 2Unicho(T)$.
En supposant que $Unicho(T) \leq Unicho(T)+1 \leq dim_2(T)$, nous concluons que $Dicho(T) \leq 2dim_2(T)$. □

La Conjecture 3.3.8 exprime également que tout arbre admettant un plongement dans un treillis booléen de dimension n, est plongeable dans un arbre binaire complet de hauteur n (et même $n-1$) (Figure 3.17).

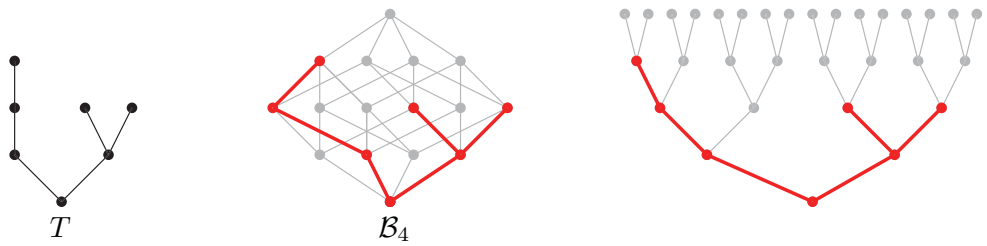


FIGURE 3.17: L'arbre T se plonge dans $\mathcal{B}_4$ et aussi dans un arbre binaire complet de hauteur 4 et même celui de hauteur 3

Dans le chapitre suivant, nous montrons qu'il existe un codage de T par vecteur de $Unicho(T)+1$ bits. Nous verrons comment récupérer la relation d'ordre entre les éléments de T dans un tel codage. Nous discutons aussi les éventuels avantages de ce codage par rapport au codage par vecteur de bits lié au calcul de la 2-dimension, qui code les éléments de T par vecteur de $dim_2(T)$ bits.

Chapitre 4

Autres codages par vecteur de bits

> You see things ; you say, 'Why ?'
> But I dream things that never
> were ; and I say 'Why not ?'
>
> George Bernard Shaw

Introduction

La notion classique de codage des ordres par vecteur de bits consiste à plonger un ordre dans un treillis booléen – $(\{0,1\}^n, \leq)$. Il est possible de généraliser cette notion en plongeant l'ordre dans l'ensemble $\{0,1\}^n$ muni d'une relation d'ordre autre que celle usuellement définie sur cet ensemble. Par exemple, on peut considérer qu'un vecteur U est inférieur à un vecteur V si U `XOR` $V = V$. En choisissant la relation d'ordre adéquate, on peut ainsi obtenir des compressions efficaces de l'ordre initial.

Le sujet de cette thèse est centré autour de la 2-dimension des ordres qui correspond à la taille minimale de leur codage par vecteur de bits. En généralisant la notion de ce type de codage, on se demande s'il existe un codage efficace des ordres par vecteur de bits de taille inférieure à la 2-dimension.

Dans ce chapitre, nous répondons positivement à cette question pour le cas des arbres binaires puis nous discutons le cas général des arbres.

1 Rappels

Nous rappelons ici les critères permettant d'évaluer l'efficacité d'un codage et aussi les caractéristiques d'un codage efficace.

Soit $P = (X, \leq)$ un ordre à n éléments. Nous évaluons l'efficacité d'un codage de P à partir (1) de sa taille, qui correspond au nombre de bits codant chaque élément de P ; (2) du temps nécessaire pour récupérer la relation d'ordre entre deux éléments de P ; (3) de la complexité (arithmétique) en temps et en espace du calcul de ce codage.

Étant donné un codage de P, nous dirons qu'il est efficace si sa taille est en $\mathcal{O}(\log_2(n))$, s'il vérifie la relation d'ordre entre deux éléments en temps linéaire par rapport à la taille du codage, et s'il code les éléments de P en temps et en espace polynomiaux par rapport à n.

2 Codages par vecteur de bits des arbres binaires complets

Cette section expose trois codages par vecteur de bits différents pour le cas des arbres binaires complets. Nous décrivons ces codages puis nous mesurons leurs critères d'évaluation. Nous en déduisons par la suite le codage par vecteur de bits le plus efficace pour les arbres binaires complets.

2.1 Codage *via* un plongement dans un treillis booléen

Soit $T = (X, \leq_T)$ un arbre binaire complet de hauteur h. Nous calculons un codage de T par vecteur de bits à travers le plongement de T dans un treillis booléen de dimension $2h$. Rappelons que la 2-dimension de T vaut exactement $2h$ (Proposition 3.1.23).

♦ Description du codage

Soit ψ une fonction de X vers $\{1, \ldots, 2h\}$ qui associe à chaque élément x de X, distant de i de la racine de T, la valeur $2i-1$ ou $2i$ de telle sorte que deux éléments avec le même parent n'aient pas la même valeur.

Nous définissons ensuite le codage ϕ qui associe à chaque élément x de T le vecteur de bits $V_x = [x_1, \ldots, x_{2h}]$ tel que x_i est à 1 s'il existe un élément a de T vérifiant $a \leq_T x$ et $\psi(a) = i$, et il est à 0 sinon, pour i de 1 à $2h$.

Montrons que ϕ est un plongement de T dans le treillis booléen $(\{0,1\}^{2h}, \leq)$. Pour cela, nous devons prouver que pour toute paire d'éléments x et y de X, $x \leq_T y$ si et seulement si $V_x \leq V_y$.

Soit x et y deux éléments de T.

1. Supposons que $x \leq_T y$. Pour i de 1 à $2h$, si $x_i = 1$ cela signifie qu'il existe un élément a de X tel que $a \leq_T x$ et $\psi(a) = i$. Comme $x \leq_T y$, nous avons $a \leq_T y$ implique $y_i = 1$. Sinon, nous avons $x_i = 0$ et y_i est dans $\{0,1\}$ implique $x_i \leq y_i$. Nous en déduisons que $V_x \leq V_y$.
2. Réciproquement, nous supposons que $V_x \leq V_y$. Montrons alors que $x \leq_T y$. En raisonnant par absurde, nous avons deux cas. Le premier cas $y \leq_T x$ implique, d'après (1) de cette preuve, $V_y \leq V_x$ ce qui contredit l'hypothèse de départ. Le deuxième cas correspond à x et y incomparables. Notons z le plus grand prédécesseur commun de x et y, et notons a et b ses successeurs immédiats qui sont respectivement des successeurs de x et y. Alors, nous avons $x_{\psi(a)} = 1$ et $y_{\psi(b)} = 1$. Nous avons également $\psi(a) \neq \psi(b)$ car a et b ont le même parent z. Soit v un élément de T tel que $\psi(v) = \psi(a)$. Les éléments a et v, et également b, sont alors à la même distance de la racine de T. Comme b est déjà un prédécesseur de y, l'élément v ne peut jamais l'être par conséquent. Ainsi, $y_{\psi(a)} = 0$. Nous en déduisons que V_x et V_y sont incomparables. Cela contredit également l'hypothèse de départ. Nous concluons que $x \leq_T y$.

Le plongement ϕ définit alors un codage par vecteur de bits de T, que nous illustrons par la Figure 4.1 (les valeurs en rouge sont calculées par la fonction ψ).

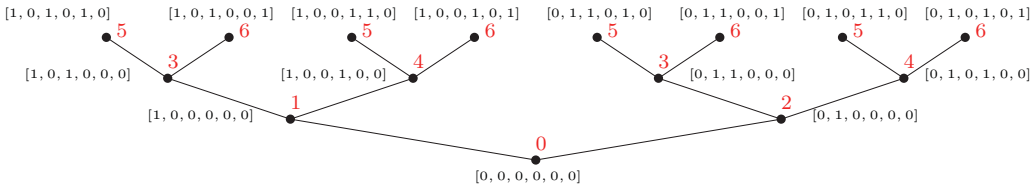


FIGURE 4.1: Codage d'un arbre binaire complet *via* un plongement dans un treillis booléen

♦ Évaluation du codage

Le codage que nous venons de définir est de taille $2h$ vu qu'il correspond au plongement de T dans le treillis booléen $(\{0,1\}^{2h}, \leq)$. De plus, tester si $x \leq_T y$ revient à tester si $V_x \leq V_y$. Ce test est effectué en temps $\mathcal{O}(h)$, qui est linéaire par rapport à la taille du codage. Finalement, le calcul du codage est de complexité en temps et en espace polynomiaux par rapport à la taille de l'arbre. En effet, il est possible de calculer l'affectation ψ ainsi que le plongement ϕ simultanément en un seul parcours de l'arbre : suivant un parcours en largeur, il suffit d'initialiser le vecteur de bits de l'élément x par celui de son parent puis mettre à 1 le $\psi(x)^{\text{ème}}$ bit du vecteur. La racine reçoit le vecteur nul (vecteur dont tous les bits valent 0). Notons que la taille de ce codage est en $\mathcal{O}(\log_2(|X|))$.

2.2 Codage *via* un étiquetage suivant un parcours infixe

Soit $T = (X, \leq_T)$ un arbre binaire complet de hauteur h. Nous proposons dans cette sous-section un codage de T inspiré de l'algorithme de Harel et Tarjan. Cet algorithme permet de trouver le plus grand ancêtre commun de deux éléments d'un arbre en temps logarithmique et se base sur un étiquetage des éléments de l'arbre suivant un parcours infixe [Harel et Tarjan, 1984].

♦ Description du codage

Nous considérons l'application ψ de X vers $\{1, \ldots, 2^{h+1} - 1\}$ qui associe au $i^{\text{ème}}$ élément visité de T, suivant un parcours infixe, la valeur i. Nous rappelons qu'un parcours infixe d'un arbre binaire consiste à visiter récursivement les éléments de son sous-arbre de gauche, puis sa racine et enfin les éléments de son sous-arbre de droite.

Nous définissons ensuite le codage ϕ qui associe à chaque élément x de T la représentation binaire de l'entier $\psi(x)$. Il s'agit d'une représentation sous forme d'un vecteur de bits de taille au plus $h+1$, i.e. $\lceil \log_2(|X|) \rceil$. Cela vient du fait que tout entier p est représenté par $\lceil \log_2(p) \rceil$ bits.

Harel et Tarjan affirment que le plus grand ancêtre commun de deux éléments x et y de T se trouve à une distance de $d = \lfloor \log_2(r) \rfloor$ des feuilles de T, avec r l'entier dont la représentation binaire est $\phi(x)$ XOR $\phi(y)$. Ils montrent ainsi qu'il s'agit de l'élément z tel que $\psi(z)$ vaut $2^{d+1}\lfloor \frac{\psi(x)}{2^{d+1}} \rfloor + 2^d$ qui vaut également $2^{d+1}\lfloor \frac{\psi(y)}{2^{d+1}} \rfloor + 2^d$. Grâce à ces résultats, ils introduisent la fonction *nca* qui prend en paramètre deux éléments de l'arbre et retourne leur plus grand ancêtre commun en temps logarithmique.

Nous considérons la relation d'ordre $\preceq_{inf}$ définie par $x \preceq_{inf} y$ si et seulement si $nca(x, y) = x$. Montrons alors que ϕ est un plongement de T dans $(\{0,1\}^{h+1}, \preceq_{inf})$. Pour cela, nous devons prouver que pour toute paire d'éléments x et y de X, $x \leq_T y$ si et seulement si $x \preceq_{inf} y$.

Par définition, nous avons $x \leq_T y$ implique $nca(x, y) = x$, i.e. $x \preceq_{inf} y$, et la réciproque est aussi vraie. Par conséquent, le plongement ϕ définit un codage par vecteur de bits de T, que nous illustrons par la Figure 4.2 (les valeurs en rouge sont calculées par l'application ψ).

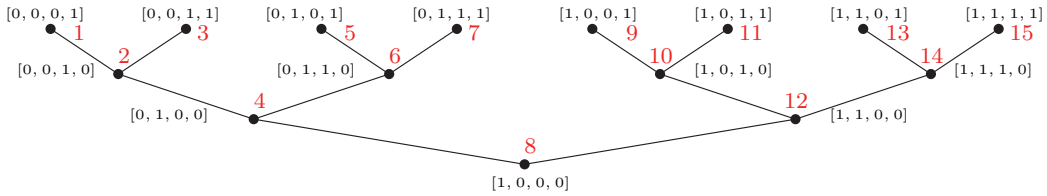


FIGURE 4.2: Codage d'un arbre binaire complet *via* un étiquetage suivant un parcours infixe

♦ Évaluation du codage

Nous avons calculé un codage par vecteur de bits de T grâce au plongement de T dans $(\{0,1\}^{h+1}, \preceq_{inf})$. La taille de ce codage est $h+1$, et elle est minimale puisque l'arbre T est de taille 2^{h+1}.

Pour répondre à la requête « $x \leq y$? », pour x et y deux éléments de T, il suffit de calculer $nca(x, y)$: trouver la valeur de d puis de $\psi(z)$. Ce calcul fait appel aux opérations : $\lfloor \log_2 \rfloor$; division entière sur une puissance de 2 ; multiplication par une puissance de 2 ; addition ; opération logique XOR. Nous traduisons quelques opérations arithmétiques en des opérations logiques :

– L'opération $\lfloor \log_2(n) \rfloor$ retourne l'indice du bit le plus à gauche de la représentation binaire de l'entier n.

– Un division entière de A sur 2^B exprime un décalage à droite de A de B bits.
– Une multiplication de A par 2^B signifie que la valeur de A est décalée à gauche B fois.

Remarquons que toutes les opérations arithmétiques et logiques qui interviennent dans le calcul de $nca(x, y)$ s'exécutent en temps linéaire par rapport à la taille des entiers ou des vecteurs de bits en entrée, et donc en $\mathcal{O}(h)$. Enfin, ce codage est calculable en temps polynomial car il nécessite un seul parcours de l'arbre pour coder ses éléments. La complexité en espace de ce codage est en $\mathcal{O}(n^2)$.

2.3 Codage *via* un étiquetage suivant un parcours en largeur

Soit $T = (X, \leq_T)$ un arbre binaire complet de hauteur h. Suivant la même stratégie du codage précédent, inspiré de l'algorithme de Harel et Tarjan, nous proposons un codage de T à partir de son étiquetage suivant un parcours en largeur.

♦ Description du codage

Étant donné un parcours en largeur de T, notons ψ l'application de X vers l'ensemble $\{1, \ldots, 2^{h+1} - 1\}$ qui associe au $i^{\text{ème}}$ élément visité de T la valeur i. Un parcours en largeur d'un arbre consiste à visiter la racine de l'arbre en premier, puis ses fils de gauche à droite, puis les fils de ses fils de gauche à droite et ainsi de suite.

Nous définissons ensuite le codage ϕ qui associe à chaque élément x de T un vecteur de bits de taille $h + 1$ correspondant à la représentation binaire de l'entier $\psi(x)$.

Nous devons maintenant trouver la relation d'ordre R pour que ϕ soit un plongement de T dans $(\{0, 1\}^{h+1}, R)$. Soit x et y deux éléments de X tels que $x \leq_T y$. Cherchons la relation entre les vecteurs de bits $\phi(x)$ et $\phi(y)$ ou tout simplement entre les entiers $\psi(x)$ et $\psi(y)$.

Comme l'étiquetage de T s'est fait suivant un parcours en largeur, nous obtenons les propriétés suivantes (voir Figure 4.3 pour une illustration) :

1. Pour tout élément x de T, distant de h_x de la racine de T, $\psi(x)$ est dans $[\![2^{h_x}, 2^{h_x+1} - 1]\!]$.
2. Soit x un élément de T distant de h_x de la racine de T. Pour tout successeur y de x, distant de h_y de la racine, la valeur $\psi(y)$ est comprise entre $2^{h_y - h_x}\psi(x)$ et $2^{h_y - h_x}\psi(x) + 2^{h_y - h_x} - 1$.

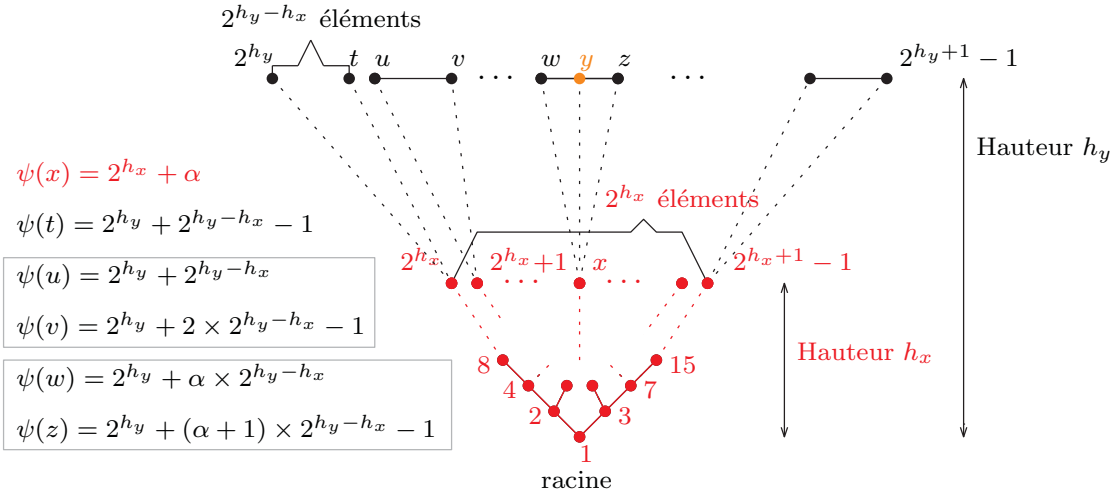


FIGURE 4.3: Propriétés du codage *via* un étiquetage suivant un parcours en largeur

La première propriété implique que tout élément x de T est distant de $\lfloor \log_2(\psi(x)) \rfloor$ de la racine de T. Il s'ensuit que $x \leq_T y$ si et seulement si $\psi(y)$ appartient à l'intervalle d'entiers $[\![\psi(x)2^{h_{xy}}, \psi(x)2^{h_{xy}} + 2^{h_{xy}} - 1]\!]$ avec $h_{xy} = \lfloor \log_2(\psi(y)) \rfloor - \lfloor \log_2(\psi(x)) \rfloor$. Nous en déduisons les équivalences suivantes :

$$\psi(x)2^{h_{xy}} \leq \psi(y) \leq \psi(x)2^{h_{xy}} + 2^{h_{xy}} - 1 \Leftrightarrow$$
$$\psi(x) \leq \frac{\psi(y)}{2^{h_{xy}}} \leq \psi(x) + 1 - \frac{1}{2^{h_{xy}}} \Leftrightarrow$$
$$\lfloor \frac{\psi(y)}{2^{h_{xy}}} \rfloor = \psi(x).$$

Enfin, nous pouvons exprimer la relation d'ordre suivante :

$$x \leq_T y \Leftrightarrow \lfloor \frac{\psi(y)}{2^{\lfloor \log_2(\psi(y)) \rfloor - \lfloor \log_2(\psi(x)) \rfloor}} \rfloor = \psi(x).$$

Comme cette relation d'ordre calcule une division de $\psi(y)$ sur une puissance de 2, nous pouvons alors l'exprimer par un décalage logique à droite de $\phi(y)$. Nous rappelons que $\phi(y)$ correspond à la représentation binaire de $\psi(y)$. Pour redéfinir cette relation d'ordre, nous utilisons le symbole >> du langage bas niveau AVR Assembler. Ce symbole indique un décalage logique de l'opérande de gauche vers la droite du nombre de bits donné par l'opérande de droite. Nous notons `EPL` la relation d'ordre suivante :

$$x \texttt{ EPL } y \Leftrightarrow \phi(y) >> \lfloor \log_2(\psi(y)) \rfloor - \lfloor \log_2(\psi(x)) \rfloor = \phi(x).$$

Il est clair que $x \leq_T y$ si et seulement si x EPL y. Par conséquent, le codage ϕ définit un plongement de T dans $(\{0,1\}^{h+1}$, EPL). Par définition, il s'agit d'un codage par vecteur de bits de T. Ce codage est illustré dans la Figure 4.4 (notons que les valeurs en rouge sont calculée par l'application ψ).

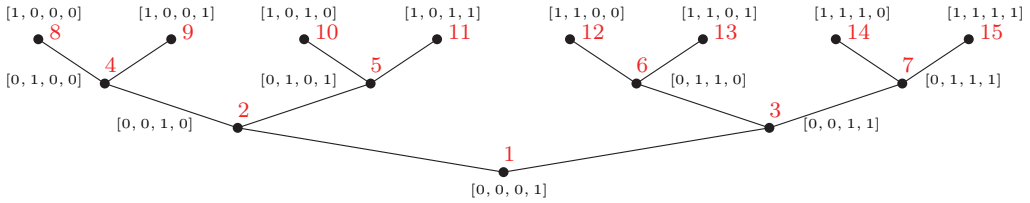


FIGURE 4.4: Codage d'un arbre binaire complet *via* un étiquetage suivant un parcours en largeur

♦ Évaluation du codage

Le codage par vecteur de bits de T est défini par le plongement de T dans $(\{0,1\}^{h+1}$, EPL). La taille de ce codage vaut alors $h+1$ et elle est minimale du fait que l'arbre T possède 2^{h+1} éléments. Le test « $x \leq_T y$? », pour x et y deux éléments de T, est calculé par x EPL y. Ce calcul s'effectue en temps $\mathcal{O}(h)$, linéaire par rapport à la taille du codage. En effet, il suffit de remarquer que ce calcul fait appel à des opérations logiques qui s'exécutent en temps linéaire par rapport à la taille de l'entrée : des vecteurs de bits de taille $h+1$. Notons que le logarithme d'un nombre correspond à l'indice du bit le plus à gauche de sa représentation binaire. Comme le codage est calculé en un seul parcours de T, sa complexité en temps est polynomiale par rapport à la taille de l'arbre. Sa complexité en espace est en $\mathcal{O}(n^2)$.

2.4 Synthèse

Nous avons exposé trois codages par vecteur de bits différents pour les arbres binaires complets. Nous synthétisons dans la Table 4.1 leurs mesures d'efficacité.

Nous constatons que les trois codages sont efficaces d'après les critères rappelés précédemment. Néanmoins, les deux derniers se montrent plus efficaces en pratique vu qu'ils offrent un codage de taille minimale. Il en résulte que l'on peut calculer un codage par vecteur de bits pour le cas des arbres binaires complets plus intéressant que celui basé sur le plongement dans un treillis booléen.

Codage par vecteur de bits	Taille du codage	Coût de « $x \leq_T y$? »	Complexités de calcul du codage
Plongement dans un treillis booléen	$dim_2(T) = 2h$	$\mathcal{O}(h)$	polynomiaux
Étiquetage suivant un parcours infixe	$\lceil \log_2(\lvert T \rvert) \rceil = h+1$	$\mathcal{O}(h)$	polynomiaux
Étiquetage suivant un parcours en largeur	$\lceil \log_2(\lvert T \rvert) \rceil = h+1$	$\mathcal{O}(h)$	polynomiaux

TABLE 4.1: Caractéristiques des codages par vecteur de bits d'un arbre binaire complet T de hauteur h

Nous traitons dans la suite le cas des arbres binaires en nous servant des codages proposés dans cette section.

3 Codages par vecteur de bits des arbres binaires

Nous rappelons dans cette section le codage par vecteur de bits des arbres binaires *via* un plongement dans un treillis booléen, puis nous donnons un autre codage par vecteur de bits plus efficace pour cette classe d'arbres.

Soit $T = (X, \leq_T)$ un arbre binaire de hauteur h et différent d'une chaîne. Comme la complexité de calcul de $dim_2(T)$ est encore ouverte (Conjecture 3.0.2), on ne sait pas s'il existe un algorithme temps polynomial calculant un plongement de T dans le treillis booléen de dimension minimale. Nous rappelons que ce plongement décrit un codage par vecteur de bits de T. Pour assurer un tel codage en temps polynomial, nous relâchons la contrainte de la minimalité de la dimension du treillis. Ainsi, il est possible de calculer un codage par vecteur de bits de T *via* un plongement dans un treillis booléen de dimension au plus $2h$, en appliquant l'une des heuristiques de codage des arbres étudiées dans le second chapitre. Notons que le codage calculé teste la relation d'ordre entre deux éléments de T en temps linéaire par rapport à la taille du codage.

Pour améliorer ce codage par vecteur de bits, nous nous inspirons de celui proposé pour le cas des arbres binaires complets basé sur un étiquetage suivant un parcours en largeur. Soit ψ une fonction de X vers $\{1, \ldots, 2^{h+1} - 1\}$ qui associe à l'élément x la valeur $\psi(x)$ telle que :

- $\psi(x) = 1$ si x est la racine de T
- $\psi(x) = 2\psi(u)$ si x est le fils de gauche de u
- $\psi(x) = 2\psi(u) + 1$ si x est le fils de droite de u

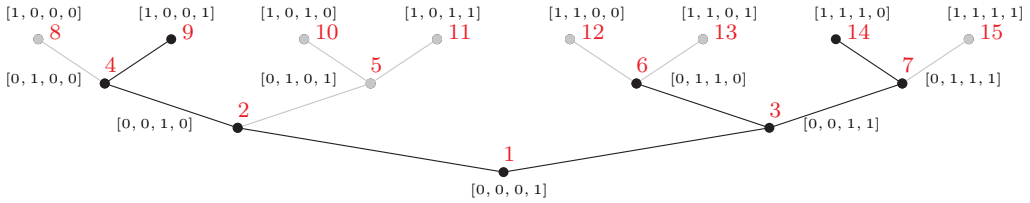


FIGURE 4.5: Codage d'un arbre binaire *via* un étiquetage suivant un parcours en largeur de l'arbre binaire complet le contenant

La fonction ψ calcule un étiquetage de T suivant un parcours en largeur de l'arbre binaire complet contenant T. D'après ce qui précède (voir la section précédente), cet étiquetage offre un plongement de l'arbre dans $(\{0,1\}^{h+1}, \texttt{EPL})$. Par conséquent, il s'agit d'un codage de T par vecteur de bits de taille $h+1$. Ce codage, illustré par la Figure 4.5, améliore effectivement celui basé sur le calcul de $dim_2(T)$ puisqu'il est calculable en temps polynomial et il assure le test de comparabilité en temps $\mathcal{O}(h)$, mais surtout parce que sa taille est plus petite que $dim_2(T)$. En effet, comme l'arbre T n'est pas une chaîne, il contient une chenille de hauteur h et à 2 feuilles, dont la 2-dimension vaut $h + 1$ (Proposition 1.3.6). D'après la monotonie du paramètre dim_2, nous déduisons que $h + 1 \leq dim_2(T)$.

4 Codages par vecteur de bits des arbres

À présent, le seul codage par vecteur de bits connu pour tout arbre est celui basé sur un plongement dans un treillis booléen. Nous proposons ici un nouveau codage par vecteur de bits des arbres, inspiré de celui proposé dans la section précédente pour le cas des arbres binaires et aussi de l'algorithme *Unicho* (Algorithme 10), décrit dans le troisième chapitre. Rappelons que cet algorithme calcule un plongement d'un arbre dans un arbre binaire de hauteur minimale.

Soit T un arbre quelconque. En appliquant l'algorithme *Unicho* sur T, nous générons un arbre binaire B de hauteur $Unicho(T)$ tel que T se plonge dans B ($T \rightsquigarrow B$). Nous en déduisons que tout codage par vecteur de bits de B peut induire un codage par vecteur de bits de T puisque $B \rightsquigarrow (\{0,1\}^n, R)$ implique $T \rightsquigarrow (\{0,1\}^n, R)$. Nous pouvons ainsi définir un nouveau codage par vecteur de bits de T calculé en deux étapes :

1. Pré-traitement : appliquer l'algorithme *Unicho* sur T afin de générer un arbre binaire B.
2. Codage : calculer un codage par vecteur de bits de B à partir du plongement de B dans $(\{0,1\}^{Unicho(T)+1}, \texttt{EPL})$.

Ce codage a une complexité en temps polynomial du fait que les deux étapes permettant de le calculer sont réalisables en temps polynomial. De plus, il permet de tester la relation d'ordre entre deux éléments x et y de T en temps linéaire par rapport à la taille du codage, puisque cela revient à tester si x EPL y. Si notre Conjecture 3.3.8 est vraie, notre codage est alors plus intéressant que le codage des arbres *via* un plongement dans un treillis booléen, puisque sa taille $Unicho(T)+1$ est plus petite que $dim_2(T)$.

5 Ouverture

Dans ce manuscrit, nous traitons le problème de codage des ordres et nous nous focalisons sur le codage par vecteur de bits. À l'instar des autres méthodes de codages existantes, nous nous intéressons à l'étude de la classe des arbres pour concevoir de meilleures heuristiques de codage des ordres par vecteur de bits ou encore pour améliorer cette méthode de codage.

Dans ce chapitre, nous avons généralisé la notion de codage par vecteur de bits, dont la notion usuelle est liée au calcul de la 2-dimension. Ainsi, nous avons montré qu'il existe des codages par vecteur de bits plus intéressants que le calcul de la 2-dimension pour certains arbres particuliers. Nous conjecturons que c'est le cas pour tout arbre différent d'une chaîne. Cette conjecture peut donner des indices pour répondre à la question principale qui se pose :

Existe-t-il des codages par vecteur de bits plus efficaces que le calcul de la 2-dimension pour tout ordre ?

Conclusion générale et perspectives

Les travaux de cette thèse s'articulent autour de la 2-dimension des ordres. Nous récapitulons dans cette conclusion certains des travaux réalisés et donnons quelques perspectives.

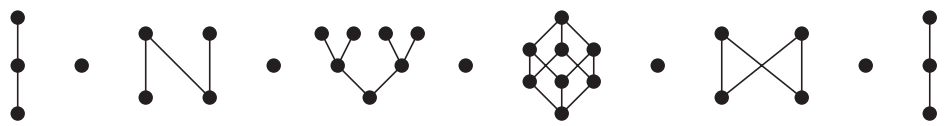
Au premier chapitre, nous avons dressé un état de l'art autour de la 2-dimension des ordres partiels. Nous avons commencé par une revue de la littérature sur les méthodes de codage des ordres existantes pour introduire par la suite le codage par vecteur de bits dont la taille minimale correspond à la 2-dimension. Nous avons ensuite cité quelques propriétés de ce paramètre (dim_2) puis expliqué le principe des heuristiques proposées pour approcher sa valeur. Ces heuristiques calculent un codage des ordres par vecteur de bits permettant ainsi de déterminer l'espace mémoire dédié pour leur stockage et de calculer le temps nécessaire pour récupérer la relation d'ordre entre chaque paire de leurs éléments.

Dans le second chapitre, nous avons proposé de nouvelles heuristiques de codage des ordres par vecteur de bits. Nous nous sommes d'abord intéressés au codage des ordres séries-parallèles à travers la technique de décomposition modulaire. Nous avons ainsi proposé une heuristique qui calcule, en temps polynomial, un codage par vecteur de bits de ces ordres de taille plus petite que celle de leur codage par l'heuristique de référence *SBSC*. Nous avons ensuite généralisé cette heuristique pour coder tout ordre en introduisant l'opérateur du « *Blow up* », permettant de traiter les nœuds premiers. Deux options ont été proposées : une première, GSB qui fait le pré-traitement *Split and Balancing* avant de calculer la décomposition modulaire de l'ordre puis son codage, et une seconde, LSB qui fait d'abord une décomposition modulaire de l'ordre initial puis code ses sous-ordres récursivement, en appliquant la phase du pré-traitement lors du codage des sous-ordres associés à un nœud premier. Les tests effectués sur des *benchmarks* connues ont montré que la meilleure taille de leur codage est retournée par l'une des deux options. Il serait intéressant de concevoir un processus unique combinant efficacement le GSB et le LSB. Nous avons enfin consacré la dernière section de ce chapitre au cas des arbres. Suite à une analyse du comportement des trois dernières heuristiques introduites pour le codage des arbres par vecteur de bits, nous nous sommes rendu compte que leur stratégie commune consiste à calculer un poids pour chaque élément d'un arbre en deux étapes : la génération d'une flat-séquence puis son codage. La taille du codage correspond au poids de la racine. L'heuristique que nous proposons suit

cette stratégie. Son efficacité s'appuie sur le fait qu'elle minimise la taille de la flat-séquence générée, ce qui minimise par conséquent les poids attribués aux éléments d'un arbre et donc la taille de son codage. Nous ouvrons des pistes de réflexion sur le calcul d'une flat-séquence de taille minimale et sur son codage optimal.

Au troisième chapitre, nous avons abordé trois conjectures autour de la 2-dimension des arbres. La première affirme que la 2-dimension exacte des arbres n-complets de hauteur h vaut exactement $h.sp(n)$. Nous avons vérifié la conclusion de la conjecture pour les arbres 2-complets et les arbres 3-complets de hauteur au plus 2. Nous l'avons par contre réfutée à l'aide des arbres 3-complets de hauteur au moins 3 ainsi que les arbres 4-complets. Nous nous demandons s'il existe une autre expression de la 2-dimension de ces arbres, même pour $h = 2$. Dans la deuxième section, nous nous sommes intéressés à une deuxième conjecture à propos d'une borne inférieure de la 2-dimension des arbres. Étant donné un arbre T avec $T_1,\ldots,T_k$ ses sous-arbres dont chacun est enraciné en un fils de sa racine,la conjecture affirme que $2^{dim_2(T)}$ est au moins $\sum_{i=1}^{k} 2^{dim_2(T_i)}$. Nous avons validé la conclusion de la conjecture pour les arbres 4-aires puis nous avons trouvé un contre exemple dans la classe des arbres 5-aires. La dernière conjecture étudiée dans ce chapitre affirme que l'algorithme *Dicho* est une 2-approximation de la 2-dimension des arbres. Dans notre étude, nous avons validé la conclusion de la conjecture dans le cas des chenilles et des arbres 2^k-complets. Nous avons également abordé quelques pistes pour la prouver dans le cas général qui restent encore ouvertes. Nous avons clôturé cette étude par une reformulation de la conjecture. Nous conjecturons que la hauteur minimale d'un arbre binaire dans lequel un arbre T peut être plongé est plus petite que la 2-dimension de T. Nous avons ainsi montré que notre conjecture implique que l'algorithme *Dicho* est une 2-approximation de la 2-dimension des arbres. Les deux conjectures font partie des perspectives.

Dans le quatrième chapitre, nous avons généralisé la notion de codage des ordres par vecteur de bits. Nous avons par la suite montré qu'il existe des codages efficaces par vecteur de bits, pour le cas des arbres binaires, de taille plus petite que leur 2-dimension. Nous pensons que ce résultat est valide pour tout arbre, et pourquoi pas pour tout ordre ?



Index

Bibliographie

Agrawal, R., Borgida, A. et Jagadish, J. V. (1989). Efficient management of transitive relationships in large data and knowledge bases. *ACM SIGMOD International Conference on Management of Data*, pages 115–146. 35, 184

Aït-Kaci, H., Boyer, R., Lincoln, P. et Nasr, R. (1989). Efficient implementation of lattice operations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(1):115–146. 44

Baldy, P. et Morvan, M. (1993). A linear time and space algorithm to recognize interval orders. *Discrete Applied Mathematics*, 46(2):173–178. 33, 184

Barbie, M., Hagedorn, M. et Kaul, A. (2001). Government debt as insurance against macroeconomic risk. 23

Beaudou, L., Ghazi, K., Kahn, G., Raynaud, O. et Thierry, É. (2016). Encoding partial orders through modular decomposition. *Proceedings of COMPSE*. 3

Benzer, S. (1959). On the topology of the genetic fine structure. *Proc Natl Acad Sci USA*, 45(11):1607–1620. 34

Borassi, M., Crescenzi, P. et Habib, M. (2016). Into the square : On the complexity of some quadratic-time solvable problems. *Electronic Notes in Theoretical Computer Science*, 322:51–67. 73, 74

Bouchet, A. (1971). *Etude combinatoire des ordonnés finis*. Thèse de doctorat, Université Joseph-Fourier-Grenoble I. 44

Brüggemann, R. et Patil, G. P. (2011). Partial order and related disciplines. *Environmental and Ecological Statistics - Ranking and Prioritization for Multi-indicator Systems*, pages 271–278. 22

Capelle, C. (1994). Representation of an order as union of interval orders. *Proceedings of the International Workshop on Orders, Algorithms, and Applications*, pages 143–161. 36

Capelle, C., Habib, M. et De Montgolfier, F. (2002). Graph decompositions and factorizing permutations. *Discrete Mathematics and Theoretical Computer Science*, 5(1):55–70. 75

Caseau, Y. (1993). Efficient handling of multiple inheritance hierarchies. *Proceedings of OOPSLA '93*, pages 271–287. 50, 56, 98

Caseau, Y., Habib, M., Nourine, L. et Raynaud, O. (1999). Encoding of multiple inheritance hierarchies and partial orders. *Computational Intelligence*, 15:50–62. 51, 54, 57, 93, 98

Caspard, N., Leclerc, B. et Montjardet, B. (2007). *Ensembles ordonnés finis : concepts, résultats et usages*. springer. 7, 8, 9, 10, 11, 12, 17

Ceroi, S. (2000). *Ordres et géométrie plane : Application au nombre de sauts*. Thèse de doctorat, Université de Montpellier 2. 44

Colomb, P., Raynaud, O. et Thierry, É. (2008). Generalized polychotomic encoding. *Proceedings of MCO '08*, pages 77–86. 3, 58, 67, 76, 112

Conrad, K. (2012). Zorn's lemma and some applications. 23

Coombs, C. H. et Smith, J. E. (1973). On the detection of structures in attitudes and developmental processes. *Psychological Review*, 80:337–351. 34

Davey, B. A. et Priestley, H. A. (2002). *Introduction to lattices and order*. Cambridge University Press. 7

Dushnik, B. et Miller, E. W. (1941). Partially ordered sets. *American Journal of Mathematics*, 63:600–610. 24, 37, 39

Faigle, U., Kern, W. et Still, G. (2013). *Algorithmic principles of mathematical programming*, volume 24. Springer Science & Business Media. 18

Fall, A. (1998). The foundations of taxonomic encoding. *Computational Intelligence*, 14(4):598–642. 44

Fernandez, P. L., Heath, L. S., Ramakrishnan, N., Tan, M. et Vergara, J. P. C. (2009). Mining posets from linear orders. 1

Filman, R. E. (2002). Polychotomic encoding : A better quasi-optimal bit-vector encoding of tree hierarchies. *Proceedings of ECOOP '02*, pages 545–561. 3, 58, 102, 103, 105

Fishburn, P. C. (1970). Intransitive indifference with unequal indifference intervals. *Journal of Mathematical Psychology*, 7(1):144–149. 33

Foutlane, A., Bouloud, A. et Ghedda, K. (1997). Restauration de la qualité des eaux des retenues de barrages. *Freshwater Contamination (Proceedings of Rabat Symposium S4)*. 1

Garg, V. K. (2015). *Introduction to Lattice Theory with Computer Science Applications*. John Wiley & Sons. 63

Gazagnaire, T. (2008). *Langages de scénarios : Utiliser des ordres partiels pour modéliser, vérifier et superviser des systèmes parallèles et répartis*. Thèse de doctorat, Université Rennes 1. 1, 22

Goralčíková, A. et Koubek, V. (1979). *A reduct-and-closure algorithm for graphs*, pages 301–307. Springer Berlin Heidelberg. 73, 74

Gray, F. (1953). Pulse code communication. 121

Greenough, T. (1976). *Representation and Enumeration of Interval Orders and Semiorders*. Thèse de doctorat, Dartmouth College, Hanove. 33

Habib, M., Huchard, M. et Nourine, L. (1995). Embedding partially ordered sets into chain-products. *Proceedings of KRUSE '95*, pages 147–161. 25, 60

Habib, M., Nourine, L., Raynaud, O. et Thierry, É. (2004). Computational aspects of the 2-dimension of partially ordered sets. *Theoretical Computer Science*, 312(2-3):401–431. 3, 4, 5, 47, 48, 49, 57, 58, 59, 99, 100, 105, 130, 137, 143, 146, 152, 153, 154

Habib, M. et Paul, C. (2010). A survey of the algorithmic aspects of modular decomposition. *Computer Science Review*, 4:41–59. 7

Harel, D. et Tarjan, R. E. (1984). Fast algorithms for finding nearest common ancestors. *siam Journal on Computing*, 13(2):338–355. 162

Hausdorff, F. (1914). *Grundzüge der Mengenlehre*. Leipzig. 23

Hausdorff, F. (1957). *Set Theory*. Chelsea. 23

Itai, A. et Rodeh, M. (1978). Finding a minimum circuit in a graph. *SIAM Journal on Computing*, 7(4):413–423. 74

Kahn, G. (2015). Codage par vecteur de bits des ensembles ordonnés, antichaînes et arbres. Mémoire de D.E.A., Université Blaise Pascal. 79, 132, 133

Karp, R. M. (1993). Mapping the genome : some combinatorial problems arising in molecular biology. *Proceedings of STOC '93*, 80:278–285. 34

Kendall, D. G. (1969). Incidence matrices, interval graphs, and seriation in archaeology. *Pacific Journal of Mathematics*, 28:565–570. 34

Konstantopoulos, T. (2010). The axiom of choice and equivalent statements. 23

Krall, A. et Grafl, R. (1997). Cacao — a 64-bit javavm just-in-time compiler. *Concurrency : Practice and Experience*, 9(11):1017–1030. 27

Krall, A., Vitek, J. et Horspool, R. N. (1997). Near optimal hierarchical encoding of types. *Proceedings of ECOOP '97*, pages 128–145. 3, 4, 52, 54, 92, 93, 95, 185

La, H. T. (2005). *Utilisation d'ordres partiels pour la caractérisation de solution robustes en ordonnancement*. Thèse de doctorat, LAAS-CNRS. 1, 22

Lam, T. (2015). The uncrossing partial order on matchings is eulerian. *Journal of Combinatorial Theory, Series A*, 135:105–111. 1

Lavallée, I. (2008). *Complexité et algorithmique avancée*. Hermann. 7

Lawler, E. (1978). Sequencing jobs to minimize total weighted completion time subject to precedence constraints. *In* Alspach, B., Hell, P. et Miller, D., éditeurs : *Algorithmic Aspects of Combinatorics*, volume 2 de *Annals of Discrete Mathematics*, pages 75–90. Elsevier. 63

Leibniz, G. W. (1989). *A study in the logical calculus*, pages 371–382. Springer Netherlands. 23

Loukil, R. (2016). "La loi de Moore n'est pas morte", assure le PDG d'Intel. *L'Usine Digitale*. 27

Madej, T. et West, D. B. (1991). The interval inclusion number of a partially ordered set. *Discrete Mathematics*, 88(2):259–277. 37, 39

Meyer, R. K. (1987). God exists ! *Noûs*, 21(3):345–361. 23

Möhring, R. H. (1985). *Algorithmic Aspects of Comparability Graphs and Interval Graphs*. Springer Netherlands. 63

Möhring, R. H. (1989). *Computationally Tractable Classes of Ordered Sets*, pages 105–193. Springer Netherlands. 63

Montgolfier, F. d. (2003). *Décomposition modulaire des graphes : théorie, extensions et algorithmes*. Thèse de doctorat, Montpellier 2, Université des Sciences et Techniques du Languedoc. 75

Nökel, K. (1991). *Temporally distributed symptoms in technical diagnosis*, volume 517. Springer Science & Business Media. 34

Novak, V. (1963). On the pseudo-dimension of ordered sets. *Czechoslovak Math. Journal*, 13:587–598. 43, 44

Pouzet, M. et Richard, D. (1984). *Orders : Description and Roles*. Elsevier. 22, 60

Pratt, V. (1986). Modelling concurrency with partial orders. *International Journal of Parallel Programming*. 1, 22

Rabinovich, T. (2005). *The dimension theory of semiorders and interval orders*. Thèse de doctorat, LAAS-CNRS. 33

Raynaud, O. et Thierry, E. (2001). A quasi optimal bit-vector encoding of tree hierarchies. application to efficient type inclusion tests. *Proceedings of ECOOP'2001*, pages 165–180. 98, 99

Roman, S. (2009). *Lattices and ordered sets*. springer. 7

Roux, C. (2009). *Initiation à la théorie des graphes*. ellipses. 7

Sipser, M. (1996). *Introduction to the theory of computation*. International Thomson Publishing, 1st édition. 7

Skorsky, M. (1992). *Endliche verbU ande diagramme und eigenschaften*. Thèse de doctorat, Darmstad, Germany. 49, 50

Sperner, E. (1928). Ein satz uber untermengen einer endlichen menge. *Math. Z*, 27:544–548. 47

Stahl, J. et Wille, R. (1986). *Preconcepts and set representation of contexts*. North–Holland, North-Holland Amsterdam. 48, 130

Szpilrajn, E. (1930). Sur l'extension de l'ordre partiel. *Fundamenta Mathematicae*, 16(1):386–389. 23

Takamizawa, K., Nishizeki, T. et Saito, N. (1981). Combinatorial problems on series-parallel graphs. *Discrete Applied Mathematics*, 3(1):75–76. 63

Thierry, É. (2001). *Sur quelques interactions entre structures de données et algorithmes efficaces pour les ordres et les graphes*. Thèse de doctorat, LIRMM, Université Montpellier II. 3, 4, 48, 49, 58, 59, 132, 142, 149

Trotter, W. T. (1975). Embedding finite posets in cubes. *Discrete Mathematics*, 12(2):165–172. 44, 48, 58, 65

Trotter, W. T. (1995). Partially ordered sets. *Handbook of Combinatorics*, 1:433–480. 7

Trotter, W. T. (1997). New perspectives on interval orders and interval graphs. *In in Surveys in Combinatorics*. Citeseer. 33

Urrutia, J. (1989). Partial orders and euclidean geometry. *In Algorithms and Order*, pages 387–434. Springer. 1

Van Emde Boas, P. (1977). Preserving order in a forest in less than logarithmic time and linear space. *Information processing letters*, 6(3):80–82. 27

Vitek, J., Horspool, R. N. et Krall, A. (1997). *Efficient type inclusion tests*, volume 32. ACM. 89

West, D. B. (2000). *Introduction to graph theory*. Prentice Hall, 2 édition. 7

Widmer, M. (1998). *Organisation industrielle : le réel apport des mathématiques*. Thèse de doctorat, Université de Fribourg, Suisse. Thèse d'habilitation. 62

Willard, D. E. (1984). New trie data structures which support very fast search operations. *Journal of Computer and System Sciences*, 28(3):379–394. 27

Yannakakis, M. (1982). The complexity of partial order dimension problem. *SIAM Journal on Algebraic Discrete Methods*, 3:351–358. 37, 38

Yannakakis, M. et Papadimitriou, C. H. (1979). Scheduling interval ordered tasks. *SIAM Journal on Computing*, 8(3):405–409. 34

Zibin, Y. et Gil, J. Y. (2001). Efficient subtyping tests with pq-encoding. *Proceedings of OOPSLA '01*, pages 96–107. 44

Appendice

A. Algorithmes

Données : Un ordre (X, R) de taille n
Résultat : Un codage de l'ordre par une matrice d'adjacence
1 Initialisation d'une matrice binaire M de taille $n \times n$ à zéro
2 **pour** *tout* (x, y) *dans* R **faire**
3 | $M[x][y] \leftarrow 1$
4 **fin**

Algorithme 11 : Codage des ordres par matrice d'adjacence

Données : Un ordre (X, R) de taille n
Résultat : Un codage de l'ordre par liste d'adjacence
1 Construire un tableau L de n listes vides
2 **pour** *tout* (x, y) *dans* R **faire**
3 | $L[x] \leftarrow L[x] \cup \{y\}$
4 **fin**

Algorithme 12 : Codage des ordres par liste d'adjacence

Données : Un ordre (X, R) de taille n réduit par transitivité
Résultat : Un codage de l'ordre par liste des successeurs immédiats
1 Construire un tableau L de n listes
2 **pour** *tout* (x, y) *dans* R **faire**
3 | $L[x] \leftarrow L[x] \cup \{y\}$
4 **fin**

Algorithme 13 : Codage des ordres par liste des successeurs immédiats

Données : Un ordre (X, R) de taille n
Résultat : Un codage de l'ordre par un ensemble d'intervalles

$T \leftarrow$ arbre couvrant optimum de P [Agrawal *et al.*, 1989]
Calculer un codage de T par intervalle unique
$O \leftarrow$ ordre topologique inverse des éléments de P
pour *tout* x *dans* O **faire**
    $S_x \leftarrow I_x \cup \emptyset$
    pour *tout* y *dans* $ImmSucc_P(x)$ **faire**
        $S_x \leftarrow S_x \cup S_y$
    fin
fin

Algorithme 14 : Codage des ordres par compression de fermeture transitive

Données : Un ordre $P = (X, R)$ de taille n
Résultat : Un codage de P par un ensemble d'intervalles

$S \leftarrow R$
$i \leftarrow 1$
tant que $S \neq \emptyset$ **faire**
    $P_i \leftarrow (X, S)$
    Enlever des comparabilités de S jusqu'à ce que $P_i = (X, R_i)$ soit un ordre d'intervalle (vérification en temps linéaire par l'algorithme [Baldy et Morvan, 1993])
    Calculer un codage de P_i par intervalle unique
    pour *tout* x *dans* X **faire**
        $code(x) \leftarrow code(x) \cup (i, a, b)$ avec $[a, b]$ code de x dans P_i
    fin
    $S \leftarrow S \backslash R_i$
    $i \leftarrow i + 1$
fin

Algorithme 15 : Codage des ordres par union d'intervalle

Données : Un ordre P
Résultat : Codage réduit de P
$G_{conflit} \leftarrow$ graphe de conflit de P
Colorer les sommets de $G_{conflit}$
pour *tout* i *dans* P **faire**
    $code(i) \leftarrow \{\}$
    si i *est dans* $G_{conflit}$ **alors**
        $code(i) \leftarrow \{couleur(i)\}$
    fin
fin

Algorithme 16 : *Simple Coloration* d'un ordre

Données : Un ordre P
Résultat : Codage de P par *SBSC*
$P' \leftarrow$ Pré-traitement *Split and Balancing* [Krall *et al.*, 1997] de P
Simple Coloration de P'
pour *tout* x *de* P **faire**
    $code(x) \leftarrow code(x) \cup \{code(y) | y \in P' \text{ and } y \leq_{P'} x\}$
fin

Algorithme 17 : Codage *SBSC* d'un ordre

Données : Un arbre T
Résultat : Un codage de T par $Cmax$

$\mathcal{O} \leftarrow$ les éléments de T suivant un parcours en largeur
$x \leftarrow 1$ // la racine de T
$code(x) \leftarrow \emptyset$ // code de l'élément x
$x_{max} \leftarrow 0$ // nombre maximal de codes utilisés et ne pouvant coder les successeurs de x
pour x *dans* $\mathcal{O}$ **faire**
    $S \leftarrow \{1 + x_{max}, \ldots, |ImmSucc_T(x)| + x_{max}\}$
    pour c *dans* S *et* y *dans* $ImmSucc_T(x)$ **faire**
        $code(y) \leftarrow code(x) \cup c$
        $y_{max} \leftarrow |ImmSucc_T(x)| + x_{max}$
    fin
fin

Algorithme 18 : Codage des arbres par $Cmax$

Données : Un arbre T
Résultat : Un codage de T par $CHNR$

$\mathcal{O} \leftarrow$ les éléments de T suivant un parcours en largeur
$x \leftarrow 1$ // la racine de T
$code(x) \leftarrow \emptyset$ // code de l'élément x
$x_{max} \leftarrow 0$ // nombre maximal de codes utilisés et ne pouvant coder les successeurs de x
pour x *dans* $\mathcal{O}$ **faire**
    $S \leftarrow$ toutes les combinaisons de $\lfloor \frac{sp(|ImmSucc_T(x)|)}{2} \rfloor$ éléments de l'ensemble $\{1 + x_{max}, \ldots, sp(|ImmSucc_T(x)|) + x_{max}\}$
    pour c *dans* S *et* y *dans* $ImmSucc_T(x)$ **faire**
        $code(y) \leftarrow code(x) \cup c$
        $y_{max} \leftarrow sp(|ImmSucc_T(x)|) + x_{max}$
    fin
fin

Algorithme 19 : Codage des arbres par $CHNR$

Données : Un arbre T
Résultat : Un codage de T par *Dicho*

```
1  // initialisation des poids des éléments de T
2  pour tout x dans T faire
3  |  si x est une feuille alors
4  |  |  w(x) ← 0
5  |  fin
6  |  sinon
7  |  |  w(x) ← −1
8  |  fin
9  fin
10 // Plongement de T dans un arbre binaire B
11 tant que il existe un élément x de poids inférieur à zéro et dont tous les
     fils ont des poids positifs faire
12 |  S ← une séquence croissante [s_1, ..., s_n] des poids des éléments de
   |    ImmSucc_T(x)
13 |  si |ImmSucc_T(x)| = 1 alors
14 |  |  w(x) ← s_1 + 1
15 |  fin
16 |  sinon si |ImmSucc_T(x)| = 2 alors
17 |  |  w(x) ← s_2 + 2
18 |  fin
19 |  sinon
20 |  |  Introduire un nouvel élément s comme fils de x et père des éléments
   |  |    de poids s_1 et s_2
21 |  |  w(s) ← s_2 + 2
22 |  fin
23 fin
24 Coder l'arbre B par l'algorithme CHNR
```

Algorithme 20 : Codage des arbres par l'algorithme *Dicho*

Données : Un arbre T
Résultat : Un codage de T par *Polychotomique*

```
1  // initialisation des poids des éléments de T
2  pour tout x dans T faire
3  |  si x est une feuille alors
4  |  |  w(x) ← 0
5  |  fin
6  |  sinon
7  |  |  w(x) ← −1
8  |  fin
9  fin
10 // Plongement de T dans un arbre T'
11 tant que il existe un élément x de poids inférieur à zéro et dont tous les
     fils ont des poids positifs faire
12 |  S ← une séquence croissante [s_1, ..., s_n] des poids des éléments de
   |    ImmSucc_T(x)
13 |  si |S| = 1 alors
14 |  |  w(x) ← s_1 + 1
15 |  fin
16 |  sinon si S est une flat-séquence alors
17 |  |  w(x) ← s_n + sp(n)
18 |  fin
19 |  sinon
20 |  |  Introduire un nouvel élément s comme fils de x et père des éléments
   |  |    de poids s_1 et s_2
21 |  |  w(s) ← s_2 + 2
22 |  fin
23 fin
24 Coder l'arbre T' par l'algorithme CHNR
```

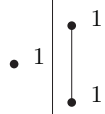
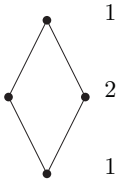
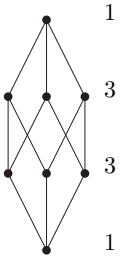
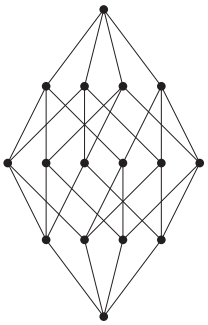
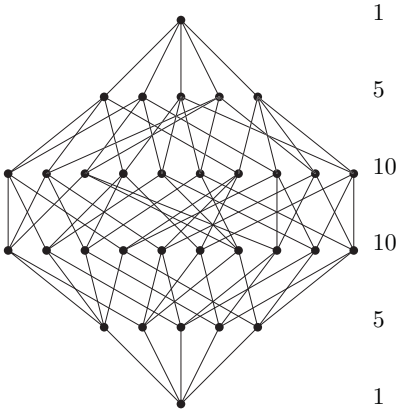
Algorithme 21 : Codage des arbres par l'algorithme *Polychotomique*

Données : Un arbre T
Résultat : Un codage de T par *Polychotomique Généralisé*

```
// initialisation des poids des éléments de T
pour tout x dans T faire
    si x est une feuille alors
        w(x) ← 0
    fin
    sinon
        w(x) ← −1
    fin
fin
// Plongement de T dans un arbre T'
tant que il existe un élément x de poids inférieur à zéro et dont tous les
 fils ont des poids positifs faire
    S ← une séquence croissante [s_1, ..., s_n] des poids des éléments de
     ImmSucc_T(x)
    si |S| = 1 alors
        w(x) ← s_1 + 1
    fin
    sinon si S est une flat-séquence alors
        w(x) ← s_n + sp(n)
    fin
    sinon si ∃k avec [s_1, ..., s_k] une flat-séquence et s_k + sp(k) ≤ s_{k+1} alors
        Introduire un nouvel élément s comme fils de x et père des éléments
         de poids s_1, ..., s_k
        w(s) = s_k + sp(k)
    fin
    sinon
        Introduire un nouvel élément s comme fils de x et père des éléments
         de poids s_1 et s_2
        w(s) ← s_2 + 2
    fin
fin
Coder l'arbre T' par l'algorithme CHNR
```

Algorithme 22 : Codage des arbres par l'algorithme *Polychotomique Généralisé*

B. Triangle de Pascal vs Fonction Sperner

	Triangle de Pascal					
Treillis Booléen						
Dimension	0	1	2	3	4	5
Sperner de		1	2	3	{4,5,6}	{7,8,9,10}

Le sperner de quelques entiers récupéré à partir du triangle de pascal