École doctorale nº 072 : Sciences Pour l'Ingénieur

Université de Lille

THÈSE

pour obtenir le grade de

Docteur de l'Université de Lille

Spécialité "Informatique"

présentée et soutenue publiquement par

Lucien MOUSIN

le 28 / 11 / 2018

Extraire et exploiter la connaissance pour mieux optimiser

Jury

Présidente:	Sophie TISON	Professeur des Universités, Université de Lille
Rapporteurs :	Nicolas JOZEFOWIEZ	Professeur des Universités, Université de Lorraine
	Adrien GOËFFON	Maître de conférences, HDR, Université d'Angers
Examinateur :	Lhassane IDOUMGHAR	Professeur des Universités, Université de Haute Alsace
Directrice :	Clarisse DHAENENS	Professeur des Universités, Université de Lille
Co-encadrante :	Marie-Eléonore KESSACI	Maître de conférences, Université de Lille

Université de Lille
Centre de Recherche en Informatique, Signal et Automatique de Lille (CRIStAL)
UMR 9189, F-59655 Lille, France

Remerciements

Mes trois années de doctorat ont été l'occasion de réaliser de nombreuses rencontres professionnelles et personnelles. Toutes ces personnes ont permis, d'une manière ou d'une autre à ces travaux d'aboutir, et donc pour ces raisons, je vais essayer de toutes les remercier en essayant de n'oublier personne...

Premièrement, je souhaite remercier Clarisse et Marie-Eléonore, directrice et co-encadrante de cette thèse, de m'avoir conseillé, soutenu, encouragé tout au long de ces trois années de doctorat. Même dans les moments de doute, vous avez réussi a me pousser à rebondir et aller jusqu'au bout de cette aventure, et c'est grâce à vous que ce manuscrit a abouti, donc merci encore. Dans le même temps, j'adresse tous mes remerciements à Laetitia, notre chef d'équipe, qui m'a suivi de près ou de loin depuis mon entrée au M1 de l'université de Lille, merci pour ce suivi et tes nombreux conseils.

Je souhaite également remercier les membres du jury : Sophie Tison, présidente, Adrien Goeffon et Nicolas Jozefowiez, rapporteurs, et Lhassane Idoumghar, examinateur de cette thèse pour l'intérêt porté à mes travaux de recherche et leurs nombreux retours.

Je remercie également les membres de notre équipe de recherche et des autres équipes pour tous les conseils et bons moments passés ensemble : pour l'époque INRIA, je remercie les permanents, Bilel, Arnaud et Fanny, ainsi que les personnes des moments conviviaux : Benjamin, Alexandre, Bayrem, Léo, Sophie, Martin et Maxence. Pour l'époque ORKAD, je remercie Anne-Lise, Aymeric, Rabin, Weerapan et ma voisine Camille, ainsi que les éphémères de l'équipe : Alexis et Christopher. Et enfin, je remercie le hors catégorie Thibault, avec qui j'ai animé le club d'algorithmie de l'université de Lille.

Passons maintenant à l'équipe pédagogique de l'université avec qui j'ai beaucoup échangé, en particulier Sophie Tison pour les cours Algorithmes et de Complexité et l'organisation des concours d'algorithmie comme le Catalyst Contest ou le Google Hash Code. Je remercie également Philippe Marquet et Yann Secq avec qui j'ai pu participer et organiser de nombreuses activités de médiation scientifique. Je remercie aussi mes étudiants, trop nombreux pour les citer, mais qui m'ont confirmé mon goût pour l'enseignement, de plus, mes enseignements ont tous été pour le moins agréables, donc merci à eux aussi.

Je remercie aussi mes amis, les meilleurs, la fine équipe de JDR et tombeur d'Escape Game : Kevin, Mélissa, Jimmy, Loise, William et Basile! Ainsi que Nessy qu'on ne voit malheureusement pas assez... Enfin, durant ces trois années j'ai eu le loisir de m'investir dans le jeu de cartes Final Fantasy, où j'ai fait la rencontre de nombreuses personnes

qui sont devenues aujourd'hui de bons amis et qui m'ont encouragé jusqu'à la fin de la thèse, je remercie donc : Cédric, Florence, Kevin, Bryan, Michael, Jose, Cécile, Perle, Quentin, Arthur, Hugues, Stéphanie, Hugo, Wim ainsi que tous les autres joueurs arrivés plus récemment... Les tournois du mardi soir, c'était le petit moment où la thèse me sortait totalement de la tête et ça faisait du bien !

Je terminerai cette page avec les remerciements de ma famille. En premier lieu, Lily présente au quotidien du début à la fin de cette thèse, qui m'a soutenu et réconforté dans les moments difficiles, et je ne lui dis jamais assez merci pour cela, donc je le dis ici, merci ! Ensuite, je remercie ma famille et ma belle-famille : ma mère, mon père, mes soeurs, mes grands-parents qui me soutiennent depuis ma naissance et m'ont encouragé dans mes choix scolaires et professionnels. Je remercie aussi Lucette, ma belle-mère et toute la belle-famille qui m'ont aussi apporté de nombreux mots d'encouragement.

Bref, merci à tous pour l'aboutissement de ces trois années de thèse !

Résumé

Les problèmes d'optimisation combinatoire de grandes tailles sont en général difficiles à résoudre de façon optimale due à des temps de calcul trop élevés. Afin de pallier ce problème, des algorithmes d'approximation tels que les heuristiques et les métaheuristiques sont utilisés pour trouver rapidement des solutions approchées de bonne qualité. Les heuristiques sont des approches développées spécifiquement pour un problème et permettent d'obtenir des solutions très rapidement. Les métaheuristiques sont des approches génériques, indépendantes du problème, permettant de trouver des solutions de bonne qualité. Ces approches présentent chacune leurs avantages et inconvénients. Nous proposons dans ce mémoire de tirer parti des avantages de ces deux approches, c'est-à-dire intégrer des connaissances spécifiques à un problème, telles que celles utilisées dans les heuristiques, dans les mécanismes génériques des métaheuristiques, afin de concevoir des nouvelles approches efficaces. Ainsi, dans ce mémoire, nous passons d'abord en revue dans la littérature les approches avec intégration de connaissances afin de proposer une taxonomie de classification de ces approches. Puis nous nous focalisons sur l'intégration de connaissances pour deux problèmes différents : le problème d'ordonnancement de type Flowshop sans temps d'attente, et le problème de sélection d'attributs en classification. Enfin, nous étudions dans ces problèmes l'impact de l'intégration de connaissances dans différents mécanismes des métaheuristiques : l'initialisation, l'opérateur de voisinage et la sélection du voisinage.

Abstract

Large-scale combinatorial optimization problems are generally hard to solve optimally due to expensive computation times. In order to tackle this problem, approximation algorithms such as heuristics and metaheuristics are used to quickly find approximate solutions. Heuristics are problem-specific approaches that provide solutions very quickly. Metaheuristics are generic approaches to finding solutions with good quality for several problems. Each of these approaches has its advantages and disadvantages. We propose in this thesis to use the advantages of these two approaches, that is to say, to integrate specific knowledge associated to a problem as heuristics do, within the mechanisms of metaheuristics in order to design new effective approaches. In this thesis report, we first review knowledge integration approaches in the literature to propose a taxonomy for a classification of these approaches. Then we focus on the integration of knowledge in two different problems : the No-Wait Flowshop Scheduling Problem, and the Feature Selection Problem in a classification context. Finally we study the impact of the integration of knowledge on these problems with different metaheuristic's mechanisms, that is initialization procedure, neighborhood operator and neighborhood selection.

Table des matières

Liste des figures

Liste des tableaux

Liste des Algorithmes

Introduction

Cette thèse se place dans le domaine de l'optimisation combinatoire et de la recherche opérationnelle. L'objectif de ces travaux est de proposer de l'intégration de connaissances dans des métaheuristiques afin de rendre ces méthodes d'optimisation plus rapides et plus efficaces. Cette thèse a été menée au sein de l'équipe ORKAD [1] du laboratoire CRIStAL [2]. Cette équipe a pour objectif de résoudre des problèmes d'optimisation combinatoire (mono-objectif et multi-objectif) de grande taille en utilisant des techniques d'extraction de connaissances.

L'optimisation combinatoire est un domaine très actif de la recherche en informatique, car il existe de nombreux problèmes de la vie quotidienne qui peuvent se modéliser comme un problème d'optimisation combinatoire. Résoudre un problème d'optimisation consiste à trouver la meilleure solution possible (ou un ensemble de meilleures solutions dans le cadre multi-objectif) dans un espace fini de solutions réalisables. Une "meilleure solution" est définie par une fonction objectif qui est soit à minimiser, soit à maximiser. Les problèmes étudiés sont généralement NP-difficiles et les méthodes exactes deviennent inutilisables lorsque les problèmes sont de grandes tailles, car trop coûteuses en temps de calcul. Pour pallier cette difficulté, des méthodes approchées ont été développées : les heuristiques et les métaheuristiques. Ces approches permettent d'obtenir des solutions de bonne qualité, mais sans garantie d'optimalité, en un temps raisonnable. Les heuristiques sont généralement des approches gloutonnes, spécifiques à un problème, qui construisent des solutions très rapidement. Les métaheuristiques, à l'opposé des heuristiques, sont des approches génériques de résolution, c'est-à-dire qu'une même métaheuristique peut-être employée pour résoudre des problèmes bien différents.

Ces méthodes approchées ont cependant aussi des désavantages. Les heuristiques sont très spécifiques à un problème, et construisent une solution itérativement, qui, pour les cas des heuristiques gloutonnes, ne tiennent pas compte des éventuelles améliorations entre les itérations. Ce qui fait que généralement la qualité obtenue par ces approches est bien loin de la qualité optimale. Les métaheuristiques pallient ce problème en proposant des modifications itératives de la solution afin de l'améliorer jusqu'à ce qu'un critère d'arrêt soit atteint (temps, optimum local ...). Cependant ces méthodes étant généralistes, elles perdent toute exploitation des données spécifiques à un problème donné. Dans ce mémoire, nous proposons d'utiliser les avantages des deux

1. Operational Research, Knowledge And Data
2. Centre de Recherche en Informatique, Signal et Automatique de Lille, UMR 9189, Université de Lille

approches : exploiter la connaissance spécifique au problème utilisé dans les heuristiques et les intégrer dans les métaheuristiques pour combiner la force d'exploration de l'espace de recherche de ces dernières et la connaissance utilisée dans les heuristiques. La Figure 1 donne une représentation de la répartition de ces approches en fonction du temps et de la qualité. En orange, notre objectif, exploiter la connaissance pour créer des nouvelles approches rapides et efficaces.

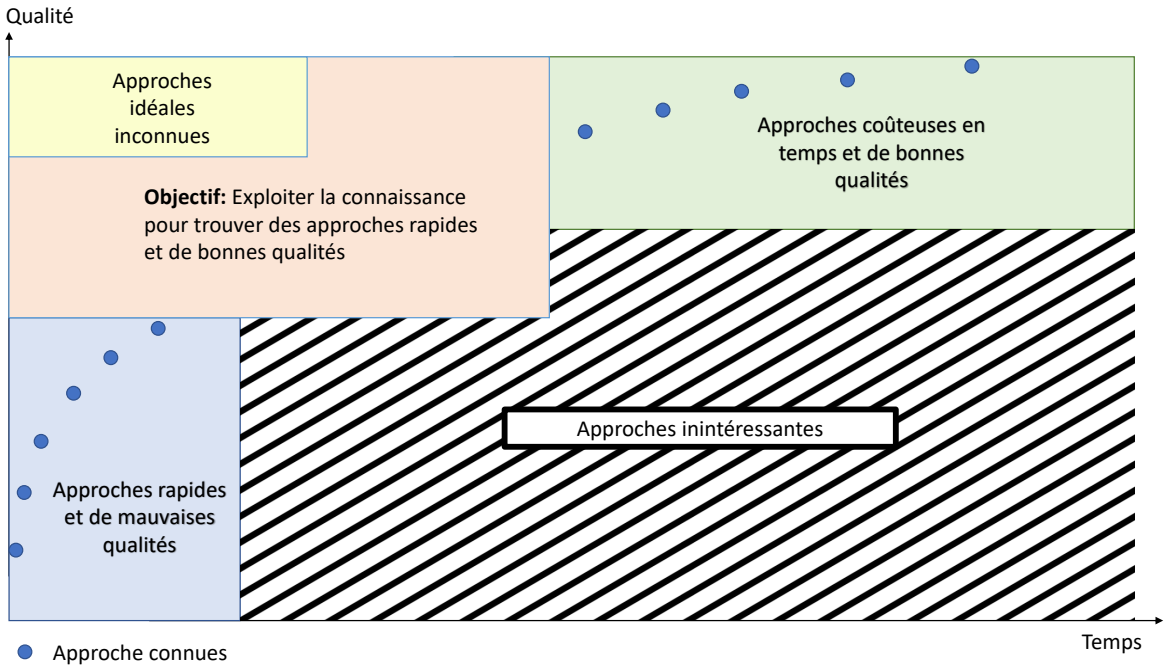


FIGURE 1 – Représentation de la répartition en qualité et en temps des approches d'aujourd'hui et l'objectif de ces travaux de thèses.

Ainsi, nous avons étudié les connaissances que nous pouvions extraire d'un problème d'optimisation pour deux problèmes, un issu d'un problème d'ordonnancement, le second d'un problème d'extraction de connaissances.

Avant d'attaquer ces problèmes, nous avons effectué une revue de la littérature afin d'étudier des méthodes d'extraction et d'intégration de connaissances déjà existantes. Cela nous a permis de définir ce qu'est *l'intégration de connaissances* au sens propre du terme. Pour intégrer de la connaissance dans une métaheuristique, il faut définir : *l'extraction, la mémorisation* et *l'exploitation* de cette connaissance. Ainsi, pour chacun des problèmes que nous avons étudiés, nous avions besoin de définir quelle est la connaissance à extraire, sous quelle forme nous devions la mémoriser (information complète, information partielle ...) et enfin où et comment nous allions l'exploiter.

Nous avons donc étudié un premier problème, le problème d'ordonnancement du Flowshop sans temps d'attente, afin de déterminer quelles connaissances nous pouvions extraire, en élaborant dans un premier temps des heuristiques constructives efficaces exploitant la connaissance du problème. Puis nous avons utilisé cette connaissance dans une métaheuristique afin de modifier le paysage de l'espace de recherche

pour atteindre des solutions de bonne qualité qui étaient jusqu'alors inaccessibles sans l'utilisation de la connaissance. Le second cas d'étude, la sélection d'attributs, est un problème où l'évaluation d'une solution est très couteuse en temps de calcul. Nous avons donc utilisé l'intégration de connaissances pour la réduction du voisinage, afin d'évaluer uniquement des solutions de bonne qualité.

Le manuscrit présentant l'ensemble de ces travaux est divisé en 5 chapitres.

Plan du mémoire

Chapitre 1

Ce premier chapitre décrit le contexte des travaux de cette thèse. Ainsi, l'optimisation combinatoire et les méthodes approchées sont définies. Puis nous présenterons les deux problèmes étudiés dans ces travaux ainsi qu'un état de la littérature pour ces problèmes.

Chapitre 2

Le second chapitre s'intéresse à l'intégration de connaissances qui est l'objectif central de nos travaux. Nous définissons donc dans un premier temps ce qu'est l'intégration de connaissances. Puis, nous montrons que pour réaliser de l'intégration de connaissances, il est nécessaire de définir trois composantes : l'extraction, qui définit où et comment nous pouvons trouver la connaissance, la mémorisation, qui définit ce qui doit être gardé ou non, certaines informations étant de très grandes tailles, il n'est pas possible de tout stocker, une sélection de l'information est ainsi nécessaire, et enfin la dernière composante, l'extraction, qui définit où et comment l'information sauvegardée dans la mémoire est exploitée. Pour cela, nous proposons une revue de la littérature complète sur ce domaine, ce qui nous a permis de définir une taxonomie des méthodes d'intégration de connaissances.

Chapitre 3

Dans le troisième chapitre, nous nous intéressons à notre premier cas d'étude, le problème d'ordonnancement de type Flowshop sans temps d'attente. Nous réalisons ainsi une étude du problème afin de définir la connaissance que nous pouvons en extraire. Cette connaissance est ensuite utilisée pour proposer deux heuristiques constructives. Les deux approches proposées utilisent une connaissance différente pour chacune d'entre elles. La première est une connaissance génotypique, c'est-à-dire, obtenue par l'étude de la structure des bonnes solutions de ce problème, la seconde est une connaissance phénotypique, c'est-à-dire, obtenue par l'étude des mesures obtenues par les bonnes solutions.

Chapitre 4

Le quatrième chapitre a été dirigé par l'étude des connaissances du chapitre précédent. Nous avons souhaité exploiter davantage la connaissance génotypique que nous avions acquise pour l'extraction de connaissances. Grâce à cette étude, nous avions pu mémoriser des sous-séquences prometteuses pour ce problème. Ensuite, l'utilisation de ces sous-séquences prometteuses a permis la modification du paysage de l'espace de recherche, ce qui a permis à l'approche utilisée d'atteindre des solutions qui étaient jusqu'alors inaccessibles par l'approche initiale. Cela nous a donc permis de montrer l'intérêt de l'apport de la connaissance dans les métaheuristiques.

Chapitre 5

Le cinquième chapitre propose une intégration de connaissances dynamique. C'est-à-dire que la mémoire va évoluer au cours du temps. Le cas d'étude de ce chapitre est le problème de sélection d'attributs. Nous nous plaçons donc dans un contexte d'évaluation coûteuse où il n'est pas possible d'explorer tout le voisinage d'une solution. De ce fait l'intégration de connaissances permet de choisir les meilleures solutions voisines potentielles afin de réduire le nombre d'évaluations par itération. La mémoire va quant à elle, être mise à jour avec les meilleures solutions trouvées contrairement aux chapitres précédents où nous utilisions une mémoire fixée a priori. Pour finir, nous montrons l'apport de l'intégration de connaissances selon deux aspects, l'aspect optimisation combinatoire et l'aspect *data-mining*.

Chapitre 1

Contexte

Sommaire

L'objectif de la thèse est de réaliser de l'intégration de connaissances dans les méthodes d'optimisation. Pour appréhender ces travaux de thèse, il est nécessaire de réaliser une présentation du domaine de l'optimisation combinatoire et des enjeux que cela représente. L'optimisation combinatoire s'applique en effet à différents problèmes du quotidien. Deux de ces problèmes seront étudiés dans ces travaux de recherche, et seront donc présentés en détail dans ce chapitre.

1.1 L'optimisation combinatoire

L'*Optimisation Combinatoire* est présent dans de nombreux domaines de l'informatique, ainsi que dans d'autres branches où l'optimisation est nécessaire, telles que la recherche opérationnelle et les mathématiques appliquées. Les problèmes d'optimisation demandent généralement de rechercher des regroupements, des affectations ou un ordonnancement, d'un ensemble discret de caractéristiques et en respectant un certain nombres de contraintes, tout en optimisant une fonction objectif. Une solution d'un problème d'optimisation combinatoire est une affectation de valeurs aux variables de décision qui satisfait les contraintes du problème. La qualité de la solution est définie par une fonction objectif qui peut être à minimiser ou à maximiser. Le but étant de trouver la meilleure solution possible. Afin de simplifier la lecture de ce chapitre, nous allons considérer être dans un contexte de minimisation, les définitions et raisonnements s'appliquant de la même façon dans un contexte de maximisation.

Donnons maintenant une définition plus formelle d'un problème d'optimisation combinatoire : l'optimisation combinatoire consiste à trouver une solution optimale dans un ensemble discret de solutions réalisables. La solution optimale est définie en accord avec une fonction objectif. La fonction objectif peut être minimisée ou maximisée. Ainsi un problème d'optimisation peut être défini par un espace de recherche Ω et une fonction objectif $f : \Omega \rightarrow \mathbb{R}$. Le but étant de trouver, dans un contexte de minimisation de la fonction objectif, $s^* \in \Omega$ tel que :

$$s^* = argmin_{s \in \Omega} f(s) \tag{1.1}$$

Dans un contexte de maximisation, l'équation 1.1 consiste alors à maximiser la fonction f. Dans un problème d'optimisation, la solution optimale s^* appelée aussi *optimum global* est définie tel que :

$$\forall s \in \Omega, f(s*) \leq f(s) \tag{1.2}$$

On définit une *instance* d'un problème d'optimisation combinatoire comme un ensemble de données particulières d'un problème à résoudre.

La plupart des problèmes d'optimisation combinatoire font partie de la classe des problèmes dits NP-difficile. C'est-à-dire qu'il n'existe pas d'algorithme efficace permettant de trouver une solution optimale en un temps raisonnable. De ce fait, on définit deux types de méthodes pour résoudre les problèmes d'optimisation combinatoire : les méthodes exactes et les méthodes approchées.

— **Les méthodes exactes** garantissent une solution optimale, mais deviennent inutilisables lorsque la taille des instances devient trop grande suite à un coût de calcul trop important.

— **Les méthodes approchées** permettent de trouver une solution de bonne qualité en un temps raisonnable, mais ne garantissent plus l'optimalité de la solution. Ce sont ces méthodes qui seront abordées dans cette thèse.

1.2 Méthodes de résolution approchées

Ces méthodes approchées de résolution de problèmes d'optimisation combinatoire se classent en deux grandes catégories : les heuristiques et les méta-heuristiques.

1.2.1 Les heuristiques

Les heuristiques sont des méthodes qui exploitent la structure d'un problème considéré afin de trouver une solution approchée de bonne qualité en un temps raisonnable, *i.e.* aussi faible que possible. En d'autres termes, elles permettent généralement de trouver une solution approchée à un problème de classe NP en un temps polynomial. Ces heuristiques peuvent être déterministes ou stochastiques.

Les heuristiques les plus simples qu'on retrouve dans la littérature sont les heuristiques gloutonnes (DEVORE et TEMLYAKOV [1996]). Ce type d'heuristique fait une succession de choix en fonction de certaines caractéristiques (Exemple : choisir la ville la plus proche parmi un ensemble de villes encore non parcourues), jusqu'à ce que la solution soit réalisable, et cela, sans retour en arrière possible.

Bien que le principe soit assez générique, il doit cependant être adapté en fonction du problème, car ces heuristiques sont généralement guidées par des caractéristiques spécifiques au problème considéré et en sont donc totalement dépendantes.

1.2.2 Les méta-heuristiques

Les méta-heuristiques sont des méthodes généralisées pour la résolution de problèmes NP. Ainsi, à l'inverse des heuristiques, elles sont indépendantes du problème considéré. De plus, des preuves de convergence vers la solution optimale ont été établies (FAIGLE et KERN [1992]) montrant que la probabilité de trouver la solution optimale augmente si on laisse un temps infini. Bien qu'en pratique il n'est pas possible de laisser un temps infini, les méta-heuristiques ont montré de très bonnes performances sur un temps fini.

Dans les méta-heuristiques nous retrouvons deux grandes stratégies : l'*intensification* et la *diversification*. L'intensification consiste à sélectionner un sous-espace de recherche prometteur et d'intensifier la recherche de solutions dans ce sous-espace. La diversification, quant à elle, permet à la méthode d'identifier des zones de l'espace non encore explorées afin de les intensifier pour trouver d'éventuelles meilleures solutions.

Basées sur ces stratégies, les méta-heuristiques recherchent donc le bon compromis entre l'intensification et la diversification. Nous pouvons donc diviser les méta-heuristiques en trois catégories. (*i*) Les méta-heuristiques *évolutionnaires* : approche inspirée de la nature qui privilégient la diversification à l'intensification. (*ii*) Les autres méta-heuristiques *inspirées de la nature* : compromis entre diversification et intensification. (*iii*) Les méta-heuristiques *à base de voisinages* : privilégient l'intensification à la diversification.

1.2.2.1 Méta-heuristiques évolutionnaires

Les méta-heuristiques évolutionnaires sont issues de la théorie de l'évolution de Charles Darwin. Le principe sur lequel reposent ces approches est que les individus les mieux adaptés à leur environnement survivent et peuvent se reproduire, alors que les plus faibles disparaissent. L'analogie avec l'optimisation combinatoire se fait facilement : les individus les mieux adaptés à leur environnement sont assimilés à des solutions de bonne qualité.

Ainsi, les méthodes évolutionnaires vont faire évoluer un ensemble d'*individus* (*i.e.* les solutions) appelé *population*. Les individus seront représentés par leur *génotype* (*i.e.* informations caractérisant un individu) et leur *phénotype* (*i.e.* la qualité d'un individu).

Cette population va évoluer pendant plusieurs *générations* afin de créer les meilleurs individus possibles. Une génération est décomposée en plusieurs étapes qui sont les suivantes :

- **Évaluation :** Le phénotype de chaque individu est évalué à partir de son génotype.
- **Reproduction :** Une partie des individus est choisie pour générer des *enfants* (*offsprings* en anglais). Deux étapes sont alors réalisées :
  - **Croisement :** Les individus choisis donnent chacun une partie de leur génotype pour générer un enfant partageant les gènes de ses parents.
  - **Mutation :** Certains enfants subissent une mutation de leur génotype avec une probabilité généralement faible.
- **Sélection :** Une sélection est effectuée parmi les parents et les enfants afin de ne garder que les meilleurs pour la prochaine génération. La sélection la plus simple consiste à garder uniquement les meilleurs individus de la population, *i.e.* ceux avec le meilleur phénotype.

Parmi les approches évolutionnaires, les stratégies les plus connues et les plus répandues sont les *algorithmes génétiques* initiés par HOLLAND [1992] qui suivent rigoureusement les principes énoncés précédemment. Mais d'autres stratégies existent aussi telles que les *algorithmes à évolution différentielle* (*Differential Evolution*) de STORN et PRICE [1997] ou les *algorithmes à estimation de distribution* (*Estimation of Distribution Algorithms* ou *EDA*) proposés par MUHLENBEIN et PAASS [1996].

1.2.2.2 Autres méta-heuristiques inspirées de la nature

Tout comme les méthodes évolutionnaires, d'autres approches inspirées de la nature existent. La particularité de ces approches est qu'elles utilisent une population d'*agents*

agissant indépendamment des autres au lieu d'une population d'individus évoluant ensemble. Les agents vont évoluer dans un système gouverné par un ensemble de règles nommées *système multi-agent* par FERBER [1999]. Ce système aboutit à terme vers un comportement pour l'ensemble des agents, où le comportement sera la convergence vers une solution d'un problème d'optimisation. Ainsi ces approches vont commencer par une grande diversification du comportement des agents pour finir par intensifier un comportement en particulier, *i.e.* intensifier une zone de l'espace de recherche.

Les méta-heuristiques inspirées de la nature les plus répandues sont les *colonies de fourmis* et les *essaims de particules*.

Les algorithmes de colonie de fourmis (*Ant Colony Optimization* ou *ACO)* proposés par DORIGO et GAMBARDELLA [1997] utilise la notion de système multi-agent où chaque agent (solution) est une fourmi. La fourmi construit une solution de manière probabiliste, en sélectionnant les attributs de la solution proportionnellement à la quantité de *phéromones* déposées sur les différents composants possibles. La fourmi dépose à chaque pas une quantité de phéromones proportionnelle à la qualité de solution. Ainsi plus un chemin a de phéromones et plus ce chemin est attractif pour les autres fourmis. Les phéromones subissent un phénomène d'évaporation à chaque itération, ce qui retire une partie des phéromones présentes dans l'environnement. Ainsi les chemins peu explorés ne sont plus privilégiés au cours de la recherche.

On a donc au début de la recherche, une exploration aléatoire des fourmis qui, au fur et à mesure des itérations avec le phénomène d'évaporation de l'intensification suite au dépôt de phéromones, font que seuls les chemins les plus intéressants seront considérés par les fourmis, et les moins intéressants seront ignorés.

Ces algorithmes sont généralement très efficaces sur des problèmes pouvant se représenter sous forme de graphes, comme les problèmes de type voyageur de commerce.

Les algorithmes à essaim de particules sont des algorithmes inspirées de la nature où les agents d'une population sont appelés *particules*. Chaque particule représente une solution d'un problème et possède une position x et une vitesse v. Chaque particule possède en plus une mémoire qui lui permet de connaitre la meilleure position qu'elles ont trouvée au cours de leur vie, notée p_{best} pour "*particule best*", ainsi que la meilleure position trouvée par les autres particules, notée g_{best} pour "*general best*". À chaque itération, les particules se déplacent dans l'espace de recherche en fonction d'un vecteur calculé comme une somme pondérée de sa vitesse, de p_{best} et de g_{best}. Sa nouvelle vitesse est alors mise à jour suivant l'équation 1.3, et sa nouvelle position suivant l'équation 1.4.

$$v_{i+1} = \omega * v_i + b_1 * (p_{best} - x_i) + b_2 * (g_{best} - x_i) \tag{1.3}$$

$$x_{i+1} = x_i + v_{i+1} \tag{1.4}$$

Où ω est le paramètre d'inertie, b_1 un paramètre influant sur la composante cognitive, *i.e.* la confiance en la meilleure solution vue par l'agent et b_2 un paramètre influant sur la composante sociale, *i.e.* la confiance au groupe.

Le point fort de cette approche est l'utilisation d'une mémoire partagée qui permet la création d'un réseau social. De plus, plusieurs stratégies d'utilisation de la mémoire partagée sont possibles : mémoire centralisée, distribuée, en étoile, etc. ce qui modifie le comportement des particules.

1.2.2.3 Méta-heuristiques à base de voisinages

Les méta-heuristiques à voisinage, appelées aussi recherches locales ou algorithmes de descente, démarrent d'une solution initiale et l'améliorent itérativement à l'aide d'un opérateur de *voisinage*.

Nous définissons ainsi le voisinage ν d'une solution s comme l'ensemble des solutions voisines de s. De plus, nous définissons aussi la notion d'*optimum local* comme une solution $s^* \in \Omega$ telle que toutes les solutions présentes dans son voisinage soient de moins bonne qualité. L'équation 1.5 formalise la notion d'optimum local :

$$\forall s \in \nu(s^*), f(s^*) \leq f(s) \tag{1.5}$$

Ces différentes notions sont illustrées sur la figure 1.1. Cette figure donne une représentation simplifiée d'une solution qui possède deux voisins (elle peut évoluer soit vers la gauche, soit vers la droite). La courbe représente l'évolution de la qualité de la solution en passant de proche en proche. Sur cette figure, nous avons représenté un minimum local, tel qu'il n'existe pas de solution voisine avec une meilleure qualité, un minimum global, tel qu'il n'existe pas de solution de plus petite qualité et un plateau tel que les solutions voisines sont de qualité équivalente.

Un grand nombre de recherches locales existent dans la littérature, mais les méthodes les plus simples sont les algorithmes de descente, aussi appelés *Hill Climbing* (Papadimitriou et Steiglitz [1982]).

Les algorithmes de descente consistent, à partir d'une solution initiale, de choisir à chaque étape un voisin qui améliore strictement la fonction objectif. Pour cela, il existe plusieurs stratégies possibles, les deux plus connues sont celle du *premier améliorant* (*first improvement*) qui choisit aléatoirement le premier voisin améliorant, et celle du *meilleur voisin* (*best improvement*) qui choisit le voisin améliorant le plus la fonction objectif. Récemment Basseur et Goëffon [2014] ont proposé aussi la stratégie du voisin le *moins améliorant* qui sélectionne parmi tous les voisins améliorants celui qui améliore le moins la fonction objectif. Pour chacune des stratégies, le critère d'arrêt est atteint lorsqu'un optimum local est rencontré. Les différentes étapes sont définies dans l'algorithme 1.1.

L'algorithme de descente est un algorithme très simple à mettre en œuvre et très rapide. Néanmoins son principal inconvénient est qu'il s'arrête dès qu'un optimum local est atteint. Pour parer à cela, il existe différentes stratégies afin de poursuivre la re-

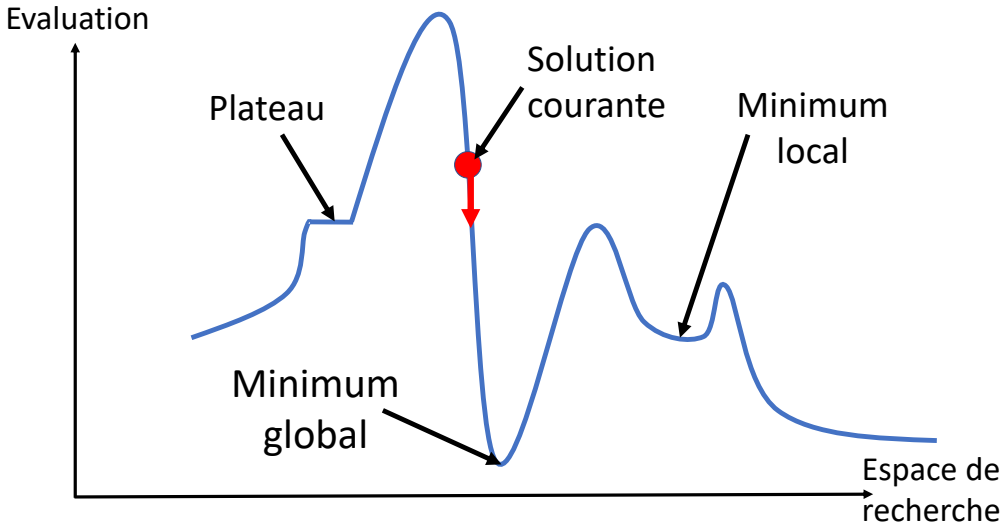


FIGURE 1.1 – Représentation d'un optimum local, d'un optimum global et d'une solution courante

Algorithme 1.1 : Algorithme de descente

Entrées : s_0 la solution initiale
Sorties : s la solution finale
$s = s_0$;
répéter
  $s' \in v(s)$ telle que $f(s') \leq f(s)$;
  $s = s'$;
jusqu'à *plus d'amélioration possible*;
retourner s

cherche lorsqu'un optimum local est atteint, c'est le cas des *recherches locales itérées*, des *recherches tabou* et des *descentes à voisinage variable*.

Les recherches locales itérées (*Iterated Local Search* ou *ILS*) comme leur nom l'indique, itèrent une recherche locale. Ainsi lorsqu'un optimum local est atteint une perturbation est appliquée sur la solution courante avant de relancer une nouvelle recherche locale. Comme la solution est perturbée à chaque optimum local, un nouveau critère d'arrêt doit être défini. Les différents critères d'arrêts les plus couramment utilisés sont le temps d'exécution, le nombre d'itérations et le nombre d'évaluations total. L'algorithme 1.2 présente les différentes étapes.

La perturbation quant à elle peut consister à appliquer plusieurs fois l'opérateur de voisinage en autorisant la dégradation de la qualité, ou bien de complètement redémarrer d'une solution choisie aléatoirement.

Algorithme 1.2 : Recherche locale itérée

Entrées : s_0 la solution initiale
Sorties : s la solution finale
$s = s_0$;
répéter
  s = rechercheLocale(s);
  si $f(s) \leq f(s^*)$ **alors**
    $s^* = s$;
  s = perturbation(s);
jusqu'à *critère d'arrêt atteint*;
retourner s

Les recherches tabou introduites par GLOVER et LAGUNA [1998], les recherches tabou sont des métaheuristiques de descente qui permettent d'échapper aux optima locaux grâce à l'utilisation d'une mémoire. En effet, la descente évolue de solution voisine en solution voisine améliorante jusqu'à la découverte d'un optimum local, c'est-à-dire, une solution sans solution voisine améliorante. Dans la recherche tabou, la méthode choisit toujours la meilleure solution voisine possible, et ce, même si elle dégrade la solution courante. Ainsi l'algorithme peut continuer d'évoluer même après avoir atteint un optimum local. Cependant, pour éviter de retomber dans un optimum local ou une solution voisine améliorante déjà visitée, la méthode va interdire de réaliser un déplacement vers les solutions récemment visitées. Ces interdictions sont gérées par une liste tabou. Cette liste tabou permet ainsi d'interdire la visite d'une solution déjà visitée ou juste d'interdire une modification locale faite récemment et permet donc à la méthode de s'éloigner de cette zone de l'espace de recherche. Les différentes étapes sont définies dans l'algorithme 1.3.

Algorithme 1.3 : Recherche tabou

Entrées : s_0 la solution initiale, L la liste tabou
Sorties : s^* la meilleure solution rencontrée
$s = s_0$;
$s^* = s$;
L = { };
répéter
  $s' \in \nu(s)$ telle que $f(s') \leq f(s)$ et $s' \notin$ L;
  si $f(s) \leq f(s^*)$ **alors**
    $s^* = s'$;
  Mise à jour de L;
  $s = s'$;
jusqu'à *critère d'arrêt atteint*;
retourner s^*

Cette méthode nécessite cependant un paramétrage de l'algorithme (caractéristiques

du mouvement tabou, longueur de liste, ...) qui peut être difficile en fonction du problème à considérer. De même, que rendant le critère d'arrêt naturel irréalisable, il est nécessaire de définir un nouveau critère d'arrêt (le temps, le nombre d'évaluations, etc.).

1.3 Problèmes étudiés

Parmi la grande variété des problèmes d'optimisation, nous nous sommes intéressés particulièrement à certains d'entre eux : le problème d'ordonnancement de type flowshop sans temps d'attente et le problème de sélection d'attributs dans un contexte de classification supervisée. Dans cette section, nous présenterons et ferons un état de l'art pour chacun de ces problèmes.

1.3.1 Problème du No-wait Flowshop

Parmi les nombreux problèmes d'ordonnancement, le Flowshop Scheduling Problem (FSP) consiste à planifier un ensemble de n tâches (ou jobs) $\{J_1, \ldots, J_n\}$, sur un ensemble de m machines $\{M_1, \ldots, M_m\}$. Plusieurs versions existent en fonction des contraintes spécifiques pouvant être considérées. Dans ces travaux de thèse, nous nous intéressons à la version sans temps d'attente de ce problème (No-Wait Flowshop Scheduling Problem - NWFSP).

1.3.1.1 Définition du problème

Le problème d'ordonnancement sans temps d'attente (NWFSP) est une variante du très connue problème PFSP (Permutation Flowshop Scheduling Problem), dans lequel aucun temps d'attente n'est autorisé entre le traitement d'une même tâche sur des machines successives [RÖCK, 1984]. Malgré cette contrainte, le NWFSP reste un problème NP-difficile.

Plus formellement, le NWFSP peut être défini comme suit. Soit J un ensemble de n tâches qui doivent être traitées sur un ensemble de m machines ordonnées; Les machines sont des ressources critiques qui ne peuvent traiter qu'une seule tâche à la fois. Une tâche J_i est composée de m opérations $\{o_{i,1}, \ldots, o_{i,m}\}$ pour les m machines respectivement. Un temps de traitement $p_{i,j}$ est associé à chaque opération $t_{i,j}$.

Comme pour le PFSP, la séquence des opérations est la même sur chaque machine, par conséquent, une solution du NWFSP est communément représentée par une permutation $\pi = \{\pi_1, \ldots, \pi_n\}$ où π_1 est la première tâche planifiée et π_n la dernière. La Figure 1.2 montre la représentation d'une solution du NWFSP dans un diagramme de Gantt. Sur cette figure, nous pouvons observer que les tâches s'exécutent dans le même ordre et sans temps d'attente entre les opérations d'une même tâche sur les différentes machines.

Notons que pour une instance de taille n, le nombre de permutations est de $n!$, soit $n!$ solutions.

Un grand nombre d'objectifs ont été définis dans les travaux de la littérature, cependant nous ne détaillerons que les objectifs étudiés pendant ces travaux de thèse. Les

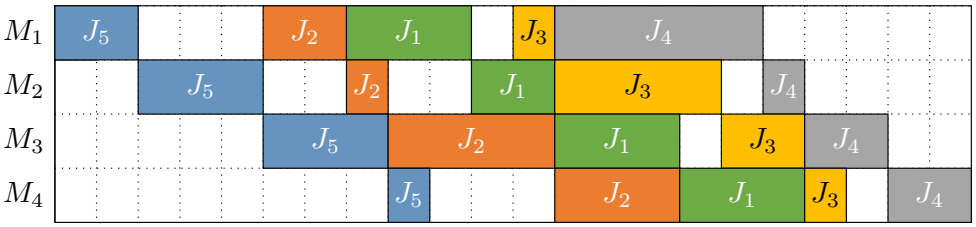


FIGURE 1.2 – Exemple d'une solution d'un problème de No-Wait Flowshop de permutation où 5 tâches sont ordonnancées sur quatre machines dans l'ordre J_5 - J_2 - J_1 - J_3 - J_4. Les données de l'instance sont disponibles dans l'annexe A.1

objectifs étudiés sont détaillés ci-dessous, notez que les formules suivantes utilisent le délai (noté $d_{i,j}$), une notion spécifique à ce problème simplifiant le calcul des objectifs. Cette notion sera détaillée dans la sous-section 1.3.1.3.

— **Le temps maximum de complétion (*Makespan*)** correspond au temps requis pour terminer toutes les tâches, *i.e.* le temps de fin de la dernière tâche, noté C_{max}. En considérant n tâches, le Makespan est calculé avec l'équation 1.6

$$C_{max} = C_n = \sum_{k=2}^{n} d_{\sigma_{k-1},\sigma_k} + \sum_{j=1}^{m} p_{\sigma_n,j} \tag{1.6}$$

La complexité de l'équation 1.6 est $O(n)$. PAN et collab. [2007a] ont montré que lors de l'application d'un opérateur de voisinage type insertion ou échange cette complexité peut être réduite de $O(n)$ à $O(1)$.

— **Le temps total de complétion (*Flowtime*)** à l'inverse du Makespan où seul le temps de complétion de la dernière tâche est considéré, le Flowtime calcule la somme de tous les temps de complétion. Cet objectif a pour but de terminer le plus de tâches possible en un temps minimal. Le Flowtime peut être calculé avec l'équation 1.7.

$$F = \sum_{i=1}^{n} C_i = \sum_{i=2}^{n} \left((n-i+1) * d_{\sigma_i,\sigma_{i+1}} + \sum_{j=1}^{m} p_{\sigma_n,j} \right) \tag{1.7}$$

La complexité de l'équation 1.7 est $O(n * m)$. Tout comme pour le makespan, PAN et collab. [2008] ont proposé des méthodes plus rapides pour calculer le Flowtime pour les opérateurs de voisinage insertion et échange.

1.3.1.2 Complexité du No-Wait Flowshop

WISMER [1972] a montré que le problème d'ordonnancement du No-wait Flowshop pouvait être formulé comme un problème de Voyageur du Commerce asymétrique (*Asymetric Traveling Salesman Problem (ATSP)*. Ainsi, pour deux machines, la matrice de distance de cet ATSP a une structure combinatoire spéciale, ce qui a permis à la méthode de GILMORE et GOMORY [1964] de résoudre le No-Wait Flowshop à deux machines avec

une complexité en temps de O($nlogn$). Enfin, RÖCK [1984] a prouvé que pour trois machines ou plus, le No-Wait Flowshop est NP-difficile.

Cependant, à l'inverse du problème de type Flowshop classique, le NWFSP possède une caractéristique qui permet de réduire le temps de calcul de la fonction objectif d'une solution.

1.3.1.3 Spécificités du No-Wait Flowshop

La non existence du temps d'attente entre deux opérations d'une même tâche implique que le décalage entre deux tâches consécutives est constant, peu importe la position de ces deux tâches dans la solution.

BERTOLISSI [2000] a ainsi défini la matrice de délai indiquant le délai ($d_{i,j}$) entre le démarrage d'une tâche i et le démarrage d'une tâche j ($i \neq j$) suivant immédiatement i sur la première machine dans l'ordonnancement. Le délai $d_{i,j}$ peut être calculé avec l'équation 1.8 qui est le temps d'exécution de la première opération de la tâche i suivit du décalage nécessaire de la tâche j pour qu'elle puisse s'exécuter sans temps d'attente entre les opérations.

$$d_{i,j} = p_{i,1} + \max_{1 \leq r \leq m} \left(\sum_{h=2}^{r} p_{i,h} - \sum_{h=1}^{r-1} p_{j,h}, 0 \right) \tag{1.8}$$

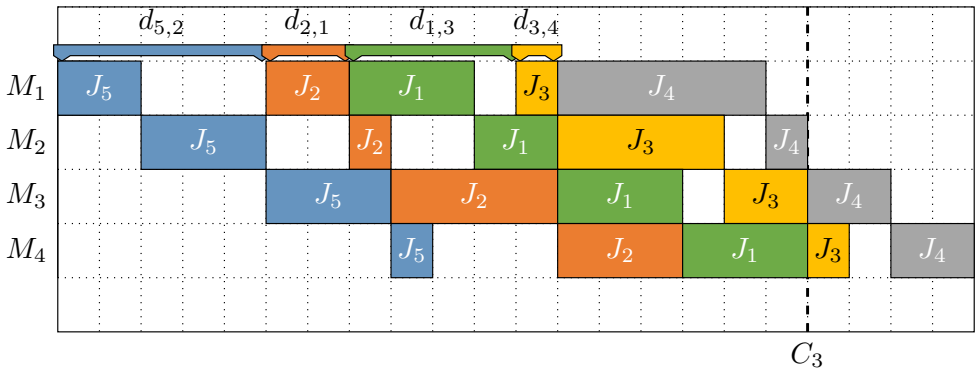


FIGURE 1.3 – Exemple d'une solution d'un problème de No-Wait Flowshop de permutation où 5 tâches sont ordonnancées sur quatre machines dans l'ordre J_5 - J_2 - J_1 - J_3 - J_4 et où les délais sont représentés ainsi que le temps de complétion C_3.

Ainsi, le temps de complétion C_i de la $i^{ème}$ tâche d'une permutation est défini par l'équation 1.10. Cette formule consiste à calculer la somme des délais entre les tâches qui précèdent la tâche i et d'ajouter le temps d'exécution de la tâche i. Comme la première tâche à ordonnancer n'est précédée par aucune tâche, le calcul de C_1 s'obtient avec uniquement le temps d'exécution de cette tâche.

$$C_1 = \sum_{j=1}^{m} p_{\sigma_1,j} \tag{1.9}$$

$$C_i = \sum_{k=2}^{i} d_{\sigma_{k-1},\sigma_k} + \sum_{j=1}^{M} p_{\sigma_i,j}, i \in \{2,...,n\} \tag{1.10}$$

La Figure 1.3 permet une meilleure compréhension du calcul du délai et du calcul du temps de complétion. Sur cette figure, le délai $d_{i,j}$ entre deux tâches consécutives i et j est indiqué. Nous pouvons aussi observer sur cette figure, que le temps de complétion C_3 représenté (qui correspond à la fin d'exécution de la troisième tâche ordonnancée, ici J_1), se calcule grâce à la somme des délais précédant la troisième tâche : $d_{5,2} + d_{2,1}$ additionnée à la durée d'exécution de la troisième tâche (ici la somme des temps d'exécution des opérations de J_1). Soit : $(5+2)+(3+2+3+3) = 18$ (la matrice des délais de cet exemple est disponible dans l'annexe A.2).

Une seconde information à extraire de l'instance identifiée par NAGANO et ARAÚJO [2013] est la matrice de GAP qui est le temps d'attente entre deux tâches i et j sur une machine k. Cette matrice ce calcule grâce à l'équation 1.11 qui s'obtient, pour le GAP entre deux tâches i et j sur la machine k, grâce à la différence entre le temps de complétion de la tâche j sur la machine $k-1$ et le temps de complétion de la tâche i sur la machine k. Pour une meilleure compréhension, l'équation 1.11 peut se reformuler en l'équation 1.12

$$GAP^k_{i,j} = (d_{i,j} + \sum_{h=1}^{k-1} p_{j,h}) - (\sum_{h=1}^{k} p_{i,h}) \tag{1.11}$$

$$GAP^k_{i,j} = d_{i,j} + \sum_{h=1}^{k-1} (p_{j,h} - p_{i,h}) - p_{i,k} \tag{1.12}$$

Grâce à nos observations, il s'est avéré qu'il peut être utile d'avoir le GAP total entre deux tâches sur toutes les machines. Ainsi, nous définissons SGAP comme la somme des GAP pour toute les machines d'un même couple de tâches i et j :

$$SGAP(i,j) = \sum_{k=1}^{m} GAP^k_{i,j} \tag{1.13}$$

La Figure 1.4 illustre la représentation du GAP pour chaque machine entre les tâches 5 et 2. Dans cet exemple, $SGAP(5,2) = 8$.

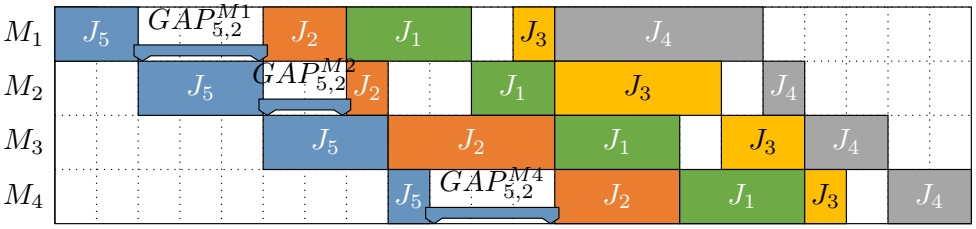


FIGURE 1.4 – Représentation du GAP entre les tâches 5 et 2.

1.3.1.4 Jeux de données

Les jeux de données habituellement utilisés dans la littérature sont les instances de TAILLARD [1993]. Ces instances proposent différentes tailles en fonction du nombre de tâches n et du nombre de machines m ($n \in \{20;50;100;200;500\}$, et $m \in \{5;10;20\}$). La durée d'exécution $p_{i,j}$ de la tâche J_i sur la machine M_j est générée selon une distribution uniforme dans l'intervalle $[1,99]$. Pour chaque combinaison $n*m$, 10 instances ont été générées.

Le tableau 1.1 référence les meilleures solutions connues pour le Makespan pour le problème d'ordonnancement du No-Wait Flowshop d'après LIN et YING [2016].
Le tableau 1.2 référence les meilleures solutions connues pour le Flowtime pour le problème d'ordonnancement du No-Wait Flowshop d'après BEWOOR et collab. [2017].

1.3.1.5 État de l'art

Suite aux nombreuses applications théoriques du No-Wait Flowshop, de nombreux algorithmes efficaces pour résoudre ce problème ont été proposés.
GILMORE et GOMORY [1964] ont résolu le cas de deux machines du problème du No-Wait Flowshop en utilisant un algorithme en $O(nlogn)$. REDDI et RAMAMOORTHY [1972] et WISMER [1972] ont été les premiers à aborder le problème du NWFSP avec plus de trois machines. Pour ce qui est des méthodes exactes, SELEN et HOTT [1986] ont proposé un modèle de programmation en nombre entier mixte pour résoudre le NWFSP multiobjectif (Makespan, Flowtime et Idle Time).

Compte tenu de la nature NP-difficile de ce problème, les méthodes de la littérature sur ce problème se sont concentrées sur le développement d'algorithmes heuristiques afin de trouver de bonnes solutions (bien que pas nécessairement optimales) à ce problème dans un temps relativement court. Les algorithmes heuristiques disponibles pour résoudre le problème du No-Wait Flowshop peuvent être classés en deux catégories principales : les heuristiques constructives et les métaheuristiques.

Plusieurs heuristiques constructives ont été proposées pour résoudre ce problème. BONNEY et GUNDRY [1976] et KING et SPACHIS [1980] ont été les pionniers des heuristiques constructives pour résoudre le problème du NWFSP. En 1976, BONNEY et GUNDRY [1976] ont développé une méthode de Slope matching (S/M) en utilisant des relations géométriques entre les temps d'exécutions cumulatifs. KING et SPACHIS [1980] ont proposé deux heuristiques à chaîne unique (*single-chain* - sc) (LBJD(sc) et LBJD(sc)*) et trois heuristiques à chaînes multiples (*multiple-chain* - mc) (LBJD(mc), MLSS(mc) et MCL(mc)). L'heuristique NEH, originellement proposée par NAWAZ et collab. [1983] pour le problème du Flowshop de permutation classique, a été très largement reprise dans la littérature du No-Wait Flowshop avec de très bonnes performances. GANGADHARAN et RAJENDRAN [1993] et RAJENDRAN [1994] ont présenté des heuristiques supplémentaires, nommées GAN-RAJ et RAJ. On peut aussi noter BIANCO et collab. [1999] et BERTOLISSI [2000] qui ont mis au point BIH et BH, deux heuristiques simples et très performantes pour ce problème. LI et collab. [2008] ont introduit une heuristique composite (CH), basée sur une méthode avec un objectif incrémental qui surpassait GAN-

Instance	Best	Instance	Best	Instance	Best	Instance	Best
20x05		50x05		100x05		200x10	
Ta01	1486	Ta31	3160	Ta61	6366	Ta91	15225
Ta02	1528	Ta32	3432	Ta62	6212	Ta92	14990
Ta03	1460	Ta33	3210	Ta63	6104	Ta93	15257
Ta04	1588	Ta34	3338	Ta64	5999	Ta94	15103
Ta05	1449	Ta35	3356	Ta65	6179	Ta95	15088
Ta06	1481	Ta36	3346	Ta66	6056	Ta96	14976
Ta07	1483	Ta37	3231	Ta67	6221	Ta97	15277
Ta08	1482	Ta38	3235	Ta68	6109	Ta98	15133
Ta09	1469	Ta39	3070	Ta69	6355	Ta99	14985
Ta10	1377	Ta40	3317	Ta70	6365	Ta100	15213
20x10		50x10		100x10		200x20	
Ta11	2044	Ta41	4274	Ta71	8055	Ta101	19531
Ta12	2166	Ta42	4177	Ta72	7853	Ta102	19942
Ta13	1940	Ta43	4099	Ta73	8016	Ta103	19759
Ta14	1811	Ta44	4399	Ta74	8328	Ta104	19759
Ta15	1933	Ta45	4322	Ta75	7936	Ta105	19697
Ta16	1892	Ta46	4289	Ta76	7773	Ta106	19826
Ta17	1963	Ta47	4420	Ta77	7846	Ta107	19946
Ta18	2057	Ta48	4318	Ta78	7880	Ta108	19872
Ta19	1973	Ta49	4155	Ta79	8131	Ta109	19784
Ta20	2051	Ta50	4283	Ta80	8092	Ta110	19768
20x20		50x20		100x20		500x20	
Ta21	2973	Ta51	6129	Ta81	10675	Ta111	46121
Ta22	2852	Ta52	5725	Ta82	10562	Ta112	46627
Ta23	3013	Ta53	5862	Ta83	10587	Ta113	46013
Ta24	3001	Ta54	5788	Ta84	10588	Ta114	46396
Ta25	3003	Ta55	5886	Ta85	10506	Ta115	46251
Ta26	2998	Ta56	5863	Ta86	10623	Ta116	46490
Ta27	3052	Ta57	5962	Ta87	10793	Ta117	46043
Ta28	2839	Ta58	5926	Ta88	10801	Ta118	46368
Ta29	3009	Ta59	5876	Ta89	10703	Ta119	46240
Ta30	2979	Ta60	5958	Ta90	10747	Ta120	46292

TABLEAU 1.1 – Qualité des solutions optimales ([LIN et YING, 2016]) pour les instances de Taillard

Instance	Best	Instance	Best	Instance	Best	Instance	Best
20x05		50x05		100x05		200x10	
Ta01	10841	Ta31	49655	Ta61	232745	Ta91	949025
Ta02	11386	Ta32	59984	Ta62	224780	Ta92	1034195
Ta03	12168	Ta33	57420	Ta63	220164	Ta93	1046902
Ta04	11438	Ta34	60470	Ta64	204798	Ta94	997214
Ta05	10204	Ta35	58590	Ta65	232933	Ta95	1034027
Ta06	11505	Ta36	57620	Ta66	232342	Ta96	1006195
Ta07	13548	Ta37	58893	Ta67	212821	Ta97	1053051
Ta08	11394	Ta38	56646	Ta68	230945	Ta98	983816
Ta09	12010	Ta39	54242	Ta69	241266	Ta99	962641
Ta10	12943	Ta40	57533	Ta70	242933	Ta100	955843
20x10		50x10		100x10		200x20	
Ta11	17395	Ta41	87114	Ta71	259015	Ta101	949025
Ta12	20603	Ta42	82820	Ta72	233285	Ta102	1245271
Ta13	15300	Ta43	64446	Ta73	249201	Ta103	1254162
Ta14	14883	Ta44	68770	Ta74	263386	Ta104	1238349
Ta15	16146	Ta45	62523	Ta75	230167	Ta105	1227214
Ta16	17899	Ta46	62005	Ta76	250354	Ta106	976118
Ta17	17667	Ta47	88750	Ta77	271318	Ta107	1243707
Ta18	19447	Ta48	67669	Ta78	230425	Ta108	983816
Ta19	20059	Ta49	67144	Ta79	250337	Ta109	962641
Ta20	21254	Ta50	61460	Ta80	254082	Ta110	955843
20x20		50x20		100x20		500x20	
Ta21	29656	Ta51	62857	Ta81	245683	Ta111	6263859
Ta22	29199	Ta52	117137	Ta82	263582	Ta112	6413646
Ta23	30158	Ta53	101810	Ta83	248834	Ta113	6437528
Ta24	31343	Ta54	100074	Ta84	239313	Ta114	6432538
Ta25	29551	Ta55	114468	Ta85	272137	Ta115	6312830
Ta26	29790	Ta56	103248	Ta86	258445	Ta116	6361035
Ta27	25506	Ta57	122880	Ta87	255264	Ta117	6539854
Ta28	28929	Ta58	119449	Ta88	384525	Ta118	6126127
Ta29	29647	Ta59	111571	Ta89	270858	Ta119	6711305
Ta30	29314	Ta60	120849	Ta90	379296	Ta120	6755722

TABLEAU 1.2 – Meilleures solutions connues ([BEWOOR et collab., 2017]) pour les instances de Taillard pour le Flowtime

RAJ et RAJ et qui avait le temps CPU le plus bas de tous les algorithmes auxquels elle s'est comparée. LAHA et CHAKRABORTY [2008] ont présenté une heuristique constructive (LC) basée sur le principe de l'insertion de groupes de tâches.

Quelques métaheuristiques ont été développées pour résoudre le problème du No-Wait Flowshop. GONAZELEZ [1995] a développé un algorithme génétique hybride (AG) qui a produit des solutions comparables aux heuristiques connues ou meilleures qu'elles. ALDOWAISAN et ALLAHVERDI [1998] ont proposé six métaheuristiques (SA, SA-1, SA-2, GEN, GEN-1, GEN-2) basées sur le recuit simulé (SA) et un algorithme génétique pour résoudre le problème. Leurs résultats ont montré que les deux meilleurs des six algorithmes étaient SA-2 et GEN-2 qui surpassaient les heuristiques GAN-RAJ et RAJ, mais nécessitaient beaucoup plus de temps de traitement. Dans la même année, SCHUSTER et FRAMINAN [2003] ont fourni deux algorithmes, dont un algorithme de recherche à voisinage variable (Variable Neighborhood Search) et un algorithme hybride utilisant un recuit simulé et un algorithme génétique (GASA). Les auteurs ont montré que les algorithmes VNS et GASA étaient supérieurs à RAJ, même s'ils n'étaient pas spécifiquement conçus pour résoudre le problème du No-Wait Flowshop.

Par la suite, GRABOWSKI et PEMPERA [2005] ont développé et comparé deux variantes d'algorithme de descente (DS, DS + M) et trois algorithmes de recherche tabou (Tabu Search - TS) (TS, TS + M, TS + MP). SCHUSTER [2006] a implémenté un algorithme de recherche tabou rapide (Fast Tabu Search - FTS) pour résoudre le problème d'ordonnancement sans temps d'attente du jobshop et du flowshop. LIU et collab. [2006] ont présenté un algorithme efficace à base d'essaims de particules hybrides (HPSO). PAN et collab. [2008, 2007a,b] ont proposé un algorithme d'essaims de particules discrètes (DPSO) et un algorithme hybride d'essaim de particules discrètes (HDPSO) ainsi qu'un algorithme amélioré de l'Iterated Greedy (IIG). QIAN et collab. [2009] ont proposé un algorithme basé sur une évolution différentielle hybride (HDE). TSENG et LIN [2010] ont proposé un algorithme génétique hybride (HGA), qui combine la force d'un algorithme génétique avec celle de la recherche locale. JARBOUI et collab. [2010] ont proposé un algorithme génétique hybride (GA-VNS) qui utilisait un VNS en tant que procédure d'amélioration dans la dernière étape de l'algorithme génétique. SAMARGHANDI et EL-MEKKAWY [2012] ont développé un algorithme hybride entre recherche tabou et essaim de particules (TS-PSO). DAVENDRA et collab. [2013] ont proposé un algorithme inspiré de la nature basé sur la "migration auto-organisée discrète" (Discret Self Organised Migrating Algorithm - DSOMA). Récemment, DING et collab. [2015] ont proposé un algorithme TMIIG (une version améliorée de l'IIG de Pan et al. par un mécanisme tabou). Plus récemment et en parallèle des ces travaux de thèse, LIN et YING [2016] ont proposé un algorithme transformant le problème du No-Wait Flowshop en un problème de Voyageur de Commerce asymétrique (ATSP) qui grâce à la programmation en nombres entiers a pu trouver les solutions exactes pour toutes les instances de Taillard.

Nous avons présenté en détail un problème d'optimisation combinatoire dans un contexte d'ordonnancement de tâches. Dans la section suivante, nous allons présenter un autre type de problème, la sélection d'attributs.

1.3.2 Problème de sélection d'attributs en classification

Aujourd'hui, dans de nombreux domaines scientifiques, nous avons besoin de superviser des systèmes de plus en plus complexes. Ces systèmes contiennent en effet des données en très grandes quantités. Le traitement de cette masse de données pose de nombreux problèmes, en particulier pour les algorithmes d'apprentissage qui construisent des modèles de prédiction pour de la classification d'individus. Une trop grande quantité de données peut donc conduire à la construction d'un modèle de mauvaise qualité ou avec du sur-apprentissage, c'est-à-dire que le modèle construit risque d'être spécifique aux données d'entrées. Cela est dû à l'existence de données non significatives qui créent du bruit et ainsi rendent le modèle incapable de faire de la prédiction sur de nouvelles données. Pour remédier à ce problème, nous utilisons la sélection d'attributs comme une phase préliminaire à la phase d'apprentissage, ce qui permet de filtrer les données pertinentes des données non pertinentes. Cette réduction des données permet donc aux méthodes d'apprentissage de découvrir de meilleurs modèles de prédiction. (Voir Figure 1.5).

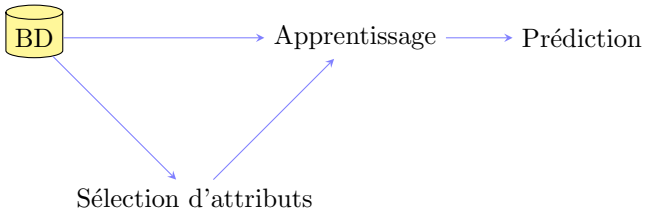


FIGURE 1.5 – Positionnement de la phase de Sélection d'attributs pour la création d'un modèle de prédiction.

1.3.2.1 Définition du problème

Dans un problème de classification, un ensemble d'observations déjà labellisé est utilisé pour apprendre un modèle de prédiction qui servira à labelliser les nouvelles observations. La sélection d'attributs a pour objectif de choisir les informations pertinentes d'un problème pour aider à la classification. Dans ce contexte, nous appellerons l'ensemble des données une *instance* qui est représentée par d observations. Chaque observation i est caractérisée par n attributs et un label. Ainsi, une instance est donc représentée par :

— Une matrice A de d lignes et n colonnes qui représentent la valeur de chaque attribut pour chaque observation.

— Un vecteur C de taille d qui représente les labels de chaque observation

D'où la représentation suivante d'une instance :

$$\mathrm{A} = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{d1} & \cdots & a_{dn} \end{bmatrix}, \mathrm{C} = \begin{bmatrix} c_1 \\ \vdots \\ c_d \end{bmatrix} \tag{1.14}$$

où $c_i \in \{1, ..., k\}$ avec k le nombre de classes.

Ces données sont ensuite divisées en deux ensembles. Le premier, appelé *ensemble d'apprentissage* qui permet à la méthode d'optimisation d'apprendre un modèle de prédiction, et le second, appelé *ensemble de validation*, est utilisé pour évaluer la robustesse du modèle de prédiction sur des nouvelles observations.

1.3.2.2 Approches de résolution

Les approches de résolution du problème de sélection d'attributs se divisent en trois type d'approches en fonction de la façon dont elles combinent l'algorithme de recherche et le classifieur. Ces trois types d'approches sont : les méthodes filtrantes, les méthodes enveloppantes et les méthodes embarquées.

Les méthodes filtrantes initiées par PUDIL et HOVOVICOVA [1998], sélectionnent les attributs indépendamment du modèle utilisé. Ainsi, les attributs les moins intéressants sont supprimés, et les attributs les plus pertinents sont conservés pour être utilisés pour la création du modèle de prédiction. Ce mécanisme est représenté par la Figure 1.6.

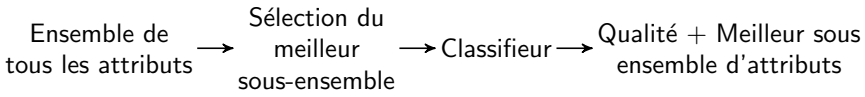


FIGURE 1.6 – Représentation d'une méthode filtrante.

Ces méthodes peuvent s'apparenter aux heuristiques des méthodes d'optimisation. Elles sont très efficaces en temps de calcul et sont de plus robustes au sur-apprentissage. Cependant, elles ne tiennent pas compte de la corrélation entre les attributs (c'est-à-dire les relations entre les attributs). Elles ont donc comme inconvénient de sélectionner des attributs redondants.

Les méthodes enveloppantes initiées par KOHAVI et JOHN [1997], évaluent des sous-ensembles d'attributs qui permettent, contrairement à la méthode précédente, de tenir compte des corrélations entre les attributs. Ainsi dans cette approche, chaque sous-ensemble d'attributs passe par un classifieur qui évalue la qualité de prédiction avec l'ensemble d'apprentissage. Ensuite, le meilleur sous-ensemble trouvé pendant la procédure de recherche est évalué sur l'ensemble de validation. Cette méthode est présentée par la Figure 1.7.

L'avantage de ce type de méthode est d'être capable de gérer les corrélations entre les attributs et donc de trouver les relations importantes entre les attributs. Cependant, ces approches génèrent souvent des modèles de prédiction avec du sur-apprentissage, c'est-à-dire que le modèle de prédiction est spécialisé sur les observations de l'ensemble d'apprentissage et peut donc réaliser de mauvaises prédictions sur des nouvelles données. De plus, le temps d'exécution de ces méthodes peut devenir coûteux en fonc-

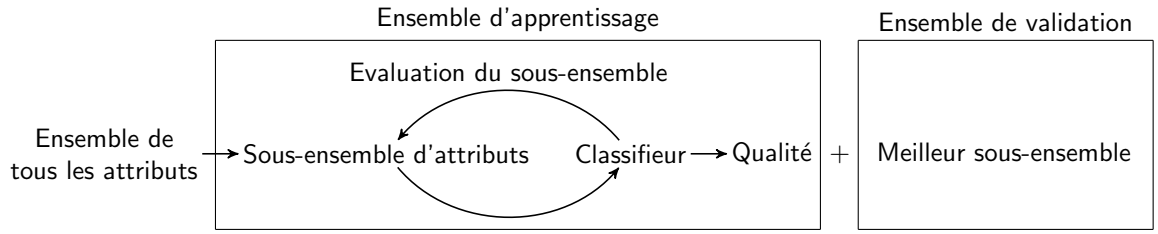


FIGURE 1.7 – Représentation d'une méthode enveloppante.

tion du classifieur utilisé et de la taille de l'instance si cette dernière possède un grand nombre d'observations et/ou d'attributs.

Les méthodes embarquées présentées par LAL et collab. [2004] sont des approches proches des méthodes enveloppantes. La différence est qu'elles utilisent les informations internes au classifieur (exemple : le vecteur de poids dans les classifieurs SVM - machines à vecteur de support) pour le processus de sélection du sous-ensemble d'attributs. Ces approches sont bien plus rapides en temps de calcul que les méthodes enveloppantes et sont aussi robustes au sur-apprentissage. Le principal inconvénient de ces approches est lié à l'utilisation de son propre classifieur, cela nécessite donc de définir a priori ce qu'est une bonne sélection, ce qui empêche une utilisation facile de ces approches.

Dans le cadre de ces travaux de thèses, nous avons souhaité faire de l'intégration de connaissances dans une méthode d'optimisation combinatoire. Ainsi la méthode qui sera évoquée dans cette thèse fait partie des approches enveloppantes.

1.3.2.3 Jeux de données

Les expérimentations de ces travaux des thèses ont été menées sur six instances de la littérature. Le Tableau 1.3 détaille les informations de ces instances.

Nom	Attributs	|T|	|V|	Référence
Schizophrenia	410	56	30	SILVA et collab. [2014]
Colon	2000	62	32	ZHU et collab. [2007]
Breast	24481	78	26	ZHU et collab. [2007]
Arcene	10000	100	100	GUYON et collab. [2006]
DNA	180	1400	600	GUERRA-SALCEDO et WHITLEY [1999]
Madelon	500	2000	600	GUYON et collab. [2004]

TABLEAU 1.3 – Description des instances : Attributs est le nombre total d'attributs, |T| est le nombre d'observations de l'ensemble d'apprentissage, |V| est le nombre d'observations de l'ensemble de validation.

Toutes ces instances sont des instances binaires (deux labels) équilibrées, c'est-à-dire qu'il y a un nombre équivalent d'observations pour chaque label.

1.3.2.4 État de l'art

De nombreuses métaheuristiques ont été développées pour la sélection d'attributs en classification. YANG et HONAVAR [1998] utilisent notamment un algorithme génétique pour une conception automatique d'un réseau de neurones pour la classification de patterns. EMMANOUILIDIS et collab. [2000] tentent une approche multi-objectif du problème de sélection d'attributs avec un algorithme génétique qui utilise un opérateur de croisement spécifique pour préserver les parties des solutions prometteuses pour conserver une bonne performance. OLIVEIRA et collab. [2006] utilisent aussi un algorithme génétique multi-objectif, mais à deux niveaux, le premier consiste à choisir un classifieur parmi un ensemble donné pour trouver le meilleur classifieur, puis dans la seconde étape d'évaluer la sélection d'attributs selon deux approches : une approche supervisée et une approche non supervisée. HAMDANI et collab. [2007] tentent une approche en utilisant l'algorithme NSGA-II pour résoudre le problème de sélection d'attributs.

DUVAL et collab. [2009] se servent d'un algorithme mémétique pour une approche embarquée en utilisant SVM comme classifieur et en combinant un opérateur de croisement spécifique et une recherche locale dédiée à l'algorithme qui utilise les informations fournies par SVM pour améliorer les résultats. La sélection d'attributs non supervisée, bien que rapide, a comme inconvénient de négliger la détection de corrélation entre les attributs. Pour résoudre ce problème, CAI et collab. [2010] utilisent des techniques de Clustering pour réaliser une sélection d'attributs non supervisés performante. GHEYAS et SMITH [2010] proposent un algorithme hybride permettant de combiner la forte capacité au Recuit Simulé à éviter d'être piégé dans les minima locaux et la forte efficacité de convergence des opérateurs de croisement des algorithmes génétiques.

CERVANTE et collab. [2012] et XUE et collab. [2013] utilisent un algorithme à essaim de particules multi-objectif pour résoudre la sélection d'attributs. INBARANI et collab. [2015] proposent d'utiliser une recherche harmonique (*Harmony search*) qui permet une convergence plus rapide que les algorithmes à descente de gradient. Récemment, AZIZ et HASSANIEN [2016] ont proposé un algorithme inspiré de la nature, *Cuckoo Search* qui permet de traiter des données avec une grande dimensionnalité pour la sélection d'attributs.

Le Tableau 1.4 représente quelques méta-heuristiques pour la résolution de la sélection d'attributs de la littérature classées par ordre chronologique. La première colonne donne le type d'algorithme, la seconde le type d'approche et la dernière la référence.

1.4 Conclusion

Ce chapitre expose le contexte scientifique des travaux présentés dans cette thèse. Nous avons présenté comment les problèmes réels peuvent s'exprimer comme des pro-

Algorithme	Approche	Référence
Sélection par mesure de similarité	Filtrante	PHUONG et collab. [2005]
Algorithme Génétique	Enveloppante	YANG et HONAVAR [1998]
Algorithme Génétique	Enveloppante	EMMANOUILIDIS et collab. [2000]
Algorithme Génétique	Enveloppante	OLIVEIRA et collab. [2006]
NSGA II	Enveloppante	HAMDANI et collab. [2007]
Algorithme Génétique + ILS	Embarquée	DUVAL et collab. [2009]
Multi-Cluster	Enveloppante	CAI et collab. [2010]
Recuit Simulé + Algorithme Génétique	Enveloppante	GHEYAS et SMITH [2010]
Essaim de particules	Enveloppante	CERVANTE et collab. [2012]
Essaim de particules	Enveloppante	XUE et collab. [2013]
Recherche Harmonique	Enveloppante	INBARANI et collab. [2015]
Cuckoo Search	Enveloppante	AZIZ et HASSANIEN [2016]

TABLEAU 1.4 – Méta-heuristiques pour le problème de sélection d'attributs

blèmes d'optimisation combinatoire. Pour cela, il existe deux types de méthodes de résolution : les méthodes exactes et les méthodes approchées. Les premières, bien qu'elles garantissent la solution optimale, ont un temps de calcul qui croit généralement exponentiellement en fonction de la taille de l'instance. À l'inverse, les méthodes approchées permettent de trouver une solution en un temps raisonnable, mais sans garantie d'optimalité. Ce sont ces dernières que nous étudions dans ce mémoire de thèse : les heuristiques et les méta-heuristiques. Nous avons apporté pour chacun de ces deux types d'approches un état de l'art afin d'avoir une vue globale des principales méthodes de résolution de la littérature. Pour rappel, l'objectif de ces travaux de thèse est d'intégrer des connaissances dans les méthodes d'optimisation de la littérature. Cela est d'autant plus vrai pour les méta-heuristiques que nous avons utilisées dans ces travaux.

Dans la seconde partie de ce chapitre, nous avons présenté les deux problèmes étudiés durant cette thèse : Le problème d'ordonnancement sans temps d'attente (No-Wait Flowshop Scheduling Problem), et la sélection d'attributs dans un contexte de classification supervisée. En plus de la présentation détaillée de ces deux problèmes, un état de l'art a été fourni pour chacun d'entre eux.

Les définitions et les informations données dans ce chapitre présentent la base théorique nécessaire pour la bonne compréhension des prochains chapitres.

Chapitre 2

Intégration de connaissances : définitions et proposition de taxonomie

Sommaire

2.1 Introduction

De nos jours, avec la rapide augmentation des tailles de données des problèmes d'optimisation, les résoudre efficacement en un temps raisonnable devient plus difficile. Pour pallier ce problème, des méthodes sophistiquées et intelligentes ont été proposées dans la littérature. Ces méthodes peuvent être génériques ou spécifiques à un problème. Cependant, il a été observé que résoudre un problème de façon complètement générique est souvent bien moins efficace que de considérer ses spécificités. Ainsi les méthodes d'optimisation peuvent être enrichies de mécanismes d'intégration de connaissances. L'intégration de connaissances utilise des données qui peuvent être obtenues a priori, par l'étude de l'instance d'un problème avant tout processus de recherche et/ou avec des informations obtenues pendant la recherche. L'intégration de connaissances peut aider à accélérer les méthodes d'optimisation : pour cela, plusieurs techniques sont possibles comme utiliser une approximation de la fonction d'évaluation afin de réduire le temps de calcul lorsque l'évaluation devient coûteuse, ou réduire l'espace de recherche en éliminant des solutions dont on estime à l'avance qu'elles sont de mauvaise qualité.

L'intégration de connaissances peut aussi aider à améliorer la qualité des méthodes de résolution en utilisant la connaissance pour diriger la recherche de solutions vers une zone prometteuse de l'espace de recherche. En fonction de son utilisation, selon YANGMING [2017] l'intégration de connaissances est utilisée en optimisation combinatoire dans quatre grands domaines :

— **Reactive Search Optimization (RSO)** qui intègre des mécanismes d'intégration de connaissances dans les méthodes d'optimisation pour résoudre des problèmes complexes. L'intégration de connaissances agit sur les paramètres de l'algorithme en s'ajustant automatiquement au cours de la recherche.

— **Hyper-heuristique (HH)** qui cherche à automatiser à l'aide de l'intégration de connaissances, les processus de sélection, combinaison, génération ou d'adaptation de plusieurs heuristiques (ou composants d'heuristiques) pour résoudre efficacement des problèmes complexes.

— **Configuration automatique d'algorithmes (AAC)** qui choisit à l'aide de l'intégration de connaissances, les meilleurs paramètres possibles pour un algorithme et un ensemble d'instances donné.

— **Sélection automatique d'algorithmes (AAS)** qui sélectionne le meilleur algorithme pour résoudre une instance donnée. Presque toutes les approches récentes de sélection d'algorithmes utilisent l'intégration de connaissances pour apprendre avec les caractéristiques de l'instance, les relations entre une instance et les performances d'un algorithme.

Dans cette thèse, nous proposons d'ajouter un cinquième domaine :

— **Modèle de substitution (SM)** qui consiste à transformer les données d'une instance à l'aide de l'intégration de connaissances en minimisant la perte d'informations pour rester le plus précis possible. Cela a pour but de modifier la structure du problème et de l'espace de recherche, pour une meilleure exploration de celui-ci.

Dans ce chapitre, nous allons nous concentrer sur les approches qui utilisent des mécanismes d'intégrations de connaissances pour aider les méthodes d'optimisation combinatoire. Dans la prochaine section, nous définissons ce qu'est l'intégration de connaissances en la décomposant en trois grandes étapes : *(i)* Extraction *(ii)* Mémorisation et *(iii)* Exploitation. La Section 2.3 présentera les techniques d'extraction de connaissances de la littérature. La Section 2.4 se concentrera sur les techniques de mémorisation et la Section 2.5 sur les techniques d'exploitation de la littérature. Pour finir, une taxonomie des méthodes d'intégration de connaissances sera proposée dans la Section 2.6.

2.2 Intégration de connaissances : vue générale

Dans cette section, nous allons définir les objectifs de l'intégration de connaissances et comment elle peut être mise en oeuvre.

2.2.1 Objectifs

L'utilisation de l'intégration de connaissances dans les méthodes d'optimisation est de plus en plus développée de nos jours. Elle est utilisée pour plusieurs aspects des méthodes d'optimisation. On retrouve le plus souvent les objectifs suivant pour l'intégration de connaissances :

— Améliorer la qualité des solutions.

— Accélérer la recherche pour trouver de meilleures solutions.

— Optimiser les paramètres d'un algorithme.

— Sélectionner les bons opérateurs d'un l'algorithme.

Les techniques d'intégration de connaissances peuvent se faire suivant deux directions :

— **Méthodes a priori** (ou off-line) qui ont pour but d'identifier des structures et des schémas intéressants pour un type de problème avant tout processus de recherche. (Exemple : HH, AAC, AAS, SM)

— **Méthodes dynamique**s (ou on-line) qui vont apprendre et utiliser des informations collectées pendant le processus de recherche. (Exemple : RSO, SM)

2.2.2 Étapes de l'intégration de connaissances

Nous avons remarqué que l'intégration de connaissances suit globalement trois étapes que nous allons détailler dans les prochaines sections :

— **Extraction :** cette étape définit quelles données vont être extraites et utilisées pour l'exploitation de connaissances. Ces données peuvent provenir de l'instance ou des solutions du problème.

— **Mémorisation :** cette étape définit la façon dont on va enregistrer les informations. Dans certains cas, il n'est pas possible de garder toute l'information extraite et des choix sont nécessaires. De plus, dans le cas des méthodes d'intégration de connaissances dynamiques, les informations doivent être mises à jour. C'est alors cette étape qui définit comment mettre à jour les informations.

— **Exploitation :** cette étape définit quant à elle la façon dont va être utilisée la connaissance dans la méthode d'optimisation.

La Figure 2.1 montre l'enchaînement des différentes étapes. Nous pouvons constater que l'"Exploitation" n'est pas forcément l'étape finale. En effet, dans le cas des méthodes dynamiques, il est généralement nécessaire de mettre à jour les informations des différentes étapes.

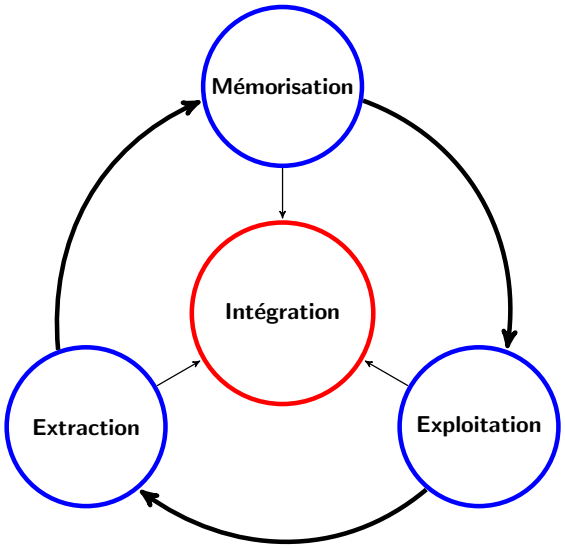


FIGURE 2.1 – Étapes de l'intégration de connaissance dans les méthodes d'optimisation

Les trois prochaines parties vont donc s'intéresser à décrire plus précisément ces trois étapes à commencer par l'Extraction de connaissances.

2.3 Extraction de connaissances

O'HAGAN et collab. [2012] dégagent deux catégories d'extraction de connaissances inspirées de la biologie :

— **Génotypique :** Correspond aux gènes d'une solution, c'est-à-dire les variables de décision.

— **Phénotypique :** Correspond aux traits observables d'une solution, c'est-à-dire les différentes métriques qu'une solution peut donner. Il s'agit principalement de la qualité de la solution, mais pour certains problèmes, des métriques spécifiques peuvent être calculées.

Le génotype et phénotype sont deux méthodes d'extraction de connaissances orientées solution. Nous distinguons aussi dans certaines approches que l'**instance** d'un problème peut-être utilisée afin d'en extraire des connaissances. Ainsi, il est possible de caractériser une instance pour choisir quelle stratégie adopter pour un type d'instance spécifique.

Nous proposons donc d'organiser les approches d'extraction de connaissances suivant ces trois aspects, en commençant par la connaissance issue de l'analyse de l'instance.

2.3.1 Connaissance à partir d'une instance

L'extraction de connaissances à partir d'une instance d'un problème est notamment très utilisée dans la problématique de Sélection automatique d'Algorithmes. Le problème de Sélection automatique d'Algorithmes a été proposé initialement par RICE [1976]. L'objectif étant qu'après avoir extrait de la connaissance d'une instance I, le meilleur algorithme possible pour cette instance soit choisi parmi un ensemble A d'algorithmes. La Figure 2.2 illustre le processus général de la Sélection Automatique d'Algorithmes.

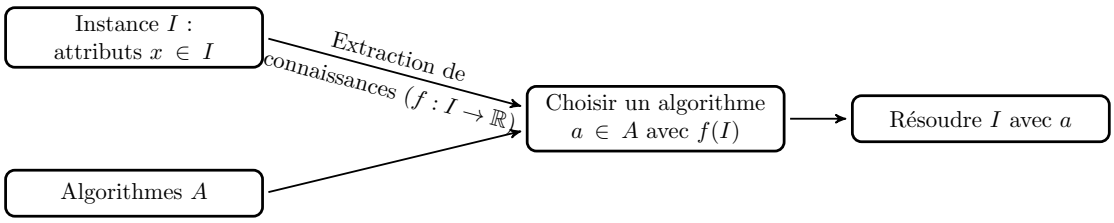


FIGURE 2.2 – Schéma d'une Sélection automatique d'Algorithmes à partir des données d'une instance.

L'état de l'art proposé par KOTTHOFF [2016] sur la Sélection automatique d'Algorithmes montre plusieurs techniques d'extraction de connaissances comme la classification, le clustering, la régression et le ranking. Par exemple, KANDA et collab. [2016] utilisent des algorithmes comme le plus proche voisin (*k-NN - k-nearest neighbor*), des arbres de décisions et un perceptron multi-couche pour déduire le modèle capable d'associer les propriétés des instances du problème du voyageur de commerce avec les méthodes d'optimisation performantes à utiliser.

TELELIS et STAMATOPOULOS [2001] utilisent un modèle de régression pour résoudre les problèmes de sac-à-dos et de partitionnement. Pour cela l'instance est utilisée pour extraire une fonction d'approximation grâce au modèle de régression. Cette fonction est ensuite utilisée en tant que guide pour la construction d'une solution dans une zone prometteuse de l'espace de recherche.

L'instance peut être aussi utilisée pour déterminer un critère d'arrêt. Idéalement, les algorithmes sont exécutés jusqu'à ce que la meilleure solution soit atteinte. Cependant, pour certains problèmes, calculer une borne pour déterminer la qualité potentielle de

la meilleure solution à trouver n'est pas facile. Dans cette optique, ARBELAEZ et O'SULLIVAN [2016] utilisent un modèle de régression linéaire sur l'instance du problème pendant l'initialisation d'une recherche locale pour déterminer la potentielle qualité de la meilleure solution afin de stopper le processus de recherche lorsque cette qualité est atteinte afin d'éviter de perdre du temps de calcul.

2.3.2 Connaissance phénotypique

La connaissance phénotypique est caractérisée par toutes les métriques obtenues à partir d'une solution. Par exemple, sa qualité ou le degré de neutralité de son voisinage.

HORN et collab. [1994] utilisent notamment des niches phénotypiques basées sur des caractéristiques partagées entre des solutions d'une population. Cela permet ainsi à l'algorithme génétique de préserver la diversité des individus en réalisant des croisements entres des solutions provenant de niches différentes.

L'analyse de paysages fait partie des données phénotypiques, qui consiste à mettre en relation les solutions avec leur qualité. WEDGE et KELL [2008] utilisent la corrélation entre la qualité et la structure des solutions (génotype) pour faire un contrôle des paramètres pour un algorithme génétique. Ainsi grâce à l'analyse du paysage, l'algorithme détermine le nombre optimal de solutions que doit contenir la population. L'analyse de paysages permet également d'établir, par exemple, une mesure du *degré de neutralité* d'une solution qui est le nombre de voisins ayant la même qualité que cette solution. MARMION et collab. [2011] utilisent la neutralité avec un système de bandits pour orienter une recherche locale vers la solution voisine neutre la plus prometteuse.

LESSMANN et collab. [2011] utilisent quant à eux des algorithmes de data-mining pour paramétrer en ligne un algorithme à essaim de particules. Pour cela des modèles de régression sont utilisés pour trouver les meilleures valeurs de paramètres à partir des valeurs utilisées précédement par l'algorithme lui-même.

2.3.3 Connaissance génotypique

La connaissance génotypique est liée à la position d'une solution dans l'espace de recherche et est donc directement liée à sa structure, *i.e.* à partir de son *génotype*.

Pour illustrer l'extraction de connaissances génotypiques, le Tableau 2.1 montre une sélection de 5 solutions en représentation binaire dont on suppose qu'elles sont les meilleures solutions trouvées. On a pour chaque gène le pourcentage de sélection. Ainsi les gènes 1, 4 et 10 sont toujours présents dans les meilleures solutions, on peut donc supposer que ces gènes sont aussi présents dans la solution optimale. De même que le pourcentage de présence, une autre information possible pourrait être la présence (ou non) simultanée de plusieurs gènes, comme le gène 5 et 8, si l'un est présent, l'autre l'est aussi. On peut aussi créer des règles plus complexes pour extraire plus d'information.

Ce génotype permet entre autre l'apprentissage de modèles statistiques pour prédire la qualité des solutions. MICHALSKI [2000] utilise dans *Learnable Evolution Model* (LEM) un classifieur pour extraire la connaissance du génotype des solutions d'une population.

1	0	1	1	0	0	1	0	0	1
1	0	0	1	0	1	1	0	0	1
1	0	1	1	1	1	1	1	0	1
1	1	1	1	1	1	0	1	0	1
1	1	1	1	0	1	0	0	0	1
100%	40%	80%	100%	40%	80%	60%	40%	0%	100%

TABLEAU 2.1 – Exemple d'extraction de la connaissance génotypique à partir des 5 meilleures solutions d'un problème

Ensuite l'algorithme mémorise ce qui caractérise les meilleures solutions et les pires solutions. Puis il exploite cette connaissance pour sélectionner les solutions dont la qualité est prédite, les meilleures solutions sont conservées et les mauvaises supprimées. Une version multi-objectif de cette approche LEMMO de JOURDAN et collab. [2005], qui utilise le même principe en utilisant la notion de dominance dans un contexte d'optimisation multi-objectif .

Toujours dans l'utilisation d'un modèle d'apprentissage, KNOWLES et NAKAYAMA [2008] utilisent dans un algorithme évolutionnaire, un modèle de régression linéaire qui est appris pendant la génération et qui met en relation la qualité et le génotype. Ensuite, ce modèle est exploité pour filtrer les individus avant de les évaluer. Ainsi si la qualité prédite est basse, les individus sont ignorés.

Le génotype permet aussi la création de nouvelles solutions comme le font LALLA-RUIZ et VOSS [2015] dans une recherche locale itérée. Après une descente d'un *Variable Neighborhood Search* (VNS), la solution obtenue est alors utilisée pour extraire la connaissance de son génotype, puis l'information est enregistrée dans une mémoire qui est ensuite exploitée à l'initialisation de la prochaine solution de la descente.

Enfin, le génotype peut aussi orienter la recherche comme l'ont réalisé SCHINDL et ZUFFEREY [2013] dans une recherche tabou avec intégration de connaissance. Dans cette méthode, le génotype de la meilleure solution trouvée lors d'un *cycle* (*i.e.* un certain nombre d'itérations) est alors utilisée pour mémoriser des combinaisons de caractéristiques qui génèrent une bonne qualité. Enfin, cette connaissance est exploitée pour orienter l'opérateur de voisinage.

2.3.4 Classification de différentes approches de la littérature

Le tableau 2.2 référence les approches où des connaissances phénotypiques (P), génotypiques (G) et de l'instance (I) sont extraites.

2.4 Mémorisation de connaissances

Lors d'une méthode de résolution, beaucoup d'informations peuvent être extraites à chaque itération. Les problèmes étudiés étant des problèmes combinatoires de grandes tailles, il n'est pas possible d'enregistrer toutes les informations. En conséquence, une

Type	base d'extraction	Approche	Référence
I	Instance	Voisinage	ARBELAEZ et O'SULLIVAN [2016]
I	Instance	Sélection d'algorithme	KANDA et collab. [2016]
P	Population	Evolutionnaire	WEDGE et KELL [2008]
P	Solution	Voisinage	MARMION et collab. [2011]
G	Population	Evolutionnaire	MICHALSKI [2000]
G	Population	Evolutionnaire	JOURDAN et collab. [2005]
G	Population	Evolutionnaire	KNOWLES et NAKAYAMA [2008]
G	Solution	Voisinage	SCHINDL et ZUFFEREY [2013]
G	Solution	Voisinage	LALLA-RUIZ et VOSS [2015]
G	Solution	Voisinage	ZHOU et collab.

TABLEAU 2.2 – Méthodes d'extraction de connaissances. (P = Phénotypique, G = Génotypique, I = Instance)

sélection de l'information est nécessaire. Nous avons identifié plusieurs possibilités pour mémoriser la connaissance. De plus, en fonction du type de mémoire utilisée, différentes fréquences de mémorisation sont possibles. La première partie de cette section présente les méthodes de mémorisation, et la seconde les différentes fréquences de mémorisation.

2.4.1 Méthodes de mémorisation

Les méthodes à base règles utilisent la connaissance extraite pour la traduire en termes de règles qui seront mémorisées afin de les utiliser au cours de la recherche. Nous avons discerné deux types de règles : *(i)* les règles *a priori* et *(ii)* les règles *dynamiques*.

(i) A priori : ces règles sont définies grâce à de la connaissance extraite a priori du problème avant tout processus de recherche et n'évoluant pas pendant la méthode d'optimisation. Ce type de règles est notamment très utilisé dans les algorithmes gloutons ou heuristiques constructives. Ces méthodes utilisent de la connaissance spécifique au problème afin de construire une solution. Par exemple, dans l'heuristique GAPH, ARAÚJO et NAGANO [2010] utilisent une propriété structurelle du problème d'ordonnancement de type Flowshop sans temps d'attente basée sur la mesure du GAP entre deux jobs pour proposer une heuristique constructive efficace pour minimiser le temps total de complétion. La connaissance du problème a permis de définir un ordre d'insertion dans une solution partielle, ainsi la règle indique qu'il faut insérer dans la solution partielle, la tâche ayant le plus petit GAP parmi les tâches non-encore ordonnancées. Ces règles peuvent aussi servir à définir un critère d'arrêt comme le font ARBELAEZ et O'SULLIVAN [2016] pour extraire un critère d'arrêt à partir de certaines caractéristiques d'une instance d'un problème pour éviter un temps de calcul plus long que nécessaire.

(ii) dynamiques : ces règles évoluent au cours de la recherche, généralement grâce à des mesures obtenues à partir des solutions construites. C'est ainsi que SANTOS et collab. [2008] proposent une hybridation entre un *GRASP* (Greedy Random Adaptive Search Procedure) et du *Data Mining*. Le DM-GRASP génère un nombre n de solutions, puis

garde uniquement les k meilleures. Ces solutions sont ensuite utilisées pour apprendre des schémas (*pattern*) présents afin de donner des règles de construction des nouvelles solutions pour la recherche. Dans LEM, un modèle de classification est appris à partir de la population à l'aide des meilleures et pires solutions de cette population. Ensuite ce modèle de classification est utilisé pour filtrer les nouvelles solutions générées afin de n'évaluer que les solutions les plus intéressantes. Toujours dans les algorithmes à population, BECERRA et COELLO [2005] utilisent dans le *Cultural Algorithm* une solution *leader* qui va diriger la recherche en forçant l'opérateur de mutation à diriger la solution sélectionnée vers la solution leader. Ainsi la règle définie est que les solutions doivent se rapprocher de leur leader : une mutation doit obligatoirement diminuer la distance à la solution leader.

On peut noter que les approches à base de voisinage sont peu représentées dans les méthodes avec des règles dynamiques dans la littérature. Cela peut s'expliquer par le fait qu'une recherche locale dispose de peu d'informations à un instant t de la recherche et donc peu difficilement établir des règles dynamiquement. Néanmoins, ce type d'approche est propice à l'utilisation des méthodes par renforcement.

Les méthodes par renforcement ont pour but d'apprendre à partir d'observations (par exemple : des solutions courantes d'un algorithme), quelle sont les actions à entreprendre en fonction de différentes situations pour maximiser le gain d'une fonction objectif à un moment donné.

On retrouve les méthodes par renforcement dans les algorithmes de colonies de fourmis de DORIGO et collab. [1996] initialement proposés pour le problème du voyageur de commerce, qui utilisent une matrice de phéromones. Dans ce type d'algorithmes, la matrice de phéromones va renforcer les décisions prises le plus fréquemment, et celles qui seront peu choisies seront oubliées au cours du temps par un principe d'évaporation. Ainsi plus la quantité de phéromones est importante est plus la probabilité de choisir un chemin est important. La quantité de phéromone est définie par la formule suivante :

$$\tau_{i,j}(t+1) = \tau_{i,j}(t) * (1-\rho) + \sum_{k=1}^{m} \Delta\tau_{i,j}^{k}(t) \tag{2.1}$$

Où ρ est un paramètre d'évaporation, afin de diminuer la quantité de phéromones si les fourmis n'empruntent plus le chemin (i, j). $\sum_{k=1}^{m} \delta\tau_{i,j}^{k}(t)$ correspond à la quantité de phéromone déposée par la fourmi k, ainsi plus une fourmi emprunte le chemin, plus la quantité de phéromones déposée sera importante.

Outre les algorithmes à colonies de fourmis, d'autres approches utilisent ce type de mémoire par renforcement, par exemple, le Learning Variable Neighborhood Search (L-VNS) de LALLA-RUIZ et VOSS [2015] dans un problème d'ordonnancement sur des machines parallèles identiques. L-VNS renforce les combinaisons de caractéristiques si elles sont présentes dans la solution, sinon ces combinaisons sont oubliées petit à petit. Cela se fait par une matrice à deux dimension $M_{i,j}$ où j représente la $j^{ième}$ tâche et i la $i^{ième}$ position dans la solution. La matrice est mise à jour selon cette formule :

$$M_{i,j} = (M_{i,j} + 1) * \alpha \tag{2.2}$$

Où $\alpha \geq 1$, si la solution a été améliorée dans l'itération, $\alpha < 1$ sinon.

Suivant le même principe, le *Learning Tabu Search* (LTS) de SCHINDL et ZUFFEREY [2013] renforce cette fois les combinaisons de caractéristiques proportionnellement à la qualité de la solution, ce qui permet un renforcement d'autant plus fort que la qualité de la solution est grande.

$$tr(a_i, a_j) = \rho * tr(a_i, a_j) + \Delta_{tr(a_i,a_j)} \tag{2.3}$$

Où ρ est le facteur d'évaporation similaire aux algorithmes de fourmis et $\Delta_{tr(a_i,a_j)}$ le facteur de renforcement.

Une autre approche par renforcement et inspirée de la nature comme les colonies de fourmis : les essaims particulaires de KENNEDY et EBERHART [1995] qui permettent de choisir une direction à prendre pour un ensemble de solutions. Les formules de renforcement sont les suivantes :

$$V_{k+1} = \omega * V_k + b_1 * (P_i - X_k) + b2 * (P_g - X_k) \tag{2.4}$$

$$X_{k+1} = X_k + V_{k+1} \tag{2.5}$$

où V_k est la vitesse, w l'inertie, P_i la qualité de la meilleure solution de la population, P_g la qualité de la meilleure solution du voisinage de P_i, b_1 et b_2 deux nombres tirés aléatoirement entre 0 et 1. Ce modèle permet, grâce à une large exploration de l'espace de recherche de la population, de renforcer la direction prise par les solutions pour converger rapidement vers un minimum local de bonne qualité.

Toutes ces méthodes ont des points communs. Nous proposons de décomposer une méthode de renforcement en deux parties : l'*historique* et la *récompense*.

L'*historique* est la valeur qui va être renforcée ou oubliée et qui est obtenue à partir des itérations passées, l'historique prend la forme suivante : $V_t * I$ où V représente la valeur de l'historique à l'instant t et I correspond à l'inertie, qui agit comme un facteur d'oubli au cours du temps, similaire à l'évaporation des algorithmes de colonies de fourmis. La *Récompense* r est une valeur qui va être ajoutée à l'itération en cours pour accentuer ou non l'importance d'une caractéristique. Ainsi nous généralisons la mise-à-jour des mémoires par renforcement sous la forme suivante :

$$V_t = V_{t-1} * I + r \tag{2.6}$$

Où V est l'information à renforcer, l'*inertie* I est un facteur compris entre 0 et 1 permettant d'oublier petit à petit la connaissance n'étant plus utilisée ou obsolète. En effet, au cours du temps, une information obtenue au début de la recherche et qui n'intervient plus au cours de la recherche n'est plus utile. La *Récompense* dépend généralement du problème ou de la méthode utilisée, elle permet le renforcement plus ou moins fort d'une information pour la suite de la recherche.

L'inconvénient de ce type de méthodes est qu'elles mémorisent l'information uniquement à partir du passé pour décider de l'instant présent. Des améliorations aux méthodes de renforcement ont donc vu le jour pour prédire plus efficacement une succession de décisions futures, ainsi au lieu de tenir compte uniquement des décisions passées, les éventuelles décisions futures sont aussi prises en compte.

C'est ce que permettent les approches à base de *Q-Learning* en complexifiant la fonction d'apprentissage.

Les méthodes Q-Learning initiées par WATKINS et DAYAN [1992] qui, à l'aide d'une fonction de gain, exprime l'utilité d'une action dans un état considéré. Cet algorithme est basé sur une simple mise à jour de la *Q-valeur* de chaque paire d'état-action (s, a). Quand, dans un état s, une action a est choisie, la Q-Valeur de cette paire $Q(s, a)$ est mise à jour avec la récompense reçue (*i.e.* à définir en fonction de l'approche souhaitée) ainsi que la meilleure Q-valeur qu'il soit possible d'atteindre dans un prochain état s'. La possibilité de tenir compte des récompenses futures est un des points forts du Q-learning. La mise à jour d'une paire état-action se fait suivant cette équation :

$$Q(s, a) = Q(s, a) + \alpha * \left(r + \gamma * max_{a'} Q(s', a') - Q(s, a)\right) \tag{2.7}$$

Dans cette équation, $\alpha \in [0, 1]$ est la vitesse d'apprentissage et r la récompense ou la pénalité en fonction de l'action a choisie dans un état s. La vitesse d'apprentissage détermine le degré avec laquelle une ancienne valeur est mise à jour. Par exemple, si la vitesse d'apprentissage est 0, aucun changement n'est effectué. D'autre part, si $\alpha = 1$, alors l'ancienne valeur est remplacée par la nouvelle estimée (*i.e.* on a $Q(s', a') - Q(s, a)$ par développement si l'on a $\alpha = 1$). Le facteur d'actualisation $\gamma \in [0, 1]$ permet d'estimer la qualité future qu'il est possible d'obtenir dans un état s'. Ainsi, si γ est proche de 0, la fonction ne considèrera que la récompense immédiate, alors que si γ tend vers 1, la fonction tiendra compte de la récompense future avec un plus grand poids.

Le Q-Learning peut aussi servir pour le choix d'opérateurs, comme l'utilisent DOS SANTOS et collab. [2014] pour choisir l'opérateur de voisinage après une descente d'un VNS afin de choisir le meilleur opérateur de voisinage. Ici, l'état est l'opérateur de voisinage courant et l'action le prochain opérateur de voisinage utiliser. Ainsi au fur et à mesure de la recherche l'algorithme apprend à déterminer quel opérateur de voisinage est à utiliser après chaque itération. Une autre possibilité est d'utiliser un Q-Learning plutôt orienté solution. Ainsi MARTÍNEZ et collab. [2011] utilisent un double Q-Learning pour le problème d'ordonnancement *Flexible Job Shop* pour choisir le placement des *Job*s sur les machines. De même, ARIN et RABADI [2017] utilise le Q-Learning dans un Meta-RASP pour construire des bonnes solutions de départ pour un problème de sac-à-dos multidimensionnel à chaque itération.

Les méthodes des bandits ou *Multi-Armed Bandits (MAB)* est un système composé de K bras indépendants, où chaque bras à une probabilité inconnue de donner une récompense. Une politique de sélection optimale est de choisir le bras qui maximise la

récompense totale sur le long terme. Plusieurs algorithmes de bandits ont été proposés dans la littérature, et l'un des plus connus est l'UCB *Upper Confidence Band* proposé par AUER et collab. [2002]. Dans cette approche, un bras i du bandit a une qualité $\hat{q}_i$ estimée empiriquement et un intervalle de confiance qui dépend du nombre de fois n_i que le bras a été choisi précédemment. Ainsi le choix du bras se fait suivant cette fonction à maximiser :

$$\text{UCB} = arg\max_{i=1..k} \hat{q}_i + \text{C} * \sqrt{\frac{2 * ln \sum_{j=1}^{\text{K}} n_j}{n_i}} \tag{2.8}$$

Où C est un facteur multiplicatif pour faire un compromis entre l'exploitation (choisir de préférence le bras avec la meilleure récompense estimée $\hat{q}_i$) et l'exploration (choisir de préférence les bras qui ont été peu choisis).

À l'aide de cette méthode, LI et collab. [2014] ont réalisé une sélection d'opérateurs adaptative (*Adaptative Operator Search (AOS)*) pour un algorithme évolutionnaire multi objectif afin de choisir les meilleurs opérateurs de croisement et de mutation parmi un ensemble. Une autre méthode, plutôt orientée sélection de solutions par MARMION et collab. [2011] utilise les bandits afin de choisir parmi le voisinage quelle solution est à explorer.

2.4.2 Fréquences de mise à jour de la mémoire

Maintenant que les différents types de mémoire ont été identifiés, il faut désormais déterminer à quel moment mettre à jour la mémoire. En effet, en fonction des méthodes utilisant une mémoire, il peut être pénalisant de mettre à jour celle-ci trop régulièrement ou pas assez souvent. Dans la première situation, il s'agit de sur apprentissage et cela restreint trop l'espace de recherche, dans la seconde l'information n'est pas pertinente et revient à une recherche aléatoire. Nous avons donc identifié plusieurs possibilités qui dépendent en partie du type de mémoire.

À chaque itération C'est la méthode la plus intuitive, car la connaissance est ainsi exploitée à chaque itération. On retrouve ce processus de mise à jour dans les algorithmes inspirés de la nature comme les colonies de fourmis ou les essaims de particules. Néanmoins, cette approche peut restreindre l'espace de recherche si le renforcement d'une zone à explorer devient trop rapidement important. De plus, si la mémoire est coûteuse à mettre à jour, cela peut vraiment ralentir l'algorithme d'optimisation.

Après un cycle Pour contrer le renforcement trop rapide de la proposition précédente, l'algorithme peut attendre un certain nombre d'itérations avant la mise à jour de la mémoire. LEM/LEMMO de MICHALSKI [2000] utilise un cycle pour alterner entre la phase d'extraction de connaissances (*i.e.* l'apprentissage du modèle de classification) et l'exploitation de celle-ci. Une autre approche, celle du Learning Tabu Search (LTS)

de SCHINDL et ZUFFEREY [2013] qui utilise aussi un cycle pour la mise à jour de la mémoire, cette fois en utilisant la meilleure solution trouvée pendant ce cycle, qu'elle soit meilleure ou non que celle trouvée depuis le début de la recherche.

Après qu'une bonne solution soit atteinte Pour garantir la mémorisation de l'information de bonne qualité, il est possible de mémoriser l'information uniquement lorsqu'un optimum local est atteint. Par exemple dans le Parallel Local Search (PLS) de ARBELAEZ et HAMADI [2011]. Les solutions d'une population évoluent ensemble à l'aide d'une recherche locale et partagent une mémoire commune. Cette mémoire est mise à jour dès qu'une solution de la population tombe dans un optimum local. Ce critère de mise à jour peut être un inconvénient à cause de l'irrégularité des optima locaux. Pour certains problèmes, les approches de résolution peuvent mettre un certain temps avant d'en trouver un.

Unique La dernière possibilité est une mémorisation unique, par exemple lors d'une initialisation d'une méthode d'optimisation, extraire des règles qui resteront valides tout le long de la recherche et ne changeront pas. Dans leur méthode, ARBELAEZ et O'SULLIVAN [2016] apprennent et définissent le critère d'arrêt dès l'initialisation et aucune information n'est extraite dans la suite de l'algorithme.

2.5 Exploitation de connaissances

La dernière composante de l'intégration de connaissance est celle de son exploitation. L'exploitation de connaissances consiste à utiliser la connaissance extraite pour l'utiliser dans les différents mécanismes de la recherche. De plus, tout comme pour la mémorisation de connaissances, on doit définir à quelle fréquence la connaissance est exploitée.

2.5.1 Les mécanismes

Paramètres L'utilisation de la connaissance peut se faire sur le paramétrage des algorithmes d'optimisation. BENLIC et collab. [2017] utilisent un mécanisme de contrôle des paramètres avec un apprentissage par renforcement pour déterminer des paramètres de la fonction de perturbation pour trouver le type et le nombre de mouvements nécessaires à un moment donné de la recherche.

Initialisation Dans cette procédure, il est possible d'exploiter la connaissance pour construire une bonne solution initiale. Dans cette optique, LALLA-RUIZ et VOSS [2015] exploitent la connaissance pour construire les solutions initiales successives d'une ILS (Iterated Local Search). Après chaque descente d'un VNS (Variable Neighborhood Search), la structure de la solution est intégrée dans une mémoire d'apprentissage, et est ensuite utilisée dans le mécanisme d'initialisation pour la descente suivante. Ainsi après chaque descente, la mémoire se renforce et devient capable à chaque itération de construire une nouvelle solution de départ de meilleure qualité.

Intensification Dans VEGAS de MARMION et collab. [2011], l'utilisation de la connaissance se fait à l'aide d'un système de bandits pour choisir efficacement la zone de l'espace de recherche à intensifier. On peut aussi utiliser la connaissance pour réduire le nombre d'évaluations, par exemple, dans le Learning Tabu Search de SCHINDL et ZUFFEREY [2013] la connaissance est utilisée pour choisir efficacement un sous-ensemble de voisins à évaluer. Le Chapitre 4 présente une extension de ces travaux.

Diversification La connaissance peut aussi s'utiliser pour diversifier les zones de recherche et donc les solutions rencontrées. Dans ce contexte, LI et collab. [2014] se servent d'une connaissance basée sur le phénotype pour la sélection d'opérateurs avec un système de bandits. Ainsi lorsqu'il n'est plus possible d'intensifier une zone de l'espace de recherche, l'algorithme sélectionne grâce à la connaissance acquise au cours de la recherche, un nouvel opérateur permettant d'atteindre de nouvelles zones de l'espace de recherche.

Évaluation Le dernier mécanisme où la connaissance peut intervenir est celui de l'évaluation. La connaissance peut permettre d'éviter des évaluations qui peuvent être coûteuses. C'est le cas des algorithmes à base de *Modèle de substitution*, où la connaissance est utilisée afin d'estimer la qualité potentielle de solution sans les évaluations. Cela est très efficace pour les approches où la fonction d'évaluation est relativement coûteuse.

2.5.2 Fréquences d'exploitation

Tout comme pour la mémorisation, il faut définir à quelle fréquence la mémoire doit être exploitée. En effet, il n'est pas toujours bon d'exploiter constamment la connaissance car cela peut faire converger rapidement un algorithme vers une zone de l'espace de recherche pouvant se révéler inintéressante. Il peut ainsi être efficace d'utiliser un comportement classique pendant une certaine durée pour explorer de nouvelles zones de l'espace de recherche, pour apprendre à nouveau avant d'utiliser à nouveau la connaissance.

Nous avons identifié plusieurs possibilités de fréquence dans la littérature :

— **Unique** : L'utilisation de la connaissance ne sert qu'une fois. C'est le cas des méthodes de construction de solutions pour l'initialisation ou pour fixer les paramètres des algorithmes.

— **Continue** : l'utilisation de la connaissance est réalisée en continue. C'est le cas dans la plupart des méthodes d'optimisation intégrant de la connaissance.

— **Cyclique** : L'utilisation de la connaissance se fait périodiquement en alternant des phases avec utilisation de la connaissance et d'autre sans comme dans LEM/LEMMO (JOURDAN et collab. [2005]) ou en alternant avec des phases d'utilisations différentes de la connaissance (Exemple : Pour intensifier ou pour diversifier).

2.6 Taxonomie des méthodes d'intégration de connaissances

Pour résumer, l'intégration de connaissances s'articule autour de trois grandes étapes : L'extraction, la mémorisation et l'exploitation. Chacune de ces étapes possède un ou plusieurs paramètres. La figure 2.3 est une extension de la figure 2.1 qui illustre ces différentes étapes ainsi que leurs paramètres.

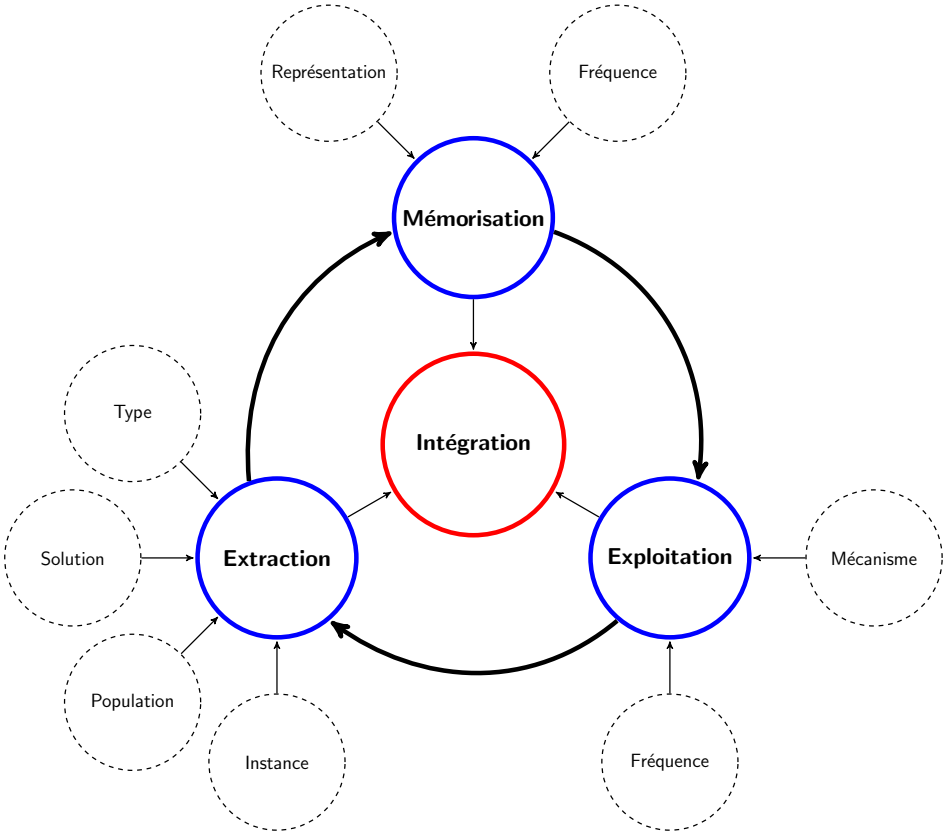


FIGURE 2.3 – Représentation de l'intégration de connaissances

Afin de classifier les approches de la littérature et de bien identifier les méthodes d'utilisation de la connaissance, nous proposons une taxonomie des méthodes d'intégration de connaissances. Cette taxonomie permet d'identifier sous la forme d'un triplet les différents paramètres des trois étapes : extraction, mémorisation et exploitation de connaissances. Ainsi un algorithme utilisant la connaissance peut se décrire sous la forme d'un triplet *Extraction* / *Mémorisation* / *Exploitation* :

$$(Source)/(Méthodes)^{Fréquence}/(Mécanisme)^{Fréquence} \tag{2.9}$$

Le tableau 2.3 référence chacune des possibilités pour chaque membre du triplet ainsi que la notation utilisée dans ce mémoire de thèse. Cette taxonomie nous permet alors de positionner quelques travaux de la littérature utilisant de l'intégration de

connaissances. Par exemple, le Learning Tabu Search de SCHINDL et ZUFFEREY [2013] est une approche qui extrait de la connaissance du génotype à partir de la meilleure solution connue d'un cycle ($G_{S_{best}}$), les caractéristiques de cette solution sont sauvegardées dans une matrice d'apprentissage par renforcement qui est mise à jour à chaque fin de cycle, soit k itérations $((ML)^k)$. Enfin, le choix des voisins à évaluer se fait dans les mécanismes d'intensification (amélioration de la solution) et de diversification (perturbation de la solution) à chaque itération $((Div + Int)^*)$. Ainsi le triplet associé à cette méthode est donc $G_{S_{best}}/(ML)^k/(Div + Int)^*$.

Étape	paramètre	valeur	notation
Extraction	Type	Phénotypique	F
Extraction	Type	Génotypique	G
Extraction	Solution	Courante	S_{cur}
Extraction	Solution	Meilleure	S_{best}
Extraction	Population	-	P
Extraction	Instance	-	I
Mémorisation	Représentation	Règles	$Rule$
Mémorisation	Représentation	Renforcement	ML
Mémorisation	Représentation	Q-Learning	QL
Mémorisation	Représentation	Bandits	MAB
Exploitation	Mécanisme	Paramétrage	$Param$
Exploitation	Mécanisme	Initialisation	$Init$
Exploitation	Mécanisme	Diversification	Div
Exploitation	Mécanisme	Intensification	Int
Exploitation	Mécanisme	Sélection	$Select$
Exploitation	Mécanisme	Évaluation	$Eval$
-	Fréquence	Toujours	*
-	Fréquence	Cyclique	k
-	Fréquence	Optimum local atteint	LO
-	Fréquence	Unique	1

TABLEAU 2.3 – Proposition d'une taxonomie des approches d'intégration de connaissances

Le tableau 2.4 présente quelques travaux de la littérature qui utilisent l'intégration de connaissances. Ces travaux sont présentés selon le type d'extraction (*Génotype* ou *Phénotype*, la méthode de mémorisation (*Méthode* et Fréquence) et la méthode d'exploitation (*Mécanisme* et *Fréquence*). On remarque dans ce tableau que les approches Génotypique et Phénotypique sont autant utilisées l'une que l'autre au cours du temps. De même qu'il ne semble pas y avoir de distinction majeure sur l'utilisation d'une population de solutions ou d'une solution unique. Cependant, dans le cadre d'une solution unique, la connaissance est extraite en très grande partie de la meilleure solution connue de l'algorithme. Peu de méthodes utilisent une solution courante. Les deux méthodes de mémorisation de connaissances les plus privilégiées sont les approches à base

de règles et par renforcement. Les approches à base de règles ont été principalement utilisées il y a plus d'une décennie, récemment, nous pouvons remarquer une augmentation des méthodes par renforcement qui sont de plus en plus utilisées. Pour finir, l'exploitation de connaissances semble privilégier la diversification des solutions.

2.7 Conclusion

Dans ce chapitre, nous avons présenté une revue de la littérature pour l'intégration de connaissances dans les méthodes d'optimisation. Nous avons ainsi pu diviser l'intégration de connaissances en trois grandes parties : L'Extraction, la Mémorisation et l'Exploitation de connaissances. Chacune de ces parties est composée d'un certain nombre de paramètres qui ont été énumérés afin de proposer une taxonomie des méthodes d'intégration de connaissances.

Méthode	Taxonomie			Référence
	Extraction	Mémorisation	Exploitation	
Algorithme de niche	F_P	$(Rule)^*$	$(Div)^*$	HORN et collab. [1994]
ACO	F_P	$(ML)^*$	$(Div+Int)^*$	DORIGO et GAMBARDELLA [1997]
LEM/LEMMO	G_P	$(Rule)^k$	$(Eval)^k$	MICHALSKI [2000]
Cultural Algorithm	$G_{S\,best}$	$(Rule)^{LO}$	$(Int)^*$	BECERRA et COELLO [2005]
LSM-DE	F_P	$\emptyset$	$(Param)^*$	WEDGE et KELL [2008]
Méta-modèle	G_P	$(Rule)^*$	$(Div+Int)^*$	KNOWLES et NAKAYAMA [2008]
DM-GRASP	G_P	$(ML)^1$	$(Init)^*$	SANTOS et collab. [2008]
GAPH	F_I	$(Rule)^*$	$(Init)^*$	ARAÚJO et NAGANO [2010]
Q-Learning JSSP	$G_{S\,cur}$	$(QL)^*$	$(Int)^*$	MARTÍNEZ et collab. [2011]
PLS-SAT	$G_{S\,best}$	$(Rule)^{LO}$	$(Init)^*$	ARBELAEZ et HAMADI [2011]
VEGAS	$F_{S\,cur}$	$(MAB)^*$	$(Select)^*$	MARMION et collab. [2011]
PSO	F_P	$(Rule)^*$	$(Int)^*$	ZEUGMANN et collab. [2011]
LTS	$G_{S\,best}$	$(ML)^k$	$(Div+Int)^*$	SCHINDL et ZUFFEREY [2013]
L-VNS	$G_{S\,best}$	$(ML)^{LO}$	$(Init)^*$	LALLA-RUIZ et VOSS [2015]
AOS	F_P	$(MAB)^{LO}$	$(Div)^*$	LI et collab. [2014]
Reactive-VNS	$F_{S\,best}$	$(QL)^{LO}$	$(Div)^*$	DOS SANTOS et collab. [2014]
Critère d'arrêt	P_I	$(Rule)^1$	$(Stop)^*$	ARBELAEZ et O'SULLIVAN [2016]
Hyper-heuristique	P_I	$(Rule)^1$	$(Param)^1$	KANDA et collab. [2016]
Meta-RASP	$G_{S\,best}$	$(QL)^{LO}$	$(Init)^*$	ARIN et RABADI [2017]
Parametrage	$F_{S\,best}$	$(ML)^{LO}$	$(Div)^*$	BENLIC et collab. [2017]

TABLEAU 2.4 – Classification des approches d'intégration de connaissances de la littérature classée dans l'ordre chronologique

Chapitre 3

Intégration de connaissances dans les procédures d'initialisation

Sommaire

3.1 Introduction

Nous avons vu dans le chapitre précédent que le mécanisme d'intégration de connaissances se décompose en trois parties : Extraction, Mémorisation et Intégration et peut-être utilisé à différents endroits de la recherche. Dans ce chapitre, nous allons commencer par présenter des méthodes d'intégration de connaissances sans mémorisation, à utiliser en tant que mécanisme d'initialisation, appliquées au problème du Flowshop sans temps d'attente (NWFSP) présenté au Chapitre 1.

Pour rappel, le NWFSP (No-Wait Flowshop Problem) impose que les opérations soient traitées sans interruption entre machines consécutives. Cette contrainte introduit des caractéristiques intéressantes que nous proposons d'exploiter dans la conception de méthodes d'intégration de connaissances de ce chapitre en tant qu'heuristiques constructives utilisées généralement comme procédure d'initialisation dans des méthodes plus sophistiquées. Ces heuristiques constructives s'apparentent à des algorithmes gloutons où la construction d'une solution est réalisée par une règle de construction spécifique au problème. En accord avec la taxonomie du chapitre précédent, la classification de ces méthodes est donc $\emptyset$/Rule/Init. Notons aussi que pour le NWFSP, il existe plusieurs objectifs possibles à considérer. Dans ce chapitre nous allons en étudier deux : le Makespan et le Flowtime, deux objectifs à minimiser.

Afin de définir une règle de construction pour chacun des deux objectifs, il faut d'abord extraire de la connaissance. Ainsi, dans ce chapitre, nous allons proposer deux approches. La première est une approche *génotypique*, où une étude sur la structure des solutions a permis de concevoir une heuristique constructive efficace pour l'objectif du Makespan pour le NWFS. La seconde est une approche *phénotypique*, où l'extraction des mesures de GAP entre les tâches a permis d'élaborer une heuristique constructive pour l'objectif du Flowtime pour le NWFS.

Dans ce contexte, l'objectif de ce chapitre est de développer plusieurs méthodes heuristiques pour le problème d'ordonnancement de flowshop sans temps d'attente. Ainsi, les contributions sont les suivantes :

1. Analyse de caractéristiques du problème avec une approche phénotypique.
2. Analyse de caractéristiques du problème avec une approche génotypique.
3. Intégration de ces observations dans la conception de nouvelles heuristiques constructives.
4. Étude de l'intérêt de l'approche proposée lorsqu'elle est utilisée seule ou comme initialisation d'une métaheuristique.

Ce chapitre est organisé comme suit : la section 3.2 présente les heuristiques constructives de la littérature pour le NWFSP utilisées dans ce chapitre. La nouvelle heuristique basée sur l'approche génotypique est détaillée dans la section 3.3 puis ses performances sont évaluées dans la section 3.4. Cette heuristique a fait l'objet d'une publication dans la conférence LION 2017 (MOUSIN et collab. [2017b]). L'heuristique basée sur l'approche phénotypique est détaillée dans la section 3.5 puis ses performances sont évaluées dans la section 3.6.

3.2 Heuristiques constructives pour le problème d'ordonnancement du No-Wait Flowshop

De nombreuses heuristiques et métaheuristiques ont été proposées dans la littérature pour résoudre les problèmes d'ordonnancement et en particulier la variante du problème d'ordonnancement de type Flowshop sans temps d'attente. Une revue de la littérature de ces méthodes a été présentée dans le Chapitre 1. Dans cette section, nous nous concentrerons sur des heuristiques spécifiques pour les deux objectifs traités, à savoir le Makespan et le Flowtime, puis nous proposerons dans les sections suivantes nos propres heuristiques constructives pour chacun des objectifs.

Avant de détailler les heuristiques constructives pour chaque objectif, nous pouvons faire une remarque sur la composition d'une heuristique constructive en nous basant sur l'article de FRAMINAN et collab. [2004] qui propose un schéma d'une heuristique constructive selon trois phases :

(i) Ordre initial : Les tâches sont organisées selon une certaine propriété basée sur les données d'une instance du problème (Exemple : l'ordre croissant des temps d'exécution de chaque tâche).

(ii) Construction de la solution : Une solution est construite de façon récursive en essayant d'insérer une ou plusieurs tâches non-ordonnancées dans une ou plusieurs positions d'un ordonnancement partiel jusqu'à ce que l'ordonnancement soit complet.

(iii) Amélioration de la solution : À partir de la solution construite, l'heuristique essaye de changer la séquence des tâches pour améliorer la solution. Ces approches améliorantes sont généralement des recherches locales de descente (*Hillclimbing*).

Ainsi une heuristique constructive pour le NWFSP peut être définie en décrivant simplement ces deux premières étapes. La troisième étape étant rarement définie dans la littérature, car cette dernière peut-être remplacée par n'importe quelle recherche locale améliorante.

Pour une bonne compréhension des heuristiques présentées dans cette section, une mesure spécifique du NWFSP doit être présentée, la mesure du GAP.

3.2.1 Une mesure spécifique au NWFSP : le GAP

NAGANO et ARAÚJO [2013] définissent le GAP entre deux tâches i et j sur une machine k comme le temps mort entre les deux tâches i et j sur la machine k (Exemple : sur la Figure 3.1, où le GAP entre les tâches J_5 et J_2 sur la machine M_4 équivaut à 3). Comme indiqué au Chapitre 1, l'équation suivante donne le calcul de la mesure du GAP :

$$\mathrm{GAP}^k_{i,j} = d_{i,j} + \sum_{h=1}^{k-1} (p_{j,h} - p_{i,h}) - p_{i,k} \tag{3.1}$$

De plus, nous proposons de définir le $\mathrm{SGAP}(i,j)$ entre deux tâches i et j comme le temps mort total entre deux tâches i et j pour l'ensemble des machines. Notons

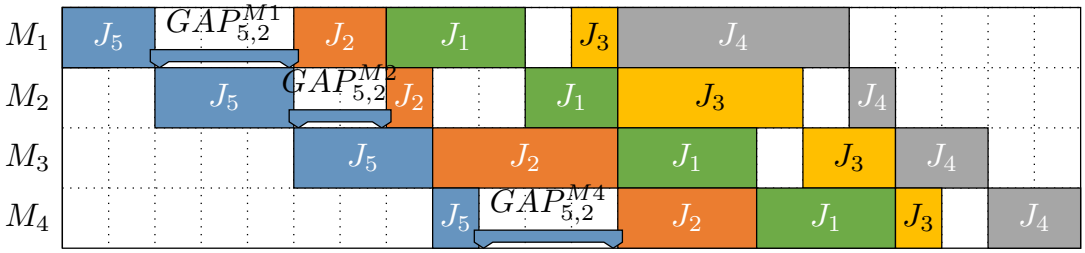


FIGURE 3.1 – Exemple montrant le GAP entre les tâches 5 et 2

que grâce à la particularité du Flowshop sans temps d'attente, le SGAP(i, j) entre deux tâches successives i et j est constant, quelle que soit leur position dans la solution. À titre d'exemple, sur la Figure 3.1, le SGAP(i, j) entre les tâches J_5 et J_2 correspond au calcul suivant : $\text{SGAP}(5,2) = \text{GAP}_{M_1,5,2} + \text{GAP}_{M_2,5,2} + \text{GAP}_{M_3,5,2} + \text{GAP}_{M_4,5,2} = 3+2+0+3 = 8$. Le SGAP s'obtient par l'équation suivante :

$$\text{SGAP}(i,j) = \sum_{k=1}^{M} \text{GAP}^{k}_{i,j} \tag{3.2}$$

Afin d'étudier plus facilement le GAP qu'une tâche peut générer avec toutes les autres, nous proposons de définir le $\text{GAP}_{total}(i)$ comme la somme des SGAP(i, j) d'une tâche i suivie par une autre tâche j, pour toutes les tâches j. Ainsi le GAP_{total} est défini comme suit :

$$\text{GAP}_{total}(i) = \sum_{\forall j \neq i} \text{SGAP}(i,j) \tag{3.3}$$

Enfin, notez que pour des raisons de facilité de lecture de ce chapitre, le GAP entre deux tâches i et j désignera toujours le SGAP(i, j). En effet, c'est le GAP généré sur l'ensemble des machines qui nous intéresse principalement.

3.2.2 Heuristiques pour minimiser le Makespan

La première heuristique considérée est l'heuristique bien connue NEH de NAWAZ et collab. [1983] proposée initialement pour le FSP classique qui s'applique avec succès sur la variante No-Wait. A l'inverse, d'autres heuristiques constructives ont été conçues spécifiquement pour le NWFSP. Citons entre autres RAJ de GANGADHARAN et RAJENDRAN [1993], BIH/BAH de BIANCO et collab. [1999] et LCH de LAHA et CHAKRABORTY [2008] Toutes ces heuristiques définissent l'ordre dans lequel les tâches sont considérées en fonction d'un critère (par exemple l'ordre décroissant des temps d'exécution de chaque tâche : Temps d'exécution de la tâche $i = \sum_m p_{i,m}$, où $p_{i,m}$ est le temps d'exécution de l'opération de la tâche i sur la machine m).

Ces heuristiques construisent des solutions de bonne qualité qui peuvent être ensuite améliorées par une métaheuristique. Dans ce chapitre, nous nous concentrons sur NEH et BIH car, non seulement elles fournissent des solutions de très bonne qualité

mais sont en plus très rapides. Ces deux heuristiques seront utilisées plus tard dans ce chapitre pour la comparaison avec l'heuristique proposée pour minimiser le Makespan.

Le principe de l'heuristique NEH est de construire itérativement une séquence de tâches π d'un ensemble de tâches J. Premièrement, les n tâches de J sont triées dans l'ordre décroissant de leur somme des temps d'exécution de chaque opération sur chaque machine. Ensuite, à chaque itération, la première tâche non planifiée restante dans l'ordre défini à l'étape précédente est insérée dans la séquence partielle π afin de minimiser la fonction objectif partielle. L'algorithme 3.1 décrit NEH.

Algorithme 3.1 : NEH

Entrées : Ensemble J de n tâches
Sorties : π la séquence des tâches ordonnancées
$\pi = \emptyset$;
`// (i) Création de l'ordre initial`
Trier l'ensemble J dans l'ordre décroissant des temps d'exécution;
`// (ii) Insertion`
pour $k = 1$ ***à*** n **faire**
    Insérer la tâche J_k dans π à la position qui minimise le Makespan partiel;
retourner π

Une fois le tri réalisé, c'est une heuristique nécessitant seulement n itérations pour construire une solution. À chaque itération, une tâche J_k est insérée. Le Makespan de la nouvelle solution partielle peut être calculé avec une complexité de $O(n)$ grâce au délai, lorsque la tâche J_k est insérée à la première position (voir section 1.3.1). Les solutions partielles, dans lesquelles la tâche J_k est insérée à une autre position dans la solution, sont voisines de la précédente donc, leur makespan peut être calculé avec une complexité de $O(1)$, selon la propriété exposée à la fin de Section 1.3.1. Ainsi, la complexité d'une itération de NEH est $O(n)$. Par conséquent, la complexité de l'exécution entière de NEH est $O(n^2)$.

La seconde heuristique considérée, BIH (Best Insertion Heuristic), est une heuristique constructive basée sur la meilleure insertion dans une solution partielle en considérant chacune des tâches non ordonnancées (contrairement à NEH qui ne considère qu'une tâche à la fois à insérer). Pour ce faire, elle va tester l'insertion de toutes les tâches non encore ordonnancées à toutes les positions de la solution partielle pour trouver le meilleur Makespan partiel. L'algorithme 3.2 présente BIH.

Si nous considérons l'ordonnancement d'une tâche comme une itération de l'algorithme, BIH construit une solution en n itérations seulement. Cependant, à chaque itération, chaque tâche est un candidat à insérer dans l'une des positions possibles pour trouver celui qui minimise le makespan partiel. Pour chaque tâche, trouver la position qui minimise le makespan partiel a une complexité maximale de $O(n)$ donc, la complexité d'une itération de BIH est $O(n^2)$. Par conséquent, la complexité de l'exécution entière de BIH est $O(n^3)$. BIH a été proposée dans la littérature après NEH afin de construire des solutions de meilleure qualité. Cependant, la complexité de BIH est en

Algorithme 3.2 : BIH

Entrées : Ensemble J de n tâches
Sorties : π la séquence des tâches ordonnancées
$\pi = \emptyset$;
tant que $J \neq \emptyset$ **faire** `// (ii) Insertion`
  Trouver la tâche $k \in J$ et la position h dans la solution π, telle que l'insertion de k à la position h dans π minimise le makespan partiel;
  Insérer la tâche k à la position h dans la solution π;
  Retirer k de l'ensemble J;
retourner π

$O(n^3)$ alors que celle de NEH est $O(n^2)$. Ainsi, bien que toutes deux soient très rapides, NEH a une meilleure complexité que BIH. Ainsi pour les problèmes de grande taille, un compromis entre le temps de calcul et la qualité peut être nécessaire.

3.2.3 Heuristiques pour minimiser le Flowtime

Concernant le critère du Flowtime (somme des dates de fin des tâches), différentes heuristiques ont également été proposées dans la littérature : L'heuristique de la meilleure insertion (BIH) de Bianco et collab. [1999], l'heuristique 1 proposée par Aldowaisan et Allahverdi [2004] (PH1) et l'heuristique basée sur le GAP de Nagano et Araújo [2013]. Ces heuristiques permettent de fournir des solutions de bonne qualité tout en étant rapides. L'heuristique BIH étant la même que celle présentée précédemment, mais en considérant l'objectif du Flowtime, elle ne sera pas ré-présentée dans cette section.

PH1 est une heuristique qui génère un ordre initial (i) en insérant à la dernière position la tâche non-ordonnancée qui minimise le Flowtime partiel à chaque insertion. Cela permet d'obtenir un ordre initial des tâches en $O(n)$. Ensuite, cet ordre initial est utilisé pour la construction de la solution (ii) en utilisant le principe de la meilleure insertion possible pour minimiser le Flowtime partiel (comme l'étape (ii) de NEH). Enfin cette étape est répétée k fois (Fixé à 10 dans l'article de Aldowaisan et Allahverdi [2004]), en considérant la dernière solution obtenue comme nouvel ordre initial à considérer pour la phase de construction de solution. À la fin, la meilleure solution des k solutions obtenues est conservée. L'algorithme 3.3 présente PH1.

L'algorithme pour générer une séquence initiale s'exécute en $O(n^2)$, l'insertion d'une tâche à la meilleure position possible a une complexité maximale en $O(n)$ cependant il faut le faire pour toutes les tâches, donc la complexité d'une itération de l'étape (ii) est en $O(n^2)$. Finalement, l'exécution de PH1 a une complexité en $O(n^2 + n^2 * k)$ puisque l'étape (ii) est exécutée k fois.

Enfin la dernière heuristique GAPH est, tout comme BIH, aussi une heuristique proposée initialement pour minimiser le Makespan utilisant la mesure du GAP (cf : Section 1.3.1.3). Puisque l'heuristique que nous allons proposer pour minimiser le Flowtime est basée elle aussi sur la mesure du GAP, nous avons donc choisi d'adapter GAPH

Algorithme 3.3 : PH1

Entrées : Ensemble J de n tâches, k nombre de répétitions de l'étape (ii)
Sorties : π_{best} la séquence des tâches ordonnancées
$\pi = \emptyset$;
$\pi_{cur} = \emptyset$;
$\pi_{SI} = \emptyset$;
pour *i de 1 à n* **faire** `// (i) Création de l'ordre initial`
  Insérer à la fin de π_{SI} la tâche $j \in J$ non présente dans π_{SI} qui minimise le Flowtime;
$\pi = \pi_{SI}$;
$\pi_{best} = \pi$;
pour *i de 1 à k* **faire** `// (ii) Insertion`
  $\pi_{cur} = \emptyset$;
  pour *j de 1 à n* **faire**
    Insérer la tâche $\pi_{SI}[j]$ à la meilleure position dans π_{cur} tel que le Flowtime soit minimal.;
  si $F(\pi) < F(\pi_{best})$ **alors**
    $\pi_{best} = \pi$;
  $\pi_{SI} = \pi$;
retourner π_{best}

au Flowtime. GAPH est similaire à BIH, sauf qu'au lieu de minimiser le Flowtime, l'heuristique va minimiser le GAP entre toutes les tâches. En d'autres termes, cela revient à minimiser le SGAP total de la solution (Somme des SGAP de l'ordonnancement). La mesure du GAP est aussi étroitement liée au Flowtime. En effet, minimiser les temps morts entre chaque tâche revient d'une certaine manière à réduire le Flowtime. L'algorithme 3.4 présente GAPH.

Algorithme 3.4 : GAPH

Entrées : Ensemble J de n tâches
Sorties : π la séquence des tâches ordonnancées
$\pi = \emptyset$;
tant que $J \neq \emptyset$ **faire** `// (ii) Insertion`
  Trouver la tâche $k \in J$ et la position h dans la solution π, telle que l'insertion de k à la position h dans π minimise le SGAP total;
  Insérer la tâche k à la position h dans la solution π;
  Retirer k de l'ensemble J;
retourner π

Au vu des fortes similarités avec BIH, il est facile de conclure que la complexité de GAPH est $O(n^3)$. Néanmoins, l'évaluation d'une solution partielle n'est pas faite de la même manière. BIH calcule le Flowtime d'une solution partielle en $O(n^2)$ alors que le GAP est une mesure constante entre chaque tâche, ce qui permet une évaluation des

solutions de GAPH en O(n). Ainsi, GAPH peut être vu comme un BIH avec une fonction d'évaluation plus rapide.

3.3 Une nouvelle heuristique pour minimiser le Makespan : IBI

Dans cette section, nous proposons une heuristique constructive conçue pour minimiser l'objectif du Makespan grâce à l'intégration de connaissances génotypiques des solutions. Pour cela, une analyse de la structure des solutions optimales est nécessaire.

3.3.1 Analyse de la structure d'une solution optimale

Comme présenté dans la section 3.2, chacune des heuristiques commence par une solution vide, à laquelle elle ajoute de manière itérative une tâche en minimisant l'objectif partiel, jusqu'à ce que toutes les tâches soient ordonnancées. Dans cette section, nous allons analyser étape par étape, l'évolution de la solution optimale pour un ensemble de tâches d'une instance partielle. Afin de comprendre la dynamique de la construction d'une telle solution. Nous avons donc observé la structure des solutions optimales pour chaque taille d'instance partielle, puis nous avons observé les différences présentes entre les solutions de taille ± 1.

Soit $\mathscr{P}_k$, une instance du NWFSP ayant k tâches à ordonnancer. Pour k petit, il est possible de connaître la solution optimale puisque l'ensemble des solutions est énumérable en un temps raisonnable. Nous pouvons donc énumérer toutes les solutions possibles pour un $k \leq 12$. Une instance $\mathscr{P}_{k+1}$ est définie comme $\mathscr{P}_{k+1} = \mathscr{P}_k$+Nouvelle tâche, cela afin de garder à chaque fois les mêmes données pour chaque taille d'instance. Ainsi nous pouvons observer pour chaque taille d'instance l'évolution de la meilleure solution entre les différentes tailles. Observer les solutions optimales des ensembles $\mathscr{P}_k$ et $\mathscr{P}_(k+1)$ revient à étudier quelles sont les opérations nécessaires pour passer d'une solution optimale de taille k à une solution optimale de taille $k+1$, où juste une nouvelle tâche à été ajoutée.

Notre proposition est d'étendre les observations réalisées sur un ensemble de sous-problèmes de petite taille, afin d'obtenir une meilleure solution sur les problèmes de grandes tailles.

Le Tableau 3.1 décrit l'évolution de la solution optimale au fur et à mesure de l'ajout de nouvelles tâches. Nous observons que dans une majorité des cas, la structure de la solution est globalement conservée, donc que son génotype change peu. En effet, une insertion simple (en rouge dans le tableau) suffit pour garder la meilleure solution. Néanmoins pour $\mathscr{P}_6$, $\mathscr{P}_9$, $\mathscr{P}_{10}$ et $\mathscr{P}_{12}$, une à deux tâches ont eu besoin d'être déplacées par rapport à la solution du problème précédent pour obtenir la solution optimale.

On peut donc déduire qu'une stratégie d'insertion comme dans l'heuristique NEH s'éloignera de la solution optimale dès la 6^e itération pour cet exemple. Car au-delà, des ré-insertions de tâches ont été nécessaires peu à peu.

$\mathscr{P}$	Solution optimale												Makespan
$\mathscr{P}_1 = 1$	1												273
$\mathscr{P}_2 = 1,2$	1	2											352
$\mathscr{P}_3 = 1..3$	3	1	2										367
$\mathscr{P}_4 = 1..4$	3	1	4	2									471
$\mathscr{P}_5 = 1..5$	3	1	5	4	2								580
$\mathscr{P}_6 = 1..6$	3	6	5	4	2	1							628
$\mathscr{P}_7 = 1..7$	3	1	5	4	2	6	7						705
$\mathscr{P}_8 = 1..8$	3	8	1	5	4	2	6	7					749
$\mathscr{P}_9 = 1..9$	9	1	5	4	2	6	7	8	3				794
$\mathscr{P}_{10} = 1..10$	9	1	4	2	6	5	10	7	8	3			851
$\mathscr{P}_{11} = 1..11$	9	11	1	4	2	6	5	10	7	8	3		952
$\mathscr{P}_{12} = 1..12$	9	1	4	2	6	5	10	7	12	11	8	3	1021

TABLEAU 3.1 – Évolution de la solution optimale sur l'instance T*a*01 (020x05) avec les 12 premières tâches pour le Makespan. En rouge, la nouvelle tâche à ajouter par rapport l'itération précédente. En bleu, les tâches ayant été réinsérées ailleurs dans la solution optimale par rapport à l'itération précédente.

La figure 3.2 explicite ce phénomène pour le passage de $\mathscr{P}_8$ à $\mathscr{P}_9$. A partir de la solution optimale de $\mathscr{P}_8$, la 9^e tâche est insérée à la position qui minimise le Makespan du problème $\mathscr{P}_9$. En comparant la solution obtenue avec la solution optimale de $\mathscr{P}_9$, on peut observer qu'elles sont très proches. En effet, seules deux réinsertions améliorantes (réinsertions des tâches 3 et 8) sont nécessaires pour obtenir la solution optimale à partir de la solution construite itérativement.

$\mathscr{P}_8$		3		8	1	5	4	2	6	7			$C^*_{max} = 749$
$\mathscr{P}_9$	Insertion de la tâche 9	3	9	8	1	5	4	2	6	7			$C_{max} = 804$
	re-insertion de la tâche 3	9		8	1	5	4	2	6	7	3		$C_{max} = 795$
	re-insertion de la tâche 8	9			1	5	4	2	6	7	8	3	$C^*_{max} = 794$

FIGURE 3.2 – Exemple de l'évolution de la structure d'une solution optimale d'un problème $\mathscr{P}_8$ de taille 8 à un problème $\mathscr{P}_9$ de taille 9.

Cette étude a été réalisée sur les 30 premières instances de Taillard, de T*a*001 à T*a*030 (c'est-à-dire pour 5 à 20 machines, sur les 12 premières tâches de chaque instance), ce qui a confirmé les observations suivantes :

— L'insertion d'une nouvelle tâche produit une solution peu éloignée de la solution optimale du nouveau problème.

— La solution construite peut-être modifiée avec peu de ré-insertions pour obtenir la solution optimale.

3.3.2 Heuristique constructive utilisant des réinsertions partielles

Suite à ces observations, nous proposons une nouvelle heuristique constructive appelée Iterated Best Insertion (IBI). A chaque itération, deux phases sont successivement réalisées : (*a*) la première tâche non planifiée restante est insérée dans la solution partielle afin de minimiser le makespan partiel (même stratégie que NEH) et (*b*), une amélioration itérative est effectuée sur cette solution partielle pour réorganiser les tâches et afin d'être plus proche de la solution optimale du sous-problème (voir Algorithme 3.5). Une amélioration itérative (aussi appelée *descente*) est une méthode qui se déplace, en utilisant un opérateur de voisinage, d'une solution voisine à une autre améliorante, pour optimiser la qualité. Elle s'arrête naturellement lorsqu'un optimum local (une solution sans solution voisine améliorante) est atteint. Ici, l'opérateur de voisinage est basé sur l'insertion et lorsqu'un mouvement possible améliorant le Makespan est trouvé, il est appliqué.

Algorithme 3.5 : IBI

Entrées : Ensemble J de n tâches, σ critère pour trier les tâches J , $cycle$ nombre d'itérations sans amélioration
Sorties : π la séquence des tâches ordonnancées
$\pi = \emptyset$;
`// (i) Création de l'ordre initial`
Trier l'ensemble J par rapport à σ;
`// (ii) Insertion`
pour $k = 1$ ***à*** n **faire**
    Insérer la tâche J_k dans π à la position qui minimise le Makespan partiel;
    si $k \equiv cycle = 0$ **alors**
        Réaliser une descente sur π;
retourner π

Deux paramètres ont été introduits dans l'algorithme proposé. Le premier est le tri initial σ pour l'ensemble des tâches J ; il indique dans quel ordre les tâches sont considérées pendant la construction. Le deuxième, appelé $cycle$, représente le nombre d'itérations sans appliquer la procédure de descente. En effet, même si l'analyse expérimentale montre que la séquence construite en phase (*a*) n'est pas très éloignée de l'optimum du sous-problème, on sait que le temps d'exploration du voisinage croît avec la taille du problème. Afin de contrôler la performance de IBI et du temps de calcul nécessaire, le paramètre $cycle$ permet à la descente d'être exécutée à un nombre régulier d'itérations uniquement.

3.4 Evaluation des performances de l'heuristique IBI

Le tri initial σ de IBI et le paramètre du $cycle$ peuvent avoir un impact sur les performances de IBI. Ainsi les performances de IBI sont évaluées en fonction de ces deux

paramètres. Cette partie présente d'abord le protocole expérimental utilisé, puis compare plusieurs versions de l'algorithme IBI, en utilisant plusieurs tris initiaux et enfin analyse l'impact du paramètre cycle.

3.4.1 Protocole Expérimental

Dans une première série d'expérimentations, nous allons tester différents tris initiaux pour déterminer quel tri sera le meilleur. Ensuite, dans une seconde série d'expérimentations, c'est l'influence de la taille du cycle qui sera testée. Une fois ces deux paramètres fixés, IBI sera comparé aux heuristiques de la littérature, en particulier NEH et BIH. Et finalement, nous comparerons les performances d'IBI en tant qu'initialisation d'une recherche locale.

L'exécution d'un algorithme sur une instance I retourne une solution π de qualité $C_{max}(\pi)$, le Makespan de la solution. Pour mesurer la qualité d'une solution, la déviation relative moyenne (*Relative Percentage Deviation - RPD*) est calculée par rapport à la meilleure solution connue (π^*) de littérature comme suit :

$$RPD(\pi) = \frac{C_{max}(\pi) - C_{max}(\pi^*)}{C_{max}(\pi^*)} * 100 \tag{3.4}$$

En raison de sa phase de descente, IBI est une méthode stochastique. Ainsi, 30 exécutions sont nécessaires pour permettre de tirer des conclusions sur les résultats obtenus. Enfin, toutes les variantes de l'algorithme sont exécutées 30 fois par instance et le test statistique de Friedman est effectué sur le RPD de toutes les exécutions pour comparer les algorithmes.

3.4.2 Expérimentations sur les paramètres d'IBI

3.4.2.1 Tri initial σ

Les heuristiques gloutonnes, telles que IBI, considèrent les tâches dans un ordre donné. Par exemple, dans NEH, les tâches sont triées dans l'ordre des sommes décroissantes des temps de traitement.

Parmi les mesures possibles pour réaliser un tri des différentes tâches, nous avons les temps d'exécutions des tâches, comme utilisés dans NEH, mais nous avons aussi les mesures de GAP définies dans la section 3.2.1. En particulier, nous allons ici utiliser le GAP_{total} qui peut-être une mesure intéressante. En effet, une valeur élevée du GAP_{total} pour une tâche indique que la tâche génère beaucoup de temps mort avec de nombreuses tâches et, de ce fait, qu'elle n'est probablement pas facile à insérer dans l'ordonnancement sans créer beaucoup de temps mort. À l'inverse, un GAP_{total} faible indique que la tâche peut s'insérer facilement dans une solution, car génère peu de temps mort pour une majorité de tâches.

Afin d'évaluer l'impact du tri initial et de définir le plus efficace pour l'heuristique IBI, nous expérimentons plusieurs tris initiaux σ, qui sont les suivants :

— N*oSort* : Aucun tri n'est réalisé, les tâches sont prises dans l'ordre donné par l'instance.

— D*ecr*GAP$_{total}$: Tri par valeur de GAP$_{total}$ décroissante.

— I*ncr*GAP$_{total}$: Tri par valeur de GAP$_{total}$ croissante.

— D*ecr*PI : Tri par valeur des durées d'exécution des tâches (somme des durées d'exécution des opérations) décroissantes.

— I*ncr*PI : Tri par valeur des durées d'exécution des tâches croissantes.

Le tableau 3.2 donne le RPD moyen sur 10 instances d'une même taille, pour chaque taille des instances de Taillard obtenu par différents tris initiaux sur les 110 instances de Taillard. Pour les instances de taille 20_10,20_20, 50_10, 50_20, 100_10 et 100_20, aucune différence significative n'est observée entre les différents tris. En revanche, pour les instances de tailles 20_5, 50_5, 100_5, 200_10 et 200_20, une différence significative est observée en ce qui concerne les tris considérant la somme des durées d'exécution. Cependant, que le tri soit par ordre croissant ou décroissant n'a pas d'influence significative. Ainsi, pour la suite des expérimentations, nous choisissons la somme croissante des durées d'exécution comme tri initial de IBI.

Instance	N*oSort*	D*ecr*GAP$_{total}$	I*ncr*GAP$_{total}$	D*ecr*PI	I*ncr*PI
20_5	**1.79**	2.32	2.57	2.13	**1.38**
20_10	2.04	1.72	1.67	1.30	1.77
20_20	1.44	1.26	1.04	1.12	1.15
50_5	3.74	3.65	4.01	**3.23**	**3.04**
50_10	2.78	3.17	3.25	2.74	2.57
50_20	2.53	2.41	2.47	2.80	2.50
100_5	4.68	4.52	4.67	**4.13**	**4.12**
100_10	3.45	3.36	3.53	3.24	3.27
100_20	3.05	2.82	2.81	2.79	2.89
200_10	4.32	4.19	4.22	**3.81**	**3.87**
200_20	3.24	3.26	3.23	**3.03**	**2.99**

TABLEAU 3.2 – RPD moyen obtenu pour chaque taille des instances de Taillard pour les différents tris. Les valeurs RPD en gras représentent les algorithmes qui surpassent les autres selon le test statistique de Friedman.

Maintenant que nous avons fixé le tri initial, il reste à déterminer la valeur du paramètre $cycle$.

3.4.2.2 Cycle

L'utilisation d'une descente à chaque itération peut être coûteuse en temps de calcul. En effet, une descente consiste à évaluer le voisinage d'une solution, or la taille du voisinage va croître avec la taille de solution, et donc au fur et à mesure de l'ajout des

tâches dans la solution, la descente sera de plus en plus coûteuse en temps de calcul. Afin de minimiser le temps de calcul, nous introduisons une notion de cycle. Un cycle est une séquence de x itérations sans descente.

Dans le Tableau 3.3, nous étudions l'impact de la taille du cycle sur la qualité des solutions et le temps d'exécution.

Instance	1		2		5		10		n	
	RPD	Temps	RPD	Temps	RPD	Temps	RPD	Temps	RPD	Temps
20_5	1.38	0.77	1.24	0.55	1.43	0.27	1.78	0.20	1.62	0.18
20_10	1.77	0.83	1.82	0.52	1.85	0.26	1.49	0.20	1.69	0.19
20_20	1.15	0.77	1.35	0.47	1.71	0.25	1.74	0.19	1.68	0.16
50_5	**3.04**	12.16	**3.25**	6.76	3.46	3.52	3.62	2.49	3.94	1.74
50_10	2.57	11.55	2.49	6.69	2.75	3.62	3.07	2.39	3.26	1.48
50_20	2.50	11.73	2.37	6.67	2.58	3.58	2.65	2.42	3.14	1.59
100_5	**4.12**	95.74	4.22	55.96	4.42	30.22	4.65	19.40	5.29	9.62
100_10	**3.27**	93.43	**3.27**	54.56	3.30	29.19	3.45	18.78	4.17	9.71
100_20	**2.89**	92.90	**2.93**	53.60	**3.07**	28.35	3.16	18.28	3.56	9.88
200_10	**3.87**	755.09	**3.91**	435.40	4.07	225.52	4.09	146.26	4.74	55.04
200_20	**2.99**	746.62	3.10	431.33	3.11	222.80	3.18	146.26	3.93	57.83

TABLEAU 3.3 – RPD moyen et temps (millisecondes) obtenus sur les instances de Taillard pour différentes tailles de cycle. Les valeurs RPD en gras représentent les algorithmes qui surpassent les autres selon le test de Friedman. Un cycle de taille x indique que la descente est exécutée toutes les x itérations. Ainsi, un cycle de taille n indique que la descente n'est exécutée qu'une seule fois, à la fin de la construction.

Ces résultats montrent que la qualité diminue avec une grande taille de cycle, mais est évidemment plus rapide car les descentes sont exécutées moins souvent. On remarque aussi une différence significative pour les instance de taille 50_5, 100_5, 100_20, 200_10 et 200_20 pour une taille de cycle de 1 et 2.

Comme l'objectif d'une telle heuristique constructive est de fournir en un temps raisonnable une solution aussi bonne que possible et que le temps requis ici, même pour les grandes instances, reste raisonnable, nous proposons de ne pas utiliser de cycle, c'est-à-dire d'exécuter la descente à chaque étape de la construction. En conclusion de ces expériences nous proposons de fixer pour le reste des expérimentions les paramètres pour IBI suivant : un tri initial basé sur la somme croissante des durées d'exécution et un cycle de valeur 1.

Dans les deux sections suivantes, nous allons analyser l'efficacité de la méthode IBI proposée selon deux situations. Premièrement, IBI sera comparée à d'autres heuristiques constructives de la littérature, et dans un second temps ces différentes heuristiques seront utilisées comme initialisation d'une recherche locale classique. Ces sections suivent le même protocole expérimental que précédemment utilisé, et les para-

mètres utilisés pour IBI sont ceux résultant des expériences menées dans cette section sur les paramètres.

3.4.3 Comparaison d'IBI avec des heuristiques de la littérature

Nous avons choisi de comparer IBI avec deux autres heuristiques de la littérature : NEH et BIH. En effet, comme exposé dans la section 3.2 NEH et BIH sont des heuristiques constructives intéressantes et efficaces pour le NWFSP :

— NEH est une heuristique classique pour les problèmes de flow-shop et a l'avantage d'être très rapide. De plus, la méthode IBI proposée peut être considérée comme une amélioration de NEH, car une descente est ajoutée à chaque étape.

— BIH est une heuristique classique pour le NWFSP et très efficace car, d'après nos expériences préliminaires, elle offre les meilleures performances parmi plusieurs heuristiques testées.

Le tableau 3.4 montre l'étude comparative entre IBI et les deux autres heuristiques constructives. Le RPD et le temps de calcul sont donnés pour montrer le gain de qualité en ce qui concerne le temps de calcul. NEH et BIH sont déterministes et ainsi, les deux valeurs indiquées pour chaque taille d'instance sont les moyennes calculées à partir des dix valeurs RPD obtenues pour les 10 instances respectivement. De plus, les valeurs d'IBI correspondent aux valeurs moyennes des 30 exécutions et des 10 instances de chaque taille d'instance.

Instance	NEH		BIH		IBI	
	RPD	Temps	RPD	Temps	RPD	Temps
20_5	3.95	0.03	3.03	0.10	**1.38**	0.77
20_10	4.18	0.02	3.60	0.09	**1.77**	0.82
20_20	2.92	0.02	**1.74**	0.08	**1.15**	0.76
50_5	6.84	0.10	5.98	1.10	**3.04**	12.16
50_10	4.59	0.09	4.10	1.13	**2.57**	11.54
50_20	4.84	0.10	4.01	1.18	**2.50**	11.72
100_5	8.14	0.32	6.85	9.07	**4.12**	95.74
100_10	6.10	0.33	5.66	9.29	**3.27**	93.43
100_20	5.17	0.33	4.65	9.30	**2.89**	92.90
200_10	6.72	1.21	5.85	72.44	**3.87**	755.09
200_20	5.74	1.21	4.51	71.95	**2.99**	746.62

TABLEAU 3.4 – RPD et temps (millisecondes) obtenus sur les instances de Taillard pour NEH, BIH et IBI (valeurs moyennes). Les valeurs de RPD en gras représentent les algorithmes surpassant statistiquement les autres selon le test de Friedman.

Les solutions construites par IBI ont une meilleure qualité (RPD plus petite) que celles construites par NEH ou BIH. Cela a été statistiquement vérifié avec le test de Friedman. Plus précisément, IBI est meilleur que NEH ou BIH pour 106 instances sur les 110

instances disponibles (Voir Annexe B pour plus de résultats). La contrepartie de cette performance concerne le temps de calcul requis pour exécuter IBI contrairement aux deux autres heuristiques. Cependant, cette durée reste raisonnable car moins d'une seconde est nécessaire même pour les instances de plus grandes tailles. IBI est donc une alternative efficace aux heuristiques classiques utilisées pour construire des solutions de bonne qualité.

3.4.4 IBI comme initialisation d'une recherche locale

Généralement, les heuristiques constructives sont utilisées comme une phase d'initialisation pour les méta-heuristiques. Afin d'évaluer la pertinence d'utiliser IBI dans une telle phase d'initialisation, l'heuristique a été combinée avec une recherche tabou (TS). Cette recherche locale est largement utilisée pour résoudre les problèmes de flowshop (GRABOWSKI et PEMPERA [2007]; GRABOWSKI et WODECKI [2004]; WANG et collab. [2010], ...), et est donc un bon candidat pour montrer la pertinence de construire des solutions avec IBI plutôt qu'avec NEH ou BIH.

Les trois heuristiques sont donc combinées avec la recherche tabou (TS). Pour être équitable, chaque approche combinée a le même temps total d'exécution fixé à $1000 * n$ millisecondes (avec n le nombre de tâches). Ces expérimentations visent à vérifier si la qualité atteinte est suffisante pour justifier l'utilisation d'IBI en tant qu'initialisation ou si son temps d'exécution *supérieur* pénalise son utilisation.

Le tableau 3.5 présente les résultats sur les valeurs de RPD moyen parmi les 30 exécutions pour chaque approche combinée à la recherche tabou pour chaque taille d'instance.

Instance	IBI	NEH+TS	BIH+TS	IBI+TS
20_5	1.38	1.12	1.30	0.83
20_10	1.77	1.30	1.35	1.39
20_20	1.15	1.15	0.71	0.73
50_5	3.04	3.17	3.49	2.63
50_10	2.57	2.68	2.53	2.15
50_20	2.50	2.53	2.41	2.03
100_5	4.12	4.66	4.55	**3.78**
100_10	3.27	3.70	3.48	**2.94**
100_20	2.89	2.97	2.93	2.62
200_10	3.87	4.25	4.20	**3.67**
200_20	2.99	3.56	**2.99**	**2.77**

TABLEAU 3.5 – RPD moyen sur les instances de Taillard pour NEH + TS, BIH + TS et IBI + TS. Les valeurs de RPD en gras représentent les algorithmes surpassant statistiquement les autres selon le test de Friedman.

Les résultats exposés dans ce tableau par rapport à ceux du tableau 3.4 indiquent

d'abord que la recherche tabou parvient à améliorer les solutions produites par les différentes heuristiques. Ils indiquent également que IBI utilisée comme une phase d'initialisation, obtient de meilleurs résultats que les autres, même si la différence n'est pas toujours statistiquement significative. En particulier pour les instances avec un nombre élevé de tâches à ordonnancer, IBI montre de très bon résultats

A titre d'illustration, la figure 3.3 montre l'évolution du makespan moyen pour une instance de 200 tâches et 20 machines (moyenne des 30 exécutions).

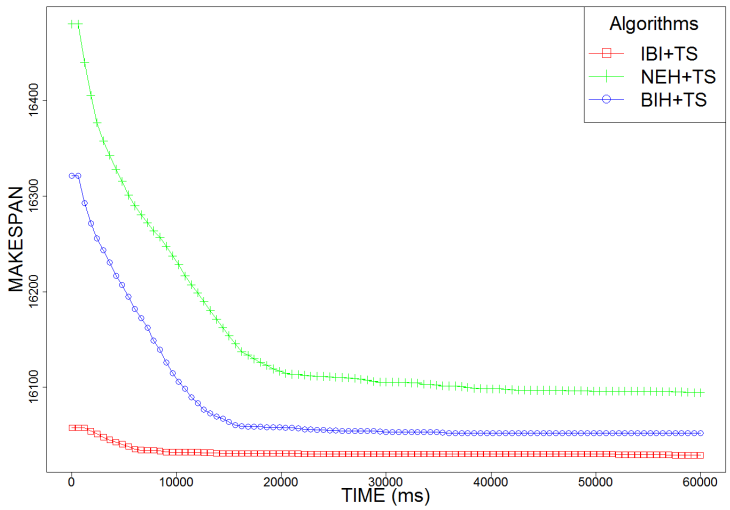


FIGURE 3.3 – Evolution de la moyenne du Makespan pour 30 exécutions sur la 7ème instance de Taillard (200 tâches, 20 machines) pour les trois approches combinées. IBI + TS est représenté par des carrés, NEH + TS par des croix et BIH + TS par des cercles.

En ce qui concerne la qualité après l'initialisation, IBI est meilleure que les deux autres heuristiques, comme montré précédemment dans le Tableau 3.4. Après la recherche tabou pour les deux autres heuristiques, la qualité finale semble être équivalente à celle obtenue par IBI sans appliquer de recherche locale. Cela montre l'efficacité de cette approche. Enfin, la recherche tabou n'aide pas beaucoup IBI. En effet, grâce aux descentes successives d'IBI, la solution fournie par IBI est déjà un optimum local. Par conséquent, il est difficile pour une recherche tabou d'améliorer la solution obtenue avec IBI.

En conclusion, ces expériences montrent la bonne performance de IBI pour les deux aspects : comme une heuristique constructive et comme initialisation d'une méta-heuristique. En particulier, ils montrent l'efficacité de l'utilisation d'une descente dans une approche constructive.

3.5 Une nouvelle heuristique pour minimiser le Flowtime : Incremental GAP

Dans cette section, nous proposons une heuristique constructive conçue pour minimiser l'objectif du Flowtime grâce à la connaissance phénotypique des solutions. À partir d'une analyse de caractéristiques phénotypiques des solutions optimales de différentes instances de petite taille, nous allons montrer que la mesure de GAP entre les tâches pouvait être utilisée dans la construction de solutions de bonne qualité.

3.5.1 Analyse du GAP d'une solution optimale

Comme vu à la section 3.2, les heuristiques constructives cherchent la meilleure insertion possible d'une tâche non encore ordonnancée dans la solution partielle en cours de construction. Ces insertions sont ainsi faites en suivant certaines règles en fonction de l'heuristique considérée pour optimiser l'objectif partiel.

Afin de définir une règle de construction d'une solution pour le Flowtime, nous avons analysé des solutions optimales et observé différentes mesures liées à ces solutions. Nous avons focalisé notre étude sur des instances de petite taille (nombre de tâches inférieur à 12), ce qui permet d'obtenir facilement les solutions optimales. Nous avons supposé que les hypothèses émises sur les instances de petite taille restent vraies sur les instances de plus grandes tailles. En effet, à partir d'un certain nombre de tâches, les méthodes exactes ne permettent plus d'obtenir la solution optimale, ainsi au-delà d'une certaine taille, nous n'avons pas de garantie d'optimalité sur les meilleures solutions connues. Puisque la qualité optimale des solutions de taille 20 pour les instances de Taillard pour l'objectif du Flowtime est connue, c'est donc celles-ci que nous avons décidé d'analyser.

Afin d'expliquer l'analyse que nous avons menée, la voici illustrée sur un exemple dans le Tableau 3.6 qui représente la solution optimale pour l'instance de Taillard T*a*01 (20 tâches et 5 machines). Nous avons observé cette solution comme lors d'un processus de construction. C'est-à-dire en considérant une à une chaque tâche dans l'ordre de la solution. Ainsi, nous avons observé à chaque étape, si le GAP entre la tâche considérée et la suivante était minimale, où s'il existait une tâche suivante telle que le GAP puisse être plus petit. Dans le cas où le GAP était effectivement minimal, le GAP est marqué en gras dans le tableau. Dans le cas contraire, nous donnons le GAP minimal possible parmi les tâches suivantes possibles, c'est-à-dire non encore considérée.

Comme on peut le constater dans ce tableau, une tâche i est généralement suivie par une tâche j telle que le GAP entre i et j soit minimal. Ainsi dans l'exemple ci-dessus, la tâche qui minimise le GAP pour la première tâche, c'est-à-dire, celle qui va démarrer le plus rapidement sur toutes les machines est la tâche 3, le GAP est plus élevé pour la première tâche, car il n'y a aucune autre tâche qui la précède, les machines sont donc en temps mort en attendant l'exécution de la première tâche. Ensuite, la tâche qui minimise le GAP en étant précédée par la tâche 3 est la tâche 17. De même que la tâche 9 est celle qui minimise le GAP avec la tâche 17, et ainsi de suite.

Il y a cependant quelques exceptions, notamment entre les couples (13;1), (1;2),

Solution	3	17	9	15	14	8	16	13	1	2
GAP	**222**	**84**	**63**	**28**	**63**	**68**	**41**	**76**	114 (101)	68 (48)
	6	10	7	20	12	11	19	4	5	18
GAP	**84**	97 (86)	**58**	181 (180)	**110**	**75**	**203**	**258**	**180**	**133**

TABLEAU 3.6 – GAP entre une tâche et la précédente de la solution optimale pour l'instance T*a*01 (020x05) de Taillard. Le GAP est en gras lorsque celui-ci est minimal par rapport aux autres tâches. Entre parenthèse, le GAP minimal possible en choisissant une autre tâche parmi les tâches non encore considérées de l'ordonnancement.

(6;10) ainsi que (7;20). Pour ces quelques exceptions, nous pouvons constater que le GAP minimal et le GAP observé sont relativement proches.

Cette étude a été réalisée sur plusieurs instances de Taillard avec des tailles différentes (50, 100 et 200 tâches), ce qui a confirmé les observations suivantes :

(i) La première tâche est généralement celle avec le plus petit GAP de départ, c'est-à-dire qui minimise le temps de démarrage des machines.

(ii) Chaque tâche est généralement suivie de celle qui minimise le GAP

(iii) Si l'observation (ii) n'est pas respectée par une tâche, alors c'est généralement la tâche qui possède le second plus petit GAP qui la suit.

3.5.2 Heuristique constructive utilisant le GAP

Suite aux observations précédentes, nous proposons une nouvelle heuristique constructive appelée Incremental GAP (IncrGAP). À chaque itération, l'heuristique ajoute à la fin de la solution courante, la tâche i qui minimise le GAP par rapport à la dernière tâche de la solution courante ($last$) (voir Algorithme 3.6). En cas d'égalité de GAP, c'est la tâche avec la plus petite somme des temps d'exécution qui est choisie.

Algorithme 3.6 : Incremental GAP (IncrGAP)

Entrées : π = solution courante, J = liste des tâches restantes non ordonnancées
Sorties : π = solution courante
Données : $last$ = dernière tâche de la solution courante
si J ≠ ∅ **alors**
  retourner π
$i = argmin_{k \in J} \text{GAP}_{last,k}$; // (i) Ordre
retourner $Incr\text{GAP}(\pi + i, J \setminus i)$; // (ii) Insertion

Nous allons illustrer le fonctionnement de l'algorithme avec les mêmes données que l'exemple de la Figure 3.1. Ainsi, les données utilisées pour les explications sont celles présentées dans le Tableau 3.7.

En considérant l'algorithme IncrGap, la tâche initiale à considérer est la tâche 4. Ensuite en se référant à la ligne J4 de la matrice des GAP, on observe que la tâche 2 a le plus

	J1	J2	J3	J4	J5
M1	12	1	5	12	19
M2	4	15	16	4	6
M3	2	16	19	11	3
M4	18	4	7	18	9
$Total$	36	36	47	45	37

(a) Matrice des temps d'exécutions de chaque tâche sur chaque machine.

	J1	J2	J3	J4	J5
$Start$	18	32	40	17	28
J1	-	27	32	21	28
J2	33	-	25	34	19
J3	79	26	-	52	65
J4	54	9	14	-	40
J5	13	36	45	22	-

(b) Matrice de GAP obtenue avec les données de l'exemple. La ligne d'indice $Start$ donne le GAP des tâches de 1 à 5 en début d'ordonnancement.

TABLEAU 3.7 – Instance construite pour les exemples.

petit GAP (9). Ainsi on ajoute la tâche 2 à la solution partielle. Ensuite la tâche 2 sera suivie de la tâche 5 qui a un plus petit GAP. Enfin les deux tâches suivantes sont choisies selon le même principe : d'abord la tâche 1, puis la tâche 3. Et ainsi, on obtient la permutation (4;2;5;1;3) qui a une qualité de 345 pour le Flowtime.

Cet algorithme, bien que simple et rapide à mettre en place, présente un inconvénient majeur qui apparaît lorsque la prochaine tâche qui doit être ordonnancée n'est pas minimale, ou en cas d'égalité sur les GAP. En effet, dans l'exemple précédent, la première tâche choisie fut la tâche 4 avec un GAP de 17, néanmoins, la tâche 1 possède un GAP de 18, ce qui est relativement proche. Si l'on avait considéré la tâche 1 au lieu de la tâche 4, alors la solution aurait été (1;4;2;5;3) avec une qualité pour le Flowtime de 340. Ce qui est une meilleure qualité que la solution précédente. Ce principe peut alors s'appliquer à d'autres niveaux de la construction de la solution et pas uniquement sur le choix de la première tâche.

Une première idée pour pallier ce problème, serait d'explorer les solutions en tenant compte des deux plus petits GAP à chaque choix et garder le meilleur embranchement, mais cela a un inconvénient majeur : l'explosion combinatoire. En effet, explorer les 2 possibilités à chaque itération revient à explorer 2^n solutions. Ce qui devient rapidement irréalisable. Néanmoins, dans certains cas l'écart du GAP entre les deux premières possibilités est assez grand. Ainsi, une autre idée est de ne considérer que les cas où l'écart entre deux GAP est petit, ce qui permet de réduire assez fortement le nombre de solutions à explorer.

Nous avons défini cet écart comme étant la tolérance τ à laquelle l'algorithme accepte d'explorer le second plus petit GAP ou non. L'Algorithme 3.7 détaille les différentes étapes.

Dans un premier temps, les tâches sont triées dans l'ordre croissant des GAP par rapport à la dernière tâche de la solution courante (cette tâche est appelée $last$) puis compare la différence entre les 2 plus petits GAP. Si la différence est inférieure à la tolérance alors l'algorithme explore les deux choix possibles. Sinon, il sélectionne la tâche avec le GAP minimal. Le résultat final est donc un *arbre de préfixe* des solutions. Pour une tolérance de 5 avec l'exemple précédent cela donnerait l'arbre de préfixe représenté sur

Algorithme 3.7 : Improved Incremental GAP (IIGap)

Entrées : π = solution courante, J = liste des tâches restantes non ordonnancées, τ = tolérance
Sorties : π = solution courante
Données : $last$ = dernière tâche de π

```
si π ≠ ∅ alors                                          // Plus de tâche à ordonnancer
    retourner π
sinon
    Trier les i tâches de J par ordre croissant des GAP_{last,i} ;        // (i) Ordre
    si GAP_{last,J[2]} − GAP_{last,J[1]} ≤ τ alors                  // (ii) Insertion
        π1 = IIGAP(π + J[1], J\J[1], τ) ;                          // Deux possibilités
        π2 = IIGAP(π + J[2], J\J[2], τ);
        si F(π1) < F(π2) alors
            retourner π1
        sinon
            retourner π2
    sinon
        retourner IIGAP(π + r[1], J\J[1], τ) ;                      // Pas d'hésitation
```

la Figure 3.4.

Le paramètre τ de la tolérance est un paramètre sensible qui peut grandement influencer la qualité et le temps d'exécution de l'heuristique. Une grande tolérance permet à l'heuristique de régulièrement explorer deux possibilités à chaque itération, ce qui devient rapidement très coûteux si cela se produit trop souvent, mais garantit une meilleure qualité. En revanche, une petite tolérance permet d'explorer moins de possibilités ce qui accroit le risque de générer une solution de moins bonne qualité.

3.6 Evaluation des performances des heuristiques IncrGAP et IIGAP

Dans une première série d'expérimentations, nous allons comparer IncrGAP aux heuristiques de la littérature pour le Flowtime : BIH, GAPH et PH1. Puis dans une seconde série d'expérimentations, nous allons tester différentes valeurs de τ pour IIGAP et voir l'influence sur la qualité des solutions de ce paramètre.

L'exécution d'algorithmes sur une instance I retourne une solution π avec la qualité $F(\pi)$, le Flowtime de la solution. Pour mesurer la qualité d'une solution, la déviation relative moyenne (*Relative Percentage Deviation - RPD*) est calculée par rapport à la meilleure solution connue (π^*) de la littérature comme suit :

$$RPD(\pi) = \frac{F(\pi) - F(\pi^*)}{F(\pi^*)} * 100 \tag{3.5}$$

Comme évoqué à la Section 1.3.1.4, les instances que nous avons utilisées pour évaluer notre heuristique sont les instances de TAILLARD [1993] qui propose 120 instances

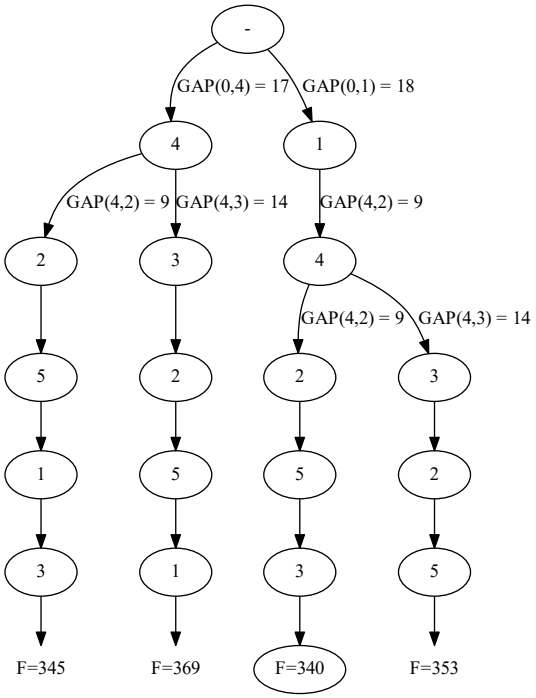


FIGURE 3.4 – Résultat de IncrementalGAPImproved sur les données de l'exemple avec une tolérance de 5

organisées en 10 instances de 12 tailles. Nos expériences sont menées sur les 10 instances disponibles de taille `N_M` où le nombre de tâches `N` va de 20 à 200 et le nombre de machines `M` de 5 à 20 (total de 110 instances sur les 120). Les RPD moyens sur les instances de même taille seront donnés. Les algorithmes ont tous été réimplémentés en C++ et les expérimentations ont été exécutées sur un processeur Intel(R) Xeon(R) 3.5Ghz. Toutes les méthodes présentées dans la section Incremental GAP sont déterministes, ainsi chaque algorithme n'a été exécuté qu'une seule fois.

3.6.1 Résultats d'IncrGAP

Les performances de BIH, GAPH, PH1 et IncrGAP sont données dans le Tableau 3.8 pour chaque taille d'instance. Le test statistique de Wilcoxon a été appliqué pour attester de la meilleure performance pour chaque taille d'instance. Ainsi, PH1 est plus performante sur 9 taille d'instances (valeurs en gras dans le tableau). Cependant, IncrGAP se montre la plus performante sur les instances de taille `100_20` et `200_20` qui sont généralement considérées comme les plus difficiles. De plus, en terme de calcul, IncrGAP se montre la plus rapide. Elle fournit d'ailleurs de bien meilleurs résultats que BIH pour un temps considérablement plus court.

En conclusion, IncrGAP se montre relativement efficace sur les instances difficiles.

	BIH		GAPH		PH1		IncrGAP	
Instances	RPD	Temps	RPD	Temps	RPD	Temps	RPD	Temps
020_05	41,05	0,41	57,60	0,19	**39,75**	0,78	44,53	0,24
020_10	41,18	0,39	58,213	0,17	**39,42**	0,79	42,27	0,21
020_20	39,29	0,41	54,00	0,18	**37,98**	0,75	39,34	0,22
050_05	58,32	8,94	88,26	2,12	**57,23**	7,34	59,16	2,11
050_10	70,69	7,06	98,05	1,71	**69,67**	5,67	70,55	1,15
050_20	75,82	6,39	104,81	1,41	**74,12**	5,25	74,86	1,25
100_05	**47,79**	86,88	74,61	10,87	**47,95**	36,13	51,41	9,38
100_10	70,01	85,83	93,91	10,68	**69,26**	35,97	69,61	7,55
100_20	114,67	86,61	148,91	10,67	113,60	35,96	**112,095**	6,25
200_10	79,33	1252,72	106,49	84,18	**78,04**	266,72	78,31	91,51
200_20	95,30	1251,38	126,71	84,18	95,07	266,95	**92,17**	43,13

TABLEAU 3.8 – RPD et temps d'exécution moyen (en ms) pour chaque taille d'instance pour les heuristiques BIH, GAPH, PH1 et IncrGAP. En gras les meilleurs résultats obtenus validés statistiquement par le test de Wilcoxon

De plus, les résultats d'IncrGAP, bien que moins bons que PH1 sont tout de même très proches, mais avec un temps d'exécution considérablement plus court. Ainsi, IncrGAP semble être une heuristique prometteuse qui peut encore être améliorée, comme c'est le cas avec IIGAP.

3.6.2 Résultats de IIGAP

Dans cette section, nous considérons IIGAP, la version améliorée d'IncrGAP qui considère un niveau de tolérance τ pour le choix de la tâche à ordonnancer lors de la construction gloutonne de la solution. Grâce à l'introduction du paramètre τ, IncrGap peut explorer un peu plus de possibilités pour les cas considérés comme critiques.

La difficulté est de définir efficacement la tolérance à appliquer pour chaque taille d'instance. En effet, le GAP généré entre les tâches sur 5 machines et 20 machines n'est pas à la même échelle. Pour 20 machines, le GAP généralement plus grand que pour 5 machines. De même, le nombre de tâches à considérer entre comptes aussi, il est plus difficile de trouver deux tâches générant un petit GAP lorsque l'on n'a pas 20 tâches, que lorsque l'on en a 200, où il y a plus de possibilités de trouver deux tâches qui génèrent peu de GAP entre-elles.

Ainsi pour déterminer la tolérance, nous avons réalisé un *ranking* des différences de GAP possibles entre les tâches. C'est-à-dire que nous avons calculé toutes les différences de GAP ($|GAP(1,2) - GAP(1,3)|$, ..., $|GAP(1,2) - GAP(1,n)|$, $|GAP(1,3) - GAP(1,4)|$, ...). Ensuite, nous les avons triées par ordre croissant. Ainsi la différence de GAP de rang k désigne la $k^{ième}$ plus petite différence de GAP parmi toutes les tâches. La raison de ce ranking est liée aux nombres de solutions potentielles à explorer. Ainsi, pour une tolérance équivalente à la différence de GAP au rang k, il y aura **au plus** 2^k solutions à visiter,

car le rang k indique le nombre maximal de choix à effectuer.

À titre d'exemple, si la liste des différences de GAP est la suivante : ($|GAP(3,2) - GAP(3,4)| = 7$, $|GAP(7,2) - GAP(7,4)| = 11$, $|GAP(1,3) - GAP(1,4)| = 14$, ...). Si l'on considère la différence au rang 2, soit une tolérance de 11, il y aura pour cet exemple, uniquement deux hésitations possibles : la tâche 3 suivie de la tâche 2 ou 4, et la tâche 7, suivie de la tâche 2 ou 4. En effet, toutes les autres différences de GAP sont supérieures à la tolérance, donc on explorera au plus 2^2 solutions, soit 4 solutions, car nous avons choisi la tolérance obtenue au rang 2.

Ainsi, cette méthode permet de trouver facilement la tolérance adéquate en fonction de l'instance, juste en calculant la différence entre les GAP.

Le Tableau 3.9 montre le RPD moyen de la solution et le temps moyen d'exécution en fonction du rang k de la tolérance pour chaque taille d'instance. En gras, ce sont les valeurs où IIGAP domine PH1, c'est-à-dire avec un meilleur RPD et un meilleur temps. Dans ce tableau nous pouvons constater l'efficacité d'IIGAP en fonction du rang de tolérance, plus le rang est élevé et meilleure est la qualité. En particulier pour les instances supérieures à `050_10` où IIGAP est toujours meilleur que PH1, seules les instances de taille `100_20` font office d'exception. Ainsi, en faisant varier la tolérance, pour un même temps d'exécution que PH1, IIGAP obtient une solution de meilleure qualité.

Néanmoins, plus la tolérance augmente, plus le temps d'exécution augmente. C'est le cas notamment de quelques instances de taille 100_05 et `200_10`, où le temps est beaucoup plus long que pour les autres instances de même taille. À titre de comparaison, parmi les 10 instances de taille `200_10`, la plus rapide met 48ms à s'éxecuter, alors que la plus longue met 28000ms. On peut donc avoir de grandes variations au niveau du temps indépendamment de la taille de l'instance. Cela dépend principalement de la différence de GAP entre les successions de tâches. Si le GAP entre les tâches est relativement proche, alors le nombre d'embranchements dans notre arbre peut rapidement augmenter.

Ce problème semble de prime abord difficile à résoudre pour une telle méthode. Néanmoins une observation des arbres des solutions nous a permis d'aboutir à des pistes pour diminuer le nombre de branches de notre arbre. La Figure 3.5 représente l'arbre des solutions obtenues avec IIGAP sur l'instance Ta01(020x05) avec une tolérance faible ($\tau = 5$). On remarque dans cet exemple que beaucoup de sous-arbres sont strictement identiques.

Chacun des sous-arbres ayant plusieurs occurrences a été représenté par une couleur différente. On se rend compte que l'algorithme calcule plusieurs fois les mêmes sous-solutions. Une piste d'amélioration de l'algorithme est donc de garder une trace des sous-arbres. Les informations que l'on a besoin de stocker sont uniquement les tâches restantes à ordonnancer. En effet, l'algorithme étant déterministe, pour un ensemble de tâches identiques, le sous-arbre généré sera le même. Ainsi, il convient de stocker pour chaque ensemble de tâches calculé, le résultat de la meilleure solution trouvée avec le sous-arbre.

Ce détail étant plus un souci d'implémentation qu'une réelle amélioration d'IIGAP, nous avons décidé de ne pas pousser nos expérimentations dans cette voie.

	RPD	Temps	RPD	Temps	RPD	Temps	RPD	Temps
	$k=0$		$k=1$		$k=2$		$k=3$	
020x05	44,53	0,24	44,27	0,31	43,96	0,35	43,42	0,62
020x10	42,27	0,21	42,21	0,25	40,85	0,33	40,75	0,35
020x20	39,34	0,22	39,00	0,27	38,80	0,30	38,07	0,42
050x05	59,16	2,11	58,95	2,98	58,81	3,21	58,72	3,63
050x10	70.55	1,15	70,52	1,22	70,06	1,60	69,62	1,77
050x20	74,87	1,25	74,56	1,22	74,41	1,37	74,27	2,08
100x05	51,42	9,38	50,8	14,22	50,27	31,42	50,01	39,01
100x10	69,62	7,55	69,53	7,94	**68,88**	**9,89**	**68,74**	**17,84**
100x20	**112,10**	**6,25**	**111,91**	**6,98**	**111,51**	**9,78**	**111,30**	**11,41**
200x10	78,32	91,51	**77.87**	**231,58**	**77.70**	**251,90**	**77.70**	**251,53**
200x20	**92,18**	**43,13**	**91,95**	**47,07**	**91,80**	**74,02**	**91,59**	**93,38**
	$k=4$		$k=5$		$k=6$		$k=7$	
020x05	43,42	0,93	43,28	1,23	43,03	2,81	42,88	3,12
020x10	39,93	0,46	39,93	0,54	39,73	0,87	39,53	1,17
020x20	37,94	0,56	37,94	0,68	37,80	1,28	37,64	1,77
050x05	58,53	4,59	58,53	5,65	57,89	8,51	57,71	9,53
050x10	**69,28**	**2,23**	**69,28**	**3,80**	**69,26**	**3,94**	69,14	11,72
050x20	**74,00**	**2,99**	**74,00**	**3,79**	**168 957**	**4,40**	73,69	5,99
100x05	49,68	39,67	49,68	123,37	49,58	131,79	49,10	1 248,23
100x10	68,36	19,53	68,27	27,72	68,27	46,95	68,12	62,41
100x20	**111,16**	**14,74**	**110,93**	**20,50**	110,93	34,43	110,87	47,22
200x10	**77,37**	**279,80**	**77,2**	**286,27**	**77,2**	**944,82**	77,01	4 685,27
200x20	**91,39**	**125,66**	**91,36**	**262,29**	91,36	276,47	91,31	520,05

TABLEAU 3.9 – RPD et temps d'exécution moyen (en ms) de IIGAP en fonction du rang k pour le choix de la tolérance. En gras les meilleurs résultats obtenus à la fois en RPD et en temps par rapport à PH1.

IIGAP est donc une heuristique rapide et efficace pour peu que le paramètre de la tolérance soit correctement choisi en fonction de l'instance considérée. Elle a d'ailleurs montré des résultats meilleurs que PH1 pour un temps d'exécution équivalent. IIGAP étant performante et très rapide, elle peut-être aisément utilisée en tant qu'initialisation d'une méthode d'optimisation.

3.7 Conclusion

Dans ce Chapitre nous avons présenté deux nouvelles heuristiques constructives pour le problème d'ordonnancement du No-Wait Flowshop, IBI pour minimiser le Makespan et IncrGAP et IIGAP qui minimisent le Flowtime.

IBI a été construite à partir de l'analyse génotypique de la construction des solutions telle que réalisée par les heuristiques de la littérature. En effet, IBI est une amélioration

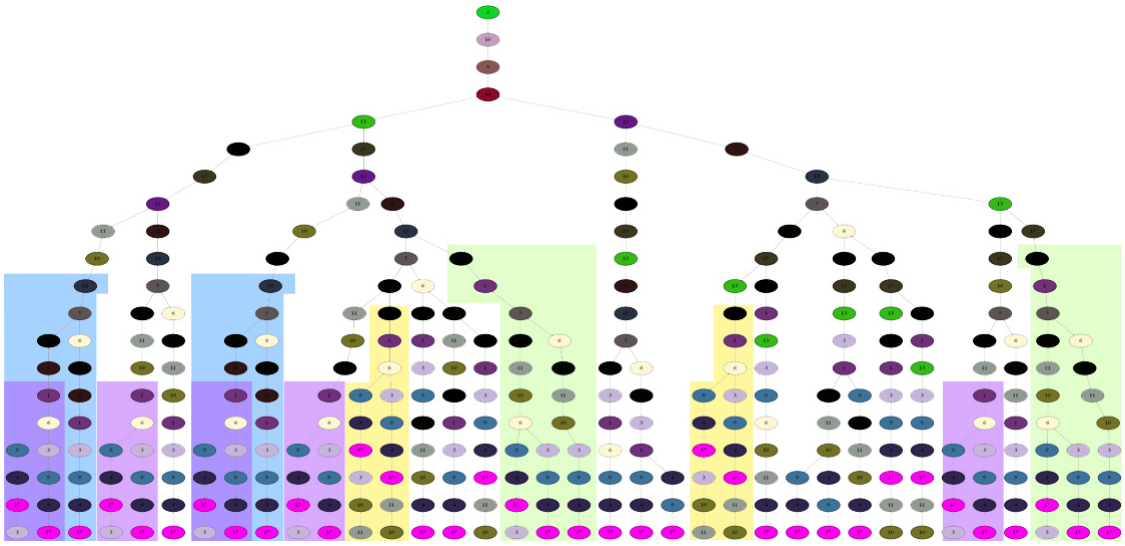


FIGURE 3.5 – Exemple de l'arbre des solutions obtenues avec IncrGAP sur l'instance de Ta01 (020x05) et $\tau = 5$

de l'heuristique NEH très utilisée dans la littérature, où une méthode d'amélioration itérative de la solution a été ajoutée à chaque insertion d'une tâches. Même si cette méthode augmente le temps d'éxécution comparé aux autres heuristiques de la littérature (NEH et BIH), l'amélioration de la qualité finale de la solution est intéressante, et a aussi été validée statistiquement par le test de Friedman.

Afin d'évaluer l'impact du temps supplémentaire accordé à IBI, nous l'avons utilisée, ainsi que les autres heuristiques comme initialisation d'une méta-heuristique. C'est-à-dire que la solution construite par l'heuristique est utilisée en tant que solution initiale d'une métaheuristique. Les expérimentations sur les instances de Taillard ont montré que IBI aide les métaheuristiques à être plus efficaces et que le temps utilisé pour l'initialisation n'est pas inutile.

Concernant le flowtime, IncrGAP et IIGAP ont, quant à elles, été construites à partir de l'analyse phénotypique des solutions optimales sur des petites instances. IIGAP a notamment montré une bien meilleure performance que PH1, une heuristique très utilisée dans la littérature pour le Flowtime. De plus, IIGAP peut fournir des meilleures solutions en faisant varier le paramètre de la tolérance. Ainsi, l'augmentation de la tolérance va augmenter le temps d'exécution de l'heuristique mais l'amélioration de la qualité finale de la solution n'en sera que meilleure.

Ce qui est intéressant dans ce travail est qu'une simple modification des heuristiques constructives permet d'améliorer fortement les résultats. Les modifications ont pu être proposées grâce à l'observation des propriétés sur des instances de petites tailles afin d'en extraire de la connaissance génotypique et phénotypique. Nous avons notamment observé dans la première section de ce chapitre que la structure des solutions optimales a permis de remarquer qu'à chaque étape de la construction d'une solution, il suffisait de faire quelques ré-insertions pour garder une solution optimale. Dans la seconde section, nous avons observé que la somme des GAP des tâches qui se suivent est minimale

dans une majorité des cas.

Nous pouvons maintenant nous demander si ces observations ne peuvent pas notamment servir dans un autre mécanisme d'une méta-heuristique que l'initialisation. Nous avons remarqué dans nos expérimentations que la structure des solutions est très stable au cours du temps. Nous avons donc choisi de concentrer nos recherches sur le génotype des solutions du No-Wait Flowshop, ce qui fait l'objet du chapitre suivant.

Chapitre 4

Modification du voisinage courant par intégration de connaissances

Sommaire

4.1 Introduction

Dans le chapitre précédent, nous avons présenté deux heuristiques constructives utilisant l'intégration de connaissances dans les mécanismes d'initialisation pour résoudre le problème d'ordonnancement de type flowshop sans temps d'attente (NWFSP). Une des approches étudiées était basée sur l'analyse du génotype de la solution, c'est-à-dire la structure des solutions. C'est pourquoi dans ce chapitre nous allons approfondir cette approche génotypique pour l'intégrer cette fois dans une métaheuristique. En effet, les heuristiques constructives, bien que rapides, ne sont pas suffisamment performantes pour donner des solutions de bonne qualité sur les grandes tailles d'instances. À l'opposé, les métaheuristiques permettent d'avoir généralement des solutions de meilleure qualité en un temps raisonnable. Cependant, les métaheuristiques sont des approches généralistes qui n'utilisent pas les connaissances spécifiques d'un problème. Leur efficacité étant liée à leur grande capacité à explorer l'espace de recherche. Dans le chapitre précédent, nous avons analysé l'évolution de la construction d'une solution de bonne qualité. Dans ce chapitre, nous allons étudier la structure des optima locaux, des solutions de bonnes qualités, tel qu'il n'existe pas de solution voisine meilleure. Ces différentes solutions vont nous permettre d'identifier des sous-séquences communes (que nous appellerons *super-jobs*), pour augmenter la performance des métaheuristiques et plus spécifiquement les recherches locales, afin d'obtenir des solutions de très bonne qualité.

En effet, comme il sera exposé dans ce chapitre, l'identification de super-jobs présente plusieurs avantages.

— Premièrement, il permet de réduire la combinatoire du problème en réduisant l'espace de recherche, ce qui permet de rendre efficace l'utilisation des méthodes performantes pour des petites tailles d'instances.

— Deuxièmement, il modifie les relations de voisinage et donc le paysage pendant l'exécution de la recherche locale, et plus précisément réduit le nombre d'optima locaux.

Les expérimentations que nous avons menées lors de cette étude ont montré que les super-jobs ont obtenu de très bons résultats sur les instances de TAILLARD [1993]. Ces travaux de recherche ont été réalisés au moment où les résultats de TMIIG de DING et collab. [2015] possédaient les meilleures solutions de la littérature. Ainsi notre objectif était d'intégrer de la connaissance dans une méthode plus simple que celle de TMIIG, mais tout en permettant d'obtenir des nouvelles meilleures solutions. Notre approche nous avait alors permis de trouver de nouvelles meilleures solutions sur certaines grandes instances de Taillard. Cette approche a fait l'objet d'une communication dans la conférence MIC 2017 ([MOUSIN et collab., 2017a]). En parallèle de nos travaux de recherche, d'autres travaux ont été menés. En particulier LIN et YING [2016] ont eu une approche différente en utilisant une méthode exacte du problème du voyageur de commerce asymétrique adaptée au NWFSP. Cette approche leur a permis d'obtenir les solutions optimales en un temps raisonnable pour ce problème. Bien que nos travaux ne permettent plus aujourd'hui d'obtenir les meilleures solutions connues pour le pro-

blème du NWFSP, ils restent intéressants pour illustrer l'apport significatif de l'intégration de connaissances au sein d'une méthode d'optimisation de type métaheuristique.

Pour exposer la façon dont nous proposons de définir et d'exploiter l'analyse menée sur les structures des solutions, ce chapitre est organisé comme suit : La section 4.2 rapporte l'analyse que nous avons menée et nous définissons les super-jobs formellement. La Section 4.3 décrit l'approche proposée pour exploiter les super-jobs, et la section 4.4 donne des résultats expérimentaux. Enfin, la section 4.5 améliore la méthode avec une version itérée et analyse ses performances. La dernière section présente quelques conclusions ainsi que quelques perspectives.

4.2 Sous-séquences prometteuses de tâches consécutives

Pour le NWFSP, chaque tâche est traitée sans interruption entre les machines successives. Par conséquent, une question se pose : cette spécificité conduit-elle à une structure particulière des meilleures solutions d'une instance donnée? Dans cette section, nous allons effectuer une analyse des solutions optimales globales et locales afin d'en extraire des informations structurelles. Cette analyse nous conduit à définir une sous-séquence prometteuse de tâches consécutives en tant que *super-job*.

4.2.1 Analyse de la structure des solutions optimales

Cette analyse vise à extraire des similitudes dans la structure des ordonnancements efficaces, c'est-à-dire des solutions de bonne qualité. Nous effectuons cette analyse sur des *petites* instances (nombre de tâches faible) avec des durées opératoires uniformément générées suivant la méthodologie des instances de Taillard (voir section 1.3.1.4). Nous étudions ici, à titre d'exemple, l'analyse d'une instance avec 12 tâches et 5 machines. Cette taille de problème (12) permet d'énumérer exhaustivement l'espace de recherche (*i.e.* de générer toutes les solutions) et donc d'identifier l'optimum global et les meilleurs optima locaux [1]. En effet, les optima locaux sont intéressants à analyser, car ils peuvent piéger les méthodes de recherches locales qui explorent l'espace de recherche en se déplaçant itérativement de voisin en voisin. Dans ce qui suit, le terme solutions optimales (ou optima) est utilisé pour traiter plus généralement les solutions optimales globales ou locales.

À titre illustratif, la figure 4.1 donne l'optimum global et les 10 meilleurs optima locaux de l'instance étudiée de taille 12. Pour cette petite instance, il est facile de voir que ces meilleurs optima partagent une structure similaire entre eux et avec l'optimum global. En effet, la tâche 8 est toujours positionnée au début de l'ordonnancement et deux sous-séquences de deux tâches consécutives sont présentes dans chacun d'eux : [12 4] en vert, et [11 2] en rouge et une sous-séquence est présente dans 10 solutions sur 11 : [9 1] colorée en bleu. Les optima locaux ne peuvent pas être améliorés en appliquant l'opérateur de voisinage insertion. Cependant, si l'on considère chaque sous-

1. Les optima locaux sont des solutions qui n'ont pas de meilleures solutions voisines *i.e.* aucun mouvement d'insertion ne pourrait conduire à une solution de meilleure qualité.

C_{max}	Solution											
Optimum global : 1021	8	3	7	5	10	[9	1]	[12	4]	6	[11	2]
Optima locaux : 1036	8	3	7	[11	2]	5	10	[9	1]	[12	4]	6
1075	8	10	[9	1]	[12	4]	7	5	3	[11	2]	6
1090	8	3	5	10	6	[12	4]	7	[11	2]	[9	1]
1103	8	3	5	10	6	[12	4]	7	[9	1]	[11	2]
1132	8	10	6	[12	4]	7	5	3	[9	1]	[11	2]
1132	8	3	7	[11	2]	5	10	9	6	[12	4]	1
∗1176	8	3	5	10	[9	1]	[11	2]	7	6	[12	4]
1189	8	10	6	[12	4]	7	5	3	[11	2]	[9	1]
1232	8	10	[9	1]	[12	4]	7	3	5	6	[11	2]
1246	8	3	7	10	[9	1]	[11	2]	5	6	[12	4]

FIGURE 4.1 – Description (C_{max}+ séquence de tâches) de l'optimum global et des 10 meilleurs optima locaux pour l'instance étudiée de taille 12. Les sous-séquences [11 2] et [12 4] colorées respectivement en rouge et vert, apparaissent dans la séquence de toutes les solutions et la sous-séquence [9 1] colorée en bleu, apparaît dans 10 solutions sur 11. En considérant ces trois sous-séquences comme une unique tâche, la recherche locale peut alors atteindre l'optimum global. Seul un optimum local (identifiable avec l'étoile ∗) n'arrive pas à être amélioré avec ces sous-séquences.

séquence de tâches consécutives comme une tâche unique, le meilleur optimum local (C_{max} = 1036) diffère de l'optimum global (C_{max} = 1021) par le simple déplacement de la sous-séquence [11 2] à la fin de la solution. De même, l'application de l'opérateur de voisinage insertion sur les autres optima locaux avec la prise en compte des sous-séquences identifiées au lieu des tâches individuelles permet à tous les optima locaux (sauf celui de C_{max} = 1176 identifié par une étoile) d'atteindre l'optimum global.

Comme mentionné précédemment, cette étude a été menée sur des petites instances pour être en mesure d'énumérer exhaustivement l'espace de recherche. Ici, seules des sous-séquences de deux tâches consécutives ont été trouvées. Cependant, la taille d'une sous-séquence n'est pas limitée à uniquement deux tâches. Ainsi, si une tâche a est toujours suivie d'une tâche b et, la tâche b est toujours suivie d'une tâche c alors, la sous-séquence [a b c] des trois tâches consécutives doit être considérée plutôt que les deux sous-séquences [a b] et [b c] séparément.

Les observations faites dans cette analyse motivent la substitution des tâches originales par des sous-séquences prometteuses de tâches à privilégier pour atteindre des solutions de meilleure qualité. Cette transformation du problème d'origine peut représenter une bonne opportunité pour résoudre des instances de grande taille. Une question se pose donc : comment définir et identifier les sous-séquences prometteuses ?

4.2.2 Définition d'un super-job

L'analyse exhaustive précédente de la structure des optima locaux pour les instances de petites tailles nous conduit à supposer qu'un comportement similaire apparaît sur les plus grandes instances. Dans cette section, nous présenterons la méthodologie que nous proposons pour identifier les sous-séquences prometteuses d'une instance inconnue à résoudre.

Indépendamment de la taille de l'espace de recherche, les solutions de bonne qualité et, plus précisément, les optima locaux qui ont une bonne qualité partagent toutes une structure similaire. Lorsque l'espace de recherche est non énumérable, il est généralement admis d'utiliser un échantillon de solutions afin d'analyser leur structure, leurs caractéristiques ...

Ici, nous proposons d'extraire les sous-séquences prometteuses d'un pool $\mathscr{P}^*$ d'optima locaux de *bonne qualité* et de définir ainsi un *super-job* avec une confiance de σ. Ce *super-job* contiendra donc les tâches consécutives qui apparaissent au moins σ de fois dans les solutions de $\mathscr{P}^*$. Par exemple, si $\sigma = 50\%$, les super-jobs sont les sous-séquences de jobs consécutifs partagés par au moins la moitié des solutions de $\mathscr{P}^*$. Notons que seules les sous-séquences les plus longues sont considérées comme des super-jobs. Par exemple, si $[a\ b]$ et $[b\ c]$ apparaissent tous les deux au moins σ de fois dans $\mathscr{P}^*$, seulement $[a\ b\ c]$ est défini comme un super-job.

Afin de faciliter les notations des équations suivantes, nous définissons SJ comme un super-job, où $\mathrm{SJ}_{.first}$ (resp. $\mathrm{SJ}_{.last}$) désigne la première tâche (resp. la dernière tâche) du super-job. Grâce à la particularité du NWFSP qui indique que le délai entre deux tâches consécutives est constant, il est alors facile de calculer la mesure du délai entre deux super-jobs :

$$\mathrm{D}_{\mathrm{SJ}_i,\mathrm{SJ}_j} = \Big(\sum_{1\leq r\leq |\mathrm{SJ}_i|-1}^{k} d_{\mathrm{SJ}_i[r],\,\mathrm{SJ}_i[r+1]}\Big) + d_{\mathrm{SJ}_{i.last},\mathrm{SJ}_{j.first}} \tag{4.1}$$

En d'autres termes, le délai D entre deux super-jobs SJ_i et SJ_j est défini par la somme des délais d de la sous-séquence de tâches du super-job SJ_i auquel on ajoute le délai d entre la dernière tâche de SJ_i et la première tâche de SJ_j.

De même, le calcul de la durée d'exécution d'un super-job est nécessaire pour calculer le temps de complétion C_i d'un super-job. Le calcul de la durée d'exécution s'obtient par la formule suivante :

$$\mathrm{P}_{\mathrm{SJ}_i} = \Big(\sum_{1\leq r\leq |\mathrm{SJ}_i|-1}^{k} d_{\mathrm{SJ}_i[r],\,\mathrm{SJ}_i[r+1]}\Big) + \sum_{j=1}^{\mathrm{M}} p_{\mathrm{SJ}_{i.last},j} \tag{4.2}$$

La durée d'exécution d'un super-job consiste donc à calculer la somme des délais d de la sous-séquence de tâches du super-job SJ_i auquel on ajoute la durée d'exécution de la dernière tâche du super-job.

Ces données étant constantes, le calcul s'effectue uniquement lors de la création du super-job. Ainsi, l'utilisation des super-jobs ne change pas la complexité de la fonction objectif puisque cette dernière se calcule alors de la même façon.

Cette méthodologie a l'avantage d'être pertinente, quelle que soit la taille du problème. Cependant, l'inconvénient principal peut être le temps de calcul requis pour générer le pool de solutions de bonne qualité. Par conséquent, la taille du pool de solutions doit être fixée avec soin : si le pool est trop grand, cela signifie que trop de temps serait dépensé pour générer les solutions du pool, s'il est trop petit, cela signifie que l'identification des super-jobs ne serait pas significative. Cet aspect sera discuté au cours de la section expérimentale.

4.2.3 Avantages des super-jobs

Les avantages de considérer des sous-séquences de tâches consécutives (super-jobs) comme des tâches uniques sont multiples. D'abord, d'un point de vue de la complexité, cela réduit la combinatoire du problème, c'est-à-dire le nombre de solutions potentielles puisque le nombre de tâches à considérer sera plus faible. Deuxièmement, d'un point de vue recherche locale, cela va modifier l'espace de recherche et le paysage induit par l'opérateur d'insertion et ainsi, de nouvelles régions de l'espace de recherche peuvent devenir accessibles (WRIGHT [1932]).

Afin de mieux comprendre l'impact des super-jobs sur le paysage, nous présentons une visualisation en 2D de la transformation du paysage dans la figure 4.2. Chaque point du graphique représente une solution. Les points bleus représentent le paysage original, en considérant les solutions construites à partir des tâches originales suivant une relation de voisinage (dans une représentation simplifiée où une solution a deux voisins), la métaphore du paysage est donc bien visible avec la présence de plusieurs vallées qui contiennent les optima locaux en leur point le plus bas. Les points rouges représentent le paysage modifié obtenu en considérant les super-jobs. De toute évidence, l'utilisation de super-jobs lisse le paysage et permet d'éviter certains optima locaux originaux qui ont disparu. On constate la présence de deux grandes vallées et d'une plus petite. Le paysage est donc plus facile à parcourir pour une recherche locale. Cependant, ce lissage peut aussi être un désavantage dans le sens où certains bons optima locaux peuvent devenir inaccessibles, de même pour l'optimum global. Ainsi, bien que le lissage du paysage a un avantage certain, il est nécessaire de considérer les tâches de façon indépendante à la fin de la recherche pour considérer l'ensemble de l'espace de recherche.

4.3 Iterated Greedy avec les Super-jobs

Les super-jobs ont été définis comme des caractéristiques structurelles communes des solutions de bonne qualité. Dans cette section, nous proposons une approche qui exploite ces super-jobs pour améliorer une heuristique déjà efficace proposée dans la littérature pour le problème d'ordonnancement du Flowshop de Permutation (PFSP), l'Iterated Greedy (IG), afin d'atteindre des meilleures solutions pour les instances de Taillard. Ainsi, cette section présente d'abord l'algorithme IG, puis l'approche proposée.

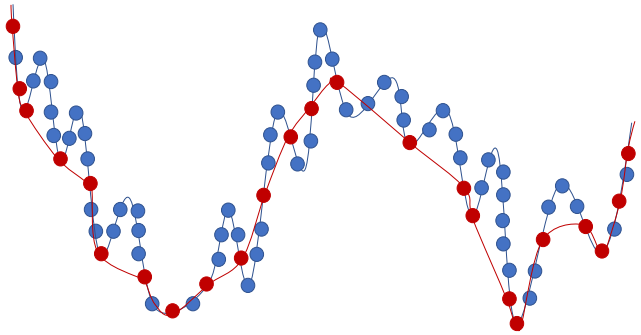


FIGURE 4.2 – Visualisation du paysage en 2D. Les points bleus représentent les solutions du problème original alors que les points rouges représentent ceux construits avec les super-jobs.

4.3.1 L'algorithme de l'Iterated Greedy

L'algorithme Iterated Greedy (IG) de RUIZ et STÜTZLE [2007], initialement proposé pour le PFSP (Permutation Flowshop Scheduling Problem), est une recherche locale itérée. La recherche locale employée dans l'IG est une descente, c'est-à-dire qu'à chaque itération, une solution voisine améliorante est choisie (*i.e.* qu'il existe une réinsertion d'une tâche dans la solution, telle que la réinsertion améliore la qualité de la solution). Ce processus est répété jusqu'à ce qu'un optimum local soit atteint, *i.e.* la solution n'est plus améliorable par une insertion. Une fois cet optimum local atteint, une phase de perturbation est réalisée. Cette perturbation consiste à supprimer des tâches d'une solution, puis de les réinsérer une à une à la meilleure position possible, *i.e.* qui minimise le Makespan partiel, comme le processus de construction de NEH, ce qui permet de relancer la recherche dans une autre zone de l'espace de recherche.

De même, afin d'éviter les situations où la perturbation aurait trop dégradé une solution, un critère d'acceptation est utilisé à la fin de la recherche locale. Ce critère d'acceptation, permet de déterminer si l'optimum local obtenu est intéressant. Si oui, l'optimum est utilisé pour la prochaine perturbation, si non, la solution de l'itération précédente est utilisée pour la prochaine perturbation. Ce critère d'acceptation est inspiré de celui du recuit simulé et utilise donc une notion de température. L'algorithme 4.1 résume toutes les étapes de l'IG.

IG est connu pour être efficace pour résoudre de nombreuses variantes du problème de type Flowshop de permutation. Cependant, même s'il est capable d'atteindre des solutions de bonne qualité en un temps raisonnable, il n'est pas capable d'atteindre, pour le NWFSP, les meilleures solutions connues pour les instances de Taillard de grandes tailles. Lorsque ces travaux de recherche ont été effectués, le meilleur algorithme pour le NWFSP était l'algorithme *Tabu-Mechanism Improved Iterated Greedy* (TMIIG) de DING et collab. [2015], inspiré de l'IG avec un mécanisme de recherche tabou. Puisque IG est très efficace pour résoudre des instances de petites et moyennes tailles et que l'utilisation des super-jobs permet de diminuer la taille du problème, nous proposons de concevoir un nouvel algorithme en exploitant à la fois l'IG et les super-jobs.

Algorithme 4.1 : Iterated Greedy (IG)

Entrées : Ensemble J de n tâches, $Temperature$ pour le critère d'acceptation
Sorties : π^* la séquence des tâches ordonnancées
π = NEH(J) ; // Méthode d'initialisation
$\pi = recherche\_locale(\pi)$;
$\pi^* = \pi$;
tant que *Critère d'arrêt non atteint* **faire**
  $\pi' = \pi$;
  pour i *de 1 à* d **faire** // Perturbation
    Supprimer une tâche aléatoirement dans π' et l'ajouter dans π_R;
  pour i *de 1 à* d **faire**
    Insérer la tâche $\pi_R[i]$ dans π' à la meilleure position de π' qui minimise le Makespan partiel;
  $\pi' = recherche\_locale(\pi')$; // Recherche locale
  si $C_{max}(\pi') < C_{max}(\pi^*)$ **alors** // Critère d'acceptation
    $\pi^* = \pi'$;
  sinon si $C_{max}(\pi') < C_{max}(\pi)$ ***ou*** $random() \leq e^{-(C_{max}(\pi')-C_{max}(\pi))/Temperature}$ **alors**
    $\pi = \pi'$;
retourner π^*

4.3.2 Iterated Greedy avec les Super-Jobs

L'algorithme Iterated Greedy avec les super-jobs (IG_{SJ}) identifie les super-jobs avec plusieurs niveaux de confiance. Ces niveaux de confiance sont traités dans l'ordre croissant pendant la recherche. C'est-à-dire que l'algorithme va d'abord chercher des solutions de bonne qualité avec des super-jobs à faible niveau de confiance, puis va au cours du temps utiliser des super-jobs avec un niveau de confiance de plus en plus élevé jusqu'à ce que l'algorithme revienne à l'instance initiale, *i.e.* sans utilisation de super-jobs.

L'algorithme 4.2 donne le pseudo-code de ce nouvel algorithme.

Compte tenu d'un pool d'optima locaux de bonne qualité, dont la génération sera discutée plus tard, et une liste croissante $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_k\}$ des niveaux de confiance, IG_{SJ} identifie d'abord les super-jobs concernant le premier niveau de confiance σ_1. Ensuite, une méthode d'initialisation génère une première solution π avec ces super-jobs identifiés. Une procédure est ensuite itérée pour chaque niveau de confiance de Σ, alternant entre (i) une phase d'identification des super-jobs (sauf pour le premier niveau de confiance σ_1) et (ii) une phase d'amélioration en utilisant IG. L'algorithme Iterated Greedy est exécuté sur la solution en considérant les super-jobs comme des tâches uniques du problème. Comme IG n'a pas de critère d'arrêt naturel, un temps maximal, ainsi qu'un nombre maximal d'itérations sans amélioration sont utilisés pour arrêter l'algorithme IG. Une fois que tous les niveaux de confiance ont été utilisés, l'algorithme renvoie la meilleure solution trouvée au cours de la recherche. La figure 4.3 illustre le comportement de IG_{SJ} avec pour k niveaux de confiance $\Sigma = \{60\%, 80\%, \ldots, \infty\}$,

Algorithme 4.2 : IG_{SJ} – Iterated Greedy avec algorithme d'apprentissage.

$\mathscr{P}^*$: pool de solutions;
$\Sigma = \sigma_1, \sigma_2, \ldots$: Liste des niveaux de confiance dans l'ordre croissant;
Sorties : π la séquence des tâches ordonnancées
SJ = identifier $(\mathscr{P}^*, \sigma_1)$; `// Identifier les Super-jobs avec une confiance de` σ_1
π = init(SJ); `// Initialiser` π `avec les SJ identifiés`
pour *i de 1 à* $|\Sigma|$ **faire**
  SJ = identifier$(\mathscr{P}^*, \sigma_i)$; `// (i) Identifier les Super-jobs avec une confiance de` σ_i
  π = IG(π, SJ); `// (ii) Exécuter IG à partir de la solution` π `avec les super-jobs identifiés SJ`
retourner π

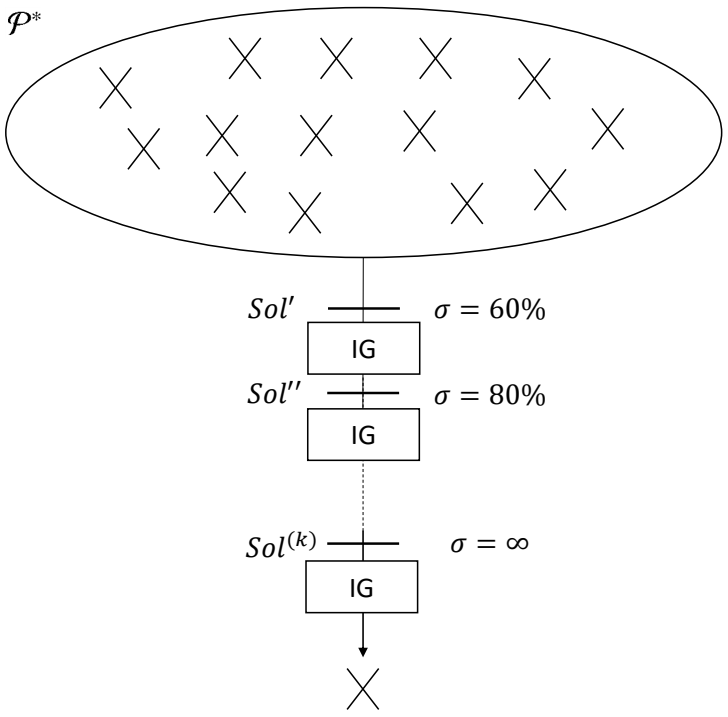


FIGURE 4.3 – Illustration de l'algorithme IG_{SJ}.

σ	Phase	Taille du problème	C_{max}	Solution π																	
60	(i)	7	-	**[2**	**8]**	3	**[4**	**20]**	6	12	**[14**	**16**	**15**	**18**	**10**	**5**	**9**	**19**	**1**	**13**	**7**	**11]**	17
	init		3021	**[4**	**20]**	17	6	3	12	**[14**	**16**	**15**	**18**	**10**	**5**	**9**	**19**	**1**	**13**	**7**	**11]**	**[2**	**8]**
	(ii)		3021	**[4**	**20]**	17	6	3	12	**[14**	**16**	**15**	**18**	**10**	**5**	**9**	**19**	**1**	**13**	**7**	**11]**	**[2**	**8]**
80	(i)	14	3021	**[4**	**20]**	17	6	3	12	**[14**	**16]**	15	18	**[10**	**5]**	9	**[19**	**1]**	13	**[7**	**11]**	**[2**	**8]**
	(ii)		3013	**[4**	**20]**	17	6	3	13	18	**[19**	**1]**	12	**[14**	**16]**	15	**[10**	**5]**	9	**[7**	**11]**	**[2**	**8]**
∞	(i)	20	3013	4	20	17	6	3	13	18	19	1	12	14	16	15	10	5	9	7	11	2	8
	(ii)		3013	4	20	17	6	3	13	18	19	1	12	14	16	15	10	5	9	7	11	2	8

TABLEAU 4.1 – IG_{SJ} dans les différentes phases sur l'instance `ta023` de Taillard avec $\Sigma = \{60\%; 70\%; \infty\}$. La phase (i) correspond à l'identification des super-jobs et la phase (ii) à l'application de l'IG. Pour chaque niveau de confiance, les super-jobs identifiés sont en gras.

où $\sigma = \infty$ signifie qu'aucun super-job n'est créé (le problème est résolu avec toutes les tâches originelles), cette dernière étape est nécessaire pour s'assurer que l'espace de recherche du problème initial puisse être exploré en entier et que les super-jobs n'empêchent pas d'atteindre un optimum local de bonne qualité.

Le tableau 4.1 présente l'évolution de la solution π pour toutes les phases d'une exécution de IG_{SJ} sur l'instance `ta023` de Taillard (20 jobs, 20 machines) avec la liste des niveaux de confiance $\Sigma = \{60\%, 80\%, \infty\}$. Dans cet exemple, avec $\sigma = 60\%$, sept super-jobs sont identifiés (phase (i)) : un de taille 12, deux de taille 2 et les tâches restantes de taille 1. Par conséquent, la taille du problème est réduite de 20 à 7. La méthode d'initialisation construit alors un optimum local avec une qualité de 3021. IG ne peut pas l'améliorer (phase (ii)). Ensuite, en considérant un niveau de confiance de $\sigma = 80\%$, certains super-jobs précédents sont décomposés (phase (i)). En effet, le plus grand super-job est décomposé en huit plus petits. La taille du problème est égale à 14 et IG parvient à trouver une meilleure solution avec une qualité de 3013 (phase (ii)). Il apparaît, dans ce cas particulier, que l'optimum global est atteint dans cette seconde phase. Ceci explique pourquoi la dernière phase ($\sigma = \infty$, revient à exécuter l'IG sur le problème de départ) n'est pas capable de produire une solution améliorante.

4.4 Evaluation des performances

Afin d'évaluer l'efficacité de l'algorithme IG_{SJ}, les expériences ont été conduites sur les instances de Taillard et comparées aux solutions obtenues par le meilleur algorithme de la littérature au moment des expérimentations (TMIIG de DING et collab. [2015]).

4.4.1 Protocole expérimental

Comme dans le chapitre 3, les instances utilisées pour évaluer les performances de la méthode proposée sont les instances de Taillard (TAILLARD [1993]) organisées en 10 instances de 12 tailles différentes. Dans ce chapitre, nous considérons en plus les grandes instances qui sont plus difficiles à résoudre. L'algorithme a été implémenté en C ++ et

Machines \ Tâches	50										100										200										500									
5				17	2			21			11		X	14	5	29	13	11	16	5	6	4																			
10													26	20	7	24	18	26	22	X	23	23	24	19	X	X	X	X	29	28	X	X										
20													13	29	12	25	26	X	29	X	19	29	X	X	X	X	X	X	X	29	X	X	X	X	28	29	23	X	X	X	X

TABLEAU 4.2 – Résultat sur les instances de Taillard (organisées par taille) pour $\Sigma_1 = \{60\%, 80\%, \infty\}$. Cellule grise : la qualité de la meilleure solution de TMIIG est atteinte. # / X : Nombre de fois où la meilleure solution de TMIIG est améliorée sur les 30 exécutions (X = toutes les exécutions).

les expériences ont été exécutées sur un processeur Intel(R) Xeon(R) 3.5GHz.

Pour ces expérimentations, nous avons fixé les paramètres ci-dessous, une analyse des paramètres du pool initial et des niveaux de confiance est menée plus loin dans ce chapitre.

— **Pool initial de solutions $\mathscr{P}^*$:** 10 solutions ont été générées à partir de 10 exécutions indépendantes d'IG avec un temps maximal de $n^2 * 10$ ms chacune. Une discussion détaillée sur ce paramètre est réalisée plus loin dans ce chapitre.

— **Niveaux de confiance :** Deux listes $\Sigma_1 = \{60\%, 80\%, \infty\}$ et $\Sigma_2 = \{60\%, 70\%, 80\%, 90\%, \infty\}$ ont été testées afin d'évaluer les performances de IG_{SJ} en fonction du lissage du paysage.

— **Méthode d'initialisation :** Heuristique de la meilleure insertion itérée (IBI - cf chapitre 3).

— **Critère d'arrêt de IG dans la phase ii :** Un temps maximal de $n_{sj}^2 * 10$ ms (où n_{sj} est le nombre de super-jobs de la phase) et un nombre maximal d'itérations sans amélioration de $50 * n_{sj}$.

Chaque exécution de IG_{SJ} sur une instance donnée $\mathscr{I}$ retourne une solution π de qualité $C_{max}(\pi)$. Pour mesurer la qualité de la solution, l'écart relatif en pourcentage (RPD) est calculé par rapport à la meilleure solution connue de la littérature π^* comme suit :

$$\text{RPD} = \frac{C_{max}(\pi) - C_{max}(\pi^*)}{C_{max}(\pi^*)} * 100 \tag{4.3}$$

Comme IG_{SJ} est stochastique, nous avons réalisé 30 exécutions de notre algorithme pour en évaluer la robustesse. Ainsi la performance pour une instance I est calculée comme étant la moyenne des 30 RPD calculés.

4.4.2 Résultats

Pour analyser les performances de l'algorithme IG_{SJ} , les résultats obtenus sont comparés avec les solutions rapportées dans DING et collab. [2015] (qui étaient jusqu'en octobre 2016 les meilleures solutions connues). Les Tableaux 4.2 (liste Σ_1) et 4.3 (liste Σ_2) indiquent, pour chaque instance (10 instances par taille), si la meilleure solution de TMIIG est atteinte (cellules colorées en gris) et si celle-ci est améliorée (cellule non vide). Par conséquent, lorsqu'un nombre est présent dans une cellule, cela indique le nombre

Machines \ Tâches	50									100										200										500									
5	1		23	2		25			17	X	14	6	X	13	14	20	8	6	6																			
10										27	17	19	X	24	28	26	X	X	22	27	19	X	X	X	X	X	29	X	X										
20										17	X	22	26	27	X	X	X	27	29	X	X	X	X	X	X	X	X	X	X	X	X	X	29	26	X	X	X	X

TABLEAU 4.3 – Résultat sur les instances de Taillard (organisées par taille) pour $\Sigma_2 = \{60\%, 70\%, 80\%, 90\%, \infty\}$. Cellule grise : la qualité de la meilleure solution de TMIIG est atteinte. # / X : Nombre de fois où la meilleure solution de TMIIG est améliorée (X = toutes les exécutions).

de fois où la méthode améliore la meilleure solution obtenue avec TMIIG au cours des 30 exécutions, un 'X' indique 30/30. Les résultats obtenus pour les 30 instances avec 20 tâches ne sont pas rapportés ici car un simple IG parvient à trouver la solution optimale. Les deux tableaux montrent que, dans tous les cas, la méthode atteint la meilleure solution de TMIIG indépendamment des deux listes de niveaux de confiance Σ_1 et Σ_2. De plus, ils montrent que toutes les instances avec 100, 200 et 500 tâches sont améliorées avec l'algorithme IG_{SJ} proposé.

Pour analyser plus en profondeur le comportement de la méthode, les tableaux 4.4 et 4.5 apportent quelques informations sur l'exécution de la méthode pour les deux listes Σ_1 et Σ_2. Les résultats présentés sont la moyenne des RPD sur les 30 exécutions pour toutes les instances d'une même taille (10). Les deux tableaux ont la même structure et les conclusions sont également similaires. Nous discuterons d'abord des analyses communes, puis soulignerons les différences au sein de la discussion.

Les premières parties de ces tableaux indiquent la taille du problème *i.e.* le nombre de tâches pour chaque phase. Ces tâches sont soit des tâches originelles, soit des super-jobs construits par la concaténation de plusieurs tâches, comme expliqué précédemment. Cette mesure permet d'estimer la combinatoire du problème. Par exemple, dans Tableau 4.4 pour les instances de taille 200, lorsque $\sigma = 60\%$, le nombre de tâches est environ de 80-90 tâches. Cela signifie que la taille du problème a été divisée par plus de 2. Dans la phase suivante, lorsque $\sigma = 80\%$, certains super-jobs sont divisés et le nombre de tâches est d'environ 130-140. La combinatoire est toujours réduite par rapport à la taille initiale du problème. Comme mentionné précédemment lorsque $\sigma = \infty$ le nombre de tâches est égal au nombre de tâches d'origine (*i.e.* 200 pour l'exemple précédent). Cette première observation indique qu'il existe une réelle différence entre l'ensemble des tâches obtenues pour les deux niveaux de confiance, ce qui montre que l'identification des super-jobs est différente.

Une autre observation est que le nombre de machines a également un impact sur l'identification des super-jobs. En effet, pour un nombre donné de tâches, plus le nombre de machines est important, plus la combinatoire est petite. En ce qui concerne les instances de taille 500, le niveau élevé de la combinatoire peut s'expliquer par l'utilisation de l'IG pour générer le pool de solutions. En effet, IG converge difficilement dans le temps imparti pour des problèmes de grandes tailles. Les solutions du pool sont donc trop diversifiées pour permettre d'identifier des sous-séquences communes de tâches.

Instances	Taille du problème		Convergence			
	60%	80%	Init	$IG_{60\%}$	$IG_{80\%}$	IG_{∞}
20×5	1.23	10.26	98.7	0.0	1.3	0.0
20×10	1.00	10.00	100.0	0.0	0.0	0.0
20×20	1.26	10.35	95.3	2.3	2.3	0.0
50×5	9.23	31.65	10.0	31.0	53.7	5.3
50×10	5.98	29.83	28.3	6.7	57.7	7.3
50×20	4.10	27.85	59.0	3.3	29.3	8.3
100×5	45.46	78.32	0.0	43.3	34.7	22.0
100×10	29.04	68.99	0.0	16.3	53.7	30.0
100×20	27.37	68.44	0.0	9.3	59.0	31.7
200×10	95.36	157.41	0.0	15.3	45.7	39.0
200×20	81.83	150.74	0.0	5.0	41.7	53.3
500×20	268.87	409.28	0.0	0.7	30.3	69.0

Instances	RPD				Temps (s)			
	Init	$IG_{60\%}$	$IG_{80\%}$	IG_{∞}	60%	80%	∞	total
20×5	0.06	0.06	0.00	0.00	0.14	0.31	0.00	0.45
20×10	0.00	0.00	0.00	0.00	0.14	0.30	0.00	0.45
20×20	0.03	0.03	0.00	0.00	0.14	0.30	0.01	0.45
50×5	0.97	0.22	0.06	0.06	0.59	1.72	1.69	3.99
50×10	0.72	0.47	0.07	0.07	0.82	1.84	0.98	3.64
50×20	0.42	0.24	0.05	0.04	0.89	1.88	0.54	3.30
100×5	3.00	0.03	0.00	-0.01	6.88	7.07	8.98	22.93
100×10	1.69	0.07	-0.05	-0.07	3.25	7.48	9.38	20.11
100×20	1.49	0.09	-0.08	-0.10	2.82	7.71	9.22	19.74
200×10	2.89	-0.13	-0.18	-0.19	30.94	35.48	46.51	112.93
200×20	1.99	-0.20	-0.30	-0.32	22.12	39.35	48.80	110.27
500×20	2.81	-0.18	-0.24	-0.25	598.84	656.63	784.92	2040.38

TABLEAU 4.4 – IG_{SJ} avec $\Sigma_1 = \{60\%, 80\%, \infty\}$. Les mesures reportées sont la moyenne des 10 instances de chaque taille.

Instances	Taille du problème				Convergence					
	60%	70%	80%	90%	Init	$IG_{60\%}$	$IG_{70\%}$	$IG_{80\%}$	$IG_{90\%}$	IG_{∞}
20×5	1.23	10.19	10.30	10.30	98.70	0.00	1.33	0.00	0.00	0.00
20×10	1.00	10.00	10.00	10.00	100.00	0.00	0.00	0.00	0.00	0.00
20×20	1.26	10.22	10.40	10.45	95.30	2.30	1.67	0.67	0.00	0.00
50×5	9.23	29.51	31.70	32.50	9.30	23.70	35.67	20.67	9.00	1.67
50×10	5.98	28.09	29.90	31.10	24.30	5.30	30.00	19.67	14.67	6.00
50×20	4.10	26.91	27.90	29.90	56.70	3.00	14.00	13.00	10.67	2.67
100×5	45.46	71.38	78.30	85.45	0.00	31.00	21.33	16.00	19.67	12.00
100×10	29.04	63.72	69.00	75.30	0.00	9.00	25.33	26.67	25.33	13.67
100×20	27.37	62.91	68.50	64.85	0.00	8.00	16.33	32.00	28.67	15.00
200×10	95.36	144.20	157.50	168.15	0.00	5.70	19.33	28.00	23.00	24.00
200×20	81.83	138.14	150.90	162.30	0.00	0.70	14.67	28.00	28.67	28.00
500×20	268.87	376.81	409.50	435.80	0.00	0.00	5.33	15.67	31.33	47.67

Instance	RPD						Temps (s)					
	Init	$IG_{60\%}$	$IG_{70\%}$	$IG_{80\%}$	$IG_{90\%}$	IG_{∞}	60%	70%	80%	90%	∞	total
20×5	0.06	0.06	0.00	0.00	0.00	0.00	0.14	0.14	0.14	0.31	0.00	0.74
20×10	0.00	0.00	0.00	0.00	0.00	0.00	0.14	0.14	0.14	0.31	0.00	0.73
20×20	0.03	0.03	0.00	0.00	0.00	0.00	0.14	0.14	0.14	0.30	0.01	0.74
50×5	0.97	0.22	0.06	0.04	0.03	0.03	0.54	1.40	1.40	1.56	1.70	6.60
50×10	0.72	0.47	0.09	0.07	0.06	0.06	0.76	1.29	1.31	1.77	0.99	6.11
50×20	0.42	0.24	0.06	0.04	0.04	0.03	0.86	1.14	1.19	1.87	0.54	5.60
100×5	3.00	0.03	0.00	-0.01	-0.02	-0.03	6.82	6.38	6.94	7.63	8.91	36.68
100×10	1.69	0.07	-0.04	-0.07	-0.09	-0.09	3.32	6.57	6.80	7.01	8.92	32.62
100×20	1.49	0.09	-0.05	-0.10	-0.12	-0.13	2.90	6.95	7.11	7.01	8.63	32.60
200×10	2.89	-0.13	-0.18	-0.20	-0.22	-0.22	31.09	30.83	32.45	35.71	43.83	173.91
200×20	1.99	-0.20	-0.28	-0.32	-0.33	-0.34	22.09	34.00	33.18	35.29	45.30	169.85
500×20	2.81	-0.18	-0.23	-0.26	-0.27	-0.28	601.14	566.95	585.10	631.06	744.54	3 128.79

TABLEAU 4.5 – IG_{SJ} avec $\Sigma_2 = \{60\%, 70\%, 80\%, 90\%, \infty\}$. Les mesures reportées sont la moyenne des 10 instances de chaque taille.

Des mesures sur la convergence de IG_{SJ} sont données dans la seconde partie des deux tableaux. La colonne, appelée *Convergence*, indique en pourcentage, le nombre de fois (sur les 30 exécutions) où la méthode - $Init$ ou IG - atteint la meilleure solution de l'exécution (moyenne des 10 instances de chaque taille), c'est-à-dire l'étape où la solution a été améliorée pour la dernière fois durant l'exécution de l'algorithme. On peut observer que dans les deux tableaux, pour les instances de taille 20, la meilleure solution est principalement atteinte lors de la phase d'initialisation. En effet, pour les instances de petites tailles, IG est très efficace et parvient à atteindre les solutions les plus connues. Pour la taille 50, IG_{SJ} parvient presque toujours à trouver sa meilleure solution avant de considérer le problème original ($\sigma = \infty$) contrairement à l'IG d'origine. Ceci valide l'utilisation de super-jobs. Cependant, pour les problèmes de plus grande taille, certaines améliorations sont encore obtenues dans la dernière phase lorsque le problème original est pris en compte.

Cette analyse est renforcée par la troisième partie des tableaux qui indique la moyenne des RPD à chaque phase. Ainsi, elle indique, à quelle distance de la meilleure solution connue de la littérature, sont les solutions atteintes après chaque phase. Pour les instances de taille 20, la meilleure solution connue, qui est optimale, est atteinte (RPD = 0). Pour la taille 50, les meilleures solutions sont souvent atteintes, mais pas pour chaque exécution. Le plus intéressant est pour les grandes instances, car les solutions sont bien meilleures que TMIIG. Une observation intéressante complémentaire pour les instances de grandes tailles est la grande amélioration entre l'initialisation et la première phase avec $\sigma = 60\%$. Dans les phases suivantes, même si les solutions sont améliorées, l'amélioration est moins importante par rapport au début, mais reste significative.

La dernière partie des deux tableaux indique le temps passé à chaque phase. Le critère d'arrêt, à chaque phase, est soit un temps maximal, soit un nombre maximal d'itérations sans amélioration, tous deux dépendant du nombre de super-jobs (voir Section 4.4.1). Ainsi, comme on peut s'y attendre, dans la plupart des cas, le temps passé augmente avec la valeur du niveau de confiance. Nous pouvons noter que le temps total est assez important (il faut encore ajouter le temps de calcul pour la génération du pool de solutions), mais l'objectif de l'approche est de montrer l'intérêt de l'intégration de connaissances pour améliorer les méthodes d'optimisation. Nous ne considérons donc pas les contraintes de temps de calcul réels.

En conclusion, l'utilisation des deux listes Σ_1 et Σ_2 donne des résultats similaires, avec un RPD moyen légèrement plus petit pour Σ_2. On remarque que les niveaux de confiance supplémentaires ont permis d'améliorer les résultats. Cela s'explique par le fait que seul un mouvement d'un nombre spécifique de tâches permet d'améliorer une solution. La découpe des super-jobs en augmentant la confiance de 20% rend le super-job trop petit pour réaliser un mouvement améliorant, alors qu'effectuer une diminution de la confiance plus progressive permet de trouver des solutions voisines améliorantes intermédiaires plus facilement. Ce qui explique l'amélioration du RPD moyen pour la liste Σ_2.

Instance	Taille Pool				
	5	10	15	20	25
20×5	0,00	0,00	0,00	0,00	0,00
20×10	0,00	0,00	0,00	0,00	0,00
20×20	0,00	0,00	0,00	0,00	0,00
50×5	0,06	0,05	0,04	0,04	0,04
50×10	0,06	0,05	0,04	0,05	0,04
50×20	0,03	0,03	0,03	0,03	0,03
100×5	0,31	0,24	0,25	0,23	0,25
100×10	0,20	0,16	0,14	0,12	0,12
100×20	0,17	0,13	0,12	0,11	0,10
200×10	0,32	0,25	0,25	0,25	0,24
200×20	0,28	0,22	0,21	0,20	0,20
500×20	0,20	0,15	0,15	0,14	0,15

TABLEAU 4.6 – Valeur moyenne du RPD pour chaque taille d'instance, en fonction de la taille de Pool d'apprentissage.

4.4.3 Discussion

Les résultats expérimentaux ont prouvé la performance de notre approche (IG_{SJ}) puisque les solutions ont été améliorées pour chaque instance de Taillard par rapport à la méthode de base. Cette section fournit une discussion sur deux aspects importants de la méthode : le temps de calcul requis pour la génération du pool de solutions, et l'analyse de la dynamique de la méthode.

Discussion sur la génération du pool $\mathscr{P}^*$ Le temps de calcul de la méthode proposée peut représenter un inconvénient lors de la résolution de grandes instances. Ce temps de calcul est en partie explicité par la génération du pool de solutions $\mathscr{P}^*$. Le fait est que la performance de l'approche est basée sur l'extraction des connaissances de ce pool de solutions initiales utilisées pour identifier les super-jobs pertinents. Nous avons constaté durant nos expérimentations que la qualité des solutions du pool impacte beaucoup les performances, et donc que les heuristiques constructives ne donnent pas des solutions de suffisamment bonne qualité pour identifier les super-jobs fiables. Par conséquent, nous avons choisi IG, avec une limite de temps, pour donner des solutions de bonne qualité pour le pool de solutions et cela prend donc du temps.

En outre, nous avons testé différentes tailles de pool comme le montre le Tableau 4.6. Nous pouvons observer que le RPD pour un pool de 5 solutions est bien plus élevé que pour des pools de 10 à 25 solutions. Cela s'explique par le fait que plus la taille est petite, plus la chance d'avoir un pool représentatif est faible. Pour les autres tailles de pools, il n'y a pas de différence significative pour le RPD.

À la suite de ces tests, nous avons décidé de générer un pool de 10 solutions seulement, car des super-jobs pertinents peuvent être découverts, avec uniquement 10 solutions. Un petit pool de solutions réduit considérablement le temps de calcul de l'approche. Cependant, le temps reste tout de même important. Par exemple, pour résoudre

une instance de 200 jobs, 400 secondes sont nécessaires pour générer une solution du pool, soit environ une heure pour les 10 solutions du pool. Cela est coûteux en temps de calcul, cependant notre objectif est d'obtenir de nouvelles meilleures solutions.

Puisque notre approche est stochastique, dans les expériences exposées, la performance est évaluée à partir de 30 exécutions de IG_{SJ} pour chaque instance. Chaque exécution génère son propre pool de solutions. Nous avons aussi conduit des expériences où un seul groupe de 10 solutions a été généré et partagé entre les 30 exécutions. Les résultats expérimentaux étaient similaires : les meilleures solutions de la littérature ont été atteintes pour les plus petites instances de Taillard, et améliorées pour les plus grandes. L'utilisation d'un tel pool partagé diminue l'ensemble du temps de calcul pour les 30 exécutions.

Lors de l'analyse de la méthode, nous avons également remarqué que pour les instances les plus importantes (avec 500 tâches principalement), un meilleur pool de solutions améliore l'identification des super-jobs et réduit davantage la combinatoire. Ainsi, nous pouvons imaginer laisser plus de temps à l'IG pour générer un seul pool de solutions de meilleure qualité, plutôt que générer un pool différent pour chaque exécution.

Discussion sur le comportement de la méthode Un autre aspect intéressant de la méthode est que sa performance repose sur la réduction de la combinatoire du problème qui est rendue possible par la structure particulière des optima locaux de bonne qualité. Dans l'espace de recherche, chaque optimum local est le «centre» d'un bassin d'attraction. Tous les bassins rendent le paysage très robuste pour les méthodes de recherche locale. La phase de perturbation de l'IG a été conçue pour échapper aux optima locaux, et donc, à partir de leurs bassins d'attraction. Cependant, les bassins d'attraction ne sont pas côte à côte, mais inclus les uns dans les autres ; le meilleur optimum local d'un grand bassin peut être le centre des autres. Par conséquent, même avec une perturbation, une méthode de recherche locale reste souvent dans le même grand bassin d'attraction. La réduction de la combinatoire du problème, avec l'identification des super-jobs, produit un effet intéressant sur le paysage. En effet, certains optima locaux originaux n'existent plus dans le paysage réduit et donc ses bassins d'attraction non plus. Par conséquent, les régions du paysage sont lissées, les bassins d'attractions originaux deviennent plus grands et la performance de l'IG de notre approche est améliorée. Par exemple, pour les instances avec 20 tâches (la plus facile des instances de Taillard), IG finit par converger vers des solutions proches des meilleures solutions de la littérature sans les atteindre à chaque fois, alors qu'avec la réduction de la combinatoire, IG atteint l'optimum à chaque fois. La réduction du nombre de bassins d'attraction aide donc l'IG à évoluer vers de meilleures zones de l'espace de recherche. L'exploitation des super-jobs efface les régions rugueuses de l'espace de recherche et augmente les performances de l'IG. Ces résultats encourageants nous conduisent à incorporer (IG_{SJ}) dans un schéma plus général.

4.5 IIG_{SJ} : Version itérée de IG_{SJ}

4.5.1 Description

Les résultats expérimentaux présentés ci-dessus ont montré que le mécanisme d'apprentissage est efficace pour les instances de petites et grandes tailles. En effet, IG_{SJ} est capable soit de trouver de nouvelles meilleures solutions, soit d'au moins atteindre les meilleures solutions connues pour les instances de Taillard. Pour les instances les plus grandes (et donc les plus difficiles) de taille 100, 200 et 500, de nouvelles meilleures solutions ont été découvertes. Cependant les valeurs de RPD successives (voir section 4.4.2 Tableaux 4.4 et 4.5) montrent que IG_{SJ} améliore encore la solution lorsque le problème original est atteint, c'est-à-dire lors de la dernière étape où aucun super-job n'est pris en compte ($\sigma = \infty$). Cela suggère que IG_{SJ} n'a pas fini de converger et peut être amélioré. Pour les instances de taille 500, nous avons remarqué que les solutions constituant le pool de départ et initiales utilisées pour identifier les super-jobs sont très diversifiées (la taille du problème était à peine réduite de moitié avec le plus faible niveau de confiance $\sigma = 60\%$) car l'IG original n'est pas suffisamment efficace pour une instance de cette taille. Sans aucun doute, cela a un effet sur les performances de l'exécution entière de IG_{SJ}. Itérer IG_{SJ} à partir des solutions obtenues à la fin des 30 exécutions pourrait améliorer la qualité ainsi qu'augmenter la taille des super-jobs identifiés et donc la performance de l'algorithme. L'approche itérative proposée (IIG_{SJ}) donnée dans l'Algorithme 4.3 est basée sur cette idée.

Algorithme 4.3 : IIG_{SJ} : IG_{SJ} itéré

Entrées : $\Sigma = \sigma_1, \sigma_2, \ldots$: liste des niveaux de confiance dans l'ordre croissant;
$\mathscr{P}_0$: ensemble initial de solutions;
I : nombre d'itérations ;
R : nombre de solutions construites à chaque itération i;
ρ : nombre de solutions utilisées pour l'apprentissage;
Données : $\mathscr{P}_i$: ensemble de solutions construites à l'itération i;
$\mathscr{P}_{tmp}$: Ensemble temporaire de solutions utilisées par IG_{SJ};
π : solution courante;
Sorties : π^* meilleure solution.

```
π* =meilleur(P_0) ;      // Initialiser la meilleure solution avec la meilleure
 solution de P_0
pour i dans 1..I faire
    /* PROCÉDURE INTERNE                                                    */
    P_i = ∅ ;                              // Initialiser P_i comme un ensemble vide
    pour k dans 1..R faire
        P_tmp = Sélectionner (ρ, P_{i-1}) ;  // Sélectionner ρ solutions parmi P_{i-1}
         et les placer dans P_tmp (i)
        π = (P_tmp, Σ) ;                                                  // (ii)
        P_i = P_{i-1} ∪ π ;                                  // Placer π dans P_i (iii)
        π* =meilleur({π, π*}) ;                  // Mémoriser la meilleure solution
retourner π*
```

IIG_{SJ} commence avec un ensemble $\mathscr{P}_0$ de R solutions, itère I fois la procédure interne et renvoie la meilleure solution π^* trouvée. La procédure interne vise à construire des ensembles de solutions de meilleure qualité afin d'espérer identifier des super-jobs proches de ceux de la solution optimale. La figure 4.4 donne une illustration de cette procédure interne. À l'itération $i \in$ I, l'algorithme commence avec $\mathscr{P}_i$, un ensemble vide où les nouvelles solutions R seront ajoutées itérativement en suivant ces trois étapes : (i) d'abord, les solutions ρ sont uniformément sélectionnées au hasard dans l'ensemble $\mathscr{P}_{i-1}$ et stockées dans un ensemble temporaire $\mathscr{P}_{tmp}$, puis (ii), IG_{SJ} est appliquée avec $\mathscr{P}_{tmp}$ et Σ pour obtenir une nouvelle solution π qui est finalement (iii), stockée dans $\mathscr{P}_i$, l'ensemble de l'itération courante i ; si π est meilleure que la solution π^* actuelle, elle la remplace. Le paramètre ρ est utilisé pour sélectionner un sous-ensemble de solutions pour maintenir la diversité dans le pool construit $\mathscr{P}_{tmp}$, sinon les mêmes super-jobs seraient identifiés pour chaque niveau de confiance de la Phase (ii).

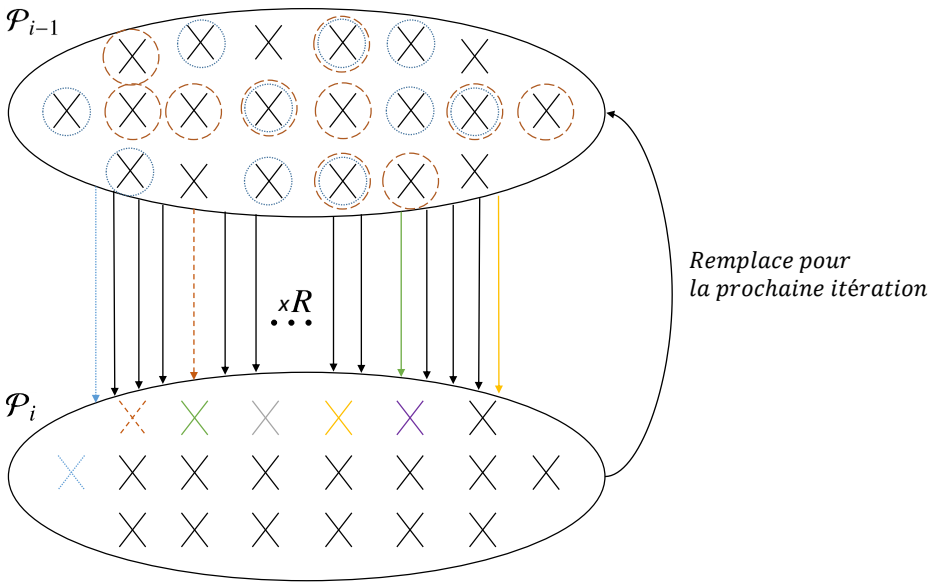


FIGURE 4.4 – Illustration de la procédure interne de IIG_{SJ} à l'itération i. Pour chaque exécution (représentée par une couleur), ρ solutions (les croix encerclées) sont uniformément sélectionnées dans $\mathscr{P}_{i-1}$ et utilisées pour identifier les super-jobs. IG_{SJ} est exécuté (flèche vers le bas) pour donner une nouvelle solution stockée dans $\mathscr{P}_i$. Ce processus est répété R fois. A la fin de l'itération i, $\mathscr{P}_i$ contient R solutions et remplacera $\mathscr{P}_{i-1}$ pour l'itération suivante $i+1$.

4.5.2 Expérimentations

4.5.2.1 Protocole expérimental

IIG_{SJ} présente des paramètres supplémentaires en plus de ceux de IG_{SJ} . Nous avons choisi de fixer le nombre R de solutions d'un pool $(\mathscr{P}_i)_{i\geq 0}$, ce qui est statistiquement rai-

Instance	Makespan	Écart	Instance	Makespan	Écart	Instance	Makespan	Écart	Instance	Makespan	Écart
Ta01	1,486	0	Ta31	**3,160**	-1	Ta61	**6,366**	-31	Ta91	**15,248**	-71
Ta02	1,528	0	Ta32	3,432	0	Ta62	**6,219**	-15	Ta92	**15,007**	-78
Ta03	1,460	0	Ta33	**3,210**	-1	Ta63	**6,108**	-13	Ta93	**15,276**	-100
Ta04	1,588	0	Ta34	**3,338**	-1	Ta64	**6,001**	-25	Ta94	**15,117**	-83
Ta05	1,449	0	Ta35	3,356	0	Ta65	**6,183**	-17	Ta95	**15,113**	-96
Ta06	1,481	0	Ta36	**3,346**	-1	Ta66	**6,058**	-16	Ta96	**14,997**	-112
Ta07	1,483	0	Ta37	3,231	0	Ta67	**6,224**	-23	Ta97	**15,300**	-95
Ta08	1,482	0	Ta38	3,235	0	Ta68	**6,115**	-15	Ta98	**15,162**	-75
Ta09	1,469	0	Ta39	**3,070**	-2	Ta69	**6,359**	-11	Ta99	**15,012**	-88
Ta10	1,377	0	Ta40	3,317	0	Ta70	**6,371**	-10	Ta100	**15,259**	-81
Ta11	2,044	0	Ta41	4,274	0	Ta71	**8,059**	-18	Ta101	**19,551**	-130
Ta12	2,166	0	Ta42	4,177	0	Ta72	**7,859**	-21	Ta102	**19,980**	-116
Ta13	1,940	0	Ta43	4,099	0	Ta73	**8,017**	-11	Ta103	**19,791**	-122
Ta14	1,811	0	Ta44	4,399	0	Ta74	**8,330**	-18	Ta104	**19,775**	-153
Ta15	1,933	0	Ta45	4,322	0	Ta75	**7,939**	-19	Ta105	**19,732**	-111
Ta16	1,892	0	Ta46	4,289	0	Ta76	**7,773**	-28	Ta106	**19,852**	-90
Ta17	1,963	0	Ta47	4,420	0	Ta77	**7,851**	-15	Ta107	**19,967**	-145
Ta18	2,057	0	Ta48	4,318	0	Ta78	**7,881**	-32	Ta108	**19,900**	-156
Ta19	1,973	0	Ta49	4,155	0	Ta79	**8,137**	-24	Ta109	**19,817**	-101
Ta20	2,051	0	Ta50	4,283	0	Ta80	**8,095**	-19	Ta110	**19,794**	-141
Ta21	2,973	0	Ta51	6,129	0	Ta81	**10,676**	-24	Ta111	**46,264**	-425
Ta22	2,852	0	Ta52	5,725	0	Ta82	**10,562**	-32	Ta112	**46,797**	-478
Ta23	3,013	0	Ta53	5,862	0	Ta83	**10,591**	-20	Ta113	**46,154**	-390
Ta24	3,001	0	Ta54	5,788	0	Ta84	**10,588**	-19	Ta114	**46,556**	-343
Ta25	3,003	0	Ta55	5,886	0	Ta85	**10,507**	-32	Ta115	**46,402**	-339
Ta26	2,998	0	Ta56	5,863	0	Ta86	**10,624**	-66	Ta116	**46,667**	-274
Ta27	3,052	0	Ta57	5,962	0	Ta87	**10,793**	-32	Ta117	**46,170**	-339
Ta28	2,839	0	Ta58	5,926	0	Ta88	**10,801**	-38	Ta118	**46,495**	-378
Ta29	3,009	0	Ta59	5,876	0	Ta89	**10,703**	-20	Ta119	**46,408**	-335
Ta30	2,979	0	Ta60	5,958	0	Ta90	**10,752**	-46	Ta120	**46,433**	-414

TABLEAU 4.7 – Comparaison des solutions avec TMIIG (L'écart est calculé par rapport à TMIIG). En gras, cela indique qu'une solution est meilleure que celle de TMIIG.

sonnable pour évaluer la performance moyenne de l'approche ; De même, nous choisissons de fixer le nombre de solutions ρ à 10 comme dans les expérimentations de la Section 4.4 où 10 solutions ont été utilisées pour identifier les super-jobs, et le nombre d'itérations I est fixé à 5, suffisant pour converger et éviter le sur-apprentissage. L'Iterated greedy (IG) de RUIZ et STÜTZLE [2007] est toujours l'algorithme utilisé dans IG_{SJ} pour améliorer les solutions. Il est arrêté selon deux conditions : soit un temps maximal de n_{sj}^2 ms (où n_{sj} est le nombre actuel de super-jobs), soit un nombre maximal d'itérations sans amélioration de $25 * n$ est atteint. Notez que ce critère d'arrêt est plus court que celui défini dans les expériences précédentes (Section 4.4). En effet, puisque le processus est itéré, la fin de la convergence n'est pas nécessaire pour chaque exécution de IG_{SG}. De plus, l'utilisation de IG_{SJ} nécessite la définition du paramètre Σ pour identifier les tâches avec différents niveaux de confiance. Nous utilisons $\Sigma = \{60\%, 70\%, 80\%, 90\%, \infty\}$ car les expérimentations précédentes ont montré de meilleurs résultats pour les instances de grandes tailles avec cette liste de valeurs

4.5.2.2 Résultats

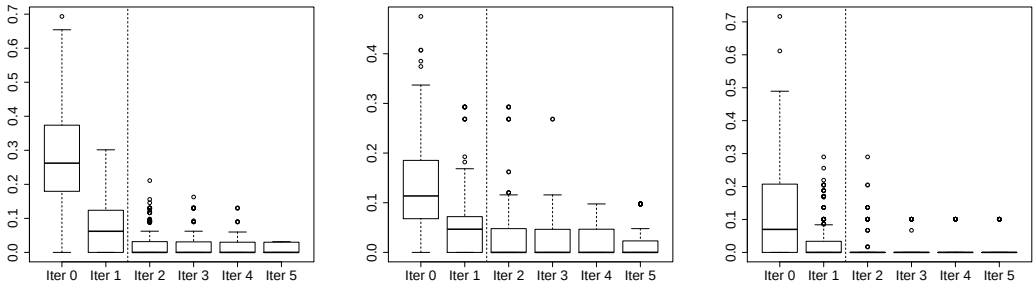
Le tableau 4.7 rapporte les meilleures solutions obtenues par IIG_{SJ} pour toutes les instances de Taillard et donne la valeur d'écart (Gap) entre les résultats obtenus par

notre approche et les meilleures solutions antérieures de la littérature obtenues par TMIIG de DING et collab. [2015]. Un écart égal à 0 signifie que IIG_{SJ} atteint les meilleures solutions de TMIIG, et un écart strictement négatif signifie qu'il trouve une solution de meilleure qualité que TMIIG. Ce tableau montre que pour les plus grandes instances de taille 100, 200 et 500 tâches, IIG_{SJ} améliore les résultats déjà obtenus avec IG_{SJ} et découvre même de meilleures solutions. La qualité de la solution la plus améliorée a eu son makespan minimisé de 1% sur l'instance Ta112 par exemple. Ces résultats montrent que IIG_{SJ} est très efficace pour résoudre des instances de grandes tailles du problème d'ordonnancement de flowshop sans temps d'attente comme les instances de Taillard.

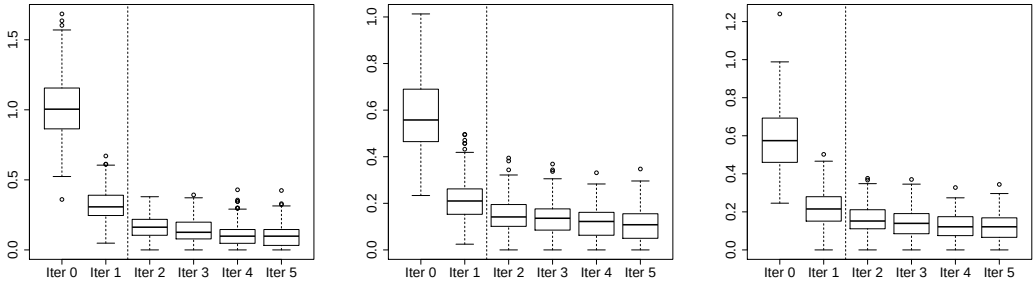
	Iter 0	Iter 1		Iter 2		Iter 3		Iter 4		Iter 5		Total
Instances	RPD	Temps (s)	RPD	Temps (s)	RPD	Temps (s)	RPD	Temps (s)	RPD	Temps (s)	RPD	Temps (s)
20×5	0.000	14	0.000	14	0.000	14	0.000	14	0.000	14	0.000	71
20×10	0.000	14	0.000	14	0.000	14	0.000	14	0.000	14	0.000	71
20×20	0.015	15	0.001	14	0.000	14	0.000	14	0.000	14	0.000	71
50×5	0.276	129	0.073	112	0.029	103	0.020	101	0.013	100	0.009	545
50×10	0.128	124	0.059	107	0.037	101	0.020	97	0.019	98	0.019	527
50×20	0.120	111	0.034	100	0.016	97	0.010	96	0.010	94	0.010	497
100×5	1.015	742	0.317	370	0.162	253	0.134	236	0.107	229	0.102	1830
100×10	0.577	332	0.215	282	0.147	254	0.134	243	0.124	231	0.112	1341
100×20	0.587	324	0.222	265	0.159	244	0.137	226	0.127	206	0.124	1264
200×10	1.431	1835	0.441	1281	0.187	1084	0.102	1005	0.085	976	0.084	6181
200×20	1.180	1726	0.366	1269	0.206	1116	0.134	1048	0.105	908	0.082	6068
500×20	1.748	32906	0.598	20789	0.299	16668	0.157	14747	0.090	11728	0.055	96838

TABLEAU 4.8 – Analyse de la dynamique de IIG_{SJ}. Les résultats obtenus pour chaque itération sont présentés selon les 12 tailles différentes des instances de Taillard N × M. La valeur RPD est calculée à partir de la qualité des meilleures solutions indiquée dans le Tableau 4.7. Les temps sont donnés en secondes. Pour chaque itération, les valeurs rapportées de RPD et de temps sont les valeurs moyennes calculées sur 200 exécutions (10 instances/taille x 20 exécutions).

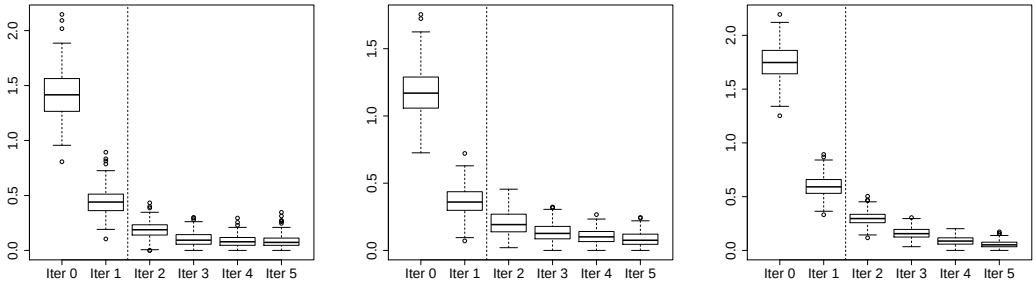
Dans ce qui suit, nous détaillons les résultats obtenus par IIG_{SJ} et discutons de l'intérêt d'utiliser IG_{SJ} itérativement. Nous présentons les résultats en regroupant les instances de Taillard en fonction des 12 tailles différentes (N × M) puisque le nombre de tâches N et le nombre de machines M impactent la résolution du problème. Dans nos expérimentations, une itération de IIG_{SJ} fournit 20 solutions. Toutes les solutions obtenues après chaque itération lors d'une exécution de IIG_{SJ} sont mémorisées afin de faire une analyse *a posteriori* pour valider l'intérêt d'itérer IG_{SJ}. Après avoir exécuté IIG_{SJ} sur les instances de Taillard, nous comparons les qualités des solutions obtenues après chaque itération avec la (nouvelle) qualité obtenue des solutions par rapport à TMIIG (valeurs rapportées dans le tableau 4.7) pour calculer la valeur de RPD (voir Section 4.4.1). Le tableau 4.8 donne le RPD moyen calculé à partir des 200 valeurs obtenues (20 solutions par itération, 10 instances par taille) pour chaque itération de IIG_{SJ}. Une valeur de RPD nulle signifie que la qualité des 200 solutions fournies à l'itération considérée est égale à la qualité de la meilleure solution connue pour chacune des 10 instances respectivement. Une valeur de RPD strictement positive signifie qu'au moins une solution fournie à la fin de l'itération n'a pas la meilleure qualité connue. Comme prévu, le RPD moyen diminue avec les itérations successives qui montrent l'intérêt d'exploiter la solution fournie par une itération pour identifier de nouveaux et meilleurs super-jobs.



(a) 50 × 05 (b) 50 × 10 (c) 50 × 20



(d) 100 × 05 (e) 100 × 10 (f) 100 × 20



(g) 200 × 10 (h) 200 × 20 (i) 500 × 20

FIGURE 4.5 – Boîte à moustache des valeurs de RPD pour 5 itérations de IIG_{SJ}.

Cette diminution est illustrée sur la figure 4.5 qui montre les boîtes à moustaches associées pour les instances les plus difficiles (20 instances de tâches sont résolues de façon optimale dès la première itération).

Une ligne sépare chaque graphique : la partie gauche correspond à la première itération (comme une exécution de IG_{SJ}) tandis que la partie droite correspond aux prochaines itérations effectuées dans IIG_{SJ}. Nous observons que plus la taille de l'instance est grande, plus l'amélioration de la qualité médiane est importante. Entre les itérations 4 et 5, pour différentes instances (*par exemple* 100 × 10, 100 × 20, 200 × 10), certaines qualités sont même détériorées. Ceci peut être expliqué par un sur-apprentissage où les solutions dans les ensembles $\mathscr{P}_3$ ou $\mathscr{P}_4$ sont trop similaires, et donc l'approche a du mal

à détecter de nouveaux super-jobs et reste coincée dans une région particulière de l'espace de recherche. Par conséquent, le nombre d'itérations de IIG_{SJ} doit être réglé avec soin pour éviter un sur-apprentissage et une perte de temps.

Le tableau 4.8 indique également, en secondes, le temps d'exécution moyen pour chaque itération et la moyenne du temps total. Comme le temps d'exécution dépend du nombre de (super-)jobs (voir section 4.5.2.1), il augmente lorsque le nombre de jobs augmente. Cette valeur atteint près de 28 heures pour les instances les plus importantes (taille 500). Évidemment, ce temps n'est pas satisfaisant en pratique. Mais, dans ces expérimentations, notre objectif est simplement d'améliorer les performances de notre approche initiale IIG_{SJ} pour améliorer les meilleures solutions trouvées par TMIIG et montrer l'intérêt de l'intégration de connaissances; ce qui a été fait. Si nous analysons plus attentivement les temps moyens pour chaque itération, nous observons une réduction du temps inversement proportionnelle à la taille de l'instance, plus cette réduction est importante, plus le nombre de tâches est élevé. Nous pouvons expliquer cela par l'augmentation de la taille des super-jobs et la diminution de leur nombre. Les solutions sont de plus en plus similaires au sein de l'ensemble et peuvent donc partager de plus grandes sous-séquences de tâches qui donnent un nombre de plus en plus petit de tâches.

4.5.3 Positionnement dans la littérature

La méthode que nous avons proposée a été dépassée par une approche plus récente que TMIIG qui a été réalisée en parallèle de notre travail. L'approche de Lin et Ying [2016] propose de réduire le problème en un problème de voyageur de commerce pour résoudre le problème de façon optimale très rapidement en utilisant un solveur existant (Gurobi).

Le tableau 4.9 montre le RPD de notre approche par rapport aux solutions optimales trouvées par Lin et Ying [2016]. On peut remarquer que le RPD est très peu différent de celui que l'on avait obtenu dans le tableau 4.8. Cela montre donc que notre approche était très proche des solutions optimales, ce qui confirme d'autant plus que l'intégration de connaissance a permis à l'IG, une méta-heuristique relativement simple, d'atteindre des solutions quasi optimales.

4.6 Conclusion

Dans ce chapitre, nous avons présenté une nouvelle approche basée sur l'algorithme Iterated Greedy utilisant l'intégration de connaissances, qui a trouvé 64 nouvelles meilleures solutions pour les instances de Taillard pour le problème d'ordonnancement de flow-shop de permutation sans temps d'attente (NWFSP) (avant l'article de Lin et Ying [2016]). La nouveauté de cette approche est de modifier le paysage lors de la recherche en réduisant d'abord la taille du problème initial puis en l'augmentant. Ce processus est possible grâce à la structure commune observée sur les solutions de bonne qualité du NWFSP qui présentent des sous-séquences communes de tâches consécutives. L'identification de ces sous-séquences a été discutée dans ce chapitre et a conduit à la définition de

Instances	RPD
20×5	0
20×10	0
20×20	0
50×5	0.009
50×10	0.019
50×20	0.060
100×5	0.102
100×10	0.112
100×20	0.085
200×10	0.083
200×20	0.083
500×20	0.058

TABLEAU 4.9 – La valeur RPD est calculée à partir des nouvelles meilleures solutions de littérature de LIN et YING [2016].

super-jobs selon un niveau de confiance. Les super-jobs sont considérés comme une seule tâche du problème, le niveau de confiance détermine donc indirectement la taille du problème à résoudre. L'approche proposée consiste à exploiter successivement ces super-jobs dans l'exécution de l'Iterated Greedy.

Dans les expérimentations, nous montrons que le nombre de super-jobs identifiés avec un niveau de confiance moyen (ici, 60%) réduit de moitié la taille du problème. Par conséquent, l'algorithme Iterated Greedy découvre plus facilement de bonnes régions de l'espace de recherche. L'augmentation du niveau de confiance décompose peu à peu les super-jobs précédemment identifiés, l'espace de recherche se révèle donc progressivement.

En outre, IG_{SJ} a été intégrée dans une approche itérative, appelée IIG_{SJ}, pour tirer parti des solutions trouvées après chaque exécution pour identifier de meilleurs super-jobs. Malgré le temps de calcul élevé de cette méthode, les performances de IG_{SJ} ont été améliorées avec succès puisque les solutions trouvées par IIG_{SJ} sont bien meilleures. Ce travail montre l'intérêt de l'utilisation des connaissances dans les méthodes d'optimisation telles que les méta-heuristiques, et montre l'impact de la qualité des connaissances utilisées.

Chapitre 5

Réduction du voisinage courant par intégration de connaissances

Sommaire

5.1 Introduction

Dans le chapitre précédent, nous avons utilisé l'intégration de connaissances dans le but de modifier le paysage de l'espace de recherche en agissant sur la structure des solutions. Dans ce chapitre, nous allons utiliser l'intégration de connaissances afin de réduire la taille du voisinage du problème présenté dans le chapitre 1 : la sélection d'attributs pour la classification supervisée.

La sélection d'attributs pour la classification supervisée est un problème composé de deux sous-problèmes : la classification supervisée et la sélection d'attributs. Pour le premier problème, la classification supervisée, nous disposons d'un ensemble d'observations, tel qu'à chaque observation, un label (aussi appelé classe) est associé. Chacune de ces observations est définie par un ensemble d'attributs (aussi appelé caractéristiques). L'objectif de la classification supervisée est de créer un modèle de classification à partir de ces données afin de prédire le label des nouvelles observations. Cependant, trouver un modèle de classification pertinent peut-être difficile si le nombre d'attributs est important. Afin de pallier cette problématique, la résolution du second problème est nécessaire.

Le problème de sélection d'attributs consiste ainsi à choisir un sous-ensemble pertinent d'attributs pour un problème donné. Dans notre problématique, la sélection d'attributs doit donc choisir les attributs qui vont permettre de rendre plus efficace la recherche du modèle associé à une classification supervisée. Ainsi la sélection d'attributs a plusieurs objectifs :

1. Réduire le sur-apprentissage, c'est-à-dire réduire la spécialisation du modèle de classification aux observations connues. Ce qui permet d'augmenter la robustesse de la classification sur la prédiction des nouvelles données.
2. Réduire le temps de calcul des algorithmes pour la classification, car ils ont moins d'attributs à traiter ce qui permet de trouver plus facilement un bon modèle de classification.
3. Simplifier la compréhension d'un modèle de classification. En effet, moins il y aura d'attributs et moins il y aura de règles complexes pour la classification.

Corne et collab. [2012] ont montré que la sélection d'attributs pour la classification supervisée peut-être modélisée sous la forme d'un problème d'optimisation combinatoire. En effet, ce problème consiste à choisir un sous-ensemble d'attributs parmi un plus grand ensemble de n attributs. Il existe ainsi, un nombre fini de solutions (2^n solutions, sachant qu'un attribut est soit sélectionné, soit pas). De plus, pour évaluer la qualité d'une solution, le sous-ensemble d'attributs est utilisé pour générer un modèle de classification, et la qualité de ce modèle détermine alors la qualité du sous-ensemble d'attributs. Cependant, la création d'un modèle de classification à partir d'un ensemble d'attributs peut-être une opération coûteuse en temps en fonction de l'algorithme de classification, ce qui peut rendre inefficaces les méthodes d'optimisation.

L'objectif de ce chapitre est ainsi de proposer une approche avec intégration de connaissances dans une méthode d'optimisation combinatoire, tout en utilisant un algorithme

de classification très performant : SVM, afin de résoudre des problèmes de très grandes tailles. Nous proposons ici d'utiliser l'intégration de connaissances pour apprendre des structures des bonnes solutions, les attributs les plus pertinents. Cela permettra entre autres de déterminer si une solution, avant même de l'évaluer, est potentiellement une bonne solution ou non. Ainsi, notre approche évaluera uniquement les solutions les plus prometteuses et évitera l'évaluation coûteuse de nombreuses solutions.

Pour cela, nous allons utiliser une recherche locale avec un mécanisme tabou et une matrice de connaissances inspirée des algorithmes à colonies de fourmis afin de sélectionner les solutions à évaluer et donc réduire le nombre d'évaluations. Cette matrice permettra aussi bien d'intensifier (recherche locale améliorante) que de diversifier (perturber les solutions afin d'explorer d'autres zones de l'espace de recherche) les solutions parcourues. De plus, contrairement à ce qui a été jusqu'alors proposé dans cette thèse, nous proposons ici une méthode dynamique, c'est-à-dire que la connaissance extraite évolue au cours de la recherche. Ainsi, conformément à notre taxonomie présentée dans le chapitre 2, la méthode proposée dans ce chapitre est de la forme suivante : $S_{cur}/ML^k/(DIV+INT)*$. Cette approche a fait l'objet d'une publication dans la conférence LION 2016 (MOUSIN et collab. [2016]).

Le reste de ce chapitre est organisé comme suit : la section 5.2 présente l'algorithme utilisé : le Learning Tabu Search. Ensuite, la section 5.3 rappelle quelques notions de la sélection d'attributs pour la classification supervisée et modélise ce problème pour une recherche locale. Ensuite, nous proposons une adaptation du Learning Tabu Search pour la sélection d'attributs dans la section 5.4. Et enfin, la section 5.5 valide notre approche grâce aux expérimentations, et compare les résultats avec l'approche classique de la recherche tabou afin de montrer la contribution du mécanisme d'apprentissage.

5.2 Une recherche tabou avec intégration de connaissances : Le Learning Tabu Search

Dans les recherches locales, et en particulier dans la recherche tabou, l'exploration du voisinage d'une solution peut être coûteuse en temps de calcul. En effet, dans la recherche tabou telle qu'elle est définie dans la littérature, au moins toutes les solutions voisines non tabou sont évaluées à chaque itération. Cependant, dans le problème de sélection d'attributs, l'évaluation d'une solution est donnée par la qualité de classification du modèle généré par l'algorithme de classification (Exemple : k plus proches voisins (k-NN), machine à vecteurs de support (SVM) ...) où une évaluation peut devenir coûteuse en temps en fonction de ce dernier. L'exploration du voisinage peut alors rapidement devenir non réalisable lorsque le nombre d'observations et/où d'attributs devient grand. Ainsi, une évaluation complète du voisinage à chaque itération ne peut pas être considérée.

Dans un même contexte d'évaluation coûteuse, ZUFFEREY et SCHINDL [2015], pour un problème de réseau de chemin de fer, ont défini une recherche tabou intégrant de la connaissance pour pallier le problème de l'évaluation coûteuse : le *Learning Tabu Search* (LTS). Dans cette approche, l'exploration du voisinage est divisée en deux étapes :

(i) l'estimation du voisinage, (ii) l'évaluation des Q voisins les plus prometteurs. Ainsi, LTS est basé sur une *fonction d'estimation* utilisée pour estimer la qualité potentielle de chaque solution voisine. Le calcul de cette estimation est basé sur l'idée suivante :

"Soit une combinaison de caractéristiques définies, si des combinaisons de caractéristiques sont souvent présentes dans les bonnes solutions pendant la recherche, alors ces combinaisons doivent être privilégiées pour la génération des nouvelles solutions"

En d'autres termes, la qualité d'une combinaison de caractéristiques est déterminée par la qualité des solutions dans lesquelles cette combinaison apparaît. La définition d'une combinaison de caractéristiques est à définir en fonction du problème (Exemple : Présence de deux attributs ensemble dans une solution, Succession de tâches consécutives dans un problème d'ordonnancement, etc.). Pour estimer cette qualité, LTS a besoin d'une mémoire pour sauvegarder la qualité de chaque combinaison de caractéristiques.

La représentation de la mémoire utilisée pour le Learning Tabu Search s'inspire grandement des phéromones utilisées dans les algorithmes à colonies de fourmis (ACO - DORIGO et collab. [1996]). Dans ce type d'algorithme, les fourmis construisent des solutions composant par composant. Le choix de ces composants est réalisé en fonction de la *trace* (appelée *trail*) laissée par les fourmis. La valeur de cette trace est proportionnelle à la qualité de la solution construite par les fourmis afin de guider plus rapidement les prochaines fourmis vers des solutions plus prometteuses. Ainsi, la valeur de *trail* va indiquer le chemin à prendre pour construire des bonnes solutions.

Dans le cas du Learning Tabu Search, la valeur de trail représente l'importance d'une combinaison de caractéristiques à être présente dans la solution. En d'autres termes, cela permet de diriger une solution afin d'obtenir ces combinaisons de caractéristiques. Une grande valeur de trail implique une combinaison de bonne qualité. Comme dans les algorithmes de fourmis, la mémoire a besoin d'être mise à jour pour que la valeur de trail des combinaisons prometteuses augmente, et à l'inverse, que la valeur diminue pour les combinaisons qui n'apparaissent plus dans les bonnes solutions. Pour l'oubli des combinaisons qui n'apparaissent plus, un mécanisme d'évaporation est généralement mis en place.

Contrairement aux algorithmes de fourmis, le Learning Tabu Search ne met pas à jour sa mémoire à chaque itération. La mise à jour se fait à un intervalle régulier, appelé *cycle*. Les meilleures solutions trouvées pendant chaque cycle sont utilisées pour mettre à jour la valeur de trail des différentes combinaisons. La taille d'un cycle est un paramètre sensible de LTS car elle a un fort impact sur les performances du mécanisme d'apprentissage. Un cycle trop petit signifie que la solution ne changera pas beaucoup avant la prochaine mise à jour, et donc que l'algorithme risque de renforcer des mauvaises combinaisons. A l'inverse, un cycle trop grand risque d'avoir des difficultés à trouver des combinaisons communes entre les solutions.

Pour éviter que l'algorithme ne converge vers une zone précise de l'espace de recherche dû à la direction forcée par l'apprentissage, un mécanisme de *diversification* est mis en place dans le but de visiter de nouvelles régions de l'espace de recherche. Ce mécanisme modifie la politique de choix du voisinage, ainsi, au lieu de choisir les Q voisins les plus prometteurs, pendant une phase de diversification, ce sont les Q voisins les

moins prometteurs qui seront choisis.

L'algorithme 5.1 donne un aperçu global du schéma d'un algorithme Learning Tabu Search. À partir d'une solution initiale, différentes étapes sont appliquées jusqu'à ce que le critère d'arrêt soit atteint. Tous les voisins non tabous sont estimés et ensuite, les Q voisins sélectionnés en accord avec la politique de diversification sont évalués. Les voisins les plus prometteurs avec l'estimation la plus forte sont choisis lorsqu'il n'y a pas de diversification. À l'inverse, les voisins avec l'estimation la plus faible sont choisis lors qu'il y a diversification. À la fin de l'exploration du voisinage, la meilleure solution depuis le début de la recherche s^*, la meilleure solution du cycle courant $\hat{s}$ et la liste tabou sont mises à jour. À la fin de chaque cycle, les valeurs de trails sont mises à jour avec la qualité de $\hat{s}$.

Algorithme 5.1 : Learning Tabu Search (LTS)

Sorties : s^* la meilleure solution trouvée
début
  $s \leftarrow$ *solution_initiale()*;
  $s^* \leftarrow$ s;
  $\hat{s} \leftarrow$ s;
  répéter
    Estimer la qualité des voisins non tabou de $\mathcal{N}(s)$;
    $N_Q \leftarrow$ Q voisins de $\mathcal{N}(s)$ sélectionnés en accord avec la politique de diversification;
    $s \leftarrow argmax_{s' \in N_Q} f(s')$;
    si $f(s) > f(\hat{s})$ **alors**
      $\hat{s} \leftarrow s$;
      si $f(s) > f(s^*)$ **alors**
        $s^* \leftarrow s$;
    Mise à jour de la liste tabou;
    si *Fin du cycle* **alors**
      Mettre à jour le trail de chaque combinaison de caractéristique avec $\hat{s}$;
      $\hat{s} \leftarrow s$;
  jusqu'à *condition d'arrêt rencontrée*;
  retourner *s**

Pour utiliser LTS dans le cadre du problème de sélection d'attributs, nous devons dans un premier temps modéliser le problème de sélection d'attributs pour une recherche locale, puis définir les différentes parties de LTS pour la sélection d'attributs, à savoir : la définition d'une combinaison de caractéristiques et la valeur de trail associée, la définition de la fonction d'estimation, la sélection des voisins prometteurs, la procédure de mise à jour du trail et le mécanisme de diversification.

5.3 Modélisation d'un problème de sélection d'attributs

Avant de modéliser le problème de sélection d'attributs comme un problème d'optimisation combinatoire, nous allons faire un rapide rappel des notions abordées dans le premier chapitre qui sont nécessaires à la bonne compréhension de ce chapitre.

5.3.1 Classification supervisée et sélection d'attributs

Dans un problème de classification supervisée, nous avons comme données d'entrée un ensemble d'observations et pour chaque observation, un label est associé. L'objectif est ainsi de créer un classifieur (qui peut-être décrit par un ensemble de règles par exemple) pour réaliser la classification de nouvelles données, c'est-à-dire leur affecter un label parmi ceux observés. L'exemple donné dans le tableau 5.1 montre un ensemble de quatre observations.

attributs / observations	Taille	Couleur yeux	Poids	Fumeur	Âge	Label
Observation 1	1m75	Bleu	95Kg	Non	27 ans	Homme
Observation 2	1m56	Marron	55Kg	Oui	15 ans	Femme
Observation 3	1m63	Bleu	75Kg	Oui	20 ans	Homme
Observation 4	1m51	Vert	48Kg	Oui	18 ans	Femme
Nouvelle observation	1m61	Bleu	67Kg	Non	21 ans	**?**

TABLEAU 5.1 – Exemple de classification d'individus homme/femme

À partir de ces observations, il faut déterminer un ensemble de règles pour réaliser une bonne prédiction, à savoir, trouver si un individu est un homme ou une femme à partir de certaines caractéristiques. Avec ces 4 observations, les règles suivantes peuvent être établies.

1. Les hommes mesurent plus de 1m60, ont plus de 19 ans, ont les yeux bleus, ont un poids de plus de 65Kg.
2. Les femmes mesurent moins de 1m60, ont moins de 19 ans, sont fumeuses, ont un poids de moins de 65Kg.

Ainsi, la nouvelle observation correspondrait plus à un homme. Or on sait que, les attributs "Couleur des yeux", "Fumeur" et "Âge" n'ont aucune incidence pour déterminer le sexe d'une personne.

La sélection d'attributs peut-être utilisée afin de ne garder qu'un ensemble pertinent d'attributs. Ainsi une bonne solution pour notre exemple pourrait être le sous-ensemble d'attributs $\{Taille, Poids\}$.

Pour résoudre ce problème (sélectionner les attributs pertinents pour trouver un bon modèle de classification supervisée), nous allons le modéliser sous la forme d'un problème d'optimisation combinatoire où l'on cherche à sélectionner un sous-ensemble d'attributs parmi n.

5.3.2 Représentation d'une solution

Nous représenterons une solution s (un sous-ensemble d'attributs) sous la forme d'un vecteur 0-1 de taille n, le nombre total d'attributs :

$$s = [a_1, ..., a_n], \text{ tel que } a_i \in \{0,1\}, \forall i \in \{1, ..., n\} \tag{5.1}$$

Le $i^{ème}$ élément du vecteur indique si l'attribut i est choisi ($a_i = 1$) ou s'il ne l'est pas ($a_i = 0$). L'espace des solutions est donc défini par tous les sous-ensembles possibles d'attributs sélectionnés parmi n, soit 2^n solutions.

Dans l'exemple de la section 5.3.1, la solution proposée serait donc représentée sous la forme $s = [1,0,1,0,0]$.

5.3.3 Évaluation d'une solution

La qualité d'une solution pour le problème de sélection d'attributs pour la classification supervisée est définie selon plusieurs critères. Les deux critères étudiés dans ces travaux sont :

— **le nombre d'attributs sélectionnés** (noté *features*) à minimiser

— **le pourcentage d'observations correctement labellisées** (noté *accuracy*) à maximiser

Le premier critère, *features*, est un critère très important. En effet, l'objectif initial de la sélection d'attributs est de réduire la taille des données à traiter par les algorithmes de classifications, de plus, cela permet aussi de simplifier la compréhension d'un modèle de classification. Ce critère est donc à minimiser et est défini par le ratio entre le nombre d'attributs sélectionnés (noté #S_F$eatures$) et le nombre total d'attributs (noté n). Dans le but d'avoir deux critères à maximiser, nous définissons la formule suivante pour ce critère :

$$features = 1 - \frac{\#\text{S\_F}eatures}{n} \tag{5.2}$$

Le second critère, le pourcentage d'observations correctement labellisées, aussi appelé *précision* ou *accuracy*, est défini par le ratio entre les observations correctement labellisées et le nombre total d'observations. L'*accuracy* est à maximiser et est défini par la formule suivante :

$$accuracy = \frac{\text{nombre d'observations correctement labellisées}}{\text{nombre total d'observations}} \tag{5.3}$$

Nous sommes donc dans un contexte bi-objectif où nous avons deux objectifs à maximiser. Cependant, nous considérerons la sélection d'attributs comme un problème mono-objectif grâce à une fonction d'agrégation f qui est une somme pondérée des deux objectifs *accuracy* et *features* :

$$f = \alpha * accuracy + (1-\alpha) * features, \text{ tel que } \alpha \in [0,1] \tag{5.4}$$

Ainsi nous cherchons à trouver le sous-ensemble d'attributs qui maximise la fonction f.

Solution initiale s	0	1	1	0	1
Solution voisine s'_1	1	1	1	0	1
Solution voisine s'_2	0	0	1	0	1
Solution voisine s'_3	0	1	0	0	1
Solution voisine s'_4	0	1	1	1	1
Solution voisine s'_5	0	1	1	0	0

TABLEAU 5.2 – Voisinage d'une solution à 5 attributs

5.3.4 Voisinage d'une solution

Comme nous avons modélisé la sélection d'attributs comme un problème 0-1, nous considérons comme opérateur de voisinage, l'opérateur *one-flip*, qui consiste à l'inversion d'un seul bit de la solution, ainsi pour toute solution s de l'espace de recherche, l'ensemble des solutions voisines est défini par :

$$\mathcal{N}_1^0(s) = \{s' \mid \exists i \in \{1,...,n\} \text{ tel que } a'_i \neq a_i \text{ et } \forall j \neq i,\ a'_j = a_j\} \tag{5.5}$$

Ainsi, pour une solution s, il existe n solutions voisines. Le tableau 5.2 montre un exemple d'une solution à 5 attributs et ses 5 solutions voisines.

Cependant, la sélection d'attributs consiste à minimiser le nombre d'attributs, ainsi une bonne solution sera plutôt représentée avec une grande quantité de bits égaux à 0. Ainsi, avec une stratégie de sélection de la première solution voisine améliorante, la probabilité d'avoir une solution voisine améliorante avec une inversion d'un bit de 0 à 1 sera beaucoup plus importante que celle d'inverser un bit de 1 à 0.

Par conséquent, dans le but de donner la même chance d'avoir une inversion de 0 à 1 et de 1 à 0 (augmenter ou diminuer le nombre d'attributs), nous avons divisé le voisinage en deux sous-voisinages. Le voisinage des ajouts ($\mathcal{N}_A$) qui est l'ensemble des solutions où le bit est inversé de 0 à 1. Le voisinage des suppressions ($\mathcal{N}_D$) qui est l'ensemble des solutions où le bit est inversé de 1 à 0. Ainsi, $\mathcal{N}_1^0(s) = \mathcal{N}_A(s) \cup \mathcal{N}_D(s)$ et $\mathcal{N}_A(s) \cap \mathcal{N}_D(s) = \emptyset$. Les voisinages $\mathcal{N}_A$ et $\mathcal{N}_D$ sont définis comme suit :

$$\mathcal{N}_A(s) = \{s' \mid \exists i \in \{1,...,n\} \text{ tel que } a'_i = 1 \text{ et } a_i = 0 \text{ et } \forall j \neq i,\ a'_j = a_j\} \tag{5.6}$$

$$\mathcal{N}_D(s) = \{s' \mid \exists i \in \{1,...,n\} \text{ tel que } a'_i = 0 \text{ et } a_i = 1 \text{ et } \forall j \neq i,\ a'_j = a_j\} \tag{5.7}$$

Ainsi, dans notre exemple précédent, les ensembles seront les suivants : $\mathcal{N}_A(s) = \{s'_1, s'_4\}$ et $\mathcal{N}_D(s) = \{s'_2, s'_3, s'_5\}$

Maintenant que la sélection d'attributs est modélisée pour une recherche locale, nous pouvons adapter l'algorithme du Learning Tabu Search pour ce problème.

5.4 Adaptation du Learning Tabu Search pour la sélection d'attributs

Dans cette section, nous modélisons le Learning Tabu Search pour le problème de sélection d'attributs.

5.4.1 Définition d'une combinaison de caractéristiques et de la valeur de trail associée

La combinaison de deux caractéristiques correspond dans notre problème à la présence simultanée ou non de deux attributs. Ainsi la qualité d'une combinaison d'attributs sera définie par la qualité des solutions où les deux attributs apparaissent ensemble dans les bonnes solutions. Dans le cadre de la sélection d'attributs, cela signifie la qualité de l'information apportée par la sélection conjointe des deux attributs.

La valeur de trail entre deux attributs a_i et a_j, notée $tr(a_i, a_j)$, indique si la combinaison des attributs a_i et a_j est prometteuse, par rapport à l'historique de la recherche. Ainsi, ces valeurs seront stockées dans une matrice symétrique ($tr(a_i, a_j) = tr(a_j, a_i)$) que nous appellerons *Matrice de trail*.

5.4.2 Estimation de la qualité des voisins

Une solution s et chaque solution voisine s' diffèrent d'un attribut a_i (l'opérateur de voisinage étant le *one-flip*). Inspiré des travaux de Zufferey et Schindl [2015], l'estimation d'une solution voisine, notée $Estim(s, a_i)$, est calculée par la quantité d'informations apportée par l'attribut a_i en relation avec les autres attributs présents dans s :

$$Estim(s'_i) = Estim(s, a_i) = \sum_{a_j \in s, a_j \neq a_i} tr(a_i, a_j) * \delta_s(a_j) \begin{cases} \delta_s(a_j) = 1 & \text{si } a_j = 1 \text{ dans s} \\ \delta_s(a_j) = 0 & \text{sinon.} \end{cases} \tag{5.8}$$

Pour une meilleure compréhension de la fonction d'estimation, nous allons supposer une matrice de trail initialisée avec quelques valeurs (cf. Tableau 5.3a). Nous allons considérer l'exemple du tableau 5.2, ainsi soit la solution $s = \{0, 1, 1, 0, 1\}$, telle que $\{a_1, a_4\} = 0$ et $\{a_2, a_3, a_4\} = 0$. Conformément à l'équation 5.8, seuls les attributs présents dans la solution sont pris en compte dans l'estimation d'une solution voisine. D'où l'estimation suivante pour s'_1 :

$$Estim(s'_1) = Estim(s, a_1) = tr(a_1, a_2) + tr(a_1, a_3) + tr(a_1, a_5) = 10 + 7 + 8 = 25 \tag{5.9}$$

Le tableau 5.3b donne l'estimation pour toutes les solutions voisines de s.

5.4.3 Exploration du voisinage

$\mathcal{N}_A$ et $\mathcal{N}_D$ sont les voisinages composés respectivement des *ajouts* et *suppressions* d'attributs. L'ajout le plus *prometteur* dans $\mathcal{N}_A$ est l'attribut a_i dans la solution s (a_1 :

	a_1	a_2	a_3	a_4	a_5
a_1	-	10	7	10	8
a_2		-	6	19	18
a_3			-	20	14
a_4				-	4
a_5					-

(a) Matrice de trail pour 5 attributs.

	a_1	a_2	a_3	a_4	a_5
Solution s	0	1	1	0	1

$Estim(s'_1) = Estim(s, a_1) = 25$
$Estim(s'_2) = Estim(s, a_2) = 24$
$Estim(s'_3) = Estim(s, a_3) = 20$
$Estim(s'_4) = Estim(s, a_4) = 43$
$Estim(s'_5) = Estim(s, a_5) = 32$

(b) Une solution et l'estimation de chaque voisin associé.

TABLEAU 5.3 – Exemple d'une matrice de trail, d'une solution, et de l'estimation de son voisinage.

$0 \rightarrow 1$), tel que la valeur de $Estim(s, a_i)$ soit la plus grande, dans le but que l'ajout de cet attribut apporte le plus d'information possible dans la solution. La suppression la plus *prometteuse* dans $\mathcal{N}_D$ est l'attribut a_i dans la solution s ($a_1 : 1 \rightarrow 0$), tel que la valeur de $Estim(s, a_i)$ soit la plus faible dans le but que la suppression de cet attribut supprime le moins d'information possible dans la solution.

Dans l'exemple du Tableau 5.3, il est important de dissocier les 2 ensembles $\mathcal{N}_A(s) = \{s'_1, s'_4\}$ et $\mathcal{N}_D(s) = \{s'_2, s'_3, s'_5\}$ pour l'exploration du voisinage. Ainsi, nous pouvons pour chaque ensemble trier les voisins dans l'ordre du plus prometteur au moins prometteur. Pour l'ensemble $\mathcal{N}_A(s)$, les voisins sont classés dans l'ordre croissant des valeurs d'estimation. Pour l'ensemble $\mathcal{N}_D(s)$, les voisins sont classés dans l'ordre décroissant des valeurs d'estimation.

D'où les ensembles triés de la façon suivante :

$$\mathcal{N}_A(s) = \{s'_4, s'_1\}$$
$$\mathcal{N}_D(s) = \{s'_3, s'_2, s'_5\}$$

Pour $\mathcal{N}_A(s)$, l'attribut le plus prometteur à l'ajout est a_4 avec une estimation de 43 qui est l'attribut apportant le plus d'information (contre 25 pour a_1). Pour $\mathcal{N}_D(s)$, l'attribut le plus prometteur à la suppression est a_3 avec une estimation de 20 qui est l'attribut portant le moins d'information (contre 24 pour a_2 et 32 pour a_5).

Pour l'exploration du voisinage, seules les Q solutions voisines les plus prometteuses sont évaluées. Ainsi, A_q (respectivement D_q) est le sous-ensemble de solutions voisines non taboues de $\mathcal{N}_A$ (respectivement $\mathcal{N}_D$) composé des q solutions voisines avec les plus grandes (respectivement les plus faibles) estimations, soit les q premières solutions de chaque ensemble trié. Finalement, toutes les solutions de $A_q \cup D_q$ sont évaluées et la meilleure solution est choisie.

5.4.4 Mise à jour de la matrice de trail

Comme mentionné dans la section précédente, la valeur de trail $tr(a_i, a_j)$ est mise à jour à chaque fin de cycle, avec la meilleure solution $\hat{s}$ trouvée durant ce cycle. La mise

à jour se fait par l'équation suivante :

$$tr(a_i, a_j) = \rho * tr(a_i, a_j) + \Delta tr(a_i, a_j) \tag{5.10}$$

Où $\rho \in [0,1]$ est le facteur d'évaporation, et $\Delta tr(a_i, a_j)$ est équivalent à la qualité de $\hat{s}$ si et seulement si, a_i et a_j sont présents ensemble dans la solution $\hat{s}$, sinon $\Delta tr(a_i, a_j) = 0$.

Ainsi, chaque paire d'attributs a_i et a_j subit dans un premier temps l'évaporation d'un facteur ρ dans le but de réduire la quantité d'information apportée par des combinaisons n'apparaissant plus dans la solution $\hat{s}$. Puis, les combinaisons d'attributs qui sont présentes dans la solution $\hat{s}$ ($a_i = a_j = 1$) voient leur valeur de trail augmenter afin de renforcer l'importance de cette combinaison.

5.4.5 Mécanisme de diversification

Le mécanisme de diversification est là pour échapper aux éventuels bassins d'attraction que le Learning Tabu Search pourrait rencontrer. C'est-à-dire, éviter de retomber toujours sur les mêmes optima locaux. Pour cela, lorsque l'algorithme est dans une phase de diversification, la stratégie de sélection des ensembles A_q et D_q pour l'exploration du voisinage est inversée. i.e. A_q (respectivement D_q) est le sous-ensemble de solutions voisines non taboues de $\mathcal{N}_A$ (respectivement $\mathcal{N}_D$) composé des q solutions voisines avec les plus faibles (respectivement les plus grandes) estimations.

Ce mécanisme dépend de deux paramètres t_1 et t_2 : la diversification est activée s'il y a t_1 itérations sans amélioration de s^* (la meilleure solution trouvée depuis le début de la recherche). La diversification est désactivée après t_2 itérations ou si s^* a été améliorée.

5.5 Evaluation des performances

5.5.1 Protocole Expérimental

Nous avons choisi de comparer le Learning Tabu Search à d'autres recherches locales : une descente (*Hill Climbing* - HC) et une recherche tabou (*Tabu Search* - TS) afin de montrer l'apport du mécanisme d'apprentissage.

L'algorithme de descente est une recherche locale classique qui a comme inconvénient majeur de stopper la recherche dès qu'un optimum local est trouvé. Dans le but de donner la même chance à tous les algorithmes, lorsque la descente atteint un optimum local, la descente redémarre à partir d'une solution aléatoire jusqu'à ce que le critère d'arrêt soit atteint, ici un temps maximal d'exécution.

La recherche tabou est une recherche locale qui utilise une mémoire pour éviter de rester bloquée dans un optimum local. La mémoire est utilisée pour stocker les solutions visitées récemment qui sont alors considérées comme *taboues*. À chaque étape, la recherche tabou choisit la meilleure solution voisine non taboue. Ainsi, la recherche

tabou est capable de s'échapper des optima locaux en passant par des solutions voisines qui détériorent le moins possible la qualité. Dans la littérature, la recherche tabou applique la stratégie du meilleur voisin améliorant pour l'exploration du voisinage. Néanmoins, cette stratégie est beaucoup trop coûteuse dans le contexte de la sélection d'attributs. Ainsi, dans notre version de la recherche tabou, nous utilisons la stratégie du premier voisin améliorant.

Un point important de l'algorithme de recherche est le choix du classifieur à utiliser pour évaluer la qualité du sous-ensemble d'attributs. Dans ces travaux de thèse, nous avons utilisé un classifieur de type *Machine à Vecteurs de Support* (Support Vector Machine, SVM, HEARST et collab. [1998]). Ce classifieur permet de découper l'espace de représentation des données en construisant des hyperplans, ce qui permet de séparer les différents labels des observations. Cependant, cette opération peut être très coûteuse en temps dès que le nombre d'observations devient important et/ou que la séparation des données est difficile de par le nombre important d'attributs les caractérisant. Notons que les performances de SVM peuvent être améliorées en définissant les paramètres adéquats. Cependant, ici, nous considérons SVM comme une boîte noire et ne cherchons donc pas à l'optimiser. L'objectif étant de montrer l'apport de la connaissance dans un contexte d'évaluation coûteuse.

Les instances utilisées pour les expérimentations sont divisées en deux parties. La première est l'ensemble d'*apprentissages*, utilisé par l'algorithme de recherche pour trouver le meilleur sous-ensemble d'attributs. Le second est l'ensemble de *validation*, utilisé pour évaluer la capacité à prédire le label sur de nouvelles données avec l'ensemble d'attributs sélectionné.

Pour chaque instance avec leur ensemble d'apprentissage et de validation, nous avons pour tous les algorithmes exécuté les opérations suivantes : (i) l'algorithme de recherche est exécuté sur l'ensemble d'apprentissage, (ii) la meilleure solution trouvée est choisie et sa qualité est évaluée sur l'ensemble de validation, (iii) ces deux étapes sont exécutées 30 fois par instances et par algorithme, (iv) les tests statistiques de Wilcoxon sont réalisés sur les qualités obtenues par l'ensemble d'apprentissage pour comparer les algorithmes, (v) les mêmes tests sont effectués pour l'ensemble de validation.

5.5.2 Description des instances

Les expérimentations ont été réalisées sur six instances de la littérature (voir Chapitre 1). Comme mentionné dans la Section 5.5.1, SVM est un classifieur coûteux en temps de calcul. Pour les instances étudiées, le temps d'exécution nécessaire pour construire et évaluer le modèle de classification peut s'avérer coûteux. Par conséquent, nous avons choisi de séparer les instances en deux groupes en accord avec le temps d'exécution nécessaire à SVM pour s'exécuter sur l'ensemble d'apprentissage.

Le premier groupe est composé des instances pour lesquelles SVM est capable de s'exécuter rapidement (inférieur à 1 seconde), et est donc composé des instances *Schizophrenia*, *Colon* et *Breast*. Le second groupe est composé des instances pour lesquelles SVM est coûteux en temps de calcul, c'est à dire les instances *Arcene*, *DNA* et *Madelon*.

Nom	Attributs	|T|	|V|	Temps SVM (s)	T_{max} (s)
Schizophrenia	410	56	30	0.01	500
Colon	2000	62	32	0.052	120
Breast	24481	78	26	0.734	500
Arcene	10000	100	100	1.123	3000
DNA	180	1400	600	1.172	500
Madelon	500	2000	600	38.089	5000

TABLEAU 5.4 – Description des instances : Attributs est le nombre total d'attributs, |T| est le nombre d'observations de l'ensemble d'apprentissage, |V| est le nombre d'observations de l'ensemble de validation, le temps d'exécution en secondes dont a besoin SVM pour construire et évaluer le modèle sur l'ensemble d'apprentissage (sans sélection d'attributs), et le temps en secondes de recherche alloué à chaque algorithme. Chaque instance est divisée en deux groupes en accord avec le temps d'exécution de SVM.

Le tableau 5.4 montre la taille des données pour chaque ensemble, ainsi que le temps d'exécution de SVM lorsqu'il est exécuté sur l'ensemble d'apprentissage sans sélection d'attributs. Notons que pour l'instance *Madelon*, SVM a besoin de 38 secondes pour évaluer la qualité sur l'ensemble d'apprentissage sans sélection d'attributs.

Le temps maximal autorisé pour chaque instance T_{max} a été déterminé par une étude de convergence de chacun des algorithmes étudiés. Le même temps d'exécution est utilisé pour les trois algorithmes et dépend principalement du temps d'exécution de SVM, puisque c'est ce que nous utilisons pour la fonction d'évaluation. Notons que l'étude de convergence pour l'instance *Arcene* a montré que la convergence était très lente pour cette instance malgré une exécution de SVM équivalente à DNA. C'est pourquoi nous avons laissé plus de temps pour cette instance, l'étude de la convergence a montré que pour chaque algorithme, le temps de convergence était inférieur à 3000 secondes.

5.5.3 Paramètres

Le Learning Tabu Search possède un grand nombre de paramètres sensibles qu'il est important de fixer avant les expérimentations. Le Tableau 5.5 présente les paramètres considérés dans ce chapitre.

Deux paramètres sensibles sont à considérer avec importance.

Le premier est q qui est le nombre de voisins prometteurs estimés pour chacun des ensembles A_q et D_q à évaluer. En effet, si q est petit, le Learning Tabu Search convergera trop rapidement, car l'algorithme aura tendance à toujours choisir les mêmes voisins, il y aura donc un manque de diversité dans le voisinage. À l'inverse, si q est grand, l'algorithme deviendra coûteux en temps de calcul, car trop de solutions seront à évaluer. Notons que q peut-être un paramètre à adapter en fonction de la taille de l'instance, cependant nos expérimentations nous ont montré que $q = 10$ semble être un bon compromis pour ces instances.

Paramètres	Valeur
Taille de la liste taboue	7
Taille des ensembles A_q et D_q (q)	10
Taille du cycle (I)	10
Facteur d'évaporation (ρ)	0.9
Nombre d'itérations sans diversification ($t1$)	10
Nombre d'itérations avec diversification ($t2$)	10
Facteur d'agrégation (α)	0.75

TABLEAU 5.5 – Paramètres du Learning Tabu Search.

Le second paramètre sensible concerne la taille du cycle (I). Si I est petit, alors le mécanisme d'apprentissage fera du sur-apprentissage parce que la recherche n'aura pas le temps de trouver des nouvelles meilleures solutions entre les mises à jour de la matrice de trail, ce qui conduira à une convergence prématurée de l'algorithme de recherche. À l'inverse, si I est trop grand, le mécanisme d'apprentissage mettra trop de temps à détecter des bonnes combinaisons d'attributs, car il n'aura pas assez d'information pertinente, et prendra aussi trop de temps à oublier les mauvaises combinaisons. Nos expérimentations nous ont conduit à fixer I = 10 ce qui semble être ici aussi un bon compromis pour ces instances.

5.5.4 Analyse des performances

L'analyse des performances est divisée en deux parties. La première se concentre sur l'aspect optimisation de la méthode (la capacité de la méthode à trouver un bon sous-ensemble d'attributs, i.e. avec une bonne qualité de classification) et l'évaluation sur l'ensemble d'apprentissage avec la fonction mono-objectif définie à la Section 5.3. La seconde partie se concentre sur l'aspect data-mining (la capacité du modèle de prédiction à labelliser correctement les nouvelles observations) et l'évaluation des résultats obtenus sur l'ensemble de validation.

Analyse des performances de la partie optimisation de l'approche : Le Tableau 5.6 montre une étude comparative entre l'approche proposée (Learning Tabu Search) et les autres approches. Pour chaque instance, l'*accuracy* est calculée avec SVM pour l'ensemble de tous les attributs afin de montrer le bénéfice obtenu par la sélection d'attributs. En détail, nous pouvons observer que le Learning Tabu Search donne généralement une meilleure (ou très proche) *accuracy* et sélectionne toujours le sous-ensemble avec le moins d'attributs que les autres algorithmes. Cela peut s'expliquer par la stratégie utilisée pour l'exploration du voisinage, en effet, le Learning Tabu Search sélectionne pour l'évaluation les q meilleurs ajouts et suppression d'attributs. Par conséquent, les suppressions d'attributs ont autant de chance d'être choisies que les ajouts. À l'inverse, le Hillclimbing et la recherche tabou ont un voisinage aléatoire. Comme le nombre d'attributs est petit, la probabilité de trouver un mouvement qui va supprimer un attribut est faible. Ce qui nécessite l'évaluation de beaucoup de voisins à chaque itération uni-

quement afin de trouver une première solution améliorante. En conclusion, le Learning Tabu Search peut trouver une solution avec une bonne *accuracy* et moins d'attributs que les autres algorithmes.

Instance	Algorithme	Qualité	Accuracy (%)	# S_Features
Schizophrenia (69.64%) LTS > HC > TS	HC	$0.992_{(0)}$	$99.946_{(0.311)}$	$11.788_{(1.933)}$
	TS	$0.968_{(0)}$	$97.132_{(2.043)}$	$16.939_{(5.123)}$
	LTS	$\mathbf{0.995}_{(0)}$	$100_{(0)}$	$8.758_{(0.792)}$
Colon (87.09%) (LTS = HC) > TS	**HC**	$\mathbf{0.998}_{(0)}$	$99.951_{(0.281)}$	$10.909_{(3.440)}$
	TS	$0.982_{(0)}$	$97.752_{(1.845)}$	$10.97_{(2.628)}$
	LTS	$\mathbf{0.996}_{(0)}$	$99.609_{(0.809)}$	$6.788_{(1.745)}$
Breast (67.3%) HC > (LTS = TS)	**HC**	$\mathbf{0.98}_{(0)}$	$97.319_{(2.717)}$	$18.394_{(6.685)}$
	TS	$0.94_{(0)}$	$92.308_{(4.022)}$	$13.121_{(2.804)}$
	LTS	$0.94_{(0)}$	$92.075_{(4.100)}$	$11.879_{(2.858)}$
Arcene (83%) LTS > HC > TS	HC	$0.999_{(0)}$	$99.879_{(0.331)}$	$21.97_{(6.356)}$
	TS	$0.971_{(0)}$	$96.273_{(3.224)}$	$22.273_{(4.252)}$
	LTS	$\mathbf{0.999}_{(0)}$	$100_{(0)}$	$14.97_{(3.147)}$
DNA (89.57%) LTS > (HC = TS)	HC	$0.941_{(0)}$	$95.71_{(0.603)}$	$19.152_{(5.274)}$
	TS	$0.941_{(0)}$	$95.762_{(0.585)}$	$19.485_{(4.651)}$
	LTS	$\mathbf{0.945}_{(0)}$	$95.234_{(0.67)}$	$13.606_{(2.904)}$
Madelon (56,45%) LTS > (HC = TS)	HC	$0.712_{(0)}$	$64.135_{(0.576)}$	$37.273_{(9.593)}$
	TS	$0.714_{(0)}$	$63.885_{(0.673)}$	$31.03_{(6.849)}$
	LTS	$\mathbf{0.731}_{(0)}$	$65.152_{(0.327)}$	$15.606_{(3.201)}$

TABLEAU 5.6 – Pour chaque algorithme, la qualité a été calculée à partir de la précision (*Accuracy*) et du nombre d'attributs sélectionnés (# S_Features) (voir la Section 5.3). La moyenne et l'écart-type (entre parenthèse) de la qualité, la précision et le nombre d'attributs sélectionnés obtenus sur l'ensemble d'apprentissage pour HC, TS et LTS sont donnés. Les qualités sont en gras pour les algorithmes dont les performances sont supérieures aux autres en accord avec les résultats du test de Wilcoxon. Pour chaque instance, la valeur de la précision obtenue par SVM sans sélection d'attributs est donnée entre parenthèse dans la première colonne. La comparaison statistique entre les algorithmes est donnée sous le nom de l'instance.

Le Tableau 5.6 montre aussi que, pour chaque instance, le Learning Tabu Search améliore également les résultats obtenus par la recherche tabou originale. Ces résultats montrent l'amélioration obtenue par le mécanisme d'apprentissage pour la sélection du voisinage.

Dans le but d'analyser le comportement des différents algorithmes, nous avons calculé leur évolution au cours du temps. La Figure 5.1 montre l'évolution de la qualité moyenne de chaque algorithme au cours du temps et donne les boîtes à moustache (après un tiers, deux tiers et à la fin du temps maximal autorisé) de la distribution de la qualité de toutes les exécutions sur l'instance *Madelon*, qui est l'instance la plus difficile à résoudre. Pour cette instance, le Learning Tabu Search a une progression rapide comparé aux deux autres méthodes. En effet, *Madelon* est une instance coûteuse, ainsi grâce

à la fonction d'estimation, le Learning Tabu Search évite un grand nombre d'évaluations. Par conséquent, le Learning Tabu Search trouve de bonnes solutions potentielles plus rapidement que les autres approches. Ces résultats montrent l'intérêt de l'utilisation d'une telle fonction d'estimation.

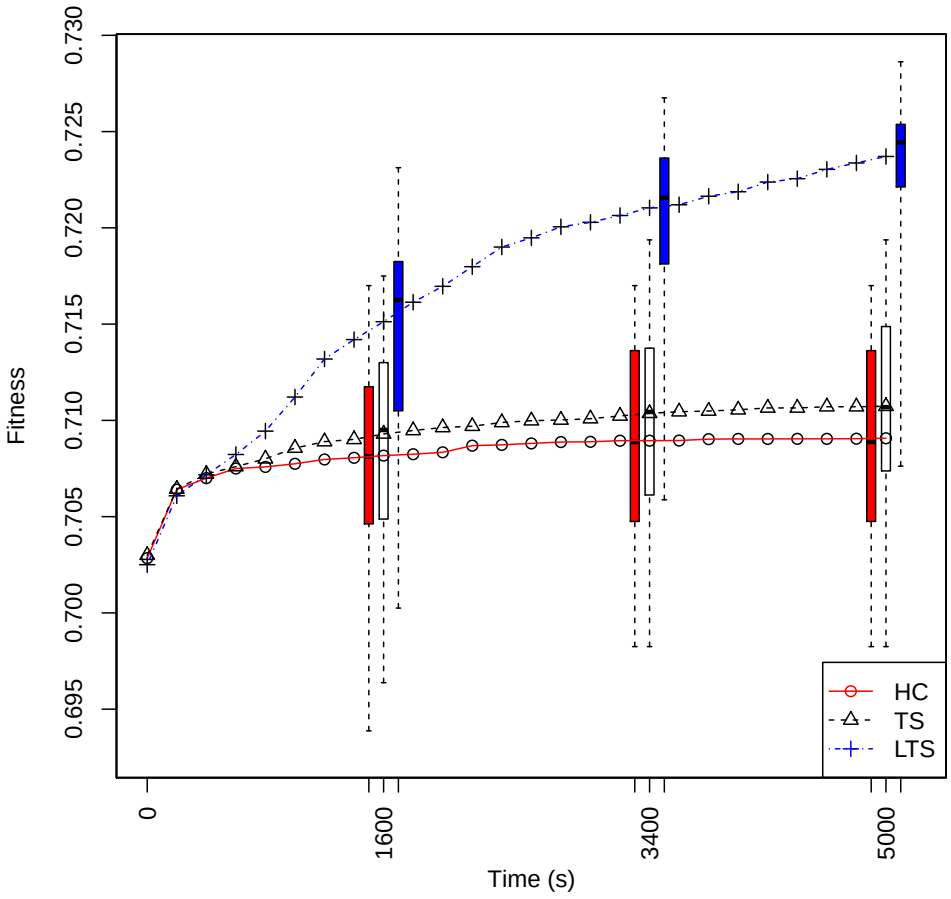


FIGURE 5.1 – Evolution de la qualité (Fitness) des algorithmes sur l'instance *Madelon*

Afin de comprendre le comportement du mécanisme d'apprentissage, nous avons aussi étudié la dynamique de l'opérateur de voisinage au cours du temps afin d'observer les évolutions au niveau des ajouts et suppressions d'attributs. Ainsi la Figure 5.2 montre, pour une exécution, l'évolution des différentes métriques (Qualité, Accuracy et # S_Features) pour l'algorithme du Learning Tabu Search sur l'instance *Madelon*. Nous pouvons observer différentes phases sur cette figure. La première phase, qui commence du début de la recherche à approximativement 100 secondes, consiste à ajouter des attributs pour améliorer la précision, et par conséquent aussi augmenter la qualité. Dans cette phase, l'ajout des attributs se fait tant que l'apport de la précision à la fonction de qualité est plus important que la perte causée par l'ajout d'un attribut. La seconde phase commence lorsque la précision est assez élevée. Dans cette phase, le mécanisme d'ap-

prentissage va choisir un certain nombre d'attributs, grâce aux valeurs dans la matrice de trail, tel que la suppression de ces attributs dans la solution impacte peu la précision. Ainsi, comme l'algorithme perd le moins d'information possible, la baisse de la précision est minimale et l'apport sur la fonction de qualité par le nombre d'attributs est suffisant pour compenser cette perte. Cela permet ainsi de favoriser la recherche des petits ensembles d'attributs. Par conséquent, le Learning Tabu Search permet un bon compromis entre la précision et le nombre d'attributs sélectionnés.

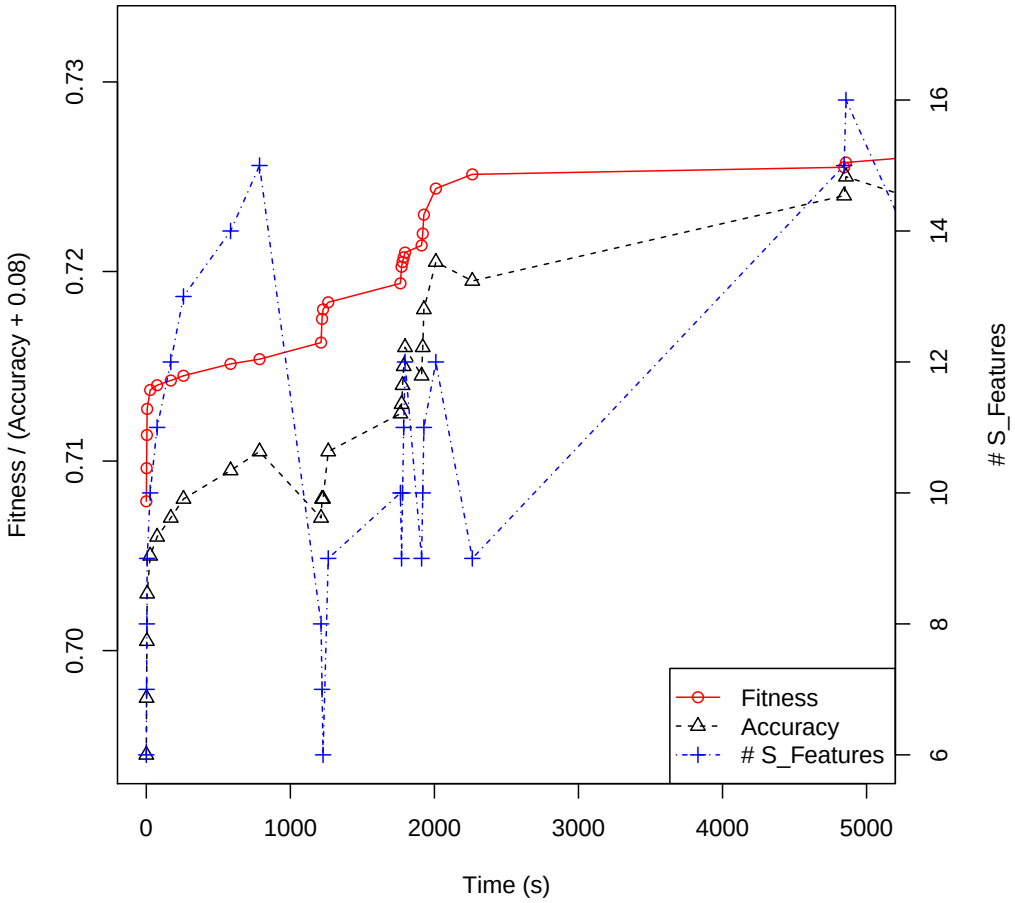


FIGURE 5.2 – Evolution de la qualité (Fitness), l'Accuracy et # S_Features pour LTS sur l'instance *Madelon*. Pour plus de lisibilité, la courbe de l'accuracy a été translatée de +0.08.

Analyse des performance de l'aspect datamining de l'approche : Le Tableau 5.7 montre les résultats des valeurs de précisions sur les ensembles d'apprentissage et de validation pour chaque instance. L'objectif est d'analyser la capacité de prédiction pour une bonne classification sur les nouvelles observations de l'ensemble de Validation qui était jusqu'alors inconnu de l'algorithme de recherche, en utilisant le sous-ensemble d'attributs

sélectionné sur l'ensemble d'apprentissage.

Instance	**Algorithme**	**Accuracy (%) Apprentissage**	**Accuracy (%) Validation**
Schizophrenia	HC	**99.946**$_{(0.311)}$	60.208$_{(9.352)}$
	TS	97.132$_{(2.043)}$	55.457$_{(10.923)}$
	LTS	**100**$_{(0)}$	**61.319**$_{(8.162)}$
		(LTS = HC) > TS	LTS > HC > TS
Colon	HC	**99.951**$_{(0.281)}$	**94.318**$_{(5.150)}$
	TS	97.752$_{(1.845)}$	91.004$_{(6.287)}$
	LTS	99.609$_{(0.809)}$	**94.127**$_{(4.863)}$
		HC > LTS > TS	(LTS = HC) > TS
Breast	HC	**97.319**$_{(2.717)}$	54.079$_{(10.263)}$
	TS	92.308$_{(4.022)}$	50.116$_{(10.333)}$
	LTS	92.075$_{(4.100)}$	52.098$_{(9.904)}$
		HC > (LTS = TS)	LTS = HC = TS
Arcene	HC	**99.879**$_{(0.331)}$	72.18$_{(4.004)}$
	TS	96.273$_{(3.224)}$	71.69$_{(4.707)}$
	LTS	**100**$_{(0)}$	**74.60**$_{(4.629)}$
		(LTS = HC) > TS	LTS > HC > TS
DNA	HC	**95.71**$_{(0.603)}$	93.63$_{(1.582)}$
	TS	**95.762**$_{(0.585)}$	93.25$_{(1.604)}$
	LTS	95.234$_{(0.67)}$	94.09$_{(1.540)}$
		(HC = TS) > LTS	LTS = HC = TS
Madelon	HC	64.135$_{(0.576)}$	57.07$_{(2.200)}$
	TS	63.885$_{(0.673)}$	56.56$_{(2.059)}$
	LTS	**65.152**$_{(0.327)}$	**59.69**$_{(1.219)}$
		LTS > (HC = TS)	LTS > (HC = TS)

TABLEAU 5.7 – La moyenne et l'écart-type (entre parenthèse) de la précision obtenue sur l'ensemble d'apprentissage et de validation pour HC, TS et LTS. Les valeurs de précision sont en gras lorsque l'algorithme est meilleur que les autres en accord avec les résultats du test de Wilcoxon. La comparaison statistique entre les algorithmes est donnée. La double ligne montre la séparation entre les instances peu-coûteuses en temps d'exécution et les autres.

La première observation que nous pouvons faire est sur les performances entre l'ensemble d'apprentissage et l'ensemble de validation. La baisse observée entre l'accuracy de l'ensemble d'apprentissage et de l'ensemble de validation révèle du sur-apprentissage, c'est-à-dire, que la solution construite sur l'ensemble d'apprentissage est spécifique à ces données. Par conséquent, la solution perd en qualité sur la prédiction des nouvelles données. Ceci est particulièrement vrai pour les instances avec peu d'observations, ce qui confirme la difficulté à trouver un bon modèle de classification s'il n'y a pas un nombre suffisant d'observations. L'écart-type obtenu sur l'ensemble de validation sur ces instances est élevé, ce qui montre un manque de robustesse sur les résultats ob-

tenus. A contrario, sur les instances avec un grand nombre d'observations, l'écart-type obtenu sur l'ensemble de validation est raisonnable. Ces solutions sont donc moins sensibles aux données utilisées pour l'apprentissage.

Comme pour l'ensemble d'apprentissage, le Learning Tabu search obtient des résultats meilleurs ou équivalents aux autres approches sur l'ensemble de validation. En particulier, nous pouvons observer une amélioration des résultats obtenus par le Learning Tabu Search comparée à la recherche tabou.

En conclusion, ces expérimentations ont montré une bonne performance du Learning Tabu Search sur l'aspect optimisation et sur l'aspect datamining de l'approche. En particulier, ces expérimentations ont montré la contribution de l'intégration de connaissances dans la méthode d'optimisation de la recherche tabou. Grâce à cette connaissance, l'algorithme a été capable de trouver un meilleur sous-ensemble d'attributs que la recherche tabou de la littérature, alors que les deux algorithmes utilisaient les mêmes composants.

5.6 Conclusion

Ce chapitre considère le problème de Sélection d'attributs pour la classification comme un problème d'optimisation et présente une adaptation de l'algorithme Learning Tabu Search pour le résoudre. L'objectif était d'être capable de traiter conjointement des instances de grandes tailles avec des classifieurs performants. En effet, ces deux éléments font que la fonction objectif devient coûteuse en temps de calcul, un aspect qui rend les méthodes d'optimisation moins performantes au vue du grand nombre d'évaluations nécessaires.

Ainsi, pour résoudre le problème de Sélection d'attributs avec une recherche locale, nous avons proposé dans un premier temps, une modélisation du problème, incluant la définition des opérateurs de voisinages à utiliser. Puis nous avons proposé une adaptation du Learning Tabu Search au problème de sélection d'attributs, qui était un algorithme initialement proposé pour un problème de réseau de chemin de fer. Quelques spécificités pour le problème de sélection d'attributs ont aussi été présentées et expliquées.

Des expérimentations ont été réalisées dans le but d'analyser l'apport de l'intégration de connaissances dans la recherche tabou. Ainsi, le Learning Tabu Search a été principalement comparé à la version originale de l'algorithme de la recherche tabou, en utilisant exactement les mêmes composants, à l'exception du mécanisme d'intégration de connaissances. Pour cela, les instances de la littérature ont été utilisées. Certaines d'entre elles avaient un faible temps de calcul pour l'évaluation, d'autres étaient au contraire très coûteuses. La principale conclusion que l'on peut faire à propos de l'aspect optimisation de la méthode (capacité à trouver une solution de bonne qualité), est que le Learning Tabu Search a obtenu des meilleurs résultats que la recherche tabou classique, plus particulièrement sur les évaluations coûteuses. Cela est possible grâce à l'utilisation d'une fonction d'estimation qui permet d'éviter de nombreuses évaluations, et permet à l'algorithme de progresser bien plus rapidement. Pour ce qui est de l'aspect

datamining (la capacité à trouver des solutions qui permettent une bonne classification sur des nouvelles données), la même conclusion est ainsi faite : Le Learning Tabu Search a obtenu de meilleurs résultats que la méthode classique de la recherche tabou. Cela s'explique par le faible nombre d'attributs sélectionnés par le Learning Tabu Search comparé aux autres méthodes. Ainsi, ces expérimentations montrent la contribution de l'intégration de connaissances.

Conclusion

Les travaux présentés dans ce mémoire de thèse traitent de l'intégration de connaissances dans les méthodes d'optimisation. Dans cette dernière partie, nous allons résumer les contributions de ces travaux de recherche, puis nous proposerons différentes perspectives d'améliorations qui pourraient suivre ce travail.

Synthèse des contributions

Taxonomie des méthodes d'intégration de connaissances : Dans ce mémoire nous avons proposé une revue complète de la littérature sur l'intégration de connaissances. Nous avons ensuite proposé de définir l'intégration de connaissances comme une composition de trois parties : Extraction, Mémorisation et Exploitation. Nous avons alors donné les différents éléments possibles pour chacune de ces parties. Pour finir, nous avons proposé une taxonomie pour classer les différentes méthodes de la littérature.

Heuristiques constructives utilisant de la connaissance : Nous avons ensuite proposé deux heuristiques constructives pour résoudre le problème d'ordonnancement du Flowshop sans temps d'attente : Iterated Best Insertion (IBI) et Incremental GAP (IncrGAP). Chacune de ces heuristiques ont présenté de nombreux avantages.

La première, IBI, est une heuristique constructive pour minimiser le Makespan, conçue grâce à l'analyse du génotype des solutions optimales sur des petites instances dont les solutions peuvent être générées exhaustivement. Grâce à cette énumération, nous avons pu étudier l'évolution de la solution optimale en ajoutant à chaque itération une nouvelle tâche. De cette façon, nous avons pu analyser le comportement que doit avoir une heuristique constructive pour conserver une solution au plus proche de la solution optimale. Nos observations nous ont conduits à comprendre qu'une simple insertion d'une tâche à la manière de l'heuristique NEH, très utilisée dans les problèmes d'ordonnancement n'est pas suffisante. En effet, nous avons observé qu'en plus de l'insertion de la nouvelle tâche, d'autres tâches devaient être réinsérées pour conserver une solution optimale. C'est ainsi que nous avons conçu IBI, sur la base de l'heuristique NEH, en ajoutant une descente améliorante à chaque insertion d'une nouvelle tâche.

Les expérimentations ont confirmé les bons résultats de cette nouvelle heuristique qui fournit des résultats bien supérieurs aux heuristiques comparées. L'unique inconvénient de cette méthode est le temps de calcul bien supérieur aux autres heuristiques. Cependant, nous avons montré l'intérêt d'utiliser IBI comme une initialisation d'une métaheuristique. Les expérimentations ont montré que l'utilisation d'IBI en tant que

méthode d'initialisation d'une recherche tabou donne de bien meilleurs résultats que les autres heuristiques habituellement utilisées.

La seconde heuristique proposée, IncrGAP, est une heuristique constructive pour minimiser le Flowtime conçu grâce à l'analyse du phénotype des meilleures solutions connues des petites instances de Taillard. Pour cela, l'analyse de la mesure du GAP a été étudiée pour comprendre comment une solution optimale est construite. Nous avons ainsi observé que les tâches sont généralement suivies par la tâche qui minimise le GAP (parmi les tâches non encore ordonnancées). Ce qui nous a permis d'élaborer une heuristique constructive très rapide et de qualité correcte. Cependant bien que plus rapide que les méthodes de la littérature, la qualité était généralement inférieure aux meilleures heuristiques de la littérature. Nous avons alors proposé une variante d'IncrGAP, IIGAP, qui permet d'explorer dans certains cas, deux possibilités au lieu d'une. Ainsi la méthode visite plus de solutions, où le nombre de solutions potentielles que va visiter IIGAP dépend du paramètre de la tolérance, qui permet de définir quand IIGAP doit explorer deux possibilités au lieu d'une seule en fonction de la proximité des GAP entre les tâches. Les résultats des expérimentations nous ont montré que pour un temps de calcul équivalent, IIGAP était meilleur que les méthodes de la littérature.

Plus généralement, nous avons proposé dans cette partie, une méthodologie pour réaliser des heuristiques constructives. Nous basons des hypothèses sur les observations faites sur des petites instances telles que toutes les solutions soient énumérables et/ou que la solution optimale soit connue. Puis nous avons testé la validité de ces observations sur des plus grandes instances.

Intégration de connaissances pour la modification du voisinage : Dans la troisième partie nous avons proposé une métaheuristique pour le problème du Flowshop sans temps d'attente pour minimiser le Makespan. Cette approche est basée sur une métaheuristique de la littérature : l'Iterated Greedy (IG) que nous avons améliorée grâce à l'intégration de connaissances. Pour cela, nous avons étudié les meilleures solutions d'une instance de petite taille qui nous permettait d'énumérer toutes les solutions possibles. Nous avons ainsi pu observer que certaines tâches étaient toujours ensemble dans toute les solutions. Nous avons ainsi défini ces groupements souvent présents ensemble comme des "super-jobs". Pour obtenir ces super-jobs, il était alors nécessaire de connaître plusieurs solutions différentes de bonne qualité. Nous devions donc générer un "pool" de solutions avec IG pour identifier les super-jobs. Une fois fait, nous relancions IG, cette fois en utilisant les super-jobs identifiés à l'étape précédente.

Les expérimentations nous ont permis de valider l'approche, dans un premier temps sur la réduction de la taille du problème initiale (jusqu'à 50% de tâches en moins), et dans un second temps sur la qualité des solutions obtenues qui était bien meilleure que celle obtenue par la meilleure méthode de la littérature au moment de ces travaux de recherche. Ensuite, comme les bonnes solutions d'IG ont permis d'obtenir de nouvelles meilleures solutions, nous avons proposé d'utiliser les nouvelles solutions trouvées pour trouver des super-jobs plus pertinents. Ainsi nous avons itéré notre approche. Bien que relativement coûteux en temps de calcul, notre méthode nous a permis de trouver de

nouvelles meilleures solutions pour de nombreuses instances de la littérature. Depuis, des travaux plus récents ont proposé une approche exacte, qui a permis de trouver les solutions optimales, et donc de dépasser les meilleures solutions que nous avions obtenues. Cependant nous avons pu constater que notre approche obtenait des solutions très proche des solutions optimales (moins de 0.08% en moyenne pour les instances étudiées). En outre, ce travail a permis de montrer l'apport de l'intégration de connaissances dans une méthode d'optimisation, en l'occurrence ici, pour l'Iterated Greedy.

Intégration de connaissances pour la réduction du voisinage : La dernière contribution de ce mémoire de thèse est la réalisation d'une méta-heuristique avec intégration de connaissances pour la réduction du voisinage. Cet algorithme, le Learning Tabu Search, a été proposé dans le cadre de la résolution du problème de sélection d'attributs pour un contexte de classification supervisée. Ce problème consiste à trouver les attributs les plus pertinents pour qu'un algorithme de classification puisse construire un modèle de prédiction efficace. Pour ce type de problème, l'évaluation d'une seule solution est coûteuse en temps de calcul, et explorer l'ensemble du voisinage d'une solution devient alors difficile. Pour pallier ce problème, nous sommes partis du constat que si des attributs sont souvent présents ensemble, dans les solutions de bonne qualité, alors elles seront présentes dans les meilleures solutions. De ce fait, chaque combinaison va être associée à un score qui sera utilisé pour estimer si une solution voisine est intéressante ou non. Ainsi, au lieu d'évaluer toutes les solutions voisines avec une fonction coûteuse, nous les estimons. Grâce aux estimations, l'algorithme évalue uniquement les solutions les plus intéressantes.

Les expérimentations ont porté sur deux aspects, l'aspect optimisation combinatoire et l'aspect datamining. Pour le premier aspect, notre algorithme a donné des résultats très satisfaisants comparés aux autres méta-heuristiques étudiées, en particulier sur les instances les plus difficiles avec beaucoup d'attributs. Pour le second aspect, cela consistait à évaluer la qualité du modèle de prédiction sur des nouvelles données. Ainsi les instances étaient divisées en deux ensembles, un ensemble d'apprentissage et un ensemble de validation. Les algorithmes étaient exécutés sur l'ensemble d'apprentissages, puis le modèle de prédiction obtenu était évalué sur l'ensemble de validation. Les résultats étaient statistiquement meilleurs que les autres approches, ce qui a permis de montrer la robustesse au sur-apprentissage de notre algorithme. Pour conclure, l'ensemble des résultats a montré l'intérêt de l'utilisation de l'intégration de connaissances pour les deux aspects.

Perspectives

Dans cette partie, nous allons présenter les améliorations qui peuvent être apportées à nos différents travaux.

Perspectives de recherche concernant IBI : Nous avons développé une heuristique basée sur une simple modification d'une heuristique constructive existante (NEH) pour

améliorer nos résultats. Cette modification a été proposée à partir des propriétés observées qui ont permis l'utilisation d'une descente améliorante à chaque étape. Ces propriétés sont : (i) seulement quelques étapes sont requises pour atteindre la solution optimale sur des petits problèmes et (ii) la descente n'est pas coûteuse en temps car le makespan d'une solution voisine pouvant être calculé en O(1) grâce aux propriétés du No-Wait Flowshop. Nous pouvons alors nous demander si de telles propriétés peuvent se retrouver dans d'autres variantes des problèmes d'ordonnancement ou plus généralement dans d'autres problèmes qui présenteraient des propriétés équivalentes. De même, nous avons intégré une descente dans une heuristique simple (NEH) mais qui n'est pas la meilleure pour ce problème. Il peut être ainsi intéressant d'étudier l'impact d'une descente à chaque itération pour les différentes heuristiques constructives pour le No-Wait Flowshop.

Perspectives de recherche concernant IncrGAP : L'heuristique que nous avons proposée est spécifique au No-Wait Flowshop et ne peut pas être transposée pour d'autres problèmes. Cependant des pistes d'améliorations sont possibles. La première concerne un problème de notre approche qui fait qu'elle exécute plusieurs fois les mêmes calculs, en proposant une structure de données efficaces, IncrGAP pourrait s'exécuter plus rapidement. Ce gain en temps pourrait permettre ainsi de fixer une tolérance plus importante ce qui nous permettrait d'obtenir des résultats bien meilleurs. De même, ce gain de temps pourrait permettre peut-être d'explorer aussi trois possibilités de tâches successives, ce qui peut être nécessaire dans de très rares cas.

Perspectives de recherche concernant les Super-jobs : L'utilisation des super-jobs a permis d'obtenir de meilleures solutions que l'algorithme TMIIG. Cependant, la génération du pool de solutions est très coûteuse en temps de calcul. Une possibilité pour pallier ce problème est d'identifier les super-jobs au cours de la recherche grâce à une méthode en ligne. Cela aurait pour avantage d'améliorer un petit peu notre pool de solutions à chaque itération, au lieu d'utiliser un pool identique pour 10 itérations. Une autre perspective est d'adapter cette approche à d'autres problèmes de permutation et ainsi, trouver comment extraire la connaissance des meilleures solutions. Plus généralement, rechercher des schémas de structure commune dans les solutions peut-être une piste intéressante à explorer.

Perspectives de recherche concernant le Learning Tabu Search : En ce qui concerne l'algorithme que nous avons proposé pour le problème de sélection d'attributs dans un contexte de classification supervisée, des expérimentations supplémentaires peuvent être nécessaires. Dans ce chapitre nous avons concentré nos recherches sur l'apport de la connaissance dans un Tabu Search que nous avons pu valider. Cependant aucune réelle comparaison aux méthodes existantes de la littérature sur ce type de problème n'a été effectuée, il peut être intéressant de positionner notre méthode par rapport aux méthodes performantes de la littérature. Une autre perspective est d'adapter ce mécanisme pour d'autres types de problèmes. Pour cela une redéfinition de la matrice d'apprentissage sera nécessaire, ainsi que la définition d'une combinaison de caracté-

ristiques. En projet de recherche à court terme, il pourrait être intéressant de combiner l'idée des super-jobs, avec le mécanisme en ligne que nous proposons dans le Learning Tabu Search. L'idée des deux approches étant étroitement liée.

Ainsi, les travaux de cette thèse ont permis de mieux comprendre les mécanismes d'intégration de connaissances au sein des méthodes d'optimisation et en particulier des heuristiques. Certains cas d'études ont montré la pertinence de ces approches. De nombreuses perspectives intéressantes quant à la généralisation des approches proposées ou à la définition de nouveaux paradigmes d'intégration de connaissances restent ouvertes.

Bibliographie

ALDOWAISAN, T. et A. ALLAHVERDI. 1998, «Total flowtime in no-wait flowshops with separated setup times», *Computers & Operations Research*, vol. 25, nº 9, doi : 10.1016/s0305-0548(98)00002-1, p. 757–765. URL http://dx.doi.org/10.1016/s0305-0548(98)00002-1. 22

ALDOWAISAN, T. et A. ALLAHVERDI. 2004, «New heuristics for m-machine no-wait flowshop to minimize total completion time», *Omega*, vol. 32, nº 5, doi :10.1016/j.omega.2004.01.004, p. 345–352. 52

ARAÚJO, D. C. et M. S. NAGANO. 2010, «An effective heuristic for the no-wait flowshop with sequence-dependent setup times problem», dans *Advances in Artificial Intelligence*, Springer Berlin Heidelberg, p. 187–196, doi :10.1007/978-3-642-16761-4_17. 36, 46

ARBELAEZ, A. et Y. HAMADI. 2011, «Improving parallel local search for SAT», dans *Lecture Notes in Computer Science*, Springer Science + Business Media, p. 46–60, doi :10.1007/978-3-642-25566-3_4. URL http://dx.doi.org/10.1007/978-3-642-25566-3_4. 41, 46

ARBELAEZ, A. et B. O'SULLIVAN. 2016, «Learning a stopping criteria for local search», dans *LION 10, Napoli, Italy, 29 may - 1 June, 2016. Revised Selected Papers*. 34, 36, 41, 46

ARIN, A. et G. RABADI. 2017, «Integrating estimation of distribution algorithms versus q-learning into meta-RaPS for solving the 0-1 multidimensional knapsack problem», *Computers & Industrial Engineering*, vol. 112, doi :10.1016/j.cie.2016.10.022, p. 706–720. 39, 46

AUER, P., N. CESA-BIANCHI et P. FISCHER. 2002, «Finite-time analysis of the multiarmed bandit problem», *Machine Learning*, vol. 47, nº 2/3, doi :10.1023/a:1013689704352, p. 235–256. URL http://dx.doi.org/10.1023/a:1013689704352. 40

AZIZ, M. A. E. et A. E. HASSANIEN. 2016, «Modified cuckoo search algorithm with rough sets for feature selection», *Neural Computing and Applications*, vol. 29, nº 4, doi :10.1007/s00521-016-2473-7, p. 925–934. 26, 27

BASSEUR, M. et A. GOËFFON. 2014, «On the efficiency of worst improvement for climbing NK-landscapes», dans *Proceedings of the 2014 conference on Genetic and evolutio-*

nary computation - GECCO '14, Association for Computing Machinery (ACM), doi :10.1145/2576768.2598268. URL http://dx.doi.org/10.1145/2576768.2598268. 12

BECERRA, R. L. et C. A. C. COELLO. 2005, «A cultural algorithm for solving the job shop scheduling problem», dans *Knowledge Incorporation in Evolutionary Computation*, Springer Science + Business Media, ISBN 978-3-642-06174-5, p. 37–55, doi :10.1007/978-3-540-44511-1_3. URL http://dx.doi.org/10.1007/978-3-540-44511-1_3. 37, 46

BENLIC, U., M. G. EPITROPAKIS et E. K. BURKE. 2017, «A hybrid breakout local search and reinforcement learning approach to the vertex separator problem», *European Journal of Operational Research*, vol. 261, nº 3, doi :10.1016/j.ejor.2017.01.023, p. 803–818. 41, 46

BERTOLISSI, E. 2000, «Heuristic algorithm for scheduling in the no-wait flow-shop», *Journal of Materials Processing Technology*, vol. 107, nº 1-3, doi :10.1016/s0924-0136(00)00720-2, p. 459–465. URL http://dx.doi.org/10.1016/s0924-0136(00)00720-2. 17, 19

BEWOOR, L., V. C. PRAKASH et S. SAPKAL. 2017, «Evolutionary hybrid particle swarm optimization algorithm for solving NP-hard no-wait flow shop scheduling problems», *Algorithms*, vol. 10, nº 4, doi :10.3390/a10040121, p. 121. xiii, 19, 21

BIANCO, L., P. DELL'OLMO et S. GIORDANI. 1999, «Flow shop no-wait scheduling with sequence dependent setup times and release dates», *INFOR : Information Systems and Operational Research*, vol. 37, nº 1, doi :10.1080/03155986.1999.11732365, p. 3–19. URL http://dx.doi.org/10.1080/03155986.1999.11732365. 19, 50, 52

BONNEY, M. C. et S. W. GUNDRY. 1976, «Solutions to the constrained flowshop sequencing problem», *Journal of the Operational Research Society*, vol. 27, nº 4, doi :10.1057/jors.1976.176, p. 869–883. 19

CAI, D., C. ZHANG et X. HE. 2010, «Unsupervised feature selection for multi-cluster data», dans *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '10*, ACM Press, doi :10.1145/1835804.1835848. 26, 27

CERVANTE, L., B. XUE, M. ZHANG et L. SHANG. 2012, «Binary particle swarm optimisation for feature selection : A filter based approach», dans *2012 IEEE Congress on Evolutionary Computation*, IEEE, doi :10.1109/cec.2012.6256452. 26, 27

CORNE, D., C. DHAENENS et L. JOURDAN. 2012, «Synergies between operations research and data mining : The emerging use of multi-objective approaches», *European Journal of Operational Research*, vol. 221, nº 3, doi :10.1016/j.ejor.2012.03.039, p. 469–479. 98

DAVENDRA, D., I. ZELINKA, M. BIALIC-DAVENDRA, R. SENKERIK et R. JASEK. 2013, «Discrete self-organising migrating algorithm for flow-shop scheduling with no-wait makespan», *Mathematical and Computer Modelling*, vol. 57, nº 1-2, doi :10.1016/j.mcm.2011.05.029, p. 100–110. 22

DEVORE, R. A. et V. N. TEMLYAKOV. 1996, «Some remarks on greedy algorithms», *Advances in Computational Mathematics*, vol. 5, nº 1, doi :10.1007/bf02124742, p. 173–187. URL http://dx.doi.org/10.1007/BF02124742. 9

DING, J.-Y., S. SONG, J. N. GUPTA, R. ZHANG, R. CHIONG et C. WU. 2015, «An improved iterated greedy algorithm with a tabu-based reconstruction strategy for the no-wait flowshop scheduling problem», *Applied Soft Computing*, vol. 30, doi :10.1016/j.asoc.2015.02.006, p. 604–613. 22, 74, 79, 82, 83, 93

DORIGO, M. et L. GAMBARDELLA. 1997, «Ant colony system : a cooperative learning approach to the traveling salesman problem», *IEEE Transactions on Evolutionary Computation*, vol. 1, nº 1, doi :10.1109/4235.585892, p. 53–66. URL http://dx.doi.org/10.1109/4235.585892. 11, 46

DORIGO, M., V. MANIEZZO et A. COLORNI. 1996, «Ant system : optimization by a colony of cooperating agents», *IEEE Transactions on Systems, Man and Cybernetics, Part B (Cybernetics)*, vol. 26, nº 1, doi :10.1109/3477.484436, p. 29–41. 37, 100

DOS SANTOS, J. P. Q., J. D. DE MELO, A. D. D. NETO et D. ALOISE. 2014, «Reactive search strategies using reinforcement learning, local search algorithms and variable neighborhood search», *Expert Systems with Applications*, vol. 41, nº 10, doi :10.1016/j.eswa.2014.01.040, p. 4939–4949. URL http://dx.doi.org/10.1016/j.eswa.2014.01.040. 39, 46

DUVAL, B., J.-K. HAO et J. C. H. HERNANDEZ. 2009, «A memetic algorithm for gene selection and molecular classification of cancer», dans *Proceedings of the 11th Annual conference on Genetic and evolutionary computation - GECCO '09*, ACM Press, doi : 10.1145/1569901.1569930. 26, 27

EMMANOUILIDIS, C., A. HUNTER et J. MACINTYRE. 2000, «A multiobjective evolutionary setting for feature selection and a commonality-based crossover operator», dans *Proceedings of the 2000 Congress on Evolutionary Computation. CEC00 (Cat. No.00TH8512)*, IEEE, doi :10.1109/cec.2000.870311. 26, 27

FAIGLE, U. et W. KERN. 1992, «Some convergence results for probabilistic tabu search», *ORSA Journal on Computing*, vol. 4, nº 1, doi :10.1287/ijoc.4.1.32, p. 32–37. URL http://dx.doi.org/10.1287/ijoc.4.1.32. 9

FERBER, J. 1999, *Multi-agent systems : an introduction to distributed artificial intelligence*, vol. 1, Addison-Wesley Reading. 11

FRAMINAN, J. M., J. N. D. GUPTA et R. LEISTEN. 2004, «A review and classification of heuristics for permutation flow-shop scheduling with makespan objective», *Journal*

of the Operational Research Society, vol. 55, n° 12, doi :10.1057/palgrave.jors.2601784, p. 1243–1255. 49

GANGADHARAN, R. et C. RAJENDRAN. 1993, «Heuristic algorithms for scheduling in the no-wait flowshop», *International Journal of Production Economics*, vol. 32, n° 3, doi : 10.1016/0925-5273(93)90042-j, p. 285–290. 19, 50

GHEYAS, I. A. et L. S. SMITH. 2010, «Feature subset selection in large dimensionality domains», *Pattern Recognition*, vol. 43, n° 1, doi :10.1016/j.patcog.2009.06.009, p. 5–13. 26, 27

GILMORE, P. C. et R. E. GOMORY. 1964, «Sequencing a one state-variable machine : A solvable case of the traveling salesman problem», *Operations Research*, vol. 12, n° 5, doi : 10.1287/opre.12.5.655, p. 655–679. URL http://dx.doi.org/10.1287/opre.12.5.655. 16, 19

GLOVER, F. et M. LAGUNA. 1998, «Tabu search», dans *Handbook of Combinatorial Optimization*, Springer US, p. 2093–2229, doi :10.1007/978-1-4613-0303-9_33. 14

GONAZELEZ, B. 1995, «A hybrid genetic algorithm approach for the "no-wait" flowshop scheduling problem», dans *1st International Conference on Genetic Algorithms in Engineering Systems : Innovations and Applications (GALESIA)*, IEE, doi :10.1049/cp:19951025. 22

GRABOWSKI, J. et J. PEMPERA. 2005, «Some local search algorithms for no-wait flow-shop problem with makespan criterion», *Computers & Operations Research*, vol. 32, n° 8, doi :10.1016/j.cor.2004.02.009, p. 2197–2212. 22

GRABOWSKI, J. et J. PEMPERA. 2007, «The permutation flow shop problem with blocking. a tabu search approach», *Omega*, vol. 35, n° 3, doi :10.1016/j.omega.2005.07.004, p. 302–311. URL http://dx.doi.org/10.1016/j.omega.2005.07.004. 61

GRABOWSKI, J. et M. WODECKI. 2004, «A very fast tabu search algorithm for the permutation flow shop problem with makespan criterion», *Computers & Operations Research*, vol. 31, n° 11, doi :10.1016/s0305-0548(03)00145-x, p. 1891–1909. URL http://dx.doi.org/10.1016/S0305-0548(03)00145-X. 61

GUERRA-SALCEDO, C. et D. WHITLEY. 1999, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '99)*, Morgan Kaufmann, ISBN 1-55860-611-4. URL https://www.amazon.com/Proceedings-Genetic-Evolutionary-Computation-Conference/dp/1558606114?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=1558606114. 25

GUYON, I., S. GUNN, A. B. HUR et G. DROR. 2004, «Result analysis of the nips 2003 feature selection challenge», dans *Proceedings of the 17th International Conference on Neural Information Processing Systems*, NIPS'04, MIT Press, Cambridge, MA, USA, p. 545–552. URL http://dl.acm.org/citation.cfm?id=2976040.2976109. 25

GUYON, I., M. NIKRAVESH, S. GUNN et L. A. ZADEH, éd.. 2006, *Feature Extraction*, Springer Berlin Heidelberg, doi :10.1007/978-3-540-35488-8. 25

HAMDANI, T. M., J.-M. WON, A. M. ALIMI et F. KARRAY. 2007, «Multi-objective feature selection with NSGA II», dans *Adaptive and Natural Computing Algorithms*, Springer Berlin Heidelberg, p. 240–247, doi :10.1007/978-3-540-71618-1_27. 26, 27

HEARST, M., S. DUMAIS, E. OSUNA, J. PLATT et B. SCHOLKOPF. 1998, «Support vector machines», *IEEE Intelligent Systems and their Applications*, vol. 13, nº 4, doi :10.1109/5254.708428, p. 18–28. 108

HOLLAND, J. H. 1992, *Adaptation in Natural and Artificial Systems*, MIT University Press Group Ltd, ISBN 0262581116. URL http://www.ebook.de/de/product/2864942/john_h_holland_adaptation_in_natural_and_artificial_systems.html. 10

HORN, J., N. NAFPLIOTIS et D. GOLDBERG. 1994, «A niched pareto genetic algorithm for multiobjective optimization», dans *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*, IEEE, doi :10.1109/icec.1994.350037. 34, 46

INBARANI, H. H., M. BAGYAMATHI et A. T. AZAR. 2015, «A novel hybrid feature selection method based on rough set and improved harmony search», *Neural Computing and Applications*, vol. 26, nº 8, doi :10.1007/s00521-015-1840-0, p. 1859–1880. 26, 27

JARBOUI, B., M. EDDALY et P. SIARRY. 2010, «A hybrid genetic algorithm for solving no-wait flowshop scheduling problems», *The International Journal of Advanced Manufacturing Technology*, vol. 54, nº 9-12, doi :10.1007/s00170-010-3009-4, p. 1129–1143. 22

JOURDAN, L., D. CORNE, D. SAVIC et G. WALTERS. 2005, «Preliminary investigation of the 'learnable evolution model' for faster/better multiobjective water systems design», dans *Lecture Notes in Computer Science*, Springer Science + Business Media, p. 841–855, doi :10.1007/978-3-540-31880-4_58. URL http://dx.doi.org/10.1007/978-3-540-31880-4_58. 35, 36, 42

KANDA, J., A. DE CARVALHO, E. HRUSCHKA, C. SOARES et P. BRAZDIL. 2016, «Meta-learning to select the best meta-heuristic for the traveling salesman problem : A comparison of meta-features», *Neurocomputing*, vol. 205, doi :10.1016/j.neucom.2016.04.027, p. 393–406. 33, 36, 46

KENNEDY, J. et R. EBERHART. 1995, «Particle swarm optimization», dans *Proceedings of ICNN'95 - International Conference on Neural Networks*, IEEE, doi :10.1109/icnn.1995.488968. 38

KING, J. R. et A. S. SPACHIS. 1980, «Heuristics for flow-shop scheduling», *International Journal of Production Research*, vol. 18, nº 3, doi :10.1080/00207548008919673, p. 345–357. 19

KNOWLES, J. et H. NAKAYAMA. 2008, «Meta-modeling in multiobjective optimization», dans *Multiobjective Optimization*, Springer Science + Business Media, p. 245–284, doi :10.1007/978-3-540-88908-3_10. URL http://dx.doi.org/10.1007/978-3-540-88908-3_10. 35, 36, 46

KOHAVI, R. et G. H. JOHN. 1997, «Wrappers for feature subset selection», *Artificial Intelligence*, vol. 97, nº 1-2, doi :10.1016/s0004-3702(97)00043-x, p. 273–324. 24

KOTTHOFF, L. 2016, «Algorithm selection for combinatorial search problems : A survey», dans *Data Mining and Constraint Programming*, Springer International Publishing, p. 149–190, doi :10.1007/978-3-319-50137-6_7. 33

LAHA, D. et U. K. CHAKRABORTY. 2008, «A constructive heuristic for minimizing makespan in no-wait flow shop scheduling», *The International Journal of Advanced Manufacturing Technology*, vol. 41, nº 1-2, doi :10.1007/s00170-008-1454-0, p. 97–109. 22, 50

LAL, T. N., O. CHAPELLE, J. WESTON et A. ELISSEEFF. 2004, «Embedded methods», dans *Feature Extraction*, Springer Berlin Heidelberg, p. 137–165, doi :10.1007/978-3-540-35488-8_6. 25

LALLA-RUIZ, E. et S. VOSS. 2015, «Minimizing total tardiness on identical parallel machines using VNS with learning memory», dans *Lecture Notes in Computer Science*, Springer Science + Business Media, p. 119–124, doi :10.1007/978-3-319-19084-6_10. URL http://dx.doi.org/10.1007/978-3-319-19084-6_10. 35, 36, 37, 41, 46

LESSMANN, S., M. CASERTA et I. M. ARANGO. 2011, «Tuning metaheuristics : A data mining based approach for particle swarm optimization», *Expert Systems with Applications*, vol. 38, nº 10, doi :10.1016/j.eswa.2011.04.075, p. 12 826–12 838. 34

LI, K., A. FIALHO, S. KWONG et Q. ZHANG. 2014, «Adaptive operator selection with bandits for a multiobjective evolutionary algorithm based on decomposition», *IEEE Transactions on Evolutionary Computation*, vol. 18, nº 1, doi :10.1109/tevc.2013.2239648, p. 114–130. URL http://dx.doi.org/10.1109/tevc.2013.2239648. 40, 42, 46

LI, X., Q. WANG et C. WU. 2008, «Heuristic for no-wait flow shops with makespan minimization», *International Journal of Production Research*, vol. 46, nº 9, doi :10.1080/00207540701737997, p. 2519–2530. 19

LIN, S.-W. et K.-C. YING. 2016, «Optimization of makespan for no-wait flowshop scheduling problems using efficient matheuristics», *Omega*, vol. 64, doi :10.1016/j.omega.2015.12.002, p. 115–125. xiii, xiv, 19, 20, 22, 74, 95, 96

LIU, B., L. WANG et Y.-H. JIN. 2006, «An effective hybrid particle swarm optimization for no-wait flow shop scheduling», *The International Journal of Advanced Manufacturing Technology*, vol. 31, nº 9-10, doi :10.1007/s00170-005-0277-5, p. 1001–1011. 22

MARMION, M.-E., C. DHAENENS, L. JOURDAN, A. LIEFOOGHE et S. VEREL. 2011, «The road to VEGAS», dans *Proceedings of the 13th annual conference on Genetic and evolutionary computation - GECCO '11*, Association for Computing Machinery (ACM), doi :10.1145/2001576.2001842. URL http://dx.doi.org/10.1145/2001576.2001842. 34, 36, 40, 42, 46

MARTÍNEZ, Y., A. NOWÉ, J. SUÁREZ et R. BELLO. 2011, «A reinforcement learning approach for the flexible job shop scheduling problem», dans *Lecture Notes in Computer Science*, Springer Science + Business Media, p. 253–262, doi :10.1007/978-3-642-25566-3_19. URL http://dx.doi.org/10.1007/978-3-642-25566-3_19. 39, 46

MICHALSKI, R. S. 2000, «Learnable evolution model : Evolutionary processes guided by machine learning», *Machine Learning*, vol. 38, nº 1/2, doi :10.1023/a:1007677805582, p. 9–40. URL http://dx.doi.org/10.1023/a:1007677805582. 34, 36, 40, 46

MOUSIN, L., L. JOURDAN, M.-E. K. MARMION et C. DHAENENS. 2016, «Feature selection using tabu search with learning memory : Learning tabu search», dans *Lecture Notes in Computer Science*, Springer International Publishing, p. 141–156, doi : 10.1007/978-3-319-50349-3_10. 99

MOUSIN, L., M.-E. KESSACI et C. DHAENENS. 2017a, «An Iterated Greedy-based Approach Exploiting Promising Sub-Sequences of Jobs to solve the No-Wait Flowshop Scheduling Problem», URL https://hal.archives-ouvertes.fr/hal-01579768. 74

MOUSIN, L., M.-E. KESSACI et C. DHAENENS. 2017b, «A new constructive heuristic for the no-wait flowshop scheduling problem», dans *Lecture Notes in Computer Science*, Springer International Publishing, p. 196–209, doi :10.1007/978-3-319-69404-7_14. 48

MUHLENBEIN, H. et G. PAASS. 1996, «From recombination of genes to the estimation of distributions i. binary parameters», dans *Parallel Problem Solving from Nature — PPSN IV*, Springer Science + Business Media, p. 178–187, doi :10.1007/3-540-61723-x_982. URL http://dx.doi.org/10.1007/3-540-61723-x_982. 10

NAGANO, M. S. et D. C. ARAÚJO. 2013, «New heuristics for the no-wait flowshop with sequence-dependent setup times problem», *Journal of the Brazilian Society of Mechanical Sciences and Engineering*, vol. 36, nº 1, doi :10.1007/s40430-013-0064-4, p. 139–151, ISSN 1806-3691. URL http://dx.doi.org/10.1007/s40430-013-0064-4. 18, 49, 52

NAWAZ, M., E. E. ENSCORE et I. HAM. 1983, «A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem», *Omega*, vol. 11, nº 1, doi :10.1016/0305-0483(83)90088-9, p. 91–95. URL http://dx.doi.org/10.1016/0305-0483(83)90088-9. 19, 50

O'HAGAN, S., J. KNOWLES et D. B. KELL. 2012, «Exploiting genomic knowledge in optimising molecular breeding programmes : Algorithms from evolutionary computing», *PLoS ONE*, vol. 7, nº 11, doi :10.1371/journal.pone.0048862, p. e48 862. URL http://dx.doi.org/10.1371/journal.pone.0048862. 32

OLIVEIRA, L. S., M. MORITA et R. SABOURIN. 2006, «Feature selection for ensembles using the multi-objective optimization approach», dans *Multi-Objective Machine Learning*, Springer Berlin Heidelberg, p. 49–74, doi :10.1007/3-540-33019-4_3. 26, 27

PAN, Q.-K., M. F. TASGETIREN et Y.-C. LIANG. 2008, «A discrete particle swarm optimization algorithm for the no-wait flowshop scheduling problem», *Computers & Operations Research*, vol. 35, nº 9, doi :10.1016/j.cor.2006.12.030, p. 2807–2839. 16, 22

PAN, Q.-K., L. WANG, M. F. TASGETIREN et B.-H. ZHAO. 2007a, «A hybrid discrete particle swarm optimization algorithm for the no-wait flow shop scheduling problem with makespan criterion», *The International Journal of Advanced Manufacturing Technology*, vol. 38, nº 3-4, doi :10.1007/s00170-007-1099-4, p. 337–347. URL http://dx.doi.org/10.1007/s00170-007-1099-4. 16, 22

PAN, Q.-K., L. WANG et B.-H. ZHAO. 2007b, «An improved iterated greedy algorithm for the no-wait flow shop scheduling problem with makespan criterion», *The International Journal of Advanced Manufacturing Technology*, vol. 38, nº 7-8, doi : 10.1007/s00170-007-1120-y, p. 778–786. 22

PAPADIMITRIOU, C. H. et K. STEIGLITZ. 1982, *Combinatorial optimization : algorithms and complexity*, Courier Corporation. 12

PHUONG, T., Z. LIN et R. ALTMAN. 2005, «Choosing SNPs using feature selection», dans *2005 IEEE Computational Systems Bioinformatics Conference (CSB'05)*, IEEE, doi :10.1109/csb.2005.22. 27

PUDIL, P. et J. HOVOVICOVA. 1998, «Novel methods for subset selection with respect to problem knowledge», *IEEE Intelligent Systems*, vol. 13, nº 2, doi :10.1109/5254.671094, p. 66–74. 24

QIAN, B., L. WANG, R. HU, D. HUANG et X. WANG. 2009, «A DE-based approach to no-wait flow-shop scheduling», *Computers & Industrial Engineering*, vol. 57, nº 3, doi : 10.1016/j.cie.2009.02.006, p. 787–805. 22

RAJENDRAN, C. 1994, «A no-wait flowshop scheduling heuristic to minimize makespan», *The Journal of the Operational Research Society*, vol. 45, nº 4, doi :10.2307/2584218, p. 472. 19

REDDI, S. S. et C. V. RAMAMOORTHY. 1972, «On the flow-shop sequencing problem with no wait in process†», *Journal of the Operational Research Society*, vol. 23, nº 3, doi : 10.1057/jors.1972.52, p. 323–331. 19

RICE, J. R. 1976, «The algorithm selection problem», dans *Advances in Computers*, Elsevier, p. 65–118, doi :10.1016/s0065-2458(08)60520-3. 33

RÖCK, H. 1984, «The three-machine no-wait flow shop is NP-complete», *Journal of the ACM*, vol. 31, nº 2, doi :10.1145/62.65, p. 336–345. URL http://dx.doi.org/10.1145/62.65. 15, 17

RUIZ, R. et T. STÜTZLE. 2007, «A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem», *European Journal of Operational Research*, vol. 177, nº 3, doi :10.1016/j.ejor.2005.12.009, p. 2033–2049. URL http://dx.doi.org/10.1016/j.ejor.2005.12.009. 79, 92

SAMARGHANDI, H. et T. Y. ELMEKKAWY. 2012, «A meta-heuristic approach for solving the no-wait flow-shop problem», *International Journal of Production Research*, vol. 50, nº 24, doi :10.1080/00207543.2011.648277, p. 7313–7326. 22

SANTOS, L. F., S. L. MARTINS et A. PLASTINO. 2008, «Applications of the DM-GRASP heuristic : a survey», *International Transactions in Operational Research*, vol. 15, nº 4, doi : 10.1111/j.1475-3995.2008.00644.x, p. 387–416. 36, 46

SCHINDL, D. et N. ZUFFEREY. 2013, «Solution methods for fuel supply of trains», *INFOR : Information Systems and Operational Research*, vol. 51, nº 1, doi :10.3138/infor.51.1.23, p. 23–30. URL http://dx.doi.org/10.3138/infor.51.1.23. 35, 36, 38, 41, 42, 44, 46

SCHUSTER, C. J. 2006, «No-wait job shop scheduling : Tabu search and complexity of subproblems», *Mathematical Methods of Operations Research*, vol. 63, nº 3, doi :10.1007/s00186-005-0056-y, p. 473–491. 22

SCHUSTER, C. J. et J. M. FRAMINAN. 2003, «Approximative procedures for no-wait job shop scheduling», *Operations Research Letters*, vol. 31, nº 4, doi :10.1016/s0167-6377(03)00005-1, p. 308–318. 22

SELEN, W. et D. HOTT. 1986, «A new formulation and solution of the flowshop scheduling problem with no in-process waiting», *Applied Mathematical Modelling*, vol. 10, nº 4, doi :10.1016/0307-904x(86)90053-3, p. 246–248. 19

SILVA, R. F., E. CASTRO, C. N. GUPTA, M. CETIN, M. ARBABSHIRANI, V. K. POTLURU, S. M. PLIS et V. D. CALHOUN. 2014, «The tenth annual MLSP competition : Schizophrenia classification challenge», dans *2014 IEEE International Workshop on Machine Learning for Signal Processing (MLSP)*, IEEE, doi :10.1109/mlsp.2014.6958889. 25

STORN, R. et K. PRICE. 1997, «Differential evolution a simple and efficient heuristic for global optimization over continuous spaces», *Journal of Global Optimization*, vol. 11, nº 4, doi :10.1023/a:1008202821328, p. 341–359. URL http://dx.doi.org/10.1023/a:1008202821328. 10

TAILLARD, E. 1993, «Benchmarks for basic scheduling problems», *European Journal of Operational Research*, vol. 64, nº 2, doi :10.1016/0377-2217(93)90182-m, p. 278–285. URL http://dx.doi.org/10.1016/0377-2217(93)90182-m. 19, 66, 74, 82

TELELIS, O. et P. STAMATOPOULOS. 2001, «Combinatorial optimization through statistical instance-based learning», dans *Proceedings 13th IEEE International Conference on Tools with Artificial Intelligence. ICTAI 2001*, IEEE Comput. Soc, doi :10.1109/ictai.2001.974466. 33

TSENG, L.-Y. et Y.-T. LIN. 2010, «A hybrid genetic algorithm for no-wait flowshop scheduling problem», *International Journal of Production Economics*, vol. 128, nº 1, doi :10.1016/j.ijpe.2010.06.006, p. 144–152. 22

WANG, C., X. LI et Q. WANG. 2010, «Accelerated tabu search for no-wait flowshop scheduling problem with maximum lateness criterion», *European Journal of Operational Research*, vol. 206, nº 1, doi :10.1016/j.ejor.2010.02.014, p. 64–72. URL http://dx.doi.org/10.1016/j.ejor.2010.02.014. 61

WATKINS, C. J. C. H. et P. DAYAN. 1992, «Q-learning», *Machine Learning*, vol. 8, nº 3-4, doi :10.1007/bf00992698, p. 279–292, ISSN 1573-0565. URL http://dx.doi.org/10.1007/BF00992698. 39

WEDGE, D. C. et D. B. KELL. 2008, «Rapid prediction of optimum population size in genetic programming using a novel genotype -», dans *Proceedings of the 10th annual conference on Genetic and evolutionary computation - GECCO '08*, Association for Computing Machinery (ACM), doi :10.1145/1389095.1389346. URL http://dx.doi.org/10.1145/1389095.1389346. 34, 36, 46

WISMER, D. A. 1972, «Solution of the flowshop-scheduling problem with no intermediate queues», *Operations Research*, vol. 20, nº 3, doi :10.1287/opre.20.3.689, p. 689–697. URL http://dx.doi.org/10.1287/opre.20.3.689. 16, 19

WRIGHT, S. 1932, *The roles of mutation, inbreeding, crossbreeding, and selection in evolution*, vol. 1, na. 78

XUE, B., M. ZHANG et W. N. BROWNE. 2013, «Particle swarm optimization for feature selection in classification : A multi-objective approach», *IEEE Transactions on Cybernetics*, vol. 43, nº 6, doi :10.1109/tsmcb.2012.2227469, p. 1656–1671. 26, 27

YANG, J. et V. HONAVAR. 1998, «Feature subset selection using a genetic algorithm», dans *Feature Extraction, Construction and Selection*, Springer US, p. 117–136, doi :10.1007/978-1-4615-5725-8_8. 26, 27

YANGMING, Z. 2017, *Learning-Driven Optimization Approaches for Combinatorial Search Problems*, thèse de doctorat, UniversitAngers. 30

ZEUGMANN, T., P. POUPART, J. KENNEDY, X. JIN, J. HAN, L. SAITTA, M. SEBAG, J. PETERS, J. A. BAGNELL, W. DAELEMANS, G. I. WEBB, K. M. TING, K. M. TING, G. I. WEBB, J. S. SHIRABAD, J. FRNKRANZ, E. HLLERMEIER, S. MATWIN, Y. SAKAKIBARA, P. FLENER, U. SCHMID, C. M. PROCOPIUC, N. LACHICHE et J. FRNKRANZ. 2011, «Particle swarm optimization», dans *Encyclopedia of Machine Learning*, Springer Science + Business Media, p. 760–766, doi :10.1007/978-0-387-30164-8_630. URL http://dx.doi.org/10.1007/978-0-387-30164-8_630. 46

ZHOU, Y., J.-K. HAO et B. DUVAL. «When data mining meets optimization : A case study on the quadratic assignment problem», . 36

ZHU, Z., Y.-S. ONG et M. DASH. 2007, «Markov blanket-embedded genetic algorithm for gene selection», *Pattern Recognition*, vol. 40, nº 11, doi :10.1016/j.patcog.2007.02.007, p. 3236–3248. 25

ZUFFEREY, N. et D. SCHINDL. 2015, «Learning tabu search for combinatorial optimization», dans *Operations Research and Enterprise Systems*, Springer International Publishing, p. 3–11, doi :10.1007/978-3-319-17509-6_1. 99, 105

Annexe A

Données d'exemple pour le NWFSP

	J1	J2	J3	J4	J5
M1	3	2	1	5	2
M2	2	1	4	1	3
M3	3	4	2	2	3
M4	3	3	1	2	1

TABLEAU A.1 – Temps d'exécution pour une instance à 5 tâches et 4 machines pour un problème d'ordonnancement de type flowshop sans temps d'attente

	J1	J2	J3	J4	J5
J1	-	5	4	3	3
J2	2	-	3	2	2
J3	2	4	-	1	3
J4	5	5	5	-	5
J5	3	5	4	2	-

TABLEAU A.2 – Matrice de délais pour une instance à 5 tâches et 4 machines pour un problème d'ordonnancement de type flowshop sans temps d'attente

Annexe B

Résultats IBI contre NEH et BIH

Instance	Best	NEH	BIH	IBI	Instance	Best	NEH	BIH	IBI
Ta01	1486	1538	1570	**1503**	Ta60	5958	6189	6226	**6127,79**
Ta02	1528	1561	1570	1563,75	Ta61	6397	6905	6828	**6714,75**
Ta03	1460	1580	1525	**1489**	Ta62	6234	6746	6703	**6522,17**
Ta04	1588	1640	1637	**1604**	Ta63	6121	6621	6503	**6372,21**
Ta05	1449	1507	1481	**1468**	Ta64	6026	6493	6417	**6272,54**
Ta06	1481	1519	1526	**1516,33**	Ta65	6200	6696	6577	**6477**
Ta07	1483	1528	1503	**1501,08**	Ta66	6074	6562	6458	**6378,21**
Ta08	1482	1505	1489	1505,08	Ta67	6247	6793	6698	**6527,79**
Ta09	1469	1513	1503	**1485,08**	Ta68	6130	6709	6630	**6450,96**
Ta10	1377	1490	1463	**1403,75**	Ta69	6370	6837	6842	**6643,67**
Ta11	2044	2075	2092	**2062,75**	Ta70	6381	6833	6845	**6689,17**
Ta12	2166	2238	2214	**2191**	Ta71	8077	8605	8502	**8345,71**
Ta13	1940	1996	1973	**1964**	Ta72	7880	8361	8235	**8234,62**
Ta14	1811	1907	1886	**1833,29**	Ta73	8082	8495	8450	**8280,5**
Ta15	1933	2053	1987	**1985**	Ta74	8348	8812	8744	**8609,71**
Ta16	1892	1958	2001	**1909,38**	Ta75	7958	8485	8448	**8225,12**
Ta17	1963	2021	2005	**2001**	Ta76	7801	8177	8161	**8023,62**
Ta18	2057	2132	2141	2144	Ta77	7866	8472	8337	**8161,96**
Ta19	1973	2144	2113	**2012,17**	Ta78	7913	8456	8249	**8208**
Ta20	2051	2129	2127	**2065,17**	Ta79	8161	8679	8501	**8456,12**
Ta21	2973	3035	2986	3002,58	Ta80	8114	8498	8543	**8404,17**
Ta22	2852	2897	2897	**2893**	Ta81	10700	11302	11224	**11087,5**
Ta23	3013	3080	3082	**3036**	Ta82	10594	11143	11151	**10906,75**
Ta24	3001	3194	3065	**3046**	Ta83	10611	11174	11090	**10884,96**
Ta25	3003	3072	3081	**3064,42**	Ta84	10607	11170	11144	**10940,88**
Ta26	2998	3130	3013	**3009**	Ta85	10539	11062	10948	**10835,83**
Ta27	3052	3152	3106	**3062,5**	Ta86	10690	11206	11071	**11029,46**
Ta28	2839	2946	2886	**2883**	Ta87	10825	11315	11334	**11187,42**
Ta29	3009	3057	3026	3046,5	Ta88	10839	11418	11450	**11147,83**
Ta30	2979	3026	3096	**3003,71**	Ta89	10723	11251	11177	**11018,29**

Ta31	3161	3497	3409	**3281,5**	Ta90	10798	11361	11308	**11100,54**
Ta32	3432	3576	3590	**3567,21**	Ta91	15319	16403	16163	**15879,25**
Ta33	3211	3467	3391	**3319,21**	Ta92	15085	16138	16068	**15802,96**
Ta34	3339	3569	3546	**3445,88**	Ta93	15376	16265	16203	**15989,83**
Ta35	3356	3607	3577	**3520,08**	Ta94	15200	16177	16026	**15777,21**
Ta36	3347	3498	3477	**3436,71**	Ta95	15209	16240	16196	**15863,08**
Ta37	3231	3420	3382	**3348,46**	Ta96	15109	16176	16025	**15790,62**
Ta38	3235	3526	3439	**3349,54**	Ta97	15395	16559	16488	**16093,71**
Ta39	3072	3301	3260	**3174,83**	Ta98	15237	16259	16239	**15932,83**
Ta40	3317	3523	3502	**3450,54**	Ta99	15100	16170	15909	**15739,67**
Ta41	4274	4446	4450	**4432,25**	Ta100	15340	16295	16176	**15966**
Ta42	4177	4335	4319	**4296,38**	Ta101	19681	20794	20543	**20303,29**
Ta43	4099	4414	4266	**4221,54**	Ta102	20096	21191	20980	**20805,71**
Ta44	4399	4565	4586	**4507,12**	Ta103	19913	21023	20940	**20554,46**
Ta45	4322	4525	4530	**4434,33**	Ta104	19928	21073	20783	**20568,79**
Ta46	4289	4482	4451	**4376,04**	Ta105	19843	20984	20599	**20484,46**
Ta47	4420	4605	4625	**4524,12**	Ta106	19942	21276	20891	**20597,79**
Ta48	4318	4568	4454	4506,58	Ta107	20112	21143	21097	**20749,5**
Ta49	4155	4362	4295	**4294,79**	Ta108	20056	21152	20869	**20637,58**
Ta50	4283	4459	4448	**4306,25**	Ta109	19918	21089	20939	**20539,67**
Ta51	6129	6298	6421	**6244,88**	Ta110	19935	21037	20782	**20512,38**
Ta52	5725	6107	5954	**5892,83**					
Ta53	5862	6208	6036	**6013,42**					
Ta54	5788	6074	5930	**5907,79**					
Ta55	5886	6218	6163	**6049,58**					
Ta56	5863	6193	6161	**6004,25**					
Ta57	5962	6213	6168	**6085,38**					
Ta58	5926	6199	6239	**6077,5**					
Ta59	5876	6105	6081	**6042,5**					

TABLEAU B.1 – Makespan obtenu sur les instances de Taillard pour NEH, BIH et IBI (valeur moyenne sur 30 exécutions). Les valeurs sont en gras lorsque IBI est meilleur que NEH et BIH en moyenne.