Méthodes d'agrégation et désagrégation de programmes linéaires en nombres entiers

Gaël Guillot

université de BORDEAUX

université de BORDEAUX

THÈSE PRÉSENTÉE

POUR OBTENIR LE GRADE DE

DOCTEUR
DE L'UNIVERSITÉ DE BORDEAUX

ECOLE DOCTORALE MATHÉMATIQUES ET INFORMATIQUE

Par **Gaël GUILLOT**

Méthodes d'agrégation et désagrégation de programmes linéaires en nombres entiers

Sous la direction de : **François CLAUTIAUX**
Co-directeur : **Boris DETIENNE**

Soutenue le 5 mars 2021

Membres du jury :

Mme. Sophie DEMASSEY	Maître assistant	CMA Mines ParisTech	Rapporteur
Mme. Safia KEDAD-SIDHOUM	Professeur des Universités	CNAM Paris	Rapporteur
Mr. Nabil ABSI	Professeur des Universités	Mines Saint-Etienne	Examinateur
Mr. Olivier BEAUMONT	Directeur de recherche	INRIA Bordeaux	Examinateur

Résumé

Dans cette thèse, nous nous intéressons à des problèmes pouvant se formuler sous la forme d'un programme dynamique et de contraintes linéaires additionnelles. Pour résoudre ces problèmes, des méthodes itératives d'agrégation de l'espace d'états ont été utilisées dans la littérature. Nous proposons un formalisme générique pour ces méthodes. Nous avons identifié une structure commune aux différentes méthodes, et mis en avant leurs ingrédients principaux. Le but de ce formalisme est d'unifier ces méthodes et de montrer les points clés pour que ces méthodes puissent être développées plus facilement pour de nouveaux problèmes. Pour permettre à ces méthodes d'être compétitives, plusieurs problématiques doivent être résolues : quelle relaxation choisir, quel espace d'état, comment raffiner la relaxation à chaque itération, quelle méthode de résolution choisir pour les relaxations, ... La difficulté de ces problématiques réside dans le fait que ces choix dépendent fortement du problème. Nous proposons des éléments qui peuvent être utilisés de manière générique, et d'autres spécifiques aux problèmes étudiés. Pour valider notre approche, nous avons appliqué l'une d'elles sur le problème de sac-à- dos temporel. La méthode *Successive Sublimation Dynamic Programming* a montré des résultats compétitifs avec la littérature, notamment grâce à un choix de dimension basé sur l'évolution du réseau, une énumération partielle des décisions possibles à chaque état et différents tests de dominances et de faisabilités. Nous comparons dans un deuxième temps les performances de notre implémentation générique avec une implémentation spécialisée sur ces problèmes. Enfin, ces méthodes ont été confrontées à un problème industriel moins classique : un problème d'utilisation de traitements phytosanitaires sur des vignes dans lequel les sous-problèmes peuvent être formulés comme des programmes dynamiques de grande taille.

Abstract

In this thesis, we are interested in problems that can be formulated in the form of a dynamic program and additional linear constraints. To solve these problems, iterative methods of state space aggregation have been used in the literature. We propose a generic formalism for these methods. We identified a common structure for the different methods, and highlighted their main ingredients. The aim of this formalism is to unify these methods and to show key points so that these methods can be developed more easily for new problems. To allow these methods to be competitive, several problems must be solved: which relaxation choose, which state space, how to refine relaxation at each iteration, which solving method to choose for the relaxations, ... The difficulty of these problems is in the fact that these choices strongly depend on the problem. We propose elements that can be used in a generic way, and others specific to the problems studied. To validate our approach, we applied one of them to the temporal knapsack problem. *Successive Sublimation Dynamic Programming* method has shown competitive results with the literature, notably thanks to a choice of dimension based on the evolution of the network, a partial enumeration of possible decisions at each state and different tests of dominance and feasibility. In a second step, we compare the performance of our generic implementation with a specialised implementation on these problems. Finally, these methods have been confronted with a less classical industrial problem: a problem of using phytosanitary treatments on vines in which the sub-problems can be formulated as large dynamic programmes.

Remerciements

Je tiens à remercier tout d'abord mes directeurs de thèse, François Clautiaux et Boris Detienne. Ils ont toujours su m'épauler et encadrer mon travail, dans les bons comme dans les mauvais moments. Je ne peux exprimer la gratitude et le respect que j'ai pour eux. Ces trois ans me laisseront un souvenir indélébile grâce à eux. Plus que des maîtres de thèse, vous êtes pour moi des modèles et des amis.

Je tiens aussi à remercier le cluster SysNum sans qui cette thèse n'aurait pas été possible.

Je remercie aussi Sophie Demassey et Safia Kedad-Sidhoum pour avoir accepté de rapporter cette thèse, ainsi que Nabil Absi et Olivier Beaumont qui ont accepté de faire partie du jury.

Je souhaite également remercier toute l'équipe RealOpt, et notamment Laurent Facq, grâce à qui toutes les expérimentations ont pu être possibles. Merci pour tout le temps que tu m'as accordé durant ces trois ans.

Merci aux chercheurs qui m'ont envoyé des résultats numériques : Fabio Furini, Enrico Malaguti, Timo Gschwind et Stefan Irnich.

Je souhaite également remercier toutes ma famille et mes amis, et plus particulièrement ma conjointe Audrey Tarricone et ma mère Denise Pontet qui m'ont toujours soutenu et sans qui je ne serais rien.

Enfin, je tiens à remercier tous ceux que j'ai oublié.

Table des matières

Chapitre 1

Introduction

Le champ scientifique de cette thèse se situe dans l'optimisation mathématique. Dans ce champ, on cherche une solution respectant un ensemble de contraintes et optimisant une ou plusieurs fonctions objectifs. Celui-ci regroupe différents types de problèmes : mono/multi-objectif, déterministe ou stochastique, discret ou continu, ... Nous nous intéressons dans ce manuscrit à des problèmes mono-objectif déterministes discrets. Pour résoudre ce type de problèmes, il existe de nombreuses approches. Ces méthodes de résolution peuvent être approchées (méta-heuristiques, algorithme génétiques, ...) ou exactes (branch-and-bound, programmation linéaire, programmation dynamique, ...).

La programmation dynamique introduite par [Bel54] est une méthode d'énumération basée sur les dominances. Cette méthode est basée sur des *états* qui contiennent les informations courantes nécessaires à la prise de décisions et des *transitions* permettant de passer d'un état à un autre, correspondant aux décisions possibles. Elle a été utilisée avec succès pour résoudre de nombreux problèmes, par exemple le problème de sac à dos ([Bal65, AF75, Tot80]), de partition de lots ("lot-sizing" : [WW58, GL88, WVHK92]) ou encore des problèmes de plus court chemin dans un graphe ([Bel58, Dre69, DS88]).

Dans de larges domaines d'application de l'optimisation combinatoire (transports, planification, découpe, ordonnancement...), la structure du problème repose sur la consommation/production de ressources limitées comme le temps ou des matières premières, rendant possible la modélisation via le paradigme de programmation dynamique. Cette modélisation se fait en intégrant les consommations de ressources dans les états du programme dynamique.

Dans ce manuscrit, on s'intéresse à la résolution de processus de décisions séquentielles pouvant se modéliser sous la forme d'un programme dynamique auquel on ajoute des contraintes linéaires additionnelles sur les séquences autorisées de décisions, correspondant par exemple à des contraintes de ressources ou de disjonctions. On se restreint aux programmes dynamiques finis qui peuvent se formuler en problèmes de chemin de plus grande valeur dans un graphe d'états. Ce type de problème générique permet de modéliser de nombreux problèmes de sac à dos, de plus court chemin sous contraintes, d'ordonnancement, de planification, ...

Plusieurs méthodes standard sont utilisées pour résoudre ce type de problème. Les méthodes d'énumération telles que le branch-and-bound sont souvent utilisées [BC89],

mais nécessitent de bonnes bornes pour le problème. Ces bornes sont obtenues en utilisant des relaxations, notamment la relaxation Lagrangienne. Cette relaxation permet de relâcher les contraintes linéaires additionnelles et de prendre en compte leur violation dans la fonction objectif. La relaxation peut être résolue en utilisant les outils classiques de la programmation linéaire ou des techniques de résolution gérant dynamiquement la taille des modèles obtenus. On peut notamment citer les techniques basées sur des reformulations de type Dantzig-Wolfe [DW60] ou les algorithmes de plan sécants [Gom60]. Dans ces approches, la formulation est tronquée, et ses variables et contraintes sont générées dynamiquement lorsqu'elles sont nécessaires à la résolution. Les algorithmes de labels [Min75, DS88] peuvent aussi être utilisés pour résoudre les programmes dynamiques avec contraintes de resources. Les labels correspondent à des chemins et contiennent les consommations de resources et les coûts des chemins. Comme les contraintes linéaires doivent être intégrées dans les labels, les dominances sont plus faibles et ces algorithmes deviennent moins performants.

D'un point de vue théorique, il est possible de reformuler un programme dynamique avec contraintes linéaires additionnelles comme un programme dynamique sans contraintes. Les contraintes linéaires additionnelles peuvent être ajoutées directement au programme dynamique en modifiant l'espace d'états. La formulation obtenue se fait souvent au prix d'un très grand nombre d'états des sous-systèmes, qui est généralement exponentiel en fonction du nombre de ressourcex, ou pseudo-polynomial en la consommation de ces ressources, ce qui interdit l'utilisation directe de ces formulations en pratique. D'autres méthodes (par exemple [CMT81] ou [ARP88]), moins utilisées en pratique, consistent à projeter l'espace d'états sur un espace de taille inférieure (agrégation de variables et de contraintes). Cela amène à résoudre successivement des relaxations de l'espace d'états. Nous nous intéressons plus particulièrement à ces méthodes dans ce manuscrit.

Plusieurs méthodes itératives d'agrégation ont été proposées dans la littérature. On peut notamment citer la méthode Successive Sublimation Dynamic Programming (SSDP) [Iba87], et la méthode Decremental State Space Relaxation [BDD06, RS08]. Le but de ces méthodes est de résoudre le problème en relâchant une partie des contraintes, et de réintégrer à chaque itération une partie de ces contraintes dans l'espace d'états. Ces méthodes peuvent être difficiles à utiliser en pratique car il est nécessaire d'ajouter différents algorithmes pour les rendre performantes, et sont souvent dépendantes du problème. De plus, ces méthodes ont été proposées indépendamment sous différents noms et utilisent des formalismes provenant de plusieurs domaines.

Ce manuscrit comporte plusieurs objectifs. Le premier est de proposer un formalisme pour les programmes dynamiques avec contraintes linéaires additionnelles ainsi que pour les méthodes itératives d'agrégation. Le deuxième objectif est de montrer que ces méthodes peuvent être utilisées et efficaces en pratique. Pour cela, nous identifions différents points clés de ces méthodes et apportons des solutions pour obtenir des résultats compétitifs. Le formalisme permet d'exprimer ces raffinements de manière générique. Ces méthodes sont implémentées dans une bibliothèque logicielle en développement dans l'équipe *RealOpt*.

Ce document est organisé de la manière suivante. Nous commençons en présentant un état de l'art concernant les programmes dynamiques avec contraintes linéaires additionnelles et les méthodes d'agrégation pour ces problèmes. Un formalisme permettant d'unifier ces notions et méthodes est introduit et permet de mettre en avant les éléments clés de ces méthodes difficiles à utiliser en pratique. Différents éléments permettant de rendre ces méthodes efficaces sont décrites en utilisant le formalisme précédent. Enfin, ces méthodes sont appliquées sur plusieurs problèmes et validées expérimentalement.

Dans le chapitre 2, nous rappelons quelques notions de recherche opérationnelle et présentons l'état de l'art de la recherche scientifique concernant les outils utilisés dans les chapitres suivants : programmation linéaire, optimisation dans les graphes ou relaxation Lagrangienne notamment. L'état de l'art se compose de quatre parties. La première partie présente les principaux outils nécessaires aux méthodes présentées dans le reste du manuscrit. La suite se compose d'une partie sur les programmes dynamiques avec contraintes linéaires additionnelles, une partie sur les méthodes d'agrégation permettant de réduire la taille de ces programmes dynamiques, et une partie sur les méthodes itératives d'agrégation de l'espace d'états. Nous explicitons les similitudes entre les différentes méthodes, et pointons les éléments qui diffèrent entre eux.

Le chapitre 3 est consacré à un formalisme unificateur pour les programmes dynamiques avec contraintes linéaires additionnelles. Le but est d'introduire un formalisme permettant d'exprimer les problèmes traités ainsi que les méthodes d'agrégation de l'espace d'états. Ce formalisme permet aussi de faire le lien avec les algorithmes de graphes, souvent utilisés dans la résolution de programmes dynamiques. Plusieurs méthodes itératives d'agrégation ont été proposées dans la littérature. Deux de ces méthodes (*SSDP* et *DSSR*) sont exprimées avec le formalisme, ce qui permet de voir les points communs et différences entre ces méthodes.

Pour que ces méthodes soient efficaces en pratique, plusieurs éléments doivent être pris en compte. Les raffinements généraux, qui ne dépendent pas du problème précis, sont exposés dans le chapitre 4. Des techniques permettant de supprimer des arcs dans les graphes représentant le programme dynamique étudié sont présentées, ainsi qu'une manière efficace de calculer une agrégation de l'espace d'états. Un des points clés des méthodes itératives d'agrégation est le choix des contraintes qui vont être intégrées à chaque itération. Ces contraintes sont sélectionnées parmi les contraintes violées par la solution de la relaxation, mais peuvent être nombreuses. Des critères permettant de savoir quelles dimensions vont être sélectionnées à chaque itération sont détaillés.

Le chapitre 5 est consacré à l'application de la méthode SSDP au problème de sac à dos temporel. Un programme dynamique original est proposé et différents tests de dominances et faisabilités permettant d'augmenter l'efficacité de la méthode sont détaillés. L'impact des tests et des différents critères de sélection de contraintes à ajouter est analysé à la fin du chapitre. L'ensemble des outils utilisés sur ce problème ont permis d'obtenir des résultats compétitifs avec ceux de la littérature.

Le but de nos méthodes génériques est de pouvoir appliquer les méthodes d'agrégation sur différents problèmes. Le chapitre 6 montre l'application de SSDP sur deux problèmes différents. Le premier est un problème d'ordonnancement sur lequel les auteurs de [TFA09] ont appliqué la méthode SSDP. Nous montrons les différents raffinements

proposés par [TFA09], et les exprimons avec notre formalisme. La comparaison des résultats obtenus avec notre framework aux résultats obtenus avec le framework spécifique de [TFA09] permet de voir les limites de notre implémentation actuelle. La deuxième partie de ce chapitre est consacrée à un problème industriel de traitements phytosanitaires de vignes. Le programme dynamique correspondant au problème est exposé et nous montrons les résultats obtenus sur ce problème complexe.

Le dernier chapitre propose des conclusions aux travaux réalisés et ouvre des perspectives pour de futurs travaux.

Chapitre 2

Etat de l'art

2.1 Introduction

Dans ce manuscrit, nous nous intéressons à des programmes dynamiques finis auxquels sont ajoutés des contraintes linéaires additionnelles. Il existe plusieurs méthodes standards pour résoudre ce type de problème : branch-and-bound, algorithmes de labels, ... Dans ce chapitre, nous présentons l'état de l'art concernant les méthodes d'agrégation de programmation dynamique. Ces méthodes, moins utilisées dans la littérature, s'appuient sur les méthodes standards pour résoudre une succession de relaxations.

Cet état de l'art se décompose en 5 sections. La première section permet d'introduire quelques notions basiques en recherche opérationnelle. La deuxième section est consacrée aux programmes dynamiques avec contraintes linéaires additionnelles. La troisième section présente une méthode de résolution pour ces problèmes. La quatrième section est consacrée aux méthodes d'agrégation en programmation dynamique. Enfin, la cinquième section est dédiée aux méthodes itératives. Les notions et algorithmes sont définis précisément et formellement dans le chapitre suivant.

2.2 Quelques bases de recherche opérationnelle

Dans cette section, nous décrivons des notions importantes de la recherche opérationnelle. Cette section permet de poser des notations et de comprendre des notions utilisées dans ce manuscrit.

2.2.1 Programmation linéaire

La programmation linéaire [Dan51] est l'étude des problèmes d'optimisation linéaire, composés d'une fonction objectif linéaire et d'un ensemble de contraintes linéaires. Le but est de trouver une solution qui minimise/maximise la fonction objectif et qui respecte l'ensemble des contraintes linéaires. Ces problèmes sont résolus régulièrement en utilisant l'algorithme du simplexe ou les méthodes de points intérieurs. Nous présentons l'écriture matricielle des problèmes d'optimisation linéaire. Soit n le nombre de variables et m le

nombre de contraintes du problème. On pose $\mathbf{x} \in \mathbb{R}^n$ un vecteur de n variables continues, deux vecteurs $\mathbf{c} \in \mathbb{R}^n$ et $\mathbf{b} \in \mathbb{R}^n$ correspondant aux coefficients des variables dans la fonction objectif et aux parties constantes des contraintes linéaires et une matrice $A \in \mathbb{R}^{m \times n}$ de coefficients dans les contraintes linéaires.

$$\max \mathbf{c}^T \mathbf{x} \tag{2.1}$$
$$\text{s.t. } Ax \leq \mathbf{b} \tag{2.2}$$

Les méthodes de points intérieurs permettent de résoudre en temps polynomial ces problèmes. Quand on ajoute des contraintes d'intégrité sur les variables de décisions, les problèmes deviennent NP-difficiles dans le cas général. On parle alors de *programmation linéaire en nombres entiers.*

$$\max \mathbf{c}^T \mathbf{x} \tag{2.3}$$
$$\text{s.t. } Ax \leq \mathbf{b} \tag{2.4}$$
$$x \in \mathbb{Z} \tag{2.5}$$

Une partie des problèmes de programmation linéaire en nombres entiers est facile à résoudre. C'est notamment le cas si la matrice de contraintes A est totalement unimodulaire (TU), c'est à dire que chaque sous-matrice carrée a un déterminant égal à 0,1 ou -1. On peut alors résoudre la relaxation linéaire du problème, car les points extrêmes du polyèdre de solutions sont entiers si la matrice de contraintes est TU.

2.2.2 Relaxation Lagrangienne

La relaxation Lagrangienne permet de relâcher une partie des contraintes, et de prendre en compte leurs violations dans la fonction objectif. Etant donné un programme linéaire décrit précédemment (2.3), on sépare la matrice de contraintes en deux matrices. Soit $m_1, m_2 \in \mathbb{R}, m_1 + m_2 = m$, on définit deux matrices de contraintes $A_1 \in \mathbb{R}^{m_1 \times n}$ et $A_2 \in \mathbb{R}^{m_2 \times n}$, ainsi que les vecteurs $b_1 \in \mathbb{R}^{m_1}$, $b_2 \in \mathbb{R}^{m_2}$ correspondant aux parties constantes des contraintes linéaires.

$$\max \mathbf{c}^T \mathbf{x} \tag{2.6}$$
$$\text{s.t. } A_1 \mathbf{x} \leq \mathbf{b_1} \tag{2.7}$$
$$A_2 \mathbf{x} \leq \mathbf{b_2} \tag{2.8}$$
$$\mathbf{x} \in \mathbb{Z} \tag{2.9}$$

Dans cette formulation, la matrice A_1 contient en général les contraintes "faciles", et A_2 les contraintes "difficiles". Soit $\pi \in \mathbb{R}_+^{m_2}$ un vecteur de multiplicateur de Lagrange, le Lagrangien est défini par :

$$L(\mathbf{x}, \pi) = \mathbf{c}^T \mathbf{x} + \pi(A_2 \mathbf{x} - \mathbf{b}_2) \tag{2.10}$$

On obtient de ce Lagrangien la relaxation Lagrangienne associée aux multiplicateurs π :

$$\max L(\mathbf{x}, \pi) = \mathbf{c}^T\mathbf{x} - \pi(A_2\mathbf{x} - \mathbf{b}_2) \tag{2.11}$$
$$\text{s.t. } A_1\mathbf{x} \leq \mathbf{b}_1 \tag{2.12}$$
$$\mathbf{x} \in \mathbb{Z} \tag{2.13}$$

La résolution du problème Lagrangien permet d'obtenir une borne duale du problème (correspondant à une borne supérieure en maximisation), qui est au moins aussi bonne que celle obtenue en utilisant la relaxation linéaire ([Geo74]). En relâchant les contraintes difficiles, on peut aussi obtenir un problème plus facile à résoudre (par exemple si la matrice A_1 est TU).

Le Lagrangien permet de définir le problème d'optimisation :

$$\max_{\pi \geq 0} \min_{\mathbf{x}} L(x, \pi)$$

appelé *problème dual Lagrangien*. Ce problème consiste à trouver le vecteur de multiplicateurs correspondant aux pénalités optimales.

Le problème dual Lagrangien est convexe et non-différentiable, plusieurs algorithmes sont utilisés dans la littérature pour approximer cette fonction (Sous-gradient, Volume, génération de colonnes, génération de contraintes, ...). Ces méthodes sont détaillées dans [WN99].

Dans ce manuscrit, nous utilisons principalement deux algorithmes pour approximer le problème dual Lagrangien. Le premier algorithme utilisé est l'algorithme de sous-gradient. Cet algorithme peut-être vu comme une généralisation de l'algorithme de gradient. Soit $f : \mathbb{R}^n \to \mathbb{R}$ une fonction convexe à minimiser. Un vecteur $\mathbf{g}$ est un sous-gradient de f en $x \in \mathbb{R}^n$ si $\forall y \in \mathbb{R}^n$, $f(y) \leq f(x) + \mathbf{g}^T.(y - x)$. L'algorithme de sous-gradient construit à partir d'un point $x^0 \in \mathbb{R}^n$ une séquence de points $x^k \in \mathbb{R}^n$ avec :

$$x^{k+1} = x^k - \delta^k g_k$$

avec δ^k la longueur du pas. Il existe plusieurs façons de calculer le pas, le plus classique est d'utiliser $\delta^k = \frac{f(x^k) - UB}{||g^k||^2}$ avec UB une borne primale du problème. Un exemple d'un autre pas est donné par [Pol69], et une analyse des propriétés de convergence de l'algorithme est donné par [AW09].

Le deuxième algorithme utilisé est l'algorithme de *Volume* [BA00]. Cet algorithme est une extension de l'algorithme de sous-gradient, et est en pratique plus efficace sur un grand nombre de problèmes. La différence majeure entre les deux algorithmes est que Volume construit une suite de solutions primales basée sur une combinaison convexe des solutions précédentes. Chaque solution primale $\tilde{x}^k$ est obtenu en utilisant :

$$\tilde{x}^k = \alpha x^k + (1 - \alpha)\tilde{x}^{k-1}, k > 1$$

avec α un paramètre entre 0 et 1. Ces méthodes permettent d'obtenir un vecteur de multiplicateurs correspondant à l'approximation du problème dual Lagrangien.

2.2.3 Graphe et plus court chemin

Le problème de plus court chemin dans un graphe orienté est un problème classique de la théorie des graphes. Il existe différents algorithmes de différentes complexités, utilisables sous certaines conditions. Soit $G = (V, E)$ un graphe avec n le nombre de sommets et m le nombre d'arcs, $s \in V$ le sommet de départ des chemins et c une fonction de coût sur les arcs. Les algorithmes sont basés sur une étiquette de distance sur chaque sommet, appelée *label*, correspondant au plus court chemin entre le sommet s et le sommet étiqueté.

Si le graphe G est acyclique, on peut utiliser *l'algorithme de Bellman* (complexité : $O(m)$). Le principe de cet algorithme est d'affecter à chaque sommet un label (en commençant par fixer le label du sommet source à 0) et d'améliorer itérativement ces labels en parcourant les sommets dans un ordre topologique. Pour chaque sommet visité, on fixe le label correspondant et on met à jour les labels des sommets voisins. On note $d(v), \forall v \in V$ le label associé au sommet v, et $c_{ij}, \forall (i,j) \in E$ le coût de l'arc (i,j). L'algorithme de Bellman est le suivant :

Algorithme 1 : Algorithme de Bellman

```
Calculer un ordre topologique σ des sommets de V tel que σ(s) = 1
d(s) ← 0; d(i) ← +∞, ∀i ∈ V \ {s};
pour p allant de σ(s) à n faire
    i ← σ^{-1}(p) (noeud en position p) ;
    pour tout j tel que (i, j) ∈ E faire
        si d(j) > d(i) + c_ij alors
            d(j) ← d(i) + c_ij ;
```

Cet algorithme est utilisable même si les coûts ne sont pas tous positifs. Il peut donc être utilisé pour calculer le plus long chemin dans un graphe (en multipliant tous les coûts par -1 et en calculant le plus court chemin).

Si les coûts des arcs sont tous positifs, alors on peut utiliser l'algorithme de Dijkstra, même si le graphe n'est pas acyclique (complexité : $O(n \log n \times m)$). Les sommets sont séparés en deux groupes : ceux qui ont été étiquetés de façon permanente, et ceux qui ont encore une étiquette temporaire. A chaque itération, on choisit le sommet parmi les étiquettes temporaires qui a le label le plus petit et on l'ajoute à l'ensemble des permanents. On met à jour ces voisins et on recommence jusqu'à ce que tous les labels

soient permanents. L'algorithme de Dijkstra est le suivant :

Algorithme 2 : Algorithme de Dijkstra

```
1  Perm ← ∅; Temp ← N
2  d(s) ← 0; d(i) ← +∞, ∀i ∈ V \ {s};
3  tant que Temp n'est pas vide faire
4      i* ← arg min d(i) : i ∈ Temp ;
5      Temp ← Temp \ {i*} ;
6      Perm ← Perm ∪ {i*} ;
7      pour tout j tel que (i*, j) ∈ E faire
8          si d(j) > d(i*) + c_{i*j} alors
9              d(j) ← d(i*) + c_{i*j} ;
```

2.3 Programmation dynamique

La programmation dynamique est une méthode de résolution introduite par Richard Bellman en 1954 [Bel54]. Il étudie les processus de décisions séquentiels et propose une méthode de résolution basée sur le principe d'optimalité.

Principe d'optimalité [Bel54] : " *An optimal policy has the property that whatever the initial state and initial decisions are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.*"

Le système étudié est défini par des *états* et des *transitions*. A chaque étape du processus séquentiel, une décision doit être prise. Un état regroupe les informations nécessaires à la prise de décision. Une décision affecte un état du système et correspond à la transformation d'un état en un autre, appelée transition. Le but est de maximiser le profit d'une séquence de transitions, appelée *politique*, d'un état initial vers un ou plusieurs états terminaux. Le principe d'optimalité permet de dire que pour une séquence de transitions optimale, n'importe quelle sous-séquence de transitions vers un état est optimale pour ce sous-problème. La fonction de récurrence définissant un programme dynamique découle directement de ce principe d'optimalité.

La formule de récurrence peut être décrite de manière plus formelle en utilisant des fonctions mathématiques. Le travail de [CMT81] permet d'introduire cette formule dans le sens "forward", c'est-à-dire en partant de l'état initial. Soit un programme dynamique défini par un ensemble d'états $\mathcal{L}$, un ensemble de transitions $\mathcal{T}$, $\ell^S \in \mathcal{L}$ l'état initial du programme dynamique et $\ell^F \in \mathcal{L}$ l'état terminal. Sans perdre de généralité, on considère que les programmes dynamiques ont un seul état terminal : s'il y a plusieurs états terminaux, on peut toujours ajouter des transitions entre chaque état terminal et un état terminal fictif. On définit la fonction $f_0(\ell), \forall \ell \in \mathcal{L}$ qui correspond au coût de la meilleure séquence de transitions entre ℓ^S et ℓ. On note $\psi(\ell), \forall \ell \in \mathcal{L}$ l'ensemble des états atteignables depuis l'état ℓ en appliquant une transition, et $\psi^{-1}(\ell) = \{\ell' | \ell \in \psi(\ell')\}$ l'ensemble des états qui atteignent ℓ. Le coût de passer de l'état ℓ à ℓ' est noté $v(\ell, \ell')$

(avec $v(\ell, \ell') = -\infty$ si $\ell' \notin \psi(\ell)$).

La récursion "forward" s'écrit :

Formule de récurrence forward

$$f_0(\ell) = \max_{\ell' \in \psi^{-1}(\ell)} \{f_0(\ell') + v(\ell', \ell)\} \tag{2.14}$$

avec la convention $f_0(\ell^S) = 0$.

Le coût de la solution de plus grand coût du programme dynamique est égal à $f_0(\ell^F)$.

La programmation dynamique est devenue une méthode de référence en optimisation. Différents formalismes ont été proposés, provenant de différents domaines. On peut notamment citer le formalisme de [KH67] basé sur les automates, ou celui de [CPRS01] basé sur les arbres de décision déterministes. Il existe de nombreux ouvrages présentant la programmation dynamique. On peux notamment citer [CC81], [Den82], [Kau67] et [Neu93] qui introduisent le principe de programmation dynamique.

Il existe plusieurs manières de modéliser un programme dynamique de type (2.14). Dans la littérature, on utilise fréquemment le *graphe de transition* pour représenter ce type de problème. La résolution du programme dynamique peut être vue comme un problème de plus long chemin dans un graphe. Cela implique qu'il est possible d'utiliser la programmation linéaire en nombres entiers pour exprimer un programme dynamique.

Le lien entre programme dynamique et problème de graphes a été présenté par plusieurs auteurs dans la littérature. La réduction d'un programme dynamique en un problème de plus court chemin dans un graphe pondéré est présentée par [Kau67]. Les auteurs montrent que les programmes dynamiques correspondant aux processus séquentiels peuvent être représentés comme des problèmes de transport dans un réseau. Cette réduction est reprise par [Mar76]. On retrouve cette réduction dans [Mor82] qui met en avant le problème majeur : la taille du graphe dépend du nombre d'états du programme dynamique, qui peut être exponentiel en la taille de l'état. Cette difficulté est aussi notée par [AMO88] où les auteurs montrent les liens entre programme dynamique et plus court chemin. L'avantage de cette technique de résolution est qu'il existe des algorithmes de complexité polynomiale en la taille du graphe si il ne contient pas de circuit de coût négatif.

Pour construire le graphe associé au programme dynamique, on associe chaque état du programme dynamique à un sommet du graphe. Il existe un arc entre deux sommets s'il existe une transition entre les deux états représentés par les sommets. Un arc correspond donc à une transition du programme dynamique. Le coût associé à un arc est le coût de la transition qu'il représente. Le problème de trouver une séquence de transitions entre l'état initial et l'état terminal qui maximise le profit est équivalent à trouver le plus long chemin entre le sommet représentant l'état initial et un des sommets représentant un état terminal dans le graphe associé. Quand il existe plusieurs états terminaux, on peut ajouter un arc artificiel entre chaque sommet représentant un état terminal et un sommet puits fictif. Le problème devient un plus long chemin entre le sommet source

et le sommet puits artificiel. Notons que le graphe associé à un problème séquentiel est acyclique, on peut donc utiliser l'algorithme de Bellman classique (2.2.3) pour résoudre le plus long chemin dans ces graphes.

Nous allons illustrer cette formulation sur un exemple en utilisant un problème classique de sac à dos binaire.

Exemple 1 (Sac à dos binaire)**.** *On considère un problème de sac à dos binaire où on doit trouver le sous-ensemble d'objets qui maximise le profit donné par chaque objet tout en respectant une contrainte de ressource. Définissons une instance de sac à dos binaire où la taille maximale du sac à dos est $W = 4$, et le nombre d'objets est $n = 3$. Chaque objet $i \in \{1, ..., n\}$ est défini par (w_i, p_i), avec w_i le poids de l'objet i et p_i le profit de l'objet i. L'ensemble des objets de l'instance est le suivant : $\{(2, 3); (2, 4); (4, 6)\}$.*

Un état du programme dynamique est un couple (i, w), où la dimension i correspond au dernier objet pouvant être pris, et la dimension w correspond à la place disponible dans le sac à dos. La formule de récurrence est la suivante :

$$\forall (i,w) \in \mathbb{N}^2, \alpha((i,w)) = \begin{cases} \max\{\alpha((i+1,w)), \alpha((i+1, w-w_{i+1})) + p_{i+1}\} & \text{si } i < n, w \geq w_{i+1} \\ \alpha((i+1,w)) & \text{si } i < n, w < w_{i+1} \\ 0 & \text{si } i \geq n \end{cases}$$

L'état initial du programme dynamique est $\ell^S = (0, W)$, et la valeur optimale est donnée par $\alpha(\ell^S)$.

Ce programme dynamique peut se formuler comme un problème de plus long chemin dans le graphe de la figure 2.1

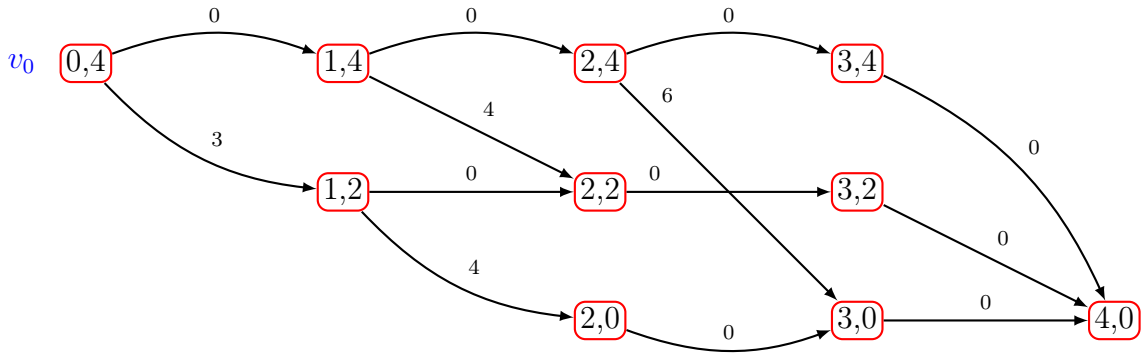


FIGURE 2.1 – Graphe du programme dynamique de l'exemple 1

Les outils de la programmation linéaire peuvent être utilisés dans le cadre de la programmation dynamique. Pour modéliser ce programme linéaire en nombres entiers, une méthode simple est d'utiliser la formulation précédente d'un programme dynamique sous forme de plus long chemin dans un graphe.

On note $\mathcal{L}$ l'ensemble des états du programme dynamique, $\mathcal{T}$ l'ensemble des transitions du programme dynamique, $x_a, a \in \mathcal{T}$, la variable binaire égale à 1 si la transition a est sélectionnée dans la solution, et $v(a)$ le coût de la transition.

$$\max \sum_{a \in \mathcal{T}} x_a * v(a) \tag{2.15}$$

$$\text{s.t.} \sum_{a \in \psi^{-1}(\ell)} x_a - \sum_{a \in \psi(\ell)} x_a = \begin{cases} -1 & \text{si } \ell = \ell^S \\ 0 & \text{si } \ell \in \mathcal{L} \setminus \{\ell^S, \ell^F\} \\ 1 & \text{si } \ell = \ell^F \end{cases}, \qquad \forall \ell \in \mathcal{L} \tag{2.16}$$

$$x_a \in \{0, 1\} \ , \forall a \in \mathcal{T} \tag{2.17}$$

Remarque 1. *Les travaux de [AMO88] ont permis de montrer que la matrice de contraintes correspondant à* (2.16)-(2.17) *est totalement unimodulaire, et donc tout les points extrêmes du polyèdre de solution sont entiers. On peut donc relâcher les contraintes d'intégralité, car les solutions optimales de la relaxation linéaire correspondent à des solutions entières.*

2.4 Méthode de résolution pour le problème de plus long chemin avec contraintes de ressources

Nous avons vu dans la section précédente comment modéliser un programme dynamique sous forme de plus long chemin dans un graphe. Les problèmes traités dans ce manuscrit se présentent sous la forme d'un programme dynamique auquel des contraintes linéaires sont ajoutées. Nous supposons que ces contraintes linéaires sont des contraintes d'inférioriété pour faciliter l'expression de certaines notions. Une séquence de transitions est réalisable si la somme des consommations des transitions de la solution est inférieure à une consommation maximale pour chaque ressource. Ces contraintes de ressources correspondent à des contraintes linéaires.

On peut remarquer que les programmes dynamiques avec contraintes linéaires additionnelles peuvent être formulés comme des problèmes de plus long chemin avec contraintes de capacités. Les auteurs de [BC89] formalisent ce problème et présentent des méthodes inspirées des algorithmes de labels pour résoudre les problèmes de plus long chemin classiques. Les labels correspondant au plus long chemin depuis la source sont remplacés par des labels plus complexes correspondant à un chemin depuis la source. Un label est donc composé du coût du chemin et pour chaque contrainte de capacité, la consommation de cette ressource sur le chemin. Contrairement aux algorithmes de plus long chemin classiques, plusieurs labels peuvent être attachés à un sommet, correspondant à des chemins différents depuis la source. Nous présentons dans cette section une de ces méthodes : le *label setting*.

Un algorithme avancé de label setting est présenté par [DDS92] et repris plus tard par [BDD06]. Ces travaux s'appuient sur les algorithmes proposés par [Min75] et [DS88] qui ont proposé une première version du label setting.

L'algorithme se base sur un graphe support. Une manière de construire ce graphe support est de considérer le graphe obtenu par la réduction du programme dynamique

initial (sans les contraintes linéaires additionnelles) en un problème de plus long chemin vu précédemment. On associe à chaque nœud du graphe support un ensemble de labels (appelé *bucket*). Chaque label correspond à un chemin entre la source et le nœud, et contient la consommation de ressources du chemin pour chaque contrainte. Un coût correspondant au coût du chemin est associé au label. L'algorithme de "label setting" est récursif : on ajoute un label sur le nœud source et on étend les labels en appliquant les transitions contenues sur les arcs aux labels, en traitant les nœuds dans un ordre topologique du graphe.

Un point important de l'algorithme est la notion de dominance. Deux labels d'un même bucket correspondent à des chemins différents. Un label domine un autre si son coût est supérieur au coût de l'autre label et pour chaque ressource, sa consommation est inférieure ou égale (pour une contrainte de type $\leq$) à la consommation de l'autre label. Un label dominé peut être retiré car il correspond à un sous-chemin qui ne peut être dans une solution optimale.

Soit $G_s = (V_s, E_s)$ le graphe support avec v_0 le sommet source du graphe. On définit $\mathscr{L}(v), \forall v \in V_s$ le bucket de labels correspondant au sommet v.

L'algorithme de label setting est résumé dans le pseudo-code suivant.

Algorithme 3 : Label Setting

```
1 Initialisation : Ajouter le label initial à l'ensemble 𝓛(v_0).
2 pour chaque sommet v dans un ordre de topologique faire
3     pour chaque label l ∈ 𝓛(v) faire
4         pour chaque arc e = (v, v') sortant de v faire
5             Extension : créer le label l' en appliquant à l la transition
              correspondant à e ;
6             Traitement du nouveau label :
7             Si l' est réalisable et n'est pas dominé par un label de 𝓛(v'), alors
              ajouter l' à 𝓛(v') ;
8             Retirer de 𝓛(v') tous les labels dominés par le nouveau label ;
```

La figure 2.2 illustre l'algorithme de label setting appliqué à l'exemple 1, avec une ressource additionnelle. On pose $r_1 = 4, r_2 = 3, r_3 = 1$ la consommation de la ressource supplémentaire de chaque objet, et $b_r = 6$ la consommation maximale.

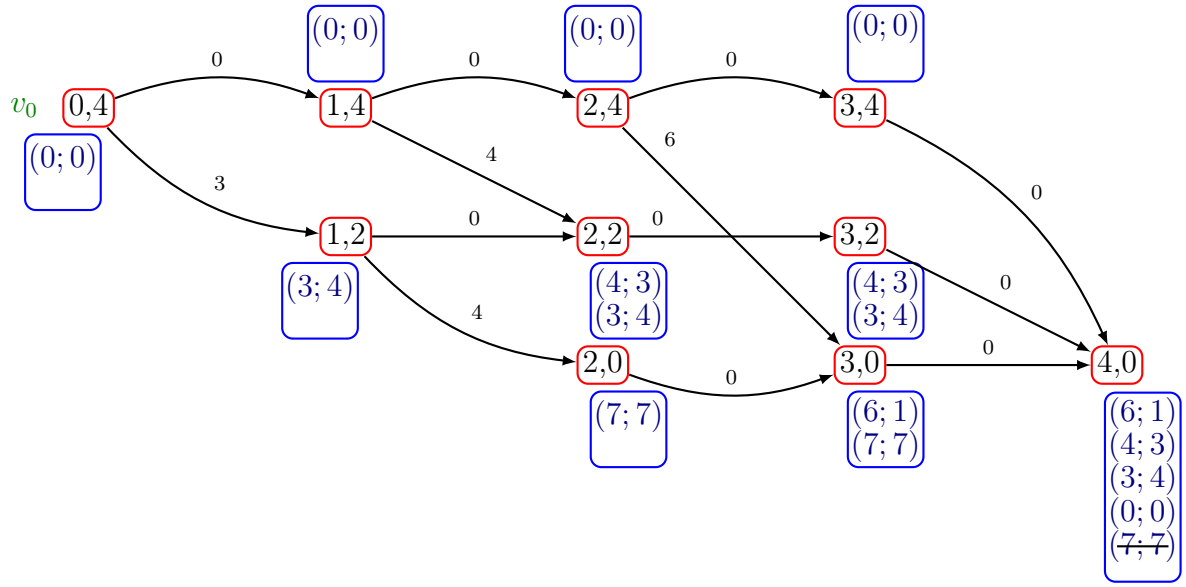


FIGURE 2.2 – Illustration de l'algorithme de label setting sur l'exemple 1 avec une contrainte de ressource supplémentaire. Pour chaque sommet, le bucket correspondant est représenté en bleu. Les labels sont notés $(c; r)$ avec c le coût du label et r la somme des consommations de la ressource supplémentaire. Les labels barrés correspondent à des labels non réalisables pour la ressource supplémentaire.

On remarque que dans le bucket du sommet correspondant à l'état $[2, 2]$ de la figure 2.2, le label $(3; 4)$ est dominé par le label $(4; 3)$ car son coût est inférieur et sa consommation de ressource est supérieure (la somme des ressources doit être inférieure à la ressource maximale). Le label peut donc être retiré.

D'autres méthodes ont été utilisées dans la littérature pour résoudre les programmes dynamiques avec contraintes linéaires additionnelles. On peut notamment citer [BC89] qui proposent un algorithme de *branch-and-bound* basé sur la relaxation Lagrangienne.

2.5 Les agrégations en programmation dynamique

La transformation d'un programme dynamique sous forme de plus long chemin dans un graphe ne constitue pas toujours une méthode efficace car la taille de l'espace d'états peut-être exponentielle par rapport à la taille des états. Plusieurs méthodes ont été mises en place pour obtenir des méthodes utilisables en pratique. T.L.Morin [Mor78] classe ces méthodes en 9 catégories et donne de nombreuses références bibliographiques à ce sujet : l'élimination d'états en utilisant des bornes (par exemple méthodes de *branch-and-bound*), le label setting, les méthodes d'approximation, les techniques de décomposition, les méthodes duales (relaxation Lagrangienne), les méthodes de voisinages, l'exploitation

de la discontinuité et convexité, l'étude de la structure des données et représentation minimale d'un programme dynamique. Plusieurs d'entre elles sont utilisées dans ce manuscrit, notamment l'élimination d'états grâce à des bornes, la relaxation Lagrangienne, ...

Dans cette section, nous nous intéressons particulièrement à l'une d'entre elles : l'agrégation. Les méthodes d'agrégation permettent de réduire la taille du programme dynamique. Les agrégations construites correspondent à des relaxations du problème et donnent des bornes sur l'optimum. Dans cette section nous allons présenter l'agrégation de l'espace d'états qui permet de regrouper des états en un seul sur-état, puis nous nous intéresserons à l'agrégation de contraintes et plus particulièrement à son couplage avec la relaxation Lagrangienne.

2.5.1 Agrégation de l'espace d'états

Pour contourner le problème du nombre exponentiel d'états des programmes dynamiques, on peut utiliser les méthodes d'agrégation de l'espace d'état. Ce type d'agrégation a été proposé par [CMT81] dans le but d'obtenir des bornes du problème. L'idée est d'utiliser une fonction de projection de l'espace d'état original dans un espace d'états plus petit. Pour chaque état original, on associe un état dans le nouvel espace.

Soit L l'ensemble d'états original et $\bar{L}$ un ensemble d'états plus petit ($|L| > |\bar{L}|$). On note $\Psi_L^{-1}(\ell), \forall \ell \in L$ l'ensemble des états ℓ' de L tel qu'il existe une transition entre ℓ' et ℓ. On définit $g : L \to \bar{L}$ une fonction de projection qui affecte à chaque état de L un état de $\bar{L}$. La fonction g doit respecter la condition suivante :

Fonction de projection
si $\ell_{i-1} \in \Psi_L^{-1}(\ell_i)$ alors $g(\ell_{i-1}) \in \Psi_{\bar{L}}^{-1}(g(\ell_i))$

Cette condition impose que pour toute transition entre deux états de L, il existe une transition entre les états projetés de ces deux états.

Si on modélise un programme dynamique en utilisant un plus court chemin dans un graphe, chaque état est représenté par un sommet dans le graphe. Des auteurs ont proposés des agrégations basées sur le graphe, et qui sont équivalentes à l'agrégation de l'espace d'états. On peut citer notamment [Hin78] et [Whi79]. Le but est de réduire la taille du graphe correspondant au programme dynamique. Les nœuds du graphe vont être agrégés dans les nœuds d'un autre graphe plus petit.

L'article de [BBS87] présente un algorithme pour agréger les nœuds d'un graphe de programme dynamique. Le but est de résoudre un plus court chemin dans un graphe agrégé, appelé *macronetwork* et noté $G^M = (V^M, E^M)$, en partitionnant les nœuds du graphe original, noté $G = (V, E)$ ($|V| > |V^M|$). Le coût d'un arc du *macronetwork* (*macroarc*) est calculé récursivement de la façon suivante. Soit $\mathcal{V}(v^M), \forall v^M \in V^M$ l'ensemble des nœuds de G contenu dans le nœud v^M, avec $\forall v \in V, \exists$ un unique $v^M \in V^M$ tq $v \in \mathcal{V}(v^M)$.

Soit $\delta^-(v^M), \forall v^M \in V^M$ l'ensemble des nœuds d'entrée du nœud du macronetwork v^M, avec $\delta^-(v^M) = \{j \in \mathcal{V}(v^M) | \exists (i,j) \in E$ avec $i \in \mathcal{V}(w^M)$ et $w^M \in V^M$ tq $w^M \neq v^M\}$.

Les nœuds d'entrée correspondent aux nœuds du réseau initial qui ont un arc entrant venant d'un nœud agrégé dans un autre nœud du macronetwork.

Le nœud d'entrée fixe d'un nœud $v^M \in V^M$ est noté $j^*_{v^M}$ et $f(i,j), i,j \in V$ le plus court chemin dans le réseau original entre i et j.

Le coût d'un *macroarc* entre $v^M \in V^M$ et $w^M \in V^M$est noté $\mathcal{C}_{v^M,w^M}$et est égal à : $\mathcal{C}_{v^M,w^M} = \{\min_{j \in \mathcal{V}(w^M)} f(j^*_{v^M}, j)\}$ et permet d'obtenir $j^*_{w^M} = \{\arg\min_{j \in \mathcal{V}(w^M)} f(j^*_{v^M}, j)\}$.

Soit $F(v^M), \forall v^M \in V^M$ le coût du plus court chemin entre le sommet source et v^M.

La figure 2.3 montre un exemple de l'algorithme SADA proposé par [BBS87] en utilisant le partitionnement suivant : $\mathcal{V}(I) = \{0,1,2\}; \mathcal{V}(II) = \{3,4,7\}; \mathcal{V}(III) = \{5,6,8\}; \mathcal{V}(IV) = \{9\}$ avec $I, II, III, IV \in V^M$

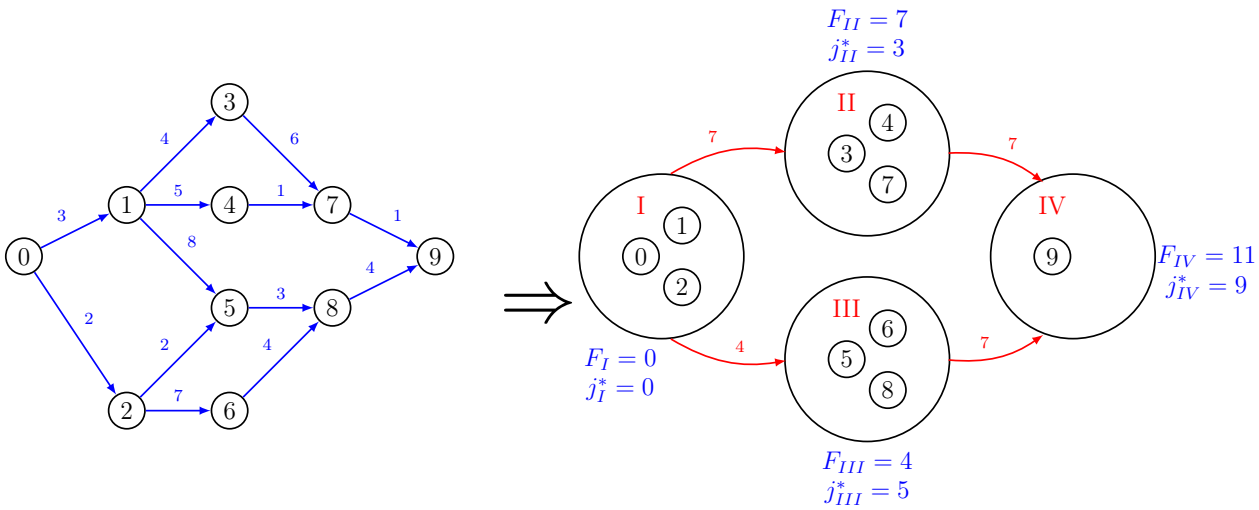


FIGURE 2.3 – Exemple SADA.

L'agrégation de l'espace d'états permet d'obtenir une borne valide du problème initial.

Pour diminuer la taille du programme dynamique, on peut décider de relâcher certaines contraintes additionnelles. L'espace d'états sera donc réduit et la résolution du problème relâché permettra d'obtenir une borne inférieure (en minimisation). Il existe plusieurs manières de relâcher des contraintes. La manière la plus simple est de ne pas les prendre en compte en les relâchant complètement.

2.5.2 Résolution de la relaxation Lagrangienne

Une autre manière classique est d'utiliser la relaxation Lagrangienne (2.2.2). Les contraintes relâchées ne seront plus vérifiées par le modèle, mais leurs violations seront pénalisées dans la fonction objectif. De nombreux articles traitent le sujet, on citera notamment [BC89] et [AMO88] qui appliquent la relaxation Lagrangienne au problème de plus court chemin avec contraintes additionnelles. Nous allons illustrer ce problème et montrer une manière de résoudre cette relaxation.

Dans les sections précédentes, nous avons vu que nos problèmes peuvent se modéliser sous forme d'un plus long chemin dans un graphe (à partir du programme dynamique (2.3)) avec contraintes de ressources. Ces contraintes peuvent être relâchées en utilisant la relaxation Lagrangienne, et l'approximation du problème dual Lagrangien nous permet d'obtenir un vecteur de multiplicateur optimale π (section 2.2.2). Nous montrons que la relaxation Lagrangienne associée à un vecteur π peut être résolue en calculant un plus long chemin dans un graphe.

Soit $G = (V, E)$ le graphe obtenu en modélisant le programme dynamique en problème de plus long chemin dans un graphe, et un ensemble de R contraintes linéaires additionnelles, avec $b_r, \forall r \in \{1, ..., R\}$ la partie constante de chaque contrainte. On note $\Delta(a, r), \forall a \in \mathcal{T}, r \in \{1, ..., R\}$ l'impact de la décision correspondant à la transition a sur la contrainte indicée par r. Le programme linéaire obtenu est :

$$\max \sum_{a \in \mathcal{T}} x_a * v(a) \tag{2.18}$$

$$\text{s.t.} \sum_{a \in \psi^{-1}(\ell)} x_a - \sum_{a \in \psi(\ell)} x_a = \begin{cases} -1 & \text{si } \ell = \ell^S \\ 0 & \text{si } \ell \in \mathcal{L} \setminus \{\ell^S, \ell^F\} \\ 1 & \text{si } \ell = \ell^F \end{cases}, \qquad \forall \ell \in \mathcal{L} \tag{2.19}$$

$$\sum_{a \in \mathcal{T}} \Delta(a, r) * x_a \leq b_r, \forall r \in \{1, ..., R\} \tag{2.20}$$

$$x_a \in \{0, 1\}\ , \forall a \in \mathcal{T} \tag{2.21}$$

On applique la relaxation Lagrangienne sur les contraintes (2.20) associé à un vecteur de multiplicateurs $\pi \in \mathbb{R}^R$:

$$\max \sum_{a \in \mathcal{T}} x_a * v(a) - \sum_{r=1}^{R} \pi_r (\sum_{a \in \mathcal{T}} \Delta(a, r) * x_a - b_r) \tag{2.22}$$

$$\text{s.t.} \sum_{a \in \psi^{-1}(\ell)} x_a - \sum_{a \in \psi(\ell)} x_a = \begin{cases} -1 & \text{si } \ell = \ell^S \\ 0 & \text{si } \ell \in \mathcal{L} \setminus \{\ell^S, \ell^F\} \\ 1 & \text{si } \ell = \ell^F \end{cases}, \qquad \forall \ell \in \mathcal{L} \tag{2.23}$$

$$x_a \in \{0, 1\}\ , \forall a \in \mathcal{T} \tag{2.24}$$

Comme chaque arc est associé à une transition, on peut construire un graphe $G_\pi = V, E$ en modifiant les coût des arcs. Soit $e \in E$ un arc du graphe G_π et $a(e)$ la transition associée à cet arc. Le coût de l'arc devient :

$$v(a) - \sum_{r=1}^{R} \pi_r (\Delta(a, r) - b_r)$$

2.6 Méthodes itératives

On a vu que les méthodes d'agrégation permettent de réduire l'espace d'états et d'obtenir des bornes duales du problème. Les méthodes itératives permettent de résoudre le problème en raffinant successivement les relaxations jusqu'à ce qu'un critère d'arrêt soit atteint. A chaque itération, la relaxation est enrichie en ajoutant à l'espace d'états des contraintes de ressources violées par la solution de la relaxation.

La première méthode utilisant les relaxations dans un algorithme itératif a été proposée par Ibaraki [Iba87], et s'appelle " *Successive Sublimation Dynamic Programming* " (SSDP). A chaque itération, l'algorithme crée explicitement le graphe du programme dynamique correspondant à l'espace d'états relâché courant, appelé graphe étendu. Le but est de construire successivement des relaxations du problème de plus en plus fines, et de supprimer des sommets et des arcs correspondant à des états et transitions non optimales pour le problème initial.

Un autre algorithme itératif est présenté dans deux articles de la littérature : [BDD06] et [RS08]. Cette algorithme, appelé par [RS08] *Decremental State Space Relaxation* (DSSR) se rapproche d'un algorithme de label setting. Dans cet algorithme, on construit un graphe support et on utilise un algorithme de labels, mais la construction et les dominances des labels se font uniquement selon les dimensions non relâchées.

2.6.1 SSDP

L'algorithme SSDP introduit par [Iba87] permet de résoudre les programmes dynamiques de grande taille en utilisant successivement des relaxations du problème. Le principe est de construire le graphe associé à une première relaxation, de supprimer des arcs non optimaux pour le problème initial, et d'enrichir à chaque itération la relaxation en ajoutant des contraintes qui étaient relâchées. A chaque itération, le graphe correspondant à la relaxation est construit explicitement et la résolution de la relaxation se fait par un algorithme de plus court chemin. La convergence est assurée si au moins une contrainte

relâchée est ajoutée à chaque itération.

Algorithme 4 : SSDP

1 **Calculer une borne primale**.
2 Calculer une borne primale du problème initial UB à partir d'une heuristique.
3 **Construire le graphe de la première relaxation**.
4 Construire le graphe G^0, correspondant à la première relaxation ;
5 $k \leftarrow 0$;
6 **Résoudre la relaxation et filtrer**.
7 Résoudre la relaxation correspondant au graphe G^k pour obtenir une solution `sol` ;
8 **si** `sol` *est réalisable et à un coût égal à UB* **alors**
9 TERMINER
10 Retirer des transitions non-optimales pour le problème initial, on obtient le graphe $\hat{G}^k$;
11 **Sublimation**.
12 Construire le nouveau graphe G^{k+1} à partir de $\hat{G}^k$ en réintroduisant de nouvelles contraintes ;
13 $k \leftarrow k + 1$;
14 Retourner en *ligne 6* ;

Les auteurs ont appliqué cet algorithme sur des problèmes classiques comme le problème de sac à dos, de voyageur de commerce ou de sac à dos contraint. Par la suite, ils ont traité un problème d'ordonnancement sur une machine [IN94].

C'est à partir de ces travaux que Tanaka a utilisé SSDP en obtenant des résultats compétitifs, d'abord sur un problème d'ordonnancement sur une machine sans temps mort [TFA09], puis sur un problème avec contraintes de précédence [TS13].

Malgré de bons résultats sur certains problèmes, SSDP a été très peu utilisé dans la littérature, car il peut être difficile à mettre en place en pratique. Pour obtenir des résultats compétitifs, Tanaka ajoute des éléments clés permettant de réduire la taille des graphes à chaque itération. On note principalement l'utilisation d'une borne duale calculée grâce à la relaxation Lagrangienne, différentes méthodes basées sur les dominances de séquences de jobs et une intégration des contraintes basée sur la taille du réseau.

2.6.2 DSSR

L'algorithme DSSR a été proposé par [BDD06] et [RS08]. Le but est d'utiliser la relaxation de l'espace d'états pour résoudre des relaxations successives. Contrairement à ce qui est fait dans SSDP, le graphe support reste le même durant tout l'algorithme. A chaque itération, un algorithme de label setting est utilisé pour calculer la borne courante. Pour obtenir un nombre acceptable de labels, on utilise une règle de dominance modifiée : seules les ressources correspondant aux contraintes non relâchées sont utilisées.

Soit $\mathcal{I}$ un ensemble de contraintes linéaires non relâchées. Les dominances sont véri-

fiées seulement pour les contraintes de $\mathcal{I}$.

Algorithme 5 : DDSR

```
1  Calculer une borne primale.
2  Initialisation :
3  Construire le graphe support
4  Ajouter le label initial à l'ensemble de labels du nœud source.
5  Extension :
6  pour chaque sommet dans un ordre de topologique faire
7      étendre les labels non dominés du sommet.
8      pour chaque label créé faire
9          Traitement du label :
10         Si le label n'est pas dominé, il faut l'ajouter à l'ensemble de labels.
11         Retirer tout les labels dominés par le nouveau label.
12 Résolution :
13 si il existe un label réalisable pour le problème initial dans l'ensemble de labels du
   sommet puits alors
14     TERMINER
15 sinon
16     ajouter des contraintes relachées à l'ensemble de contraintes I
17 Retourner en 3.
```

Contrairement à SSDP, la méthode DSSR a fréquemment été utilisée dans la littérature, notamment pour la résolution de sous-problèmes des méthodes de génération de colonnes. La méthode a été proposée au départ pour résoudre des problèmes de plus court chemin élémentaire avec contraintes de capacités ([RS08], [BDD06]). La plupart du temps, la méthode est utilisée pour résoudre les sous-problèmes obtenus par génération de colonnes (*branch-and-price*) sur des problèmes de tournées de véhicules. On peut notamment cité l'article de [BSU18] qui traite un problème de minimisation de latences, dérivé du problème de voyageur de commerce, l'article de [ESB10] qui traite de planification d'avions ou encore l'article de [KZBV16] qui traite un problème de tournées de véhicules avec profits. Les travaux de [CSVV18] montrent que l'on peut étendre cet algorithme aux hypergraphes sur un problème de découpe.

2.7 Conclusion

Cet état de l'art a permis de rappeler les bases de recherche opérationnelle sur lesquelles reposent le document. A partir de ces éléments, nous avons décrit les principales méthodes de résolution pour les programmes dynamiques avec contraintes additionnelles. Ces méthodes sont clairement proches, même si elles utilisent des outils différents pour converger. Dans le chapitre suivant, nous présenterons un formalisme unificateur qui nous permet de caractériser ces algorithmes de manière rigoureuse.

Chapitre 3

Un formalisme unificateur pour la résolution de programmes dynamiques avec contraintes linéaires additionnelles

3.1 Introduction

L'état de l'art a permis de présenter la problématique de programmation dynamique sous contraintes à laquelle nous nous intéressons. Des méthodes directes de résolution ont été proposées dans la littérature, en particulier des méthodes itératives d'agrégation permettant de contourner le problème du grand nombre de dimensions.

Dans ce chapitre, nous présentons un formalisme pour les programmes dynamiques avec contraintes linéaires additionnelles. Le but est d'unifier les méthodes de résolution directes et les méthodes itératives pour la résolution de ces problèmes. Plusieurs formalismes ont été proposés dans la littérature, mais aucun à notre connaissance ne permet d'exprimer l'ensemble des méthodes utilisées.

La première partie de ce chapitre présente les problèmes décrits sous forme de programme dynamique et de contraintes linéaires additionnelles. Les algorithmes basés sur la modélisation en problème de plus court chemin dans un graphe et les algorithmes de labels sont ensuite introduits. Comme nous nous intéressons à des programmes dynamiques de grande taille, la deuxième partie de ce chapitre introduit les méthodes d'agrégation qui permettent de formaliser des méthodes itératives basées sur l'agrégation de l'espace d'état. Nous montrons comment deux méthodes classiques (SSDP et DSSR) peuvent s'exprimer dans ce cadre général. Notons que les notations introduites dans ce formalisme sont différentes des notations utilisées dans le chapitre précédent.

3.2 Programme dynamique avec contraintes linéaires additionnelles

Dans ce manuscrit, nous nous intéressons à des problèmes pouvant se formuler sous la forme d'un programme dynamique et de contraintes linéaires additionnelles. On peut formuler ces contraintes linéaires comme des contraintes liées à des ressources qui sont consommées par les transitions. Cette première partie est consacrée au formalisme pour ces problèmes ainsi qu'aux méthodes classiques de résolution.

3.2.1 Programme dynamique et décisions séquentielles

Un programme dynamique modélise un processus de décision séquentiel. Chaque décision correspond à la transition d'un état dans un autre. Nous devons donc définir les états et les transitions d'un programme dynamique. Soit $P \in \mathbb{N}$ le nombre de dimensions du programme dynamique.

Définition 1 (Ensemble d'états)**.** *Un état d'un programme dynamique est un vecteur* $\boldsymbol{\ell} = (\ell_1, ..., \ell_P) \in \mathbb{R}^P$*. L'ensemble des états définis dans* $\mathbb{R}^P$ *est noté* $\mathcal{L}^P$ *.*

Une transition correspond à la transformation d'un état $\boldsymbol{\ell} \in \mathcal{L}^P$ dans un autre état $\boldsymbol{\ell}' \in \mathcal{L}^P$. On note $\mathcal{T}^P$ l'ensemble des transitions du programme dynamique et on définit donc le vecteur de $\mathbb{R}^P$ correspondant à la variation entre ces deux états.

Définition 2 (Vecteur de variation correspondant à une transition)**.** *Le vecteur de variation d'une transition* $\mathbf{t} \in \mathcal{T}^P$ *est donné par la fonction* $\Delta^P : \mathcal{T}^P \to \mathbb{R}^P$*. L'état d'arrivée de la transition* $\mathbf{t} \in \mathcal{T}^P$ *appliquée à l'état* $\boldsymbol{\ell} \in \mathcal{L}^P$ *est défini par :*

$$\begin{array}{rccc} F & : & \mathcal{L}^P \times \mathcal{T}^P & \to & \mathcal{L}^P \\ & & (\boldsymbol{\ell}, \mathbf{t}) & \mapsto & \boldsymbol{\ell}' = \boldsymbol{\ell} + \Delta^P(\mathbf{t}) \end{array}$$

avec $+$ *la somme des composantes des deux vecteurs.*

Nous nous intéressons à des processus séquentiels. Une manière de les modéliser est de définir une dimension particulière, appelée *dimension principale*. La particularité de cette dimension est que la variation correspondante contenue dans les transitions est strictement positive. Ces programmes dynamiques ont donc la propriété qu'un état ne peut être atteint qu'une fois dans une séquence de transitions. Par convention, on considérera la dimension d'indice 1 comme dimension principale.

Hypothèse 1. *Pour chaque transition* $\mathbf{t} \in \mathcal{T}^P$*,* $\Delta^P_1(\mathbf{t}) > 0$*.*

Les décisions applicables à un état donné diffèrent selon les états. On définit l'ensemble des transitions applicables depuis chaque état. Ces ensembles permettent de définir les séquences de transitions réalisables.

Définition 3 (Ensemble de transitions applicables)**.** *L'ensemble des transitions applicables depuis un état $\boldsymbol{\ell} \in \mathcal{L}^P$ est donné par la fonction $\psi^P(\boldsymbol{\ell})$:*

$$\begin{array}{rccc} \psi^P & : & \mathcal{L}^P & \rightarrow & \mathbb{P}(\mathcal{T}^P) \\ & & \boldsymbol{\ell} & \mapsto & \{\mathbf{t}_1, ..., \mathbf{t}_{z(\boldsymbol{\ell})}\} \end{array}$$

avec $z(\boldsymbol{\ell})$ le nombre de transitions appliquables depuis $\boldsymbol{\ell}$.

La solution d'un programme dynamique est une séquence de paires état/transition $\mu = ((\boldsymbol{\ell}^0, \mathbf{t}_1), ..., (\boldsymbol{\ell}^{z(\mu)-1}, \mathbf{t}_{z(\mu)}))$, avec $z(\mu)$ le nombre de transition dans la séquence. On définit ensuite une solution réalisable. Soit $\boldsymbol{\ell}^S$ l'état source du programme dynamique et $\boldsymbol{\ell}^F$ l'état terminal.

Définition 4 (Solution réalisable de DP)**.** *Une solution $\mu = ((\boldsymbol{\ell}^0, \mathbf{t}_1), ..., (\boldsymbol{\ell}^{z(\mu)-1}, \mathbf{t}_{z(\mu)}))$ est réalisable si et seulement si elle respecte les 3 propriétés suivantes :*

1. $\boldsymbol{\ell}^0 = \boldsymbol{\ell}^S$ *et* $F(\boldsymbol{\ell}^{z(\mu-1)}, \mathbf{t}_{z(\mu)}) = \boldsymbol{\ell}^F$
2. $\mathbf{t}_i \in \psi^P(\boldsymbol{\ell}^{i-1}), i = 1, ..., z(\mu)$
3. $\boldsymbol{\ell}^i = F(\boldsymbol{\ell}^{i-1}, \mathbf{t}_i), i = 1, ..., z(\mu)$

La propriété 1 assure que la première transition est appliquée à l'état source et la dernière permet de produire un des états terminaux du programme dynamique. Pour chaque transition de la séquence, elle doit appartenir à l'ensemble des transitions possibles depuis l'état de départ de la transition (propriété 2), et que l'état suivant est bien obtenu à partir de ce couple état-transition (propriété 3).

Pour calculer le coût d'une solution, on définit la fonction de coût d'une transition appliquée à un état.

Définition 5 (Fonction de coût)**.** *La fonction de coût $f^P : \mathcal{L}^P \times \mathcal{T}^P \rightarrow \mathbb{R}$ correspond au coût d'utiliser la transition $\mathbf{t}$ à partir de l'état $\boldsymbol{\ell}$.*

Pour résoudre un programme dynamique, on cherche une solution réalisable qui maximise $\sum_{i=1}^{z(\mu)} f^P(\boldsymbol{\ell}_{i-1}, \mathbf{t}_i)$. On peut ainsi définir la fonction de récurrence qui définit ce programme dynamique :

DP

$$\max \alpha(\boldsymbol{\ell}^S)$$

$$\forall \boldsymbol{\ell} \in \mathcal{L}^P, \alpha(\boldsymbol{\ell}) = \begin{cases} \max\limits_{\mathbf{t} \in \psi^P(\boldsymbol{\ell})} \{f^P(\boldsymbol{\ell}, \mathbf{t}) + \alpha(F(\boldsymbol{\ell}, \mathbf{t}))\} & \text{si } \boldsymbol{\ell} \neq \boldsymbol{\ell}^F \\ 0 & \text{sinon} \end{cases}$$

Note : on adopte la convention $\max \emptyset = -\infty$.

Dans la suite de ce manuscrit, ce programme dynamique sera noté **DP**.

3.2.2 Modélisation d'un programme dynamique en plus court chemin dans un graphe

Les programmes dynamiques décrits précédemment peuvent être modélisés sous forme de plus court chemin dans un graphe. Pour construire ce graphe, nous introduisons une fonction de mapping entre les états du programme dynamique et les sommets du graphe.

Définition 6 (Fonction de mapping)**.** *Soit $G = (V, E)$ un multigraphe avec V l'ensemble des sommets et E l'ensemble des arcs, $|V| = |\mathcal{L}|$ et $|E| = |\mathcal{T}|$. On définit la fonction bijective :*

$$\begin{array}{rccc} g & : & V & \to & \mathcal{L} \\ & & v & \mapsto & \boldsymbol{\ell} \end{array}$$

qui associe à chaque sommet du graphe un état du programme dynamique.

Il existe un arc $e \in E$ correspondant à la transition $\mathbf{t} \in \mathcal{T}$ entre v_1 et v_2 si et seulement si :

1. $\mathbf{t} \in \psi^P(g(v_1))$
2. $g(v_2) = F(g(v_1), \mathbf{t})$

Nous utilisons pour un arc $e \in E$ la notation $e = (v_1, v_2, \mathbf{t}, c)$ avec :

- $v_1 \in V$ la queue de l'arc
- $v_2 \in V$ la tête de l'arc
- $\mathbf{t} \in \mathcal{T}$ la transition associée à l'arc
- $c \in \mathbb{R}$ le coût de l'arc ($c = f^P(g(v_1), \mathbf{t})$)

L'algorithme de construction de graphe à partir d'un programme dynamique est le

suivant :

Algorithme 6 : Création du graphe associé à un programme dynamique

Input : DP
Output : $G(V, E)$
Construire $\mathbb{T}$ une liste de sommets vide
Créer le sommet v^S tel que $g(v^0) = \boldsymbol{\ell}^S$
Créer le sommet v^F tel que $g(v^f) = \boldsymbol{\ell}^F$
Ajouter v^S à $\mathbb{T}$
V $\leftarrow \{v^S, v^F\}$
while $\mathbb{T}$ *n'est pas vide* **do**
    Soit $v \in \mathbb{T}$ tel que $g(v)_1 = \min\{g(u)_1 : u \in \mathbb{T}\}$
    $\mathbb{T} \leftarrow \mathbb{T} - \{v\}$
    for $\mathbf{t} \in \psi^P(g(v))$ **do**
        Créer le sommet v' tel que $g(v') = F(g(v), \mathbf{t})$
        Si $v' \notin V$, alors $V \leftarrow V \cup \{v'\}$
        Créer l'arc $e = (v, v', \mathbf{t}, f^P(g(v), t))$
        E $\leftarrow$ E $\cup \{e\}$
        if $v' \notin \mathbb{T}$ **then**
            Ajouter v' à $\mathbb{T}$

Remarque 2. *Comme la dimension principale est strictement croissante (cf hypothèse 1), le graphe G construit est acyclique.*

Résoudre le problème de plus long chemin entre la source et le sommet puits fictif dans ce graphe est équivalent à résoudre le programme dynamique.

3.2.3 Programme dynamique avec contraintes linéaires additionnelles

Les problèmes que nous traitons se présentent sous la forme d'un programme dynamique et de contraintes linéaires additionnelles. On notera DP+LSC ce type de problème dans ce manuscrit. Ces contraintes peuvent être vues comme des contraintes de ressources : chaque transition consomme des ressources et la somme des consommations de chaque ressource dans une solution doit être inférieure ou égale à une valeur de consommation maximale. Notons $R \in \mathbb{N}$ le nombre de contraintes additionnelles, nous devons exprimer ces contraintes dans notre formalisme. Pour chaque contrainte $r \in \{1, ..., R\}$, on définit une valeur totale maximale b_r, et on note $\mathbf{b}$ le vecteur de ces valeurs maximales. En plus des variations d'états du programme dynamique, les transitions contiennent les consommations de ressources des contraintes linéaires. On définit donc la fonction $a : \mathcal{T} \rightarrow \mathbb{R}^R$ qui renvoie un vecteur des consommation associés à une transition dans les contraintes linéaires. Enfin, on définit des valeurs o_r^- et o_r^+ correspondant aux valeurs minimales et maximales des ressources, ainsi que $\mathbf{o}^-$ et $\mathbf{o}^+$ les vecteurs de ces valeurs minimales et maximales. On peut, sans perdre de généralité, toujours fixer ses valeurs en prenant pour

chaque ressource les valeurs $-\infty$ et $+\infty$.

Pour définir une solution réalisable de ce type de problème, on reprend la définition d'une solution réalisable d'un problème DP auquel on ajoute les contraintes linéaires. Une solution réalisable est définie ainsi :

Définition 7 (Solution réalisable de DP+LSC)**.** *Une solution* $\mu = ((\boldsymbol{\ell}^0, \mathbf{t}_1), ..., (\boldsymbol{\ell}^{z(\mu)-1}, \mathbf{t}_{z(\mu)}))$ *est réalisable si et seulement si elle respecte les 5 propriétés suivantes :*

1. $\boldsymbol{\ell}^0 = \boldsymbol{\ell}^S$ *et* $F(\boldsymbol{\ell}^{z(\mu-1)}, \mathbf{t}_{z(\mu)}) = \boldsymbol{\ell}^F$
2. $\mathbf{t}_i \in \psi^P(\boldsymbol{\ell}^{i-1}), i = 1, ..., z(\mu)$
3. $\boldsymbol{\ell}^i = F(\boldsymbol{\ell}^{i-1}, \mathbf{t}_i), i = 1, ..., z(\mu)$
4. $\sum_{i=1}^{z(\mu)} a(\mathbf{t}_i) \leq \mathbf{b}$
5. $\mathbf{o}^- \leq \sum_{i=1}^{j} a(\mathbf{t}_i) \leq \mathbf{o}^+, \forall j = 1, ..., z(\mu)$

3.2.4 Algorithme pour résoudre DP + LSC

Dans la section précédente, nous avons vu qu'un problème DP peut être modélisé comme un problème de plus court chemin dans un graphe. Il apparait naturellement qu'un problème DP+LSC peut être formulé comme un problème de plus court chemin avec contraintes de ressources. Les travaux de [ID05] ont listé plusieurs méthodes de résolutions. Dans cette section, nous allons voir différents algorithmes de résolution de ces problèmes.

Dans [DS88], les auteurs ont introduit l'algorithme de "label setting", repris plus tard par [RS08]. Cet algorithme est utilisé pour résoudre les problèmes de plus court chemin avec contraintes de ressources.

Dans cet algorithme, on utilise le *graphe support* qui correspond au graphe de DP. A chaque sommet sont associés des labels et un coût pour chaque label. Chaque label représente un chemin entre le sommet source et le sommet associé au label. Pour chaque sommet, on étend les labels associés pour créer de nouveaux labels. La solution optimale est donnée par le label de meilleur coût sur le sommet terminal fictif.

Soit $G^{DP} = (V^{DP}, E^{DP})$ le graphe obtenu en modélisant le problème DP sous forme de plus court chemin dans un graphe. Un label est un vecteur $\iota \in \mathbb{R}^R$. A chaque sommet $v \in V^{DP}$, on associe un ensemble de labels $\beta(v)$, avec un coût pour chaque label. Soit $\tilde{c}(\iota)$ le coût d'un label $\iota \in \beta(v)$.

On définit ensuite l'opérateur de dominance $\succeq$ permettant de supprimer des labels ne pouvant correspondre à une solution optimale.

Définition 8 (Dominance entre deux labels)**.** *Soit $v \in V^{DP}$ et $\iota^1, \iota^2 \in \beta(v)$. Le label ι^1 domine le label ι^2 ($\iota^1 \succeq \iota^2$) si et seulement si :*

$$1)\tilde{c}(\iota^1) \geq \tilde{c}(\iota^2)$$

$$2)\iota_i^1 \leq \iota_i^2, \forall i \in \{1, ..., R\}$$

et au moins une des relations est stricte.

Algorithme 7 : Label Setting

```
1 Ajouter ι^0 le label initial à l'ensemble de labels β(v^S).
2 pour chaque sommet v ∈ V^DP dans un ordre topologique faire
3     pour chaque label ι ∈ β(v) faire
4         pour chaque arc e = (v, v', t, c) ∈ δ^+(v) faire
5             créer le label ι'. ∀i ∈ {1, ..., R}, ι'_i = ι_i + a(t_i).
6             c̃(ι') = c̃(ι) + c
7             si Pour tout k ∈ β(v) , k ⊁ ι' alors
8                 Ajouter ι' à β(v')
9                 Retirer les labels j de β(v') tq ι' ⪰ j
```

Un label ι est réalisable si et seulement si :

$$\forall r \in R, \iota_r \leq b_r$$

La valeur optimale correspond au label **réalisable** de meilleur coût dans l'ensemble de labels de v^F (sommet terminal).

3.3 Intégrer les contraintes dans l'espace d'états

Pour traiter les problèmes DP+LSC, nous pouvons intégrer les contraintes linéaires dans l'espace d'états. Pour cela, nous allons augmenter le nombre de dimensions des états du programme dynamique et prendre en compte les contraintes linéaires additionnelles dans la récurrence.

Grâce à la structure particulière des contraintes additionnelles, on peut, pour toute instance DP+LSC, construire un programme dynamique intégrant ces contraintes dans la relation de récurrence. Le programme dynamique intégrant ces contraintes sera noté *EDP*.

Le nombre de dimensions des nouveaux états sera noté $m = P + R$. On définit $\mathcal{L}$ l'ensemble des états admissibles de EDP définis dans $\mathbb{R}^m$.

$$\mathcal{L} = \{\boldsymbol{\ell} \in \mathbb{R}^m : \boldsymbol{\ell}_{|_P} \in \mathcal{L}^P, \boldsymbol{\ell}_i \in [o_i^-, o_i^+] \forall i \in \{P+1, ..., m\}\}$$

Pour obtenir l'état initial du programme dynamique EDP $\tilde{\boldsymbol{\ell}}^S \in \mathcal{L}$, il suffit de concaténer $\boldsymbol{\ell}^S$ et $\mathbf{b^0}$, avec $\mathbf{b^0}$ un vecteur de 0 de taille R. Pour obtenir un état terminal unique,

on ne peut pas se contenter des transitions de $\mathcal{T}^P$, car $\mathcal{L}$ peut contenir plusieurs états $\boldsymbol{\ell} \in \mathcal{L}$ tel que $\boldsymbol{\ell}_{|_P} = \boldsymbol{\ell}^F$, avec $|_P$ un opérateur qui tronque un vecteur aux P premières composantes.

On définit donc $\tilde{\boldsymbol{\ell}}^F \in \mathcal{L}$ l'état terminal de EDP en concaténant $\boldsymbol{\ell}^F$ et $\mathbf{b}$, et en fixant $\tilde{\boldsymbol{\ell}}_1^F = \boldsymbol{\ell}_1^F + 1$ (pour respecter les propriétés de la dimension principale) et on introduit un ensemble de transitions additionnelles $\mathcal{T}^*$ avec :

$$\mathcal{T}^* = \{\mathbf{t}^{\boldsymbol{\ell},F} : \boldsymbol{\ell} \in \mathcal{L}, \boldsymbol{\ell}_{|_P} = \boldsymbol{\ell}^F\}$$

($\forall \boldsymbol{\ell} \in \mathcal{L}$, il existe une transition $\mathbf{t}^{\boldsymbol{\ell},F} \in \mathcal{T}^*$ entre $\boldsymbol{\ell}$ et $\tilde{\boldsymbol{\ell}}^F$ si et seulement si $\boldsymbol{\ell}_{|_P} = \boldsymbol{\ell}^F$)

Le vecteur de variation correspondant aux transitions de $\mathcal{T}^*$ est noté $\Delta^* : \mathcal{T}^* \to \mathbb{R}^m$.

L'ensemble des transitions $\mathcal{T}$ de EDP est défini par $\mathcal{T} = \mathcal{T}^P \cup \mathcal{T}^*$. On définit ensuite la fonction qui permet de calculer à partir d'une transition du programme dynamique EDP le vecteur de variation de la transition.

Définition 9 (Vecteur de variation des transitions de EDP)**.** *La fonction $\Delta : \mathcal{T} \to \mathbb{R}^m$ permettant d'obtenir le vecteur de variation d'une transition de EDP est définie par :*

$$\forall \mathbf{t} \in \mathcal{T}, \Delta(\mathbf{t}) = \begin{cases} \Delta_i^P(\mathbf{t}) & \textit{si } i \leq P \textit{ et } \mathbf{t} \in \mathcal{T}^P \\ a_{i-P}(\mathbf{t}) & \textit{si } i > P \textit{ et } \mathbf{t} \in \mathcal{T}^P \\ \Delta^*(\mathbf{t}) & \textit{si } \mathbf{t} \in \mathcal{T}^* \end{cases}$$

On définit la fonction donnant l'ensemble des transitions applicables depuis un état.

Définition 10 (Ensemble des transitions applicables depuis un état de EDP)**.** *L'ensemble des transitions applicables $\psi : \mathcal{L} \to \mathbb{P}(\mathcal{T})$ est défini par :*

$$\forall \boldsymbol{\ell} \in \mathcal{L}, \psi(\boldsymbol{\ell}) = \begin{cases} \psi^P(\boldsymbol{\ell}_{|_P}) & \textit{si } \boldsymbol{\ell}_{|_P} \neq \boldsymbol{\ell}^F \\ \mathbf{t}^{\boldsymbol{\ell},F} & \textit{si } \boldsymbol{\ell}_{|_P} = \boldsymbol{\ell}^F \textit{ et } \forall i > P, \boldsymbol{\ell}_i \leq \mathbf{b}_{i-P} \\ \emptyset & \textit{sinon} \end{cases}$$

Il reste à définir la fonction de coût $f : \mathcal{L} \times \mathcal{T} \to \mathbb{R}$ correspondant au coût d'utiliser la transition $\mathbf{t}$ à partir de l'état $\boldsymbol{\ell}$.

$$\forall \boldsymbol{\ell} \in \mathcal{L}, \forall \mathbf{t} \in \mathcal{T}, f(\boldsymbol{\ell}, \mathbf{t}) = \begin{cases} f^P(\boldsymbol{\ell}_{|_P}, \mathbf{t}) & \text{si } \mathbf{t} \in \mathcal{T}^P \\ 0 & \textit{sinon} \end{cases}$$

Le programme dynamique étendu EDP est défini par la fonction de récurrence suivante :

EDP

$$\max \alpha(\tilde{\boldsymbol{\ell}}^S)$$

$$\forall \boldsymbol{\ell} \in \mathcal{L}, \alpha(\boldsymbol{\ell}) = \begin{cases} \max\limits_{\mathbf{t} \in \psi(\boldsymbol{\ell})} \{f(\boldsymbol{\ell}, \mathbf{t}) + \alpha(\boldsymbol{\ell} + \Delta(\mathbf{t}))\} & \text{si } \boldsymbol{\ell} \neq \tilde{\boldsymbol{\ell}}^F \\ 0 & \text{sinon} \end{cases}$$

suivant la convention $\max \emptyset = -\infty$.

En intégrant les contraintes additionnelles dans l'espace d'états, on a obtenu un autre programme dynamique qui peut être modélisé comme un problème de plus court chemin dans un graphe, comme vu dans la section 3.1.2. On peut aussi obtenir un problème équivalent en n'intégrant qu'une partie des contraintes linéaires dans l'espace d'états. On définit alors un nouveau problème de type DP+LSC pouvant être résolu en utilisant les algorithmes précédents.

Remarque 3. *Il existe un nombre exponentiel de problèmes équivalents correspondant à l'intégration dans l'espace d'états d'une sous partie des contraintes linéaires additionnelles.*

Remarque 4. *Résoudre le problème EDP où l'intégralité des contraintes linéaires ont été intégrées dans l'espace d'états par l'algorithme de label setting est équivalent à résoudre le problème de plus court chemin dans le graphe correspondant à EDP.*

3.4 Agrégation

Nous avons vu que le programme dynamique EDP défini précédemment était de taille exponentielle en le nombre de contraintes intégrées dans l'espace d'états. Les méthodes d'agrégation permettent de travailler sur des programmes dynamiques de tailles plus petites et permettent d'obtenir une borne pour le programme dynamique EDP. Dans cette partie, nous allons voir comment définir une agrégation et comment on peut détecter des sommets et des arcs du graphe agrégé qui ne sont l'agrégation d'aucun sommet/arc du graphe initial appartenant à une solution optimale.

3.4.1 Mapping des états

Pour obtenir une relaxation du problème, il est possible d'agréger l'espace d'états du programme dynamique. Pour ce faire, nous allons relâcher une partie des contraintes linéaires additionnelles. L'ensemble des dimensions de EDP est noté $\mathcal{I} = \{1, ..., P, P + 1, ..., m\}$. On note les dimensions du programme dynamique d'origine $\mathcal{I}^{DP} = \{1, ..., P\}$, et les dimensions correspondant aux contraintes linéaires additionnelles $\mathcal{I}^{LC} = \{P + 1, ..., m\}$. Définissons un programme dynamique relâché.

Définition 11 (Programme dynamique relaché)**.** *Soit $\mathcal{K} \subset \mathcal{I}^{LC}$ l'ensemble des dimensions correspondant à une contrainte relâchée, et $\mathcal{J} = \mathcal{I}^{LC} \setminus \mathcal{K}$ les dimensions correspondant à des contraintes non relachées. La relaxation obtenue est définie par le programme dynamique $EDP^{\mathcal{J}}$ de structure similaire à EDP, dans lequel les contraintes portant sur les composantes de $\mathcal{K}$ des états sont relâchées.*

Nous introduisons à présent une définition formelle de $EDP^{\mathcal{J}}$. Deux états sont agrégés si leurs composantes dans $\mathcal{J}$ sont identiques. En pratique, cela signifie que l'on va considérer les états ayant les mêmes composantes dans $\mathcal{J}$ comme un seul état. Cette agrégation est représentée par les figures 3.1 et 3.2.

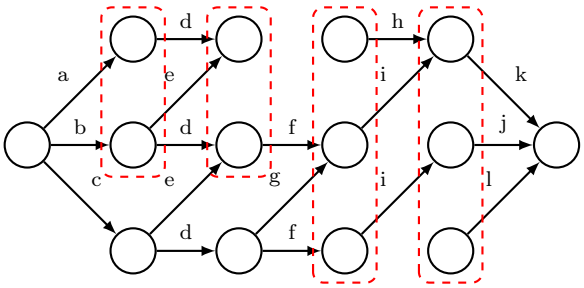


FIGURE 3.1 – Représentation de EDP. Les zones pointillées correspondent à des états qui ont les mêmes composantes dans $\mathcal{J}$.

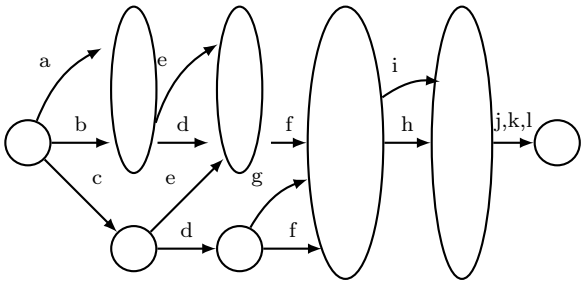


FIGURE 3.2 – Représentation de $EDP^{\mathcal{J}}$

Remarque 5. *Les transitions "doublons" entre deux états agrégés ne sont représentées qu'une seule fois dans le graphe agrégé (par exemple, la transition d entre les deux sommets agrégés de la figure 3.2).*

Nous allons désormais formaliser le programme dynamique relâché. Nous introduisons la notion d'empreinte qui va permettre d'exprimer l'agrégation d'états d'un programme dynamique EDP.

Définition 12 (Empreinte d'un état)**.** *Etant donné un état $\boldsymbol{\ell} \in \mathcal{L}$ et un ensemble $J \subseteq \{1, ..., m\}$, la fonction qui associe à un état $\boldsymbol{\ell}$ son empreinte sur J est définie par :*

$$\begin{array}{rccl} id_J & : \mathcal{L} & \to & (\mathbb{R} \cup \{*\})^m \\ & \boldsymbol{\ell} & \mapsto & \boldsymbol{e}_i = \begin{cases} \boldsymbol{\ell}_i & si\ i \in J \\ * & sinon \end{cases} \end{array}$$

Notons que deux états du programme dynamique original ayant la même empreinte seront considérés identiques dans le programme dynamique relâché. Chaque état du programme dynamique original possède une empreinte sur n'importe quel ensemble J.

Définition 13 (Ensemble des empreintes possible sur un ensemble J)**.** *L'ensemble des empreintes possibles pour l'ensemble des composantes J est noté ID_J, et est défini par :*

$$ID_J = \{\boldsymbol{e} \in (\mathbb{R} \cup \{*\})^m : \boldsymbol{e}_i \in \mathbb{R}\ \forall i \in J, \boldsymbol{e}_i = *\ \forall i \notin J\}$$

Un état du programme dynamique relâché est représenté par une empreinte. Dans ce programme dynamique, une empreinte $\boldsymbol{e}$ représente un ensemble $S_J(\boldsymbol{e})$ d'états du programme dynamique EDP de base : $S_J(\boldsymbol{e}) = \{\boldsymbol{\ell} \in \mathcal{L} : id_J(\boldsymbol{\ell}) = \boldsymbol{e}\}$. Comme le coût d'une transition du programme dynamique de base est calculé seulement à partir d'une transition et des P premières composantes de l'état de départ de la transition, le coût d'une transition dans le programme dynamique relâché peut être obtenu à partir des empreintes.

Le programme dynamique relâché est donné par la fonction de récurrence suivante :

$$\underline{EDP^{\mathcal{J}}}$$

$$\forall \boldsymbol{e} \in ID_{\mathcal{J}}, \hat{\alpha}(\boldsymbol{e}) = \begin{cases} \max\limits_{\mathbf{t} \in \cup_{\boldsymbol{\ell} \in S_{\mathcal{J}}(\boldsymbol{e})} \psi(\boldsymbol{\ell})} \{f^P(\boldsymbol{e}_{|_P}, \mathbf{t}) + \hat{\alpha}(\boldsymbol{e} + \Delta(\mathbf{t}))\} & \text{si } \boldsymbol{e} \neq id_{\mathcal{J}}(\tilde{\boldsymbol{\ell}}^F) \\ 0 & \text{sinon} \end{cases}$$

$$\text{avec } \boldsymbol{e} + \Delta(\mathbf{t}) = \left\{ \boldsymbol{e}' \in (\mathbb{R} \cup \{*\})^m : \forall i \in \{1, ..., m\}, \boldsymbol{e}'_i = \begin{cases} \boldsymbol{e}_i + \Delta_i(\mathbf{t}) & \text{si } i \in \mathcal{J} \cup \mathcal{I}^{DP} \\ * & \text{si } i \in \mathcal{K} \end{cases} \right\}$$

3.4.2 Fonction d'agrégation

Nous avons vu que la fonction de mapping permet de représenter un ensemble d'états en utilisant un seul état, caractérisé par une empreinte. Une difficulté est de déterminer les transitions issues de ces empreintes. Cette difficulté découle du fait que l'ensemble des états représentés contient des états réalisables et non-réalisables pour EDP. Dans le programme dynamique précédent, les transitions sortantes d'un état du programme dynamique relâché sont les transitions sortantes de tous les états ayant une empreinte égale à celle de l'état relâché, qu'ils soient réalisables ou non.

Par ailleurs, on peut parfois détecter directement que certaines transitions ne peuvent pas mener à des solutions réalisables si elles sont appliquées à certains états.

Pour traiter ces ensembles d'états de manière implicite, on a besoin en pratique d'avoir une description compacte de ces ensembles. On cherche un compromis entre la taille de la description et sa précision. Nous devons d'abord définir la fonction d'agrégation qui permet d'identifier les états qui vont être considérés.

Définition 14 (Fonction d'agrégation)**.** *On définit la fonction :*

$$agg : \mathbb{P}(\mathcal{L}) \to \mathbb{P}(\mathcal{L})$$

qui à un groupe d'états du programme dynamique EDP, en associe un sur-ensemble.

La fonction d'agrégation permet de définir le sur-ensemble des états réalisables qui sera considéré pour identifier les transitions sortantes d'un état caractérisé par une empreinte.

Différentes possibilités s'offrent pour le choix de cette fonction. Par exemple, la fonction identité $agg_{Id} : W \to W$ donne une description parfaite mais ne permet pas de gain d'occupation mémoire, car on doit conserver tous les états réalisables. Au contraire, la fonction qui renvoie l'ensemble des états ayant la même empreinte que celle des états réalisables permet un gain d'occupation mémoire (seule l'empreinte est conservée) mais ne permet pas d'avoir une description précise. Nous allons définir la récurrence d'un programme dynamique relâché à partir d'une fonction d'agrégation. Le programme dynamique obtenu sera noté $EDP^{\mathcal{J}}_{agg}$. Les états de ce programme dynamique sont caractérisés par une empreinte.

L'ensemble des états du programme dynamique $EDP^{\mathcal{J}}_{agg}$ correspond à l'ensemble des empreintes possibles $ID_{\mathcal{J}}$.

On génère des ensembles d'états de manière implicite. L'ensemble des états atteignables du programme dynamique original correspondant à une même empreinte est calculé récursivement.

On définit la fonction définissant les états atteignables de EDP à partir d'un ensemble d'états de EDP.

Définition 15 (Ensemble d'états atteignables à partir d'un état $\boldsymbol{\ell} \in \mathcal{L}$)**.** *La fonction $N : \mathcal{L} \to \mathbb{P}(\mathcal{L})$ qui pour un état donné $\boldsymbol{\ell} \in \mathcal{L}$ renvoie les états atteignables à partir de $\boldsymbol{\ell}$ via une transition, est défini par*

$$N(\boldsymbol{\ell}) = \{\boldsymbol{\ell} + \mathbf{t} | \mathbf{t} \in \psi(\boldsymbol{\ell})\},\ et\ \mathcal{N}(L) = \bigcup_{\boldsymbol{\ell} \in L} N(\boldsymbol{\ell}),\ avec\ L \subseteq \mathcal{L}$$

Cet ensemble peut être vu comme l'ensemble des voisins atteignables de $\boldsymbol{\ell}$.

On définit ensuite l'ensemble des états atteignables de EDP correspondant à un état de $EDP^{\mathcal{J}}_{agg}$.

Définition 16 (Ensemble des états atteignables de EDP à partir d'un état de $EDP^{\mathcal{J}}_{agg}$)**.**

$$\mathcal{A}(\boldsymbol{e}) = \begin{cases} \{\boldsymbol{\ell} \in \mathcal{L} : \boldsymbol{\ell}_{|_P} = \boldsymbol{\ell}^S\} & si\ \boldsymbol{e} = id_{\mathcal{J}}(\tilde{\boldsymbol{\ell}}^S) \\ \{\boldsymbol{\ell} \in \bigcup\limits_{\boldsymbol{e}' \in EDP^{\mathcal{J}}_{agg} : \boldsymbol{e}'_1 < \boldsymbol{e}_1} \mathcal{N}(agg(\mathcal{A}(\boldsymbol{e}'))) \mid id_{\mathcal{J}}(\boldsymbol{\ell}) = \boldsymbol{e}\} & sinon \end{cases}$$

Nous pouvons désormais exprimer la relation de récurrence qui décrit le programme dynamique $EDP^{\mathcal{J}}_{agg}$:

$$\begin{array}{l} \underline{EDP^{\mathcal{J}}_{agg}} \\ \max \alpha^{\mathcal{J}}(id_{\mathcal{J}}(\tilde{\boldsymbol{\ell}}^{\mathbf{S}})) \\ \forall \boldsymbol{e} \in ID_{\mathcal{J}}, \alpha^{\mathcal{J}}(\boldsymbol{e}) = \begin{cases} \max\limits_{\mathbf{t} \in \cup_{\boldsymbol{\ell} \in agg(\mathcal{A}(\boldsymbol{e}))} \psi(\boldsymbol{\ell})} \{f(\boldsymbol{e}_{|_P}, \mathbf{t}) + \alpha^{\mathcal{J}}(id_{\mathcal{J}}(\boldsymbol{\ell} + \mathbf{t}))\} & \text{si } \boldsymbol{e} \neq id_{\mathcal{J}}(\tilde{\boldsymbol{\ell}}^F) \\ 0 & \text{sinon} \end{cases} \end{array}$$

Il reste à définir correctement la fonction d'agrégation agg. Nous devons montrer que cette fonction permet d'obtenir une relaxation valide du problème.

Proposition 1. *Une condition suffisante pour que EDP^{J}_{agg} soit une relaxation de EDP est que $agg(E) \supseteq E, \forall E \in \mathbb{P}(\mathcal{L})$*

Le résultat découle du fait que $\tilde{\boldsymbol{\ell}}^S$ est réalisable, et par récurrence, toutes les transitions réalisables initiales sont conservées en utilisant agg. Les coûts des transitions ne sont pas changés donc la relaxation est valide.

Pour illustrer ces fonctions d'agrégation, nous présentons les deux fonctions d'agrégation basiques appliquées à un problème de sac à dos binaire disjonctif.

Exemple 2 (Sac à dos binaire disjonctif)**.** *On considère un problème de sac à dos binaire 1 auquel on ajoute des contraintes linéaires indiquant si deux objets ne peuvent être pris ensemble dans une solution réalisable. Définissons une instance de sac à dos binaire où*

la taille maximale du sac à dos est $W = 4$*, et le nombre d'objets est* $n = 3$*. Chaque objet* $i \in \{1, ..., n\}$ *est défini par* (w_i, p_i)*, avec* w_i *le poids de l'objet* i *et* p_i *le profit de l'objet* i*. L'ensemble des objets de l'instance est le suivant :* $\{(2, 5); (2, 3); (2, 2)\}$*. On ajoute deux contraintes linéaires de disjonction : l'objet* 1 *et l'objet* 2 *ne peuvent être pris ensemble, ainsi que l'objet* 1 *et l'objet* 3*. Soit* x_i *la variable binaire égale à 1 si l'objet* i *est sélectionné dans le sac à dos. Les deux contraintes linéaires correspondantes sont :*

$$x_1 + x_2 \leq 1$$

$$x_1 + x_3 \leq 1$$

Un état du programme dynamique est un couple (i, w)*, où la dimension* i *correspond au dernier objet pouvant être pris, et la dimension* w *correspond à la place disponible dans le sac à dos. On peut intégrer les deux contraintes linéaires en ajoutant deux dimensions au programme dynamique. Un état du programme dynamique étendu est un tuple* (i, w, d_1, d_2) *où* d_1 *correspond au nombre d'objets dans le sac à dos de la première contrainte de disjonction (1 ou 2), et* d_2 *le nombre d'objets de la deuxième contrainte.*

La première fonction d'agrégation présentée, notée agg_{Id}*, permet d'obtenir une description parfaite de l'ensemble des états réalisables.*

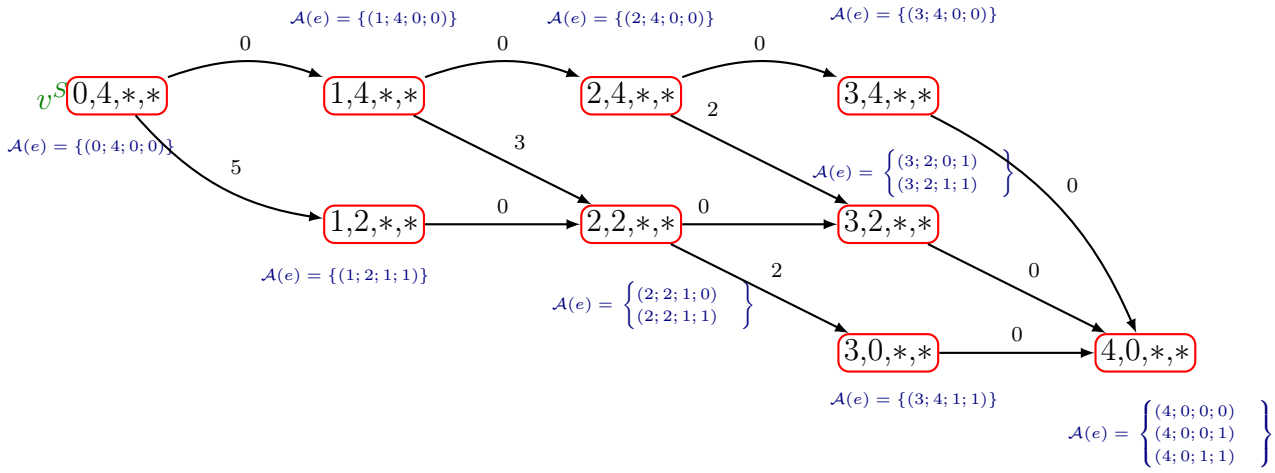


FIGURE 3.3 – Graphe obtenu en utilisant l'opérateur agg_{Id}, avec $agg_{Id}(\mathcal{A}(\boldsymbol{e})) = \mathcal{A}(\boldsymbol{e})$.

La deuxième fonction, notée $agg_{\boldsymbol{e}}$*, correspond à ne garder que l'empreinte des états et est définie par :*

$$\begin{array}{rccc} agg_e & : & \mathbb{P}(\mathcal{L}) & \rightarrow & \mathbb{P}(\mathcal{L}) \\ & & \boldsymbol{\ell} & \mapsto & \{\boldsymbol{\ell}' \in \mathcal{L} : id_{\mathcal{J}}(\boldsymbol{\ell}') = id_{\mathcal{J}}(\boldsymbol{\ell})\} \end{array}$$

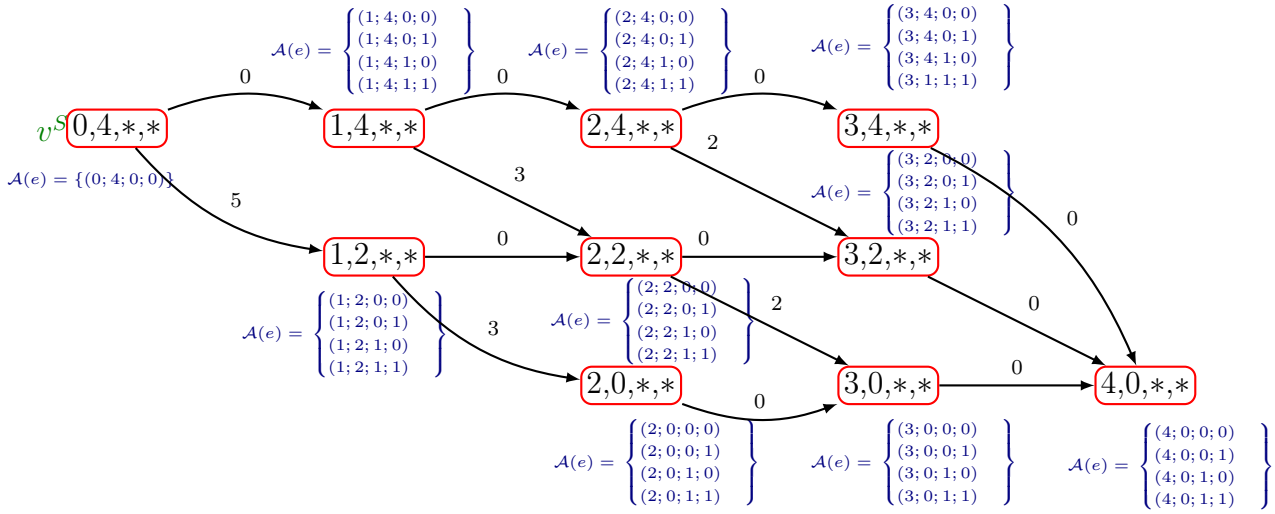


FIGURE 3.4 – Graphe obtenu en utilisant l'opérateur agg_e.

Proposition 2. *Le programme dynamique* $EDP^{\mathcal{J}}_{agg_{\boldsymbol{e}}}$ *obtenu en utilisant la fonction d'agrégation* $agg_{\boldsymbol{e}}$ *est équivalent à* $EDP^{\mathcal{J}}$.

Preuve. Pour une empreinte $\boldsymbol{e}$, le sur-ensemble des états réalisables considérés $agg(\mathcal{A}(\boldsymbol{e}))$ correspond à l'ensemble des états ayant cette empreinte. Les états de $\mathcal{A}(\boldsymbol{e})$ ont par définition la même empreinte.

$$agg_{\boldsymbol{e}}(\mathcal{A}(\boldsymbol{e})) = \{\boldsymbol{\ell} \in \mathcal{L} | id_{\mathcal{J}}(\boldsymbol{\ell}) = \boldsymbol{e}\} = S_{\mathcal{J}}(\boldsymbol{e})$$

□

3.5 Méthodes itératives

Nous avons vu dans les sections précédentes comment formaliser la relaxation d'un programme dynamique. La résolution de ce programme nous permet d'obtenir une borne duale du problème. On peut réintégrer des contraintes pour résoudre le problème, et nous avons vu comment intégrer toutes les contraintes dans le programme dynamique ou en utilisant un algorithme de labels. Dans cette section, nous allons présenter deux méthodes permettant de réintégrer les contraintes de manière itérative. Ces méthodes présentent un avantage majeur : elles permettent de résoudre optimalement le problème sans prendre en compte l'intégralité des contraintes additionnelles du problème.

3.5.1 Méthode générique

On définit différents statuts pour les dimensions de nos problèmes DP+LSC. Les dimensions du programme dynamique initial ainsi que les contraintes linéaires intégrées à l'espace d'états $\mathcal{J}^{DP} \subset \mathcal{J}$ seront dites **NET**. La validité de ces contraintes est vérifiée

par construction du graphe. Les dimensions correspondant aux contraintes linéaires vérifiées par l'algorithme de résolution seront dites **PRIMAL** et celles correspondant à des contraintes relâchées **OUT**.

NET	**PRIMAL**	**OUT**
$\mathcal{I}^{DP} \cup \mathcal{J}^{DP}$	$\mathcal{I}^{LC} \setminus \{\mathcal{J}^{DP} \cup \mathcal{K}\}$	$\mathcal{K}$

Le principe de ces méthodes est de résoudre à chaque itération une relaxation où les dimensions relâchées sont **OUT**, puis de raffiner la relaxation en prenant en compte des dimensions **OUT**.

On définit l'algorithme générique itératif :

Algorithme 8 : Méthode itérative générique

1 **Etape 1 : Définir les statuts des dimensions**
2 Définir les ensembles de dimensions **NET**, **PRIMAL** et **OUT**.
3 **Etape 2 : Construire la relaxation**
4 Construire le graphe correspondant à l'espace d'états des dimensions **NET**.
5 **Etape 3 : Résoudre la relaxation**
6 Résoudre le problème correspondant aux dimensions **NET** et **PRIMAL** en utilisant un algorithme de résolution des problèmes DP+LSC. Si la solution obtenue est réalisable pour le problème initial, **TERMINER**.
7 **Etape 4 : Raffiner la relaxation**
8 Modifier les ensembles **NET**, **PRIMAL** et **OUT**.
9 Retour à l'**étape 2**.

Pour assurer la convergence de l'algorithme, on peut s'assurer que la somme du nombre de dimensions en **NET** et en **PRIMAL** est strictement croissante. A partir de cette méthode, on peut définir différents algorithmes. Les trois premières étapes sont communes aux différents algorithmes possibles. La différence se fait sur la modification des ensembles de l'étape 4.

Le choix des dimensions qui vont permettre de raffiner la relaxation est donc un point clé de ces méthodes. La suite de ce chapitre présente deux méthodes itératives spécifiques.

3.5.2 SSDP

L'algorithme SSDP résoud des relaxations successives en construisant explicitement le graphe correspondant à chaque itération.

Dans l'algorithme 4, la construction de la première relaxation correspond à l'étape 2 de l'algorithme générique. Le graphe correspondant à cette relaxation est construit comme décrit dans la section 3.2.2. La résolution de la relaxation correspond à l'étape 3 correspond au plus court chemin dans le graphe précédemment construit.

A chaque itération, on choisit un sous-ensemble de dimensions **OUT** qu'on intègre dans l'ensemble **NET**. L'ensemble des dimensions **PRIMAL** est vide, on peut donc

résoudre les relaxations en utilisant l'algorithme de Bellman pour trouver un plus court chemin dans le graphe construit. Cette étape décrit le raffinement de la relaxation de l'algorithme générique.

La particularité de SSDP se situe dans l'étape 4 de l'algorithme générique. La construction de la nouvelle relaxation, appelée Sublimation dans l'algorithme SSDP, est réalisée à partir du graphe obtenu à l'itération précédente. Les performances de l'algorithme dépendent fortement du filtrage : les transitions et états supprimés dans le graphe d'une relaxation ne sont pas pris en compte dans la construction de la projection. Un filtrage basé sur les multiplicateurs de Lagrange et une implémentation efficace de la sublimation sont présentés dans la section 4.3.

3.5.3 DSSR

L'algorithme DDSR peut être vu comme la version itérative de l'algorithme de label setting. La différence avec SSDP se situe dans le changement de statuts des dimensions. Le sous-ensemble de dimensions **OUT** utilisé pour raffiner la relaxation va être ajouté aux dimensions **PRIMAL**. Le graphe support est construit à partir des dimensions **NET**, et cette ensemble de dimensions ne changera pas au cours de l'algorithme. Ce graphe est construit de la même manière que SSDP, mais il ne sera pas modifié durant les différentes itérations. La résolution de la relaxation se fait à partir de l'algorithme de label setting. La construction et la dominance des labels sont réalisées à partir des dimensions **PRIMAL**. Un label est réalisable s'il respecte les contraintes correspondant à des dimensions PRIMAL.

3.6 Conclusion

Dans ce chapitre, nous avons exprimé un formalisme permettant de décrire les problèmes pouvant se formuler comme un programme dynamique et des contraintes linéaires additionnelles, ainsi que les méthodes itératives d'agrégation permettant de résoudre ces problèmes. Ce formalisme a permis de mettre en avant une structure commune à ces méthodes. Ces méthodes sont basées sur un graphe support, correspondant au programme dynamique initial, et d'un algorithme de plus court chemin sous contraintes dans ce graphe. Les contraintes linéaires peuvent être prises en compte soit en les ajoutant dans l'espace d'états du programme dynamique initial, soit lors de la résolution de l'algorithme de plus court chemin sous contraintes. A partir de cette description générale, nous décrivons les ingrédients importants pour permettre aux méthodes d'être efficace en pratique dans le chapitre suivant.

Chapitre 4

Méthodes génériques permettant d'améliorer les méthodes itératives

4.1 Introduction

Les méthodes itératives de résolution de programmes dynamiques avec contraintes linéaires additionnelles ont été appliquées à différents problèmes avec succès. La méthode SSDP a notamment obtenu des résultats compétitifs sur des problèmes d'ordonnancement (cf [TFA09] et [TS13]), et DDSR sur des problèmes de plus court chemin élémentaire avec différentes contraintes (cf [RS08]). Malgré ces résultats, l'utilisation de ces méthodes reste rare dans la littérature. La raison principale réside dans la difficulté à mettre en place les différents éléments algorithmiques nécessaires à leur efficacité. Pour qu'elles soient performantes, il ne suffit pas d'implémenter ces méthodes, il faut ajouter des ingrédients qui vont permettre à ces méthodes d'être compétitives.

Nous mettons en avant deux points clés de ces méthodes. Le premier point clé est l'élimination d'états et de transitions non optimales. Si on arrive à supprimer au cours des itérations des états et des transitions qui ne peuvent appartenir à une solution optimale du problème initial, les projections de ces états et transitions pourront être supprimées des différentes relaxation. Le but est donc de trouver des règles génériques qui permettent cette suppression. Le deuxième point clé réside dans le choix des contraintes à ajouter à chaque itération. Plusieurs questions se posent : combien de contraintes doit-on réinjecter, quels critères permettent de choisir ces contraintes, ...

La première partie du chapitre est consacrée à un test de réalisabilité des états basé sur la notion de *dimension principale* des états. Une procédure d'élimination de transitions basée sur la relaxation Lagrangienne est ensuite présentée. La troisième section est consacrée à une fonction originale d'agrégation permettant un compromis entre précision de l'agrégation et quantité de données stockées. La quatrième partie présente le concept classique de dominance. Ensuite, nous présentons plusieurs techniques permettant de choisir les dimensions à ajouter à chaque itération de nos méthodes, et nous terminons en introduisant le concept d'énumération partielle des transitions.

4.2 Test de réalisabilité

Les programmes dynamiques étudiés dans ce manuscrit ont la particularité d'avoir une dimension principale strictement croissante. Pour chaque contrainte linéaire, on peut définir un intervalle de cette dimension, appelé *span* = [span_min, span_max] qui correspond à l'intervalle où la valeur de la ressource correspondante peut varier.

Définition 17 (Span d'une ressource)**.** *On définit pour chaque contrainte linéaire* $r \in \{1, ..., R\}$ *le span par rapport à la dimension principale par :*

$$span\_min_r = \min\{\boldsymbol{\ell}_1 : \boldsymbol{\ell} \in \mathcal{L}, \mathbf{t} \in \psi(\boldsymbol{\ell}), \Delta_{P+r}(\mathbf{t}) > 0\}$$

$$span\_max_r = \max\{\boldsymbol{\ell}_1 : \boldsymbol{\ell} \in \mathcal{L}, \mathbf{t} \in \psi(\boldsymbol{\ell}), \Delta_{P+r}(\mathbf{t}) > 0\}$$

Pour chaque contrainte linéaire $r \in \{1, ..., R\}$, le $span\_min_r$ correspond à la plus petite valeur de la dimension principale des états ayant une transition sortante impactant la contrainte linéaire. Comme la consommation des ressources n'est pas monotone, la valeur d'une ressource peut dépasser la valeur maximale de la ressource de l'état terminal. Le span permet de supprimer des états (ou labels) qui ne peuvent être dans un chemin réalisable.

Proposition 3. *Soit* $\boldsymbol{\ell} \in \mathcal{L}$ *un état d'une relaxation associée à* $\mathcal{J}$*.*

$$\textit{Si } \exists j \in \mathcal{J} \textit{ tq } \ell_j > b_j \textit{ et } \ell_1 > span\_max_j \textit{ alors } \boldsymbol{\ell} \textit{ n'est pas réalisable.}$$

Comme la valeur maximale du span a été dépassée, la valeur de la ressource ne pourra plus être modifiée et ne sera jamais réalisable.

4.3 Filtrage Lagrangien et bornes de complétion

Un des points important de la méthode de [Iba87] est l'élimination d'états et de transitions non optimales dans les relaxations successives. En utilisant l'algorithme de Bellman sur un graphe on peut obtenir le coût du plus court chemin depuis la source vers chaque sommet (forward), ainsi que le plus court chemin du puit vers chaque sommet (backward). Si on donne une borne supérieure du problème UB, alors un arc du graphe peut être supprimé si son coût additionné au plus court chemin entre le sommet source du graphe et la queue de l'arc et au plus court chemin entre le sommet puits du graphe et la tête de l'arc est plus grand que la borne UB. On appelle cette procédure le *filtrage* d'arc.

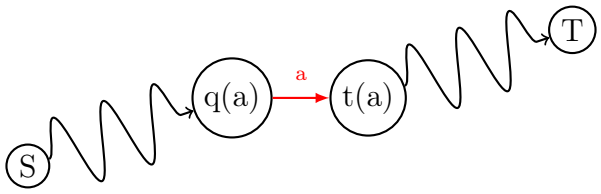


FIGURE 4.1 – Illustration du filtrage sur un arc a. L'arc a est supprimé si tout chemin ayant la forme ci-dessus a un coût trop faible.

Pour renforcer ce filtrage, on peut utiliser la relaxation Lagrangienne. Les dimensions OUT seront donc prises en compte dans l'objectif au lieu d'être relâchées complètement. A chaque itération de SSDP, la relaxation est basée sur l'ensemble de contraintes $\mathcal{K}$ correspondant aux dimensions OUT. Soit $\pi \in \mathbb{R}^m$ un vecteur de multiplicateurs de Lagrange associé aux contraintes de $\mathcal{K}$. Pour simplifier les notations, on note que π est toujours de taille m (on peut fixer à 0 les multiplicateurs associés aux contraintes qui ne sont pas relâchées).

Soit $\mathcal{L}_\mathcal{K}(\pi)$ la fonction duale Lagrangienne associée au vecteur de multiplicateurs π. Pour un ensemble $\mathcal{K}$ et un vecteur de multiplicateurs π, $\mathcal{L}_\mathcal{K}(\pi)$ donne une borne duale du problème initial. Pour obtenir une bonne borne duale, on doit trouver une approximation du problème dual Lagrangien $\min_{\pi \in \mathbb{R}^m} \mathcal{L}_\mathcal{K}(\pi)$. Ce problème peut être résolu par les techniques de sous-gradient classiques ou une de ses améliorations comme l'algorithme de Volume (2.2.2). Pour chaque vecteur π, on peut calculer $\mathcal{L}_\mathcal{K}(\pi)$ en modifiant le coût des arcs du graphe $G^\mathcal{K} = (V^\mathcal{K}, E^\mathcal{K})$ correspondant à la relaxation de l'ensemble $\mathcal{K}$ pour qu'ils prennent en compte la pénalisation de ces contraintes relâchées. Plus précisément, pour chaque arc $e = (v, v', t, c) \in E^\mathcal{K}$, le coût de e devient $c+ < \pi, \Delta(t) >$. Notons $G^\mathcal{K}_\pi$ le graphe avec les coûts modifiés par les multiplicateurs de Lagrange π. Un exemple de ce graphe modifié sera présenté dans la section 5.5.

Le filtrage peut être appliqué sur le graphe $G^\mathcal{K}_\pi$. On note $\hat{\gamma}^\pi_\mathcal{K}(v), \forall v \in V^\mathcal{K}$ le coût maximal d'un chemin entre le sommet source et v, et $\hat{\alpha}^\pi_\mathcal{K}(v), \forall v \in V^\mathcal{K}$ le coût maximal d'un chemin entre v et le sommet puits dans $G^\mathcal{K}_\pi$. Pour chaque sommet $v \in V^\mathcal{K}$, les valeurs $\hat{\alpha}^\pi_\mathcal{K}(v)$ et $\hat{\gamma}^\pi_\mathcal{K}(v)$ peuvent être calculées en deux passes en utilisant respectivement l'algorithme de programmation dynamique "forward" et "backward" de Bellman.

Proposition 4. *[Iba87] Soit $\mathcal{K}$ un ensemble de contraintes relâchées, $e = (v, v', t, c) \in E^\mathcal{K}$ et $\pi \in \mathbb{R}^m$ un vecteur de multiplicateur de Lagrange. La valeur suivante est une borne supérieure sur le coût de n'importe quel chemin de $G^\mathcal{K}_\pi$ entre le sommet source v^S et le sommet puits v^F utilisant l'arc e :*

$$\hat{\gamma}^\pi_\mathcal{K}(v) + c+ < \pi, \Delta(t) > + \hat{\alpha}^\pi_\mathcal{K}(v')$$

Ce résultat permet de retirer des transitions inutiles dans le graphe $G^\mathcal{K}$: si la borne supérieure obtenue d'un arc e est plus petite que la borne inférieure connue du problème, alors l'arc e ne peut être dans une solution optimale du problème.

Le filtrage Lagrangien est utilisé par la méthode SSDP lors de la résolution de la relaxation. L'étape de sublimation 11 permet de renforcer la relaxation courante en ajoutant des contraintes qui ont été violées dans la solution courante. Le graphe correspondant à une relaxation est construit à partir du graphe filtré à l'itération précédente. Une partie des contraintes de $\mathcal{K}$ est intégrée à l'espace d'état, on note $\mathcal{K}' \subset \mathcal{K}$ le nouvel ensemble de dimensions relâchées.

On définit la fonction $g^{\mathcal{K}}$ qui associe à un sommet de $G^{\mathcal{K}}$ un état de $EDP^{\mathcal{J}}_{agg}$ et la fonction $\hat{v}_{\mathcal{K}}(\boldsymbol{\ell}) = \{v \in V^{\mathcal{K}} | g^{\mathcal{K}}(v) = id_{\mathcal{K}}(\boldsymbol{\ell})\}, \forall \boldsymbol{\ell} \in \mathcal{L}$. Pour un sommet $v \in V^{\mathcal{K}}$, soit $\Gamma^+(v)$ (respectivement $\Gamma^-(v)$) l'ensemble des arcs sortant (respectivement arcs entrant). Pour un arc $a \in E^{\mathcal{K}}$, on note $\mu(a)$ la transition associée à l'arc a.

Soit $\rho_{\mathcal{K}}$ la fonction qui associe à un état $\boldsymbol{\ell}$ l'ensemble des transitions qui n'ont pas été filtrées durant l'itération correspondant à $\mathcal{K}$.

$$\rho_{\mathcal{K}}(\boldsymbol{\ell}) = \begin{cases} \emptyset & \text{si } \hat{v}_{\mathcal{J}}(\boldsymbol{\ell}) \notin V^{\mathcal{K}} \\ \{\mu(a) : a \in \Gamma^+(\hat{v}_{\mathcal{K}}(\boldsymbol{\ell})\} & \text{sinon} \end{cases}$$

Le programme dynamique construit lors de la phase de sublimation correspondant à l'ensemble $\mathcal{K}'$ est obtenu en remplaçant dans (3.4.2) la fonction $\psi(\boldsymbol{\ell})$ par la fonction $\hat{\psi}_{\mathcal{K}'}$:

$$\hat{\psi}_{\mathcal{K}'}(\boldsymbol{\ell}) = \psi(\boldsymbol{\ell}) \cap \rho_{\mathcal{K}}(\boldsymbol{\ell})$$

Le graphe de la relaxation $\mathcal{K}'$ est donc construit à partir du graphe de la relaxation $\mathcal{K}$. Les coûts "forward" de Bellman peuvent être calculés lors de la construction du graphe, ce qui permet de supprimer des arcs grâce aux bornes de complétion définies par la proposition suivante.

Proposition 5. *Bornes de complétion. Un arc $e = (v, v', t, c+ < \pi, \Delta(t) >)$ peut être retiré du graphe $G^{\mathcal{K}'}_{\pi}$ si*

$$\hat{\gamma}^{\pi}_{\mathcal{K}'}(v) + c+ < \pi, \Delta(t) > + \hat{\alpha}^{\pi}_{\mathcal{K}}(v_{\mathcal{K}}) < LB$$

avec $v_{\mathcal{K}} = \{v \in V^{\mathcal{K}} | g^{\mathcal{K}}(v) = id_{\mathcal{K}}(g^{\mathcal{K}'}(v'))\}$

4.4 Encodage efficace de la fonction d'agrégation

Dans la section 3.3.2, nous avons présenté la récurrence d'un programme dynamique associée à une fonction d'agrégation. Il existe plusieurs fonctions d'agrégation possibles respectant la propriété 1. C'est le cas de la fonction agg_{Id} introduite précédemment. Nous présentons désormais une nouvelle fonction d'agrégation respectant cette condition.

Définition 18 (Fonction d'agrégation utilisant les bornes des ressources)**.** *On définit la fonction d'agrégation :*

$$\widetilde{agg}(L) = \{l \in \mathbb{R}^m : \forall i \in \{1, ..., m\}, \min\{w_i : w \in L\} \leq l_i \leq \max\{w_i : w \in L\}\}, \forall L \in \mathbb{P}(\mathbb{R}^m)$$

Proposition 6. *L'opérateur $\widetilde{agg}$ respecte les conditions pour que EDP^J_{agg} soit une relaxation.*

La preuve est triviale, car $\forall \mathbf{l} \in L, \forall i \in \{1, ..., m\}, \min\{w_i : w \in L\} \leq l_i \leq \max\{w_i : w \in L\}$

L'opérateur $\widetilde{agg}$ permet d'obtenir un sur-ensemble des états réalisables, ce qui peut amener à considérer des états qui ne sont pas réalisables. La figure 4.2 permet d'illustrer le fonctionnement de l'opérateur $\widetilde{agg}$. Dans cette exemple, les états 2 et 4 sont des états atteignables du programme dynamique original, et l'ensemble de ces états ont la même empreinte. L'opérateur $\widetilde{agg}$ correspond au sur-ensemble composé des états $1, 2, 4$ et 5.

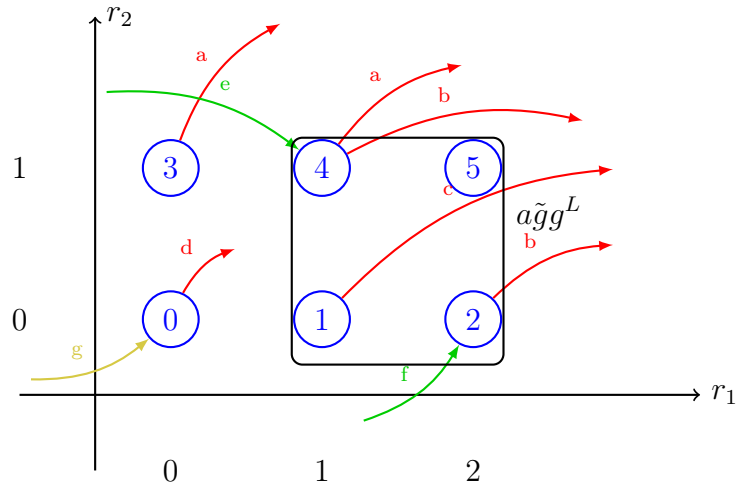


FIGURE 4.2 – Illustration de l'agrégateur $a\tilde{g}g$ sur un exemple à deux ressources. Les états atteignables du programme dynamique sont $L = \{2, 4\}$.

Une implémentation possible de cet agrégateur est de représenter un état par deux vecteurs de bornes supérieures et inférieures des états représentés.

Observation 1. *Soit $\mathcal{J}$ un ensemble de contraintes non relâchées et e un état de $EDP^{\mathcal{J}}_{a\tilde{g}g}$. On peut représenter l'état $\boldsymbol{\ell}$ par deux vecteurs $\boldsymbol{UB^\ell} \in \mathbb{R}^m$ et $\boldsymbol{LB^\ell} \in \mathbb{R}^m$ avec*

$$\boldsymbol{UB}^{\boldsymbol{\ell}}_i = \max_{l \in \mathcal{A}(e)} l_i \text{ et } LB^{\boldsymbol{\ell}}_i = \min_{l \in \mathcal{A}(e)} l_i, \forall i \in \{1, ..., m\}$$

Nous avons vu que cet agrégateur permet d'avoir un sur-ensemble des états réalisables. On peut définir des fonctions d'agrégation plus précises, mais le coût algorithmique serait plus important. L'avantage de cet agrégateur est de pouvoir utiliser seulement deux vecteurs pour représenter un état (au lieu d'un vecteur représentant l'empreinte). Nous illustrons notre agrégateur sur un exemple de sac à dos disjonctif.

Exemple 3 (Sac à dos binaire disjonctif)**.** *On considère le problème de sac à dos disjonctif présenté précédemment 2, en posant la taille maximale du sac à dos $W = 4$, et*

le nombre d'objets $n = 3$. *L'ensemble des objets* (w_i, p_i), *avec* w_i *le poids de l'objet* i *et* p_i *le profit de l'objet* i, *est le suivant :* $\{(2,5);(2,3);(2,2)\}$. *On ajoute deux contraintes linéaires de disjonction. Soit* x_i *la variable binaire égale à 1 si l'objet* i *est sélectionné dans le sac à dos.*

$$x_1 + x_2 \leq 1$$

$$x_1 + x_3 \leq 1$$

L'objet 1 *et l'objet* 2 *ne pourront pas être pris tous les deux dans une solution réalisable. Il en est de même pour l'objet* 1 *et l'objet* 3.

Un état du programme dynamique est un tuple (i, w, d_1, d_2), *où la dimension* i *correspond au dernier objet pouvant être pris, la dimension* w *correspond à la place disponible dans le sac à dos, et les dimensions* d_1 *et* d_2 *correspondent au nombre d'objets sélectionnés dans les contraintes de disjonction. Le graphe 4.3 représente le programme dynamique où les contraintes de disjonction ont été relâchées. Pour faciliter la lecture, un sommet du graphe sera représenté par les valeurs de l'état* $\boldsymbol{\ell}$ *associé à ce sommet, avec* $(\ell_1, \ell_2, \frac{UB_3^{\boldsymbol{\ell}}}{\boldsymbol{LB}_3^{\boldsymbol{\ell}}}, \frac{UB_4^{\boldsymbol{\ell}}}{\boldsymbol{LB}_4^{\boldsymbol{\ell}}})$.

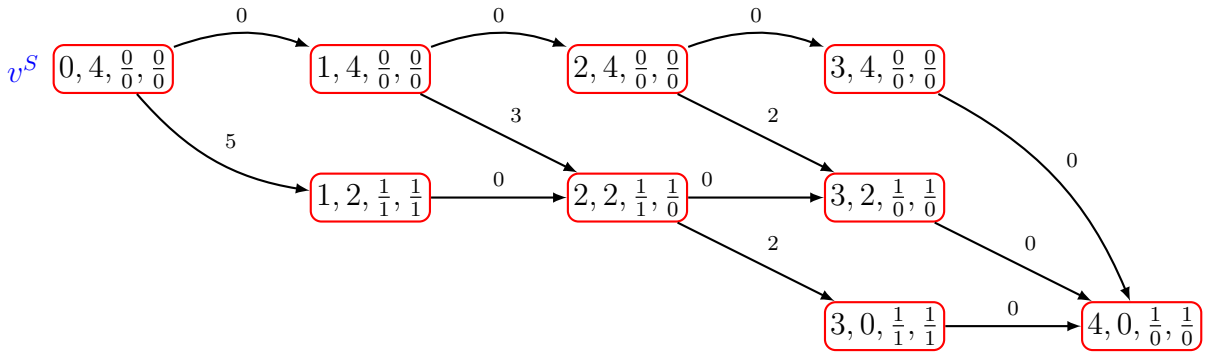


FIGURE 4.3 – Graphe du programme dynamique de la relaxation des contraintes de disjonction de l'exemple 3.

Remarque 6. *Pour chaque état* $\boldsymbol{\ell}$, *on note que* $\forall i \in NET, UB_i^{\boldsymbol{\ell}} = \boldsymbol{LB}_i^{\boldsymbol{\ell}}$. *En effet, comme ces dimensions correspondent à des contraintes non relâchées, la valeur de l'état est connue.*

4.5 Dominances

Un état du programme dynamique agrégé représente un ensemble de chemins dans le programme dynamique initial. On souhaite retirer des transitions qui sont dans des chemins qui ne peuvent correspondre à un chemin optimal. Pour retirer ces solutions, il existe deux types de tests : les tests de dominance et les tests de faisabilité.

Les tests de dominance ont été utilisés depuis longtemps dans la littérature, et les travaux de [JC11] ont permis de donner une description formelle de ceux-ci. Le but est de supprimer des solutions partielles qui ne peuvent pas correspondre à une solution optimale car il existe une solution dominante. Une solution S domine une solution S' si la réalisabilité de S' implique la réalisabilité de S, et que la valeur de S est meilleure que la valeur de S'.

Pour illustrer ces dominances, on s'intéresse au sac à dos binaire (cf 1). Un objet i domine un objet j si $w_i \leq w_j$, $p_i \geq p_j$ et une des deux inégalités est stricte. Toute solution partielle contenant l'objet j mais ne contenant pas l'objet i est dominée par la solution où on a pris l'objet i à la place de l'objet j. On en déduit que toute solution optimale contenant l'objet i doit contenir l'objet j.

4.6 Choix des dimensions à ajouter

Une question importante pour les méthodes itératives est le choix des contraintes qui vont être réintégrées pour obtenir une nouvelle relaxation. On doit choisir un sous-ensemble de contraintes qui ne seront plus relâchées. Ces contraintes peuvent être vérifiées soit en les ajoutant dans la construction du graphe (**NET**) ou par l'algorithme de label (**PRIMAL**). Si au moins une contrainte violée dans la solution optimale de la relaxation est ajoutée, alors la solution de la nouvelle relaxation sera différente. Plusieurs éléments permettent de savoir si le choix d'une dimension est bon : relaxation plus forte, graphe correspondant à la relaxation de petite taille, convergence rapide de la méthode, ... Intéressons nous d'abord à l'ensemble des contraintes violées dans la solution de meilleur coût.

4.6.1 Critères de sélection des dimensions

Plusieurs critères de sélection ont été proposés dans la littérature, notamment dans [BDD06]. Les auteurs appliquent la méthode itérative DSSR sur un problème de plus court chemin élémentaire avec contraintes de ressources. Le problème consiste à trouver un chemin dans un graphe respectant un certain nombre de contraintes de ressources, auquel on ajoute une contrainte de capacité 1 modélisant la contrainte d'élémentarité. Le premier critère consiste à ajouter les contraintes correspondant au nombre de visite d'un sommet les plus violées dans la solution optimale (soit le plus visité, soit tout les sommets les plus visités). On peut généraliser ce critère en choisissant les contraintes de ressource les plus violées.

Dans [TFA09], l'auteur donne un autre critère en utilisant SSDP sur un problème d'ordonnancement. On sélectionne parmi les jobs qui n'ont pas été ordonnancés, ceux qui ont le moins d'occurence dans le graphe. Le but est d'ajouter des contraintes violées qui ne feront pas trop augmenter la taille du graphe.

A chaque itération des méthodes itératives, on doit sélectionner une ou plusieurs dimensions correspondant à des contraintes violées dans la solution de la relaxation courante.

Nous allons présenter plusieurs critères permettant de sélectionner les contraintes à ajouter. Comme il n'existe pas de preuve théorique permettant de donner le meilleur critère, le choix de celui ci dépend fortement du problème traité. Un critère peut s'avérer très efficace pour un problème et inefficace pour un autre.

Critère 1 (Multiplicateur Lagrangien)**.**

Si on utilise la relaxation Lagrangienne, on peut associer à chaque contrainte un multiplicateur de Lagrange. On choisit la dimension correspondant à une contrainte linéaire qui a le plus grand multiplicateur de Lagrange (en valeur absolue). Plus le multiplicateur est grand, plus la violation sera pénalisée dans la fonction objectif. Ce critère est un des plus naturel, mais n'est pas toujours le plus efficace. De plus, les multiplicateurs ont un sens quand on ajoute une seule dimension à chaque itération, mais on ne peut pas assurer que l'ajout de plusieurs dimensions avec un multiplicateur élevé permet de résoudre plus rapidement le problème.

Critère 2 (Violation de la contrainte de ressource)**.**

Ce critère vient directement de l'article de [BDD06]. A chaque itération, on résout une relaxation du problème et on obtient une solution du problème relâché. Cette solution peut ne pas être réalisable pour le problème initial, et implique qu' une ou plusieurs contraintes linéaires ont été violées. Le principe de ce critère est de sélectionner la ou les dimensions correspondant aux dimensions les plus violées.

Ce critère est efficace par exemple sur le problème de plus court chemin élémentaire avec contraintes de ressources. La violation d'une contrainte correspond à un nœud visité plusieurs fois.

Pour certains problèmes, ce critère peut s'avérer inefficace. C'est le cas du problème du sac à dos temporel, traité plus tard dans ce manuscrit, où la violation d'une contrainte est égale à 1 dans tous les cas. On ne peut donc pas choisir de dimension par rapport à sa violation dans ce problème.

Critère 3 (Taille du graphe)**.**

Quand on ajoute une dimension en **NET**, on augmente la taille du graphe. C'est le cas pour la méthode SSDP qui construit un nouveau graphe relâché à chaque itération. Le contrôle de la taille du graphe est primordial pour cette méthode. On ne connait pas l'impact de l'ajout d'une dimension dans les ressources **NET**, mais on peut estimer l'augmentation si on ajoute une seule dimension.

Proposition 7 (Estimation de l'augmentation du réseau)**.** *Soit $i \in \mathcal{K}$ une dimension relâchée qui va être ajoutée dans les dimensions **NET**, et $G^{\mathcal{K}} = (V^{\mathcal{K}}, E^{\mathcal{K}})$. Le nombre de sommets du nouveau graphe est borné par :*

$$card(V^{\mathcal{K}\setminus\{i\}}) \leq \sum_{v \in V^{\mathcal{K}}} (UB_i^{\ell(v)} - LB_i^{\ell(v)}) + 1$$

Le nombre de sommets ajoutés dans le graphe d'une itération en ajoutant une dimension **OUT** en **NET** peut être calculé en appliquant l'algorithme de Bellman sur le réseau précédent.

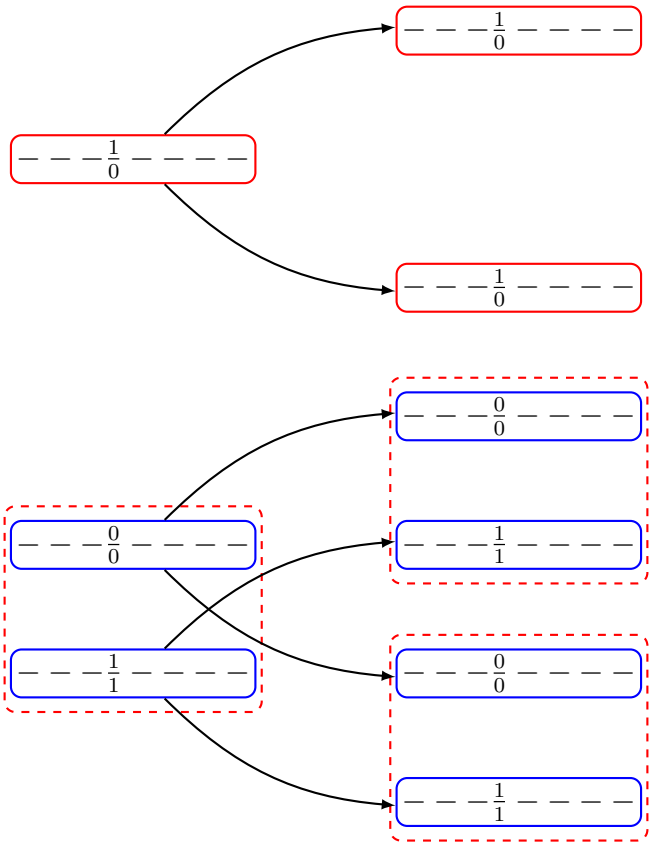


FIGURE 4.4 – Représentation de l'estimation du nombre de sommets ajoutés. L'extrait de graphe du haut représente une relaxation, où seulement une dimension relâchée est présentée. L'extrait de graphe du bas correspond à une relaxation où la dimension a été ajoutée en **NET**.

Critère 4 (Nombre de solutions où la ressource est violée)**.**

Les dimensions qui peuvent être ajoutées à l'espace d'états correspondent à des contraintes violées dans la solution de la relaxation utilisée. Quand les contraintes sont relâchées en utilisant la relaxation Lagrangienne, une solution non réalisable est obtenue pour chaque ensemble de multiplicateurs de Lagrange. L'idée de ce critère est de calculer le nombre de solutions dans lesquelles chaque contrainte est violée. Si une contrainte est violée dans plusieurs solutions duales, l'ajout de la dimension permet d'éliminer plusieurs solutions non réalisables.

4.6.2 Stratégies de sélection des dimensions

Nous avons plusieurs critères permettant de sélectionner la contrainte violée qui va être ajoutée à l'espace d'états. On peut sélectionner un sous-ensemble de contraintes violées et les ajouter durant la même itération à l'espace d'états. En pratique, l'ajout de plusieurs dimensions est plus efficace sur certains problèmes (par exemple le sac à dos temporel détaillé dans le chapitre suivant). Le problème de cette technique est que l'on augmente rapidement la taille du réseau (pour SSDP) ou le nombre de labels (DSSR) en ajoutant plusieurs contraintes en même temps.

Pour sélectionner ces dimensions, nous proposons plusieurs stratégies basées sur le span des ressources. Le but est de sélectionner un ensemble de ressources qui ont un span différent pour éviter les recombinaisons lors de l'ajout des dimensions. Soit $\mathcal{K}^{\neq} \subset \mathcal{K}$ l'ensemble des contraintes violées à une itération, et $G^{int} = (\mathcal{K}^{\neq}, E^{int})$ le graphe d'intervalle relatif aux intervalles $[span\_min_j, span\_max_j), \forall j \in \mathcal{K}^{\neq}$. Un arc dans ce graphe représente une paire de dimensions qui ne doivent pas être ajoutées ensemble (si l'on veut éviter une croissance trop rapide du graphe).

La première méthode d'ajout, appelée *stable pondéré*, revient à chercher un stable dans G^{int} maximisant un critère vu précédemment, tout en maitrisant la taille du réseau obtenu. Soit χ un des critères de sélection et χ^{size} le critère d'estimation de l'augmentation du réseau (critère 3). On définit un paramètre $TAILLEMAX$ donnant l'augmentation maximum autorisé.

Stable pondéré

$$\max \sum_{i \in \mathcal{K}^{\neq}} \chi_i x_i \tag{4.1}$$

$$\sum_{i \in \mathcal{K}^{\neq}} \chi_i^{size} x_i \leq \text{TAILLEMAX} \tag{4.2}$$

$$x_i + x_j \leq 1, \forall (i,j) \in E^{int} \tag{4.3}$$

$$x_i \in \{0,1\}, \forall i \in \mathcal{K}^{\neq} \tag{4.4}$$

Fixer $\chi = \chi^{size}$ ne semble pas adéquat car cela correspond à prendre le stable qui maximise l'augmentation du réseau. On peut définir un critère hybride permettant de prendre en compte cette augmentation.

Critère 5 (Hybride de l'augmentation du réseau χ^{hyb})**.** *Soit* $maxG^{\mathcal{K}^{\neq}} = \max_{i \in \mathcal{K}^{\neq}} \chi_i^{size}$

$$\forall j \in \mathcal{K}^{\neq}, \chi^{hyb} = \frac{maxG^{\mathcal{K}^{\neq}} - \chi_j^{size}}{maxG^{\mathcal{K}^{\neq}}}$$

La deuxième stratégie, appelée *stable de cardinalité*, consiste à résoudre le modèle précédent en utilisant des profits unitaires pour obtenir un stable de cardinalité maximale (noté $Cmax$). On cherche ensuite un stable de cardinalité supérieure à $Cmax * k^{stable}$, où k^{stable} est un paramètre dans $(0,1]$. Pour cela, on résout le problème suivant :

Stable de cardinalité

$$\max \sum_{i \in \mathcal{K}^{\neq}} \chi_i x_i \tag{4.5}$$

$$\sum_{i \in \mathcal{K}^{\neq}} x_i \geq Cmax * k^{stable} \tag{4.6}$$

$$x_i + x_j \leq 1, \forall (i,j) \in E^{int} \tag{4.7}$$

$$x_i \in \{0,1\}, \forall i \in \mathcal{K}^{\neq} \tag{4.8}$$

Cette stratégie vise à contourner un inconvénient majeur de la stratégie de stable pondéré, qui ajoute parfois trop peu de nouvelles contraintes, conduisant à une convergence lente de l'algorithme global. Selon toute vraisemblance, ce problème est également NP-complet, mais il est résolu efficacement par les solveurs modernes (c'est-à-dire que le temps nécessaire pour résoudre le problème est négligeable par rapport au temps global de l'algorithme).

La troisième stratégie, appelée *k-coloration*, permet de prendre en compte les contraintes violées aux itérations précédentes, en considérant un graphe plus grand que G^{int}. Pour exprimer cet ensemble, notons $\mathcal{K}^+ = \mathcal{J} \cup \mathcal{K}^{\neq}$ et définissons $E^{int}_+ = E^{int} \cap (\mathcal{K}^+ \times \mathcal{K}^+)$. On considère le sous-graphe $G^{int}_+ = (\mathcal{K}^*, E^{int}_+)$ de G^{int} induit par $\mathcal{K}^+$.

Soit k^{color} le nombre de couleurs qui ont été attribuées à l'itération courante. A chaque itération, on cherche un sous-graphe k^{color}-colorable de G^{int}_+de poids maximum. Si la solution est différente de $\mathcal{J}$, alors on ajoute les contraintes sélectionnées. Si la solution ne contient que $\mathcal{J}$, alors on augmente la valeur de k^{color} de une unité, et on résout à nouveau le modèle. Le paramètre k^{color} est initialisé à 1. Nous résolvons à plusieurs reprises le modèle suivant, avec z_{ij} une variable binaire qui indique si la contrainte i est assignée à la couleur j.

$$\max \sum_{i \in \mathcal{K}^{\neq}} \sum_{j=1,\ldots,k^{\text{color}}} \psi_i z_{ij}$$

$$z_{ij} + z_{\ell j} \leq 1, \forall (i,\ell) \in E^{int}_+, j \in \{1,\ldots,k^{\text{color}}\}$$

$$\sum_{j=1,\ldots,k^{\text{color}}} z_{ij} \leq 1, \forall i \in \mathcal{K}^{\neq}$$

$$\sum_{j=1,\ldots,k^{\text{color}}} z_{ij} = 1, \forall i \in \mathcal{K}^{+}$$

$$z_{ij} \in \{0,1\}, \forall i \in \mathcal{K}^+, j \in \{1,\ldots,k^{\text{color}}\}$$

Ce problème est également résolu avec un solveur de programmation linéaire. Le temps de résolution est négligeable par rapport au temps global.

Pour finir, nous définissons une quatrième stratégie, appelée *Adaptative*. Cette stratégie consiste à favoriser l'amélioration du gap aux premières itérations, et de favoriser le contrôle de la taille du réseau quand il devient trop grand. Pour les premières itérations,

on aimerait fermer le gap entre la borne primale et la borne duale le plus rapidement possible pour permette au filtrage Lagrangien d'être efficace. Mais quand le nombre de sommets du graphe devient trop important, il est plus important de contrôler cette taille, car un réseau trop grand peut amener à un sous-problème Lagrangien impossible à résoudre. Par conséquent, la stratégie adaptative utilise l'une des stratégies ci-dessus dans les premières itérations, et lorsque le nombre de sommets est supérieur à un seuil donné, le choix est uniquement basé sur la taille attendue du réseau.

4.6.3 Dimensions non violées

A chaque itération des méthodes itératives, on ajoute une ou plusieurs dimensions relâchées de **OUT** dans **NET** ou **PRIMAL**. Les dimensions pouvant être sélectionnées correspondent à des contraintes violées dans la solution de la relaxation courante. Mais on peut aussi ajouter des dimensions correspondant à des contraintes qui n'ont pas été violées dans la solution relâchée. Si aucune dimension correspondant à une contrainte violée n'est ajoutée, alors la solution de la relaxation suivante sera la même. Si on utilise la relaxation Lagrangienne, on obtient une solution relâchée à chaque itération de sous gradient. L'idée est de sélectionner un sous-ensemble de ces solutions relâchées et de prendre en compte les dimensions correspondant aux contraintes les plus souvent violées. On augmente donc le nombre de dimensions de **OUT** pouvant être ajoutées à l'espace d'états.

4.7 Enumération partielle

Si on considère une succession de transitions réalisables, cette chaîne de transitions peut ne pas être réalisable mais ne pas être retirée du réseau car chaque transition est réalisable localement, c'est-à-dire qu'il existe un chemin réalisable comportant cette transition ou que l'agrégation choisie ne permet pas de détecter l'irréalisabilité. L'idée est donc de combiner plusieurs transitions consécutives en une seule transition afin de renforcer localement certaines contraintes, y compris les contraintes qui n'ont pas été ajoutées à l'espace d'état.

Notre implémentation de cette idée utilise un paramètre k^{enum} qui contrôle la profondeur de l'énumération des transitions consécutives. Une nouvelle transition sera une chaîne de k^{enum} transitions initiales successives. Pour un état, au lieu de considérer les n transitions possibles à chaque état, on va considérer les $O(n^{k^{enum}})$ séquences possibles de k^{enum} transitions successives. La figure 4.5 illustre cette idée.

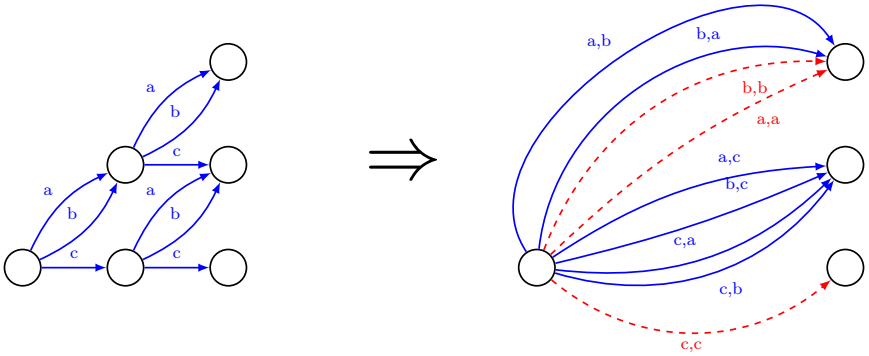


FIGURE 4.5 – Illustration de l'énumération partielle. Dans cet exemple, une transition ne peut pas être deux fois dans une solution optimale. Les séquences (a, a),(b, b) et (c, c) sont supprimées.

Si deux transitions sont incompatibles, alors les nouvelles transitions où ces deux transitions sont énumérées peuvent être retirées du programme dynamique. Cette énumération se montre efficace notamment sur les problèmes où le nombre de transitions possibles à chaque état est faible. Cette énumération permet d'obtenir un programme dynamique équivalent avec un nombre moins important d'états.

4.8 Conclusion

Dans ce chapitre, nous avons présenté différentes techniques permettant d'améliorer les méthodes itératives d'agrégation. Ces techniques sont génériques et ne dépendent donc pas du problème traité. Elles permettent de supprimer des états et des transitions qui ne peuvent être dans une solution optimale du problème original. L'ajout des dimensions est un élément clé de ces méthodes, mais il n'existe pas de preuve théorique permettant de déterminer qu'un critère est plus efficace qu'un autre sur l'ensemble des problèmes abordés. Les différentes techniques proposées permettent en pratique de trouver de bons critères pour une partie de ces problèmes, mais des techniques spécifiques aux problèmes sont souvent nécessaires pour rendre les méthodes itératives d'agrégation efficaces. Dans la suite de ce manuscrit, nous présentons différentes applications de ces méthodes.

Chapitre 5

Application détaillée de la méthode SSDP sur le problème de sac à dos temporel

5.1 Introduction

Le chapitre précédent présente des techniques génériques permettant d'améliorer les méthodes itératives de résolution de programmes dynamiques avec contraintes linéaires additionnelles. Il est parfois nécessaire de proposer des techniques spécifiques au problème étudié pour permettre aux méthodes itératives d'être efficaces en pratique.

Dans ce chapitre, nous appliquons la méthode SSDP à un problème de sac à dos temporel. Les résultats obtenus par [TFA09] ont montré que cette méthode peut être efficace en pratique. Le but est d'appliquer les techniques génériques précédentes à un problème et de fournir d'autres techniques propres au problème permettant d'obtenir des résultats compétitifs avec la littérature.

Une formulation originale du problème est présentée, et des tests de réalisabilité ainsi que diverses méthodes d'ajout de contraintes linéaires sont détaillés, ainsi qu'une analyse numérique de l'impact des différentes améliorations. Les résultats présentés dans ce chapitre ont été publiés dans [CDG20].

5.2 Présentation du problème

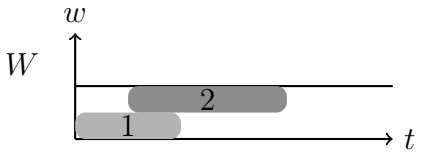
Le problème de sac à dos temporel (TKP) est une généralisation du problème de sac à dos classique, où les contraintes de capacité sont considérées durant une période de temps et les objets sont ajoutés dans le sac durant un intervalle de temps qui est différent pour chaque objet. La figure 5.1 représente un exemple d'instance de TKP.

Le nom "Temporal Knapsack" a été introduit par [BFH+05], mais le problème a été traité sous différents noms dans la littérature : problème d'allocation de bande passante ([CHT02]), d'allocation de ressource ([CCKR02]), voire problème d'ordonnancement sur plusieurs machines. Le problème peut se formuler ainsi.

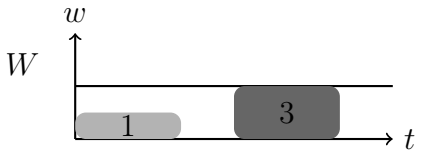
$W = 8$

i	w_i	p_i	$[s_i, f_i)$
1	4	2	$[0, 2)$
2	4	20	$[1, 4)$
3	8	200	$[3, 5)$

(a) Une instance de TKP



(b) Une première solution de profit 22



(c) Une seconde solution de profit 202

FIGURE 5.1 – Une instance de TKP avec trois objets et ces deux solutions maximales.

Problème 1 (Temporal Knapsack Problem)**.** *Soit $\mathbb{I} = \{1, \ldots, n\}$ un ensemble d'objets. Chaque objet $i \in \mathbb{I}$ a un profit $p_i \in \mathbb{R}^+$, un poids $w_i \in \mathbb{N}$, et un intervalle de temps $[s_i, f_i)$, où $s_i, f_i \in \mathbb{N}$ et $s_i < f_i$. De plus, notons $W \in \mathbb{N}$ le poids du sac à dos.*

Une solution réalisable est un sous-ensemble $\mathbb{I}'$ de $\mathbb{I}$ tel que pour chaque valeur $t \in \mathbb{N}$, la somme des poids des objets de $\mathbb{I}'$ dont l'intervalle de temps contient t est plus petite que W.

Le problème de sac à dos temporel est de trouver un sous-ensemble $\mathbb{I}'$ de $\mathbb{I}$ avec le profit maximal.

Dans sa version générale, le problème de sac à dos temporel est NP-difficile au sens fort [BSW14].

Les premiers résultats proposés pour ce problème sont centrés sur des caractérisations théoriques : un cas particulier polynomial [AS87], et des approximations [CHT02, CCKR02]. Deux programmes dynamiques ont été proposés par [CHT02] et [CFM13].

Les travaux les plus récents proposent des algorithmes de "branch-and-price" basés sur la reformulation Dantzig-Wolfe [CFM13]. L'idée est de partitionner l'horizon de temps en plusieurs périodes de temps consécutives (blocs). Pour chaque bloc, les variables correspondant aux objets dont l'intervalle de temps chevauche deux blocs sont dupliquées. Chaque sous-problème est un TKP plus petit, et le problème maître assure que les variables dupliquées correspondant au même objet ont la même valeur. Ces résultats ont été améliorés par [GI17] en utilisant une technique innovante de stabilisation permettant de renforcer le "branch-and-price" de [CFM13].

5.3 Modélisation

Rappelons tout d'abord la formulation de programmation entière utilisée dans la littérature (voir [CFM13] et [CFMT16].

Dans ce modèle, chaque variable binaire x_i est égale à un si l'objet i est sélectionné, zéro sinon. Comme indiqué dans [CFM13], il est suffisant de contrôler la contrainte de capacité uniquement aux dates d'entrée s_i des objets i.

$$\max \sum_{i \in \mathbb{I}} p_i x_i \tag{5.1}$$

$$\text{s.t.} \sum_{i \in \mathbb{I}: s_i \leq s_j < f_i} w_i x_i \leq W, \quad j \in \mathbb{I} \tag{5.2}$$

$$x_i \in \{0,1\}, \quad i \in \mathbb{I} \tag{5.3}$$

Pour simplifier la présentation de notre programme dynamique, nous proposons une formulation alternative du sac à dos temporel. Dans ce programme linéaire en nombres entiers, nous voyons le problème comme une succession d'événements où les décisions de prendre un objet ou de l'enlever doivent être prises.

Soit $\mathcal{E} = (e_1, \ldots, e_{2n})$ une liste triée d'indices appelés *événement*. Chaque événement e est relié à un objet $i(e) \in \mathbb{I}$. Il existe deux types d'événements : les événements représentant le début de la fenêtre de temps d'un objet ($\mathcal{E}^{\text{in}}$) et les événements représentant la fin d'une fenêtre de temps ($\mathcal{E}^{\text{out}}$).

Notons $\hat{t}(e)$ l'instant de temps relié à l'événement e, *i.e.* $s_{i(e)}$ si $e \in \mathcal{E}^{\text{in}}$ et $f_{i(e)}$ si $e \in \mathcal{E}^{\text{out}}$.

Les événements e de $\mathcal{E}$ sont triés de la façon suivante : $e \prec e'$ si $\hat{t}(e) < \hat{t}(e')$ ou $(\hat{t}(e) = \hat{t}(e') \wedge e \in \mathcal{E}^{\text{out}} \wedge e' \in \mathcal{E}^{\text{in}})$ (les égalités sont départagées arbitrairement).

Notons également $2n+1$ l'indice d'un événement fictif dont le temps correspondant est plus grand que le temps correspondant de tous les événements.

Présentons maintenant le nouveau programme linéaire en nombres entiers. Chaque variable de décision est reliée à un événement. Pour chaque événement e, on définit une variable binaire y_e qui indique si l'action correspondant à l'événement e est réalisée ou non. Si $e \in \mathcal{E}^{\text{in}}$, la décision correspond à ajouter l'objet $i(e)$ à la solution courante. Si $e \in \mathcal{E}^{\text{out}}$, la décision correspond à retirer l'objet $i(e)$ de la solution courante. Il est évident que dans une solution réalisable, un objet ne peut quitter le sac que s'il y est entré.

Chaque variable ϕ_e ($e = 1, \ldots, 2n$) est égale au poids cumulé des objets du sac à la fin de l'événement e.

$$\max \sum_{e \in \mathcal{E}} \frac{1}{2} p_{i(e)} y_e \tag{5.4}$$

$$\phi_1 = w_{i(1)} y_1 \tag{5.5}$$

$$\phi_e = \phi_{e-1} + w_{i(e)} y_e \quad e \in \mathcal{E}^{\text{in}} \setminus \{1\} \tag{5.6}$$

$$\phi_e = \phi_{e-1} - w_{i(e)} y_e \quad e \in \mathcal{E}^{\text{out}} \tag{5.7}$$

$$\phi_e \leq W \quad e = 1, \ldots, 2n \tag{5.8}$$

$$\phi_{2n} = 0 \tag{5.9}$$

$$y_e - y_{e'} = 0 \quad e \in \mathcal{E}^{\text{in}}, e' \in \mathcal{E}^{\text{out}}, i(e) = i(e') \tag{5.10}$$

$$y_e \in \{0,1\}, \quad e = 1, \ldots, 2n \tag{5.11}$$

$$\phi_e \in \mathbb{R}_+, \quad e = 1, \ldots, 2n \tag{5.12}$$

La fonction objectif est similaire à celle du modèle (5.1). La seule différence est que le profit est réparti sur les deux événements de chaque objet. Notons que la répartition en deux parts égales est arbitraire, et que n'importe quelle paire de valeurs réelles dont la somme est égale à $p_{i(e)}$ est valide. Les contraintes (5.5)–(5.7) assurent que la consommation de capacité à la fin de chaque événement est conforme à la composition du sac à dos. Les contraintes (5.8) et (5.9) garantissent que les contraintes de capacité sont satisfaites. Les contraintes (5.10) permettent de vérifier que si un objet entre dans le sac, alors il doit sortir. Notons que la contrainte (5.9) est redondante si aucune autre contrainte n'est relâchée.

5.4 Programme dynamique

Dans la littérature, deux programmes dynamiques ont été proposés. Dans l'article [CHT02], les auteurs ont proposé un programme dynamique où les états sont triés par niveau correspondant aux événements décrits précédemment. Un état $(e, \mathbf{d})$ est caractérisé par l'événement courant $e \in \mathcal{E}$ et $\mathbf{d} \in \{0,1\}^n$ le vecteur caractéristique de l'ensemble d'objets présents dans le sac. Dans l'article [CFM13], un autre programme dynamique a été proposé. Il est basé sur une reformulation du problème comme un problème de chemin de profit maximum dans un graphe de taille exponentielle. Ce graphe possède une couche pour chaque contrainte de sac à dos (5.2). Dans chaque couche, un nœud est créé pour chaque sous-ensemble d'objets réalisable qui peut être dans le sac à dos. Ensuite, il existe un arc entre deux nœuds de deux couches successives si et seulement si leurs contenus sont compatibles (*i.e.* si un objet est présent dans le premier sous-ensemble, alors il est présent dans le deuxième, et inversement).

Le coût d'un arc est égal à la somme des profits des objets qui sont ajoutés pour obtenir la nouvelle configuration. Selon les auteurs, cette méthode permet de résoudre les petites instances de la littérature, mais ne peut pas être appliquée quand il y a trop d'objets pouvant être présents dans le sac à dos à un instant de temps car le nombre d'états du programme dynamique est exponentiel en cela.

Notre programme dynamique est basé sur le concept d'*événement*. C'est une adaptation du programme dynamique de [CHT02] où nous ajoutons aux états une information redondante (la consommation de capacité), qui sera utile dans notre relaxation. Le modèle fonctionne de la même manière que le modèle (5.4)–(5.12) dans le sens où la capacité courante est mise à jour à chaque événement récursivement, en s'assurant que la contrainte de capacité reste satisfaite et que les décisions relatives aux objets restent cohérentes.

Nous allons décrire le programme dynamique en définissant les *états* et les *transitions*. On définit un *état* comme un tuple $(e, w, \mathbf{d})$ où $e \in \mathcal{E}$ est l'événement courant, $w \in \mathbb{Z}_+$ l'occupation courante du sac à dos et $\mathbf{d} \in \{0,1\}^n$ le vecteur caractéristique de l'ensemble d'objets présents dans le sac à dos. Notons que w est redondant, car il peut être déduit du vecteur $\mathbf{d}$. Une *transition* est définie par un tuple $(\Delta_e, \Delta_w, \Delta_{\mathbf{d}}, p)$ où $\Delta_e \in \mathbb{Z}_+$ décrit l'augmentation de l'indice de l'événement courant, $\Delta_w \in \mathbb{Z}$ la consommation de la capacité quand la décision est prise, $\Delta_{\mathbf{d}} \in \{-1,0,1\}^n$ un vecteur qui met à jour le contenu

du sac à dos et $p \in \mathbb{R}_+$ le profit obtenu quand la décision est prise.

Les décisions possibles pouvant être prises sont définies par ψ, la fonction qui associe pour chaque état un ensemble de transitions réalisables. Soit $\mathbf{0}$ le vecteur nul de taille n, et $\boldsymbol{\varepsilon}_k \in \{0,1\}^n$ le vecteur caractéristique de l'ensemble $\{k\}$, pour $k \in \mathbb{I}$. Pour chaque état réalisable $(e, w, \mathbf{d})$, la fonction $\psi((e, w, \mathbf{d}))$ est calculée ainsi :

$$\psi((e,w,\mathbf{d})) = \begin{cases} \left\{(1,0,\mathbf{0},0),(1,w_{i(e)},\boldsymbol{\varepsilon}_{i(e)},\frac{1}{2}p_{i(e)})\right\} & \text{if } e \in \mathcal{E}^{\text{in}} \wedge w + w_{i(e)} \leq W \\ \{(1,0,\mathbf{0},0)\} & \text{if } e \in \mathcal{E}^{\text{in}} \wedge w + w_{i(e)} > W \\ \left\{(1,-w_{i(e)},-\boldsymbol{\varepsilon}_{i(e)},\frac{1}{2}p_{i(e)})\right\} & \text{if } e \in \mathcal{E}^{\text{out}} \wedge \mathbf{d}_{i(e)} = 1 \\ \{(1,0,\mathbf{0},0)\} & \text{if } e \in \mathcal{E}^{\text{out}} \wedge \mathbf{d}_{i(e)} = 0 \end{cases} \tag{5.13}$$

Quand on considère un événement $e \in \mathcal{E}^{\text{in}}$, deux transitions sont possibles : la première correspond à sélectionner l'objet $i(e)$ et la deuxième à ne pas sélectionner l'objet $i(e)$ (l'objet peut être sélectionné seulement si la capacité restante est suffisante). Quand on considère un événement $e \in \mathcal{E}^{\text{out}}$, une seule transition est possible et elle dépend de la valeur de $\mathbf{d}_{i(e)}$. La fonction de coût α de chaque état $(e, w, \mathbf{d})$ est exprimé de manière récursive en "backward". Dans le reste de ce manuscrit, lorsque deux vecteurs sont considérés, les symboles + et − représentent respectivement l'addition et la soustraction composante par composante.

$$\alpha((e,w,\mathbf{d})) = \begin{cases} \max\limits_{(\Delta_e,\Delta_w,\Delta_{\mathbf{d}},p)\in\psi((e,w,\mathbf{d}))} \{p + \alpha((e+\Delta_e, w+\Delta_w, \mathbf{d}+\Delta_{\mathbf{d}}))\} & \text{if } e \in \{1,\ldots,2n\} \\ 0 & \text{if } e = 2n+1, w = 0, \boldsymbol{d} = \mathbf{0} \end{cases} \tag{5.14}$$

La valeur optimale du TKP est $\alpha((1,0,\mathbf{0}))$.

Comme nous l'avons vu dans la section 3.2.2, le programme dynamique peut être modélisé comme un problème de plus court chemin dans un graphe. La figure 5.2 illustre le graphe correspondant au programme dynamique (5.14) appliqué à l'instance de l'exemple 5.1a.

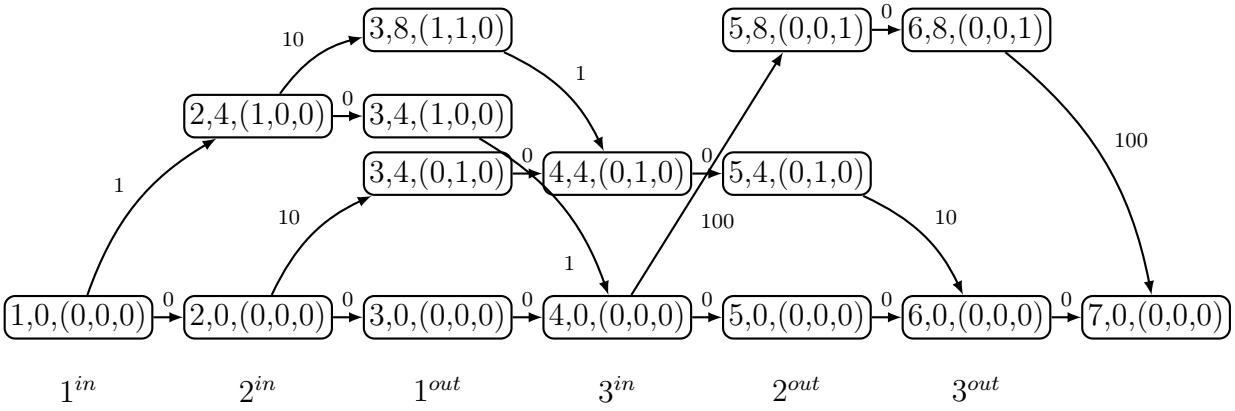


FIGURE 5.2 – Graphe correspondant au programme dynamique (5.14) appliqué à l'instance de l'exemple 5.1a.

A une itération de l'algorithme, la relaxation est basé sur l'ensemble $\mathcal{J}$ de dimensions correspondant à des contraintes (5.10). On note r_i la dimension de la contrainte (5.10) correspondant à l'objet i. Nous utilisons la relaxation $EDP^{\mathcal{J}}_{a\tilde{g}g}$ présentée dans la section précédente. La figure suivante permet d'illustrer cette relaxation.

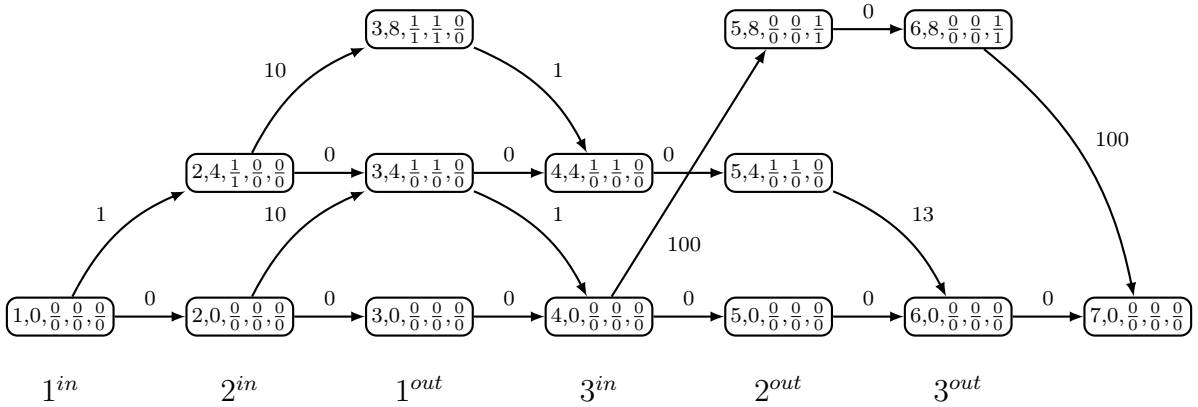


FIGURE 5.3 – Représentation du graphe correspondant au programme dynamique relaché pour $\mathcal{J} = \emptyset$ en utilisant l'opérateur $a\tilde{g}g$. La solution optimale de la relaxation a un coût de 211.

5.5 Suppression d'arcs et d'états du programme dynamique

Nous avons vu qu'il existe plusieurs méthodes pour supprimer des arcs et des états du programme dynamique. Ces suppressions peuvent être faites dans les relaxations

successives et doivent être valables dans les relaxations suivantes. Dans cette section, nous allons montrer les différents techniques mises en place pour le problème de sac à dos temporel.

Nous avons vu dans la section 4.3 que la relaxation Lagrangienne nous permet de filtrer une partie des transitions du graphe et permet d'obtenir une borne duale du problème.

Soit $\boldsymbol{\pi} \in \mathbb{R}^n$ le vecteur de multiplicateurs de Lagrange associé aux contraintes (5.10) pour les indices de $\mathcal{K}$. Pour faciliter les notations, on suppose que $\boldsymbol{\pi}$ et $\mathbf{d}$ sont toujours de taille n. Pour un ensemble donné $\mathcal{J}$, et un vecteur de multiplicateurs $\boldsymbol{\pi}$, la fonction duale Lagrangienne est définie par :

$$L_{\mathcal{J}}(\boldsymbol{\pi}) = \max \sum_{e \in \mathcal{E}^{\text{in}}} \left(\frac{1}{2} p_{i(e)} + \boldsymbol{\pi}_{i(e)} \right) y_e + \sum_{e \in \mathcal{E}^{\text{out}}} \left(\frac{1}{2} p_{i(e)} - \boldsymbol{\pi}_{i(e)} \right) y_e \tag{5.15}$$

$$(5.6) - (5.9), (5.11), (5.12) \tag{5.16}$$

$$y_e - y_{e'} = 0 \quad e \in \mathcal{E}^{\text{in}}, e' \in \mathcal{E}^{\text{out}}, i(e) = i(e'), r_{i(e)} \in \mathcal{J} \tag{5.17}$$

Dans la figure 5.4, nous représentons le graphe obtenu en ajoutant les coûts Lagrangiens à la relaxation de l'espace d'états initiale. Il y a un multiplicateur pour chaque contrainte (5.10). On choisit les multiplicateurs $(0, -3, 0)$ pour pénaliser la possibilité de prendre l'objet 2 dans le sac sans le sortir. La nouvelle solution optimale est la même que dans la figure 5.3, mais son coût est de 208, correspondant à une meilleure borne. Notons que si on choisit le vecteur de multiplicateurs $(0, -10, 0)$, la relaxation permet de résoudre le problème optimalement.

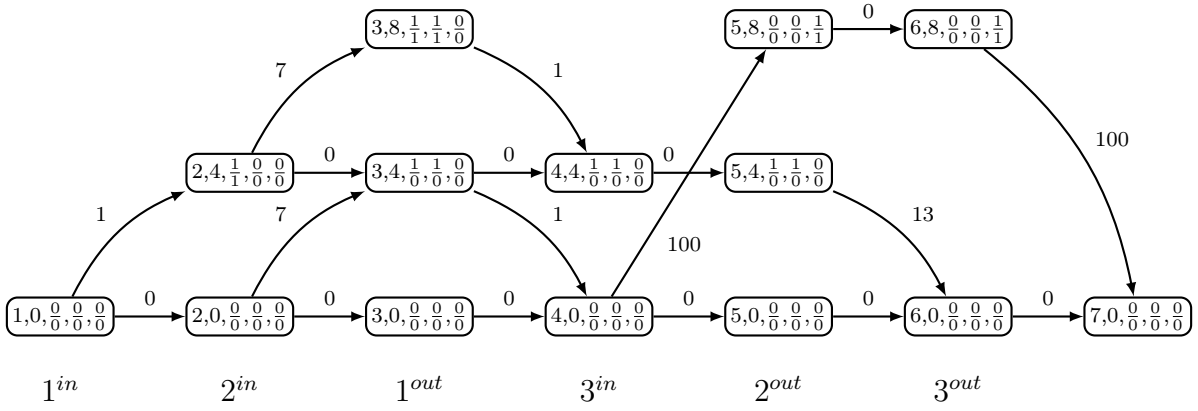


FIGURE 5.4 – Représentation du graphe correspondant au programme dynamique relaché pour $\mathcal{J} = \emptyset$,avec les coûts Lagrangiens $(0, -3, 0)$, en utilisant l'opérateur $a\tilde{g}g$. Ajouter l'objet 2 a désormais un profit de 7 au lieu de 10, et retirer l'objet 2 a désormais un profit de 13 au lieu de 10.

Dans la suite, un *état réalisable* est défini comme un état qui peut être généré depuis

le sommet source en appliquant une séquence de transitions réalisables. On note $\mathcal{L}^+$ l'ensemble des états réalisables, et $G_{\mathcal{J}}$ le graphe correspondant au programme dynamique $EDP^{\mathcal{J}}_{a\tilde{g}g}$. On rappel que $V_{\mathcal{J}}$ est l'ensemble des sommets considérés durant la résolution de la relaxation associée à $\mathcal{J}$. Ainsi, chaque sommet de $V_{\mathcal{J}}$ qui n'est pas lié à un état réalisable de $EDP^{\mathcal{J}}_{a\tilde{g}g}$ peut être supprimé du graphe sans nuire à la validité de l'algorithme. Une condition suffisante pour que le processus de résolution globale soit valide est que pour tout $\mathcal{J}$, il existe un chemin dans le graphe $G_{\mathcal{J}}$ associé à la relaxation dont les arcs forment une séquence optimale de transition.

Nous avons vu que les sommets $v \in V^{\mathcal{J}}$ du graphe sont représentés par une empreinte $\boldsymbol{e}(v)$, et que chaque empreinte $\boldsymbol{e}(v)$ représente un sur-ensemble d'états de $\mathcal{L}$ caractérisé par deux vecteurs $\mathbf{LB}^{\boldsymbol{e}(v)}$ et $\mathbf{UB}^{\boldsymbol{e}(v)}$. Pour une empreinte $\boldsymbol{e} = (e, w, \mathbf{LB}, \mathbf{UB})$ tel que $e \in \mathcal{E}^{\text{out}}$, si $r_{i(e)} \in \mathcal{J}$, deux transitions sont possibles (enlever ou non l'objet $i(e)$). Dans certains cas, tous les états de l'espace d'états original représenté par l'empreinte $\boldsymbol{e}$ contiennent $i(e)$. Dans d'autres cas, aucun état ne contient l'objet. Dans ces cas, il n'y a qu'une seule transition possible.

Pour chaque sommet, on note $\mathbf{d}^{\eta}(v)$ le vecteur qui pour chaque objet indique s'il est présent ou non dans le sac, avec :

$$\forall i \in \mathbb{I}, \mathbf{d}^{\eta}(v)_i = \begin{cases} 1 & \text{si } LB_{r_i}^{\boldsymbol{e}(v)} = 1 \\ 0 & \text{si } UB_{r_i}^{\boldsymbol{e}(v)} = 0 \\ \emptyset & \textit{sinon} \end{cases} \tag{5.18}$$

On a donc $\mathbf{d}^{\eta}(v)_i = 1$ si l'objet est dans le sac à dos pour l'état correspondant à v, même si la contrainte correspondant à l'objet est relâchée. Toutes les transitions amenant à cet état contiennent donc l'objet i.

Pour retirer des états et transitions en utilisant des tests de réalisabilité, on attache à chaque sommet une information correspondant à des contraintes redondantes. Pour chaque sommet $v \in V^{\mathcal{J}}$, on définit $q^{\eta}_{\min}(v)$ (respectivement $q^{\eta}_{\max}(v)$) une borne inférieure (resp. supérieure) du nombre d'objets pouvant être dans le sac à dos des états de $\boldsymbol{e}(v)$. On rappelle que pour une empreinte $\boldsymbol{e}$, $a\tilde{g}g(\mathcal{A}(\boldsymbol{e}))$ représente un sur-ensemble des états atteignables.

Plusieurs techniques peuvent être implémentés pour détecter les états qui ne sont pas réalisables aux premières étapes de la méthode.

Le premier test de faisabilité vérifie que le nombre d'objets dans le sac à dos est cohérent.

Proposition 8. *Soit $\mathcal{J} \subseteq \mathcal{I}$ et $v \in V_{\mathcal{J}}$. Si $card(\{i \in \mathbb{I} : \mathbf{d}^{\eta}(v)_i \neq 0\}) < q^{\eta}_{\min}(v)$ ou $card(\{i \in \mathbb{I} : \mathbf{d}^{\eta}(v)_i = 1\}) > q^{\eta}_{\max}(v)$ alors $a\tilde{g}g(\mathcal{A}(\boldsymbol{e}(v))) \cap \mathcal{L}^+ = \emptyset$.*

Un autre test de faisabilité est basé sur l'ensemble des poids possibles des sous-ensembles d'objets qui peuvent être dans le sac à dos à un événement donné. L'idée est de pré-calculer l'ensemble de ces poids réalisables pour chaque événement et de supprimer les sommets dont la consommation de ressources de l'état associé ne correspond à aucune configuration possible. Pour chaque événement e, on note :

$\mathcal{F}(e) = \{\sum_{i\in S} w_i : S \subseteq \mathbb{I}(e), \sum_{i\in S} w_i \leq W\}$, qui correspond à tout les poids atteignables par un sous-ensemble d'objets. Chacun de ces ensembles peut être calculé en $\mathcal{O}(nW)$ en utilisant un algorithme de programmation dynamique simple. On déduit de cet ensemble la proposition suivante :

Proposition 9. *Soit $\mathcal{J} \subseteq \mathcal{I}$ et $v \in V^{\mathcal{J}}$. Si $w \notin \mathcal{F}(e)$ alors $\tilde{agg}(\mathcal{A}(\boldsymbol{e}(v))) \cap \mathcal{L}^+ = \emptyset$.*

Cette règle peut être améliorée en considérant les informations supplémentaires obtenues à partir de $\mathbf{d}^\eta(s)$. On pré-calcule, pour chaque événement e et chaque objet $i \in \mathbb{I}(e)$, $\mathcal{F}^+(e,i)$ et $\mathcal{F}^-(e,i)$ qui sont respectivement les poids possibles atteignables en utilisant l'objet i, et les poids atteignables sans l'objet i. On a $\mathcal{F}^+(e,i) = \{\sum_{j\in S\cup\{i\}} w_j : S \subseteq \mathbb{I}(e) \setminus \{i\}, \sum_{j\in S} w_j \leq W - w_i\}$ et $\mathcal{F}^-(e,i) = \{\sum_{j\in S} w_j : S \subseteq \mathbb{I}(e) \setminus \{i\}, \sum_{j\in S} w_j \leq W\}$.

Proposition 10. *Soit $\mathcal{J} \subseteq \mathcal{I}$, $v \in V_{\mathcal{J}}$ et $i \in \mathbb{I}(e)$. Si $\mathbf{d}^\eta(v)_i = 1$ et $w \notin \mathcal{F}^+(e,i)$ alors $\tilde{agg}(\mathcal{A}(\boldsymbol{e}(v))) \cap \mathcal{L}^+ = \emptyset$. De la même manière, si $\mathbf{d}^\eta(v)_i = 0$ et $w \notin \mathcal{F}^-(e,i)$, alors $\tilde{agg}(\mathcal{A}(\boldsymbol{e}(v))) \cap \mathcal{L}^+ = \emptyset$.*

Le résultat suivant permet de détecter les sommets liés aux états qui ne peuvent être générés qu'en supprimant un objet du sac à dos sans l'ajouter au préalable. Dans de tels cas, le poids enregistré dans l'état peut devenir inférieur au poids des objets dont la présence dans le sac à dos est connue à coup sûr.

Proposition 11. *Soit $\mathcal{J} \subseteq \mathcal{I}$ et $v \in V_{\mathcal{J}}$. Si $\sum_{i\in\mathbb{I}:\mathbf{d}^\eta(v)_i=1} w_i > w$ alors $\tilde{agg}(\mathcal{A}(\boldsymbol{e}(v))) \cap \mathcal{L}^+ = \emptyset$.*

Pour une empreinte $(e, w, \mathbf{LB}, \mathbf{UB})$, le test de faisabilité suivant intègre les limites du nombre d'objets dans le sac à dos pour assurer la cohérence de l'ensemble des objets potentiellement dans le sac à dos en respectant w.

Proposition 12. *Soit $\mathcal{J} \subseteq \mathcal{I}$ et $v \in V_{\mathcal{J}}$. Soit $S^1 = \{i \in \mathbb{I}(e) : \mathbf{d}^\eta(v)_i \neq 0\}$ l'ensemble des objets potentiellement dans le sac à dos. Si $\max\left\{\sum_{i\in S} w_i : S \subseteq S^1, |S| \leq q^\eta_{\max}(v)\right\} < w$ ou $\min\left\{\sum_{i\in S} w_i : S \subseteq S^1, |S| \geq q^\eta_{\min}(v)\right\} > w$ alors $\tilde{agg}(\mathcal{A}(\boldsymbol{e}(v))) \cap \mathcal{L}^+ = \emptyset$.*

Dans le problème de sac à dos classique, un objet domine un autre si son profit est supérieur et son poids est inférieur. On peut étendre cette règle de dominance au problème de sac à dos temporel.

Proposition 13. *L'objet i est dominé par l'objet j si $p_i \leq p_j$, $w_i \geq w_j$, $s_i \leq s_j$, $f_i \geq f_j$ et au moins une de ces inégalités est stricte.*

Preuve. Comme j a un plus grand profit et un poids inférieur à i, et que sa fenêtre de temps est incluse dans celle de i, on peut remplacer i par j dans toutes solutions réalisables pour obtenir une solution réalisable de meilleur coût. □

L'ensemble de ces tests permet de supprimer des transitions et des états dans les relaxations successives. Les résultats obtenus sont étudiés dans la section suivante.

5.6 Résultats expérimentaux

Dans cette section, nous exposons les résultats expérimentaux obtenus sur le problème de sac à dos temporel. Nous évaluons l'impact de chaque raffinement sur les performances de SSDP. Enfin, nous comparons nos résultats avec ceux de la littérature et ceux obtenus en utilisant un solveur commercial de programmation linéaire en nombres entiers. Dans cette section, nous considérons qu'une instance est résolue si l'algorithme trouve une solution optimale et prouve son optimalité.

Toutes nos expérimentations ont été effectuées en utilisant un Intel Xeon E5-2680 v3 2,5 GHz 2 Dodeca-core Haswell avec 128Go RAM. Pour chaque instance, notre code a été exécuté sur 6 threads et avec une limite de 32Go de RAM. Tous les modèles considérés dans les sous-programmes sont résolus avec IMB ILOG Cplex 12.7.

Nous utilisons les instances proposées dans [CFM13], qui sont divisées en deux groupes. Les résultats pour le premier groupe I ne sont pas reportés car les méthodes de [GI17] et nos méthodes peuvent résoudre toutes les instances à l'optimalité dans un temps court. Pour le deuxième groupe (nommé U), les 100 instances sont composées de 1000 objets, et une taille de sac à dos allant de 500 à 520. Chaque objet a un profit compris entre 1 et 100. Nous montrons d'abord l'impact des raffinements sur notre algorithme afin de déterminer les meilleurs paramètres de la méthode, puis nous comparons les résultats obtenus aux méthodes de la littérature.

5.6.1 Impact des tests de dominance et réalisabilité

Dans la suite de ce manuscrit, on note *SSDP** la combinaison de techniques ayant donnée les meilleurs résultats (cette combinaison sera détaillée au cours de la section). Pour tester l'efficacité des paramètres vu dans la section précédente, nous comparons les résultats obtenus à la méthode SSDP* en désactivant chaque paramètre. Pour chaque méthode, nous reportons dans la figure 5.5 le nombre d'instances résolues par instant de temps, avec une limite de temps de 3 heures.

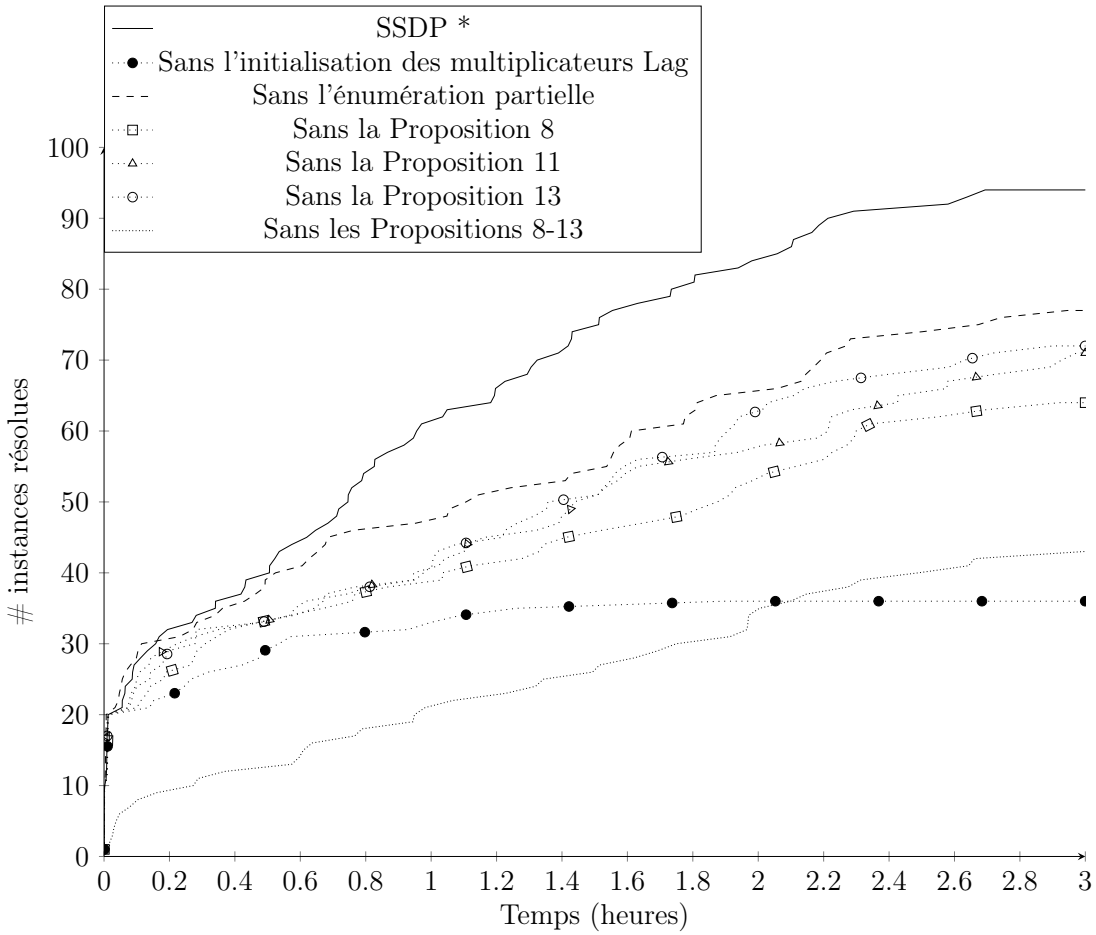


FIGURE 5.5 – Nombre d'instances de sac à dos temporel de type U ([CFMT16]) résolues par rapport au temps pour la meilleure version de notre algorithme et 6 versions obtenues en désactivant certaines techniques.

Nous nous intéressons d'abord à l'impact de l'énumération partielle. Le nombre d'instances résolues optimalement augmente de 20% quand on utilise l'énumeration partielle. Cela signifie que l'énumération de séquences de transitions permet d'obtenir des informations utiles pour l'algorithme de filtrage. Pour illustrer l'impact de l'énumération partielle, le tableau 5.1 montre la taille du premier réseau construit pour différentes valeurs de k^{enum}. Quand le nombre de transitions successives considérées augmente, le nombre de nœuds dans le réseau diminue et le nombre d'arcs augmente (des combinaisons d'arcs sont remplacées par des arcs simples). Bien que la consommation de mémoire et le temps de résolution vont être plus élevés, cela peut être avantageux dans une certaine mesure : sélectionner un arc signifie choisir plus de décisions simultanément et plus d'impact sur le coût Lagrangien de la solution. Ainsi, ces longues séquences de décisions sont plus facilement écartées par le filtrage Lagrangien. Cela explique également, avec la suppression de courtes séquences irréalisables, pourquoi le réseau avec $k^{enum} = 2$ a moins d'arcs que celui avec $k^{enum} = 1$.

k^{enum}	1	2	3	4	5	6
# noeuds moyen	703 k	384 k	264 k	202 k	162 k	135 k
# transitions moyen	1,392 k	1,344 k	1,639 k	2,269 k	3,155 k	4,658 k

TABLE 5.1 – Taille moyenne du premier réseau pour différentes valeurs de k^{enum}, après l'étape de filtrage. "k" représente des milliers.

Nous nous intéressons désormais à l'impact de l'initialisation des multiplicateurs de Lagrange.

Initialiser les multiplicateurs de Lagrange à la première itération en utilisant les valeurs duales du modèle (5.4)-(5.12) permet de résoudre plus de 2,5 fois plus d'instances qu'en initialisant les multiplicateurs à 0 (94 contre 36 sur 100 instances). Il y a deux explications possibles : soit l'algorithme de Volume utilisé converge plus rapidement grâce à l'initialisation, soit la borne duale obtenue après convergence est meilleure. Le tableau 5.2 contient le temps moyen de convergence de l'algorithme de Volume pour le premier réseau, le gap moyen avec la borne primale et le nombre moyen de sommets et arcs dans le réseau après convergence.

	Temps moyen (s.)	# nœuds moyen	# arcs moyen	gap moyen
SSDP* avec init	118	161 k	1,650 k	0.42%
SSDP* sans init	191	188 k	2,136 k	1.53%

TABLE 5.2 – Impact de l'initialisation des multiplicateurs de Lagrange sur la solution de la première relaxation et sur la taille du réseau obtenu après le premier filtrage. "k" signifie des milliers.

Les tests de réalisabilité proposés dans les Propositions 8 à 13 ont un impact crucial sur les performances de notre algorithme. L'ajout de ces tests permet de résoudre 40 instances supplémentaires en moins d'une heure, et 11 instances supplémentaires en moins de 3 heures. Ces tests permettent de retirer environ 20% des nœuds et 32% des arcs du premier réseau obtenu en utilisant $k^{enum} = 4$. On remarque que chacun de ces tests permet d'améliorer significativement les performances de l'algorithme.

5.6.2 Critères de choix de dimension

Nous avons vu dans la section 4.6 plusieurs critères de sélection. Nous présentons l'impact de ces critères sur les résultats obtenus. Pour cela, nous détaillons le calcul de chaque critère appliqué au problème de sac à dos temporel. A chaque étape de sublimation de SSDP, des dimensions correspondant à des contraintes violées sont intégrées dans l'espace d'états. Pour une itération donnée, notons π^q le vecteur de multiplicateurs de Lagrange qui permet d'obtenir la q^{me} meilleure borne duale pendant l'étape de résolution de la relaxation, et y^q la solution trouvée.

Le premier critère 1 sera noté χ^1 et correspond au vecteur de multiplicateurs de

Lagrange π permettant d'obtenir la meilleure solution duale à chaque itération.

$$\chi_i^1 = |\pi_i|, \forall i \in \mathbb{I} \text{ tq } r_i \in \mathcal{J}$$

Le critère 3, noté χ^2, est calculé comme décrit dans la section précédente. Enfin, le critère 4 correspondant au nombre de solutions où une contrainte est violée sera noté χ^3. Les solutions obtenues lors de l'algorithme de Volume sont conservées dans une liste $(y^1, ..., y^{k^{nbsol}})$, avec k^{nbsol} un paramètre de la méthode correspondant au nombre de solutions conservées à chaque itération (dans nos expérimentations, ce paramètre est fixé à $k^{nbsol} = 20$). Le critère χ^3 est donné par :

$$\chi_i^3 = \frac{1}{k^{nbsol}} \sum_{q=1}^{k^{nbsol}} \left| \boldsymbol{y}^q_{e^{in}(i)} - \boldsymbol{y}^q_{e^{out}(i)} \right|$$

Le détail des configurations utilisées pour les tests est présenté dans le tableau 5.3.

Configuration	Stratégie	Critère χ_i
SSDP*	Stable de cardinalité	$\chi_i^2 (1 - \chi_i^3)$
Stable	Stable pondéré	$\frac{-\chi_i^2 + \max_{j \in \mathcal{J} \neq} \chi_j^2}{\max_{j \in \mathcal{J} \neq} \chi_j^2} \chi_i^3$
NbNodes	Stable de cardinalité	χ_i^2
Adaptative	Stable de cardinalité	D'abord χ_i^3, puis χ_i^2
LagMult	Stable de cardinalité	χ_i^1
KColor	K-Coloration	$\frac{-\chi_i^2 + \max_{j \in \mathcal{J} \neq} \chi_j^2}{\max_{j \in \mathcal{J} \neq} \chi_j^2} \chi_i^3$

TABLE 5.3 – Paramètres des méthodes testées.

Des expérimentations préliminaires nous ont permis de fixer le paramètre $k^{nbsol} = 20$ pour le critère χ^3. La méthode a donné des performances moins bonnes pour une valeur de k^{nbsol} inférieure à 10, mais ne semble pas affectée par des valeurs supérieures à 20. Pour les méthodes utilisant la stratégie *stable de cardinalité*, la valeur du paramètre k^{stable} a été fixée à $0,7$. Le critère de sélection utilisé pour la méthode SSDP* tente de prendre en compte le nombre de sommets supplémentaires estimé et la fréquence de violation de la contrainte dans les meilleures solutions relâchées trouvées. Lors de l'utilisation de la stratégie *stable pondéré*, réduire le nombre attendu de sommets ajoutés revient à ne sélectionner aucune nouvelle contrainte. C'est pourquoi nous maximisons le complément au nombre maximum de sommets supplémentaires attendu. Cette valeur est pondérée par la fréquence de violation de la contrainte. Enfin, la configuration *Adaptative* tente d'améliorer la borne duale le plus rapidement possible en ajoutant les contraintes les plus souvent violées. Quand le réseau devient trop grand (la limite a été fixée à 4.000k sommets), on tente de contrôler la taille du réseau. La comparaison de ces différentes méthodes de sélection est montrée dans la figure 5.6.

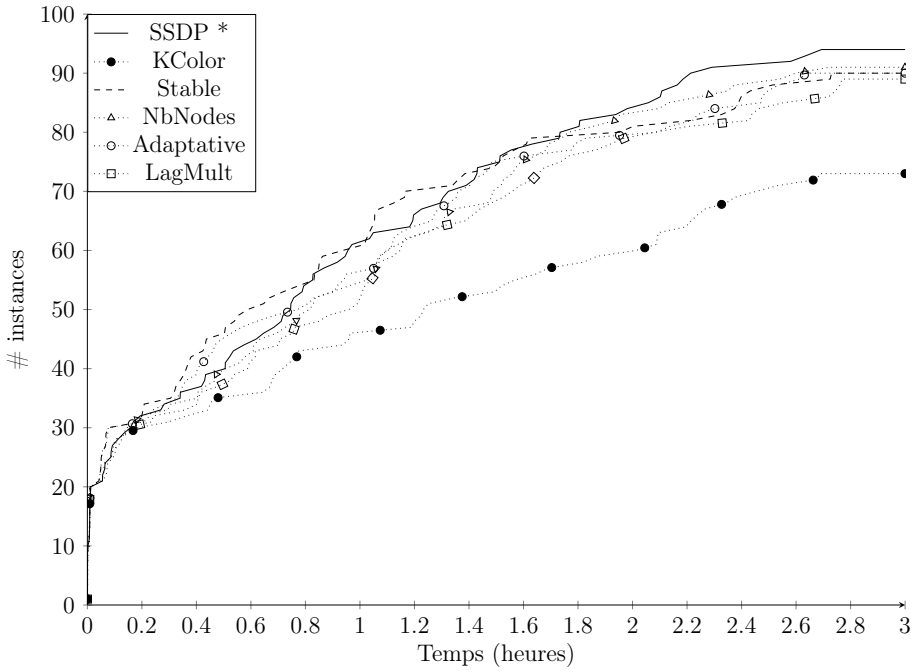


FIGURE 5.6 – Nombre d'instances résolues par rapport au temps pour les différentes méthodes utilisées pour déterminer les dimensions à ajouter à chaque itération.

La première remarque est que la stratégie de k-coloration n'est clairement pas compétitive comparée aux stratégies de stable sur ce problème. Le fait que cette stratégie fonctionne mal montre que notre méthode pour évaluer la taille du réseau utilisant les stratégies de stable est utile, et qu'on ne peut pas se fier uniquement à la structure d'intervalle des contraintes. Les configurations basées sur les stratégies de stable ont un comportement similaire. La configuration *stable* donne de bons résultats avec une limite de temps moyenne, mais ses performances se dégradent pour un temps limite plus grand. Ceci peut s'expliquer par le fait que plus la taille du réseaux augmente, moins il y a de nouvelles contraintes ajoutées. Quand une taille critique est atteinte, nous avons remarqué que seules quelques contraintes sont ajoutées en général. Les méthodes basées sur les stables ne souffrent pas de ce phénomène. En effet, pour les instances de type U composées de 1000 objets, les configurations *SSDP** et *Stable* réintroduisent entre 5 et 20 contraintes dans plus de 75% des itérations, alors que le nombre maximum de contraintes ajoutées par l'algorithme à chaque itération est respectivement de 29 et 41. Le nombre total de contraintes ajoutées est inférieur ou égal à 15 pour 20% des instances, supérieur à 274 (resp. 267) pour 20% des instances pour la configuration *SSDP** (resp. *Stable*). Le nombre maximum de contraintes ajoutées pour une seule instance est de 328 (resp. 348).

5.6.3 Comparaison avec la littérature

Nous comparons notre méthode avec l'algorithme de [GI17]. Ils ont implémenté un *branch-and-price* sans heuristique primale et en utilisant comme stratégie de sélection de nœuds *best first*. Les expérimentations ont été réalisées sur un Intel(R) Core(TM) i7-2600 3,4 GHz avec 16 Go de mémoire en utilisant un thread. Pour pouvoir comparer les résultats obtenus par [GI17] et ceux de notre algorithme, nous avons utilisé une machine similaire (avec le même processeur et la même quantité de mémoire) en utilisant un seul thread. Les résultats sont donnés dans la figure 5.7.

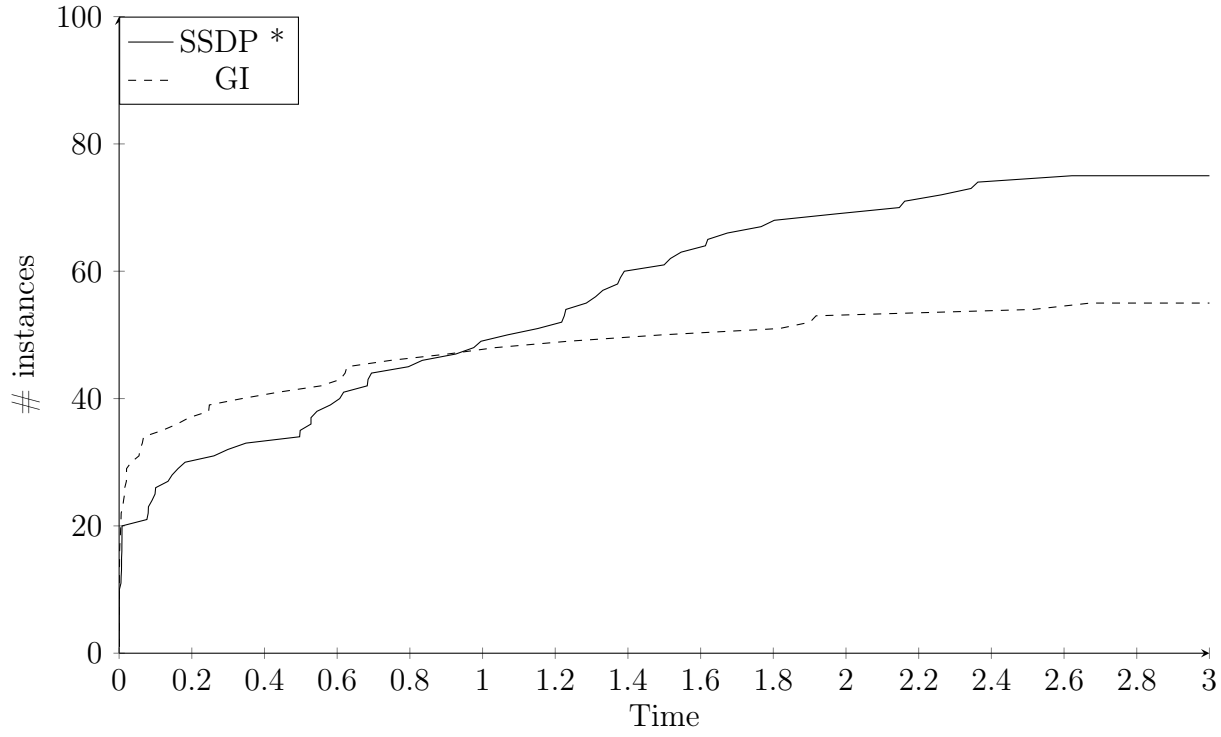


FIGURE 5.7 – Nombre d'instances résolues par rapport au temps de notre meilleure implémentation (SSDP*) et de l'algorithme de [GI17] sur un Intel(R) Core(TM) i7-2600 3,4 GHz avec 16 Go de mémoire en utilisant un thread.

Les résultats montrent que la méthode de [GI17] est plus efficace quand on fixe un temps de calcul court. En 30 minutes, notre méthode permet de résoudre à l'optimalité 35 instances contre 41 pour [GI17]. Pour 1 heure de temps de calcul, les deux méthodes ont des résultats similaires (47 et 49 instances résolues). Quand on étend le temps de résolution à 3 heures, notre méthode résout à l'optimalité 50% des instances non résolues en 1 heure, alors que la méthode de branch-and-price ne résout que quelques instances suplémentaires (55 instances pour le branch-and-price, 75 instances pour notre méthode).

5.6.4 Comparaison avec le solveur CPLEX

Nous comparons les résultats obtenus par notre méthode et les résultats obtenus en utilisant le solveur de programmation linéaire IBM Ilog CPLEX sur le modèle (5.1)-(5.3). La figure 5.8 montre les résultats obtenus en utilisant un Intel Xeon E5-2680 v3 2,5 GHz 2 Dodeca-core Haswell avec 128Go RAM en utilisant 1 thread et 6 threads.

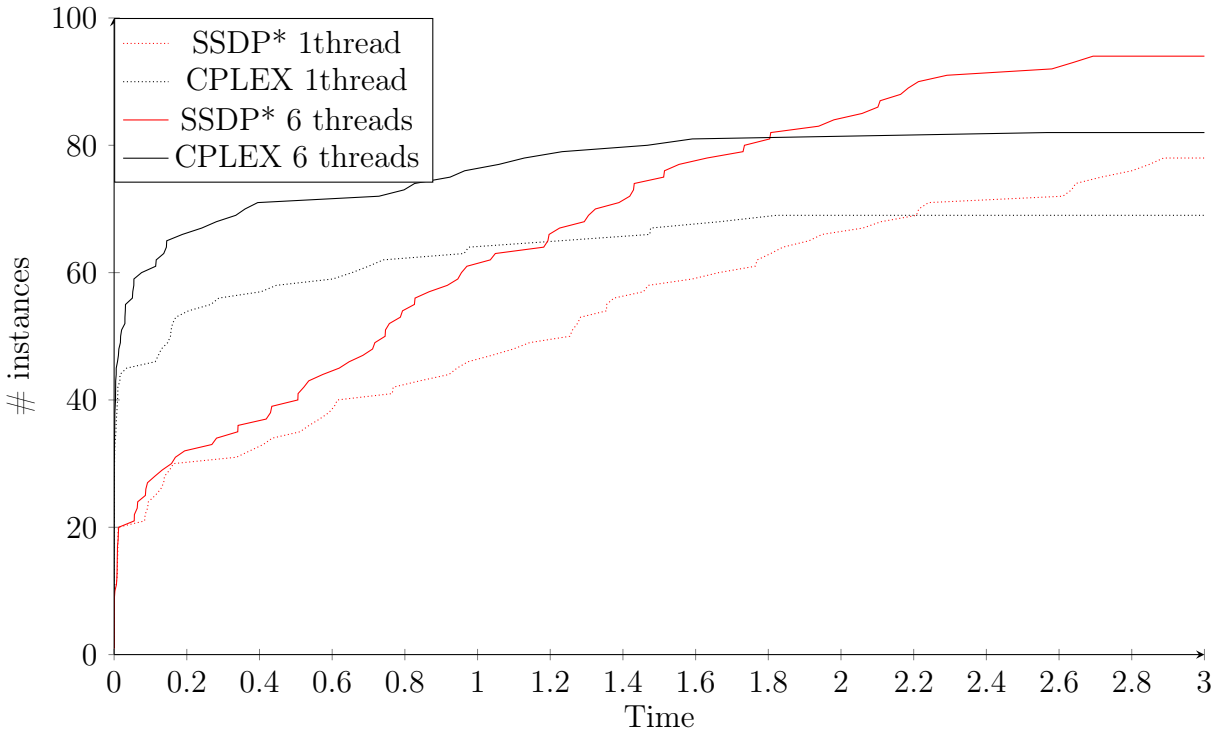


FIGURE 5.8 – Nombre d'instances résolues par rapport au temps pour notre méthode (SSDP*) et le solveur CPLEX sur le modèle (5.1)-(5.3) (CPLEX).

Notons que seulement 5 instances n'ont pas été résolues par les deux méthodes. Notre méthode SSDP* permet de résoudre plus d'instances en 3 heures.

Quand on utilise un thread, SSDP* permet de résoudre à l'optimalité 78 instances contre 69 pour CPLEX. La résolution en utilisant CPLEX est beaucoup plus rapide sur de nombreuses instances, notamment pour les instances numérotées entre 1 et 55. Pour la plupart d'entre elles, le solveur CPLEX est capable de les résoudre en quelques secondes, alors qu'il faut plusieurs minutes (voire heures) pour les autres méthodes. Cela peut être expliqué par les procédures puissantes intégrées dans ces solveurs sur les contraintes de sac à dos (par exemple en utilisant des coupes), et par de très bonnes heuristiques génériques. Pour les instances numérotées de 55 à 99, CPLEX permet de résoudre 16 instances à l'optimalité. La structure des instances permet d'expliquer ce phénomène : chaque série de 10 instances consécutives a une structure similaire, notamment le nombre maximum d'objet dans une clique. Ce nombre augmente avec le numéro de l'instance. Il

ressort de ces expérimentations que les méthodes basées sur la programmation linéaire sont très sensibles à ce paramètre.

L'utilisation de plusieurs threads permet d'améliorer les résultats des deux méthodes. Après 3 heures, SSDP* permet de résoudre 94 instances à l'optimalité contre 82 pour CPLEX. Le temps nécessaire à la construction du graphe et à la sublimation représente un large pourcentage du temps total. Seule la résolution du problème Lagrangien est parallélisée pour SSDP*, ce qui explique pourquoi SSDP* n'est pas 6 fois plus rapide en utilisant 6 threads. Notons que seulement 2 instances sont résolues optimalement par CPLEX (U69 et U78), contre 14 pour SSDP*.

5.6.5 Sensibilité de la méthode à la capacité du sac à dos

La taille de notre programme dynamique dépend de la capacité du sac à dos W. Lorsque la capacité augmente, le temps et la mémoire nécessaires pour créer le graphe correspondant augmentent également. A l'inverse, les solveurs d'optimisation linéaire sont généralement moins sensibles à la valeur de ce paramètre.

Nous avons effectué des expérimentations sur des instances avec une capacité de sac à dos plus grande. Pour créer ces instances, nous avons implémenté le générateur d'instance de [CFM13] qui a permis d'obtenir les instances vues précédemment dans cette section. Les auteurs ont généré 20 classes d'instances différentes ($I1$ à $I10$ et $U1$ à $U10$). Nous avons généré 4 nouvelles instances par classe : deux avec $W = 1000$ et deux avec $W = 10000$. Ces instances créées ont la même structure que les instances générées par [CFM13] mais avec une capacité de sac à dos plus grande.

Pour ces expérimentations, la proposition 11 n'a pas été utilisée pour notre implémentation de SSDP, car la mémoire requise était trop importante. Le tableau 5.4 montre le résultat de ces expérimentations. Pour chaque type d'instance (I et U), et chaque capacité de sac à dos (1000 et 10000), nous indiquons le nombre d'instances (sur les 20 générées) qui ont été résolues par CPLEX seulement, par SSDP seulement, et par les deux méthodes avec un temps limite de 3 heures.

TABLE 5.4 – Sensibilité de notre méthode sur la taille du sac à dos.

Type	Capacité	#Status des instances (sur 20)			
		seulement CPLEX	seulement SSDP	résolues par les deux	non résolues
I	1000	0	12	8	0
U	1000	5	0	7	8
I	10000	0	11	9	0
U	10000	14	0	0	6

Les résultats montrent que la méthode SSDP est moins performante quand la taille du sac à dos augmente et qu'on désactive la proposition 11. Néanmoins, les performances obtenues restent compétitives avec une taille de sac à dos plus grande (47 instances résolues contre 43 pour CPLEX). Pour les instances de type I, les deux méthodes permettent de résoudre à l'optimalité toutes les instances avec $W = 100$. En augmentant la capacité

du sac à dos, CPLEX n'a pu résoudre que 8 instances sur les 20 générées avec $W = 1000$, et 9 instances avec $W = 10000$, alors que SSDP parvient à toutes les résoudre à l'optimalité. CPLEX est capable de trouver rapidement une bonne solution, mais ne parvient pas à fermer le gap après 3 heures de calcul. SSDP est capable de résoudre toutes les grandes instances de type I, bien qu'il nécessite un temps de calcul plus important (respectivement 284s et 596s en moyenne pour $W = 1000$ et $W = 10000$). A l'inverse, CPLEX est plus efficace pour les instances de type U. Pour $W = 10000$, le solveur est capable de résoudre 14 instances, tandis que SSDP n'est capable d'en résoudre aucune, car la mémoire nécessaire pour stocker le graphe correspondant au premier programme dynamique est trop grand.

5.7 Conclusion

Nous avons montré que la méthode SSDP permet d'obtenir des résultats compétitifs sur le problème de sac à dos temporel. Notre implémentation a permis de résoudre optimalement 94 des 100 instances de la littérature en fixant un temps limite de 3 heures. Pour permettre à la méthode d'être efficace, différents tests de réalisabilité ainsi que des techniques d'ajout de dimensions ont été implémentées. L'analyse des résultats a permis de montrer l'impact des différentes techniques proposées. Les travaux nécessaires pour développer et implémenter ces techniques expliquent pourquoi ces méthodes sont peu utilisées dans la littérature.

Chapitre 6

Application de SSDP à des problèmes de planification

6.1 Introduction

Nous avons vu dans le chapitre précédent que la méthode SSDP permettait d'obtenir des résultats compétitifs sur le problème de sac à dos temporel. Les techniques génériques décrites précédemment ainsi que des techniques propres à ce problème ont permis à la méthode d'être efficace. Les résultats obtenus nous amènent à tester la méthode sur d'autres problèmes.

Dans cette section, nous nous intéressons à deux applications de la méthode SSDP. La première nous permet de comparer notre approche à une des rares applications de la méthode dans la littérature [TFA09] sur un problème d'ordonnancement à une machine. Les différents raffinements de la méthode sont étudiés et exprimés avec notre formalisme et comparés avec les résultats obtenus dans [TFA09]. La deuxième correspond à un problème industriel complexe de traitements phytosanitaires des vignes. Une modélisation est proposée et nous montrons le programme dynamique correspondant utilisé pour appliquer nos méthodes itératives.

6.2 Ordonnancement

Les problèmes d'ordonnancement sont des problèmes classiques de la littérature. La méthode SSDP a rapidement été appliquée sur ce type de problème par [IN94] puis repris par [TFA09]. Le but de cette section est de comparer notre implémentation générique de méthodes itératives à un algorithme spécifiquement dédié à ce problème.

6.2.1 Description du problème

On considère un problème d'ordonnancement sur une machine sans préemption. On cherche à minimiser la somme des retards pondérés. La notation classique en ordonnancement est $1||w_iT_i$. La figure 6.1 représente un exemple d'instance de ce problème.

Problème 2 ($1||w_iT_i$)**.** *Soit $J = 1, ..., n$ un ensemble de jobs à ordonnancer. Chaque job $j \in J$ a un poids $w_j \in \mathbb{N}$, une date échue $d_j \in \mathbb{N}$ et une durée $p_j \in \mathbb{N}$. Une solution réalisable est une séquence de n jobs, où chaque job de J a été ordonnancé exactement une fois. On note C_j la date fin du job j dans une séquence. Le problème consiste à trouver un ordonnancement des n jobs qui minimise la somme des retards pondérés par le poids des jobs.*

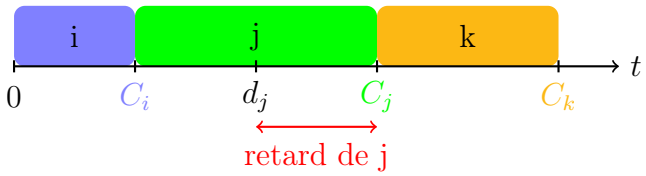


FIGURE 6.1 – Une instance de $1||w_iT_i$ avec 3 jobs

Le problème est NP-difficile au sens fort ([Law77], [LKB77]). Les premières approches étaient basées sur la programmation dynamique et le branch-and-bound. Ces méthodes sont résumés dans [ARPVW90]. Il existe plusieurs règles de dominance qui permettent de discriminer certaines solutions. Une partie d'entre elles ont été établies par [Emm69] et [Elm68].

De bonnes bornes inférieures ont été données dans la littérature, notamment par [ARP88] grâce à une méthode basée sur la relaxation de l'espace d'états. Des heuristiques ont aussi été développées, on citera notamment [CPvdV02] qui introduit une méthode basée sur la programmation dynamique dans un algorithme de recherche locale.

Parmi les travaux récents, l'article de [TFA09] attire plus particulièrement notre attention. Les auteurs ont repris les travaux de [IN94] qui ont appliqué la méthode SSDP au problème de minimisation de la somme pondérée des retards. L'apport de [TFA09] a permis de montrer que cette méthode peut être compétitive. Pour obtenir ces résultats, les auteurs ont implémenté un algorithme dédié à ce problème. Le but ici est de comparer leurs résultats avec notre implémentation générique, et de pouvoir appliquer d'autres méthodes d'agrégation sur ce problème.

6.2.2 Modélisation et programme dynamique

La modélisation de ce problème sous forme de programme linéaire en nombres entiers a été proposée dans de nombreux articles de la littérature (par exemple [PWW69] ou [SW92]).

Nous reprenons la formulation utilisée par [TFA09]. Dans ce modèle, chaque variable de décision binaire x_{jt}, $j \in J$, $1 \leq t \leq T$ indique si le job j finit en t (ie $C_j = t$).

Soit $f_j(t) = \max\{0, (t - d_j)w_j\}$ le coût de terminer le job j en t.

$$\min_{x} \quad \sum_{i=1}^{n}\sum_{t=1}^{T} f_i(t)x_{it} \tag{6.1}$$

$$\sum_{i=1}^{n}\sum_{s=t}^{\min\{T,t+p_i-1\}} x_{is} \quad = \quad 1, \quad 1 \leq t \leq T \tag{6.2}$$

$$\sum_{t=p_i}^{T} x_{it} \quad = \quad 1, \quad 1 \leq j \leq n \tag{6.3}$$

$$x_{it} \quad \in \quad \{0,1\} \quad 1 \leq j \leq n, 1 \leq t \leq T \tag{6.4}$$

La contrainte (6.2) assure qu'il n'y a qu'un seul job à chaque instant de temps, et la contrainte (6.3) assure que tous les jobs sont ordonnancés exactement une fois. Notons que T correspond au temps de fin de l'ordonnancement, et correspond à $T = \sum_{j \in J} p_j$. Les contraintes (6.3) correspondent aux contraintes linéaires qui vont être relâchées (ensemble $\mathcal{K}$). En relâchant ces contraintes, un job peut être ordonnancé plusieurs fois ou ne pas être ordonnancé.

Dans la littérature, l'article de [HK62] propose un programme dynamique pour ce problème, repris par la suite par [ARP88].

On définit un état comme un tuple $(t, \mathbf{d})$ où t est le temps courant et $\mathbf{d} \in \{0,1\}^n$ le vecteur caractéristique de l'ensemble des jobs ordonnancés au temps t.

On définit ensuite une transition comme un tuple $(j, \Delta_t, \Delta_d, p)$ où $j \in J$ le job correspondant à la transition, $\Delta_t \in \mathbb{N}$ l'augmentation du temps quand on ordonnance le job, $\Delta_d \in \{0,1\}^n$ le vecteur qui met à jour les jobs déjà ordonnancés et $p \in \mathbb{N}$ le retard pondéré obtenu.

La fonction ψ qui associe à chaque état l'ensemble des transitions applicables est définie par :

$$\psi((t, \mathbf{d})) = \{(j, p_j, \boldsymbol{\varepsilon}_j, f_j(t + p_j) : j \in J \setminus \{i\} \text{ tq } \mathbf{d}_j = 0\}$$

La fonction de coût récursive est exprimée de la façon suivante :

$\underline{DP_1}$

$$\alpha((t, \mathbf{d})) = \begin{cases} \max\limits_{(j,\Delta_t,\Delta_d,p)\in\psi((t,\mathbf{d}))} \{p + \alpha((t + \Delta_t, \mathbf{d} + \Delta_d))\} & \text{si } t \leq T \\ 0 & \text{sinon} \end{cases}$$

6.2.3 Raffinement de SSDP

Nous avons vu dans les sections précédentes la nécessité de supprimer des arcs et des sommets dans les graphes successifs. Les auteurs de [IN94] ont introduit plusieurs raffinements, qui ont été repris et étoffés par [TFA09]. Dans cette section, nous décrivons l'ensemble de ces méthodes permettant d'améliorer les performances de SSDP sur ce problème.

Borne primale initiale.

Plusieurs heurisitiques permettent de trouver une borne primale à ce problème. Les auteurs de [IN94] ont utilisé comme borne primale le meilleur ordonnancement trouvé entre un algorithme glouton où le job qui impacte le moins le profit est choisi et une heuristique basée sur l'ordre EDD ("Earliest Due Date", [BB81]) où les jobs sont ordonnancés selon leur date échue. Les auteurs de [TFA09] ont amélioré cette borne en utilisant une technique similaire : on sélectionne le meilleur ordonnancement obtenu a partir de trois heuristiques (deux algorithmes gloutons en "forward" et "backward", et une heuristique utilisant la règle SPT("shortest processing time order") qui ordonnance les jobs selon leur durée). Ils appliquent ensuite sur la meilleure séquence trouvée un algorithme de "Dynasearch" ([CPvdV02], [GDCT04]). Cette technique est basée sur le voisinage de la solution. Pour une séquence de jobs donnée, l'algorithme de dynasearch cherche à améliorer la solution en échangeant la position de deux jobs (successifs ou non) ou en retirant un job de la séquence et en l'insérant au début ou à la fin de la séquence.

Dominance sur deux jobs successifs.

Les auteurs de [TFA09] ont proposé une règle de dominance permettant de supprimer des séquences non optimales en considérant deux jobs successifs. On note $j \to i(t)$ pour une solution si le job i se termine en t ($C_i = t$) et j est ordonnancé juste avant i ($C_j = t - p_i$). On définit $\Xi_{j\to i}(t)$ la somme des coûts des jobs i et j quand $j \to i(t)$, avec $\Xi_{j\to i}(t) = f_j(t - p_i) + f_i(t)$.

Une séquence de jobs contenant $j \to i(t)$ ne peut être optimale si

$$\Xi_{i\to j}(t) < \Xi_{j\to i}(t)$$

On peut facilement trouver une séquence de meilleur coût en inversant i et j. Cette règle permet de définir la proposition suivante :

Proposition 14. *Dominance de jobs successifs [TFA09] Il existe au moins une séquence de jobs optimale telle que deux jobs successifs i et j ($j \to i(t)$) satisfont*

$$\Xi_{j\to i}(t) < \Xi_{i\to j}(t) \tag{6.5}$$

ou

$$\Xi_{j\to i}(t) = \Xi_{i\to j}(t), \quad j = R_{ij} \tag{6.6}$$

où R_{ij} définit une relation d'ordre arbitraire entre i et j. R_{ij} permet de casser les symétries quand deux jobs peuvent être échangés sans changer le coût de la séquence.

A partir de cette proposition, on peut définir l'ensemble des jobs pouvant être ordonnancés successivement. Soit $\mathcal{P}_i(t)(i \in J, p_i + 1 \leq t \leq T)$ l'ensemble des jobs $j \neq i$ qui satisfont $p_i + p_j \leq t$ et (6.5) ou (6.6).

Pour renforcer les relaxations successives de SSDP, les auteurs de [TFA09] ont ajouté la contrainte suivante au problème :

Un job i terminant en t doit être précédé directement par un job $j \in \mathcal{P}_i(t)$ (6.7)

Pour prendre en compte cette contrainte dans le programme dynamique, une dimension correspondant au dernier job ordonnancé est ajouté à l'espace d'états. On définit un état du nouveau programme dynamique comme un tuple $(t, j, \mathbf{d})$, avec t et $\mathbf{d}$ définis comme dans le programme dynamique précédent $DP1$, et $j \in J$ le dernier job ordonnancé. Les transitions sont les mêmes que dans le programme dynamique défini précédemment.

L'ensemble des transitions ψ applicables depuis un état est donné par :

$$\psi((t, j, \mathbf{d})) = \{(k, p_k, \boldsymbol{\varepsilon}_k, f_k(t + p_k) | k \in J \setminus \{j\} \text{ tel que } j \in \mathcal{P}_k(t + p_k) \text{ et } \mathbf{d}_k = 0\}$$

On définit ainsi un programme dynamique intégrant les contraintes de deux jobs successifs :

$\underline{DP_2}$

$$\alpha((t, j, \mathbf{d})) = \begin{cases} \max\limits_{(i,\Delta_t,\Delta_d,p)\in\psi((t,j,\mathbf{d}))} \{p + \alpha((t + \Delta_t, i, \mathbf{d} + \Delta_d))\} & \text{si } t \leq T \\ 0 & \text{sinon} \end{cases}$$

Choix des dimensions à ajouter

L'objectif principal des auteurs est de maitriser la taille du graphe à chaque itération. Le choix des dimensions dépend de la mémoire utilisée. Soit M le ratio d'occupation de la mémoire (avec M =(mémoire utilisée)/(taille de la mémoire)). Le nombre m_c de dimensions à ajouter à chaque itération est déterminé par :

$$m_c = \begin{cases} 1 & \text{si } 2^{-2} < M \\ 2 & \text{si } 2^{-3} < M \leq 2^{-2} \\ 3 & \text{si } M \leq 2^{-3} \end{cases}$$

Les auteurs ont ensuite défini deux critères de sélection dépendant de la violation et du nombre d'occurences de chaque job.

Les violations des contraintes relachées dans une solution correspondent soit à un job qui n'a pas été ordonnancé, soit un job qui a été ordonnancé plusieurs fois. Le premier critère est utilisé quand le ratio $M < 2^{-6}$:

Critère 6. *Les jobs qui ne sont pas ordonnancés dans la solution de la relaxation sont sélectionnés. S'il y en a plus que m_c, alors ceux qui apparaissent le moins dans le graphe correspondant à la relaxation sont sélectionnés. S'il y en a moins que m_c, on sélectionne ceux qui apparaissent le moins dans le graphe parmi ceux qui sont ordonnancés plusieurs fois.*

Le nombre d'occurences de chaque job dans le graphe de la relaxation peut être calculé en un parcours de graphe. Chaque transition $(j, \Delta_t, \Delta_d, p)$ compte pour une occurence du job j.

Quand le ratio d'occupation de la mémoire devient trop important ($M \geq 2^{-6}$), un deuxième critère est utilisé :

Critère 7. *Les jobs sont sélectionnés par nombre d'occurences croissant dans le graphe correspond à la relaxation.*

Le nombre d'occurences d'un job se rapproche du critère 3 présenté dans le chapitre précédent. L'estimation de la taille du réseau en utilisant notre agrégateur peut être vu comme une extension du nombre d'occurences.

Two cycle L'ajout de la contrainte (6.7) permet de supprimer les solutions où deux jobs successifs sont identiques. La section 4.3 montre comment supprimer des arcs dans le graphe correspondant à une relaxation en utilisant le filtrage Lagrangien. Ce filtrage peut être amélioré quand la dimension correspondant à la contrainte (6.7) est ajoutée à l'espace d'état. L'idée est de vérifier le coût d'un arc sous la contrainte qu'un job n'est pas ordonnancé plusieurs fois dans une séquence de trois arcs.

Pour faciliter la compréhension, nous allons illustrer cette technique de filtrage en utilisant pour l'ensemble $\mathcal{J}$ l'ensemble contenant uniquement la contrainte (6.7), mais la technique reste valable pour des ensembles de contraintes plus grands. De plus, on notera $v_{t,i} \in G_\pi^{\mathcal{J}}$ le sommet correspondant à l'empreinte $\boldsymbol{e} = (t, i, d)$. Pour un arc $a = (v; v') \in E^{\mathcal{J}}$, on note $\mathbf{t}(v) = (i, \Delta_t, \Delta_d, p)$ la transition associée à l'arc et $f(a) = f((v; v')) = p+ < \pi, \Delta_d >$ le coût de l'arc a dans le graphe $G_\pi^{\mathcal{J}}$.

Nous avons vu que le coût maximal d'un chemin vers chaque sommet de $G_\pi^{\mathcal{J}}$ peut être calculé avec l'algorithme de Bellman. Le coût de ce chemin vers un sommet $v_{t,i}$ est donné par :

$$\hat{\gamma}_{\mathcal{J}}^{\pi}(v_{t,i}) = \max_{(v_{t-p_i,j}; v_{t,i}) \in E^{\mathcal{J}}} f((v_{t-p_i,j}; v_{t,i})) + \hat{\gamma}_{\mathcal{J}}^{\pi}(v_{t-p_i;j})$$

On introduit l'élément $\lambda_{\mathcal{J}}^{\pi}(v_{t,i})$ correspondant au job séquencé juste avant l'arc permettant d'obtenir le meilleur coût $\hat{\gamma}_{\mathcal{J}}^{\pi}(v_{t,i})$, ie :

$$\lambda_{\mathcal{J}}^{\pi}(v_{t,i}) = \arg \max_{\substack{j \\ (v_{t-p_i,j}; v_{t,i}) \in E^{\mathcal{J}}}} f((v_{t-p_i,j}; v_{t,i})) + \hat{\gamma}_{\mathcal{J}}^{\pi}(v_{t-p_i;j})$$

Le coût maximal du chemin de la source vers un sommet $v_{t,i}$ n'ayant pas le job $\lambda(v_{t,i})$ ordonnancé juste avant est noté $\hat{\gamma}_{\mathcal{J},2}^{\pi}(v_{t,i})$ et est donné par :

$$\hat{\gamma}_{\mathcal{J},2}^{\pi}(v_{t,i}) = \max_{\substack{(v_{t-p_i,j}; v_{t,i}) \in E^{\mathcal{J}} \\ j \neq \lambda(v_{t,i})}} f((v_{t-p_i,j}; v_{t,i})) + \hat{\gamma}_{\mathcal{J}}^{\pi}(v_{t-p_i;j})$$

Ces différentes valeurs permettent d'obtenir le coût $h_{\mathcal{J}}^{\pi}((v_{t-p_i,j}; v_{t,i}))$ depuis la source vers $v_{t,i}$ passant par l'arc $(v_{t-p_i,j}; v_{t,i})$, en respectant la contrainte de non duplication de job sur 3 arcs successifs.

$$h_{\mathcal{J}}^{\pi}((v_{t-p_i,j}; v_{t,i})) = \begin{cases} \hat{\gamma}_{\mathcal{J}}^{\pi}(v_{t,i}) + f((v_{t-p_i,j}; v_{t,i})) & \text{si } i \neq \lambda_{\mathcal{J}}^{\pi}(v_{t,i}) \\ \hat{\gamma}_{\mathcal{J},2}^{\pi}(v_{t,i}) + f((v_{t-p_i,j}; v_{t,i})) & \text{si } i = \lambda_{\mathcal{J}}^{\pi}(v_{t,i}) \end{cases}$$

De cette manière, on peut obtenir les coûts backward du sommet puits vers un sommet $v_{t,i}$ respectant la contrainte de non duplication de job sur 3 arcs successifs. Le coût du chemin maximal entre $v_{t,i}$ et le sommet puits dans $G_\pi^{\mathcal{J}}$ est :

$$\hat{\alpha}^{\pi}_{\mathcal{J}}(v_{t,i}) = \max_{(v_{t,i};v_{t+p_j,j})\in E^{\mathcal{J}}} f((v_{t,i};v_{t+p_j,j})) + \hat{\alpha}^{\pi}_{\mathcal{J}}(v_{t+p_j;j})$$

Le job $\Lambda^{\pi}_{\mathcal{J}}(v_{t,i})$ correspond au job ordonnancé par l'arc donnant le coût maximal.

$$\Lambda^{\pi}_{\mathcal{J}}(v_{t,i}) = \arg \max_{\substack{j \\ (v_{t,i};v_{t+p_j,j})\in E^{\mathcal{J}}}} f((v_{t,i};v_{t+p_j,j})) + \hat{\alpha}^{\pi}_{\mathcal{J}}(v_{t+p_j;j})$$

Le coût maximal du chemin de $v_{t,i}$ vers le puits n'ayant pas le job $\Lambda(v_{t,i})$ ordonnancé juste après est noté $\hat{\alpha}^{\pi}_{\mathcal{J},2}(v_{t,i})$ et est donné par :

$$\hat{\alpha}^{\pi}_{\mathcal{J},2}(v_{t,i}) = \max_{\substack{(v_{t,i};v_{t+p_j,j})\in E^{\mathcal{J}} \\ j\neq\hat{\Lambda}^{\pi}_{\mathcal{J}}(v_{t,i})}} f((v_{t,i};v_{t+p_j,j})) + \hat{\alpha}^{\pi}_{\mathcal{J}}(v_{t+p_j;j})$$

On obtient le coût $H^{\pi}_{\mathcal{J}}((v_{t,i},v_{t+p_j,j}))$ de $v_{t,i}$ au puits en passant par l'arc $(v_{t,i};v_{t+p_j,j})$ qui est égal à :

$$H^{\pi}_{\mathcal{J}}((v_{t,i};v_{t+p_j,j})) = \begin{cases} \hat{\alpha}^{\pi}_{\mathcal{J}}(v_{t+p_j,j}) + f((v_{t,i};v_{t+p_j,j})) & \text{si } i \neq \lambda^{\pi}_{\mathcal{J}}(v_{t+p_j,j}) \\ \hat{\alpha}^{\pi}_{\mathcal{J},2}(v_{t+p_j,j}) + f((v_{t,i};v_{t+p_j,j})) & \text{si } i = \lambda^{\pi}_{\mathcal{J}}(v_{t+p_j,j}) \end{cases}$$

Un arc $(v_{t-p_i,j};v_{t,i})$ ne peut pas appartenir à une solution optimale et peut être retiré de $G^{\mathcal{J}}_{\pi}$ si :

$$h^{\pi}_{\mathcal{J}}((v_{t-p_i,j};v_{t,i})) + H^{\pi}_{\mathcal{J}}((v_{t-p_i,j};v_{t,i})) - f((v_{t-p_i,j};v_{t,i})) < LB$$

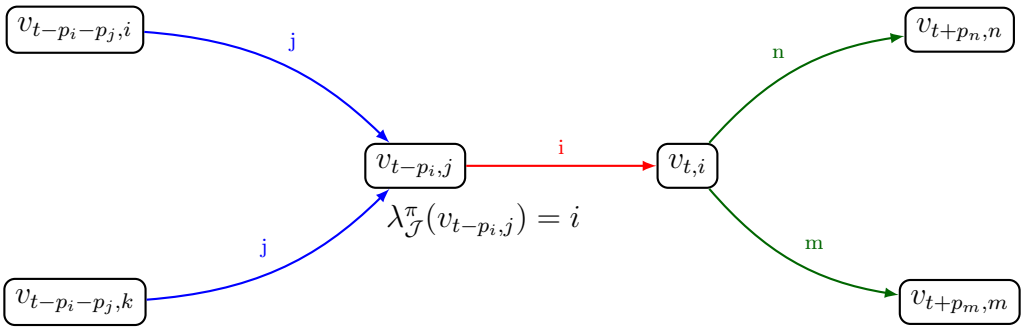


FIGURE 6.2 – Illustration du filtrage *Two cycle*. On suppose que $\hat{\gamma}^{\pi}_{\mathcal{J}}(v_{t-p_i-p_j,k}) < \hat{\gamma}^{\pi}_{\mathcal{J}}(v_{t-p_i-p_j,i})$. Le coût maximal $h^{\pi}_{\mathcal{J}}((v_{t-p_i,j};v_{t,i}))$ du chemin entre la source et le sommet $v_{t,i}$ en passant par l'arc $(v_{t-p_i,j};v_{t,i})$ ne peut pas être calculé à partir du coût maximal du chemin entre la source et le sommet $v_{t-p_i,j}$ car le job i a déjà été ordonnancé dans ce chemin la $(\lambda^{\pi}_{\mathcal{J}}(v_{t-p_i,j}) = i)$.

Sous gradient

A chaque itération, les multiplicateurs de Lagrange associés à la relaxation courante sont calculés en approximant le problème dual Lagrangien. La procédure de sous-gradient (cf section 2.2.2) proposée par [TFA09], adaptée de [Fis85], donne un paramétrage permettant d'obtenir une bonne borne en évitant de converger trop rapidement si la borne n'a pas été améliorée. Les principaux apports sont un pas spécifique pour le problème et un critère d'arrêt adapté. Notons que les multiplicateurs de Lagrange sont initialisés à 0.

Procédure globale

Nous détaillons la procédure globale utilisée par [TFA09]. La première relaxation est construite à partir du modèle 6.1 en relâchant les contraintes (6.3). Le programme dynamique obtenu correspond au programme dynamique 6.2.2 où le vecteur $\mathbf{d}$ est relâché. La contrainte (6.7) est ajouté à l'espace d'état, et le programme dynamique obtenu correspond à 6.2.3 en relâchant le vecteur $\mathbf{d}$. Le graphe est filtré en utilisant le filtrage Lagrangien 4.3 et le *two cycle* 6.2.3.

6.2.4 Résultats expérimentaux

Nous comparons les résultats obtenus dans la figure 6.1 avec l'implémentation de [TFA09]. Les tests ont été réalisés sur un Intel Xeon E5-2680 v3 2,5 GHz 2 Dodeca-core Haswell avec 128Go RAM en utilisant 1 thread. Le code de [TFA09] a été implémenté en *C*, et le notre en *C++*. On note *MCF* notre implémentation de la méthode en utilisant l'agrégateur 4.4 et les améliorations proposées dans la section 6.2.3, et *TFA* l'implémentation de [TFA09].

Instances	TFA (moyenne)	MCF (moyenne)	# instances résolues (MCF)
wt040	0.07	1.85	125/125
wt050	0.12	4.19	125/125
wt100	1.01	51.87	125/125
wt150	3.60	225.64	123/125
wt200	9.64	539.92	116/125
wt250	21.89	1114.51	101/125

TABLE 6.1 – Comparaison des temps de calcul moyen entre notre implémentation et celle de [TFA09] sur plusieurs instances. Les instances de classes *wtX* correspondent à des instances contenant X jobs. Pour MCF, les temps moyens correspondent aux temps moyens sur les instances résolues.

Les temps de calculs utilisant l'implémentation spécifique de [TFA09] sont meilleurs que ceux obtenus avec notre implémentation générique. Ces résultats peuvent être expliqués par plusieurs facteurs. Il existe encore une différence entre les deux implémentations. Le filtrage utilisé par [TFA09] compare la valeur obtenue pour un arc à $UB - 1$. Comme les coûts sont entiers, si le filtrage permet de retirer l'intégralité des arcs, alors la borne UB correspond à la solution optimale. Notre implémentation utilise la valeur UB.

Le temps nécessaire pour obtenir la première relaxation dans notre implémentation est plus long que le temps global de [TFA09] sur les grandes instances alors que le temps de calcul du premier réseau (avant l'optimisation des multiplicateurs de Lagrange) est court (moins de 10 secondes pour toutes les instances). Ces résultats préliminaires montrent que les performances de notre bibliothèque logicielle peuvent être améliorées.

6.3 Traitements phytosanitaires sur des vignes

Dans cette section, on s'intéresse à un problème industriel de traitements phytosanitaires de vignes. On dispose d'un ensemble de vignes que l'on souhaite traiter contre certaines maladies. Dans le problème original, l'ensemble des vignes est divisé en parcelles. Le problème que l'on va définir correspond au sous-problème correspondant à une parcelle. Le but est de choisir un ensemble de traitements qui permet de protéger les vignes des différentes maladies au cours du temps. La composition des traitements choisis doit respecter des normes environnementales. On cherche un ensemble de traitements phytosanitaires réalisables qui minimise le coût. Nous décrirons les critères de réalisabilité d'une séquence de traitements. L'objectif de cette section est de voir le comportement de nos méthodes sur des programmes dynamiques de grandes tailles avec des structures complexes.

6.3.1 Description du problème

Pour chaque parcelle, un ensemble de maladies R doit être traité sur un horizon de temps de T périodes. Chaque maladie $r \in R$ doit être traitée dans un (ou plusieurs) intervalles de temps $T_r = [e_r, l_r] \in [1, T]$. Pour traiter ces maladies, on dispose d'un ensemble P de traitements disponibles sur le site. Pour chaque traitement $p \in P$, on note R^p l'ensemble des maladies traitées par le produit p. La durée d'application d'un traitement est noté d_p et l'ensemble de périodes où le produit peut être appliqué T_p. On note également D_{rpt} la durée d'application de $p \in P$ contre la maladie $r \in R$ s'il est appliqué en $t \in T$. Chaque produit est composé de plusieurs composants $c \in C$, avec C^p l'ensemble des composants du produit $p \in P$, et P^c l'ensemble des produits contenant le composant $c \in C$. Le coût d'appliquer un produit $p \in P$ au temps t est noté c_{pt}. Une illustration du problème est donnée dans la figure 6.3

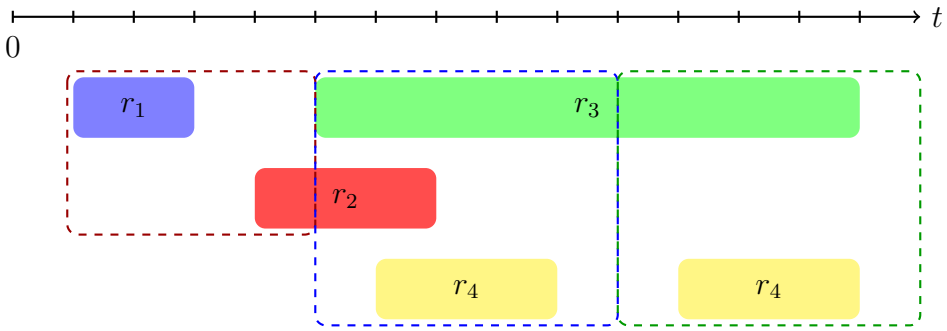


FIGURE 6.3 – Illustration d'une instance du problème de traitements phytosanitaires des vignes

Il existe plusieurs règles permettant de modéliser le problème. A chaque instant de temps, on ne peut appliquer qu'un seul traitement sur le site. De plus, on note $Succ(p)$ l'ensemble des traitements pouvant être appliqués juste après $p \in P$. Chaque traitement déverse des produits toxiques (P^{pol} est l'ensemble des produits contenant des substances toxiques). La quantité déversée dépend du temps t d'application du traitement et est notée tox_{pt}. Cette quantité est limitée par un seuil de tolérance tol.

Le nombre d'utilisations de chaque composant sur l'ensemble des périodes $c \in C$ est limité par N_c.

6.3.2 Programme linéaire en nombres entiers

On peut modéliser le problème sous forme de programme linéaire en nombres entiers. Chaque variable binaire $\delta_{pt}, \forall p \in P, t \in [1, T]$ est égale à 1 si le produit p est appliqué au temps t.

$$\min \sum_{p\in P, t\in T_p} c_{pt}\delta_{pt} \tag{6.8}$$

$$\sum_{p\in P_r, \tau\in T_p : \tau\in[t-D_{rp\tau}+1,t]} \delta_{p\tau} \geq 1 \quad \forall r \in R, t \in T_r \tag{6.9}$$

$$\sum_{p\in P} \delta_{pt} \leq 1 \quad \forall t \in [1,T] \tag{6.10}$$

$$\sum_{p\in P_c, t\in[1,T]} \delta_{pt} \leq N_c \quad \forall c \in C^N \tag{6.11}$$

$$\sum_{p\in P^{pol}, t\in T_p} tox_{pt}\delta_{pt} \leq \text{tol} \tag{6.12}$$

$$\delta_{pt} + \sum_{\substack{p'\in P_r \\ p'\notin Succ_{pr}}} \delta_{p',t+d} \leq 1 + \sum_{p'\in Succ_{pr}} \sum_{\tau=t+1}^{t+d-1} \delta_{p'\tau} \quad \forall r \in R, p \in P_r, t \in T_p, d \in [1, D_{max}] \tag{6.13}$$

$$\delta_{pt} \in \{0,1\} \quad \forall p,t \tag{6.14}$$

Détaillons les contraintes de ce modèle. La contrainte (6.9) impose que chaque maladie doit être traitée à chaque instant de temps par un produit. La contrainte (6.10) limite le nombre de produits utilisés à chaque instant de temps. Le nombre d'utilisation de chaque composant est controlé par les contraintes (6.11), et le seuil de tolérance à la toxicité par la contrainte (6.12). Enfin, les contraintes (6.13) correspondent aux contraintes sur les successeurs des différents produits appliqués.

6.3.3 Programme dynamique

Nous décrivons le programme dynamique en définissant les états et les transitions. Un état du programme dynamique est défini par un tuple(t, D, L, tox, Q) avec :

- $t \in [0,T]$ le temps courant.
- $D \in \mathbb{N}^R$ pour chaque maladie r, la prochaine date où la maladie doit être traitée.
- $L \in \mathbb{N}^R$ pour chaque maladie r, la dernière mixture utilisée pour traiter r.
- $tox \in \mathbb{R}$ la toxicité accumulée jusqu'au temps t.
- $Q \in \mathbb{N}^C$ pour chaque composant c, le nombre d'utilisation avant t.

Une transtion du programme dynamique est un tuple $(\Delta_D, \Delta_L, \Delta_{tox}, \Delta_Q, p)$ où $\Delta_D \in \mathbb{N}^R$ est le vecteur contenant les prochaines dates où les maladies doivent être traités,$\Delta_L \in \mathbb{N}^R$ est le vecteur qui contient la dernière mixture utilisée contre chaque maladie, $\Delta_{tox} \in \mathbb{N}$

la consommation de toxicité quand la décision est prise, $\Delta_Q \in \mathbb{N}^C$ le vecteur qui met à jour le nombre d'utilisation de chaque composant et $p \in \mathbb{N}$ le coût obtenu quand la décision est prise.

Afin d'exprimer l'ensemble de transitions ϕ pouvant entre appliquées à un état, on définit plusieurs fonctions facilitant son écriture.

Définition 19. *Mixtures pouvant être appliquées*

$\forall t \in [0, T], D \in \mathbb{N}^R, L \in \mathbb{N}^R$, l'ensemble des mixtures pouvant être appliquées au temps t après les mixtures L et permettant de traiter les maladies D est donné par :

$$P(t, D, L) = (\bigcap_{r:D_r=t} P_r) \cap (\bigcap_{r \in R} Succ(L_r))$$

Les mixtures possibles doivent pouvoir traiter l'ensemble des maladies devant être traitées au temps t et doivent respecter les règles de précédence.

Définition 20. *Prochaines dates où les maladies doivent être traitées*

$$d(t, D, p) = \begin{cases} \forall r \in R^p, d(t, D, p)_r = min\{s : s \in T_r, s \geq t + d_p\} \\ \forall r \notin R^p, d(t, D, p)_r = D_r \end{cases}$$

Les prochaines dates où les maladies doivent être traitées augmentent de la durée du traitement appliqué.

Définition 21. *Dernière mixture appliquée pour chaque maladie*

$$a(L, p) = \begin{cases} \forall r \in R^p, a(L, p)_r = p \\ \forall r \notin R^p, a(L, p)_r = L_r \end{cases}$$

L'ensemble des transitions appliquables à un état (t, D, L, tox, Q) est défini par :

$$\psi(t, D, L, tox, Q) = \bigcup_{p \in P(t,D,L)} \{(d(t, D, p), a(L, p), tox_p, C_p, c_{p,t})\}$$

La fonction de coût α d'un état permet de définir la formule de récurrence du programme dynamique :

$$\alpha(t, D, L, tox, Q) = \min_{(\Delta_D, \Delta_L, \Delta_{tox}, \Delta_Q, p) \in \psi((t,D,L,tox,Q))} \{\alpha(t + 1, \Delta_D, \Delta_L, tox + \Delta_{tox}, Q + \Delta_Q) + p\}$$

6.3.4 Résultat

Nous comparons les résultats obtenus par notre méthode et les résultats obtenus en utilisant le solveur de programmation linéaire IBM Ilog CPLEX. La figure 6.2 montre les résultats obtenus en utilisant un Intel Xeon E5-2680 v3 2,5 GHz 2 Dodeca-core Haswell avec 128Go RAM en utilisant 6 threads. Les résultats obtenus pour SSDP ont

été obtenus en utilisant le critère 1 correspondant aux multiplicateurs Lagrangien et en ajoutant une seule contrainte à chaque itération. Le temps limite est fixé à une heure.

TABLE 6.2 – Temps obtenus avec SSDP et le solveur de programmation linéaire CPLEX sur quelques instances du problème. (TL correspond aux instances non résolues dans la limite de temps)

Instance	CPLEX	SSDP
14	75.2359	TL
16	77.8647	1677.21
32	73.8244	1202.49
38E	119.188	TL
57W	116.422	TL
58D	73.9504	1511.32
61	76.4248	398.067
79W	73.4444	TL
85N	78.4938	1649.06
Bergieu2	9.22907	4.83955
Claous3	8.85647	4.77527
Guillette3	11.6667	7.25896
Montissan2	8.2929	7.37301
Peyvignau4b	9.18966	5.13024
Sablons1a	8.25281	8.28955
Troisponts2	12.566	7.93878
Troisponts4c	12.5662	8.02739
VieuxSémillons1	8.28681	7.25734

Les performances obtenues en utilisant CPLEX sont meilleures que celles obtenues avec notre méthode. SSDP n'arrive pas à résoudre à l'optimalité 4 instances, alors que CPLEX résout l'intégralité des instances testées. Notons que pour les 9 premières instances, le temps de résolution de SSDP est considérablement plus long que celui de CPLEX, mais pour les instances plus faciles (9 dernières), le temps de résolution de SSDP est plus court que celui de CPLEX.

Il y a plusieurs facteurs qui expliquent les différences de résultats de SSDP. Pour commencer, les bornes utilisées pour ce problème sont de mauvaises qualités. La borne primale a été fixée à une valeur arbitraire. Trouver une solution réalisable pour ce problème est difficile ce qui ne permet pas pour l'instant d'avoir une bonne heuristique. La borne duale donnée par la relaxation est elle aussi de mauvaise qualité. Ensuite, nous n'avons pas ajouté de tests de réalisabilité pour supprimer des transitions. Le premier graphe obtenu sur les instances qui n'ont pas été résolues par SSDP sont déjà très grands (plusieurs millions de sommets, et des dizaines de millions d'arcs).

6.4 Conclusion

Dans ce chapitre, nous avons appliqué la méthode SSDP a deux problèmes de planification. Le premier problème traité est un problème d'ordonnancement sur lequel la méthode SSDP a été utilisé dans la littérature. L'analyse des résultats nous permet de comparer notre implémentation générique avec une implémentation spécifique proposée par [TFA09]. Les différences de temps de calculs permettent de montrer les limites de notre implémentation générique. Le temps utilisé pour résoudre le sous problème Lagrangien et pour construire le graphe de la nouvelle relaxation à chaque itération est plus important dans notre implémentation. Le deuxième problème est un problème de traitement phytosanitaire des vignes. L'application de la méthode sur ce problème industriel a permis de montrer les difficultés rencontrées sur un problème de structure diiférente. Deux difficultés sont apparues lors de l'application : la mauvaise qualité des bornes initiales qui impactent l'efficacité de SSDP et des contraintes linéaires dont la partie droite peut prendre une valeur supérieure à 1 (à l'inverse des autres problèmes traités dans cette thèse, où le membre de droite est égal à 1).

Chapitre 7

Conclusions et perspectives

Dans ce manuscrit, nous avons proposé un formalisme unificateur permettant de décrire des problèmes modélisés sous la forme d'un programme dynamique et de contraintes linéaires additionnelles. Il permet d'exprimer différentes méthodes de résolution de la littérature, en particulier les méthodes itératives d'agrégation de l'espace d'états. Le formalisme permet de mettre en évidence les structures similaires de ces méthodes. En effet, les méthodes proposées sont toutes basées sur un graphe support, correspondant au programme dynamique initial auquel des contraintes linéaires peuvent être ajoutées, et un algorithme de résolution de plus court chemin prenant en compte les contraintes linéaires additionnelles restantes. La différence majeure entre ces méthodes réside dans la manière de contrôler les contraintes linéaires additionnelles qui sont ajoutées au fur et à mesure de ces méthodes. Les contraintes peuvent être vérifiées dans la structure du graphe support ou par l'algorithme de résolution (label setting, branch-and-bound, ...).

Dans le chapitre 4, nous avons identifié plusieurs éléments clés permettant d'utiliser ces méthodes itératives en pratique. Ces éléments ont été implémentés dans un code générique facilitant la conception de méthodes itératives. L'agrégateur présenté dans la section 4.4 montre que l'on peut construire un agrégateur définissant un sur-ensemble plus précis des états agrégés sans trop augmenter l'espace mémoire nécessaire. Plusieurs critères de sélection de contraintes ont été proposés et peuvent être utilisés sur tous les problèmes qui peuvent s'exprimer dans le formalisme. Nous avons identifié des briques génériques permettant d'améliorer la performance de ces algorithmes et montré leur intérêt et leurs limites dans les applications.

Ces éléments ont été appliqués sur le problème de sac à dos temporel dans le chapitre 5. Notre implémentation a permis de montrer que la méthode SSDP peut obtenir des résultats compétitifs avec la littérature sur ce problème. Une analyse approfondie des résultats obtenus permet de voir l'impact de chacun de ces éléments sur les performances de la méthode. Notre implémentation a permis de résoudre 94 des 100 instances de la littérature en fixant le temps limite à 3 heures. A notre connaissance, la seule implémentation efficace de cette méthode a été proposé par [TFA09] sur un problème d'ordonnancement, ce qui a permis de montrer que cette méthode peut être utilisée en pratique mais nécessite de nombreux ajustements. En effet, de nombreux raffinements ont été apportés à SSDP pour être efficace sur le problème. Une partie de ces raffinements

a pu être proposé de manière générique, mais d'autres sont propres au problème.

Le chapitre 6 a permis de montrer les limites de notre implémentation générique actuelle. La comparaison des résultats avec l'implémentation de [TFA09] a montré un écart important des temps de calculs entre les deux implémentations. La construction du graphe à chaque itération en intégrant les nouvelles contraintes est l'étape où les différences de temps sont les plus notables. Les résultats obtenus sur le problème de traitements phytosanitaires des vignes montrent la difficulté à résoudre des problèmes plus complexes. Ces résultats peuvent être expliqués par deux éléments : la mauvaise qualité des bornes et l'importance des contraintes relâchées qui sont trop structurantes. L'intégration d'inégalités valides dans le problème serait une piste à étudier.

Ce travail a permis de faire apparaitre plusieurs perspectives. Le formalisme proposé nous amène à considérer une nouvelle méthode itérative basée sur l'hybridation de SSDP et DSSR. Comme la méthode DDSR se base sur un graphe support, une manière de construire ce graphe est d'utiliser la méthode SSDP. Cette méthode construit de manière explicite le graphe correspondant aux relaxations successives. Il est donc possible de commencer par utiliser la méthode SSDP sur quelques itérations, puis en partant d'une relaxation obtenue au cours de la méthode, utiliser la méthode DSSR. Cette hybridation permettrait de profiter du filtrage utilisé pour la construction du graphe grâce à SSDP, et d'utiliser les dominances de labels de la méthode DSSR. Cette hybridation a été implémenté mais ne donne pas encore de résultats satisfaisants. Pour améliorer celle-ci, une meilleure implémentation de DSSR est nécessaire.

Nous avons proposé plusieurs éléments génériques dans l'implémentation et le raffinement de ces méthodes. Ces méthodes sont applicables sur de nombreux problèmes et peuvent être testées grâce à la bibliothèque logicielle dédiée. Certaines techniques ont été utilisées sur des problèmes spécifiques et ont permis d'améliorer les méthodes présentées. C'est le cas par exemple de l'énumération partielle proposée pour le problème de sac à dos temporel qui peut être généralisée en énumérant des transitions successives.

Un des points clés des méthodes itératives est le choix des dimensions à ajouter à chaque itération. Nous avons considéré uniquement le cas où des dimensions sont ajoutées à l'espace d'états, mais il est possible d'enlever des dimensions. La désagrégation du programme dynamique doit prendre en compte les transitions qui ont été retirées. La problématique de choix de dimensions est la même que pour l'ajout de dimensions. Il faut aussi être vigilant car la convergence de ces algorithmes est assurée par l'ajout d'au moins une dimension par itération.

Les méthodes présentées dans ce manuscrit sont basées sur l'agrégation de l'espace d'états, mais d'autres types d'agrégations peuvent être utilisés. On peut notamment citer [CHM$^+$17] qui proposent une méthode d'agrégation pour les problèmes de flot avec contraintes additionnelles.

Les travaux de [CSVV18] ont montré que la méthode DSSR peut être étendue aux programmes dynamiques modélisés sous la forme d'hypergraphe. La possibilité d'utiliser les méthodes d'agrégation itératives sur ce type de problème est prometteuse, malgré de nombreuses questions techniques et d'adaptation des raffinements proposés dans le chapitre 4.

Nous nous sommes intéressés à une famille restreinte de programmes dynamiques.

Ces méthodes peuvent être généralisées pour des familles de programmes dynamiques avec contraintes de resources plus complexes. On peut notamment citer les travaux de [Par16] qui présentent des problèmes de ce type dans le cadre des monoïdes. Les méthodes d'agrégation ne sont pas encore utilisées sur ce type de programme dynamique plus complexe à notre connaissance.

Bibliographie

[AF75] Joachim H Ahrens and Gerd Finke. Merging and sorting applied to the zero-one knapsack problem. *Operations Research*, 23(6) :1099–1109, 1975.

[AMO88] Ravindra K Ahuja, Thomas L Magnanti, and James B Orlin. Network flows. 1988.

[ARP88] TS Abdul-Razaq and CN Potts. Dynamic programming state-space relaxation for single-machine scheduling. *Journal of the Operational Research Society*, 39(2) :141–152, 1988.

[ARPVW90] TS Abdul-Razaq, Chris N Potts, and Luk N Van Wassenhove. A survey of algorithms for the single machine total weighted tardiness scheduling problem. *Discrete Applied Mathematics*, 26(2-3) :235–253, 1990.

[AS87] E. M. Arkin and E.B. Silverberg. Scheduling with fixed start and end times. *Discrete Applied Mathematics*, 18 :1–8, 1987.

[AW09] Kurt M Anstreicher and Laurence A Wolsey. Two well-known properties of subgradient optimization. *Mathematical Programming*, 120(1) :213–220, 2009.

[BA00] Francisco Barahona and Ranga Anbil. The volume algorithm : producing primal solutions with a subgradient method. *Mathematical Programming*, 87(3) :385–399, 2000.

[Bal65] Michel Louis Balinski. Integer programming : methods, uses, computations. *Management science*, 12(3) :253–313, 1965.

[BB81] Kenneth R Baker and JWM Bertrand. An investigation of due-date assignment rules with constrained tightness. *Journal of Operations Management*, 1(3) :109–120, 1981.

[BBS87] James C Bean, John R Birge, and Robert L Smith. Aggregation in dynamic programming. *Operations Research*, 35(2) :215–220, 1987.

[BC89] John E Beasley and Nicos Christofides. An algorithm for the resource constrained shortest path problem. *Networks*, 19(4) :379–394, 1989.

[BDD06] Natashia Boland, John Dethridge, and Irina Dumitrescu. Accelerated label setting algorithms for the elementary resource constrained shortest path problem. *Operations Research Letters*, 34(1) :58–68, 2006.

[Bel54] Richard Bellman. The theory of dynamic programming. Technical report, RAND Corp Santa Monica CA, 1954.

[Bel58] Richard Bellman. On a routing problem. *Quarterly of applied mathematics*, 16(1) :87–90, 1958.

[BFH+05] Mark Bartlett, Alan M. Frisch, Youssef Hamadi, Ian Miguel, S. Armagan Tarim, and Chris Unsworth. The Temporal Knapsack Problem and Its Solution. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 3524, pages 34–48. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[BSU18] Teobaldo Bulhoes, Ruslan Sadykov, and Eduardo Uchoa. A branch-and-price algorithm for the minimum latency problem. *Computers & Operations Research*, 93 :66–78, 2018.

[BSW14] P.S. Bonsma, Jens Schulz, and Andreas Wiese. A constant-factor approximation algorithm for unsplittable flow on paths. *SIAM journal on computing*, 43(2) :767–799, 2014.

[CC81] Leon Cooper and Mary W. Cooper. *Introduction to dynamic programming.* Number v. 1 in International series in modern applied mathematics and computer science. Pergamon Press, Oxford ; New York, 1st ed edition, 1981.

[CCKR02] Gruia Calinescu, Amit Chakrabarti, Howard Karloff, and Yuval Rabani. Improved approximation algorithms for resource allocation. In *International Conference on Integer Programming and Combinatorial Optimization*, pages 401–414. Springer, 2002.

[CDG20] François Clautiaux, Boris Detienne, and Gaël Guillot. An iterative dynamic programming approach for the temporal knapsack problem. *European Journal of Operational Research*, 2020.

[CFM13] A. Caprara, F. Furini, and E. Malaguti. Uncommon dantzig-wolfe reformulation for the temporal knapsack problem. *INFORMS Journal on Computing*, 25(3) :560–571, 2013.

[CFMT16] Alberto Caprara, Fabio Furini, Enrico Malaguti, and Emiliano Traversi. Solving the temporal knapsack problem via recursive dantzig–wolfe reformulation. *Information Processing Letters*, 116(5) :379–386, 2016.

[CHM+17] François Clautiaux, Saïd Hanafi, Rita Macedo, Marie-Emilie Voge, and Cláudio Alves. Iterative aggregation and disaggregation algorithm for

pseudo-polynomial network flow models with side constraints. *European Journal of Operational Research*, 258(2) :467–477, 2017.

[CHT02] B. Chen, Refael Hassin, and Michal Tzur. Allocation of bandwidth and storage. *IIE Transactions*, 34(5) :501–507, 2002.

[CMT81] Nicos Christofides, A. Mingozzi, and P. Toth. State-space relaxation procedures for the computation of bounds to routing problems. *Networks*, 11(2) :145–164, 1981.

[CPRS01] Yu-Li Chou, Stephen M Pollock, H Edwin Romeijn, and Robert L Smith. A formalism for dynamic programming. *Department of Industrial and Operations Engineering, The University of Michigan*, 2001.

[CPvdV02] Richard K Congram, Chris N Potts, and Steef L van de Velde. An iterated dynasearch algorithm for the single-machine total weighted tardiness scheduling problem. *INFORMS Journal on Computing*, 14(1) :52–67, 2002.

[CSVV18] François Clautiaux, Ruslan Sadykov, François Vanderbeck, and Quentin Viaud. Combining dynamic programming with filtering to solve a four-stage two-dimensional guillotine-cut bounded knapsack problem. *Discrete Optimization*, 29 :18–44, 2018.

[Dan51] George B Dantzig. Maximization of a linear function of variables subject to linear inequalities. *Activity analysis of production and allocation*, 13 :339–347, 1951.

[DDS92] Martin Desrochers, Jacques Desrosiers, and Marius Solomon. A new optimization algorithm for the vehicle routing problem with time windows. *Operations research*, 40(2) :342–354, 1992.

[Den82] Eric V Denardo. Dynamic programming : Theory and application. prentice hall. *New Jersey*, 1982.

[Dre69] Stuart E Dreyfus. An appraisal of some shortest-path algorithms. *Operations research*, 17(3) :395–412, 1969.

[DS88] Martin Desrochers and François Soumis. A generalized permanent labelling algorithm for the shortest path problem with time windows. *INFOR : Information Systems and Operational Research*, 26(3) :191–212, 1988.

[DW60] George B Dantzig and Philip Wolfe. Decomposition principle for linear programs. *Operations research*, 8(1) :101–111, 1960.

[Elm68] Salah E Elmaghraby. The one-machine sequencing problem with delay costs. *Journal of Industrial Engineering*, 19 :105–108, 1968.

[Emm69] Hamilton Emmons. One-machine sequencing to minimize certain functions of job tardiness. *Operations Research*, 17(4) :701–715, 1969.

[ESB10] Niklaus Eggenberg, Matteo Salani, and Michel Bierlaire. Constraint-specific recovery network for solving airline recovery problems. *Computers & operations research*, 37(6) :1014–1026, 2010.

[Fis85] Marshall L Fisher. An applications oriented guide to lagrangian relaxation. *Interfaces*, 15(2) :10–21, 1985.

[GDCT04] Andrea Grosso, Federico Della Croce, and Roberto Tadei. An enhanced dynasearch neighborhood for the single-machine total weighted tardiness scheduling problem. *Operations Research Letters*, 32(1) :68–72, 2004.

[Geo74] Arthur M Geoffrion. Lagrangean relaxation for integer programming. In *Approaches to integer programming*, pages 82–114. Springer, 1974.

[GI17] Timo Gschwind and Stefan Irnich. Stabilized column generation for the temporal knapsack problem using dual-optimal inequalities. *OR Spectrum*, 39(2) :541–556, 2017.

[GL88] Andre Gascon and Robert C Leachman. A dynamic programming solution to the dynamic, multi-item, single-machine scheduling problem. *Operations Research*, 36(1) :50–56, 1988.

[Gom60] Ralph Gomory. An algorithm for the mixed integer problem. Technical report, RAND CORP SANTA MONICA CA, 1960.

[Hin78] K Hinderer. On approximate solutions of finite-stage dynamic programs. In *Dynamic programming and its applications*, pages 289–317. Elsevier, 1978.

[HK62] Michael Held and Richard M Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied mathematics*, 10(1) :196–210, 1962.

[Iba87] Toshihide Ibaraki. Successive sublimation methods for dynamic programming computation. *Annals of Operations Research*, 11(1) :397–439, December 1987.

[ID05] Stefan Irnich and Guy Desaulniers. Shortest path problems with resource constraints. In *Column generation*, pages 33–65. Springer, 2005.

[IN94] Toshihide Ibaraki and Yuichi Nakamura. A dynamic programming method for single machine scheduling. *European Journal of Operational Research*, 76(1) :72–82, 1994.

[JC11] Antoine Jouglet and Jacques Carlier. Dominance rules in combinatorial optimization problems. *European Journal of Operational Research*, 212(3) :433–444, 2011.

[Kau67] Arnold Kaufmann. *Dynamic programming : Sequential scientific management*, volume 37. Academic Press, 1967.

[KH67] Richard M Karp and Michael Held. Finite-state processes and dynamic programming. *SIAM Journal on Applied Mathematics*, 15(3) :693–718, 1967.

[KZBV16] Morteza Keshtkaran, Koorush Ziarati, Andrea Bettinelli, and Daniele Vigo. Enhanced exact solution methods for the team orienteering problem. *International Journal of Production Research*, 54(2) :591–601, 2016.

[Law77] Eugene L Lawler. A pseudopolynomial algorithm for sequencing jobs to minimize total tardiness. In *Annals of discrete Mathematics*, volume 1, pages 331–342. Elsevier, 1977.

[LKB77] Jan Karel Lenstra, AHG Rinnooy Kan, and Peter Brucker. Complexity of machine scheduling problems. In *Annals of discrete mathematics*, volume 1, pages 343–362. Elsevier, 1977.

[Mar76] Alberto Martelli. An application of heuristic search methods to edge and contour detection. *Communications of the ACM*, 19(2) :73–83, 1976.

[Min75] Michel Minoux. Plus court chemin avec contraintes : algorithmes et applications. In *Annales des télécommunications*, volume 30, pages 383–394. Springer, 1975.

[Mor78] Thomas L Morin. Computational advances in dynamic programming. In *Dynamic programming and its applications*, pages 53–90. Elsevier, 1978.

[Mor82] Thomas L Morin. Monotonicity and the principle of optimality. *Journal of Mathematical Analysis and Applications*, 88(2) :665–674, 1982.

[Neu93] K Neumann. Dynamic programming basic concepts and applications. In *Optimization in Planning and Operation of Electric Power Systems*, pages 31–56. Springer, 1993.

[Par16] Axel Parmentier. *Quelques Algorithmes pour des problèmes de plus court chemin et d'opérations aériennes*. Theses, Université Paris-Est, November 2016.

[Pol69] Boris Teodorovich Polyak. Minimization of nonsmooth functionals. *Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki*, 9(3) :509–521, 1969.

[PWW69] A Alan B Pritsker, Lawrence J Waiters, and Philip M Wolfe. Multiproject scheduling with limited resources : A zero-one programming approach. *Management science*, 16(1) :93–108, 1969.

[RS08] Giovanni Righini and Matteo Salani. New dynamic programming algorithms for the resource constrained elementary shortest path problem. *Networks : An International Journal*, 51(3) :155–170, 2008.

[SW92] Jorge P Sousa and Laurence A Wolsey. A time indexed formulation of non-preemptive single machine scheduling problems. *Mathematical programming*, 54(1-3) :353–367, 1992.

[TFA09] Shunji Tanaka, Shuji Fujikuma, and Mituhiko Araki. An exact algorithm for single-machine scheduling without machine idle time. *Journal of Scheduling*, 12(6) :575–593, 2009.

[Tot80] Paolo Toth. Dynamic programming algorithms for the zero-one knapsack problem. *Computing*, 25(1) :29–45, 1980.

[TS13] Shunji Tanaka and Shun Sato. An exact algorithm for the precedence-constrained single-machine scheduling problem. *European Journal of Operational Research*, 229(2) :345–352, 2013.

[Whi79] Ward Whitt. Approximations of dynamic programs, ii. *Mathematics of Operations Research*, 4(2) :179–185, 1979.

[WN99] Laurence A Wolsey and George L Nemhauser. *Integer and combinatorial optimization*, volume 55. John Wiley & Sons, 1999.

[WVHK92] Albert Wagelmans, Stan Van Hoesel, and Antoon Kolen. Economic lot sizing : an o (n log n) algorithm that runs in linear time in the wagner-whitin case. *Operations Research*, 40(1-supplement-1) :S145–S156, 1992.

[WW58] Harvey M Wagner and Thomson M Whitin. Dynamic version of the economic lot size model. *Management science*, 5(1) :89–96, 1958.