THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par : *l'Institut National des Sciences Appliquées de Toulouse (INSA de Toulouse)*

Présentée et soutenue le *28 avril 2022* **par :**

Léa BLAISE

Modélisation et résolution de problèmes d'ordonnancement au sein du solveur d'optimisation mathématique LocalSolver

JURY

Jean-Charles BILLAUT	Professeur des Universités	Rapporteur
Laetitia JOURDAN	Professeur des Universités	Rapporteur
Olivier HUDRY	Professeur des Universités	Président du jury
Christine SOLNON	Professeur des Universités	Examinateur
Christian ARTIGUES	Directeur de Recherche	Directeur de Thèse
Thierry BENOIST	Directeur Associé LocalSolver	Directeur de Thèse

École doctorale et spécialité :
MITT : Domaine Mathématiques : Mathématiques appliquées
Unité de Recherche :
Laboratoire d'analyse et d'architecture des systèmes (LAAS)
Directeur(s) de Thèse :
Christian ARTIGUES et *Thierry BENOIST*
Rapporteurs :
Jean-Charles BILLAUT et *Laetitia JOURDAN*

Résumé

Résoudre un problème d'ordonnancement consiste à organiser la réalisation de tâches au cours du temps : déterminer leur répartition sur les différentes ressources disponibles ainsi que leurs dates d'exécution. Les problèmes d'ordonnancement se rencontrent dans tous les domaines de l'industrie et des services, et sont très étudiés dans la littérature. Le travail de cette thèse se concentre sur les problèmes d'ordonnancement de type disjonctif et/ou packing, avec ou sans flexibilité des ressources. L'ensemble des contributions algorithmiques de la thèse ont été implémentées au sein du solveur d'optimisation mathématique LocalSolver, dont les techniques de résolution combinent des méthodes exactes, telles que la programmation linéaire, non linéaire et par contraintes, et heuristiques, comme la recherche locale et des algorithmes constructifs.

Le travail de cette thèse répond à deux problématiques principales, liées au traitement de ce type de problèmes d'ordonnancement par LocalSolver. Le premier objectif se dégageant de ces problématiques consiste à permettre aux utilisateurs du solveur de modéliser simplement un grand nombre de problèmes d'ordonnancement disjonctif. En tirant profit de la richesse du formalisme de modélisation ensembliste de LocalSolver, on propose des formulations génériques, adaptables à différentes familles de problèmes d'ordonnancement disjonctif, permettant d'exprimer simplement les notions de tâches, de relations de précédence, ou encore de contraintes de non-chevauchement des tâches affectées à une même ressource disjonctive. Les formulations génériques ainsi choisies reposent sur l'utilisation combinée de deux types de variables de décision offertes par LocalSolver : les variables entières, permettant de modéliser les dates de début et durées de tâches, et les variables de listes, représentant leur ordre sur les différentes ressources disjonctives.

Le second objectif de la thèse consiste à améliorer les performances de LocalSolver sur les problèmes d'ordonnancement étudiés, en intégrant différents algorithmes de résolution les plus génériques possibles à la composante de recherche locale du solveur. Cette généricité des contributions est cruciale : en effet, on ne cherche pas à améliorer les performances du solveur sur un unique problème, ni même seulement sur les problèmes d'ordonnancement, mais sur tous les problèmes présentant des structures caractéristiques de l'ordonnancement disjonctif ou du packing.

Les contributions algorithmiques de cette thèse peuvent être regroupées en trois grandes catégories : des algorithmes d'initialisation, des mouvements de recherche locale, et un algorithme de propagation de contraintes. On présente deux algorithmes constructifs d'initialisation des variables ensemblistes (sets et listes), aidant le solveur à trouver une solution réalisable immédiatement sur des problèmes comme ceux de l'Aircraft Landing ou de l'Assembly Line Balancing, et accélérant ainsi la recherche de solutions de bonne qualité sur ces problèmes. On présente également des mouvements de recherche locale, reposant sur la détection de structures spécifiques dans le modèle (contraintes de non-chevauchement des tâches, relations de précédence, contraintes de packing). On présente également un algorithme de réparation de solutions par propagation de contraintes, appelé après chaque mouvement de recherche locale conduisant à une solution infaisable. Notre algorithme diffère de la propagation classique de la programmation par contraintes par plusieurs points. Par exemple, il ne propage que les réductions de domaine excluant la valeur courante des variables, et peut prendre des décisions arbitraires lorsqu'il rencontre une contrainte pouvant être réparée de différentes manières. On démontre que dans certains cas l'algorithme présente des propriétés lui assurant de trouver une réparation s'il en existe une. Cet algorithme permet de pallier les difficultés rencontrées par la recherche locale sur les problèmes d'ordonnancement aux contraintes très serrées (passer d'une bonne solution à une autre nécessite de réaliser de nombreux changements sur les variables). L'intégration de ces mouvements et de cet algorithme de réparation au sein de la recherche locale de LocalSolver apporte des gains de performance importants sur divers problèmes (Job Shop et variantes, Unit Commitment, Assembly Line Balancing, ou encore Bin Packing).

Mots clés : solveur, ordonnancement disjonctif, packing, recherche locale, propagation de contraintes

Abstract

Solving a scheduling problem consists in organizing the realization of tasks in the course of time: determining their distribution on the various available resources as well as their execution dates. Scheduling problems are encountered in all domains of industry and services, and are often studied in the literature. This thesis focuses on disjunctive scheduling and/or packing problems, with or without flexibility on the resources. All the algorithmic contributions of this thesis have been implemented within the mathematical optimization solver LocalSolver, whose resolution techniques combine exact methods, such as linear, nonlinear and constraint programming, and heuristics, such as local search and constructive algorithms.

This thesis addresses two main issues related to the treatment of this type of scheduling problems by LocalSolver. The first objective is to easily allow users to model many disjunctive scheduling problems. By taking advantage of LocalSolver's set-based modeling formalism, we propose generic formulations, adaptable to different disjunctive scheduling problem families, in order to simply express the concepts of tasks, precedence relations, or non-overlap constraints on the tasks assigned to the same disjunctive resource. These generic formulations are based on the combined use of two types of decision variables offered by LocalSolver: integer variables, representing the start dates and durations of tasks, and list variables, representing their order on the various disjunctive resources.

The second objective of the thesis is to improve the performance of LocalSolver on the considered scheduling problems, by integrating different resolution algorithms, as generic as possible, to the local search component of the solver. The genericity of the contributions is crucial: indeed, we do not aim at improving the performance of the solver on a single problem, nor even only on scheduling problems, but on all problems presenting certain characteristic structures often encountered in disjunctive scheduling or in packing.

The algorithmic contributions of this thesis can be grouped into three main categories: initialization algorithms, local search moves, and a constraint propagation algorithm. We present two constructive initialization algorithms for set and list variables, which help the solver to immediately find a feasible solution to problems such as the Aircraft Landing or Assembly Line Balancing Problems, and thus accelerate the search for good quality solutions. We also present several local search moves, based on the detection of specific structures in the model (non-overlap constraints, precedence relations, packing constraints). We also present a solution repair algorithm by constraint propagation, integrated into the solver's local search component, and called after each local move leading to an infeasible solution. Our algorithm differs from classical constraint propagation in several ways. For example, it only propagates domain reductions that exclude the current values of the variables, and can make arbitrary decisions when it encounters a constraint that can be repaired in different ways. We prove that in some cases the algorithm has properties that ensure that the propagation will succeed if the solution can be repaired. This algorithm overcomes the difficulties encountered by local search on tightly constrained scheduling problems (moving from a good solution to another requires making changes on a large number of variables). The integration of these local moves and this repair algorithm into LocalSolver's local search component brings significant performance gains on various problems (Job Shop and variants, Unit Commitment, Assembly Line Balancing, or Bin Packing).

Key words: solver, disjunctive scheduling, packing, local search, constraint propagation

Remerciements

Je souhaite tout d'abord adresser ces remerciements à mes directeurs de thèse, pour m'avoir fait confiance et m'avoir permis de mener à bien ce projet de thèse. Je remercie tout d'abord Thierry Benoist, avec qui j'ai eu la chance de travailler au quotidien chez LocalSolver, pour son encadrement scientifique et technique de grande qualité, et son investissement dans mes travaux. Je remercie également Christian Artigues, qui m'a suivie d'un peu plus loin depuis le LAAS-CNRS mais avec autant d'implication, pour ses conseils précieux et sa disponibilité malgré un emploi du temps chargé.

Je souhaite également remercier mes rapporteurs, Jean-Charles Billaut et Laetitia Jourdan, pour leur assiduité dans leur travail de relecture de mon manuscrit, ainsi que pour la pertinence de leurs nombreuses questions lors de la soutenance. Je remercie également les membres du jury, Olivier Hudry et Christine Solnon, pour l'intérêt qu'ils ont porté à mon travail, et pour les échanges enrichissants que j'ai pu avoir avec eux lors de la soutenance. Je souhaite remercier tout particulièrement Olivier Hudry, qui a accepté la présidence du jury, et qui a suivi mon parcours depuis mon entrée à Télécom ParisTech en 2015, pour ses conseils avisés, sa disponibilité et sa pédagogie exceptionnelle.

Je tiens également à remercier toute l'équipe de LocalSolver, aux côtés de laquelle j'ai eu le plaisir de travailler durant mes trois années de thèse. Je remercie également l'équipe ROC du LAAS-CNRS, avec laquelle je n'ai malheureusement pas eu l'occasion d'échanger autant que je l'aurais souhaité, mais qui m'a toujours très bien accueillie.

Enfin, je souhaite remercier ma famille et mes amis pour leur soutien. Je remercie en particulier Anna, Séverine, et ma mère Sophie, pour m'avoir aidée à améliorer la qualité de mon manuscrit par leurs relectures.

Table des matières

Chapitre 1

Introduction

1.1 Contexte de la thèse

1.1.1 Présentation de LocalSolver

Les problèmes d'optimisation mathématique qui se posent dans les entreprises sont souvent non linéaires, combinatoires, et de très grande taille. Les utilisateurs des outils d'aide à la décision et de recherche opérationnelle souhaitent cependant obtenir des solutions de bonne qualité à ces problèmes, c'est-à-dire respectant les contraintes posées par les utilisateurs (ou modélisateurs), et de qualité supérieure aux solutions qui pourraient éventuellement être construites à la main. De plus, ces solutions doivent être obtenues très rapidement : en quelques minutes, voire quelques secondes. Ce besoin d'obtenir des solutions de qualité en des temps d'exécution très courts a conduit à la création de LocalSolver, solveur de programmation mathématique global de nouvelle génération, développé par la société LocalSolver.

LocalSolver est un « solveur » : il suffit de lui fournir une définition mathématique du problème à résoudre, de façon purement déclarative, sous la forme d'un ensemble de contraintes à respecter et d'objectifs à minimiser ou à maximiser. Une fois cette modélisation faite, l'utilisateur délègue entièrement à LocalSolver la recherche de solutions et de bornes inférieures. Ce fonctionnement de type « model & run » nécessite la mise au point d'algorithmes puissants et génériques au sein du solveur, issus de différentes techniques de la recherche opérationnelle, à la fois exacts et heuristiques. La composante historique de LocalSolver, décrite dans [15] et [43], consiste en une recherche locale à voisinage variable [63, 67] qui lui permet de faire coopérer des transformations locales et rapides de la solution courante avec des modifications plus profondes utilisant des techniques algorithmiques exploitant la structure du problème. Cependant, des composantes exactes sont également intégrées à LocalSolver depuis plusieurs années (programmation linéaire, non linéaire et par contraintes).

Pour modéliser et résoudre des problèmes combinatoires riches, LocalSolver a introduit une modélisation à base d'ensembles et de listes. Ces structures de plus haut niveau que les traditionnelles variables entières et continues de la programmation linéaire mixte, également utilisées dans d'autres solveurs notamment de programmation par contraintes [44, 2, 42], permettent de modéliser de façon naturelle et compacte des problèmes comprenant des notions d'ordre, comme des problèmes de tournées de véhicules, ou des concepts ensemblistes, comme des problèmes de packing. Ces concepts ont permis de conjuguer une grande simplicité de modélisation avec une extrême rapidité de résolution. La généricité des objets introduits (ensembles et listes d'entiers) a permis en outre de ne pas limiter leur usage à des familles de problèmes restreintes. Ainsi, même si les listes sont la structure de choix pour modéliser les problèmes de tournées, elles permettent par exemple aussi de modéliser des problèmes de planification de lignes de production ou d'organisation d'ateliers, ou encore de conception de réseaux.

1.1.2 Sujet de la thèse

Définition de l'ordonnancement. Résoudre un problème d'ordonnancement consiste à organiser la réalisation de tâches dans le temps : déterminer leurs dates d'exécution et l'ordre dans lequel elles sont traitées sur l'ensemble des ressources disponibles. Ces problèmes sont caractérisés par deux catégories principales de contraintes. D'une part, les tâches sont souvent soumises à des contraintes temporelles : elles doivent être réalisées à l'intérieur de fenêtres de temps prédéterminées, et respecter des relations de précédence avec les autres tâches. Dans cette catégorie de contraintes temporelles, les tâches peuvent être préemptives (elles peuvent être interrompues puis reprises) ou non préemptives (elles doivent être exécutées en une fois). D'autre part, on trouve des contraintes portant sur la disponibilité des ressources requises. Par exemple, une ressource peut être renouvelable (elle peut de nouveau être utilisée après l'exécution d'une tâche) ou consommable (la quantité de ressource utilisée par une tâche n'est plus disponible après son exécution), ou encore disjonctive (elle ne peut alors exécuter qu'une seule tâche à la fois) ou cumulative (elle peut exécuter plusieurs tâches simultanément sous réserve du respect d'une capacité limitée). On trouve généralement des objectifs liés au temps (minimiser la durée totale de l'ordonnancement ou makespan, le nombre de tâches en retard, la somme des écarts à une date de fin cible pour chaque tâche...) ou à des coûts (minimisation du coût horaire d'utilisation des ressources, maximisation des revenus engendrés...). Du fait de la grande diversité des contraintes temporelles, des ressources, et des fonctions-objectif rencontrées, il existe une multitude de problèmes d'ordonnancement différents. Dans cette thèse, on se concentrera principalement sur les problèmes d'ordonnancement disjonctif, avec des tâches non préemptives.

Les problèmes d'ordonnancement sont parmi les problèmes d'optimisation combinatoires les plus étudiés dans la littérature, du fait des enjeux théoriques et applicatifs majeurs qu'ils recouvrent. Sur le premier point, nombre de problèmes d'ordonnancement sont généralement NP-difficiles au sens fort. Sur le deuxième point, on trouve des problèmes d'ordonnancement dans tous les secteurs de l'industrie et des services. Il est donc naturellement souhaitable de permettre aux utilisateurs de LocalSolver de modéliser et résoudre efficacement ce type de problèmes.

Objectifs de la thèse. Cette thèse s'articule autour de deux objectifs principaux. D'une part, on souhaite permettre aux utilisateurs de LocalSolver de modéliser simplement de nombreux problèmes d'ordonnancement disjonctif. D'autre part, le second objectif consiste à résoudre ces problèmes de façon efficace au moyen de modèles et d'algorithmes de recherche locale les plus génériques possibles.

Du point de vue de la modélisation, l'objectif est donc de proposer des formulations génériques au sein du formalisme de modélisation de LocalSolver, permettant d'exprimer facilement les notions de tâches, de relations de précédence et de contraintes de non-chevauchement, afin de pouvoir modéliser de façon simple et compacte la plupart des problèmes d'ordonnancement disjonctif. En effet, avant les travaux de la thèse, la modélisation des problèmes d'ordonnancement restait délicate, en particulier pour exprimer les contraintes de non-chevauchement ou de précédence entre certaines tâches. Il était souvent nécessaire de passer par une modélisation à base d'ordre de priorité, fonctionnant avec un algorithme externe calculant l'ordonnancement à partir de la liste de priorité. Une telle approche est généralement efficace, mais nécessite une certaine expertise pour sa mise en œuvre. Comme pour les listes utilisées dans les tournées, le formalisme ainsi élaboré n'est pas dédié à l'ordonnancement, mais permet de modéliser une grande variété de problèmes. Cette généricité du formalisme, nécessitant d'identifier les concepts mathématiques minimaux nécessaires pour modéliser les problèmes d'ordonnancement, est une des clés du paradigme choisi pour LocalSolver.

En termes de résolution, l'accent a été mis sur l'obtention rapide de très bonnes solutions et sur la généricité des algorithmes implémentés. En effet, LocalSolver étant basé sur un formalisme très générique, celui-ci ne « sait pas » précisément quel type de problème lui a été soumis. Les

algorithmes mis en œuvre doivent donc s'appliquer de façon très large, en se basant uniquement sur les équations définissant le modèle. En termes de recherche locale générique (qui est la composante de LocalSolver à laquelle cette thèse s'intéresse), il importe donc de trouver des opérateurs de voisinage qui travaillent sur une représentation adaptée de la solution, et qui bénéficient du mécanisme d'évaluation incrémentale de LocalSolver pour s'exécuter le plus rapidement possible. En outre, ces algorithmes sont conçus pour pouvoir interagir avec les autres algorithmes de LocalSolver. On souhaite pouvoir modéliser et résoudre des problèmes combinant des caractéristiques de problèmes d'ordonnancement avec d'autres types de variables et d'expressions, plutôt que de construire un « silo » d'ordonnancement indépendant au sein de LocalSolver. En effet, cette intégration au reste du formalisme est particulièrement appréciée par les utilisateurs des variables de listes et d'ensembles, par exemple pour les problèmes de tournées de véhicules. Il est donc naturellement souhaitable d'en faire de même pour les éléments de modélisation introduits à l'occasion de cette thèse.

1.2 Introduction aux aspects techniques de LocalSolver

1.2.1 Formalisme de modélisation de LocalSolver

Le formalisme de modélisation de LocalSolver est différent de celui des solveurs de programmation linéaire classiques, et comporte plusieurs caractéristiques propres. Par exemple, il est possible de définir des expressions intermédiaires, qui se distinguent des variables de décision. Ces expressions sont utilisées pour modéliser des quantités pouvant être totalement déduites à partir des valeurs des variables de décision. Elles sont définies de façon directe, sans ajout de contraintes. Leur utilisation facilite ainsi l'écriture du modèle. Afin d'offrir davantage de possibilités de modélisation, ces expressions peuvent de plus être définies au moyen d'opérateurs non linéaires variés (min, max, produit, puissance, racine carrée, modulo, valeur absolue, partie entière, fonctions trigonométriques, condition ternaire, « *at* » sur un tableau, relations logiques...).

LocalSolver peut être utilisé via des API (Python, Java, C++, C#), mais il possède également son propre langage de modélisation, nommé LSP pour *LocalSolver Programming language*, permettant de définir un modèle d'optimisation de façon purement déclarative. Dans la suite de la thèse, tous les exemples de modélisation seront écrits en LSP.

Variables de décision ensemblistes. Une caractéristique majeure et innovante du formalisme de modélisation de LocalSolver est la possibilité d'écrire des modèles ensemblistes. En effet, en plus des variables de décision numériques classiques (booléens, entiers, réels), LocalSolver offre deux types de variables de décision ensemblistes : les sets et les listes. Une variable de set de domaine n, définie dans le modèle par l'écriture `set(n)`, représente un sous-ensemble (non-ordonné) de $\{0, ..., n-1\}$ [1]. Dans toute solution, la valeur d'une variable de set est donc un ensemble d'entiers, et non un simple nombre. Les variables de sets sont naturellement utilisées pour modéliser les problèmes dans lesquels on crée des regroupements d'éléments. Par exemple, dans le problème du Bin Packing, chaque conteneur est représenté par une variable de set, dont la valeur est égale à l'ensemble des éléments qu'il contient. Ces variables sont également utilisées dans les problèmes de clustering : chaque set représente alors une catégorie.

De façon similaire, une variable de liste de domaine n, définie dans le modèle par l'écriture `list(n)`, représente une permutation d'un sous-ensemble de $\{0, ..., n-1\}$. Les variables de listes sont alors naturellement utilisées pour définir des ordres. Par exemple, les problèmes de tournées de véhicules sont modélisés à partir de variables de listes : chaque liste est associée à un véhicule, et représente l'ordre des points visités par ce véhicule. Les variables de listes jouent également un rôle

1. Le domaine de définition d'une telle variable de set est donc l'ensemble des parties de $\{0, ..., n-1\}$. Cependant, ce domaine de définition étant totalement déterminé par l'entier n, on emploiera par abus de langage le terme « domaine » dans toute la thèse pour désigner le nombre maximal d'éléments que peut contenir la variable.

crucial dans la modélisation des problèmes d'ordonnancement, puisqu'elles peuvent être utilisées pour représenter l'ordre des tâches affectées à une ressource disjonctive.

Il existe plusieurs opérateurs spécifiques aux variables de sets et de listes. Les opérateurs « partition », « disjoint » et « cover » servent à définir des contraintes globales sur des familles de variables ensemblistes de même type et de même domaine. Si `mySetVariables` est un ensemble de variables de sets (ou de listes) de domaine n, l'écriture `constraint partition(mySetVariables)` (resp. `constraint disjoint(mySetVariables)`, resp. `constraint cover(mySetVariables)`) indique que chaque élément de $\{0, ..., n-1\}$ doit être présent exactement une fois (resp. au plus une fois, resp. au moins une fois) dans l'ensemble des variables de `mySetVariables`. D'autres opérateurs permettent d'obtenir des informations sur le contenu des différents sets et listes : l'opérateur `count` donne le nombre d'éléments présents dans la variable, `contains` permet de tester l'appartenance d'un élément à un set ou une liste, `indexOf` donne la position d'un élément dans une liste, `find` permet de retrouver la variable de set ou liste contenant un élément donné.

Opérateurs variadiques et lambda-fonctions. LocalSolver offre la possibilité d'appliquer des opérateurs variadiques n-aires (somme, min, max, opérateurs logiques) avec un nombre de termes dynamique. Cela permet par exemple de calculer une somme dont le nombre de termes n'est pas constant, et dépend de la valeur d'autres expressions, comme une variable de set ou de liste. Ces opérateurs variadiques sont typiquement utilisés pour calculer une somme sur les éléments d'une variable ensembliste. Par exemple, dans le problème du Bin Packing, on peut calculer la somme des poids des éléments i contenus dans un bin k modélisé par une variable de set `bin[k]` de la façon suivante :

```
binWeight[k] <- sum(bin[k], i => elementWeight[i]);
```

De même, dans le problème du Voyageur de Commerce, si l'on note n le nombre de villes et `order` la variable de liste représentant l'ordre dans lequel les différentes villes sont visitées, la distance parcourue peut être calculée à l'aide d'une somme variadique :

```
totalDistance <- sum(0..n-2, i => distance[order[i]][order[i+1]]) + distance[order[n-1]][0];
```

Comme on peut le voir sur ces deux exemples, l'expression des termes à sommer est donnée par une lambda-fonction (écrite `i => elementWeight[i]` ci-dessus pour le problème du Bin Packing par exemple). Comme expliqué dans le Chapitre 2, cette notion est cruciale dans la modélisation des problèmes d'ordonnancement, notamment pour écrire les contraintes de non-chevauchement entre les tâches.

Remarque 1.1 (Notations). Comme expliqué à la Section précédente, la modélisation des problèmes d'ordonnancement disjonctif avec LocalSolver est un des enjeux majeurs de cette thèse. Dans les différents Chapitres constituant la thèse, on présentera donc plusieurs exemples en code LSP. On présentera ainsi les modèles LSP complets de certains problèmes, mais aussi des expressions génériques permettant de modéliser certaines contraintes rencontrées dans de nombreux problèmes d'ordonnancement. Pour simplifier les notations et par souci de clarté, on utilisera à la fois une police mathématique pour désigner des objets tels que les tâches et les ressources, et une police monotype pour désigner les variables de décision et constantes intervenant dans le modèle. Par exemple, on pourra s'intéresser à une ressource disjonctive notée r, et à une variable de liste notée `order[r]` représentant l'ordre des tâches qui lui sont affectées. Une tâche t s'exécutant sur la ressource r sera alors représentée par l'élément `t` dans la liste `order[r]`. Autre exemple, on pourra également considérer le temps de transition `setup[r][prev][next]` à respecter entre la fin d'une tâche t_{prev} et le début de la tâche suivante t_{next} sur une ressource r.

1.2.2 Structures internes de LocalSolver

On décrit ici les principaux outils algorithmiques et mathématiques utilisés dans les différentes composantes de LocalSolver. On se concentre en particulier sur la composante de recherche locale [43, 15], dans laquelle s'inscrivent les travaux de cette thèse. La recherche locale est une technique d'amélioration itérative d'une solution initiale, mettant en œuvre des opérateurs de voisinage, algorithmes élémentaires et généralement très rapides permettant de trouver la meilleure solution voisine d'une solution selon un voisinage prédéfini (par exemple, la permutation de deux tâches consécutives dans la séquence des tâches affectées à une ressource disjonctive). Selon les types et la variété des voisinages considérés, on trouve différentes méthodes de recherche locale. En particulier, la recherche à voisinage variable (Variable Neighborhood Search ou VNS) [63, 67], sur laquelle est basé LocalSolver, alterne différents types de voisinages dans un but d'intensification de la recherche autour d'une bonne solution, et de diversification de la recherche pour s'extraire des minima locaux. Lorsque le voisinage considéré est de taille exponentielle, on parle de recherche locale à voisinage étendu (Large Neighborhood Search ou LNS). Dans ce cas, la recherche de la meilleure solution dans le voisinage est effectuée au moyen d'un algorithme polynomial, ou du moins très efficace, généralement adapté à la structure du sous-problème définissant le voisinage.

Les algorithmes de recherche locale traditionnels sont souvent dédiés à la résolution d'un problème particulier. Les voisinages qu'ils explorent sont alors basés sur la structure métier du problème. Une telle approche ne convenant pas dans un solveur générique, ce principe a été industrialisé dans la composante historique de LocalSolver, aboutissant ainsi à une recherche locale générique, avec des voisinages basés uniquement sur la structure mathématique du modèle, et applicable à des problèmes quelconques.

Graphe d'évaluation. Les équations définissant le modèle sont représentées à l'intérieur de LocalSolver au moyen d'un graphe d'évaluation. Le graphe d'évaluation est un graphe acyclique orienté. Les feuilles correspondent aux variables de décision et aux constantes. Les racines du graphe sont les contraintes et objectifs du problème, et les nœuds internes correspondent aux autres expressions intermédiaires.

Exemple 1.1 (Graphe d'évaluation d'une contrainte de précédence). On considère une contrainte de précédence entre deux tâches t_1 et t_2, écrite en LSP sous la forme suivante :

```
1 constraint start[t2] >= start[t1] + duration[t1],
```

Le sous-graphe d'évaluation correspondant à cette contrainte est représenté sur la Figure 1.1 : les feuilles sont les variables de décision (de type « *integer* ») représentant les dates de début des deux tâches, et les nœuds internes (de type « *sum* » et « *geq* ») sont des expressions intermédiaires.

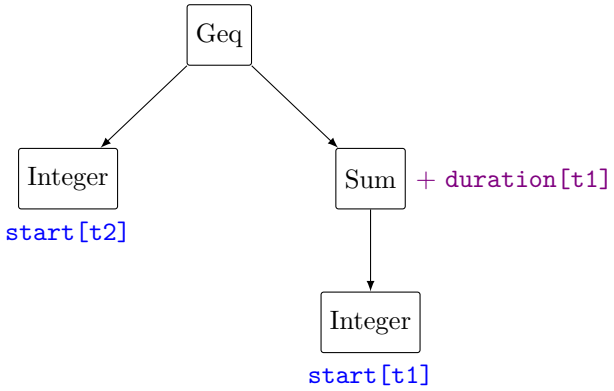


FIGURE 1.1 – Exemple de graphe d'évaluation

A chaque itération de la recherche locale, on utilise le graphe d'évaluation pour évaluer les contraintes et objectifs de façon incrémentale. En effet, puisque la plupart des voisinages utilisés pour passer d'une solution à une autre ne modifient que peu de variables, il est intéressant de ne pas réévaluer tous les nœuds du graphe, mais seulement ceux ayant été modifiés. L'incrémentalité permet ainsi d'accélérer largement le processus d'évaluation des solutions, comme le montre l'exemple suivant. On considère n variables numériques x_i avec $0 \leq i < n$, et un objectif égal à la somme de tous les $x_i x_j$, $0 \leq i, j < n$. L'objectif est donc un nœud de type « somme », dont descendent n^2 enfants de type « produit ». On suppose que l'un des x_i a été modifié. Sans incrémentalité, la complexité d'évaluation de l'objectif est alors $O(n^2)$. Celle-ci peut cependant être réduite à $O(n)$ si l'on ne visite que les nœuds du graphe ayant été impactés par le changement. De même, si l'on considère un objectif égal à la somme des x_i, $0 \leq i < n$, et si l'une de ces variables a été modifiée, alors l'évaluation de l'objectif se fait avec une complexité de $O(1)$, contre $O(n)$ sans incrémentalité. L'incrémentalité permet également d'arrêter la propagation avant d'atteindre les racines du graphe lorsqu'on rencontre un nœud dont la valeur reste inchangée (min ou max par exemple). Chaque nœud du graphe d'évaluation est donc propagé de façon incrémentale, selon un algorithme efficace propre au type du nœud.

On se sert également du graphe d'évaluation pour détecter certaines structures particulières dans les contraintes et objectifs du problème. On peut par exemple détecter des sous-graphes révélant des contraintes de packing, ou une fonction objectif de minimisation d'une distance parcourue, typique des problèmes de tournées de véhicules, ou encore une structure de ressources disjonctives, comme le montre le Chapitre 2. La détection de ces structures dans le graphe d'évaluation donne une information précieuse au solveur, qu'il peut alors exploiter au cours de la résolution.

Voisinages et mouvements. Une caractéristique de la recherche locale de LocalSolver est sa stratégie d'exploration des voisinages, appelée recherche randomisée. A chaque itération de la recherche locale, on choisit un voisinage à explorer. Cette exploration ne se fait cependant pas de façon exhaustive : on choisit une transformation dans le voisinage considéré, de façon aléatoire ou après une exploration très limitée suivant le voisinage choisi, et on l'applique à la solution courante. Cette procédure est appelée « mouvement » de la recherche locale, ou encore « transformation locale », dans le reste du document.

Chaque type de variables (booléens, entiers, réels, sets, listes) est associé à un ensemble de mouvements spécifiques. Ces mouvements peuvent correspondre à des voisinages restreints aléatoires très simples, comme par exemple une inversion de valeur sur un booléen, un échange entre les valeurs de deux entiers, un décalage de la valeur d'un réel, l'insertion d'un élément dans un set, ou encore l'échange de deux indices d'une liste. Plusieurs mouvements sur les variables numériques sont basés sur le principe des chaînes d'éjection, et modifient les variables associées à une même contrainte. Enfin, certains mouvements sont dédiés à certaines classes de problèmes, et sont ainsi appliqués aux modèles dans lesquels certaines structures particulières ont pu être détectées dans le graphe d'évaluation. Par exemple, on trouve des mouvements sur les variables numériques associés à des contraintes de couverture ou de sac-à-dos, des mouvements sur les sets basés sur une structure de packing ou de clustering, ou encore des mouvements sur les listes particulièrement adaptés aux problèmes de tournées de véhicules.

Comme expliqué plus haut, l'exploration des voisinages associés à ces mouvements n'est pas exhaustive, mais aléatoire. Afin d'augmenter la probabilité de réaliser un mouvement améliorant, on peut utiliser un algorithme de ciblage des variables à modifier par le mouvement choisi. On utilise ainsi la structure du graphe d'évaluation pour cibler des modifications de variables susceptibles d'amener à des améliorations, en explorant par exemple les nœuds impliqués dans des contraintes violées, ou associés à un sens de variation avantageux dans un objectif.

Heuristiques. Une recherche locale à voisinages variables comme celle de LocalSolver comporte deux heuristiques principales, dites de sélection et d'acceptation. L'heuristique de sélection est

appelée au début de chaque itération, et détermine le voisinage à explorer. Dans LocalSolver, cette heuristique est basée sur des méthodes d'apprentissage statistique, et utilise l'historique des résultats des mouvements aux itérations précédentes (taux d'acceptation ou d'infaisabilité) pour choisir un voisinage prometteur. Afin de garantir une diversification importante, l'heuristique de sélection s'assure également que chaque voisinage est régulièrement exploré.

L'heuristique d'acceptation intervient quant à elle à la fin de chaque itération. Son rôle est de déterminer si les changements effectués lors du mouvement que l'on vient de réaliser doivent être conservés, ou s'ils doivent être annulés pour restaurer la solution précédente. Plusieurs heuristiques d'acceptation différentes sont implémentées dans LocalSolver, et sont appelées lors de différentes phases de la recherche. Elles consistent généralement à accepter un mouvement si celui-ci ne dégrade pas la qualité de la solution. La qualité des solutions est évaluée grâce à une mesure de faisabilité, dont la valeur dépend du nombre de violations de contraintes et de leur amplitude, puis en fonction de la valeur des différents objectifs. Cependant, afin d'éviter que la recherche locale ne reste bloquée dans un minimum local, LocalSolver utilise également des heuristiques d'acceptation autorisant un certain nombre de dégradations si aucune nouvelle amélioration n'a été trouvée pendant un certain nombre d'itérations.

Optimisation globale. En plus de la recherche locale, LocalSolver intègre des composantes exactes de programmation linéaire et non linéaire, comportant tous les composants standards d'un solveur MIP/MINLP [24]. Parmi les algorithmes implémentés, on trouve un code de reformulation du modèle (linéarisation, transformation des opérateurs variadiques), un presolve dédié, des solveurs numériques pour résoudre des relaxations linéaires et convexes (simplexe, lagrangien augmenté, point intérieur), un branch and bound couplé à une propagation et à des coupes, ou encore des heuristiques primales. Ces composantes exactes complètent la composante historique de recherche locale de LocalSolver, en permettant au solveur de calculer des bornes inférieures sur de nombreux problèmes. Les différentes composantes du solveur collaborent de plus en échangeant des solutions : une solution trouvée au cours d'une recherche arborescente peut être rapidement améliorée par la composante de recherche locale, et inversement, une solution réalisable de bonne qualité trouvée par recherche locale permet de couper efficacement des pans entiers d'un arbre de recherche.

Le travail de cette thèse s'inscrit cependant exclusivement dans la composante de recherche locale de LocalSolver, et les composantes exactes ne seront donc plus mentionnées par la suite.

1.3 Revue de littérature en ordonnancement

1.3.1 Problèmes d'ordonnancement classiques

Dans cette Section, on présente plusieurs problèmes d'ordonnancement disjonctif classiques, issus de domaines allant de l'ordonnancement d'atelier à la planification de production, que l'on étudiera dans les Chapitres suivants.

Problème du Flow Shop. Le problème du Flow Shop [90] est décrit de la façon suivante. On considère un ensemble de n jobs devant être exécutés sur un ensemble de k machines. Chaque job est divisé en k activités ordonnées, chacune devant être réalisée sur une machine particulière : $\forall j \in \{0, ..., n-1\}$, $\forall m \in \{0, ..., k-1\}$, la m-ième tâche du job j est réalisée sur la machine m. Les machines sont disjonctives (elles ne peuvent traiter qu'un job à la fois), mais elles peuvent fonctionner en parallèle. Dans la variante considérée (Flow Shop dit « de permutation »), les jobs doivent être traités par toutes les machines dans le même ordre. L'objectif du problème consiste à trouver la permutation sur les jobs permettant de minimiser le makespan, soit la date de fin du dernier job sur la dernière machine.

Le problème du Flow Shop est un problème d'ordonnancement disjonctif simple, que l'on peut modéliser avec LocalSolver en utilisant une unique variable de décision, de type liste. En effet, la seule décision à prendre correspond à l'ordre dans lequel les jobs seront traités par les différentes machines, ce qui correspond parfaitement au cas d'usage typique d'une variable de liste. En plus d'une modélisation concise, l'utilisation des variables de listes permet également à LocalSolver d'obtenir de très bonnes performances sur ce problème, y compris dans les versions du solveur antérieures aux travaux de la thèse. Il constitue donc un bon exemple d'utilisation des variables de listes en ordonnancement. Le code LSP complet de la modélisation LocalSolver du problème du Flow Shop est donné dans le Modèle 1.1.

```
1  // Ordre des jobs (permutation).
2  jobs <- list(nbJobs);
3
4  // Tous les jobs doivent être effectués.
5  constraint count(jobs) == nbJobs;
6
7  // Dates de fin sur la première machine (0) :
8  // fin du job en position j = fin du job en position j-1 + durée du job en position j.
9  end[0] <- array(0..nbJobs-1, (j, prev) => prev + processingTime[0][jobs[j]]);
10
11 // Dates de fin sur les autres machines : le job en position j sur la machine m peut commencer si :
12 //  - le job en position j-1 sur la machine m est terminé ;
13 //  - le job en position j sur la machine m-1 est terminé.
14 for[m in 1..nbMachines-1] {
15     end[m] <- array(0..nbJobs-1, (j, prev) => max(prev, end[m-1][j]) + processingTime[m][jobs[j]]);
16 }
17
18 // Minimisation du makespan : date de fin du dernier job sur la dernière machine.
19 makespan <- end[nbMachines-1][nbJobs-1];
20 minimize makespan;
```

Modèle 1.1 – Code LSP pour le problème du Flow Shop

On commence par définir la variable de décision (ligne 2) : une liste de domaine n (noté `nbJobs` dans le code). La valeur de cette liste correspond à l'ordre dans lequel seront traités les jobs sur chacune des machines. Tous les jobs devant être traités, on contraint la taille de la liste à être égale au nombre de jobs (ligne 5).

A partir de la valeur de la liste, on peut calculer les dates de fin de chacune des tâches, en commençant par les tâches affectées à la machine 0 (à traiter en premier). Sur cette machine, chaque tâche commence dès la fin de la tâche précédente. Le calcul des dates de fin sur la première machine (ligne 9) se fait au moyen d'une fonction variadique à deux arguments : `j`, le numéro du job traité, et `prev`, la valeur de la case précédente dans le tableau (ou 0 lorsqu'on calcule la valeur de la première case). Ainsi, la date de fin du job en position j sur la première machine est égal à la somme de la date de fin de la tâche en position $j-1$ sur cette machine (`prev`) et de sa durée sur cette machine (`processingTime[0][jobs[j]]`). On passe ensuite au calcul des dates de fin sur les autres machines (lignes 14 à 16). La date de début du job en position j sur une machine m quelconque est égale au maximum entre la date de fin du job en position $j-1$ sur la machine m et la date de fin du job en position j sur la machine $m-1$.

Enfin, on calcule le makespan, égal à la date de fin du dernier job exécuté sur la dernière machine (ligne 19), et on indique au solveur qu'il s'agit d'un objectif à minimiser (ligne 20).

Problème du Job Shop (et variantes). Le problème du Job Shop [92] présente des similitudes avec celui du Flow Shop, mais est plus complexe. Là encore, on considère un ensemble de n jobs, divisés en k tâches chacun, à réaliser sur un ensemble de k machines. Chaque tâche est toujours affectée à une machine particulière, mais l'ordre des machines n'est pas le même pour tous les jobs. Les machines sont toujours disjonctives, et peuvent toujours fonctionner en parallèle, mais

elles peuvent traiter les jobs dans un ordre différent. L'objectif du problème est également la minimisation du makespan.

Du fait de ces différences, le problème du Job Shop est plus difficile à modéliser que celui du Flow Shop. En effet, si l'on essaie de modéliser le problème du Job Shop en suivant le principe du modèle présenté au paragraphe précédent pour le problème du Flow Shop, c'est-à-dire en n'utilisant que des variables de listes représentant l'ordre des tâches sur chaque machine, la date de fin d'une tâche quelconque ne peut plus être exprimée aussi simplement que dans le problème du Flow Shop. En effet, dans le problème du Flow Shop, l'ordre des machines est le même pour chaque job. Lorsqu'on calcule la date de fin de la tâche en position j sur une machine m, on a donc accès à la date de fin de la tâche précédente dans le job (puisqu'il s'agit de la tâche en position j sur la machine $m-1$). Au contraire, dans le problème du Job Shop, on n'a aucune garantie de ce type, et on ne peut donc pas exprimer les dates de fin des tâches de la même façon. Trouver une modélisation simple et concise pour un problème comme celui du Job Shop constitue donc un des enjeux de cette thèse, exploré en détail dans le Chapitre 2.

Dans un souci de robustesse et de généralisation, plusieurs variantes du problème du Job Shop sont étudiées dans cette thèse (voir Chapitres 2 et 3). On considère ainsi le problème du Job Shop flexible [30], différant du Job Shop par le fait que chaque tâche, au lieu d'être affectée à une machine particulière, peut être affectée à une machine quelconque parmi un sous-ensemble de machines compatibles, avec des durées potentiellement différentes sur les différentes machines. Le degré de « flexibilité » peut varier selon le benchmark étudié : on peut trouver des instances dans lesquelles quelques tâches seulement peuvent être affectées à quelques machines seulement, ou des instances dans lesquelles chaque tâche est compatible avec chaque machine.

Une autre variante du problème du Job Shop est le problème de l'Open Shop [91]. Dans ce problème, chaque tâche est toujours affectée à une machine particulière. La différence avec le problème du Job Shop est que les tâches de chaque job peuvent être effectuées dans un ordre quelconque.

Enfin, on peut considérer des variantes des problèmes du Job Shop, du Job Shop flexible ou de l'Open Shop dans lesquelles on introduit des temps de transition à respecter entre deux tâches consécutives sur une même machine.

Problème du Unit Commitment simplifié. On s'intéresse maintenant au problème du Unit Commitment [23], dans une version simplifiée purement combinatoire. On considère un ensemble de n usines, dont on doit décider du fonctionnement sur un ensemble de H pas de temps. A chaque pas de temps, chaque usine peut être en fonctionnement, ou éteinte. Les plages de fonctionnement et de non-fonctionnement (fenêtres de temps entre deux plages de fonctionnement consécutives, durant lesquelles l'usine est éteinte) ont des durées minimales sur chaque usine. A chaque pas de temps, chaque usine en fonctionnement a un certain niveau de production, et la production totale de l'ensemble des usines en fonctionnement doit être supérieure à une demande. Les usines ont des coûts de fonctionnement différents : à la fois un coût de fonctionnement à chaque pas de temps, et un coût de démarrage au début de chaque plage de fonctionnement. Suivant le nombre de pas de temps séparant deux plages de fonctionnement, on peut considérer que l'usine effectue un démarrage « à froid » ou « à chaud », associés à des coûts de démarrage différents. Le but du problème est de minimiser le coût total.

La description du problème du Unit Commitment est très différente de celle du problème du Job Shop. Pourtant, on montre dans le Chapitre 2 que les aspects propres à l'ordonnancement des deux problèmes peuvent être modélisés de façon similaire, en considérant chaque plage de production dans le problème du Unit Commitment comme une tâche de durée variable.

Autres problèmes. D'autres types de problèmes d'ordonnancement seront également étudiés dans la suite de la thèse. Par exemple, on parlera dans le Chapitre 2 du problème de l'Aircraft Landing [80], dont le but est de choisir l'ordre d'atterrissage d'un ensemble d'avions, en respectant

un temps de séparation entre chaque paire d'avions consécutifs, et en minimisant les coûts liés à des dates d'atterrissage en avance ou en retard par rapport à des dates cibles. Par ailleurs, on s'intéressera dans le Chapitre 4 au problème de l'Assembly Line Balancing [72], consistant à répartir un ensemble de tâches dans un minimum de stations de travail, en respectant des relations de précédence entre les tâches et une durée totale maximale dans chaque station de travail.

1.3.2 État de l'art

On s'intéresse dans cette partie aux méthodes de résolution classiques, exactes ou heuristiques, utilisées pour résoudre les problèmes d'ordonnancement.

Programmation par contraintes. La programmation par contraintes [68, 83] est un paradigme de résolution exact, utilisé pour résoudre des problèmes combinatoires. Elle repose sur l'utilisation active des contraintes du problème, qui sont propagées afin de réduire l'espace des solutions à explorer. La programmation par contraintes est souvent utilisée pour résoudre les problèmes d'ordonnancement. En effet, une des forces de cette technique réside dans la variété des contraintes, notamment globales, qu'il est possible de définir, et dans la façon dont chacune peut être propagée. Par exemple, si l'on considère les contraintes de non-chevauchement des tâches affectées à une même machine, auxquelles cette thèse s'intéresse particulièrement, on trouve dans la littérature (par exemple [10]) différents algorithmes efficaces pour la propagation de ces contraintes, permettant de resserrer les bornes des variables entières représentant les dates de début des tâches. Une première méthode pour propager ces contraintes consiste à envisager l'ordonnancement comme un emploi du temps : au plus une tâche peut être exécutée à chaque pas de temps. Une autre façon de réduire le domaine des variables associées aux dates de début des tâches consiste à considérer les tâches deux à deux (la tâche t doit s'exécuter avant la tâche t', ou inversement). On peut également citer l'exemple de l'algorithme « *edge finding* », qui comprend à la fois une méthode de branchement et de resserrage des bornes, afin de déterminer si certaines tâches peuvent, doivent, ou ne peuvent pas s'exécuter en première ou en dernière position au sein d'un sous-ensemble de tâches. De façon similaire, de nombreuses autres contraintes caractéristiques des problèmes d'ordonnancement (ressource cumulatives, tâches préemptives, temps de transition, etc.) peuvent être propagées grâce à des algorithmes de filtrage efficaces.

Ces techniques de programmation par contraintes sont utilisées pour résoudre de nombreux problèmes d'ordonnancement [11]. Elles permettent par exemple de résoudre des problèmes d'ordonnancement d'atelier, comme dans [13] pour résoudre le problème du Job Shop, ou dans [8], où les auteurs s'intéressent à une version du problème avec temps de transition, ou également dans [48] pour résoudre le problème de l'Open Shop. Ces méthodes sont également utilisées en ordonnancement cumulatif, comme par exemple pour résoudre le Resource-Constrained Project Scheduling Problem dans [58]. On retrouve également ces techniques de programmation par contraintes dans différents solveurs libres [52, 79, 86, 51] et commerciaux [61].

Approches SAT. Les problèmes d'ordonnancement peuvent également être abordés par des approches de type SAT. Ce type d'approche consiste à reformuler le problème en utilisant des variables booléennes, et un ensemble de clauses à satisfaire portant sur ces variables. La résolution est souvent abordée par des algorithmes d'apprentissage de clauses par conflits [69]. Hybridées avec des techniques de programmation par contraintes, ces algorithmes permettent d'obtenir de très bonnes performances sur les problèmes d'ordonnancement. De telles approches sont par exemple étudiées sur des problèmes d'ordonnancement disjonctif dans [57] et [89], mais aussi pour des problèmes cumulatifs comme le Resource-Constrained Project Scheduling Problem dans [33] et [87]. Il existe également des solveurs hybrides implémentant des techniques de génération de clauses retardée et obtenant de bonnes performances pour les problèmes d'ordonnancement [31] : les clauses

utilisées par le solveur pour diriger la recherche ne sont pas toutes définies au départ, mais générées à la volée en fonction des résultats de la propagation de contraintes.

Programmation linéaire en nombres entiers. La programmation linéaire en nombres entiers (PLNE) est un paradigme de résolution exact, dans lequel on ne considère que des variables entières ou booléennes, sur lesquelles sont exprimés des contraintes et un objectif linéaires, et dont la résolution repose généralement sur des algorithmes de type *Branch and Bound* ou *Branch and Cut*. Cette technique est souvent utilisée pour résoudre des problèmes d'ordonnancement, comme par exemple dans [41]. Elle a cependant plusieurs inconvénients : on remarque d'une part que la modélisation de certaines contraintes sous forme linéaire peut être fastidieuse, et d'autre part que la complexité de résolution et la mauvaise qualité des relaxations continues ne permettent en général pas de résoudre des problèmes de grande taille. Néanmoins, les approches de PLNE permettent, au prix de temps de calcul importants, d'obtenir de bonnes bornes inférieures (par exemple [7] pour le Resource-Constrained Project Scheduling Problem). Dans certains cas, la PLNE peut être compétitive face à la programmation par contraintes, notamment pour les problèmes préemptifs [81]. Enfin, les matheuristiques consistent en des approches de recherche locale à voisinage étendu (voir paragraphe suivant) dans lesquelles un sous-problème est résolu par PLNE, ce qui permet de contrôler l'explosion des temps de résolution [50, 29, 54].

Recherche locale dédiée et métaheuristiques. La recherche locale, notamment dans ses variantes VNS et LNS (voir Section 1.2.2), peut être appliquée aux problèmes d'ordonnancement [94, 70, 71, 73, 47, 56, 74, 96]. Ces méthodes permettent généralement d'obtenir les meilleures solutions connues sur des problèmes difficiles, par exemple sur des problèmes d'ordonnancement d'atelier tels que celui du Job Shop [95], ou des problèmes d'ordonnancement de projets tels que le Resource-Constrained Project Scheduling Problem (RCPSP) [1]. Néanmoins, ces approches reposent généralement sur des structures spécifiques au problème d'ordonnancement résolu qu'il est difficile de généraliser, notamment si l'on n'utilise pas un objectif standard comme le makespan. Par exemple, pour les problèmes cumulatifs de type RCPSP, les meilleurs résultats sont obtenus en utilisant des voisinages basés sur des permutations de tâches au sein de listes de priorité [56] (voir paragraphe suivant), alors que pour le problème du Job-Shop et certaines de ses variantes dites régulières, les algorithmes de recherche locale travaillent sur des structures d'ordres partiels issues du modèle du graphe disjonctif [70, 71, 47, 65, 88]. Ces deux structures classiques de représentations d'ordonnancement ne permettent pas de traiter le cas de problèmes plus généraux comme les problèmes d'avance/retard pour lesquels il n'est plus dominant de considérer les ordonnancements au plus tôt, et où une représentation centrée sur les dates de début des tâches peut s'avérer nécessaire [9]. On peut également citer l'exemple des problèmes avec temps de transfert (temps de préparation ou de transport d'unités de ressources entre tâches) pour les ressources cumulatives, qui nécessitent des modèles de flots d'unités de ressources [82].

La recherche locale est une des composantes des métaheuristiques, qui proposent des mécanismes généraux faisant appel à des heuristiques dédiées comme les algorithmes de listes présentés ci-dessous (recuit simulé, méthodes taboues, GRASP, scatter search, algorithmes génétiques, algorithmes par colonies de fourmis, hyper-heuristiques). Les métaheuristiques sont fréquemment utilisées en ordonnancement, et fournissent de bons résultats [56, 93]. Elles peuvent être intégrées au sein d'approches hybrides [76].

Algorithmes de listes. De nombreux problèmes d'ordonnancement peuvent être résolus avec des algorithmes de listes. Le principe général d'un tel algorithme est le suivant. On définit une liste de priorité L sur les tâches, correspondant à une permutation des tâches. Tant que la liste L est non vide, on considère le plus petit instant s auquel au moins une tâche de L est exécutable, en tenant compte des contraintes de précédence, de la disponibilité des ressources, des fenêtres de temps des tâches, etc. On note t la tâche disponible à la date s ayant la plus haute priorité dans la liste L :

on décide d'ordonnancer la tâche t à partir de la date s, et on la retire de la liste L. Suivant le problème considéré (nombre et nature des ressources, des contraintes de précédence, de l'objectif, etc.), il existe de nombreuses variantes d'algorithmes de listes, sur lesquels il est parfois possible de prouver l'existence d'une liste conduisant à un ordonnancement optimal. Comme évoqué au paragraphe précédent, ces algorithmes de listes sont fréquemment utilisés au sein de recherches locales : on effectue des transformations élémentaires sur la liste de priorité afin d'obtenir des ordonnancements de meilleure qualité.

Cette technique de résolution peut être exploitée pour de nombreux problèmes d'ordonnancement classiques, comme celui du Job Shop en ordonnancement d'atelier, avec la règle de Giffler et Thompson [45]. On peut également citer les algorithmes « *parallel* » et « *serial schedule generation scheme* » (pSGS et sSGS), présentés notamment dans [55], pour le Resource-Constrained Project Scheduling Problem.

Autres approches. La littérature est riche d'approches de type *Branch and Bound* dédiées, c'est-à-dire ne faisant pas appel à des solveurs de programmation linéaire, de programmation par contraintes, ou SAT, mais à des branchements, des ajustements et des calculs de bornes spécifiques. Leur but est de résoudre exactement les problèmes de la plus grande taille possible. Il existe des algorithmes célèbres, comme le *Branch and Bound* de Carlier [27] pour le problème à une machine, celui de Carlier et Pinson ayant permis de résoudre pour la première fois l'instance FT10 du problème du Job Shop [28], ou encore celui de Demeulemeester et Herroelen, resté pendant des années la meilleure méthode exacte pour le Resource-Constrained Project Scheduling Problem [36]. Néanmoins, ces approches souffrent de leur difficulté à être étendues à d'autres problèmes. Elles ont cependant jeté les bases de principes qui ont ensuite été intégrés dans des solveurs plus génériques. Par exemple, on peut citer les sélections immédiates de Carlier et Pinson [28], à la base des algorithmes de propagation de contraintes pour l'ordonnancement disjonctif, ou les « *cutsets* » de Demeulemeester et Herroelen, précurseurs des « *no good* » intégrés dans les solveurs à base de génération de clauses retardée [36].

Parmi les approches ne rentrant pas dans la classification de ce Chapitre, on trouve également les algorithmes basés sur la programmation dynamique. La programmation dynamique consiste à résoudre un problème en le décomposant en sous-problèmes : on résout successivement chacun des sous-problèmes, du plus petit au plus grand, en utilisant les résultats calculés précédemment. Pour certains problèmes d'ordonnancement relativement simples, des algorithmes de programmation dynamique efficaces existent. Pour des problèmes plus complexes, l'espace d'états explose mais des approches par relaxation d'espaces d'états permettent d'avoir de bonnes bornes inférieures [35, 34]. Un modèle de programmation dynamique peut être transformé en diagramme de décision dont la relaxation ou la restriction peut donner des bornes inférieures et supérieures de qualité pour une variété de problèmes d'ordonnancement [32].

Pour finir, les techniques de machine learning sont désormais utilisées pour apprendre des politiques d'ordonnancement efficaces, au sein de méthodes de recherche arborescente ou d'heuristiques, via des approches d'apprentissage par renforcement ou par réseaux de neurones [5, 75].

1.4 Plan de la thèse

Le Chapitre 2 s'intéresse à la conception d'une modélisation simple et concise des problèmes d'ordonnancement disjonctif, comme les différentes variantes du problème du Job Shop, ou le problème du Unit Commitment, décrits dans la Section 1.3.1. On se concentre en particulier sur les contraintes de précédence et non-chevauchement sur les tâches, typiques de ce type de problèmes, et jusqu'alors difficiles à exprimer efficacement avec LocalSolver. On montre que l'utilisation combinée de variables entières et de listes, contrôlant respectivement les dates de début et l'ordre des tâches, permet de répondre à cette problématique de modélisation. Afin d'obtenir de bonnes

performances sur les modèles exploitant cette nouvelle façon d'écrire les contraintes d'ordonnancement, la recherche locale doit être enrichie de nouveaux mouvements. On présente alors ces mouvements, basés sur la détection de contraintes de non-chevauchement entre les tâches, pensés pour l'ordonnancement mais restant néanmoins très génériques.

Dans le Chapitre 3, on s'intéresse aux difficultés encore rencontrées par la recherche locale générique de LocalSolver sur les problèmes d'ordonnancement disjonctif. Ces difficultés s'expliquent par le caractère très serré des contraintes de précédence et de non-chevauchement des tâches, à cause duquel il est impossible de passer d'une bonne solution à une autre en ne déplaçant qu'un petit nombre de tâches. Améliorer la solution courante nécessite alors de modifier la valeur d'un grand nombre de variables de décision, ce qui est contraire au principe de recherche locale à voisinages restreints. La solution retenue pour pallier ce problème consiste à réparer les solutions non réalisables obtenues à l'issue des mouvements de la recherche locale. On montre comment notre algorithme de réparation des solutions, basé sur la propagation de certaines contraintes spécifiques, permet d'améliorer largement les performances de LocalSolver sur les problèmes d'ordonnancement disjonctif.

Le Chapitre 4 se concentre sur une autre famille de problèmes d'ordonnancement, combinant des contraintes de précédence et des contraintes de packing, en prenant l'exemple du problème de l'Assembly Line Balancing. On commence par montrer comment le formalisme ensembliste de LocalSolver permet de modéliser ce problème de façon simple et concise. On montre ensuite comment les algorithmes implémentés au cours de la thèse améliorent les performances de la composante de recherche locale de LocalSolver, et lui permettent ainsi de résoudre très efficacement ce problème. On présente ainsi deux principaux algorithmes : un algorithme constructif, utilisé à la fois en tant que mouvement de recherche locale et comme algorithme d'initialisation, ainsi qu'un mouvement de recherche locale basé sur le principe des chaînes d'éjection, exploitant la structure de packing du problème, et décliné en deux versions.

Enfin, le Chapitre 5 conclut cette thèse en donnant une vue d'ensemble des améliorations apportées sur la résolution des problèmes d'ordonnancement, et en donnant quelques perspectives de futurs travaux. On évoquera ainsi des branches de l'ordonnancement non traitées pendant cette thèse, comme les problèmes cumulatifs et préemptifs, et la façon dont ils pourraient être traités, notamment du point de vue de la modélisation.

Contributions

La contribution principale de cette thèse correspond à l'élaboration des algorithmes de résolution présentés dans les différents Chapitres, mais aussi à leur implémentation au sein de LocalSolver. En effet, les objectifs de la thèse répondent avant tout à un besoin industriel. De ce fait, l'intégration des algorithmes élaborés à un logiciel complexe existant représente une part importante du travail de la thèse. L'ensemble des contributions algorithmiques de cette thèse ont ainsi été implémentées progressivement dans différentes versions commerciales du solveur, depuis LocalSolver 9.0, sorti en août 2019, jusqu'à LocalSolver 11.0, prévu pour mars 2022.

La contribution principale du Chapitre 2 correspond ainsi à l'élaboration de mouvements de recherche locale pensés pour l'ordonnancement, mais restant toutefois le plus génériques possibles, permettant d'exploiter efficacement la modélisation des ressources disjonctives décrite au début du Chapitre. Contrairement aux autres mouvements plus classiques de la recherche locale, ces mouvements modifient à la fois les variables entières représentant les dates de début des tâches et les variables de listes représentant leur ordre sur les différentes ressources, et assurent ainsi une certaine cohérence entre les valeurs des deux types de variables à l'issue de la transformation. Les mouvements de recherche locale présentés dans ce Chapitre, ainsi que les principes de modélisation sur lesquels ils s'appuient, ont été présentés lors de la conférence ROADEF 2021 [20].

Dans le Chapitre 3, il s'agit de l'élaboration et de l'implémentation de l'algorithme de réparation des solutions par propagation de contraintes, dans lequel chaque type de contraintes traité possède

son propre algorithme de filtrage. Cet algorithme de réparation est appliqué à chaque itération de la recherche locale, et doit donc être exécuté très rapidement. Pour cette raison, il diffère de la propagation classique de la programmation par contraintes par plusieurs points. Une réduction de domaine n'est propagée que si elle coupe la valeur courante de la variable. De plus, lors de la propagation de certains types de contraintes, il est possible de propager des réductions de domaines arbitraires, choisies aléatoirement. Les contributions du Chapitre 3 ont donné lieu à la publication d'un article, pour la conférence PPSN 2020 [17]. Elles ont également été présentées lors des conférences ROADEF 2020 [16] et PMS 2020-2021 [18].

Enfin, dans le Chapitre 4, la contribution principale consiste en l'élaboration et en l'implémentation au sein de LocalSolver des différentes versions des deux algorithmes décrits au cours du Chapitre, visant à améliorer les performances du solveur sur les problèmes présentant à la fois des contraintes de précédence et une structure de packing, comme celui de l'Assembly Line Balancing. Le premier est un algorithme constructif, permettant à la fois d'obtenir une première solution réalisable dès le début de la recherche lorsqu'il est utilisé en tant qu'algorithme d'initialisation des variables, et d'améliorer les performances de la recherche locale lorsqu'il est intégré au sein d'un mouvement de type « *destroy and repair* ». Le second algorithme présenté est un mouvement de recherche locale basé sur le principe des chaînes d'éjection. Il est applicable à tous types de problèmes présentant une structure de packing, et consiste à réarranger les éléments présents dans un ensemble de k variables de sets, afin de vider l'une d'entre elles. Les résultats numériques montrent que l'intégration de ces algorithmes permet à LocalSolver d'obtenir d'excellents résultats sur le problème de l'Assembly Line Balancing : ces résultats sont non seulement nettement meilleurs que ceux de CP Optimizer, mais ils améliorent également la meilleure solution connue de la littérature sur 59% des 500 instances du benchmark de référence introduit par [72]. Les contributions du Chapitre 4 ont été ou seront présentées lors des conférences ROADEF 2022 [21] et PMS 2022 [19].

Une autre partie importante du travail d'implémentation consiste à détecter dans le graphe d'évaluation les structures d'ordonnancement sur lesquelles s'appuient les algorithmes de résolution élaborés au cours de la thèse. Cette phase cruciale de détection est abordée dans les différents Chapitres : détection des tâches et ressources disjonctives dans le Chapitre 2, des contraintes de précédence et non-chevauchement généralisées dans le Chapitre 3, ou encore des structures de variables de sets ordonnées dans le Chapitre 4.

Chapitre 2

Modélisation de tâches et ressources disjonctives avec LocalSolver

2.1 Introduction

Ce Chapitre a pour but de montrer comment les différents types de variables de décision offerts par LocalSolver permettent de modéliser efficacement de nombreux problèmes d'ordonnancement, en n'utilisant que des opérateurs génériques. Plus précisément, on se concentrera sur les avantages apportés par l'utilisation combinée de variables de décision entières et de listes pour modéliser les problèmes d'ordonnancement disjonctif. D'une part, les variables entières constituent un choix naturel pour modéliser les décisions intervenant dans la plupart des problèmes d'ordonnancement : les dates de début des tâches, et éventuellement leurs durées. D'autre part, les variables de listes permettent de modéliser l'ordre dans lequel ces tâches sont ordonnancées. Comme expliqué dans la Section 1.2.1, une variable de liste de domaine n est une variable de décision dont la valeur est une permutation d'un sous-ensemble de $\{0, ..., n-1\}$. Ces variables de listes sont naturellement utilisées pour définir des ordres. Elles sont donc utiles pour modéliser les problèmes d'ordonnancement disjonctif, dans lesquels les tâches doivent être ordonnées sur les différentes ressources. En effet, si chaque tâche est associée à un indice entre 0 et $n-1$, la valeur de la liste permet de modéliser l'ordre des tâches [1].

Dans toute solution faisable, l'information de l'ordre donné par la liste est redondante avec les valeurs des dates de début des tâches. Cependant, on montre dans la suite du Chapitre que l'utilisation simultanée de ces deux types de variables apporte différents avantages. Du point de vue de la modélisation, elle permet une écriture plus directe et plus concise de certaines contraintes ou expressions. De plus, les informations plus riches qu'elle apporte sur la structure du problème (notamment la présence de ressources disjonctives) sont plus facilement exploitables par les algorithmes de résolution du solveur.

Les principes de modélisation présentés dans ce Chapitre ne sont toutefois pas utiles pour tous les problèmes d'ordonnancement disjonctif. En effet, comme expliqué en introduction dans la Section 1.3.1, certains problèmes, comme celui du Flow Shop par exemple, sont simples à décrire et peuvent être modélisés avec LocalSolver en utilisant uniquement des variables de listes. Les listes représentent alors l'ordre des tâches sur les différentes machines, et les dates de début des tâches

1. Cette utilisation des variables de listes pour l'ordonnancement peut rappeler les variables d'ordre existant dans certains autres solveurs, comme les variables de séquences d'intervalles (`IloSequence`) pour le solveur CP Optimizer (IBM) [59]. L'usage des variables de listes de LocalSolver n'est toutefois pas restreint aux problèmes d'ordonnancement.

peuvent être déduites simplement et de façon optimale à partir de cet ordre. On trouve également des problèmes d'ordonnancement par batches (comme le problème de l'Assembly Line Balancing, présenté dans le Chapitre 4) pouvant être modélisés en utilisant uniquement des variables de sets (représentant les différents batches). Dans ce Chapitre, on se concentre sur les problèmes pour lesquels les dates de début des tâches ne peuvent au contraire pas être déduites des valeurs des autres variables (listes représentant des ordres, sets représentant des batches), comme le problème du Job Shop, présenté dans la Section 1.3.1.

Le Chapitre est organisé de la façon suivante. La Section 2.2 présente la façon la plus simple d'exprimer un lien entre des variables entières et de listes, qui se traduit par une simple notion d'ordre sur les variables entières. On montre comment cette notion d'ordre peut être exploitée pour trouver une première solution réalisable rapidement, en présentant un algorithme d'initialisation des variables de listes en fonction des bornes des variables entières associées. Les Sections 2.3 et 2.4 se concentrent sur les problèmes comportant une ou plusieurs ressources disjonctives. On présente une formulation générique pour les contraintes de non-chevauchement entre les tâches dans la Section 2.3, inscrite dans le formalisme de modélisation de LocalSolver. Dans la Section 2.4, on montre comment la détection de ces contraintes au sein du solveur permet une résolution plus efficace des problèmes visés, grâce à des structures dédiées exploitées dans des mouvements de recherche locale implémentés au cours de la thèse. Les contributions algorithmiques de ce Chapitre ont été présentées lors de la conférence ROADEF 2021 [20].

2.2 Lien entre listes et entiers : simple notion d'ordre

2.2.1 Modélisation : tableau d'entiers, indicé par une liste

Comme mentionné dans la Section 2.1, de nombreux problèmes d'ordonnancement ont pour variables de décision des entiers représentant les dates de début des tâches qui les constituent. Ces tâches étant souvent réalisées dans un certain ordre, l'usage d'une variable de liste représentant cet ordre aide à la modélisation du problème. En effet, les dates de début seules ne permettent pas d'accéder facilement à la i-ème tâche de l'ordonnancement, ou aux tâches voisines d'une tâche donnée. En revanche, si l'on a non seulement défini un tableau de variables entières noté `start`, représentant les dates de début des tâches, mais également une variable de liste notée `order`, représentant leur ordre, leur expression devient facilement accessible. En effet, les variables entières sont alors ordonnées par la variable de liste. On peut ainsi très simplement exprimer la date de début de la tâche réalisée en position i : `start[order[i]]`.

De façon plus générale, cette Section s'intéresse à l'exploitation de structures de la forme `intArray[order[index]]` au sein du solveur, où `intArray` est un tableau de variables entières, `order` une variable de liste, et `index` une expression entière quelconque. La présence d'une telle structure dans le modèle dénote une intention d'ordonner les entiers du tableau `intArray` par la variable de liste `order`.

Exemple 2.1 (Problème de l'Aircraft Landing). Le but du problème de l'Aircraft Landing [80] consiste à choisir les dates d'atterrissage d'un ensemble d'avions. Chaque avion peut atterrir à l'intérieur d'une fenêtre de temps `[earliestTime, latestTime]` prédéterminée, autour d'une date cible `targetTime`. Un temps de séparation `separationTime`, défini pour toutes paires d'avions, doit en outre être respecté entre les dates d'atterrissage de deux avions consécutifs. L'objectif est de minimiser la somme des coûts de pénalité associés à un atterrissage en avance ou en retard par rapport à la date cible.

Pour modéliser le problème de l'Aircraft Landing efficacement avec LocalSolver, on peut justement choisir d'utiliser des variables entières associées à une variable de liste. Le principe du modèle est le suivant. On définit une variable de liste, notée `landingOrder`, pour modéliser l'ordre d'atterrissage des avions : le i-ème élément de la liste correspond à l'indice du i-ème avion à atterrir. A partir de cet ordre, on souhaite déterminer la date d'atterrissage de chaque avion p, en

utilisant une fonction récursive sur la liste `landingOrder`. L'avion p_i, en position i, doit respecter le temps de séparation avec l'avion en position $i-1$: la date `landingTime[i]` d'atterrissage de p_i doit être supérieure ou égale à la somme de la date `landingTime[i-1]` d'atterrissage de son prédécesseur et du temps de séparation `separationTime[landingOrder[i-1]][landingOrder[i]]`. Cette date est notée $t_i^{\min}$ dans la suite du paragraphe. Si la date $t_i^{\min}$ est supérieure à la date d'atterrissage cible de l'avion p_i, alors il est inutile d'attendre pour le faire atterrir : il n'est en effet jamais intéressant de faire atterrir un avion avec plus de retard que nécessaire. L'avion p_i doit donc atterrir à la date $t_i^{\min}$. Dans le cas contraire, c'est-à-dire si la date $t_i^{\min}$ est inférieure à la date d'atterrissage cible de l'avion p_i, il n'existe pas de choix systématiquement dominant pour la date d'atterrissage de p_i. En effet, on peut choisir d'attendre et de le faire atterrir à sa date cible, et ainsi éviter de payer une pénalité d'avance. Cependant, il peut être plus intéressant de faire atterrir p_i en avance, afin d'éviter de payer plus tard une pénalité plus importante liée au retard d'un des avions suivants. Afin de choisir une date d'atterrissage pour p_i (supérieure ou égale à $t_i^{\min}$ et à sa date d'atterrissage au plus tôt, et inférieure ou égale à sa date d'atterrissage cible), on utilise un autre type de variables de décision : pour chaque avion p, la variable entière `preferredTime[p]`, comprise entre `earliestTime[p]` et `targetTime[p]`, représente la date d'atterrissage « préférée » de l'avion p, lorsque celui-ci a la possibilité d'atterrir en avance. La date d'atterrissage `landingTime[i]` de l'avion p_i en position i est donc égale au maximum entre $t_i^{\min}$ et `preferredTime[landingOrder[i]]`.

Le code LSP complet pour la modélisation du problème de l'Aircraft Landing est donné par le Modèle 2.1. On en donne une description ligne par ligne au paragraphe suivant.

```
1  // Variable de liste : landingOrder[i] est l'indice du i-ème avion à atterrir.
2  landingOrder <- list(nbPlanes);
3
4  // Tous les avions doivent atterrir.
5  constraint count(landingOrder) == nbPlanes;
6
7  // Variables entières : date d'atterrissage "préférée" pour chaque avion s'il a la possibilité
       d'atterrir en avance.
8  preferredTime[p in 0..nbPlanes-1] <- int(earliestTime[p], targetTime[p]);
9
10 // Dates d'atterrissage des avions, définies récursivement (par ordre croissant).
11 landingTime <- array(0..nbPlanes-1, (p, prev) => max(preferredTime[landingOrder[p]],
12         p > 0 ? prev + separationTime[landingOrder[p-1]][landingOrder[p]] : 0));
13
14 // Coût de pénalité associé à la date d'atterrissage de chaque avion.
15 for [p in 0..nbPlanes-1] {
16     local planeIndex <- landingOrder[p];
17     constraint landingTime[p] <= latestTime[planeIndex];
18     cost[p] <- (landingTime[p] < targetTime[planeIndex] ? earlinessCost[planeIndex] :
19             tardinessCost[planeIndex]) * abs(landingTime[p] - targetTime[planeIndex]);
20 }
21
22 // Minimisation de la somme des coûts de pénalité.
23 totalCost <- sum[p in 0..nbPlanes-1] (cost[p]);
24 minimize totalCost;
```

Modèle 2.1 – Code LSP pour le problème de l'Aircraft Landing

On commence par définir la variable de liste `landingOrder`, représentant l'ordre d'atterrissage des avions (ligne 2). Chaque avion doit atterrir : la taille de la liste doit donc être égale au nombre d'avions (ligne 5). On introduit ensuite le deuxième type de variables de décision : les variables entières `preferredTime`, correspondant à la date d'atterrissage « préférée » de chaque avion (ligne 8), lorsque ceux-ci ont la possibilité d'atterrir en avance. Comme expliqué précédemment, un avion n'a jamais intérêt à atterrir après sa date cible s'il n'y est pas contraint par les avions précédents. Sa date d'atterrissage préférée se situe donc entre sa date d'atterrissage au plus tôt et sa date cible.

La date d'atterrissage réelle `landingTime` de chaque avion est définie par une fonction récursive

(lignes 11 à 12) : elle correspond au minimum entre sa date préférée et la date à partir de laquelle il peut atterrir en respectant le temps de séparation avec l'avion précédent (l'expression `prev` correspond ici à la case précédente dans le tableau `landingTime`, c'est-à-dire à la date d'atterrissage de l'avion précédent). Les éléments du tableau `landingTime` sont des expressions intermédiaires, et non des variables de décision : leurs valeurs sont entièrement déterminées par celles de la liste `landingOrder` et des entiers du tableau `preferredTime`. Pour calculer les dates d'atterrissage du tableau `landingTime`, on doit accéder à la date préférée du i-ème avion à atterrir, pour toute valeur de i, à savoir `preferredTime[landingOrder[i]]`. Comme mentionné ci-dessus, cette expression est de la forme `intArray[order[index]]`, où `intArray` est un tableau de variables entières, `order` une variable de liste, et `index` une expression entière quelconque. Dans la Section 2.2.2, on montre comment la détection de cette structure peut être exploitée pour initialiser efficacement la liste `order`.

Une fois la date d'atterrissage `landingTime` de chaque avion calculée, elle est contrainte à ne pas dépasser sa date d'atterrissage au plus tard (ligne 17). A partir de cette date, on calcule le coût lié à l'avance ou au retard de chaque avion (lignes 18 à 19). Enfin, on minimise la fonction objectif, égale à la somme des coûts (lignes 23 à 24).

2.2.2 Initialisation des listes en fonction des bornes des entiers associés

La composante de recherche locale au sein de LocalSolver ne comporte pas seulement des mouvements élémentaires sur les variables de décision, mais également des algorithmes constructifs d'initialisation des variables ensemblistes. En effet, si l'on contraint la taille d'une liste à être égale à son domaine, ou si l'on impose une partition sur un ensemble de listes, il est beaucoup plus efficace de chercher à remplir ces listes de façon à respecter ces contraintes structurantes dès le début de la recherche. Il existe ainsi différents algorithmes constructifs dans le solveur, exécutés dans les premières itérations de la recherche locale afin de remplir efficacement les variables de listes ou de sets en fonction des structures détectées dans le modèle. Par exemple, pour un problème de packing, si l'on détecte une fonction de poids associée aux éléments d'un ensemble de sets, ainsi qu'une capacité maximale autorisée sur ces sets, ceux-ci seront remplis de façon à ne pas excéder leur capacité.

De façon similaire, si le modèle comporte une structure de la forme `intArray[order[index]]`, suggérant que les entiers du tableau `intArray` sont ordonnés par la variable de liste `order`, on souhaite initialiser la liste efficacement, en fonction des informations dont on dispose sur les entiers qui lui sont associés. Dans cette Section, on décrit ainsi un nouvel algorithme constructif d'initialisation des variables de listes, utile sur les problèmes dans lesquels on détecte une notion d'ordre sur des variables entières.

Exemple 2.2 (Problème de l'Aircraft Landing). Dans le problème de l'Aircraft Landing, décrit à la Section 2.2.1 (Exemple 2.1), les variables entières ont des bornes très hétérogènes. En effet, ces variables correspondent à la date d'atterrissage préférée de chaque avion lorsqu'il a la possibilité d'atterrir en avance, située entre sa date d'arrivée au plus tôt et sa date cible, qui lui sont propres. On remarque alors que de nombreuses valeurs de la variable de liste conduisent à des solutions infaisables : si l'on choisit de faire atterrir en premier un avion dont la date d'atterrissage au plus tôt est très tardive, alors tous les avions suivants seront très en retard. Au contraire, si l'on choisit de faire atterrir les avions dans l'ordre croissant de leurs dates d'atterrissage au plus tôt ou au plus tard, on a de grandes chances d'obtenir une solution réalisable très rapidement, après seulement quelques ajustements locaux sur la liste, voire même immédiatement. Ce principe pouvant être appliqué à d'autres problèmes présentant une structure similaire, il est généralisé afin d'en faire un algorithme d'initialisation des variables de listes dans LocalSolver.

Algorithme d'initialisation des listes ordonnant des variables entières. Avant de commencer la phase de résolution, on cherche à savoir s'il existe des variables de listes liées à des

variables entières dans le modèle. Pour chaque variable de liste, notée `order`, on vérifie si le modèle comporte des expressions de la forme `intArray[order[index]]`, avec `intArray` un tableau d'expressions entières non constantes (variables de décision, sommes de variables de décision, ...) et `index` une expression entière quelconque. Si une telle structure est détectée, on s'intéresse aux bornes supérieures et inférieures des entiers du tableau `intArray`. Si celles-ci sont toutes égales, la détection de la structure `intArray[order[index]]` ne nous permet pas de trouver une valeur initiale pertinente pour la liste `order`. Cependant, s'il existe des différences parmi ces bornes, on retient que la liste `order` est liée au tableau d'entiers `intArray`, et que l'ordre qu'elle définit doit également correspondre à celui des entiers du tableau. Cette information pourra alors être exploitée au cours de la résolution.

Dans les premières itérations de la recherche locale de LocalSolver, on cherche en particulier à respecter les contraintes les plus structurantes du modèle, telles qu'une partition sur un ensemble de listes, ou une taille minimale pour une liste, en remplissant efficacement ces listes. Ainsi, lorsqu'on cherche à remplir une liste `order`, que l'on a détectée comme étant associée à un tableau d'entiers `intArray`, on applique l'algorithme d'initialisation suivant, que l'on note $\mathcal{A}_{\text{init}}$.

On considère que la liste `order` possède une taille cible, supérieure à sa taille actuelle. Cette taille cible peut avoir été donnée de façon directe par une contrainte, comme c'est par exemple le cas dans le problème de l'Aircraft Landing : tous les avions devant atterrir, la taille de la liste doit être égale au nombre d'avions. Elle peut également avoir été décidée plus tôt dans l'algorithme d'initialisation, par exemple si la liste `order` fait partie d'une partition. En plus des entiers du tableau `intArray`, la liste `order` peut également être associée à une fonction de poids sur ses éléments, et à une capacité maximale. Enfin, on note `missing` l'ensemble des éléments qu'il est possible d'ajouter dans la liste `order`, c'est-à-dire tous les éléments n'étant pas déjà contenus dans `order` ou dans une autre liste avec laquelle `order` forme une partition.

Tant que la taille cible et la capacité maximale de la liste `order` n'ont pas été atteintes, et que l'ensemble `missing` n'est pas vide, on cherche à ajouter un nouvel élément. On sélectionne ainsi un élément `i`, choisi aléatoirement dans l'ensemble `missing` de façon à respecter la capacité maximale de la liste `order`, et on l'ajoute à la liste. Lorsque l'on ne peut plus ajouter de nouvel élément dans la liste `order`, on réordonne les éléments présents en fonction des bornes des entiers du tableau `intArray`, en considérant la moyenne entre la borne inférieure et la borne supérieure de chaque entier. Ainsi, pour deux éléments `i` et `j` dans la liste `order`, si la moyenne des bornes de `intArray[i]` est plus petite que la moyenne des bornes de `intArray[j]`, alors l'élément `i` sera présent avant l'élément `j` dans la liste triée.

Exemple 2.3 (Application de l'algorithme au problème de l'Aircraft Landing). On présente ici le déroulement de l'algorithme d'initialisation $\mathcal{A}_{\text{init}}$ sur une petite instance du problème de l'Aircraft Landing.

Description de l'instance. Le nombre d'avions `nbPlanes` vaut 5. Pour chaque avion p, les coûts de pénalité associés à un atterrissage en avance `earlinessCost[p]` ou en retard `tardinessCost[p]` valent 1. Pour chaque paire d'avions p_1 et p_2, le temps de séparation `separationTime[p1][p2]` à respecter entre leurs dates d'atterrissage respectives vaut 2. Les dates d'atterrissage au plus tôt (`earliestTime`), cible (`targetTime`), et au plus tard (`latestTime`) sont données dans la Table 2.1.

Avion	`earliestTime`	`targetTime`	`latestTime`
0	1	2	20
1	10	13	20
2	0	1	20
3	5	7	20
4	9	10	20

TABLE 2.1 – Dates d'atterrissage au plus tôt, cible, et au plus tard.

Détection de la structure de variables entières ordonnées par une variable de liste. Pour calculer la date d'atterrissage de l'avion en position i, on accède à la valeur de sa date d'atterrissage préférée s'il a la possibilité d'atterrir en avance, à savoir `preferredTime[landingOrder[i]]` (ligne 11). On reconnaît ici une structure de la forme `intArray[order[index]]`, et on en déduit que les variables entières du tableau `preferredTime` sont ordonnées par la variable de liste `landingOrder`. On peut alors utiliser les bornes de ces variables entières pour initialiser la liste.

Déroulement de l'algorithme $\mathcal{A}_{init}$. La liste `landingOrder` représentant l'ordre d'atterrissage des avions, initialement vide, a pour taille cible le nombre d'avions `nbPlanes`, soit 5. L'ensemble `missing` des éléments qu'il est possible d'ajouter à la liste comprend tous les avions. Il n'y a aucune contrainte de capacité à respecter. On remplit alors la liste en y ajoutant successivement des éléments au hasard : on obtient par exemple la valeur $\{3, 1, 4, 2, 0\}$.

On trie ensuite la liste `order` selon la moyenne des bornes (`earliestTime`, `targetTime`) des variables de décision entières `preferredTime` qui lui sont associées. Celle-ci est alors initialisée avec la valeur $\{2, 0, 3, 4, 1\}$.

Les variables de décision étant initialisées (par défaut, les valeurs initiales des variables entières `preferredTime` sont égales à leurs bornes inférieures `earliestTime`), le solveur peut désormais calculer les valeurs des autres expressions :

- `landingTime[0]` = `preferredTime[2]` $= 0$
- `landingTime[1]` $= \max($`preferredTime[0]`$,$ `landingTime[0]` $+ 2) = 2$
- `landingTime[2]` $= \max($`preferredTime[3]`$,$ `landingTime[1]` $+ 2) = 5$
- `landingTime[3]` $= \max($`preferredTime[4]`$,$ `landingTime[2]` $+ 2) = 9$
- `landingTime[4]` $= \max($`preferredTime[1]`$,$ `landingTime[3]` $+ 2) = 11$
- `totalCost` $= 6$

Sur cet exemple, la solution ainsi obtenue est réalisable : aucun avion n'atterrit après sa date d'atterrissage au plus tard. Cette solution est représentée sur la Figure 2.1. Chaque couleur correspond à un avion p : sa date d'atterrissage `landingTime` est représentée par une croix sur l'axe temporel, le domaine [`earliestTime[p]`,`targetTime[p]`] de la variable `preferredTime[p]` est représenté sous l'axe temporel, et le temps de séparation que l'avion suivant doit respecter est représenté en gris.

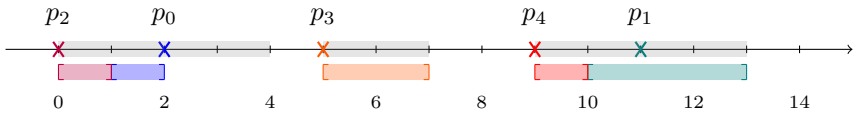


FIGURE 2.1 – Solution obtenue après application de l'algorithme $\mathcal{A}_{\text{init}}$

2.2.3 Résultats numériques sur le problème de l'Aircraft Landing

Cet algorithme d'initialisation des variables de listes associées à des tableaux d'entiers permet de trouver une solution faisable au problème de l'Aircraft Landing en quelques itérations seulement, quelle que soit la taille de l'instance. Ce résultat est d'autant plus appréciable qu'il est très difficile de trouver une solution réalisable sur les grosses instances de ce problème lorsque ce mécanisme est désactivé : si l'on n'utilise que les mouvements de la recherche locale, à partir d'une initialisation de la liste ne tenant pas compte de son lien avec les variables entières, il faut attendre plusieurs minutes avant d'atteindre la faisabilité.

La Table 2.2 présente les résultats obtenus par LocalSolver sur les instances d'Aircraft Landing proposées par Beasley dans [80]. Sur chacune de ces instances, comprenant entre 10 et 500 avions, on considère le temps en secondes nécessaire à LocalSolver pour atteindre une première solution réalisable, ainsi que l'écart relatif entre la meilleure solution connue et la solution retournée par

LocalSolver après une seconde et cinq minutes de recherche, le tout avec et sans l'algorithme d'initialisation $\mathcal{A}_{\text{init}}$. La valeur de la meilleure solution connue est tirée de [84].

Instance	Nb. avions	Temps faisabilité (s)		Gap 1 seconde		Gap 5 minutes	
		Avec	Sans	Avec	Sans	Avec	Sans
1	10	0	0	0%	0%	0.0%	0.0%
2	15	0	0	0%	0%	0.0%	0.0%
3	20	0	0	0%	251%	0.0%	0.0%
4	20	0	0	0%	3.2%	0.0%	0.0%
5	20	0	0	0%	0%	0.0%	0.0%
6	30	0	4	0%	/	0.0%	0.0%
7	44	0	2	0%	/	0.0%	0.0%
8	50	0	0	237%	556%	0.0%	0.0%
9	100	0	3	55%	/	0.0%	0.0%
10	150	0	10	71%	/	1.2%	2.9%
11	200	0	26	115%	/	4.1%	6.0%
12	250	0	63	117%	/	3.8%	18%
13	500	0	>300	145%	/	14%	/

TABLE 2.2 – Écart à la meilleure solution connue sur le problème de l'Aircraft Landing

La réduction du temps nécessaire pour atteindre une première solution faisable, rendue possible grâce à l'ajout de l'algorithme d'initialisation $\mathcal{A}_{\text{init}}$, permet à LocalSolver d'obtenir de bonnes performances sur le problème de l'Aircraft Landing. On constate en effet un écart à la meilleure solution connue inférieur à 5% après cinq minutes de calcul, sauf sur la plus grande instance (500 avions), pour laquelle il faut attendre un peu plus longtemps (on obtient par exemple un écart de 1.7% en trente minutes).

2.3 Utilisation des variables entières et de listes pour modéliser les ressources disjonctives

A la Section précédente, on a vu comment la présence au sein du modèle d'une simple structure de la forme `intArray[order[index]]`, suggérant une intention d'ordonner les entiers du tableau `intArray` par la variable de liste `order`, peut être exploitée pour construire des solutions initiales de bonne qualité.

Lorsque le problème comprend explicitement des ressources disjonctives, exprimées sous forme de contraintes dans modèle, on peut faire un lien encore plus fort entre les variables de listes représentant l'ordre des tâches et les variables entières représentant leurs dates de début. Cette Section s'intéresse ainsi aux contraintes de ressources disjonctives, et montre comment elles peuvent être modélisées de façon simple et concise avec LocalSolver. On montrera dans la Section suivante comment leur détection peut être exploitée pour améliorer les performances de la recherche locale.

2.3.1 Modélisation des ressources disjonctives à partir de contraintes de non-chevauchement des tâches

La présence de ressources disjonctives est caractérisée par des contraintes de non-chevauchement des tâches. Ces contraintes peuvent être exprimées de différentes manières. Une formulation simple et concise consiste à exploiter l'ordre des tâches : chaque tâche ne peut commencer qu'après la date de fin de la tâche qui la précède. Cette formulation de la contrainte de non-chevauchement des tâches est traduite par l'expression suivante dans le formalisme de modélisation de LocalSolver :

```
1 constraint and(0..nbTasks-2, i => start[order[i+1]] >= end[order[i]]);
```

où :

- `order` est une variable de liste, représentant l'ordre des tâches ;
- `nbTasks` est une expression entière, fixe ou variable, correspondant au nombre de tâches concernées par la contrainte (elle est ainsi généralement égale à la taille de la liste `order`) ;
- `start` est un tableau de variables entières, représentant les dates de début des tâches ;
- `end` est un tableau d'expressions entières, représentant les dates de fin des tâches (dates de début définies dans le tableau `start` plus durées fixes ou variables).

Cette contrainte est construite à partir d'une lambda-fonction au sein d'une fonction « et » variadique (voir Section 1.2.1). Elle se lit de la façon suivante : « pour toute valeur de `i` entre 0 et `nbTasks`−2, `start[order[i+1]]` doit être supérieur ou égal à `end[order[i]]` ».

2.3.2 Problèmes d'ordonnancement classiques

Les contraintes de non-chevauchement des tâches sont à la base de nombreux problèmes d'ordonnancement disjonctif, qu'il s'agisse d'ordonnancement d'atelier (problème du Job Shop par exemple) ou de planification de production (problème du Unit Commitment par exemple). Dans cette Section, on montre comment la formulation générique de ces contraintes, présentée dans la Section 2.3.1, peut être adaptée afin de modéliser divers problèmes d'ordonnancement disjonctif.

Problème du Job Shop. Le problème du Job Shop [92], décrit dans la Section 1.3.1, est un problème classique d'ordonnancement d'atelier comportant plusieurs ressources disjonctives. Le problème comporte un nombre `nbJobs` de jobs, chacun étant divisé en `nbMachines` tâches à réaliser dans un ordre donné sur chacune des machines disjonctives. On connaît la durée et la machine attribuée à chaque tâche. L'objectif du problème consiste à minimiser le makespan.

La modélisation LocalSolver du problème du Job Shop se fait en combinant deux types de variables de décision : des variables entières et des variables de liste. Pour toute machine m et tout job j, on note `start[j][m]` la variable entière représentant la date de début de la tâche du job j exécutée sur la machine m. Pour toute machine m, on note `order[m]` la variable de liste représentant l'ordre des tâches exécutées sur la machine m. Le code LSP complet pour la modélisation du problème du Job Shop est donné par le Modèle 2.2.

```
1  // Variables entières : dates de début des tâches.
2  start[j in 0..nbJobs-1][m in 0..nbMachines-1] <- int(0, maxStart);
3  end[j in 0..nbJobs-1][m in 0..nbMachines-1] <- start[j][m] + duration[j][m];
4
5  // Contraintes de précédence.
6  for [j in 0..nbJobs-1][k in 0..nbMachines-2]
7      constraint start[j][machine[j][k+1]] >= end[j][machine[j][k]];
8
9  // Variables de listes : ordre des tâches sur chaque machine.
10 order[0..nbMachines-1] <- list(nbJobs);
11 for [m in 0..nbMachines-1] constraint count(order[m]) == nbJobs;
12
13 // Contraintes de non-chevauchement des tâches sur chaque machine.
14 for [m in 0..nbMachines-1] {
15     constraint and(0..nbJobs-2, j => start[order[m][j+1]][m] >= end[order[m][j]][m]);
16 }
17
18 // Objectif : minimisation du makespan.
19 makespan <- max[j in 0..nbJobs-1](end[j][machine[j][nbMachines-1]]);
20 minimize makespan;
```

Modèle 2.2 – Code LSP pour le problème du Job Shop

On commence par définir les variables de décision entières représentant les dates de début des tâches (ligne 2), ainsi que des expressions intermédiaires représentant leurs dates de fin (ligne 3). En notant `machine[j][k]` la machine sur laquelle s'exécute la tâche en position k sur le job j,

on peut alors exprimer les contraintes de précédence entre les paires de tâches consécutives d'un même job (lignes 6 à 7).

On définit ensuite les variables de listes représentant l'ordre des tâches sur chaque machine (ligne 10). Chaque job ayant une tâche affectée à chaque machine, on contraint la taille de la liste à être égale au nombre de jobs (ligne 11). On peut alors exprimer les contraintes de non-chevauchement des tâches sur chaque machine (lignes 14 à 16), en suivant la formulation générique présentée à la Section 2.3.1 : pour toute machine m et tout indice j, la tâche en position $j+1$ sur la machine m doit commencer après la fin de la tâche en position j sur la machine m.

Enfin, on définit l'objectif, égal au maximum sur tous les jobs de la date de fin de la dernière tâche du job (lignes 19 à 20).

Remarque 2.1. Cette formulation des contraintes de non-chevauchement des tâches permet en outre de tenir compte de contraintes structurantes supplémentaires sur les machines, comme par exemple des plages de disponibilité. Si une machine m comprend k périodes fixes durant lesquelles elle est indisponible et ne peut donc traiter aucune tâche, on peut modéliser ses plages d'indisponibilité comme des tâches constantes fictives. Pour cela, il suffit d'imposer à la liste `order[m]` d'avoir une taille égale à `nbTasks`$+k$, et d'ajouter les dates de début (constantes) et les durées (constantes également) de ces plages d'indisponibilité aux tableaux `start[m]` et `duration[m]` respectivement. L'écriture de la contrainte de non-chevauchement des tâches affectées à cette machine ne change pas, et correspond toujours à celle présentée dans le Modèle 2.2.

Remarque 2.2. Une modélisation alternative des contraintes de non-chevauchement des tâches dans le problème du Job Shop peut se faire au moyen de disjonctions : pour chaque paire de tâches t_1 et t_2 affectées à une même machine, t_2 doit commencer après la fin de t_1, ou inversement. On a alors $O($`nbMachines` $\times$ `nbJobs`$^2)$ contraintes. La formulation présentée plus haut, comportant `nbMachines` contraintes variadiques de taille $O($`nbJobs`$)$, a donc l'avantage d'être beaucoup plus compacte.

Problème du Job Shop flexible. Le problème du Job Shop flexible [30], décrit dans la Section 1.3.1, est une variante du problème du Job Shop dans laquelle on peut choisir la machine sur laquelle chaque tâche sera exécutée, parmi un sous-ensemble de machines compatibles, avec des durées potentiellement différentes. Comme pour le problème du Job Shop, la modélisation du problème du Job Shop flexible avec LocalSolver se fait en utilisant des variables entières ainsi que des variables de listes : la date de début de chaque tâche t est modélisée par la variable entière `start[t]`, et l'ordre des tâches sur chaque machine m est modélisé par la variable de liste `order[m]`. Le code LSP complet pour la modélisation du problème du Job Shop flexible est donné par le Modèle 2.3.

```
1  // Variables de listes : ordre des tâches sur chaque machine.
2  order[m in 0..nbMachines-1] <- list(nbTasks);
3
4  // Chaque tâche est affectée à exactement une machine.
5  constraint partition[m in 0..nbMachines-1](order[m]);
6
7  // Chaque tâche doit être placée sur une machine avec laquelle elle est compatible.
8  for[t in 0..nbTasks-1][m in 0..nbMachines-1] {
9      if (incompatible[t][m]) {
10         constraint not contains(order[m], t);
11     }
12 }
13
14 // Variables entières : dates de début des tâches.
15 start[t in 0..nbTasks-1] <- int(0, maxStart);
16
17 // Machine sélectionnée pour chaque tâche.
18 taskMachine[t in 0..nbTasks-1] <- find(order, t);
```

```
19
20 // Variables intermédiaires : la durée d'une tâche dépend de la machine sélectionnée.
21 duration[t in 0..nbTasks-1] <- taskProcessingTime[t][taskMachine[t]];
22 end[t in 0..nbTasks-1] <- start[t] + duration[t];
23
24 // Contraintes de précédence entre deux tâches consécutives d'un même job.
25 for [j in 0..nbJobs-1][o in 0..nbOperations[j]-2] {
26     local t1 = jobOperationTask[j][o];
27     local t2 = jobOperationTask[j][o+1];
28     constraint start[t2] >= end[t1];
29 }
30
31 // Contraintes de non-chevauchement des tâches sur chaque machine.
32 for [m in 0..nbMachines-1] {
33     constraint and(0..count(order[m])-2, t => start[order[m][t+1]] >= end[order[m][t]]);
34 }
35
36 // Objectif : minimisation du makespan.
37 makespan <- max[t in 0..nbTasks-1] (end[t]);
38 minimize makespan;
```

Modèle 2.3 – Code LSP pour le problème du Job Shop flexible

On commence par définir les variables de listes représentant l'ordre des tâches affectées aux différentes machines (ligne 2). Le domaine de chaque liste est égal au nombre total de tâches `nbTasks`, et non au nombre de jobs comme dans le problème du Job Shop. En effet, dans le problème du Job Shop, chaque machine exécute exactement une tâche de chaque job : ces tâches peuvent donc être identifiées par le numéro du job auquel elles appartiennent. Puisque l'ensemble des tâches exécutées sur chaque machine n'est pas fixe dans le problème du Job Shop flexible, il est ici nécessaire d'identifier chaque tâche par un numéro entre 0 et le nombre total de tâches moins 1. Puisque chaque tâche doit être exécutée sur exactement une machine, on contraint l'ensemble des variables de listes à former une partition (ligne 5). On exprime ensuite les contraintes de compatibilité entre les tâches et les machines (lignes 8 à 12) : si une tâche t est incompatible avec une machine m, alors la variable de liste `order[m]` ne peut contenir l'élément `t`.

On définit les variables de décision entières représentant les dates de début des tâches (ligne 15). Pour chaque tâche t, on définit ensuite la variable intermédiaire `taskMachine[t]`, représentant la machine sélectionnée pour t (ligne 18), dont la valeur est calculée grâce à l'opérateur `find`[2]. A partir de la variable intermédiaire `taskMachine[t]`, on peut exprimer la durée `duration[t]` (ligne 21), puis la date de fin `end[t]` (ligne 22) de chaque tâche t. On peut alors exprimer les contraintes de précédence entre toutes les paires de tâches consécutives sur un même job (lignes 25 à 29). On écrit ensuite les contraintes de non-chevauchement des tâches affectées à chaque machine (lignes 32 à 34), en suivant la formulation générique présentée à la Section 2.3.1. Le nombre de tâches affectées à chaque machine m n'étant pas fixe, l'intervalle sur lequel s'applique la lambda-fonction est variable, et va de 0 à `count(order[m])`-2. Enfin, on définit l'objectif, égal à la plus grande date de fin (lignes 37 à 38).

Remarque 2.3. La formulation alternative pour le problème du Job Shop, présentée dans la Remarque 2.2, ne fonctionne pas pour le problème du Job Shop flexible. En effet, cette formulation à base de disjonctions suppose de connaître à l'avance les tâches affectées à chaque machine, ce qui n'est pas le cas ici. Au contraire, la formulation basée sur l'utilisation d'une lambda-fonction au sein d'une fonction « et » variadique, présentée à la Section 2.3.1, est parfaitement adaptée aux problèmes dans lesquels le nombre de tâches affectées à chaque machine est variable.

Le problème du Job Shop flexible s'accompagne parfois de temps de transition à respecter entre deux tâches consécutives sur une même machine [88]. En notant `setup[m][t1][t2]` le temps

2. L'expression `find(collectionsArray, x)` renvoie l'indice dans le tableau `collectionsArray` de la variable ensembliste contenant l'élément `x`, ou -1 si aucune de ces collections ne contient l'élément `x`.

de transition à respecter entre la date de fin d'une tâche t_1 et la date de début d'une tâche t_2, s'exécutant de façon consécutive sur une machine m, la contrainte de non-chevauchement des tâches affectées à la machine m s'écrit de la façon suivante, et remplace la ligne 33 dans le Modèle 2.3.

```
constraint and(0..count(order[m])-2, t => start[order[m][t+1]] >= end[order[m][t]]
    + setup[m][order[m][t]][order[m][t+1]]);
```

Problème de l'Open Shop. Le problème de l'Open Shop [91], décrit à la Section 1.3.1, est une autre variante du problème du Job Shop, dans laquelle les tâches composant chaque job peuvent être exécutées dans un ordre quelconque. Le problème ne comporte donc pas de contraintes de précédence, mais des contraintes de non-chevauchement des tâches composant un même job : les jobs peuvent ainsi être assimilés à des ressources disjonctives. La modélisation du problème de l'Open Shop est également basée sur l'utilisation de variables de décisions entières et de listes. La date de début de la tâche d'un job j affectée à une machine m est modélisée par la variable entière `start[j][m]`, l'ordre des machines exécutant un job j est modélisé par la variable de liste `machinesOrder[j]`, et l'ordre des tâches exécutées sur une machine m (identifiées par le numéro du job auquel elles appartiennent) est modélisé par la variable de liste `jobsOrder[m]`. Le code LSP complet pour la modélisation du problème de l'Open Shop est donné par le Modèle 2.4.

```
1  // Variables entières : dates de début des tâches.
2  start[j in 0..nbJobs-1][m in 0..nbMachines-1] <- int(0, maxStart);
3  end[j in 0..nbJobs-1][m in 0..nbMachines-1] <- start[j][m] + duration[j][m];
4
5  // Variables de listes : ordre des machines pour chaque job.
6  machinesOrder[0..nbJobs-1] <- list(nbMachines);
7  for [j in 0..nbJobs-1] {
8      constraint count(machinesOrder[j]) == nbMachines;
9      // Non-chevauchement des tâches composant un même job.
10     constraint and(0..nbMachines-2, i => start[j][machinesOrder[j][i+1]]
11             >= end[j][machinesOrder[j][i]]);
12 }
13
14 // Variables de listes : ordre des jobs sur chaque machine.
15 jobsOrder[0..nbMachines-1] <- list(nbJobs);
16 for [m in 0..nbMachines-1] {
17     constraint count(jobsOrder[m]) == nbJobs;
18     // Non-chevauchement des tâches affectées à une même machine.
19     constraint and(0..nbJobs-2, i => start[jobsOrder[m][i+1]][m] >= end[jobsOrder[m][i]][m]);
20 }
21
22 // Objectif : minimisation du makespan.
23 makespan <- max[j in 0..nbJobs-1][m in 0..nbMachines-1](end[j][m]);
24 minimize makespan;
```

Modèle 2.4 – Code LSP pour le problème de l'Open Shop

On commence par définir les variables entières représentant les dates de début des tâches (ligne 2), ainsi que les expressions intermédiaires représentant leurs dates de fin (ligne 3). On définit ensuite les variables de listes représentant l'ordre des machines exécutant chaque job (ligne 6). Puisque chaque job comprend une tâche par machine, on contraint la taille de ces listes à être égale au nombre de machines (ligne 8). On peut ensuite exprimer les contraintes de non-chevauchement des tâches composant chacun des jobs, en suivant la formulation générique présentée à la Section 2.3.1 (lignes 10 à 11). De même, on définit ensuite les variables de listes représentant l'ordre des tâches exécutées sur chaque machine (ligne 15), et on contraint leur taille à être égale au nombre de jobs (ligne 17). On exprime alors les contraintes de non-chevauchement des tâches affectées à une même machine (ligne 19), en suivant le même principe que précédemment. Enfin, on définit l'objectif, égal à la plus grande date de fin (lignes 23 à 24).

Problème du Unit Commitment. On considère le problème du Unit Commitment [23], dans une version simplifiée purement combinatoire, introduite dans la Section 1.3.1. On dispose d'un nombre `nbUnits` d'usines, dont on doit décider du fonctionnement au cours du temps. A chaque pas de temps, chaque usine peut être en fonctionnement, ou éteinte. Chaque plage de fonctionnement sur une usine u possède une durée minimale `minTimeOn[u]`. De même, chaque plage de non-fonctionnement entre deux plages de fonctionnement possède une durée minimale `minTimeOff[u]`. Chaque usine a un niveau de production fixe, associé à un coût de production fixe lorsque l'usine est en fonctionnement. A chaque pas de temps, la production totale des usines en fonctionnement doit être supérieure à une demande donnée. De plus, on définit des coûts de démarrage au début de chaque plage de fonctionnement. Suivant la durée séparant deux plages de fonctionnement sur une usine, on considère que celle-ci effectue un démarrage « à chaud » ou « à froid », associés à des coûts différents. L'objectif consiste à minimiser le coût total de fonctionnement des usines, égal à la somme des coûts de production et des coûts de démarrage.

Le problème du Unit Commitment décrit ci-dessus est un problème d'ordonnancement de production (ou plus exactement un problème de planification de production [3]), très différent des problèmes d'ordonnancement d'atelier abordés dans les paragraphes précédents. Cependant, la modélisation LocalSolver de ce problème peut se faire sur le même principe, en utilisant des variables de décision entières et de listes, à partir desquelles on définit des contraintes de non-chevauchement des tâches affectées à une même ressource disjonctive, reprenant la formulation générique de la Section 2.3.1. Le principe de cette modélisation consiste à assimiler les plages de production sur les différentes usines à des tâches, optionnelles, dont les dates de début et les durées sont variables. Des variables de listes permettent par ailleurs de représenter la séquence des plages de fonctionnement sur chaque usine. Le code LSP complet pour la modélisation du problème du Unit Commitment simplifié est donné par le Modèle 2.5.

```
// Variables de listes : séquence des plages de fonctionnement sur chaque usine.
tasks[u in 0..nbUnits-1] <- list(maxNbTasks);
constraint disjoint[u in 0..nbUnits-1](tasks[u]);

// Variables entières : dates de début et durées des plages de fonctionnement.
start[x in 0..maxNbTasks-1] <- int(0, nbTimesteps);
duration[x in 0..maxNbTasks-1] <- int(minDuration, maxDuration);
end[x in 0..maxNbTasks-1] <- start[x] + duration[x];

for [u in 0..nbUnits-1] {
    local c <- count(tasks[u]);

    // Durée minimale d'une plage de fonctionnement.
    constraint and(0..c-1, i => duration[tasks[u][i]] >= minTimeOn[u]);

    // Durée minimale des plages de non-fonctionnement (non-chevauchement des plages de
     fonctionnement).
    constraint and(0..c-2, i => start[tasks[u][i+1]] >= end[tasks[u][i]] + minTimeOff[u]);

    // Coût de démarrage sur chaque usine.
    startingCost[u] <- c == 0 ? 0 : costColdStart[u]
            + sum(0..c-2, i => start[tasks[u][i+1]] > end[tasks[u][i]] + timeColdStart[u]
            ? costColdStart[u] : costHotStart[u]);

    // A chaque pas de temps, on détermine si l'usine est allumée ou éteinte.
    isUnitOn[u][t in 0..nbTimesteps-1] <- or(0..c-1, i => start[tasks[u][i]] <= t
            && end[tasks[u][i]] > t);
}
```

3. Dans la littérature, on parle plutôt de planification de la production que d'ordonnancement. En ordonnancement, on considère une liste de tâches à faire, tandis qu'en planification, on doit construire les tâches ou plages de production, que l'on ne connaît pas à l'avance. Le problème du Unit Commitment est donc plus un problème de planification. Cependant, on montre dans la suite du paragraphe qu'il peut être modélisé comme un problème d'ordonnancement de tâches optionnelles.

```
28
29 // La production doit être supérieure à la demande à chaque pas de temps.
30 for [t in 0..nbTimesteps-1] {
31     totalPower[t] <- sum[u in 0..nbUnits-1](isUnitOn[u][t] * unitPower[u]);
32     constraint totalPower[t] >= powerDemand[t];
33 }
34
35 // Objectif : minimisation des coûts de production et de démarrage.
36 totalProductionCost <- sum[u in 0..nbUnits-1][t in 0..nbTimesteps-1]
37         (isUnitOn[u][t] * productionCost[u][t]);
38 totalStartingCost <- sum[u in 0..nbUnits-1](startingCost[u]);
39 minimize totalProductionCost + totalStartingCost;
```

Modèle 2.5 – Code LSP pour le problème du Unit Commitment (simplifié)

On commence par définir les variables de listes, représentant la séquence des plages de fonctionnement de chaque usine (ligne 2). On peut calculer une borne supérieure `maxNbTasks` du nombre total de plages de fonctionnement (à partir des durées minimales des plages de fonctionnement et non-fonctionnement). Le domaine des variables de listes est alors égal à `maxNbTasks`. Chaque tâche correspondant à une plage de fonctionnement optionnelle, on impose une contrainte de type « *disjoint* » sur l'ensemble des variables de listes (ligne 3). Ainsi, chaque élément de $\{0, ...\texttt{maxNbTasks} - 1\}$ appartient à au plus une variable de liste : chaque plage de fonctionnement (tâche) est affectée à au plus une usine. On définit ensuite les variables de décision entières correspondant aux dates de début et aux durées des plages de fonctionnement (lignes 6 à 8).

On peut alors écrire les contraintes liées aux durées des plages de fonctionnement et de non-fonctionnement. On utilise une contrainte de type « et » variadique pour imposer une durée minimale égale à `minTimeOn[u]` à toutes les plages de fonctionnement de chaque usine u (ligne 14). On impose ensuite une durée minimale aux plages de non-fonctionnement grâce à des contraintes de non-chevauchement des plages de fonctionnement (ligne 17) : pour chaque usine u et chaque indice i, la plage de fonctionnement en position $i + 1$ sur l'usine u doit commencer au moins `minTimeOff[u]` pas de temps après la fin de la plage de fonctionnement en position i sur l'usine u. On peut ensuite calculer le coût de démarrage associé à chaque usine u en fonction des durées des plages de non-fonctionnement (lignes 20 à 22). Si aucune plage de production n'est définie sur cette usine, le coût est nul (ligne 20). Sinon, la première plage de production démarre à froid (ligne 20). En fonction de la durée des plages de non-fonctionnement précédant chacune des plages de production suivantes, on détermine si celles-ci démarrent à chaud ou à froid (lignes 21 à 22).

Afin de mesurer la production des usines à chaque pas de temps, on calcule pour chaque usine u et chaque pas de temps t l'expression intermédiaire `isUnitOn[u][t]` (lignes 25 à 26) : l'usine est en fonctionnement si elle possède une plage de fonctionnement commençant avant t et terminant strictement après t. On peut alors calculer la production totale à chaque pas de temps (ligne 31), que l'on contraint à être supérieure ou égale à la demande (ligne 32). On calcule également le coût de production total des usines (lignes 36 à 37). Enfin, on définit l'objectif, égal à la somme des coûts de production et des coûts de démarrage (ligne 39).

Remarque 2.4. Dans le problème du Unit Commitment simplifié défini ici, les tâches sont toutes équivalentes : elles indiquent simplement que l'usine est en fonctionnement. On peut cependant imaginer des variantes de ce problème, dans lesquelles les usines fabriquent des produits différents (par exemple, des usines de produits chimiques capables de fabriquer différents composants). On peut alors définir une modélisation dans laquelle chaque tâche est associée à un type, et correspond à une plage de production d'un composant particulier. Cela ne change pas la formulation de la contrainte de durée minimale des plages de non-fonctionnement (toujours écrite sous la forme d'une contrainte de non-chevauchement des tâches).

Remarque 2.5. Une formulation alternative classique pour le problème du Unit Commitment, récemment exploitée par exemple dans [53], consiste à caractériser l'état de fonctionnement de

chaque usine à chaque pas de temps par des variables de décision booléennes. Cependant, la formulation à base de tâches présentée ici permet de modéliser beaucoup plus simplement les contraintes associées aux durées minimales des plages de fonctionnement et de non-fonctionnement.

2.4 Mouvements de recherche locale sur les tâches

La formulation des contraintes de non-chevauchement décrite à la Section précédente permet ainsi de décrire des ressources disjonctives dans de nombreux cas particuliers correspondant à des problèmes très différents. De plus, elle permet de faire un lien très clair entre la variable de liste représentant l'ordre des tâches et les tableaux contenant les expressions entières représentant les dates de début et durées (fixes ou variables) des tâches.

2.4.1 Structure dans LocalSolver

Afin de mieux résoudre les problèmes d'ordonnancement disjonctif, on crée au sein de LocalSolver des structures liées à la présence de tâches et de ressources disjonctives, exploitables par les mouvements de la recherche locale. L'activation de ces structures lors de la résolution d'un problème donné repose sur la détection dans le modèle de contraintes de non-chevauchement des tâches, telles que décrites dans la Section 2.3. Plus précisément, ces structures sont créées si l'on détecte des contraintes dont l'expression peut être ramenée à la forme canonique suivante :

```
constraint and(0..nbTasks-2, i => start[order[i+1]] >= start[order[i]] + duration[order[i]]
    + setup[order[i]][order[i+1]]);
```

où :

- `order` est une variable de liste associée à une ressource disjonctive, représentant l'ordre des tâches sur cette ressource.
- `nbTasks` est une expression entière quelconque (constante ou variable, généralement égale à la taille de la liste `order`).
- `start` est un tableau d'expressions entières, comprenant au moins une variable de décision entière, et éventuellement des constantes. La présence de constantes dans le tableau `start` peut être due par exemple au fait que la première tâche doit commencer à l'instant 0, ou indiquer la présence de tâches fictives fixes (voir Remarque 2.1).
- `duration` est un tableau d'expressions entières positives contenant des constantes et/ou des variables de décision entières.
- `setup` est un tableau bidimensionnel de constantes entières, éventuellement toutes nulles, représentant les temps de transition à respecter entre la date de fin d'une tâche et la date de début de la tâche suivante.

Lorsque de telles contraintes de non-chevauchement des tâches sont détectées dans le modèle, trois types de structures sont créées : les ressources, les tâches, et les familles de tâches.

Ressources. Chaque variable de liste intervenant dans une contrainte de non-chevauchement des tâches est interprétée comme une ressource disjonctive. Chaque structure de ressource créée dans le solveur est alors associée à une variable de liste `order` (représentant l'ordre des tâches qu'elle exécute), à l'ensemble de tâches qu'elle est susceptible d'exécuter, et éventuellement à un tableau bidimensionnel de temps de transition `setup`. Afin d'alléger les notations, et lorsque le contexte est clair, on emploiera parfois le terme « ressource » pour désigner la variable de liste associée à une ressource. Par exemple, on pourra parler du « domaine » [4] ou de la « taille » [5] d'une ressource.

4. Le domaine d'une variable de liste correspond au nombre maximal d'éléments qu'elle peut contenir. Le domaine de la ressource qui lui est associée correspond alors au nombre maximal de tâches que celle-ci peut exécuter.

5. La taille d'une variable de liste correspond au nombre d'éléments qu'elle contient dans une solution donnée. La taille de la ressource qui lui est associée correspond alors au nombre de tâches présentes sur cette ressource dans

Tâches. Chaque structure de tâche est créée à partir d'un couple d'expressions entières (variables de décision ou constantes) de la forme (`start[i]`, `duration[i]`), avec `i` entre 0 et `nbTasks` − 1. Ces deux expressions sont respectivement interprétées comme la date de début et la durée de la tâche. On distingue deux types de tâches : les tâches à durée fixe (lorsque le tableau `duration` ne contient que des constantes), et les tâches à durée variable (lorsque le tableau `duration` contient des variables entières).

Familles de tâches. Les tâches sont regroupées dans des structures de familles de tâches. Une famille de tâches regroupe l'ensemble des tâches associées à une même contrainte de non-chevauchement, c'est-à-dire dont les dates de début ont été détectées à partir d'un même tableau `start`, et dont les durées sur la ressource concernée ont été détectées à partir d'un même tableau `duration`. Toutes les tâches composant une famille sont de même type. Ainsi, si le tableau `duration` ne contient que des constantes, chaque tâche a une durée fixe. Inversement, si le tableau `duration` contient au moins une variable entière, toutes les tâches de la famille sont considérées comme ayant des durées variables (même si certaines peuvent en pratique avoir des durées constantes).

Chaque famille de tâches est également associée à une ou plusieurs ressources, révélées par les différentes variables de listes intervenant dans une contrainte de non-chevauchement impliquant le tableau `start` définissant les tâches de la famille. Par exemple, dans le problème du Job Shop, dont le modèle est donné dans la Section 2.3.2, chaque tâche est affectée à l'avance à une ressource donnée. On a donc autant de familles de tâches que de ressources, et chaque famille de tâches est associée à une unique ressource. Au contraire, dans le problème du Job Shop flexible, dont le modèle est également donné dans la Section 2.3.2, les tâches peuvent être exécutées par plusieurs ressources. Les variables de listes représentant ces ressources sont donc contraintes à former une partition, et le tableau `start` intervenant dans l'expression des différentes contraintes de non-chevauchement est unique. On a donc une unique famille de tâches, associée à toutes les ressources à la fois. Le problème de l'Open Shop est également intéressant à illustrer : on associe une famille de tâches à chaque machine, mais également à chaque job. En effet, dans le modèle présenté à la Section 2.3.2 pour le problème de l'Open Shop, les jobs sont assimilés à des ressources disjonctives. Dans ce problème, chaque tâche appartient à un job donné, et est exécutée par une machine donnée : elle appartient donc à deux familles de tâches à la fois.

En fonction des contraintes sur les variables de listes représentant les ressources associées à une famille de tâches, on peut de plus déterminer si la famille contient ou non des tâches optionnelles. Si une famille de tâches est associée à une unique ressource contrainte à avoir une taille égale à son domaine (problème du Job Shop ou de l'Open Shop par exemple), ou si elle est associée à un ensemble de ressources contraintes à former une partition (problème du Job Shop flexible par exemple), alors chacune des tâches doit être exécutée : la famille ne contient pas de tâches optionnelles. Si aucune contrainte de ce type n'est détectée (problème du Unit Commitment par exemple), alors toutes les tâches ne sont pas nécessairement ordonnancées, et on considère que la famille peut contenir des tâches optionnelles.

2.4.2 Mouvements d'ordonnancement

Dans cette Section, on s'intéresse aux mouvements élémentaires implémentés au sein de la recherche locale de LocalSolver. On passe en revue l'ensemble des mouvements pouvant s'appliquer sur les problèmes d'ordonnancement comportant une ou plusieurs ressources disjonctives. On décrira plus particulièrement les mouvements spécifiques à ces problèmes, qui s'appuient sur les structures de familles de tâches présentées à la Section 2.4.1, créées à partir des contraintes de non-chevauchement détectées dans le modèle.

Tous les mouvements présentés ici ne sont cependant pas activés lors de la résolution de tous les problèmes comportant des contraintes de non-chevauchement des tâches. En effet, la recherche

la solution considérée.

locale de LocalSolver a plutôt été pensée pour conserver la faisabilité des solutions une fois celle-ci atteinte. Par souci d'homogénéité, les mouvements d'ordonnancement implémentés au cours de la thèse respectent également ce principe. De ce fait, les mouvements ne pouvant de façon certaine pas aboutir à une solution réalisable sur un modèle donné sont désactivés sur ce modèle. Par exemple, certains mouvements supposent de modifier la durée de certaines tâches, et ne sont donc actifs que sur les problèmes présentant des familles de tâches de durées variables. D'autres mouvements cherchent à déplacer des tâches d'une ressource vers une autre, et ne sont donc appelés que sur les problèmes comportant des familles de tâches associées à plusieurs ressources. De même, certains mouvements consistent à ajouter à une ressource une nouvelle tâche non encore ordonnancée, ou au contraire à supprimer l'affectation d'une tâche. On décide alors de n'appliquer ces mouvements qu'aux problèmes comprenant des tâches optionnelles.

Les caractéristiques principales des mouvements sur les familles de tâches (durées quelconques ou variables, nombre de tâches concernées, nombre de ressources concernées, présence de tâches optionnelles requise ou non) sont résumées dans la Table 2.3. Ces mouvements sont présentés en détail dans la suite de la Section. Ils sont classés en quatre catégories : tâches de durées quelconques ou variables, impliquant une ou plusieurs ressources.

Mouvement	Durée des tâches	Nb. tâches concernées	Nb. ressources concernées	Tâches optionnelles
Insertion	Quelconque	1	1	Oui
k-échange des dates de début	Quelconque	2 à 5	1	Non
Remplacement	Quelconque	2	1	Oui
Décalage	Quelconque	1	1	Non
Changement de date de début	Variable	1	1	Non
k-échange des durées	Variable	2 à 5	1	Non
Fusion	Variable	2	1	Oui
Fractionnement	Variable	1	1	Oui
Glissement d'interstice	Variable	2	1	Non
Changement de ressource	Quelconque	1	2	Non
Échange de ressources	Quelconque	2	2	Non
Fractionnement et changement de ressource	Variable	1	2	Oui
k-inversion sur un petit intervalle	Variable	Variable	3 à 6	Oui

Table 2.3 – Tableau récapitulatif des mouvements d'ordonnancement

Notations. Chaque mouvement présenté dans la suite de cette Section s'applique lors d'une itération quelconque de la recherche locale, à partir d'une solution dite « initiale » notée $\mathcal{S}$.

Remarque 2.6. Tous les mouvements d'ordonnancement présentés dans la suite de cette Section commencent par la sélection d'une famille de tâches appropriée (avec des tâches de durées variables, associée à plusieurs ressources, ou présentant des tâches optionnelles lorsque la nature du mouvement l'impose). Toutes les tâches et ressources étant par la suite sélectionnées et modifiées pendant le déroulement du mouvement sont associées à cette famille de tâches.

2.4.2.1 Mouvements classiques

Certains mouvements très génériques s'appliquent sur de nombreux modèles, indépendamment de la présence de contraintes de non-chevauchement : ce sont typiquement les mouvements modifiant uniquement les dates de début ou les durées (mouvements sur les variables entières), ainsi que les mouvements modifiant uniquement l'ordre des tâches (mouvements sur les variables de listes). Ces mouvements ne sont cependant pas prévus pour maintenir la relation entre ces deux types de

variables. Certains modifient une variable de liste, et ainsi l'ordre sur les tâches que contient la ressource associée à cette liste, sans tenir compte des valeurs des dates de début des tâches. Les autres modifient les variables entières, et ainsi les dates de début et durées des tâches, sans réordonner la liste représentant leur ordre afin de tenir compte de ces changements. Pour exploiter au mieux la structure donnée par une contrainte de non-chevauchement, on a donc besoin de nouveaux mouvements, spécifiques aux problèmes d'ordonnancement, exploitant toute l'information fournie par cette contrainte, et permettant ainsi de modifier la solution courante tout en maintenant sa faisabilité.

2.4.2.2 Mouvements sur les familles de tâches quelconques

On commence par considérer les mouvements s'appliquant sur toutes les familles de tâches, qu'elles soient composées de tâches à durées fixes ou variables, et associées à une ou plusieurs ressources disjonctives. Cette catégorie comprend quatre mouvements, notés « insertion », « k-échange des dates de début », « remplacement », et « décalage ».

Insertion. Le mouvement d'insertion consiste à ajouter une nouvelle tâche sur une ressource. On commence par sélectionner une ressource r, et une tâche t compatible avec r et non affectée dans la solution initiale $\mathcal{S}$ (ce mouvement ne s'applique donc qu'à partir de familles de tâches présentant des tâches optionnelles). On choisit ensuite au hasard une position éligible pour l'insertion de la tâche t dans la liste représentant l'ordre des tâches sur la ressource r : pour qu'une nouvelle tâche puisse être insérée entre les positions i et $i+1$ de la liste, il faut que l'écart entre la date de fin de la i-ème tâche t_{prev} et la date de début de la $(i+1)$-ième tâche t_{next} soit suffisant, c'est-à-dire supérieur à `setup[prev][t] + minDuration + setup[t][next]` (où `minDuration` est la durée minimale de la tâche t). On choisit alors une date de début (et éventuellement une durée si celle-ci n'est pas fixe) pour la tâche t, de sorte que celle-ci ne chevauche ni la tâche qui la précède sur la ressource r, ni celle qui lui succède : la tâche t est placée soit au plus près de t_{prev}, soit au plus près de t_{next}, soit de façon à remplir tout l'intervalle disponible entre la fin de t_{prev} et le début de t_{next} (si la durée de t est variable), soit de façon aléatoire.

Exemple 2.4 (Mouvement d'insertion). On considère une ressource r sur laquelle on veut insérer une nouvelle tâche t', de durée fixe. Le mouvement d'insertion correspondant est illustré par la Figure 2.2. Cinq tâches, notées t_1 à t_5 et représentées en gris clair, sont déjà présentes sur la ressource. Le temps de transition entre toute paire de tâches vaut 1, et est représenté en gris foncé. La durée d'exécution de la tâche t' vaut 9 : on en déduit les positions d'insertion possibles pour t' sur la ressource r, représentées en vert. Un possible déroulement du mouvement d'insertion est le suivant :

— On choisit une position d'insertion pour t' parmi les trois possibles : ici entre t_3 et t_4.
— On choisit une date de début pour t' : ici 58.

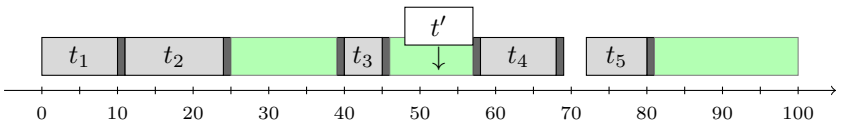


FIGURE 2.2 – Exemple de mouvement d'insertion.

k-échange des dates de début. On choisit aléatoirement k tâches affectées à une même ressource r, et on réalise un échange circulaire sur les valeurs des variables entières représentant leurs dates de début. La variable de liste correspondant à l'ordre des tâches sur la ressource r est mise à jour pour rendre compte de ces échanges.

Exemple 2.5 (Mouvement de 3-échange des dates de début). On considère les tâches t_1 à t_5, affectées à une même ressource r, dont l'ordonnancement est représenté sur la Figure 2.3. On veut réaliser un 3-échange circulaire sur les dates de début des tâches t_2, t_4 et t_5. Le déroulement du mouvement est le suivant : t_2 prend la date de début initiale de t_4, soit 42, puis t_4 prend la date de début initiale de t_5, soit 58, puis t_5 prend la date de début initiale de t_2, soit 11.

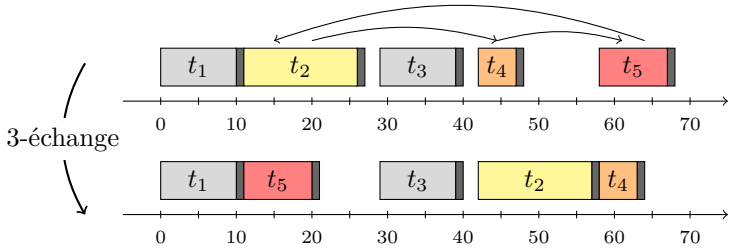


FIGURE 2.3 – Exemple de mouvement de 3-échange des dates de début.

Remplacement. On sélectionne une tâche t_1, de date de début s_1 et de durée d_1, affectée à une ressource r quelconque. On sélectionne une autre tâche, notée t_2, compatible avec r et non affectée dans la solution initiale $\mathcal{S}$. On remplace alors t_1 par t_2 : l'élément `t1` est remplacé par l'élément `t2` dans la variable de liste représentant la ressource r, et les variables entières représentant la date de début et la durée (dans le cas de tâches de durées variables uniquement) de la tâche t_2 prennent respectivement les valeurs s_1 et d_1.

Ce mouvement est utile lorsque les tâches ne sont pas toutes équivalentes, par exemple si elles sont associées à un type particulier (ce dont la contrainte de non-chevauchement qui permet de les détecter ne tient pas compte). Par exemple, on peut considérer une variante du problème du Unit Commitment modélisant la production de différents réactifs dans une usine de produits chimiques, dans laquelle chaque tâche correspond à une plage de production d'un réactif donné. L'application du mouvement de remplacement sur ce problème correspond alors à modifier le réactif produit sur l'une des plages de production.

Exemple 2.6 (Mouvement de remplacement). On considère une ressource r comprenant trois tâches, notées t_1 à t_3. La Figure 2.4 illustre un mouvement de remplacement sur cette ressource :

— La tâche t_2, de type « rouge », est retirée de la ressource r.
— La tâche t_4, de type « vert » et non encore affectée, est placée sur la ressource r avec la même plage d'exécution.

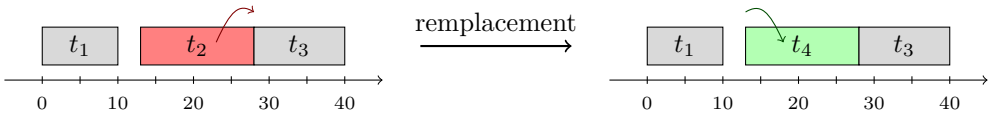


FIGURE 2.4 – Exemple de mouvement de remplacement

Décalage. On sélectionne au hasard une tâche t affectée à une ressource r quelconque, et on la décale en modifiant sa date de début d'une quantité choisie aléatoirement, de façon à toujours respecter la contrainte de non-chevauchement sur la ressource r. En notant t_{prev} et t_{next} les tâches précédant et suivant t respectivement sur la ressource r, on décale ainsi la tâche t en s'assurant que sa date de début reste supérieure à `end[prev] + setup[prev][t]`, et que sa date de fin reste inférieure à `start[next] - setup[t][next]`.

Exemple 2.7 (Mouvement de décalage). On considère les tâches t_1 à t_3, affectées à une ressource r, dont l'ordonnancement est représenté sur la Figure 2.5. On souhaite appliquer le mouvement de décalage sur la tâche t_2. Afin de continuer à respecter les temps de transition avec ses tâches

voisines t_1 et t_3, la date de début de t_2 doit être supérieure ou égale à 9, et inférieure ou égale à 13. On choisit par exemple de décaler t_2 vers la gauche, de sorte que sa nouvelle date de début soit égale à 9.

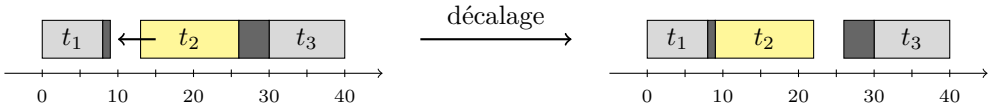


FIGURE 2.5 – Exemple de mouvement de décalage.

2.4.2.3 Mouvements sur les familles de tâches de durées variables

On considère maintenant les mouvements d'ordonnancement s'appliquant sur les familles de tâches de durées variables uniquement, qu'elles soient associées à une ou plusieurs ressources. Les cinq mouvements de cette catégorie sont notés « changement de date de début », « k-échange des durées », « fusion », « fractionnement », et « glissement d'interstice ».

Changement de date de début à date de fin fixée. On sélectionne une tâche t affectée à une ressource r quelconque. Tout en laissant fixée la date de fin de la tâche t, on décale sa date de début de façon à toujours respecter la contrainte de non-chevauchement sur la ressource r.

Bien que ce mouvement soit très simple, il est utile de l'ajouter à notre collection de mouvements d'ordonnancement. En effet, décaler uniquement la date de début d'une tâche, sans modifier sa date de fin, implique de faire des modifications opposées sur deux variables entières, représentant la date de début et la durée de la tâche (contrairement à une modification de la date de fin seulement d'une tâche, qui consiste à modifier une unique variable entière, représentant sa durée). Cette double modification synchronisée ayant peu de chances d'être réalisée au hasard et sans ciblage particulier lors d'un mouvement classique sur les variables entières, il est pertinent qu'elle fasse l'objet d'un mouvement d'ordonnancement.

Exemple 2.8 (Mouvement de changement de date de début à date de fin fixée). On considère les tâches t_1 et t_2, affectées à une ressource r, dont l'ordonnancement est représenté sur la Figure 2.6. On souhaite appliquer le mouvement de changement de date de début à date de fin fixée sur la tâche t_2. Afin de continuer à respecter la contrainte de non-chevauchement sur la ressource r, la date de début de t_2 doit être supérieure ou égale à 10. On choisit par exemple d'étendre la tâche t_2, de sorte que sa nouvelle date de début soit égale à 13 (et sa nouvelle durée égale à 27).

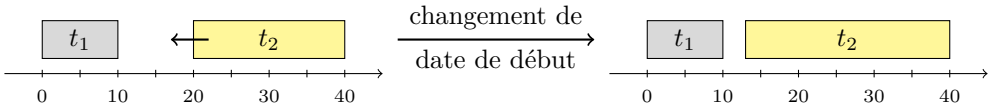


FIGURE 2.6 – Exemple de mouvement de changement de date de début à date de fin fixée.

k-échange des durées. On choisit aléatoirement k tâches affectées à une même ressource r, et on réalise un échange circulaire sur les variables entières représentant leurs durées. L'ordre des tâches ne change pas ; il n'est donc pas nécessaire de modifier la variable de liste associée à la ressource r.

Exemple 2.9 (Mouvement de 3-échange des durées). On considère les tâches t_1 à t_5, affectées à une même ressource r, dont l'ordonnancement est représenté sur la Figure 2.7. On veut réaliser un 3-échange circulaire sur les durées des tâches t_2, t_4 et t_5. Le déroulement du mouvement est le suivant : t_2 prend la durée initiale de t_5, soit 9, puis t_4 prend la durée initiale de t_2, soit 15, puis t_5 prend la durée initiale de t_4, soit 5.

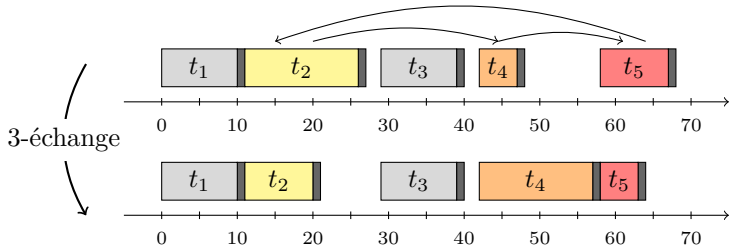


FIGURE 2.7 – Exemple de mouvement de 3-échange des durées.

Fusion. On sélectionne aléatoirement deux tâches consécutives t_1 et t_2 sur une ressource r, et on les fusionne. Afin de ne conserver qu'une seule tâche au lieu de deux, on supprime l'affectation de la tâche t_2 (l'élément `t2` est ainsi supprimé de la variable de liste associée à la ressource r). Les variables entières représentant la date de début et la durée de la tâche t_1, fusionnée avec t_2, sont modifiées afin de conserver deux attributs parmi les trois suivants : date de début initiale de t_1, date de fin initiale de t_2, durée initiale totale de t_1 et t_2.

Exemple 2.10 (Mouvement de fusion). On considère les tâches t_1, t_2 et t_3, affectées à une même ressource r, dont l'ordonnancement est représenté sur la Figure 2.8. On souhaite fusionner les tâches t_1 et t_2. Le déroulement du mouvement est le suivant :

— On supprime l'affectation de la tâche t_2.
— On souhaite par exemple conserver la date de début de t_1 et la durée totale des deux tâches : la durée de t_1 est donc modifiée pour prendre la valeur $10 + 15 = 25$.

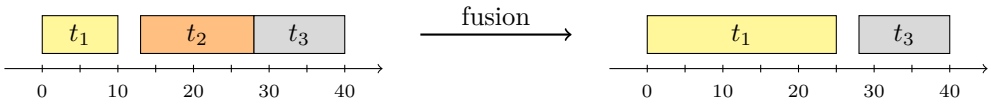


FIGURE 2.8 – Exemple de mouvement de fusion

Fractionnement. On sélectionne une tâche t_1 affectée à une ressource r, et on la sépare en deux. Pour cela, on ajoute une nouvelle tâche t_2 (non affectée dans la solution initiale $\mathcal{S}$) sur la ressource r à la suite de t_1. On ajoute ainsi l'élément `t2` juste après l'élément `t1` dans la variable de liste associée à la ressource r. On modifie ensuite la date de début de t_2, ainsi que les durées de t_1 et t_2, afin que la date de fin de t_2 soit égale à la date de fin initiale de t_1 dans la solution $\mathcal{S}$, et en s'assurant de respecter la contrainte de non-chevauchement sur la ressource r. L'écart entre la date de fin de t_1 et la date de début de t_2 doit ainsi être supérieur ou égal à `setup[t1][t2]`.

Exemple 2.11 (Mouvement de fractionnement). On considère les tâches t_1 et t_2, affectées à une ressource r, dont l'ordonnancement est représenté sur la Figure 2.9. On considère également une tâche t_3 non affectée. Les temps de transition entre t_1 et t_2, et entre t_2 et t_3 sur la ressource r sont respectivement égaux à 2 et 3. On souhaite appliquer le mouvement de fractionnement sur la tâche t_2. Le déroulement du mouvement est le suivant :

— On ajoute la tâche t_3 sur la ressource r à la suite de t_2.
— On coupe la tâche t_2, par exemple à la date 23 : la durée de t_2 passe de 27 à 10.
— On modifie la date de début de t_3 de sorte que celle-ci commence juste après t_2, en respectant le temps de transition : la nouvelle date de début de t_3 est égale à 26.
— On modifie la durée de t_3 de sorte que celle-ci se termine à la date de fin initiale de t_2, soit 40 : la nouvelle durée de t_3 est égale à 14.

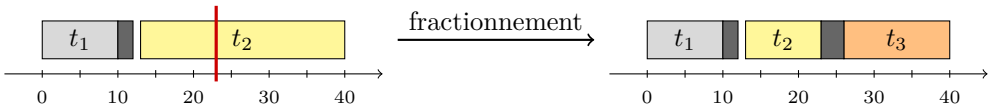


FIGURE 2.9 – Exemple de mouvement de fractionnement.

Glissement d'interstice. On sélectionne deux tâches consécutives t_1 et t_2 au hasard sur une ressource r, et on fait glisser l'interstice entre les deux, en appliquant un décalage opposé à la date de fin de t_1 et à la date de début de t_2.

Exemple 2.12 (Mouvement de glissement d'interstice). On considère les tâches t_1 et t_2, consécutives sur la ressource r, dont l'ordonnancement est représenté sur la Figure 2.10. On leur applique le mouvement de glissement d'interstice : la durée (et donc la date de fin) de t_1 ainsi que la date de début de t_2 sont augmentées de 13, tandis que la durée de t_2 est diminuée de la même quantité.

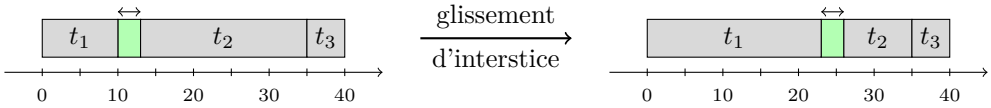


FIGURE 2.10 – Exemple de mouvement de glissement d'interstice.

2.4.2.4 Mouvements sur les familles de tâches multi-ressources

On présente ici les mouvements s'appliquant sur les familles de tâches associées à plusieurs ressources, et composées de tâches à durée fixe ou variable. Cette catégorie comporte deux mouvements, notés « changement de ressource » et « échange de ressources ».

Changement de ressource (avec ajustement). On sélectionne une tâche t, que l'on déplace de sa ressource courante r_1 vers une nouvelle ressource aléatoire r_2, compatible avec t, en conservant sa date de début (et sa durée si celle-ci est variable). L'élément `t` est donc supprimé de la variable de liste associée à la ressource r_1, et ajouté à la liste associée à r_2 à la « bonne » position, c'est-à-dire en garantissant que les tâches sont rangées dans la liste par dates de début croissantes.

Après la réalisation de ce changement de ressource, il est possible que la tâche t chevauche la tâche qui la précède sur la ressource r_2, ou que le temps de transition entre ces deux tâches ne soit pas respecté : la contrainte de non-chevauchement des tâches affectées à la ressource r_2 est violée. Si les tâches ont des durées variables, on cherche alors à réparer cette contrainte : on peut choisir de réduire la tâche t en décalant sa date de début afin d'éviter ce chevauchement, ou de fusionner les deux tâches. Il en est de même en cas de chevauchement avec la tâche suivant t sur la ressource r_2 : on peut choisir de réparer la contrainte en réduisant la tâche t par un décalage de sa date de fin, ou en fusionnant les deux tâches.

Exemple 2.13 (Mouvement de changement de ressource avec ajustement). On considère les ressources r_1 et r_2, représentées sur la Figure 2.11. Les temps de transition entre toutes paires de tâches sur r_1 et r_2 sont respectivement égaux à 1 et 2. On souhaite déplacer la tâche t_2, initialement affectée à la ressource r_1, vers la ressource r_2 : on insère t_2 sur r_2 entre les tâches t'_1 et t'_2. Après avoir réalisé le changement de ressource, on remarque que le temps de transition entre les tâches t_2 et t'_2 n'est pas respecté. On choisit alors de réduire la durée de la tâche t_2 de 11 à 9 afin de décaler sa date de fin.

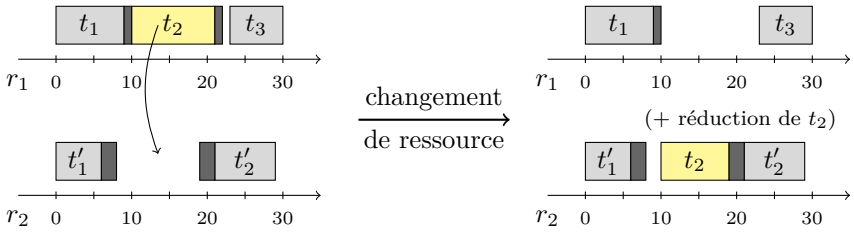


FIGURE 2.11 – Exemple de mouvement de changement de ressource avec ajustement.

Échange de ressources (avec ajustement). On sélectionne aléatoirement deux tâches t_1 et t_2, respectivement affectées aux ressources r_1 et r_2, ayant des plages d'exécution proches, et telles que chacune des deux tâches est compatible avec chacune des deux ressources. On échange les deux tâches : l'élément `t1` présent dans la variable de liste associée à la ressource r_1 est échangé avec l'élément `t2` présent dans la liste associée à r_2.

Comme pour le mouvement de changement de ressource, si la contrainte de non-chevauchement des tâches affectées à la ressource r_1 ou r_2 est violée, et si les tâches sont à durées variables, on cherche à réparer la contrainte. Si la tâche t_1 (resp. t_2) chevauche la tâche qui la précède sur la ressource r_2 (resp. r_1), on peut choisir de la réduire en décalant sa date de début, ou de fusionner les deux tâches. De même, en cas de chevauchement de t_1 (resp. t_2) avec la tâche qui lui succède sur la ressource r_2 (resp. r_1), on peut choisir de la réduire en décalant sa date de fin, ou de fusionner les deux tâches.

Exemple 2.14 (Mouvement d'échange de ressources avec ajustement). On considère les ressources r_1 et r_2, représentées sur la Figure 2.12. On souhaite échanger la tâche t_2, initialement affectée à la ressource r_1, avec une tâche affectée à r_2 et ayant une plage d'exécution proche : on choisit alors la tâche t'_2. Après avoir réalisé l'échange, on remarque que les tâches t_1 et t'_2 se chevauchent sur la ressource r_1, et que les tâches t_2 et t'_3 se chevauchent sur la ressource r_2. On choisit alors de réparer les contraintes de non-chevauchement en décalant la date de début de t'_2 de 10 à 11, et en fusionnant t_2 avec t'_3.

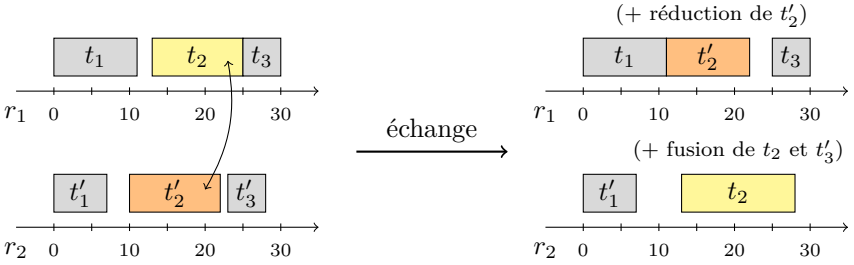


FIGURE 2.12 – Exemple de mouvement d'échange de ressources avec ajustement.

2.4.2.5 Mouvements sur les familles de tâches de durées variables et multi-ressources

Enfin, on présente les mouvements s'appliquant sur les familles de tâches de durées variables, associées à plusieurs ressources. Les deux mouvements de cette catégorie sont notés « fractionnement et changement de ressource » et « k-inversion sur un petit intervalle ».

Fractionnement et changement de ressource. Le mouvement de fractionnement et changement de ressource combine deux autres mouvements précédemment décrits. On sélectionne une tâche t_1 affectée à une ressource r_1, dont les dates de début et de fin sont respectivement notées s_1 et e_1. On sélectionne également une tâche t_2 non affectée dans la solution initiale $\mathcal{S}$, et compatible avec une seconde ressource r_2. On applique le mouvement de fractionnement sur la tâche t_1 : on

choisit une date x entre s_1 et e_1, et on modifie les dates de début et durées des tâches t_1 et t_2 de sorte que t_1 soit désormais ordonnancée entre les instants s_1 et x et t_2 entre les instants x et e_1, ou inversement. On affecte ensuite la tâche t_2 à la ressource r_2, en suivant une procédure similaire à celle du mouvement de changement de ressource.

Exemple 2.15 (Mouvement de fractionnement et changement de ressource). On considère deux ressources r_1 et r_2, représentées sur la Figure 2.13. On applique le mouvement de fractionnement et changement de ressource à la tâche t_1, affectée à la ressource r_1. On sépare t_1 en deux au niveau de l'instant $x = 16$: la durée de t_1 est raccourcie, passant de 25 à 16, et on considère une nouvelle tâche t_3, initialement non affectée, pour laquelle on choisit une date de début et une durée respectivement égales à 16 et 9. On place alors la tâche t_3 sur la ressource r_2. Les tâches t_3 et t'_2 se chevauchent sur la ressource r_2 : on les fusionne pour assurer le respect de la contrainte de non-chevauchement.

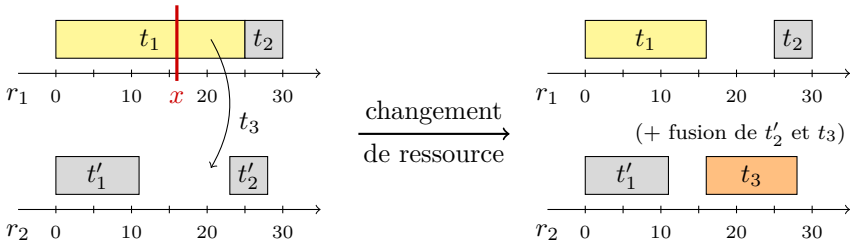


FIGURE 2.13 – Exemple de mouvement de fractionnement et changement de ressource

k-inversion sur un petit intervalle. On choisit aléatoirement k ressources ainsi qu'un petit intervalle de temps I (appartenant à la plage d'exécution d'une tâche prise aléatoirement sur l'une des ressources sélectionnées). On « inverse » alors le fonctionnement de chacune de ces ressources sur cet intervalle de temps. Pour une ressource majoritairement en état de fonctionnement, c'est-à-dire comprenant une tâche en cours d'exécution sur cet intervalle, alors la ressource doit être au repos. Si l'exécution de cette tâche dépasse peu l'intervalle de temps choisi, on la supprime (on supprime l'élément correspondant dans la variable de liste associée à la ressource). Sinon, on se contente de décaler sa date de début, ou de décaler sa date de fin, ou de la fractionner en deux tâches de part et d'autre de l'intervalle I (on suit alors une procédure similaire à celle du mouvement de fractionnement). Pour une ressource majoritairement au repos, on ajoute une nouvelle tâche sur l'intervalle de temps choisi, que l'on peut éventuellement fusionner avec ses tâches voisines pour assurer le respect de la contrainte de non-chevauchement.

Exemple 2.16 (Mouvement de 3-inversion sur un petit intervalle). On considère les ressources r_1, r_2 et r_3, illustrées sur la Figure 2.14, et on suppose que la durée minimale d'une tâche vaut 3. On applique sur les trois ressources choisies le mouvement d'inversion sur un petit intervalle. On choisit pour cela l'intervalle $I = [11, 15]$. La ressource r_1 étant initialement en fonctionnement sur la totalité de l'intervalle I, elle doit désormais y être au repos : on fractionne alors la tâche t_2 en deux petites tâches, de part et d'autre de I. La ressource r_2 est quant à elle majoritairement au repos dans l'intervalle I : on étend alors la tâche t'_1 de sorte que celle-ci se termine à la fin de l'intervalle I. Enfin, la ressource r_3 doit être mise au repos. La durée minimale d'une tâche étant de 3, on ne peut se contenter de réduire la tâche t''_2 : on la supprime.

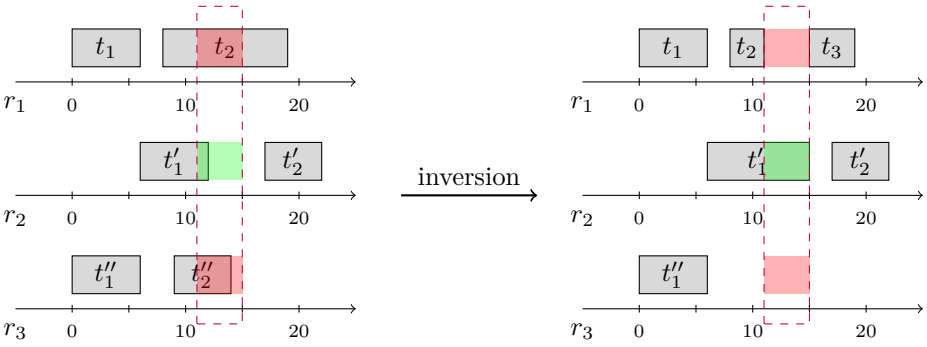


FIGURE 2.14 – Exemple de mouvement de 3-inversion sur un petit intervalle.

2.4.3 Résultats numériques

Dans cette Section, on s'intéresse aux améliorations de performance apportées par l'ajout au sein de la recherche locale de LocalSolver des mouvements présentés dans la Section 2.4.2. On présente ainsi des résultats numériques obtenus en activant ou en désactivant ces mouvements, sur différents problèmes d'ordonnancement disjonctif dont la définition est donnée dans les Sections 1.3.1 et 2.3.2.

Problème du Job Shop. Le problème du Job Shop comporte des tâches de durées fixes, affectées à une machine particulière, et non optionnelles. Deux des mouvements décrits à la Section 2.4.2 s'appliquent donc sur ce problème : les mouvements de k-échange des dates de début et de décalage. On évalue les performances de LocalSolver sur quatre classes d'instances classiques du problème du Job Shop : la classe FT de Fisher et Thompson [40], la classe LA de Lawrence [62], la classe ORB d'Applegate et Cook [6], et la classe TAI de Taillard [92].

Problème du Job Shop flexible. Le problème du Job Shop flexible comporte des tâches de durées fixes, pouvant être affectées à plusieurs machines, et non optionnelles. Quatre des mouvements décrits à la Section 2.4.2 s'appliquent donc sur ce problème : les mouvements de k-échange des dates de début, de décalage, de changement de ressource et d'échange de ressources. On évalue les performances de LocalSolver sur le problème du Job Shop flexible sur les instances de Barnes, Behnke, Brandimarte, Dauzere, Fattahi, Kacem et Hurink. On compare les résultats obtenus à la meilleure solution connue, donnée dans [14].

Problème du Job Shop flexible avec temps de transition. Les mouvements s'appliquant sur le problème du Job Shop flexible avec temps de transition sont les mêmes que pour le problème du Job Shop flexible. On évalue les performances de LocalSolver sur ce problème à partir des instances de Job Shop flexible, auxquelles on ajoute des temps de transition aléatoires, dont l'ordre de grandeur varie entre 10% et la valeur moyenne des durées et cinq fois cette valeur. Ces instances ayant été générées aléatoirement, on ne dispose pas de la valeur de la solution optimale, ou de la meilleure solution connue selon la littérature : on compare donc nos résultats à ceux obtenus par le solveur de programmation par contraintes CP Optimizer 20.1.0 après 60 secondes de calcul.

Problème du Unit Commitment simplifié. Le problème du Unit Commitment simplifié comporte des tâches optionnelles, de durées variables, et pouvant être affectées à plusieurs ressources. L'ensemble des treize mouvements décrits à la Section 2.4.2 s'appliquent donc sur ce problème. On évalue les performances de LocalSolver sur ce problème à partir des instances de [4] : le nombre d'usines varie de 10 à 100, et le nombre de pas de temps est de 24. Le problème étudié ici est cependant une version simplifiée et purement combinatoire du problème du Unit Commitment. On

ne dispose donc pas de la valeur de la solution optimale, ni d'une meilleure solution connue de la littérature.

L'ensemble des résultats numériques donnés ici ont été calculés sur une machine équipée d'un processeur Intel i7 8750 à 2.20 GHz et de 16 Go de mémoire RAM. Ils sont présentés dans la Table 2.4. Pour chaque problème considéré, on compare les performances de LocalSolver en activant et en désactivant l'ensemble des mouvements d'ordonnancement pouvant s'appliquer sur ce problème, en 60 secondes de calcul. Pour chaque catégorie d'instances, on donne la moyenne du pourcentage d'amélioration obtenu grâce à l'ajout de ces mouvements, ainsi que la valeur moyenne de l'écart à la meilleure solution connue avec et sans ces mouvements.

Problème	Instances	Nombre d'instances	Amélioration	Gap avec mouvements	Gap sans mouvements
Job Shop	FT	3	0.1%	1.2%	1.3%
	LA	40	0.1%	1.4%	1.4%
	ORB	10	0.5%	1.9%	2.4%
	TAI	80	0.0%	7.6%	7.6%
Job Shop flexible	Barnes	21	-0.2%	4.4%	4.2%
	Behnke	60	2.6%	2.5%	5.1%
	Brandimarte	10	1.1%	5.0%	6.2%
	Dauzere	18	0.3%	5.5%	5.8%
	Fattahi	20	0.5%	0.6%	1.2%
	Kacem	4	0.0%	0.0%	0.0%
	Hurink – e	66	0.0%	3.5%	3.5%
	Hurink – r	66	0.3%	2.8%	3.1%
	Hurink – v	66	0.1%	0.3%	0.4%
Job Shop flexible avec temps de transition	$\simeq 10\%$ durées	104	0.4%	2.8%	3.3%
	$\simeq$ durées	105	0.2%	1.8%	2.0%
	$\simeq 5\times$ durées	104	0.0%	2.5%	2.5%
Unit Commitment	24×10	30	2%	–	–
	24×20	27	7%	–	–
	24×50	25	10%	–	–
	24×75	19	11%	–	–
	24×100	25	14%	–	–
	24×150	21	34%	–	–

TABLE 2.4 – Comparaison des performances de LocalSolver avec et sans les mouvements sur les familles de tâches

Sur les problèmes d'ordonnancement d'atelier, on constate que l'ajout de ces quelques mouvements (deux mouvements pour le problème du Job Shop, quatre mouvements pour les problèmes du Job Shop flexible avec ou sans temps de transition) permet d'encore améliorer les performances déjà très bonnes de LocalSolver. Sur le problème du Unit Commitment simplifié, présentant plus de degrés de liberté et sur lequel on peut alors appliquer une plus grande variété de mouvements, on constate que cet ajout apporte un gain de performance très significatif, particulièrement sur les instances de plus grande taille.

2.5 Conclusion

Dans ce Chapitre, on a montré comment l'utilisation combinée des variables de décision entières et de listes permet de modéliser efficacement de nombreux problèmes d'ordonnancement disjonctif. D'une part, les variables entières permettent classiquement de représenter les dates de début

des tâches, ainsi que leurs durées lorsque celles-ci sont variables. Les variables de listes permettent d'autre part de représenter l'ordre de ces tâches sur les différentes ressources disjonctives. L'utilisation de ces deux types de variables au sein du même modèle facilite la modélisation de nombreuses expressions, comme notamment les contraintes de non-chevauchement des tâches affectées à une même ressource disjonctive. Ces contraintes sont alors modélisées grâce à des lambda-fonctions au sein de fonctions de type « et » variadique, et s'expriment de la façon suivante : « pour toute valeur de i, la tâche en position $i + 1$ sur la ressource doit commencer après la fin de la tâche en position i sur cette ressource ». Cette formulation générique des contraintes de non-chevauchement des tâches permet d'écrire une modélisation simple et concise pour de nombreux problèmes d'ordonnancement d'atelier (Job Shop, Job Shop flexible, Open Shop), mais également pour des problèmes de planification de production (Unit Commitment).

La modélisation LocalSolver des problèmes d'ordonnancement disjonctif présente ainsi des structures de variables entières ordonnées par une ou plusieurs variables de listes, qui peuvent être exploitées au sein du solveur pour permettre une résolution plus efficace. Tout d'abord, les bornes de ces variables entières, lorsqu'elles sont différentes, permettent de donner une valeur initiale cohérente à la variable de liste représentant leur ordre. Cette initialisation efficace des variables de listes permet d'obtenir une première solution réalisable très rapidement sur des problèmes comme celui de l'Aircraft Landing. Par ailleurs, la présence de contraintes de non-chevauchement permet de mettre en évidence la présence de tâches et de ressources disjonctives dans le modèle. On peut alors ajouter des mouvements adaptés à l'ordonnancement au sein de la recherche locale de LocalSolver. Les mouvements ainsi créés modifient à la fois les valeurs des variables entières et des variables de listes. Ils sont pensés pour maintenir la faisabilité des contraintes de non-chevauchement des tâches, et ont ainsi plus de chances d'aboutir à des solutions réalisables que les mouvements classiques ne modifiant qu'un seul type de variables à la fois. On montre que l'implémentation de ces mouvements au sein de la recherche locale de LocalSolver permet d'améliorer ses performances sur des problèmes d'ordonnancement très différents, comme ceux du Job Shop ou du Unit Commitment.

Chapitre 3

Réparation de solutions par propagation de réseaux d'inégalités

3.1 Introduction

Dans ce Chapitre, on s'intéresse aux problèmes comportant un réseau de contraintes définies par des inégalités linéaires entre deux ou trois variables, ou par des disjonctions ou chaînes de telles inégalités. Ce type de réseau de contraintes binaires ou ternaires est caractéristique de nombreux problèmes d'ordonnancement. En effet, ces expressions peuvent servir à modéliser les contraintes de précédence entre les tâches, ou de non-chevauchement des tâches présentes sur une même ressource disjonctive, souvent rencontrées en ordonnancement, comme par exemple dans le problème du Job Shop [40]. Toutefois, ce type de structure est également typique des problèmes de packing, de layout ou encore d'extraction minière.

Ces problèmes sont très contraints : dans une bonne solution d'une instance du problème du Job Shop, les contraintes de précédence et de ressources disjonctives sont souvent très serrées. De ce fait, passer d'une solution de makespan x à une solution de makespan $x-1$ nécessite de nombreux petits changements sur les dates de début des tâches. Cette situation est illustrée sur la Figure 3.1 : on considère une petite instance du problème du Job Shop, comportant quatre jobs (chaque couleur correspondant à un job), et quatre machines (chaque ligne correspondant à une machine). On constate que pour passer de la solution initiale de makespan 40, représentée à gauche, à la solution de makespan 39 représentée à droite, il a fallu modifier la date de début de six tâches parmi les seize tâches composant le problème.

Pouvoir passer d'une bonne solution réalisable à une autre en utilisant une recherche locale à voisinages restreints aléatoires est alors très improbable : il faudrait cibler de manière aléatoire le bon ensemble de variables entières, représentant les dates de début des tâches, et les décaler toutes de la bonne quantité. Or, les algorithmes de la composante de recherche locale de LocalSolver, décrits dans [15] et [43], sont principalement basés sur des voisinages restreints (inversions, échanges, décalages, insertions, ...). Pour ces raisons, une recherche locale de ce type peut avoir des difficultés à converger vers de bonnes solutions.

Dans la vaste littérature de l'ordonnancement d'atelier par recherche locale (par exemple [13], [77] et [94]), ces difficultés sont surmontées en exploitant des représentations de solutions dédiées de niveau supérieur, comme le graphe disjonctif. Dans ce travail, on ne cherche cependant pas simplement à construire une recherche locale efficace sur les problèmes d'ordonnancement d'atelier comme celui du Job Shop, mais à améliorer les performances du solveur LocalSolver sur une large

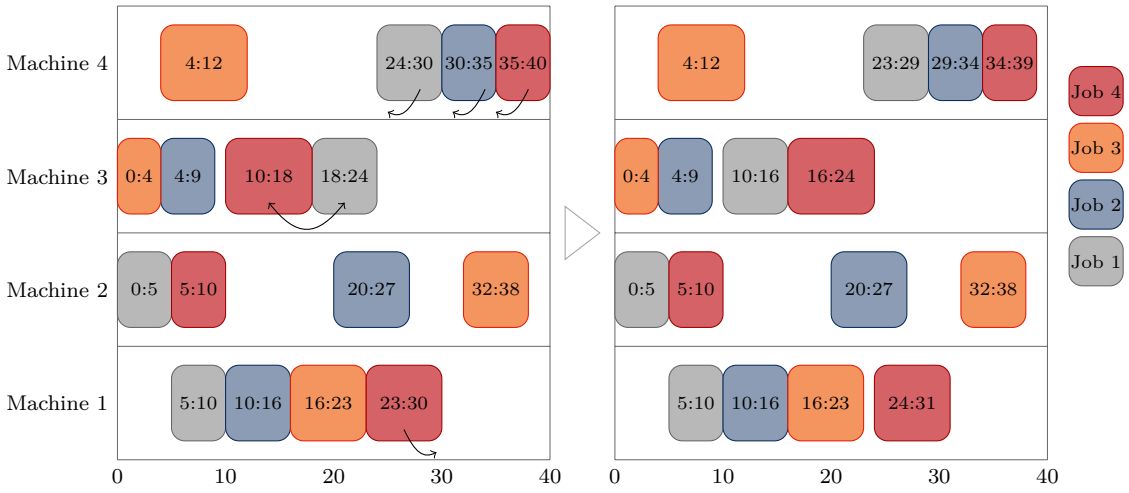


FIGURE 3.1 – Problème du Job Shop – Passage d'une solution de makespan 40 à une solution de makespan 39

gamme de problèmes d'optimisation. On souhaite ainsi conserver des éléments de modélisation simples, pour élaborer une méthode de résolution adaptable et généralisable au plus grand nombre de problèmes. On choisit donc de se concentrer sur la représentation directe du problème, à base de variables de décision entières représentant les dates de début des tâches.

Pour pallier les difficultés de la recherche locale sur les problèmes visés, on élabore un algorithme de réparation de solutions basé sur la propagation de contraintes : une solution prometteuse mais infaisable est réparée progressivement, une contrainte à la fois. Cette procédure graduelle peut être comparée aux algorithmes de chaînes d'éjection, initialement proposés par Glover (1996) [46] pour générer des voisinages de mouvements composés pour le problème du voyageur de commerce, et étudiés plus récemment par Ding *et al.* (2019) [37] dans le contexte des problèmes d'ordonnancement. Ces méthodes donnent d'excellents résultats grâce à des mouvements puissants qui améliorent la recherche locale. Notre mécanisme de réparation peut également être comparé à l'heuristique min-conflicts introduite par Minton *et al.* (1992) [66] pour résoudre les problèmes de satisfaction de contraintes (CSP), et plus récemment appliquée au domaine de l'ordonnancement par Ahmeti et Musliu (2018) [3]. Cette méthode consiste à créer une affectation complète mais inconsistante pour les variables d'un CSP, et à réparer les violations de contraintes jusqu'à ce que l'affectation soit consistante. L'heuristique min-conflicts consiste à sélectionner une variable impliquée dans une contrainte violée, et à lui donner pour valeur celle qui minimise le nombre de violations de contraintes restantes.

Ce Chapitre est organisé de la façon suivante. La Section 3.2 présente de façon formelle le mécanisme de réparation mis en place au sein de LocalSolver. La méthode décrite ici est plus simple que celles mentionnées ci-dessus, puisque les contraintes violées sont réparées dans l'ordre où elles sont rencontrées, en modifiant les valeurs des variables impliquées juste assez pour réparer la contrainte courante. Cependant, elle donne des résultats satisfaisants en pratique, possède de fortes propriétés théoriques et a l'avantage d'être très rapide, ce qui est crucial pour pouvoir l'intégrer dans un solveur performant. Dans les Sections 3.3 et 3.4, l'algorithme est appliqué aux contraintes binaires sur les variables booléennes et numériques respectivement. Il est montré que certaines contraintes ont des propriétés très fortes garantissant la réussite du mécanisme de réparation, et une caractérisation de la forme de ces contraintes est donnée. Dans la Section 3.5, certaines contraintes plus complexes sont introduites : les disjonctions et les chaînes, pour lesquelles le mécanisme de réparation est adapté. La Section 3.6 est dédiée aux contraintes ternaires. Enfin, des résultats numériques sont détaillés dans la Section 3.7. Notre méthode est appliquée

aux problèmes du Job Shop (flexible ou non, avec ou sans temps de transition), de l'Open Shop et du Unit Commitment, pour lesquels la procédure de réparation améliore considérablement les performances de nos algorithmes de recherche locale. Les contributions de ce Chapitre ont donné lieu à la publication d'un article pour la conférence PPSN 2020 [17], et ont été présentées lors des conférences PMS 2020-2021 [18] et ROADEF 2020 [16].

3.2 Mécanisme de réparation : définitions et algorithme général

Dans cette Section, on donne la description formelle du mécanisme de réparation mis en œuvre dans LocalSolver pour compléter ses algorithmes de recherche locale, qui consiste à réparer progressivement une solution infaisable, une contrainte à la fois.

Définition 3.1 (Contrainte). *Une contrainte est une relation sur les variables d'un problème d'optimisation qu'une solution doit satisfaire. Elle se caractérise par son ensemble réalisable, qui est un sous-ensemble du produit cartésien des domaines de ses variables. L'ensemble des variables impliquées dans une contrainte $\mathcal{C}$ est noté $var(\mathcal{C})$.*

Dans la suite du Chapitre, lorsque le contexte est clair, le terme « contrainte » pourra désigner soit l'équation de définition de la contrainte, soit son ensemble réalisable.

On considère une itération quelconque de la recherche locale, et on suppose que cette itération commence à partir d'une solution initiale réalisable, notée $\mathcal{S}_0$, dans laquelle chaque variable X a une valeur initiale x_0, et un domaine initial $\mathcal{D}_X = [\underline{x}, \overline{x}]$. Une transformation locale est appliquée à $\mathcal{S}_0$: on a désormais une solution $\mathcal{S}$, que l'on suppose infaisable. La valeur de chaque variable X dans la solution actuelle $\mathcal{S}$ est notée $x \in \mathcal{D}_X$.

Propriété 3.1 (Respect des décisions précédentes). Afin que le mécanisme de réparation étende la transformation locale plutôt que de l'annuler, on impose de ne jamais revenir sur une décision antérieure. En d'autres termes, si une variable a déjà été modifiée (au cours de la transformation locale, ou lors de la réparation d'une contrainte), elle peut être modifiée à nouveau dans le même sens, mais elle ne peut pas être modifiée dans le sens opposé. Ainsi, une modification de la valeur d'une variable équivaut à une réduction de domaine : lorsqu'elle augmente (resp. diminue), sa borne inférieure (resp. supérieure) est ajustée en conséquence.

En raison de la Propriété 3.1, la phase de réparation est équivalente à une sorte de propagation de contraintes particulière, dont l'algorithme de filtrage sera noté « demi borne consistance » (HBC pour « *half bound consistency* ») dans le reste du Chapitre. En effet, le domaine initial $\mathcal{D}_X = [\underline{x}, \overline{x}]$ de chaque variable X sera réduit par des modifications successives d'une seule de ses bornes tout au long de la propagation. La première modification de la valeur x de X (dans la transformation locale, ou pendant la propagation) détermine laquelle des bornes de X verra ses modifications se propager. Par conséquent, tout au long de l'itération, le domaine non vide d'une variable X est toujours de la forme $[\underline{x}, b]$ ou $[b, \overline{x}]$, avec $b \in [\underline{x}, \overline{x}]$.

On décrit maintenant la procédure de réparation, également fournie sous forme de pseudo-code dans l'Algorithme 3.1. Après avoir effectué la transformation locale, toutes les contraintes impliquant une variable qui vient d'être modifiée (variables X telles que $x \neq x_0$) sont mises dans la file de propagation.

Tant que la file n'est pas vide, on applique l'algorithme de filtrage HBC introduit ci-dessus (voir lignes 5 à 19 de l'Algorithme 3.1) sur sa contrainte de tête $\mathcal{C}$, de la façon suivante. Si $\mathcal{C}$ est déjà vérifiée, elle est ignorée, et on passe directement à la contrainte suivante. Sinon, on calcule un nouveau domaine candidat pour chaque variable $X \in var(\mathcal{C})$, dont les bornes plus étroites sont choisies pour assurer sa consistance avec $\mathcal{C}$. Si au moins un de ces domaines réduits est vide, alors $\mathcal{C}$ ne peut pas être réparée : la propagation échoue, et les contraintes restant dans la file de

propagation sont ignorées. Sinon, chaque variable $X \in var(\mathcal{C})$ se voit attribuer une nouvelle valeur x'. Si x est toujours dans le domaine réduit de X, alors $x' = x$. Sinon, x' est la valeur du domaine réduit de X qui est la plus proche de x : il s'agit de l'une des bornes réduites de X. Ainsi, x' est la projection de x sur le domaine réduit de X.

On considère alors la solution $\mathcal{S}'$, dans laquelle chaque variable $X \in var(\mathcal{C})$ prend la valeur x'. Si cette solution satisfait $\mathcal{C}$, les réductions de domaine se propagent : pour chaque variable $X \in var(\mathcal{C})$ telle que $x \neq x'$, sa nouvelle borne x' est propagée (son autre borne étant laissée inchangée), et toutes les autres contraintes impliquant X sont ajoutées à la file de propagation. Si $\mathcal{S}'$ viole la contrainte, alors il existe plusieurs façons de la réparer, et plusieurs façons de choisir une nouvelle valeur valide pour les variables, dont aucune n'est *a priori* meilleure que les autres. On détaille la manière dont cette situation est traitée dans les Sections suivantes. Dans les Sections 3.3 et 3.4, on montre que certaines classes de contraintes peuvent toujours être réparées par projection des variables sur leurs domaines réduits lorsqu'il est possible de les réparer. Dans les Sections 3.5 et 3.6, on explique comment certaines contraintes plus complexes, n'étant pas toujours réparables par simple projection, sont réparées de façon aléatoire.

Lorsque la file de propagation est vide, soit la propagation échoue car il n'existe aucune solution réalisable respectant les décisions de la transformation locale, et l'algorithme revient à sa solution initiale $\mathcal{S}_0$, soit une solution réalisable est trouvée. Dans ce dernier cas, pour chaque variable, si son domaine a été réduit, alors chaque réduction de domaine subie au cours de la propagation correspond à une modification de la même borne, et sa valeur actuelle est égale à cette borne modifiée.

Algorithme 3.1 Une itération de la recherche locale

Entrée : Solution initiale réalisable $\mathcal{S}_0$ (valeur initiale de chaque variable X notée x_0)
Transformation locale sur $\mathcal{S}_0$: solution courante $\mathcal{S}$ (valeur de chaque variable X notée x)
$q \leftarrow \{\mathcal{C} : \exists X \in var(\mathcal{C}), x \neq x_0\}$
tant que $q \neq \emptyset$ **faire**
  $\mathcal{C} \leftarrow q.pop()$
  Application de l'algorithme de filtrage HBC sur $\mathcal{C}$
  si $\mathcal{C}$ vérifiée **alors**
    continue
  fin si
  $\forall X \in var(\mathcal{C})$, $\mathcal{D}'_X \leftarrow$ réduction de $\mathcal{D}_X$ par rapport à $\mathcal{C}$
  $\forall X \in var(\mathcal{C})$, $x' \leftarrow$ projection de x sur $\mathcal{D}'_X$: solution $\mathcal{S}'$
  si $\exists X,\ \mathcal{D}'_X = \emptyset$ **alors**
    échec *$\mathcal{C}$ ne peut pas être réparée*
    break
  sinon si $\mathcal{S}'$ non réalisable **alors**
    voir Sections 3.3, 3.4, 3.5, 3.6 *plusieurs façons de réparer $\mathcal{C}$*
  sinon
    $q \leftarrow q \cup \{\mathcal{C}' \neq \mathcal{C} :\ \exists X \in var(\mathcal{C}) \cap var(\mathcal{C}'),\ x \neq x'\}$
    $\forall X \in var(\mathcal{C})$, $x' > x$ (resp. $x' < x$), mise à jour de la borne inf. (resp. sup.) x' de X
    $\mathcal{S} \leftarrow \mathcal{S}' :\ \forall X \in var(\mathcal{C}),\ x \leftarrow x'$
  fin si
fin tant que
si échec **alors**
  retour à $\mathcal{S}_0$
fin si

3.3 Réparation de contraintes binaires portant sur des variables booléennes

On considère tout d'abord sur le cas simple des contraintes binaires portant sur des variables booléennes (*leq*, *nand*, *or*, *xor*) :

$$X \leq Y; \quad X + Y \leq 1; \quad X + Y \geq 1; \quad X + Y = 1$$

Lorsqu'on considère des variables booléennes, la Propriété 3.1, qui interdit de revenir sur les décisions prises pendant l'itération courante, implique que chaque variable ne peut être modifiée qu'une seule fois au cours de chaque itération de la recherche locale. Si l'on ne considère que des contraintes binaires portant sur des variables booléennes, cela implique que chaque fois qu'une contrainte violée est propagée, il existe au plus une façon de la réparer. En effet, puisque la contrainte se trouve dans la file de propagation, au moins une de ses variables doit déjà avoir été modifiée. Dans ce cas, cette variable ne peut pas être modifiée à nouveau, et la contrainte ne peut être réparée qu'en modifiant la valeur de l'autre variable.

Cette procédure est similaire à une itération de l'algorithme de *limited backtracking* pour le problème 2-SAT, décrit dans [39]. Cet algorithme se déroule de la façon suivante. Initialement, aucune variable n'a de valeur. Tant qu'il existe une variable sans valeur, et qu'on n'a pas prouvé que le problème n'était pas satisfiable, on répète l'étape suivante :

- S'il existe une clause dont exactement une des variables a déjà une valeur, et qu'elle ne permet pas de satisfaire la clause, alors on assigne à l'autre variable la valeur lui permettant de satisfaire la clause.
- S'il existe une clause non satisfaite mais dont les deux variables ont déjà une valeur, alors on revient en arrière, et on annule toutes les affectations réalisées jusqu'au dernier choix arbitraire que l'on a fait pour assigner une valeur à une variable. Si l'on n'a aucun choix arbitraire à annuler, alors le problème n'est pas satisfiable.
- Sinon, on doit faire un choix arbitraire pour continuer à affecter des valeurs aux variables : on choisit au hasard une variable sans valeur, et on lui affecte une valeur aléatoire.

Lorsque l'on ne considère que des contraintes binaires portant sur des variables booléennes, l'algorithme de réparation de solutions par propagation décrit à la Section 3.2 peut être assimilé à une phase de propagation de l'algorithme de *limited backtracking*. On part d'une solution réalisable $\mathcal{S}_0$, dans laquelle chaque variable X a une valeur x_0 provisoire. On fait ensuite quelques choix arbitraires lors de la transformation locale, affectant à certaines variables une valeur définitive. On commence alors la phase de réparation des contraintes. Lorsque l'on applique l'algorithme de filtrage HBC sur une contrainte violée, deux configurations se présentent. Si seule l'une des variables a déjà été modifiée, et possède ainsi une valeur définitive, alors on assigne à l'autre variable la valeur définitive permettant de respecter la contrainte. Si les deux variables ont une valeur définitive, alors la contrainte ne peut pas être réparée, et la propagation échoue. Dans ce cas, la différence entre l'algorithme de limited backtracking et notre algorithme de réparation de solutions est que l'on ne prend jamais de décisions arbitraires au cours de la propagation, que l'on pourrait annuler en rencontrant une contrainte impossible à réparer. En effet, ces décisions arbitraires ont été prises lors de la transformation locale, que l'on choisit de ne pas reconsidérer (Propriété 3.1 de respect des décisions précédentes).

Proposition 3.1. *Si l'on ne considère que des contraintes binaires portant sur des variables booléennes, et s'il existe une solution réalisable qui respecte les décisions de la transformation locale, alors l'algorithme de réparation est garanti de la trouver.*

Schéma de preuve. Cela résulte de la similitude entre notre algorithme de réparation sur des contraintes binaires booléennes et l'algorithme de *limited backtracking* (voir [39]). On ne fait aucun choix arbitraire, excepté lors de la transformation locale. Les seules modifications faites lors de la

propagation sont nécessaires à l'obtention d'une solution réalisable. □

Exemple 3.1. On considère un petit problème avec trois variables booléennes X, Y et Z, et trois contraintes binaires booléennes $\mathcal{C}_1 : X + Y \leq 1$, $\mathcal{C}_2 : Y + Z \geq 1$ et $\mathcal{C}_3 : X \geq Z$. On suppose que la solution initiale réalisable $\mathcal{S}_0$ est telle que $x_0 = 0$, $y_0 = 1$ et $z_0 = 0$. On suppose qu'après la transformation locale, la solution actuelle $\mathcal{S}$ est infaisable et vérifie $x = 1$, $y = y_0 = 1$ et $z = z_0 = 0$. La propagation se déroule de la façon suivante :

— Modification de X (transformation locale) $\Rightarrow q = \{\mathcal{C}_1, \mathcal{C}_3\}$
— Propagation de $\mathcal{C}_1 : X + Y \leq 1$.
Réparation par modification de $Y : y = 0$ $\Rightarrow q = \{\mathcal{C}_3, \mathcal{C}_2\}$
— Propagation de $\mathcal{C}_3 : X \geq Z$. Contrainte déjà vérifiée $\Rightarrow q = \{\mathcal{C}_2\}$
— Propagation de $\mathcal{C}_2 : Y + Z \geq 1$.
Réparation par modification de $Z : z = 1$ $\Rightarrow q = \{\mathcal{C}_3\}$
— Propagation de $\mathcal{C}_3 : X \geq Z$. Contrainte déjà vérifiée $\Rightarrow q = \emptyset$

On trouve ainsi une solution réalisable : $x = 1$, $y = 0$, $z = 1$.

3.4 Réparation de contraintes binaires portant sur des variables numériques

On considère désormais les contraintes binaires portant sur des variables numériques. On commence par décrire l'expression des contraintes binaires spécifiques réellement propagées dans LocalSolver. On donne ensuite des propriétés plus générales sur la forme des contraintes binaires vérifiant certaines propriétés utiles, garantissant une propagation efficace.

3.4.1 Inégalités linéaires binaires portant sur des variables numériques

En plus des contraintes binaires booléennes, on considère désormais les inégalités de la forme

$$aX + bY \leq c \tag{3.1}$$

où X et Y sont des variables de décision entières ou réelles, et a, b et c sont des constantes.

Remarque 3.1. Le cas particulier où $|a| = 1$ et $b = -a$ correspond aux contraintes de précédence généralisées rencontrées dans les problèmes d'ordonnancement. En effet, si l'on considère une tâche t, dont la date de début et la durée sont respectivement modélisées par la variable de décision entière S_t et la constante d_t, ainsi qu'une tâche t', dont la date de début est modélisée par la variable de décision entière $S_{t'}$, alors la contrainte de précédence entre t et t' peut s'écrire

$$S_{t'} \geq S_t + d_t \iff S_t - S_{t'} \leq -d_t$$

ce qui correspond bien à la forme de l'Equation 3.1.

Détection des inégalités linéaires binaires à propager. Avant de commencer la recherche et de s'intéresser à la réparation des contraintes violées, la phase de détection des contraintes que l'on souhaite propager est cruciale. En effet, la forme canonique des inégalités linéaires binaires, décrite par l'Equation 3.1, peut en pratique correspondre à de nombreuses écritures différentes dans le modèle : les deux variables de décision peuvent être du même côté de l'inégalité ou non, on peut utiliser le signe « inférieur ou égal » ou « supérieur ou égal », la constante c peut être unique ou séparée en plusieurs composantes, etc. Dans les structures internes de LocalSolver, chacune de ces écritures se traduit par un graphe d'évaluation [1] particulier, à partir duquel on doit déterminer

1. Voir Section 1.2.2

si la contrainte correspond à l'expression recherchée (ici, une inégalité linéaire binaire) ou non. Deux écritures très différentes d'une même contrainte conduisent à deux graphes d'évaluation très différents également. Afin de garantir une réparation efficace quelle que soit la modélisation choisie, on doit donc s'assurer d'être très robuste sur la détection des contraintes correspondant à des inégalités linéaires binaires, ainsi que de toutes les autres contraintes que l'on est capable de propager.

Pour cela, lors de la phase de détection précédant le début de la recherche, on analyse chaque contrainte du modèle portant sur un nœud dont le type correspond à une comparaison (nœuds de type « *leq* », « *geq* », « *lt* » ou « *gt* »). L'algorithme de détection ainsi appliqué consiste en une exploration récursive du sous-graphe de la contrainte, durant laquelle on construit progressivement une reformulation de la contrainte sous forme canonique. Tant que l'on ne rencontre aucun nœud rendant la contrainte incohérente avec la forme canonique que l'on cherche à reconnaître, compte tenu de ce qui a déjà été détecté, l'exploration continue. Lorsque tout le sous-graphe a été exploré, on vérifie que l'expression détectée correspond effectivement à la forme canonique de l'Equation 3.1 (bon nombre de variables de décision, variables de décision toutes différentes, etc.).

Exemple 3.2 (Détection d'une contrainte de précédence). Afin d'illustrer la nécessité d'une détection robuste, on considère l'exemple de l'écriture d'une contrainte de précédence entre deux tâches. On commence par définir trois tableaux, notés `duration`, `start` et `end`, tels que pour toute tâche t, `duration[t]` est une constante représentant la durée de t, `start[t]` est une variable de décision entière représentant sa date de début, et `end[t]` est une expression entière représentant sa date de fin. Le début de modèle correspondant est donné dans le Modèle 3.1.

```
// Lecture des données.
duration[t in 0..nbTasks-1];
// Variables de décision.
start[t in 0..nbTasks-1] <- int(0, maxStart);
// Expressions intermédiaires.
end[t in 0..nbTasks-1] <- start[t] + duration[t];
```

Modèle 3.1 – Exemple de modélisation de contrainte de précédence – Mise en place

On souhaite écrire une contrainte de précédence entre deux tâches t_1 et t_2. Il existe de nombreuses façons d'écrire cette contrainte. On en présente quatre, plus ou moins proches de l'Equation 3.1, et on donne pour chacune une représentation du graphe d'évaluation correspondant (Figures 3.2 à 3.5). Bien que la contrainte soit simple, on constate que les graphes d'évaluation correspondant aux différentes écritures possibles peuvent prendre des formes variées (entre 4 et 6 nœuds, de types différents).

— Première version :

```
constraint start[t1] - start[t2] <= -duration[t1];
```

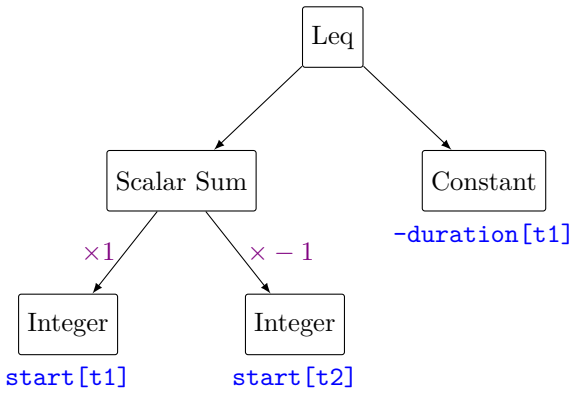


FIGURE 3.2 – Graphe d'évaluation de la contrainte de précédence – première version

— Deuxième version :

```
constraint start[t2] >= end[t1];
```

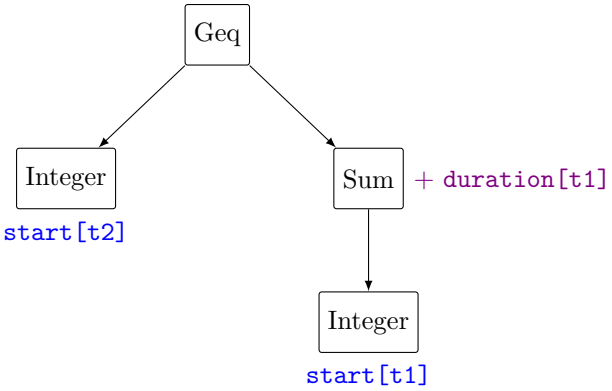


FIGURE 3.3 – Graphe d'évaluation de la contrainte de précédence – deuxième version

— Troisième version :

```
constraint start[t1] + duration[t1] - start[t2] <= 0;
```

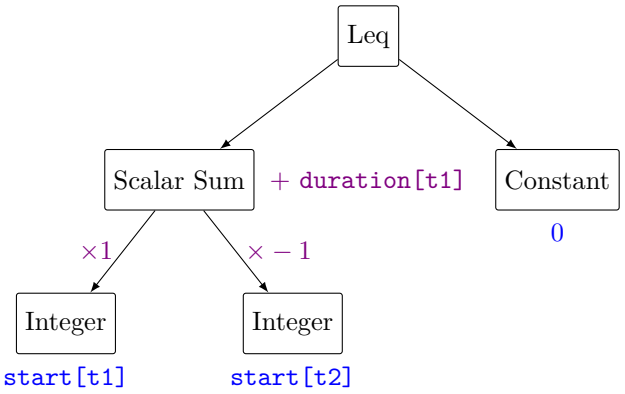


FIGURE 3.4 – Graphe d'évaluation de la contrainte de précédence – troisième version

— Quatrième version :

```
constraint start[t2] - end[t1] >= 0;
```

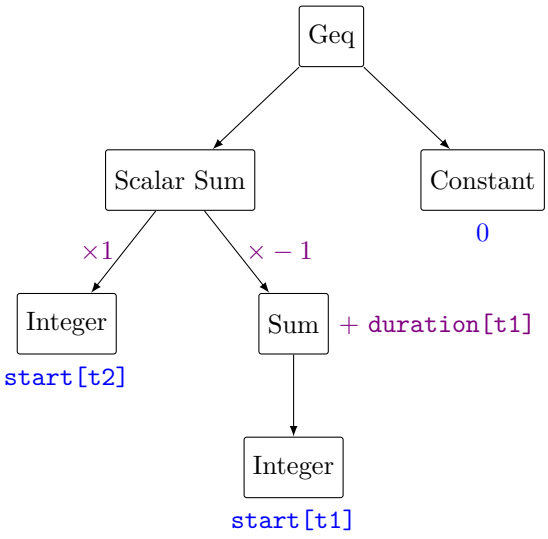


FIGURE 3.5 – Graphe d'évaluation de la contrainte de précédence – quatrième version

Comme expliqué précédemment, la détection des contraintes correspondant à des inégalités linéaires binaires se fait grâce à un algorithme explorant récursivement le sous-graphe de la contrainte dans le graphe d'évaluation du modèle. Sans donner le détail de cet algorithme de détection, on montre son application sur un exemple. On considère pour cela le graphe de la deuxième version de la contrainte de précédence présentée ci-dessus, correspondant à l'écriture suivante.

```
1 constraint start[t2] >= end[t1];
```

Le déroulement de l'algorithme est le suivant.

- Le nœud racine de la contrainte est de type « *geq* » : on lui applique donc l'algorithme de détection des inégalités linéaires binaires. On commence à construire la forme canonique $aX + bY \leq c$ (Equation 3.1) que l'on cherche à reconnaître : les variables X et Y et les coefficients a et b sont indéfinis, et la constante c est nulle.
- On explore le fils gauche du nœud racine de la contrainte : il s'agit d'un nœud de type « variable entière » (`start[t2]`), ce qui est jusqu'ici cohérent avec la forme canonique de l'Equation 3.1. On continue alors la construction de cette forme canonique : on retient que X correspond à la variable `start[t2]`, avec un coefficient $a = -1$ (du fait du sens de l'inégalité).
- On explore le fils droit du nœud racine de la contrainte : il s'agit d'un nœud de type « somme » (`end[t1]`), possédant un unique enfant. Or, il reste une variable à détecter (puisque la forme canonique comprend deux variables, et qu'une première variable a déjà été détectée à l'étape précédente). La partie du sous-graphe que l'on a explorée est donc toujours cohérente avec la forme de l'Equation 3.1. La somme possède également un terme constant (`duration[t1]`). On met alors à jour la valeur de la constante c dans la forme canonique que l'on est en train de construire : c vaut désormais $-$`duration[t1]`.
- On explore l'unique enfant du nœud somme : il s'agit d'un nœud de type « variable entière » (`start[t1]`), ce qui reste cohérent par rapport à la forme de l'Equation 3.1. On continue alors la construction de la forme canonique : on retient que Y correspond à la variable `start[t1]`, avec un coefficient $b = 1$.
- L'exploration du sous-graphe de la contrainte est terminée. On n'a rencontré aucune incohérence par rapport à la forme de l'Equation 3.1. On vérifie alors que la forme canonique construite lors de l'exploration du graphe est correcte : on a bien détecté le bon nombre de variables numériques, soit deux, et ces deux variables sont bien différentes. La contrainte est donc bien détectée comme une inégalité linéaire binaire, d'équation canonique $-$`start[t1]` $+$ `start[t2]` $\leq -$`duration[t1]`.

3.4.2 Mécanisme de réparation et propriétés sur les inégalités linéaires binaires

La propagation de ces contraintes linéaires binaires suit le déroulement de l'algorithme général de réparation décrit dans la Section 3.2. En termes plus concrets, l'application de l'algorithme de filtrage HBC à une contrainte $\mathcal{C}$ d'équation $aX + bY \leq c$ se déroule de la façon suivante.

Si l'inégalité est déjà vérifiée, la contrainte est ignorée. Si les variables ne peuvent pas être décalées suffisamment dans la bonne direction pour réparer $\mathcal{C}$, la propagation échoue.

Sinon, si une seule des variables peut être décalée dans la bonne direction (par symétrie, on suppose qu'il s'agit de la variable X), l'algorithme applique la seule modification nécessaire et suffisante sur X pour réparer $\mathcal{C}$: X prend la valeur $x = \frac{c-by}{a}$.

Enfin, si les variables X et Y peuvent toutes deux être modifiées dans le sens de la réparation de la contrainte $\mathcal{C}$, alors, comme expliqué dans la Section 3.2, celle-ci peut être réparée de différentes façons. Cependant, on montre que le caractère réalisable de la solution initiale $\mathcal{S}_0$ garantit que ce dernier cas ne se produit jamais.

Proposition 3.2. *On considère une contrainte $\mathcal{C}$, d'équation $aX + bY \leq c$, rencontrée au cours de la propagation. Si la solution initiale $\mathcal{S}_0$ est réalisable, alors il existe au plus une façon de réparer $\mathcal{C}$.*

Preuve. On suppose que la contrainte $\mathcal{C}$ est violée. En notant x_0 et y_0 la valeur des variables X et Y dans la solution initiale $\mathcal{S}_0$, et x et y leur valeur lors de la propagation de $\mathcal{C}$, on a $ax + by > c$. On suppose qu'il existe plusieurs façons de réparer la contrainte, ce qui implique que X et Y peuvent toutes deux être décalées dans le sens de la réparation. On a donc $ax \leq ax_0$, $by \leq by_0$, et $ax_0 + by_0 > c$, ce qui implique que la contrainte $\mathcal{C}$ était également violée dans la solution initiale. □

Proposition 3.3. *Si la solution initiale $\mathcal{S}_0$ est réalisable, et s'il existe une solution réalisable compatible avec les décisions de la transformation locale, alors l'algorithme est garanti de la trouver.*

Preuve. Puisque la solution initiale $\mathcal{S}_0$ est réalisable, si une contrainte linéaire binaire est violée, alors au moins l'une de ses deux variables a déjà été modifiée dans le sens de la violation. Du fait de la Propriété 3.1 de respect des décisions précédentes, elle ne peut pas être modifiée dans le sens de la réparation. On sait alors que dans toute solution réalisable qui respecte les décisions de la transformation locale, l'autre variable a également été modifiée. Sa valeur doit ainsi avoir été modifiée dans le sens de la réparation, suffisamment pour réparer la contrainte. Les seules décisions de réparation que prend l'algorithme sont nécessaires pour obtenir une solution réalisable. En outre, d'après la Propriété 3.2, à chaque fois qu'une contrainte violée est rencontrée au cours de la propagation, il existe exactement une manière nécessaire et suffisante de la réparer. Ainsi, l'algorithme ne peut jamais prendre une « mauvaise » décision qui l'empêcherait de trouver une solution réalisable à la fin, et il n'est jamais confronté à une configuration dans laquelle il existe plusieurs façons de réparer la contrainte. S'il existe une solution vérifiant toutes les contraintes, et respectant les décisions de la transformation locale, l'algorithme finit donc par la trouver. □

Exemple 3.3. On considère deux tâches t et t', dont les dates de début sont modélisées par des variables de décision notées S_t et $S_{t'}$, et dont les durées sont $d_t = 5$ et $d_{t'} = 6$. On suppose que t doit être ordonnancée avant t' : les deux tâches sont liées par une contrainte de précédence $\mathcal{P}$ d'équation $S_{t'} \geq S_t + d_t$.

On considère une solution réalisable $\mathcal{S}_0$, telle que $s_{t0} = 2$ et $s_{t'0} = 7$. On suppose que la date de début de la tâche t a été modifiée lors de la transformation locale, et vaut $s_t = 4$. La contrainte de précédence $\mathcal{P}$ est donc violée, et l'unique réparation nécessaire consiste à modifier la date de début de t' : on pose $s_{t'} = 9$. Cette situation est représentée sur la Figure 3.6.

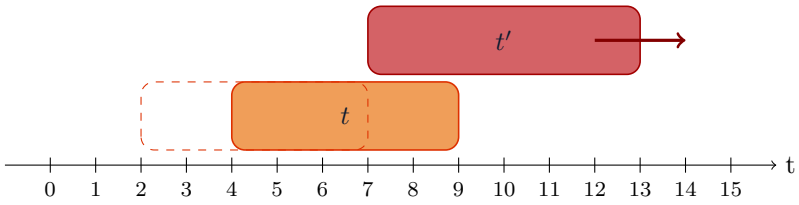


FIGURE 3.6 – Réparation d'une contrainte de précédence

3.4.3 Extensions

On rencontre parfois des contraintes dont l'équation est très proche de l'Equation 3.1 décrivant une inégalité linéaire binaire, sans y correspondre parfaitement. Si l'on se limitait strictement aux contraintes respectant l'Equation 3.1, celles-ci ne seraient alors pas détectées comme des inégalités à propager. Par exemple, dans le problème du Job Shop flexible, chaque tâche peut être affectée à différentes machines, sur lesquelles elle peut avoir des durées différentes. La durée d'une tâche n'est donc pas une constante, mais une expression entière dépendant de la machine qui a été choisie pour la tâche. Dans ce cas, les contraintes de précédence entre deux tâches ne sont donc pas des

inégalités linéaires binaires. Cependant, les difficultés rencontrées par les algorithmes de recherche locale à voisinages restreints sur les problèmes d'ordonnancement, décrites dans la Section 3.1, se posent également lorsque les durées des tâches sont flexibles. On souhaite donc être capable de propager les contraintes de précédence avec durées flexibles, bien que leur expression diffère légèrement de celle des inégalités linéaires binaires.

On généralise donc l'expression des inégalités détectées, que l'on appelle alors « inégalités linéaires binaires paramétrées », de la forme

$$aX + bY \leq \sum_i P_i \tag{3.2}$$

où chaque paramètre P_i prend l'une des formes suivantes :

- Paramètre constant : $P_i = c$, avec c une constante
- Paramètre booléen : $P_i = cB$, avec B une variable booléenne, différente de X et Y, et c une constante
- Paramètre « *at* » : $P_i = d[Z]$, avec Z une variable entière ou booléenne, différente de X et Y, et d un tableau de constantes
- Paramètre « *find* » : $P_i = d[\text{find}(\mathcal{L}, c_0)][c_1][...][c_k]$, avec d un tableau de constantes de dimension $k+1$, $\mathcal{L}$ une famille de variables de listes formant une partition, et chaque indice c_j une constante entière.

S'il n'y a qu'un seul paramètre, de type constant, la contrainte est une inégalité linéaire binaire classique. Le paramètre *find* correspond aux contraintes de précédence avec durées flexibles, décrites en détail dans l'Exemple 3.4. On donne un exemple d'application pour les paramètres booléens et *at* dans la Section 3.5.3.

Exemple 3.4 (Détection d'une contrainte de précédence avec durées flexibles). Afin d'illustrer la détection des inégalités linéaires binaires paramétrées, on considère l'exemple de l'écriture d'une contrainte de précédence entre deux tâches de durées flexibles. Le Modèle 3.2 donne une ébauche de modélisation pour un tel problème [2]. Les différentes machines sont modélisées par des variables de listes regroupées dans le tableau `machines` (tel que `machines[m]` représente l'ordre des tâches affectées à la machine m), et formant une partition (chaque tâche est affectée à exactement une machine). Pour toute tâche t, `start[t]` est une variable de décision entière représentant la date de début de t, `chosenMachine[t]` correspond à la machine à laquelle est affectée t dans la solution courante, et `duration[t]` correspond à la durée de t sur cette machine (obtenue à partir du tableau bidimensionnel constant `machineDurations`).

```
1  // Lectures des données : durée de chaque tâche sur chaque machine.
2  machineDurations[m in 0..nbMachines-1][t in 0..nbTasks-1];
3
4  // Variables de décision.
5  start[t in 0..nbTasks-1] <- int(0, maxStart);
6  machines[m in 0..nbMachines-1] <- list(nbTasks);
7  constraint partition(machines);
8
9  // Expressions intermédiaires.
10 chosenMachine[t in 0..nbTasks-1] <- find(machines, t);
11 duration[t in 0..nbTasks-1] <- machineDurations[chosenMachine[t]][t];
12
13 // Contrainte de précédence.
14 constraint start[t2] >= start[t1] + duration[t1];
```

Modèle 3.2 – Exemple de modélisation de contrainte de précédence avec durées flexibles

2. Le Modèle 3.2 correspond par exemple à une ébauche de modélisation pour le problème du Job Shop flexible, défini dans les Sections 1.3.1 et 2.3.2. Le modèle LocalSolver pour le problème du Job Shop flexible est présenté en détail dans la Section 2.3.2.

La contrainte de précédence entre les tâches t_1 et t_2 (ligne 14 du Modèle 3.2) est représentée dans le solveur par le graphe d'évaluation illustré sur la Figure 3.7. On constate que le graphe à détecter devient plus complexe que les graphes des contraintes de précédence simples donnés dans l'Exemple 3.2. Comme précédemment, sa forme peut varier en fonction de l'écriture de la contrainte dans le modèle. Par souci de robustesse, on doit là encore être capable de détecter chaque variante possible pour le graphe d'évaluation correspondant.

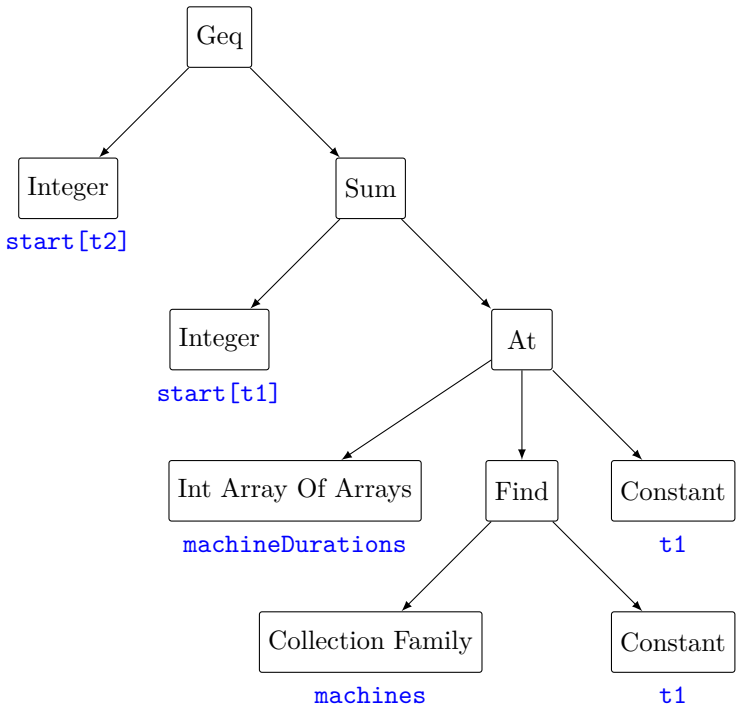


FIGURE 3.7 – Graphe d'évaluation de la contrainte de précédence – avec durées flexibles

Mécanisme de réparation. La réparation des inégalités linéaires binaires paramétrées suit le même principe que la réparation des inégalités non paramétrées. Comme expliqué dans la Section 3.2, on souhaite réparer ces contraintes en modifiant juste assez les valeurs des variables, sans aller à l'encontre des décisions prises lors des mouvements de la recherche locale. On considère que les valeurs des variables intervenant dans l'expression des paramètres (variables booléennes, ou variables entières pour les paramètres de type *at*, ou variables de listes pour les paramètres *find*) doivent être décidées par les mouvements de la recherche locale uniquement, et on choisit donc de ne pas les modifier lors des réparations. La réparation d'une inégalité linéaire binaire paramétrée $\mathcal{C}$, décrite par l'Equation 3.2, se fait alors en modifiant les valeurs des variables X et Y, et en laissant fixes les valeurs des paramètres. Pour réparer la contrainte $\mathcal{C}$, on commence donc par évaluer la valeur courante p_i de chaque paramètre P_i. En notant $c' = \sum_i p_i$, la contrainte $\mathcal{C}$ est alors réparée comme une inégalité linéaire binaire classique $\mathcal{C}'$ d'équation $aX + bY \leq c'$.

3.4.4 Condition suffisante sur la réparabilité des contraintes binaires

Dans cette Section, on décrit une condition sur toute contrainte binaire garantissant que, s'il existe une solution réalisable respectant les décisions de la transformation locale, la propagation réussira. Une telle propriété est très utile, puisqu'elle assure une grande efficacité dans la procédure de réparation.

Comme vu dans la Section 3.4.2, l'algorithme de propagation est garanti de trouver une solution réalisable (lorsqu'il en existe une) si et seulement si il existe toujours une façon nécessaire de réparer

chaque contrainte violée rencontrée. En effet, on ne réalise alors que des modifications nécessaires à l'obtention d'une solution réalisable, et on est garanti de ne jamais prendre de « mauvaise » décision empêchant d'aboutir à une telle solution. Au contraire, si cette propriété n'est pas vérifiée, l'algorithme peut se retrouver confronté à une configuration dans laquelle plusieurs réparations sont possibles, sans indication de laquelle sera la plus avantageuse.

Définition 3.2 (Réparation nécessaire). *On appelle réparation nécessaire toute modification des variables impliquées dans l'expression d'une contrainte violée nécessaire à sa réparation et conduisant à une solution réalisable. Comme décrit dans la Section 3.2, lorsque l'on applique l'algorithme de filtrage HBC sur une contrainte, on considère la solution dans laquelle la valeur de chaque variable X est la projection de sa valeur actuelle x sur son domaine réduit. Si cette solution est réalisable, alors il s'agit d'une réparation nécessaire.*

On introduit ainsi une condition sur toute contrainte binaire $\mathcal{C}$ portant sur deux variables X et Y, garantissant que s'il existe une solution réalisable du point de vue de $\mathcal{C}$ et respectant les décisions précédentes, alors la projection de la valeur actuelle de X et Y sur leur domaine réduit respecte la contrainte $\mathcal{C}$, et constitue donc une réparation nécessaire. Dans la suite de la Section, une solution dans laquelle les variables X et Y ont pour valeur x et y respectivement sera notée (x, y). De plus, pour alléger les notations, on désignera par « solution réalisable » (resp. « solution non réalisable ») toute solution réalisable (resp. non réalisable) du point de vue de la contrainte $\mathcal{C}$, c'est-à-dire respectant (resp. ne respectant pas) $\mathcal{C}$.

Définition 3.3 (Réparation minimale). *On considère la réparation d'une solution non réalisable (x, y) en une solution réalisable (x', y'). Cette réparation est minimale si toute solution intermédiaire $(\lambda x + (1-\lambda)x', \mu y + (1-\mu)y')$ avec $\lambda, \mu \in]0, 1[$ est non réalisable.*

Remarque 3.2. Par définition, toute réparation nécessaire est minimale. Cependant, une réparation minimale n'est pas toujours nécessaire. Par exemple, deux solutions (x, y_1) et (x, y_2) avec $y_1 < y < y_2$ peuvent toutes deux correspondre à une réparation minimale.

Définition 3.4 (Contrainte biconvexe). *Une contrainte binaire $\mathcal{C}$ sur les variables réelles X et Y est appelée biconvexe si*

$$\begin{cases} \forall x \in \mathcal{D}_X,\ \mathcal{C}^x = \{y \in \mathcal{D}_Y : (x, y) \in \mathcal{C}\} \text{ est convexe} \\ \forall y \in \mathcal{D}_Y,\ \mathcal{C}^y = \{x \in \mathcal{D}_X : (x, y) \in \mathcal{C}\} \text{ est convexe} \end{cases}$$

Une contrainte binaire $\mathcal{C}$ sur les variables entières X et Y est appelée biconvexe si

$$\begin{cases} (x_1, y_1) \text{ et } (x_2, y_1) \text{ réalisables} & \Rightarrow & \forall \tilde{x} \in [x_1, x_2],\ (\tilde{x}, y_1) \text{ réalisable} \\ (x_1, y_1) \text{ et } (x_1, y_2) \text{ réalisables} & \Rightarrow & \forall \tilde{y} \in [y_1, y_2],\ (x_1, \tilde{y}) \text{ réalisable} \end{cases}$$

Définition 3.5 (Contrainte chemins-connexe). *Une contrainte binaire $\mathcal{C}$ sur les variables réelles X et Y est chemins-connexe si pour deux solutions réalisables (x, y) et (x', y'), il existe un chemin réalisable continu entre elles. C'est-à-dire :*

$$\forall (x, y), (x', y') \in \mathcal{C},\ \exists f : [0, 1] \to \mathcal{C} \text{ continue}, f(0) = (x, y), f(1) = (x', y')$$

Une contrainte binaire $\mathcal{C}$ sur des variables entières X et Y est chemins-connexe si pour deux solutions réalisables (x, y) et (x', y'), il existe un chemin de solutions « voisines » [3] réalisables entre elles. C'est-à-dire :

$$\forall (x, y), (x', y') \in \mathcal{C},\ \exists \{f_0, ..., f_n\} \in \mathcal{C}, \begin{cases} f_0 = (x, y), f_n = (x', y') \\ \forall i < n,\ \|f_i - f_{i+1}\|_\infty \leq 1 \end{cases}$$

3. Deux solutions (x_1, y_1) et (x_2, y_2) sont voisines lorsque $\|(x_1, y_1) - (x_2, y_2)\|_\infty \leq 1 \Leftrightarrow |x_1 - x_2| \leq 1$ et $|y_1 - y_2| \leq 1$

Lemme 3.1. *Si une contrainte binaire $\mathcal{C}$ sur des variables entières ou réelles X et Y est biconvexe et chemins-connexe, alors pour deux solutions réalisables quelconques (x, y) et (x', y'), il existe un chemin de solutions réalisables $(\tilde{x}, \tilde{y})$ tel que $x \leq \tilde{x} \leq x'$ (resp. $x \geq \tilde{x} \geq x'$) et $y \leq \tilde{y} \leq y'$ (resp. $y \geq \tilde{y} \geq y'$).*

Un tel chemin sera appelé « chemin réalisable borné ».

Preuve. Par symétrie, on suppose $x \leq x'$. Soient $\mathcal{D} = \{(\tilde{x}, \tilde{y}) : \tilde{x} \geq x\}$ et $\mathcal{C}' = \mathcal{C} \cap \mathcal{D}$. Puisque $\mathcal{C}$ est chemins-connexe, pour tous points A et B dans $\mathcal{C}'$, il existe un chemin f de A vers B dans $\mathcal{C}$. Supposons que f contient un point dans $\mathcal{C} \setminus \mathcal{D}$. Alors, A et B étant tous deux dans $\mathcal{D}$, f traverse la bordure de $\mathcal{D}$ un nombre pair de fois (et au moins deux fois). Soient (x, y_1) et $(x, y_2) \in \mathcal{C}'$ le premier et le dernier de ces points d'intersection. Puisque $\mathcal{C}$ est biconvexe, $\forall \tilde{y} \in [y_1, y_2]$, la solution $(x, \tilde{y})$ est réalisable. On peut alors définir un nouveau chemin f' de A vers B, confondu avec f sauf entre (x, y_1) et (x, y_2), où il est confondu avec ce segment. f' est entièrement dans $\mathcal{C}'$.

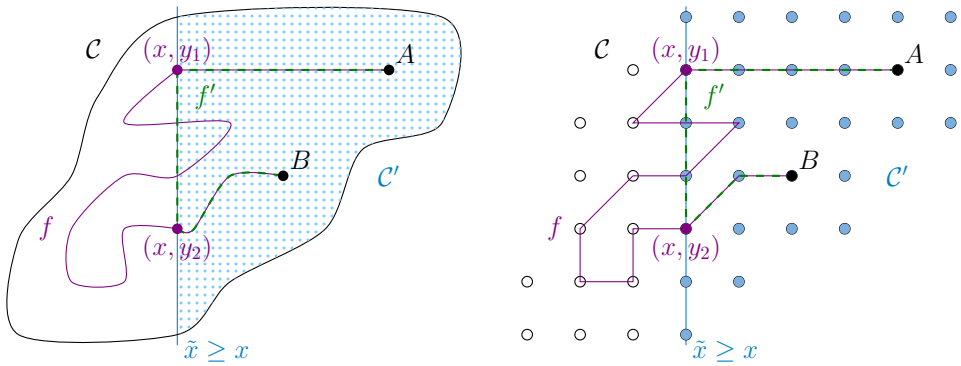


FIGURE 3.8 – Chemin réalisable borné – variables réelles (gauche) et entières (droite)

Comme illustré sur la Figure 3.8, $\mathcal{C}'$ est donc chemins-connexe. Par extension, l'intersection de $\mathcal{C}$ et du rectangle formé par les deux points diagonaux (x, y) et (x', y') l'est aussi. □

Proposition 3.4. *Si une contrainte binaire $\mathcal{C}$ sur des variables entières ou réelles X et Y est biconvexe et chemins-connexe (et fermée dans le cas des variables réelles), et s'il est possible de la réparer, alors elle admet une réparation nécessaire. Cette réparation consiste à projeter les variables sur leurs domaines réduits.*

Preuve. On suppose que la solution initiale (x_0, y_0) est réalisable.

1. Tout d'abord, on suppose que seule la valeur de X a changé : lorsque la contrainte $\mathcal{C}$ est propagée, la solution courante (x, y_0) est infaisable (par symétrie, on suppose $x > x_0$).
   (a) Il est impossible de réparer la contrainte en modifiant uniquement la valeur de X :
   S'il existe $(x_1, y_0) \in \mathcal{C}$ avec $x_1 > x$, alors puisque $\mathcal{C}$ est biconvexe, on a aussi $(x, y_0) \in \mathcal{C}$, ce qui contredit notre hypothèse de départ.
   (b) S'il existe une solution réalisable (x, y), alors il existe exactement une réparation minimale modifiant seulement Y :
   Supposons qu'il existe $y > y_0$ et $y' < y_0$ tels que $(x, y) \in \mathcal{C}$ et $(x, y') \in \mathcal{C}$. Puisque $\mathcal{C}$ est biconvexe, alors $(x, y_0) \in \mathcal{C}$, ce qui contredit notre hypothèse de départ. Ainsi, il existe au plus une réparation minimale modifiant seulement Y.
   S'il existe une solution réalisable (x, y) (avec $y > y_0$, par symétrie), alors $\mathcal{C} \cap \{(x, \tilde{y}) : y_0 \leq \tilde{y} \leq y\}$ étant soit fermé (dans le cas de variables réelles) soit fini (dans le cas de variables entières), il existe une réparation minimale modifiant seulement Y.
   (c) S'il existe une façon de réparer la contrainte en modifiant les deux variables, alors il existe une réparation minimale modifiant seulement Y, qui consiste à projeter Y sur son domaine réduit :

Supposons qu'il existe $x' > x$ et y' (par symétrie, on peut supposer $y' > y_0$) tels que $(x', y') \in \mathcal{C}$. D'après le Lemme 3.1, il existe un chemin réalisable borné f de (x_0, y_0) vers (x', y'), et f contient un point (x, y_1) avec $y_0 < y_1 \leq y'$. Puisque $y_1 \leq y'$, il existe exactement une réparation minimale impliquant d'augmenter la valeur de Y. D'après 1b, il existe exactement une réparation minimale modifiant seulement Y.

Les paragraphes 1c et 1b prouvent que s'il existe une solution réalisable (x', y') avec $x' \geq x$ et $y' > y_0$ (resp. $y' < y_0$), alors $\forall \tilde{x} \geq x, \forall \tilde{y} \leq y_0$ (resp. $\forall \tilde{y} \geq y_0$), la solution $(\tilde{x}, \tilde{y})$ est infaisable. Ainsi, il existe exactement une réparation minimale $(x, y^\star)$, vérifiant $y^\star > y_0$, et $y^\star$ est la projection de Y sur son domaine réduit.

2. On suppose maintenant que les deux variables ont été modifiées : lorsque la contrainte est propagée, la solution courante (x, y) est non réalisable (par symétrie, on suppose $x > x_0$ et $y > y_0$).
Comme illustré sur la Figure 3.9, on montre que s'il est possible de réparer la contrainte, alors il existe une réparation minimale modifiant une des deux variables seulement.

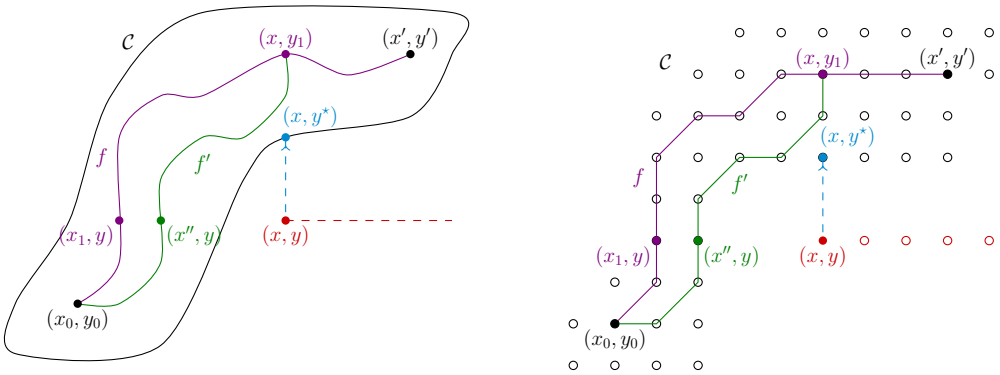


FIGURE 3.9 – Réparation nécessaire – variables réelles (gauche) et entières (droite)

On suppose que la contrainte est réparable : il existe $x' \geq x$ et $y' \geq y$ tels que $(x', y') \in \mathcal{C}$. D'après le Lemme 3.1, il existe un chemin réalisable borné f de (x_0, y_0) vers (x', y'), et f contient deux points (x, y_1) et (x_1, y), vérifiant $x_0 \leq x_1 \leq x'$ et $y_0 \leq y_1 \leq y'$. Si $y_1 > y$, alors, d'après le Lemme 3.1, il existe un chemin réalisable borné f' de (x_0, y_0) vers (x, y_1), et f' contient un point (x'', y) tel que $x_0 \leq x'' < x$. Puisque $\mathcal{C}$ est biconvexe, $\forall \tilde{x} \geq x$, on a $(\tilde{x}, y) \notin \mathcal{C}$. Puisque (x_1, y) est faisable, on a $x_1 < x$. De même, si $x_1 > x$, alors $y_1 < y$. Par symétrie, supposons que $y_1 > y$: la solution réalisable (x, y_1) respecte les décisions de la transformation locale, mais pas (x_1, y). Ainsi, il existe exactement une réparation minimale impliquant d'augmenter la valeur de Y, et exactement une réparation minimale modifiant seulement Y.

Tout comme au paragraphe 1 de la preuve, il existe exactement une réparation minimale $(x, y^\star)$, vérifiant $y^\star > y_0$, et $y^\star$ est la projection de Y sur son domaine réduit. □

3.5 Réparation de disjonctions et de chaînes d'inégalités linéaires binaires portant sur des variables numériques

3.5.1 Disjonctions et chaînes d'inégalités linéaires binaires

En plus des contraintes mentionnées précédemment, on considère désormais les disjonctions d'inégalités de la forme

$$\bigvee_i (a_i X_i + b_i Y_i \leq c_i) \tag{3.3}$$

où les X_i et Y_i sont des variables entières ou réelles, et les a_i, b_i et c_i sont des constantes quelconques, ainsi que les chaînes d'inégalités de la forme

$$\bigwedge_i \left(aX_{L[f(i)]} + bX_{L[g(i)]} \leq c_{L[h(i)]}\right) \tag{3.4}$$

où X est un tableau de variables entières ou réelles, L est une variable de liste représentant un ordre sur les variables de décision du tableau X (voir Chapitre 2), a, b et les éléments du tableau c sont des constantes quelconques, et f, g et h sont des fonctions entières quelconques.

Remarque 3.3. Lorsque les a_i et b_i sont respectivement égaux à 1 et -1, et que $X_{2i} = Y_{2i+1}$ et $Y_{2i} = X_{2i+1}$ pour toute valeur de i, les disjonctions de la forme de l'Equation 3.3 correspondent à des contraintes de packing ou de ressources disjonctives, dans lesquelles les objets ou tâches sont considérés deux à deux. De même, lorsqu'on a $a = 1$ et $b = -1$, et que $f(i) = h(i) = i$ et $g(i) = i + 1$ pour toute valeur de i, la chaîne de l'Equation 3.4 décrit une contrainte de ressource disjonctive, dans laquelle les tâches sont ordonnées selon la variable de liste L. Une illustration de l'utilisation des disjonctions et chaînes d'inégalités linéaires binaires en tant que contraintes de non-chevauchement des tâches affectées à une même ressource disjonctive est donnée dans l'Exemple 3.5.

Exemple 3.5. Les contraintes de ressources disjonctives – non-chevauchement des tâches ordonnancées sur une même machine – sont souvent rencontrées dans le domaine de l'ordonnancement. Lorsque le nombre de tâches n sur une ressource est fixe, sa nature disjonctive peut être décrite en utilisant $O(n^2)$ disjonctions de précédences :

$$\forall 1 \leq i < j \leq n,\ (S_j \geq S_i + d_i) \vee (S_i \geq S_j + d_j)$$

où S_i et d_i, respectivement, désignent la date de début (variable de décision) et la durée (constante) d'une tâche i. Dans cette formulation, on considère que si deux tâches i et j sont ordonnancées sur la même machine, alors la tâche j doit commencer après la fin de la tâche i, ou inversement.

Cependant, en utilisant une contrainte chaînée, on peut réduire le nombre de contraintes à $O(n)$. En effet, comme expliqué dans le Chapitre 2, dans toute solution réalisable, les tâches sont ordonnancées dans un certain ordre. On peut alors formuler la contrainte avec une variable de liste L : la valeur de la variable de liste est une permutation, définissant l'ordre des tâches.

$$\forall 1 \leq i < n,\ S_{L[i+1]} \geq S_{L[i]} + d_{L[i]}$$

Dans cette formulation, on considère que la $(i + 1)$-ième tâche ordonnancée sur la machine doit commencer après la fin de la i-ème tâche, créant une contrainte chaînée de taille $O(n)$.

Détection des disjonctions et chaînes d'inégalités linéaires binaires. Comme pour les inégalités linéaires binaires, on cherche à rendre la phase de détection des disjonctions et chaînes très robuste, afin d'obtenir de bons résultats quelle que soit l'écriture du modèle.

La détection des disjonctions dans le graphe d'évaluation est la plus simple. On cherche simplement à détecter une contrainte sur un nœud de type « *or* », dont tous les enfants sont détectés comme étant des inégalités linéaires binaires, éventuellement paramétrées. On donne une ébauche du graphe d'évaluation correspondant sur la Figure 3.10.

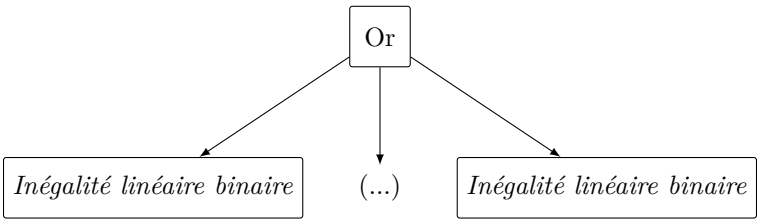


FIGURE 3.10 – Graphe d'évaluation d'une disjonction d'inégalités linéaires binaires

L'expression d'une contrainte chaînée est plus complexe. Elle passe par l'utilisation d'une fonction « *and* » variadique parcourant la liste L. On donne un exemple de contraintes chaînées dans le Modèle 3.3, correspondant à la modélisation des contraintes de non-chevauchement des tâches ordonnancées sur chaque machine dans le problème du Job Shop [4]. Le graphe d'évaluation de la contrainte de non-chevauchement sur une machine m (ligne 13 du Modèle 3.3) est représenté sur la Figure 3.11.

```
1  // Lecture des données.
2  duration[m in 0..nbMachines-1][j in 0..nbJobs-1];
3  // Variables de décision : date de début de chaque tâche.
4  start[m in 0..nbMachines-1][j in 0..nbJobs-1] <- int(0, maxStart);
5  // Variables de décision : ordre des tâches sur chaque machine.
6  order[m in 0..nbMachines-1] <- list(nbJobs);
7  // Expressions intermédiaires.
8  end[m in 0..nbMachines-1][j in 0..nbJobs-1] <- start[m][j] + duration[m][j];
9
10 // Contrainte de non-chevauchement des tâches sur chaque machine.
11 for [m in 0..nbMachines-1] {
12     constraint count(order[m]) == nbJobs;
13     constraint and(1..nbJobs-1, i => start[m][order[m][i]] >= end[m][order[m][i-1]]);
14 }
```

Modèle 3.3 – Exemple de modélisation d'une contrainte de non-chevauchement des tâches

Le sous-graphe représenté en haut à gauche de la Figure 3.11 (sous-graphe parent du nœud de type « *geq* ») correspond à la structure d'une fonction « *and* » variadique. Le « *and* » variadique possède deux enfants dans le graphe : d'une part un nœud de type « *range* », correspondant au domaine sur lequel on applique la fonction, et d'autre part une fonction variadique à valeur entière. Le nœud correspondant à cette fonction, de type « *int function* », a également deux enfants : l'argument de la fonction, de type « *int fun arg* » (noté `i` dans le modèle), et le sous-graphe correspondant à l'expression de la fonction. Ce sous-graphe, dont la racine est un nœud de type « *geq* » dans l'exemple considéré, a une forme semblable à celle du graphe d'évaluation d'une inégalité linéaire binaire (voir Section 3.4.1), avec toutefois quelques différences liées au contexte de fonction variadique dans lequel il se trouve (par exemple, on accède aux variables de décision entières par des nœuds de type « *at* » sur des tableaux, et non de façon directe comme dans les exemples précédents).

Afin d'améliorer la lisibilité, la représentation du graphe d'évaluation de la contrainte chaînée sur la Figure 3.11 s'éloigne légèrement de la forme du véritable graphe d'évaluation présent dans les structures internes de LocalSolver. Les triples flèches partant des nœuds de type « *array* » servent à indiquer que ces nœuds ont plusieurs enfants de même type, et non un seul comme représenté sur la Figure 3.11 (par exemple, le nœud de type « *int array* » correspondant au tableau de variables de décisions entières `start[m]`, représenté en bas à gauche sur la Figure 3.11, possède `nbJobs` enfants de type « *integer* » : `start[m][0]`, `start[m][1]`, ..., `start[m][nbJobs-1]`). De plus, certains nœuds du graphe ont été dupliqués sur la Figure 3.11 (nœuds de type « *int fun arg* » ou « *list* » par exemple).

4. Le modèle LocalSolver complet pour le problème du Job Shop est présenté en détail dans la Section 2.3.2.

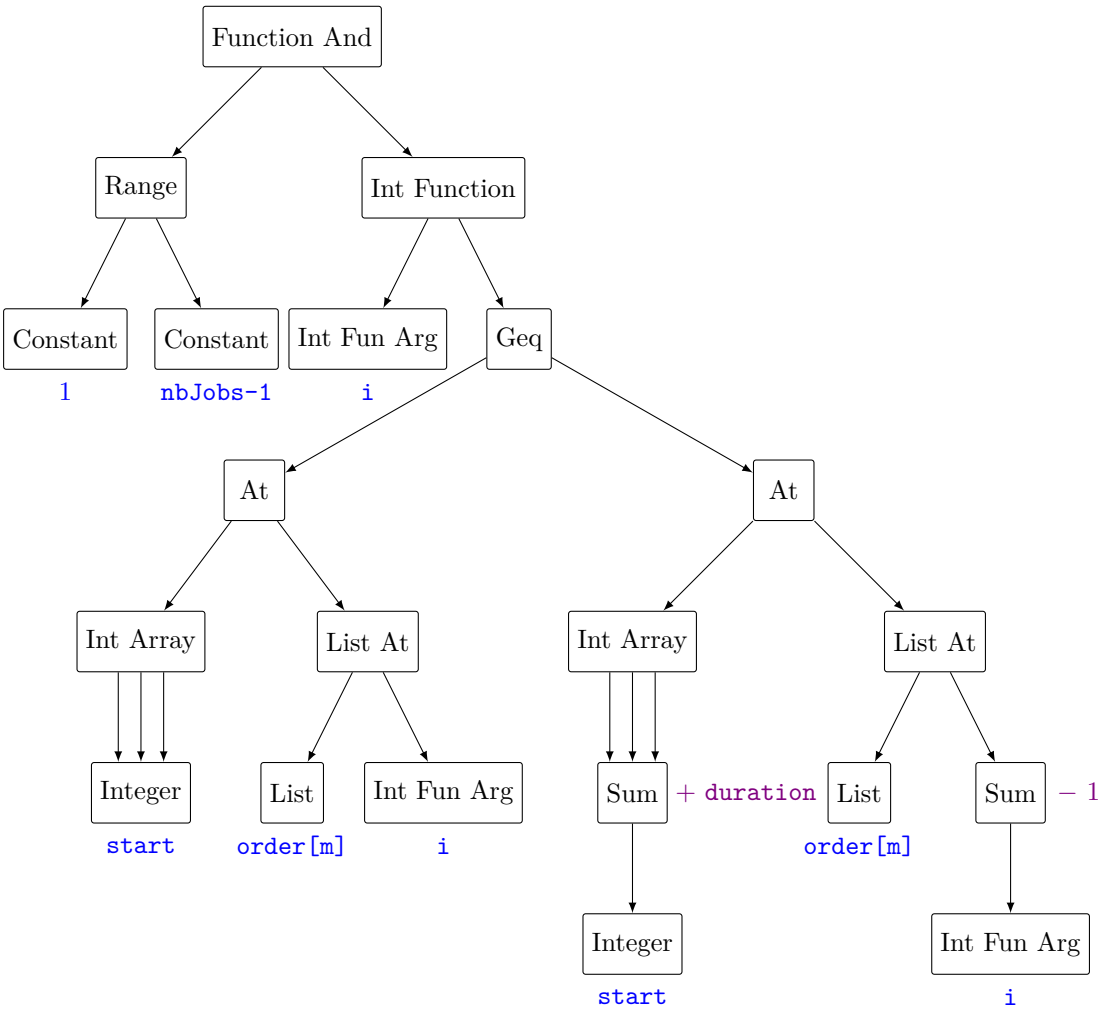


FIGURE 3.11 – Graphe d'évaluation de la contrainte de non-chevauchement

Comme pour les inégalités linéaires binaires, la détection des contraintes chaînées dans le modèle doit être robuste, et s'applique à partir de toutes les contraintes portant sur un nœud de type « *and* » variadique. L'algorithme de détection consiste à explorer récursivement tout le sous-graphe de la contrainte, en construisant progressivement une reformulation de la contrainte sous forme canonique, correspondant à l'Equation 3.4. Cette robustesse dans la détection est là encore nécessaire. En effet, la complexité des graphes d'évaluation des chaînes d'inégalités linéaires binaires est encore accentuée par la diversité des formes qu'ils peuvent prendre en fonction de l'écriture de la contrainte dans le modèle. Par exemple, si l'on inverse l'ordre d'indexation des tableaux `start` et `duration` (job puis machine, au lieu de machine puis job), et si l'on n'utilise pas le tableau de variables intermédiaires `end` dans l'écriture de la contrainte de non-chevauchement, une partie du graphe correspondant est radicalement transformée. Une ébauche du modèle correspondant à ces modifications est donnée dans le Modèle 3.4.

```
1 // Lecture des données.
2 duration[j in 0..nbJobs-1][m in 0..nbMachines-1];
3 // Variables de décision : date de début de chaque tâche.
4 start[j in 0..nbJobs-1][m in 0..nbMachines-1] <- int(0, maxStart);
5 // Variables de décision : ordre des tâches sur chaque machine.
6 order[m in 0..nbMachines-1] <- list(nbJobs);
7 // Expressions intermédiaires.
8 end[j in 0..nbJobs-1][m in 0..nbMachines-1] <- start[j][m] + duration[j][m];
```

```
9
10 // Contrainte de non-chevauchement des tâches sur chaque machine.
11 for [m in 0..nbMachines-1] {
12     constraint count(order[m]) == nbJobs;
13     constraint and(0..nbJobs-2, i => start[order[m][i]][m] + duration[order[m][i]][m]
14             <= start[order[m][i+1]][m]);
15 }
```

Modèle 3.4 – Exemple de modélisation d'une contrainte de non-chevauchement des tâches

Le graphe d'évaluation résultant de cette écriture alternative des contraintes de non-chevauchement est représenté sur la Figure 3.12. Par souci de gain de place, on ne représente ici que la partie du graphe correspondant à l'expression de la fonction variadique utilisée à l'intérieur de l'expression « *and* ». Comme sur la Figure 3.11, certains nœuds de la Figure 3.12 ont été dupliqués afin d'améliorer la lisibilité.

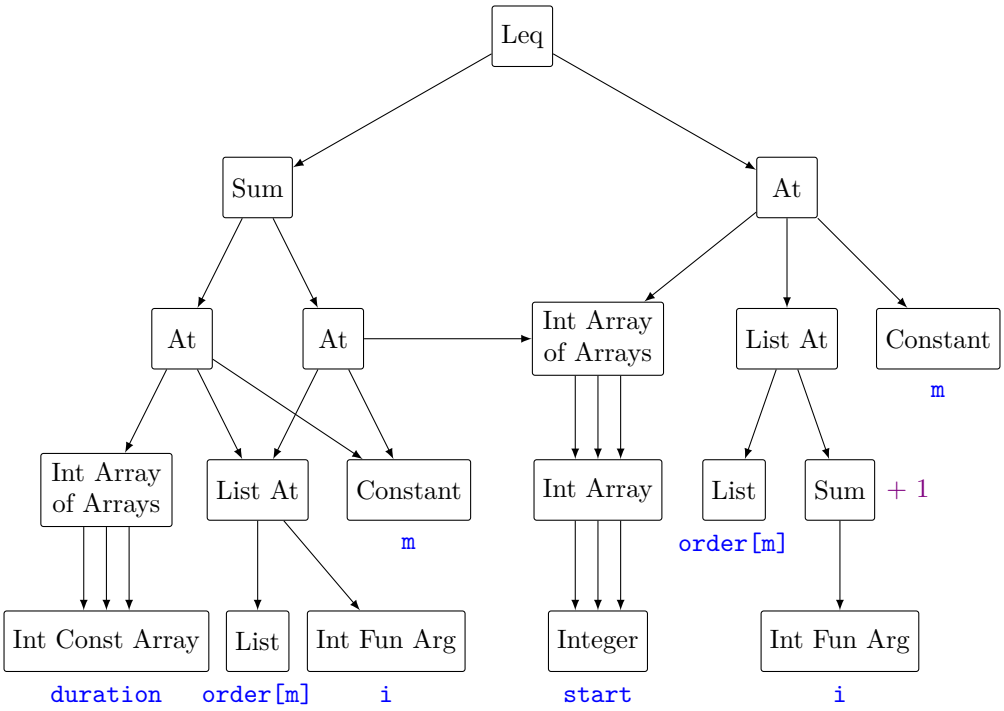


FIGURE 3.12 – Graphe d'évaluation de la contrainte de non-chevauchement

3.5.2 Mécanisme de réparation

Lorsque les contraintes du problème comprennent des disjonctions ou des chaînes, les propriétés citées dans la Section 3.4.2 ne sont plus valables. En effet, les contraintes décrites par les Equations 3.3 et 3.4 admettent rarement une réparation nécessaire. Afin de les réparer efficacement, on utilise alors un algorithme de réparation légèrement différent de l'Algorithme 3.1. La méthode alors appliquée ne repose plus sur de simples projections, et n'a ainsi plus le caractère déterministe de l'algorithme de filtrage HBC, mais suppose au contraire de faire des choix aléatoires lorsqu'il n'existe aucune réparation nécessaire.

Réparation d'une disjonction. On suppose qu'une contrainte $\mathcal{C}$, écrite sous la forme d'une disjonction d'inégalités linéaires binaires, est violée. Puisqu'aucune des inégalités de la disjonction ne doit *a priori* prévaloir sur les autres, on en choisit une au hasard que l'on tente de réparer. S'il est impossible de réparer cette inégalité tout en respectant les décisions prises précédemment, alors

on tente de réparer l'inégalité suivante, et ainsi de suite. Si aucune inégalité de la disjonction ne peut être réparée, la propagation échoue.

La procédure de réparation de l'inégalité ainsi choisie est la suivante. On note $aX + bY \leq c$ cette inégalité. Si une seule de ses variables peut être décalée dans le sens de la réparation, alors il existe une unique façon de réparer cette inégalité, et la contrainte est réparée comme décrit dans la Section 3.4.2.

Cependant, contrairement aux cas précédents, il est également possible que les deux variables puissent être décalées dans le sens de la réparation. En effet, si l'inégalité $aX + bY \leq c$ choisie ne correspond pas à celle qui permettait de vérifier la contrainte $\mathcal{C}$ dans la solution initiale réalisable $\mathcal{S}_0$, il est possible que les valeurs de X et Y n'aient pas été modifiées, ou qu'elles aient été modifiées dans le « bon » sens. Dans ce cas, l'algorithme choisit au hasard comment modifier leur valeur pour réparer la contrainte. Pour cela, on note $\Delta = aX + bY - c > 0$ la distance à la faisabilité, et δ_X et δ_Y les parts de la réparation respectivement attribuées à X et Y, vérifiant $\delta_X + \delta_Y = \Delta$. On choisit alors parmi quatre façons équiprobables de réparer l'inégalité :

- Soit X répare seul la contrainte : $\delta_X = \Delta$, et $\delta_Y = 0$.
- Soit Y répare seul la contrainte : $\delta_X = 0$, et $\delta_Y = \Delta$.
- Soit on partage la réparation équitablement entre X et Y : $\delta_X = \delta_Y = \frac{1}{2}\Delta$.
- Soit on partage la réparation aléatoirement entre X et Y : $\delta_X = random(1, \Delta - 1)$, et $\delta_Y = \Delta - \delta_X$.

On applique alors la réparation choisie : X prend la valeur $x' = x - \frac{\delta_X}{a}$, et Y prend la valeur $y' = y - \frac{\delta_Y}{b}$.

Remarque 3.4. En pratique, on tient également compte des domaines de X et Y dans le choix de la réparation effectuée. Après avoir choisi les valeurs de δ_X et δ_Y, on les ajuste pour s'assurer que l'on a bien $x' \in \mathcal{D}_X$ et $y' \in \mathcal{D}_Y$, tout en maintenant la relation $\delta_X + \delta_Y = \Delta$.

Exemple 3.6 (Réparation d'une disjonction). On considère deux tâches t et t', dont les dates de début sont modélisées par des variables de décision notées S_t et $S_{t'}$, et dont les durées sont $d_t = 5$ et $d_{t'} = 6$. On suppose que les deux tâches sont liées par une contrainte de non-chevauchement $\mathcal{C}$ d'équation

$$(S_{t'} \geq S_t + d_t) \vee (S_t \geq S_{t'} + d_{t'}) .$$

On suppose que dans la solution $\mathcal{S}$ obtenue à l'issue de la transformation locale, la date de début de t a diminué et vaut désormais $s_t = 4$, et que la date de début de t' vaut $s_{t'} = 6$. La contrainte $\mathcal{C}$ est donc violée. La solution $\mathcal{S}$ est représentée sur la Figure 3.13.

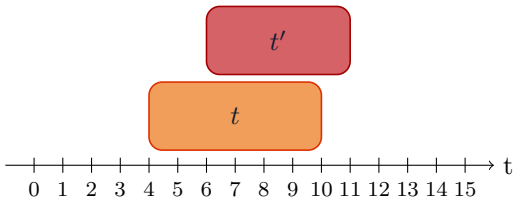
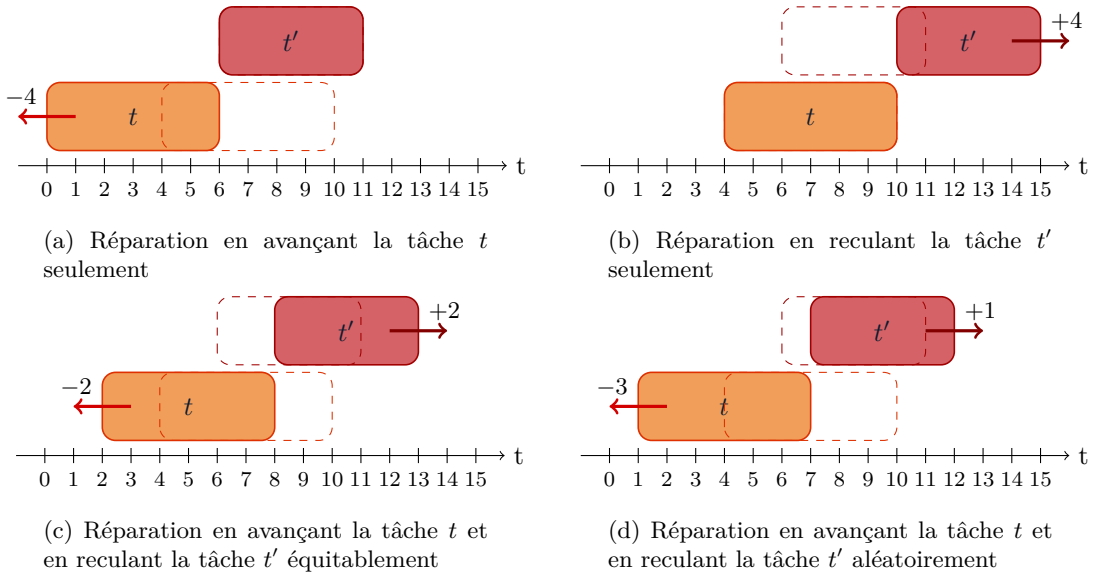


FIGURE 3.13 – Solution $\mathcal{S}$: la contrainte de non-chevauchement est violée

Lors de la propagation de la contrainte $\mathcal{C}$, on suppose que l'on choisit de réparer la première inégalité de la disjonction : la tâche t doit être ordonnancée avant la tâche t'. Chacune des deux variables S_t et $S_{t'}$ pouvant être modifiée dans le sens de la réparation, il existe plusieurs façons de réparer la contrainte. On choisit alors une réparation au hasard, parmi les quatre possibilités représentées sur la Figure 3.14 : diminuer la valeur de S_t seulement (Figure 3.14a), augmenter la valeur de $S_{t'}$ seulement (Figure 3.14b), diminuer la valeur de S_t et augmenter celle de $S_{t'}$ de la même quantité (Figure 3.14c), ou modifier les valeurs de S_t et $S_{t'}$ aléatoirement (Figure 3.14d).



(a) Réparation en avançant la tâche t seulement

(b) Réparation en reculant la tâche t' seulement

(c) Réparation en avançant la tâche t et en reculant la tâche t' équitablement

(d) Réparation en avançant la tâche t et en reculant la tâche t' aléatoirement

FIGURE 3.14 – Différentes façons de réparer la contrainte $\mathcal{C}$

Exemple 3.7 (Problème d'ordonnancement). On considère un problème d'ordonnancement avec trois tâches. La tâche t, de durée 3 et de date de release 1, et la tâche t', de durée 2, ne doivent pas se chevaucher, et doivent toutes deux être ordonnancées avant la tâche t'', de durée 4. Le problème comporte donc trois variables entières S, S' et S'' (dates de début des tâches), deux contraintes de précédence $\mathcal{P}_1 : S - S'' \leq -3$ et $\mathcal{P}_2 : S' - S'' \leq -2$, et une contrainte de ressource disjonctive $\mathcal{R} : (S - S' \leq -3) \vee (S' - S \leq -2)$. On suppose que la solution initiale réalisable $\mathcal{S}_0$ est telle que $s_0 = 1$, $s'_0 = 4$ et $s''_0 = 6$. On suppose qu'après la transformation locale, la solution courante $\mathcal{S}$ est irréalisable et vérifie $s = s_0 = 1$, $s' = s'_0 = 4$ et $s'' = 5$. On note q la file de propagation. Un déroulement possible pour l'algorithme de propagation est le suivant :

— Modification de S'' (transformation locale) $\Rightarrow q = \{\mathcal{P}_1, \mathcal{P}_2\}$
— Propagation de $\mathcal{P}_1$: $S - S'' \leq -3$. Contrainte déjà vérifiée $\Rightarrow q = \{\mathcal{P}_2\}$
— Propagation de $\mathcal{P}_2$: $S' - S'' \leq -2$.
Réparation par modification de S' : $s' = 3$ $\Rightarrow q = \{\mathcal{R}\}$
— Propagation de $\mathcal{R}$: $(S - S' \leq -3) \vee (S' - S \leq -2)$.
Réparation par modification de S et S' (choix aléatoire) : $s = 2$, $s' = 0$ $\Rightarrow q = \{\mathcal{P}_1, \mathcal{P}_2\}$
— Propagation de $\mathcal{P}_1$: $S - S'' \leq -3$. Contrainte déjà vérifiée $\Rightarrow q = \{\mathcal{P}_2\}$
— Propagation de $\mathcal{P}_2$: $S' - S'' \leq -2$. Contrainte déjà vérifiée $\Rightarrow q = \emptyset$

Une solution réalisable a été trouvée. Elle vérifie $s = 2$, $s' = 0$, $s'' = 5$, comme illustré sur la Figure 3.15.

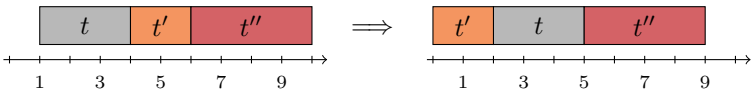


FIGURE 3.15 – Solutions initiale et réparée

Réparation d'une chaîne. Du point de vue de l'algorithme de réparation, une chaîne de longueur n n'est pas appréhendée comme une seule grande contrainte portant sur $O(n)$ variables, mais comme un ensemble de n inégalités distinctes. Les chaînes sont ainsi propagées un indice à la fois. Pour chaque indice i, l'inégalité correspondante dans la chaîne décrite par l'Equation 3.4 fait intervenir trois variables de décision : les deux variables numériques $X_{L[f(i)]}$ et $X_{L[g(i)]}$, ainsi que la variable de liste L. Contrairement aux cas précédemment évoqués, pour lesquels on ne s'intéressait

qu'à des inégalités binaires (seules ou au sein de disjonctions), dont la réparation reposait uniquement sur deux variables numériques, on peut désormais également réparer les inégalités considérées en modifiant la valeur de la variable de liste.

Plus précisément, on s'autorise à modifier la liste L en échangeant les éléments présents en position $f(i)$ et $g(i)$. On note $e_1 = L[f(i)]$, $e_2 = L[g(i)]$ et $e_3 = L[h(i)]$ les éléments respectivement en position $f(i)$, $g(i)$ et $h(i)$ dans la valeur courante de la variable de liste L. Ainsi, si l'on décide de ne pas modifier la valeur de L, l'inégalité à réparer est notée $\mathcal{C}$, vérifiant :

$$\mathcal{C} : \ aX_{e_1} + bX_{e_2} \leq c_{e_3}$$

Si l'on décide au contraire de réaliser l'échange, l'inégalité à réparer, alors notée $\mathcal{C}'$, devient :

$$\mathcal{C}' : \ aX_{e_2} + bX_{e_1} \leq c_{e_4}$$

où l'expression e_4 vérifie :

$$e_4 = e_2 \text{ si } e_3 = e_1 \quad ; \quad e_4 = e_1 \text{ si } e_3 = e_2 \quad ; \quad e_4 = e_3 \text{ sinon.}$$

La réparation de l'indice i d'une contrainte chaînée se fait alors de la façon suivante. Si l'inégalité $\mathcal{C}$ est respectée, elle est ignorée. Sinon, si l'inégalité $\mathcal{C}'$ est respectée, on échange les positions des éléments e_1 et e_2 dans la liste L, et on ajoute les autres contraintes impliquant les variables X_{e_1} et X_{e_2} à la file de propagation.

Sinon, il est nécessaire de modifier la valeur d'au moins une des deux variables numériques pour réparer l'inégalité. On commence par décider de réaliser l'échange des éléments e_1 et e_2 dans la liste L ou non :

— S'il est impossible de réparer l'inégalité $\mathcal{C}$, alors on réalise l'échange.
— Sinon, s'il est impossible de réparer l'inégalité $\mathcal{C}'$, alors on laisse la valeur de la liste inchangée.
— Sinon, on réalise l'échange avec une probabilité $\frac{1}{2}$.

Une fois la valeur de la liste L fixée, la procédure pour réparer l'inégalité ($\mathcal{C}'$ si l'on a choisi de réaliser l'échange, et $\mathcal{C}$ sinon) est la même que celle décrite plus haut pour la réparation d'une inégalité dans une disjonction.

Une autre particularité des contraintes chaînées se situe dans la façon dont elles sont ajoutées à la file de propagation. En effet, puisque les chaînes sont propagées un indice à la fois, on prend toujours soin de n'ajouter que des indices particuliers de la chaîne à la file de propagation, et non l'intégralité de la chaîne. Lorsqu'une variable numérique X intervenant dans l'expression d'une contrainte chaînée est modifiée, et que l'on ajoute à la file de propagation toutes les contraintes impliquant X (après la transformation locale, ou suite à la réparation d'une autre contrainte), on vérifie quels indices de la chaîne impliquent réellement la variable X, en fonction de la valeur courante de la variable de liste, et on les ajoute à la file de propagation. Il en est de même lorsque la variable X a été modifiée lors de la réparation d'un indice de la chaîne : on ajoute à la file de propagation tous les autres indices de la chaîne impliquant la variable X.

Exemple 3.8 (Réparation d'une chaîne). On considère un ensemble de quatre tâches t_0 à t_3, dont les dates de début sont modélisées par des variables de décision notées S_0 à S_3, et dont les durées sont $d_0 = 5$, $d_1 = 2$, $d_2 = 3$ et $d_3 = 4$. On suppose que ces tâches sont ordonnées par une variable de liste L, et sont liées par une contrainte de non-chevauchement exprimée sous la forme d'une chaîne $\mathcal{C}$:

$$\bigwedge_{i=0}^{2} \left(S_{L[i+1]} \geq S_{L[i]} + d_{L[i]}\right)$$

On considère une solution initiale réalisable $\mathcal{S}_0$, vérifiant $L = [1, 2, 0, 3]$, $s_{1,0} = 0$, $s_{2,0} = 3$, $s_{0,0} = 6$ et $s_{3,0} = 11$, représentée sur la Figure 3.16. On suppose que la date de début de t_3 a été modifiée lors de la transformation locale, et qu'on a désormais $s_3 = 10$: la solution courante $\mathcal{S}$ est infaisable,

et doit être réparée. On note q la file de propagation, et $\mathcal{C}[i]$ l'inégalité correspondant à l'indice i de la contrainte chaînée $\mathcal{C}$ pour toute valeur de i. Un déroulement possible pour l'algorithme de propagation est le suivant :

— Modification de S_3 (transformation locale) $\Rightarrow q = \{\mathcal{C}[2]\}$
— Propagation de $\mathcal{C}[2] : S_{L[3]} \geq S_{L[2]} + d_{L[2]}$.
  On choisit aléatoirement de laisser la liste inchangée.
  On doit donc réparer l'inégalité $S_3 \geq S_0 + 5$.
  Réparation par modification de S_0 : $s_0 = 5$ $\Rightarrow q = \{\mathcal{C}[1]\}$
— Propagation de $\mathcal{C}[1] : S_{L[2]} \geq S_{L[1]} + d_{L[1]}$.
  On choisit aléatoirement d'échanger les indices 1 et 2 dans la liste.
  On doit donc réparer l'inégalité $S_2 \geq S_0 + 5$.
  Réparation par modification de S_0 et S_2 (choix aléatoire) : $s_0 = 2$, $s_2 = 7$ $\Rightarrow q = \{\mathcal{C}[0], \mathcal{C}[2]\}$
— Propagation de $\mathcal{C}[0] : S_{L[1]} \geq S_{L[0]} + d_{L[0]}$.
  L'inégalité $S_0 \geq S_1 + 2$ est déjà vérifiée $\Rightarrow q = \{\mathcal{C}[2]\}$
— Propagation de $\mathcal{C}[2] : S_{L[3]} \geq S_{L[2]} + d_{L[2]}$.
  L'inégalité $S_3 \geq S_2 + 3$ est déjà vérifiée $\Rightarrow q = \emptyset$

Une solution réalisable a été trouvée. Elle vérifie $L = [1, 0, 2, 3]$, $s_1 = 0$, $s_0 = 2$, $s_2 = 7$ et $s_3 = 10$, comme illustré sur la Figure 3.16.

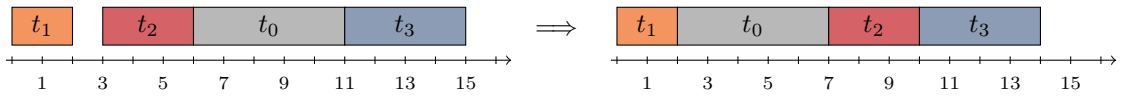


FIGURE 3.16 – Solutions initiale et réparée

Proposition 3.5. *Si la solution initiale $\mathcal{S}_0$ est réalisable, et s'il existe une solution réalisable respectant les décisions de la transformation locale, il y a toujours une probabilité non nulle pour que la propagation réussisse, en supposant que l'algorithme prenne toujours les « bonnes » décisions aléatoires lors des réparations successives des contraintes.*

Schéma de preuve. La preuve suit le même principe que celle de la Proposition 3.3 (lorsque l'on considère uniquement des contraintes binaires portant sur des variables booléennes, et des inégalités linéaires binaires portant sur des variables numériques, et s'il existe une solution réalisable respectant les décisions de la transformation locale, l'algorithme est garanti de la trouver). □

3.5.3 Extensions

Comme expliqué à la Section 3.4.3, on souhaite être capable de propager des contraintes dont l'expression ne correspond pas tout à fait aux Equations 3.1, 3.3 et 3.4 décrivant des inégalités linéaires binaires, ou des disjonctions et chaînes de telles inégalités. En effet, en étendant la détection à des contraintes dont l'expression est proche de ces équations, notre algorithme de réparation devient plus robuste aux variations du modèle, et permet ainsi au solveur d'obtenir de meilleures performances sur des problèmes plus complexes. Dans cette Section, on donne ainsi plusieurs exemples de modèles présentant des contraintes proches de disjonctions ou chaînes d'inégalités linéaires binaires.

Exemple 3.9 (Contrainte de non-chevauchement des objets dans un problème de packing). On considère un problème de packing en deux dimensions, dans lequel chaque objet i est un rectangle de longueur `l[i]` et de largeur `L[i]`, et sur lequel on peut éventuellement appliquer une rotation. Les différents objets ne doivent pas se chevaucher. On note `X[i]` et `Y[i]` les coordonnées du coin inférieur gauche de l'objet i. Les contraintes de non-chevauchement sont écrites sous la forme de disjonctions, en considérant les objets deux à deux. Ainsi, pour deux objets i et j, soit i est situé

après j sur l'axe des abscisses, soit j est situé après i sur l'axe des abscisses, soit i est situé après j sur l'axe des ordonnées, soit j est situé après i sur l'axe des ordonnées. Si l'on suppose que la rotation de chaque objet est fixée, chacune des quatre branches de cette disjonction correspond à une inégalité linéaire binaire. En pratique cependant, le terme exprimant la rotation de chaque objet n'est pas une constante, et intervient dans l'expression de la contrainte. Afin d'être capable de propager les contraintes de non-chevauchement entre les objets, on doit donc détecter des inégalités linéaires binaires sous leur forme paramétrée. On présente deux modélisations différentes pour le problème, correspondant à deux types de paramètres différents (paramètres booléens et paramètres *at*, définis dans la Section 3.4.3), et on donne une représentation des graphes d'évaluation correspondants.

— Première version : avec des paramètres booléens.
Pour tout objet i, on note `rotation[i]` une variable booléenne égale à 1 si et seulement si l'objet i est tourné de 90°. La taille de l'objet i selon l'axe des abscisses devient alors `rotation[i]*l[i] + (1-rotation[i])*L[i]`. Le Modèle 3.5 donne une ébauche de modélisation pour ce problème. Le graphe d'évaluation de la contrainte de non-chevauchement entre deux objets i et j (lignes 11 à 16 du Modèle 3.5) est représenté sur la Figure 3.17 (par symétrie et par souci de place, on ne représente que la première branche de la disjonction).

```
1  // Lecture des données : longueur et largeur de chaque objet.
2  l[i in 0..nbObjects-1];
3  L[i in 0..nbObjects-1];
4  // Variables entières : position des objets.
5  X[i in 0..nbObjects-1] <- int(0, maxX);
6  Y[i in 0..nbObjects-1] <- int(0, maxY);
7  // Variables booléennes : objet tourné de 90 degrés ou non.
8  rotation[i in 0..nbObjects-1] <- bool();
9
10 // Contrainte de non-chevauchement.
11 for [i in 0..nbObjects-1][j in i+1..nbObjects-1] {
12     constraint (X[i] >= X[j] + rotation[j]*l[j] + (1-rotation[j])*L[j])
13             || (X[j] >= X[i] + rotation[i]*l[i] + (1-rotation[i])*L[i])
14             || (Y[i] >= Y[j] + (1-rotation[j])*l[j] + rotation[j]*L[j])
15             || (Y[j] >= Y[i] + (1-rotation[i])*l[i] + rotation[i]*L[i]);
16 }
```

Modèle 3.5 – Exemple de modélisation de problème de packing 2D – avec des paramètres booléens

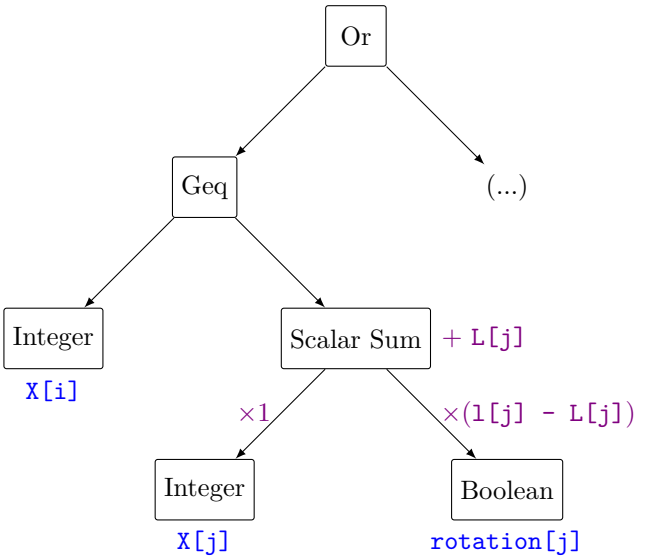


FIGURE 3.17 – Graphe d'évaluation de la contrainte de non-chevauchement – avec des paramètres booléens

— Deuxième version : avec des paramètres *at*.
Pour tout objet i et tout entier r entre 0 et le nombre maximum de rotations possibles pour l'objet i, on note `l[i][r]` et `L[i][r]` la taille de l'objet i selon les axes des abscisses et des ordonnées respectivement en choisissant la rotation r. La variable entière `rotation[i]` représente la rotation choisie pour l'objet i. La taille d'un objet i selon l'axe des abscisses est donc donnée par l'expression `l[i][rotation[i]]`. Le Modèle 3.6 donne une ébauche de modélisation pour ce problème. Le graphe d'évaluation de la contrainte de non-chevauchement entre deux objets i et j (lignes 11 à 14 du Modèle 3.6) est représenté sur la Figure 3.18 (par symétrie et par souci de place, on ne représente que la première branche de la disjonction).

```
1  // Lecture des données : longueur et largeur de chaque objet en fonction de sa rotation.
2  l[i in 0..nbObjects-1][r in 0..nbRotations-1];
3  L[i in 0..nbObjects-1][r in 0..nbRotations-1];
4  // Variables entières : position des objets.
5  X[i in 0..nbObjects-1] <- int(0, maxX);
6  Y[i in 0..nbObjects-1] <- int(0, maxY);
7  // Variables entières : rotation choisie pour chaque objet.
8  rotation[i in 0..nbObjects-1] <- int(0, nbRotations-1);
9
10 // Contrainte de non-chevauchement.
11 for [i in 0..nbObjects-1][j in i+1..nbObjects-1] {
12     constraint (X[i] >= X[j] + l[j][rotation[j]] || X[j] >= X[i] + l[i][rotation[i]])
13             || (Y[i] >= Y[j] + L[j][rotation[j]] || Y[j] >= Y[i] + L[i][rotation[i]]);
14 }
```

Modèle 3.6 – Exemple de modélisation de problème de packing 2D – avec des paramètres *at*

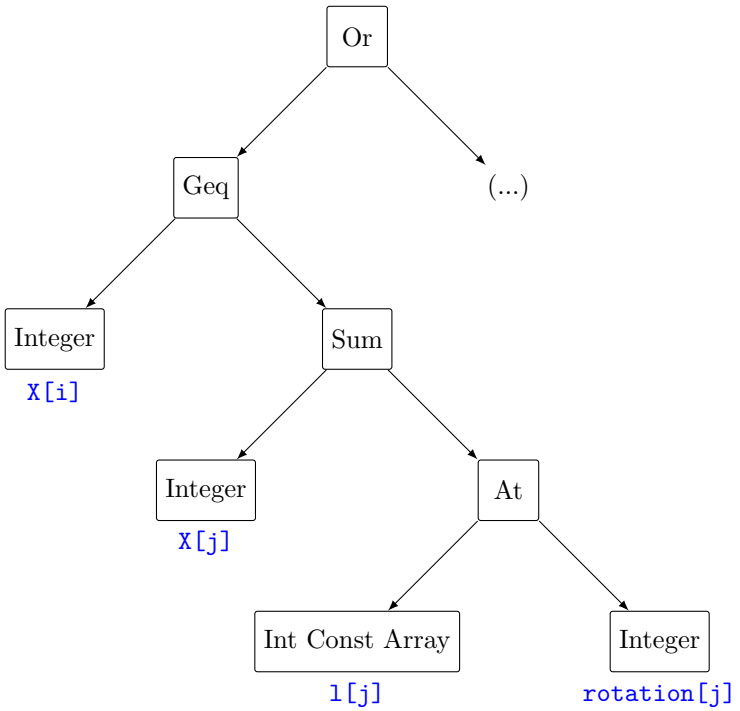


FIGURE 3.18 – Graphe d'évaluation de la contrainte de non-chevauchement – avec des paramètres *at*

On constate que les graphes d'évaluation correspondant à ces deux écritures de la contrainte de non-chevauchement des objets prennent là encore des formes très différentes. Dans les deux cas, les variables de décision représentant la rotation des objets sont considérées comme des « paramètres » des inégalités linéaires binaires des disjonctions formant les contraintes de non-chevauchement des

objets. Comme expliqué dans la Section 3.4.3, on considère que les valeurs de ces variables doivent être décidées par les mouvements de la recherche locale uniquement. Celles-ci ne sont donc pas modifiées au cours de la réparation des contraintes de non-chevauchement violées. La rotation de chaque objet est ainsi considérée comme fixe au cours de la propagation.

Exemple 3.10 (Contrainte de non-chevauchement des tâches dans le problème du Job Shop flexible). Comme expliqué à la Section 3.4.3, un autre type de paramètre que l'on souhaite détecter dans les inégalités linéaires binaires est le paramètre de type *find*, rencontré par exemple dans les problèmes d'ordonnancement avec durées flexibles. Le Modèle 3.7 donne une ébauche de modélisation pour le problème du Job Shop flexible [5]. Les différentes machines sont modélisées par des variables de listes, telles que `order[m]` représente l'ordre des tâches affectées à la machine m. Pour chaque tâche t, `start[t]` est une variable de décision entière représentant la date de début de t, `chosenMachine[t]` correspond à la machine à laquelle est affectée t, et `end[t]` correspond à la date de fin de t (sachant que t est ordonnancée sur la machine `chosenMachine[t]`).

```
1  // Lecture des données : durée de chaque tâche sur chaque machine.
2  duration[m in 0..nbMachines-1][t in 0..nbTasks-1];
3  // Variables de décision : date de début de chaque tâche.
4  start[t in 0..nbTasks-1] <- int(0, maxStart);
5  // Variables de décision : ordre des tâches sur chaque machine.
6  order[m in 0..nbMachines-1] <- list(nbTasks);
7  constraint partition(order);
8
9  // Expressions intermédiaires : machine choisie pour chaque tâche.
10 chosenMachine[t in 0..nbTasks-1] <- find(order, t);
11 end[t in 0..nbTasks-1] <- start[t] + duration[chosenMachine[t]][t];
12
13 // Contrainte de non-chevauchement des tâches sur chaque machine.
14 for [m in 0..nbMachines-1] {
15     constraint and(0..count(order[m])-2, i => start[order[m][i+1]] >= end[order[m][i]]);
16 }
```

Modèle 3.7 – Contrainte de non-chevauchement des tâches dans le problème du Job Shop flexible

On remarque que l'écriture de la contrainte de non-chevauchement (ligne 15 du Modèle 3.7) est très proche de celle donnée à la Section 3.5.1 (ligne 13 du Modèle 3.3) pour une contrainte de non-chevauchement sur des tâches de durées constantes. Cependant, les durées n'étant ici pas des constantes mais des expressions entières reposant sur un nœud de type « *find* », cette écriture aboutit à un graphe d'évaluation très différent au sein de LocalSolver. Le graphe d'évaluation de la fonction variadique associée à la contrainte de non-chevauchement des tâches du Modèle 3.7 est donné dans la Figure 3.19. Comme précédemment, certains nœuds sont dupliqués afin d'améliorer la lisibilité.

Une particularité du paramètre *find* lorsqu'il est utilisé au sein de contraintes chaînées, comme ici dans une contrainte de non-chevauchement des tâches affectées à une même machine, est que sa valeur est en réalité constante dans ce contexte. En effet, pour une machine m donnée, la contrainte de non-chevauchement associée à m est une contrainte variadique, qui itère sur les éléments appartenant à la variable de liste `order[m]`, c'est-à-dire sur les indices des tâches affectées à la machine m. Ainsi, pour toute tâche t rencontrée au sein de la contrainte, sa machine courante `chosenMachine[t]` est égale à m, et sa durée est connue et égale à `duration[m][t]`.

Ce cas de figure particulier pour le paramètre *find* est celui que l'on cherche à détecter dans les chaînes d'inégalités linéaires binaires. Afin de s'assurer que l'expression du paramètre sera en réalité toujours constante lors de la propagation de la contrainte, on doit vérifier lors de la phase de détection que la valeur du nœud « *find* » est constante, et égale à l'indice de la liste sur laquelle porte la contrainte. Pour cela, on vérifie que lorsqu'on accède à un élément du tableau faisant intervenir les nœuds de type « *find* » (le tableau `end` ici), on accède bien à un indice appartenant à

5. Le modèle LocalSolver complet pour le problème du Job Shop flexible est décrit dans la Section 2.3.2.

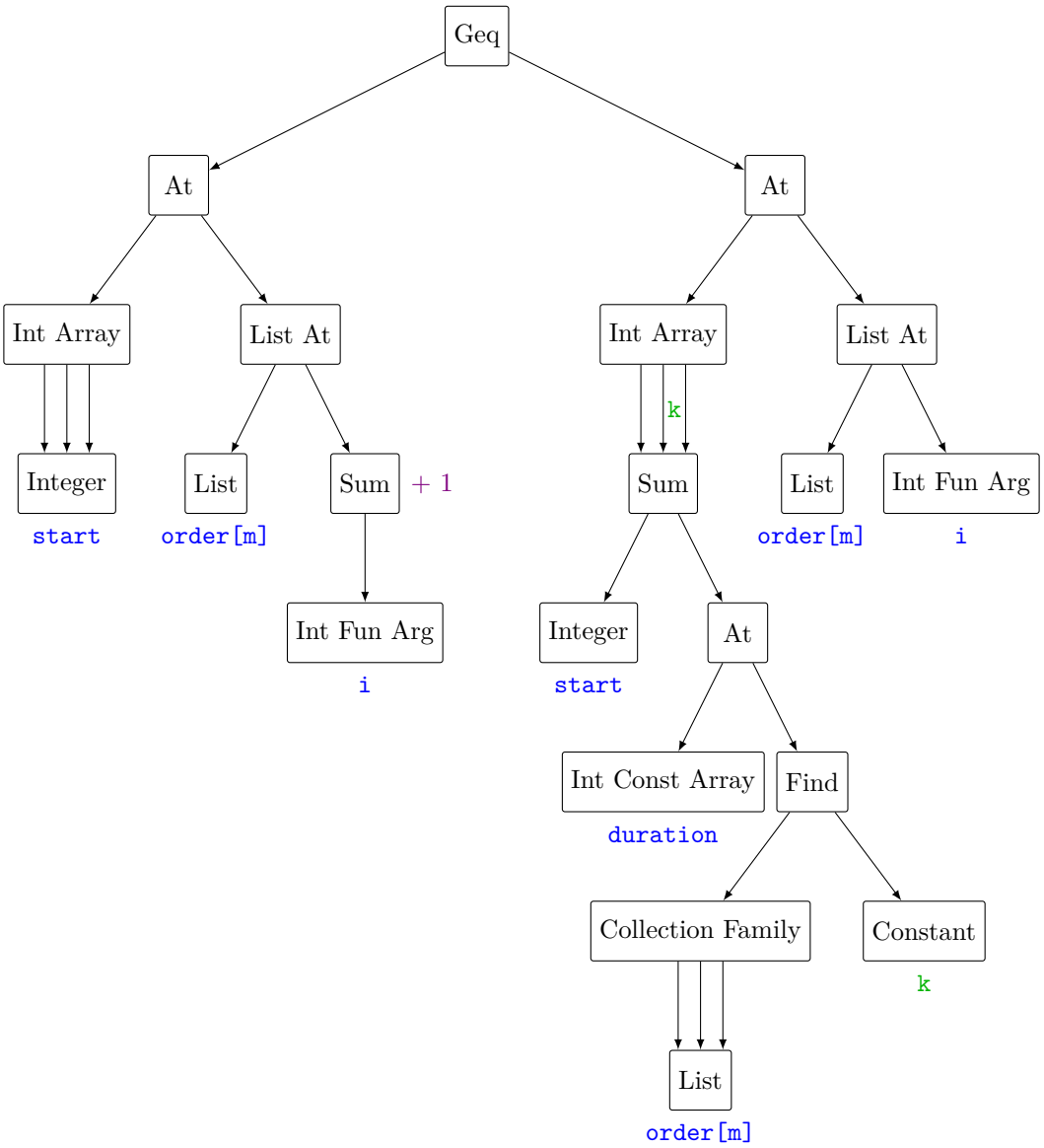


FIGURE 3.19 – Graphe d'évaluation de la contrainte de non-chevauchement – avec des durées flexibles

la liste considérée (indice `order[m][i]` ici). Dans le cas de l'exemple considéré ici, cela correspond à vérifier que l'on accède bien à la durée d'une tâche affectée à la machine m, et non à une autre machine quelconque. Dans le graphe d'évaluation de la contrainte, cela correspond à vérifier deux propriétés sur le nœud « *at* » représentant l'expression considérée (l'expression `end[order[m][i]]` ici, correspondant au nœud « *at* » en haut à droite de la Figure 3.19). Tout d'abord, le deuxième fils de ce nœud (l'indice auquel on accède) doit être de type « *list at* », et porter sur la liste considérée dans la contrainte (`order[m]` ici). Ensuite, on doit vérifier que la k-ième case du tableau est bien associée à l'élément `k` (élément de la liste, représentant une tâche). Pour cela, on vérifie que le deuxième enfant du nœud « *find* » présent dans la case numéro k dans le tableau est une constante égale à k (les deux « k » sont représentés en vert sur la Figure 3.19).

Exemple 3.11 (Présence de tâches constantes dans un problème d'ordonnancement disjonctif). On rencontre parfois des problèmes dans lesquels certaines tâches sont fixes, ayant une date de début et une durée constantes. Plusieurs situations peuvent amener à la présence de ce type de tâches. Par

exemple, dans un problème très contraint, les bornes d'une variable représentant la date de début d'une tâche peuvent être resserrées jusqu'à ne plus être qu'un singleton. On peut également choisir de définir des tâches fictives constantes pour modéliser des plages de temps pendant lesquelles une machine est indisponible, et ne peut traiter aucune tâche « réelle ». Dans les deux cas, ces tâches constantes sont impliquées dans les contraintes de non-chevauchement sur les machines. Contrairement aux cas précédemment évoqués, le tableau `start` contenant les expressions entières modélisant les dates de début des tâches ne contient plus uniquement des variables de décision entières, mais également des constantes. La forme du graphe d'évaluation correspondant est alors légèrement transformée, ce qui rend la détection des chaînes d'inégalités linéaires binaires plus complexe.

Lorsque toutes les tâches ont une date de début variable, il suffit de vérifier que les variables entières intervenant dans la i-ème case des tableaux `start` et `end` sont les mêmes pour toute valeur de i. Cependant, si certaines dates de début sont constantes, les cases `start[i]` et `end[i]` ne contiennent pas les mêmes nœuds constants. Par exemple, on considère un ensemble de quatre tâches t_0 à t_3, dont les dates de début sont des variables entières notées S_0 à S_3, et dont les durées sont $d_0 = 4$, $d_1 = 6$, $d_2 = 3$ et $d_3 = 7$. On suppose que la tâche t_2 est fixe : sa date de début est constante, et vaut $S_2 = 20$. Le contenu des tableaux `start` et `end` est donné dans la Table 3.1.

	`start[0]`	`start[1]`	`start[2]`	`start[3]`
Type	*integer*	*integer*	*constant*	*integer*
Détail	S_0	S_1	20	S_3

	`end[0]`	`end[1]`	`end[2]`	`end[3]`
Type	*sum*	*sum*	*constant*	*sum*
Détail	$S_0 + 4$	$S_1 + 6$	23	$S_3 + 7$

TABLE 3.1 – Contenu des tableaux `start` et `end`

On constate que la case `start[0]` correspond bien à la variable de décision S_0, et que la case `end[0]` est bien un nœud somme, ayant pour unique enfant S_0. Le même schéma est répété pour les tâches t_1 et t_3. Pour la tâche t_2 cependant, les cases `start[2]` et `end[2]` contiennent deux nœuds constants différents. La durée de t_2 n'est pas explicitement donnée. On peut toutefois retrouver facilement sa valeur : $d_2 =$ `end[2]` $-$ `start[2]` $= 3$.

Exemple 3.12 (Ordonnancement disjonctif avec temps de transition). Certains problèmes d'ordonnancement disjonctif font également intervenir des temps de transition à respecter entre deux tâches consécutives. Dans ce cas, l'écriture de la contrainte de non-chevauchement des tâches ordonnancées sur une même machine change, et devient

```
1 and(0..nbTasks-2, i => start[order[i+1]] >= end[order[i]] + setup[order[i]][order[i+1]]);
```

Afin d'être capable de propager ce type de contraintes de non-chevauchement, on étend la détection des chaînes d'inégalités linéaires binaires aux équations de la forme

$$\bigwedge_i \left(aX_{L[f_1(i)]} + bX_{L[f_2(i)]} \leq c_{L[f_3(i)]} + d_{L[f_4(i)],L[f_5(i)]} \right) \tag{3.5}$$

où X est un tableau de variables de décision entières ou réelles, L est une variable de liste, a, b, et les éléments des tableaux c et d sont des constantes, et les f_i sont des fonctions entières quelconques.

3.6 Réparation d'inégalités linéaires ternaires portant sur des variables numériques

On étend le mécanisme de réparation aux inégalités linéaires ternaires, de la forme

$$aX + bY + cZ \leq d \tag{3.6}$$

où X, Y et Z sont des variables de décision entières ou réelles, et où a, b, c et d sont des constantes.

Remarque 3.5. Le cas particulier où $a = c = 1$ et $b = -1$ correspond aux contraintes de précédence généralisées entre deux tâches de durées variables. En effet, si l'on considère deux tâches t et t', dont les dates de début et les durées sont respectivement modélisées par les variables de décision S_t et $S_{t'}$, et D_t et $D_{t'}$, alors la contrainte de précédence entre t et t' peut s'écrire

$$S_{t'} \geq S_t + D_t$$

ce qui correspond bien à la forme de l'Equation 3.6.

On considère également les disjonctions et chaînes d'inégalités linéaires ternaires, de la forme

$$\bigvee_i (a_i X_i + b_i Y_i + c_i Z_i \leq d_i) \tag{3.7}$$

$$\bigwedge_i \left(aX_{L[f_1(i)]} + bX_{L[f_2(i)]} + cY_{L[f_3(i)]} \leq d_{L[f_4(i)]}\right) \tag{3.8}$$

Remarque 3.6. Comme précédemment, si tous les a et c sont égaux à 1 et tous les b à -1, ces équations décrivent des contraintes de non-chevauchement entre des tâches de durées variables (considérées deux à deux dans le cas d'une disjonction, et ordonnées par la variable de liste L dans le cas d'une chaîne).

Remarque 3.7. Les extensions décrites dans les Sections 3.4.3 et 3.5.3 – paramètres (booléens, *at*, *find*), temps de transition, tâches constantes – s'appliquent également aux inégalités linéaires ternaires, ainsi qu'aux disjonctions et chaînes de telles inégalités.

Mécanisme de réparation. Comme dans la Section 3.5, lorsque la contrainte courante est une inégalité linéaire ternaire, ou une disjonction ou une chaîne de telles inégalités, il existe rarement une réparation nécessaire. L'algorithme de filtrage appliqué à ces contraintes est alors non déterministe, et repose là encore sur des choix aléatoires lorsqu'il n'existe aucune réparation nécessaire.

La réparation d'une contrainte linéaire ternaire violée $\mathcal{C}$, d'équation $aX + bY + cZ \leq d$, se déroule de la façon suivante. Si une ou deux de ses variables seulement peuvent être modifiées dans le sens de la réparation, alors la contrainte $\mathcal{C}$ est réparée comme une inégalité linéaire binaire dans une disjonction (voir Section 3.5). Si les trois variables peuvent être modifiées dans le sens de la réparation, l'algorithme choisit entre onze méthodes de réparation équiprobables :

- On répare la contrainte en modifiant une seule variable.
- On répare la contrainte en modifiant deux variables sur les trois, de façon équitable.
- On répare la contrainte en modifiant deux variables sur les trois, en répartissant la part de réparation attribuée à chacune de façon aléatoire.
- On répare la contrainte en modifiant les trois variables, de façon équitable.
- On répare la contrainte en modifiant les trois variables, en répartissant la part de réparation attribuée à chacune de façon aléatoire.

Le mécanisme de réparation d'une disjonction ou chaîne d'inégalités linéaires ternaires est similaire à celui décrit dans la Section 3.5 pour la réparation des disjonctions et chaînes d'inégalités linéaires binaires.

Exemple 3.13 (Réparation de contraintes ternaires). On considère un problème d'ordonnancement avec trois tâches, t, t' et t'', dont les dates de début et les durées sont modélisées par des variables de décision entières respectivement notées S, S' et S'', et D, D' et D''. Le problème comporte deux contraintes de précédence : $\mathcal{P}_1 : S'' \geq S + D$ et $\mathcal{P}_2 : S'' \geq S' + D'$, et une contrainte de non-chevauchement $\mathcal{R} : (S' \geq S + D) \vee (S \geq S' + D')$. On suppose que la solution initiale réalisable $\mathcal{S}_0$, représentée sur la Figure 3.20, est telle que $s_0 = 0$, $d_0 = 3$, $s'_0 = 3$, $d'_0 = 2$, $s''_0 = 5$ et $d''_0 = 4$. On suppose que la durée de t' a été modifiée lors de la transformation locale, et qu'on a désormais $d' = 4$: la solution courante $\mathcal{S}$ est infaisable, et doit être réparée. On note q la file de propagation. Un déroulement possible de l'algorithme de propagation est le suivant :

— Modification de D' (transformation locale) $\Rightarrow q = \{\mathcal{P}_2, \mathcal{R}\}$
— Propagation de $\mathcal{P}_2 : S'' \geq S' + D'$.
  Réparation par modification de S' et S'' (choix aléatoire) : $s' = 2$, $s'' = 6$ $\Rightarrow q = \{\mathcal{R}, \mathcal{P}_1\}$
— Propagation de $\mathcal{R} : (S' \geq S + D) \vee (S \geq S' + D')$.
  Réparation par modification de S et S' (choix aléatoire) : $s = 4$, $s' = 0$ $\Rightarrow q = \{\mathcal{P}_1, \mathcal{P}_2\}$
— Propagation de $\mathcal{P}_1 : S'' \geq S + D$.
  Réparation par modification de D (choix aléatoire) : $d = 2$ $\Rightarrow q = \{\mathcal{P}_2, \mathcal{R}\}$
— Propagation de $\mathcal{P}_2 : S'' \geq S' + D'$. Déjà vérifiée $\Rightarrow q = \{\mathcal{R}\}$
— Propagation de $\mathcal{R} : (S' \geq S + D) \vee (S \geq S' + D')$. Déjà vérifiée $\Rightarrow q = \emptyset$

Une solution réalisable a été trouvée. Elle vérifie $s = 4$, $d = 2$, $s' = 0$, $d' = 4$, $s'' = 6$ et $d'' = 4$, comme illustré sur la Figure 3.20.

FIGURE 3.20 – Solutions initiale et réparée

3.7 Résultats numériques

Dans cette Section, on s'intéresse aux gains de performance apportés par l'ajout de notre algorithme de réparation de solutions au sein de la recherche locale de LocalSolver. Tous les résultats numériques donnés dans cette Section ont été calculés sur une machine équipée d'un processeur Intel i7 8750 à 2.20 GHz et de 16 Go de mémoire RAM.

Remarque 3.8. Dans les Sections précédentes, on a toujours supposé que l'itération de recherche locale considérée commençait avec une solution initiale $\mathcal{S}_0$ réalisable, afin de pouvoir garantir certaines propriétés souhaitables. Cependant, le mécanisme de réparation peut également être appliqué lorsque $\mathcal{S}_0$ est irréalisable, avec une probabilité de succès plus faible. De ce fait, il est également appelé dans les premières étapes de la recherche locale de LocalSolver, avant qu'une solution réalisable ne soit trouvée. Les résultats numériques présentés dans cette Section ont ainsi été obtenus en activant le mécanisme de réparation à chaque itération de la recherche locale, que celles-ci commencent avec une solution réalisable ou non.

3.7.1 Réparation de contraintes binaires dans des problèmes d'ordonnancement d'atelier

Dans cette Section, on examine les problèmes du Job Shop, de l'Open Shop, du Job Shop flexible, et du Job Shop flexible avec temps de transition, décrits dans les Sections 1.3.1 et 2.3.2. Dans chacun de ces problèmes, n jobs sont divisés en m activités chacun (une activité par machine pour les problèmes du Job Shop et de l'Open Shop). Dans les problèmes du Job Shop flexible avec et sans temps de transition, on doit décider de l'affectation de chaque tâche à une machine compatible. Les machines sont disjonctives : les contraintes de non-chevauchement des tâches qui

leur sont associées peuvent être modélisées en utilisant soit $O(mn^2)$ disjonctions (pour les problèmes du Job Shop et de l'Open Shop seulement, puisqu'on ne peut utiliser les disjonctions que lorsque l'affectation des tâches aux machines est fixe), soit $O(m)$ chaînes de taille $O(n)$. Dans les problèmes du Job Shop et du Job Shop flexible avec et sans temps de transition, les activités de chaque job sont ordonnées : il y a $O(n^2)$ contraintes de précédence, tandis que dans le problème de l'Open Shop, les jobs peuvent être considérés comme des ressources disjonctives. Pour chacun des quatre problèmes, l'objectif est de minimiser le makespan.

On compare les performances de LocalSolver avec et sans notre mécanisme de réparation sur chacun de ces problèmes. Pour le problème du Job Shop, on considère quatre classes d'instances classiques : la classe FT de Fisher et Thompson [40], la classe LA de Lawrence [62], la classe ORB d'Applegate et Cook [6], et la classe TAI de Taillard [92]. Pour le problème de l'Open Shop, on considère les instances classiques de Taillard [91]. Pour le problème du Job Shop flexible, on utilise les instances de Barnes, Behnke, Brandimarte, Dauzere, Fattahi, Kacem et Hurink, pour lesquelles la valeur de la meilleure solution connue est donnée par [14]. Pour le problème du Job Shop flexible avec temps de transition, on utilise les instances de Job Shop flexible, auxquelles on ajoute des temps de transition aléatoires, dont l'ordre de grandeur varie entre 10% et la valeur moyenne des durées et cinq fois cette valeur.

3.7.1.1 Premiers résultats

On commence par présenter les résultats obtenus avec LocalSolver 9.0 (août 2019), première version de LocalSolver intégrant notre mécanisme de réparation. Les Tables 3.2 et 3.3 donnent l'écart moyen à l'optimum avec et sans réparation, après 10 et 60 secondes de calcul, sur les problèmes du Job Shop et de l'Open Shop respectivement [6]. Les résultats présentés ici correspondent aux premiers travaux de la thèse, et montrent que l'ajout de notre algorithme de réparation au sein de LocalSolver a rapidement permis une très nette amélioration de ses performances. On voit en effet que celui-ci atteint des solutions de qualité en des temps de calcul très courts. On constate de plus qu'une grande proportion (77% en moyenne pour le problème du Job Shop, et 62% en moyenne pour le problème de l'Open Shop) des mouvements améliorants appliqués lors de la recherche locale conduisent à une solution initialement infaisable, mais réparée avec succès par notre algorithme.

	Nombre d'instances	Gap 10s avec rép.	Gap 10s sans rép.	Gap 60s avec rép.	Gap 60s sans rép.	% mouvements réparés
FT	3	5%	73%	2%	15%	75%
LA	40	8%	246%	3%	91%	76%
ORB	10	6%	120%	3%	22%	81%

Table 3.2 – Écart à l'optimum sur le problème du Job Shop

Taille (jobs × machines)	Nombre d'instances	Gap 10s avec rép.	Gap 10s sans rép.	Gap 60s avec rép.	Gap 60s sans rép.	% mouvements réparés
4×4	10	0.4%	41%	0.0%	41%	52%
5×5	10	2.7%	65%	1.3%	65%	58%
7×7	10	5.0%	102%	2.8%	93%	65%
10×10	10	7.2%	569%	4.8%	261%	72%

Table 3.3 – Écart à l'optimum sur le problème de l'Open Shop

6. La première version de notre mécanisme de réparation évaluée ici n'intégrait pas encore la détection de tous les types de contraintes évoqués dans ce Chapitre, et ne permettait notamment pas encore de traiter les problèmes du Job Shop flexible avec et sans temps de transition.

Bien que positifs, on peut noter que ces résultats restent en deçà de ceux des algorithmes d'ordonnancement dédiés ou des méthodes spécialisées (ordonnancement basé sur des contraintes, recherche locale basée sur des graphes disjonctifs) présentés par exemple dans [89] et [97] pour le problème du Job Shop, et dans [48] pour le problème de l'Open Shop. Toutefois, comme déjà mentionné au début du Chapitre, on s'intéresse ici à des formes plus générales des contraintes de précédence et non-chevauchement des tâches, et on souhaite garder des éléments de modélisation simples et non restreints à des types de problèmes spécifiques.

3.7.1.2 Résultats récents

On présente ici des résultats numériques plus récents, obtenus avec LocalSolver 11.0 (mars 2022). Différents algorithmes visant à obtenir des gains de performance sur les problèmes d'ordonnancement disjonctif ont été implémentés entre les versions 9.0 et 11.0 de LocalSolver. Parmi ces algorithmes, on trouve notamment les mouvements de recherche locale présentés dans le Chapitre 2, mais également des algorithmes constructifs que l'on ne détaillera pas dans cette thèse. Du fait de l'amélioration générale des résultats, l'impact de notre mécanisme de réparation sur les performances du solveur est désormais plus faible, mais il reste néanmoins visible.

Pour chacun des quatre problèmes cités au début de la Section, et pour chaque catégorie d'instances, la Table 3.4 donne la moyenne du pourcentage d'amélioration apporté par notre mécanisme de réparation, ainsi que l'écart moyen à l'optimum ou à la meilleure solution connue avec et sans le mécanisme de réparation, après 60 secondes de calcul. Les instances de Job Shop flexible avec temps de transition ayant été générées aléatoirement, on ne dispose pas de la valeur de la solution optimale, ou de la meilleure solution connue selon la littérature : on compare donc nos résultats à ceux obtenus par le solveur de programmation par contraintes CP Optimizer 20.1.0, après 60 secondes de calcul également.

Problème	Instances	Nombre d'instances	Amélioration	Gap avec réparation	Gap sans réparation
Job Shop	FT	3	0.1%	1.2%	1.3%
	LA	40	0.8%	1.4%	2.2%
	ORB	10	1.3%	1.9%	3.2%
	TAI	80	0.9%	7.6%	8.6%
Job Shop flexible	Barnes	21	4.4%	1.4%	5.9%
	Behnke	60	2.5%	3.9%	6.5%
	Brandimarte	10	5.0%	5.0%	10.4%
	Dauzere	18	5.5%	1.3%	6.9%
	Fattahi	20	0.6%	1.6%	2.2%
	Kacem	4	0.0%	2.3%	2.3%
	Hurink – e	66	3.5%	1.9%	5.5%
	Hurink – r	66	2.8%	2.1%	4.9%
	Hurink – v	66	0.3%	0.5%	0.8%
Job Shop flexible avec temps de transition	$\simeq$ 10% durées	104	1.7%	2.8%	4.6%
	$\simeq$ durées	105	1.5%	1.8%	3.3%
	$\simeq 5\times$ durées	104	1.2%	2.5%	3.7%
Open Shop	4×4	10	1.9%	0.0%	2.1%
	5×5	10	0.9%	1.0%	1.9%
	7×7	10	0.0%	0.2%	0.3%
	10×10	10	0.1%	0.0%	0.1%
	15×15	10	0.0%	0.0%	0.0%
	20×20	10	0.0%	0.0%	0.0%

TABLE 3.4 – Comparaison des performances de LocalSolver avec et sans le mécanisme de réparation

3.7.2 Réparation de contraintes ternaires dans le problème du Unit Commitment

Dans cette Section, on considère le problème du Unit Commitment, récemment étudié dans [38]. On considère une version simplifiée et purement combinatoire du problème, dans laquelle le niveau de production de chaque usine allumée est fixé à son taux de production moyen. La modélisation LocalSolver du problème, présentée en détail dans la Section 2.3.2, est la suivante. Les plages de production de chaque usine sont modélisées comme des tâches de durées variables, pouvant être affectées à n'importe quelle usine. Les usines sont alors assimilées à des ressources disjonctives, sur lesquelles on impose des contraintes de non-chevauchement des tâches, écrites sous forme de chaînes : deux plages de production consécutives sur une usine doivent être séparées par au moins la durée minimale de non-fonctionnement de l'usine. La date de début et la durée de chaque tâche étant variables, ces contraintes reprennent la forme des chaînes d'inégalités linéaires ternaires décrites dans la Section 3.6.

Dans la Table 3.5, on mesure les gains de performance apportés par notre algorithme de réparation, en 10 et 60 secondes de calcul, sur LocalSolver 9.5 (mars 2020). On donne également le pourcentage de transformations locales améliorantes ayant eu besoin d'être réparées. On utilise les instances de [4] : le nombre d'usines varie de 10 à 100, et le nombre de pas de temps est de 24.

Nombre d'usines	Nombre d'instances	Amélioration 10s	Amélioration 60s	% mouvements réparés
10	30	4%	3%	3%
20	27	7%	6%	3%
50	25	13%	10%	3%
75	19	12%	12%	4%
100	25	7%	14%	4%

TABLE 3.5 – Amélioration des performances – problème du Unit Commitment simplifié

Bien que le taux de mouvements ayant nécessité une réparation pour être améliorants soit assez faible (entre 3% et 4%), on constate que la réparation des contraintes ternaires permet d'améliorer les résultats obtenus par LocalSolver sur le problème du Unit Commitment. Le gain de performance observé est particulièrement important sur les instances de plus grande taille.

3.8 Conclusion

Dans ce Chapitre, on a examiné une famille de problèmes d'optimisation, caractérisée par un réseau d'inégalités linéaires binaires et ternaires. Ce type de réseau permet notamment de modéliser des contraintes de précédence ou de non-chevauchement généralisées, portant sur des tâches de durées fixes (inégalités binaires) ou variables (inégalités ternaires). On a introduit un algorithme de réparation de solutions basé sur de la propagation de contraintes, qui surmonte les difficultés rencontrées par les algorithmes de recherche locale à voisinages restreints présents dans LocalSolver sur les problèmes visés. Pour fonctionner, l'algorithme s'appuie sur une détection robuste des nombreuses formes pouvant être prises par les contraintes que l'on souhaite propager, pouvant varier fortement en fonction de l'écriture du modèle. Notre algorithme de propagation présente deux spécificités principales. D'une part, une réduction de domaine n'est propagée que si elle exclut la valeur actuelle de la variable. D'autre part, la valeur de chaque variable doit toujours être modifiée dans la même direction. On a également décrit certaines propriétés souhaitables sur les contraintes, qui assurent le succès de la procédure de réparation.

La principale limite de notre approche est la possibilité d'échec sur des réparations de contraintes complexes, ce qui entraîne le rejet du mouvement proposé. Toutefois, cette limitation est largement tempérée en pratique par la rapidité du processus global d'itération, qui permet de tester

un grand nombre de mouvements dans un court laps de temps, et qui conduit généralement à des réparations réussies. En conséquence, son intégration dans LocalSolver améliore nettement ses performances sur les problèmes ciblés, non seulement sur les problèmes d'ordonnancement classiques tels que les problèmes du Job Shop, de l'Open Shop, du Job Shop flexible avec ou sans temps de transition et du Unit Commitment, mais également sur certaines instances industrielles de packing multidimensionnel et d'extraction minière.

Chapitre 4

Le problème de l'Assembly Line Balancing

4.1 Introduction et définition du problème

Le problème de l'Assembly Line Balancing auquel on s'intéresse dans ce Chapitre est décrit dans [25] de la façon suivante. On considère un ensemble de n tâches, dont les durées sont connues et fixes. Les tâches sont partiellement ordonnées par des relations de précédence entre certaines paires de tâches. Le problème consiste à définir une séquence de « stations de travail » réalisant l'ensemble des tâches. Chaque station de travail est ainsi dédiée à la réalisation d'un sous-ensemble de tâches. La durée totale passée dans chaque station (la somme des durées des tâches qui sont affectées à cette station) ne doit pas excéder un certain temps de cycle c. Afin de respecter les relations de précédence entre les tâches, aucune tâche ne peut être affectée à une station de travail strictement plus tardive que l'une de ses tâches suivantes. Deux tâches liées par une relation de précédence peuvent cependant être affectées à une même station. L'objectif du problème est alors de minimiser le nombre de stations de travail utilisées.

Ce problème est un problème classique de la recherche opérationnelle, connu sous le nom de « type 1 Simple Assembly Line Balancing Problem » (SALB-1). Une vue d'ensemble de diverses méthodes de résolution du problème est donnée dans [12, 85], ou plus récemment dans [64]. D'autres articles récents s'intéressent à différentes variantes du problème. Par exemple, [78] étudie la version robuste du problème de l'Assembly Line Balancing, tandis que [22] considère le Robotic Assembly Line Balancing Problem.

La Figure 4.1 illustre un exemple d'instance et de solution du problème. On considère ici dix tâches, notées t_0 à t_9. Les relations de précédence entre les tâches sont représentées au moyen d'un graphe de précédence (par exemple, la tâche t_0 doit être exécutée dans une station de travail de numéro inférieur ou égal à celle des tâches t_1 et t_4). La durée de chaque tâche est représentée en

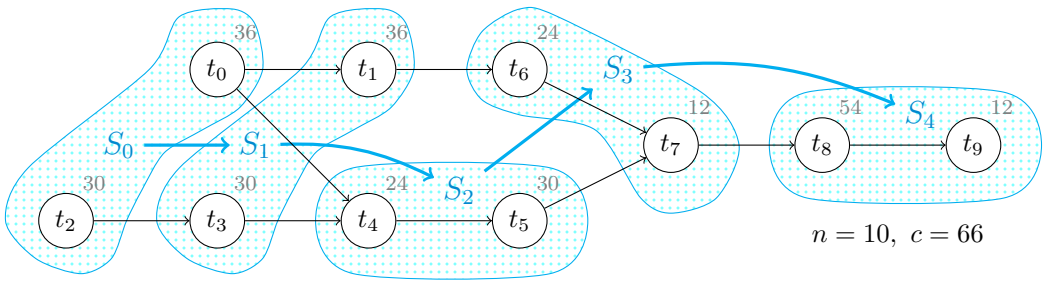


FIGURE 4.1 – Exemple : dix tâches réparties dans cinq stations de travail

gris en haut à droite du nœud correspondant. Le temps de cycle à respecter est de $c = 66$. La solution optimale comporte cinq stations de travail, notées S_0 à S_4, représentées en bleu sur la Figure 4.1.

Notations. Dans l'ensemble du Chapitre, on utilisera les notations suivantes pour décrire le problème de l'Assembly Line Balancing.

- Le nombre de tâches est noté n.
- Les tâches sont exprimées par des lettres minuscules : par exemple t, t', t_1, t_2.
- Les stations sont exprimées par des lettres majuscules : par exemple S, S', S_1, S_2. Une station est un ensemble de tâches : par exemple $S = \{t_1, t_2\}$.
- La station choisie pour une tâche t est notée $S(t)$.
- La durée d'une tâche t est notée d_t.
- La durée totale passée dans une station de travail S est notée $D(S) = \sum_{t \in S} d_t$.
- Le temps de cycle est noté c.
- On note $\preceq$ la relation de précédence entre deux tâches ou deux stations. Pour deux stations S_1 et S_2, $S_1 \preceq S_2$ signifie que S_1 est placée avant S_2 dans l'ordre des stations : l'ensemble des tâches attribuées à S_1 sera effectué avant l'ensemble des tâches affectées à S_2. Pour deux tâches t_1 et t_2, $t_1 \preceq t_2$ signifie que t_1 et t_2 sont liées par une relation de précédence : t_2 doit être affectée à une station de travail plus tardive que t_1 (ou à la même station). Ainsi, si $t_1 \preceq t_2$, alors dans toute solution réalisable, on a $S(t_1) \preceq S(t_2)$.

Dans ce Chapitre, on montre comment les algorithmes implémentés au sein de LocalSolver, bien que génériques, en font un solveur de choix pour résoudre le problème de l'Assembly Line Balancing. Dans la suite de la Section 4.1, on présente la modélisation du problème avec LocalSolver, à partir de variables de décision ensemblistes (variables de sets). On présente également le benchmark utilisé dans tout le Chapitre pour évaluer les performances des algorithmes présentés. Dans la Section 4.2, on s'intéresse au graphe de précédence révélé par le modèle à base de variables de sets. On présente un algorithme constructif de répartition des tâches dans les stations. Cet algorithme est utilisé d'une part pour construire une solution initiale réalisable, et d'autre part au sein d'un mouvement de la recherche locale. Dans la Section 4.3, on s'intéresse à la dimension « packing » du problème, mise en évidence par la formulation à base de variables de sets. On présente deux mouvements de recherche locale construits sur le principe des chaînes d'éjection, et applicables à tous types de problèmes présentant une structure de packing. Enfin, dans la Section 4.4, on présente une synthèse des gains de performance apportés par l'intégration du travail de la thèse au sein de LocalSolver sur le problème de l'Assembly Line Balancing. Les contributions algorithmiques de ce Chapitre ont été ou seront présentées lors des conférences ROADEF 2022 [21] et PMS 2022 [19].

4.1.1 Modélisation du problème avec LocalSolver

La modélisation la plus simple et naturelle du problème avec LocalSolver utilise des variables ensemblistes [1] pour représenter les stations. Dans le formalisme de modélisation de LocalSolver, une variable de set de domaine n représente un sous-ensemble (non-ordonné) de $\{0, ..., n-1\}$. La valeur d'une variable de set est donc un ensemble, et non un simple nombre. On associe donc naturellement chaque station à une variable de set : la valeur de la variable de set correspond alors au sous-ensemble des tâches affectées à cette station.

Le code LSP complet de la modélisation ensembliste du problème de l'Assembly Line Balancing est donné par le Modèle 4.1.

```
1 // Variables de sets : stations[s] contient les tâches assignées à la station s.
2 stations[s in 0..maxNbStations-1] <- set(nbTasks);
3
```

1. Voir Section 1.2.1.

```
4  // Chaque tâche est assignée à exactement une station.
5  constraint partition(stations);
6
7  // chosenStation[t] est le numéro de la station à laquelle est assignée la tâche t.
8  chosenStation[t in 0..nbTasks-1] <- find(stations, t);
9
10 // Chaque tâche doit être réalisée avant ses successeurs.
11 for [t in 0..nbTasks-1][succ in successors[t]] {
12     constraint chosenStation[t] <= chosenStation[succ];
13 }
14
15 // La durée totale passée dans chaque station ne doit pas excéder le temps de cycle.
16 for [s in 0..maxNbStations-1] {
17     stationTime[s] <- sum(stations[s], t => processingTime[t]);
18     constraint stationTime[s] <= cycleTime;
19 }
20
21 // Minimisation du nombre de stations utilisées.
22 stationUsed[s in 0..maxNbStations-1] <- count(stations[s]) > 0;
23 nbStations <- sum[s in 0..maxNbStations-1] (stationUsed[s]);
24 minimize nbStations;
```

Modèle 4.1 – Code LSP pour le problème de l'Assembly Line Balancing

On commence par définir les variables de décision du problème : des variables de sets, de domaine n (noté `nbTasks` dans le modèle), représentant les stations de travail (ligne 2). On définit autant de variables de sets qu'il y a de tâches à placer (`maxNbStations` est égal à `nbTasks`). En effet, une solution réalisable triviale consiste à ranger les tâches selon un ordre topologique respectant les relations de précédence entre elles, et à placer la i-ème tâche de cet ordre dans la station en position i. La variable `stations[s]` représente alors l'ensemble des tâches attribuées à la station de travail en position s. On impose une contrainte de partition sur l'ensemble des stations (ligne 5). On s'assure ainsi que dans toute solution réalisable, chaque élément de $\{0, ..., n-1\}$ est présent exactement une fois sur l'ensemble des variables de sets, c'est-à-dire que chacune des tâches est assignée à exactement une station de travail.

On définit ensuite les expressions intermédiaires `chosenStation` (ligne 8). Pour toute tâche t, `chosenStation[t]` représente la station $S(t)$ qui a été choisie pour la tâche t. Ces expressions ne sont pas des variables de décision ; leur valeur est calculée par l'opérateur `find`[2], qui retourne le numéro de la station contenant t. A partir de ces expressions, on peut écrire les contraintes de respect des relations de précédence entre les tâches (lignes 11 à 13). Pour deux tâches t et t_{succ} telles que $t \preceq t_{succ}$, on impose de vérifier $S(t) \preceq S(t_{succ})$: `chosenStation[t] <= chosenStation[succ]`.

Pour chaque station de travail S, on calcule ensuite `stationTime[s]`, correspondant à la durée totale $D(S)$ passée dans la station, grâce à une lambda-fonction (ligne 17). On contraint ensuite cette durée à être inférieure ou égale au temps de cycle c (ligne 18).

Pour chaque station de travail S, on vérifie si S est effectivement utilisée : on compte le nombre de tâches qui lui sont attribuées en vérifiant la taille de la variable de set à laquelle elle correspond (ligne 22). Enfin, on minimise le nombre total de stations de travail effectivement utilisées (lignes 23 à 24).

Remarque 4.1. Les indices des stations de travail n'ont pas d'importance : seul leur ordre compte. Ainsi, on considère que deux solutions utilisant le même nombre de stations sont d'une qualité équivalente. On n'impose par exemple pas que toutes les stations de travail utilisées soient consécutives, et que toutes les stations vides soient regroupées à la fin. Au contraire, il est tout à fait possible d'avoir une station de travail vide entre deux stations utilisées.

En plus d'être très court, on remarque que ce modèle, basé sur l'utilisation de variables de décision ensemblistes, est également très compact. En effet, grâce à l'utilisation des lambda-fonctions

2. L'expression `find(collectionsArray, x)` renvoie l'indice dans le tableau `collectionsArray` de la variable ensembliste contenant l'élément `x`, ou -1 si aucune de ces collections ne contient l'élément `x`.

pour calculer le temps passé dans chaque station, et de l'opérateur `count` pour vérifier si une station est utilisée, on n'a besoin que d'un nombre linéaire d'expressions pour exprimer les contraintes de respect du temps de cycle ainsi que l'objectif.

De plus, l'utilisation de variables de décision ensemblistes confère au modèle une structure forte et exploitable par les algorithmes du solveur. En effet, les liens entre les tâches sont mis en évidence, puisque celles-ci sont représentées par les valeurs contenues à l'intérieur d'une même variable de set. De même, les liens entre les stations de travail sont explicités par la contrainte de partition entre les variables de sets qui les représentent. Les Sections 4.2 et 4.3 montrent comment les structures avantageuses offertes par cette modélisation ensembliste peuvent être exploitées efficacement au sein du solveur.

Remarque 4.2. Les algorithmes décrits dans les Sections 4.2 et 4.3 n'étant pas dédiés uniquement au problème de l'Assembly Line Balancing, mais pouvant au contraire être appliqués à tous types de problèmes présentant une structure de packing (et une structure de variables ensemblistes ordonnées dans le cas des algorithmes présentés dans la Section 4.2), on utilisera, en plus du vocabulaire dédié, un vocabulaire le plus générique possible. On parlera ainsi du « poids » d_t de chaque élément t (ce qui correspond à la durée d_t d'une tâche t dans le problème de l'Assembly Line Balancing), et du poids $D(S)$ de chaque variable de set S, égal à la somme des poids des éléments qu'elle contient (durée totale $D(S)$ des tâches assignées à une station de travail S). On parlera également de la « capacité » c des variables de sets (ce qui correspond au temps de cycle c dans le problème de l'Assembly Line Balancing).

4.1.2 Benchmark utilisé

Le benchmark utilisé est celui des instances de très grande taille publié par Otto *et al.* dans [72]. Ce benchmark est constitué de cinq cent vingt-cinq instances de mille tâches, regroupées en vingt-et-une catégories de vingt-cinq instances chacune. Chaque catégorie est caractérisée par la structure du graphe de précédence des instances, la densité moyenne du graphe de précédence des instances, et la distribution des durées des tâches.

On distingue trois catégories pour la structure du graphe de précédence :

— Graphe contenant des tâches de degré au moins huit, noté « BN » pour « bottleneck ».
— Graphe contenant au moins 40% de tâches chaînées, noté « CH ».
— Graphe quelconque (noté « MIXED »).

La densité moyenne du graphe de précédence est de 0.2, 0.6, ou 0.9.

On distingue trois catégories également pour la distribution des durées des tâches :

— Pic dans les durées courtes.
— Pic dans les durées moyennes.
— Distribution bimodale.

Les caractéristiques de chacune des vingt-et-une catégories d'instances sont résumées dans la Table 4.1. Les valeurs des meilleures solutions connues, dont les moyennes sur chaque catégorie d'instances sont listées dans la Table 4.1, sont soit les solutions de référence présentées dans [72], soit les solutions obtenues dans ce Chapitre avec LocalSolver quand elles améliorent la solution de référence, ce qui est le cas sur 308 instances parmi les 525 présentes dans le benchmark, soit 59% des instances. Pour ces instances, la valeur de la meilleure solution connue utilisée est celle obtenue à l'issue de 10 minutes de calcul avec LocalSolver 10.5. Dans tout le Chapitre, on utilisera ces valeurs pour évaluer la qualité des solutions obtenues.

On remarque que les valeurs des meilleures solutions connues des instances d'une même catégorie sont très proches, et dépendent fortement de la distribution des durées des tâches dans cette catégorie (entre 130 et 140 stations de travail pour une distribution des durées avec un pic dans les durées faibles, entre 220 et 230 stations de travail pour une distribution des durées bimodale, et entre 500 et 550 stations de travail pour une distribution des durées avec un pic dans les durées

moyennes). Pour mieux mettre en évidence l'impact de ce paramètre sur les performances des algorithmes décrits dans les Sections 4.2 et 4.3 (par exemple, les deux mouvements de recherche locale faisant l'objet de la Section 4.3 sont particulièrement efficaces sur les instances très combinatoires, pour lesquelles la meilleure solution connue utilise plus de 500 stations de travail), on rappellera la valeur moyenne de la meilleure solution connue sur chaque catégorie d'instances dans chacun des tableaux de résultats présentés dans ce Chapitre.

Instances	Structure du graphe de précédence	Densité moyenne des précédences	Distribution des durées	Meilleure solution connue moyenne
1 – 25	BN	0.2	pic durées faibles	137
26 – 50	BN	0.2	pic durées moyennes	523
51 – 75	BN	0.2	bimodale	229
76 – 100	BN	0.6	pic durées faibles	138
101 – 125	BN	0.6	pic durées moyennes	539
126 – 150	BN	0.6	bimodale	227
151 – 175	CH	0.2	pic durées faibles	139
176 – 200	CH	0.2	pic durées moyennes	532
201 – 225	CH	0.2	bimodale	231
226 – 250	CH	0.6	pic durées faibles	139
251 – 275	CH	0.6	pic durées moyennes	556
276 – 300	CH	0.6	bimodale	226
301 – 325	MIXED	0.2	pic durées faibles	138
326 – 350	MIXED	0.2	pic durées moyennes	522
351 – 375	MIXED	0.2	bimodale	229
376 – 400	MIXED	0.6	pic durées faibles	137
401 – 425	MIXED	0.6	pic durées moyennes	547
426 – 450	MIXED	0.6	bimodale	224
451 – 475	MIXED	0.9	pic durées faibles	138
476 – 500	MIXED	0.9	pic durées moyennes	570
501 – 525	MIXED	0.9	bimodale	229

TABLE 4.1 – Récapitulatif des caractéristiques des instances

Performances de LocalSolver avant les travaux de la thèse. Dans les Tables 4.2 et 4.3, on montre les performances de LocalSolver avant les travaux effectués pendant cette thèse. Les résultats présentés ici, ainsi que tous les résultats numériques donnés dans la suite du Chapitre, ont été calculés sur une machine équipée d'un processeur Intel i5 6500 à 3.20 GHz et 16 Go de mémoire RAM. On donne ainsi les résultats obtenus par LocalSolver 10.5 en désactivant chacun des algorithmes décrits plus loin dans ce Chapitre, avec une solution initiale naïve ajoutée dans le modèle, en deux minutes de calcul. On compare ces résultats à la meilleure solution connue. Dans la Table 4.2, on présente, pour chaque catégorie d'instances, les moyennes sur les vingt-cinq instances de la catégorie de la valeur de la meilleure solution connue, de la valeur de la solution obtenue avec LocalSolver avant les travaux de la thèse, et de l'écart relatif entre ces deux solutions. La première ligne de la Table 4.3 se lit de la façon suivante. LocalSolver obtient un résultat strictement plus élevé, donc moins bon, que la meilleure solution connue sur 523 instances, soit 99.6% des instances. Sur ces instances, les écarts moyen et maximal à la meilleure solution connue sont respectivement de 3.0% (ou 9.6 stations de travail) et 9.3% (ou 55 stations de travail).

On constate que les résultats obtenus par LocalSolver sont plutôt bons : l'écart moyen à la meilleure solution connue sur l'ensemble des instances est de 3.0%. Cet écart est inférieur à 2.5% sur la moitié des instances, et inférieur à 1% sur 99 instances, soit 19% des instances du benchmark (par exemple, il est de 0.5% en moyenne sur les instances 26 à 50). On voit cependant que LocalSolver n'atteint que très rarement la meilleure solution connue, et que les résultats sont significativement moins bons sur d'autres catégories d'instances (l'écart à la meilleure solution connue est d'environ

Instances	Moyenne best known	Moyenne LS	Gap LS / Best
1 – 25	136	137	0.8%
26 – 50	507	509	0.5%
51 – 75	228	230	0.9%
76 – 100	137	140	2.5%
101 – 125	514	522	1.4%
126 – 150	226	231	1.9%
151 – 175	138	142	2.8%
176 – 200	508	520	2.2%
201 – 225	230	236	2.3%
226 – 250	138	143	3.5%
251 – 275	527	559	5.8%
276 – 300	225	235	4.5%
301 – 325	138	139	0.7%
326 – 350	505	512	1.3%
351 – 375	228	230	1.1%
376 – 400	136	140	2.9%
401 – 425	521	533	2.3%
426 – 450	223	231	3.7%
451 – 475	136	146	6.8%
476 – 500	546	590	7.5%
501 – 525	226	243	6.8%
Toutes	–	–	3.0%

TABLE 4.2 – Performances de LocalSolver avant les travaux de la thèse

Instances	Nb instances		Gap moyen		Gap max	
LS > Best	523	99.6%	9.6	3.0%	55	9.3%
LS = Best	2	0.4%				

TABLE 4.3 – Performances de LocalSolver avant les travaux de la thèse

7% pour les instances 451 à 500 par exemple). L'écart à la meilleure solution connue reste cependant raisonnable même sur ces instances difficiles : il est inférieur à 7% pour 94% des instances, et ne dépasse jamais 9.3%. Dans les Sections suivantes, on montre comment ces résultats peuvent être très significativement améliorés.

4.2 Algorithme constructif

Dans cette Section, on présente différentes variantes d'un algorithme constructif de répartition des tâches dans les différentes stations de travail. Dans l'ensemble du Chapitre, cet algorithme sera noté $\mathcal{A}_{greedy}$. Celui-ci repose sur la détection de structures utilisées dans le modèle à base de variables de sets, présenté la Section 4.1.1 (Modèle 4.1). Les deux utilisations principales de l'algorithme $\mathcal{A}_{greedy}$ au sein de LocalSolver sont les suivantes. D'une part, on montre dans la Section 4.2.1 qu'il permet de trouver très rapidement une solution initiale réalisable au problème lorsqu'il est appliqué à l'ensemble des tâches. D'autre part, on montre à la Section 4.2.2 qu'il permet d'améliorer les performances du solveur lorsqu'il est appliqué à un sous-ensemble de tâches au sein d'un mouvement de type « *destroy and repair* » dans la recherche locale. De plus, on montre dans la Section 4.2.3 que la structure de sets ordonnés sur laquelle se base l'algorithme peut être exploitée dans les autres mouvements de la recherche locale, pour mieux cibler les différents sets modifiés par ces mouvements.

4.2.1 Construction d'une solution initiale réalisable

Détection d'une relation d'ordre entre les variables de sets. L'algorithme $\mathcal{A}_{greedy}$ est basé sur la détection d'une relation d'ordre entre les différentes variables de collections (variables ensemblistes) du problème. Dans le cas du problème de l'Assembly Line Balancing étudié ici, on utilise des variables de sets. Cependant, l'algorithme $\mathcal{A}_{greedy}$ n'est pas dédié à ce problème particulier, et peut être appliqué à tout autre problème présentant une relation d'ordre entre des variables ensemblistes, qu'il s'agisse de sets ou de listes. Ainsi, afin de déterminer s'il est utile d'appliquer l'algorithme pour initialiser les variables de sets ou listes du modèle reçu par le solveur, on cherche à détecter des structures permettant de déterminer si ces variables sont ordonnées.

Dans le problème de l'Assembly Line Balancing, cette relation d'ordre est visible au travers des contraintes de précédence entre les tâches. Dans le Modèle 4.1, présentant la modélisation ensembliste du problème, on récupère la valeur de `chosenStation[t]`, correspondant à la station de travail dans laquelle on a placé chaque tâche t (ligne 8). Or, `chosenStation[t]` est une expression entière. On a donc associé une valeur entière à chaque variable de set, correspondant à un identifiant entier pour la station de travail associée. Ce sont ces identifiants entiers qui définissent la relation d'ordre entre les variables de sets. En effet, ils sont ensuite utilisés dans le modèle pour écrire les contraintes de respect des relations de précédence entre les tâches (lignes 12 à 14). Pour deux tâches t et t_{succ} telles que $t \preceq t_{succ}$, on doit avoir $S(t) \preceq S(t_{succ})$: l'identifiant `chosenStation[t]` de la station où a été placée la tâche t doit être inférieur ou égal à l'identifiant `chosenStation[succ]` de la station où a été placée la tâche t_{succ}.

De façon plus formelle, on suppose que le modèle comporte m variables de collections de domaine n (soit m variables de sets, ou m variables de listes) regroupées en une famille. On appelle « famille » un ensemble de variables de collections de même type (sets ou listes), de même domaine, et liées par une contrainte globale. Dans le cas du problème de l'Assembly Line Balancing, il s'agit d'une partition ; dans le cas général, la famille peut aussi être un « disjoint » ou un « cover » [3]. Ces variables collections sont notées V_0 à V_{m-1}. Chacune de ces variables est implicitement associée à un identifiant entier unique compris entre 0 et $m-1$. Ces identifiants définissent ainsi un ordre sur les variables. Pour tout élément x entre 0 et $n-1$, on peut ainsi récupérer l'identifiant de la variable de set ou liste contenant x. On cherche alors à détecter la présence de contraintes d'inégalité entre deux expressions entières servant d'identifiants aux variables de sets contenant deux éléments particuliers. Les expressions entières que l'on reconnaît comme étant de tels identifiants ont une écriture simple, reposant sur l'utilisation de l'opérateur dédié `find`. L'opérateur `find` prend en arguments la famille de variables de collections, et un élément x quelconque entre 0 et $n-1$, et renvoie l'indice i correspondant à la variable de collection V_i contenant l'élément x dans la famille. Les variables de sets ou listes sont alors simplement ordonnées par leurs indices dans la famille.

En plus de la relation d'ordre entre les différentes variables de collections, l'algorithme $\mathcal{A}_{greedy}$ est également basé sur la détection de structures de « packing » sur ces collections : les éléments ont des poids, et les collections ont des capacités maximales à respecter. La détection de telles structures étant exploitée depuis les premières versions de LocalSolver, et n'ayant pas de lien avec cette thèse, elle ne sera pas décrite ici.

Dans la suite de la Section 4.2, on supposera pour simplifier que les variables de collections du modèle sont des sets, regroupés en une partition, comme c'est le cas pour le problème de l'Assembly Line Balancing. L'algorithme $\mathcal{A}_{greedy}$ s'applique cependant aussi lorsque les variables de collections sont des listes et non des sets, et lorsque celles-ci sont regroupées en un « disjoint » ou un « cover » et non une partition. On élargit ainsi le spectre des problèmes sur lesquels on pourra appliquer l'algorithme $\mathcal{A}_{greedy}$. En effet, l'application de cet algorithme est pertinente dans tout modèle

3. Imposer une contrainte de type « disjoint » (resp. « cover ») sur une famille de collections de domaine n signifie que chaque élément de $\{0, ..., n-1\}$ doit être présent au plus une fois (resp. au moins une fois) sur l'ensemble des collections de la famille.

comportant des variables de collections ordonnées, et non seulement pour le problème particulier qu'est l'Assembly Line Balancing.

Déroulement de l'algorithme $\mathcal{A}_{greedy}$ pour la construction d'une solution initiale. On note $\mathcal{S} = \{S_0, S_1, ..., S_{n-1}\}$ l'ensemble des variables de sets (stations de travail) disponibles, rangées par ordre croissant, c'est-à-dire $\forall 0 \leq k \leq n-2,\ S_k \preceq S_{k+1}$. Initialement, chaque set est vide : $\forall k,\ S_k = \emptyset$. L'algorithme $\mathcal{A}_{greedy}$ consiste classiquement à remplir les variables de sets par rangs croissants, en insérant un par un les éléments (tâches) n'ayant plus de prédécesseurs. Pour cela, on introduit deux types d'ensembles. Pour chaque élément t, on note P_t l'ensemble des prédécesseurs non assignés de t. Au début de l'algorithme, aucun élément n'est assigné. Pour tout élément t, $P_t = \{t' \mid t' \preceq t\}$ correspond donc à l'ensemble des prédécesseurs de t. On définit ensuite l'ensemble E des éléments éligibles à l'insertion. Un élément est considéré comme éligible à l'insertion s'il n'a aucun prédécesseur non encore assigné : $E = \{t \mid P_t = \emptyset\}$. Au début de l'algorithme, l'ensemble E des éléments éligibles correspond donc à l'ensemble des éléments n'ayant aucun prédécesseur. Pour toute valeur d, on définit également $E(d) = \{t \in E \mid d_t \leq d\}$, l'ensemble des éléments éligibles de poids inférieur ou égal à d (soit l'ensemble de tâches éligibles de durée inférieure ou égale à d dans le cas de l'Assembly Line Balancing).

Le déroulement de l'algorithme $\mathcal{A}_{greedy}$ est le suivant. On note k l'indice de la variable de set courante, dans laquelle on cherche à insérer de nouveaux éléments, et on pose $k = 0$. Tant que l'ensemble E des éléments éligibles n'est pas vide, on applique la procédure suivante.

On considère $E' = E(c - D(S_k)) = \{t \in E \mid D(S_k) + d_t \leq c\}$ l'ensemble des éléments éligibles de poids suffisamment faible pour être insérés dans la variable de set courante S_k en respectant la capacité (temps de cycle) c. Si E' est vide, alors aucun élément ne peut être inséré dans S_k. On incrémente donc la valeur de k, pour passer à la variable de set suivante, actuellement vide. Sinon, on choisit un élément t dans E' en utilisant une fonction de priorité (aléatoire en pratique), et on insère t dans la variable de set courante S_k. Le poids total $D(S_k)$ de S_k se voit donc augmenté d'une quantité d_t. L'élément t étant désormais affecté, on le retire de l'ensemble E des éléments éligibles. On met ensuite à jour les successeurs de t. Pour chaque élément t_{succ} vérifiant $t \preceq t_{succ}$, on retire t de l'ensemble $P_{t_{succ}}$ des prédécesseurs non assignés de t_{succ}. Si $P_{t_{succ}}$ devient vide, alors l'élément t_{succ} devient éligible : on l'ajoute à l'ensemble E.

Le déroulement de l'algorithme $\mathcal{A}_{greedy}$ est décrit dans l'Algorithme 4.1. Pour en faciliter la lecture, on utilise ici les termes correspondant à l'application de l'algorithme au cas de l'Assembly Line Balancing (les variables de sets correspondent aux stations de travail, et les éléments qu'elles contiennent correspondent aux tâches).

Algorithme 4.1 Algorithme constructif $\mathcal{A}_{greedy}$ pour le problème de l'Assembly Line Balancing

Entrée : $\mathcal{S} = \{S_0, S_1, ..., S_{n-1}\}$ ensemble des stations disponibles vides : $\forall k,\ S_k = \emptyset$ et $S_k \preceq S_{k+1}$
Sortie : $\mathcal{S}$ forme une partition

1: $\forall t,\ P_t \leftarrow \{t' \mid t' \preceq t\}$ *prédécesseurs non assignés (initialement, tous les prédécesseurs)*
2: $E \leftarrow \{t \mid P_t = \emptyset\}$ *tâches éligibles : sans prédécesseur non assigné*
3: $k \leftarrow 0$ *indice de la station de travail courante*
4: **tant que** $E \neq \emptyset$ **faire**
5: $E' \leftarrow \{t \in E \mid D(S_k) + d_t \leq c\}$ *tâches éligibles de durée compatible*
6: **si** $E' = \emptyset$ **alors**
7: $k \leftarrow k + 1$ *aucune tâche ne peut être ajoutée à S_k : passage à la station suivante*
8: **sinon**
9: $t \leftarrow random(E')$
10: $S_k \leftarrow S_k \cup \{t\}$
11: $E \leftarrow E \setminus \{t\}$
12: **pour** t_{succ} tel que $t \preceq t_{succ}$ **faire**
13: $P_{t_{succ}} \leftarrow P_{t_{succ}} \setminus \{t\}$ *mise à jour des successeurs de t*

14: **si** $P_{t_{succ}} = \emptyset$ **alors**
15: $E \leftarrow E \cup \{t_{succ}\}$ *t_{succ} devient éligible*
16: **fin si**
17: **fin pour**
18: **fin si**
19: **fin tant que**

Proposition 4.1. *L'algorithme $\mathcal{A}_{greedy}$ donne toujours une solution réalisable s'il en existe une. De plus, il existe un ordre topologique sur les tâches, compatible avec les relations de précédence, tel que l'algorithme $\mathcal{A}_{greedy}$ conduit à la solution optimale si cet ordre est utilisé en tant que fonction de priorité.*

Preuve. On suppose que l'instance a été construite correctement, c'est-à-dire que le réseau des relations de précédence entre les tâches ne comprend aucun cycle, et que chaque tâche a une durée inférieure ou égale au temps de cycle. Sous ces conditions, on sait que le problème admet au moins une solution faisable. En effet, une solution réalisable triviale au problème consiste à ranger les tâches selon un ordre topologique (ce qui est possible si le réseau des relations de précédence ne comporte aucun cycle), et à placer la i-ème tâche selon cet ordre dans la station de travail en position i (ce qui est possible si la durée de chaque tâche est inférieure ou égale au temps de cycle). On cherche à prouver que, sous ces conditions, à la fin du déroulement de l'algorithme $\mathcal{A}_{greedy}$ tel que décrit dans l'Algorithme 4.1, la répartition des tâches dans les différentes stations de travail forme bien une solution réalisable.

Par construction, l'ajout d'une tâche à une station ne peut se faire qu'en respectant le temps de cycle et les relations de précédence avec les tâches antérieures. Il reste à montrer que chaque tâche a bien été placée dans une station à la fin de l'algorithme. On suppose que certaines tâches n'ont pas été affectées. On définit un ordre topologique sur les tâches tenant compte des relations de précédence, et on note t la première tâche non placée selon cet ordre. La tâche t n'a pas été placée, donc elle n'a jamais été ajoutée à l'ensemble des tâches éligibles E. Il existe donc au moins un prédécesseur de t non assigné. Or, tous les prédécesseurs de t sont placés avant t dans l'ordre topologique que l'on a défini, et ont donc été assignés. A la fin du déroulement de l'algorithme $\mathcal{A}_{greedy}$ tel que décrit par l'Algorithme 4.1, on a donc bien une solution réalisable au problème.

Pour la seconde partie de la preuve, on considère la fonction de priorité donnée par l'ordre topologique de la répartition des tâches dans les stations de travail dans une solution optimale. De façon évidente, cette fonction de priorité permet à l'algorithme $\mathcal{A}_{greedy}$ d'obtenir cette solution optimale. □

Complexité. La complexité de l'algorithme $\mathcal{A}_{greedy}$ tel que décrit dans l'Algorithme 4.1 est $O(n^2)$. En effet, puisque les éléments sont insérés un par un dans les variables de sets, celui-ci comporte $O(n)$ étapes. A chaque étape, on parcourt l'ensemble des éléments éligibles pour déterminer lesquels sont insérables dans la variable de set courante. S'il existe des éléments insérables, on en choisit un, puis on parcourt ses successeurs pour les mettre à jour. La complexité de chaque étape est ainsi $O(n)$. Toutefois, si l'on décide de toujours insérer le plus petit élément éligible, et non un élément éligible aléatoire, on peut réduire cette complexité à $O(m + n \log n)$, où m est le nombre de contraintes de précédence, en plaçant les éléments de l'ensemble E dans un tas, trié par poids croissants. En effet, l'étape de sélection du prochain élément à insérer se fait alors en $O(\log n)$ (retrait du plus petit élément du tas) au lieu de $O(n)$. Chaque relation de précédence est parcourue exactement une fois lors des mises à jour des prédécesseurs non assignés, pour une complexité totale de $O(m)$. Enfin, chaque élément est inséré dans le tas E exactement une fois (lorsque son ensemble de prédécesseurs non assignés devient vide), avec une complexité de $O(\log n)$.

Remarque 4.3. On peut faire l'analogie entre le problème SALB-1 et le Single-Resource Resource-Constrained Project Scheduling Problem (RCPSP) : on considère une unique ressource de capacité

c, des tâches t de durées unitaires et consommant une quantité d_t de ressource, et des contraintes de précédence de date de début à date de début et sans délai. Dans ce cas, l'algorithme $\mathcal{A}_{greedy}$ correspond à l'algorithme d'ordonnancement de liste « *parallel schedule generation scheme* » (pSGS), présenté notamment dans [55]. L'algorithme pSGS garantit l'existence d'une liste conduisant à un ordonnancement optimal pour le problème du RCPSP à durées unitaires, conformément à ce que la Proposition 4.1 affirme à propos de l'algorithme $\mathcal{A}_{greedy}$ pour le problème du SALB-1, mais pas dans le cas du problème du RCPSP à durées quelconques.

Exemple 4.1 (Exécution de l'algorithme $\mathcal{A}_{greedy}$ pour la construction d'une solution initiale). On considère un petit exemple comprenant $n = 6$ tâches, de temps de cycle $c = 50$, représenté sur la Figure 4.2. Les durées des tâches ainsi que leurs relations de précédence sont données dans la Table 4.4.

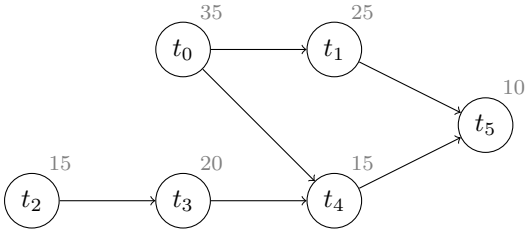


FIGURE 4.2 – Représentation graphique de l'instance

Tâche	t_0	t_1	t_2	t_3	t_4	t_5
Durée	35	25	15	20	15	10
Prédécesseurs	$\emptyset$	$\{t_0\}$	$\emptyset$	$\{t_2\}$	$\{t_0, t_3\}$	$\{t_1, t_4\}$

TABLE 4.4 – Caractéristiques de l'instance

— On commence par initialiser les différents ensembles : pour toute tâche t, P_t correspond à l'ensemble des prédécesseurs de t ; chaque station S est vide ; E correspond à l'ensemble des tâches sans aucun prédécesseur. Le contenu des différents ensembles est donné par la Table 4.5.

P_{t_0}	P_{t_1}	P_{t_2}	P_{t_3}	P_{t_4}	P_{t_5}	E
$\emptyset$	$\{t_0\}$	$\emptyset$	$\{t_2\}$	$\{t_0, t_3\}$	$\{t_1, t_4\}$	$\{t_0, t_2\}$

	S_0	S_1	S_2	S_3	S_4	S_5
S	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$D(S)$	0	0	0	0	0	0

TABLE 4.5 – Application de l'algorithme $\mathcal{A}_{greedy}$ – Initialisation

— On commence par remplir la station S_0. L'ensemble des tâches éligibles et de durée compatible est $E' = E(c - D(S_0)) = E(50) = \{t_0, t_2\}$. On choisit une tâche au hasard dans E' : par exemple t_2. On place alors t_2 dans S_0, et on actualise la valeur des différents ensembles (voir Table 4.6).

P_{t_0}	P_{t_1}	P_{t_2}	P_{t_3}	P_{t_4}	P_{t_5}	E
$\emptyset$	$\{t_0\}$	$\emptyset$	$\boldsymbol{\emptyset}$	$\{t_0, t_3\}$	$\{t_1, t_4\}$	$\{t_0, \boldsymbol{t_3}\}$

	S_0	S_1	S_2	S_3	S_4	S_5
S	$\{\boldsymbol{t_2}\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$D(S)$	**15**	0	0	0	0	0

TABLE 4.6 – Application de l'algorithme $\mathcal{A}_{greedy}$ – Insertion de t_2 dans S_0

— On continue de remplir la station S_0. L'ensemble des tâches éligibles de durée compatible est désormais $E' = E(c - D(S_0)) = E(35) = \{t_0, t_3\}$. On choisit une tâche au hasard dans E' : par exemple t_3. On place alors t_3 dans S_0, et on actualise la valeur des différents ensembles (voir Table 4.7).

P_{t_0}	P_{t_1}	P_{t_2}	P_{t_3}	P_{t_4}	P_{t_5}	E
$\emptyset$	$\{t_0\}$	$\emptyset$	$\emptyset$	$\{t_0\}$	$\{t_1, t_4\}$	$\{t_0\}$

	S_0	S_1	S_2	S_3	S_4	S_5
S	$\{t_2, \boldsymbol{t_3}\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$D(S)$	**35**	0	0	0	0	0

TABLE 4.7 – Application de l'algorithme $\mathcal{A}_{greedy}$ – Insertion de t_3 dans S_0

— L'ensemble des tâches éligibles de durée compatible est désormais $E' = E(15) = \emptyset$. On ne peut plus ajouter de nouvelle tâche dans S_0.
On passe alors au remplissage de la station S_1 : on a $E' = E(50) = \{t_0\}$. On place donc la tâche t_0 dans S_1, et on actualise la valeur des différents ensembles (voir Table 4.8).

P_{t_0}	P_{t_1}	P_{t_2}	P_{t_3}	P_{t_4}	P_{t_5}	E
$\emptyset$	$\boldsymbol{\emptyset}$	$\emptyset$	$\emptyset$	$\boldsymbol{\emptyset}$	$\{t_1, t_4\}$	$\{\boldsymbol{t_1}, \boldsymbol{t_4}\}$

	S_0	S_1	S_2	S_3	S_4	S_5
S	$\{t_2, t_3\}$	$\{\boldsymbol{t_0}\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$D(S)$	35	**35**	0	0	0	0

TABLE 4.8 – Application de l'algorithme $\mathcal{A}_{greedy}$ – Insertion de t_0 dans S_1

— On continue le remplissage de la station S_1. L'ensemble des tâches éligibles de durée compatible est $E' = E(15) = \{t_4\}$. On place donc la tâche t_4 dans S_1, et on actualise la valeur des différents ensembles (voir Table 4.9).

P_{t_0}	P_{t_1}	P_{t_2}	P_{t_3}	P_{t_4}	P_{t_5}	E
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\{t_1\}$	$\{t_1\}$

	S_0	S_1	S_2	S_3	S_4	S_5
S	$\{t_2, t_3\}$	$\{t_0, \boldsymbol{t_4}\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$D(S)$	35	**50**	0	0	0	0

TABLE 4.9 – Application de l'algorithme $\mathcal{A}_{greedy}$ – Insertion de t_4 dans S_1

— L'ensemble des tâches éligibles de durée compatible est désormais $E' = E(0) = \emptyset$. On ne peut plus ajouter de nouvelle tâche dans S_1.
On passe donc au remplissage de la station S_2 : on a $E' = E(50) = \{t_1\}$. On place donc la tâche t_1 dans S_2, et on actualise la valeur des différents ensembles (voir Table 4.10).

P_{t_0}	P_{t_1}	P_{t_2}	P_{t_3}	P_{t_4}	P_{t_5}	E
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\boldsymbol{\emptyset}$	$\{\boldsymbol{t_5}\}$

	S_0	S_1	S_2	S_3	S_4	S_5
S	$\{t_2, t_3\}$	$\{t_0, t_4\}$	$\{\boldsymbol{t_1}\}$	$\emptyset$	$\emptyset$	$\emptyset$
$D(S)$	35	50	**25**	0	0	0

TABLE 4.10 – Application de l'algorithme $\mathcal{A}_{greedy}$ – Insertion de t_1 dans S_2

— On continue le remplissage de S_2. On a désormais $E' = E(25) = \{t_5\}$. On place donc la tâche t_5 dans S_2, et on actualise la valeur des différents ensembles (voir Table 4.11).

P_{t_0}	P_{t_1}	P_{t_2}	P_{t_3}	P_{t_4}	P_{t_5}	E
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\boldsymbol{\emptyset}$

	S_0	S_1	S_2	S_3	S_4	S_5
S	$\{t_2, t_3\}$	$\{t_0, t_4\}$	$\{t_1, \boldsymbol{t_5}\}$	$\emptyset$	$\emptyset$	$\emptyset$
$D(S)$	35	50	**35**	0	0	0

TABLE 4.11 – Application de l'algorithme $\mathcal{A}_{greedy}$ – Insertion de t_5 dans S_2

— L'ensemble E des tâches éligibles est maintenant vide : l'algorithme s'arrête. On a bien réparti l'ensemble des tâches dans les différentes stations de travail. La solution ainsi obtenue, représentée sur la Figure 4.3, utilise trois stations.

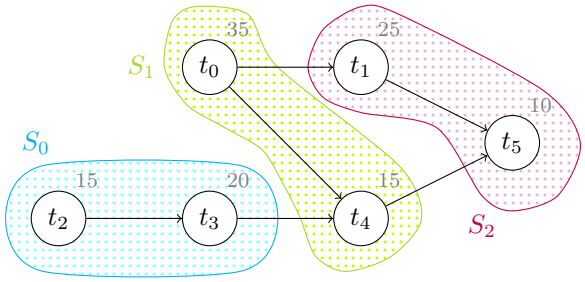


FIGURE 4.3 – Solution obtenue à l'issue de l'algorithme $\mathcal{A}_{greedy}$

Résultats numériques. L'intégration de l'algorithme $\mathcal{A}_{greedy}$ en tant qu'algorithme d'initialisation des variables de sets au sein de LocalSolver permet de trouver une solution réalisable au problème de l'Assembly Line Balancing immédiatement, quelle que soit la taille de l'instance. Ce résultat est particulièrement appréciable sur les très grandes instances du problème, pour lesquelles LocalSolver pouvait mettre plusieurs dizaines de secondes à converger vers une solution réalisable en partant d'un remplissage aléatoire des variables de sets.

De plus, l'utilisation de l'algorithme $\mathcal{A}_{greedy}$ en phase d'initialisation permet de commencer la recherche avec une solution non seulement réalisable mais également de bonne qualité. En effet, on constate un écart à la meilleure solution connue très faible sur beaucoup d'instances : il est de 7.0% en moyenne, mais est inférieur à 5% sur 326 instances, et inférieur à 2.5% sur 113 instances (soit 62% et 22% des instances respectivement). Les instances sur lesquelles cet écart est supérieur à 5% correspondent majoritairement aux instances pour lesquelles la meilleure solution connue utilise plus de la moitié des stations de travail disponibles, c'est-à-dire lorsque les stations non vides contiennent en moyenne un peu moins de deux tâches. Même sur ces instances, l'écart à la meilleure solution connue ne dépasse pas 16% : la solution obtenue à l'issue de l'algorithme $\mathcal{A}_{greedy}$ constitue une bonne base pour démarrer la recherche locale. Les résultats obtenus sont regroupés dans la Table 4.12 : pour chaque catégorie d'instances, on présente la valeur moyenne

de la meilleure solution connue sur les instances de la catégorie, ainsi que l'écart relatif moyen de la solution construite par l'algorithme $\mathcal{A}_{greedy}$ par rapport à la meilleure solution connue.

Instances	Moyenne best known	Gap moyen $\mathcal{A}_{greedy}$
1 – 25	136	2.2%
26 – 50	507	14.7%
51 – 75	228	3.8%
76 – 100	137	2.8%
101 – 125	514	15.7%
126 – 150	226	4.0%
151 – 175	138	2.5%
176 – 200	508	14.7%
201 – 225	230	3.4%
226 – 250	138	2.1%
251 – 275	527	13.5%
276 – 300	225	3.4%
301 – 325	138	2.1%
326 – 350	505	14.6%
351 – 375	228	3.8%
376 – 400	136	2.5%
401 – 425	521	14.0%
426 – 450	223	4.3%
451 – 475	136	3.9%
476 – 500	546	12.8%
501 – 525	226	5.6%
Toutes	–	7.0%

TABLE 4.12 – Gap moyen à l'issue de $\mathcal{A}_{greedy}$ sur chaque catégorie d'instances

4.2.2 Mouvement « *destroy and repair* »

Adaptation de l'algorithme $\mathcal{A}_{greedy}$ pour la recherche locale. L'algorithme $\mathcal{A}_{greedy}$ peut être adapté afin d'être utilisé en tant que mouvement de la recherche locale de LocalSolver. En effet, on peut choisir de l'appliquer pour remplir un sous-ensemble de variables de sets plutôt que la totalité de ces variables. Dans le cas du problème de l'Assembly Line Balancing par exemple, on peut choisir de l'appliquer non pas pour répartir la totalité des tâches dans les différentes stations, mais pour réorganiser les tâches déjà placées dans un sous-ensemble de stations seulement. On ré-optimise ainsi la solution courante, en détruisant puis en reconstruisant une partie de la solution. Le déroulement du mouvement de recherche locale basé sur l'algorithme $\mathcal{A}_{greedy}$ est le suivant.

On commence par sélectionner un ensemble de ℓ variables de sets consécutives, avec $2 \leq \ell \leq 10$. On note cet ensemble $\mathcal{S} = \{S_0, S_1, ..., S_{\ell-1}\}$. Le mouvement consiste alors à vider les variables de sets de l'ensemble $\mathcal{S}$ de leurs éléments, puis à réaffecter ces éléments dans les différents sets de $\mathcal{S}$ selon l'algorithme $\mathcal{A}_{greedy}$. Pour plus de diversification, on pourra choisir d'appliquer l'algorithme de « gauche à droite », c'est-à-dire en remplissant les variables de sets par ordre croissant, comme pour la construction de la solution initiale, ou de « droite à gauche », c'est-à-dire en remplissant les variables de sets par ordre décroissant.

Le déroulement de l'algorithme est le même que celui décrit à la Section 4.2.1, avec seulement quelques différences. On ne considère pas l'ensemble des éléments de $\{0, ..., n-1\}$ (tâches), mais seulement le sous-ensemble $\mathcal{T}^{\mathcal{S}} = \bigcup_{k=0}^{\ell-1} S_k$ des éléments initialement contenus dans les différents sets de $\mathcal{S}$. Pour chaque élément t de $\mathcal{T}^{\mathcal{S}}$, on s'intéresse alors à l'ensemble $P_t^{\mathcal{S}}$ des prédécesseurs (resp. successeurs) de t dans $\mathcal{T}^{\mathcal{S}}$: $P_t^{\mathcal{S}} = \{t' \in \mathcal{T}^{\mathcal{S}} \mid t' \preceq t\}$ si l'on applique l'algorithme de gauche à droite (resp. $P_t^{\mathcal{S}} = \{t' \in \mathcal{T}^{\mathcal{S}} \mid t \preceq t'\}$ si l'on applique l'algorithme de droite à gauche). De même,

l'ensemble des éléments éligibles à l'insertion devient $E = \{t \mid P_t^{\mathcal{S}} = \emptyset\}$.

Les variables de sets n'étant ici pas initialement vides, on commence par les vider. On note k l'indice de la station courante, dans laquelle on cherche à réinsérer les tâches, et on pose $k = 0$ (resp. $k = \ell - 1$ dans la version de droite à gauche). Tant que l'ensemble E des tâches éligibles à l'insertion n'est pas vide, et que l'on a $0 \leq k < \ell$, on applique la procédure décrite dans l'Algorithme 4.1 à la Section 4.2.1.

Une différence majeure entre les deux versions de l'algorithme $\mathcal{A}_{greedy}$ (initialisation ou mouvement de la recherche locale) se situe à l'issue de l'algorithme. En phase d'initialisation, on garantit que l'ensemble des variables de sets forme une partition après application de l'algorithme. En phase de recherche locale, ce n'est cependant plus le cas. Soit $\ell' \leq \ell$ le nombre de sets non vides dans $\mathcal{S}$ au début du mouvement. A la fin de l'algorithme, quatre situations différentes sont possibles.

- Si l'algorithme a rempli strictement moins de ℓ' variables de sets, alors la solution obtenue à l'issue du mouvement utilise moins de variables de sets que la solution initiale. Pour un problème dans lequel on veut minimiser le nombre de sets non vides, comme celui de l'Assembly Line Balancing, dans lequel on minimise le nombre de stations de travail utilisées, le mouvement a donc été améliorant.
- Si la solution obtenue à l'issue du mouvement utilise le même nombre de variables de sets, et a donc la même valeur que la solution initiale, alors on a réalisé un mouvement « horizontal ». Même si le mouvement n'a pas été strictement améliorant, il peut être utile pour diversifier les solutions explorées.
- Si la solution obtenue à l'issue du mouvement utilise davantage de variables de sets que la solution initiale, alors le mouvement a été dégradant. Suivant la stratégie de recherche utilisée, il peut cependant être tout de même accepté pour favoriser la diversification.
- Si l'algorithme se termine alors que tous les éléments n'ont pas pu être placés dans les variables de sets de $\mathcal{S}$, celui-ci conduit à une solution infaisable. Les changements faits au cours du mouvement sont alors annulés, et on revient à la solution initiale.

Le déroulement de l'algorithme $\mathcal{A}_{greedy}$ en tant que mouvement de la recherche locale est décrit dans sa version de droite à gauche dans l'Algorithme 4.2. Pour en faciliter la lecture, on utilise ici les termes correspondant à l'application de l'algorithme au cas de l'Assembly Line Balancing (les variables de sets correspondent aux stations de travail, et les éléments qu'elles contiennent correspondent aux tâches).

Algorithme 4.2 Adaptation de l'algorithme $\mathcal{A}_{greedy}$ à la recherche locale pour le problème de l'Assembly Line Balancing – de droite à gauche

Entrée : $\mathcal{S} = \{S_0, S_1, ..., S_{\ell-1}\}$ sous-ensemble de stations disponibles non toutes vides vérifiant $\forall k,\ S_k \preceq S_{k+1}$

Sortie : Réorganisation des tâches contenues dans les stations de $\mathcal{S}$

1: $\mathcal{T}^{\mathcal{S}} = \bigcup_{k=0}^{\ell-1} S_k$ *ensemble des tâches à réaffecter*
2: $\forall k \in \{0, ..., \ell - 1\},\ S_k \leftarrow \emptyset$
3: $\forall t \in \mathcal{T}^{\mathcal{S}},\ P_t^{\mathcal{S}} \leftarrow \{t' \in \mathcal{T}^{\mathcal{S}} \mid t \preceq t'\}$ *successeurs non assignés dans* $\mathcal{T}^{\mathcal{S}}$
4: $E \leftarrow \{t \mid P_t^{\mathcal{S}} = \emptyset\}$ *tâches éligibles : sans successeur non assigné*
5: $k \leftarrow \ell - 1$ *indice de la station de travail courante*
6: **tant que** $E \neq \emptyset$ et $0 \leq k < \ell$ **faire**
7: $E' \leftarrow \{t \in E \mid D(S_k) + d_t \leq c\}$ *tâches éligibles de durée compatible*
8: **si** $E' = \emptyset$ **alors**
9: $k \leftarrow k - 1$ *aucune tâche ne peut être ajoutée à* S_k *: passage à la station précédente*
10: **sinon**
11: $t \leftarrow random(E')$
12: $S_k \leftarrow S_k \cup \{t\}$
13: $E \leftarrow E \setminus \{t\}$
14: **pour** $t_{pred} \in \mathcal{T}^{\mathcal{S}}$ tel que $t_{pred} \preceq t$ **faire**

15: $P_{t_{pred}} \leftarrow P_{t_{pred}} \setminus \{t\}$ *mise à jour des prédécesseurs de t*
16: **si** $P_{t_{pred}} = \emptyset$ **alors**
17: $E \leftarrow E \cup \{t_{pred}\}$ *t_{pred} devient éligible*
18: **fin si**
19: **fin pour**
20: **fin si**
21: **fin tant que**

Exemple 4.2 (Exécution de l'algorithme $\mathcal{A}_{greedy}$ dans un mouvement améliorant). On considère une instance à $n = 10$ tâches, de temps de cycle $c = 50$, représentée sur la Figure 4.4.

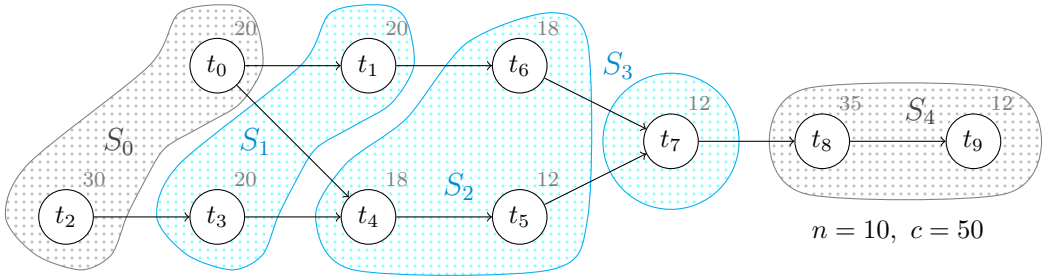


FIGURE 4.4 – Solution initiale, utilisant cinq stations de travail

On choisit par exemple d'appliquer le mouvement de droite à gauche aux stations S_1, S_2 et S_3, en bleu sur la Figure 4.4. Les tâches concernées par le mouvement sont donc $\mathcal{T}^{\mathcal{S}} = \{t_1, t_3, t_4, t_5, t_6, t_7\}$. Une fois les trois stations sélectionnées vidées de leurs tâches, le déroulement du mouvement est le suivant :

- On commence par remplir la station S_3, car on applique l'algorithme de droite à gauche. La seule tâche de $\mathcal{T}^{\mathcal{S}}$ n'ayant aucun successeur dans $\mathcal{T}^{\mathcal{S}}$ étant t_7, on a $E' = E(50) = \{t_7\}$. On place t_7 dans S_3.
- L'ensemble des tâches éligibles de durée compatible est désormais $E' = E(38) = \{t_5, t_6\}$. On choisit au hasard une tâche dans E' : par exemple t_6, et on la place dans S_3.
- On a maintenant $E' = E(20) = \{t_1, t_5\}$. On choisit une tâche au hasard dans E' : par exemple t_1, et on la place dans S_3.
- On a maintenant $E' = E(0) = \emptyset$. Ne pouvant insérer aucune autre tâche dans S_3, on passe à la station précédente, c'est-à-dire S_2.
- On a $E' = E(50) = \{t_5\}$: on place t_5 dans S_2.
- On a $E' = E(38) = \{t_4\}$: on place t_4 dans S_2.
- On a $E' = E(20) = \{t_3\}$: on place t_3 dans S_2.
- L'ensemble E des tâches éligibles est vide car toutes les tâches ont été placées : l'algorithme s'arrête. La solution obtenue à l'issue de ce mouvement, représentée sur la Figure 4.5, n'utilise plus que quatre stations au lieu de cinq. Le mouvement a bien été améliorant.

Exemple 4.3 (Exécution de l'algorithme $\mathcal{A}_{greedy}$ conduisant à une solution infaisable). On a montré dans la Section 4.2.1 que l'application de l'algorithme $\mathcal{A}_{greedy}$ à l'ensemble des tâches et des stations pour construire une solution initiale conduit toujours à une solution réalisable. Ce n'est cependant plus le cas lorsqu'on l'applique à un sous-ensemble de tâches et de stations dans le cadre d'un mouvement de la recherche locale. En effet, il est possible que de « mauvais » choix aléatoires conduisent à devoir utiliser plus de stations que le nombre disponible.

Par exemple, si l'on considère la solution à quatre stations représentée sur la Figure 4.5, et si l'on applique l'algorithme $\mathcal{A}_{greedy}$ de gauche à droite sur les tâches des stations S_2 et S_3, on peut obtenir le déroulement suivant.

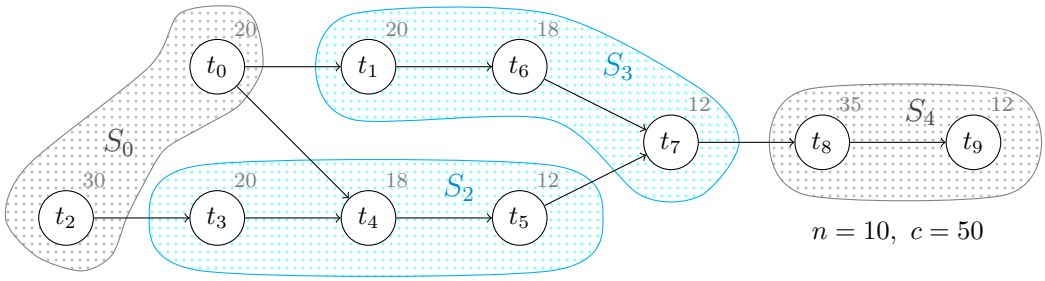


FIGURE 4.5 – Solution utilisant quatre stations de travail après application du mouvement

— On commence par remplir la station S_2. Les tâches t_1 et t_3 n'ayant aucun prédécesseur dans le sous-ensemble de tâches concerné par le mouvement, on a $E' = E(50) = \{t_1, t_3\}$. On choisit une tâche au hasard dans E' : par exemple t_1, et on la place dans S_2.
— On a maintenant $E' = E(30) = \{t_3, t_6\}$. On choisit une tâche au hasard dans E' : par exemple t_3, et on la place dans S_2.
— On a maintenant $E' = E(10) = \emptyset$: on passe à la station suivante, c'est-à-dire S_3.
— On a maintenant $E' = E(50) = \{t_4, t_6\}$. On choisit une tâche au hasard dans E' : par exemple t_6, et on la place dans S_2.
— On a maintenant $E' = E(32) = \{t_4\}$: on place t_4 dans S_2.
— On a maintenant $E' = E(14) = \{t_5\}$: on place t_5 dans S_2.
— On a maintenant $E' = E(2) = \emptyset$. On doit donc passer à la station de travail suivante, mais il n'y a plus aucune autre station disponible. La tâche t_7 n'ayant pas pu être placée, le mouvement échoue.

Résultats numériques. L'intégration de l'algorithme $\mathcal{A}_{greedy}$ en tant que mouvement de type « *destroy and repair* » au sein de la recherche locale de LocalSolver permet d'améliorer significativement ses performances sur le problème de l'Assembly Line Balancing.

La Table 4.13 présente la comparaison des résultats obtenus après deux minutes de calcul par différentes versions de LocalSolver sur les différentes catégories d'instances de mille tâches du benchmark utilisé [72]. La notation « LS_0 » désigne LocalSolver 10.5, utilisant l'algorithme $\mathcal{A}_{greedy}$ pour construire une solution initiale réalisable, mais pas au sein de la recherche locale. La notation « $\text{LS}_{d\&r}$ » (pour « *destroy and repair* ») désigne LocalSolver 10.5, utilisant l'algorithme $\mathcal{A}_{greedy}$ à la fois pour la construction d'une solution initiale réalisable et dans la recherche locale en tant que mouvement de type « *destroy and repair* ». On montre l'écart relatif moyen de la valeur des solutions obtenues par LS_0 et $\text{LS}_{d\&r}$ par rapport à la meilleure solution connue sur chaque catégorie d'instances, ainsi que la valeur moyenne de la meilleure solution connue sur les instances de chaque catégorie.

La Table 4.14 présente le nombre d'instances sur lesquelles $\text{LS}_{d\&r}$ obtient des résultats de meilleure qualité, de moins bonne qualité, ou égaux à ceux obtenus avec LS_0, ou à la meilleure solution connue. On présente également les écarts moyens et maximaux sur ces groupes d'instances.

On voit que l'ajout de l'algorithme $\mathcal{A}_{greedy}$ au sein de la recherche locale de LocalSolver permet d'améliorer largement ses performances : on obtient des résultats strictement meilleurs sur 83% des instances. L'écart à la meilleure solution connue n'est plus que de 1.0% en moyenne, et ne dépasse pas 3.4%. Ces améliorations sont particulièrement visibles sur les instances avec une forte densité de précédences : sur les instances 451 à 525, dont la densité du graphe de précédence est de 0.9, les résultats sont améliorés de 5 à 6 points de pourcentage en moyenne. Il reste cependant une marge de progression non négligeable. LocalSolver n'atteint encore que rarement la meilleure solution connue (18% des instances du benchmark), même si l'écart moyen est plutôt faible (1.3% d'écart en moyenne sur les instances où LocalSolver ne parvient pas à atteindre la meilleure solution connue). De plus, on remarque quelques dégradations de performances par rapport à la version

Instances	Best known	LS_0 / Best	$LS_{d\&r}$ / Best
1 – 25	136	0.8%	0.2%
26 – 50	507	0.5%	0.8%
51 – 75	228	0.9%	0.0%
76 – 100	137	2.5%	0.4%
101 – 125	514	1.4%	1.8%
126 – 150	226	1.9%	0.2%
151 – 175	138	2.8%	0.7%
176 – 200	508	2.2%	2.1%
201 – 225	230	2.3%	0.0%
226 – 250	138	3.5%	1.1%
251 – 275	527	5.8%	2.4%
276 – 300	225	4.5%	0.7%
301 – 325	138	0.7%	0.0%
326 – 350	505	1.3%	0.9%
351 – 375	228	1.1%	0.2%
376 – 400	136	2.9%	0.7%
401 – 425	521	2.3%	2.0%
426 – 450	223	3.7%	0.9%
451 – 475	136	6.8%	1.8%
476 – 500	546	7.5%	1.5%
501 – 525	226	6.8%	0.9%
Toutes	–	3.0%	1.0%

TABLE 4.13 – Evaluation des performances de LocalSolver avec le mouvement de recherche locale basé sur l'algorithme $\mathcal{A}_{greedy}$

Instances	Nb instances		Gap moyen		Gap max	
$LS_{d\&r} < LS_0$	436	83%	7.0	2.6%	44	8.0%
$LS_{d\&r} > LS_0$	61	12%	2.2	0.4%	6	1.1%
$LS_{d\&r} = LS_0$	28	5%				
$LS_{d\&r} >$ Best	431	82%	5.0	1.3%	18	3.4%
$LS_{d\&r} =$ Best	94	18%				

TABLE 4.14 – Evaluation des performances de LocalSolver avec le mouvement de recherche locale basé sur l'algorithme $\mathcal{A}_{greedy}$

de LocalSolver n'ayant pas le mouvement de recherche locale basé sur l'algorithme $\mathcal{A}_{greedy}$. Ces dégradations se trouvent principalement sur les instances pour lesquelles la meilleure solution connue utilise plus de la moitié des stations de travail disponibles, chacune ne contenant donc que très peu de tâches (instances 101 à 125 par exemple).

4.2.3 Ciblage des sets « proches » dans les autres mouvements de la recherche locale

Si de nombreux mouvements sur les variables de sets de la recherche locale de LocalSolver tiennent compte de la structure de packing du modèle, le mouvement basé sur l'algorithme constructif $\mathcal{A}_{greedy}$ décrit à la Section 4.2.2 est le seul à tenir également compte de la relation d'ordre entre les différentes variables de sets. Pour cette raison, les mouvements réalisant des échanges entre les éléments de différents sets échouent souvent sur le problème de l'Assembly Line Balancing, et plus généralement sur les problèmes présentant une relation d'ordre entre des variables de sets. En effet, lorsque ceux-ci déplacent un élément d'un set vers un autre d'indice potentiellement très éloi-

gné, la probabilité qu'une ou plusieurs contraintes de précédence soient violées est très forte. Dans cette Section, on montre comment la structure de variables de sets ordonnées, détectée en amont de l'application de l'algorithme $\mathcal{A}_{greedy}$, peut également être exploitée pour aider les différents mouvements de la recherche locale à mieux choisir les sets qu'ils modifient.

Comme montré dans la Section 4.2.1, la détection des relations de précédence entre les tâches permet de révéler un ordre sur les variables de sets, représentant les stations de travail dans le problème de l'Assembly Line Balancing. Grâce à cet ordre, on peut facilement déterminer si deux stations de travail, ou deux variables de sets dans le cas général, sont « proches » ou non. Ainsi, plus deux sets sont « proches », et plus le déplacement d'un élément de l'un vers l'autre a de chances de respecter les relations de précédence du problème. Il est donc intéressant de faire en sorte que les mouvements effectuant des déplacements d'éléments choisissent des variables de sets proches les unes des autres. Ainsi, lorsqu'une relation d'ordre est détectée entre les différentes variables de sets du modèle, on ajoute un ciblage automatique sur les sets proches à l'ensemble des mouvements portant sur les variables de sets. A chaque fois que l'on réalise un de ces mouvements, on a alors une certaine probabilité de tenter de choisir en priorité des sets proches les uns des autres. Certains mouvements ciblent déjà des sets présentant des caractéristiques particulières : par exemple des sets vides ou non vides, des sets peu remplis ou au contraire très remplis. Dans ce cas, on combine les deux caractéristiques dans le ciblage : on cherche des sets proches vides, ou des sets proches non vides, etc.

Pour déterminer si deux sets donnés sont proches, on a besoin d'un critère simple et rapide à évaluer. Puisque le ciblage des sets proches dans les mouvements de la recherche locale a pour but d'éviter de briser les relations de précédence entre les éléments, on s'intéresse au remplissage des sets. En effet, plus il y a d'éléments dans chaque set, et plus on a de chances de créer un conflit de précédence en déplaçant un élément de k sets vers la gauche ou vers la droite. Le nombre moyen d'éléments par set non vide étant directement lié au nombre de sets vides ou non vides, on définit la distance maximale δ_{max} jusqu'à laquelle deux sets sont considérés comme proches comme proportionnelle au nombre de sets non vides au sein de la famille.

On remarque également que la nécessité de cibler des sets proches dans les mouvements de la recherche locale est d'autant plus forte que le nombre moyen d'éléments par set non vide est élevé. En effet, si l'on déplace un élément vers un set distant de plus de δ_{max} du set de départ, alors on a davantage de chances de briser une relation de précédence lorsque les sets contiennent un nombre important d'éléments. La probabilité de tenter de choisir en priorité des sets proches dans les mouvements est donc choisie pour être inversement proportionnelle à la limite δ_{max} du nombre de sets proches de part et d'autre d'un set donné.

Résultats numériques Le ciblage des sets proches dans l'ensemble des mouvements de sets de la recherche locale de LocalSolver permet encore d'améliorer les performances du solveur sur le problème de l'Assembly Line Balancing.

Les Tables 4.15 et 4.16 présentent la comparaison des résultats obtenus par différentes versions de LocalSolver sur chacune des catégories d'instances de mille tâches du benchmark utilisé [72]. On compare ainsi quatre versions de LocalSolver, différant par l'activation ou non du mouvement de recherche locale basé sur l'algorithme $\mathcal{A}_{greedy}$, décrit à la Section 4.2.2, et du ciblage des sets proches. Chaque version utilise l'algorithme $\mathcal{A}_{greedy}$ pour la construction d'une solution initiale réalisable. La notation LS_0 désigne LocalSolver 10.5, sans le mouvement de recherche locale basé sur l'algorithme $\mathcal{A}_{greedy}$, et sans ciblage des sets proches. Pour les trois autres versions, l'indice « *d&r* » indique que le mouvement de recherche locale basé sur l'algorithme $\mathcal{A}_{greedy}$ est activé, et l'exposant « *target* » indique que le ciblage des sets proches est activé.

Dans la Table 4.15, on montre d'une part l'efficacité du ciblage seul en évaluant l'écart à la meilleure solution connue des valeurs obtenues par LS_0 et LS^{target} (voir les deux colonnes centrales), et d'autre part l'efficacité du ciblage combiné à l'utilisation de l'algorithme $\mathcal{A}_{greedy}$ dans la recherche locale en évaluant l'écart à la meilleure solution connue des valeurs obtenues par

$LS_{d\&r}$ et $LS_{d\&r}^{target}$ (voir les deux colonnes de droite). On donne également la valeur moyenne de la meilleure solution connue sur les instances de chaque catégorie.

Instances	Best known	LS_0 / Best	LS^{target} / Best	$LS_{d\&r}$ / Best	$LS_{d\&r}^{target}$ / Best
1 – 25	136	0.8%	0.2%	0.2%	0.2%
26 – 50	507	0.5%	0.3%	0.8%	0.5%
51 – 75	228	0.9%	0.2%	0.0%	0.0%
76 – 100	137	2.5%	0.4%	0.4%	0.7%
101 – 125	514	1.4%	1.1%	1.8%	1.5%
126 – 150	226	1.9%	0.2%	0.2%	0.2%
151 – 175	138	2.8%	0.7%	0.7%	0.7%
176 – 200	508	2.2%	1.6%	2.1%	1.7%
201 – 225	230	2.3%	0.9%	0.0%	0.6%
226 – 250	138	3.5%	2.1%	1.1%	0.7%
251 – 275	527	5.8%	2.8%	2.4%	2.4%
276 – 300	225	4.5%	2.0%	0.7%	0.7%
301 – 325	138	0.7%	0.0%	0.0%	0.4%
326 – 350	505	1.3%	0.5%	0.9%	0.5%
351 – 375	228	1.1%	0.2%	0.2%	0.4%
376 – 400	136	2.9%	1.1%	0.7%	0.7%
401 – 425	521	2.3%	1.4%	2.0%	1.4%
426 – 450	223	3.7%	2.4%	0.9%	0.7%
451 – 475	136	6.8%	3.5%	1.8%	1.4%
476 – 500	546	7.5%	3.3%	1.5%	1.6%
501 – 525	226	6.8%	4.2%	0.9%	0.9%
Toutes	–	3.0%	1.4%	1.0%	0.9%

TABLE 4.15 – Gains de performances apportés par l'ajout du ciblage des sets proches dans les mouvements de la recherche locale

La Table 4.16 présente le nombre d'instances sur lesquelles $LS_{d\&r}^{target}$ obtient des résultats de meilleure qualité, de moins bonne qualité, ou égaux à ceux obtenus par LS_0, ou à la meilleure solution connue. On présente également les écarts moyens et maximaux sur ces groupes d'instances.

Instances	Nb instances		Gap moyen		Gap max	
$LS_{d\&r}^{target} < LS_0$	481	92%	6.9	2.4%	45	8.2%
$LS_{d\&r}^{target} > LS_0$	19	4%	1.6	0.3%	4	0.8%
$LS_{d\&r}^{target} = LS_0$	25	5%				
$LS_{d\&r}^{target} >$ Best	442	84%	4.0	1.1%	16	2.9%
$LS_{d\&r}^{target} =$ Best	83	16%				

TABLE 4.16 – Gains de performances apportés par l'ajout du ciblage des sets proches dans les mouvements de la recherche locale

On voit que l'ajout du ciblage des sets proches dans les mouvements de la recherche locale permet d'améliorer encore davantage les performances de LocalSolver sur le problème de l'Assembly Line Balancing. En effet, on obtient des résultats strictement meilleurs avec $LS_{d\&r}^{target}$ qu'avec LS_0 sur 92% des instances du benchmark. L'écart moyen à la meilleure solution connue, égal à 0.9%, est également plus faible que sans ce ciblage. L'effet du ciblage seul est encore plus facilement remarquable : on voit que l'ajout du ciblage des sets proches dans LS^{target} lui permet d'obtenir des résultats moyens strictement meilleurs que LS_0 sur chaque catégorie d'instances. L'écart est

particulièrement important sur les instances 451 à 525, pour lesquelles la densité du graphe de précédence est la plus forte (amélioration de 4.2 points de pourcentage sur les instances 476 à 500 par exemple). On constate donc que le ciblage seul, ou le mouvement de recherche locale basé sur l'algorithme $\mathcal{A}_{greedy}$ seul, permettent tous deux d'améliorer les performances du solveur, et que l'utilisation combinée du mouvement et du ciblage conduit à obtenir des résultats encore meilleurs.

4.3 Mouvement de packing à base de chaînes d'éjection

Dans la Section 4.2, on a montré comment il était possible d'améliorer les performances de LocalSolver sur le problème de l'Assembly Line Balancing en exploitant en particulier la relation d'ordre entre les différentes stations de travail. Dans cette Section, on cherche à tirer profit de la structure de packing révélée par le modèle : chaque tâche t a une durée d_t, et la somme des durées des tâches affectées à chaque station de travail ne peut dépasser le temps de cycle c. Cette structure de packing n'est pas propre au problème de l'Assembly Line Balancing. Au contraire, on la retrouve dans de nombreux problèmes d'optimisation, comme bien sûr le problème du Bin Packing, ou par exemple les problèmes de tournées de véhicules avec capacités.

On s'intéresse ici en particulier aux instances très combinatoires pour lesquelles les variables de sets ne contiennent que très peu d'éléments, même dans une bonne solution. En effet, dans notre benchmark d'instances du problème de l'Assembly Line Balancing, les instances les plus difficiles, sur lesquelles les résultats obtenus sont les plus éloignés des meilleures solutions connues, correspondent aux instances les plus combinatoires (écart moyen de 1.5%, contre 0.6% pour les autres instances). De même, les instances très combinatoires du problème du Bin Packing sont réputées difficiles. On cherche donc à améliorer les performances de LocalSolver en particulier sur les instances très combinatoires des problèmes présentant une structure de packing.

4.3.1 Revue de littérature

Dans [26], les auteurs proposent une recherche locale itérative pour le problème du Bin Packing avec conflits, basée sur différents voisinages locaux ou larges. Le but de l'article se rapproche ainsi de celui de cette Section : résoudre un problème de packing, avec éventuellement des contraintes supplémentaires (éléments incompatibles à placer dans des bins différents dans le cas du problème du Bin Packing avec conflits, ou relations de précédence entre les tâches pour le problème de l'Assembly Line Balancing), en utilisant une recherche locale.

Les auteurs définissent ainsi plusieurs mouvements de recherche locale à voisinages larges. On s'intéresse en particulier au mouvement « chaînes d'éjection », dont on s'inspire dans cette Section pour créer un nouveau mouvement à intégrer à la recherche locale de LocalSolver. Le principe des chaînes d'éjection [46] est une technique classiquement utilisée pour définir des mouvements de recherche locale complexes, construits en composant plusieurs mouvements simples. Plus formellement, les procédures de chaînes d'éjection consistent en des séquences de mouvements élémentaires, où un changement effectué à l'étape k sur l'état, la position, l'affectation ou la valeur des éléments sélectionnés entraîne, à l'étape $k+1$, l'« éjection » d'autres éléments de leur état, leur position, leur affectation, ou leur valeur actuelle. Cette technique est particulièrement utile lorsque l'on cherche à résoudre des problèmes très contraints pour lesquels il est difficile de construire une structure de voisinage telle que toute solution réalisable admette une solution voisine réalisable également. En effet, l'utilisation de voisinages basés sur des chaînes d'éjection permet alors d'obtenir une solution réalisable à l'issue d'un enchaînement de petites transformations qui, seules, auraient conduit à des solutions non réalisables.

Le déroulement du mouvement décrit par les auteurs correspond à l'algorithme suivant. On commence par définir un ordre quelconque sur l'ensemble des bins non vides dans la solution courante : on les note alors $S_0, S_1, ..., S_{k-1}$. Cet ordre sert à réduire l'espace d'exploration du mouvement : on ne peut déplacer un élément d'un bin S_i vers un bin S_j que si $i < j$. On définit

ensuite un graphe $G = (V, A)$, illustré sur la Figure 4.6, avec $V = V_{elem} \cup V_{bin} \cup \{v_{source}\}$: V_{elem} contient un sommet v_t pour chaque élément t, et V_{bin} contient un sommet v_S pour chaque bin S non vide dans la solution courante. Les arcs de A modélisent des déplacements d'éléments :

- Pour toute paire (t, t') d'éléments tels que $S(t)$ est placé avant $S(t')$, l'arc $a_{t,t'}$ entre v_t et $v_{t'}$ modélise le fait de remplacer l'élément t' par l'élément t dans le bin $S(t')$.
- Pour tout élément t et tout bin S placé avant $S(t)$, les arcs $a_{source,t}$ et $a_{S,t}$ entre v_{source} ou v_S et v_t modélisent le fait de retirer l'élément t de son bin $S(t)$ dans la solution courante.
- Pour tout élément t et tout bin S placé après $S(t)$, l'arc $a_{t,S}$ entre v_t et v_S modélise le fait d'insérer l'élément t dans le bin S.

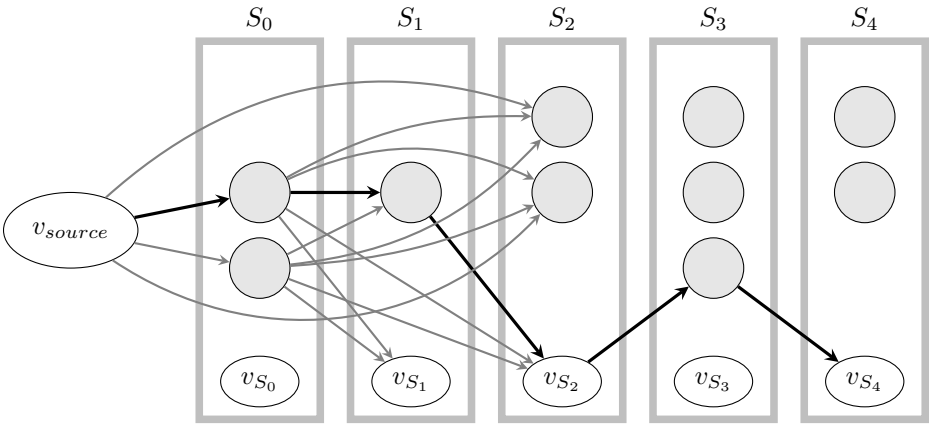


FIGURE 4.6 – Graphe du mouvement de chaîne d'éjection, et solution possible

On associe un coût à chaque arc a, correspondant au coût pour effectuer le déplacement modélisé par l'arc a. Ce coût est calculé en fonction des violations de contraintes de capacité ou d'exclusion induites ou réparées par le déplacement. On calcule un plus court chemin dans ce graphe, de v_{source} vers un sommet quelconque de V_{bin}. Ce chemin correspond à une séquence de déplacements d'éléments. Grâce à l'utilisation des sommets de V_{bin}, cette séquence peut être composée de plusieurs chaînes d'éjection disjointes.

4.3.2 Description du mouvement $\mathcal{M}_{chain}$ – version « chaîne d'éjection » et version « déplacements horizontaux »

En s'inspirant des travaux de [26], on définit un nouveau mouvement pour la recherche locale de LocalSolver, exploitant la structure de packing préalablement détectée dans le modèle. Ce mouvement est également basé sur le principe des chaînes d'éjection : il correspond à une série de déplacements d'éléments vers de nouvelles variables de sets. Cependant, contrairement au mouvement de chaînes d'éjection de l'article, il n'impose pas de réaliser les déplacements d'éléments dans un certain ordre, ne s'applique que sur un sous-ensemble de variables de sets non vides, et choisit les déplacements d'éléments à effectuer de manière gloutonne. Il sera noté $\mathcal{M}_{chain}$ dans la suite du Chapitre. Le mouvement $\mathcal{M}_{chain}$ est décliné en deux versions, notées version « chaîne d'éjection » et version « déplacements horizontaux », décrites dans cette Section.

4.3.2.1 Principe général des deux versions du mouvement

Le principe général du mouvement $\mathcal{M}_{chain}$ est le suivant. On commence par choisir un ensemble de k variables de sets non vides $\mathcal{S} = \{S_0, ..., S_{k-1}\}$, avec $10 \leq k \leq 40$. Si les variables de sets du problème sont ordonnées, comme c'est le cas dans le problème de l'Assembly Line Balancing, on peut choisir de remplir $\mathcal{S}$ avec des sets consécutifs ou quelconques. On note $\mathcal{T}^{\mathcal{S}}$ l'ensemble des éléments contenus dans les différentes variables de sets de $\mathcal{S}$. Le but est de réorganiser les éléments

de $\mathcal{T}^{\mathcal{S}}$, de sorte à ne plus utiliser que $k-1$ variables de sets parmi les k variables présentes dans $\mathcal{S}$. Pour cela, on effectue une série de petites transformations. On utilise deux types de telles transformations, donnant ainsi deux versions du mouvement.

Dans la première version du mouvement $\mathcal{M}_{chain}$, ces transformations consistent à éjecter un élément d'une variable de set, afin de pouvoir y placer un autre élément, qui lui aussi aura été préalablement éjecté de son set de départ. Cette version du mouvement sera alors notée version « chaîne d'éjection ». On remarque qu'à chacune des étapes intermédiaires du mouvement, un des éléments de $\mathcal{T}^{\mathcal{S}}$ (l'élément éjecté) n'appartient plus à aucune variable de set. Ainsi, si le problème comporte une contrainte de partition entre les différentes variables de sets, comme c'est le cas pour les problèmes de l'Assembly Line Balancing et du Bin Packing par exemple, chacune des étapes intermédiaires du mouvement donne une solution non réalisable.

Dans la deuxième version du mouvement $\mathcal{M}_{chain}$, ces transformations consistent à déplacer un élément d'une variable de set vers une autre. Le mouvement correspond ainsi à une succession de petits mouvements « horizontaux », ne modifiant pas le nombre total de variables de sets utilisées, et n'invalidant pas l'éventuelle contrainte de partition liant les variables de sets du problème. Cette version du mouvement sera alors notée version « déplacements horizontaux ».

Les versions « chaîne d'éjection » et « déplacements horizontaux » du mouvement $\mathcal{M}_{chain}$ sont décrites en détail dans la suite de cette Section.

4.3.2.2 Mouvement $\mathcal{M}_{chain}$ version « chaîne d'éjection »

Le principe de la version « chaîne d'éjection » du mouvement $\mathcal{M}_{chain}$ consiste à sélectionner un élément, que l'on éjecte de son set de départ, et à l'insérer dans une nouvelle variable de set, d'où l'on éjecte alors un autre élément, de poids plus faible et donc plus « avantageux ». On répète cette procédure jusqu'à ce que l'insertion de l'élément courant dans une nouvelle variable de set ne nécessite plus d'éjection pour respecter les contraintes de capacité.

On note $S_{cur} \in \mathcal{S}$ la variable de set courante, initialement choisie comme celle ayant le poids le plus faible. On choisit ensuite l'élément courant t_{cur} aléatoirement dans S_{cur}. Le but du mouvement est de réarranger les éléments contenus dans les variables de $\mathcal{S} \setminus \{S_{cur}\}$, afin de pouvoir y placer t_{cur}. Si le poids de l'élément courant t_{cur} est strictement supérieur à la somme des espaces encore libres dans les autres variables de sets, alors le mouvement échoue. En effet, cela revient à dire que la somme des poids des éléments de $\mathcal{S} \setminus \{S_{cur}\}$ et de t_{cur} est supérieure à la somme des capacités des variables de sets de $\mathcal{S} \setminus \{S_{cur}\}$: il est alors impossible de respecter toutes les contraintes de capacité. Dans le cas contraire, on éjecte l'élément courant t_{cur} de la variable de set courante S_{cur}. On répète alors la procédure suivante.

On considère l'ensemble des variables de sets susceptibles d'accueillir l'élément t_{cur} tout en respectant la contrainte de capacité qui leur est associée : il s'agit de l'ensemble des variables de sets $S \neq S_{cur}$ telles que $D(S) + d_{t_{cur}} \leq c$. Si cet ensemble est non vide, alors on y choisit une variable de set quelconque S_{next}. On insère t_{cur} dans S_{next} : le mouvement est un succès.

Sinon, l'élément t_{cur} n'est pas directement insérable dans une nouvelle variable de set : on cherche alors à l'insérer à la place d'un autre élément de poids plus faible (et ainsi plus facilement insérable à l'étape suivante). On considère alors l'ensemble $\mathcal{T}_{next}$ des éléments échangeables avec t_{cur}, et de poids plus faible. Les éléments de $\mathcal{T}_{next}$ doivent ainsi avoir un poids strictement inférieur à $d_{t_{cur}}$, et appartenir à une variable de set ayant suffisamment de place disponible pour accueillir t_{cur} après leur éjection : $\mathcal{T}_{next} = \{t \in \mathcal{T}^{\mathcal{S}} \mid d_t < d_{t_{cur}},\ S(t) \neq S_{cur},\ D(S(t)) - d_t + d_{t_{cur}} \leq c\}$. Si $\mathcal{T}_{next}$ est vide, alors il n'y a plus d'échange possible : le mouvement échoue. Sinon, on sélectionne t_{next} l'élément de plus petit poids dans $\mathcal{T}_{next}$, et on note $S_{next} = S(t_{next})$ la variable de set qui le contient. On insère t_{cur} dans S_{next}, et on en éjecte t_{next}. L'élément courant devient alors t_{next}, et la variable de set courante devient S_{next}.

Le déroulement de la version « chaîne d'éjection » du mouvement $\mathcal{M}_{chain}$ est décrit dans l'Algorithme 4.3. Pour en faciliter la lecture, on utilise ici les termes correspondant à l'application

du mouvement au problème du Bin Packing (les variables de sets correspondent aux bins, et les éléments qu'elles contiennent correspondent aux objets).

Algorithme 4.3 Adaptation de la version « chaîne d'éjection » du mouvement $\mathcal{M}_{chain}$ pour le problème du Bin Packing

Entrée : Ensemble $\mathcal{S} = \{S_0, ..., S_{k-1}\}$ de bins non vides.

$\mathcal{T}^{\mathcal{S}} \leftarrow \bigcup_{S \in \mathcal{S}} S$
$S_{cur} \leftarrow \arg\min_{S \in \mathcal{S}}(D(S))$ *plus petit bin*
$t_{cur} \leftarrow random(S_{cur})$
si $d_{t_{cur}} > \sum_{S \neq S_{cur}} c - D(S)$ **alors**
  renvoyer échec *pas assez de place disponible dans les autres bins pour placer* t_{cur}
fin si
$S_{cur} \leftarrow S_{cur} \setminus \{t_{cur}\}$ *on éjecte l'objet sélectionné de son bin courant*
pour $0 \leq i < i_{\max}$ **faire**
  $\mathcal{S}_{next} \leftarrow \{S \in \mathcal{S} \mid S \neq S_{cur},\ D(S) + d_{t_{cur}} \leq c\}$ *bins pouvant accueillir* t_{cur}
  si $\mathcal{S}_{next} \neq \emptyset$ **alors**
    $S_{next} \leftarrow random(\mathcal{S}_{next})$
    $S_{next} \leftarrow S_{next} \cup \{t_{cur}\}$ t_{cur} *peut être inséré dans un nouveau bin* S_{next}
    $S_{\min} \leftarrow \arg\min_{S \in \mathcal{S}}(D(S))$ *plus petit bin*
    $S'_{\min} \leftarrow \arg\min_{S \neq S_{\min} \in \mathcal{S}}(D(S))$ *deuxième plus petit bin*
    si $D(S_{\min}) + D(S'_{\min}) \leq c$ **alors**
      $S'_{\min} \leftarrow S'_{\min} \cup S_{\min}$ *fusion des deux plus petits bins*
      $S_{\min} \leftarrow \emptyset$
    fin si
    renvoyer succès
  fin si
  $\mathcal{T}_{next} \leftarrow \{t \in \mathcal{T}^{\mathcal{S}} \mid d_t < d_{t_{cur}},\ S(t) \neq S_{cur},\ D(S(t)) - d_t + d_{t_{cur}} \leq c\}$ *objets échangeables avec* t_{cur}
  si $\mathcal{T}_{next} = \emptyset$ **alors**
    renvoyer échec *plus d'échange possible*
  fin si
  $t_{next} \leftarrow \arg\min_{t \in \mathcal{T}_{next}}(d_t)$
  $S_{next} \leftarrow S(t_{next})$
  $S_{next} \leftarrow S_{next} \setminus \{t_{next}\} \cup \{t_{cur}\}$ *on éjecte* t_{next} *de* S_{next} *pour y placer* t_{cur}
  $S_{cur} \leftarrow S_{next}$ *nouveau bin courant*
  $t_{cur} \leftarrow t_{next}$ *nouvel objet courant*
fin pour

Remarque 4.4. A chaque étape de l'algorithme du mouvement $\mathcal{M}_{chain}$ version « chaîne d'éjection », on éjecte un élément t_{next} de sa variable de set courante pour y insérer l'élément courant t_{cur}. Ces deux éléments vérifiant l'inégalité $d_{t_{next}} < d_{t_{cur}}$, l'algorithme est composé d'au plus N étapes, où N est égal au nombre total d'éléments dans l'ensemble $\mathcal{S}$ des variables de sets sélectionnées au début du mouvement.

Complexité. La complexité du mouvement $\mathcal{M}_{chain}$ version « chaîne d'éjection », tel que décrit dans l'Algorithme 4.3 est alors $O(N^2)$. En effet, l'exploration de l'ensemble $\mathcal{S}_{next}$ des variables de sets pouvant accueillir l'élément courant t_{cur} se fait en $O(k)$ (avec $k \leq N$ puisque les variables de sets sélectionnées sont toutes non vides), et l'exploration de l'ensemble $\mathcal{T}_{next}$ des éléments échangeables avec t_{cur} et de poids inférieur, au sein duquel on sélectionne le plus petit élément t_{next}, se fait en $O(N)$. On peut réduire la taille de l'ensemble des éléments explorés lors de la recherche de t_{next} en triant l'ensemble $\mathcal{T}^{\mathcal{S}}$ de tous les éléments des sets sélectionnés. Cependant, l'élément t_{next} ne doit

pas seulement avoir un poids inférieur à celui de t_{cur}. Il doit également appartenir à une variable de set $S(t_{next})$ capable d'accueillir t_{cur} après éjection de t_{next}. Même si les éléments sont triés par poids croissants, l'ensemble des éléments à explorer pour trouver t_{next} a toujours une taille $O(N)$, et la complexité de l'algorithme demeure inchangée.

Remarque 4.5. Comme expliqué au paragraphe précédent, le but du mouvement $\mathcal{M}_{chain}$ est de réorganiser les éléments présents dans un ensemble de k variables de sets non vides, afin de ne plus utiliser que $k-1$ de ces variables. Pourtant, la version « chaîne d'éjection » du mouvement $\mathcal{M}_{chain}$ consiste à retirer un unique élément de la variable de set de poids le plus faible, puis à réaliser des échanges successifs avec d'autres éléments jusqu'à pouvoir insérer l'élément courant dans une autre variable de set tout en respectant sa contrainte de capacité. A la fin du mouvement, si celui-ci se termine sur un succès, une seule variable de set (la variable de set de poids le plus faible) contient un nombre d'éléments plus faible qu'au début du mouvement. De plus, ce nombre n'a diminué que de 1. Si l'objectif du problème est de minimiser le nombre de sets non vides, le mouvement ne peut donc être améliorant que si cette variable de set ne contenait initialement qu'un seul élément. Cependant, le mouvement $\mathcal{M}_{chain}$ version « chaîne d'éjection » creuse l'écart entre les poids des k variables de sets sélectionnées : il diminue encore le poids de la variable de poids le plus faible en lui retirant un élément, et augmente les poids des autres variables. Même s'il n'est pas directement améliorant, il peut ainsi permettre d'obtenir une solution plus facilement améliorable par les prochains mouvements appelés lors de la recherche locale. De plus, afin d'avoir de meilleures performances sur les instances quelconques, on ajoute une étape à la fin du mouvement : si celui-ci se termine sur un succès, on tente de fusionner les deux variables de sets de poids les plus faibles.

De plus, le mouvement $\mathcal{M}_{chain}$ a été pensé pour améliorer les performances de LocalSolver plus particulièrement sur les instances très combinatoires de Bin Packing ou d'Assembly Line Balancing, pour lesquelles les variables de sets non vides ne contiennent que très peu d'éléments, même dans une bonne solution. Sur ce type d'instances, il est effectivement probable que la variable de set de poids le plus faible parmi un groupe de k variables non vides ne contienne qu'un seul élément. Si le mouvement se termine sur un succès, il est alors bien améliorant.

Exemple 4.4 (Mouvement $\mathcal{M}_{chain}$ version « chaîne d'éjection »). On considère un petit ensemble de huit éléments, répartis dans cinq variables de sets, sur lesquels on applique la version « chaîne d'éjection » du mouvement $\mathcal{M}_{chain}$. La capacité de chaque variable de set est $c = 1000$, et le poids d_t de chaque élément t est donné dans la Table 4.17.

t	0	1	2	3	4	5	6	7
d_t	499	408	631	501	419	579	336	549

TABLE 4.17 – Caractéristiques de l'instance

On part d'une solution utilisant cinq variables de sets, représentée à gauche sur la Figure 4.7 : les variables de sets sont représentées sur l'axe des abscisses, et leurs poids sont représentés sur l'axe des ordonnées. Chaque élément t est représenté par un rectangle, étiqueté par le numéro de l'élément t et son poids d_t.

Le déroulement du mouvement est le suivant :

— La variable de set courante est celle ayant le poids le plus faible : $S_{cur} = S_4$, de poids $D(S_4) = 549$. L'élément courant t_{cur} est choisi aléatoirement dans S_{cur}, qui ne contient qu'un seul élément : on a donc $t_{cur} = t_7$, de poids $d_{t_7} = 549$. On éjecte t_7 de S_4.

— On constate qu'aucune variable de set $S \neq S_4$ ne peut accueillir t_7 : on cherche donc à insérer t_7 à la place d'un autre élément de poids plus faible. On s'intéresse donc aux éléments $t \notin S_4$ dont le poids d_t vérifie $d_t \leq d_{t_7}$ et $D(S(t)) - d_t + d_{t_7} \leq c$. Les éléments candidats sont t_0 et t_3. Le poids $d_{t_0} = 499$ de t_0 étant plus faible, on choisit $t_{next} = t_0$. On éjecte t_0 de sa

variable de set courante S_0, et on y place t_7. La variable de set courante devient $S_{cur} = S_0$, et l'élément courant devient $t_{cur} = t_0$.

— En suivant le même raisonnement, on éjecte l'élément t_4 de la variable de set S_2, pour y placer t_0.

— On éjecte t_6 de S_3 pour y placer t_4.

— L'élément courant est désormais t_6, de poids $d_{t_6} = 336$. On constate que $c - D(S_1) = 369 \geq d_{t_6}$: on insère t_6 dans S_1. Le mouvement est un succès.

Le mouvement ainsi réalisé est améliorant : à la fin de son déroulement, on n'utilise plus que quatre variables de sets sur les cinq initialement utilisées. La solution obtenue à l'issue du mouvement est représentée à droite sur la Figure 4.7.

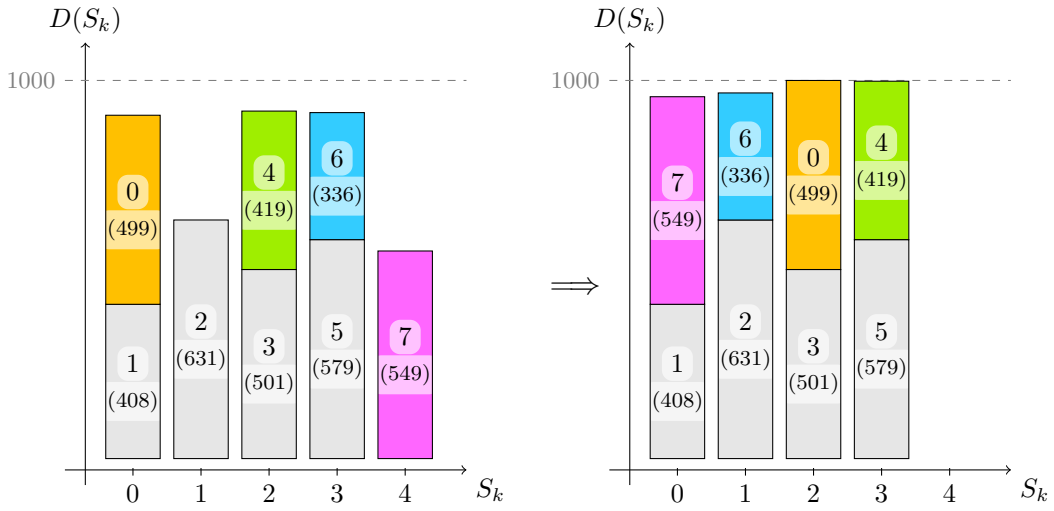


FIGURE 4.7 – Solution initiale (gauche) et solution obtenue après le mouvement $\mathcal{M}_{chain}$ version « chaîne d'éjection » (droite)

4.3.2.3 Mouvement $\mathcal{M}_{chain}$ version « déplacements horizontaux »

Le principe de la version « déplacements horizontaux » du mouvement $\mathcal{M}_{chain}$ consiste à déplacer successivement des éléments de leur set courant S vers une nouvelle variable de set S', de façon à obtenir le plus d'espace libre possible dans le set S, jusqu'à pouvoir le fusionner avec une autre variable de set tout en respectant les contraintes de capacité. Le déroulement du mouvement est le suivant.

On note S_{cur} la variable de set courante, initialement choisie pour être celle ayant le poids le plus faible, et on répète la procédure suivante. On s'intéresse aux éléments de $\mathcal{S} \setminus \{S_{cur}\}$ dont le déplacement vers S_{cur} est réalisable et créerait un plus grand espace libre. Il s'agit de l'ensemble des éléments $t \in \mathcal{T}^{\mathcal{S}} \setminus S_{cur}$ vérifiant $D(S_{cur}) + d_t \leq c$ (déplacement réalisable) et $c - D(S(t)) + d_t > c - D(S_{cur})$ (plus grand espace libre). Si aucun tel élément n'existe, alors le mouvement s'arrête (s'il ne s'agit pas de la première itération, on a réalisé un mouvement « horizontal »). Sinon, on note t_{next} l'élément dont le déplacement vers la variable de set courante S_{cur} crée le plus grand espace libre, et S_{next} la variable de set à laquelle il appartient. On déplace l'élément t_{next} de S_{next} vers S_{cur}. On considère alors les deux variables de sets les moins remplies dans $\mathcal{S}$. S'il est possible de les fusionner, alors on les fusionne : le mouvement est un succès. Sinon, la variable de set courante devient S_{next}, et on recommence.

Le déroulement de la version « déplacements horizontaux » du mouvement $\mathcal{M}_{chain}$ est décrit dans l'Algorithme 4.4. Pour en faciliter la lecture, on utilise ici les termes correspondant à l'application du mouvement au problème du Bin Packing (les variables de sets correspondent aux bins, et les éléments qu'elles contiennent correspondent aux objets).

Algorithme 4.4 Adaptation de la version « déplacements horizontaux » du mouvement $\mathcal{M}_{chain}$ pour le problème du Bin Packing

Entrée : Ensemble $\mathcal{S} = \{S_0, ..., S_{k-1}\}$ de bins non vides.

$\mathcal{T}^{\mathcal{S}} \leftarrow \bigcup_{S \in \mathcal{S}} S$
$S_{cur} \leftarrow \arg\min_{S \in \mathcal{S}}(D(S))$ *plus petit bin*
pour $0 \leq i < i_{\max}$ **faire**
  $\mathcal{T}_{next} = \{t \in \mathcal{T}^{\mathcal{S}} \mid S(t) \neq S_{cur},\ D(S_{cur}) + d_t \leq c,\ c - D(S(t)) + d_t > c - D(S_{cur})\}$
    objets dont le déplacement vers S_{cur} crée un plus grand espace libre
  si $\mathcal{T}_{next} = \emptyset$ **alors**
    break *plus de déplacement possible*
  fin si
  $t_{next} = \arg\max_{t \in \mathcal{T}_{next}}(D(S_{cur}) - D(S(t)) + d_t)$ *objet donnant le plus grand espace libre*
  $S_{next} \leftarrow S(t_{next})$
  $S_{next} \leftarrow S_{next} \setminus \{t_{next}\}$
  $S_{cur} \leftarrow S_{cur} \cup \{t_{next}\}$ *déplacement de t_{next} de S_{next} vers S_{cur}*
  $S_{\min} \leftarrow \arg\min_{S \in \mathcal{S}}(D(S))$ *plus petit bin*
  $S'_{\min} \leftarrow \arg\min_{S \neq S_{\min} \in \mathcal{S}}(D(S))$ *deuxième plus petit bin*
  si $D(S_{\min}) + D(S'_{\min}) \leq c$ **alors**
    $S_{\min} \leftarrow S_{\min} \cup S'_{\min}$ *fusion des deux plus petits bins*
    $S'_{\min} \leftarrow \emptyset$
    renvoyer succès
  fin si
  $S_{cur} \leftarrow S_{next}$ *nouveau bin courant*
fin pour

Proposition 4.2. *Le critère utilisé à chaque itération de la version « déplacements horizontaux » du mouvement $\mathcal{M}_{chain}$ pour choisir le prochain élément à déplacer, qui vise à obtenir un plus grand espace libre, implique de toujours déplacer un élément de poids strictement plus élevé qu'à l'étape précédente. L'algorithme comporte donc au plus N étapes, où N est égal au nombre total d'éléments dans l'ensemble $\mathcal{S}$ des variables de sets sélectionnées au début du mouvement. Comme pour la version « chaîne d'éjection », la complexité de la version « déplacements horizontaux » du mouvement $\mathcal{M}_{chain}$ tel que décrit par l'Algorithme 4.4 est alors $O(N^2)$.*

Preuve (Eléments de poids croissants). On considère le déroulement suivant. On suppose que l'algorithme a successivement réalisé le déplacement d'un élément t d'une variable de set S_2 vers une autre variable de set S_1 à l'itération i, puis le déplacement d'un élément t' d'une autre variable de set S_3 vers S_2 à l'itération $i+1$, et que les poids des éléments t et t' sont tels que $d_t \geq d_{t'}$. Deux configurations se distinguent alors :

- Si $S_3 \neq S_1$, alors le déplacement de t' de S_3 vers S_1 aurait été possible à l'itération i, puisque $D(S_1) + d_{t'} \leq D(S_1) + d_t \leq c$, et aurait créé un plus grand espace libre (dans S_3) que le déplacement de t (dans S_2). L'algorithme aurait donc choisi de déplacer directement t' de S_3 vers S_1 à l'itération i.
- Si $S_3 = S_1$, alors le raisonnement précédent n'est plus valable. Au début de l'itération i, l'élément t appartient au set S_2, et l'élément t' appartient au set S_1. On note D_1 et D_2 la somme des poids des autres éléments appartenant à S_1 et S_2 respectivement : au début de l'itération i, on a $D(S_1) = D_1 + d_{t'}$ et $D(S_2) = D_2 + d_t$. Le déplacement de t de S_2 vers S_1 à l'itération i crée un plus grand espace vide dans S_2 ($c - D_2 > c - (D_1 + d_{t'})$), d'où $D_2 < D_1 + d_{t'}$. Le déplacement de t' de S_1 vers S_2 à l'itération $i+1$ crée ensuite un plus grand espace vide dans S_1, d'où $D_2 > D_1 + d_t$. Puisque les poids des éléments t et t' vérifient $d_t \geq d_{t'}$, on a alors $D_2 > D_2$, ce qui est absurde.

Les poids des éléments déplacés au cours du déroulement de l'algorithme sont donc bien strictement croissants. □

Exemple 4.5 (Mouvement $\mathcal{M}_{chain}$ version « déplacements horizontaux »). On considère le même petit ensemble de huit éléments répartis dans cinq variables de sets que précédemment, sur lesquels on applique la version « déplacements horizontaux » du mouvement $\mathcal{M}_{chain}$. La capacité de chaque variable de set est $c = 1000$, et le poids d_t de chaque élément t est donné dans la Table 4.17. On part de la même solution initiale que dans l'exemple précédent, utilisant cinq variables de sets, représentée à gauche sur la Figure 4.8.

Le déroulement du mouvement est le suivant.

— La variable de set courante est celle ayant le poids le plus faible : $S_{cur} = S_4$, de poids $D(S_4) = 549$.

— On s'intéresse aux éléments dont le déplacement vers S_4 créerait un plus grand espace vide dans leur variable de set de départ, c'est-à-dire aux éléments $t \notin S_4$ dont le poids d_t vérifie $D(S_4)+d_t \leq c$ et $c-D(S(t))+d_t > c-D(S_4)$. Les éléments candidats sont t_1 et t_4. L'espace libéré par t_1 étant plus grand, on choisit $t_{next} = t_1$. On déplace donc t_1 de S_0 vers S_4. La variable de set courante devient alors S_0.

Les deux variables de sets de poids les plus faibles sont S_0 et S_4. La somme de leurs poids vaut $499 + 629 = 1028 > c = 1000$. On ne peut pas fusionner S_0 et S_1, et on passe donc à l'itération suivante.

— En suivant le même raisonnement, on déplace l'élément t_3 de S_2 vers S_0.

— On déplace t_5 de S_3 vers S_2. Les deux variables de sets de poids les plus faibles sont désormais S_3 et S_1. La somme de leurs poids vaut $336 + 631 = 967 \leq c = 1000$. On fusionne donc les variables de sets S_3 et S_1, en déplaçant l'élément t_2 de S_1 vers S_3. Le mouvement est un succès.

Le mouvement ainsi réalisé est améliorant : à la fin de son déroulement, on n'utilise plus que quatre variables de sets sur les cinq initialement utilisées. La solution obtenue à l'issue du mouvement est représentée à droite sur la Figure 4.8.

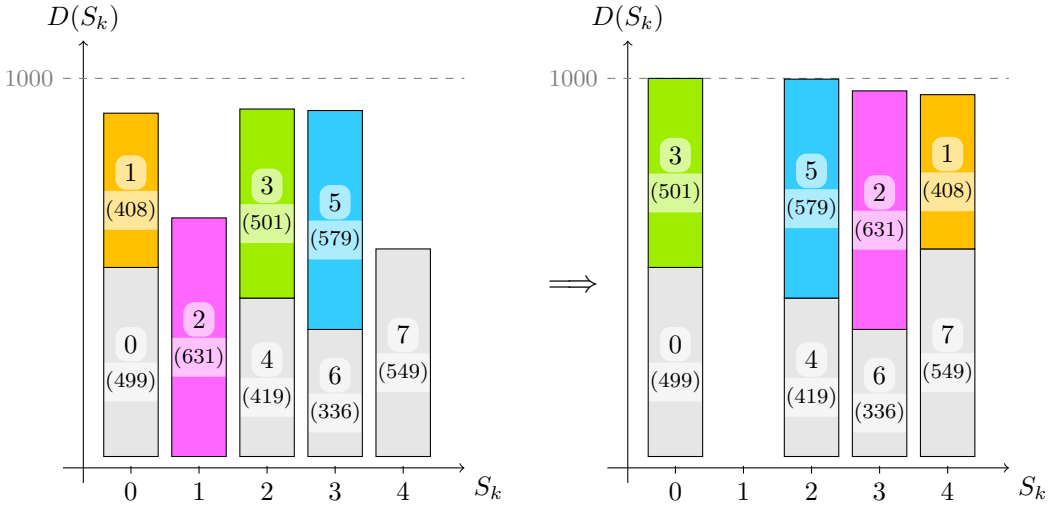


FIGURE 4.8 – Solution initiale (gauche) et solution obtenue après le mouvement $\mathcal{M}_{chain}$ version « déplacements horizontaux » (droite)

4.3.3 Propriétés sur le mouvement $\mathcal{M}_{chain}$

Dans cette Section, on décrit plusieurs propriétés notables sur le mouvement $\mathcal{M}_{chain}$. On s'intéresse ainsi d'abord à sa construction, puis à la forme des solutions auxquelles il conduit, et enfin

à ses performances.

Intérêt de passer plusieurs fois par une même variable de set. A chaque étape du mouvement $\mathcal{M}_{chain}$, on choisit un nouvel élément, noté t_{next} (élément à éjecter dans la version « chaîne d'éjection », ou élément à déplacer dans la version « déplacements horizontaux »). On peut remarquer qu'on considère pour cela les éléments présents dans n'importe quelle variable de set différente du set courant S_{cur} : on parcourt donc notamment les variables de sets ayant déjà été modifiées. Afin de réduire sa complexité, on peut envisager une version alternative du mouvement $\mathcal{M}_{chain}$, dans laquelle on ne s'autoriserait pas à modifier plusieurs fois une même variable de set : on n'aurait alors à parcourir que les sets non encore visités pour choisir le prochain élément t_{next}. Cependant, une telle modification de l'algorithme du mouvement $\mathcal{M}_{chain}$ n'est en pratique pas souhaitable. En effet, il existe des situations dans lesquelles les critères « chaîne d'éjection » et « déplacements horizontaux » du mouvement $\mathcal{M}_{chain}$ ne permettent de trouver une solution améliorante que si l'on s'autorise à ajouter ou retirer plusieurs éléments à une même variable de set.

Exemple 4.6. On considère un petit ensemble de dix éléments, répartis dans six variables de sets, sur lesquels on applique les deux versions du mouvement $\mathcal{M}_{chain}$, d'abord en imposant de ne modifier qu'une seule fois chaque variable de set, puis en levant cette restriction. La capacité de chaque variable de set est $c = 1000$, et le poids d_t de chaque élément t est donné dans la Table 4.18. La solution initiale est représentée à gauche sur les Figures 4.9 et 4.10.

t	0	1	2	3	4	5	6	7	8	9
d_t	591	316	457	486	547	510	401	614	529	431

TABLE 4.18 – Caractéristiques de l'instance

On commence par appliquer la version « chaîne d'éjection » du mouvement $\mathcal{M}_{chain}$ à la solution initiale. A chaque étape de l'algorithme, on ne s'autorise à éjecter un élément t_{next} pour le remplacer par l'élément courant t_{cur} que si $S(t_{next})$ n'a jamais été modifié. Le déroulement du mouvement est le suivant :

— La variable de set ayant le plus petit poids est $S_{cur} = S_2$: on éjecte $t_{cur} = t_4$ de S_2.
— L'élément de plus petit poids pouvant être remplacé par t_4 est t_5 : on éjecte t_5 de S_3 pour y placer t_4.
— On éjecte t_2 de S_1 pour y placer t_5.
— On éjecte t_9 de S_5 pour y placer t_2.
— Le seul élément t de poids $d_t < d_{t_9}$ et tel que $S(t)$ peut accueillir t_9 après l'éjection de t est t_6. Or, $S(t_6) = S_3$ a déjà été modifié : on en a déjà éjecté t_5 pour y insérer t_4. On n'a donc plus aucune éjection possible.

On constate que le mouvement échoue. Cependant, si l'on s'autorise à modifier à nouveau la variable de set S_3, l'algorithme continue :

— On éjecte t_6 de S_3 pour y insérer t_9.
— On éjecte t_1 de S_0 pour y insérer t_6.
— On insère t_1 dans S_4.

Le mouvement se termine sur un succès : on a une solution améliorante utilisant cinq variables de sets, représentée à droite sur la Figure 4.9.

On applique maintenant la version « déplacements horizontaux » du mouvement $\mathcal{M}_{chain}$, à partir de la même solution initiale. A chaque étape de l'algorithme, on ne s'autorise à déplacer un élément t_{next} que si $S(t_{next})$ n'a jamais été modifié. Le déroulement du mouvement est le suivant :

— La variable de plus petit poids est S_2.
— On déplace t_6 de S_3 vers S_2.

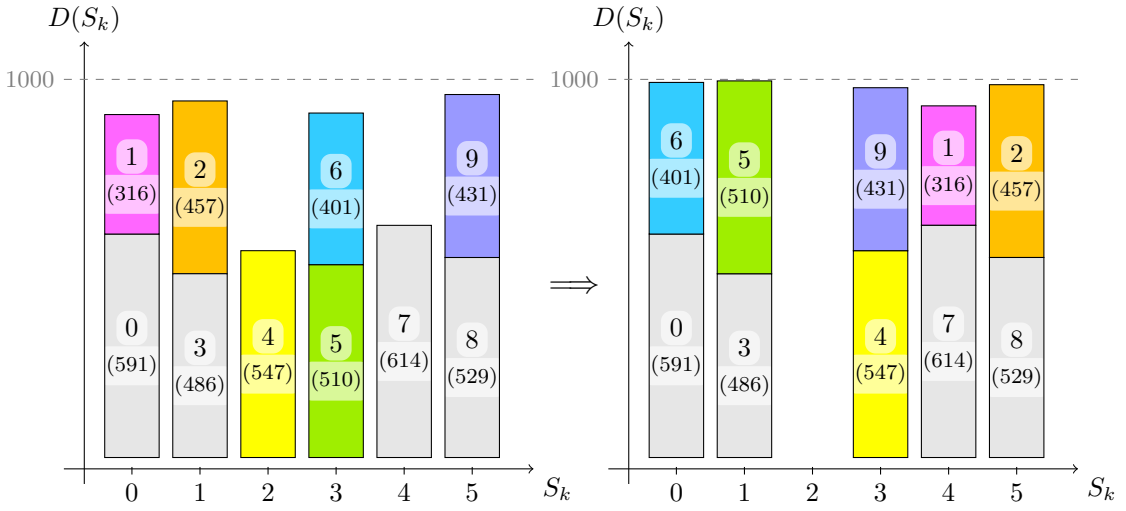


FIGURE 4.9 – Solution initiale (gauche) et solution obtenue après le mouvement $\mathcal{M}_{chain}$ version « chaîne d'éjection » (droite) : on doit modifier plusieurs fois la variable S_3

— On déplace t_3 de S_1 vers S_3.
— On déplace t_8 de S_5 vers S_1.
— Le seul élément t pouvant être inséré dans S_5 et dont le déplacement créerait un plus grand espace libre dans $S(t)$ est t_4. Or, $S(t_4) = S_2$ a déjà été modifié : on y a inséré t_6. On n'a donc plus aucun déplacement possible.

On constate que le mouvement s'arrête sans avoir amélioré la solution. Cependant, si l'on s'autorise à modifier plusieurs fois la variable S_2, l'algorithme continue :

— On déplace t_4 de S_2 vers S_5.
— On déplace t_0 de S_0 vers S_2.
— Les variables S_0 et S_4 peuvent être fusionnées : on déplace t_7 de S_4 vers S_0.

Le mouvement se termine sur un succès : on a une solution améliorante utilisant cinq variables de sets, représentée à droite sur la Figure 4.10.

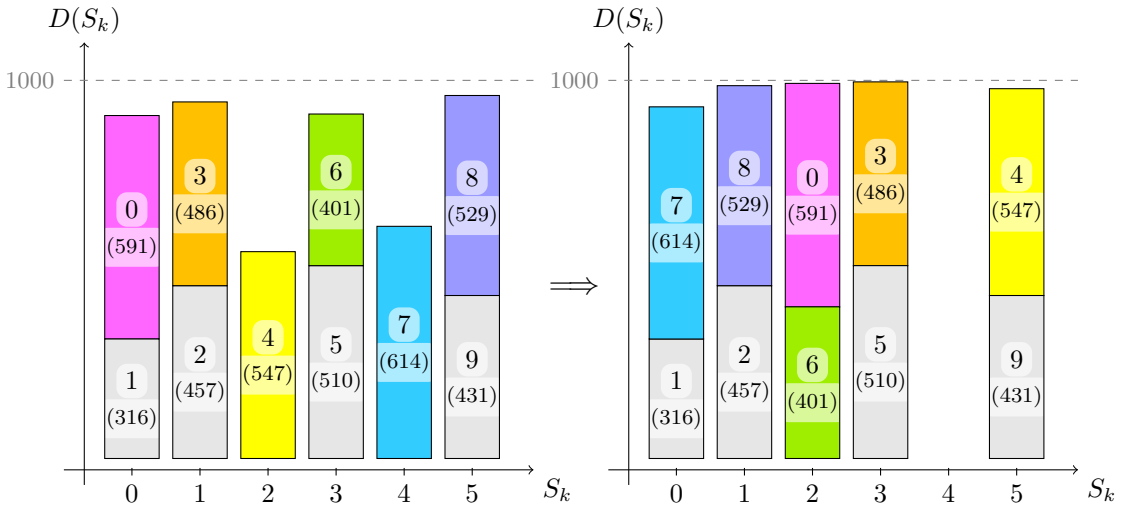


FIGURE 4.10 – Solution initiale (gauche) et solution obtenue après le mouvement $\mathcal{M}_{chain}$ version « déplacements horizontaux » (droite) : on doit modifier plusieurs fois la variable S_2

Ainsi, non seulement l'algorithme autorise de modifier plusieurs fois une même variable de

set au cours du mouvement, mais cela est même parfois nécessaire à l'obtention d'une solution améliorante.

Comparaison des solutions obtenues à l'issue des deux versions du mouvement. On s'intéresse ici aux différences et similitudes entre les solutions obtenues à l'issue des deux versions du mouvement $\mathcal{M}_{chain}$, lorsque celles-ci sont améliorantes.

Les deux versions du mouvement $\mathcal{M}_{chain}$ consistent à réorganiser progressivement le contenu des variables de sets sélectionnées en effectuant des déplacements successifs d'éléments, dans le but de s'approcher peu à peu d'une situation dans laquelle il est possible de fusionner deux variables de sets. Pour ce faire, elles utilisent deux stratégies opposées, mais répondant toutefois à la même logique globale. D'une part, la version « chaîne d'éjection » du mouvement $\mathcal{M}_{chain}$ consiste à éjecter un élément, que l'on échange avec des éléments d'autres sets, de poids de plus en plus faibles, jusqu'à pouvoir l'insérer dans une variable de set tout en respectant sa contrainte de capacité. On réalise ainsi une succession de déplacements d'éléments de poids de plus en plus faibles. D'autre part, la version « déplacements horizontaux » du mouvement $\mathcal{M}_{chain}$ consiste à déplacer des éléments de façon à obtenir un espace libre de plus en plus grand dans leur variable de set de départ, jusqu'à ce que celle-ci puisse être fusionnée avec une autre variable de set. On réalise ainsi une succession de déplacements d'éléments de poids de plus en plus élevés [4].

Proposition 4.3. *Dans le cas général, les solutions obtenues à l'issue des deux versions du mouvement (lorsque celui-ci est améliorant) sont différentes. Toutefois, dans le cas particulier où les variables de sets sélectionnées contiennent au plus deux éléments, ces solutions sont très similaires : le contenu des différentes variables de sets est le même, seul leur ordre diffère. Pour un problème dans lequel toutes les variables de sets sont interchangeables, comme celui du Bin Packing par exemple, ces solutions sont donc équivalentes.*

Schéma de preuve. Dans le cas où chaque variable de set sélectionnée contient au plus deux éléments, on peut considérer ces variables de sets comme des paires d'éléments. On appellera alors « complémentaire » d'un élément t l'élément t' appartenant à la même « paire d'éléments » que t, c'est-à-dire tel que $S(t) = \{t, t'\}$.

A chaque itération du mouvement $\mathcal{M}_{chain}$ version « chaîne d'éjection », on éjecte le plus petit élément tel que son complémentaire peut être associé à l'élément courant. D'autre part, à chaque itération de la version « déplacements horizontaux », on déplace l'élément que l'on peut associer à l'unique élément du set courant tel que son complémentaire est le plus petit. Ainsi, les deux versions du mouvement $\mathcal{M}_{chain}$ tendent à créer les mêmes paires d'éléments, soit en déplaçant les éléments de poids faibles vers les sets contenant les éléments de plus grands poids pour la version « chaîne d'éjection », soit en déplaçant les éléments de plus grands poids vers les sets contenant les éléments de poids faibles pour la version « déplacements horizontaux ». □

Bien que les solutions renvoyées par les deux versions du mouvement $\mathcal{M}_{chain}$ soient très similaires, il est tout de même intéressant d'utiliser les deux versions du mouvement $\mathcal{M}_{chain}$ au sein de la recherche locale de LocalSolver, même dans le cas particulier où toutes les variables de sets considérées ne contiennent que deux éléments. En effet, pour des problèmes présentant des variables de sets ordonnées, comme celui de l'Assembly Line Balancing, les solutions renvoyées par les deux versions du mouvement $\mathcal{M}_{chain}$ ne sont pas équivalentes, puisqu'elles diffèrent par l'ordre des sets. On peut alors observer des situations dans lesquelles la solution renvoyée par l'une des versions du mouvement est bien améliorante, tandis que celle obtenue à l'issue de l'autre version ne respecte pas toutes les relations de précédence entre les éléments (tâches). L'utilisation combinée des deux versions du mouvement permet en outre une plus grande diversification des solutions explorées par la recherche locale.

4. Voir Proposition 4.2

Exemple 4.7 (Solutions équivalentes). On considère l'instance de l'Exemple 4.6. En étudiant les solutions améliorantes obtenues à l'issue des deux versions du mouvement, représentées sur les Figures 4.9 et 4.10, on constate que celles-ci sont bien équivalentes du point de vue du contenu des variables de sets. Les deux versions du mouvement créent les mêmes paires d'éléments, dans le même ordre :

- On crée la paire d'éléments $\{t_4, t_6\}$ en déplaçant t_4 vers S_3 dans la version « chaîne d'éjection » (resp. en déplaçant t_6 vers S_2 dans la version « déplacements horizontaux »).
- On crée la paire $\{t_3, t_5\}$ en déplaçant t_5 vers S_1 (resp. en déplaçant t_3 vers S_3).
- On crée la paire $\{t_2, t_8\}$ en déplaçant t_2 vers S_5 (resp. en déplaçant t_8 vers S_1).
- On crée la paire $\{t_4, t_9\}$ en déplaçant t_9 vers S_3 (resp. en déplaçant t_4 vers S_5).
- On crée la paire $\{t_0, t_6\}$ en déplaçant t_6 vers S_0 (resp. en déplaçant t_0 vers S_2).
- On crée la paire $\{t_1, t_7\}$ en déplaçant t_1 vers S_4 (resp. en déplaçant t_7 vers S_0).

On obtient donc bien deux solutions équivalentes du point de vue du contenu des sets, mais différentes par l'ordre des sets.

Lorsque les variables de sets sélectionnées au début du mouvement peuvent contenir strictement plus de deux éléments, les solutions obtenues à l'issue des deux versions du mouvement ne sont plus équivalentes, et on remarque que la version « chaîne d'éjection » est plus souvent améliorante. En effet, la version « chaîne d'éjection » déplace des éléments de poids décroissants, tandis que la version « déplacements horizontaux » déplace des éléments de poids croissants. Or, lorsque les variables de sets contiennent un plus grand nombre d'éléments, ceux-ci ont en moyenne des poids plus faibles. Dans la version « déplacements horizontaux », on a donc plus de chances d'arriver rapidement à une itération où plus aucun élément seul n'a un poids assez élevé pour que son déplacement vers le set courant crée un plus grand espace vide dans son set de départ. Il faudrait alors déplacer des blocs d'éléments, et non des éléments seuls. La version « déplacements horizontaux » du mouvement se termine donc souvent après un plus petit nombre d'itérations, sans avoir trouvé de solution améliorante. Cependant, même dans ce cas, il produit une solution réalisable (ou respectant au moins la contrainte de partition et les contraintes de capacité), dans laquelle les écarts entre les poids des différentes variables de sets sont plus importants que dans la solution initiale. De plus, les deux versions du mouvement conduisent à des solutions différentes. Il est donc utile de conserver ces deux versions dans la recherche locale de LocalSolver.

Statistiques sur l'efficacité du mouvement. Le mouvement $\mathcal{M}_{chain}$ a été pensé pour aider la recherche locale de LocalSolver à sortir des minima locaux dans lesquels elle pourrait être bloquée. En effet, on trouve parfois des situations dans lesquelles il est nécessaire de modifier un grand nombre de variables de sets pour trouver une solution améliorante. Le mouvement $\mathcal{M}_{chain}$ a ainsi pour but de permettre au solveur de sortir de ces situations difficiles, en réorganisant totalement la répartition des éléments dans un sous-ensemble de variables de sets. Afin d'évaluer ses performances, on s'intéresse donc à sa capacité à améliorer des solutions non trivialement améliorables.

Pour cela, on génère un grand nombre de petites instances aléatoires sur lesquelles on applique le mouvement $\mathcal{M}_{chain}$. On considère ainsi des instances comportant 10 variables de sets, de capacité 1000, comportant chacune entre 1 et 4 éléments. Afin d'étudier les performances du mouvement uniquement sur les instances difficiles décrites ci-dessus, on élimine les instances pour lesquelles il n'existe aucune solution améliorante, ainsi que les instances trop simples, pouvant être améliorées en modifiant moins de 3 variables de sets. On divise les instances restantes en trois catégories, en fonction du nombre maximum d'éléments dans chaque variable de set : moins de 2 éléments par set, moins de 3 éléments par set, et moins de 4 éléments par set. La Table 4.19 donne le nombre d'instances générées, ainsi que le pourcentage d'instances améliorées par les versions « chaîne d'éjection » et « déplacements horizontaux » pour chaque catégorie.

Les instances pour lesquelles chaque variable de set contient au plus deux éléments sont les plus combinatoires. Pour rappel, ce sont ces instances qui sont visées en particulier par le mouvement

	$\|S\| \leq 2$	$\|S\| \leq 3$	$\|S\| \leq 4$
Nombre d'instances	40 000	100 000	700 000
Améliorations « chaîne d'éjection »	99.98%	31%	18%
Améliorations « déplacements horizontaux »	99.98%	28%	12%

TABLE 4.19 – Pourcentage d'améliorations trouvées par chaque version du mouvement

$\mathcal{M}_{chain}$. Les performances du mouvement sur ces instances sont excellentes : le mouvement $\mathcal{M}_{chain}$ trouve une amélioration sur près de 100% des solutions pouvant être améliorées. Comme expliqué au point précédent, on remarque que les performances des deux versions du mouvement sont équivalentes.

Remarque 4.6. Les quelques instances à deux éléments par set sur lesquelles le mouvement $\mathcal{M}_{chain}$ ne trouve pas d'amélioration ont toutes la même forme. Pour chacune d'elles, l'amélioration à trouver peut être vue comme une double ou triple chaîne d'éjection : on retire un élément de poids important d'une variable de set, dans laquelle on place ensuite deux éléments de poids faibles.

On remarque que les performances du mouvement $\mathcal{M}_{chain}$ sont moins bonnes sur les instances pour lesquelles les variables de sets peuvent contenir jusqu'à 3 ou 4 éléments. Toutefois, cela ne signifie pas que le mouvement échoue sur toutes les instances pour lesquelles il n'a pas abouti à une amélioration. En effet, sur beaucoup d'instances, on a réalisé avec succès un mouvement horizontal, qui ne change pas le nombre de variables de sets utilisées, mais qui fournit une solution plus facilement améliorable par la suite. Comme expliqué au point précédent, on remarque que la version « chaîne d'éjection » du mouvement trouve plus souvent des solutions améliorantes que la version « déplacements horizontaux ».

4.3.4 Généralisation du mouvement $\mathcal{M}_{chain}$

Si les deux versions du mouvement sont particulièrement pertinentes pour les problèmes de l'Assembly Line Balancing et du Bin Packing, celui-ci ne leur est cependant pas dédié. Au contraire, tous les algorithmes implémentés au sein de LocalSolver, et notamment les mouvements de la recherche locale, visent toujours à être les plus génériques possibles. Afin de respecter ce principe, les deux versions du mouvement $\mathcal{M}_{chain}$ sont généralisées de façon à s'adapter aux contraintes et structures supplémentaires plus complexes éventuellement présentes dans le problème.

Les algorithmes des versions « chaîne d'éjection » et « déplacements horizontaux » du mouvement $\mathcal{M}_{chain}$ décrits dans la Section 4.3.2 prennent l'hypothèse que toutes les variables de sets du problème ont une même capacité c, ce qui correspond à la définition classique des problèmes de l'Assembly Line Balancing (toutes les stations de travail doivent respecter le même temps de cycle c) et du Bin Packing (tous les conteneurs ont la même capacité c). L'implémentation des deux versions du mouvement $\mathcal{M}_{chain}$ dans LocalSolver est cependant plus générique, et prévoit que chaque variable de set S puisse avoir sa propre capacité c_S.

On rencontre parfois des problèmes dans lesquels on impose à des groupes d'éléments d'appartenir à une même variable de set. Dans ce cas, les deux versions du mouvement $\mathcal{M}_{chain}$ décrites à la Section 4.3.2 ont de fortes chances de mener à des solutions infaisables. Leur implémentation au sein de LocalSolver tient donc compte de l'éventuelle existence de groupes d'éléments. Chaque élément t appartenant à un tel groupe G est alors assimilé à un super élément équivalent à tout le groupe G : on considère que son poids n'est pas égal à d_t, mais à $d_G = \sum_{t' \in G} d_{t'}$, et tous les autres éléments de G sont automatiquement déplacés avec lui. Tous les éléments t de G sont alors équivalents : retirer ou ajouter t à une variable de set S fait diminuer ou augmenter son poids total $D(S)$ d'une quantité d_G.

Il est également possible d'avoir des appartenances imposées (un élément t contraint à appartenir à une variable de set S donnée) ou interdites (un élément t ne pouvant pas appartenir à une

variable de set S). Afin de ne pas violer les contraintes d'appartenances imposées, on s'assure que l'élément retiré de sa variable de set courante à chaque itération du mouvement (élément éjecté dans la version « chaîne d'éjection », ou élément déplacé dans la version « déplacements horizontaux ») ne correspond pas à une appartenance imposée. De même, on vérifie à chaque itération qu'aucun déplacement d'un élément vers une nouvelle variable de set ne correspond à une appartenance interdite. De plus, si un élément t fait partie d'un groupe d'éléments G, on ne peut le déplacer au cours du mouvement que si chaque élément de G peut bien être retiré de sa variable de set courante $S(t)$ et ajouté à sa nouvelle variable de set. Du fait des appartenances imposées et interdites, on teste également la possibilité de fusion des deux variables de sets de poids les plus faibles $S_{\min}$ et $S'_{\min}$ (à chaque itération dans la version « déplacements horizontaux », ou à la fin du mouvement en cas de succès dans la version « chaîne d'éjection ») dans les deux sens : on vérifie s'il est possible de déplacer tous les éléments de $S_{\min}$ vers $S'_{\min}$, ou inversement.

4.3.5 Résultats numériques

Problème de l'Assembly Line Balancing. L'ajout des versions « chaîne d'éjection » et « déplacements horizontaux » du mouvement $\mathcal{M}_{chain}$ au sein de la recherche locale de LocalSolver permet au solveur d'obtenir de très bonnes performances sur le problème de l'Assembly Line Balancing.

La Table 4.20 présente la comparaison des résultats obtenus par quatre versions de LocalSolver, différant par l'activation ou non de chacune des versions du mouvement $\mathcal{M}_{chain}$, sur les différentes catégories d'instances de mille tâches du benchmark utilisé [72]. La notation $\text{LS}_{d\&r}^{target}$ correspond à LocalSolver 10.5, en activant tous les algorithmes décrits à la Section 4.2 (algorithme $\mathcal{A}_{greedy}$ en tant qu'initialisation et mouvement de la recherche locale, ciblage des sets « proches »), et en désactivant les deux versions du mouvement $\mathcal{M}_{chain}$. La colonne correspondante dans le tableau reprend les résultats de la Table 4.15 (Section 4.2.3). Les notations LS_{eject} et LS_{horiz} correspondent respectivement à l'activation des versions « chaîne d'éjection » et « déplacements horizontaux » du mouvement $\mathcal{M}_{chain}$, en plus des algorithmes décrits à la Section 4.2. La notation $\text{LS}^{\star}$ correspond à l'activation de tous les algorithmes décrits dans ce Chapitre (notamment les deux versions du mouvement $\mathcal{M}_{chain}$). On montre l'efficacité des deux versions du mouvement $\mathcal{M}_{chain}$, prises séparément et de façon combinée, en évaluant l'écart à la meilleure solution connue des valeurs obtenues par $\text{LS}_{d\&r}^{target}$, LS_{eject}, LS_{horiz} et $\text{LS}^{\star}$. On donne également la valeur moyenne de la meilleure solution connue sur les instances de chaque catégorie, afin de mettre en évidence l'efficacité particulière du mouvement $\mathcal{M}_{chain}$ sur les instances très combinatoires, pour lesquelles la meilleure solution connue utilise plus de cinq cents stations de travail.

La Table 4.21 présente le nombre d'instances sur lesquelles LS_{eject}, LS_{horiz} et $\text{LS}^{\star}$ obtiennent des résultats de moins bonne qualité ou égaux à la meilleure solution connue. On présente également les écarts moyens et maximaux sur ces groupes d'instances.

On voit que l'ajout du mouvement $\mathcal{M}_{chain}$ permet d'améliorer encore davantage les performances de LocalSolver sur le problème de l'Assembly Line Balancing. L'ajout de chacune des deux versions (« chaîne d'éjection » et « déplacements horizontaux ») séparément apporte déjà un gain de performance. En effet, on constate que les résultats de LS_{eject} et LS_{horiz} sont meilleurs que ceux de $\text{LS}_{d\&r}^{target}$ sur presque tous les groupes d'instances, et que l'écart moyen à la meilleure solution connue passe de 0.9% à moins de 0.6%. On constate que les deux versions du mouvement $\mathcal{M}_{chain}$ donnent des résultats similaires : avec les versions « chaîne d'éjection » et « déplacements horizontaux », on atteint la meilleure solution connue sur 39% et 27% des instances respectivement, avec un écart moyen de 0.6% et 0.5%. L'utilisation combinée des deux versions du mouvement permet d'atteindre des performances encore meilleures : la meilleure solution connue est atteinte sur 50% des instances, avec un écart moyen de 0.5% sur les 50% d'instances restantes, et de 0.3% seulement sur l'ensemble des instances. Le gain de performance est particulièrement visible sur les instances pour lesquelles la meilleure solution connue comporte plus de 500 stations de travail utilisées. Ces

Instances	Best known	$LS_{d\&r}^{target}$ / Best	LS_{eject} / Best	LS_{horiz} / Best	$LS^{\star}$ / Best
1 – 25	136	0.2%	0.2%	0.2%	0.2%
26 – 50	507	0.5%	0.4%	0.2%	0.1%
51 – 75	228	0.0%	0.0%	0.0%	0.0%
76 – 100	137	0.7%	0.4%	0.4%	0.4%
101 – 125	514	1.5%	1.2%	0.2%	0.1%
126 – 150	226	0.2%	0.0%	0.2%	0.0%
151 – 175	138	0.7%	0.4%	0.7%	0.4%
176 – 200	508	1.7%	1.6%	0.4%	0.4%
201 – 225	230	0.6%	0.0%	0.4%	0.0%
226 – 250	138	0.7%	0.7%	0.7%	0.4%
251 – 275	527	2.4%	1.7%	0.7%	0.8%
276 – 300	225	0.7%	0.2%	0.4%	0.4%
301 – 325	138	0.4%	0.0%	0.0%	0.0%
326 – 350	505	0.5%	0.7%	0.2%	0.2%
351 – 375	228	0.4%	0.0%	0.2%	0.0%
376 – 400	136	0.7%	0.7%	0.7%	0.7%
401 – 425	521	1.4%	0.9%	0.2%	0.0%
426 – 450	223	0.7%	0.2%	0.4%	0.4%
451 – 475	136	1.4%	1.4%	1.8%	1.4%
476 – 500	546	1.6%	1.3%	0.3%	0.5%
501 – 525	226	0.9%	0.9%	1.3%	0.4%
Toutes	–	0.9%	0.6%	0.5%	0.3%

TABLE 4.20 – Gains de performances apportés par l'ajout des deux versions du mouvement $\mathcal{M}_{chain}$ sur le problème de l'Assembly Line Balancing

Instances	Nb instances		Gap moyen		Gap max	
LS_{eject} > Best	322	61%	4.2	1.0%	14	2.6%
LS_{eject} = Best	203	39%				
LS_{horiz} > Best	382	73%	1.6	0.6%	4	2.2%
LS_{horiz} = Best	143	27%				
$LS^{\star}$ > Best	264	50%	1.6	0.5%	5	1.5%
$LS^{\star}$ = Best	261	50%				

TABLE 4.21 – Gains de performances apportés par l'ajout des deux versions du mouvement $\mathcal{M}_{chain}$ sur le problème de l'Assembly Line Balancing

instances, pour lesquelles les variables de sets non vides contiennent moins de deux éléments en moyenne, même dans une bonne solution, sont en effet plus particulièrement la cible visée par le mouvement $\mathcal{M}_{chain}$. Alors que l'écart à la meilleure solution connue était le plus grand avec $LS_{d\&r}^{target}$ sur ces instances, il devient très faible avec $LS^{\star}$: on passe par exemple de 1.5% à 0.1% d'écart moyen sur les instances 101 à 125, et de 1.4% à 0.0% sur les instances 401 à 425.

Problème du Bin Packing. Le mouvement $\mathcal{M}_{chain}$ est purement un mouvement d'optimisation de packing. Il s'applique donc également à des problèmes de packing sans relations de précédence, comme le problème classique du Bin Packing (minimisation du nombre de bins pour contenir un ensemble donné d'objets). L'ajout des deux versions du mouvement $\mathcal{M}_{chain}$ permet ainsi d'améliorer les performances de LocalSolver sur ce problème. On compare les résultats de LocalSolver avec et sans le mouvement $\mathcal{M}_{chain}$ sur les instances proposées par Gschwind et Irnich en 2006 dans [49]. Ce benchmark est composé de deux cent quarante instances, regroupées en douze catégories de vingt instances chacune, comprenant entre 1212 et 5486 objets.

Instances	Nb objets	LS_0 / Best	$LS^\star$ / Best
csAA125	1329	0.7%	0.7%
csAA250	2587	0.6%	**0.5%**
csAA500	5161	0.7%	**0.5%**
csAB125	1319	0.2%	0.2%
csAB250	2651	0.2%	**0.1%**
csAB500	5259	0.2%	**0.1%**
csBA125	1340	0.7%	**0.6%**
csBA250	2640	0.5%	**0.4%**
csBA500	5240	0.8%	**0.6%**
csBB125	1333	0.4%	**0.3%**
csBB250	2631	0.2%	**0.1%**
csBB500	5233	0.2%	**0.1%**
Toutes		0.44%	**0.36%**

TABLE 4.22 – Gains de performances apportés par l'ajout des deux versions du mouvement $\mathcal{M}_{chain}$ sur le problème du Bin Packing

Dans la Table 4.22, on présente l'écart entre les solutions renvoyées par LocalSolver avec et sans le mouvement $\mathcal{M}_{chain}$ (notés respectivement $LS^\star$ et LS_0) et la meilleure borne connue. On donne également la taille moyenne des instances de chaque catégorie. L'ajout des deux versions du mouvement $\mathcal{M}_{chain}$ permet d'améliorer les résultats de LocalSolver sur 58% des instances, faisant passer l'écart moyen à la meilleure borne inférieure connue de 0.44% à 0.36%.

4.4 Synthèse des résultats numériques

On récapitule ici les progrès faits sur le problème de l'Assembly Line Balancing grâce aux travaux de la thèse. On compare ainsi deux versions de LocalSolver 10.5 : une version notée LS_0, pour laquelle tous les algorithmes décrits dans ce Chapitre ont été désactivés, et une version notée $LS^\star$, pour laquelle tous ces algorithmes sont activés. On compare également les résultats de LocalSolver à ceux obtenus par le solveur de programmation par contraintes CP Optimizer 20.1.0 (noté CPO dans la suite), souvent considéré comme un solveur de référence pour les problèmes d'ordonnancement. On utilise les paramètres de recherche par défaut pour chaque solveur. Le modèle utilisé pour évaluer les performances de CP Optimizer est donné dans [60]. La Table 4.23 présente l'écart à la meilleure solution connue des valeurs obtenues par LS_0, $LS^\star$ et CPO en deux et dix minutes de calcul. On donne également la valeur moyenne de la meilleure solution connue sur les instances de chaque catégorie, ainsi que le nombre et le pourcentage d'instances sur lesquelles chaque solveur parvient à trouver une solution ayant un écart à la meilleure solution connue inférieur à 1%. La Table 4.24 présente le nombre d'instances sur lesquelles $LS^\star$ obtient des résultats de meilleure qualité, de moins bonne qualité, ou égaux à la valeur obtenue par LS_0 ou par CPO, ou à la meilleure solution connue. On présente également les écarts moyens et maximaux sur ces groupes d'instances. La Figure 4.11 représente l'écart à la meilleure solution connue des valeurs obtenues par $LS^\star$ et CPO en deux minutes de calcul, en fonction de la valeur de la meilleure solution connue.

On voit que les performances de LocalSolver sur le problème de l'Assembly Line Balancing se sont considérablement améliorées : les résultats obtenus en deux minutes de calcul sont strictement meilleurs sur 99% des instances, avec des écarts importants (environ 3% en moyenne, et jusqu'à près de 10%). Ces améliorations sont particulièrement visibles sur les instances pour lesquelles LocalSolver rencontrait le plus de difficultés avant la thèse : l'écart à la meilleure solution connue passe par exemple de 7.5% à 0.5% (et 0.0% en dix minutes) sur les instances 476 à 500, et de 6.8% à 0.4% (et 0.0% en dix minutes) sur les instances 501 à 525.

Instances	Best known	Gap / Best 2 minutes			Gap / Best 10 minutes		
		LS_0	CPO	$LS^\star$	LS_0	CPO	$LS^\star$
1 – 25	136	0.8%	0.3%	0.2%	0.3%	0.2%	0.1%
26 – 50	507	0.5%	3.3%	0.1%	0.1%	3.3%	0.0%
51 – 75	228	0.9%	0.4%	0.0%	0.2%	0.4%	0.0%
76 – 100	137	2.5%	0.7%	0.4%	1.8%	0.3%	0.1%
101 – 125	514	1.4%	4.5%	0.1%	0.7%	4.4%	0.0%
126 – 150	226	1.9%	0.4%	0.0%	0.9%	0.4%	0.0%
151 – 175	138	2.8%	0.4%	0.4%	1.1%	0.4%	0.1%
176 – 200	508	2.2%	4.4%	0.4%	1.0%	4.4%	0.0%
201 – 225	230	2.3%	0.4%	0.0%	1.3%	0.4%	0.0%
226 – 250	138	3.5%	0.7%	0.4%	2.1%	0.7%	0.4%
251 – 275	527	5.8%	5.2%	0.8%	3.7%	5.1%	0.0%
276 – 300	225	4.5%	0.4%	0.4%	3.5%	0.4%	0.1%
301 – 325	138	0.7%	0.4%	0.0%	0.7%	0.2%	0.0%
326 – 350	505	1.3%	3.3%	0.2%	0.5%	3.3%	0.0%
351 – 375	228	1.1%	0.4%	0.0%	0.7%	0.4%	0.0%
376 – 400	136	2.9%	0.7%	0.7%	1.8%	0.5%	0.2%
401 – 425	521	2.3%	4.9%	0.0%	0.9%	4.9%	0.0%
426 – 450	223	3.7%	0.7%	0.4%	2.6%	0.6%	0.1%
451 – 475	136	6.8%	1.4%	1.4%	4.9%	0.8%	0.7%
476 – 500	546	7.5%	4.3%	0.5%	5.3%	3.8%	0.0%
501 – 525	226	6.8%	1.1%	0.4%	5.2%	0.8%	0.0%
Toutes	–	3.0%	1.9%	0.3%	1.9%	1.7%	0.1%
Instances	–	97	321	517	209	338	521
< 1% gap		18%	61%	98%	40%	64%	99%

TABLE 4.23 – Comparaison des performances de LocalSolver avant et après la thèse, ainsi qu'avec le solveur CP Optimizer, sur le problème de l'Assembly Line Balancing

Instances	Temps	Nb instances		Gap moyen		Gap max	
$LS^\star < LS_0$	2 min.	519	99%	9	2.9%	54	9.8%
$LS^\star > LS_0$	2 min.	0	0%				
$LS^\star = LS_0$	2 min.	6	1%				
$LS^\star <$ CPO	2 min.	373	71%	11.1	2.3%	32	6.2%
$LS^\star >$ CPO	2 min.	6	1%	1	0.7%	1	0.7%
$LS^\star =$ CPO	2 min.	149	28%				
$LS^\star >$ Best	2 min.	261	50%	1.6	0.5%	5	1.5%
$LS^\star =$ Best	2 min.	261	50%				
$LS^\star >$ Best	10 min.	82	16%	1.1	0.7%	2	1.4%
$LS^\star =$ Best	10 min.	443	84%				

TABLE 4.24 – Comparaison des performances de LocalSolver avant et après la thèse, ainsi qu'avec le solveur CP Optimizer, sur le problème de l'Assembly Line Balancing

On constate également que LocalSolver est désormais nettement plus performant que CP Optimizer sur ce problème. En effet, le résultat obtenu par LocalSolver est strictement meilleur sur 71% des instances, et au moins aussi bon sur 99% des instances. L'écart moyen à la meilleure solution connue est beaucoup plus faible pour LocalSolver (0.3% en deux minutes, 0.1% en dix minutes) que pour CP Optimizer (1.9% en deux minutes, 1.7% en dix minutes). On remarque également que LocalSolver parvient à trouver une solution ayant moins de 1% d'écart avec la meilleure solution connue sur un bien plus grand nombre d'instances que CP Optimizer (98% des instances

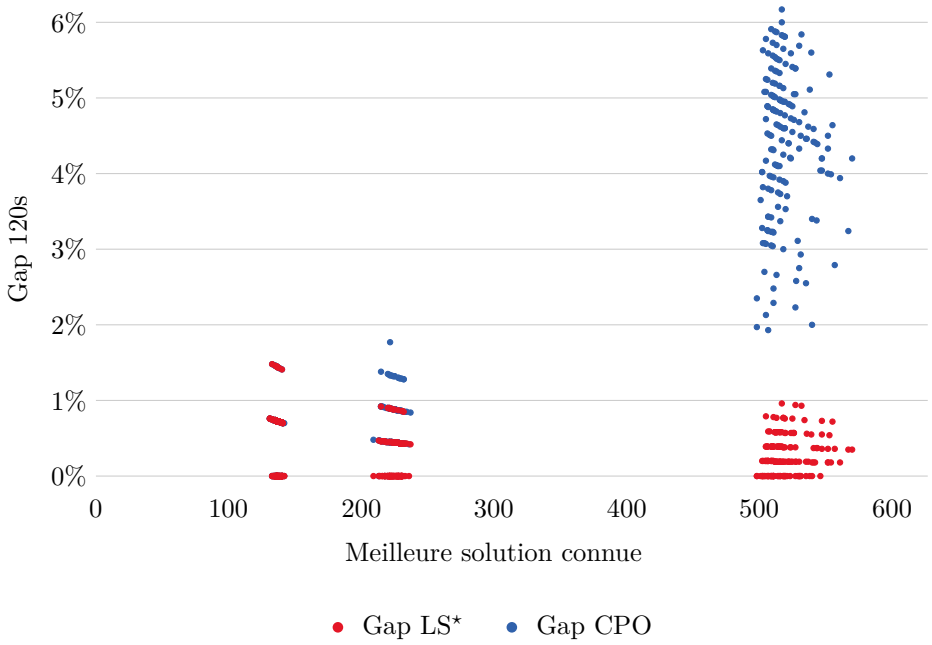


FIGURE 4.11 – Écart à la meilleure solution connue des solutions renvoyées par LS⋆ et CPO

pour LocalSolver en deux minutes, contre 61% pour CP Optimizer). L'écart de performance entre les deux solveurs est particulièrement visible sur les instances pour lesquelles la meilleure solution connue utilise plus de 500 stations de travail. En effet, le résultat renvoyé par CP Optimizer sur ces instances se trouve le plus souvent entre 3% et 5% de la meilleure solution connue, tandis que la valeur obtenue par LocalSolver est le plus souvent entre 0% et 1% de la meilleure solution connue. Ce résultat est clairement visible sur la Figure 4.11.

Enfin, on constate que LocalSolver atteint la meilleure solution connue sur la moitié des instances en deux minutes, et sur 84% des instances en dix minutes de calcul, avec un écart très faible sur les instances restantes. Pour rappel, la valeur de la meilleure solution connue utilisée ici correspond au minimum entre la valeur de référence donnée par [72] et celle obtenue avec LocalSolver. En effet, LocalSolver améliore la valeur de la meilleure solution connue donnée par [72] sur 59% des instances.

4.5 Conclusion

Dans ce Chapitre, on a montré comment les travaux de la thèse ont permis de faire de LocalSolver un solveur de choix pour résoudre le problème de l'Assembly Line Balancing. LocalSolver étant cependant un solveur générique, tous les algorithmes implémentés sont également utiles pour de nombreux autres problèmes présentant des caractéristiques communes dans leur structure. On s'est d'abord intéressé à la structure de variables de sets ordonnées révélée par les contraintes de respect des relations de précédence entre les tâches. On a ainsi implémenté un algorithme constructif reposant sur cette structure, permettant d'une part d'obtenir une solution initiale réalisable immédiatement, et d'autre part d'améliorer la solution courante au cours de la recherche, en l'intégrant dans un mouvement de la recherche locale. On a de plus montré comment cette structure pouvait également être exploitée par l'ensemble des mouvements de sets de la recherche locale, en ciblant en particulier des variables de sets « proches » les unes des autres. On s'est ensuite intéressé à la structure de packing du problème, traduite par les contraintes de respect du temps de cycle. En étudiant particulièrement les instances pour lesquelles il restait un écart à la meilleure solution connue significatif, on a élaboré un nouveau mouvement de recherche locale exploitant cette

structure de packing. Ce mouvement, décliné en deux versions, est basé sur le principe des chaînes d'éjection : on réalise des déplacements successifs d'éléments entre les différentes variables de sets. L'ajout de ce mouvement à la recherche locale de LocalSolver permet d'obtenir de très bonnes performances sur le problème de l'Assembly Line Balancing, mais également sur le problème du Bin Packing.

Chapitre 5

Conclusion et perspectives

5.1 Conclusion

Cette thèse porte de façon générale sur les problèmes d'ordonnancement, dont la résolution consiste à organiser la réalisation de tâches au cours du temps : déterminer leur répartition sur les différentes ressources disponibles ainsi que leurs dates d'exécution. Plus particulièrement, le travail de la thèse s'est concentré sur les problèmes d'ordonnancement disjonctif, dans lesquels chaque ressource ne peut exécuter qu'une seule tâche à la fois, et présentant des tâches non préemptives, ne pouvant pas être interrompues au cours de leur exécution. Les problèmes d'ordonnancement se rencontrent dans tous les domaines de l'industrie et des services, et sont parmi les problèmes les plus étudiés de la littérature, du fait des enjeux théoriques et applicatifs majeurs qu'ils recouvrent. Il était donc naturellement souhaitable de permettre aux utilisateurs de LocalSolver de modéliser et résoudre plus efficacement ce type de problèmes.

Le travail de cette thèse répond à deux problématiques principales, liées au traitement des problèmes d'ordonnancement disjonctif par le solveur d'optimisation mathématique LocalSolver. Le premier objectif se dégageant de ces problématiques était de permettre aux utilisateurs de LocalSolver de modéliser simplement de nombreux problèmes d'ordonnancement disjonctif. En tirant profit de la richesse du formalisme de modélisation ensembliste de LocalSolver, nous avons alors proposé des formulations génériques permettant d'exprimer simplement les notions de tâches, de relations de précédence, ou encore de contraintes de non-chevauchement des tâches affectées à une même ressource disjonctive. Les principaux critères utilisés dans le choix du formalisme retenu étaient d'abord la simplicité et la concision des formulations proposées, mais également leur généricité. En effet, cette généricité des opérateurs et expressions est à la base du paradigme de modélisation de LocalSolver.

Le deuxième objectif de la thèse était de permettre à LocalSolver d'obtenir très rapidement des solutions de très bonne qualité sur divers problèmes d'ordonnancement disjonctif. Là encore, la généricité des algorithmes de résolution implémentés était un point essentiel à leur bonne intégration au sein de LocalSolver. Le but des algorithmes élaborés au cours de la thèse n'était en effet pas d'améliorer les performances de LocalSolver sur un problème d'ordonnancement particulier, ni même seulement sur les problèmes d'ordonnancement, mais sur tous types de problèmes présentant des structures souvent rencontrées dans le domaine de l'ordonnancement disjonctif. Certaines contributions ciblant en particulier les problèmes d'ordonnancement sont ainsi également pertinentes sur d'autres types de problèmes, comme par exemple les problèmes de packing. En effet, le solveur ne « sait pas » quel type de problème lui a été soumis, et les algorithmes qui le constituent s'appliquent donc de façon très large, en se basant uniquement sur les équations définissant le modèle. Afin d'assurer une certaine cohérence des algorithmes implémentés au cours de la thèse avec le code existant, et pour leur permettre d'interagir efficacement sur des problèmes combinant des caractéristiques d'ordonnancement avec d'autres types de variables et d'expressions, il était

alors crucial de conserver un important degré de généricité.

Comme expliqué ci-dessus, la modélisation des problèmes d'ordonnancement disjonctif constituait l'un des enjeux majeurs de cette thèse. Le Chapitre 2 s'est alors intéressé à l'élaboration de formulations génériques permettant de modéliser les contraintes structurant la plupart des problèmes d'ordonnancement disjonctif : les relations de précédence et les contraintes de non-chevauchement des tâches. Les formulations génériques ainsi choisies reposent sur l'utilisation combinée de deux types de variables de décision offertes par LocalSolver : les variables entières, et les variables de listes. Les variables entières sont utilisées de façon classique, et permettent de modéliser les décisions intervenant dans la majorité des problèmes d'ordonnancement, à savoir les dates de début des tâches, ainsi que leurs durées lorsque celles-ci sont variables. Ces variables entières permettent d'exprimer très simplement les relations de précédence entre les tâches. Les variables de listes constituent quant à elles une spécificité du formalisme de modélisation ensembliste de LocalSolver : une variable de liste de domaine n correspond à une permutation d'un sous-ensemble de $\{0, ..., n-1\}$. En ordonnancement, les variables de listes permettent ainsi de modéliser l'ordre des tâches affectées aux différentes ressources disjonctives du problème. La présence de variables de listes dans le modèle permet de faciliter l'écriture de certaines expressions, comme les contraintes de non-chevauchement des tâches. La formulation générique choisie pour modéliser ces contraintes repose sur l'utilisation d'une lambda-fonction au sein d'une fonction « et » variadique, et correspond à la formulation suivante : « pour toute valeur de i, la tâche ordonnancée en position $i+1$ sur la ressource disjonctive considérée doit commencer après la date de fin de la tâche en position i sur cette ressource ». On a alors montré que cette formulation générique pouvait être adaptée afin de modéliser de nombreux problèmes d'ordonnancement disjonctif parfois très différents, de l'ordonnancement d'atelier (Job Shop par exemple) à la planification de production (Unit Commitment par exemple).

Afin d'exploiter au mieux cette modélisation à base de variables entières et de listes, nous avons tout d'abord élaboré un algorithme d'initialisation des variables de listes en fonction des bornes des variables entières qui leur sont associées. L'intégration de cet algorithme au solveur l'aide à trouver des solutions réalisables très rapidement sur les problèmes visés. Par exemple, il permet d'obtenir une solution réalisable immédiatement sur des problèmes comme celui de l'Aircraft Landing, et ainsi d'accélérer considérablement l'obtention de solutions de très bonne qualité. La présence de ressources disjonctives dans le problème, révélée par les contraintes de non-chevauchement des tâches, est également exploitée dans la recherche locale de LocalSolver. Treize types de mouvements élémentaires, dont l'activation repose sur la détection de structures particulières, ont ainsi été élaborés. Certains impliquent de modifier la durée des tâches, et s'appliquent donc uniquement lorsque les tâches ont des durées variables. D'autres reposent sur le déplacement de tâches d'une ressource à une autre, et ne sont donc actifs que lorsque l'affectation des tâches n'est pas fixe. D'autres encore consistent à ajouter de nouvelles tâches sur une ressource, ou au contraire à supprimer l'affectation d'une tâche, et sont uniquement appelés sur les problèmes comportant des tâches optionnelles. L'ajout de ces mouvements au sein de la recherche locale de LocalSolver a permis d'améliorer les performances du solveur sur différents types de problèmes. Ils permettent en effet d'obtenir un écart à la meilleure solution connue très faible sur différents problèmes d'ordonnancement d'atelier, comme les problèmes du Job Shop ou du Job Shop flexible, mais également d'obtenir de très importants gains de performance sur des problèmes de planification de production, comme le problème du Unit Commitment.

Même en lui ajoutant des mouvements dédiés, la recherche locale générique et à voisinages restreints de LocalSolver, lorsqu'elle est utilisée seule, rencontre des difficultés sur les problèmes d'ordonnancement disjonctif. En effet, les contraintes structurant ces problèmes, c'est-à-dire les contraintes de précédence et de non-chevauchement des tâches affectées à une même ressource disjonctive, sont souvent très serrées dans une solution de bonne qualité. Pour passer d'une bonne

solution à une autre, il est donc souvent nécessaire de modifier les dates de début d'un grand nombre de tâches. Avec la modélisation générique des problèmes d'ordonnancement présentée dans le Chapitre 2, cela correspond à modifier la valeur d'un grand nombre de variables entières, ce qui est contraire au principe de la recherche locale à voisinages restreints. Après l'application d'un mouvement de recherche locale sur une solution de bonne qualité, on obtient alors souvent une solution non réalisable. Plus largement, le Chapitre 3 n'est pas seulement consacré aux problèmes d'ordonnancement disjonctif, mais à tous types de problèmes caractérisés par un réseau d'inégalités linéaires binaires et ternaires, sur lesquels la recherche locale rencontre les mêmes difficultés. Ces réseaux correspondent en effet notamment aux contraintes de précédence ou de non-chevauchement généralisées rencontrées en ordonnancement, portant sur des tâches de durées fixes (inégalités binaires) ou variables (inégalités ternaires). Toutefois, on retrouve également ces contraintes généralisées dans des problèmes de packing, de layout, ou d'extraction minière.

Afin de pallier les difficultés rencontrées par la recherche locale sur ces problèmes, nous avons élaboré un algorithme de réparation de solutions par propagation des contraintes linéaires binaires et ternaires décrites au paragraphe précédent. Dans un souci de robustesse, un soin particulier a été apporté à la détection des formes génériques de ces contraintes dans le graphe d'évaluation du modèle. En effet, l'expression d'une contrainte correspondant à l'une des formes canoniques que l'on souhaite propager peut présenter de nombreuses variations d'écriture, se traduisant par des graphes d'évaluation très différents. Il était alors nécessaire de détecter toutes ces variations, grâce à une approche systématique, pour obtenir les meilleures performances possibles sur le plus grand nombre de modèles.

Notre algorithme de réparation, appliqué après chaque mouvement de la recherche locale aboutissant à une solution non réalisable, diffère de la propagation classique de la programmation par contraintes par plusieurs points. Tout d'abord, une réduction du domaine d'une variable X n'est propagée que si elle exclut sa valeur courante x. Dans ce cas, la variable X se voit attribuer une nouvelle valeur, égale à la projection de son ancienne valeur x sur son domaine réduit. Une autre différence est la possibilité de prendre des décisions aléatoires au cours de la propagation. Si l'on rencontre une contrainte pouvant être réparée de différentes manières, dont aucune ne prévaut *a priori* sur les autres, une réparation est choisie de manière aléatoire. Afin de réparer les solutions irréalisables en amplifiant le mouvement réalisé plutôt qu'en l'annulant, on impose de plus de ne jamais revenir sur une décision prise précédemment, et ainsi de modifier la valeur de chaque variable toujours dans la même direction. Chaque type de contraintes propagées possède son propre algorithme de filtrage, déterministe lorsqu'on peut garantir qu'une unique réparation nécessaire existe, et non déterministe dans le cas contraire. On a également décrit certaines propriétés souhaitables sur les contraintes propagées, qui assurent l'existence d'une réparation nécessaire et garantissent ainsi le succès de la procédure de réparation.

L'intégration de cet algorithme de réparation au sein de LocalSolver a permis, comme souhaité, d'améliorer considérablement les performances de la recherche locale sur les problèmes visés. En effet, la plupart des solutions améliorantes trouvées au cours de la recherche ont été construites grâce à un mouvement conduisant initialement à une solution infaisable, mais réparée par notre algorithme. Des gains de performance importants ont ainsi été constatés, non seulement sur des problèmes d'ordonnancement disjonctif académiques comme celui du Job Shop, de l'Open Shop ou du Unit Commitment, mais également sur des instances industrielles de packing multidimensionnel et d'extraction minière.

Le Chapitre 4, s'est concentré sur une autre classe de problèmes d'ordonnancement disjonctif, en prenant pour exemple le problème de l'Assembly Line Balancing. Ce problème n'est pas modélisé avec des contraintes de non-chevauchement exprimées à partir de variables entières et de listes comme dans les Chapitres précédents, mais utilise des variables de sets pour seules décisions. Les contributions de ce Chapitre montrent comment les différentes structures du problème (contraintes de packing sur les variables de sets, et relations de précédence entre les éléments des sets) peuvent

être exploitées pour élaborer des algorithmes de résolution efficaces.

La première contribution présentée consiste en un algorithme constructif attribuant les tâches aux différentes stations de travail, en respectant la capacité des stations de travail et les relations de précédence entre les tâches. Tout d'abord intégré à LocalSolver en tant qu'algorithme d'initialisation des variables de sets, cet algorithme permet de construire une première solution réalisable pour le problème. La recherche est alors accélérée, puisqu'elle démarre immédiatement à partir d'une solution de bonne qualité. Ce résultat est d'autant plus appréciable que la recherche locale pouvait auparavant mettre plusieurs dizaines de secondes à atteindre la faisabilité. Cet algorithme a également été intégré au sein de la recherche locale de LocalSolver, dans un mouvement de type « *destroy and repair* », consistant à réorganiser les tâches attribuées à un sous-ensemble de stations de travail. L'ajout de ce mouvement a permis d'obtenir des gains de performance significatifs sur le problème de l'Assembly Line Balancing. Afin d'améliorer encore davantage les performances du solveur, la structure de variables de sets ordonnées utilisée dans notre algorithme constructif est également exploitée dans les autres mouvements de la recherche locale portant sur des variables de sets : elle est utilisée pour cibler des échanges d'éléments entre des sets « proches », et ainsi diminuer la probabilité de violer des contraintes de précédence en réalisant ces échanges.

Le deuxième algorithme présenté dans le Chapitre 4 est également un mouvement de recherche locale, exploitant cette fois uniquement la structure de packing révélée par les contraintes de capacité sur les stations de travail, et basé sur le principe des chaînes d'éjection. Le principe de ce mouvement consiste à sélectionner k variables de sets non vides, et à réarranger les éléments qu'elles contiennent de façon à pouvoir vider l'une d'entre elles. Ce mouvement est décliné en deux versions. La première version consiste à éjecter et réinsérer dans d'autres sets des éléments de poids de plus en plus faibles, jusqu'à pouvoir placer l'élément éjecté sans avoir à réaliser de nouvelle éjection. La seconde version consiste à déplacer des éléments de poids de plus en plus grands, de façon à libérer de plus en plus d'espace dans leur set de départ, jusqu'à vider entièrement une variable de set. L'intégration des deux versions de ce mouvement à la recherche locale de LocalSolver lui permet d'obtenir d'excellentes performances sur le problème de l'Assembly Line Balancing. En effet, LocalSolver parvient à améliorer la valeur de la meilleure solution connue sur la moitié des instances d'un benchmark de 525 instances de grande taille, et obtient des résultats significativement meilleurs que le solveur de programmation par contraintes CP Optimizer (IBM) sur l'ensemble des instances. Cet algorithme est de plus générique et non dédié au problème de l'Assembly Line Balancing : son intégration à LocalSolver a également permis d'améliorer nettement ses performances sur d'autres problèmes, comme par exemple sur des instances difficiles du problème du Bin Packing.

5.2 Perspectives

Le travail de cette thèse a donné lieu au sein de LocalSolver à d'importants gains de performance sur divers problèmes d'ordonnancement disjonctif. De nombreuses pistes restent cependant à explorer, et donnent des perspectives de travaux futurs, tant du point de vue de la modélisation que de celui de la résolution des problèmes d'ordonnancement. Sur le premier point, si le travail de cette thèse a permis de définir un formalisme de modélisation concis et efficace pour les problèmes caractérisés par des ressources disjonctives et des tâches non préemptives, un travail similaire reste à faire sur d'autres branches de l'ordonnancement. On pourra alors réfléchir à la manière de modéliser les ressources discrètes, les fonctions d'intensité, les batches, ou encore la préemption, en ajoutant éventuellement de nouveaux opérateurs, tout en conservant des éléments de modélisation les plus génériques possibles. De même, si l'on a d'abord souhaité se concentrer sur la recherche de solutions de bonne qualité, on pourra par la suite s'intéresser également au calcul de bornes sur les problèmes d'ordonnancement.

Calcul de bornes. Le travail de la thèse s'est concentré sur l'obtention rapide de solutions de qualité sur divers problèmes d'ordonnancement disjonctif, en enrichissant la composante de recherche locale de LocalSolver, sans chercher dans un premier temps à améliorer la qualité des bornes inférieures sur ces problèmes. Une première perspective pour de futurs travaux serait alors de calculer de meilleures bornes inférieures sur les problèmes d'ordonnancement. En plus de la composante de recherche locale, LocalSolver possède des composantes exactes (CP, MIP, MINLP), fournissant notamment des bornes inférieures et des preuves d'optimalité sur de nombreux problèmes. Ces composantes traitent principalement des modèles écrits à partir de variables numériques classiques, ainsi que certains modèles ensemblistes pouvant être reformulés en variables numériques (par exemple, on calcule des bornes pour le problème de l'Assembly Line Balancing en linéarisant automatiquement le modèle ensembliste présenté dans le Chapitre 4). Les contraintes de non-chevauchement des tâches, présentes dans les modèles d'ordonnancement disjonctif à base de variables entières et de listes décrits dans le Chapitre 2, ne sont cependant actuellement pas reformulées. Les bornes inférieures fournies par le solveur sur ce type de modèles sont alors calculées en prétraitement par propagation, notamment à partir des contraintes de précédence entre les tâches. Plusieurs pistes sont alors possibles pour améliorer la qualité des bornes calculées sur ces problèmes. On pourra ainsi linéariser les contraintes de non-chevauchement des tâches, pour que les problèmes concernés puissent être traités par les composantes exactes déjà présentes dans le solveur, ou encore intégrer de nouvelles composantes exactes capables de traiter directement ces modèles ensemblistes, à base de programmation par contraintes par exemple.

Fonctions d'intensité. Du point de vue de la modélisation, une première évolution possible pour le traitement des problèmes d'ordonnancement par LocalSolver serait de s'intéresser aux problèmes avec des ressources disjonctives présentant des fonctions d'intensité, c'est-à-dire pour lesquelles l'« intensité » du travail pouvant être fourni varie au cours du temps. On pourra dans un premier temps s'intéresser en particulier aux fonctions d'intensité booléennes. Ce type de fonctions permet par exemple de modéliser des plages de disponibilité (intensité égale à 1) et de non-disponibilité (intensité égale à 0) sur les ressources. Il est actuellement possible de modéliser les plages de non-disponibilité comme des tâches fictives fixes, intégrées dans la contrainte de non-chevauchement des tâches, comme expliqué dans le Chapitre 2. Cependant, cette modélisation suppose qu'aucune tâche ne peut commencer avant une plage de non-disponibilité, et terminer après. On pourrait au contraire souhaiter modéliser des problèmes dans lesquels les tâches peuvent s'exécuter de part et d'autre d'une plage de non-disponibilité (une tâche commence à être exécutée, puis se met en pause lorsque la ressource devient indisponible, et reprend lorsque celle-ci est de nouveau disponible), ce qui constituerait un premier pas vers la gestion de tâches préemptives. Dans ce cas, la présence de plages de disponibilité et de non-disponibilité se traduit par le fait que la durée de chaque tâche est non pas une constante, mais une fonction de sa date de début. Une tâche de durée d sur une ressource toujours disponible doit s'exécuter sur d pas de temps : ici, cela correspond à dire que la date de fin de la tâche est le premier instant tel que la ressource a été disponible pendant d pas de temps depuis sa date de début. Pour s'assurer d'obtenir de bonnes performances sur ce type de problèmes, il faudra alors modifier les mouvements de recherche locale présentés dans le Chapitre 2, ainsi que les algorithmes de filtrage permettant de propager les contraintes de précédence et non-chevauchement des tâches, décrits dans le Chapitre 3, afin de traiter efficacement ce nouveau type de tâches, dont les durées sont variables mais totalement déterminées par leurs dates de début.

Ressources discrètes ou cumulatives. Si le travail de la thèse s'est concentré sur le traitement des ressources disjonctives, on pourra par la suite étudier d'autres types de ressources, comme les ressources discrètes, aussi appelées ressources cumulatives. Tout comme la présence de ressources disjonctives se traduit par des contraintes de non-chevauchement des tâches dans le modèle, les ressources discrètes sont associées à des contraintes de capacité. Chaque tâche utilisant une ressource discrète en consomme une certaine quantité. A chaque instant t, la somme des quantités consom-

mées par les tâches utilisant une certaine ressource discrète à l'instant t ne doit pas dépasser la capacité de cette ressource. Comme pour les contraintes de non-chevauchement des tâches décrites dans le Chapitre 2, on pourrait modéliser ces contraintes de capacité en exploitant les fonctions variadiques et lambda-fonctions offertes par le formalisme de modélisation de LocalSolver. En utilisant une fonction « et » variadique sur le temps (pour chaque instant de 0 à l'horizon), et une lambda-fonction exprimant la contrainte de capacité à chaque pas de temps, on peut exprimer cette contrainte complexe en n'utilisant que peu de mémoire, même si l'horizon est très grand. En effet, avec une telle formulation, la contrainte est définie en intention et non en extension. Pour garantir de bonnes performances sur les problèmes présentant de telles contraintes, il faudra alors détecter automatiquement ces contraintes pour les gérer efficacement au sein du solveur, notamment en s'assurant de toujours les traiter sous leur forme compacte, sans jamais les « dérouler ». Par la suite, on pourra également réaliser sur ces ressources discrètes un travail similaire à celui réalisé pendant la thèse sur les ressources disjonctives : ajouter des mouvements spécifiques à la recherche locale, en suivant le même principe que dans le Chapitre 2, et définir un algorithme de filtrage pour les contraintes de capacité, afin de les intégrer à l'algorithme de réparation de solutions par propagation décrit dans le Chapitre 3.

Ordonnancement par batches. D'autres problèmes comportent des ressources par batches : des tâches de types et/ou durées compatibles sont regroupées dans des batches pour être traitées ensemble par la ressource. Les variables de sets, souvent utilisées dans les problèmes de packing et de clustering, ou encore dans le problème de l'Assembly Line Balancing présenté dans le Chapitre 4, semblent un choix naturel pour représenter les batches. Si les ressources traitant ces batches peuvent être considérées comme des ressources disjonctives (exécutant des batches de tâches et non des tâches simples), elles peuvent être modélisées à partir de variables de listes représentant l'ordre de différents batches. On distingue souvent le batching en série (la durée d'un batch est égale à la somme des durées de tâches qui le composent) du batching en parallèle (la durée d'un batch est égale au maximum des durées des tâches qui le composent). Dans les deux cas, une modélisation à base de variables de sets permettrait d'exprimer facilement la durée de chaque batch, en utilisant des fonctions variadiques de type « somme » et « max » respectivement.

Planification de production avec aspects continus. L'ensemble des problèmes étudiés pendant la thèse étaient purement combinatoires. De nombreux problèmes d'ordonnancement, notamment en planification de production, sont cependant mixtes, et présentent des aspects continus en plus de leurs aspects combinatoires. C'est par exemple le cas du problème du Unit Commitment, étudié dans une version simplifiée purement combinatoire dans les Chapitres 2 et 3. Dans la version classique du problème, on doit également décider du niveau de production de chaque usine à chaque instant, avec des contraintes sur l'évolution de ce niveau de production au cours du temps. LocalSolver possède déjà des algorithmes exploitant la structure de ce type de problèmes mixtes, utiles notamment lorsque les aspects continus consistent en un sous-problème linéaire, que l'on résout alors à chaque itération de la recherche locale après avoir fixé la valeur des variables entières et ensemblistes. Il serait pertinent d'élaborer de nouveaux mouvements de recherche locale, exploitant simultanément les différentes structures présentes dans les problèmes d'ordonnancement mixtes, afin de chercher des solutions améliorantes en se basant à la fois sur les aspects combinatoires et continus du modèle.

Bibliographie

[1] A. Agarwal, S. Colak, and S. Erenguc. Metaheuristic Methods. In C. Schwindt and J. Zimmermann, editors, *Handbook on Project Management and Scheduling Vol.1*, International Handbooks on Information Systems, chapter 0, pages 57–74. Springer, December 2015.

[2] M. Ågren. Set variables and local search. In M. Wallace, editor, *Principles and Practice of Constraint Programming – CP 2004*, pages 788–788, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[3] A. Ahmeti and N. Musliu. Min-conflicts heuristic for multi-mode resource-constrained projects scheduling. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '18, page 237–244. Association for Computing Machinery, 2018.

[4] A. Angulo, D. Espinoza, and R. Palma. Thermal unit commitment instances for paper : A polyhedral-based approach applied to quadratic cost curves in the unit commitment problem. http ://www.dii.uchile.cl/∼daespino/UC_instances _archivos/portada.htm.

[5] V. Antuori, E. Hébrard, M. J. Huguet, S. Essodaigui, and A. Nguyen. Leveraging reinforcement learning, constraint programming and local search : a case study in car manufacturing. In C. Springer, editor, *International Conference on Principles and Practice of Constraint Programming*, pages 657–672, September 2020.

[6] D. Applegate and W. Cook. A computational study of the job-shop scheduling problem. *ORSA Journal on Computing*, 3 :149–156, 1991.

[7] J. A. Araujo, H. G. Santos, B. Gendron, S. D. Jena, S. S. Brito, and D. S. Souza. Strong bounds for resource constrained project scheduling : Preprocessing and cutting planes. *Computers & Operations Research*, 113 :104782, Jan 2020.

[8] C. Artigues and D. Feillet. A branch and bound method for the job-shop problem with sequence-dependent setup times. *Annals of Operations Research*, page 25 pages, 2007.

[9] F. Ballestín and N. Trautmann. An iterated-local-search heuristic for the resource-constrained weighted earliness-tardiness project scheduling problem. *International Journal of Production Research*, 46 :6231–6249, 11 2008.

[10] P. Baptiste, C. Pape, and W. Nuijten. *Constraint-Based Scheduling : Applying Constraint Programming to Scheduling Problems*. International Series in Operations Research & Management Science. Springer US, 2012.

[11] R. Barták, M. A. Salido, and F. Rossi. Constraint satisfaction techniques in planning and scheduling. *Journal of Intelligent Manufacturing*, 21(1) :5–15, 2010.

[12] I. Baybars. A survey of exact algorithms for the simple assembly line balancing problem. *Management Science*, 32(8) :909–932, 1986.

[13] J. Beck, T. Feng, and J.-P. Watson. Combining constraint programming and local search for job-shop scheduling. *INFORMS Journal on Computing*, 23 :1–14, 2011.

[14] D. Behnke and M. J. Geiger. Test instances for the flexible job shop scheduling problem with work centers. Technical report, Helmut-Schmidt-Universität, Universität der Bundeswehr Hamburg, 2012.

[15] T. Benoist, B. Estellon, F. Gardi, R. Megel, and K. Nouioua. LocalSolver 1.x : A black-box local-search solver for 0-1 programming. *4OR*, 9 :299–316, 2011.

[16] L. Blaise. Réparation de solutions par propagation de réseaux d'inégalités dans LocalSolver. EasyChair Preprint no. 2498, ROADEF 2020.

[17] L. Blaise, C. Artigues, and T. Benoist. Solution Repair by Inequality Network Propagation in LocalSolver. In T. Bäck, M. Preuss, A. Deutz, H. Wang, C. Doerr, M. Emmerich, and H. Trautmann, editors, *Parallel Problem Solving from Nature – PPSN XVI*, pages 332–345, Cham, 2020. Springer International Publishing.

[18] L. Blaise, C. Artigues, and T. Benoist. Solution Repair by Inequality Network Propagation in LocalSolver. In 17th International Workshop on Project Management and Scheduling, editors, *Book of extended abstracts*, 2020.

[19] L. Blaise, T. Benoist, and C. Artigues. Solving the Assembly Line Balancing Problem with LocalSolver. 18th International Workshop on Project Management and Scheduling, https ://pms2022.sciencesconf.org/375719, 2022.

[20] L. Blaise, T. Benoist, and C. Artigues. Modélisation de ressources disjonctives avec LocalSolver. https ://roadef2021.sciencesconf.org/350276, ROADEF 2021.

[21] L. Blaise, T. Benoist, and C. Artigues. Résolution du problème de l'Assembly Line Balancing avec LocalSolver. https ://roadef2022.sciencesconf.org/375723, ROADEF 2022.

[22] L. Borba, M. Ritt, and C. Miralles. Exact and heuristic methods for solving the robotic assembly line balancing problem. *European Journal of Operational Research*, 270(1) :146–156, 2018.

[23] A. Borghetti, A. Frangioni, F. Lacalandra, and C. Nucci. Lagrangian heuristics based on disaggregated bundle methods for hydrothermal unit commitment. *IEEE Transactions on Power Systems*, 18(1) :313–323, 2003.

[24] S. Boulmier. *Optimisation globale avec LocalSolver*. PhD thesis, Université Grenoble Alpes, 2020.

[25] N. Boysen, M. Fliedner, R. Klein, and A. Scholl. Assembly line balancing problem. https ://assembly-line-balancing.de/salbp/.

[26] R. Capua, Y. Frota, L. S. Ochi, and T. Vidal. A study on exponential-size neighborhoods for the bin packing problem with conflicts. *Journal of Heuristics*, 24(4) :667–695, August 2018.

[27] J. Carlier. The one-machine sequencing problem. *European Journal of Operational Research*, 11(1) :42–47, 1982. Third EURO IV Special Issue.

[28] J. Carlier and E. Pinson. An algorithm for solving the job-shop problem. *Management Science*, 35(2) :164–176, 1989.

[29] D. M. Carvalho and M. C. Nascimento. Hybrid matheuristics to solve the integrated lot sizing and scheduling problem on parallel machines with sequence-dependent and non-triangular setup. *European Journal of Operational Research*, 296(1) :158–173, 2022.

[30] J. Chambers and J. W. Barnes. Reactive search for flexible job shop scheduling. *Graduate program in Operations Research and Industrial Engineering, The University of Texas at Austin, Technical Report Series, ORP98-04*, 1998.

[31] G. Chu, P. J. Stuckey, A. Schutt, T. Ehlers, G. Gange, and K. Francis. Chuffed, a lazy clause generation solver. *URL : https ://github. com/chuffed/chuffed*, 2018.

[32] A. A. Cire and W.-J. van Hoeve. Multivalued decision diagrams for sequencing problems. *Operations Research*, 61(6) :1411–1428, 2013.

[33] G. H. I. de Azevedo and A. A. Pessoa. A sat based exact method to the rcpsp/max. In Blumenau-SC, editor, *XLIX Simposio Brasileiro de Pesquisa Operacional*, 2017.

[34] V. L. de Lima, C. Alves, F. Clautiaux, M. Iori, and J. M. Valério de Carvalho. Arc flow formulations based on dynamic programming : Theoretical foundations and applications. *European Journal of Operational Research*, 296(1) :3–21, 2022.

[35] E. A. G. de Souza, M. S. Nagano, and G. A. Rolim. Dynamic programming algorithms and their applications in machine scheduling : A review. *Expert Systems with Applications*, 190 :116180, 2022.

[36] E. L. Demeulemeester and W. S. Herroelen. New benchmark results for the resource-constrained project scheduling problem. *Management Science*, 43(11) :1485–1492, 1997.

[37] J. Ding, L. Shen, Z. Lü, and B. Peng. Parallel machine scheduling with completion-time-based criteria and sequence-dependent deterioration. *Computers & Operations Research*, 103 :35 – 45, 2019.

[38] N. Dupin and E.-g. Talbi. Parallel matheuristics for the discrete unit commitment problem with min-stop ramping constraints. *International Transactions in Operational Research*, 27(1) :219–244, 2020.

[39] S. Even, A. Itai, and A. Shamir. On the complexity of timetable and multicommodity flow problems. *SIAM J. Comput.*, 5 :691–703, 1976.

[40] H. Fisher and G. Thompson. Probabilistic learning combinations of local job-shop scheduling rules. *Industrial Scheduling*, 1963.

[41] C. Floudas and X. Lin. Mixed integer linear programming in process scheduling : Modeling, algorithms, and applications. *Annals of Operations Research*, 139(1) :131–162, 2005.

[42] A. M. Frisch, M. Grum, C. Jefferson, B. M. Hernández, and I. Miguel. The design of ESSENCE : A constraint language for specifying combinatorial problems. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence*, IJCAI'07, page 80–87, San Francisco, CA, USA, 2007. Morgan Kaufmann Publishers Inc.

[43] F. Gardi, T. Benoist, J. Darlay, B. Estellon, and R. Megel. *Mathematical Programming Solver Based on Local Search*. John Wiley & Sons, Ltd, 2014.

[44] C. Gervet. *Set Intervals in Constraint Logic Programming : Definition and implementation of a language*. Theses, Université de Franche Comté Besançon, Sep 1995.

[45] B. Giffler and G. L. Thompson. Algorithms for solving production-scheduling problems. *Operations Research*, 8(4) :487–503, 1960.

[46] F. Glover. Ejection chains, reference structures and alternating path methods for traveling salesman problems. *Discrete Applied Mathematics*, 65(1) :223 – 253, 1996.

[47] D. Godard, P. Laborie, and W. Nuijten. Randomized large neighborhood search for cumulative scheduling. In *ICAPS*, 2005.

[48] D. Grimes, E. Hebrard, and A. Malapert. Closing the open shop : Contradicting conventional wisdom. In *Principles and Practice of Constraint Programming*, pages 400–408, 09 2009.

[49] T. Gschwind and S. Irnich. Dual inequalities for stabilized column generation revisited. *INFORMS Journal on Computing*, 28(1) :175–194, 2016.

[50] E. Guzman, B. Andres, and R. Poler. Matheuristic algorithm for job-shop scheduling problem using a disjunctive mathematical model. *Computers*, 11(1), 2022.

[51] E. Hebrard. Mistral, a constraint satisfaction library. *Proceedings of the Third International CSP Solver Competition*, 3(3) :31–39, 2008.

[52] N. Jussien, G. Rochart, and X. Lorca. Choco : an Open Source Java Constraint Programming Library. In *CPAIOR'08 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08)*, pages 1–10, Paris, France, France, 2008.

[53] B. Knueven, J. Ostrowski, and J.-P. Watson. On mixed-integer programming formulations for the unit commitment problem. *INFORMS Journal on Computing*, 32(4) :857–876, 2020.

[54] C. Koch, T. Arbaoui, Y. Ouazene, F. Yalaoui, H. De Brunier, N. Jaunet, and A. De Wulf. A matheuristic approach for solving a simultaneous lot sizing and scheduling problem with client prioritization in tire industry. *Computers & Industrial Engineering*, 165 :107932, Mar 2022.

[55] R. Kolisch. Serial and parallel resource-constrained project scheduling methods revisited : Theory and computation. *European Journal of Operational Research*, 90(2) :320–333, 1996.

[56] R. Kolisch and S. Hartmann. Experimental investigation of heuristics for resource-constrained project scheduling : An update. *European Journal of Operational Research*, 174 :23–37, 10 2006.

[57] M. Koshimura, H. Nabeshima, H. Fujita, and R. Hasegawa. Solving open job-shop scheduling problems by sat encoding. *IEICE Transactions on Information and Systems*, E93.D(8) :2316–2318, 2010.

[58] S. Kreter, A. Schutt, and P. Stuckey. Using constraint programming for solving rcpsp/max-cal. *Constraints*, 22(3) :432–462, 2017.

[59] P. Laborie. An update on the comparison of mip, cp and hybrid approaches for mixed resource allocation and scheduling. In W.-J. van Hoeve, editor, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 403–411. Springer International Publishing, 2018.

[60] P. Laborie. Solving the simple assembly line balancing problem with cp optimizer. https ://www.linkedin.com/pulse/solving-simple-assembly-line-balancing-problem-cp-philippe-laborie/, 2020.

[61] P. Laborie, J. Rogerie, P. Shaw, and P. Vilím. Ibm ilog cp optimizer for scheduling. *Constraints*, 23(2) :210–250, apr 2018.

[62] S. Lawrence. Resource-constrained project scheduling : an experimental investigation of heuristic scheduling techniques (supplement). Technical report, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1984.

[63] J. Lenstra and E. Aarts. *Local search in combinatorial optimization*. Wiley-Interscience series in discrete mathematics and optimization. Wiley-Interscience, 1997.

[64] Z. Li, I. Kucukkoc, and Q. Tang. A comparative study of exact methods for the simple assembly line balancing problem. *Soft Computing*, 24, 2020.

[65] Y. Mati, S. Dauzère-Pérès, and C. Lahlou. A general approach for optimizing regular criteria in the job-shop scheduling problem. *European Journal of Operational Research*, 212(1) :33–42, July 2011.

[66] S. Minton, M. D. Johnston, A. B. Philips, and P. Laird. Minimizing conflicts : a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58(1) :161 – 205, 1992.

[67] N. Mladenović and P. Hansen. Variable neighborhood search. *Computers and Operations Research*, 24(11) :1097–1100, 1997.

[68] U. Montanari. Networks of constraints : Fundamental properties an applications to picture processing. *Information Sciences*, 7 :95 – 132, 1974.

[69] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. M. Chaff. Engineering an efficient sat solver. In *38th Annual Design Automation Conference*, pages 530–535, 2001.

[70] E. Nowicki and C. Smutnicki. A fast taboo search algorithm for the job shop problem. *Management Science*, 42(6) :797–813, 1996.

[71] E. Nowicki and C. Smutnicki. An advanced tabu search algorithm for the job shop problem. *Journal of Scheduling*, 8 :145–159, 04 2005.

[72] A. Otto, C. Otto, and A. Scholl. Systematic data generation and test design for solution algorithms on the example of SALBPGen for assembly line balancing. *European Journal of Operational Research*, 228(1) :33–45, 2013.

[73] M. Palpant, C. Artigues, and P. Michelon. Lssper : Solving the resource-constrained project scheduling problem with large neighbourhood search. *Annals OR*, 131 :237–257, 10 2004.

[74] P. M. Pardalos, O. V. Shylo, and A. Vazacopoulos. Solving job shop scheduling problems utilizing the properties of backbone and "big valley". *Computational Optimization and Applications*, 47 :61–76, 2010.

[75] J. Park, J. Chun, S. H. Kim, Y. Kim, and J. Park. Learning to schedule job-shop problems : representation and policy learning using graph neural network and reinforcement learning. *International Journal of Production Research*, 59(11) :3360–3377, 2021.

[76] R. Pellerin, N. Perrier, and F. Berthaut. A survey of hybrid metaheuristics for the resource-constrained project scheduling problem. *European Journal of Operational Research*, 280(2) :395–416, 2020.

[77] B. Peng, Z. Lü, and T. Cheng. A tabu search/path relinking algorithm to solve the job shop scheduling problem. *Computers & Operations Research*, 53 :154 – 164, 2015.

[78] J. Pereira and E. Alvarez-Miranda. An exact approach for the robust assembly line balancing problem. *Omega*, 78 :85–98, 2018.

[79] L. Perron. Operations research and constraint programming at google. In J. Lee, editor, *Principles and Practice of Constraint Programming – CP 2011*, pages 2–2, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[80] H. Pinol and J. Beasley. Scatter search and bionomic algorithms for the aircraft landing problem. *European Journal of Operational Research*, 171 :439–462, 06 2006.

[81] O. Polo-Mejía, C. Artigues, P. Lopez, and V. Basini. Mixed-integer/linear and constraint programming approaches for activity scheduling in a nuclear research facility. *International Journal of Production Research*, 58(23) :7149–7166, 2020.

[82] J. Poppenborg and S. Knust. A flow-based tabu search algorithm for the rcpsp with transfer times. *OR Spectrum*, 38 :305–334, 2016.

[83] F. Rossi, P. v. Beek, and T. Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., USA, 2006.

[84] A. Salehipour and M. Ahmadian. A heuristic algorithm for the aircraft landing problem. In *22nd International Congress on Modelling and Simulation*, 2017.

[85] A. Scholl and C. Becker. State-of-the-art exact and heuristic solution procedures for simple assembly line balancing. *European Journal of Operational Research*, 168 :666–693, 02 2006.

[86] C. Schulte and P. J. Stuckey. Efficient constraint propagation engines. *ACM Trans. Program. Lang. Syst.*, 31(1), dec 2008.

[87] A. Schutt, T. Feydy, P. J. Stuckey, and M. G. Wallace. Solving rcpsp/max by lazy clause generation. *Journal of Scheduling*, 16(3) :273–289, 2013.

[88] L. Shen, S. Dauzère-Pérès, and J. S. Neufeld. Solving the flexible job shop scheduling problem with sequence-dependent setup times. *European Journal of Operational Research*, 265(2) :503–516, 2018.

[89] M. Siala, C. Artigues, and E. Hébrard. Two Clause Learning Approaches for Disjunctive Scheduling. In G. Pesant, editor, *Principles and Practice of Constraint Programming*, volume 9255 of *Lecture Notes in Computer Science*, pages 393–402, Cork, Ireland, 2015.

[90] E. Taillard. Some efficient heuristic methods for the flow shop sequencing problem. *European Journal of Operational Research*, 47(1) :65–74, 1990.

[91] E. Taillard. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64(2) :278 – 285, 1993.

[92] E. Taillard. Parallel taboo search techniques for the job shop scheduling problem. *ORSA journal on Computing*, 6(2) :108–117, 1994.

[93] E.-G. Talbi, F. Yalaoui, and L. Amodeo. *Metaheuristics for Production Systems*, volume 60 of *Operations Research/Computer Science Interfaces Series (ORCS)*. Springer, 2016.

[94] R. Vaessens, E. Aarts, and J. Lenstra. Job shop scheduling by local search. *INFORMS Journal on Computing*, 8(3) :302–317, 1996.

[95] J. J. van Hoorn. The current state of bounds on benchmark instances of the job-shop scheduling problem. *Journal of Scheduling*, 21 :127–128, 2018.

[96] P. Vilim, P. Laborie, and P. Shaw. Failure-directed search for constraint-based scheduling. In L. Michel, editor, *Integration of AI and OR Techniques in Constraint Programming*, pages 437–453, Cham, 2015. Springer International Publishing.

[97] J. Zhang, G. Ding, Y. Zou, S. Qin, and J. Fu. Review of job shop scheduling research and its new perspectives under industry 4.0. *Journal of Intelligent Manufacturing*, 2017.