UNIVERSITÉ DE MONTRÉAL

ALGORITHMES DE DÉNOMBREMENT D'EXTENSIONS LINÉAIRES D'UN ORDRE PARTIEL ET APPLICATION AUX PROBLÈMES D'ORDONNANCEMENT DISJONCTIF

RACHID CHERKAOUI EL AZZOUZI
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
DÉCEMBRE 2015

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

ALGORITHMES DE DÉNOMBREMENT D'EXTENSIONS LINÉAIRES D'UN ORDRE PARTIEL ET APPLICATION AUX PROBLÈMES D'ORDONNANCEMENT DISJONCTIF

présenté par : CHERKAOUI EL AZZOUZI Rachid
en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de :

M. BILODEAU Guillaume-Alexandre, Ph. D., président
M. PESANT Gilles, Ph. D., membre et directeur de recherche
M. GALINIER Philippe, Doctorat, membre

REMERCIEMENTS

J'aimerais remercier toutes les personnes m'ayant aidé dans la réalisation de ce projet en particulier mon directeur de recherche, professeur Gilles, qui fut une source d'inspiration et de bons conseils. J'aimerais également remercier les membres du laboratoire Quosséça pour leur soutien et leur agréable compagnie.

RÉSUMÉ

En programmation par contraintes, une contrainte de ressource unaire est un ensemble de permutations valides des activités chacune avec une fenêtre de temps et une durée. Cette contrainte est généralisée si on considère des préséances entre activités données sous la forme d'un ensemble partiellement ordonné. Un problème d'ordonnancement disjonctif peut être modélisé par une ou plusieurs contraintes de ressource unaire auxquelles s'ajoutent des contraintes supplémentaires telles que des disjonctions entre activités de différentes ressources ou des contraintes de séquences. La recherche d'une solution au problème se fait par une série de décisions de la position relative d'une paire d'activités associées à une contrainte dont l'ordre n'est pas encore connu. L'algorithme utilisé dans le choix de la paire ainsi que la position relative est appelé heuristique de branchement. Dans le contexte de l'heuristique maxSD, il s'agit de calculer les densités de solutions de toutes les assignations de paires d'activités à un ordre et ensuite de brancher sur celle de densité maximum. Pour adapter cette heuristique aux problèmes d'ordonnancement avec contraintes de ressource unaire, on considérera les densités de permutations dans lesquelles une activité est placée avant l'autre dans l'ordre partiel associé à chaque contrainte. Pour ce faire, on propose deux algorithmes exact et heuristique pour le calcul des densités de permutations dans un ensemble partiellement ordonné. Ces algorithmes sont utilisés dans l'heuristique de branchement pour résoudre la version de satisfaction de contraintes du problème Job-Shop, un cas typique d'ordonnancement avec ressources unaires.

ABSTRACT

In constraint programming a unary resource constraint is a set of valid permutations of activities each with a time window and a duration. This constraint is generalized if we consider precedence constraints between activities given by a partially ordered set. A disjunctive scheduling problem can be stated as a combination of one or more such constraints for which some additional constraints such as disjunction or sequence of activities on different resources may be added. In this model, a solution is found by a series of decisions on the relative position of a pair of activities on a same resource and for which the order is unknown. The algorithm used to select the pair and the order is called a branching heuristic. In the context of maxSD, densities of all assignments of pairs and order are computed and the assignment of maximum density is selected. In order to adapt this heuristic for scheduling problems with unary resources, we will consider the permutations of the partial order in which the rank of an activity is superior to another. For that, we propose exact and heuristic algorithms that compute the density of permutations in a partially ordered set. These algorithms are then used in branching to solve the constraint satisfaction version of the Job-Shop scheduling problem, a typical use case of scheduling with unary resource constraints.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

CSP	Constraint Satisfaction Problem
CSOP	Constraint Satisfaction Optimization Problem
CBS	Counting Based Search
IBS	Impact Based Search
ADP	Approximate Dynamic Programming

LISTE DES ANNEXES

CHAPITRE 1 INTRODUCTION

Les problèmes combinatoires notamment les problèmes d'ordonnancement peuvent être résolus en utilisant différents modèles et différentes stratégies. En programmation par contraintes (PPC), un grand nombre de contraintes globales sont conçues permettant de simplifier la modélisation tout en restant flexible. La contrainte de ressource unaire pour les problèmes d'ordonnancement disjonctif en est un exemple. Dans un modèle disjonctif, la recherche de solution se fait en fixant tour à tour les variables binaires de chaque disjonction. Dans ce chapitre, on donne une brève introduction à la PPC ainsi qu'aux problèmes d'ordonnancement disjonctif. L'objectif de la recherche et le plan de mémoire sont présentés dans les deux dernières sections.

1.1 Programmation par contraintes

La PPC est un paradigme de programmation déclaratif utilisé dans la résolution de problèmes combinatoires variés. Une des principales caractéristiques de la PPC est la facilité à modéliser les différents problèmes par la séparation de la définition de ces derniers des algorithmes utilisés. Les problèmes sont modélisés sous forme de variables chacune avec un domaine désignant l'ensemble des valeurs que la variable peut se faire assigner. Les assignations de valeurs aux variables sont sujettes à des contraintes. Chaque contrainte restreint les combinaisons d'assignations d'un sous-ensemble de variables se trouvant dans sa portée. L'ensemble des variables et contraintes forme un réseau appelé réseau de contraintes. La recherche de solution se fait par une série d'instanciations des variables. Chaque instanciation se fait propager dans le réseau et "réveille" certaines contraintes dont les variables dans leur portée ont subi un changement pour effectuer le filtrage des domaines c.-à-d. la suppression des valeurs incompatibles avec la contrainte. Cela permet le maintien, jusqu'à un certain niveau, de la cohérence dans les domaines des variables. Lorsque toutes les variables se font assigner une valeur dans leurs domaines respectifs alors une solution est trouvée. Formellement, un problème de satisfaction de contraintes est défini par :

Définition 1.1. (problème de satisfaction de contraintes (CSP)) *est un triplet* (X, D, C) *où* $X = \{x_1, \ldots, x_n\}$ *est un ensemble fini de variables,* $D = \{D_1, \ldots, D_n\}$ *est un ensemble fini de domaines associés aux variables* $x_i \in D_i$ $(1 \leq i \leq n)$ *et* $C = \{c_1, \ldots, c_k\}$ *est un ensemble fini de contraintes (relations) sur des sous-ensembles de* X*. Une solution au CSP est un* $n-tuple$ $(v_1, \ldots, v_n)$ *de valeurs assignées aux variables de* X *qui satisfait (appartient à) chaque contrainte* c_j *dans* C*.*

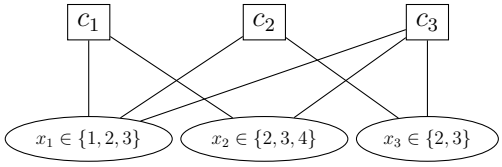


Figure 1.1 Réseau de contraintes de l'exemple 1.1

Exemple 1.1. Soient $X = \{x_1, x_2, x_3\}$, $D_1 = \{1, 2, 3\}$, $D_2 = \{2, 3, 4\}$ et $D_3 = \{2, 3\}$. Les contraintes sont $c_1 : x_1 + x_2 \leq 5$, $c_2 : x_1 + x_3 \geq 4$ et $c_3 : allDiff$ [1](x_1, x_2, x_3). En terme de relation on a respectivement $(v_1, v_2) \in \{(1,2),(1,3),(1,4),(2,2),(2,3),(3,2)\}$, $(v_1, v_3) \in \{(1,3),(2,2),(2,3),(3,2),(3,3)\}$ et $(v_1, v_2, v_3) \in \{(1,2,3),(1,3,2),(1,4,2),(1,4,3),(2,4,3),(3,4,2)\}$. Le réseau de contraintes correspondant à ce CSP est montré à la figure 1.1

Dans une recherche de solution avec choix de variable et de valeur selon l'ordre lexicographique (lexico), le solutionneur PPC commencera par assigner la valeur 1 à la variable x_1. Cette assignation sera propagée dans le réseau ce qui entraînera le "réveil" de la contrainte c_2, car la variable x_1 se trouve dans sa portée (figure 1.2(a)). Ainsi, la valeur 2 est retirée du domaine de la variable x_3. Cette modification dans le domaine de la variable x_3 déclenche à son tour une nouvelle propagation et la contrainte c_3 réagit pour retirer la valeur 3 du domaine de la variable x_2 (figure 1.2(b)). Le triplet $(v_1, v_2, v_3) = (1, 2, 3)$ est une solution possible.

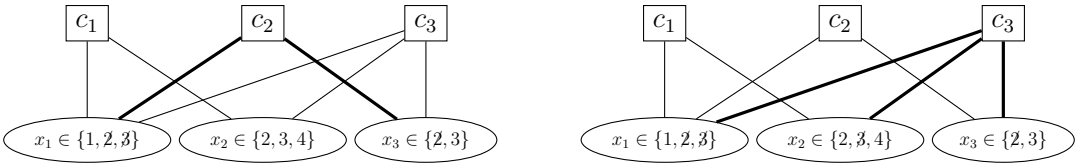


Figure 1.2 Propagation et réduction des domaines des variables du CSP de l'exemple 1.1

La recherche de solution est représentée par un arbre binaire qui mémorise les décisions de branchement qui ont été prises. Si un mauvais choix est fait, c.-à-d. a mené vers une impasse, alors le solutionneur fait retour arrière pour éliminer ce choix. La figure 1.3 montre un exemple de recherche de solution du CSP de l'exemple 1.1 en choisissant la variable avec la plus petite taille du domaine et la valeur est choisie aléatoirement parmi les valeurs dans le domaine. Cette heuristique correspond à une variante de l'heuristique de branchement

1. Indique que les valeurs des variables doivent être distinctes.

dom (Haralick and Elliott, 1980). Contrairement au premier cas, ici le solutionneur doit faire des retours arrière. Une bonne stratégie de recherche tente de trouver une solution avec un nombre minimum de retours arrière si le CSP est réalisable, sinon une meilleure stratégie serait celle permettant de réduire au maximum l'arbre de recherche pour prouver dans un temps raisonnable la non-réalisabilité du CSP.

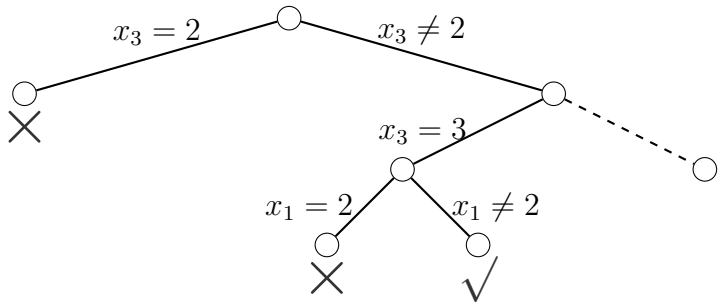


Figure 1.3 Exemple d'arbre de recherche pour le CSP de l'exemple 1.1

La stratégie de recherche est un aspect important de la programmation par contraintes. Pour la plupart des problèmes combinatoires, qui sont $\mathcal{NP}$-Difficiles, on ne connaît pas de stratégie de recherche qui trouve systématiquement, en temps polynomial, une solution si elle existe ou prouve la non-réalisabilité du CSP dans le cas contraire. Plusieurs heuristiques sont alors utilisées. Le choix de variable ou de valeur par lexico ou dom sont des stratégies parmi plusieurs qui sont intégrées dans la plupart des solutionneurs de programmation par contraintes. Il existe d'autres heuristiques génériques et robustes qui seront présentées en détail dans le chapitre 2. Il est aussi possible d'utiliser une heuristique conçue spécialement pour un problème qui exploite la structure particulière de ce dernier.

1.2 Problèmes d'ordonnancement disjonctif

L'ordonnancement est un domaine d'application parmi les plus étudiés en recherche opérationnelle. Il s'agit, en général, du processus de trouver un placement optimal des activités, tâches ou évènements dans le temps en respectant les ressources disponibles. Dans le cas où cette ressource est de capacité 1, alors il s'agit de problème d'ordonnancement disjonctif. Des exemples de ressources unaires peuvent être une piste d'atterrissage, un véhicule qui fait une tournée, une machine dans un atelier, etc. Cette ressource est "consommée" par un ensemble d'activités, chacune avec une plage de temps indiquant le plus tôt et le plus tard qu'elle peut commencer ; on parle d'exécution ou traitement des activités avec fenêtres de temps. Pour les situations décrites précédemment les activités seraient les avions, les endroits à visiter et les travaux à exécuter respectivement. Plusieurs versions d'optimisation de ce problème sont

considérées dans la littérature. Par exemple, on peut tolérer le début d'exécution d'une activité en dehors de sa fenêtre de temps, dans ce cas on s'intéresserait à minimiser les pénalités associées au retard maximum, la somme des retards ou autre. Dans la version de satisfaction de contraintes, les activités doivent commencer dans leurs fenêtres de temps respectives. De manière générale, chaque activité i aurait :

— temps de disponibilité e_i
— temps de fin l_i
— temps de traitement p_i

Les activités doivent être traitées sans interruption et ne doivent pas se chevaucher dans le temps. Décider s'il existe un ordonnancement tel que toutes les activités s'exécutent entièrement dans leurs fenêtres de temps respectives est équivalent à décider s'il existe un ordonnancement des activités avec temps de disponibilité tel que le retard maximum est égal à 0. Ce dernier est une instance de la version de décision du problème d'optimisation sur une seule machine avec objectif, la minimisation du retard maximum qui a été largement étudié et connu $\mathcal{NP}$-Difficile (Lenstra et al., 1977).

Exemple 1.2. Soit $A = \{1, \ldots, 6\}$, $(e_i)_{1 \leq i \leq |A|} = (6, 20, 10, 0, 18, 10)$, $(l_i)_{1 \leq i \leq |A|} = (34, 56, 40, 24, 42, 30)$ et $(p_i)_{1 \leq i \leq |A|} = (6, 10, 6, 10, 4, 10)$. Une solution à cet exemple est montrée à la figure 1.4.

En PPC, ce problème est considéré comme une contrainte plutôt qu'un problème à résoudre. Il s'agit d'une autre principale caractéristique de la PPC qui est d'utiliser des contraintes

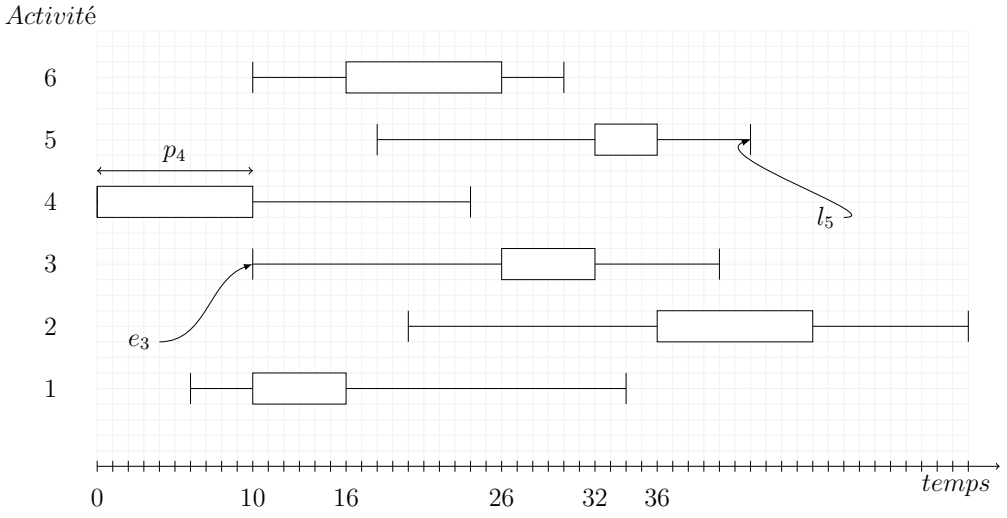


Figure 1.4 Une solution à l'exemple 1.2

globales. L'avantage est que cela permet d'encapsuler les petites contraintes associées au problème pour développer des algorithmes de filtrage puissants qui permettent une meilleure réduction des domaines des variables. Dans le cas de contrainte de ressource unaire, les variables de temps de départ des activités sont filtrées par des algorithmes conçus spécialement à cet effet. La détection de préséances entre activités est un exemple parmi plusieurs (voir Vilím, 2004). Dans sa version de base, cet algorithme vérifie pour chaque paire d'activités $i, j \in A$ si l'activité i peut être placée avant l'activité j, sinon, cette information est propagée et les domaines des temps de départ des activités subissent des réductions pour maintenir la cohérence des bornes. Par exemple, dans l'exemple 1.2, l'activité 6 doit s'exécuter avant l'activité 5 car le contraire n'est pas réalisable (figure 1.5). Ceci a comme conséquence de réduire la fenêtre de temps de départ de l'activité 5 par 2 unités. Cet algorithme s'exécute en temps polynomial, mais ne garantit pas la détection de toutes les préséances.

Une autre raison qui justifie la représentation du problème de ressource unaire par une contrainte unique est que beaucoup de problèmes du monde réel, en particulier les problèmes de planification, peuvent se modéliser sous forme d'une combinaison de ces contraintes. Le problème de tournée de véhicules en est un exemple. Dans ce cas, on aura des contraintes supplémentaires pour le temps de déplacement ou de capacité qui seront ajoutés au modèle. Un autre exemple de problème typique dont la structure est une combinaison de contraintes de ressources unaires est le problème d'ordonnancement dans un atelier à cheminements multiples (Job-Shop) qui sera décrit dans le chapitre 4.

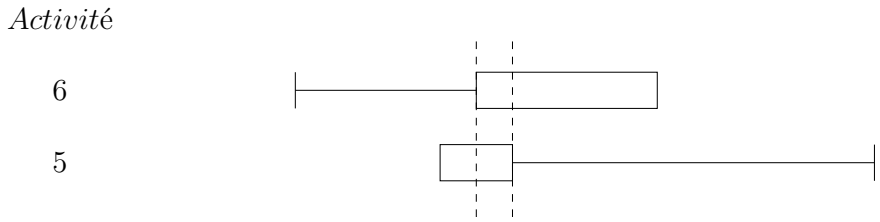


Figure 1.5 Exemple de détection de préséance

1.3 Objectifs de recherche

Pour la résolution des problèmes d'ordonnancement, il s'agit en général de décider des temps de départ des activités. Toutefois, pour plusieurs de ces problèmes notamment le problème Job-Shop, il est suffisant d'exprimer la solution en terme de permutations d'activités de chaque ressource, les temps de départ pouvant être déduits facilement. En utilisant un modèle disjonctif, trouver une solution passe par une série de décisions de la position relative d'une

activité i par rapport à une autre activité j. Dans ce cas, une solution partielle est un ordre partiel sur l'ensemble des activités de chaque ressource et une solution complète est une permutation de ces derniers. L'ordre partiel des activités est tout simplement un graphe orienté sans cycles nommé graphe de préséances dont les sommets sont les activités et les arcs (i, j) indiquent que i doit passer avant j.

L'objectif de cette recherche est de combiner un modèle disjonctif simple et la stratégie de recherche basée sur la densité des solutions maxSD, pour résoudre les problèmes d'ordonnancement disjonctif. La méthodologie utilisée consiste à développer un algorithme exact et heuristique pour calculer la proportion des permutations des activités dans le graphe de préséance associé à une ressource unaire dans lesquelles le rang d'une activité i est supérieur à celui de l'activité j qu'on appellera densité de permutations de la paire (i, j). En particulier, on propose un algorithme de mise à jour incrémentielle du nombre d'extensions linéaires d'un ordre partiel basé sur l'algorithme de programmation dynamique pour le cas exact. Les résultats empiriques montrent que la mise à jour incrémentielle est considérablement plus rapide que le recalcul du nombre d'extensions linéaires pour chacune des paires (i, j). Pour le cas heuristique, on propose deux algorithmes : Le premier sert à estimer le nombre d'extensions linéaires en se basant sur les résultats théoriques de la litérature mettant en lien l'entropie de graphe et le nombre d'extensions linéaires. Le deuxième algorithme heuristique sert à estimer la densité de permutations de la paire d'activités i et j en analysant le changement de la cardinalité de la relation d'ordre après ajout de la paire (i, j).

1.4 Plan du mémoire

Ce mémoire est organisé comme suit :

Dans le chapitre 2, on présente le contexte d'application de la stratégie de recherche basée sur le calcul de la densité de permutations. Ensuite, la revue de littérature sur les principales heuristiques de branchement est présentée, en particulier l'heuristique basée sur le dénombrement des solutions qui constitue le cadre de ce travail de recherche. La deuxième section de la revue de littérature sera consacrée aux algorithmes de dénombrement des permutations avec contraintes de préséances.

Le chapitre 3 est la contribution principale de ce travail de recherche. On y présente une adaptation de l'algorithme de programmation dynamique pour calculer de manière incrémentielle les densités de permutations des préséances dans un ensemble partiellement ordonné. Ensuite, on propose une heuristique pour estimer ces densités en se basant sur la structure du graphe de préséances. Un algorithme pour estimer le nombre de permutations par le calcul

de l'entropie d'un graphe est également présenté.

Les résultats expérimentaux sont présentés dans le chapitre 4. Les résultats de la performance en temps de calcul de l'algorithme exact ainsi que la précision du classement de l'algorithme heuristique sont présentés. En deuxième partie, on utilise un modèle simple pour le problème d'ordonnancement JSP combiné avec une stratégie de recherche basée sur la densité de solutions qui utilise les algorithmes présentés dans le chapitre 3. Les résultats obtenus sont comparés à ceux obtenus avec les autres heuristiques pour un même modèle.

La synthèse des travaux réalisés, les limitations ainsi que les améliorations futures sont présentées dans le dernier chapitre.

CHAPITRE 2 CONTEXTE ET REVUE DE LITTÉRATURE

Ce chapitre se divise en deux parties. Dans la première partie, on présente le cadre d'utilisation de l'heuristique basée sur le calcul de densité de permutations. La deuxième partie est consacrée à la revue de littérature. Tout d'abord, une revue des principales heuristiques de branchement en programmation par contraintes est présentée, en particulier l'heuristique basée sur le dénombrement de solutions CBS. La dernière section est consacrée à la revue de littérature sur les algorithmes exacts et approximatifs pour le calcul du nombre de permutations d'un graphe de préséances.

2.1 Mise en contexte

2.1.1 Représentation d'une ressource unaire

Soit la contrainte de ressource unaire définie sur l'ensemble des activités $A = \{1, 2, \ldots\}$ où chaque activité $i \in A$ a un temps de traitement p_i, le temps le plus tôt qu'elle peut commencer e_i ainsi que le temps le plus tard qu'elle peut terminer l_i. On cherche le temps de départ $t_i \in D_i$ de chaque activité $i \in A$ tel que $D_i = [e_i, l_i - p_i]$. Nous présentons deux modèles :

Modèle 1 : Les variables de rang $r_i \in \{1, \ldots, |A|\}$ sont déclarées pour chaque activité $i \in A$. La contrainte 2.1(a) garantit que les rangs prennent des valeurs distinctes. Le lien entre le rang et le temps de départ des activités est assuré par 2.1(b).

$$\begin{array}{lll} allDiff(r_i) & 1 \leq i \leq |A| & (a) \\ r_i < r_j \Leftrightarrow t_i + p_i \leq t_j & \forall i, j \in A,\ i \neq j & (b) \end{array} \tag{2.1}$$

Dans ce modèle, les variables de décision sont les rangs r_i. Le solutionneur PPC tentera d'assigner des valeurs à r_i pour chaque activité i pendant la recherche. Les variables de temps de départ des activités sont dépendantes. Selon la nature du problème, la taille de l'espace de recherche (produit cartésien des domaines des variables) et les contraintes supplémentaires, ce modèle peut être plus ou moins efficace.

Modèle 2 : Pour ce problème ainsi que plusieurs autres problèmes d'ordonnancement, trouver une permutation des activités est en général suffisant pour décrire une solution en

terme de temps de départ. En effet, si π est une permutation des activités de A, alors il peut être vérifié que si la solution

$$t_{\pi(k)} = \begin{cases} \max\{e_{\pi(k)}, t_{\pi(k-1)} + p_{\pi(k-1)}\} & 2 \leq k \leq |A| \\ e_{\pi(1)} & k = 1 \end{cases} \tag{2.2}$$

est non réalisable, alors aucune autre solution pour cette même permutation ne le sera. Si au contraire cette solution est réalisable, alors on a une solution. De plus, s'il s'agit d'un problème d'optimisation dont la fonction objectif est de minimiser le temps de complétion de la dernière activité traitée, alors la solution 2.2 est la meilleure solution correspondante à la permutation π. Dans ce deuxième modèle, chaque paire d'activités (i, j) est associée à une variable booléenne b_{ij} dont le domaine est $\{0, 1\}$ tel que

$$b_{ij} = \begin{cases} 0 \Rightarrow t_i + p_i \leq t_j \\ 1 \Rightarrow t_j + p_j \leq t_i \end{cases} \qquad \forall i, j \in A,\ i < j \tag{2.3}$$

Un aspect intéressant que procure le modèle 2 par rapport au modèle 1 est le fait qu'une solution partielle est un ordre partiel sur les activités qui peut être représenté par un graphe de préséances. Ce dernier n'est pas un concept nouveau. Le graphe de préséances associé à une ressource unaire est utilisé pour la propagation et le filtrage des domaines des variables de temps de départs des activités (Focacci et al., 2000; Vilım, 2007).

Définition 2.1. (graphe de préséance) *Soit le modèle 2.3, c une contrainte de ressource unaire et* $\{b_{ij}\}_{1 \leq i < j \leq |A|}$ *l'ensemble des variables dans la portée de c. Le graphe de préséances P associé à la contrainte c est défini par la paire* (A, R) *telle que* $R = \{(i, j) : b_{ij} = 0 \vee b_{ji} = 1\}$ *est l'ensemble des arcs du graphe. Chaque arc* $(i, j) \in R$ *représente une préséance entre une paire d'activités* (i, j)*.*

Le modèle 2 est appelé modèle disjonctif. Dans ce modèle, les variables de décision sont les variables booléennes b_{ij}. Le solutionneur PPC tente alors de trouver une solution si elle existe en décidant tour à tour, pour toutes les paires d'activités (i, j) non encore ordonnées, de placer l'activité i avant j par l'assignation $b_{ij} = 0$ ou bien $b_{ij} = 1$ pour j avant i. Ces opérations se traduisent par l'ajout de l'arc (i, j) pour $b_{ij} = 0$ ou l'arc (j, i) pour $b_{ij} = 1$ dans le graphe de préséances P. L'ajout d'arcs peut engendrer également de nouvelles préséances par la fermeture transitive. En résumé, les arcs dans le graphe de préséances associé à une contrainte de ressource unaire sont établis par :

— les décisions de branchement
— la détection de préséances

— la fermeture transitive

L'avantage d'utiliser le modèle 2 par rapport au modèle 1 est la possibilité d'exploiter le graphe de préséances pour guider la recherche en utilisant la densité de solutions comme heuristique de branchement. Les densités sont calculées par le dénombrement des permutations compatibles avec les contraintes de préséances du graphe associé à chaque ressource. Cette heuristique est décrite en détail dans la section 2.2.2

Plusieurs algorithmes de filtrage ont été développés pour les contraintes de ressource unaire notamment l'algorithme de détection de préséances (Vilim, 2004) vu dans le chapitre d'introduction. Il existe d'autres algorithmes tels que *Overload Checking*, *Edge Finding* et *Not-First/Not-Last* (Baptiste and Le Pape, 1996). Ces algorithmes sont surtout utilisés dans les modèles où les variables de décision sont les variables de temps de départ des activités et par conséquent sont considérés "lourds" par rapport au modèle disjonctif. La lourdeur de ces modèles est dû à la "surcharge" de la recherche de solution par les différents algorithmes de filtrage mentionnés ci-haut.

2.1.2 Contraintes dénombrables

Dans le chapitre précédent, on a vu qu'une contrainte est simplement une relation sur un sous-ensemble de variables du CSP. Le dénombrement de solutions associées à cette contrainte est défini par :

Définition 2.2. (dénombrement de solutions (Pesant, 2005)). *Soit* $c(x_1 \dots x_n)$ *une contrainte et* $\{D_i\}_{1 \leq i \leq n}$ *un ensemble fini de domaines respectifs. On note* $\#c(x_1, \dots x_n)$ *le nombre de* $n-$*tuples dans la relation correspondante appelé* dénombrement de solutions.

Plusieurs contraintes disposent d'algorithmes pour le dénombrement de solutions (Pesant and Quimper, 2008; Zanarini and Pesant, 2009; Pesant et al., 2012; Brockbank et al., 2013). Certains sont exacts et d'autres approximatifs. Pour la contrainte de ressource unaire, compter le nombre de solutions est au moins aussi difficile que compter le nombre de permutations. Toutefois, pour le dénombrement des permutations, il s'agit d'un problème connu pour lequel des algorithmes approximatifs existent. En exploitant le graphe de préséances associé à la contrainte, on peut doter la contrainte avec un algorithme dédié qui calcule le nombre de permutations compatibles avec les contraintes de préséances. La distinction entre le nombre de permutations et le nombre de solutions d'une contrainte à ressource unaire est que chaque solution correspond à une et une seule permutation alors qu'une permutation désigne plusieurs solutions. Par exemple, la permutation associée à la solution présentée à la figure 2.1 est $4-1-6-3-5-2$, alors que plusieurs solutions sont possibles pour cette même per-

mutation. Pour obtenir d'autres solutions pour une même permutation π des activités de A, il suffit de déplacer les activités dans leurs intervalles de liberté $\Delta_i = t'_i - t_i$ où t_i est définie par 2.2 et t'_i est définie comme suit :

$$t'_{\pi(k)} = \begin{cases} \min\{l_{\pi(k)}, t'_{\pi(k+1)}\} - p_{\pi(k)} & 1 \leq k < |A| \\ l_{\pi(|A|)} - p_{\pi(|A|)} & k = |A| \end{cases} \tag{2.4}$$

en choisissant l'activité $j = \underset{i \in A}{\operatorname{argmin}} \Delta_i$, on obtient l'ensemble des solutions associées à une permutation π en déplaçant chacune des activités i par Δ_j. Bien entendu, il ne s'agit pas de toutes les solutions possibles pour la permutation π. Un exemple est montré à la figure 2.1.

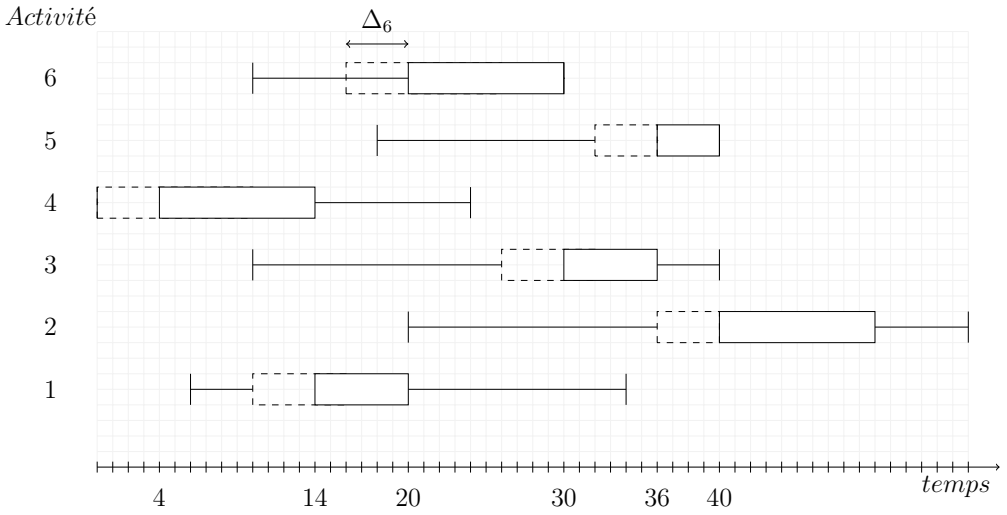


Figure 2.1 Exemple de solutions associées à une permutation

Plusieurs problèmes d'ordonnancement ou de tournée de véhicules peuvent être modélisés sous forme d'un ensemble de contraintes de ressources unaires reliées entre elles par des contraintes supplémentaires telles que les contraintes de préséances, de restriction de rangs ou de séquences. Un exemple classique est le problème Job-Shop qui sera décrit plus en détail dans le chapitre 4. Pour ces problèmes, on considère le modèle disjonctif 2.3. Le graphe de préséances entre activités associées à chaque ressource unaire est exploité pour guider la recherche en utilisant l'heuristique basée sur le calcul de densités de solutions. L'intuition derrière est de dire que si pour une paire d'activités (i, j), la proportion de permutations dans lesquelles l'activité i se trouve avant j est suffisamment élevée alors, si une solution existe, fort probablement que i sera ordonnée avant j.

2.2 Revue de littérature

2.2.1 Heuristiques de branchement

Les heuristiques de branchement sont généralement classées selon deux catégories : statiques SVO (Static Variable Ordering) et dynamiques DVO (Dynamic Variable Ordering). Comme leur nom l'indique, les heuristiques de la première catégorie établissent l'ordre de choix des variables de branchement avant la recherche et ce choix n'est jamais affecté pendant la recherche. Un exemple est le choix de variables selon l'ordre lexicographique vu dans le chapitre 1. La famille d'heuristiques DVO est considérée plus efficace, car on y exploite l'information collectée pendant la recherche. Le classement de variables pour cette catégorie suit généralement le principe du "échouer d'abord" (*first-fail principle*) introduit par Haralick and Elliott (1980). L'idée est de se dire "pour réussir, brancher là où on a le plus de chance d'échouer". Un exemple de cette famille est le choix de variable avec le plus petit domaine dom vu dans le chapitre 1. D'autres heuristiques plus élaborées ont été proposées pour rendre la stratégie de recherche plus robuste. Parmi ces heuristiques, IBS(*Impact-Based Search*), WDEG(*Weighted Degree*) et CBS(*Counting-Based Search*) qui constituent l'état de l'art dans ce domaine.

Heuristique basée sur l'impact (IBS) proposée par Refalo (2004), cette heuristique choisit la variable x_i qui déclenche la plus grande réduction dans l'espace de solutions suite à l'assignation d'une valeur d dans son domaine D_i. Une fois la variable choisie, la valeur dans le domaine de cette variable ayant causé le plus petit impact est sélectionnée. La réduction de l'espace de solutions est estimée par le ratio du produit cartésien des domaines des variables avant (P_{avant}) et après ($P_{après}$) l'assignation $x_i = d$. L'impact de la paire (x_i, d) est défini par :

$$I(x_i = d) = 1 - \frac{P_{apres}}{P_{avant}}$$

Le calcul exact de l'impact d'une variable x_i nécessite le calcul de l'impact de l'ensemble des assignations de x_i aux valeurs dans son domaine pour en calculer la moyenne. De plus ce travail doit se faire à chaque nœud de l'arbre de recherche et cela pour toutes les variables non instanciées. Cette approche devient très coûteuse et est remplacée plutôt par le calcul de la moyenne des impacts associés aux assignations qui ont été observées (branchements) jusqu'à un certain niveau de l'arbre de recherche. Ceci est basé sur l'idée que l'impact d'une variable ne changera pas beaucoup d'un nœud à un autre. Par contre, une étape d'initialisation est nécessaire à la racine de l'arbre et ensuite les impacts des variables sont mis à jour après chaque décision de branchement.

Heuristique basée sur le degré (DDEG) proposée par Brélaz (1979), cette heuristique consiste à choisir la variable avec le plus petit domaine et ensuite de briser l'égalité en choisissant la variable x ayant le plus grand degré dynamique ddeg. Le degré dynamique de x est le nombre de variables non instanciées dans la porté des contraintes où x appartient. Bessiere and Régin (1996) et Smith and Grant (1997) combinent dom et ddeg en minimisant le ratio dom/ddeg.

Heuristique basée sur le conflit (WDEG) proposée par Boussemart et al. (2004), cette heuristique utilise une technique d'apprentissage permettant de prioriser les variables dans la portée des contraintes qui causent le plus d'échecs. Au départ, chaque contrainte c possède un poids initialisé 1. À chaque variable x_i est associé un degré pondéré α_{wdeg}-wdeg- qui est la somme des poids des contraintes auxquelles elle appartient. Le degré pondéré de la variable x_i est calculé par :

$$\alpha_{wdeg}(x_i) = \sum_{c \in C} poids[c] \quad | \quad x_i \in Vars(c) \wedge |FutVars(c)| > 1$$

où $poids[c]$ est le poids de la contrainte c, $Vars(c)$ est l'ensemble des variables dans la portée de c et $FutVars(c) \subseteq Vars(c)$ indique l'ensemble des variables non instanciées. À chaque fois qu'une contrainte est impliquée dans un échec, son degré est incrémenté de 1. En choisissant de brancher sur la variable qui maximise $\alpha_{wdeg}(x_i)$ ou qui minimise dom/wdeg, l'heuristique tend à choisir les variables les plus contraintes.

2.2.2 Heuristique basée sur le dénombrement

Proposée par Zanarini and Pesant (2009) cette heuristique exploite l'information obtenue à partir du nombre de solutions associées à une contrainte. L'idée de départ est de considérer les contraintes avec peu de solutions comme étant critiques par rapport à la satisfiabilité de tout le CSP. Cette idée a été généralisée pour chaque variable dans la contrainte en calculant la proportion de solutions dans lesquelles la variable x_i est assignée à une valeur d de son domaine appelée densité de solutions de l'assignation $x_i = d$.

Définition 2.3. (densité de solution (Zanarini and Pesant, 2009)) *Soit $c(x_1 \dots x_k)$ une contrainte et $\{D_i\}_{1 \leq i \leq k}$ un ensemble fini de domaines respectifs, x_i une variable dans la portée de c et une valeur $d \in D_i$. On appelle*

$$\sigma(x_i, d, c) = \frac{\#c(x_1, \dots, x_{i-1}, d, x_{i+1}, \dots x_k)}{\#c(x_1, \dots, x_k)}$$

la densité de solution de la paire (x_i, d)*. Elle indique la proportion de solutions de* c *avec* $x_i = d$*.*

L'heuristique maxSD dérive de la définition 2.3. Dans cette heuristique, les choix de branchement sont établis en calculant, pour chaque contrainte c, la densité de solutions de toutes les assignations de valeurs aux variables dans la portée de c non encore instanciées. L'assignation de densité maximum est choisie (algorithme 1).

```
1 max ← 0;
2 pour chaque contrainte c(x_1, ..., x_k) faire
3     pour chaque variable non instanciée x_i ∈ {x_1, ..., x_k} faire
4         pour chaque valeur d ∈ D_i faire
5             si σ(x_i, d, c) > max alors
6                 (x*, d*) = (x_i, d);
7                 max = σ(x_i, d, c);
8 retourner la décision de branchement x* = d*;
```

Algorithme 1 : Heuristique de branchement basée sur la densité (maxSD)

Il existe d'autres variantes de l'heuristique basée sur le dénombrement. Par exemple, on peut décider de choisir la contrainte ayant le plus petit nombre de solutions et ensuite choisir la variable dans la portée de cette contrainte qui maximise la densité de solutions. Une autre possibilité est de choisir la variable avec le plus petit domaine et ensuite, pour toutes les contraintes auxquelles appartient la variable, choisir la valeur qui maximise la densité de solutions. Les données empiriques actuelles indiquent que la performance de maxSD est, dans l'ensemble, au moins aussi bonne que celle de ces variantes (Pesant et al., 2012).

2.2.3 Algorithmes de dénombrement

Algorithmes exacts soit un graphe de préséances $P = (S, R)$ où $S = \{1, \ldots, n\}$ est l'ensemble des sommets et $R \subseteq S \times S$ est l'ensemble de préséances ou arcs dans P. On suppose que R est fermé transitivement et sans cycle. Soit π une permutation des sommets de S telle que pour toute paire $(i, j) \in R$ on a $\pi^{-1}(i) < \pi^{-1}(j)$. On note $\#P(S, R)$ ou simplement $\#P(S)$, le nombre total de ses permutations. Calculer $\#P(S)$ est connu pour être $\#\mathcal{P}$-Complet (Brightwell and Winkler, 1991), la version de dénombrement des problèmes $\mathcal{NP}$-Complets. Pruesse and Ruskey (1994) proposent un algorithme qui génère toutes les permutations du graphe P en temps constant amorti par rapport au nombre de permutations. Leur stratégie est basée sur l'idée que pour les problèmes de génération dans un ensemble, les objets combinatoires

successifs diffèrent seulement par des petites variations bien définies. Dans le cas des permutations de P, les variations sont faites par une ou deux transpositions en partant d'une permutation initiale. Une adaptation de l'algorithme pour calculer la proportion de permutations dans lesquelles un sommet i est placé avant j est donnée en temps $\mathcal{O}(n^2 + \#P)$. Cet algorithme est considéré par ses auteurs comme étant "rapide". En effet, il l'est pour la génération des permutations, mais pas pour le cas où on cherche uniquement à en connaître le nombre d'autant plus que ce nombre peut avoisiner $n!$.

Un algorithme de programmation dynamique pour calculer $\#P(S)$ est connu (Edelman et al., 1989). Cet algorithme exploite la propriété d'auto-réductibilité du problème de dénombrement. En effet, si on choisit le premier sommet de la permutation $\pi(1)$, alors le problème se réduit à trouver les permutations des sommets à placer dans $\pi(2), \ldots, \pi(n)$. La formule récursive utilisée par le programme dynamique est donnée par :

$$\#P(S) = \begin{cases} \sum_{i \in MIN(S)} \#P(S \setminus \{i\}) & S \neq \emptyset \\ 1 & S = \emptyset \end{cases} \tag{2.5}$$

où $MIN(S)$ est l'ensemble des sommets de P n'ayant pas d'arcs entrants dans S. L'idée est basée sur le fait que toutes les permutations de P commencent nécessairement par un des sommets dans $MIN(S)$. Le nombre total des permutations est la somme de toutes les permutations commençant par chacun des sommets dans $MIN(S)$. Par la propriété de réductibilité, les sous-problèmes sont résolus récursivement. En inversant les paires (i, j) dans R, les éléments maximaux $MAX(P)$ de P n'ayant pas d'arcs sortants dans S, deviennent minimaux. Ainsi la formule 2.5 s'applique récursivement pour les sous-graphes subséquents. Stachowiak (1988) utilise cette observation pour montrer que

$$2 \times \#P(S) = \sum_{i \in MIN(S) \cup MAX(S)} \#P(S \setminus \{i\}) \tag{2.6}$$

En mots, on peut choisir soit d'appliquer la formule 2.5 à travers les éléments minimaux ou maximaux de P et la somme est égale à $2 \times \#P$.

Les deux formules récursives 2.5 et 2.6 sont théoriques et à notre connaissance, aucune implémentation n'a été proposée. De plus, l'application de cette formule après chaque ajout d'un arc dans le graphe n'est pas efficace. Idéalement, on préférerait exploiter la table des valeurs calculées par l'algorithme de programmation dynamique pour la mise à jour incrémentielle du nombre de permutations après l'ajout d'un arc.

Une récurrence similaire a été utilisée par Held and Karp (1962) pour résoudre différents pro-

blèmes de séquencement, en particulier le problème de voyageur de commerce et le problème d' équilibrage de la ligne d'assemblage (*Assembly-line balancing problem*). La similarité se trouve dans le fait que l'espace des états généré par la récurrence proposée initialement par Held and Karp et l'espace des états générés par 2.5 est le même. Dans leur article, Held and Karp utilisent le terme "sous-ensemble réalisable" pour désigner ces états. Dans la notation contemporaine, il s'agit d'un bas-ensemble ou plus connue par le terme anglais *downset* de P. Plus récemment, Cire et al. (2012) utilisent la même récursion pour compiler un diagramme appelé *MDD permutation*. Ce diagramme a des usages multiples. Il sert entre autres, comme représentation compacte de l'espace des états ou de manière équivalente de l'espace des solutions d'une contrainte disjonctive. Il est utilisé également pour représenter les domaines des variables. Vu que la taille de l'espace des états ou noeuds dans le diagramme tend à devenir exponentielle, une version "relâchée" du *MDD* est utilisée. La relaxation est basée sur des heuristiques qui ajoutent progressivement les états par raffinement.

Algorithmes approximatifs puisque le calcul de $\#P(S)$ a été démontré difficile, les recherches se sont concentrées sur la conception d'algorithmes approximatifs. Banks et al. (2010) propose un algorithme utilisant la technique TPA (Tootsie Pop Algorithm). Cet algorithme est utilisé pour estimer la taille d'un ensemble par une série d'échantillonnages jusqu'à atteindre une cible (un objet dans l'ensemble). Chaque fois qu'un objet est choisi, l'ensemble est réduit au maximum tout en s'assurant que l'objet y demeure. Le nombre de tentatives nécessaires est une variable aléatoire avec une distribution de *Poisson* dont le paramètre est le logarithme naturel de la taille de l'ensemble. La distribution sert à estimer la taille par le nombre de tentatives. Ce processus est répété plusieurs fois pour en augmenter la précision. Pour que cet algorithme soit utilisable dans le problème de dénombrement des permutations, il est nécessaire de transformer l'ensemble des permutations, qui est un ensemble discret, en un ensemble continu mesurable. Ceci est réalisé en faisant une correspondance entre un vecteur x de réels $x \in (0, n]^n$ et une permutation π par le calcul de plafond $\lceil x(i) \rceil = \pi(i),\ 1 \leq i \leq n$. De plus, la mesure de Lebesgue de l'ensemble des vecteurs x qui mappent une même permutation est 1. Ainsi, la mesure de l'ensemble des vecteurs qui correspondent à l'ensemble des permutations est égale au nombre de permutations. En ayant un espace mesurable représentant l'espace de permutations, l'utilisation de l'algorithme TPA est rendue possible. L'algorithme donne une approximation avec un facteur $1 + \epsilon$ et une probabilité $1 - \delta$ pour un nombre d'opérations élémentaires d'au plus $\mathcal{O}(n^3(ln(n)(ln(\#P))^2\epsilon^{-2}ln(\delta^{-1})))$.

Un autre axe de recherche sur les algorithmes d'approximations pour ce problème de dénombrement exploite un résultat important trouvé par Stanley (1986) faisant le lien entre le nombre de permutations de P est le volume de corps convexes. Soit l'ensemble $\mathbb{R}^A$ de toutes

les fonctions $f : A \to \mathbb{R}$. Stanley propose le polytope de l'ordre (Ordre Polytope) du graphe G, $\mathcal{O}(G)$ étant un sous-ensemble de $\mathbb{R}^A$ défini par :

$$\begin{aligned} 0 \leq f(x) \leq 1 \quad & \forall x \in S \\ f(x) \leq f(y) \quad & \forall (x, y) \in R \end{aligned}$$

On note que si aucun arc ne se trouve dans P, alors le volume du polytope est égal à 1. Stanley démontre que

$$vol(\mathcal{O}(P)) = \frac{\#P(S)}{|S|!}$$

L'interprétation géométrique est que chaque permutation correspond à un simplexe dans $\mathcal{O}(P)$ dont le volume est $1/|S|!$. Une des conséquences de ce résultat est que les approches utilisées dans l'approximation des volumes convexes deviennent utilisables pour l'approximation du nombre de permutations de P.

En lien avec les volumes de corps convexes, Kahn and Kim (1995) démontrent que le logarithme du nombre de permutations est dans l'ordre exact de l'entropie H, tel que définie par Körner (1973), du graphe complément de P,

$$log(\#P(S)) \in \Theta(nH(\overline{P}))$$

Le calcul de $H(P)$ est un problème de programmation convexe qui peut être résolu en temps polynomial. Néanmoins, calculer l'entropie exactement demeure coûteux. Une approximation de l'entropie de P a été proposée par Cardinal et al. (2013) qui consiste à calculer l'entropie de Shannon de la distribution obtenue à partir d'une décomposition en chaînes vorace du graphe P. L'algorithme proposé retire à chaque itération la chaîne de taille maximum dans le graphe. Cependant, si plusieurs chaînes de même taille sont candidates, alors l'algorithme choisit une chaîne de manière arbitraire ou selon l'ordre lexicographique. Ceci a comme impact de diminuer les chances d'extraire des chaînes de plus grandes tailles dans les itérations suivantes et par conséquent influence la valeur de l'entropie calculée qui tend à être plus grande et donc moins précise. L'algorithme proposé est dans l'ordre de $\mathcal{O}(|P|^{2.5})$.

CHAPITRE 3 ALGORITHMES POUR LES DENSITÉS DE PERMUTATIONS

Dans un ensemble partiellement ordonné P si deux éléments $i, j \in P$ sont incomparables, alors la proportion de permutations $\sigma(i, j)$ dans lesquelles le rang de i est inférieur au rang de j est appelée densité de permutations de la paire (i, j). Dans un contexte de branchement utilisant l'heuristique maxSD, les densités doivent être calculées pour l'ensemble des paires incomparables de P dont le nombre est dans l'ordre de $\mathcal{O}(|P|^2)$ et cela pour chaque contrainte de ressource unaire. Dans le cas d'un algorithme de dénombrement exact, et partant du fait qu'un tel algorithme serait exponentiel dans le cas général, l'utilisation d'un algorithme incrémentiel est nécessaire pour garantir un seuil de performance acceptable jusqu'à une certaine taille de P, à partir de laquelle l'utilisation d'un algorithme d'approximation s'impose. En général, puisque la densité d'une paire $\sigma(i, j)$ est un rapport du nombre de permutations de P avant et après l'ajout de la paire (i, j) à la relation d'ordre associée à P, une approximation de la densité passera par l'approximation du nombre de permutations.

Dans ce chapitre, on propose une implémentation de l'algorithme de programmation dynamique par une structure de données qui permet de calculer les densités de permutations de toutes les paires non comparables dans P par mise à jour incrémentielle de la table de valeurs calculées par l'algorithme DP. Ensuite, on propose une heuristique qui donne une estimation des densités de permutations par l'analyse de la structure du poset. Pour avoir une estimation du nombre de permutations, on propose un algorithme qui se base sur les résultats de Kahn and Kim (1995) et Cardinal et al. (2013)

3.1 Définitions

Pour ce chapitre, les termes graphe de préséance et permutation seront remplacés par une autre terminologie équivalente : respectivement ensemble partiellement ordonné et extension linéaire. D'autres définitions, principalement de la théorie des ensembles ordonnés, seront utilisées. En voici quelques-unes :

Définition 3.1. (ensemble partiellement ordonné). Un ensemble partiellement ordonné (ou poset pour *Partially Ordered Set*) est une paire $P = (S, R)$ composée d'un ensemble $S = \{1, \ldots, n\}$ et une relation binaire R transitive, irréflexive et asymétrique. Deux éléments i, j de S sont comparables si $(i, j) \in R$ ou $(j, i) \in R$ sinon ils sont incomparables. Par abus de notation, P est également utilisé pour désigner l'ensemble S.

Définition 3.2. (extension linéaire) une extension linéaire est un ordre total (permutation) des éléments de P qui respecte l'ordre partiel défini par R.

Définition 3.3. (Chaîne et antichaîne) Un ensemble $C \subseteq P$ est une chaîne si $\forall i, j \in C$ alors i et j sont comparables. Si la cardinalité de C est maximum, alors la valeur de $|C|$ indique la hauteur de P. Par opposition, A est une antichaîne si $\forall i, j \in A$ alors i et j sont incomparables. Si la cardinalité de A est maximum, alors la valeur de $|A|$ indique la largeur de P. On considère que tous les ensembles singletons sont des chaînes et des antichaînes.

Définition 3.4. (bas-ensemble et haut-ensemble). Un ensemble $L \subseteq S$ est un bas-ensemble si $\forall i \in L$, $\forall j \in S$ si $(j, i) \in R$ alors $j \in L$. L'ensemble de tous les bas-ensembles de P est noté $\mathcal{L}$, en particulier $L[i] \in \mathcal{L}$ tel que $L[i] = \{j \in S : (j, i) \in R\} \cup \{i\}$ est le bas-ensemble de l'élément i. Similairement, un ensemble $U \subseteq S$ est un haut-ensemble si $\forall i \in U$,$\forall j \in S$ si $(i, j) \in R$ alors $j \in U$. L'ensemble de tous les haut-ensembles de P est noté $\mathcal{U}$, en particulier $U[i] \in \mathcal{U}$ tel que $U[i] = \{j \in S : (i, j) \in R\} \cup \{i\}$ est le haut-ensemble de l'élément i.

Définition 3.5. (ensemble minimal et maximal). Les ensembles minimaux et maximaux de P sont définis respectivement par $MIN(P) = \{i : i \in S,\ L[i] = \{i\}\}$ et $MAX(P) = \{i : i \in S,\ U[i] = \{i\}\}$.

Exemple 3.1. Soit le poset P tel que $S = (\{1, \ldots, 6\}$ et la relation d'ordre $R = \{(1, 2), (4, 2), (4, 3), (4, 5), (4, 6), (6, 2), (6, 5)\}$. Une extension linéaire de P est $4-3-6-1-5-2$. L'ensemble $C = \{4, 5, 6\}$ est une chaîne et est de taille maximum. La hauteur de P est $h = |C| = 3$. L'ensemble $A = \{1, 3, 6\}$ est une antichaîne et est de taille maximum. La largeur de P est $k = |A| = 3$. L'ensemble $L = \{3, 4, 5, 6\}$ est un bas-ensemble et $L[2] = \{1, 2, 4, 6\}$. L'ensemble $U = \{2, 5, 6\}$ est un haut-ensemble et $U[2] = \{2\}$. Notons que le bas-ensemble $L = \{3, 4, 5, 6\}$ n'est le bas-ensemble d'aucun élément de P.

Un poset P est généralement donné dans une forme réduite, c.-à-.d. qu'on conserve uniquement les paires résultantes de la réduction transitive de la relation R. Le sous-ensemble $R' \subseteq R$ est réduction transitive de R si la fermeture transitive de R' est égale à R et R' est de cardinalité minimum. Un élément j "couvre" i dans P si $(i, j) \in R'$ et R' est appelée relation de couverture pour le terme anglais *cover relation*. Pour visualiser un poset, on utilise un graphe orienté sans cycle tel que montré à la figure 3.1.

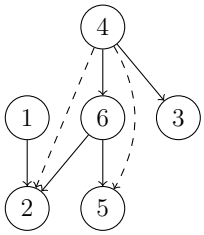


Figure 3.1 Poset de l'exemple 3.1 donné par un graphe orienté sans cycle. Dans sa forme réduite, les paires dont les arcs sont montrés en tirets seront retirés par réduction transitive.

3.2 Algorithme exact

3.2.1 Dénombrement des extensions linéaires

Soit $P(S, R)$ un poset et $i, j \in S$ deux éléments incomparables de P : on sait que $\#P(S, R \cup (i, j)) + \#P(S, R \cup (j, i)) = \#P(S)$. Le calcul du nombre d'extensions linéaires se réduit donc au calcul du nombre d'extensions linéaires de poset plus "faciles". Cette opération est répétée jusqu'à obtention d'un poset dont les éléments S forment une chaîne ou une antichaîne. Dans le premier cas on a $\#P(S) = 1$ et dans le deuxième $\#P(S) = |S|!$. En choisissant minutieusement deux éléments i et j dans une antichaîne de P on peut concevoir un algorithme de la famille *diviser-pour-régner* pour le problème de dénombrement.

Exemple 3.2. On considère le poset de l'exemple 3.1 : En ajoutant la paire $(4, 1)$ on obtient un poset P' tel que $MIN(P') = \{4\}$. Ainsi, le retrait de l'élément $\{4\}$ n'aura pas d'impact sur le nombre des extensions linéaires de P'. En retirant cet élément, le poset se décompose en un élément isolé $\{3\}$ et un poset dont les éléments, sont $\{1, 2, 5, 6\}$. Le nombre d'extensions linéaires de P' est donc $\#P'(S) = (|\{1, 2, 5, 6\}| + 1) \times \#P(\{1, 2, 5, 6\})$ où le premier terme correspond aux insertions possibles de l'élément $\{3\}$ dans une permutation du reste. Pour l'étape suivante, on choisit la paire $(1, 6)$. Maintenant en ajoutant la paire $(1, 4)$, les éléments $\{1\}$ et $\{4\}$ seront retirés successivement, car ils deviennent les seuls éléments minimaux. Le poset résultant sera aussi décomposé et on obtiendra une première antichaîne $\{2, 5\}$ après retrait du seul élément minimal $\{6\}$. Les détails du calcul sont montrés à la figure 3.2.

Il existe d'autres façons d'exploiter la propriété d'auto réductibilité du problème de dénombrement des extensions linéaires d'un poset. Par exemple, pour un élément i donné, notons $h_p(i)$ le nombre d'extensions linéaires dans lesquelles l'élément i se trouve à la position p ; il est clair que $\#P(S) = \sum_{1 \leq p \leq |S|} h_p(i),\ \forall i \in S$ [1]. Pour ramener le problème en

1. Cette définition apparaît dans Biró and Trotter (2011) sous le nom de *Height Sequence* où ils donnent une preuve combinatoire que la séquence $(h_p(i))_{1 \leq p \leq |S|}$ est log-concave.

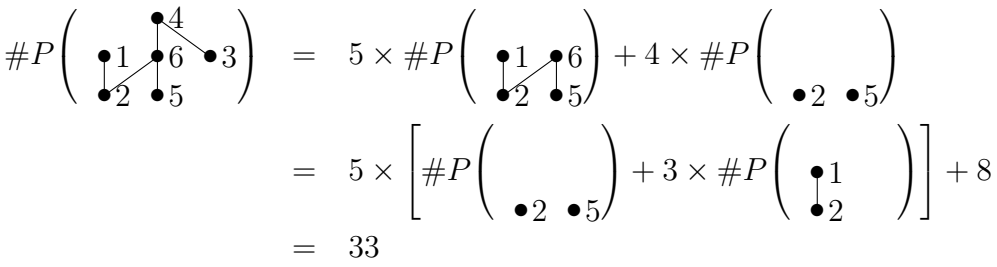


Figure 3.2 Calcul du nombre d'extensions linéaires du poset de l'exemple 3.1

sous-problèmes plus petits, on utilise plutôt le dual, c.-à-d. on fixe la position p et on fait varier les éléments $i \in S$ pouvant se trouver à la position p. Encore une fois, il est clair que $\#P(S) = \sum_{i \in S} h_p(i),\ \forall 1 \leq p \leq |S|$. Maintenant, supposant qu'on a une position p tel qu'il existe un seul élément i qui peut s'y retrouver, alors on peut vérifier facilement que $h_p(i) = \#P(L[i]) \times \#P(U[i])$ et donc $\#P(S) = h_p(i)$. Dans le cas général on aura

$$h_p(i) = \sum_{L \in \mathcal{L}, |L| = p-1, i \notin L, S \setminus (L \cup \{i\}) \in \mathcal{U}} \#P(L) \times \#P(S \setminus (L \cup \{i\})) \tag{3.1}$$

Exemple 3.3. Soit le poset de l'exemple 3.1. Les éléments $i \in \{1, \ldots, 6\}$ pouvant se trouver à la position $p = 3$ sont tels que $|L[i]| \leq 3$ et $|U[i]| \leq |S| - 3 + 1$. On a

$i = 1,\ L[1] = \{1\},\ U[1] = \{1, 2\}$
$i = 2,\ L[2] = \{1, 2, 4, 6\},\ U[2] = \{2\}$
$i = 3,\ L[3] = \{3, 4\},\ U[3] = \{3\}$
$i = 4,\ L[4] = \{4\},\ U[4] = \{2, 3, 4, 5, 6\}$
$i = 5,\ L[5] = \{4, 5, 6\},\ U[5] = \{5\}$
$i = 6,\ L[6] = \{4, 6\},\ U[6] = \{2, 5, 6\}$

Ses éléments sont {1,3,5,6}. D'après 3.1, on a

$h_3(1) = \#P(\{3, 4\}) \times \#P(\{2, 5, 6\}) + \#P(\{4, 6\}) \times \#P(\{2, 3, 5\}) = 1 \times 2 + 3! = 8$
$h_3(3) = \#P(\{4, 6\}) \times \#P(\{1, 2, 5\}) + \#P(\{1, 4\}) \times \#P(\{2, 5, 6\}) = 1 \times 3 + 2 \times 2 = 7$
$h_3(5) = \#P(\{4, 6\}) \times \#P(\{1, 2, 3\}) = 1 \times 3 = 3$
$h_3(6) = \#P(\{1, 4\}) \times \#P(\{2, 3, 5\}) + \#P(\{3, 4\}) \times \#P(\{1, 2, 5\}) = 2! \times 3! + 1 \times 3 = 12 + 3 = 15$

On trouve $\#P(\{1, \ldots, 6\}) = \sum_{i \in \{1,3,5,6\}} h_3(i) = 8 + 7 + 3 + 15 = 33$.

Si on choisit $p = 1$, les éléments i pouvant se trouver au premier rang des extensions linéaires sont les éléments de l'ensemble minimal de P. Puisque $L = \emptyset$ est le seul bas-ensemble tel que $|L| = 1 - 1 = 0$ alors la sommation disparaît. De plus, on a $\#P(\emptyset) = 1$ et $S \setminus (\emptyset \cup \{i\}) \in \mathcal{U}$

(définition 3.4). L'équation 3.1 devient $h_1(i) = \#P(S \setminus \{i\}),\ \forall i \in MIN(P)$ et

$$\#P(S) = \begin{cases} \sum\limits_{i \in MIN(P)} h_1(i) & S \neq \emptyset \\ 1 & sinon \end{cases} \tag{3.2}$$

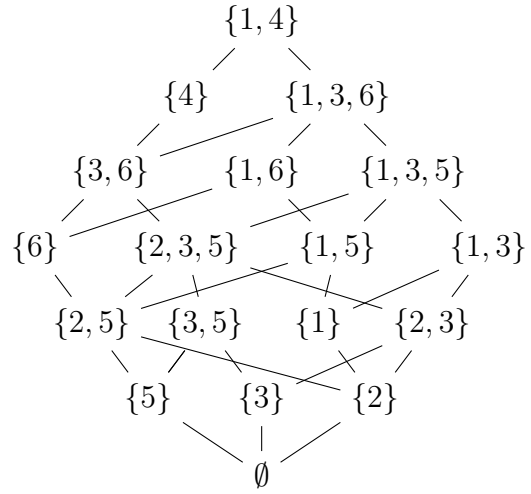
On remarque qu'on se ramène à la récursion 2.5. L'avantage de la généralisation 3.1 ainsi que l'approche utilisée dans l'exemple 3.2 est de lever le voile sur les possibilités de concevoir un algorithme pour le dénombrement des extensions linéaires d'un poset se basant sur le patron de conception *diviser-pour-régner*. Un tel algorithme serait probablement plus rapide que l'algorithme de programmation dynamique, car il utiliserait plusieurs "raccourcis" pour réduire le temps de calcul. Cependant, si l'objectif est de calculer les densités de toutes les paires incomparables du poset, alors l'algorithme de programmation dynamique est mieux adapté, car on aura besoin de toutes les valeurs intermédiaires pour la mise à jour incrémentielle. Dans la section 3.2.3, on explique comment on peut exploiter le résultat 3.1 dans la mise à jour du nombre d'extensions linéaires.

3.2.2 Compilation de l'algorithme de programmation dynamique

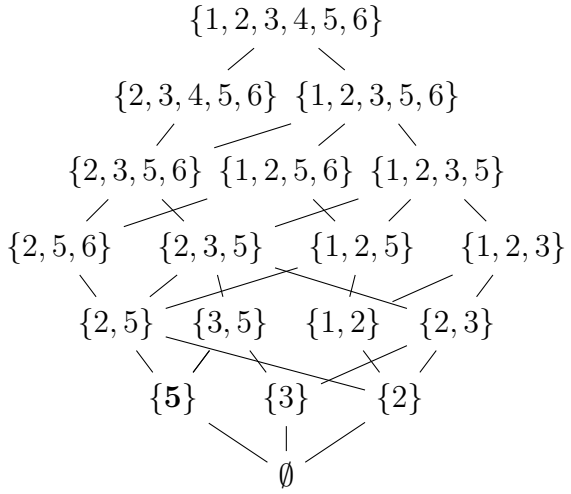
La compilation du programme dynamique consiste à générer et mémoriser l'espace des états dans une structure de données. D'après 3.2, l'algorithme commence à partir de l'ensemble des éléments minimaux $MIN(P)$ du poset P qu'on choisit comme état de départ. Selon les définitions 3.3 et 3.5, $MIN(P)$ est une antichaîne. À partir de l'état de départ, l'algorithme transite à travers tous les éléments $i \in MIN(P)$ en retirant tour à tour i de l'ensemble S, donnant lieu à un nouveau poset P' dont les éléments $S' = S \setminus \{i\}$ forment un haut-ensemble. Intuitivement, on peut penser que l'algorithme transite à partir de tous les haut-ensembles de P et ainsi $\mathcal{U}$ représente l'ensemble des états $\mathcal{S}$ de l'algorithme de programmation dynamique 3.2. Pour le démontrer, on procède par induction. Soit $\mathcal{S}^i$ l'ensemble des états à l'étage i où chaque état $U \in \mathcal{S}^i$ est tel que $i = |P| - |U|$. À l'étage $i = 0$ on a $\mathcal{S}^0 = S \in \mathcal{U}$. À l'étage $i = 1$, on a $\mathcal{S}^1 = \bigcup_{j \in MIN(P)} P \setminus \{j\}$ de plus il s'agit de tous les haut-ensembles de cardinalité $|P| - 1$. Supposons que $\forall U \in \mathcal{U}$ on a $U \in \mathcal{S}^n$ tel que $|U| = |P| - n,\ 1 < n < |P|$. Soit $U' \in \mathcal{U}$ un haut-ensemble de P tel que $|U'| = |U| - 1 = |P| - (n+1)$. $\exists j \in S$ tel que $U' \cup \{j\} \in \mathcal{U}$ de plus $|U' \cup \{j\}| = |P| - n$ et $U' \cup \{j\} \in \mathcal{S}^n$ par hypothèse d'induction et donc nécessairement l'algorithme va transiter de l'état $U \in \mathcal{S}^n$ de l'étage $|P| - n$ vers l'état $U' \setminus \{j\} \in \mathcal{S}^{n+1}$ de l'étage $|P| - (n+1)$ en retirant j et par conséquent $\mathcal{S} = \mathcal{U}$.

En utilisant une approche similaire, on peut démontrer que l'ensemble $\mathcal{L}$ des bas-ensembles de P représente également l'espace des états $\mathcal{S}$ de l'algorithme de programmation dynamique

3.2. Comme indication, si $U \in \mathcal{U}$ est un haut-ensemble, alors $(S \setminus U) \in \mathcal{L}$ est un bas-ensemble de P. Encore plus intéressant, on a $\mathcal{S} = \mathcal{A}$. En effet, si $U \in \mathcal{U}$ est un haut-ensemble, alors U est un sous ensemble de P partiellement ordonné. Soit $A = MIN(U)$, d'après la définition de l'ensemble minimal, A est une antichaîne. De plus, A est unique, car à chaque poset est associé un ensemble d'éléments minimaux unique. Maintenant, soit $A \in \mathcal{A}$ une antichaîne de P, alors $\bigcup_{i \in A} U[i]$ est un haut-ensemble de P. En mots, il existe une correspondance 1 : 1 entre les éléments de $\mathcal{U}$ et $\mathcal{A}$ tel que montré à la figure 3.3.



(a) Espace des états de la récurrence 3.2 encodé par $\mathcal{A}$ l'ensemble des antichaînes du poset de l'exemple 3.1.



(b) Espace des états de la récurrence 3.2 encodé par $\mathcal{U}$ l'ensemble des haut-ensembles du poset de l'exemple 3.1.

Figure 3.3 Deux représentations possibles de l'espace des états de la récurrence 3.2. À chaque état U de la figure (b) correspond un ensemble unique A de la figure (a) tel que $MIN(U) = A$. Les arêtes montrent les transitions entre états.

Un algorithme bien connu qui génère un sous-ensemble de $\mathcal{A}$ est l'algorithme de tri topologique. La différence par rapport à l'algorithme de programmation dynamique pour le problème de dénombrement est que le premier se contente de choisir un élément à chaque étage alors que le dernier doit transiter à travers tous les éléments de l'état courant. Pour compiler le programme dynamique, on utilise une adaptation de l'algorithme de tri topologique en ajoutant la fonctionnalité de choix exhaustifs. Pour s'assurer que l'algorithme ne transite pas plus qu'une seule fois par un état, on utilise une table associative dont les clefs sont les entiers correspondant à la représentation binaire d'un sous-ensemble de P : dans notre cas il s'agit des ensembles de $\mathcal{A}$ (figure 3.4). L'unicité des clefs est assurée par le fait que chaque sous-poset de P est associé à un ensemble unique d'éléments minimaux $A = MIN(P)$.

L'algorithme 2 compile le programme dynamique par une approche du bas vers le haut

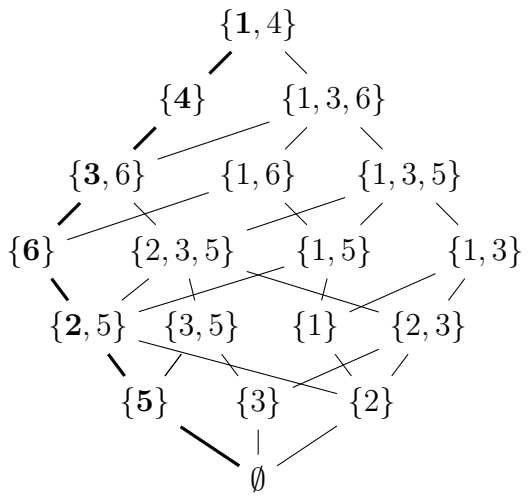


Figure 3.4 Trace de l'algorithme de tri topologique selon l'ordre lexicographique donnant la solution 1-4-3-6-2-5.

bottom-up en utilisant l'ensemble $\mathcal{U}$ pour encoder chaque état. Selon le problème à résoudre, une représentation de l'espace des états pourrait être plus appropriée qu'une autre. Dans le cas de dénombrement des extensions linéaires, ce choix est indifférent. Par contre, pour la mise à jour incrémentielle, les états seront encodés par les ensembles de $\mathcal{U}$ et de $\mathcal{L}$: dans le premier cas, on utilisera $\mathcal{S} = \mathcal{U}$ et dans le deuxième $\mathcal{S} = \mathcal{L}$.

À la base, l'algorithme 2 est un algorithme de tri topologique si on désactive les lignes $12-19$ et en itérant directement sur les éléments de A à la ligne 2. A étant l'ensemble des éléments minimaux d'un graphe orienté sans cycle. Dans ce cas, on ignore la clef c et le tri topologique est l'ordre dans lequel on ajoute les éléments i de la ligne 3 dans une liste. Les lignes $12-19$ assurent la recherche exhaustive lorsque les appels récursifs sont faits, c.-à-d. si l'algorithme choisit de retirer un élément $i \in A$ à la ligne 3, alors au retour il choisira un autre élément de A tout simplement. En particulier, les lignes $12-15$ sont la contrepartie des lignes $6-9$ permettant de rétablir les éléments de A tels qu'ils étaient avant l'appel récursif à la ligne 11. Notons qu'il est nécessaire que la relation R soit une relation de couverture, car sinon la condition à la ligne 14 ne fonctionnera pas correctement.

La clef c est utilisée par l'algorithme pour accéder aux entrées de la table associative $\mathcal{S}$ qui mémorise les états de l'algorithme de programmation dynamique. L'accès à un état dans $\mathcal{S}$ est indiqué par l'opérateur [ ]. La clef est initialisée à $2^{|P|-1}$ dans l'algorithme 3 pour indiquer que tous les éléments du poset sont présents : il s'agit du premier état U, qui est le haut-ensemble P et également le poset. Pour chaque élément i que l'algorithme retire de U, la clef est recalculée en enlevant la valeur correspondante par l'opérateur de décalage pour les bits

Algorithme 2 : CompilationBasEnHaut(R,A,$\mathcal{S}$,δ,c)

```
1  t ← taille de A;
2  tant que t > 0 faire
3      i ← retirer dernier élément de A;
4      c ← c − (1 ≪ i);
5      si S ne contient pas la clef c alors
6          pour chaque j ∈ S : (i, j) ∈ R faire
7              δ[j] ← δ[j] − 1;
8              si δ[j] = 0 alors
9                  Ajouter j en dernier dans A;
10         MIN(S[c]) ← A;
11         CompilationBasEnHaut(R,A,S,δ,c);
12         pour chaque j ∈ S : (i, j) ∈ R faire
13             δ[j] ← δ[j] + 1;
14             si δ[j] = 1 alors
15                 Retirer dernier élément dans A;
16     Ajouter i à l'avant dans A;
17     c ← c + (1 ≪ i);
18     le(S[c]) ← le(S[c]) + le(S[c − (1 ≪ i)]);
19     t ← t − 1;
```

$\ll$ (ligne 4). Ensuite l'information qu'on veut conserver au niveau de l'état U dont la clef est c est ajoutée. Pour le dénombrement, on s'intéresse au nombre d'extensions linéaires $le(\mathcal{S}[c])$ de chaque sous poset de $U \subseteq P$ (ligne 18). On conserve également au niveau de chaque état, l'ensemble minimal $MIN(\mathcal{S}[c])$ correspondant tel qu'indiqué à la ligne 10 de l'algorithme 2.

Analyse L'algorithme standard pour le calcul de la réduction transitive est dans l'ordre de $\mathcal{O}(n^3)$ où $|P| = n$. La complexité de l'algorithme 3 est dominée par l'appel à la ligne 11. Pour l'algorithme 2, on tente une analyse amortie de la borne supérieure du nombre des opérations élémentaires. Soit $\mathcal{A}_i$ l'ensemble des états de l'étage i et k la largeur de P. La boucle de la ligne 2 fait au plus k itérations puisque A est une antichaîne. Les lignes $3-4$ et $16-19$ sont toutes des opérations qui s'exécutent en temps constant. Pour simplifier, on regroupe ces opérations en une seule opération élémentaire. Le nombre de fois où la condition à la ligne 5 est vérifiée correspond à la cardinalité de l'étage suivant $|\mathcal{A}_{i+1}|$ et cela pour tous les états de $\mathcal{A}_i$ réunis. Le nombre d'itérations des boucles à la ligne 6 et 12 sont bornées par k chacune en supposant l'utilisation de la réduction transitive de R. Pour simplifier, on regroupe les opérations des lignes $7-10$ et les lignes $13-15$ en une seule opération élémentaire. Maintenant, on compte le nombre de ces opérations à chaque étage : À l'étage 0, on a $k + 2 \times k \times |\mathcal{A}_1|$ opérations. À l'étage suivant on a $k \times |\mathcal{A}_1| + 2 \times k \times |\mathcal{A}_2|$,.... Le

Algorithme 3 : DénombrementExtensionLinéaire($P(S, R)$)

```
1  R ← réduction transitive de R;
2  U ← table associative pour encoder les états;
3  c ← 2^|S| − 1, A ← ∅;
4  δ ← tableau de |S| éléments initialisés à la valeur 0;
5  pour chaque i ∈ {1, ..., |S|} faire
6      δ[i] ← calculer degré entrant;
7      si δ[i] = 0 alors
8          ajouter i en dernier dans A;
9  MIN(U[c]) ← A;
10 le(U[0]) ← 1;
11 CompilationBasEnHaut(R,A,U,δ,c);
12 retourner le(U[c]);
```

nombre total d'opérations est $\sum_{1\leq i\leq n} k \times |\mathcal{A}_{i-1}| + 2 \times k \times |\mathcal{A}_i| = \sum_{0\leq i<n} 3 \times k \times |\mathcal{A}_i| + 2 \times k$ qui est dans l'ordre de $\mathcal{O}(k \times |\mathcal{A}|)$. Puisque $|A| \leq k,\ \forall A \in \mathcal{A}$ alors $|\mathcal{A}| \leq \binom{n}{k}$. La complexité asymptotique de l'algorithme 2 est $\mathcal{O}(k \times n^k)$.

Tel que mentionné dans le chapitre 2, la récursion dans l'algorithme de programmation dynamique pour calculer le nombre d'extensions linéaires est la même utilisée par d'autres algorithmes de programmation dynamique pour résoudre les problèmes de séquencement. L'algorithme 2 est conçu de manière à refléter cette propriété en séparant la génération des états de l'action à prendre pendant les transitions entre états. Pour le problème de dénombrement des extensions linéaires, à chaque transition d'un état U vers un état $U \setminus \{i\}$ à travers l'élément i, il suffit de mettre à jour la valeur $\#P(U)$ associée à l'état U qui représente le nombre d'extensions linéaires du sous-poset $P(U, R)$. La mise à jour est calculée par $\#P(U) = \#P(U) + \#P(U \setminus \{i\})$ (ligne 18 de l'algorithme 2) sachant que les valeurs initiales sont telles que $\#P(U) = 0$ si $U \neq \emptyset$ sinon $\#P(U) = 1$.

Pour donner une forme basique de généricité à l'algorithme 2 pour qu'il soit utilisable dans d'autres problèmes que celui du dénombrement, il suffit de placer l'action à faire pendant les transitions entre les états dans une fonction de *callback* et de passer le pointeur de la fonction à l'algorithme. Bien sûr, un peu d'ingénierie logicielle serait nécessaire pour permettre la mémorisation de différents types d'informations propres à chaque problème au niveau des états. Dans le cas de dénombrement des extensions linéaires, seulement la valeur de $\#P(U, R)$ est mémorisée tel que montré à la figure 3.5.

Un aspect intéressant à mentionner est que l'algorithme 3 calcule implicitement le nombre d'antichaînes dans un ensemble partiellement ordonné P sachant qu'une antichaîne est un ensemble indépendant du graphe de comparabilité de P. La complexité de cet algorithme

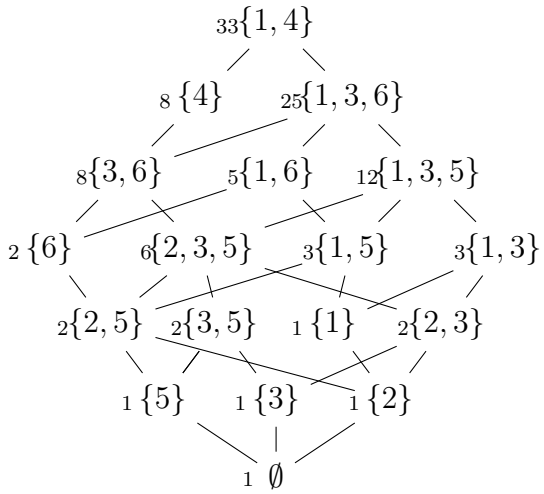


Figure 3.5 Calcul du nombre d'extensions linéaires de l'exemple 3.1 par l'algorithme 3. Devant chaque état A, le nombre d'extensions linéaires $\#P(U)$ où $MIN(U) = A$. Visuellement, il s'agit du nombre de chemins de l'état A à l'état $\emptyset$.

n'est pas tout à fait amortie dans la taille de $\mathcal{A}$, mais peut être qu'il serait possible de l'améliorer.

3.2.3 Algorithme de mise à jour

Le but de la mise à jour est de calculer la densité de permutations $\sigma(i, j)$ d'une paire d'éléments incomparables (i, j) dans un poset P. Ceci peut être réalisé soit directement $\sigma(i, j) = \frac{\#P(S|i<j)}{\#P(S)}$ en recalculant le nombre d'extensions linéaires $\#P(S|i < j)$ dans lesquelles i est classé avant j, ou indirectement $\sigma(i, j) = 1 - \frac{\#P(S|j<i)}{\#P(S)}$ en retirant le nombre d'extensions linéaires de P dans lesquelles j est classé avant i. Dans le premier, le recalcul du nombre d'extensions linéaires consiste à repasser par les étages dont les états doivent subir une mise à jour. On rappelle qu'après exécution de l'algorithme 3, à chaque état U est associée une valeur représentant le nombre d'extensions linéaires du sous-poset dont les éléments sont U. En disposant les états par étages selon leur cardinalité, le nombre d'extensions linéaires après ajout de la paire (i, j) est recalculé par

$$\#P(S|i < j) = \begin{cases} \sum_{k \in MIN(S)} \#P(S \setminus \{k\}|i < j) & si |S| > |U[i]| \\ \#P(S) & sinon \end{cases} \tag{3.3}$$

La différence par rapport à 3.2 est qu'ici les appels récursifs s'arrêtent à l'étage $|U[i]|$. Au-delà de cet étage, toutes les valeurs associées aux états générés initialement n'auront pas

à subir de modifications, puisqu'aucune transition à travers j vers un état contenant i ne peut s'y retrouver. De telles transitions reflètent le positionnement de j avant i dans une extension linéaire. Pour montrer que l'étage $|U[i]|$ est l'étage limite, il suffit de remarquer que les positions p_i possibles pour le classement de i dans toute extension linéaire sont telles que $|L[i]| \leq p_i \leq n - |U[i]| + 1$. Si j peut transiter vers un état U tel que $i \in U$ au-delà de l'étage $|U[i]|$ alors la position p_j de l'élément j dans le classement dans de telles permutations est telle que $n - |U[i]| \leq p_j < p_i$ ce qui est contradictoire avec $p_i \leq n - |U[i]|$.

Cette version d'algorithme de mise à jour du nombre d'extensions linéaires aura une performance variable, car elle dépend de la cardinalité du haut-ensemble $U[i]$. De plus, tous les états dans les étages allant jusqu'à $|U[i]|$ en partant de l'étage $|S|$ doivent être visités même si plusieurs n'auront pas d'impact sur le nouveau nombre d'extensions linéaires. Un exemple de mise à jour par 3.3 est montré à la figure 3.6.

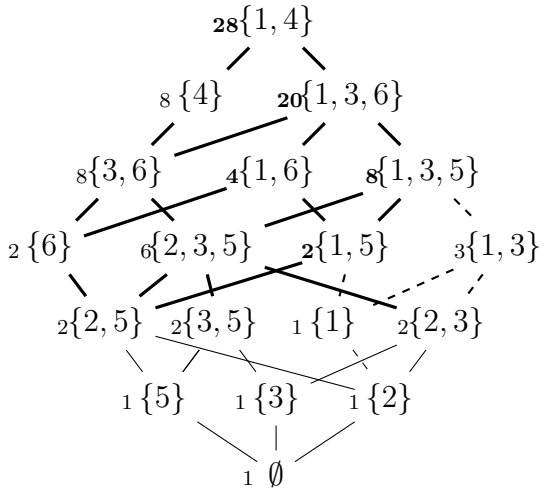


Figure 3.6 Espace des états $\mathcal{A}$ après ajout de la paire $(1, 5)$. En gras, les valeurs mises à jour. Les états $\{1, 3\}$ et $\{1\}$ sont retirés, car ils ne sont plus compatibles avec la nouvelle paire. Les transitions impliquant ces 2 états sont montrées en tirets.

Dans une mise à jour incrémentielle, seulement les états impliqués dans le changement du nombre d'extensions linéaires doivent être visités. Une deuxième approche consiste à éliminer les extensions linéaires dans lesquelles j est classé avant i. En effet, l'ajout de la paire (i, j) se traduit par le retrait de certains états de $\mathcal{U}$. Un état U non plus compatible avec l'ajout de la paire (i, j) est tel que $i \in U$ et $j \notin U$. Deux états attirent notre attention : le premier est $U = U[i]$ le haut-ensemble de l'élément i. Cet état est définitivement non plus compatible et doit être retiré, car l'ajout de la paire (i, j) a un impact direct sur $U[i]$ par la définition 3.4. Le deuxième état remarquable est $L[j]$ le bas-ensemble de l'élément j. Encore une fois, cet état ne sera plus compatible après l'ajout de la paire (i, j) car i doit s'ajouter à l'ensemble $L[j]$. Le

retrait du bas-ensemble $L[j]$ doit s'accompagner du retrait du haut-ensemble $U = S \setminus L[j]$. Pour cibler exactement tous les états qui ne sont plus compatibles par l'ajout de la paire (i, j), on définit l'ensemble $\{U^{\uparrow}\} = \{U' \in \mathcal{U} : U \subseteq U'\}$ et $\{U^{\downarrow}\} = \{U' \in \mathcal{U} : U' \subseteq U\}$. Par cette définition, on a $\{S^{\downarrow}\} = \{\emptyset \uparrow\} = \mathcal{U}$. On peut vérifier que les états qui doivent être retirés suite à l'ajout de la paire (i, j) sont $\{U[i]^{\uparrow}\} \cap \{(S \setminus L[j])^{\downarrow}\}$. En effet, soit U un haut-ensemble tel que $U \in \{U[i]^{\uparrow}\}$ alors par définition, on a $i \in U$. Si de plus $U \in (S \setminus L[j])^{\downarrow}\}$ alors $j \notin U$ car $j \in L[j]$ et donc $j \notin S \setminus L[j]$. Or, on sait que les étages non plus compatibles avec l'ajout de la paire (i, j) sont ceux contenant i mais pas j.

L'algorithme de mise à jour doit visiter uniquement les états de $\{U[i]^{\uparrow}\} \cap \{(S \setminus L[j])^{\downarrow}\}$. Ensuite, si une transition impliquant l'élément j est trouvée alors il faut retirer la contribution dans le nombre d'extensions linéaires calculé initialement suite à cette transition. La contribution dans le nombre des extensions linéaires d'un état retiré U dont j est l'élément de transition, est le produit $\#P(U) \times \#P(S \setminus (U \cup \{j\})$ selon 3.1. Pour retirer correctement la contribution de l'état U, il nous faut le nombre des extensions linéaires du sous-poset dont les éléments sont le bas-ensemble $L = S \setminus (U \cup \{j\})$ ou de manière générale les valeurs associées aux éléments des états de $\mathcal{L}$. Pour ce faire, une deuxième compilation du programme dynamique est nécessaire, cette fois, c'est l'espace $\mathcal{L}$ qui sera utilisé pour encoder les états. Aucune modification de l'algorithme 3 n'est nécessaire : il suffit de l'adapter en inversant les paires dans la relation de P. En d'autres mots, on utilise le poset $P^{inv}(S, R^{inv})$ comme entrée du programme tel que $R^{inv} = \{(i, j) : (j, i) \in R\}$. Donc une étape d'initialisation qui consiste à compiler le programme dynamique à travers les bas-ensembles *et* haut-ensembles est nécessaire pour obtenir une mise à jour incrémentielle. À partir des éléments cités plus haut, la mise à jour du nombre d'extensions linéaires $\#P(S|i < j)$ après l'ajout de la paire (i, j) au poset P est calculée par $\#P(S|i < j) = \#P(S) - \#P(S|j < i)$ tel que :

$$\#P(S|j < i) = \#P(S) \times \#P(\overline{S} \setminus \{j\}) + \sum_{k \in MIN(S), k \neq i, k \notin U[j]} \#P(S \setminus \{k\}|j < i)) \tag{3.4}$$

La densité de la paire (i, j) est $\sigma(i, j) = 1 - \frac{\#P(S \setminus U[j]|j<i)}{\#P(S)}$.

L'algorithme 4 ressemble à l'algorithme 2 dans les transitions sauf qu'ici, seulement les états impliqués dans le changement du nombre d'extensions linéaires seront visités. La première mise à jour calculée à la ligne 2 est celle du retrait de l'état U correspondant au haut-ensemble de j dont la clef est passée en paramètre par l'algorithme 5. Pour le calcul de cette contribution, l'algorithme va chercher le nombre d'extensions linéaires du bas-ensemble opposé dépourvu de j dont la clef indiquée à la ligne 2 de l'algorithme 4 par $c^{inv} - (1 \ll j)$

Algorithme 4 : MiseAJourBasEnHaut(c,i,j)

1 $c^{inv} \leftarrow$ inverser la clef c ;
2 $ule \leftarrow le(\mathcal{U}[c]) \times le(\mathcal{L}[c^{inv} - (1 \ll j)])$;
3 $A \leftarrow MIN(\mathcal{U}[c])$;
4 $t \leftarrow$ taille de A;
5 **tant que** $t > 0$ **faire**
6 $s \leftarrow$ retirer dernier élément dans A ;
7 $c^{suiv} \leftarrow c - (1 \ll s)$;
8 **si** $s \neq i$ *et* $s \notin U[j]$ *et* $\mathcal{U}[c^{suiv}]$ *pas encore visité* **alors**
9 $ule \leftarrow ule +$ MiseAJourBasEnHaut(c^{suiv},i,j);
10 Ajouter s à l'avant dans A;
11 $t \leftarrow t - 1$;
12 retourner ule;

Algorithme 5 : MiseAJourExtensionLinéaire($P(S,R)$,i,j)

1 $c \leftarrow$ calculer la clef du bas ensemble de j;
2 $c^{inv} \leftarrow$ inverser la clef c pour avoir le haut-ensemble correspondant;
3 $ule \leftarrow$ MiseAJourBasEnHaut(c^{inv},i,j);
4 $c^{top} \leftarrow 2^{|S|} - 1$;
5 retourner $le(\mathcal{U}[c^{top}]) - ule$;

ensuite calculer le produit avec $\#P(U)$. Cela se traduit tout simplement par le retrait d'une partie des extensions linéaires où j est positionné avant i. Le reste des extensions linéaires de la mise à jour est calculé récursivement (ligne 9 de l'algorithme 4). Les conditions à la ligne 8 permettent de revisiter seulement les états $\{U[i]^{\uparrow}\} \cap \{(S \setminus L[j])^{\downarrow}\}$.

La mise à jour incrémentielle est rendue possible grâce un travail de compilation supplémentaire par rapport à la première version. Dans le contexte où l'on doit calculer les mises à jour pour toutes les paires non comparables de P, ce travail supplémentaire est justifié et sera appuyé expérimentalement dans le chapitre 4. Deux petits détails à ajouter : Le premier est qu'il est possible que l'algorithme tente de repasser par un même état plus qu'une seule fois, et par conséquent multiplier le retrait des contributions de ce dernier. Pour éviter que cela se produise, il suffit d'utiliser une structure de donnée de booléens indiquant que la contribution d'un état a déjà été retirée. Le deuxième détail est la condition d'arrêt $k \in U[j]$ dans 3.4. En fait, lors d'une transition vers un état U à travers un élément k, si cet élément se trouve dans le haut-ensemble de j, alors tous les autres états qui vont suivre $\{U^{\downarrow}\} \cap \{U[i]^{\uparrow}\}$ doivent être retirés, *mais* ne seront pas impliqués dans la mise à jour, car aucun autre état ne peut transiter à travers j vers les états de $\{U^{\downarrow}\}$ sinon j serait incomparable avec k ce qui est absurde du fait que $k \in U[j]$.

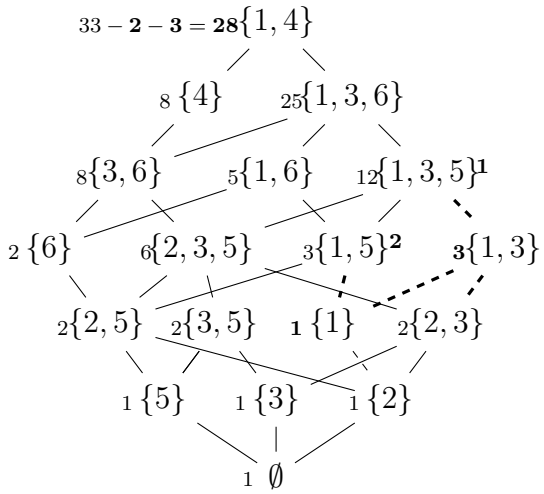


Figure 3.7 Mise à jour par 3.4 du nombre d'extensions linéaires du poset de l'exemple 3.1 après ajout de la paire $(1, 5)$. Deux opérations ont été nécessaires : la première est $\#P(\{1, 2, 3\}) \times \#P(\{4, 6\}) = 3$ et la deuxième est $\#P\{1, 2\} \times \#P(\{3, 4, 6\}) = 2$. Il s'agit du nombre d'extensions linéaires à retirer où 5 est avant 1.

3.3 Algorithme heuristique

3.3.1 Estimation des densités

L'utilisation de l'algorithme exact présenté dans la section précédente est limitée par la taille n du poset ainsi que sa largeur k. De ce fait, il est nécessaire de prévoir une alternative qui sera utilisée pour des tailles plus grandes de n et k. On retient de la section précédente que l'ajout d'une paire (i, j) d'éléments incomparables à la relation R du poset P se traduit par le retrait de l'ensemble des extensions linéaires de P non plus compatibles, c.-à-d. dans lesquelles j est classé avant i, mais ce n'est pas tout. En général, l'ajout de la paire (i, j) est en réalité un ajout de plusieurs paires $\{(i', j') : (i', j') \in L[i] \times U[j],\ (i', j') \notin R\}$. Autrement dit, si on note R' étant la nouvelle relation de P, alors $R' = R \cup \{(i', j')\}$. Puisque R est une relation transitive, alors on note simplement $R' = R \cup (i, j)$. Une formule exacte pour compter la réduction du nombre de ces extensions linéaires en analysant le changement dans la relation d'ordre de P serait fort probablement non-polynomiale par la nature du problème. Le but est plutôt d'estimer la proportion de réduction dans le nombre d'extensions linéaires par le changement dans la cardinalité de la relation R ou de manière équivalente la cardinalité de l'ensemble des paires d'éléments non comparables $\overline{R}$, et de traduire cette réduction en densité.

L'heuristique est basée sur l'idée que la valeur de $|R \cup (i, j)| - |R|$ est inversement propor-

tionnelle à la densité de la paire (i, j). En mots, si l'ajout de la paire (i, j) cause un grand changement dans R ou de manière équivalente dans $\overline{R}$, alors ceci est un indicateur que la densité de la paire est de petite valeur. Si au contraire, un petit changement se produit, alors ça serait un indicateur que la densité de la paire (i, j) est élevée. Pour normaliser les densités pour toutes les paires non comparables, on choisit de diviser par le nombre total de ces derniers $|\overline{R}|$. La densité de la paire (i, j) est ainsi estimée par

$$\tilde{\sigma}(i, j) = \frac{|R \cup (i, j)| - |R|}{|\overline{R}|} \tag{3.5}$$

Dans le cas exact, on peut décider de choisir de calculer la densité $\sigma(i, j)$ soit par l'ajout de la paire (i, j) à la relation de R ou bien (j, i) et de déduire la valeur par 1-$\sigma(j, i)$. Dans le cas heuristique on n'a pas forcement $\tilde{\sigma}(i, j) = 1 - \tilde{\sigma}(j, i)$. Toutefois, si dans le cas exact on a $\sigma(i, j) > \sigma(j, i)$ alors, heuristiquement on aura $\frac{|R\cup(i,j)|-|R|}{|R\cup(j,i)|-|R|} < 1$. De plus cette valeur s'approcherait de 0 quand $\sigma(i, j)$ tend vers 1. Sinon, si $\sigma(i, j)$ tend vers 0.5 alors $\frac{|R\cup(i,j)|-|R|}{|R\cup(j,i)|-|R|}$ prendrait des valeurs proches de 1. Une fonction f décroissante sur l'intervalle $x \in [0, 1]$ telle que $f(0) = 1$ et $f(1) = 0.5$ est $f := 1 - \frac{x}{2}$. On peut alors réécrire 3.5 par

$$\tilde{\sigma}(i, j) = 1 - \frac{|R \cup (i^*, j^*)| - |R|}{2 \times (|R \cup (j^*, i^*)| - |R|)} \tag{3.6}$$

où $(i^*, j^*) = \text{argmin}_{\{(i,j),(j,i)\}} |R \cup (i, j)|$. L'avantage d'utiliser 3.6 par rapport à 3.5 est que l'information obtenue par le changement dans la cardinalité de la relation est exploitée dans les deux sens en plus d'obtenir les densités d'une paire et son opposée ayant une somme égale à 1 pour ainsi imiter le cas exact.

Exemple 3.4. La densité de permutations de la paire $(1, 5)$ du poset de l'exemple 3.1 est $\frac{28}{33} \approx 0.85$. Pour estimer cette densité, on applique 3.6. Pour la paire $(1, 5)$ on a $L[1] \times U[5] \cap R = \{(1, 5)\}$ donc $|R\cup(1, 5)|-|R| = 1$. Pour $(5, 1)$, on a $L[5]\times U[1]\cap R = \{(5, 1), (5, 2), (6, 1), (4, 1)\}$ donc $|R \cup (5, 1)| - |R| = 4$. La première constatation est que la densité de $(1, 5)$ serait plus élevée que celle de la paire $(5, 1)$. On trouve $\tilde{\sigma}(i, j) = \tilde{\sigma}(1, 5) = 1 - \frac{1}{8} = 0.875$. Les densités de l'ensemble des paires incomparables sont montrées à la figure 3.8.

Pour cet exemple, on remarque que le classement trouvé par l'heuristique est presque identique à celui de l'algorithme exact. Toutefois, la formule 3.5 a comme faiblesse la tendance à produire plusieurs cas ex aequo. Pour briser l'égalité, on utilise la cardinalité du produit cartésien $|L[i] \times U[j]|$ et $|L[i'] \times U[j']|$ de deux paires ex aequo (i, j) et (i', j'). La paire dont la valeur calculée est petite aura la priorité. Ce bris d'égalité reflète l'idée intuitive qu'en général, la paire (i, j) de densité maximum est celle dont l'élément i est tel que $i \in MIN(P)$

Exacte		Heuristique	
(i,j)	$\sigma(i,j)$	(i,j)	$\tilde{\sigma}(i,j)$
$(1,5)$	0.85	$(1,5)$	0.87
$(4,1)$	0.76	$(4,1)$	0.87
$(6,3)$	0.73	$(1,3)$	0.83
$(1,3)$	0.73	$(6,3)$	0.83
$(3,2)$	0.70	$(3,2)$	0.83
$(3,5)$	0.64	$(3,5)$	0.75
$(5,2)$	0.57	$(5,2)$	0.70
$(1,6)$	0.54	$(1,6)$	0.50

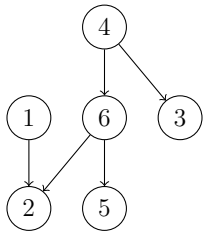


Figure 3.8 Classement des paires non comparables du poset de l'exemple 3.1 et valeurs de densités calculées par l'algorithme exact et heuristique.

et que $j \in MAX(P)$. Dans ce cas on a une valeur égale à 1 et dans le cas opposé on aura une valeur proche de $(\lceil \frac{n}{2} \rceil - 1)^2$. Dans l'exemple de la figure 3.8 on a $|L[1] \times U[5]| = 1$ et $|L[4] \times U[1]| = 2$. La paire $(1,5)$ sera classée avant $(4,1)$.

3.3.2 Estimation du nombre d'extensions linéaires

L'algorithme heuristique présenté dans la section précédente calcule une estimation des densités de permutations des paires incomparables (i,j) dans un poset P sans passer par le calcul du nombre de permutations. Dans le contexte de branchement par l'heuristique CBS, le calcul de la densité est la finalité et donc justifiable en particulier pour les problèmes dont la structure est principalement composée de contraintes de même type, dans notre cas, des contraintes de ressources unaires. Cependant, dans certaines variantes de l'heuristique CBS, le nombre de solutions associé à la contrainte est sollicité. Dans ces cas la possibilité de donner le nombre de permutations ou une estimation de ce dernier est nécessaire quand le dénombrement exact devient coûteux.

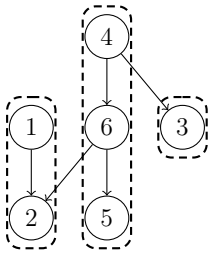
Décomposition en chaînes La décomposition en chaînes d'un poset $P(S,R)$ consiste à partitionner l'ensemble S en sous-ensembles $\{C_i\}_{1 \leq i \leq k}$ tel que C_i est une chaîne de P et $\bigcup_{1 \leq i \leq k} C_i = S$ et $\bigcap_{1 \leq i \leq k} C_i = \emptyset$. Cette décomposition sert, entre autres, à estimer l'entropie $H(\overline{P})$ du graphe d'incomparabilité de P par $H(\overline{P}) = \sum_{i=1}^{k} -\frac{|C_i|}{|S|} \log \frac{|C_i|}{|S|}$. L'algorithme vorace proposé par Cardinal et al. (2013), retire à chaque itération une chaîne de taille maximum dans le but de minimiser le nombre total de chaînes de la décomposition et ainsi minimiser la valeur de l'entropie. Si on note $C[i]$ une chaîne C dont le dernier élément est i, alors on peut

calculer pour chaque élément du poset, la chaîne de taille maximum dont il est le dernier élément par programmation dynamique comme suit :

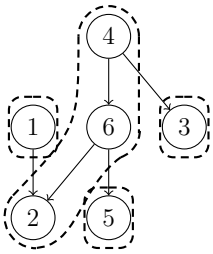
$$C[i] = \begin{cases} C[j] \cup \{i\} : j = \underset{\substack{k \prec i \\ (k,i) \in R}}{\operatorname{argmax}} |C[k]| & i \notin MIN(P) \\ \{i\} & sinon \end{cases} \tag{3.7}$$

Où "$\prec$" indique l'ordre selon le tri topologique des éléments de S. La chaîne de taille maximum C_{max} est telle que $max = \underset{i \in MAX(P)}{\operatorname{argmax}} |C[i]|$.

En temps normal, cet algorithme a une complexité de $\mathcal{O}(n^3)$ et cette complexité est dominée par la fermeture transitive de R. Cardinal et al. (2013) ont réussi à montrer qu'une décomposition peut être obtenue en $\mathcal{O}(n^{2.5})$ en supposant que la fermeture transitive est calculée par un algorithme de complexité $\mathcal{O}(n^{2.3})$ basé sur la multiplication matricielle. Or dans le cas pratique, la différence serait négligeable pour les tailles de posets d'utilisation normale. On pense qu'une amélioration de l'algorithme serait celle qui a un impact sur la valeur de l'entropie obtenue plutôt que la complexité. En effet, un poset admet plusieurs décompositions possibles par l'algorithme vorace, car durant la construction de la plus longue chaîne, il peut y avoir plusieurs choix possibles de sélection d'éléments et ces choix auront un impact sur la construction de la chaîne à l'itération suivante. L'exemple de la figure 3.9 illustre cette situation pour le poset de l'exemple 3.1.



(a) Le choix de la première longue chaîne $\{4, 5, 6\}$ permet une décomposition minimum en 3 chaînes.



(b) Le choix de la première longue chaîne $\{2, 4, 6\}$ cause une décomposition en 4 chaînes.

Figure 3.9 Impact du choix entre les chaînes ex-æquo.

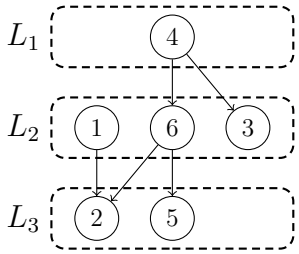
Parmi les deux décompositions trouvées dans l'exemple à la figure 3.9, celle dont le nombre de chaînes est 3 donne une meilleure estimation de la valeur de l'entropie. On rappelle que le calcul de l'entropie de graphe est un problème de minimisation. On propose comme amé-

lioration d'ajouter un critère vorace pour la sélection des éléments pendant la construction de la chaîne. Dans leur article, Cardinal et al. (2013) mentionnent la possibilité de classer les éléments d'un poset selon m niveaux où m est la taille de la plus grande chaîne de P. Le niveau l_i d'un élément i peut être calculé dynamiquement par :

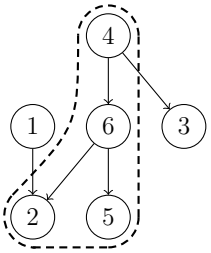
$$l_i = \begin{cases} \max\limits_{\substack{j \prec i \\ (j,i) \in R}} \{l_j\} + 1 & i \notin MIN(P) \\ 1 & sinon \end{cases} \tag{3.8}$$

où "$\prec$" est l'ordre selon le tri topologique des éléments de P.

Maintenant, on introduit un nouveau classement des éléments de P : les éléments fixes et les éléments libres. Un élément i est fixe si $l_i + l'_i = m + 1$ où l'_i est le niveau de l'élément i du poset $P(S, R_{inv})$ tel que $R_{inv} = \{(j, i) : (i, j) \in R\}$. En mots, on inverse les paires de la relation R du poset initial et on recalcule les niveaux des éléments par 3.8. Un élément libre i aura la propriété $l_i + l'_i \leq m$. On nomme l'ensemble des éléments fixes par F. Soit $L_p = \{i \ : \ l_i = p, i \in F\}$, on a $1 \leq p \leq m$ de plus la chaîne C_{max} de taille maximum est telle que $C_{max} \in \prod_{1 \leq p \leq m} L_p$.



(a) La taille de la plus grande chaîne détermine le nombre de niveaux.



(b) Les niveaux des éléments $\{2, 4, 5, 6\}$ sont fixes contrairement à $\{1, 3\}$.

Figure 3.10 Niveaux des éléments dans un poset.

On remarque que si l'ensemble des éléments L_p du niveau p contient un élément unique, alors cet élément doit nécessairement faire partie de la chaîne C_{max} et sa position dans la chaîne est p. Pour construire la chaîne la plus longue, deux possibilités sont à considérer : la première est de maintenir une liste de chaînes candidates pendant tout le processus de sélection des éléments à travers les niveaux $\{L_p\}_{1 \leq p \leq m}$. Chaque chaîne aura un score initialisé à 0. À chaque niveau, on calcule le meilleur choix d'élément i parmi les éléments fixes du niveau L_{p+1} selon la valeur qui minimise la somme des degrés entrants et sortants de l'élément i dans le graphe de comparabilité de P. On ajoute ensuite l'élément choisi aux chaînes candidates selon leurs

scores et on met à jour le score associé à la chaîne. Au dernier niveau, on retient la chaîne avec le meilleur score. On remarque qu'à chaque niveau L_p on doit calculer un score pour toutes (ou presque) les paires d'éléments dans L_p et L_{p+1}. Pour un poset dont la taille de la chaîne la plus longue est m on aura, par une analyse amortie, un nombre d'éléments dans chaque niveau égal à $\lceil \frac{n}{m} \rceil - 1$. Le nombre d'opérations est donc borné par $\sum_{1 \leq p \leq m} (\lceil \frac{n}{m} \rceil - 1)^2$ qui est dans l'ordre de $\mathcal{O}(n^2)$. L'autre possibilité plus simple est de garder une seule chaîne candidate en se déplaçant à travers les niveaux et d'ajouter dans chacun l'élément i selon le score décrit plus haut (Algorithme 7).

Algorithme 6 : DécompositionEnChaineVorace($P(S, R)$)

```
1 C ← ∅,r ← |S|;
2 tant que r > 0 faire
3     C ← PlusLongueChaine(P(S, R));
4     Ajouter C en dernier dans C;
5     pour chaque i ∈ C faire
6         S ← S \ {i}, R ← R \ {(j, k) : j = i ∨ k = i};
7         r ← r − 1;
8 renvoyer C;
```

Cette amélioration de l'algorithme de décomposition vorace permet de choisir la chaîne la plus longue en s'assurant que son retrait cause le minimum de "désordre" dans le poset et ainsi conserver la chance de trouver d'autres chaînes avec une taille plus grande dans les itérations suivantes et par conséquent avoir un nombre moindre de chaînes. En appliquant l'algorithme 6 sur l'exemple 3.1, on retrouve la décomposition montrée dans la figure 3.9(a) dont le nombre de chaînes reflète mieux le poset. Dans ce cas, il s'agit de la meilleure décomposition selon le Théorème de Dilworth's (Dilworth, 1950).

Estimation par entropie Une conséquence directe du résultat de Kahn and Kim (1995) est que le nombre d'extensions linéaires du poset P peut être borné par

$$2^{c_1 \times nH(\overline{P})} \leq \#P(S) \leq 2^{c_2 \times nH(\overline{P})} \tag{3.9}$$

où $n = |P|$ et $H(\overline{P})$ est l'entropie du graphe d'incomparabilité de P et $0 \leq c_1 < c_2 < 1$. Dans le cas exact, Cardinal et al. (2013) ont trouvé la constante $c_1 = 0.5$. Dans le cas de l'entropie calculée par l'algorithme vorace de décomposition en chaîne, la constante c sera déterminée statistiquement par régression linéaire multiple. On pense que c dépend de k la largeur du poset et de n la taille du poset. Les coefficients de régression linéaire pour la prédiction de la constance sont montrés dans le chapitre 4.

Algorithme 7 : PlusLongueChaine($P(S, R)$)

```
1  rang ← calculer les rangs des éléments de P(S, R) par 3.8;
2  P(S, R^inv) ← inverser les paires dans R;
3  rangInv ← calculer les rangs des éléments de P(S, R^inv) par 3.8;
4  C ← ∅, h ← valeur maximum dans rang;
5  si h = 1 alors
6      ajouter le premier élément i ∈ S à la chaîne C;
7  sinon
8      N ← tableau de h ensembles pour chaque niveau;
9      pour chaque i ∈ {1, ..., |S|} faire
10         si rang[i] + rangInv[i] = h + 1 alors
11             ajouter i dans N[rang[i]];
12     P ← tableau des poids des éléments de S initialisés à 0;
13     pour chaque i ∈ {1, ..., |S|} faire
14         pour chaque j ∈ S : (j, i) ∈ R ∨ (i, j) ∈ R faire
15             P[i] ← P[i] + 1;
16     l ← 1;
17     tant que l ≤ h faire
18         (i*, j*) ← (0, 0);
19         si l = 1 alors
20             pour chaque (i, j) ∈ N[l] × N[l + 1] : (i, j) ∈ R faire
21                 (i*, j*) ← meilleure paire qui minimise P[i] + P[j];
22             ajouter i* en dernier dans C;
23             ajouter j* en dernier dans C;
24             l ← l + 2;
25         sinon
26             i ← dernier élément de C;
27             pour chaque j ∈ N[l] : (i, j) ∈ R faire
28                 (i*, j*) ← meilleure paire qui minimise P[i] + P[j];
29             ajouter j* en dernier dans C;
30             l ← l + 1;
31 retourner C;
```

CHAPITRE 4 RÉSULTATS EXPÉRIMENTAUX

Dans ce chapitre, les résultats expérimentaux sont présentés. Tout d'abord, l'approche utilisée pour générer les posets est décrite à la première section. Dans les deux sections qui suivent, on présente les différents résultats expérimentaux des algorithmes exact et heuristique en particulier la performance en temps de calcul de l'algorithme exact ainsi que la précision de l'heuristique d'estimation. Pour évaluer l'efficacité de l'heuristique de branchement basée sur le dénombrement des permutations, on choisit le problème d'ordonnancement en atelier à cheminement multiple "Job-Shop". La description du problème, l'approche de résolution ainsi que les résultats comparatifs aux autres heuristiques de branchement sont présentés à la dernière section. Les discussions et commentaires accompagnent les résultats présentés tout au long de ce chapitre.

4.1 Algorithmes pour les posets

4.1.1 Génération de posets

Pour la génération d'un poset P, deux points seront à considérer : le premier concerne la largeur $w(P)$. En effet, l'analyse théorique de l'algorithme exact ainsi que l'entropie de la décomposition en chaînes utilisée dans l'estimation du nombre d'extensions linéaires montrent l'influence de $w(P)$ sur ces derniers. Ainsi, pour l'analyse empirique, les posets doivent être regroupés non seulement par leur taille, mais aussi par largeur. Le deuxième point concerne la connectivité des éléments de P. Pour l'évaluation de la performance des algorithmes, en particulier le cas exact, il n'est pas intéressant de le faire si le graphe de comparabilité de P n'est pas connexe, car le nombre d'extensions linéaires ainsi que les densités de permutations peuvent être déduites facilement en appliquant l'algorithme séparément pour chaque composante connexe, d'autant plus que ces posets ne serviront pas à évaluer l'heuristique de branchement, mais simplement à évaluer la performance en fonction de la taille et la largeur.

Un poset est simplement un graphe orienté sans cycles. Pour générer un tel graphe, une approche simple attribuée à Erdős & Rényi utilise deux paramètres n et m pour générer un graphe orienté de n sommets et m arcs choisis aléatoirement (voir Cordeiro et al., 2010). Pour obtenir des posets avec des largeurs différentes pour une même taille n, on pourrait varier la valeur de m ; la diminution de m ferait croître la largeur et vice versa. Toutefois, il devient difficile de générer des posets avec une largeur suffisamment grande pour nos tests sans obtenir des sommets isolés.

Nos posets sont générés en utilisant une approche différente. L'idée est de visualiser un poset comme un ensemble de k chaînes placées verticalement et reliées entre elles par des arcs. D'après le théorème de Dilworth (Dilworth, 1950), tout poset dispose de cette représentation où $k = w(P)$. De plus, chaque chaîne C_i contient un élément $a_i \in C_i$ tel que $A = \{a_i\}_{1 \leq i \leq k}$ est une antichaîne et est maximum.

Algorithme 8 : générerPosetAléatoire(n,k)

Initialiser $P \leftarrow \{1, \ldots, n\}$, $R \leftarrow \emptyset$, $E \leftarrow P$, $A = \{a_i\} \leftarrow \emptyset$, $\mathcal{C} = \{C_i\} \leftarrow \emptyset$;
$E \leftarrow$ brouiller l'ensemble E ;
tant que $|E| > 0$ **faire**
    $e \leftarrow$ dernier élément de E;
    si $\mathcal{C}$ *est vide* **alors**
        $a_1 \leftarrow e, C_1 \leftarrow \{e\}$;
    sinon
        $p \leftarrow \frac{k-|A|}{|E|}$;
        si $p >$ *Unif([0 1])* **alors**
            $A \leftarrow A \cup \{e\}, \mathcal{C} \leftarrow \mathcal{C} \cup \{\{e\}\}$;
            $j \leftarrow Unif([1, |\mathcal{C}| - 1])$;
            $s \leftarrow$ choisir un élément uniformément au hasard dans C_j tel que $s \neq a_j$;
            si $s \in D[a_j]$ **alors**
                $R \leftarrow R \cup (s, e)$;
            sinon
                $R \leftarrow R \cup (e, s)$;
        sinon
            $j \leftarrow Unif([1, |\mathcal{C}|])$;
            $s \leftarrow$ choisir un élément uniformément au hasard dans C_j;
            Insérer l'élément e avant s dans C_j et mettre à jour R;
            $j' \leftarrow Unif([1, |\mathcal{C}|])$;
            si $j \neq j'$ **alors**
                $s \leftarrow$ choisir un élément uniformément au hasard dans $C_{j'}$;
                si *($e \in D[a_j]$ et $s \in D[a_{j'}]$) ou (($e \in U[a_j]$ et $s \in U[a_{j'}]$)* **alors**
                    si *Unif[0,1]> 0.5* **alors**
                        $R \leftarrow R \cup (e, s)$;
                    sinon
                        $R \leftarrow R \cup (s, e)$;
                si $e \in D[a_j]$ *et* $s \in U[a_{j'}]$ **alors**
                    $R \leftarrow R \cup (e, s)$;
                sinon
                    $R \leftarrow R \cup (s, e)$;
    $E \leftarrow E \setminus \{e\}$;
retourner (P, R);

L'algorithme 8 génère un poset (P, R) avec une taille et une largeur définies respectivement

par les paramètres n et k. Au départ le poset est formé d'éléments isolés $P = E = \{1 \dots n\}$ et $R = \emptyset$. À chaque itération de la ligne 3, selon la probabilité $p = \frac{k-|A|}{|E|}$ (ligne 8) où A est maximum antichaîne à l'itération courante, un élément isolé $e \in E$ est soit ajouté à l'une des chaînes existantes de P ou bien utilisé pour instancier une nouvelle. Le calcul de p permet de garantir qu'on aura exactement k chaînes instanciées. Les préséances (arcs) entre éléments appartenant à deux chaînes différentes C_i et C_j doivent garantir de ne pas affecter l'incomparabilité des éléments $a_i, a_j \in A : a_i \in C_i$ et $a_j \in C_j$. La règle générale est que les arcs $(D[a_i] \setminus \{a_i\}) \times U[a_j]$ ou $D[a_i] \times (U[a_j] \setminus \{a_j\})$ peuvent être ajoutés à la relation R sans affecter la largeur de P. La connectivité est assurée par le fait qu'à chaque instanciation d'une nouvelle chaîne par l'élément e, ce dernier est connecté à un élément de l'une des chaînes existantes. On peut penser que cela crée un biais, car "force" les éléments de l'antichaîne à être reliés par des préséances, mais à cause de la réduction transitive, ces liaisons pourraient disparaître (visuellement) et le poset prend une "allure" normale. En annexe, on montre deux exemples de poset A.1 et A.2 générés par l'algorithme 8.

Tableau 4.1 Quelques statistiques sur les exemplaires de posets

n	w	nb d'exempl.	$\lvert\overline{R}\rvert$			$\lvert R'\rvert$		
			Min.	Max.	Moy.	Min.	Max.	Moy.
20	4	10	59	97	77.0	23	26	24.4
	8	10	96	148	117.1	23	26	24.5
	12	10	140	165	149.8	21	26	23.6
	16	10	139	164	156.2	19	21	20.4
30	4	10	143	206	178.5	34	38	36.7
	8	10	216	266	259.1	34	42	37.9
	12	10	289	335	317.8	37	42	39.9
	16	10	319	362	344.5	35	40	37.1
40	4	10	176	305	231.6	45	51	47.5
	8	10	363	476	420.0	49	58	52.5
	12	10	460	591	535.4	48	57	53.2
	16	10	549	614	579.4	47	54	52.1
50	4	10	228	458	339.4	53	64	59.1
	8	10	495	727	632.6	62	71	66.2
	12	10	744	948	836.8	66	75	71.1
	16	10	807	976	907.5	65	76	69.5

Les posets sont générés par séries de taille $n = \{20, 30, 40, 50\}$ et de largeur $k \in \{4, 8, 12, 16\}$. Chaque série d'exemplaires contient 10 posets pour un total de 160. Le tableau 4.1 montre quelques statistiques pour le nombre d'incomparabilités $|\overline{R}|$ des posets de chaque série d'exemplaires ainsi que la cardinalité $|R'|$ de la relation d'ordre des exemplaires après réduction

transitive.

4.1.2 Estimation par entropie

Tel que mentionné dans le chapitre précédent, le nombre d'extensions linéaires du poset P peut être estimé par $2^{c\times|P|\times H(P)}$ où $H(P)$ est l'entropie de la distribution obtenue de la décomposition en chaînes vorace de P et c est une constante qu'on déterminera statistiquement par régression linéaire en fonction de $|P|$ et $w(P)$. L'algorithme utilisé dans la détermination de la constante c est celui proposé dans le chapitre précédent. Pour justifier le choix de cet algorithme par rapport à l'algorithme de décomposition standard de Cardinal et al. (2013), on compare les valeurs moyennes d'entropie de toutes les séries d'exemplaires de posets. En principe, une valeur d'entropie plus petite serait plus précise c.-à-d. proche de la valeur exacte. Ceci n'est pas démontré formellement, mais l'idée est tirée du fait qu'à la base, le calcul d'entropie d'un graphe est un problème de minimisation. Les résultats de la comparaison sont montrés à la figure 4.1.

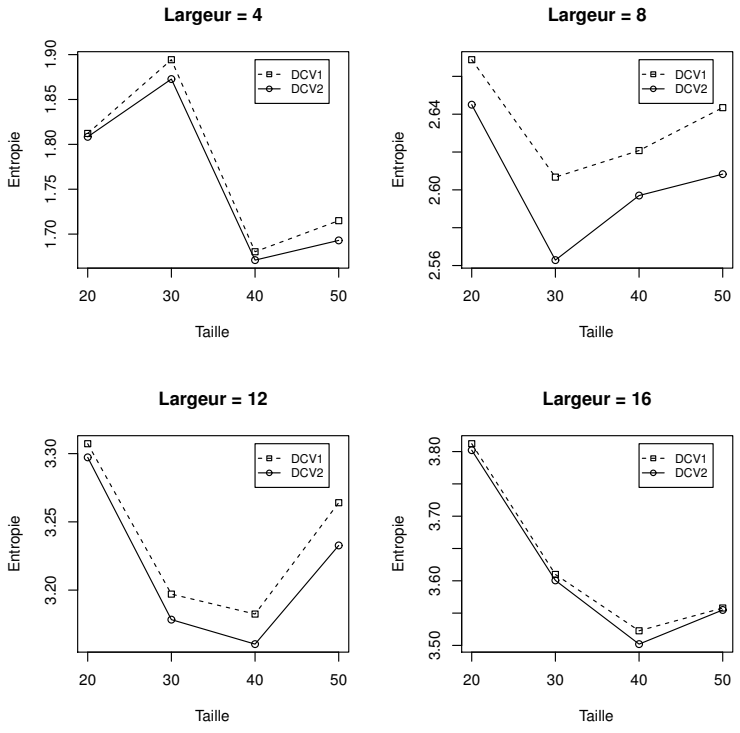


Figure 4.1 Comparaison selon la largeur des valeurs d'entropies moyennes calculées par les algorithmes de décomposition standard (DCV1) et celui proposé (DCV2)

Pour le test de régression, en utilise un échantillon de 1000 posets dont les tailles varient

entre 20 et 50 et les largeurs entre 4 et 16 pour un maximum de 10 par série. Les coefficients du modèle linéaire sont montrés dans le tableau 4.2.

Tableau 4.2 Coefficients de la régression linéaire pour estimer la constante c

	Estim.	Err. Std.
(Intercept.)	0.6210974	0.0055104
$\|P\|$	0.0025041	0.0001231
$w(P)$	0.0004076	0.0002980

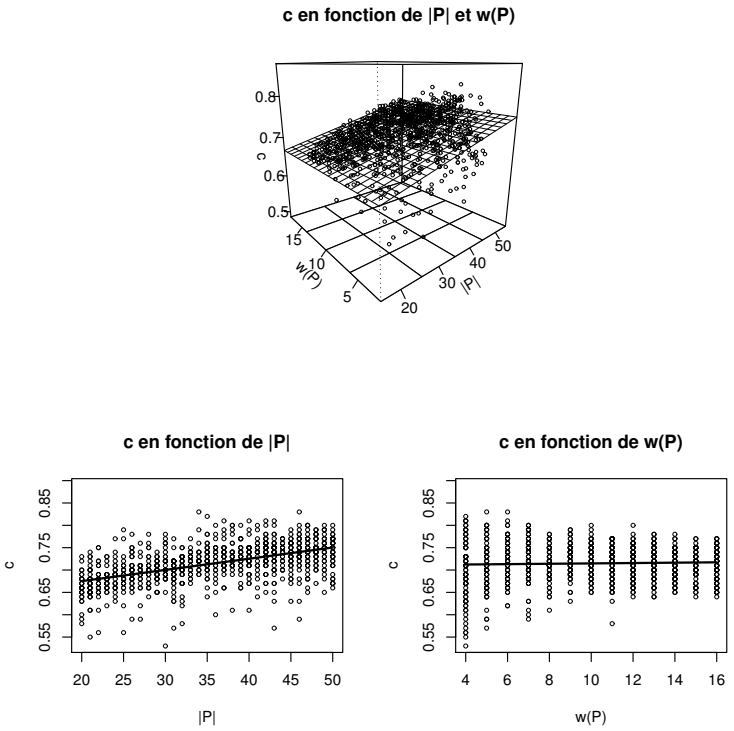


Figure 4.2 Graphiques du modèle de régression linéaire pour la prédiction de la constante c

D'après la figure 4.2 on remarque que la constante c varie entre 0.5 et 0.8. Tel que mentionné dans le chapitre précédent, théoriquement, cette constante est bornée inférieurement par 0.5 (Cardinal et al., 2013) en supposant que l'entropie est calculée exactement. Ici, on a une entropie qui n'est pas exacte pourtant, la borne est respectée. Ceci appuie les dires de Cardinal et al. (2013) que l'entropie calculée à partir de la décomposition en chaînes vorace est une bonne approximation. Une autre remarque est que la constante croît en fonction de la

taille du poset mais pas en fonction de la largeur. Cela contredit notre prédiction initiale. Par contre, cela pourrait être expliqué par le fait que l'influence de la largeur agit sur l'entropie calculée $H(P)$, plutôt que ça soit directement sur la constante c.

En utilisant le modèle de la régression linéaire montré dans le tableau 4.2 on prédit la valeur de la constante c pour chacun des posets dans le jeu de données du tableau 4.1. Cette valeur sert ensuite à estimer le nombre d'extensions linéaires $\#P'$. Le tableau 4.3 montre les résultats obtenus de l'estimation par entropie. L'erreur relative $\sigma = \frac{|\#P' - \#P|}{\#P}$ est utilisée comme mesure de la précision.

Tableau 4.3 Erreur relative de l'estimation par entropie

$\|P\|$	$w(P)$	Nb d'exempl.	Ordre de grandeur de $\#P$	$\sigma(\%)$		
				Min.	Max.	Moy.
20	4	10	6.6	21.80	662.50	182.20
	8	10	9.9	24.77	1281.00	256.90
	12	10	12.6	13.98	243.30	99.36
	16	10	14.3	99.50	1634.00	410.50
30	4	10	11.5	27.42	403.40	153.90
	8	10	16.0	19.55	540.50	108.00
	12	10	20.0	6.89	260.10	78.43
	16	10	22.2	8.21	756.80	137.70
40	4	10	13.3	37.62	5020.00	1408.00
	8	10	22.6	45.14	4699.00	559.60
	12	10	27.5	53.62	488.20	114.70
	16	10	30.2	21.29	636.40	122.70
50	4	10	17.9	28.25	4551.00	1400.00
	8	10	28.7	15.58	2863.00	632.70
	12	10	36.6	29.66	1634.00	410.50
	16	10	39.6	25.22	2349.00	487.10

Les résultats montrés dans le tableau 4.3 indiquent que la précision est assez faible. Par exemple, pour la série d'exemplaires $(40, 4)$, en moyenne l'estimation est 15 fois plus élevées (ou moins élevées) que la vraie valeur. Ceci n'est pas surprenant étant donné l'ordre de grandeur des valeurs à estimer (13.3 pour cette série). L'ordre de grandeur du nombre d'extensions linéaires est calculé par le logarithme de base 10 de $\#P$. Pour une estimation grossière du nombre d'extensions linéaires, la précision est acceptable en particulier si l'on s'intéresse uniquement à l'ordre de grandeur du nombre.

4.1.3 Densités des permutations

Les algorithmes exact et heuristique pour calculer les densités de permutations appartiennent à deux classes différentes. D'après l'analyse théorique, le temps d'exécution de l'algorithme exact croît exponentiellement avec la taille du poset, car dans le cas général la largeur dépend de la taille. Par contre, l'heuristique est un algorithme polynomial. Les résultats expérimentaux de la performance des algorithmes sont conduits différemment : pour l'algorithme exact, on montre les temps d'exécution [1], par contre, pour l'heuristique on montre la précision. Le but de présenter les temps de calcul de l'algorithme exact est de montrer, empiriquement, l'avantage de la mise à jour incrémentielle du nombre d'extensions linéaires par rapport au recalcul de ce dernier. Dans le cas heuristique, la compléxité des algorithmes est, en gros, dominée par la fermeture transitive de la relation d'ordre du poset et donc le temps de calcul serait négligeable par rapport au cas exact à partir d'une certaine taille. Pour cela, on se contente d'en montrer la précision en comparant avec les résultats du cas exact.

Tableau 4.4 Temps d'exécution de l'algorithme exact

$\|P\|$	$w(P)$	Temps d'initialisation(s)			Temps de mise à jour(s)		
		Min.	Max.	Moy.	Min.	Max.	Moy.
20	4	0.0004	0.0013	0.0007	0.000005	0.000019	0.000010
	8	0.0042	0.0122	0.0065	0.000086	0.000212	0.000122
	12	0.0304	0.0567	0.0410	0.000985	0.001947	0.001289
	16	0.2935	0.4491	0.3516	0.017990	0.031090	0.023110
30	4	0.0015	0.0041	0.0027	0.000014	0.000040	0.000027
	8	0.0130	0.0425	0.0256	0.000118	0.000602	0.000316
	12	0.1207	0.3565	0.2139	0.002276	0.008474	0.004714
	16	1.0020	3.0810	2.1350	0.037170	0.107500	0.070870
40	4	0.0016	0.0055	0.0030	0.000017	0.000038	0.000025
	8	0.0321	0.1319	0.0649	0.000312	0.001577	0.000719
	12	0.3333	2.2840	1.1400	0.005409	0.040800	0.019590
	16	4.5910	10.4900	8.0940	0.103700	0.264900	0.191100
50	4	0.0029	0.0107	0.0060	0.000026	0.000058	0.000035
	8	0.0603	0.2952	0.1610	0.000520	0.002896	0.001532
	12	1.1850	19.5400	6.8730	0.016160	0.344500	0.106700
	16	16.3100	129.4000	52.1200	0.376900	3.073000	1.150000

Le temps d'initialisation indiqué dans le tableau 4.4 est celui de la compilation de l'algorithme. Cette étape est exécutée une seule fois, ensuite les appels subséquents servent à mettre à jour la table des valeurs pour déduire la densité de permutations d'une paire d'éléments non

1. Processeur : Intel(R) Xeon(R) CPU E5530 @ 2.40GHz. Mémoire : 50Go

comparables. L'influence de la largeur sur le temps d'exécution est évidente et mieux visible à la figure 4.3.

Malgré sa complexité élevée, l'algorithme exact n'est pas complètement inutilisable. En effet, pour calculer uniquement le nombre d'extensions linéaires d'un poset ainsi que les densités des permutations pour une utilisation normale dans un temps raisonnable, on peut envisager cet algorithme pour des tailles de poset allant jusqu'à 50 ou plus en autant que la largeur ne dépasse pas 16. Par contre, dans le contexte de branchement, un temps d'initialisation de 52.12 secondes et une mise à jour de 1.15 secondes en moyenne pour la série $50 - 16$ est inacceptable. Pour cela, dans le contexte de branchement, on choisit d'utiliser l'algorithme exact pour les posets de taille $10 - 30$ et de largeur allant jusqu'à 10. Pour ces posets, le temps d'initialisation se compte en millisecondes et la mise à jour en microsecondes. Puisque la consommation en mémoire de l'algorithme exact est linéaire dans le nombre d'antichaînes du poset alors, ceci ne posera pas de problème pour une largeur limitée.

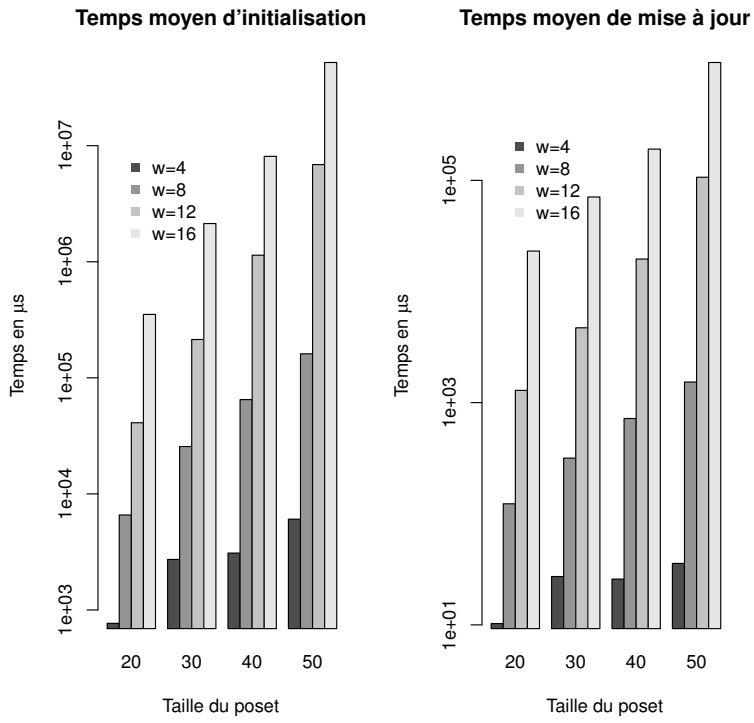


Figure 4.3 Influence de la largeur sur le temps d'exécution de l'algorithme exact

L'heuristique pour calculer les densités de permutations est polynomiale dans la taille du poset. En gros, cet algorithme est dominé par le temps de calcul de la fermeture transitive. Pour cela, pour les tailles de posets de $20-50$, le temps d'exécution est négligeable. Par contre,

Tableau 4.5 Mesure de corrélation du classement par l'algorithme heuristique

$\|P\|$	$w(P)$	Nb d'exempl.	Coefficient de corrélation			Même paire max. densité
			Min.	Max.	Moy.	
20	4	10	0.5425	0.8078	0.69	6/10
	8	10	0.5991	0.7939	0.6968	6/10
	12	10	0.6357	0.7748	0.7055	9/10
	16	10	0.5558	0.8083	0.7014	9/10
30	4	10	0.5654	0.7216	0.6444	9/10
	8	10	0.5538	0.6853	0.6233	6/10
	12	10	0.5996	0.7166	0.6741	8/10
	16	10	0.6104	0.714	0.6736	7/10
40	4	10	0.5165	0.64	0.5853	7/10
	8	10	0.5597	0.671	0.6292	5/10
	12	10	0.5891	0.7005	0.6492	6/10
	16	10	0.5885	0.691	0.6404	9/10
50	4	10	0.4633	0.7064	0.6209	6/10
	8	10	0.5539	0.6651	0.6132	5/10
	12	10	0.6005	0.687	0.6426	5/10
	16	10	0.6118	0.6899	0.6458	10/10

on s'intéresse à la précision du classement des densités des paires non comparables trouvé par l'algorithme. La comparaison est réalisée par la mesure du coefficient de corrélation entre le classement trouvé par l'algorithme exact et celui trouvé par l'heuristique. Il existe plusieurs mesures de coefficient de corrélation entre classements. Dans notre cas, une mesure pondérée est mieux appropriée, car dans le contexte de branchement les premiers branchements sont ceux qui comptent le plus. On utilise une mesure appelée "Apcorr" (Yilmaz et al., 2008) qui donne une valeur qui varie entre -1 et 1 ; 1 si les classements sont identiques et -1 sinon. Une implémentation de cette mesure est disponible pour le logiciel R.

D'après le tableau 4.5, les valeurs de corrélation indiquent que les classements par l'algorithme exact et heuristique ont une similarité apparente avec une tendance à diminuer lorsque le nombre de paires non comparables devient grand. Cependant, on constate que la première paire de densité maximum est souvent identique à celle choisie par l'algorithme exact. Encore mieux, cette propriété n'a pas l'air de diminuer avec l'augmentation du nombre de paires dans le posets. Par exemple, pour les posets de la série $50 - 16$ dont le nombre de paires est en moyenne de 907.5, l'algorithme heuristique a réussi à trouver 100% des paires de maximums désistés. Dans le contexte de branchement, on peut s'attendre à ce que les choix de branchements de l'heuristique et l'algorithme exact ne soient pas très différents, en supposant

qu'on n'a que des contraintes de ce type.

4.2 Application au problème d'ordonnancement "Job-Shop"

4.2.1 Description

Un exemplaire du problème Job-Shop est défini par n tâches (*Jobs*) et m machines. Chaque tâche est composée de m opérations chacune avec une durée connue qui s'exécutera sur chacune des machines selon une séquence fixe. L'objectif est de trouver un ordonnancement des tâches (opérations) sur les machines qui minimise le temps de terminaison appelé communément *makespan*. Dans notre cas, on s'intéresse à la version de satisfaction de contraintes, c.-à-d., pour une valeur de *makespan* donnée, on cherche un ordonnancement réalisable des opérations sur chacune des machines. L'ajout de la contrainte pour le *makespan* impose une fenêtre de temps pour chaque opération par la cohérence de borne. On peut donc voir le problème Job-Shop comme un ensemble de m contraintes de ressources unaires auxquelles s'ajoutent des contraintes pour les séquences d'opérations d'une même tâche.

Une solution à ce problème est obtenue en décidant de la permutation des tâches pour chaque machine. On n'a pas à se soucier des temps de départ, car, tel que mentionné au chapitre 2, pour une permutation donnée des tâches sur une ressource unaire, les temps de départ sont déduits facilement. Pour notre heuristique de branchement basée sur le dénombrement des permutations, il s'agit d'un cas d'utilisation typique, car on dispose uniquement de m contraintes dénombrables pour lesquelles on compte le nombre de permutations du graphe de préséance associé. La présence de contraintes temporelles, bien que non dénombrables dans le contexte proposé dans ce travail de recherche, elles influencent les préséances et par conséquent les permutations.

Le modèle utilisé pour Job-Shop est celui adopté par Grimes et al. (2009). Dans ce modèle, à chaque paire de tâches $1 \leq i < j \leq n$ est associée une variable booléenne b_{ij} pour chacune des m contraintes de ressource unaire pour un total de $m \times (n \times (n-1))/2$ variables. Soit p_i, p_j le temps de traitement respectivement des tâches i, j sur une même machine, les variables b_{ij} et les variables de temps de départs t_i, t_j pour cette machine sont reliées par les contraintes suivantes :

$$b_{ij} = \begin{cases} 0 \Leftrightarrow t_i + p_i \leq t_j \\ 1 \Leftrightarrow t_j + p_i \leq t_i \end{cases} \tag{4.1}$$

Les contraintes additionnelles pour les séquences des opérations d'une même activité sont

ajoutées au modèle.

4.2.2 Heuristiques de branchement

Pour les heuristiques de branchement basées sur les densités de permutations, on dispose de deux algorithmes : exact et heuristique. Les densités sont calculées en alternant entre les deux algorithmes selon la largeur du graphe de préséance associé à la machine pour laquelle on compte le nombre de permutations. En se basant sur le tableau 4.4 on choisit d'utiliser l'algorithme exact lorsque la largeur est inférieure ou égale à 10. Pour les valeurs de n, cela ne pose pas de problème, car les exemplaires de Job-Shop utilisés dans la littérature sont en général de taille $n \in [10, 50]$ pour le nombre de tâches. La largeur du graphe de préséance est calculée en utilisant l'algorithme d'appariement maximum dans un graphe biparti (Ford and Fulkerson, 1962).

Deux variantes d'heuristiques basées sur le dénombrement de solutions seront considérées : la première est l'heuristique maxSD. Selon le schéma de maxSD, les choix de branchements sont établis en calculant la densité de permutations pour chaque paire de tâches i, j dont la variable binaire correspondante b_{ij} n'est pas encore instanciée, et cela pour toutes les machines. La machine dont la variable est de densité maximum est choisie et le choix de branchement est fait pour l'assignation correspondante. On note que pour ce problème en particulier, on n'a pas à calculer les densités pour chaque valeur dans le domaine de la variable, il suffit de choisir celle dont le calcul est plus "facile" et déduire l'autre. Cela va de même pour le cas heuristique. La densité de l'assignation $b_{ij} = 0$ est considérée plus facile à calculer si $|D[i] \times U[j]| < |D[j] \times U[i]|$ sinon on choisit plutôt de calculer la densité pour $b_{ij} = 1$. En général, si la cardinalité du produit cartésien du bas ensemble de i et haut-ensemble de j est petite, alors ceci est indicateur que le nombre d'opérations nécessaires dans la mise à jour du nombre d'extensions linéaires est également moindre.

Dans la deuxième heuristique, en choisit la variable b_{ij} dont la somme des domaines de temps de départ de la paire de tâches i, j correspondante est minimum. L'idée est d'adapter l'heuristique dom à notre problème. En temps normal, c'est la variable de décision, dans notre cas b_{ij}, dont le domaine est minimum qui doit être considérée, mais puisque ces variables sont booléennes alors cela n'a pas de sens, car les variables non instanciées auront toutes la même taille. Pour cela on se tourne vers les domaines de temps de départ des activités correspondantes. Cette idée a été adoptée par Grimes et al. (2009) pour l'heuristique domWdeg. Dans notre cas, il s'agit de domMaxSD où le choix de valeur dans l'assignation est celui qui maximise la densité.

Parce que la performance de nos heuristiques de branchements pour ce modèle de Job-Shop

dépend du nombre de préséances, on décide d'ajouter une étape d'initialisation pour examiner les assignations non réalisables et ainsi retirer les valeurs correspondantes des domaines des variables. Il s'agit d'une opération connue par le terme anglais *probing*. Pour ce problème en particulier, le retrait de valeur du domaine d'une variable est équivalent à instancier la variable ce qui se traduit par l'ajout d'au moins une préséance. Si l'assignation de l'autre valeur est non réalisable, alors l'exemplaire du CSP l'est aussi. Cette technique est attrayante, mais coûteuse et ne peut être utilisée que dans une étape d'initialisation pour la même raison que IBS fait une mise à jour approximative de ses impacts au lieu de faire le recalcul à chaque fois.

Les deux heuristiques seront comparées à IBS et domDDeg. En raison de difficultés techniques à utiliser l'heuristique domWDeg, on utilise à la place domDDeg. En bref, cette heuristique choisit la variable qui contraint le plus grand nombre de variables non instanciées. Combinée à dom, dans ce cas, la variable est choisie selon le rapport dom/ddeg (Smith and Grant, 1997) mais dans notre cas, les domaines des variables non instanciées ont tous la même taille et donc seulement le degré ddeg qui influence son choix.

4.2.3 Jeu de données

Les exemplaires du problème Job-Shop sont disponibles dans http://www.cril.univ-artois.fr/ lecoutre/benchmarks.html. Il s'agit de la version CSP de certains des exemplaires du problème Job-Shop les plus utilisés dans la littérature notamment ceux de Taillard. Ici, les fenêtres de temps pour les variables de temps de départ des activités sont ajoutées. En gros, tous les domaines des variables sont contraints à une valeur maximum, en particulier, la dernière opération de chaque tâche. Trois valeurs maximum sont considérées : 95%, 100% et 105% de la valeur optimale (ou meilleure valeur connue) de chaque exemplaire auquel on retire le temps d'exécution de la dernière opération dans la séquence de chaque activité. Ainsi, pour chaque exemplaire, on obtient 3 CSPs dont un est non réalisable.

La description du jeu de données est montrée dans le tableau 4.6. Les exemplaires de la série "tai-15" sont ceux disponibles dans le lien mentionné ci-haut. Le reste des exemplaires sont obtenus en suivant la même approche. Parmi les exemplaires utilisés dans la littérature, on choisit uniquement celles dont les valeurs optimales sont connues. Les valeurs optimales sont données et prouvées dans ce lien http://bach.istc.kobe-u.ac.jp/csp2sat/jss/.

La dernière colonne du tableau 4.6 indique la proportion des exemplaires réalisables (SAT) et ceux non réalisables (UNSAT). Pour l'exécution, un temps limite de 1200s est accordé à chaque exemplaire. Le nombre d'exemplaires résolus par les différentes heuristiques ainsi que le nombre d'échecs (*backtrack*) et temps moyens sont montrés respectivement dans les

Tableau 4.6 Exemplaires pour le problème Job-Shop

Série d'éxmepl.	n	m	Nb d'éxempl.	SAT/UNSAT
la-10-5	10	5	15	10/5
la-15-5	15	5	12	8/4
la-20-5	20	5	15	10/5
la-10	10	10	13	10/3
la-15-10	15	10	15	10/5
la-20-10	20	10	9	6/3
la-30-10	30	10	9	6/3
la-15	15	15	15	10/5
orb-10	10	10	30	20/10
tai-15	15	15	30	20/10

tableaux 4.7 et 4.8.

Tableau 4.7 Nombre d'exemplaires résolus

Série d'éxempl.	maxSD		domMaxSD		IBS		domDDeg	
	SAT	UNSAT	SAT	UNSAT	SAT	UNSAT	SAT	UNSAT
la-10-5	1	0	10	4	10	5	10	5
la-15-5	0	0	3	0	7	0	1	0
la-20-5	0	0	4	0	2	0	0	1
la-10	1	0	9	3	10	3	8	3
la-15-10	0	0	1	2	2	1	0	0
la-20-10	0	0	0	0	0	0	0	0
la-30-10	0	0	0	0	0	0	0	0
la-15	0	0	1	2	0	0	0	1
orb-10	0	0	15	10	15	9	9	6
tai-15	0	0	4	2	3	0	1	2

Les résultats obtenus pour l'heuristique maxSD sont loin de nos attentes. L'heuristique a réussi difficilement à résoudre deux exemplaires parmi les 163 alors que les autres heuristiques ont réussi à résoudre un nombre respectable d'exemplaires en particulier l'heuristique domMaxSD et IBS. Cette situation nécessite une explication, car l'heuristique maxSD a prouvé son efficacité dans la résolution de problèmes CSP qu'ils soient réalisables ou pas (Pesant et al., 2012). On attribue les résultats inattendus de l'heuristique maxSD aux facteurs suivants :

Premièrement, tel qu'il a été mentionné dans le chapitre 2, les permutations des activités utilisées dans le calcul les densités ne sont pas toutes valides si l'on considère les temps

Tableau 4.8 Nombre d'échecs et temps moyen d'exécution pour les exemplaires résolus

Série d'exempl.	maxSD		domMaxSD		IBS		domDDeg	
	bkt	t(s)	bkt	t(s)	bkt	t(s)	bkt	t(s)
la-10-5	674346	377.16	73057	12.70	3805	0.53	769797	42.84
la-15-5	-	-	173439	48.34	77591	7.93	89976	6.50
la-20-5	-	-	4826	2.76	785149	318.06	1725639	538.07
la-10	1310736	965.27	7692	3.09	11872	3.72	313599	55.42
la-15-10	-	-	52921	48.40	691079	402.73	-	-
la-20-10	-	-	-	-	-	-	-	-
la-30-10	-	-	-	-	-	-	-	-
la-15	-	-	203745	271.40	-	-	392029	179.63
orb-10	-	-	211005	87.35	206203	60.73	1356821	144.11
tai-15	-	-	29464	31.78	199927	162.81	1714927	509.39

de départs des activités. Cependant, le schéma de maxSD ne nous oblige pas à prendre en considération les temps de départs dans le dénombrement des permutations, car il s'agit tout simplement de contraintes de différents types suivant le modèle disjonctif adopté. Mais puisque les permutations semblent avoir plus d'influence sur la direction de la recherche de solutions, alors on pensait que cela n'aurait pas un aussi grand impact. Or d'après les résultats obtenus, cette hypothèse est mise en doute.

Le premier facteur évoque une autre possibilité pouvant expliquer les résultats obtenus : il s'agit des contraintes temporelles. Normalement, dans le schéma de maxSD, on doit calculer les solutions pour toutes les contraintes, or pour ce problème, on ne dispose pas d'algorithme de dénombrement pour les contraintes temporelles, surtout que les variables de décision ne sont pas les variables de temps de départs, d'ailleurs, c'est la raison derrière l'adoption du modèle disjonctif. De plus, maxSD est souvent utilisée dans des problèmes où l'on ne dispose pas nécessairement d'algorithmes de dénombrement de solutions de toutes les contraintes du problème, pourtant les résultats sont satisfaisants. Il semble que pour le problème Job-Shop, les contraintes temporelles sont très critiques dans la direction de la recherche.

À ces facteurs s'ajoute le fait qu'au départ de la recherche, il n'y a pas beaucoup d'information à exploiter par l'heuristique, car les graphes de préséances associés aux machines ne montrent aucune "tendance" particulière. Les choix de branchement pris au départ seront alors faits de façon presque arbitraire ce qui influence grandement la recherche, car ces décisions sont prises en haut de l'arbre de recherche. Les heuristiques comme IBS, ou Wdeg sont des heuristiques adaptatives, et peuvent "corriger" le critère utilisé dans les prises de décisions dans les choix de branchement, alors que maxSD n'est pas une heuristique adaptative. Pour ce problème,

cette propriété serait cruciale.

L'heuristique domMaxSD donne de bons résultats, qu'on peut considérer les meilleurs, car le nombre d'exemplaires résolus est 70/163 et les nombres d'échecs moyens ainsi que le temps moyen sont en général meilleurs que le reste des heuristiques. Cependant, il faut noter que la "compétitivité" de cette heuristique est surtout attribuable au choix de variable b_{ij}, qui est établi par le calcul de $|D_i(t_i)| + |D_j(t_j)|$ où t_i est la variable temps de départ de l'activité i et D_i est le domaine de t_i. Le choix de valeurs est établi par maxSD, mais puisqu'on a seulement deux valeurs dans le domaine de b_{ij}, alors, un choix aléatoire aurait des résultats très semblables tels que montrés dans le tableau 4.9 à la colonne indiquée par domRandom. Notons que les résultats de cette heuristique sont obtenus sans aucune répétition.

Tableau 4.9 Nombre d'exemplaires résolus

Série d'éxempl.	maxWSD		domRandom		domMaxSD	
	SAT	UNSAT	SAT	UNSAT	SAT	UNSAT
la-10-5	6	3	10	4	10	4
la-15-5	4	2	5	0	3	0
la-20-5	1	0	3	0	4	0
la-10	0	0	9	5	9	3
la-15-10	0	0	0	2	1	2
la-20-10	0	0	0	0	0	0
la-30-10	0	0	0	0	0	0
la-15	0	0	0	1	1	2
orb-10	3	2	13	10	15	10
tai-15	0	0	4	2	4	2

Pour mettre en perspective l'influence de l'information obtenue à partir des domaines des variables de temps de départ des activités, on décide de pondérer l'heuristique maxSD dans le calcul de la densité d'une assignation $\sigma(b_{ij}, 0)$ par le rapport $(D'_i \times D'_j)/(D_i \times D_j)$ où D_i et D_j sont les domaines des variables temps de départ t_i et t_j de la paire d'activités (i, j) et $D'_i = min\{max\{D_i\}, max\{D_j\} - p_i\} - min\{D_i\}$ et $D'_j = max\{D_j\} - max\{min\{D_j\}, min\{D_i\} + p_i\}$. Les résultats de cette heuristique qu'on dénote par maxWSD aux tableaux 4.9 et 4.10 montrent une nette amélioration par rapport à maxSD standard.

Pour les heuristiques IBS et domDDeg, la première confirme son positionnement parmi les meilleures heuristiques dans la PPC. Pour ce problème, il a réussi à résoudre 68/163 exemplaires, mais un temps et nombre d'échecs plus élevés que domMaxSD. Probablement que si l'heuristique pouvait exploiter les domaines de temps de départ des activités, les résultats seraient meilleurs. La deuxième, domDDeg, donne des résultats moyens relativement aux

Tableau 4.10 Nombre d'échecs et temps moyen d'exécution pour les exemplaires résolus

Série d'exempl.	maxWSD		domRandom		domMaxSD	
	bkt	t(s)	bkt	t(s)	bkt	t(s)
la-10-5	532620	357.99	58381	16.83	73057	12.70
la-15-5	255078	282.42	1767	1.15	173439	48.34
la-20-5	20877	21.55	3	0.31	4826	2.76
la-10	-	-	11826	7.25	7692	3.09
la-15-10	-	-	91685	161.50	52921	48.40
la-20-10	-	-	-	-	-	-
la-30-10	-	-	-	-	-	-
la-15	-	-	93612	292.51	203745	271.40
orb-10	286715	347.91	168057	113.80	211005	87.35
tai-15	-	-	13067	32.32	29464	31.78

autres, mais cela était attendu, car cela aurait également mieux fonctionné si dom faisait référence au domaine de temps de départ plutôt que celui des variables binaires qui n'a aucune contribution ici.

CHAPITRE 5 CONCLUSION

Ce travail de recherche fut une première tentative dans l'application de l'heuristique de branchement basée sur les densités de solutions dans le contexte des problèmes d'ordonnancement notamment les problèmes d'ordonnancement disjonctif. Dans ce chapitre, on rappelle les principales contributions en particulier en ce qui concerne les algorithmes proposés pour le calcul des densités de permutations d'un ensemble partiellement ordonné. On mentionne également les limitations de notre solution ainsi que les perspectives futures.

5.1 Synthèse des travaux

Nous avons proposé une adaptation de l'algorithme de tri topologique d'un graphe orienté pour le calcul des extensions linéaires d'un poset. Cet algorithme est implanté de manière à permettre la mise à jour incrémentielle de la table des valeurs pour calculer la proportion des permutations dans lesquelles le rang d'un élément est supérieur à un autre qu'on a appelé densité de permutations. L'analyse asymptotique de cet algorithme montre que sa complexité dépend de la largeur du poset et peut donc être utilisé en autant que la largeur ne dépasse pas un certain seuil. Ce dernier varie selon le contexte d'utilisation.

Pour les posets de grande taille, un algorithme heuristique est proposé. Cet algorithme est basé sur l'examen de la cardinalité de la relation d'ordre du poset avant et après l'ajout d'une paire d'éléments non comparables à la relation. Les coefficients de corrélation calculés pour un échantillon de poset montrent que les classements des paires trouvés par les algorithmes exact et heuristique ont une similarité apparente en particulier la paire de densité maximum. Cet algorithme est utile ; car le nombre d'extensions linéaires d'un poset devient énorme à partir d'une certaine taille et sa représentation sur une machine devient imprécise voire même futile. Dans beaucoup de situations, on s'intéresserait plutôt aux densités de permutations des paires ; dans ce cas les résultats de l'heuristique sont satisfaisants.

Pour l'estimation du nombre d'extensions linéaires d'un poset, nous avons proposé une amélioration de l'algorithme de décomposition en chaînes vorace proposé par Cardinal et al. (2013) pour que le résultat du nombre de chaînes trouvées soit moindre. Ceci a comme effet d'améliorer la précision de l'entropie calculée et par conséquent donner une meilleure estimation du nombre d'extensions linéaires du poset en exploitant le résultat de Kahn and Kim (1995) qui montre le lien entre le nombre d'extensions linéaires d'un ensemble partiellement ordonné et l'entropie du graphe d'incomparabilité du poset.

Ces algorithmes sont utilisés dans le contexte de branchement en utilisant l'heuristique maxSD pour la résolution du problème Job-Shop par la PPC. Les résultats obtenus sont loin d'être compétitifs avec ceux des autres heuristiques de branchement comme IBS ou dom, mais nous mentionnons quelques facteurs possibles pouvant expliquer ces résultats, notamment la présence des contraintes temporelles pour lesquelles on ne dispose pas d'algorithmes de dénombrement de solutions. Par ailleurs, pour les problèmes dont les solutions sont des permutations, l'utilisation des algorithmes de dénombrement proposés serait, dans le contexte de l'heuristique maxSD, plus avantageuse. Un tel problème où une solution est exprimée par une permutation des variables est le fameux problème du voyageur de commerce (TSP). Bien entendu, il s'agit d'un problème pur qui pourrait être résolu plus efficacement par d'autres approches, mais en supposant qu'on dispose d'une combinaison de TSPs, alors le problème serait plus intéressant à résoudre par la PPC. En littérature, cette généralisation désigne une famille de problèmes connue sous le nom de "Problèmes de tournées de véhicules" qui est un domaine largement étudié.

5.2 Limitations de la solution proposée

Pour l'algorithme exact, sa limitation est claire. Il s'agit d'un algorithme qui est exponentiel dans la taille du poset et donc ne peut être utilisé que dans certains cas. Heureusement, pour les problèmes d'ordonnancement impliquant les ressources unaires, les solutions sont telles que le nombre de variables par ressource est relativement petit, par exemple, le problème de Job-Shop. Aussi, si on considère les problèmes de tournée de véhicules, alors même si le nombre de villes ou de clients à visiter est grand, celui de chaque véhicule serait relativement petit dans un horizon raisonnable.

Une autre limitation de l'algorithme exact est l'utilisation des opérateurs de décalage. L'algorithme tel que proposé ne peut pas être utilisé pour un poset de taille supérieure à 64. Par contre, cela peut être amélioré facilement en remplaçant les clefs des entrées dans la table par des mots (*string*).

Pour l'heuristique maxSD, dans le contexte du calcul des densités de permutation, une limitation se trouve dans le nombre élevé d'assignations dont les valeurs de densité sont potentiellement maximums en particulier pour les poset dont le nombre de paires incomparables est élevé. Par exemple, il n'est pas rare de se retrouver avec plusieurs assignations partageant les valeurs $0.9999\ldots$, dans ce cas, est-ce le choix d'une assignation dont la valeur de densité est 0.99997 est mieux qu'une autre dont la valeur est 0.99996 ? Surtout que le nombre de telles assignations est non négligeable.

5.3 Améliorations futures

L'algorithme exact pour le calcul des extensions linéaires peut être amélioré en utilisant la parallélisation. Par exemple, on peut utiliser deux processus pour compiler à travers les bas-ensembles et les haut-ensembles simultanément. Cela réduira le temps de calcul d'initialisation par 2, probablement le temps de la mise à jour aussi. Une autre amélioration possible est de calculer les permutations valides par rapport aux fenêtres de temps des activités. Cette étape ne sera pas difficile programmaticalement, mais le défi serait de faire la mise à jour incrémentielle. Un tel algorithme ouvrirait les portes à l'utilisation d'une version approximative en se basant sur les techniques utilisées dans ADP.

Pour l'estimation du nombre d'extensions linéaires d'un poset par entropie, on pense qu'une décomposition minimale et vorace en même temps donnerait encore une meilleure précision par rapport à l'algorithme proposé dans le chapitre 3. Une telle décomposition peut être obtenue en utilisant l'algorithme de couplage de poids maximum dans un graphe biparti valué. Le travail consisterait à trouver une pondération adéquate des arêtes du graphe.

Pour maxSD dans le contexte de l'ordonnancement disjonctif, il s'agit d'une heuristique qui a du potentiel. Ce travail a permis de constater qu'il serait difficile d'atteindre de bons résultats si on considère uniquement les permutations des activités. Les perspectives futures dans cette direction seraient de développer une technique permettant d'inclure les domaines des variables de temps de départ des activités dans le calcul des densités de permutations notamment par pondération de ces derniers. Une telle solution pourrait éventuellement être généralisée dans une version pondérée de l'heuristique maxSD impliquant les variables de décision et les variables dépendantes dans un CSP.

RÉFÉRENCES

J. Banks, S. Garrabrant, M. L. Huber, et A. Perizzolo, "Using tpa to count linear extensions", *arXiv preprint arXiv :1010.4981*, 2010.

P. Baptiste et C. Le Pape, "Edge-finding constraint propagation algorithms for disjunctive and cumulative scheduling", dans *Proceedings of the fifteenth workshop of the UK planning special interest group*, vol. 335. Citeseer, 1996, pp. 339–345.

C. Bessiere et J.-C. Régin, "Mac and combined heuristics : Two reasons to forsake fc (and cbj ?) on hard problems", dans *Principles and Practice of Constraint Programming—CP96*. Springer, 1996, pp. 61–75.

C. Biró et W. T. Trotter, "A combinatorial approach to height sequences in finite partially ordered sets", *Discrete Mathematics*, vol. 311, no. 7, pp. 563–569, 2011.

F. Boussemart, F. Hemery, C. Lecoutre, et L. Sais, "Boosting systematic search by weighting constraints", dans *ECAI*, vol. 16, 2004, p. 146.

D. Brélaz, "New methods to color the vertices of a graph", *Communications of the ACM*, vol. 22, no. 4, pp. 251–256, 1979.

G. Brightwell et P. Winkler, "Counting linear extensions", *Order*, vol. 8, no. 3, pp. 225–242, 1991.

S. Brockbank, G. Pesant, et L.-M. Rousseau, "Counting spanning trees to guide search in constrained spanning tree problems", dans *Principles and Practice of Constraint Programming*. Springer, 2013, pp. 175–183.

J. Cardinal, S. Fiorini, G. Joret, R. M. Jungers, et J. I. Munro, "Sorting under partial information (without the ellipsoid algorithm)", *Combinatorica*, vol. 33, no. 6, pp. 655–697, 2013.

A. A. Cire, W. J. van Hoeve *et al.*, "Mdd propagation for disjunctive scheduling." dans *ICAPS*, 2012.

D. Cordeiro, G. Mounié, S. Perarnau, D. Trystram, J.-M. Vincent, et F. Wagner, "Random graph generation for scheduling simulations", dans *Proceedings of the 3rd International ICST*

Conference on Simulation Tools and Techniques. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2010, p. 60.

R. P. Dilworth, "A decomposition theorem for partially ordered sets", *Annals of Mathematics*, pp. 161–166, 1950.

P. Edelman, T. Hibi, et R. P. Stanley, "A recurrence for linear extensions", *Order*, vol. 6, no. 1, pp. 15–18, 1989.

F. Focacci, P. Laborie, et W. Nuijten, "Solving scheduling problems with setup times and alternative resources." dans *AIPS*, 2000, pp. 92–101.

L. Ford et D. R. Fulkerson, *Flows in networks*. Princeton Princeton University Press, 1962, vol. 1962.

D. Grimes, E. Hebrard, et A. Malapert, "Closing the open shop : Contradicting conventional wisdom", dans *Principles and Practice of Constraint Programming-CP 2009*. Springer, 2009, pp. 400–408.

R. M. Haralick et G. L. Elliott, "Increasing tree search efficiency for constraint satisfaction problems", *Artificial intelligence*, vol. 14, no. 3, pp. 263–313, 1980.

M. Held et R. M. Karp, "A dynamic programming approach to sequencing problems", *Journal of the Society for Industrial and Applied Mathematics*, pp. 196–210, 1962.

J. Kahn et J. H. Kim, "Entropy and sorting", *Journal of Computer and System Sciences*, vol. 51, no. 3, pp. 390–399, 1995.

J. Körner, "Coding of an information source having ambiguous alphabet and the entropy of graphs", dans *6th Prague conference on information theory*, 1973, pp. 411–425.

J. K. Lenstra, A. R. Kan, et P. Brucker, "Complexity of machine scheduling problems", *Annals of discrete mathematics*, vol. 1, pp. 343–362, 1977.

G. Pesant, "Counting solutions of csps : A structural approach", dans *IJCAI*, 2005, pp. 260–265.

G. Pesant et C.-G. Quimper, "Counting solutions of knapsack constraints", dans *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. Springer, 2008, pp. 203–217.

G. Pesant, C.-G. Quimper, et A. Zanarini, “Counting-based search : Branching heuristics for constraint satisfaction problems.” *J. Artif. Intell. Res.(JAIR)*, vol. 43, pp. 173–210, 2012.

G. Pruesse et F. Ruskey, “Generating linear extensions fast”, *SIAM Journal on Computing*, vol. 23, no. 2, pp. 373–386, 1994.

P. Refalo, “Impact-based search strategies for constraint programming”, dans *Principles and Practice of Constraint Programming–CP 2004*. Springer, 2004, pp. 557–571.

B. M. Smith et S. A. Grant, “Trying harder to fail first”, *RESEARCH REPORT SERIES-UNIVERSITY OF LEEDS SCHOOL OF COMPUTER STUDIES LU SCS RR*, 1997.

G. Stachowiak, “The number of linear extensions of bipartite graphs”, *Order*, vol. 5, no. 3, pp. 257–259, 1988.

R. P. Stanley, “Two poset polytopes”, *Discrete & Computational Geometry*, vol. 1, no. 1, pp. 9–23, 1986.

P. Vilim, “O (n log n) filtering algorithms for unary resource constraint in : Proceedings of cpaior 2004”, *Nice, France*, 2004.

P. Vilím, “O (nlog n) filtering algorithms for unary resource constraint”, dans *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. Springer, 2004, pp. 335–347.

P. Vilım, “Global constraints in scheduling”, Thèse de doctorat, PhD thesis, Charles University in Prague, Faculty of Mathematics and Physics, Department of Theoretical Computer Science and Mathematical Logic, KTIML MFF, Universita Karlova, Malostranské námestı 2/25, 118 00 Praha 1, Czech Republic, 2007.

E. Yilmaz, J. A. Aslam, et S. Robertson, “A new rank correlation coefficient for information retrieval”, dans *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 2008, pp. 587–594.

A. Zanarini et G. Pesant, “Solution counting algorithms for constraint-centered search heuristics”, *Constraints*, vol. 14, no. 3, pp. 392–413, 2009.

ANNEXE A

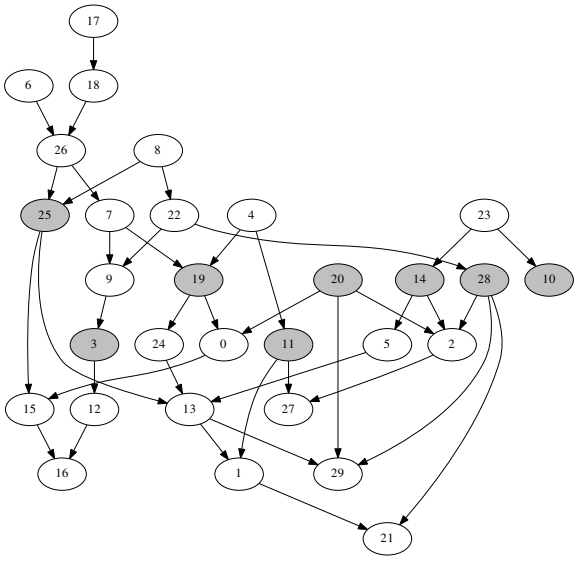


Figure A.1 Poset de taille 30 et largeur 8. En gris, les éléments de l'antichaîne maximum.

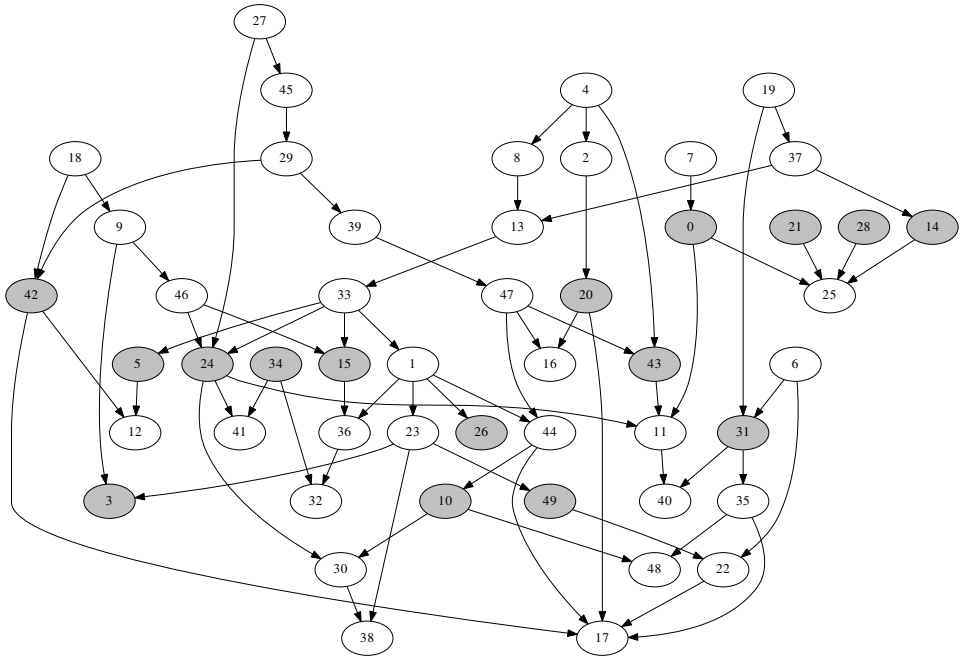


Figure A.2 Poset de taille 50 et largeur 16. En gris, les éléments de l'antichaîne maximum.