Optimisation boîte grise massivement parallèle et large échelle

Lorenzo Canonne

Université de Lille

Thèse

pour obtenir le grade de :

Docteur de l'Université de Lille

dans la spécialité

« Informatique »

par

Lorenzo Canonne

Optimisation boîte grise massivement parallèle et large échelle

Thèse soutenue le 19 décembre 2023 devant le jury composé de :

M.	Adrien Goëffon	Professeur, Université d'Angers	(Rapporteur)
M.	Lhassane Idoumghar	Professeur, Université de Haute-Alsace	(Rapporteur)
Mme	Corinne Lucet-Vasseur	Professeur, Université de Picardie	(Examinatrice)
M.	Cyril Fonlupt	Professeur, Université du Littoral Côte d'Opale	(Examinateur)
M.	Giuseppe Lipari	Professeur, Université de Lille	(Examinateur, Président)
M.	Bilel Derbel	Professeur, Université de Lille	(Directeur de Thèse)

Laboratoire inria-université de Lille
Unité de recherche CRIStAL

École doctorale MADIS-631
Mathématiques-Sciences du numérique et de leurs interactions

Résumé

L'optimisation boîte grise se distingue de l'optimisation boîte noire par le fait que des informations soient disponibles sur la structure du problème que l'on souhaite résoudre. Ces informations permettent de concevoir des approches très efficaces, car adaptées aux particularités des problèmes considérés ; on peut ainsi, en un temps raisonnable, traiter des problèmes de plus en plus grands et ce très efficacement. À ces avancées algorithmiques s'ajoutent l'accroissement de la puissance des supercalculateurs, celle-ci étant principalement le fruit de la multiplication des unités de calcul au sein d'un même système. Cependant, pour tirer parti efficacement de cette immense puissance de calcul, il faut adapter et/ou concevoir de nouveaux algorithmes. Cette thèse vise non seulement à l'élaboration d'approches boîtes grises massivement parallèles, mais également à obtenir une compréhension plus fine de la dynamique et des synergies des opérateurs boîtes grises les plus puissants, et ce, dans un environnement parallèle, mais aussi séquentiel.

Plus précisément, nous nous concentrons sur les problèmes d'optimisation pseudo-booléens k-bornés modélisables par des paysages Mk et plus particulièrement sur des instances de très grande taille. La communauté a proposé des algorithmes avancés exploitant les informations disponibles sur la structure de ce type de problèmes génériques. Parmi ceux-ci, on retrouve des recherches locales de type hill climber capables de trouver les mouvements améliorants en temps constant ; ainsi que des opérateurs de croisement recombinant deux optima locaux afin d'en obtenir un nouveau, tout en ayant la garantie que ce dernier soit au moins d'aussi bonne qualité que la meilleure des deux solutions croisées, et ce, en un temps linéaire.

Dans ce contexte, nos contributions se regroupent en deux parties. Dans une première partie, nous nous concentrons sur la conception d'approches parallèles. Nous proposons d'abord une variante d'un hill climber basée sur la coloration de graphes et fonctionnant en mémoire partagée. Nous proposons ensuite une nouvelle approche complètement distribuée, coopérative et massivement parallèle, fonctionnant sur l'un des plus puissants calculateurs au monde, le Fugaku. Dans une seconde partie, nous nous concentrons sur la dynamique de recherche induite par l'utilisation des croisements boîte grise. Nous décrivons deux contributions qui améliorent successivement l'état de l'art existant. La première concerne la conception et l'analyse de nouvelles stratégies d'échappement avancées, tandis que la seconde s'intéresse à l'ajout d'une phase d'initialisation permettant de diriger la recherche vers des zones prometteuses de l'espace de recherche. Nous concluons nos investigations par une étude comparative approfondie de la synergie entre les approches proposées et les différents opérateurs de croisement de la littérature, permettant ainsi de mieux appréhender la dynamique de recherche des approches proposées, et de discuter des pistes d'amélioration fondées sur de solides observations empiriques.

Abstract

In contrast to blackbox optimization, graybox optimization assumes that some information is available about the structure of the problem to be solved. This information allows us to design highly efficient solving approaches. As a result, increasingly large-size problems can be solved very efficiently, in a reasonable amount of time. In addition to such algorithmic advances, the power of supercomputers continues to increase very substantually, mainly as a result of the increasing number of computing units that can be used. However, in order to make the most effective use of the implied computing power, new algorithms need to be adapted and/or designed. The aim of this thesis is both to develop massively parallel gray-box approaches, and to gain a more fundamental understanding of the dynamics of the latest graybox operators, in both parallel and sequential environments.

More precisely, we focus on solving large-size k-bounded pseudo-Boolean optimization problems modeled as MK-landscapes. In recent years, the community has made a lot of advances in exploiting the information available on the structure of these generic optimization problems. This includes the design of hill climbers, and local search based heuristics, that are able to find improving moves in constant time, as well as evolutionary crossover operators that can successfully recombine two local optima in linear time in the size of the problem.

In this context, our contributions can be grouped into two parts. In the first part, we focus on the design of scalable parallel solving approaches. We propose a shared memory parallel hill climber variant based on graph coloring. We then investigate the design and analysis of a fully distributed, cooperative and massively parallel approach running on one of the world's most powerful computers, the Fugaku. In the second part, we focus on the search dynamics induced by the use of graybox crossovers. We describe two contributions improving the existing state-of-the-art solving approaches. The first contribution relates to the design and analysis of novel escape strategies, and the second one focuses on the design of a warming initialisation phase to guide the search towards more promising areas of the search space. We conclude our investigations with a comprehensive in-depth comparative study on the synergy between the proposed approaches and the various crossover operators from in the literature; thereby enabling a better fundamental understanding of the search dynamics of the proposed approaches, and highlighting new empirical findings that should lead to new improved algorithms.

Table des matières

Introduction

Contexte général

L'optimisation combinatoire consiste à trouver la (ou les) meilleure solution parmi un ensemble très grand, mais fini, de solutions possibles. De nombreux problèmes appliqués peuvent se modéliser comme des problèmes d'optimisation combinatoire. On peut citer notamment les problèmes de satisfaisabilité logique, de tournées de véhicules, d'ordonnancements, etc. Parmi les problèmes d'optimisation combinatoire, il existe également des problèmes de nature fondamentale, tels que par exemple les problèmes de verre de spin ou les paysages NK, qui permettent de modéliser un panorama large de problèmes et d'applications. Ces derniers problèmes présentent aussi l'avantage d'être simple de formulation, facilement configurable pour générer des instances avec un degré de difficulté variable, et offrent ainsi un intéressant champ d'étude pour concevoir de nouveaux algorithmes d'optimisation, et en évaluer la puissance et la dynamique de recherche. Dans cette thèse, nous nous plaçons dans ce cadre d'étude. Nous considérons la résolution de problèmes d'optimisation combinatoire modélisables par des paysages NK. Ces derniers représentent en outre des problèmes difficiles d'optimisation pseudo-booléenne, pour lesquels différentes approches de résolutions de type heuristiques ont été développées. On peut notamment classer les approches existantes dans deux grandes catégories, les approches dites boîte noire et les approches dites boîte grise. Tandis que les premières ne supposent aucune connaissance à priori sur les problèmes à résoudre, les secondes font usage de l'information relative à la structure des problèmes et de leur formulation, telle que par exemple les interactions entre les variables. De telles informations permettent de concevoir des algorithmes beaucoup plus puissants que dans un cadre purement boîte noire. C'est donc précisément dans le cadre de ce type de méthodes que se situe le périmètre général des travaux de cette thèse. Plus spécifiquement, dans le cadre de cette thèse, on s'intéressera à :

— des opérateurs de recherche locale très efficaces capables de trouver des mouvements améliorants en temps constant (de façon indépendante à la dimension du problème) ;
— des opérateurs génétiques de croisement, s'appuyant sur la partition du graphe

d'interaction des variables, et permettant d'explorer de façon implicite un nombre éventuellement exponentiel de solutions enfants en un temps linéaire ;
— des algorithmes (méta-) heuristiques plus avancés intégrant et hybridant ces différents composants algorithmiques de façon transparente, pour une capacité globale de recherche extrêmement puissante.

D'un côté, toutes les approches boîte grise précédemment mentionnées sont intrinsèquement séquentielles. De ce point de vue, elles ne permettent pas de tirer profit de façon immédiate de la puissance de calcul offerte par les (super)calculateurs modernes.

Ainsi, on s'intéresse dans cette thèse à l'élaboration d'algorithmes d'optimisation capables d'intégrer des ressources toujours plus nombreuses et massivement parallèle, afin de résoudre des problèmes difficiles de très grande taille. D'un autre côté, l'amélioration des approches séquentielles existantes représente également un défi majeur. Les dernières avancées reposent sur la conception d'opérateurs de recherche de plus en plus puissants. Cependant, peu de travaux se sont intéressés à la dynamique de recherche des algorithmes existants, et à la façon de les améliorer. Ainsi, on s'intéresse également à l'analyse fine et systématique de la dynamique des algorithmes existants, afin d'en extraire la connaissance nécessaire à la conception de nouveaux algorithmes, toujours plus puissants. Ces deux axes de recherche ont donné lieu à de nombreuses contributions que nous résumons brièvement ci-après.

Contributions

Les contributions présentées dans ce manuscrit peuvent se décomposer en deux axes principaux. Le premier se concentre sur la conception et l'analyse d'approches d'optimisation parallèles, tandis que le second se concentre sur la conception et l'analyse de nouvelles approches de résolution dans un environnement de calcul séquentiel. L'intégralité de nos travaux se place dans un paradigme boîte grise pour l'optimisation de problèmes pseudo-booléens dits k-bornés.

Au sein du premier axe de recherche, nos contributions visent de façon générale à développer des méthodes parallèles capables de passer à l'échelle, et de s'attaquer à des problèmes de plus en plus grands et difficiles de façon efficace. Ainsi, notre première contribution présente une recherche locale boîte grise parallèle, se basant sur la coloration du graphe d'interaction des variables, et fonctionnant en mémoire partagée. Cette approche est capable de s'exécuter sur plus d'une quarantaine de cœurs de calcul. Dans une seconde contribution, nous introduisons un framework massivement parallèle inspiré des modèles en îles, et fonctionnant de façon asynchrone et complètement décentralisée. Dans ce cadre, nous introduisons une variante coopérative de l'algorithme état de l'art du domaine, présentant divers niveaux de parallélisme. Notre approche est expérimentée sur le super-

calculateur Fugaku, deuxième plus puissant au monde, avec un nombre de cœurs allant jusqu'à plus de dix mille.

Au sein du second axe de recherche, nos travaux visent à améliorer l'algorithme état de l'art du domaine en se concentrant sur trois aspects complémentaires. Dans un premier temps, nous proposons des nouveaux mécanismes d'échappement basés sur deux stratégies, l'une pro-active et l'autre rétroactive. Ces mécanismes conduisent à une première amélioration substantielle des performances de l'algorithme état de l'art existant. Dans un second temps, une étude approfondie de la dynamique de recherche montre qu'un biais fort, et relativement néfaste en termes de la qualité des solutions finales trouvées, est introduit par l'utilisation d'opérateurs de croisement boîte grise pourtant connus pour leur extrême efficacité. En conséquence, nous proposons un nouvel algorithme simple s'appuyant sur l'intégration d'une phase explorative initiale capable par la suite de fournir des solutions à haut potentiel d'amélioration. Nous en étudions alors la dynamique et la performance, et nous montrons que ce nouvel algorithme est capable d'améliorer l'état de l'art sur un large panel d'instances difficiles et de grandes dimensions. Enfin, dans un troisième temps, nous considérons de comparer de façon plus systématique les combinaisons possibles entre les différentes variantes d'algorithmes proposés et les derniers opérateurs de croisement boîte grise. On s'intéresse ici non seulement à la performance des algorithmes ainsi obtenus, mais également à la dynamique de recherche sous-jacente. À cet effet, nous proposons des nouveaux outils et concepts pour une analyse expérimentale plus fine permettant de mieux comprendre les différences de dynamique et de performances observées entre les variantes d'algorithmes considérées.

Plan du manuscrit

Ce manuscrit de thèse s'organise en 3 parties et 7 chapitres, que l'on peut résumer comme suit :

— La première partie est composée de deux chapitres d'introduction. Le chapitre 1 présente d'un point de vue générale le domaine de l'optimisation combinatoire et les principales méthodes de résolutions qui nous intéressent dans la suite. Le chapitre 2 se consacre au paradigme boîte grise et à la présentation des méthodes état de l'art du domaine.

— La deuxième partie est dédiée à nos contributions autour de l'optimisation parallèle. Elle se compose des chapitres 3 et 4, lesquels présentent, respectivement, une recherche locale parallèle en mémoire partagée basé sur la coloration de graphe, et un framework massivement parallèle, distribué et asynchrone, associé à une variante coopérative de l'algorithme état de l'art séquentiel.

— La troisième partie est dédiée à nos contributions autour de la conception et à l'ana-

lyse d'approches séquentielles. Elle se divise en trois chapitres, chacun décrivant une nouvelle contribution de façon incrémentale. Ainsi, le chapitre 5 propose de revisiter l'algorithme état de l'art séquentiel en proposant deux variantes introduisant des mécanismes d'échappement différents. Le chapitre 6 propose l'ajout d'une phase exploratoire en guise d'initialisation avant l'utilisation des opérateurs de croisement ; créant ainsi un algorithme capable d'améliorer encore davantage les approches proposées précédemment. Le chapitre 7 étudie la synergie entre les approches proposées dans les chapitres 5 et 6 lorsque combiner aux derniers opérateurs de croisements boîte grise. En plus d'étudier leurs performances relatives, cette contribution s'intéresse également à l'explication des différences observées au niveau de la dynamique de recherche.

Le manuscrit se termine par une conclusion retraçant l'intégralité des contributions, puis discutant de quelques pistes de recherches inspirées des différents résultats obtenus.

Résumé des publications

Les travaux menés dans le cadre de cette thèse ont donné lieu à cinq articles publiés et présentés à la conférence internationale de référence de notre domaine GECCO (ACM SIGEVO Genetic and Evolutionary Computation Conference). Ci-après, nous listons ces publications, en indiquant entre parenthèses les chapitres du manuscrit où elles apparaissent. Il est à noter que les contributions de deux de ces publications ne sont pas reportées dans ce manuscrit, et les contributions de deux des cinq chapitres de contributions (Chapitres 4 et 7) sont au moment de l'écriture de ce manuscrit en cours de publication dans des revues.

— To Combine or not to Combine Graybox Crossover and Local Search ? Lorenzo Canonne, Bilel Derbel, Francisco Chicano, Gabriela Ochoa. GECCO 2023 : 257-265. (Chapitre 6)

— Local Optima Markov Chain : A New Tool for Landscape-aware Analysis of Algorithm Dynamics. Francisco Chicano, Gabriela Ochoa, Bilel Derbel, Lorenzo Canonne. GECCO 2023 : 284-292. (Non reporté dans ce manuscrit)

— An Investigation of Geometric Semantic GP with Linear Scaling. Giorgia Nadizar, Fraser Garrow, Berfin Sakallioglu, Lorenzo Canonne, Sara Silva, Leonardo Vanneschi. GECCO 2023 : 1165-1174. (Non reporté dans ce manuscrit)

— DRILS revisited : On the combination of perturbation with graybox optimization techniques. Lorenzo Canonne, Bilel Derbel. GECCO 2022 : 204-212. (Chapitre 5)

— A graph coloring based parallel hill climber for large-scale NK-landscapes. Bilel Derbel, Lorenzo Canonne. GECCO 2021 : 216-224. (Chapitre 4)

Première partie

Optimisation Combinatoire et état de l’art des méthodes boîte grise

Chapitre 1

Optimisation Combinatoire et problèmes pseudo-booléens k-bornés

1.1 Problèmes d'optimisation pseudo-booléens

1.1.1 Problèmes d'optimisation

Un problème d'optimisation est défini par une fonction objective f et un ensemble de solutions X, appelé espace de recherche. L'espace de recherche contient l'ensemble des solutions potentielles possibles pour le problème. La fonction f associe à chaque élément x de l'ensemble X une valeur correspondant à sa qualité, aussi appelée fitness. Le but est alors de trouver une solution de l'ensemble X qui maximise ou minimise la fonction objective f. En pratique, il n'y a pas de différence entre les problèmes de maximisation ou de minimisation, puisqu'il suffit, pour passer de l'un à l'autre, de considérer l'opposé de la fonction objective. Sans perte de généralité, nous considérerons dans ce manuscrit des problèmes de maximisation. Les problèmes considérés seront de type mono-objectif. Cela signifie qu'il n'y a qu'un seul objectif à optimiser. Plus formellement, un problème de maximisation mono-objectif se définit comme suit :

$$x^* = \arg\max_{x \in X} f(x)$$

La fonction objective n'est pas nécessairement injective, ce qui veut dire que plusieurs solutions peuvent éventuellement avoir la même qualité, et que donc la solution optimale x^* n'est pas forcément unique.

1.1.2 Optimisation Combinatoire pseudo-booléenne

On distingue deux grands types de problèmes d'optimisation, suivant la nature des variables à optimiser. Dans le cas de l'optimisation continue, les variables sont des nombres

réels ; l'espace de recherche est donc en théorie de taille infinie. Tandis que dans le cas de l'optimisation combinatoire, l'espace de recherche est discret et fini. C'est ce dernier type de problème que nous traiterons dans ce manuscrit.

On peut citer quelques célèbres problèmes d'optimisation concrets tels que : les problèmes d'ordonnancement, les problèmes de tournée de véhicules ou les problèmes de satisfaisabilités.

Parmi la famille des problèmes d'optimisation combinatoire, on distingue un sous-ensemble appelé problèmes d'optimisation pseudo-booléens. La particularité de ceux-ci réside dans le fait que les variables sont sous forme binaire et ne peuvent prendre que deux valeurs : 0 ou 1. La fonction objective, quant à elle, n'a pas de restrictions particulières ; dans le cas le plus général, elle retourne un réel.

1.1.3 Problèmes pseudo-booléens k-bornés

Dans cette thèse, nous nous intéressons à des problèmes dits k-bornés. Plus précisément, on considère des problèmes pseudo-booléens, avec donc des variables binaires, ayant la particularité d'avoir une fonction objective qui se décrit comme la somme de plusieurs sous-fonctions, chacune d'entre elles dépendant d'au plus k variables. Plus formellement, la fonction objective d'un tel problème est supposée être la somme de m sous-fonctions comme donnée par la formule suivante :

$$f(x) = \sum_{i=1}^{m} f^i(x, masque_i^k)$$

Avec chaque sous-fonctions f^i définissant elle-même un problème pseudo-booléen, mais dépendant de seulement k variables, i.e., $masque_i^k$ indiquant un sous-ensemble de k variables qui servent à calculer la contribution de chaque sous-fonction f^i à la fonction globale f. Généralement pour chaque sous-fonction, il y a donc au plus 2^k différentes valeurs de contribution possibles correspondant à l'ensemble des configurations que l'on peut obtenir avec les k variables binaires associées. Les problèmes considérés dans ce manuscrit et présentés dans les paragraphes suivants sont tous des problèmes pseudo-booléens k-bornés.

Paysages NK

Introduits par Stuart Kauffman en 1987 [20] et en 1989 pour leurs formes modernes [21], les paysages NK sont souvent utilisés dans le domaine de l'optimisation combinatoire ; et ce, en particulier, en raison de leur généricité, leur difficulté, et leur capacité à fournir de façon flexible des instances de problèmes avec une rugosité paramétrables. La définition de ces paysages repose essentiellement sur l'utilisation de deux paramètres : le premier,

N, correspond aux nombres de variables du problème, et donc à sa taille ; le deuxième, K, représente le degré d'épistasie. Ce dernier influe sur le nombre de variables impactées par chaque sous-fonction. Une instance de paysage NK est composée d'autant de sous-fonctions qu'il y a de variables, soit N sous-fonctions, chacune d'entre elles dépendant de $K+1$ variables. La fonction objective est donc définie comme la somme suivante :

$$f(x) = \sum_{l=1}^{N} f^{(l)}(x_{i_{(l,0)}}, x_{i_{(l,1)}}, \ldots, x_{i_{(l,K)}})$$

Pour chaque sous-fonction, les contributions correspondant au $K+1$ configurations possibles sont des nombres réels appartenant à l'intervalle $[0,1]$. Il existe une variante souvent utilisée dans la littérature, appelée NKQ, et qui introduit un troisième paramètre, un entier Q. Ce dernier sert à définir le codomaine des sous-fonctions. Ainsi, pour un problème NKQ, les contributions sont des entiers et non plus des réels, appartenant à l'intervalle $[0, Q[$. Cela permet d'affiner la rugosité du paysage en donnant davantage de contrôle sur la possible existence de plus ou moins de plateaux (c'est-à-dire de neutralité entre les solutions).

On distingue en général deux grandes catégories de paysages NK en fonction de la façon dont les sous-fonctions sont définies. Le premier type concerne les paysages dits adjacents. Dans un paysage adjacent, une sous-fonction i dépend de la variable i ainsi que des K variables suivantes, de $i+1$ jusqu'à $(i+K) \mod N$. Ces paysages sont faciles à optimiser, puisqu'ils peuvent être résolus en temps polynomial en utilisant des techniques de programmation dynamique [41]. Le deuxième type concerne les paysages aléatoires. Dans ceux-ci, les variables associées à chaque sous-fonction sont choisies aléatoirement (à l'exception de la première variable de chaque sous-fonction qui correspond bien sûr à l'indice de la sous-fonction). Ces problèmes sont NP-difficiles pour $K > 1$[41].

Paysages Mk

Formalisés par Darrell Whitley [40], les paysages Mk sont des problèmes pseudo-booléens avec M sous-fonctions, chacune d'entre elles dépendant d'au plus k variables. Contrairement au paysage NK, il n'y a pas de contrainte particulière sur les sous-fonctions. En ce sens, les paysages Mk sont moins restrictifs et permettent de modéliser/généraliser de nombreux problèmes pseudo-booléens tels que MAX-kSAT, les paysages NK ou bien le problème des verres de spin.

1.2 Recherche locale : les bases

1.2.1 Préliminaire

Afin de résoudre un problème d'optimisation combinatoire, la communauté scientifique a proposé de très nombreuses méthodes. Pour simplifier le propos, on peut considérer deux catégories, les méthodes de résolution exactes et les méthodes heuristiques.

Comme leur nom l'indique, les premières garantissent, en un temps plus ou moins long, de trouver le résultat optimal. En théorie, il suffit de tester toutes les solutions possibles, on parle de méthodes exhaustives. En pratique, le nombre de solutions à considérer croît exponentiellement et il devient très vite impossible d'utiliser ces méthodes pour des problèmes algorithmiquement très difficiles. L'utilisation de techniques avancées comme par exemple l'algorithme Branch-and-Bound [23, 24], avec ces différentes composantes de branchement et d'élagage, permet d'augmenter l'efficacité de ce type d'approches exhaustives en organisant l'exploration de l'espace de recherche sous forme d'arbre. L'usage d'un arbre d'exploration est couplé à l'utilisation d'une fonction permettant de calculer une borne supérieure ou inférieure pour l'ensemble des solutions appartenant à la sous-arborescence. Ceci permet, lorsque un sous-arbre n'est pas prometteur, d'éviter d'avoir à l'explorer, limitant donc le nombre de solutions à calculer tout en s'assurant que la solution optimale est trouvée.

Cependant, ce type d'approche reste limité en fonction de la spécificité du problème à résoudre, et surtout dans le cas où le nombre de variables à optimiser est très important. En particulier, dans le cas des paysages NK aléatoires qui sont au cœur de cette thèse, les meilleures méthodes de résolution exactes ne sont capables de trouver la solution optimale que pour des problèmes de quelques dizaines de variables seulement [30], limitant ainsi leur intérêt. En pratique, il faut donc s'en remettre à d'autres approches en acceptant le fait que trouver la solution optimale est potentiellement impossible à garantir en un temps raisonnable. Des heuristiques sont ainsi souvent utilisées et offrent de très bons résultats.

De façon générale, les algorithmes de résolution heuristiques permettent de trouver en un temps raisonnable (souvent polynomial) une solution de bonne qualité à un problème d'optimisation. Bien que ces approches soient souvent très efficaces en pratique, elles ne permettent pas de garantir que la solution trouvée soit optimale. On peut décomposer la classe des algorithmes heuristiques en deux catégories. Les heuristiques qui construisent une solution en choisissant au fur et à mesure la valeur des variables, on parle de méthodes constructives ; et celles qui tentent d'améliorer la qualité d'une solution en se déplaçant de façon spécifique dans l'espace de recherche, souvent par le biais de petites modifications successives. Ce sont ces dernières approches qui seront étudiées dans ce manuscrit, en particulier les approches à base de *recherche locale*. Avant d'aller plus loin, il nous faut introduire quelques notions supplémentaires.

1.2.2 Voisinage et distance de Hamming

Considérons une solution quelconque x. On appelle voisinage de la solution x, souvent noté $N(x)$, l'ensemble des solutions que l'on peut atteindre en partant de cette dernière et en lui appliquant une fonction chargée de l'altérer légèrement. Par exemple, cette fonction peut consister en l'échange des valeurs de 2 variables. Dans ce cas, la taille du voisinage d'une solution composée de n variables est $n(n-1)$, i.e., il existe localement autant de voisins possible parmi lesquels l'algorithme peut être amené à choisir. Dans le cas des problèmes pseudo-booléens qui nous intéressent, les voisinages sont souvent définis à l'aide de la distance de Hamming. La distance de Hamming représente alors le nombre de variables différentes entre deux solutions voisines. Ainsi, il est possible, par exemple, de définir un voisinage comme étant l'ensemble des solutions à une distance de Hamming 1, c'est-à-dire l'ensemble des solutions que l'on peut obtenir en changeant l'état d'une seule variable. Dans le cas de variables binaires, étant donné que chaque variable n'a que 2 états possibles, la taille du voisinage à distance 1 d'un problème de taille n est n.

1.2.3 Hill Climber et optima locaux

De façon générale, les méthodes qui se basent sur la notion de voisinage afin de se déplacer itérativement dans l'espace de recherche s'appellent des recherches locales. Pour citer un exemple populaire, l'algorithme 2-opt [16] en est un exemple parmi les plus connus. Il consiste à échanger 2 arcs d'une tournée pour le problème du voyageur de commerce, et est en pratique relativement efficace pour trouver des améliorations successives d'une tournée.

Quand toutes les solutions voisines ne sont pas améliorantes, on dit que la solution courante est un optimum local, c'est-à-dire la meilleure de son voisinage. Il existe de nombreuses stratégies pour choisir vers quel voisin se déplacer [33, 3, 2, 4]. Suivant la stratégie mise en place, il est alors possible d'accepter de se déplacer vers une solution du voisinage moins bonne afin de poursuivre la recherche [22, 7] ou de changer de voisinage [28]. Dans le paragraphe suivant, nous présentons une recherche locale des plus basiques : le Hill Climber (ou algorithme de descente locale).

Le Hill Climber est un algorithme de recherche locale dont la stratégie de recherche consiste à se déplacer uniquement vers les voisins qui ne dégradent pas la solution courante, puis d'itérer jusqu'à ce qu'il n'y ait plus d'améliorations possibles ; à la suite de quoi, la recherche ne progresse plus et s'arrête naturellement. La solution ainsi obtenue est alors un optimum local. Malgré son extrême simplicité, il existe de nombreuses variantes de cet algorithme qui entraînent des comportements et des performances éventuellement très variables. Il faut ainsi une bonne connaissance du problème, et bien souvent de nombreuses expériences, afin de choisir la variante la plus adéquate [2, 4]. Les variantes

les plus connues concernent la stratégie du choix du voisin suivant à chaque itération (appelée aussi règle pivot) :

— *meilleur améliorant* : l'idée est de parcourir l'ensemble des solutions du voisinage et de se déplacer vers la meilleure solution ;
— *premier améliorant* : on parcourt le voisinage et on choisit la première solution améliorante ;
— *plus petit améliorant* : étant en quelque sorte le dual du *meilleur améliorant*, on parcourt ici l'ensemble du voisinage, mais on choisit cette fois la solution améliorante qui offre la plus petite amélioration possible.

Suivant le type et la taille du voisinage considéré, les Hill Climbers peuvent converger très vite vers un optimum local. Une fois cet optimum local atteint, la recherche est coincée. Or, il n'y a aucune garantie que cet optimum local soit la meilleure solution au problème d'optimisation que l'on essaie de résoudre. De ce fait, en pratique, on utilise soit des variantes de cet algorithme pour s'échapper des optima locaux et poursuivre la recherche, soit on intègre le Hill Climber dans des algorithmes plus complexes. Dans cette thèse, nous nous intéressons en particulier à un type d'algorithme à base de recherche locale itérée. Ceci est décrit dans le paragraphe suivant.

1.2.4 Recherche locale itérée

La recherche locale itérée [25] est une métaheuristique à la base des algorithmes états de l'art pour la résolution des paysages NK et Mk. Comme son nom l'indique, une métaheuristique généralise, d'un certain point de vue, le concept d'algorithmes de résolution heuristique, en intégrant des stratégies de recherche de haut-niveau suffisamment générales pour être appliquées à de nombreux types de problèmes [26]. En ce qui concerne la recherche locale itérée ; et comme son nom l'indique, cette dernière se base sur le concept simple qui consiste à répéter itérativement une recherche locale plus basique. Il s'agira dans le cadre de nos travaux d'un algorithme de type Hill Climber, mais cela peut être tout autre type de recherche locale. L'idée générale, et générique, est alors d'introduire une perturbation sur un optimum local afin de pouvoir mener la recherche dans d'autres régions de l'espace de recherche. Ceci dans l'espoir d'obtenir et de découvrir de nouveaux optima locaux de meilleure qualité. Comme l'on peut le voir dans le pseudo-code de l'algorithme 1, l'architecture générale d'une recherche locale itérée est très simple. Cependant, sa performance en pratique dépend très fortement de certains choix algorithmiques, dont voici les 3 principaux :

— L'algorithme de recherche locale (dans notre cas un Hill Climber) et les différents choix inhérents à ce dernier : le voisinage, la stratégie de déplacement (par exemple le choix du meilleur ou du premier améliorant pour le Hill Climber).
— La perturbation : si elle est trop faible, cela peut conduire à rester bloqué dans une

Algorithme 1 : Recherche locale itérée

```
   // Initialisation d'une solution suivie d'une recherche locale
1  x_0 ← Générateur aléatoire();
2  x* ← Recherche Locale(x_0);
3  repeat
       // On itére ensuite les 3 composantes principales jusqu'à ce que la
          condition d'arrêt soit satisfaite
4      x' ← Perturbation(x*) ;
5      x*' ← Recherche Locale(x') ;
6      x* ← Critère d'acceptation(x*,x*') ;
7  until temps restant ou objectif de qualité atteint;
```

région de l'espace de recherche, voir même à retomber sur le même optimum local. Si elle est trop forte, alors cela revient en quelque sorte à recommencer la recherche en partant d'une solution aléatoire.

— Le critère d'acceptation : la recherche locale itérée étant un algorithme qu'on qualifie d'algorithme à trajectoire, elle fonctionne typiquement et le plus souvent "sans" mémoire. Ainsi, quand on obtient un nouvel optimum local, on doit décider si on le rejette ou si on l'accepte pour continuer la recherche. Parmi les stratégies utilisées, on retrouve les 3 suivantes : n'accepter que les optima locaux améliorants, accepter les optima locaux améliorants ou de même qualité, accepter tous les optima locaux.

Il n'y a malheureusement pas de recommandation générale et définitive à suivre, indépendamment du problème et du contexte d'optimisation, pour choisir la configuration optimale d'une recherche locale itérée. C'est dans ce contexte général que plusieurs de nos contributions peuvent se situer, à savoir la conception de recherches locales efficaces pour le type de problèmes qui nous intéressent.

Nous avons maintenant fini cette brève introduction des notions nécessaires pour présenter les algorithmes état de l'art sur les problèmes qui seront considérés dans ce manuscrit. Le chapitre suivant sera donc consacré à une introduction rapide du concept d'optimisation boîte grise puis à une revue détaillée des meilleurs méthodes de la littérature consacrées à la résolution des paysages NK et Mk.

Chapitre 2

Optimisation boîte grise

2.1 Paradigme

L'optimisation boîte grise fait référence à un paradigme général en optimisation dans lequel des informations sur la structure du problème sont disponibles, et peuvent être utilisées lors du processus d'optimisation. En ce sens, elle diffère de l'optimisation boîte-noire où aucune information n'est supposée disponible ni connue par le processus d'optimisation. C'est un aspect largement étudié par la communauté scientifique de par sa capacité à permettre le développement d'algorithmes et de techniques d'optimisation très efficaces. Par exemple, l'information (boîte grise) disponible permet de réduire le temps de calcul de la fonction objective, ou bien d'introduire des opérateurs avancés comme nous le verrons plus tard avec des opérateurs de croisements particuliers. Ainsi, pour des paysages Mk, dans un contexte d'optimisation boîte grise, on supposera que la définition des variables intervenant dans les sous-fonctions est connue par le solveur. Autrement dit, on sait quelles variables interagissent entre elles et au sein de quelles sous-fonctions ; ceci de par la disponibilité de cette information dans la formulation/définition formelle des paysages. Cela conduit à l'élaboration de méthodes de résolution très efficaces. Nous les présenterons dans la suite de cette section. Avant cela, il nous faut introduire une représentation de ces problèmes sur laquelle repose toutes les approches état de l'art existantes : le graphe d'interaction des variables ou VIG [40].

2.2 Graphe d'interactions des variables et autres structures de données boîte grise

Pour mieux se représenter un paysage Mk, on s'intéresse, dans un contexte d'optimisation boîte grise, aux interactions entre les variables. Pour modéliser ses interactions, on utilise un graphe, noté VIG (pour 'Variable Interaction Graph'). L'idée est la suivante :

$$f^0(x_0,x_1,x_2),\ f^1(x_1,x_4,x_2),\ f^2(x_2,x_6,x_4),\ f^3(x_3,x_9,x_0),\ f^4(x_4,x_1,x_5),$$

$$f^5(x_5,x_7,x_1),\ f^6(x_6,x_0,x_2),\ f^7(x_7,x_9,x_0),\ f^8(x_8,x_6,x_4),\ f^9(x_9,x_{14},x_7)$$

$$f^{10}(x_{10},x_4,x_{13}),\ f^{11}(x_{11},x_6,x_8),\ f^{12}(x_{12},x_{10},x_{11}),\ f^{13}(x_{13},x_4,x_5),\ f^{14}(x_{14},x_9,x_3)$$

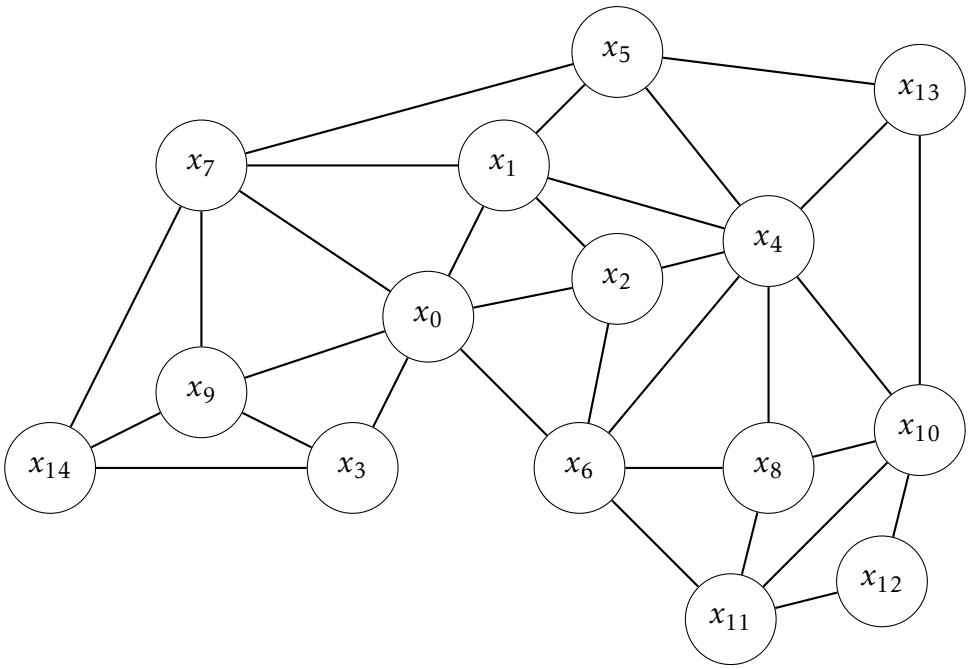

FIGURE 2.1 – *Exemple de représentation d'un paysage NK aléatoire sous forme de graphe d'interactions des variables*

chaque sommet du graphe VIG représente une variable de notre problème, et il existe une arête entre deux variables/sommets si celles-ci font toutes les deux partie d'au moins une sous-fonction. Une illustration est donnée dans la figure 2.1 qui expose un exemple de graphe d'interaction pour un paysage Mk, dont les sous-fonctions correspondantes sont donc supposées connues. Pour compléter cette représentation à base de graphe, une autre structure de données est nécessaire pour stocker pour chaque variable l'ensemble des sous-fonctions dans lesquelles celle-ci apparaît, et ce, typiquement sous-forme de liste d'adjacences. Ces deux structures sont à la base de tous les algorithmes présentés dans la suite de ce manuscrit. Comme ces différentes structures de données seront à la base de tous les algorithmes de ce manuscrit, nous définissons plus formellement quelques notations afin de pouvoir les réutiliser invariablement plus tard :

— $VIG(x)$: ensemble des voisins de x dans le graphe d'interactions des variables ;
— $F^-(l)$: ensemble des variables apparaissant dans la sous-fonction l ;
— $L^-(x)$: ensemble des sous-fonctions impactées par la variable x.

2.3 Hamming Ball Hill Climber

2.3.1 Historique et contexte

Dans un souci de lisibilité et de concision, nous ne pouvons retracer l'intégralité de l'historique des algorithmes de type Hill Climber proposés par la communauté scientifique. Nous passerons donc en revue les dernières avancées qui précèdent l'état de l'art, avant de nous concentrer sur l'algorithme appelé 'Hamming Ball Hill Climber' (également abrégé en HBHC dans la suite) spécifiquement décrit dans le contexte des paysages Mk.

Whitley et al [38] ont proposé un Hill Climber avec une stratégie 'meilleur améliorant'. Leur approche est prouvée extrêmement plus rapide qu'une approche naïve qui consisterait à réévaluer la qualité d'un voisin sans utiliser l'information boîte grise sur le paysage Mk. Pour cela, ils se basent sur la décomposition de Walsh de la fonction objective. Chen et al [8] ont ensuite amélioré ce résultat en utilisant des dérivées secondes de fonctions pseudo-booléennes. De cette façon, ils améliorent encore davantage la performance de leur approche, réduisant la complexité pour trouver les mouvements améliorants de $O(k^2 \cdot 2^k)$ à $O(k^3)$ pour un voisinage basé sur une distance de Hamming 1. Ce résultat est enfin généralisé par Chicano et al [10], en décrivant l'algorithme 'Hamming Ball Hill Climber' qui introduit l'identification efficace de mouvements améliorants dans des boules de Hamming de rayon quelconque. Cet algorithme est à la base de l'état de l'art actuel dans la résolution des paysages Mk. Nous décrivons dans la suite cet algorithme, en commençant par la notion essentielle de boule de Hamming.

2.3.2 Boule de Hamming et tailles des voisinages

Commençons d'abord par décrire ce qu'est une *sphère* de Hamming. Étant donnée une solution, il s'agit de l'ensemble des solutions voisines qui diffèrent d'exactement r variables, où r est un paramètre appelé rayon. Il est facile de calculer la taille d'un tel voisinage, i.e., il est égal à $\binom{N}{r}$ le nombre de combinaisons de taille r parmi N. Par extension, on parle de boule de Hamming lorsque l'on s'intéresse à l'ensemble des solutions qui diffèrent d'au plus r variables. Ainsi, une boule de Hamming de rayon $s \leq r$ est inclue dans la boule de rayon r. Une boule de Hamming de rayon r est donc composée de l'union des sphères de Hamming de rayon plus petit ou égal ; soit r, $r-1$, …, 1. On en déduit ainsi que la taille du voisinage induit par une boule de Hamming de rayon r croît exponentiellement avec r.

En pratique, dans un cadre d'optimisation boîte grise et, plus particulièrement, relativement aux paysages Mk, la taille d'un voisinage basé sur les boules de Hamming de rayon $r \geq 2$ peut être drastiquement réduite, et ce, sans impacter la dynamique de la recherche. En effet, revenons sur le graphe d'interaction des variables, le VIG. Pour cela, aidons-nous de l'exemple de la figure 2.2. Comme on peut le voir, il s'agit d'un VIG pour un problème

$$f^0(x_0,x_1,x_2),\ f^1(x_1,x_4,x_2),\ f^2(x_2,x_6,x_4),\ f^3(x_3,x_9,x_0),\ f^4(x_4,x_1,x_5),$$
$$f^5(x_5,x_7,x_1),\ f^6(x_6,x_0,x_2),\ f^7(x_7,x_9,x_0),\ f^8(x_8,x_6,x_4),\ f^9(x_9,x_{14},x_7)$$
$$f^{10}(x_{10},x_4,x_{13}),\ f^{11}(x_{11},x_6,x_8),\ f^{12}(x_{12},x_{10},x_{11}),\ f^{13}(x_{13},x_4,x_5),\ f^{14}(x_{14},x_9,x_3)$$

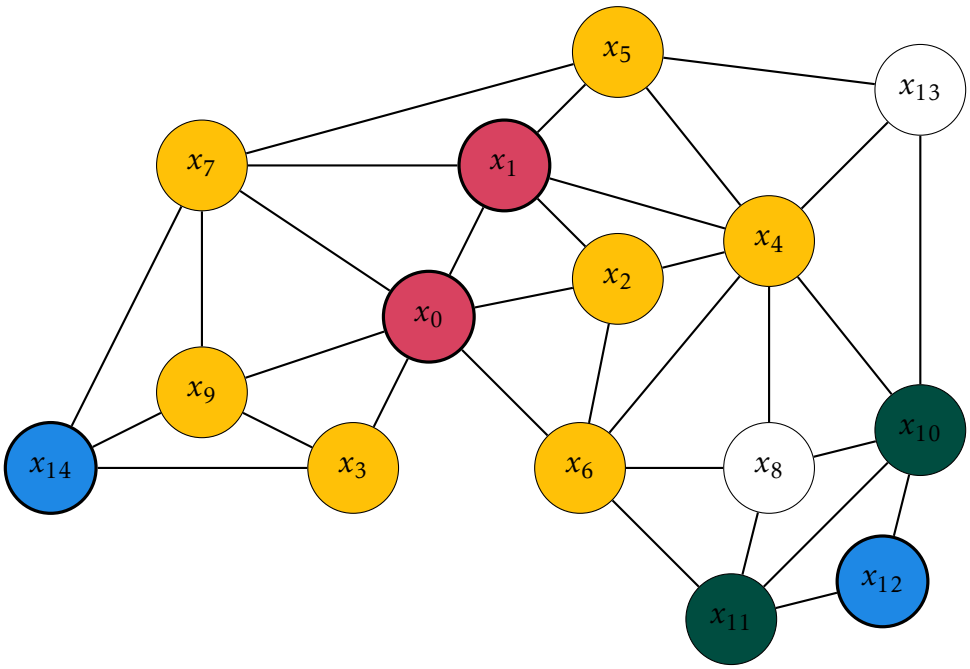

FIGURE 2.2 – *Exemple de mouvements composés de 2 variables à considérer ou non (respectivement en rouge et en bleu) dans une boule de Hamming de rayon 2. En jaune l'ensemble des variables candidates à partir desquelles il peut-être intéressant de former un mouvement de 3 variables en partant du mouvement composé de x_0 et x_1 (en rouge). En vert les variables impactées par le changement de x_{12}*

NK avec 15 variables et 3 variables par sous-fonction. Commençons par considérer le cas d'une boule de rayon 1. Dans ce cas, chaque mouvement est composé d'une seule variable. L'intégralité du voisinage doit être considérée, dans le sens où il faut tester chaque variable et voir si elle améliore ou non la qualité de la solution. Supposons maintenant qu'il n'y ait plus aucun mouvement améliorant dans un voisinage de rayon 1. Testons si le fait de changer 2 variables en même temps peut apporter de nouvelles améliorations.

Prenons par exemples les variables x_{12} et x_{14} (en bleu), ces dernières forment un mouvement dans une boule de Hamming de rayon 2. Cependant, on voit qu'elles ne sont pas connectées dans le VIG. Comme elles n'interagissent pas, leur impact si elles étaient changées en même temps est le même que si elles étaient changées l'une après l'autre. Il est donc inutile de considérer leur combinaison dans une boule de Hamming de rayon 2. Si l'on prend l'exemple de la variable x_{12} on voit que les 2 seuls mouvements qui sont à considérer dans une boule de rayon 2 sont ceux avec x_{11} et x_{10} (en vert). Autrement dit, il

Algorithme 2 : Mise à jour efficace des scores dans une boule de hamming de rayon r

Input : Une solution x, un mouvement T, le tableau des scores S, une boule de Hamming de rayon r M^r

// Pour toutes les sous-fonctions impactées par au moins une variable composant le mouvement T

1 **for** $\forall\, l \in [1 \dots m] \mid (\exists\, t \in T, l \in L^-(t))$ **do**

  // Pour tous les mouvements de la boule de hamming dont au moins une variable dépend de la sous-fonction l

2 **for** $\forall\, S \in M^r \mid (\exists\, v \in V, v \in F^-(l))$ **do**

    // on met à jour le score du mouvement V

3    $S_V \leftarrow S_V + f^l(x \oplus T \oplus V) - f^l(x \oplus T) - f^l(x \oplus V) + f^l(x)$;

4 **end**

5 **end**

n'y a donc, à distance 2, que 2 mouvements contenant x_{12} à considérer, contre 14 dans le cas d'une approche boîte noire naïve.

Un autre exemple concerne le mouvement à rayon 2 $\{x_0, x_1\}$ (en rouge). On voit en jaune l'ensemble des variables avec lesquelles il est possible de construire un mouvement à rayon 3. En règle générale, on peut former un mouvement dans un rayon $s \geq r$ en considérant pour chaque mouvement dans la boule de rayon r, l'union de l'ensemble des variables voisines (dans le VIG) aux variables contenues dans ce mouvement. On voit ici que x_4 et x_5 sont voisines de x_1 ; que x_3, x_6, x_9 sont voisines de x_0 ; tandis que x_2 et x_7 sont voisines à la fois de x_0 et x_1.

En pratique, pour des fonctions k-bornées, il est possible de drastiquement réduire la taille du voisinage à considérer dans un algorithme de type Hill Climber quand on s'intéresse aux mouvements composés de plusieurs variables, en particulier quand $k \ll n$.

Dans le cas de notre exemple, on peut voir que sur les 105 mouvements à deux variables possibles, seulement 32 sont à considérer. En effet, une fois qu'il n'y a plus de mouvements améliorants à une variable, alors les seuls mouvements potentiellement améliorants à deux variables sont ceux composés d'une paire de variables interagissant dans le VIG. Or, ici, on constate que seulement 32 paires remplissent ce critère, permettant ainsi de réduire le nombre de solutions à considérer dans le voisinage à distance au plus 2 d'une solution courante de 120 à 47, avec 15 solutions qui diffèrent d'une variable et 105 solutions qui diffèrent de deux variables, mais seulement 32 potentiellement améliorantes.

2.3.3 Notion de scores et mise à jour efficace

L'efficacité de l'algorithme HBHC 'Hamming Ball Hill Climber' [10] réside dans sa capacité à maintenir à jour une liste de *scores* pour tous les *mouvements* dans une boule de Hamming. Commençons d'abord par définir plus précisément la notion de mouvement et de score. Étant donnée une solution courante, un mouvement dans la boule de Hamming de rayon r est simplement le fait de changer $s \leq r$ variables simultanément. Par abus de langage (et de notation), un mouvement désignera aussi le sous ensemble des variables à changer (à flipper). Le score quant à lui désigne le changement attendu dans la fonction objective (relativement à la solution courante), si un tel mouvement est effectivement réalisé, i.e., si les variables impliquées dans le mouvement sont effectivement changées/flipées. Un score positif indique que le mouvement est améliorant, alors qu'un score négatif indique qu'il est détériorant par rapport à la qualité de la solution courante. L'idée de l'algorithme HBHC réside alors dans le maintien et la mise à jour efficace de la liste des scores des différents mouvements possibles dans une boule de Hamming donnée, comme discuté un peu plus en détail ci-après.

Considérons tous les mouvements possibles dans une boule de Hamming d'un certain rayon r. Il est clair que l'ensemble des mouvements possibles est invariant et ne dépend pas de la solution courante. Ceci est noté M^r dans la suite. L'algorithme commence par une phase d'initialisation où il faut calculer une première fois le score relatif à chaque mouvement possible de M^r, en supposant avoir une solution initiale de départ (typiquement générée de façon aléatoire). Au cours de la recherche locale, le maintien de ces scores peut alors se faire en utilisant une procédure extrêmement efficace telle que décrit de façon générique dans l'Algorithme 2.

L'idée principale est de remarquer que si l'on effectue un mouvement, alors la mise à jour de tous les scores peut se faire de façon incrémentale en prenant en compte les interactions connues entre les variables dans chaque sous-fonction $f^{(l)}$, et ce sans avoir à tout recalculer. En particulier, le score d'un mouvement peut se décomposer comme la somme (linéaire) de *sous-scores* calculés relativement à chaque sous-fonction définissant la fonction objective globale. Par exemple, le score d'un mouvement v, contenant des variables impactant uniquement 2 sous-fonctions, disons l et j, correspond exactement à la somme des 2 sous-scores relativement aux changements de ces deux sous-fonctions $f^{(l)}$ et $f^{(j)}$.

Ainsi, après un mouvement donné T, le sous-score de tout mouvement V, relativement à une sous-fonction l, consiste à calculer les deux sous-scores suivants : $S_V^{(l)}(x \oplus T) = f^{(l)}(x \oplus T \oplus V) - f^{(l)}(x \oplus T)$ et $S_V^{(l)}(x) = f^{(l)}(x \oplus V) - f^{(l)}(x)$; respectivement le sous-score du mouvement V associé à la sous-fonction l appliqué à la solution $x \oplus T$ et celui associé à la solution x. On peut ensuite utiliser ces deux valeurs pour mettre à jour le score S_V relativement à V, en lui soustrayant $S_V^{(l)}(x)$ et en ajoutant $S_V^{(l)}(x \oplus T)$, comme indiqué

à ligne 3. Cette procédure est ensuite itérée par le biais de deux boucles imbriquées : la première parcourt toutes les sous-fonctions impactées par T (ligne 1), et la seconde parcourt tous les mouvements dont le score dépend de cette sous-fonction (ligne 2).

La complexité de cette procédure pour la mise à jour des scores ne dépend à priori *pas* de la taille du problème ; mais exclusivement du mouvement à effectuer T, et du nombre de mouvements possibles dans M^r ayant une interaction avec T. En effet, comme discuté précédemment, un mouvement n'implique en pratique qu'un nombre restreint de mise à jour des scores. En particulier, dans le travail original de Chicano et al [10], il est démontré le théorème 1, qui stipule que la complexité de la mise à jour des scores ne dépend pas de la dimension n du problème. Ceci est particulièrement intéressant lorsqu'on on s'attaque à des problèmes de très grandes dimensions, et explique en grande partie l'extrême efficacité de HBHC.

Théorème 1 (10)**.** *Soit f une fonction définie sur $\mathbb{B}^n$ qui peut se décomposer linéairement sous la forme de somme de sous-fonction $f^{(l)}$, chacune dépendant d'au plus k variables booléennes et chaque variable apparaissant dans au plus c sous-fonctions. Considérons M^r l'ensemble des mouvements $v \in \mathbb{B}^r$ dans la boule de Hamming de rayon r. On a alors :*

- *La cardinalité de M^r est $O((3kc)^r n)$.*
- *Les scores $S_V(x)$ pour $V \in M^r$ peuvent être mise à jour quand on se déplace d'une solution x à une solution $x \oplus T$ en utilisant l'algorithme 2 en un temps $O(b(k)(3kc)^r|T|)$, avec $b(k)$ une borne sur le temps requis pour évaluer une sous-fonction $f^{(l)}$. Ce temps est indépendant de n si k, r et c sont indépendants de n.*
- *Il suffit de vérifier les scores $S_V(x)$ avec $V \in M^r$ de la solution courante x afin de déterminer la présence d'un mouvement améliorant dans la boule de rayon r autour de x. Si les scores sont convenablement stockés en mémoire alors cette étape peut être réalisée en temps constant.*

La mise à jour des scores étant possible de façon efficace, il devient alors possible dans chaque itération d'identifier les mouvements améliorant de façon tout aussi efficace ; ce sont les mouvements qui ont des scores positifs. Nous devons cependant faire particulièrement attention à des considérations techniques d'implémentation, qui ne demeurent pas moins importantes en pratique d'un point de vue de la complexité globale de l'algorithme. En effet, parcourir naïvement la liste des scores est coûteux, car cela dépend de la dimension n et du nombre de mouvements dans M^r. Pour ce faire, nous utilisons dans nos implémentations une structure de données permettant le maintien, en plus des valeurs des scores, de deux ensembles : l'un contenant les mouvements améliorants et l'autre les non-améliorants. Lorsque un mouvement est effectué, en plus de mettre à jour les scores, ces deux ensembles sont mis à jour. Le temps requis pour maintenir à jour ces structures est alors le même que celui nécessaire pour la mise à jour des valeurs de scores. Autrement

dit, la complexité globale suite à un mouvement est toujours la même que celle annoncée lors de la mise à jour des scores dans le théorème précédent.

2.3.4 Algorithme du Hamming Ball Hill Climber

Algorithme 3 : Hamming Ball Hill Climber (HBHC)

Input : Une solution x, un rayon r

1 $M^r \leftarrow$ IntialisationBouleDeHamming (VIG, r);
// Initialisation des scores des mouvements composant M^r
2 $S \leftarrow$ IntialisationDesScores(x, M^r);
// Tant qu'il y a des mouvements améliorants
3 **while** $\exists\, S_v \in M^r \mid S_v > 0$ **do**
// Choisir un mouvement améliorant
4 $T \leftarrow$ SélectionMouvementAméliorant (S);
// Mettre à jour les scores
5 MiseÀJourDesScores(x, T, S);
// Effectuer le mouvement choisi
6 $x \leftarrow x \oplus T$
7 **end**

Nous avons donc maintenant tous les ingrédients constituants l'algorithme 'Hamming Ball Hill Climber'. Rappelons que dans la suite du manuscrit, nous utiliserons l'acronyme HBHC pour faire référence à celui-ci. Étudions en détail son fonctionnement avec le pseudo-code de l'algorithme 3. En entrée, l'algorithme reçoit une solution x initiale et un rayon r. La première étape (ligne 1) consiste à initialiser M^r, la boule de Hamming de rayon r, c'est-à-dire à instancier tous les mouvements à considérer comme nous l'avons présenté plus tôt. Ensuite, le score correspondant à chacun des mouvements composant la boule de Hamming M^r est calculé, puis stocké, relativement à la solution initiale x (ligne 2).

Une fois ces étapes d'initialisation terminées, l'algorithme itère sur le contenu de la boucle While (Tant que) jusqu'à convergence (ligne 3 à 6). Celle-ci consiste à tirer un mouvement dans la liste des améliorants (ligne 4), puis à mettre à jour les scores et les structures de données par le biais de l'algorithme 2. Les mouvements qui étaient améliorants et qui ne le sont désormais plus sont déplacés de l'ensemble des améliorants vers celui des non-améliorants, et inversement. On effectue ensuite le mouvement sur la solution x. Toutes les structures de données de maintien des scores et des listes améliorants/non-améliorants étant à jour, on peut donc recommencer une nouvelle itération. Une fois que

l'ensemble des mouvements améliorants est vide, l'algorithme s'arrête naturellement : la solution x courante est un optimum local.

Concernant les stratégies de sélection des mouvements à effectuer, les auteurs recommandent de considérer en priorité les mouvements améliorants de plus petite taille, c'est-à-dire ceux composés du minimum de variables ; pour ce faire, il nous faut utiliser $2r$ ensembles pour stocker les mouvements. Par exemple, supposons une boule de rayon 2, on a donc 4 listes pour stocker les mouvements :

— ceux améliorants composés d'une variable,
— ceux améliorants composés de 2 variables,
— ceux non-améliorants composés d'une variable,
— ceux non-améliorants composés de 2 variables.

Bien qu'on gardera à jour tous ces ensembles à chaque itération, on privilégiera toujours l'ensemble des mouvements améliorants composés d'une variable avant de passer à la taille supérieure. Suivant les structures de données utilisées, l'algorithme peut-être déterministe ou stochastique : si l'on utilise une pile ou une file, prendre le premier ou le dernier mouvement améliorant rend l'algorithme déterministe (en supposant bien sûr une même solution initiale), tandis que si l'on utilise un ensemble et qu'on tire au sort parmi les mouvements améliorants, l'algorithme sera stochastique.

Les structures utilisées dans HBHC, ainsi que la mise à jour des scores, peuvent être réutilisée dans le cas d'algorithmes plus généraux utilisant cette recherche locale. En effet, si l'on considère, par exemple, l'ajout d'une perturbation, alors chaque changement appliqué à la solution courante peut l'être par le biais de l'algorithme 2 de mise à jour efficace des scores.

2.4 Croisement par partitions

2.4.1 Historique et contexte

Proposé dans un premier temps par Darell Whitley [39] pour le problème du voyageur de commerce, *le croisement par partitions* est un croisement boîte grise entre deux individus. Il a la particularité d'être capable, la majeure partie du temps, d'obtenir de nouveaux optima locaux en partant de deux optima locaux. (Ceci est originalement appelé *tunnélisation* entre optima locaux). Dans le contexte du problème de voyageur de commerce, où une solution est une tournée composée d'arcs, la transmission génétique à la base de ce croisement repose sur deux règles : (i) lorsque un arc est présent dans les deux parents alors les descendants doivent en hériter, et (ii) pour les arcs restants, les descendants sont construits en utilisant uniquement les arêtes héritées de l'un des deux parents. Plus en détails, les arcs communs sont retirés pendant l'étape de croisement de sorte qu'ils ne

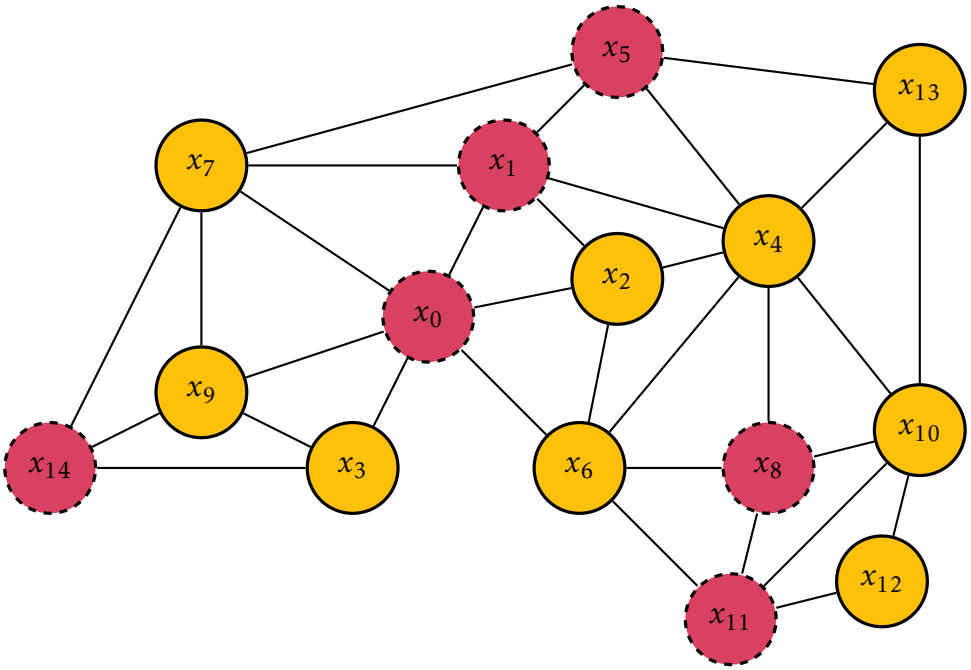

FIGURE 2.3 – *Exemple de partitions (en jaune) obtenues en croisant deux optima locaux. Les partitions sont obtenues en retirant les variables dont l'état est le même dans les deux parents (en rouge et en pointillés), par exemple les solutions* 00000 00000 00000 *et* 00111 01101 10110.

restent que des partitions formant des chemins dans le graphe. Pour chaque partition, les descendants héritent de la configuration de l'un ou l'autre des parents. Ceci aboutit à la création de deux nouvelles solutions valides puisqu'il s'agit bien de circuits, et qui de plus sont la plupart du temps des optima locaux.

Cet opérateur de croisement fut ensuite adapté par R. Tinós et al [36] pour les problèmes pseudo-booléens. Dans le cas du problème de voyageur de commerce, les graphes utilisés pour construire les partitions correspondaient de façon naturelle aux représentations des solutions. Tandis que dans un problème pseudo-booléens, cette approche n'est pas transposable de façon immédiate. Cependant, les auteurs proposent de se baser sur la représentation du problème, plus précisément sur les liens de dépendance entres les variables, afin d'obtenir un graphe qui servira de support au croisement par partition entre deux optima locaux. Ce graphe n'est autre que le graphe d'interaction des variables, le VIG, tel qu'introduit précédemment. Dans la suite, nous discutons de la façon dont ces partitions sont obtenues pour les problèmes pseudo-booléens boîte grise qui nous intéressent.

2.4.2 Partitionnement du VIG pour les problèmes pseudo-booléens k-bornés

Considérons un problème pseudo-booléen k-borné quelconque. Supposons sans perte de généralité un paysage NK, et appuyons nous sur la figure 2.3 comme un exemple illustrant un graphe d'interaction des variables. Nous rappelons qu'un sommet du graphe désigne une variable du problème et que les arêtes modélisent les interactions entre variables telles que données par la définition même du paysage NK. Contrairement au croisement par partitions pour le voyageur de commerce, le graphe est ici inaltérable puisqu'il correspond à la représentation du problème et non pas à la représentation d'une solution. En revanche, comme les nœuds du graphe, sont associés aux variables du problème, leurs *états* dépend lui de la solution. Supposons à cet effet que nous disposons de deux solutions, appelons les parents , que l'on aimerait croiser, par exemple 00000 00000 00000 et 00111 01101 10110 dans la figure 2.3. Chaque sommet/variabe peut alors se retrouver avec les combinaisons d'états suivantes : $\{0,0\},\{0,1\},\{1,0\},\{1,1\}$; la première combinaison indiquant que la variable correspondante vaut 0 pour les deux parents, la deuxième combinaison indiquant que la variable vaut respectivement 0 et 1 dans le premier et le second parent, et ainsi de suite. On remarque que parmi ces quatre configurations possibles d'un sommet du graphe, il y a deux configurations pour lesquelles les parents ont les mêmes valeurs et deux pour lesquelles les valeurs sont différentes. C'est sur cela que les auteurs dans [36] s'appuient pour créer leur partitionnement du graphe.

Plus précisément, l'idée originale consiste à créer un sous-graphe du VIG dans lequel sont retirés tous les sommets dont les variables associées ont une valeur identique dans les deux parents. Par exemple, dans la figure 2.3, les sommets retirés sont affichés en rouge et en pointillés. Les variables restantes (en jaune) forment un sous-graphe composé des variables dont les valeurs diffèrent pour les deux parents. Les partitions correspondent donc aux composantes connexes de ce sous-graphe. Dans la figure 2.3, on a deux partitions : $\{x_2, x_4, x_6, x_{10}, x_{12}, x_{13}\}$ et $\{x_3, x_7, x_9\}$. C'est en se basant sur ces partitions qu'une nouvelle solution va pouvoir être obtenue, tel qu'expliqué dans la section suivante.

2.4.3 Croisement par partitions (PX)

Le but du croisement par partitions, noté PX pour 'Partition Crossover', est de construire la meilleure solution possible en considérant que les variables d'une même composante connexe du graphe ainsi partitionné héritent de toutes les valeurs issues du premier parent, ou alors du deuxième parent. Plus précisément, la solution obtenue avec un croisement par partition est construite de la façon suivante. Pour les variables dont les deux parents partagent le même état, leurs valeurs sont directement héritées, et sont donc les mêmes que pour les deux parents. En ce qui concerne maintenant les autres variables,

il faut considérer chaque composante connexe de façon indépendante. Dans la suite par souci de simplicité, on appellera partition une telle composante connexe.

Étant donné que les partitions ne sont pas reliées entre elles (puisqu'elles forment des composantes connexes) dans le sous-graphe obtenu en éliminant les variables ayant la même valeur dans les deux parents, elles peuvent être considérées comme des (sous-) problèmes d'optimisation indépendants les uns des autres. Pour obtenir une nouvelle solution, on considère alors pour chaque partition la valeur de (sous-)fitness associée à chacun des deux parents. Pour ce faire, il suffit de calculer la somme des contributions pour les sous-fonctions, dont au moins une variable apparaît dans la partition ; et ce, pour chacun des deux parents. Autrement dit, on considère le sous-problème d'optimisation induit par chaque partition de façon indépendante, en se basant sur la formulation boîte grise initiale. On compare ensuite les valeurs de fitness obtenues en héritant des valeurs des variables d'une partition soit complètement du premier parent, soit complètement du deuxième parent. Dans la nouvelle solution ainsi construite, les variables de la partition considérée héritent de la configuration du *meilleur* parent.

À titre d'exemple, considérons de nouveau le cas de la figure 2.3. Pour la première partition ($\{x_2, x_4, x_6, x_{10}, x_{12}, x_{13}\}$), imaginons qu'on obtienne une (sous-)fitness de 86 si on héritait des valeurs des variables du premier parent, et une valeur de 34 si on les héritait du deuxième parent ; et qu'on obtienne respectivement les valeurs 26 et 48 dans le cas de l'autre partition ($\{x_3, x_7, x_9\}$). Alors, la nouvelle solution héritera de l'état de la configuration du premier parent pour les variables $\{x_2, x_4, x_6, x_{10}, x_{12}, x_{13}\}$ et du deuxième pour les variables $\{x_3, x_7, x_9\}$. Pour les variables restantes, comme les deux parents sont égaux, le choix du parent pour l'héritage n'a pas d'importance ; le résultat sera le même que l'on choisisse l'un ou l'autre. Dans notre exemple, le croisement par partitions donnerait alors la solution suivante : 00010 00101 00000. (Rappelons que les solutions croisées sont : 00000 00000 00000 et 00111 01101 10110). Notons que dans le cas où les deux parents auraient la même fitness sur une partition, ce qui peut arriver même si les variables ont des valeurs différentes, alors le choix du parent utilisé pour l'héritage sur cette partition est fait de façon équiprobable. Cela termine la description du croisement par partition [36].

Comme mentionnée précédemment, il est important de noter, que ces règles du choix d'héritages des partitions, basées sur leur qualité, impliquent que la nouvelle solution ainsi construite est au moins aussi bonne que le meilleur des deux parents. Il est, en effet, très facile de s'en rendre compte, puisque les partitions sont indépendantes et que donc leurs contributions à la fitness globale l'est aussi. En choisissant la configuration ayant la meilleure contribution de fitness dans chaque partition, on a la garantie d'obtenir une solution aussi bonne, sinon meilleur, que le meilleur des deux parents. Le pire cas étant de retomber sur le premier ou le deuxième parent ; ce qui peut se produire si toutes les

partitions sont systématiquement meilleur avec la configuration de l'un ou de l'autre des deux parents.

Du point de vue de la complexité de cet opérateur de croisement, il est tout aussi facile de voir qu'il a une complexité relativement très attractive par rapport à sa capacité d'explorer l'espace des enfants possibles. En effet, soit q le nombre de partitions obtenues relativement à deux parents données en entrée. D'un côté, la solution résultante du croisement est garantie d'être la meilleure parmi les $2^q - 2$ descendants possible. Comme la valeur de q peut en pratique être relativement élevée, le croisement par partition possède la capacité de déterminer le meilleur enfant possible parmi un nombre éventuellement exponentiel de possibilités. D'un autre côté, cela ne demande qu'un temps linéaire en la taille du graphe d'interaction.

En outre, ce croisement par partitions possède une autre propriété puissante. En effet, les solutions obtenues par croisement par partitions sont très souvent des optima locaux. Il est par exemple montré expérimentalement dans le papier introduisant ce croisement [36] que pour des paysages NK aléatoires, cela est vrai dans 83% des cas expérimentés. Pour se convaincre que ce n'est pas systématiquement le cas, et si l'on reprend notre exemple avec la figure 2.3, alors on peut voir que les variables qui connectent les deux partitions dans le VIG peuvent être changées pour entraîner des améliorations supplémentaires après le croisement, soit ici les variables $\{x_0, x_1, x_5\}$. En se basant sur ce type d'observation, des versions plus avancées de ce croisement par partition ont été proposées depuis. Les deux sous-sections suivantes leurs sont consacrées.

2.4.4 Croisement avec analyse des points d'articulation (APX)

La première variante du croisement par partitions, appelée APX pour 'Articulation Point Crossover', a été proposée par Chicano et al [13]. Elle introduit l'étude des points d'articulation des partitions afin de tenter de subdiviser ces dernières en de plus petites sous-partitions. Pour rappel, un point d'articulation d'un graphe est un sommet du graphe qui, lorsqu'il est retiré, augmente le nombre de composantes connexes. En d'autres termes, son retrait transforme un graphe connexe en un graphe non-connexe. Ainsi, si le graphe en question est une partition telle que défini dans la sous-section précédente, on voit qu'un point d'articulation permet de diviser cette partition en de nouvelles composantes connexes de plus petite tailles. Ainsi, cela augmente fortement le nombre de solutions explorées lors d'un croisement, et ainsi le pouvoir d'exploration et d'exploitation de l'opérateur, puisque seul le meilleur enfant parmi toutes les combinaisons possibles est retenu au final. Supposons que sur q partitions initiales, seulement l'une d'entre elle possède un point d'articulation qui divise sa partition en 2, alors la meilleure solution retournée ne sera plus la meilleure parmi les $2^q - 2$ descendants possibles, mais la meilleure parmi $2^{q+1} - 2$. On constate donc que ce nombre augmente en fonction du nombre de

partitions ayant un point d'articulation et du nombre de composantes connexes crées par le retrait de ces derniers. Nous développons davantage ce concept dans ce qui suit sans pour autant expliciter tous les détails par soucis de concision et de clarté. Le lecteur est invité à se référer au papier original pour une description complète [13].

L'algorithme du croisement par partitions avec analyse des points d'articulation est composé de deux phases de complexité linéaire. Il s'agit dans un premier temps de parcourir le graphe à l'aide de l'algorithme de Tarjan [34] pour identifier les points d'articulation. Ce dernier s'exécute en un temps linéaire en la taille du graphe. Une fois les points d'articulation trouvés, il faut les parcourir. Pour chacun d'entre eux, il faut calculer les contributions de fitness associées à leur retrait. La complexité de cette étape dépend, pour chaque point d'articulation, du nombre de composantes connexes créées lors de son retrait du graphe.

Dans [13], les auteurs ont décidé de limiter leur étude au cas où un seul point d'articulation par partition serait considéré pour effectuer la recombinaison. Ainsi, ils considèrent un critère glouton où le point d'articulation avec le meilleur potentiel d'amélioration de la solution courante est retenu ; en cas d'égalité, le choix est fait aléatoirement.

Afin de mieux comprendre les implications pratique de cela, basons nous sur la figure 2.4 et étudions plus en détail le fonctionnement de cet opérateur de croisement. Dans cette figure, on peut voir deux partitions : celle composée des variables $\{x_3, x_7, x_9\}$ que nous nommerons Π^1, et celle composée des variables $\{x_2, x_4, x_6 x_{10}, x_{12}, x_{13}\}$ que nous nommerons Π^2. Dans Π^1, on peut voir qu'il n'y a qu'un seul point d'articulation, la variable x_9, nommons le AP_1^1. Celui-ci divise la partition en deux composantes $C^1_{(1,1)}$ et $C^1_{(1,2)}$, toutes deux composées d'une seule variable, respectivement x_3 et x_7. De même que la partition Π^2 est composée de 2 points d'articulation $AP_1^2 = x_4$ et $AP_2^2 = x_{10}$; les nouvelles composantes créées par retrait des points d'articulation sont notés $C^2_{(1,1)} = \{x_2, x_6\}$ et $C^2_{(1,2)} = \{x_{10}, x_{12}, x_{13}\}$ pour AP_1^2 et $C^2_{(2,1)} = \{x_2, x_4, x_6, x_{13}\}$ et $C^2_{(2,2)} = \{x_{12}\}$ pour AP_2^2. Ces différents points d'articulation et leurs nouvelles composantes respectives sont découverts au fur et à mesure que l'algorithme de Tarjan parcourt le graphe. Pour chaque nouvelle composante, la contribution de la fitness de celle-ci, relativement à la fonction objective initiale, est calculée et stockée pour les 4 cas suivants :

— La composante hérite de P_1 en sachant que le point d'articulation hérite de P_1 ;
— La composante hérite de P_2 en sachant que le point d'articulation hérite de P_1 ;
— La composante hérite de P_1 en sachant que le point d'articulation hérite de P_2 ;
— La composante hérite de P_2 en sachant que le point d'articulation hérite de P_2.

Une fois les points d'articulation trouvés et donc, de fait, leurs composantes associées et leurs contributions de fitness, il faut appliquer une étape supplémentaire. Elle consiste en une technique de programmation dynamique sur l'arbre de décision permettant d'hériter les valeurs des variables dans les différentes composantes. Ceci est maintenant illustré à

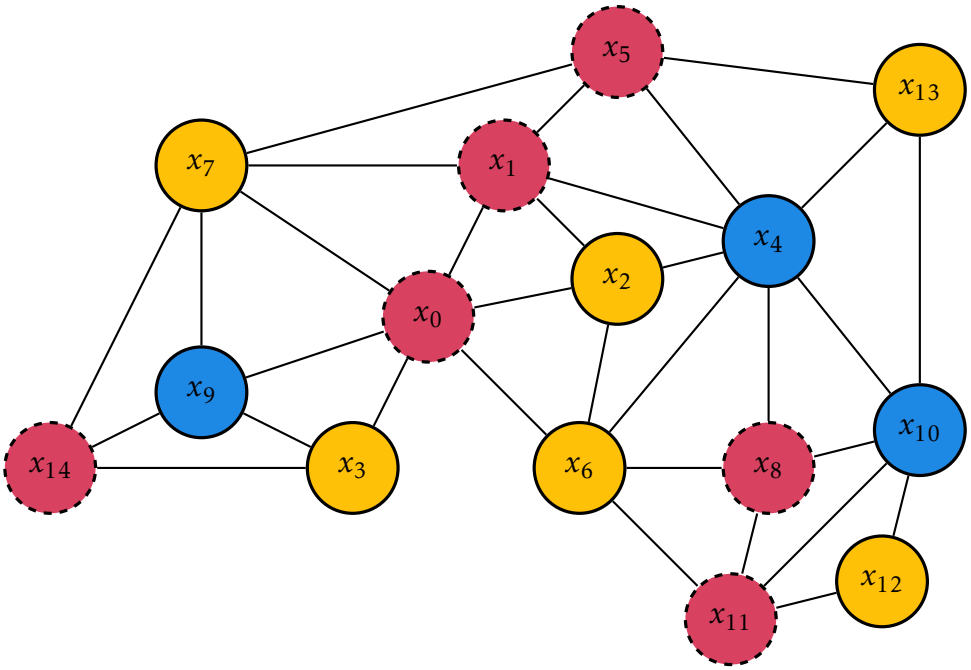

FIGURE 2.4 – *Exemple de points d'articulation (en bleu) sur des partitions (en jaune + bleu) obtenues en croisant deux optima locaux. Les partitions sont obtenues en retirant les variables dont l'état est le même dans les deux parents (en rouge et en pointillés), par exemple les solutions* $P_1 = 00000\ 00000\ 00000$ *et* $P_2 = 00111\ 01101\ 10110$.

l'aide de la figure 2.5. L'idée générale est que comme les données des composantes sont déjà stockées, il suffit de remonter dans un arbre de décision et de faire le choix qui maximise la fitness finale ; en cas d'égalité, un tirage équiprobable a lieu. Autrement dit, du fait que l'on se limite à un seul point d'articulation et que les partitions sont indépendantes, tout comme les composantes créées, alors on a la garantie que l'algorithme complet revient à créer un arbre de décision et à lui appliquer une programmation dynamique dont la profondeur en termes d'affectation et de décision est au plus de 3 pour chaque partition :

1. Dans un premier temps, le choix du parent dont hérite chaque composante en fonction de l'état du point d'articulation ;
2. Le choix du parent dont hérite le point d'articulation en prenant en compte les contributions de fitness des différentes combinaisons d'états des composantes qui lui sont associées ;
3. Et pour finir, le choix du point d'articulation qui maximise la valeur de la fonction objective.

Dans le cas de notre exemple dans la figure 2.5, on voit que la meilleure fitness pour la partition Π^2 est obtenue en considérant le point d'articulation AP_1^2, soit une amélioration de 59, et ce, en héritant de la solution P_2 pour le point d'articulation et la première composante, tandis que la seconde composante hérite de P_1.

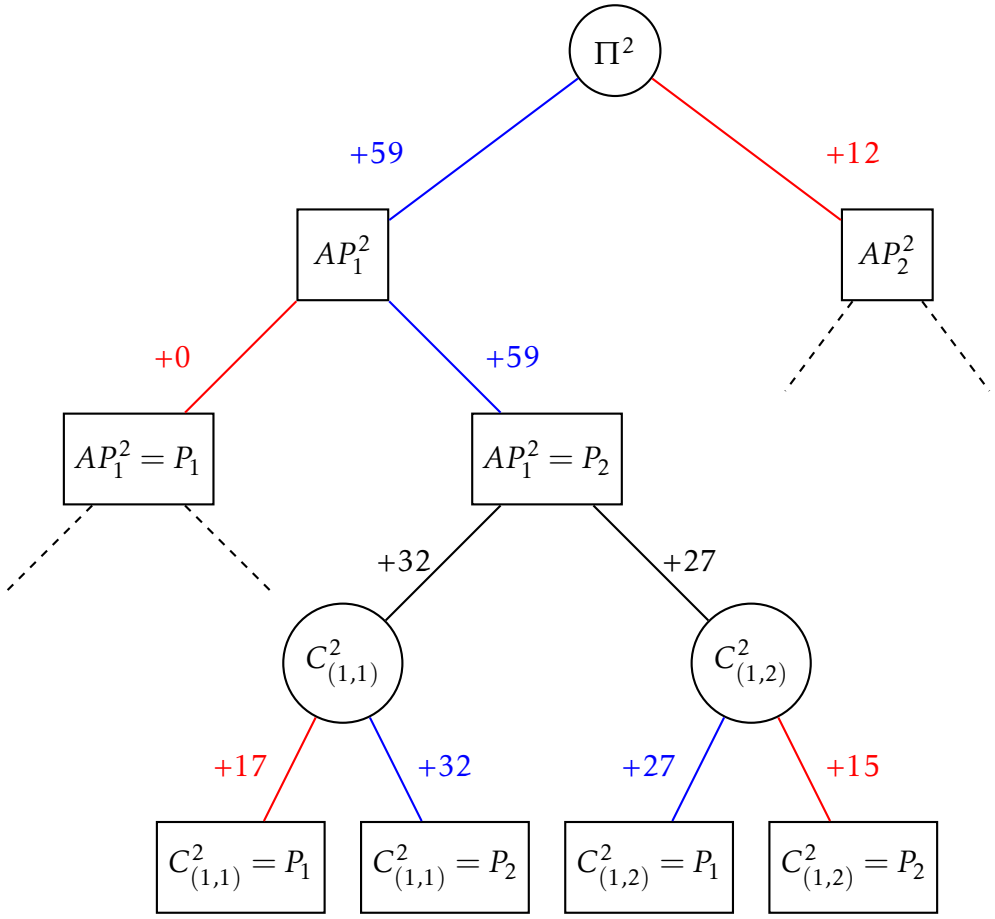

FIGURE 2.5 – *Arbre de décision partiel associé à l'opérateur de croisement par partitions avec étude des points d'articulation appliqué au cas présenté dans la figure 2.4, et plus précisément à la partition* $\Pi^2 = \{x_2, x_4, x_6, x_{10}, x_{12}, x_{13}\}$. *Les rectangles représentent les décisions à prendre, c'est-à-dire des comparaisons dans le but de maximiser la fitness. Les données associées aux feuilles de l'arbre sont calculées et stockées lors de l'exécution de l'algorithme de Tarjan pour trouver les points d'articulations, tandis que les décisions supérieures sont effectuées dans la deuxième phase, celle de l'analyse des points d'articulations.*

Pour conclure, il est important de remarquer que, si l'on s'en tient à l'opérateur de croisement sans prendre en compte la dynamique de recherche induite lorsque intégrée dans un algorithme de recherche heuristique plus avancée, alors le croisement par partitions avec analyse des points d'articulation ne peut qu'être meilleur (et au pire égal) au croisement par partitions classique, en terme de la qualité de l'enfant qu'il est capable de construire à partir de deux parents. En effet, sans perte de généralité, considérons le cas d'une partition unique. Dans le cas où il n'y a pas de point d'articulation alors il est évident que le résultat sera le même pour les deux croisements, la partition hérite pleinement de l'un ou de l'autre parent. Maintenant, considérons le cas où il y a au moins 1 point d'articulation, alors si celui-ci n'apporte pas une amélioration, on retombera dans le pire des cas également sur l'état de l'un des deux parents ou bien sûr une combinaison des deux qui aboutissent à la même fitness que le croisement par partition ; et ce en raison de la technique de programmation dynamique qui permet de maximiser la fitness finale de l'enfant.

2.4.5 Croisement par partitions avec potentiel dynastique (DPX)

Récemment, en 2022, Chicano et al [14] ont proposé un nouveau croisement qui pousse encore plus loin l'utilisation des partitions en proposant de les explorer de façon plus exhaustive en utilisant une technique de programmation dynamique, et ce, sans se limiter aux points d'articulation. Cet opérateur est appelé DPX, en référence à son potentiel dynastique de recherche ('Dynastic Potential Crossover'). L'opérateur DPX se base sur une panoplie de techniques incluant une décomposition en arbre de cliques et plusieurs algorithmes issue de la théorie des graphes appliqués au graphe d'interaction des variables. Expliquer de façon détaillée le fonctionnement de ce nouveau type de croisement est en dehors du périmètre de ce manuscrit. Cependant, par soucis de complétude, nous allons expliquer son principe de fonctionnement global. En particulier, nous donnons la procédure générale utilisée pour construire un enfant, et expliquons en quoi elle est théoriquement supérieure en termes de qualité de la solution enfant trouvée comparée aux versions PX et APX.

Chicano et al[14] s'inspirent précisément de techniques antérieures utilisées dans la création des arbres de jonctions pour les réseaux bayésiens [5]. Supposons une partition comme expliqué auparavant pour les autres types de croisement existants. L'algorithme commence par utiliser une recherche de cardinalité maximale ('maximum cardinality search') suivie d'une procédure de remplissage afin d'avoir un graphe cordal [35]. Cette étape étant en soit destinée à résoudre un problème NP-Difficile, la procédure utilisée ne garantit pas que le graphe cordal soit de taille minimale, mais elle s'exécute en temps linéaire. Une fois ce graphe cordal obtenu, un arbre de clique est créé afin de donner un ordre d'élimination aux variables pour la phase de programmation dynamique. Cette phase

Algorithme 4 : Pseudo-code de la procédure de croisement par partitions avec potentiel dynastique (DPX)

Input : Deux optima locaux x et y

1 Construction des partitions
2 Recherche de cardinalité maximale
3 Procédure de remplissage pour rendre le graphe cordal
4 Construction de l'arbre de cliques
5 Assignation des sous fonctions aux cliques
6 Séparation des variables des cliques en deux ensembles en fonction de β (entièrement exploré ou non)
7 Programmation dynamique et remplissage des tables
8 Construction de la nouvelle solution à partir des données des tables

est suivie d'une assignation des sous-fonctions aux différentes cliques, avant que l'étape de programmation dynamique puisse à proprement dit commencer. La meilleure solution est ainsi construite à partir des données issues de la programmation dynamique, la procédure complète étant résumée de façon très générale dans le pseudo-code de l'algorithme 4.

La figure 2.6 montre un exemple de la méthode de construction d'un arbre de cliques appliqué à une partition (dans ce cas, le graphe est déjà cordal). Dans cet exemple, la clique de plus grande taille ne comporte que 3 variables. Cependant, en pratique, ce nombre peut croître très vite de sorte que le nombre de solutions à explorer durant la phase de programmation dynamique devienne rapidement rédhibitoire. Pour contrer cela, les auteurs proposent d'introduire un paramètre servant à limiter l'exploration des cliques de trop grandes tailles. Ce paramètre nommé β divise chaque clique en deux parties, les β variables dont toutes les combinaisons vont être explorées durant la phase de programmation dynamique, et les autres qui hériteront en groupe de l'un ou l'autre des parents. Ainsi, DPX permet d'augmenter le nombre potentiel d'enfants pouvant être construit en préservant une partie du matériel génétique des deux parents, d'où le nom 'Dynastic Potential Crossover'. Le lecteur est de nouveau invité à consulter le détail de l'algorithme de programmation dynamique utilisé dans l'article original [14].

L'ajout de la phase de programmation dynamique sur l'arbre des cliques permet à DPX de surpasser, ou au pire d'égaler, PX et APX dans le cas où le nombre de variables explorées ne serait pas limité. En pratique, même pour $\beta = 0$, PX ne peut pas être meilleur. Cependant, APX peut être meilleur et ce même pour des valeurs de β plus grandes. Cela est dû au fait que les points d'articulation peuvent être explorés de façon différente entre les deux croisements. Pour minimiser ce comportement, les auteurs recommandent de privilégier l'exploration des points d'articulation parmi les β variables à explorer, ce qui rajoute une complexité supplémentaire dans la conception de cet opérateur.

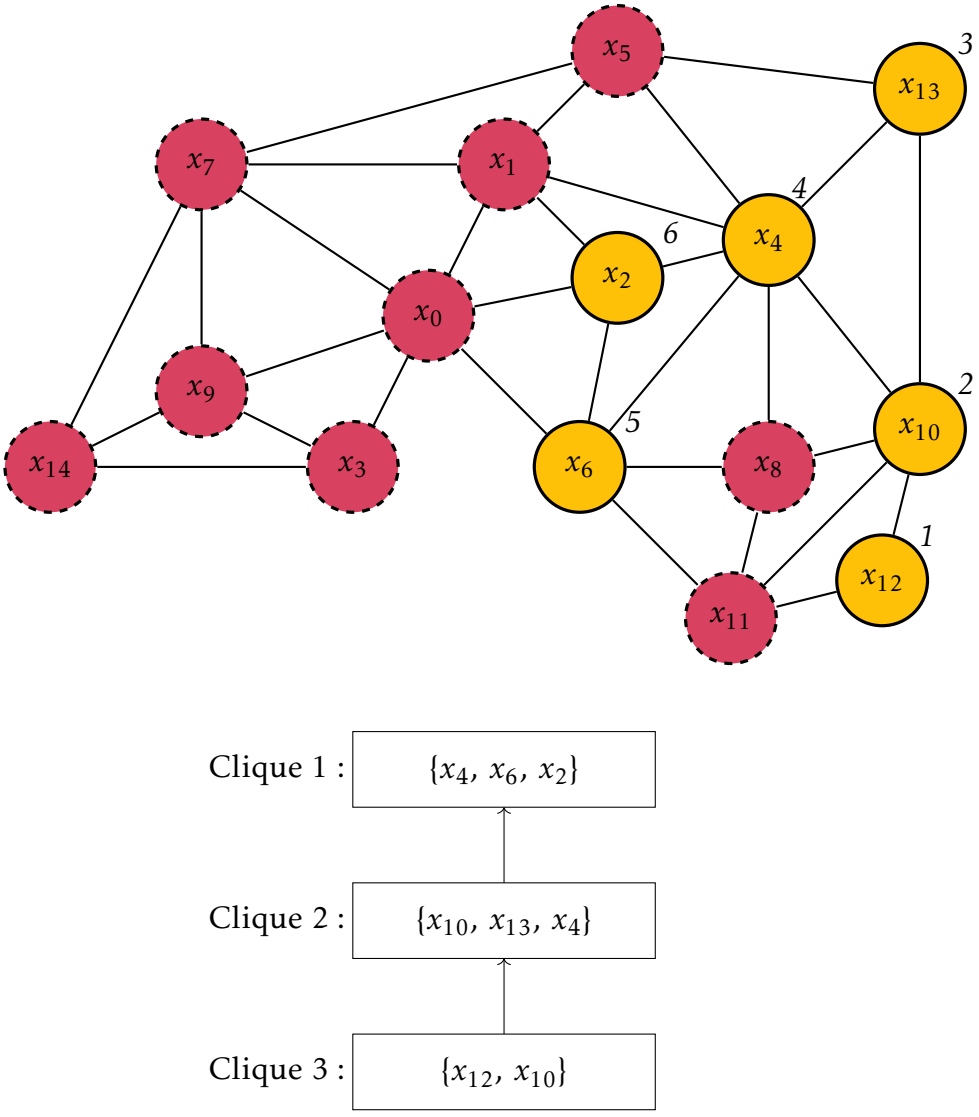

FIGURE 2.6 – *Exemple d'arbre de cliques appliqué sur une partition (en jaune) obtenue en croisant deux optima locaux. Les partitions sont obtenues en retirant les variables dont l'état est le même dans les deux parents (en rouge et en pointillés), par exemple les solutions* 00000 00000 00000 *et* 00101 01000 10110. *L'ordre de visite des variables lors d'une recherche de cardinalité maximum commençant par la variable x_2 est indiqué en italique et en haut à droite de chaque variable de la partition*

Concernant les aspects liés à la complexité et au temps d'exécution en pratique, DPX est sensiblement plus long à exécuter que PX et APX ; ces deux derniers étant plus ou moins identiques en termes de vitesse d'exécution. De même, la quantité de mémoire nécessaire pour DPX est beaucoup plus grande que pour PX et APX. Ces deux problèmes font qu'en pratique DPX est difficilement utilisable pour des problèmes de très grandes tailles et/ou avec un budget/temps de calcul réduit, comme nous le verrons plus loin dans ce manuscrit avec des paysages NK ayant 1 million de variables, et des temps d'exécution de l'ordre de quelques secondes.

Maintenant, que nous avons présenté les croisements les plus récents pour les problèmes pseudo-booléens, nous allons nous pencher sur leur intégration concrète dans des algorithmes de recherche plus avancés, tels qu'ils ont été initialement décrits dans la littérature. Dans le reste de ce chapitre, nous présenterons en effet deux algorithmes pouvant intégrer ces différents croisements, dont un qui représente l'état de l'art pour les problèmes NK de très grandes tailles.

2.5 Hybridation entre recherche locale et croisement boîte grise

2.5.1 Contexte

La recherche sur les opérateurs de croisement est étroitement liée à l'étude des algorithmes génétiques, John Holland en est l'un des pionniers [19]. Le fonctionnement de base des algorithmes génétiques s'inspire fortement de la théorie de l'évolution : le principe est de faire évoluer une population d'individus (des solutions au problème d'optimisation) en itérant sur plusieurs générations les étapes suivantes :

1. Évaluation : les solutions se voient attribuer une fitness par le biais de la fonction objective.
2. Sélection : cette étape consiste à sélectionner des individus pour les utiliser soit lors du croisement, soit pour les conserver dans la génération suivante ; il existe une multitude de stratégie de sélection différente dans la littérature [1, 17].
3. Reproduction (croisement, et mutation) : une fois un groupe d'individus sélectionné, ces derniers sont croisés entre eux et/ou mutés, là encore, il existe de nombreuses stratégies. On peut citer parmi les croisements génériques les plus connus les opérateurs suivants :
   — le croisement uniforme [32] : chaque gène est hérité aléatoirement d'un des deux parents avec un tirage équiprobable ou bien pondéré en fonction de l'écart de fitness.
   — le croisement multi-points : plusieurs points, dont le nombre est spécifié initialement par le concepteur, sont tirés aléatoirement parmi l'ensemble des gènes, permettant ainsi de les découper en plusieurs portions. Les portions de gènes sont ensuite échangées entre les deux parents pour former des nouvelles solutions enfants.

C'est cette classe d'algorithme qui a été naturellement adopté dans l'article initial introduisant l'algorithme de croisement par partitions (PX) [36]. Cependant, le temps de calcul nécessaire à l'évolution d'une population peut vite devenir conséquent. De même,

la performance de l'opérateur de croisement est très fortement corrélée à la distance de Hamming entre les deux individus croisés. Il est donc difficile de maintenir un intervalle de distance optimale entre individus au sein d'une population. C'est dans ce contexte que F. Chicano et al proposent deux nouveaux algorithmes [12] reposants sur l'utilisation des opérateurs de croisements boîtes grises pour les problèmes pseudo-booléens. Dans les deux cas, ces croisements sont associés au sein des algorithmes proposés à l'utilisation du 'Hamming Ball Hill Climber' HBHC avec un rayon 1 en guise de recherche locale.

2.5.2 HiReLS

Le premier algorithme proposé par F. Chicano et al [12] est un algorithme à base de population. Il se nomme HiReLS, acronyme de 'Hierarchical Recombination Local Search'. Expliquons en détail le fonctionnement correspondant au pseudocode 5. La population de cet algorithme est maintenue sous la forme d'une pyramide inversée. On commence par générer des paires de solutions aléatoires suivies de recherche locale pour obtenir des optima locaux. Ensuite, ces paires d'optima locaux, formant le niveau initial de la pyramide, sont combinées à l'aide du croisement par partition ; nous pouvons ensuite appliquer la recherche locale aux solutions résultantes. Nous obtenons ainsi des solutions qui sont des optima locaux et dont la valeur de fitness n'est pas inférieure à celle des parents. Appelons optima locaux de niveau 1 ces solutions, et optima locaux de niveau 0 les solutions initiales générées après avoir appliqué la recherche locale à des solutions aléatoires. La valeur moyenne de fitness des optima locaux de niveau 1 est supérieur ou au pire égale à celle des optima locaux de niveau 0. Nous pouvons obtenir des optima locaux de niveau 2 en recombinant deux optima locaux de niveau 1 à l'aide du croisement par partitions, suivi d'une recherche locale. La fitness moyenne des solutions au niveau $i+1$ étant supérieur à celle du niveau i, la combinaison d'optima locaux de niveau 1 devrait fournir de meilleures solutions, en général, que la combinaison d'optima locaux de niveau 1 et de niveau 0. Cette idée peut être appliquée de manière itérative pour trouver des optima locaux à différents niveaux avec des valeurs moyennes de fitness croissantes.

On voit ainsi que la progression dans la pyramide repose sur le succès des croisements de paires d'individus de niveau inférieur. Par exemple, pour obtenir une solution de niveau 2, il est nécessaire de réussir un croisement entre deux individus de niveau 1, cela ayant nécessité le croisement de 2 paires d'individus de niveau 0, etc.

2.5.3 DRILS

Le second algorithme proposé par F. Chicano et al [12] est, contrairement à HiReLS, un algorithme hybride à base de trajectoire s'inspirant d'une recherche locale itérée combinée

Algorithme 5 : HiReLS :

```
   // On commence avec une pile vide
1  stack ← ∅ ;
2  repeat
       // On crée un nouvel optimum local
3      current ← HBHC(random()) ;
       // On ajoute cet individu au premier niveau de la pyramide
4      current.level ← 0 ;
       // Si la pile est vide ou bien si le niveau dans la pyramide de la dernière
          solution de la pile est supérieur à 0
5      if stack.isEmpty() or stack.peek().level > 0 then
           // On ajoute la solution courante à la pile
6          stack.push(current);
7      else
           // Si la condition précédente n'est pas remplie, cela indique que l'on peut
              faire des croisements et progresser dans la pyramide
8          pxSuccess ← true ;
           // Tant qu'il y a des croisements possible ; c'est-à-dire si la pile n'est
              pas vide, qu'aucun croisement réalisé au cours de cette itération n'a
              échoué et qu'il y a deux solutions de même niveau à croiser ; alors on
              effectue les opérations suivantes
9          repeat
               // On dépile top
10             top ← stack.pop() ;
               // On croise la solution courante et top pour obtenir child
11             child ←PX(top,current) ;
12             pxSuccess ← child ≠ top and child ≠ current ;
               // Si le croisement réussi, alors on utilise une recherche locale sur
                  child, puis on copie le résultat dans la solution courante, et on
                  fait progresser cette solution au niveau supérieur
13             if pxSuccess then
14                 current ← HBHC(child) ;
15                 current.level + + ;
16             end
17         until !stack.isEmpty() and pxSuccess and stack.peek().level = current.level;
           // S'il n'y a pas eu de croisement, ou que tous les croisements effectués
              ont réussi et qu'il n'y a plus de croisement possible
18         if pxSuccess then
               // Alors on ajoute la solution courante à la pile
19             stack.push(current) ;
20         end
21     end
22 until not stopping condition;
```

Algorithme 6 : DRILS

```
1  current ← HBHC(random());
2  repeat
3  |  next ← HBHC(perturb(current));
4  |  child ← PX(current, next);
5  |  if child = current or child = next then
6  |  |  current ← next;
7  |  end
8  |  else
9  |  |  current ← HBHC(child);
10 |  end
11 until not stopping condition;
```

à un opérateur de croisement, d'où son nom DRILS, pour 'Deterministic Recombination and Iterated Local Search' (recherche locale itérée et recombinaison déterministe).

Il est à noter que bien que son nom suggère le contraire, le croisement n'est pas nécessairement déterministe mais plutôt stochastique dans le cas où deux partitions seraient de même qualité lors du croisement. Le déterminisme dans le nom de l'algorithme fait ici référence à la capacité du croisement à trouver le meilleur enfant parmi éventuellement un large nombre de possibilités de façon certaine. Comme pour toute recherche locale itérée, DRILS repose sur un hyperparamètre α qui correspond à l'intensité de la perturbation. La boucle principale de l'algorithme consiste à générer un nouvel optimum local appelé *next* dans le pseudo-code de l'Algorithme 6. Ce dernier est obtenu en perturbant la solution courante, et en lui appliquant ensuite une recherche locale. Les deux solutions *current* et *next* sont ensuite croisées pour obtenir une nouvelle solution *child*. Ici, comme l'opérateur de croisement supposé est de type PX, soit on retombe sur l'un des deux parents auquel cas on continue la recherche avec *next*, ce qui sert de mécanisme d'échappement ; soit on applique de nouveau une recherche locale à la solution *child* si il n'est éventuellement pas l'optimum local. On continue ainsi de façon itérative jusqu'à ce qu'un critère d'arrêt soit satisfait. Il est à noter que bien que DRILS intègre initialement le croisement par partitions PX, sa conception n'en ait en aucun cas dépendante, et d'autres types de croisement pourraient être utilisés.

2.5.4 Discussion

Les performances des deux algorithmes, DRILS et HiReLS, dépendent des caractéristiques des paysages NK sur lesquelles ils sont exécutés. En particulier, il a été montré que HiReLS donne de bons résultats pour les paysages NK dits adjacents. Cependant, il est bien

connu que ces derniers ne sont pas difficiles, et peuvent être résolus en temps polynomial en utilisant des techniques de programmation dynamique, raison pour laquelle nous ne considérons d'ailleurs que les paysages aléatoires dans ce manuscrit ; paysages connus comme NP-difficiles lorsque les sous-fonctions sont composées d'au moins 3 variables. En considérant des paysages aléatoires avec $K \geq 2$, DRILS est montré très clairement supérieur à HiReLS peu importe la taille et la rugosité des paysages. Notons que la performance de DRILS est très fortement liée à la valeur du paramètre de perturbation α, comme cela est le cas dans toute recherche locale itérée de ce type. Dans la suite de ce document, nous allons nous intéresser à améliorer DRILS, considérée par la communauté comme étant l'état de l'art pour les paysages Mk au moment où cette thèse a commencé.

2.6 Conclusion

Dans ce chapitre, nous avons introduit les principaux composants nécessaires à la conception d'algorithmes pour l'optimisation boîte grise de problèmes pseudo-booléens k-bornées. Nous avons décrit en détail le principe d'une recherche locale basique HBHC utilisant le graphe d'interaction (VIG) et nous avons discuté de divers opérateurs de croisement à base de partitions. Enfin, nous avons décrit l'intégration et l'hybridation de ces opérateurs de recherche locale et de croisement dans des algorithmes plus évolués. En particulier, étant donné les performances de DRILS sur les paysages Mk et NK aléatoires, celui-ci sera le principal sujet d'étude de ce manuscrit.

Dans la suite, le document s'organise en 2 parties. La première partie concernera la conception d'approches parallèles pour l'optimisation boîte grise. Dans cette partie, le premier chapitre est consacré à l'étude de la parallélisation de l'algorithme HBHC dans un environnement en mémoire partagée. Le deuxième chapitre se penche ensuite sur la parallélisation massive de DRILS et plus particulièrement sur les approches coopératives et distribuées, influencées par les travaux sur les modèles en îles. Ce chapitre est complété par une étude expérimentale sur le Fugaku, l'un des plus puissants supercalculateurs au monde. Cette dernière contribution peut-être vue comme un framework dans lequel DRILS n'est qu'un composant qui peut à priori être remplacé par une variante fonctionnant sur le même principe.

La seconde partie du reste du document se concentre quant à elle sur la compréhension de la dynamique de recherche induite par DRILS, et des améliorations que l'on peut y apporter dans un environnement de calcul séquentiel. Comme nous le verrons dans le second chapitre de la première partie, le framework massivement parallèle proposé repose sur l'utilisation de DRILS considéré comme un seul bloc s'exécutant de façon séquentielle, mais coopérativement à d'autres instances de DRILS. Ainsi, les améliorations apportées à l'algorithme séquentiel de DRILS pourront dans le futur bénéficier également aux ap-

proches parallèles. Cette partie se divise donc en trois chapitres. Le premier chapitre étudie les mécanismes d'échappements de DRILS en introduisant deux variantes DRILS+ et DRILS++ qui proposent respectivement une stratégie d'échappement rétroactive et proactive. Cette partie est ensuite complétée d'expérimentations sur un grand nombre de paysages NK de rugosités et de tailles différentes. Le deuxième chapitre concerne l'utilisation d'une phase d'initialisation précédant DRILS, et sur l'étude du biais d'intensification introduit par l'usage de plusieurs croisements successifs sur une même solution de départ. Enfin, nous concluons nos investigations avec un chapitre traitant du comportement des différentes variantes de DRILS proposées, et de leurs combinaisons avec les opérateurs de croisement par partitions les plus récents, et ce, sur un large panel d'instances de tailles et de rugosités différentes. Nous nous concentrerons également sur la dynamique de recherche de DRILS et en particulier sur l'impact des itérations successives de croisements réussies sur les itérations suivantes.

Deuxième partie

Conception et étude d’algorithmes boîte grise parallèles

Chapitre 3

Hill Climber parallèle basée sur la coloration de graphe

3.1 Motivation

Comme discuté dans le chapitre précédent, une grande partie de l'efficacité des algorithmes état de l'art boîte grise repose sur la conception de recherches locales très efficaces. Ces dernières utilisent les informations disponibles sur la structure du problème afin d'améliorer grandement leur vitesse d'exécution. Considérons l'algorithme HBHC décrit dans la section 2.3. Cet algorithme s'appuie sur le graphe d'interaction des variables (VIG) afin de trouver un mouvement améliorant en temps constant. L'idée sous-jacente est que si l'on utilise une structure contenant le score associé au changement de chaque variable, alors il est possible lorsque l'on change l'une d'entre elles de conserver cette structure à jour de façon efficace. Pour ce faire, on a uniquement besoin de modifier les scores des variables impactées, c'est-à-dire celles qui sont adjacentes à la variable modifiée dans le VIG. Pour chaque variable impactée, il suffira alors de recalculer le score relativement aux sous-fonctions qui dépendent à la fois de celle-ci et de la variable modifiée.

Cette approche est très efficace peu importe la taille et la rugosité du problème. Cependant, une recherche locale peut nécessiter beaucoup de mouvements avant de converger vers un optimum local. Ainsi, il est intéressant d'étudier la possibilité de pouvoir effectuer *plusieurs* mouvements *en parallèle*, et ce, afin d'accélérer le temps d'exécution de la recherche locale. Malheureusement, la dynamique et les structures de données sur lesquelles reposent HBHC ne sont utilisables en tant que tels que dans un environnement de calcul séquentiel. Il faut donc penser à une approche différente dans un contexte de calcul parallèle. Le principal défi pour une parallélisation efficace en mémoire partagée d'un tel algorithme est la gestion des accès concurrents ; c'est-à-dire s'assurer que deux threads n'essayent pas de modifier la même donnée en même temps. Il faut également s'assurer que le résultat en termes de modifications des valeurs de variables soit cohérent.

En effet, imaginons que deux variables (ou plus généralement deux mouvements) soient toutes deux améliorantes. Ceci est seulement vrai lorsque l'une ou l'autre reste dans le même état. Nous n'avons à priori donc aucune garantie que modifier les deux variables en même temps de façon parallèle soit également améliorant. Autrement dit, si deux threads découvrent en même temps ces deux mouvements améliorants, et les effectuent naïvement de façon concurrente, cela peut poser des problèmes de correction conceptuelle lors de la parallélisation de la recherche locale. Nous devons donc trouver un moyen de s'assurer que, lorsque l'on veut effectuer plusieurs mouvements en parallèle, les uns n'interfèrent pas avec les autres.

Pour ce faire, une idée est de pouvoir *partitionner* les mouvements améliorants de telle sorte à ce qu'ils puissent effectivement s'opérer en parallèle. Ce type de problématique de partitionnement est bien connu en théorie des graphes, et peut s'appréhender en le modélisant comme un problème de *coloration de graphe*.

Dans ce chapitre, constituant notre première contribution, nous allons montrer comment, en se basant sur l'idée d'utiliser une coloration du graphe représentant les interactions de variable dans un paysage NK, il est possible de concevoir un algorithme de Hill Climber parallèle en mémoire partagée. Dans un premier temps, nous décrivons une première approche utilisant une représentation sous forme d'un graphe spécifique aux paysages NK. Nous étudierons son efficacité en termes de qualité des optima locaux obtenus, mais aussi en termes de temps de calcul et d'accélération parallèle. Nous verrons dans un second temps comment cette approche peut se généraliser à d'autres problèmes k-bornés par le biais de l'utilisation d'une autre forme de représentation du problème, cette fois plus générale : le graphe d'interaction des variables (VIG). Là aussi, nous étudierons les performances en termes de vitesse, de scalabilité et de qualité, cette fois-ci sur des paysages Mk. Pour finir, nous verrons qu'il est possible d'utiliser les informations du VIG afin d'augmenter encore plus la vitesse de notre approche en évitant de refaire certains calculs inutilement.

3.2 Hill Climber parallèle basée sur une coloration à distance 2 du graphe NK

3.2.1 Graphe NK

Commençons par rappeler une spécificité des paysages NK tels que définis formellement dans la section 1.1.3 du premier chapitre. Dans ceux-ci, il y a autant de sous-fonctions que de variables, et la i-ième sous-fonction dépend nécessairement de la i-ième variable ainsi que de K autres. Définissons donc ce que l'on appellera le *graphe NK* associé à un paysage NK. Il s'agit d'un graphe orienté. Sans ambiguïté possible, chaque sommet d'un

$$f^0(x_0,x_1,x_2),\ f^1(x_1,x_4,x_2),\ f^2(x_2,x_6,x_4),\ f^3(x_3,x_9,x_0),\ f^4(x_4,x_1,x_5),$$
$$f^5(x_5,x_7,x_1),\ f^6(x_6,x_0,x_2),\ f^7(x_7,x_9,x_0),\ f^8(x_8,x_6,x_4),\ f^9(x_9,x_{14},x_7)$$
$$f^{10}(x_{10},x_4,x_{13}),\ f^{11}(x_{11},x_6,x_8),\ f^{12}(x_{12},x_{10},x_{11}),\ f^{13}(x_{13},x_4,x_5),\ f^{14}(x_{14},x_9,x_3)$$

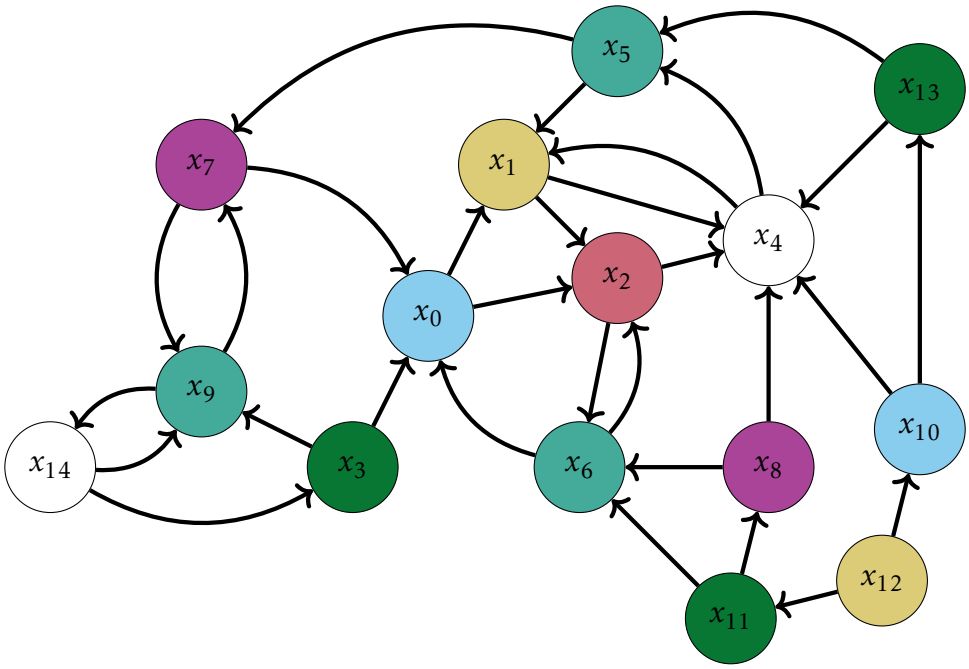

FIGURE 3.1 – *Exemple de coloration à distance 2 d'un paysage NK aléatoire représenté sous forme de graphe NK.*

graphe NK représentera à la fois une variable et la sous-fonction associée. Les arcs d'un graphe NK sont définis de la manière suivante, de sorte à ce qu'ils décrivent les appartenances des variables aux sous-fonctions : il existe un arc allant du sommet i au sommet j si la variable j dépend de la fonction i. De plus, on ne considère pas les arcs formant une boucle sur un même sommet. La figure 3.1 illustre un exemple d'un tel graphe.

3.2.2 Coloration

Une fois le graphe NK obtenu, nous allons maintenant le colorier de sorte à obtenir des partitions de variables linéairement indépendantes. Rappelons qu'une coloration (simple et propre) d'un graphe est une affectation de couleurs aux sommets du graphe telle que deux sommets adjacents (reliés par un arc) n'ont pas la même couleur. Si l'on colorie directement le graphe NK obtenu précédemment, la propriété de linéarité et d'indépendance ne serait pas respectée. En effet, considérons une sous-fonction i dépendant donc de la variable i et (disons) des variables j et k. De part la définition de graphe NK, une coloration

de ce dernier considérerait éventuellement que j et k sont indépendantes et leur affecterait une même couleur. Or, ces deux variables ne le sont pas puisqu'elles appartiennent à la même sous-fonction. Heureusement, il existe un moyen simple de régler ce problème. Il s'agit de colorier le graphe à distance 2 en considérant les arcs comme des arêtes. Une coloration à distance 2 est par extension une coloration propre telle que tout sommet a une couleur différente de tout autre sommet se trouvant à distance 2 dans le graphe d'origine. On utilise ici une notion de distance qui ne prend pas en compte le fait que le graphe NK soit orientée ; autrement dit, deux sommets sont à distance 2 s'ils sont séparés par deux arcs peu importe l'orientation de ces arcs. En utilisant une telle coloration, toutes les variables appartenant à une même sous-fonction sont bien de couleurs différentes, car la distance les séparant est au maximum égale à 2.

Bien entendu, le problème de coloration de graphe est lui-même un problème d'optimisation complexe et algorithmiquement difficile, lorsqu'il s'agit de minimiser le nombre de couleurs utilisées, appelé nombre chromatique. Cependant, notre but ici n'est pas de trouver la meilleure coloration possible par rapport au nombre chromatique, mais simplement une coloration permettant d'inférer une partition des variables de taille raisonnablement équilibrée afin de profiter du parallélisme que nous souhaitons mettre en œuvre. En effet, chaque couleur définit de fait un sous-ensemble de sommets indépendants, lesquelles forment une partition de l'ensemble de tous les sommets. Intuitivement parlant, plus le nombre de couleur est petit, plus la taille des sous-ensembles indépendants tend à augmenter ; offrant ainsi un plus grand degré de parallélisme. Dans la suite, nous considérerons une coloration où les sommets sont considérés dans un ordre aléatoire, nous verrons que cette dernière permet d'obtenir de très bons résultats.

3.2.3 Description de l'algorithme de descente parallèle

Nous présentons dans cette section notre conception d'un algorithme parallèle efficace, capable de rivaliser avec HBHC en termes de qualité, en offrant également la possibilité d'être plus rapide à mesure que l'on augmente le nombre de threads s'exécutant en parallèle.

Pour cela, considérons une coloration à distance 2 du graphe NK. Cette coloration définit une partition des sommets en sous-ensemble de sommets. Par construction, les sommets appartenant à chaque sous-ensemble se trouvent à une distance au moins 3 les uns des autres. La partie principale de notre algorithme consiste alors à itérer sur les différents sous-ensembles, tels que définis par la coloration du graphe NK, et d'effectuer des mouvements améliorants en parallèle jusqu'à ce qu'il n'y en ait plus. Un mouvement consiste ici à flipper la valeur d'une variable/sommet. Il est important de remarquer que les mouvements possibles pour un même sous-ensemble, constitués donc de sommets de même couleur, sont, en effet, indépendants et aucune interaction entre les variables

impliquées n'est possible par la définition même de la coloration à distance 2 du graphe NK. Ils peuvent donc être effectués en parallèle, ce qui résout le problème de conflit et de concurrence. Cependant, se pose désormais le défi suivant : trouver efficacement si un mouvement est améliorant ou non. Rappelons que notre objectif principal est d'améliorer la vitesse de la recherche locale par le biais de la parallélisation. Si l'on recalcule le score de chaque mouvement, et même si on le faisait en parallèle, cela serait très probablement plus lent que l'approche séquentielle HBHC, qui elle est capable de trouver les mouvements améliorants en temps constant. Si l'on essaye maintenant de conserver les scores des variables à jour, on retombe sur des problèmes d'accès concurrents. On doit donc trouver un autre moyen d'accélérer l'identification des mouvements améliorants ainsi que de les effectuer, le tout dans un environnement parallèle.

Avant de décrire notre proposition plus en détail, nous avons besoin d'introduire la notation suivante. L'ensemble des indices des sous-fonctions où la variable s apparaît, à l'exception de la sous-fonction s, sera noté $I^{-}(s)$. Ce qui correspond dans notre graphe NK à l'ensemble des sommets connectant les arêtes incidentes à s. Ainsi, si l'on reprend la figure 3.1, on a $I^{-}(0) = \{3,6,7\}$; remarquons que 0 n'en fait pas partie. Rappelons aussi que la coloration à distance 2 du graphe NK implique que les variables d'une même sous-fonction appartiennent à des sous-ensembles indépendants de couleurs différentes.

Voyons maintenant en détail l'algorithme présenté en pseudocode 7. Il commence par une phase d'initialisation (ligne 1) qui consiste à calculer et à stocker la contribution initiale de chaque sous-fonction dans un tableau F. Une fois cette phase d'initialisation effectuée, la boucle principale peut commencer. Cette dernière consiste à parcourir toutes les couleurs (ligne 4) séquentiellement, puis pour chacune de ces couleurs à effectuer *en parallèle* la boucle suivante (ligne 5) : pour toutes les variables de la couleur courante q, chaque thread va parcourir les variables qui lui ont été attribuées et effectuer les opérations suivantes pour chacune d'entre elle (notons la variable courante s) :

1. Le thread calcule la contribution de la sous-fonction ayant le même indice que la variable s qu'il est en train de traiter, cette valeur est stockée dans le tableau $F^{\oplus}$ (ligne 6) ;
2. On calcule ensuite la différence entre la nouvelle contribution et l'ancienne de la sous-fonction s, on stocke cette différence dans Δ_s (ligne 7)
3. on réitère avec toutes les sous-fonctions dépendant de s (ligne 8 à 11), on a ainsi à la fin Δ_s qui correspond au score de s.
4. Si ce score est supérieur à 0 (ligne 12), alors on applique ce mouvement à la solution partagée x (ligne 14), puis on met à jour les contributions des sous-fonctions pour prendre en compte ce changement. (ligne 15)

On peut remarquer ligne 13, que l'on utilise un booléen appelé LO que l'on passe à *faux* lorsqu'un mouvement améliorant est effectué, ce dernier nous sert de critère de

Algorithme 7 : Hill climber basé sur la coloration du graphe NK^2

Input : Une solution x et une coloration du graphe NK^2

```
   // Initialisation des contributions de chaque sous-fonction
1  for i ∈ {1,...,n} do F[i] ← f^(i)(x) ; F^⊕[i] ←⊥ ;
2  repeat
3      LO ← true ;
4      for q ∈ {1,...,c} do // Boucle sur les variables de couleur q
5          for s ∈ C_q do (en parallèle)
               // Calcul de la contribution de la sous-fonction s si on change s
6              F^⊕[s] ← f^(s)(x ⊕ s) ;
7              Δ_s ← F^⊕[s] − F[s] ;
               // Calcul de la contribution des autres sous-fonctions dépendants
                  de s si on change s
8              for ℓ ∈ I^−(s) do
9                  F^⊕[ℓ] ← f^(ℓ)(x ⊕ s) ;
10                 Δ_s ← Δ_s + F^⊕[ℓ] − F[ℓ] ;
11             end
               // Si Δ_s > 0, alors s est un mouvement améliorant
12             if Δ_s > 0 then
13                 LO ← false;
                   // on change la valeur de s et on met à jour les contributions
14                 x ← x ⊕ s ;
15                 for ℓ ∈ {s} ∪ I^−(s) do F[ℓ] ← F^⊕[ℓ] ;
16             end
17             Barrière de synchronisation;
18         endfor
19     end
20 until LO = true;
```

convergence, en effet, on voit qu'une fois que l'on a bouclé sur toutes les couleurs, et donc de fait sur toutes les variables, ce booléen est remis à *vrai* (ligne 3). Ainsi, le Hill Climber s'arrête quand toutes les variables ont été testées sans qu'aucune d'entre elles ne soit améliorante (ligne 20).

En raison du fait que les variables soient testées et potentiellement modifiées en parallèle, il faut apporter la plus grande importance à ce que les threads n'opèrent pas sur des couleurs différentes. Pour ce faire, nous utilisons une barrière de synchronisation

N	$k=4$			$k=6$			$k=8$			$\forall\, k$		
	▲	≈	∇	▲	≈	∇	▲	≈	∇	▲	≈	∇
10	2	18	0	3	16	1	1	19	0	6	53	1
30	1	19	0	1	19	0	2	18	0	4	56	0
100	2	18	0	12	8	0	10	10	0	24	36	0
300	13	7	0	17	3	0	14	6	0	44	16	0
1000	20	0	0	20	0	0	20	0	0	60	0	0
$\forall\, N$	38	62	0	53	46	1	47	53	0	138	161	1

TABLE 3.1 – *Nombres d'instances où le Hill Climber parallèle basé sur une coloration du graphe NK est meilleur (▲), pire (∇) ou non statistiquement différent (≈) du Hamming Ball Hill Climber à distance 1 ; le test utilisé est celui de Mann-Whitney-Wilcoxon avec un seuil de significativité de 0.05. Les valeurs de N sont affichées en milliers.*

(ligne 17) afin de s'assurer que tous les threads aient fini d'itérer sur une couleur avant de passer à la suivante.

3.2.4 Expérimentation

Dans cette section, nous étudions la compétitivité de notre approche parallèle face à l'algorithme séquentiel état de l'art, Hamming Ball Hill Climber (HBHC), en utilisant une distance de Hamming de 1. Pour ce faire, nous considérons un large panel de paysages NK aléatoire de tailles et de rugosités différentes : $N \in \{10\,000,\ 30\,000,\ 100\,000,\ 300\,000,\ 1\,000\,000\}$; $k \in \{4,\ 6,\ 8\}$ (soit $K = \in \{3,\ 5,\ 7\}$). Nous utilisons 20 instances par combinaison de N et K, et nous effectuons 20 exécutions indépendantes pour chaque instance. Les expériences sont réalisées en utilisant un processeur AMD EPYC 7 642 de 48 cœurs. Les algorithmes ont été implémentés en C dans un environnement en mémoire partagée utilisant la technologie OMP.

Qualité des optima locaux

Nous commençons par comparer la qualité des optima locaux obtenus avec notre algorithme parallèle comparé à HBHC. En effet, il est à noter que les mouvements effectués dans les versions parallèles et séquentielles, ne sont pas les mêmes ; ce qui peut mener à des optima locaux différents. Nous utiliserons un test de Mann-Withney-Wilcoxon avec un seuil de significativité de 0.05. Nos résultats sont résumés dans le tableau 3.1.

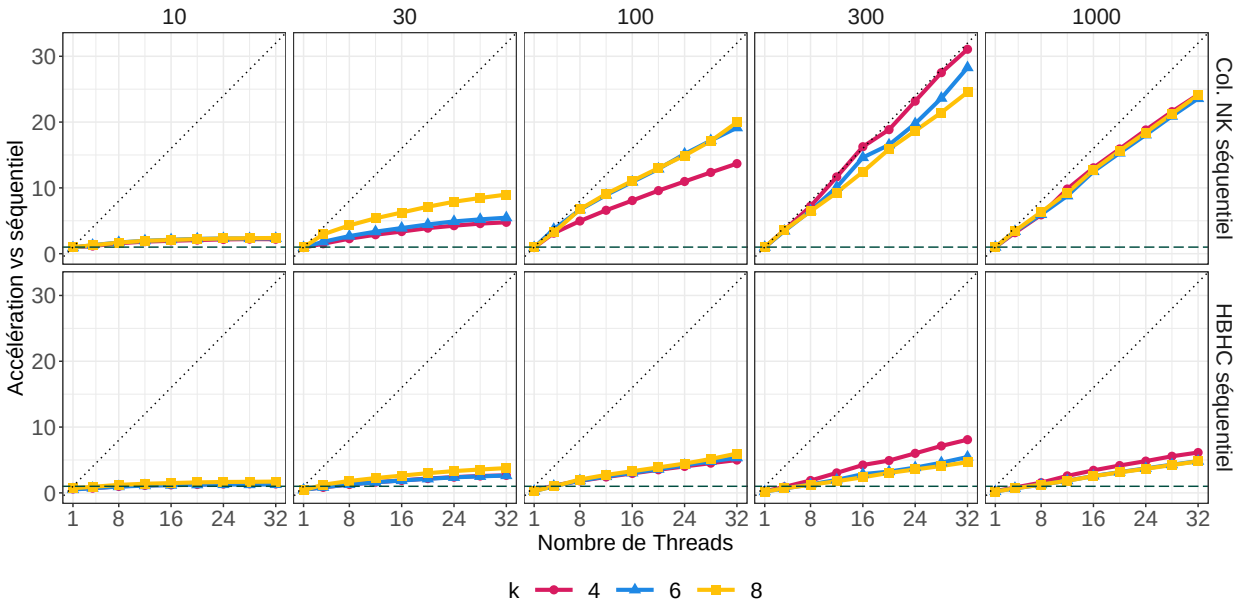

FIGURE 3.2 – *Accélération moyenne constatée pour un nombre de threads croissant du Hill Climber parallèle basé sur une coloration du graphe NK. L'accélération est calculée par rapport au temps du même algorithme avec 1 thread pour la première ligne et par rapport au HBHC séquentiel pour la seconde ligne de la figure. La couleur correspond à la valeur de k. De gauche à droite, nous montrons les résultats obtenus dans l'ordre croissant les valeurs de N exprimées en milliers.*

Comme on peut le voir, la différence de qualité est fortement dépendante de la taille des instances considérées et assez peu de la rugosité. Pour les problèmes de petites tailles, on voit que la plupart du temps les algorithmes ont des performances similaires, l'approche parallèle étant meilleure sur 6 instances et moins bonne sur une seule, tandis que sur les 53 restantes, il n'y a pas de différence significative de qualité. On voit ensuite clairement que l'écart se creuse à mesure que la taille de l'instance augmente. Pour les quatre plus grandes tailles d'instances considérées, HBHC n'est jamais meilleur, il est même battu de plus en plus souvent, jusqu'à être 100% du temps statistiquement moins bon que notre algorithme parallèle basé sur la coloration du graphe NK. En résumé, sur toutes les instances agrégées, l'approche parallèle est statistiquement meilleure sur 138 des 300 instances et n'est battue que sur une seule de toutes ces instances. Ceci montre clairement la supériorité de notre approche.

Accélération parallèle

Maintenant que nous avons vu que l'algorithme parallèle était très compétitif en termes de qualité des optima locaux obtenus, nous étudions le gain en terme de temps de calcul nécessaire pour les atteindre. Pour ce faire, appuyons-nous sur la figure 3.2. Nous pouvons y voir l'accélération obtenue en fonction du nombre de threads comparée

au même algorithme avec un thread pour la première ligne de la figure et à l'algorithme séquentiel HBHC pour la seconde ligne. De gauche à droite, on retrouve par ordre croissant les différentes tailles d'instances en milliers, et la rugosité est représentée à l'aide de couleurs différentes. La ligne oblique en pointillé noire dans la diagonale représente une accélération linéaire, tandis que celle horizontale, en vert, représente le seuil à partir duquel l'accélération est supérieure à 1.

Une première vue globale nous rassure quant à la scalabilité de notre algorithme puisque l'accélération en fonction du nombre de threads est toujours croissante.

Intéressons-nous désormais à la première rangée de sous-figures. L'accélération est clairement fonction de la taille N du problème. Jusqu'à 300 000 variables, plus la taille augmente plus l'accélération est importante, jusqu'à être linéaire pour $N = 300\,000$ et $k = 4$. Ensuite, l'accélération redescend pour les plus grandes instances considérées tout en restant importante : environ 25 avec 32 threads pour 1 million de variables, et ce, pour toutes les valeurs de k. Dans le même temps, on constate que l'impact de la rugosité est plus limité et ne semble pas suivre de tendance aussi clair que pour la taille des instances. Les résultats montrent qu'on peut obtenir une accélération intéressante même sur des instances de 30 000 variables : entre 9 et 10 pour $k = 8$. Cependant, pour la plus petite taille de problème considérée l'accélération est assez limitée, de l'ordre d'un facteur 2 ou 3.

Analysons maintenant l'accélération comparée à l'algorithme séquentiel HBHC telle que montrée dans la deuxième rangée de sous-figures. On peut voir que la tendance est la même, mais avec une accélération nettement plus faible. Cela s'explique par l'extrême efficacité de l'algorithme HBHC, et sa capacité à trouver et à conserver la liste des mouvements améliorants en temps constant. Il est donc relativement difficile d'obtenir une accélération supérieure à 1. Notre approche dépassant cependant ce seuil pour les 4 tailles d'instances les plus grandes, avec une accélération moyenne supérieure à 5 pour $N \in \{100\,000,\ 300\,000,\ 1\,000\,000\}$ et sur toutes les valeurs de k.

Ces premiers résultats sont prometteurs dans le sens que notre nouvelle recherche locale découvre des solutions de meilleures qualités et plus rapidement que HBHC sur une grande variété d'instances de paysages NK de grandes tailles et de rugosités différentes.

Dans la suite, nous nous consacrerons à la généralisation de cette recherche locale parallèle pour les paysages Mk ; ainsi qu'à l'amélioration de la vitesse d'exécution et des accélérations parallèles que nous pouvons obtenir.

3.3 Généralisation aux paysages Mk par la coloration du graphe d'interaction des variables

Jusqu'à maintenant, notre approche se basait sur un graphe NK qui tient compte des spécificités des paysages NK. Il est cependant possible d'utiliser une représentation plus

$$f^0(x_0,x_1,x_2),\ f^1(x_1,x_4,x_2),\ f^2(x_2,x_6,x_4),\ f^3(x_3,x_9,x_0),\ f^4(x_4,x_1,x_5),$$
$$f^5(x_5,x_7,x_1),\ f^6(x_6,x_0,x_2),\ f^7(x_7,x_9,x_0),\ f^8(x_8,x_6,x_4),\ f^9(x_9,x_{14},x_7)$$
$$f^{10}(x_{10},x_4,x_{13}),\ f^{11}(x_{11},x_6,x_8),\ f^{12}(x_{12},x_{10},x_{11}),\ f^{13}(x_{13},x_4,x_5),\ f^{14}(x_{14},x_9,x_3)$$

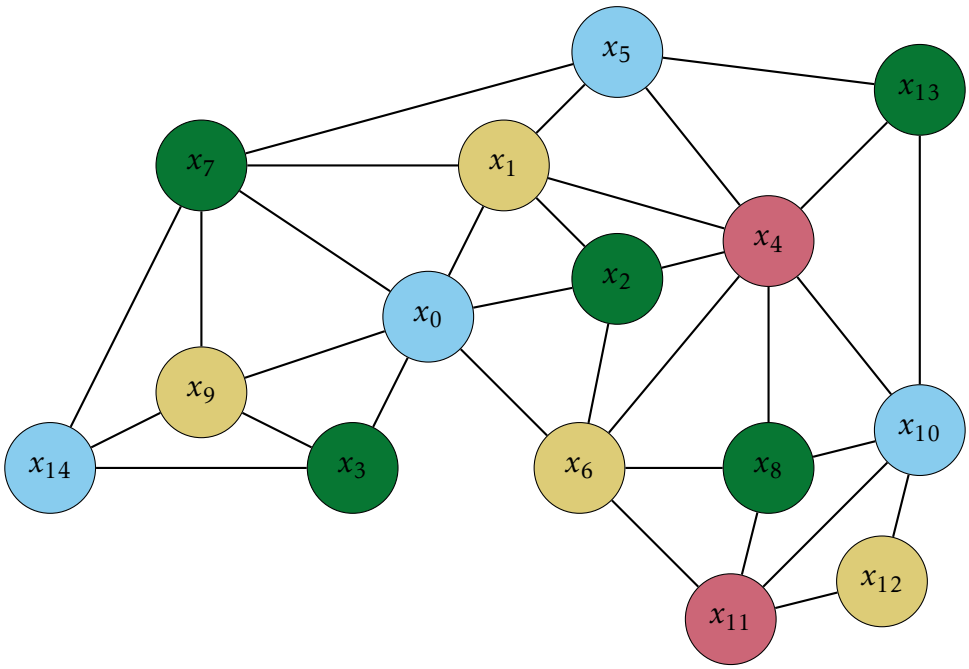

FIGURE 3.3 – *Exemple de coloration du graphe d'interactions des variables associé à un paysage NK aléatoire*

générale qui fonctionne pour tous les problèmes d'optimisation pouvant se formuler sous la forme d'un paysage Mk. Cette représentation repose sur le graphe d'interaction des variables (VIG) associés à un problème. Rappelons que dans ce graphe, chaque variable est représentée par un sommet et deux sommets sont reliés s'ils apparaissent tous deux dans une même sous-fonction. Si l'on se place du point de vue des sous-fonctions, cela veut dire que chaque sous-fonction est représentée par une clique dans le VIG.

Souvenons-nous que, dans la section précédente, l'algorithme parallèle basé sur la coloration du graphe NK devait remplir deux critères. Le premier est qu'il faut partitionner les variables en sous-ensembles qui n'interagissent pas. Nous remarquons ainsi que ce critère reste parfaitement vérifié si l'on considère une coloration à distance 1 du VIG (à la place du graphe NK). En effet, les variables du VIG sont par définition reliées entre elles si et seulement si elles interagissent. Le deuxième critère est qu'il faut s'assurer que dans un sous-ensemble de variables indépendantes, toutes paires de variables ne dépendent pas d'une même sous-fonction. Cette seconde propriété reste de même vérifiée en considérant une coloration propre du graphe VIG ; par le fait que la coloration d'une clique de taille k

va nécessiter k couleurs. Ainsi, deux variables d'une sous-fonction ne pourront jamais se retrouver dans la même partition. La figure 3.3 montre un exemple d'une telle coloration sur le même paysage NK utilisé précédemment, mais en considérant le graphe VIG associé (et non le graphe NK). On peut remarquer que le graphe VIG est un peu plus dense, mais qu'il nécessite moins de couleurs, car la coloration n'a plus besoin d'être à distance 2.

3.3.1 Propagation d'états basée sur les liens entre variables

Au-delà de la capacité de généralisation apportée par le VIG, nous proposons également d'utiliser les informations relatives aux interactions entre variables afin de, tout comme le fait HBHC, réduire le nombre de calculs nécessaires pour trouver un optimum local. L'idée est la suivante. Pour savoir si une variable est améliorante lorsque sa valeur change, il faut calculer son score au moins une fois. Il y a alors deux cas possibles pour une variable : soit elle est améliorante, soit elle ne l'est pas. Considérons le cas où une variable n'est pas améliorante dans une itération donnée. À la prochaine itération, si aucune des variables voisines (dans le VIG) n'a changée, alors la variable ne peut pas devenir améliorante ; car il faut qu'au moins l'une de ses voisines ait été changée. Ainsi, il n'est pas utile de recalculer/revérifier son score. Maintenant, dans le deuxième cas où la variable est améliorante (donc avec un score positif), alors on peut changer sa valeur. Mais dans ce cas, si aucune de ses voisines ne change d'état alors nul besoin de la reconsidérer dans l'itération suivante, puisque son score sera nécessairement négatif.

Basé sur cette réflexion simple, nous proposons alors d'introduire ce que nous appelons la propagation d'états dans notre algorithme (voir le pseudo-code de l'algorithme 8). L'idée est simple. Chaque variable commence dans un état éveillé (correspondant à une valeur de $E[i] = 0$ dans l'algorithme 8). Une fois la variable considérée, qu'elle soit améliorante ou non, on la passe dans un état endormi (valeur de 1 dans l'algorithme). Si la variable a été changée alors on réveille toutes les variables adjacentes à celle-ci dans le VIG, ce qui correspond à la fonction PropagationÉtats (ligne 17). Bien entendu, dans la boucle principale, il ne faut considérer maintenant que les variables dont l'état est éveillé, ce qui correspond à la ligne 7. Si un thread passe sur une variable endormie, il continuera directement avec la suivante. Nous montrons dans la suite que cette idée simple de propagation d'état permet un gain substantiel en termes de temps d'exécution.

3.3.2 Expérimentations

Dans cette partie, nous analysons la performance du nouvel algorithme parallèle basé sur la coloration du VIG en termes de qualité et également en terme de vitesse d'exécution. Les expériences sont faites en considérant un vaste ensemble de paysages Mk avec $M \in \{1, 2, 4\}$; $N \in \{10\,000, 30\,000, 100\,000, 300\,000, 1\,000\,000\}$; $k \in \{4, 6, 8\}$. Ici, M

Algorithme 8 : Hill Climber parallèle basé sur la coloration du VIG avec propagation d'états

Input : Une solution x et une coloration du graphe d'interactions

```
// Initialisation des contributions de chaque sous-fonction
for i ∈ {1,...,m} do F[i] ← f^(i)(x); F^⊕[i] ←⊥ ;
// On initialise l'état des n variables à 0
for i ∈ {1,...,n} do E[i] = 0 ;
repeat
    LO ← true ;
    for q ∈ {1,...,c} do // Boucle sur les variables de couleur q
        for s ∈ C_q do (en parallèle)
            // Si s est dans un état activé
            if E[s] == 0 then
                // Calcul de la nouvelle contribution des sous-fonctions
                   dépendantes de s si on change s
                Δ_s ← 0;
                for ℓ ∈ L^-(s) do
                    F^⊕[ℓ] ← f^(ℓ)(x ⊕ s) ;
                    Δ_s ← Δ_s + F^⊕[ℓ] − F[ℓ] ;
                end
                // Si Δ_s > 0, alors s est un mouvement améliorant
                if Δ_s > 0 then
                    LO ← false;
                    // On change la valeur de s et on met à jour les
                       contributions
                    x ← x ⊕ s ;
                    for ℓ ∈ L^-(s) do F[ℓ] ← F^⊕[ℓ] ;
                    // on active les voisins de s
                    PropagationÉtats(E,s);
                end
                E[s] = 1;
            end
            Barrière de synchronisation;
        endfor
    end
until LO = true;
```

représente le multiplicateur à appliquer aux nombres de variables pour obtenir le nombre de sous-fonctions. Pour chaque combinaison de paramètres, nous générons 20 instances aléatoires et effectuons pour chacune d'entre elles 20 exécutions de chaque algorithme. Pour la version parallèle, nous utiliserons un nombre croissant de threads jusqu'à 48. Le processeur sur lequel est réalisé cette série d'expériences est le même que dans la section précédente, à savoir un AMD EPYC 7 642 de 48 cœurs. L'implémentation est réalisée en C et nous utilisons la technologie OMP pour la gestion de l'exécution parallèle en mémoire partagée.

Qualité des optima locaux

Commençons par l'analyse de la qualité des optima locaux obtenues par rapport à l'algorithme séquentiel HBHC. Comme auparavant, nous utilisons un test de Mann-Whitney-Wilcoxon avec un seuil de significativité de 0.05. Dans le tableau 3.2, on peut voir que les résultats dépendent des instances considérées. Pour les instances avec la plus faible rugosité, $M = 1$ et $k = 4$, on voit que HBHC est mieux sur les instances de grandes tailles. Cependant, à mesure que la valeur de M ou k augmente, HBHC ne parvient plus à battre l'approche parallèle basée sur la coloration du VIG mis à part pour quelques très rares exceptions. Le plus souvent pour les instances de tailles moyennes, les résultats sont mitigés ; aucun algorithme ne parvenant à prendre le dessus sur l'autre, tandis que sur les plus grandes instances, l'algorithme parallèle parvient à surpasser HBHC sur la majorité des instances.

Si l'on regarde les résultats agrégés pour les différentes valeurs de M (avant-dernière ligne dernière colonne du tableau), on voit une tendance globale se dessiner en fonction de N. Pour $N = 10\,000$ et $30\,000$, les résultats sont très similaires, avec 167 instances pour chaque valeur de N où l'on n'observe pas de différence significative, respectivement 10 et 9 instances où la version parallèle est meilleur, et seulement 3 et 4 instances où elle est battue. Pour les valeurs de $N = 100\,000$ et $300\,000$, on observe plus souvent des résultats significatifs, avec seulement 143 et 116 comparaisons au-dessus du seuil de 0.05, tandis que le nombre de résultats significatifs en faveur de l'algorithme parallèle croît plus rapidement en fonction de N que celui en sa défaveur, respectivement 25 et 46 contre 12 et 18. Enfin, pour les plus grandes instances, celles de 1 million de variables, l'algorithme parallèle surpasse HBHC séquentiel sur 110 instances et n'est battu que dans 22 sur 200 instances considérées.

En résumé, si l'on considère la performance globale sur un total de 900 instances considérées : sur 641 d'entre elles il n'y a pas de différence significative et sur les 259 restantes, 200 sont en faveur de l'approche parallèle contre seulement 59 en faveur de l'approche séquentielle état de l'art HBHC. Notons que 54 de ces 59 instances concernent les paysages les moins rugueux, soit $M = 1$ et $k = 4$. Sur les autres paysages, plus rugueux

M	N	$k=4$			$k=6$			$k=8$			$\forall\, k$		
		▲	≈	▽	▲	≈	▽	▲	≈	▽	▲	≈	▽
1	10	0	18	2	2	17	1	1	19	0	3	54	3
	30	0	18	2	0	20	0	0	20	0	0	58	2
	100	0	8	12	0	20	0	2	18	0	2	46	12
	300	0	2	18	1	19	0	5	15	0	6	36	18
	1000	0	0	20	1	18	1	13	7	0	14	25	21
2	10	0	20	0	0	20	0	3	17	0	3	57	0
	30	0	19	1	1	19	0	2	17	1	3	55	2
	100	1	19	0	7	13	0	4	16	0	12	48	0
	300	3	17	0	9	11	0	9	11	0	21	39	0
	1000	7	12	1	18	2	0	18	2	0	43	16	1
4	10	0	20	0	1	19	0	3	17	0	4	56	0
	30	3	17	0	2	18	0	1	19	0	6	54	0
	100	2	18	0	5	15	0	4	16	0	11	49	0
	300	5	15	0	10	10	0	4	16	0	19	41	0
	1000	18	2	0	19	1	0	16	4	0	53	7	0
$\forall\, M$	10	0	58	2	3	56	1	7	53	0	10	167	3
	30	3	54	3	3	57	0	3	56	1	9	167	4
	100	3	45	12	12	48	0	10	50	0	25	143	12
	300	8	34	18	20	40	0	18	42	0	46	116	18
	1000	25	14	21	38	21	1	47	13	0	110	48	22
$\forall\, M$	$\forall\, N$	39	205	56	76	222	2	85	214	1	200	641	59

TABLE 3.2 – *Nombre d'instances où le Hill Climber parallèle basé sur une coloration du graphe d'interaction des variables est meilleur (▲), pire (▽) ou non statistiquement différent (≈) du HBHC à distance 1. Le test utilisé est celui de Mann-Whitney-Wilcoxon avec un seuil de significativité de 0.05. Les valeurs de N sont exprimées en milliers.*

et donc plus difficiles, l'approche à base de coloration du VIG surpasse 200 fois HBHC, en étant moins performante 5 fois seulement.

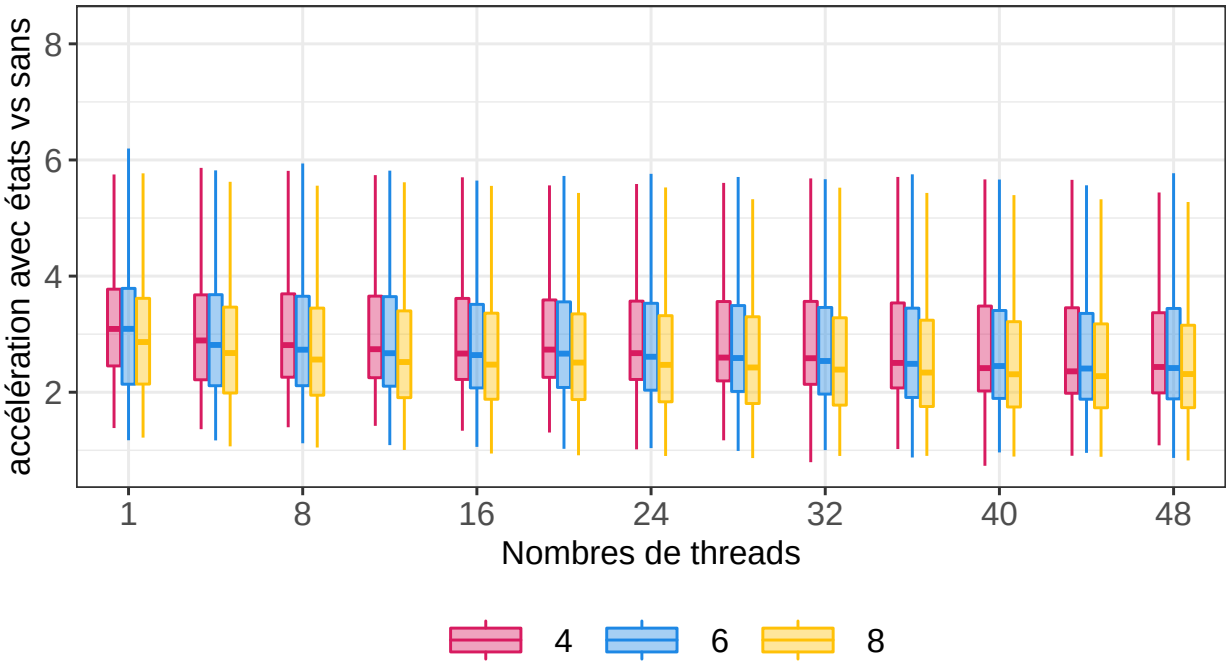

FIGURE 3.4 – *Accélération constatée pour un même nombre de threads lorsque l'on ajoute la propagation d'états à l'algorithme de Hill Climber parallèle basée sur une coloration du graphe d'interactions des variables. Les données sont agrégées suivant M et N et affichées pour 3 valeurs de k : 4, 6 et 8, le nombre de threads varie de 1 à 48*

Impact de la propagation d'états sur la vitesse et la scalabilité

La propagation d'états dans notre algorithme ne change en rien la dynamique de la recherche parallèle et donc la qualité des optima locaux trouvées qui reste donc globalement supérieur à HBHC, comme analysé de façon détaillé dans le paragraphe précédent. Cependant, la propagation d'état a pour but d'éviter les calculs inutiles et donc d'améliorer la vitesse d'exécution de l'algorithme. Observons donc comment la propagation d'états impacte la vitesse et la scalabilité de notre algorithme parallèle basé sur une coloration du VIG.

Pour ce faire, la figure 3.4 regroupe toutes les valeurs de M et N, et se concentre sur le gain de temps dans notre implémentation parallèle, pour un même nombre de threads, et ce, pour différentes valeurs de k représentées par des couleurs différentes. On peut voir que l'accélération apportée par l'utilisation de la propagation d'état est importante. En effet, le résultat principal de cette figure est que l'accélération est substantielle et constante suivant le nombre de threads, entre 2 et 4.

Voyons maintenant plus en détail comment, pour un nombre de threads donné, l'accélération évolue en fonction de M et N. Ainsi la figure 3.5 nous montre, pour 48 threads, l'accélération obtenue en fonction : de N sur l'axe des abscisses (en milliers), de k sous

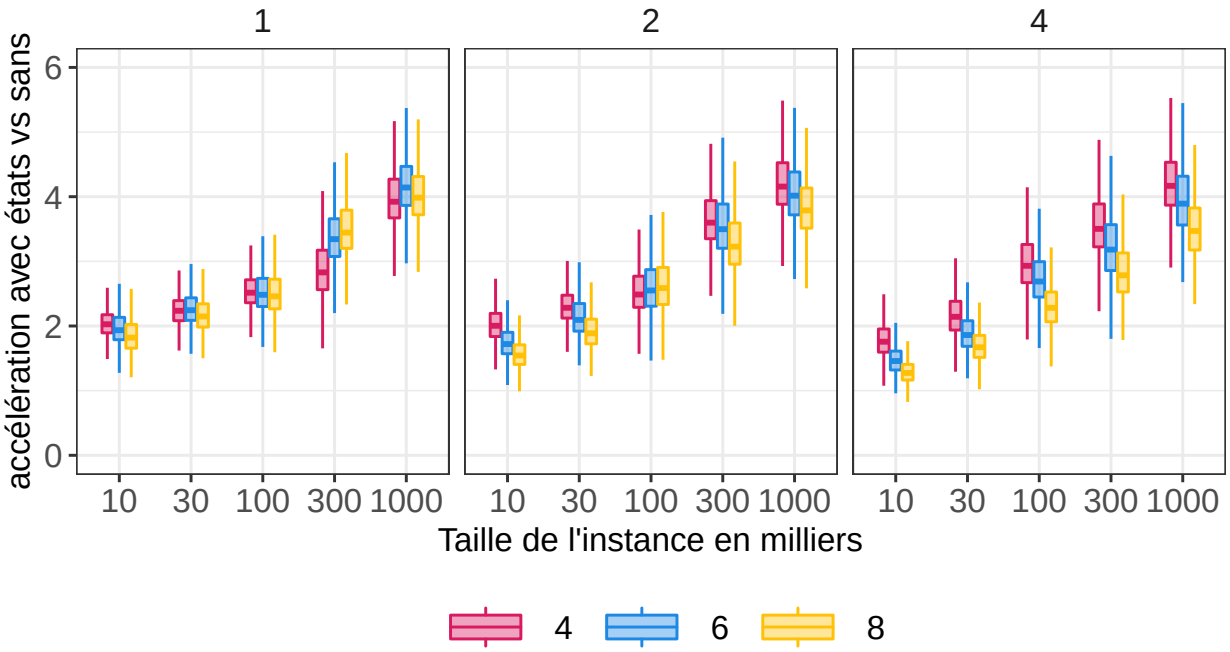

FIGURE 3.5 – *Accélération constatée avec 48 threads lorsque l'on rajoute la propagation d'états à l'algorithme de Hill Climber parallèle basée sur une coloration du graphe d'interactions des variables. Les données sont affichées pour un nombre croissant de variables N (en milliers) et pour 3 valeurs de k* : 4, 6 *et* 8 *représenté par une couleur. Enfin chaque colonne présente une valeur de M, de gauche à droite* 1, 2 *et* 4.

forme de couleurs différentes, et de M de gauche à droite. On peut donc voir que l'accélération dépend principalement de N et croît en fonction de la taille de l'instance allant de 1.5 en moyenne pour les plus petites ($N = 10\,000$) à environ 4 pour les plus grandes ($N = 1\,000\,000$). On observe quelques légères variations en fonction de M et k, cependant insuffisantes pour en déduire une tendance.

Ceci prouve que l'utilisation de la propagation d'états a un impact significatif sur la vitesse d'exécution de l'algorithme parallèle proposé. En évitant de reconsidérer inutilement des variables qui ne peuvent pas être améliorantes, on permet à notre algorithme d'être encore plus compétitif, et ce, sans altérer la qualité des solutions obtenues. De ce fait, dans la suite, nous ne considérerons plus que la version de l'algorithme parallèle basée sur la coloration du VIG avec propagation d'états. Il nous reste désormais à étudier l'accélération obtenue par rapport au HBHC séquentiel.

Accélération parallèle et scalabilité

Débutons notre analyse par l'étude de la scalabilité de l'algorithme parallèle basé sur la coloration du VIG avec propagation d'état. La figure 3.6 nous montre l'accélération obtenue

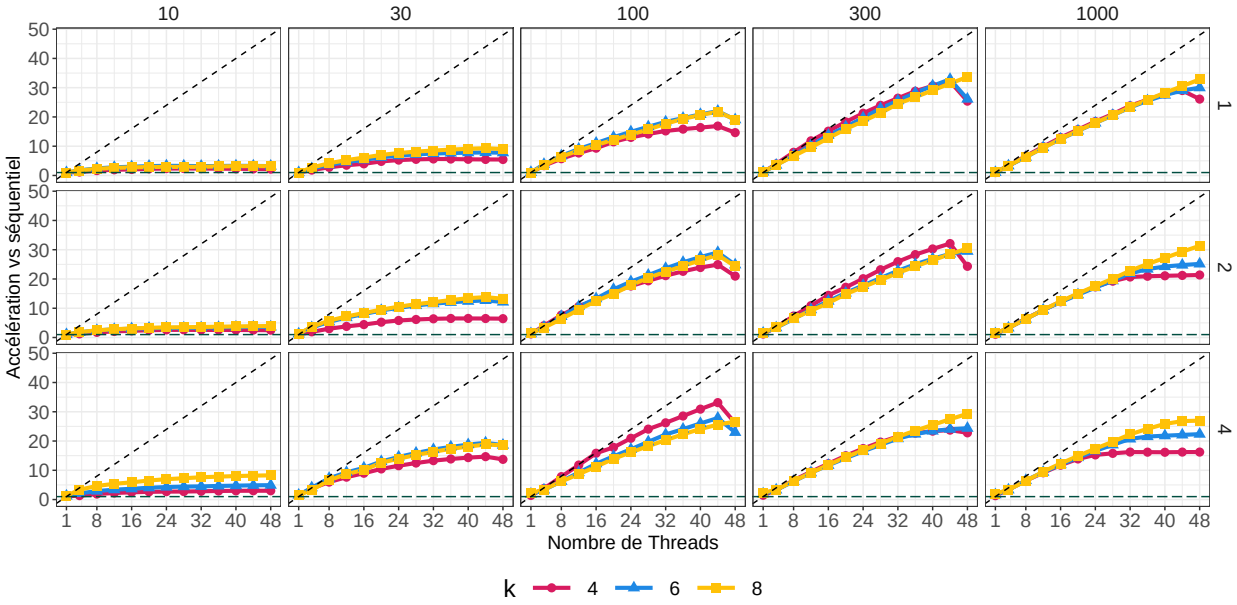

FIGURE 3.6 – *Accélération moyenne constatée pour un nombre de threads croissant de l'algorithme parallèle basé sur une coloration du graphe d'interaction des variables avec propagation d'états. L'accélération est calculée par rapport au temps du même algorithme avec 1 thread. La couleur correspond à la valeur de k, de gauche à droite, nous avons les valeurs de N en milliers, et de haut en bas les différentes valeurs pour M, le tout dans l'ordre croissant.*

pour un nombre croissant de threads allant jusqu'à 48 comparée au temps nécessaire pour le même algorithme avec 1 thread. De gauche à droite et par ordre croissant nous retrouvons les différentes valeurs de N en milliers, de haut en bas les différentes valeurs de M, à savoir 1, 2 et 4 ; les couleurs représentent les différentes valeurs de k. Précisons un point important, on constate que l'accélération est croissante suivant le nombre de threads, mais pour une raison probablement liée aux aspects techniques de l'unité de calcul utilisé (que nous n'avons pas approfondi ici), lorsque l'on utilise l'intégralité des cœurs du processeur, on constate une baisse de vitesse ; nous avons essayé sur un autre processeur avec un nombre de cœurs plus restreint et le comportement est équivalent lorsque l'on utilise tous les cœurs. Ainsi, nous omettrons de notre analyse le cas où l'on utilise 48 cœurs. En plus du cas séquentiel, toutes les expériences ont été effectuées pour tous les nombres de threads qui sont multiples de 2. Dans un souci de lisibilité, nous montrerons sur les deux figures suivantes les nombres de threads qui sont multiples de 4, en plus du cas séquentiel.

L'accélération est positivement impactée par la taille du problème. On constate un facteur très faible pour les instances de taille 10 000. Pour celles-ci, l'ajout de threads ne permet pas d'accélérer la recherche, et ce peu importe les valeurs de M et k. En ce qui concerne les plus grandes instances, soit $N \geq 100\,000$, on obtient une accélération significative de 15 à 30 suivant les valeurs de k et M. D'ailleurs, on remarque que k n'im-

M	K	$N = 10$	$N = 30$	$N = 100$	$N = 300$	$N = 1000$	$\forall N$
1	4	2.40	5.21	13.15	25.49	29.02	15.05
	6	3.00	7.25	18.06	22.78	24.57	15.13
	8	3.62	9.84	20.11	20.41	24.63	15.72
2	4	2.26	5.12	15.33	16.45	20.87	12.00
	6	2.85	7.50	13.65	14.98	16.52	11.10
	8	3.54	8.85	11.81	14.08	15.56	10.76
4	4	2.10	6.06	9.27	11.45	16.30	9.04
	6	2.75	7.57	7.70	8.82	9.36	7.24
	8	3.73	6.71	6.98	7.44	8.46	6.66
$\forall M$	4	2.25	5.46	12.58	17.80	22.06	12.03
	6	3.25	7.44	13.14	15.53	16.81	11.16
	8	4.10	8.47	12.96	13.98	16.21	11.05
$\forall M$	$\forall K$	3.20	7.12	12.89	15.77	18.54	11.50

TABLE 3.3 – *Accélération moyenne de l'algorithme parallèle basé sur la coloration du graphe d'interactions des variables avec propagation d'états en utilisant 46 threads versus HBHC séquentiel. Les valeurs de N sont affichées en milliers.*

pacte la scalabilité que sur peu d'instances, quand k est plus petit, l'accélération tend à être plus importante, mais sans réelle tendance de fond. À contrario, la valeur de M a un impact assez clair. Plus cette valeur est grande, plus l'accélération est faible. Cet effet est particulièrement marqué sur les grandes instances. En résumé, nous voyons que l'algorithme parallèle basé sur la coloration du VIG est capable d'atteindre des accélérations importantes sur des problèmes de tailles supérieures ou égales à 30 000 atteignant dans les meilleurs cas des valeurs supérieures à 30 pour 44 threads.

Pour la comparaison avec HBHC séquentiel, nous étudions dans la table 3.3 l'accélération moyenne obtenue avec 46 threads. Nous pouvons voir que celle-ci dépend fortement des valeurs de M, N et k. Elle est proportionnelle à la taille du problème, comme on peut le voir sur la dernière ligne, où l'on agrège les résultats par rapport aux valeurs de M et de k. Pour les plus petites instances, l'algorithme parallèle est en moyenne 3.20 fois plus rapide. Cette moyenne monte jusqu'à 18.54 pour les problèmes à 1 million de variables. Intéressons-nous désormais à l'impact de k, quand $N = 100\,000$ et pour toutes les valeurs de M ainsi que pour $N = 100\,000$ et $M = 1$, l'accélération augmente quand k augmente. Pour les instances plus grandes ou lorsque que $N = 100\,000$ et $M > 1$, on observe le comportement opposé. Concernant le dernier paramètre, M, une tendance

générale ressort : l'accélération est inversement proportionnelle à la valeur de M. On obtient une très bonne accélération moyenne de 11.50 sur toutes les instances combinées, et dans les meilleurs cas, pour les instances de très grandes tailles et de faibles rugosités, on observe une accélération moyenne qui peut atteindre 29.

3.4 Conclusion

Pour résumer, nous avons dans un premier temps proposé un algorithme Hill Climber parallèle pour les paysages NK, basé sur la coloration à distance 2 du graphe NK et fonctionnant en mémoire partagé. Nous avons montré qu'une telle approche était capable d'améliorer grandement la vitesse à laquelle nous pouvons obtenir des optima locaux par rapport à l'algorithme séquentiel état de l'art : le Hamming Ball Hill Climber (HBHC). De plus, dans 46% des cas la qualité moyenne de ces optima locaux est meilleure, n'étant inférieur que dans moins de 1% des cas.

Dans une seconde partie, nous avons généralisé cet algorithme aux modèles Mk en étendant la coloration au graphe d'interactions des variables. Nous avons également utilisé ce graphe pour introduire ce que l'on a appelé la propagation d'états, permettant de réduire significativement le nombre de calcul à effectuer et donc le temps d'exécution. Sur la base d'une analyse approfondie et rigoureuse suite à nombreuses expérimentations, nous avons montré que l'algorithme parallèle basé sur la coloration du graphe d'interaction des variables avec propagations d'états permet de tirer profit de la puissance de calcul allant jusqu'à 46 cœurs. Il obtient ainsi des optima locaux en moyenne 11.50 plus vite que l'algorithme HBHC séquentiel ; ces derniers étant même significativement meilleurs dans 22% des cas, et moins bon dans seulement 6.5% (et de qualité non statistiquement différente dans le reste des cas).

Cependant, dans nos expérimentations, nous avons noté une disparité en termes d'accélération suivant le nombre de changements de variables à effectuer avant d'atteindre un optima local. Lorsque ce nombre est élevé, l'accélération peut atteindre un facteur impressionnant de 29, mais lorsqu'il est faible alors on stagne aux alentours d'une accélération d'environ 3 pour 46 threads. Ce dernier point entraîne certaines limitations quant à l'efficacité attendue de notre Hill Climber parallèle lorsque utilisé dans une métaheuristique plus avancée, comme la recherche locale itérée DRILS [12] décrite dans la section 2.5.3. En effet, dans DRILS, l'algorithme HBHC n'est qu'une composante parmi d'autres. La dynamique de recherche consiste à perturber légèrement un optimum local, et donc en principe à le dégrader, avant de faire appel à une nouvelle itération de HBHC. En pratique, cette perturbation ne concerne que quelques pourcents des variables, ce qui pourrait réduire le nombre de mouvements que doit effectuer HBHC. Or, un faible nombre de mouvements

peut être un frein pour obtenir des accélérations intéressantes lorsque le nombre de cœurs est amené à augmenter de façon substantielle.

Une deuxième limitation de notre algorithme parallèle est cette fois-ci d'ordre technologique. En effet, notre approche est pensée pour fonctionner sur une unité de calcul en mémoire partagée (OMP). Cependant, la puissance des supercalculateurs modernes repose principalement sur la multiplication des processeurs, et donc sur l'usage supplémentaire d'autres technologies de parallélisation telles que la communication par échange de messages (MPI). Si l'on veut tirer profit efficacement d'une plus grande puissance de calcul, l'utilisation d'un Hill Climber parallèle n'est pas suffisante, et d'autres techniques pour exploiter cette puissance de calcul sont souhaitables.

Dans le chapitre suivant, nous nous attaquerons précisément à la conception d'algorithmes parallèles passant à l'échelle, en proposant une approche massivement parallèle capable de tirer profit de la puissance de dizaines de milliers de cœurs.

Chapitre 4

Approche coopérative massivement parallèle pour DRILS

4.1 Motivation

Dans le chapitre précédent, nous avons présenté une recherche locale parallèle dans un environnement à mémoire partagée. Comme commenté en fin du chapitre, malgré son efficacité dans un environnement de calcul à mémoire partagée, notre approche se heurte à des considérations algorithmiques et technologiques qui en limitent l'intérêt dans le cadre du développement d'une approche boîte grise avancée massivement parallèle et à très grande échelle. Dans ce chapitre, nous allons nous intéresser spécifiquement à cet aspect en proposant une conception parallèle capable de concurrencer l'algorithme de recherche locale itérée DRILS, tout en passant à l'échelle.

Avant d'aller plus loin dans le détail de notre proposition, rappelons que DRILS s'appuie sur l'application successive d'une recherche locale simple, en l'occurrence HBHC, d'un croisement boîte grise par partitions, en l'occurrence PX, puis d'une perturbation visant à s'échapper des optima locaux. Ainsi, en multipliant les perturbations des optima locaux, DRILS multiplie les itérations pour permettre à la recherche locale et au croisement de continuer à être fructueux. De ce fait, notre objectif dans la suite de ce chapitre, n'est plus d'accélérer le plus possible une itération de DRILS en la parallélisant de l'intérieur, mais plutôt d'effectuer le plus d'itérations séquentielles en parallèle. Cela est dû, en partie, au fait que la seule parallélisation d'une itération de DRILS ne permette de toute façon pas d'atteindre de bonnes propriétés en termes de scalabilité parallèle à grande échelle.

Dans ce cadre, une approche alternative, serait d'utiliser une métaheuristique à base de population. En effet, l'utilisation d'une population offre un parallélisme naturel que l'on pourrait exploiter. Un tel algorithme a été proposé, il s'agit de l'algorithme HiReLS, tel que décrit dans la section 2.5.2. Rappelons que HiReLS repose sur une population organisée sous forme de pyramide où les individus progressent dans la hiérarchie par croisements

entre individus d'un même niveau. Proposés dans le même papier que DRILS, HiReLS n'est cependant pas du tout compétitif sur les paysages NK aléatoires. Pour ces raisons et malgré le fonctionnement intrinsèquement séquentiel de DRILS (ce dernier étant un algorithme à trajectoire et donc fonctionnant avec une seule solution), il nous semble que la meilleure option est de considérer une itération de DRILS comme un seul bloc de base pour concevoir un algorithme massivement parallèle.

Afin de tirer profit de l'immense puissance de calcul offerte par les supercalculateurs modernes pour résoudre des problèmes toujours plus larges et plus difficiles, il nous faut opter pour une approche capable de combiner efficacement des technologies de mémoires partagées et d'échanges de messages. En effet, comme discuté plus loin, les approches développées dans ce chapitre sont destinées à être exécutées sur le super-calculateur Fugaku, le deuxième plus puissant au monde en 2023 selon le classement du TOP'500 [1]. Il est donc important de garder en tête que les développements décrits dans ce chapitre sont aussi, et en partie, guidée par notre volonté de déployer les approches proposées de façon effective en utilisant les ressources de calculs offertes par un tel supercalculateur, massivement parallèle et large échelle.

4.2 Architecture parallèle haut niveau

DRILS est un algorithme séquentiel faisant évoluer une seule solution suivant une certaine trajectoire dans l'espace de recherche. De ce fait, si l'on veut paralléliser DRILS, l'approche doit se penser à l'échelle de la solution. Une première idée est donc de se concentrer sur les composantes de DRILS (recherche locale et croisement). La parallélisation de la recherche locale a déjà été étudiée dans la partie précédente. Comme nous l'avons déjà mentionnée, au vu des accélérations que nous avons pu obtenir dans le chapitre précédent, il ne semble pas intéressant de continuer sur cette voie dans un cadre de calcul massivement parallèle et large échelle. En ce qui concerne le croisement par partition, celui-ci est réalisé en temps linéaire et n'offre pas de possibilités de parallélisme suffisamment prometteuses pour être considérées. Pour résumer, la parallélisation de l'algorithme séquentiel DRILS n'est pas évidente pour les raisons suivantes :

— Les solutions sont évaluées très rapidement, il suffit de lire la contribution de chaque sous-fonction dans une table et de les additionner. C'est d'ailleurs l'un des points les plus fort de l'algorithme difficilement améliorable.
— Les composantes de DRILS sont extrêmement efficaces en séquentiel et n'offrent pas de degrée de parallélisme suffisant pour rendre une approche parallèle compétitive.
— Une itération de DRILS est très rapide et repose sur l'enchaînement de composantes qui nécessitent la complétion de l'étape précédente.

1. www.top500.org

En conséquence, nous nous concentrerons dans un premier temps sur une stratégie haut niveau dans notre élaboration d'une version parallèle de DRILS. En suivant la terminologie de [15], nous opterons pour une méthodologie dite de *marches multiples*. Dans notre cas, la marche correspondra à une itération de DRILS. Notre objectif est alors de proposer un algorithme permettant de dépasser les performances obtenues par l'approche parallèle la plus naïve qui consiste à lancer plusieurs exécutions séquentielles de DRILS indépendamment les unes des autres. Bien que très simple, cette approche naïve apporte de la diversité à DRILS en visitant une plus grande zone de l'ensemble de l'espace de recherche. Nous utiliserons donc cette méthode afin de comparer les différentes approches que nous proposerons et étudierons par la suite.

Pour le succès de nos développements, il nous faut donc proposer des règles de coopération parallèles entre les différents composants de DRILS afin de tirer efficacement profit de la puissance de calcul mise à disposition. Bien que les aspects liés à la conception d'approches coopératives parallèles aient été bien étudiés dans le passé, et que le corpus de travaux relatifs aux problématiques sous-jacentes soit en évolution croissante[15,18], l'élaboration d'algorithmes à base de marches multiples massivement parallèles reste une tâche très difficile qui nécessite une connaissance fine de la classe de problèmes que l'on souhaite résoudre et de ces spécificités. Autrement dit, il n'existe pas, à notre connaissance, une seule approche, standard et universellement efficace, pour la conception et le déploiement de tels algorithmes parallèles, et ce, de façon systématique. Dans la suite, nous commençons donc par énoncer nos choix de conception, avant de détailler de façon incrémentale, et la plus pédagogique possible, les différentes approches développées.

4.2.1 Choix architecturaux et technologiques

Dans toutes les approches développées dans ce chapitre, nous considérerons deux niveaux de parallélisme, chacun associé à une technologie différente :

— Au sein d'un processeur, il s'agit d'un parallélisme entre les cœurs le composant. Ce niveau de parallélisme utilise Open MP (Multi Processing) et fonctionne en mémoire partagée ;
— Entre les processeurs, il repose sur la technologie MPI (Message Passing Interface) et se base sur l'échange de données entre ces deniers par le biais de messages.

De façon générale, chaque cœur (et le thread associé) sera en charge d'effectuer des itérations de DRILS, chaque cœur faisant donc évoluer la trajectoire de sa propre solution. Les communications, qu'elles soient au sein d'un environnement en mémoire partagée ou bien en échange de messages, consisteront uniquement en l'envoi et en la réception de la meilleure solution trouvée par l'un des threads. Nous détaillerons cela par la suite. Pour le moment, gardons simplement en tête que nos architectures impliquent l'échange de meilleures solutions entre les différents threads exécutants des itérations de DRILS

en parallèle et de façon concurrente et coopérative. Il est à noter que dans ce cadre, l'échange d'une solution fait référence à la fois à la représentation binaire ; mais aussi aux informations nécessaires au bon déroulement des composantes boîte grise telles que le score d'un mouvement, les structures associées aux sous-fonctions, etc. Ceci afin d'éviter pour un thread de les recalculer lors des communications.

Le partage de la meilleure solution peut impacter fortement la dynamique de recherche d'un algorithme s'exécutant en parallèle. En particulier, DRILS dispose en effet d'un mécanisme d'échappement qui lui permet de ne pas stagner dans des zones de l'espace de recherche ; quand il ne parvient pas, par le biais de son opérateur de croisement, à construire de nouvelles solutions de meilleures qualités, ou bien de qualités égales. Le lecteur est invité à se rappeler que la version séquentielle de DRILS croise deux parents obtenus par recherche locale, et que le second parent est retenu pour l'itération suivante en cas d'échec d'amélioration. Ainsi, cela permet de lutter contre la stagnation de la recherche, et en théorie de visiter une plus grande zone de l'espace de recherche. Cependant, en supposant que différents threads exécutent des trajectoires différentes de DRILS en parallèle, le fait de les synchroniser entre eux à travers l'échange de la meilleure solution trouvée vient en quelque sorte contrebalancer la diversité de la recherche que pourrait offrir l'utilisation parallèle de plusieurs threads. Cela revient à déplacer la zone de recherche pour tous les threads en un point donné de l'espace de recherche, et ainsi à intensifier la recherche aux alentours de cette solution pour un certain temps. Le mécanisme d'échappement et le caractère stochastique de la perturbation viennent par la suite rajouter de la diversité à la recherche globale.

En conséquence, il n'est pas trivial de trouver un équilibre au compromis exploration versus exploitation. Nous avons mentionné la version indépendante, qui d'un point de vue haut niveau, privilégie au maximum l'exploration. Tandis qu'on peut aisément concevoir une approche cherchant à synchroniser le plus vite possible tous les threads parallèles autour de la meilleure solution ; cette version privilégierait, d'un point de vue haut niveau, l'exploitation. Dans la suite, nous proposerons une approche basée sur cette idée ; nous verrons qu'elle lève de nouvelles questions en termes d'architecture et de dynamique de recherche.

4.2.2 Une première approche hiérarchique maître-travailleur

L'architecture la plus communément utilisée, et la plus directe, pour partager la meilleure solution à large échelle est d'utiliser un processus maître chargé de recevoir les meilleures solutions et de les diffuser aux travailleurs. Ainsi, nous proposons une approche asynchrone, détaillée dans le paragraphe suivant et résumée à travers le schéma de la figure 4.1.

Cette première approche prend en compte deux niveaux de parallélisme : Le plus haut

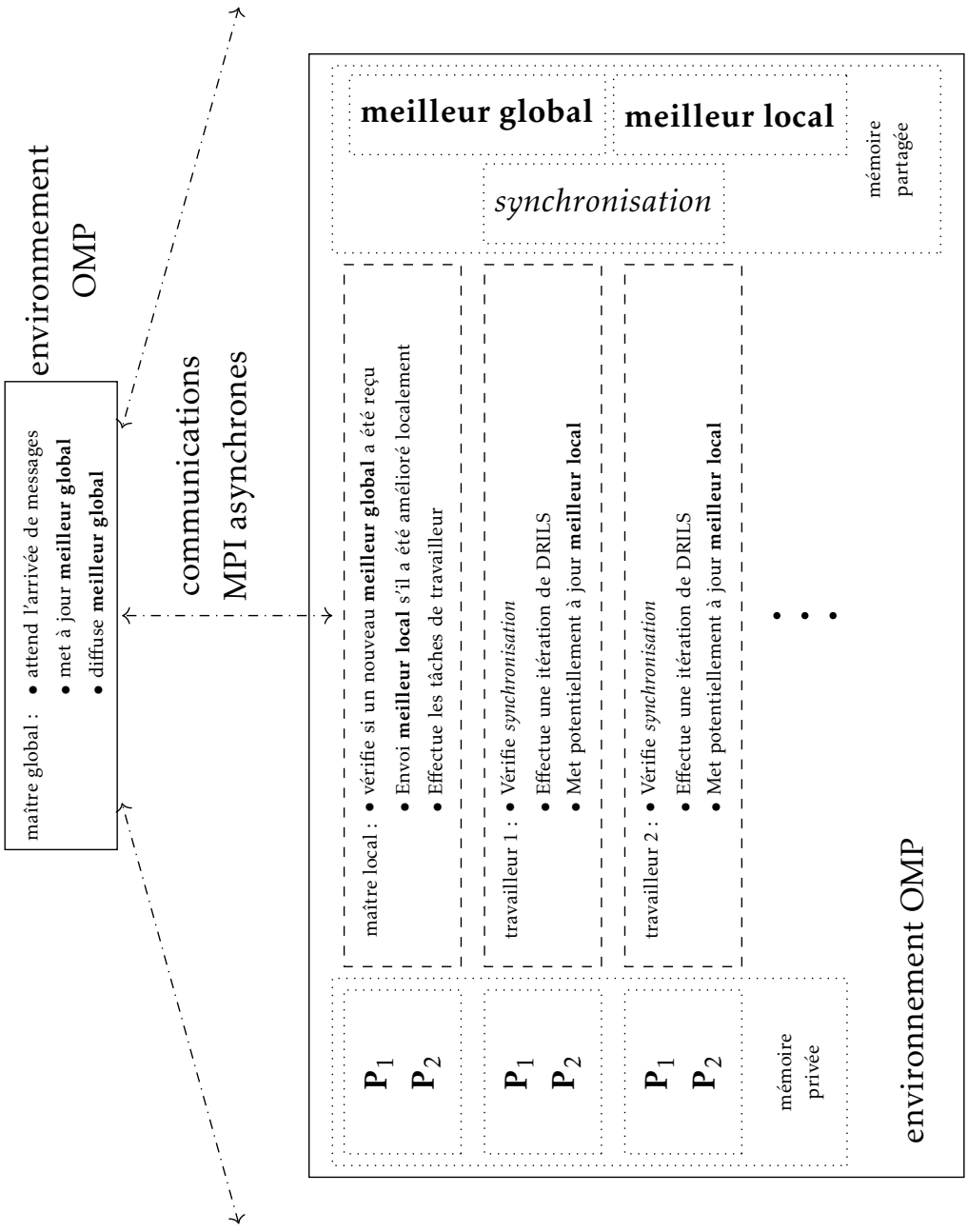

FIGURE 4.1 – *Architecture parallèle hiérarchique maître-travailleurs, combinant OMP et MPI de façon asynchrone (en italique un tableau de booléens, en gras les solutions).*

niveau concerne les échanges entre processeurs. L'un d'eux se voit en effet attribuer le rôle de *maître global*. C'est ce maître qui s'occupe de la réception et de la diffusion des solutions aux autres, les processeurs travailleurs. Au niveau des processeurs ayant le rôle de travailleurs, chaque thread effectue des itérations de DRILS avec sa solution courante locale. Une fois une nouvelle (meilleur) solution reçue, les threads se synchronisent localement sur celle-ci et continuent leur recherche de façon indépendante. Au sein d'un

même environnement en mémoire partagé, l'un des threads des processeurs travailleurs est dédié au rôle de *maître local* qu'il cumule en plus de ses taches de travailleur. Le maître local s'occupe de vérifier si l'un des threads présent localement autour de lui (dans l'environnement OMP) a trouvé une nouvelle meilleure solution. Dans ce cas, il ordonne aux autres threads OMP locaux de se synchroniser avec cette dernière. Ensuite, il l'envoie au maître global. Le maître local se charge aussi de recevoir les solutions diffusées par le maître global. L'intégralité des communications est faite de façon asynchrone.

La gestion de la synchronisation à l'échelle du processeur, et de son environnement OMP en mémoire partagée, est gérée par un tableau de booléen (appelé 'synchronisation' dans la figure 4.1). À chaque fois qu'une nouvelle solution est reçue, et si les threads doivent se synchroniser, alors le maître local change les valeurs du tableau de synchronisation. Avant d'effectuer une itération de DRILS, chaque thread vérifie l'état de synchronisation à l'indice lui correspondant : s'il doit se synchroniser, alors il le fait puis remet le booléen dans son état initial.

Comme discutée précédemment, une telle architecture implique le choix assumé d'intensifier la recherche dans le voisinage de la meilleure solution, puisqu'elle est partagée globalement et immédiatement par tous les threads du système. Ce choix de conception peut-être sous-optimal car manquant de diversité. Un compromis entre cette approche et l'approche totalement indépendante est donc une alternative intéressante à considérer. De plus, cette approche très hiérarchique peut poser des problèmes purement techniques. En effet, lorsque le nombre de processeurs devient important, le processeur maître global, en charge de recevoir toutes les solutions, peut se retrouver submergé sous un flot de messages asynchrone. Un tel goulot d'étranglement peut ralentir la vitesse d'exécution globale. Pour ces raisons, nous proposons dans la section suivante une approche distribuée utilisant un modèle dit de modèle en îles.

4.2.3 Approche décentralisée et modèle en îles

L'architecture sous forme d'îles [6] permet de distribuer l'effort de recherche, et par conséquent la puissance de calcul, de façon décentralisée entre plusieurs groupes de processeurs. Le graphe représentant l'ensemble des groupes de processeurs (les îles) ainsi que leurs interactions possibles peut être construit suivant différentes topologies de graphe. Les îles peuvent ainsi communiquer avec leurs voisins directs et uniquement ceux-ci tels que défini par la topologie du graphe de communication considéré. Il est bien connu que la façon dont on défini ce graphe, et les règles de communications utilisées, peuvent avoir un impact important sur la performance. Dans ce travail, nous étudierons les deux topologies suivantes communément utilisées : une topologie de communication aléatoire, et une topologie sous forme d'anneau. La première que nous abrégerons en topologie aléatoire dans la suite du document consiste à tirer au sort un voisin de façon équiprobable à chaque

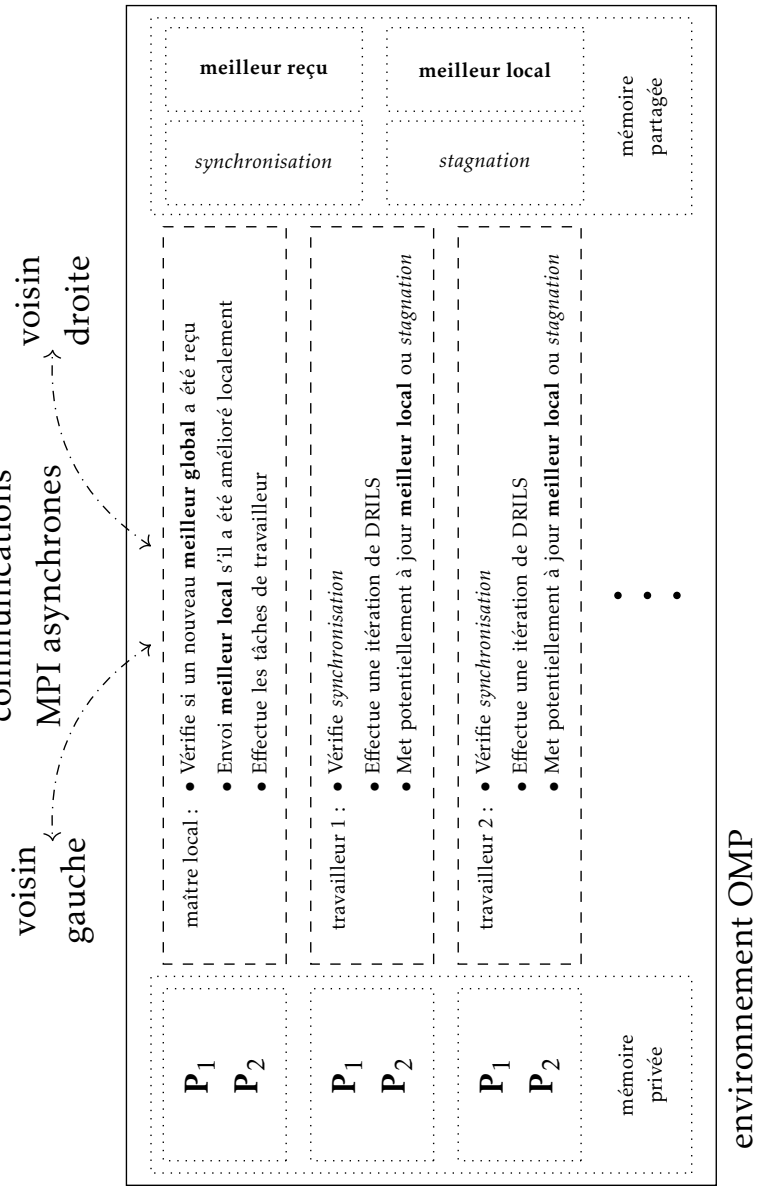

FIGURE 4.2 – *Architecture distribuée sous forme d'îles avec les règles de communications R1 et R2. Implementé en OMP et MPI (en gras les solutions, en italique les tableaux de booléens).*

fois qu'une île doit envoyer un message. Il s'agit donc d'un graphe dynamique qui évolue à chaque communication. Comme son nom l'indique, dans la seconde topologie en anneau, chaque île est connectée à deux autres îles de façon à ce que le graphe induit suive la forme d'un anneau/cycle simple.

Cette approche, décentralisée en île, rend obsolète l'utilisation d'un processeur maître global dédié à la propagation des messages vers tous les autres. Cette tâche étant maintenant effectuée de manière implicite et décentralisée par chaque processeur. Ainsi, chaque processeur se comportera comme une île, et le fonctionnement au niveau de l'île reste iden-

tique à celui des processeurs travailleurs définis dans le modèle hiérarchique. Seul diffère la façon avec laquelle les communications par échanges de messages (MPI) sont effectuées. Dans la figure 4.2, on peut voir les deux voisins avec lesquels une île communique dans le cadre d'une topologie en anneau.

Si l'on considère uniquement les nouvelles topologies en conservant la même politique de propagation de la meilleure solution que dans le modèle hiérarchique, on se rend compte que la seule différence est la disparition du goulot d'étranglement que l'on pouvait avoir autour du maître global. Les communications étant maintenant réparties plus équitablement. En outre, la meilleure solution finit toujours par se propager à l'ensemble des îles, en un temps plus ou moins long suivant la topologie. Ainsi, ces variantes décentralisées restent des approches très orientée "exploitation", et concentre la puissance de calcul autour de la dernière meilleure solution trouvée, modulo le temps qu'elle met à se propager entre les différentes îles. Dans la section suivante, nous introduisons un ensemble de nouvelles règles qui établissent un compromis plus fin entre exploration et exploitation tout en réduisant encore davantage le nombre de communications, afin de limiter au maximum la charge de travail induite par le transfert des solutions entre les îles.

4.2.4 Une politique de communication avancée pour le modèle en île

Commençons par la première règle visant à minimiser les communications sans altérer de façon substantielle la dynamique haut niveau globale de l'architecture parallèle en îles. Nous nommerons cette règle R1 dans la suite du document, pour plus de lisibilité.

Pour comprendre la motivation et la nécessité de cette règle, il est important de se pencher sur la dynamique de DRILS dans les premiers moments de la recherche. Les modèles de problèmes d'optimisation sur lesquelles nous travaillons sont très rugueux, avec peu de plateaux, et les composantes boîte grise utilisées par DRILS sont très rapides. En conséquent, dans les premières phases de la recherche, la meilleure solution est très souvent améliorée dans un très petit intervalle de temps. Si on prend en compte le nombre de threads travaillant chacun à améliorer la solution actuelle et le délai de synchronisation qui existe en raison de l'implémentation asynchrone, on peut se retrouver avec une quantité très importante de messages envoyés au cours des premiers moments de la recherche. Ceci peut entraîner un ralentissement général de la recherche.

Pour illustrer cela, prenons un exemple simple et considérons deux threads A et B. Supposons que A améliore la meilleure solution et que B améliore également la meilleure solution légèrement après A. Peu importe que la solution de A soit meilleure ou pas que celle de B, comme B et A n'ont pas eu le temps de se synchroniser (et donc de savoir qui est meilleur et quelle solution devrait être propagée ou non), les deux solutions risquent fortement de se retrouver en transit dans le réseau d'îles. Ceci n'est pas en soit un problème dans cet exemple, mais si l'on considère plusieurs milliers ou dizaines de

milliers de threads travaillant simultanément (ce qui est le cas dans ce chapitre), plus le fait qu'on ne communique pas qu'une solution brute, mais également les données nécessaires au fonctionnement des algorithmes boîte grise, alors on comprend facilement qu'il est intéressant de tenter de réduire ces communications.

La première règle de communication R1 que nous proposons stipule simplement qu'il faut attendre que tous les threads de l'île aient échoué au moins une fois à améliorer la nouvelle meilleure solution trouvée localement après une itération de DRILS avant de communiquer. Cette règle est gérée à l'aide d'un tableau de booléens (stagnation dans la figure 4.2). Cette règle est expressément introduite afin de limiter les communications. Elle fut conçue dans le but de pallier des problèmes de ralentissement observées en raison du grand nombre de messages à échanger pendant les premières phases de la recherche.

Passons maintenant à la règle suivante que nous proposons, et que nous appelons la règle R2. Celle-ci a pour but de ralentir la propagation de la meilleure solution, et ce dans le but de conserver davantage de diversité dans la recherche entre les différentes îles. Jusqu'à maintenant, quand une île recevait une solution, elle la propageait de façon systématique aux îles voisines, et ainsi de suite. Ceci a l'effet de permettre la propagation rapide d'une meilleure solution trouvée dans une île donnée, sous réserve qu'elle soit meilleure que les autres trouvées/reçues localement par les autres îles. Une telle solution finit donc par très rapidement s'imposer à toutes les îles. La règle R2 peut alors se résumer ainsi : la propagation d'une solution d'une île a une autre ne se fait que si l'île a été capable d'améliorer cette solution localement. Ainsi, nous changerons la nomenclature de meilleure solution globale, à meilleure solution reçue pour désigner la meilleure solution jamais reçue par une île. En effet, il faut maintenant comprendre qu'en raison de cette nouvelle règle, la meilleure solution jamais reçue par cette île est fort probablement différente et possiblement moins bonne que la meilleure des meilleures solutions trouvées par l'ensemble des îles.

Il est important d'insister sur le fait que cette règle introduit un équilibre différent entre exploitation et exploration, car la limite de propagation d'une solution introduit un biais qui favorise la propagation des solutions qui sont capables d'être améliorées plus souvent. Prenons l'exemple de deux solutions A et B et d'une topologie en anneau. Supposons que A et B ont été trouvées dans des îles très éloignés du réseau, mais qu'ils aient exactement la même fitness. Supposons que A soit améliorée plusieurs fois, mais que chaque amélioration soit petite, et que B soit améliorée une seule fois, mais suffisamment pour que les qualités de A et B soient les mêmes. La solution A et ces versions améliorées auront ainsi été propagées plus de fois que celle de B qui ne l'aura été qu'une fois ; de sorte que d'avantage d'îles travailleront dans une zone plus proche de A que de B (si l'on pose l'hypothèse que A et B sont éloignées et que les améliorations successives n'ont pas entraînées de si gros changements sur les solutions). Si l'on considère un grand nombre

	Hiérarchique		Modèles en îles			
	Maître/Travailleurs		Aléatoire		Anneau	
R1	✗	✓	✗	✓	✗	✓
R2	—		✗	✓	✗	✓
label	M/T	M/T+	IAlea	IAlea+	IAnn	IAnn+

Table 4.1 – *Résumé des combinaisons des topologies et des règles de communications étudiées*

d'îles, couplé au fait que la difficulté à trouver des améliorations croît exponentiellement au fur et à mesure que la recherche progresse, alors la règle R2 devrait permettre de conserver un niveau de diversité relativement important au sein des différentes îles, tout en intensifiant localement la recherche sur des zones qui ont été capables dans le passé de fournir le plus d'améliorations successives.

4.2.5 Combinaisons entre règles de communications et topologies

L'idée de proposer des approches sous forme d'îles est venue en réponse aux problèmes qu'introduisait l'approche hiérarchique maître-travailleurs. Cependant, la volonté de créer des versions parallèles intermédiaires, entre les approches purement exploration et purement exploitation, couplée à certains problèmes techniques, nous a amené à proposer de nouvelles règles de communication : les règles R1 et R2. En considérant le nombre de combinaisons possibles entre les règles (2 règles soient 4 combinaisons) et les 3 architectures proposées (hiérarchique, îles avec topologie aléatoire ou en anneau), nous nous retrouvons avec 12 combinaisons d'algorithme parallèles. En pratique, seulement la moitié est intéressante à considérer. En effet, la règle R2 couplée au modèle hiérarchique n'a pas de sens. De même, pour les modèles en îles, nous considérons uniquement le cas où les deux règles sont activées ou non en même temps. Ainsi, nous étudierons les combinaisons détaillées dans le tableau 4.1 à savoir : hiérarchique avec ou sans R1 (notées dans la suite M/T, M/T+), et aléatoire ou anneau avec ou sans R1 et R2 (noté dans la suite IAlea, IAnn et IAlea+ IAnn+, le + servant à indiquer l'utilisation des règles de communications avancées). Au total, six versions de conception parallèle de DRILS sont proposées et étudiées de façon expérimentale dans la suite.

4.2.6 Supercalculateur Fugaku et protocole expérimental

Avant de décrire le protocole expérimental, il est important de clarifier un aspect technique important lié à l'utilisation du supercalculateur Fugaku [29, 31] dans nos expérimentations.

L'architecture du Fugaku est basée sur un système NUMA 'Non uniform memory access', soit accès mémoire non-uniforme. Ainsi, un processeur du Fugaku correspond en réalité à 4 plus petites unités disposant chacune de sa mémoire et de 12 cœurs physiques de calculs. Ils sont appelés nœuds NUMA, et sont plus ou moins équivalent à un processeur classique. Pour lever toute ambiguïté, nous parlerons de nœuds NUMA à la place de processeurs.

Afin d'étudier les différentes approches que nous avons présentées précédemment, nous avons choisi 9 configurations de paysages NKQ large échelle et difficiles. La taille et le degré d'interactions, respectivement N et k, des instances considérées appartiennent aux ensembles suivantes $N = \{1\ 000,\ 5\ 000,\ 10\ 000\}$, $k = \{3,\ 4,\ 5\}$, la valeur de Q a été fixée à 64, suivant ainsi la valeur standard utilisée dans la littérature. Afin de comparer de façon rigoureuse les variantes entre elles et en prenant compte du comportement stochastique de DRILS, nous appliquerons le protocole suivant. Chaque algorithme est lancé 10 fois avec une graine différente. Les exécutions durent 15 minutes. Étant donné que l'on s'intéresse également à la scalabilité, nous avons utilisé un nombre croissant de nœuds NUMA {4, 16, 64, 256}. Ainsi, dans cette première étude, nous utiliserons au maximum $256 * 12 = 3\ 072$ cœurs physiques de calculs.

L'algorithme DRILS reposant sur un hyperparamètre de perturbation, noté α, nous avons pris soin de tester pour chaque instance et pour chaque configuration un large ensemble de valeurs possibles que voici : $\alpha = \{0.01,\ 0.025,\ 0.05,\ 0.1,\ 0.15\}$. Cette phase de calibrage a été réalisée sur un nœud du Fugaku ce qui correspond à 4 nœuds NUMA de 12 cœurs chacun. Rappelons aussi que pour des problèmes NKQ aussi difficiles, la solution optimale n'est pas connue en pratique. Ainsi, pour comparer les algorithmes entre eux, nous utiliserons la valeur de l'écart de fitness relatif, noté rfd (de l'anglais 'relative fitness deviation'). Cette dernière mesure l'écart entre la meilleure solution trouvée par un algorithme (disons x) et la meilleure solution jamais trouvée (c.à.d. la meilleure solution connue à défaut de la solution optimale exacte) pour l'instance en question (disons x^*). Plus formellement, la rfd se calcule avec la formule suivante :

$$rfd(x) = \frac{f(x^*) - f(x)}{f(x^*)}$$

Ainsi plus cette valeur est proche de 0, meilleure est la qualité de la solution x, une valeur de 0 indiquant que la solution est la meilleure jamais trouvée pour cette instance.

4.2.7 Résultats

Dans un premier temps, référons-nous à la figure 4.3 dans le but d'analyser la scalabilité des approches proposées, ainsi que l'impact combiné des différentes topologies et des politiques de communication. Dans un souci de lisibilité, les résultats sont agrégés suivant les différentes valeurs de k.

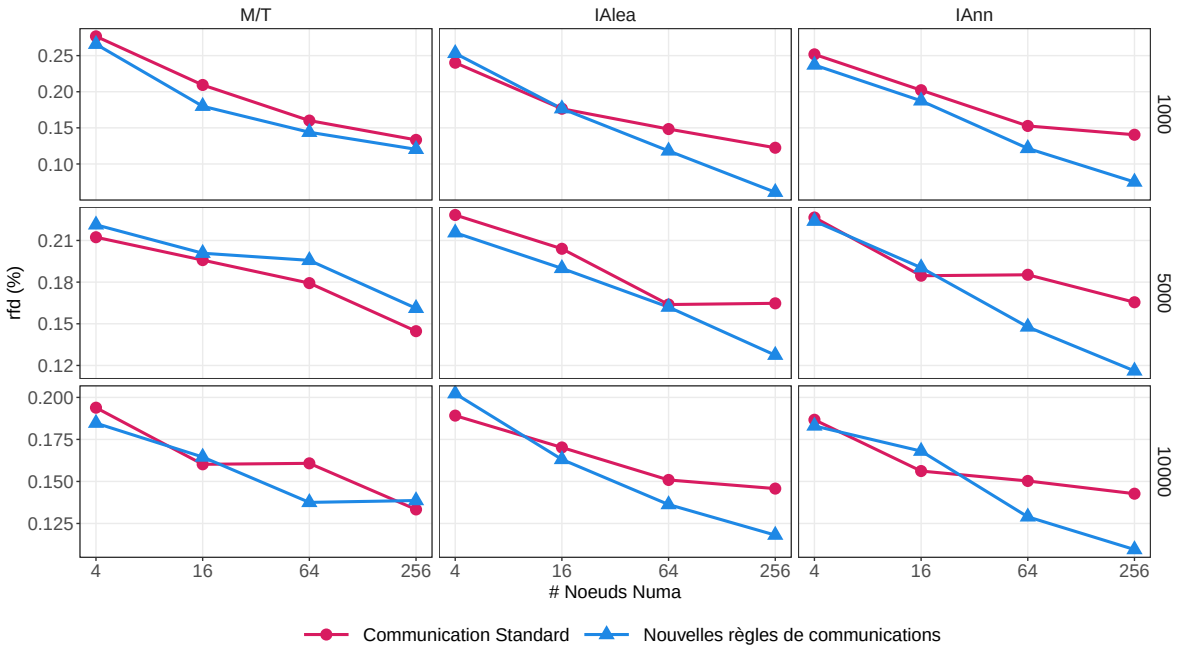

FIGURE 4.3 – *Impact des règles de communications. Rfd (en %) pour le modèle Maître/Travailleur (M/T) et les modèles en îles (IALEA pour la topologie aléatoire, IANN pour celle en anneau) avec et sans les nouvelles règles de communications proposées. De haut en bas, la taille des instances varie de* 1 000 *à* 10 000, *les valeurs de k sont quant à elles agrégées.*

La première observation est clairement que l'ajout de puissance de calcul, en termes de nœuds NUMA, permet d'augmenter la qualité des solutions trouvées peu importe la topologie ou la politique de communication. On remarque que les modèles en îles sont en moyenne meilleurs que le modèle hiérarchique. Maintenant, considérons l'impact des règles R1 et R2. Pour l'approche hiérarchique, l'impact de R1 est mitigé et dépend de l'instance considérée. En ce qui concerne les modèles en île, l'impact des règles R1 et R2 est beaucoup plus bénéfique à mesure que l'on ajoute des ressources de calcul. On peut donc conclure que l'introduction d'un délai dans la propagation de la meilleure solution, par le biais des règles de communications, à un impact bel et bien positif.

Dans la suite, nous considérerons toujours les variantes avec les nouvelles règles de communications activées, R1 pour hiérarchique, R1 et R2 pour les modèles en îles. Ainsi, l'abréviation des différents modèles sera suivie d'un + pour caractériser cet ajout (M/T+, IAnn+, IAlea+).

Dans la figure 4.4, nous comparons la qualité des solutions trouvées par le modèle hiérarchique M/T+ et les modèles en îles (IAnn+, IAlea+) par rapport à la version séquentielle (Seq) et à la version naïve, indépendante (Ind) de DRILS. Toujours dans un souci de visibilité les données sont agrégées, suivant N cette fois.

Comme nous pouvions nous y attendre, toutes les variantes parallèles, y compris celle naïve avec des DRILS indépendants, offrent de meilleurs résultats que la version

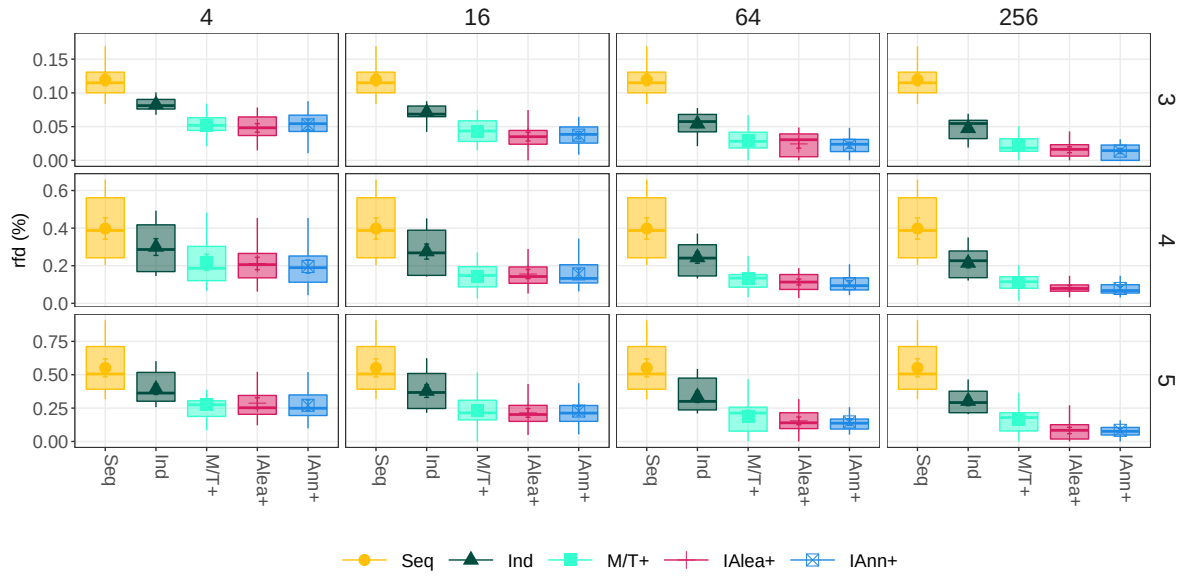

FIGURE 4.4 – *Performance des différentes approches proposées basé sur la rfd (en %). Les 5 versions sont : DRILS séquentiel (Seq), DRILS indépendant (Ind), l'approche Maître/Travailleurs avec la règle R2 (M/T+) et les approches en îles avec les règles R1 et R2 (IAlea+ et IAnn+ suivant la topologie). Chaque colonne correspond à un nombre de nœuds NUMA allant de* 4 *à* 256*, et chaque ligne représente un niveau de rugosité défini par une valeur de k allant de* 3 *à* 5*. La taille des instances est agrégée.*

séquentielle. Plus on augmente le nombre de nœuds NUMA, plus les performances sont bonnes. Concernant la variante indépendante qui sert de base de comparaison pour nos approches parallèles, on peut voir que peu importe la variante contre laquelle on la compare, elle est beaucoup moins bien que cette dernière. En ce qui concerne les expériences avec un nombre réduit de nœuds NUMA, on voit que la variante M/T+ obtient des résultats similaires aux modèles en îles (IAnn+ et IAlea+). Ces derniers creusant cependant l'écart en leurs faveurs à mesure que l'on augmente la puissance de calcul, et sont ceux qui performent le mieux sur tous les problèmes avec 256 nœuds NUMA. Ceci montre que l'approche décentralisée en îles est une bonne solution lorsque l'on atteint un certain seuil de puissance de calcul. Notons cependant qu'il ne semble pas y avoir d'écart notable en termes de performance entre IAnn+ et IAlea+, ce qui indique que la topologie n'a que peu d'importance en comparaison aux règles de communications R1 et R2 qui, elles, bénéficient énormément à la qualité des approches en îles.

4.3 Une approche coopérative hybride de DRILS

Dans cette partie, nous proposons d'introduire un nouveau niveau de collaboration en ajoutant une couche de parallélisme au niveau de l'environnement en mémoire partagée.

4.3.1 Motivation

Dans les approches proposées précédemment, la coopération consistait en l'envoi d'une solution si celle-ci permettait d'améliorer soit globalement la meilleure solution, soit localement dans le cas des approches distribuées. Comme le montraient nos résultats expérimentaux, l'utilisation d'un modèle en îles combinée à une politique de communication avancée, permet d'obtenir des résultats qui dépassent de loin ceux que peut obtenir une version naïve, indépendante et sans coopération. Cela prouve, du moins empiriquement, que la coopération est une composante essentielle pour la conception d'un algorithme massivement parallèle efficace. Cependant, quand on atteint une certaine qualité de solution et qu'il devient très difficile de continuer à trouver des améliorations, cette forme simple de coopération n'est plus suffisante, et la dynamique de la recherche se rapproche d'un DRILS indépendant, car la condition nécessaire à l'échange de message est justement de trouver une solution améliorant la meilleure solution. On peut donc légitimement se demander s'il est possible d'introduire un nouveau degré de coopération qui resterait constant peu importe l'état actuelle de la recherche ? Et, si oui, est ce qu'une telle approche pourrait être bénéfique comme le fut la coopération introduite dans la partie précédente ? Cette section répond précisément à ces questions ; et ceci en deux étapes.

Premièrement, nous introduisons un nouveau niveau de parallélisme qui permet une coopération plus fine entre les différents threads d'un environnement en mémoire partagé. Deuxièmement, nous incorporons un tel niveau de coopération dans le cadre du modèle en îles avec les règles de communications R1 et R2 activées, et ce, en testant expérimentalement l'approche ainsi obtenue sur un large nombre d'instances et en utilisant jusqu'à 1 024 nœuds NUMA du Fugaku, soit 12 288 cœurs physiques (1 024 nœuds de 12 cœurs).

4.3.2 Coopération asynchrone dans l'itération de DRILS

Considérons un scénario simple où nous disposons de deux threads exécutant des itérations de DRILS en parallèle. Au niveau de l'itération, chaque thread a besoin de deux optima locaux (les deux parents) pour effectuer un croisement. Disons que ces deux threads commencent avec le même parent que l'on notera p. En partant de ce parent, chacun des 2 threads effectuera les opérations classiques de DRILS, une perturbation aléatoire suivie d'une recherche locale. Bien que la recherche locale peut être déterministe, le comportement stochastique de la perturbation aléatoire implique que les (deuxièmes) parents trouvés par chaque thread seront très certainement différents. De plus, ces deux threads ne termineront très probablement pas leurs calculs en même temps. Supposons sans perte de généralité que le premier thread termine en premier. Notons le nouveau parent qu'il a trouvé p_2^1. Ce thread dispose désormais de tout ce dont il a besoin pour

continuer son itération de DRILS : il lui suffit désormais de croiser p et p_2^1. Une fois ce croisement terminé, on parle ici du croisement suivi de la recherche locale et du critère d'acceptation, ce thread va commencer une nouvelle itération en partant de la nouvelle solution obtenue avec le croisement, notons la p'. Considérons qu'entre temps le deuxième thread ait terminé sa recherche locale, il a donc une solution p_2^2 prête pour être utilisée par l'opérateur de croisement. Il se pose maintenant la question suivante : est-il préférable de croiser p_2^2 avec p ? C'est à dire comme DRILS le ferait classiquement, ou bien alors peut-on envisager plutôt d'utiliser p' ? Auquel cas, on profiterait, si p' était meilleur que p, d'un croisement entre deux meilleures solutions. Comme ces dernières ont toutes les deux été obtenues en partant de p et en appliquant une petite perturbation, elles devraient toujours satisfaire la contrainte de proximité nécessaire pour le bon fonctionnement du croisement boîte grise (PX). En effet, il est empiriquement connu que le croisement a plus de chance de fonctionner avec des parents relativement proches. Nous avons ici utilisé un exemple de deux threads partageant une même solution (premier parent) de départ ; mais ce scénario, et les questions qu'il soulève, pourrait se généraliser à davantage de threads.

Pour résumer, et en se basant sur les questions soulevées dans l'exemple précédent, notre idée est de faire travailler plusieurs threads de façon coopérative au sein de petits groupes dans lesquels : (i) les recherches locales sont faites parallèlement de façon asynchrone et (ii) les croisements sont effectués itérativement sur une même solution partagée par tous les threads de ce groupe, en l'occurrence le premier parent. Il faut cependant faire attention à certains aspects algorithmiques et techniques. Premièrement, même s'il est possible d'effectuer des recherches locales en parallèle, car chacun des threads travaille sur la construction d'une solution sans interaction avec les autres threads, on doit faire attention à la phase de croisement, car elle modifie la solution partagée. Il faut donc apporter un soin particulier à l'implémentation et bloquer l'accès en lecture et écriture à la solution courante partagée dés lors que : (i) un thread est soit en train de copier cette solution pour commencer un nouveau cycle de perturbation et de recherche locale, ou (ii) lorsqu'un thread est en train de l'utiliser pour faire un croisement. Auquel cas la solution sera nécessairement altérée durant le processus (en raison du critère d'acceptation notamment). Deuxièmement, on doit faire attention au temps passé à attendre pour avoir accès à la solution partagée. À cet effet, on ne souhaite pas ralentir le processus de recherche dans sa globalité en créant des groupes trop grands. Auquel cas le temps d'attente deviendrait probablement contre-productif. Notons qu'en pratique la recherche locale prend 6 à 8 fois le temps d'un croisement. Ainsi, tant que la taille du groupe reste raisonnable, ce problème n'en est pas vraiment un. Troisièmement, et cela rejoint le problème précédent, supposons un groupe de 20 threads, il se peut que le dernier thread doive croiser sa solution avec la solution courante qui aura précédemment était croisée 20 fois. Dans ce cas-là, il devient difficile de garantir une quelconque proximité entre les solutions qui vont être croisées.

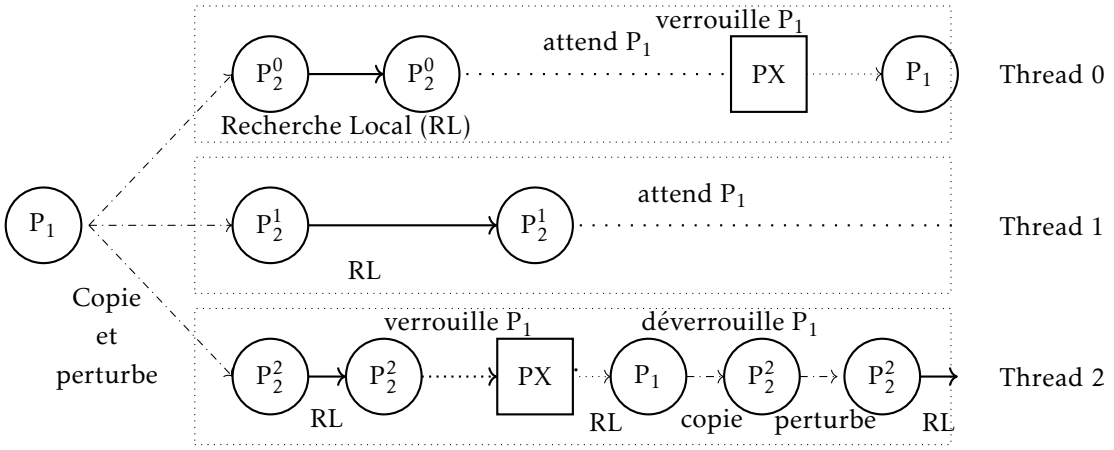

FIGURE 4.5 – *DRILS coopératif et asynchrone avec un sous-groupe composé de 3 threads. Le premier parent (i.e P_1) est partagé, les seconds parents (i.e les P_2^i) sont privés. Le troisième thread est le premier à compléter sa recherche locale ; il est donc le premier à pouvoir verrouiller P_1, il effectue son croisement suivi d'une recherche locale, puis copie P_1 dans P_2^2 avant de déverrouiller P_1. Pendant ce temps, les autres threads sont, soit en train d'attendre, soit de finir leur recherche locale. Notons que l'axe du temps n'est pas représentatif et sert juste de support pour l'explication.*

On comprend donc qu'avant même de penser à l'implémentation d'un tel algorithme coopératif au niveau des itérations de DRILS, il faut trouver un juste-milieu entre des groupes trop grands qui poseraient des problèmes pour le croisement, et des groupes trop petits, qui ne profiteraient pas pleinement de la capacité d'enchaîner très rapidement des croisements successifs.

Dans la suite, nous détaillons l'implémentation de cette idée de coopération entre itérations de DRILS au niveau mémoire partagée, ainsi que son incorporation dans le modèle en îles décrit précédemment dans ce chapitre.

4.3.3 DRILS coopératif et asynchrone

Pour détailler le fonctionnement de notre nouvelle proposition d'algorithme DRILS parallèle et coopératif, nous nous appuyons sur la figure 4.5 illustrant la coopération en mémoire partagée au sein d'un même groupe de threads OMP. Notons que l'exemple suppose une taille de groupe égale à 3 threads. Le premier parent, p_1, est partagé par tous les threads. Celui-ci est obtenu au tout début de l'exécution en partant d'une solution aléatoire suivie d'une recherche locale. Cette solution sera utilisée en tant que premier parent dans tous les croisements effectués par les threads de ce groupe.

Chaque thread copie p_1 dans une solution stockée dans sa mémoire privée (p_2^0, p_2^1...). Cette solution privée va ensuite être perturbée puis utilisée pour obtenir un nouvel optimum local par le biais d'une recherche locale. Considérons maintenant le premier thread à terminer sa recherche locale. Supposons que c'est le thread 2 pour suivre l'exemple

de la figure. Ce thread est donc prêt à effectuer un croisement sur p_1. Cependant, comme p_1 est dans la mémoire partagée, il faut s'assurer qu'aucun autre thread n'est en train de le modifier ou de le copier. Pour cela, on utilise un verrou OMP sur p_1. Le fonctionnement technique est alors très simple. Si le verrou n'est pas activé, alors n'importe quel thread peut l'activer, et ainsi avoir l'exclusivité sur la ressource jusqu'à ce qu'il n'en ait plus besoin auquel cas le verrou est désactivé. Les threads voulant utiliser cette ressource verrouillée doivent attendre qu'elle soit déverrouillée.

Revenons à notre exemple, le thread 2 accède donc à la ressource p_1 en premier et bloque le verrou correspondant. Ce même thread peut désormais croiser sa solution privée p_2^2 avec p_1 et ainsi modifier p_1 puis le recopier dans sa solution privée p_2^2 avant de libérer le verrou sur p_1, et repartir sur un nouveau cycle de perturbation et de recherche locale. Comme on peut le voir sur la figure, le thread 0 a fini sa recherche locale pendant que le thread 2 effectuait son croisement. Le thread 0 tente donc d'accéder à la solution p_1, mais en raison du verrou, il doit attendre que le thread 2 ait terminé. Une fois le verrou désactivé, le thread 0 itère à son tour le croisement, et ainsi de suite.

4.3.4 Modèle en îles coopératif et asynchrone

Voyons maintenant comment incorporer cette nouvelle version de DRILS dans un modèle en îles qui, rappelons le, est le modèle qui munie des règles de communication R1 et R2 obtenait les meilleurs résultats dans nos expériences précédentes. Pour cela, nous utilisons la figure 4.6 qui résume notre architecture complète.

Le cadre définit précédemment au niveau des interactions entre nœuds NUMA ne change pas étant donné que la modification décrite ici ne touche que l'exécution coopérative des itérations de DRILS au sein d'un même (petit) groupe de threads. Nous faisons donc le choix d'utiliser une topologie en anneau, et l'intégration de ce nouveau niveau de parallélisme est quasi-immédiat. Chaque île contient désormais plusieurs sous-groupes travaillant chacun de façon coopérative, telle que discuté précédemment. Nous retrouvons en particulier un maître local chargé des communications suivant les règles R1 et R2, la mémoire privée et la mémoire partagée contenant les mêmes données que précédemment, à l'exception de la solution p_1. En effet, dans les versions précédentes, chaque thread avait sa version privée de p_1. Maintenant, chaque sous-groupe de threads dispose simplement de sa solution partagée p_1. L'accès à ces p_1 se fait de façon restrictive uniquement aux membres du même sous-groupe de threads correspondants, et avec l'utilisation d'un verrou tel qu'expliqué dans la section précédente. Le maître local cumule donc ce rôle en plus de sa participation à un sous-groupe au sein de la même île.

Si une nouvelle meilleure solution est trouvée par un sous-groupe, alors le maître locale la propage et les autres sous-groupes de threads se synchronisent en suivant les mêmes règles que précédemment. Il en va de même si la solution provient d'une communication

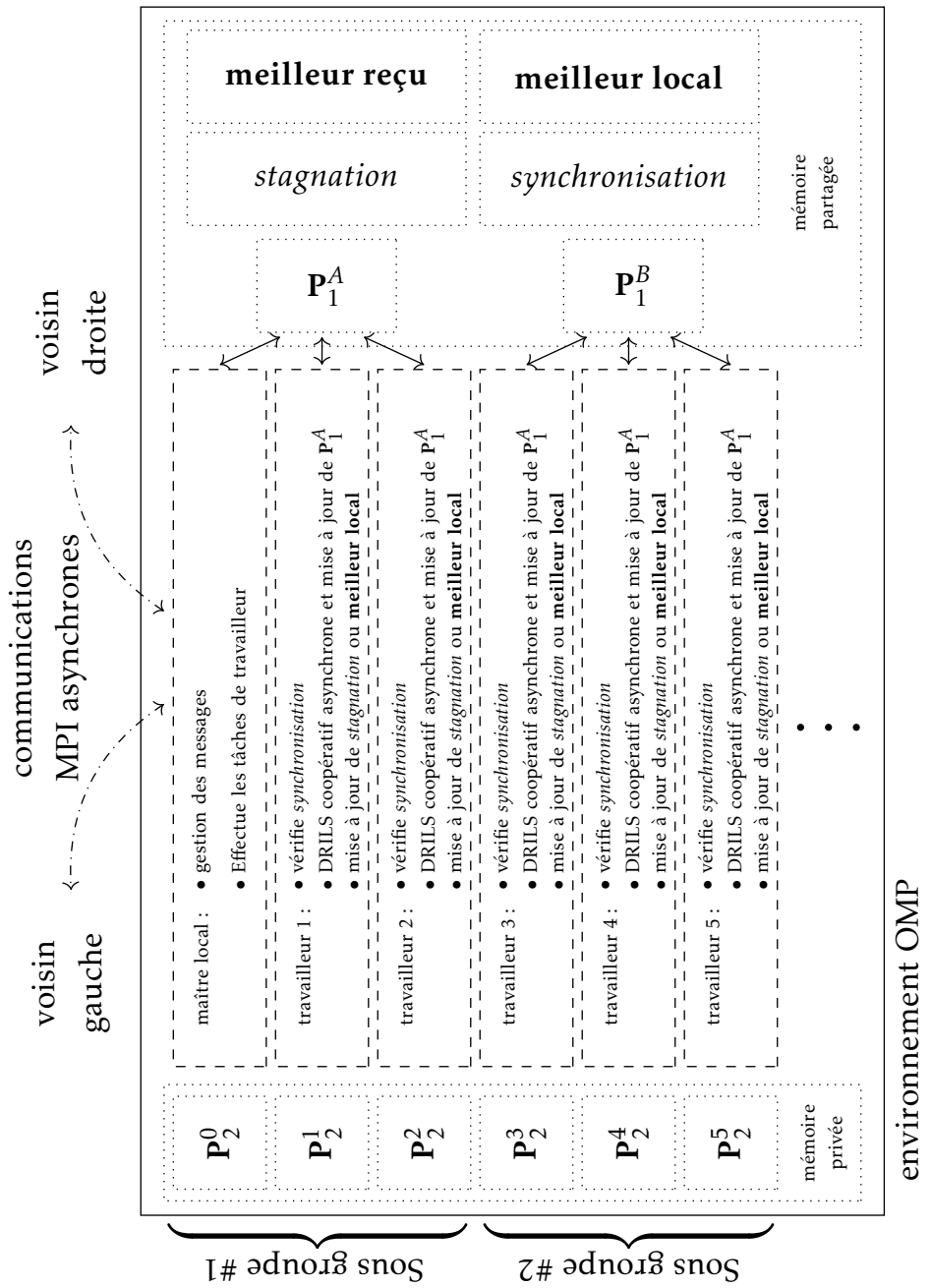

FIGURE 4.6 – *Architecture distribuée sous forme d'îles avec les règles de communications R1 et R2 utilisant la version coopérative et asynchrone de DRILS. Implémenté en OMP et MPI (en gras les solutions, en italique les tableaux de booléens).*

avec un nœud voisin. Rappelons que toutes les communications entre sous-groupes d'un même nœud et au sein d'un sous-groupe sont gérées via l'utilisation de la mémoire partagée.

Ceci étant, il est important d'apporter la plus grande vigilance à un point technique, qui peut avoir un impact algorithmique important. Quand une nouvelle solution est reçue, si

celle-ci remplace p_1 et qu'on ne fait rien de plus, alors quand le premier thread, terminant une recherche locale après que p_1 ait été modifiée, va vouloir effectuer un croisement, il concernera des parents fort probablement très éloignés. En effet, le deuxième parent aura très probablement été généré en partant du parent partagé ; mais qui aura été remplacé entre temps. On ne peut donc garantir à quel point le nouveau p_1 était proche ou pas de celui qu'il a remplacé. On pourrait se dire à première vue qu'on a juste perdu une itération de DRILS dans le pire des cas, et que de toute façon avec cette nouvelle version de DRILS, on peut les enchaîner encore plus rapidement. Mais il faut prendre en considération le critère d'acceptation ; ce dernier stipule que si le croisement n'a pas été bénéfique alors on remplace la solution courante par le dernier optimum local obtenu. Sans attention ni traitement supplémentaire, ceci peut alors avoir un impact critique. De façon plus détaillée, on se retrouverait en effet dans une situation où on avait un p_1 au moment où un thread l'a copié pour générer un p_2. Entre temps, ce p_1 a été croisé un certain nombre de fois. Appelons p_1' la solution résultante de ce processus. Maintenant, supposons qu'une nouvelle meilleure solution soit reçue et qu'elle remplace p_1', appelons la p_1". Autant nous pouvions garantir que p_1' et p_2 serait relativement proche, autant, on ne peut pas s'appuyer sur cette hypothèse pour p_1" et p_2. Si le croisement entre p_1" et p_2 n'est pas prolifique, la solution p_2 va être copiée dans p_1". On voit ainsi que la solution reçue est écrasée par une solution qui nous ramène dans l'endroit de l'espace de recherche dans lequel on était avant de recevoir cette nouvelle meilleure solution, annulant donc une grande partie les effets de la coopération inter-îles. Pour éviter cette situation critique, nous décidons donc, lors de l'étape de synchronisation, de forcer les threads à générer une nouvelle solution en partant de la *nouvelle* meilleure solution reçue, et ainsi éviter de l'écraser par inadvertance. Ceci termine la description de notre nouvelle version coopérative de DRILS.

4.3.5 Analyse expérimentale

Description du protocole

Pour réaliser nos analyses expérimentales, nous nous basons sur le même protocole que décrit précédemment. Cependant, nous considérons une condition d'arrêt de 1 heure d'exécution. Nous étendons également le nombre de nœuds NUMA en considérant des expériences utilisant jusqu'à 1024 nœuds, de manière à avoir une idée plus détaillée sur la capacité de passage à l'échelle des différentes variantes parallèles. La phase de calibrage des paramètres change également quelque peu ; car nous devons maintenant considérer le paramètre, noté Γ, correspondant au nombre de threads par sous-groupe. Étant donné que les environnements en mémoire partagée fonctionnent sur 12 cœurs en raison de l'architecture physique NUMA du Fugaku (mais également pour les contraintes sur la

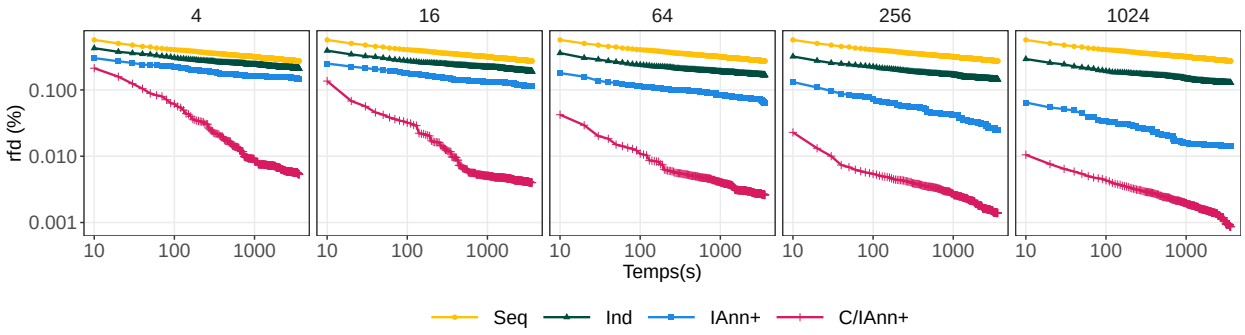

FIGURE 4.7 – *Profil de convergence de DRILS séquentiel (Seq), DRILS indépendant (Ind) et des modèles en îles utilisant une topologie en anneau et les règles de communication R1 et R2, se basant sur DRILS classique (IAnn+) ou DRILS coopératif et asynchrone (C/IAnn+) . L'axe y représente la rfd (en %, échelle logarithmique) pour différents nombre de coeurs NUMA (de 4 à 1024) en fonction du temps de calcul (en secondes, échelle logarithmique). Les différentes instances sont agrégées.*

taille optimale des sous-groupes exposées précédemment), nous considérerons les valeurs suivantes : $\Gamma = \{2, 3, 4, 6\}$.

En nous appuyant sur les résultats déjà obtenus dans nos expérimentations précédentes, nous nous concentrerons sur les 4 algorithmes suivants : DRILS séquentiel (seq) ; DRILS indépendant (ind) ; l'approche en île avec une topologie en anneau et les règles R1 et R2 activées avec DRILS (IAnn+) ou bien avec la nouvelle version de DRILS coopératif que nous venons de décrire (C/Iann+)

Comparaison globale des performances

Afin de comparer les performances des différentes approches les unes par rapports aux autres, nous avons calculé la rfd des solutions obtenues à la fin du temps imparti. Nous nous basons sur un test de Wilcoxon-Mann-Whitney avec un seuil de significativité de 0.05 pour comparer les approches par paires, et ainsi pouvoir les classer. Nous utiliserons pour cela leur rang, que nous définirons comme étant, pour une approche donnée, le nombre d'autres approches ayant obtenues un meilleur score que cette dernière. Ainsi, un rang 0 signifie qu'aucun autre algorithme n'a obtenu de meilleurs résultats et un rang de 3 signifie que tous les autres algorithmes sont meilleurs (car nous comparons 4 approches).

Le tableau 4.2 montre le rang ainsi que la rfd moyenne pour chaque approche, et ceci pour chaque instance et nombre de nœuds NUMA considérés. Il est frappant de voir que l'approche C/Iann+ surpasse globalement et de loin les autres, suivie par Iann+. Ceci apporte une preuve expérimentale du bien-fondé de l'ajout d'un niveau de coopération au sein même d'une itération de l'algorithme DRILS. Si l'on analyse plus finement les résultats, on voit que C/Iann+ et Iann+ ne sont pas statistiquement différents dans (seulement) 3 cas, pour les problèmes de plus petite taille (1000) et avec un très grand nombre de noeuds

N	k	noeuds NUMA	Seq		Ind		IAnn+		C/IAnn+	
			rang	rfd	rang	rfd	rang	rfd	rang	rfd
1000	3	4	3	0.12810	2	0.05159	1	0.03129	**0**	**0.00085**
		16	3	0.12810	2	0.04271	1	0.02008	**0**	**0.00000**
		64	3	0.12810	2	0.02896	1	0.05920	**0**	**0.00000**
		256	3	0.12810	2	0.01966	**0**	**0.00000**	**0**	**0.00000**
		1024	3	0.12810	2	0.01163	**0**	**0.00000**	**0**	**0.00000**
	4	4	3	0.58562	2	0.40221	1	0.26719	**0**	**0.00020**
		16	3	0.58562	2	0.34029	1	0.16557	**0**	**0.00000**
		64	3	0.58562	2	0.29000	1	0.12891	**0**	**0.00000**
		256	3	0.58562	2	0.24377	1	0.08899	**0**	**0.00000**
		1024	3	0.58562	2	0.21505	1	0.03890	**0**	**0.00000**
	5	4	3	0.74178	2	0.52786	1	0.36187	**0**	**0.00000**
		16	3	0.74178	2	0.46848	1	0.26383	**0**	**0.00000**
		64	3	0.74178	2	0.44237	1	0.12773	**0**	**0.00000**
		256	3	0.74178	2	0.35469	1	0.03109	**0**	**0.00000**
		1024	3	0.74178	2	0.30169	**0**	**0.00000**	**0**	**0.00000**
5000	3	4	3	0.12159	2	0.08696	1	0.07395	**0**	**0.01283**
		16	3	0.12159	2	0.07768	1	0.05163	**0**	**0.00966**
		64	3	0.12159	2	0.07196	1	0.03948	**0**	**0.00610**
		256	3	0.12159	2	0.05917	1	0.02804	**0**	**0.00310**
		1024	3	0.12159	2	0.05824	1	0.02181	**0**	**0.00203**
	4	4	3	0.40187	2	0.32495	1	0.24953	**0**	**0.03518**
		16	3	0.40187	2	0.30485	1	0.19454	**0**	**0.03514**
		64	3	0.40187	2	0.28729	1	0.14052	**0**	**0.02242**
		256	3	0.40187	2	0.26563	1	0.11517	**0**	**0.00779**
		1024	3	0.40187	2	0.25035	1	0.09425	**0**	**0.00347**
	5	4	3	0.62489	2	0.48940	1	0.38042	**0**	**0.10340**
		16	3	0.62489	2	0.45724	1	0.28796	**0**	**0.08816**
		64	3	0.62489	2	0.40964	1	0.24761	**0**	**0.04784**
		256	3	0.62489	2	0.39591	1	0.21883	**0**	**0.02830**
		1024	3	0.62489	2	0.36828	1	0.18309	**0**	**0.01096**
10000	3	4	3	0.10742	2	0.08045	1	0.05779	**0**	**0.00759**
		16	3	0.10742	2	0.07565	1	0.05020	**0**	**0.00490**
		64	3	0.10742	2	0.06618	1	0.03668	**0**	**0.00374**
		256	3	0.10742	2	0.06096	1	0.02936	**0**	**0.00252**
		1024	3	0.10742	2	0.05967	1	0.02596	**0**	**0.00118**
	4	4	3	0.27109	2	0.18812	1	0.14857	**0**	**0.04661**
		16	3	0.27109	2	0.18568	1	0.14694	**0**	**0.03788**
		64	3	0.27109	2	0.16891	1	0.09805	**0**	**0.02399**
		256	3	0.27109	2	0.16307	1	0.08484	**0**	**0.01001**
		1024	3	0.27109	2	0.15279	1	0.07292	**0**	**0.00663**
	5	4	3	0.53350	2	0.35681	1	0.28710	**0**	**0.12339**
		16	3	0.53350	2	0.32304	1	0.25017	**0**	**0.06838**
		64	3	0.53350	2	0.30785	1	0.17890	**0**	**0.03109**
		256	3	0.53350	2	0.26401	1	0.16691	**0**	**0.02476**
		1024	3	0.53350	2	0.26954	1	0.13791	**0**	**0.01945**

Table 4.2 – *Rfd moyenne (en %) et rang des différentes approches comparées, obtenu à l'aide d'un test de Wilcoxon avec un seuil de significativité de 0.05. En gras la meilleure approche par nombre de noeuds et par instance.*

NUMA (256 et 1024 pour k=3, 1024 pour k=5). Cependant, on remarque également que ces résultats peuvent être obtenus par C/Iann+ avec une puissance de calcul beaucoup

plus réduite. Pour ces instances, la scalabilité de C/Iann+ est relativement faible ; mais ces performances sont impressionnantes même à la plus petite échelle. Ainsi, pour cette taille d'instance, peu importe la valeur de k, on peut noter que 16 nœuds NUMA permettent avec C/IAnn+ d'obtenir des résultats au moins aussi bons, voir de loin meilleur, que ceux de Iann+ utilisant 1024 nœuds NUMA ; soit avec 64 fois moins de ressources.

En s'intéressant maintenant aux instances les plus difficiles, on voit que C/IAnn+ conserve une très bonne capacité de recherche. Pour toutes les instances considérées, bien que C/IAnn+ soit meilleur avec 4 nœuds NUMA comparé à IAnn+ avec 1024, il continue à s'améliorer avec l'ajout d'avantage de nœuds NUMA. Ceci même quand on atteint 1024 nœuds, soit plus de 10 000 cœurs, sans montrer aucun signe de stagnation. Nous reviendrons sur cet aspect dans une prochaine section ; mais étudions avant cela le comportement des différentes approches tout au long de leur exécution.

Profil de convergence

Nous nous intéressons ici à la dynamique de la recherche et plus précisément à l'évolution dans le temps de la moyenne des qualités des meilleures solutions trouvées. Pour cela, nous nous appuyons sur la figure 4.7 qui montre le profil de convergence des différentes approches en fonction du nombre de nœuds NUMA. Dans un souci de lisibilité, les données de toutes les instances ont été agrégées. Notons également que la version séquentielle, bien que fonctionnant avec 1 seul cœur, a été rajoutée dans toutes les (sous-) figures dans le but de servir de point de comparaison.

On peut ainsi voir l'impressionnante différence qui se creuse, puis se maintient voir même s'élargit, tout au long de l'exécution entre C/IAnn+ et les autres algorithmes. Il est en particulier frappant de noter la vitesse avec laquelle la qualité des solutions trouvées par C/IAnn+ dépasse celle que IAnn+ va mettre 1 heure à trouver.

Accélération Parallèle

En se basant sur l'analyse des profils de convergences, on peut affirmer que la force de C/IAnn+ ne vient pas seulement de sa capacité de passage à l'échelle en terme du nombre d'unités de calculs. En fait, les résultats sur un petit nombre de nœuds NUMA nous montrent que l'écart de performance vient principalement du nouveau mécanisme de coopération au sein des petits sous-groupes NUMA. La nouvelle dynamique de recherche introduites par cette approche rend la notion d'accélération parallèle très difficile à calculer. Pour définir une telle accélération, il nous faut trouver une valeur cible, afin de comparer le temps est nécessaire aux différentes approches pour l'atteindre. Or, comme nous l'avons vue, l'écart entre C/IAnn+ et IAnn+ est extrêmement grand. D'un côté, si l'on prend comme cible les meilleurs scores obtenus par IAnn+, alors ces cibles sont beaucoup trop simples pour C/IAnn+ ; et l'accélération obtenue ne reflétera donc pas de façon juste la

k	ϵ	N	4		16		64		256		1024	
			acc	nacc	acc	nacc	acc	nacc	acc	nacc	acc	nacc
3	$5 \cdot 10^{-5}$	1000	1.00	1.00	6.64	1.66	33.50	2.28	36.50	0.57	36.50	0.14
		5000	1.00	1.00	9.33	2.33	70.56	4.41	112.90	1.76	112.90	0.44
		10000	1.00	1.00	2.81	0.70	8.44	0.53	16.64	0.26	24.96	0.10
		moyenne	1.00	1.00	**6.26**	1.56	**37.5**	2.41	**55.35**	0.86	**58.12**	0.23
4	$2.5 \cdot 10^{-4}$	1000	1.00	1.00	2.44	0.61	32.00	2.00	53.33	0.83	64.00	0.25
		5000	1.00	1.00	1.96	0.49	4.50	0.28	11.26	0.18	28.68	0.11
		10000	1.00	1.00	2.35	0.59	3.92	0.25	7.53	0.12	13.91	0.05
		moyenne	1.00	1.00	**2.25**	0.56	**13.47**	0.84	**24.04**	0.38	**35.52**	0.14
5	$5 \cdot 10^{-4}$	1000	1.00	1.00	1.36	0.34	4.15	0.26	14.82	0.23	32.60	0.13
		5000	1.00	1.00	0.77	0.19	2.44	0.15	3.11	0.05	4.94	0.02
		10000	1.00	1.00	2.05	0.51	3.89	0.24	5.31	0.08	7.80	0.03
		moyenne	1.00	1.00	**1.39**	0.35	**3.49**	0.22	**7.75**	0.12	**15.11**	0.06
		moyenne			**3.3**	0.82	**18.15**	1.16	**29.1**	0.45	**32.92**	0.14

TABLE 4.3 – *Ratio d'accélération moyen acc et nacc, pour un nombre croissant de noeuds NUMA. les lignes sont regroupées par valeur de k. La qualité cible est fixée à* $(1-\epsilon)\overline{f}_{4noeuds}$ *; i.e., l'accélération vaut* 1 *pour 4 noeuds NUMA. L'accéleration moyenne des différentes tailles d'instances est affichée en gras. Les accélérations normalisées super-linéaire sont soulignées.*

différence de performances entre les deux approches. D'un autre côté, si l'on prend les meilleures solutions obtenues par C/IAnn+ comme cible, alors IAnn+ ne pourra pas les atteindre en un temps raisonnable ; et ce même si on fixe un temps de calcul très restreint pour C/IAnn+ comme vue dans la section précédente.

Ainsi, bien qu'il soit difficile de comparer l'accélération parallèle des différentes approches entre elles, il est possible d'étudier l'accélération parallèle d'une approche en se basant sur les résultats obtenus avec un nombre réduit de cœurs. Cela demande de choisir avec précaution les cibles que nous allons définir et également de prendre en compte le caractère stochastique des algorithmes utilisés. Avant de poursuivre, et pour éviter toute ambiguïté, définissons de façon plus formelle les notions d'accélération (acc) et d'accélération normalisée (nacc) que nous utiliserons dans la suite. Pour cela, considérons deux algorithmes A et B, travaillant sur la même instance. Notons p_A et p_B le nombre de nœuds NUMA utilisés par chaque algorithme (avec $p_A \geq p_B$). Supposons maintenant donnée une qualité de solution cible qui soit atteignable par ces deux algorithmes, et appelons la f^*. L'accélération de A relativement à B ainsi que l'accélération normalisée correspondant sont définies de la façon suivante :

$$\begin{aligned} \mathsf{acc}(f^\star, A, B) &= \frac{T(f^\star, B)}{T(f^\star, A)} \\ \mathsf{nacc}(f^\star, A, B) &= \frac{p_B}{p_A} \cdot \mathsf{acc}(f^\star, A, B) \end{aligned}$$

Où $T(f^*, \cdot)$ correspond au temps moyen nécessaire à l'algorithme pour trouver une solution avec une qualité au moins égale à f^*.

En tant que base de comparaison, nous utiliserons C/IAnn+ avec 4 nœuds NUMA. Nous étudierons donc l'accélération de C/IAnn+ avec 16, 64, 256 et 1024 nœuds NUMA par rapport à ce même algorithme utilisant 4 nœuds. Dans le but de satisfaire la condition nécessaire d'atteignabilité de la cible f^*, nous nous devons de prendre une cible qui soit atteignable à coup sûr peu importe le nombre de nœuds. Rappelons que le caractère aléatoire des algorithmes ne garantit pas qu'à temps de calcul égal, C/IAnn+ avec 16 nœuds soit capable d'obtenir une qualité au moins aussi bonne que C/IAnn+ avec 4 nœuds. D'un autre côté, choisir une valeur trop facile ne serait pas représentatif et léserait l'accélération obtenue avec les plus grands nombres de nœuds. Pour résoudre ce dilemme, il est possible de choisir une cible qui soit très légèrement inférieure au meilleur score moyen obtenu avec 4 nœuds, de sorte à garantir qu'avec 16 nœuds et plus, C/IAnn+ soit toujours capable d'atteindre cette cible. Ainsi, nous définirons f^* comme étant $(1-\epsilon) \cdot \bar{f}_{4\text{nodes}}$, avec $\bar{f}_{4\text{nodes}}$ la qualité moyenne obtenue par C/IAnn+ avec 4 nœuds NUMA après 1 heure de calcul. Concernant ϵ, nous utiliserons une valeur dépendante de k. En effet, nous avons expérimentalement noté que plus k croît, plus la dispersion des qualités obtenues après une heure de calcul est grande. Ainsi, nous voulons que la valeur de ϵ soit proportionnelle à celle de k afin que la difficulté des cibles soit inversement proportionnelle à k. Les valeurs suivantes sont donc utilisées : pour $k = 3$, $\epsilon = 5 \cdot 10^{-5}$; pour $k = 4$, $\epsilon = 2.5 \cdot 10^{-4}$ et pour $k = 5$, $\epsilon = 5 \cdot 10^{-4}$;

Il est important de rappeler que ces cibles sont très difficiles et hors de portée de la version naïve de DRILS indépendant parallèle, et très rarement atteintes par IAnn+ (uniquement pour les problèmes les plus simples). L'interprétation des accélérations calculées spécifiquement pour C/IAnn+, avec un nombre variable de ressources, doit donc être appréciée à sa juste valeur, c'est-à-dire, sachant que les autres algorithmes parallèles n'auraient jamais pu (ou très difficilement dans de rares exécutions) atteindre les même cibles utilisées pour calculer ces accélérations.

Dans le tableau 4.3, nous montrons l'accélération et sa variante normalisée obtenue par C/IAnn+. D'un côté, on peut voir que l'accélération est fonction de l'instance considérée. Plus l'instance est difficile, c'est-à-dire plus le problème est rugueux et le nombre de variables important, moins l'accélération est élevée. Cela n'est pas surprenant puisque dans un environnement d'optimisation parallèle, augmenter les ressources de calculs n'implique pas systématiquement que le problème peut être résolu plus rapidement, particulièrement dans le cadre de problème (et de cibles) très difficiles. D'un autre côté,

l'accélération est globalement importante quand on regarde la difficulté des instances considérées, avec parfois des accélérations supra-linéaire. Ceci indique clairement que l'accélération n'est pas seulement due à l'augmentation de la puissance de calcul, mais très probablement à : la dynamique différente induite par notre conception, à la gestion du compromis exploration versus exploitation, ainsi qu'à l'aspect coopératif de notre nouvelle version parallèle de DRILS.

Sur les instances les moins rugueuses, soit $k = 3$, l'accélération augmente régulièrement jusqu'à atteindre une valeur de 58. Pour les problèmes les plus rugueux, soit $k = 5$, l'accélération maximale constatée dépend de la taille des instances. Ainsi pour $N = 1\,000$, on obtient une accélération de 32, contre 7 pour celle à 10 000 variables. En moyenne, C/IAnn+ obtient une accélération de 33 si l'on considère le plus grand nombre de nœuds NUMA soit 1024. Ceci est relativement impressionnant si l'on prend en compte que cette accélération est obtenue en se comparant à la version de C/IAnn+ utilisant 4 noeuds, qui rappelons le, est capable d'obtenir des résultats inatteignables par les autres approches, et ce même avec une puissance de calcul 256 fois supérieure.

Finalement, il est intéressant de noter qu'à l'exception des 2 instances les plus simples considérées, C/IAnn+ ne montre pas de signe de stagnation et ce même avec 1024 nœuds NUMA soit 12 288 cœurs de calcul.

Impact des hyper-paramètres

Dans un souci de complétude, nous concluons notre analyse par l'étude de l'impact des hyper-paramètres sur la qualité des solutions obtenues par C/IAnn+. Rappelons que DRILS dispose d'un hyper-paramètre, α qui correspond à l'intensité de la perturbation, et que notre version coopérative introduit un nouvel hyper-paramètre Γ correspondant à la taille des sous-groupes de threads. La figure 4.8 illustre la rfd moyenne obtenue en fonction de α pour à la fois l'algorithme DRILS séquentiel, et pour l'algorithme C/IAnn+ parallèle avec des sous-groupes de taille 2 et 6.

La première observation est que les meilleurs résultats sont obtenus avec la plus grande taille de sous-groupes. On voit également que DRILS est de base sensible à la valeur de α et que cette sensibilité est similaire pour des petits sous-groupes, mais exacerbée lorsque leur taille augmente. Il est aussi à noter que la forme des courbes représentant la performance en fonction de α est la même entre DRILS et C/IAnn+ pour les sous-groupes de taille 2. Cette dernière est simplement translatée vers le bas (indiquant de meilleures performances pour C/IAnn+). Si l'on examine maintenant les sous-groupes de taille 6, on voit que la valeur optimale de α n'est plus la même.

En résumé, on voit que l'hyper-paramètre Γ introduit dans C/IAnn+ peut faire varier de façon très importante la qualité de la recherche, bien que les résultats soient meilleurs peu importe la valeur expérimentée de cet hyper-paramètre. Pour des sous-groupes de petites

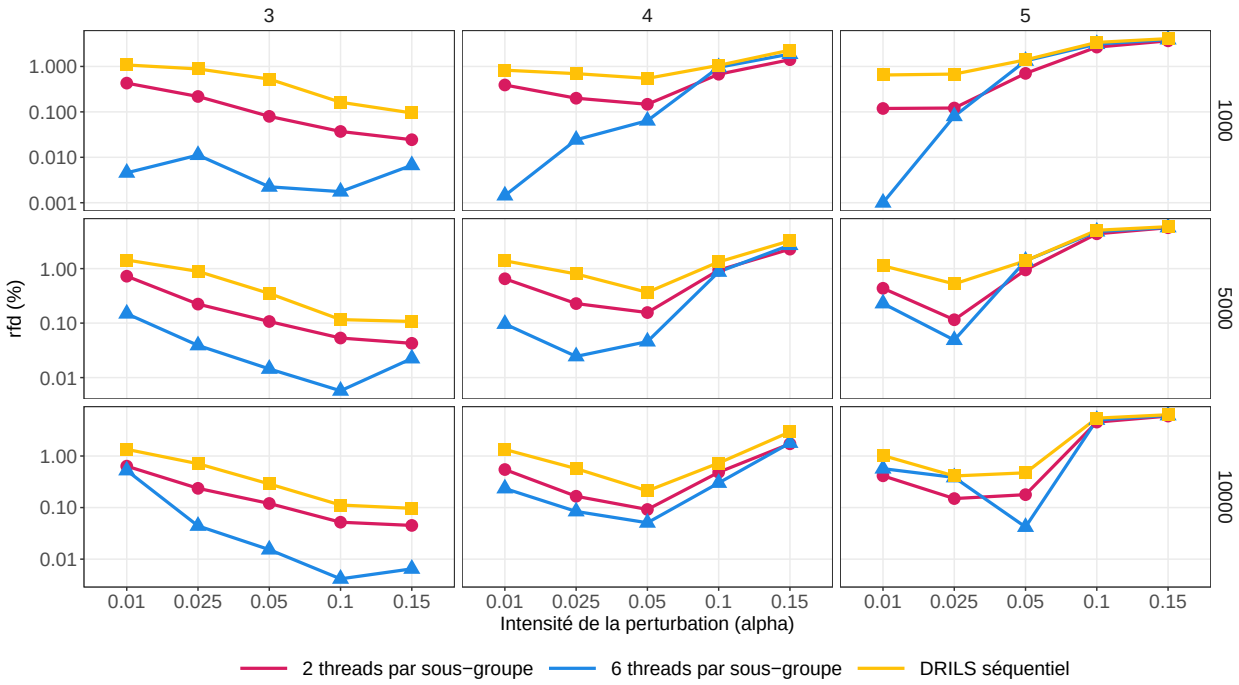

FIGURE 4.8 – *Rfd (en %) pour DRILS séquentiel et C/IAnn+ en fonction de l'intensité de la perturbation α, et de la taille des sous-groupes. Ces résultats sont ceux obtenues durant la phase de calibrage tel que défini dans la partie description du protocole. de gauche à droite, la rugosité du paysage $k = 3$, 4 et 5. de haut en bas, le nombre de variables $N = 1000$, 5000 et 10000.*

tailles Γ, les valeurs optimales de la perturbation α sont les mêmes que la version classique séquentielle de DRILS. On peut ainsi suivre les recommandations de la littérature à ce sujet. Néanmoins, pour obtenir de meilleurs résultats, il faut s'orienter vers des tailles de sous-groupes plus conséquents, mais avec la nécessité de passer par une phase de calibration pour obtenir la valeur optimale de α. En pratique, cette phase peut être bien souvent accélérée en choisissant de commencer par des valeurs de α proches de celles proposées dans la littérature. Ceci sort cependant du périmètre de notre étude, et est laissé comme une intéressante perspective de recherche.

4.4 Conclusion et perspectives

En partant de DRILS, l'algorithme séquentiel état de l'art pour les paysages NK à large échelle, nous avons proposé une approche massivement parallèle basée sur un modèle décentralisé en îles, et utilisant des règles de communications avancées permettant l'élaboration d'une stratégie coopérative de haut niveau capable de trouver un équilibre efficace entre exploration et exploitation. Nous avons par la suite enrichi cette approche en proposant un autre niveau de parallélisme plus bas niveau, cette fois-ci au sein même de DRILS. Cette variante parallèle coopérative et asynchrone offre une dynamique nouvelle à

DRILS. L'ensemble formant une approche capable d'obtenir des résultats impressionnants avec un nombre restreint de processeurs et sur un nombre important d'instances très difficiles de paysages NKQ. En outre, nous avons eu l'occasion de montrer la capacité de scalabilité de notre approche sur l'un des plus puissants supercalculateurs du monde, le Fugaku. En se basant sur des expériences et un protocole rigoureux, nous avons montré que notre approche pouvait efficacement tirer parti d'une puissance de calcul allant jusqu'à 12 288 cœurs sans signes de stagnation sur les instances les plus difficiles.

Ce travail conclu donc la première partie de nos contributions de thèse, dédiée à nos travaux pour la résolution de problèmes d'optimisation boîte grise dans un contexte de calcul parallèle. Comme nous l'avons décrit, l'approche massivement parallèle proposée dans ce chapitre peut se voir comme un framework de haut-niveau utilisant l'algorithme séquentiel DRILS comme brique de base. Ainsi, toute amélioration de la version séquentielle de DRILS, et/ou tout nouvel opérateur de recherche locale et/ou de croisement, peuvent potentiellement être utilisés et intégrer de façon immédiate afin d'obtenir de meilleures performances.

Dans la seconde partie de ce manuscrit, nous proposons d'étudier plus précisément la dynamique de recherche de l'algorithme séquentiel DRILS, ainsi qu'un certain nombre de variantes qui améliorent sa conception. Comme nous allons le voir, les contributions de cette deuxième partie sont donc de nature différente en comparaison aux travaux décrits jusqu'à maintenant ; puisqu'on s'intéressera à l'étude et à l'amélioration de DRILS dans un environnement de calcul exclusivement séquentiel.

Troisième partie

Conception et étude d’algorithmes boîte grise séquentiels

Chapitre 5

Nouveaux mécanismes d'échappement pour DRILS

Dans la troisième partie de ce manuscrit, nous nous concentrons sur l'étude et l'amélioration de DRILS lorsqu'exécuté dans un environnement de calcul séquentiel. Dans ce chapitre, nous nous intéressons plus précisément à un aspect important de DRILS, celui des mécanismes lui permettant de s'échapper des optima locaux. Nous commençons par effectuer quelques observations empiriques, afin de mieux introduire notre motivation pour la conception de nouvelles variantes de DRILS intégrant de nouveaux mécanismes d'échappement dédiées. Avant de commencer, nous rappelons que DRILS utilise en plus de la recherche locale, le croisement par partitions, décrit plus en avant dans ce manuscrit. Bien que la conception de DRILS est transparente au type de croisement utilisé, ses performances peuvent être impactées de façon substantielle en fonction du croisement utilisé. Nous considérons l'utilisation du croisement boîte grise PX, qui a été historiquement le premier à être intégré dans DRILS. Plus tard dans ce manuscrit, nous dédierons un chapitre à l'étude approfondie de l'impact de l'utilisation d'autres opérateurs de croisement.
s

5.1 Motivation

5.1.1 Considérations préliminaires

Le critère d'acceptation est un élément clé de la dynamique de recherche et donc des performances des algorithmes inspirés de la recherche locale itérée. Tant que l'algorithme est capable de trouver des solutions de meilleures qualités, alors la question de l'acceptation ou non de celles-ci ne se pose à priori pas. Il n'y a en effet pas beaucoup d'intérêt à refuser une solution qui est meilleure que celle que nous avions précédemment, si l'on a la certitude que les améliorations pourront continuer tout au long de la recherche. Cepen-

dant, une telle certitude n'est souvent pas possible et la question d'accepter une nouvelle solution ou bien de la rejeter, en fonction de la rugosité du paysage, peut s'avérer cruciale. En pratique, la version standard de l'algorithme de recherche locale itérée, accepte la nouvelle solution à partir du moment où celle-ci ne diminue pas la qualité de la solution précédente, et rejette les solutions de moins bonnes qualités. Le fait de ne pas accepter de dégrader la solution peut cependant bloquer la recherche dans une région de l'espace dont il sera très difficile, voir impossible, de sortir.

5.1.2 Croisement par partitions, perturbation et critère d'acceptation dans DRILS

Dans le cas de DRILS, qui est un algorithme de recherche locale hybride, l'utilisation du croisement vient changer complètement la donne. En effet, le critère d'acceptation dans DRILS ne concerne pas la solution précédente comme dans le cas d'un ILS classique, mais plutôt à la fois : la solution qui représente le premier parent 1, la nouvelle solution générée qui représente le deuxième parent 2, et la nouvelle solution/enfant obtenu par croisement des deux parents. En particulier, rappelons que dans le cas du croisement par partition, l'enfant ne peut être dans le pire des cas qu'au moins aussi bon que le meilleur des deux parents, et dans le meilleur des cas meilleurs que les deux (en héritant des partitions améliorantes du génome de chacun). Ainsi, on ne peut jamais se retrouver dans le cas où l'on devrait choisir d'accepter ou de refuser une solution de qualité inférieure. Le choix adopté dans la conception originale de DRILS est alors d'accepter l'enfant à condition que ce dernier soit différent de ses deux parents. En d'autres termes, l'enfant doit hérité d'au moins une partition de chaque parent. Si cette condition n'est pas remplie, alors un mécanisme d'échappement est proposé. Il consiste à accepter le deuxième parent. Or, il est important de remarquer que ce deuxième parent a été généré par une perturbation d'intensité α suivie d'une recherche locale. Ainsi, dans le cas où le croisement ne serait pas fructueux, DRILS a exactement le même fonctionnement qu'une itération de recherche locale itérée où l'on accepte la nouvelle solution peu importe sa qualité. La perturbation entre donc en considération dans le mécanisme d'échappement au même titre que le critère d'acceptation.

Ces choix de conception ne sont pas anodins. En particulier, le fait de considérer le génome et non pas la qualité de la solution dans le critère d'acceptation peut être important. En effet, revenons brièvement sur le fonctionnement du croisement par partition (voir section 2.4.3). Dans le cas où les deux parents auraient la même qualité pour une composante connexe dans le graphe d'interaction, alors l'enfant héritera aléatoirement du génome d'un des deux parents pour celle-ci. Ainsi, imaginons qu'il y ait deux partitions de même qualité, alors on a 50% de chance de retomber sur l'un des deux parents et 50% de chance d'hériter des deux. Dans le dernier cas, on ne remplit pas la condition nécessaire

à l'activation du mécanisme d'échappement. Encore une fois, la probabilité que cela se produise dépend fortement du nombre de plateaux contenu dans le paysage de fitness et est proportionnelle à celui-ci.

En ce sens, il y a une part d'aléatoire dans le critère d'acceptation de DRILS. Ce comportement rappelle fortement le fonctionnement d'autres mécanismes probabilistes, tel que le recuit simulé, sauf qu'ici la probabilité d'accepter une solution de moins bonne qualité est donnée de façon implicite par l'opérateur de croisement. Ainsi, le mécanisme d'échappement est plus rarement déclenché en considérant le génome plutôt que la qualité des solutions. Comme nous le verrons plus tard, ce choix n'est pas toujours le meilleur. Il est bénéfique lorsqu'il faut davantage chercher autour de la solution courante, tandis qu'il peut s'avérer néfaste quand il est plus intéressant de favoriser l'exploration de l'espace de recherche.

5.1.3 Étude expérimentale préliminaire

Afin de mieux comprendre l'intérêt de la perturbation et du critère d'acceptation dans la conception de DRILS, mais aussi de voir l'impact du croisement par partitions, nous allons dans cette même section de motivation générale donner une étude expérimentale du comportement relatif des trois algorithmes de bases suivants. (i) La version standard de DRILS. (ii) Une version élitiste de DRILS où l'on accepte que les solutions améliorantes. Dans ce cas, on ne continue donc pas avec le second parent quand le croisement n'est pas prolifique, mais avec le meilleur des deux parents de sorte à toujours repartir de la meilleure solution trouvée lors de la recherche. (iii) Enfin, une recherche locale itérée sans croisement par partitions ; où l'on accepte toujours la nouvelle solution. Ainsi, la comparaison entre DRILS et ce troisième algorithme (une ILS simple non-élitiste) permettra de comprendre l'impact du croisement ; tandis que la comparaison entre DRILS et sa version élitiste permettra de mieux comprendre l'utilité ou non d'accepter ou de dégrader la solution courante.

Protocole Expérimental

En suivant les travaux de la littérature et les différents croisements boîtes grises [12, 13, 14], nous nous concentrerons sur des instances NKQ avec une valeur de Q de 64. Nous considérerons quatre valeurs différentes de degré de non-linéarité, $K \in \{2, 3, 4, 5\}$. Le nombre de variables N est composé des puissances de 10, dans l'intervalle entre mille et 1 million de variables. Le temps de calcul est fixé à 15 minutes. Pour chaque instance et chaque algorithme, nous effectuons 10 exécutions avec des graines différentes. Concernant l'hyper-paramètre α relatif à l'intensité de la perturbation, nous testons systématiquement

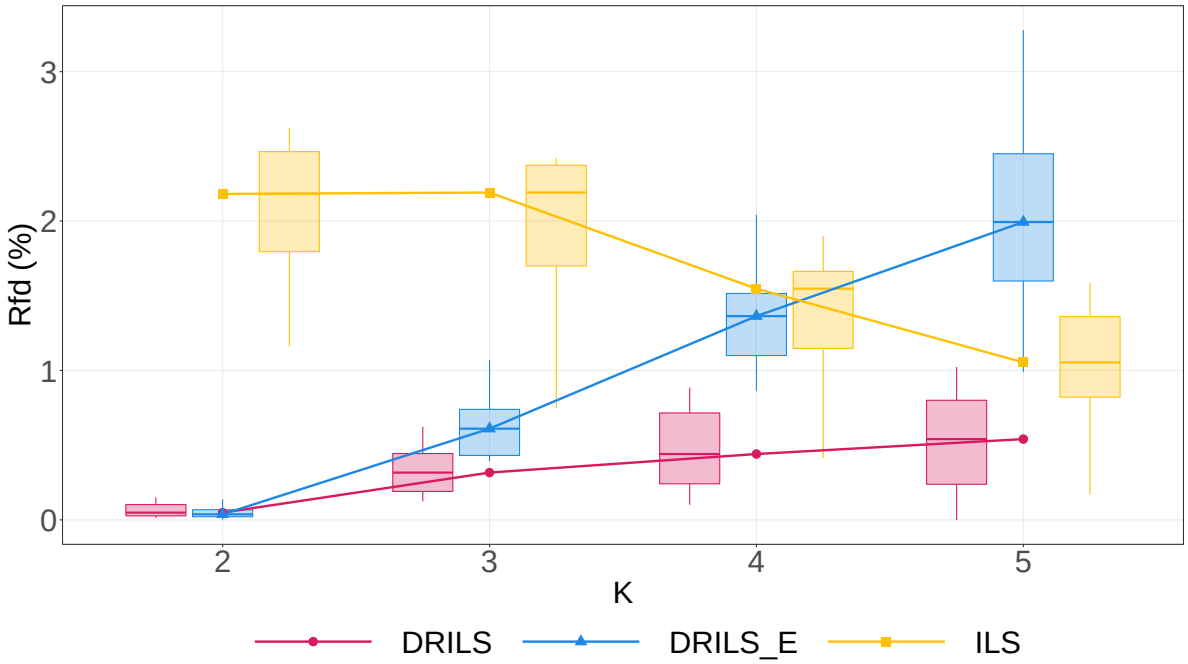

FIGURE 5.1 – *Rfd moyenne par rapport à la meilleure solution trouvée pour des instances NKQ de tailles allant de 1 000 à 1 million et pour des valeurs de K de 2 à 5. Les différents algorithmes considérés sont : DRILS, DRILS_E soit DRILS où l'on accepte que les solutions améliorantes et pour finir ILS, c'est-à-dire une recherche locale itérée où l'on accepte toujours la nouvelle solution. Les valeurs de N sont agrégées et on peut voir l'évolution de la performance des algorithmes en fonction de la rugosité de l'instance.*

les valeurs suivantes (en %) : {0.1, 0.25, 0.5, 1, 2.5, 5, 10, 15}. Les résultats sont donnés en utilisant la meilleure configuration pour chaque algorithme et combinaison de N et K.

Performances des différents algorithmes

Premièrement, intéressons-nous à la performance des différents algorithmes. La figure 5.1 nous montre la qualité moyenne des différentes solutions trouvées par les 3 algorithmes, et ceux pour différentes rugosités des instances. Cette qualité est mesurée à l'aide de la rfd par rapport à la meilleure solution jamais trouvée. Ceci nous permet notamment d'agréger les instances de tailles différentes, pour une meilleure lisibilité et afin de pouvoir se concentrer sur l'impact de la rugosité.

Dans un premier temps, regardons l'impact du critère d'acceptation de DRILS comparé à sa variante élitiste où l'on accepte que les solutions améliorantes. On peut voir que pour la plus petite valeur de K, soit K=2, ou dit autrement 3 variables par sous-fonction, les deux algorithmes trouvent des solutions qui sont de même qualité. Quand la rugosité augmente, on voit que l'écart se creuse en faveur de DRILS. La dispersion des résultats augmente elle aussi en fonction de la rugosité. Ainsi, on peut conclure que plus le problème est rugueux,

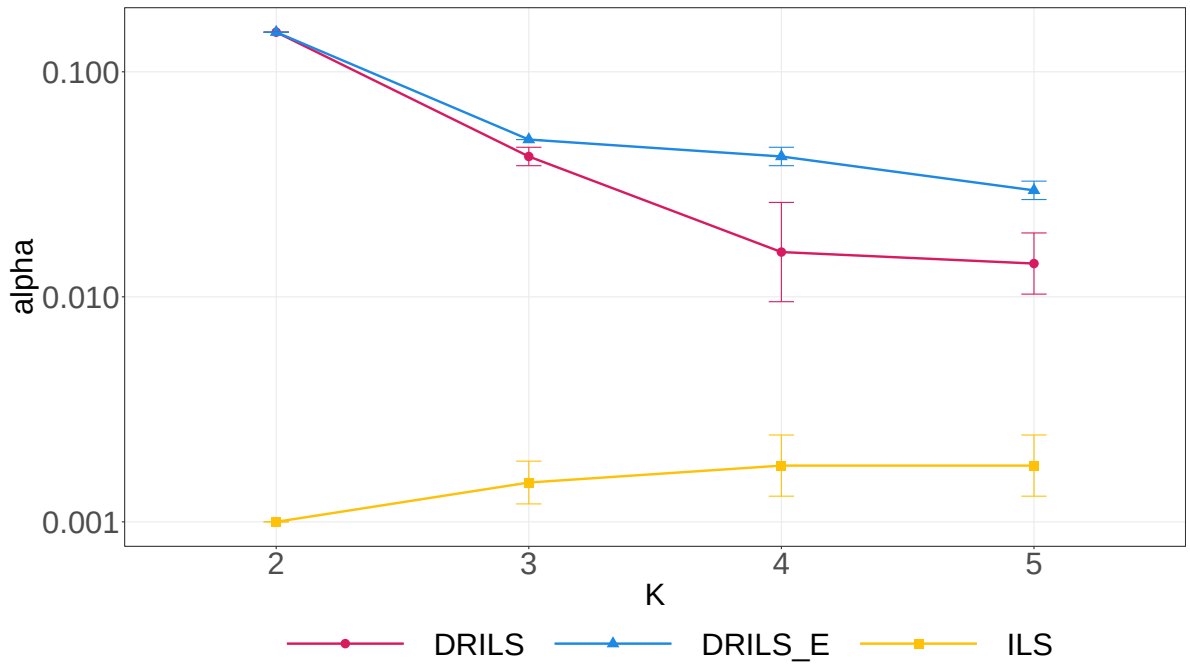

FIGURE 5.2 – *Meilleur paramètre pour les algorithmes DRILS, DRILS élite et ILS en fonction de la rugosité de l'instance, les valeurs de N sont agrégées.*

plus il est important d'accepter de dégrader la solution quand la recherche ne parvient plus à améliorer la solution courante.

Dans un second temps, on peut étudier l'importance du croisement dans DRILS. Pour ce faire, comparons DRILS avec ILS, la recherche locale itérée sans croisement où l'on accepte toutes les solutions. On voit que ILS est systématiquement moins bon que DRILS. Cependant, là où, lorsque la rugosité K augmente, DRILS avait davantage de difficulté à continuer à fournir des solutions très proches de la meilleure solution jamais trouvée, on observe le comportement inverse pour ILS. Autrement dit, plus l'instance est rugueuse plus ILS devient compétitif en moyenne en comparaison à DRILS. Il est aussi à noter que la dispersion de performance est beaucoup plus régulière suivant K, c'est-à-dire, qu'on observe une dispersion importante des résultats peu importe la rugosité des instances considérées. Une dernière remarque intéressante est le croisement des performances relatives entre DRILS élitiste et ILS lorsque la valeur de K atteint 4. Avant cette valeur, ILS est clairement moins performant, tandis que cela s'inverse pour des valeurs de K plus grands. On voit donc que le croisement par partitions fonctionne très bien pour les instances peu rugueuses ; mais à mesure que la rugosité augmente la performance décroît fortement en comparaison à une recherche locale simple sans croisement (surtout si un critère d'acceptation élitiste était utilisé avec le croisement).

Étude des paramètres optimaux

Concentrons-nous maintenant sur la valeur optimale de l'intensité de perturbation α pour les différents algorithmes en fonction de la rugosité de l'instance. Comme on peut le voir dans la figure 5.2, la perturbation optimale évolue en fonction de K, pour les algorithmes à base de croisement, c'est à dire DRILS et DRILS élitiste. On voit que la valeur optimale de α diminue à mesure que la rugosité augmente, les deux ayant une valeur optimale de 15% pour K=2, avant de décroître, à un rythme plus lent pour DRILS élitiste en comparaison à DRILS. Ceci peut s'expliquer par le besoin de DRILS élitiste de générer des solutions un peu plus éloignées, dans le but de s'échapper lorsque ce dernier ne parvient plus à progresser. DRILS (non-élitiste) va lui avoir la capacité à s'échapper plus naturellement du fait de son critère d'acceptation. Maintenant, en ce qui concerne ILS, on peut voir qu'à l'inverse des variantes de DRILS, la valeur optimale de la perturbation est positivement corrélée à la rugosité. Néanmoins, la croissance est très lente et les valeurs de α restent très petites, inférieures à 1% même pour les instances les plus rugueuses.

Discussion

De ces expériences préliminaires, on peut tirer des conclusions intéressantes afin d'améliorer notre compréhension de DRILS. On peut ainsi résumer l'apport de ces expériences en deux points :

1. Le choix du parent dans DRILS est important, et doit être associé avec une perturbation appropriée pour générer le deuxième parent utilisée pour le croisement ;
2. une recherche locale itérée où l'on accepte toutes les solutions peut, dans le cas où la perturbation est bien calibrée, aider à s'échapper efficacement des optima locaux, comme on le voit en comparant les performances de DRILS élitiste et ILS pour les instances les plus rugueuses.

En se basant sur ces observations, nous proposons dans la section suivante des variantes de DRILS dans le but d'améliorer ses performances par l'introduction de nouvelles stratégies d'échappement.

5.2 Intégration de nouveaux mécanismes d'échappement dans DRILS

En considérant que DRILS explore l'espace des optima locaux, nous aimerions améliorer sa capacité à découvrir des optima locaux prometteurs, tout en évitant de rester bloqué. Il est ainsi important de se rappeler que le croisement par partition est très efficace pour explorer une grande proportion de solutions enfants possibles. Cependant, il peut être trop

Algorithme 9 : DRILS+ : Une stratégie d'échappement rétroactive pour DRILS

```
   Input : α, β
1  current ← HBHC(random()) ;
2  repeat
3  |  next ← HBHC(perturb(current),α) ;
4  |  child ← PX(current, next) ;
5  |  if child = current or child = next then
6  |  |  current ← HBHC(perturb(current),β) ;
7  |  else
8  |  |  current ← HBHC(child);
9  |  end
10 until time left;
```

axé sur l'intensification lorsqu'il s'agit de s'échapper et d'explorer de nouvelles régions. En revanche, la recherche locale itérée avec un critère d'acceptation où l'on ne rejette aucune solution peut permettre d'accéder plus facilement à des optima locaux plus diversifiés, et donc éventuellement plus prometteur. En outre, la perturbation ne joue pas le même rôle selon que la recherche locale soit effectuée seule ou accompagnée d'un croisement. En particulier, le réglage optimal de l'intensité de la perturbation peut être différent en fonction des autres choix de conception comme l'a montré notre analyse préliminaire. Par conséquent, nous étudions l'idée d'utiliser non pas une, mais deux forces de perturbation personnalisées, tout en alternant soigneusement entre la recherche locale et le croisement pour tenter d'inférer un compromis intensification / diversification plus adéquat dans la conception DRILS.

Dans la suite, nous proposons donc deux variantes simples, dénommées DRILS+ et DRILS++, basées sur deux visions légèrement différentes. Dans la première, nous adoptons une stratégie d'échappement rétroactive, c'est-à-dire que le mécanisme d'échappement est appelé quand la recherche est bloquée. Dans la deuxième, nous adoptons une stratégie d'échappement proactive, en essayant d'agir avant que la recherche ne soit bloquée. Ces deux approches sont décrites plus en détail dans la suite.

5.2.1 DRILS+ : Une approche rétroactive

Comme le montre l'algorithme 9, DRILS+ applique une seconde perturbation de force β si la condition initiale de ligne 5 est remplie. En fait, supposons que le croisement n'ait pas réussi à aboutir à la création d'un enfant qui diffère de ses parents. Cela signifie que, bien qu'il ait la capacité d'explorer une énorme proportion d'enfants possibles, aucun d'entre eux n'est améliorant. En supposant que le deuxième parent a été généré à l'aide d'un

facteur de perturbation bien calibré (maximisant les chances du croisement de trouver une amélioration), nous interprétons cela comme un signal que la recherche commence à stagner. Ainsi, au lieu de reprendre la recherche à partir du dernier optimum local généré (comme le fait DRILS), nous proposons dans DRILS+ de reprendre la recherche à partir d'un nouvel optimum local fraîchement obtenu en appliquant une perturbation *différente*, d'intensité β, appliquée sur le premier parent précédent (ligne 6). La motivation est de se déplacer vers un optimum local qui n'aurait pas été trouvé en utilisant la force de perturbation α (lignes 8 et 9). Rappelons que pour que le croisement par partition atteigne son meilleur potentiel, la valeur de α est d'une importance cruciale, puisque c'est cette perturbation qui permettra d'obtenir ou non un deuxième parent capable d'améliorer la solution courante par le biais du croisement. Par conséquent, la conception d'une deuxième perturbation dans la nouvelle variante DRILS+ nous permet de séparer clairement le rôle de la perturbation : (i) la perturbation β doit être considérée comme un moyen de contrôler l'intensité que l'on veut mettre lors de la phase de dégradation de la solution afin de s'échapper chaque fois que la recherche est bloquée, et (ii) la perturbation α est liée à la génération du deuxième parent utilisé pour le croisement. Ces deux aspects étant en effet fondamentalement différents, alors même que dans la variante initiale de DRILS, aucune distinction explicite n'y est faite. La version DRILS+ proposée étend donc la version DRILS initiale dans ce sens.

Notons que DRILS+ peut être considéré comme un algorithme dans lequel la perturbation β pour promouvoir la diversité est effectuée de manière rétroactive. En effet, la perturbation β n'est appliquée qu'après avoir détecté que l'algorithme stagne et ne parvient plus à effectuer des croisements fructueux. Une autre option consiste à ne pas attendre qu'un tel événement se produise et de procéder de manière proactive afin de minimiser la probabilité d'être piégé. C'est précisément l'idée de notre deuxième variante, appelée DRILS++.

5.2.2 DRILS++ : Une approche proactive

Comme le montre l'algorithme 10, dans DRILS++, nous alternons entre : (i) une itération de recherche locale itérée pure, sans croisement, en utilisant une perturbation β et un critère d'acceptation où la nouvelle solution est toujours acceptée, et (ii) le croisement par partitions utilisant une perturbation α pour générer le deuxième parent. Plus précisément, avec une probabilité γ, le premier parent est dégradé à l'aide de la perturbation β, et il est systématiquement remplacé par l'optimum local obtenu par la recherche locale qui en suit. Sinon, avec une probabilité de $1 - \gamma$, nous procédons exactement comme dans DRILS. D'après notre étude préliminaire, ILS s'est en effet avéré moins bon que DRILS, mais il a montré un certain potentiel pour s'échapper des optima locaux avec une force de perturbation plus faible. Par conséquent, la conception de DRILS++ vise à promouvoir la

Algorithme 10 : DRILS++ : Une stratégie d'échappement pro-active pour DRILS

```
Input : α, β, γ
current ← HBHC(random()) ;
repeat
    if random(0,1) < γ then
        current ← HBHC(perturb(current),β) ;
    else
        next ← HBHC(perturb(current),α) ;
        child ← PX(current, next) ;
        if child = current or child = next then
            current ← next ;
        else
            current ← HBHC(child);
        end
    end
until time left;
```

diversité en considérant une perturbation β dédiée comme dans DRILS+. Cependant, il alimente, avec une probabilité γ, un nouveau premier parent pour effectuer un croisement avec l'espérance d'augmenter la probabilité de succès de ce dernier. Notons aussi que cette deuxième variante DRILS++ implique l'utilisation d'un nouveau paramètre γ.

5.3 Analyse Éxpérimentale

5.3.1 Protocole

Pour tester l'efficacité de nos deux nouvelles variantes de DRILS, à savoir DRILS+ et DRILS++, nous utilisons les mêmes instances que lors de l'étude préliminaire réalisée au début de ce chapitre. Concernant le temps d'exécution, nous utilisons également 15 minutes et nous testons toutes les combinaisons de paramètres possibles en considérant les ensembles suivants : {0.1, 0.25, 0.5, 1, 2.5, 5, 10, 15} (en %) pour α et β et $\{25, 50, 75\}$ (en %) pour γ. Notons que pour DRILS++ utiliser une valeur de γ de 0 ou 1 revient à retomber respectivement sur DRILS ou bien ILS.

Pour les besoins de notre analyse, nous calculons les valeurs optimales de α, β et γ, fournissant donc la plus petite rfd telle que définie dans la section 4.2.6. Ceci est effectué pour chaque instance afin d'avoir une analyse fine et juste des performances de chaque algorithme lorsqu'il est configuré de manière optimale. En outre, nous calculons la perturbation qui fournit les meilleures valeurs rfd moyennes par rapport à l'ensemble des

instances, ainsi que, lorsqu'elles sont divisées en différents groupes en fonction de K ou N. Dans ce cas, la meilleure combinaison, globale ou par groupe d'instances, des paramètres peut être différente de la meilleure configuration par instance, et il en va de même pour les performances sous-jacentes.

Notre objectif est aussi de fournir une analyse complète de l'impact de la perturbation et de la robustesse des algorithmes. Pour chaque paramètre, nous utilisons un test de Wilcoxon avec un seuil de significativité de 0.05 pour évaluer les différences statistiques. Nous commençons par analyser les performances relatives des algorithmes lorsqu'ils sont configurés de manière optimale. Cette analyse est suivie d'une étude détaillée de l'impact de la configuration de la force des différentes perturbations utilisées.

5.3.2 Analyse de performance

Performance globale

Dans le tableau 5.1, nous indiquons les rangs des différents algorithmes obtenus en utilisant les meilleures valeurs de perturbation. Le rang d'un algorithme est calculé comme le nombre d'autres algorithmes qui sont significativement meilleurs que l'algorithme considéré. Le meilleur rang possible est donc 0, qui indique que l'algorithme n'est jamais battu. Nous indiquons également les variantes qui sont significativement meilleures (▲), pires (▽) ou non signifactivement différentes (≈) de DRILS.

Sur l'ensemble des instances (coin inférieur droit du tableau 5.1), DRILS+ et DRILS++ sont tous les deux classés premiers (ce qui correspond au rang 0), avec DRILS+ étant significativement meilleur que DRILS. Cela fournit une première preuve empirique (sur l'ensemble des paysages) de l'importance d'une conception précise de la perturbation lors de l'hybridation des différentes techniques boîte grise. Pour la suite, nous analyserons plus attentivement les performances en fonction des caractéristiques du paysage (dimension et rugosité).

Impact de la rugosité des instances sur les performances

En observant les différents rangs des instances regroupées par K (c'est-à-dire la dernière ligne), on constate que DRILS+ s'avère significativement meilleur que DRILS pour K = 2, alors qu'il a une performance comparable pour les grandes valeurs de K = 3, 4, 5. En revanche, DRILS++ est significativement meilleur que DRILS pour les grandes valeurs de K = 3, 4, 5 (remarquez que nous avons arrondi la valeur rfd à deux décimales pour des raisons de présentation). Cela signifie que pour les paysages faciles, les opérateurs de croisement boîtes grises sont extrêmement efficaces ; cependant, ils commencent à se heurter à des difficultés lorsque la rugosité du problème augmente. La perturbation entre en jeu et permet d'observer des différences significatives avec DRILS.

N	*Algorithmes*	$K=2$			$K=3$			$K=4$			$K=5$			$\forall K$		
1'000	DRILS	2	(0.11)		1	(0.56)		2	(0.58)		2	(0.33)		1	(0.70)	
	DRILS+	**0**	(0.00)	▲	**0**	(0.02)	▲	**0**	(0.10)	▲	**0**	(0.06)	▲	0	(0.09)	▲
	DRILS++	2	(0.12)	≈	1	(0.52)	≈	1	(0.50)	≈	1	(0.27)	▲	1	(0.70)	≈
10'000	DRILS	1	(0.10)		1	(0.16)		2	(0.13)		**0**	(0.15)		1	(0.40)	
	DRILS+	**0**	(0.02)	▲	**0**	(0.06)	▲	**0**	(0.06)	▲	**0**	(0.14)	≈	0	(0.21)	▲
	DRILS++	1	(0.09)	≈	1	(0.15)	≈	**0**	(0.09)	▲	**0**	(0.11)	≈	1	(0.40)	≈
100'000	DRILS	2	(0.04)		2	(0.21)		2	(0.33)		2	(0.71)		1	(0.28)	
	DRILS+	1	(0.03)	▲	**0**	(0.05)	▲	**0**	(0.04)	▲	**0**	(0.06)	▲	0	(0.07)	▲
	DRILS++	**0**	(0.01)	▲	1	(0.12)	▲	1	(0.09)	▲	1	(0.17)	▲	1	(0.27)	≈
1'000'000	DRILS	1	(0.02)		1	(0.41)		1	(0.85)		1	(0.98)		1	(0.53)	
	DRILS+	1	(0.02)	≈	1	(0.41)	≈	1	(0.84)	≈	1	(0.98)	≈	1	(0.53)	≈
	DRILS++	**0**	(0.00)	▲	**0**	(0.04)	▲	**0**	(0.05)	▲	**0**	(0.04)	▲	0	(0.25)	▲
$\forall N$	DRILS	2	(0.07)		1	(0.34)		1	(0.53)		1	(0.65)		1	(0.48)	
	DRILS+	0	(0.02)	▲	0	(0.26)	≈	0	(0.44)	≈	1	(0.66)	≈	0	(0.39)	▲
	DRILS++	1	(0.06)	▲	0	(0.25)	▲	0	(0.29)	▲	0	(0.32)	▲	0	(0.43)	≈

TABLE 5.1 – *Comparaison statistiques des performances obtenues avec les meilleurs paramètres pour des temps de calcul de 15 minutes entre les différentes variantes de DRILS proposées. Le Test utilisé est celui de Wilcoxon avec un seuil de significativité de 0.05. Sur la première sous colonne le rang : un rang de 0 indiquant que l'algorithme n'est jamais significativement moins bon qu'au moins l'un des autres ; ensuite la rfd moyenne puis en dernier la significativité du test entre DRILS+ (ou DRILS++) et DRILS,* ▲ *indiquant que l'algorithme est mieux que DRILS.*

Puisque DRILS+ attend que la recherche ne s'améliore plus avant d'effectuer la perturbation β, il donne plus de temps au croisement pour intensifier la recherche autour des parents actuels, tout en offrant une chance raisonnable de s'échapper. Toutefois, pour les paysages très vastes et très rugueux, cela ne semble pas suffisant pour contrebalancer le biais d'intensification très fort du croisement par partitions, et la recherche est très probablement piégée plus rapidement. En revanche, la conception proactive de DRILS++ permet d'anticiper une telle situation en favorisant la diversité sans attendre que le croisement explore intensivement les derniers parents découverts. Il est intéressant de noter que le seul groupe d'instances pour lequel DRILS++ est significativement plus performant que DRILS+ est pour K = 5, qui est l'instance la plus difficile ; et les seules instances pour lesquelles DRILS+ est significativement plus performant que DRILS++ sont les instances

où K = 2. Cela met en lumière les différents comportements liés au compromis intensification/diversification apporté par ces deux stratégies : proactive et rétroactive, ainsi que leurs points forts et faibles en fonction de la rugosité de l'instance.

Il est important de noter que ces observations ne sont valables que lorsque les algorithmes sont configurés avec la meilleure intensité de perturbation pour un ensemble d'instances regroupées par rugosité. C'est-à-dire, une configuration globale ne tenant pas compte de l'hétérogénéité des paysages. La situation peut être différente lorsque l'on utilise des configurations plus fines et calibrées par instance en tenant compte de la taille des instances, N. Ceci est analysé ci-après.

Impact des instances sur les performances

Concernant les performances relatives suivant la dimension du paysage N (dernière colonne du tableau), DRILS+ s'avère globalement plus efficace que DRILS++ pour presque toutes les instances de dimension $N \leq 10^5$, sauf pour $N = 10^5$ et K = 2, et $N = 10^4$ et $K = 4, 5$. Pour les plus grandes instances d'un million de variables avec $K \geq 2$, DRILS++ surpasse DRILS+ de manière significative.

Pour compléter, en considérant les performances des algorithmes avec des paramètres calibrés pour chaque couple (N,K), on peut voir que DRILS+ est statistiquement meilleur que DRILS sur 11 des 16 instances considérées. Ce chiffre est de 10 pour DRILS++. Notons que ni DRILS+, ni DRILS++ ne sont jamais statistiquement moins bons que DRILS. En ce qui concerne le rang des algorithmes, on peut voir que DRILS + obtient 11 fois le meilleur rang possible contre 7 pour DRILS++. Rappelons que le meilleur rang est octroyé à un algorithme pour une instance s'il n'est jamais battu par les autres algorithmes.

5.3.3 Analyse de la dynamique recherche

Dynamique de recherche en fonction du temps

Intéressons-nous maintenant à l'évolution de la meilleure solution trouvée par les différents algorithmes pour chaque instance en fonction du temps d'exécution. Ceci est montré dans la figure 5.3. Cette figure nous permet de mieux comprendre la dynamique de recherche, et de constater visuellement l'écart entre les qualités des solutions obtenues.

Commençons par les plus petites instances. On voit que pour K=3, DRILS et DRILS++ ont le même profil tout au long de la recherche tandis que DRILS+ est d'entrée meilleur que ces deux versions, et continue à maintenir un écart important jusqu'à la fin de la recherche. Pour les problèmes plus rugueux, on constate que les profils de DRILS et DRILS++ diffèrent, avec DRILS++ étant toujours légèrement meilleur que DRILS. Même si à la fin d'exécution, les deux courbes tendent à se rejoindre, l'écart est suffisant pour que le test statistique conclue à la supériorité de DRILS++. Une nouvelle fois DRILS+ est de

loin le meilleur algorithme. Même si on peut remarquer que durant les premières dizaines de secondes les courbes de DRILS+ et DRILS++ se chevauchent, l'écart se creuse ensuite et s'intensifie jusqu'à la fin de la recherche. Pour cette taille d'instance, il semble que les solutions trouvées par DRILS+ soient clairement hors de portée de DRILS et DRILS++ en un temps raisonnable si l'on se fit à la convergence des courbes.

Pour N=10 000, on peut voir que les tendances sont moins marquées. Si l'on s'intéresse à K=3, on voit qu'une nouvelle fois DRILS et DRILS++ ont le même profil, tandis que DRILS+ est meilleur tout du long de la recherche avec un écart qui se creuse au fil du temps, mais qui demeure beaucoup plus faible que pour l'instance de plus petite taille et de même rugosité. Pour K=5, on peut voir que toutes les courbes se chevauchent. DRILS++ tend à être très légèrement meilleur, mais pas suffisamment pour que les tests statistiques soient significatifs.

Pour les instances de 100 000 variables, DRILS est toujours moins bon que les autres algorithmes si l'on compare à temps égal, hormis pour K=5 et 10 secondes où DRILS et DRILS++ ont la même qualité. En ce qui concerne DRILS+ et DRILS++, on peut voir que les deux courbes se croisent, DRILS++ étant meilleur au début de la recherche, mais se faisant dépasser par DRILS+ après un temps plus ou moins long suivant la rugosité du paysage : plus celle-ci est importante plus tard sera le croisement des profils. Ici, on voit clairement à la fin l'écart de performance entre les différents algorithmes ; avec DRILS+ meilleur que DRILS++ lui-même meilleur que DRILS.

Enfin, terminons cette analyse avec les instances à 1 million de variables. On peut voir que DRILS++ est largement meilleur après une centaine de secondes en comparaison à DRILS et DRILS+ et continue de creuser l'écart. On remarque également que DRILS et DRILS+ ont la même courbe de convergence. Cela peut s'expliquer par le fait que DRILS+ n'est jamais bloqué en 900 secondes. De ce fait, son comportement est exactement le même que celui de DRILS.

Probabilité de succès des croisements

Pour compléter notre étude, nous analysons comment la capacité du croisement à trouver des enfants améliorants est affectée par la perturbation. À cette fin, nous reportons une expérience supplémentaire en utilisant deux ensembles d'instances avec $N \in \{10^4, 10^5\}$ et $K \in \{3, 4, 5\}$. Pour chaque exécution d'algorithme, nous enregistrons le nombre de fois où le croisement a réussi à trouver un enfant améliorant. Nous enregistrons également le nombre de fois où un enfant améliorant a été lui-même amélioré par le Hill Climber effectué juste après le croisement. Nous indiquons ensuite la probabilité de succès empirique du croisement par partitions, et la probabilité de succès du Hill Climber (ou, de manière équivalente, la probabilité que l'enfant ne soit pas un optimum local).

Ceci est illustré dans la figure 5.4 en utilisant la configuration optimale (par instance)

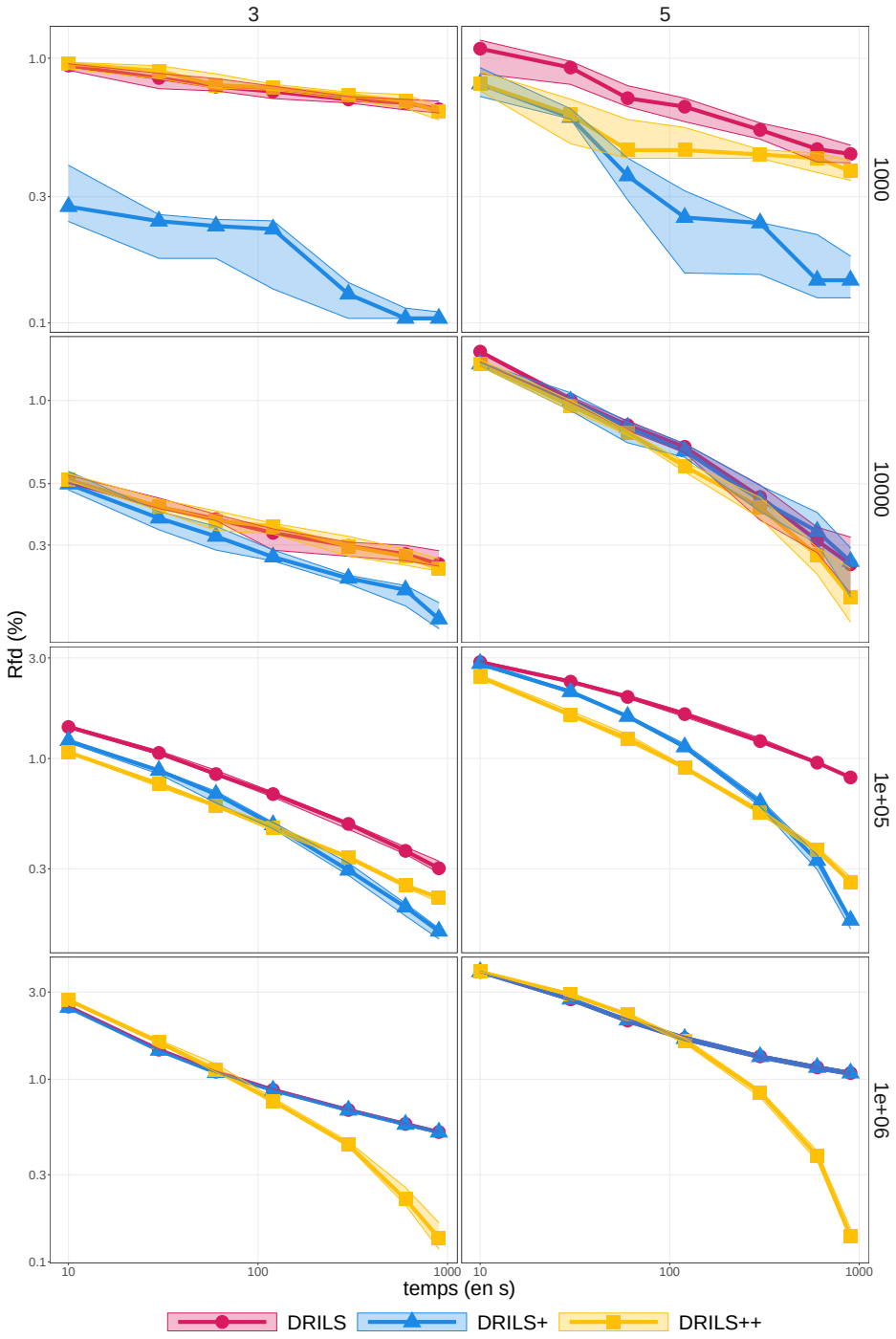

FIGURE 5.3 – *Exemple de la dynamique de recherche des différents algorithmes en fonction du temps pour $K = 3$ et 5 et des valeurs de N croissantes.*

de chaque algorithme, et en agrégeant les valeurs de probabilité sur les différentes exécutions et les différentes valeurs de K. Nous pouvons voir que la probabilité de succès du croisement par partitions est la plus élevée pour DRILS++. Elle est presque de 1 pour $N = 10^5$, alors qu'elle est d'environ 0.8 pour DRILS et DRILS+. Cela ne peut être attribué

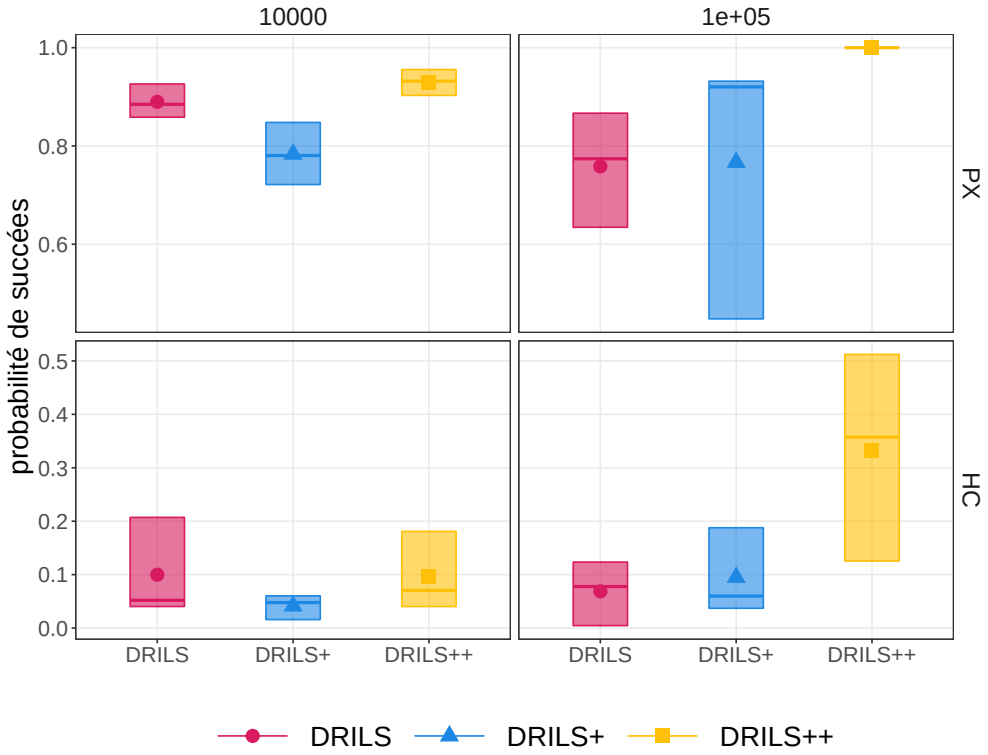

FIGURE 5.4 – *Probabilité de succès du croisement par partitions (PX) ou du Hill Climber suivant (HC), en fonction de l'algorithme et de la taille du problème, les valeurs de K sont agrégées ($K = \{3, 4, 5\}$).*

qu'à la conception de la perturbation β. Puisque pour $\gamma > 0$, DRILS++ effectue globalement moins de croisements que DRILS et DRILS+. Il est intéressant de noter que cela ne diminue pas les chances de trouver des enfants améliorants. Au contraire, la perturbation β permet à DRILS++ de se déplacer vers de nouveaux optima locaux pour lesquels le croisement a plus de chances de réussir. En outre, la probabilité que les enfants améliorants trouvés par DRILS++ ne soient pas des optima locaux augmente. Cela peut s'expliquer par la capacité de DRILS++ à visiter des régions de recherche diversifiées où il est plus probable de trouver des améliorations. Notez que cela ne signifie pas nécessairement que DRILS++ améliore systématiquement la meilleure solution, étant donné que la probabilité de succès est calculée par rapport au premier parent actuel. Cependant, cela montre que la diversité introduite par la perturbation β peut être bénéfique pour la probabilité de succès du croisement, et donc pour le processus de recherche dans son ensemble.

5.3.4 Analyse de l'intensité des perturbations

Dans cette section, nous analysons l'effet des différentes intensités de perturbation, ainsi que les valeurs optimales suivant l'algorithme, la taille et la rugosité de l'instance. Dans le tableau 5.2, nous indiquons les valeurs optimales de α, β et γ pour chaque algorithme ; en

N	Algorithmes	$K=2$			$K=3$			$K=4$			$K=5$		
		α	β	γ	α	β	γ	α	β	γ	α	β	γ
1'000	DRILS	15.0			5.0			0.1			0.2		
	DRILS+	10.0	0.1		15.0	1.0		2.5	1.0		2.5	1.0	
	DRILS++	15.0	2.5	25.0	5.0	0.5	25.0	0.1	0.1	75.0	0.2	0.1	75.0
10'000	DRILS	15.0			5.0			2.5			2.5		
	DRILS+	15.0	2.5		5.0	2.5		5.0	2.5		2.5	2.5	
	DRILS++	15.0	0.1	50.0	5.0	0.2	25.0	2.5	0.2	25.0	2.5	0.1	50.0
100'000	DRILS	15.0			2.5			5.0			2.5		
	DRILS+	5.0	15.0		5.0	10.0		2.5	5.0		2.5	5.0	
	DRILS++	15.0	0.5	25.0	5.0	0.5	75.0	2.5	1.0	50.0	2.5	1.0	50.0
1'000'000	DRILS	15.0			5.0			5.0			2.5		
	DRILS+	15.0	15.0		5.0	10.0		5.0	15.0		2.5	1.0	
	DRILS++	15.0	1.0	25.0	5.0	2.5	25.0	2.5	1.0	50.0	2.5	1.0	50.0

TABLE 5.2 – *Meilleure configurations des hyper-paramètres trouvée par algorithme pour chaque combinaison de taille d'instance et de rugosité sur des temps de calcul de 15 minutes.*

d'autres termes celles ayant la plus faible rfd. Nous indiquons les valeurs calculées pour les différents ensembles d'instances regroupées par N ou K.

Impact du paramètre α

Tout d'abord, nous pouvons constater que les paramètres optimaux de α suivent la même tendance pour les différents algorithmes et instances. Pour la plus petite valeur de K = 2, la meilleure intensité de perturbation α se situe dans la fourchette la plus élevée des valeurs expérimentées ($\alpha = 15\%, 10\%$). Pour les plus grandes valeurs de K = 3, 4, 5, la meilleure valeur de α diminue progressivement jusqu'à 2.5 %. Ce résultat est en fait cohérent pour tous les algorithmes et conforme à notre analyse préliminaire. Nous pouvons donc conclure que la perturbation α doit définitivement être fixée de manière à ce que le second parent puisse se croiser efficacement avec le premier. En d'autres termes, le choix de la meilleure valeur de α devrait être guidé par la capacité d'alimenter le croisement avec des couples de parents prometteurs, et ce, en fonction de K. Rappelons-nous que dans la conception originale de DRILS, la perturbation α jouait deux rôles à la fois : (i) générer un second parent pour le croisement et (ii) mettre à jour le premier parent au cas où le

croisement ne serait pas productif. En revanche, DRILS+ et DRILS++ font une différence claire entre ces deux rôles en introduisant la deuxième perturbation β.

Impact du paramètre β

Il est intéressant d'observer les valeurs de β pour les différents algorithmes. Commençons par le cas le plus simple et le plus généralisable, celui de DRILS++. En effet, on voit que la valeur optimale de β est globalement stable pour toutes les instances considérées. On peut tout de même considérer deux cas, les instances de plus petites tailles et celles de plus grandes tailles. Pour les premières, à savoir 1 000 et 10 000, la perturbation est très faible à une exception près pour K=2 et N=10 000. Sinon, la valeur optimale se situe entre 0.1 et 0.5 % seulement ; ce qui indique un besoin très faible de diversification pour les plus petites instances. Tandis que pour les instances les plus grandes, la perturbation est légèrement plus forte avec une moyenne de 1%. Cette perturbation est à considérer conjointement à la stratégie proactive, qui consiste à accepter une dégradation légère de solution avec une probabilité comprise entre 25% et 75%. On peut facilement comprendre que dans le cas d'une dégradation trop forte, la diversification apportée vient contrebalancer les efforts du croisement qui tentent étape après étape d'améliorer la qualité de la solution.

Passons maintenant à DRILS+. Pour ce dernier, on voit clairement une évolution en fonction de la taille des instances. Tout comme DRILS++, l'intensité optimale de la perturbation β augmente proportionnellement à la dimension du problème. Pour les instances à 1000 variables, on voit que β est plus petit, voir beaucoup plus petit, que α, de l'ordre de 1% la majorité du temps. Ceci indique une forte nécessité d'intensification autour de la zone de recherche active. Pour N = 10 000, on voit l'augmentation de la valeur de β qui atteint 2.5% pour toutes les valeurs de K considérées. Ici aussi, la valeur de α reste plus grande, mais avec un écart qui diminue. On a une valeur de α 2 fois plus grandes que β sur la moitié des instances ; tandis que pour K=5, on voit que $\alpha = \beta$ et que ces valeurs sont les mêmes que le α optimal de DRILS. Ceci explique pourquoi pour N=10 000, K=5 est la seule instance où les algorithmes DRILS et DRILS++ ont des performances similaires.

Si l'on passe maintenant aux instances à 100 000 variables, on voit apparaître le besoin de diversification à travers le ratio entre α et β. En effet, jusqu'à maintenant, les instances nécessitaient davantage d'intensification avec des valeurs β en général plus petite que α. Cependant, désormais, le ratio est de 2 à 3 en faveur de β. Ceci indique que dans le cas où la recherche se bloquerait, l'intensité de la perturbation pour s'échapper de cette région est deux fois plus importante que la perturbation utilisée pour générer le deuxième parent. En ce qui concerne les instances à 1 million de variables, les résultats sont moins probants, dans le sens où, étant donné la taille des instances et donc le temps nécessaire à

exécuter DRILS+, l'algorithme se retrouve très rarement en situation de faire appel à son mécanisme d'échappement.

Impact du paramètre γ

Terminons notre analyse avec le dernier paramètres γ, propre à DRILS++. Rappelons que ce paramètre désigne la probabilité d'effectuer ou non une itération de recherche locale itérée à la place d'une itération de DRILS. D'après le tableau 5.2, les valeurs les plus performantes de γ sont principalement 25% et 50%. Cela signifie que l'exécution de la perturbation β la moitié ou le quart du temps est un choix assez judicieux. Étant donné que la meilleure perturbation β s'est avérée être relativement petite, un tel réglage permet en fait de visiter de nouveaux optima locaux, assez proches, mais conduisant le croisement à trouver des améliorations de bonne qualité.

5.4 Conclusion

Dans ce chapitre, nous avons revisité la conception du mécanisme de perturbation de l'algorithme état de l'art DRILS. Ceci a conduit à la conception de nouveaux schémas d'hybridation améliorés exposant de nouveaux compromis intensification/diversification basée sur des stratégies d'échappement proactive et rétroactive. En plus d'être en mesure d'obtenir des améliorations significatives, notre travail fournit des preuves de l'importance de la conception de l'hybridation, et démontre la nécessité d'utiliser des perturbations d'intensités différentes pour chaque étape de l'algorithme, à savoir la génération du deuxième parent utilisé pour le croisement (première perturbation α) et le mécanisme d'échappement et de diversification (perturbation β, et également paramètre γ pour l'approche proactive).

En outre, nos résultats montrent que la capacité impressionnante du croisement par partition à explorer un grand nombre de solutions, doit être équilibrée par des mécanismes d'échappement soigneusement conçus en tenant compte de la dimension et de la rugosité du paysage considéré. Nos travaux démontrent que la perturbation peut en fait jouer différents rôles, influençant la capacité du croisement à continuer à améliorer la solution courante, et permettant de visiter des régions prometteuses inexplorées.

Chapitre 6

Initialisation explorative pour DRILS

6.1 Introduction

Dans ce chapitre, nous poursuivons nos investigations pour l'amélioration de l'algorithme DRILS et de ces variantes proposées dans le chapitre précédent. Pour ce faire, nous proposons une nouvelle variante, appelée PRILS pour 'Pipelined Recombination and Iterated Local Search' (Recombinaison et recherche locale itérée en pipeline). Cette nouvelle variante s'appuie sur une conception extrêmement simple basée sur des observations empiriques solides.

Dans un premier temps, nous présentons en effet une analyse empirique qui amène très naturellement à la conception de cette nouvelle variante. Le but est de permettre au lecteur de comprendre d'un point de vue plus fondamental la dynamique de recherche et la motivation derrière cette nouvelle conception. Nous introduisons ensuite l'algorithme PRILS et analysons ses performances. Concernant le croisement utilisé, nous nous concentrerons dans ce chapitre sur le croisement par partitions. Les différentes combinaisons possibles à partir des variantes proposées et des opérateurs de croisement boîte grise existants, seront étudiées dans le prochain chapitre.

6.2 Analyse des optima locaux obtenus par ILS et DRILS+

6.2.1 Analyse basée sur le génotype des optima locaux

Protocole expérimental

Considérons des paysages NKQ aléatoires avec la configuration suivante $N = 100\,000$, $K = 5$, $Q = 64$. Nous proposons d'exécuter 5 000 itérations d'une recherche locale itérée (ILS) avec un critère d'acceptation non-élitiste et en utilisant un facteur de perturbation $\alpha = 0.01$. Nous proposons d'exécuter également l'algorithme DRILS+ dé-

crit dans le chapitre précédent pour le même nombre d'itérations, avec les paramètres $\alpha = 0.025$ et $\beta = 0.05$. Les deux algorithmes sont exécutés à partir du même premier optimum local générée initialement à partir d'une solution aléatoire à laquelle une recherche locale Hill Climber est appliquée. Nous considérons 10 exécutions indépendantes pour chaque algorithme, ainsi que 10 instances aléatoires différentes ; ce qui aboutit au final à 100 exécutions par algorithme.

Pour chaque exécution, nous enregistrons les solutions courantes (qui sont des optima locaux) utilisées par chaque algorithme (ILS ou DRILS+) au début de chaque itération au cours de la recherche. Pour chaque variable binaire, nous proposons ensuite de calculer le nombre de fois où sa valeur assignée (0 ou 1) dans une telle solution a changé au cours des différentes itérations d'une exécution donnée. Par conséquent, en agrégeant les différentes exécutions et instances, nous sommes en mesure de calculer le pourcentage de variables dont l'assignation dans la solution courante change pour un nombre donné de fois au cours de l'exécution d'ILS ou de DRILS+. Autrement dit, nous sommes en mesure d'étudier la probabilité empirique que l'affectation d'une variable change pour un nombre donné de fois au cours d'une exécution. Ceci est montré dans l'histogramme de la figure 6.1.

Interprétation

Nous pouvons constater que ILS et DRILS+ impliquent des distributions très différentes. Il est plutôt surprenant de noter que DRILS+ a tendance à modifier moins fréquemment l'affectation des variables des solutions visitées par rapport à ILS. Par exemple, 9,6% des variables ne sont jamais modifiées dans DRILS+, alors que ce chiffre tombe à 2,3% pour ILS. Seulement 3% des variables changent plus de 10 fois au cours des 5 000 itérations pour DRILS+, ce qui contraste avec les 22,5% dans le cas de ILS. Cela indique clairement que, malgré sa supériorité, DRILS+ agit sur une région plus restreinte de l'espace de recherche par rapport à ILS. Ceci est très probablement dû au pouvoir d'intensification du croisement boîte grise PX sous-jacent. Par conséquent, bien que DRILS+ soit plus performant que ILS, il est probable qu'il intensifie la recherche sur une région spécifique, tout en manquant de diversification. En revanche, ILS est capable de visiter des régions plus diversifiées comme le montre la propriété des génotypes des solutions rencontrées durant le processus de recherche, tout en manquant cependant d'intensification (puisque ILS sans croisement a été montré moins bon que DRILS et ses variantes avec croisement).

Il est intéressant de noter que la force de DRILS+ comparé à DRILS est justement cette capacité de diversification plus importante quand la recherche est piégée. Cependant, on se rend compte dans cette expérience que malgré cette diversification plus importante, DRILS+ reste un algorithme très porté sur l'intensification. Le fait que beaucoup de variables reste figées dans leur état initial nous indique que dés le début de la recherche une grande partie de la solution va être fixée par DRILS+, et va rester dans cet état même

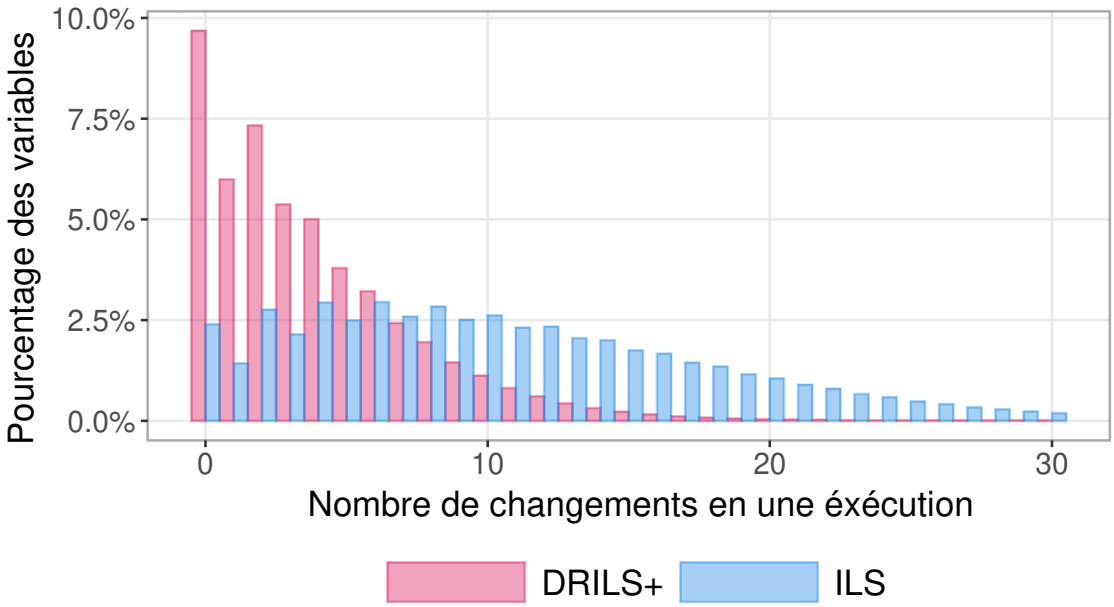

FIGURE 6.1 – *Histogramme représentant le pourcentage de variables qui changent exactement X fois au cours d'une exécution de 5000 itérations de ILS (en bleu) ou DRILS+ (en rouge), la valeur X étant représentée sur l'axe des abscisses.*

après plusieurs milliers d'étapes, et donc probablement également après de nombreuses très fortes perturbations. Rappelons que la perturbation β est fixé à 5% ce qui indique tout de même un fort changement potentiel dans la solution, si l'on compare au 1% seulement utilisé par l'ILS en guise d'intensité de perturbation.

De cette expérience, on peut tirer la conclusion importante suivante. Lorsque l'on croise le comportement de DRILS+ avec sa très bonne performance, cet algorithme apparaît très adéquat pour intensifier la recherche en utilisant le croisement par partitions, tout en apportant légèrement plus de diversité que DRILS. Cependant, il ne permet d'explorer qu'une région relativement restreinte de l'espace de recherche, dû à un 'biais' fort impliquant que certaines variables restent figées dans leur état initial tout au long de la recherche. Il semble donc intéressant d'étudier la possibilité d'explorer dans un premier temps l'espace de recherche avec ILS sans utiliser de croisement, afin (on l'espère) d'être capable de visiter des régions plus prometteuses à l'aide de la puissance d'intensification de DRILS+.

6.2.2 Analyse basée sur le phénotype des optima locaux

Protocole expérimental

Dans cette deuxième expérience, nous considérons les mêmes instances que dans l'expérience précédente. Nous exécutons 10 fois indépendamment les algorithme ILS

avec $\alpha = 0.01$ et DRILS+ avec $\alpha = 0.025$ et $\beta = 0.05$, en commençant également par une solution générée aléatoirement suivi de recherche locale Hill Climber. L'ensemble des solutions obtenues à la fin des itérations de DRILS+ et ILS sont stockées dans deux ensembles distincts, notés S_{DRILS+} et S_{ILS}. Ces ensembles contiennent donc, de fait, des optima locaux. Dans la mesure où la qualité des meilleures solutions obtenues par DRILS+ sont hors d'atteinte pour ILS, même après un temps de calcul assez long, nous avons décidé de ne conserver dans l'ensemble S_{DRILS+} que les solutions dont la qualité est inférieure ou égale à la meilleure qualité obtenue par ILS. Le nombre de solutions ainsi considérées au final est de l'ordre de 8 millions. Notons que nous ne prenons pas en compte le temps nécessaire pour obtenir ces solutions, et que l'on s'intéresse uniquement à leur qualité et à l'algorithme utilisé pour leur obtention.

Maintenant que les données de ces deux ensembles ont été collectées, nous pouvons détailler plus précisément l'expérience et les observations que nous souhaitons effectuer. En effet, considérons une solution, disons x, provenant d'un ensemble des deux ensembles (obtenu par ILS ou DRILS+). Nous itérons alors 10 fois le processus suivant : générer un second parent y en utilisant une perturbation $\alpha = 0.025$ suivi d'un Hill Climber pour s'assurer que y soit un optimum local. On croise ensuite x et y pour obtenir un enfant z, auquel on applique également un Hill Climber. Ce processus est donc exactement une itération standard de DRILS (sans considérer le critère d'acceptation). Une fois ce processus terminé, pour chaque itération et pour chaque solution de chaque ensemble, soit S_{DRILS+} et S_{ILS}, nous calculons les deux métriques suivantes :

— La distance de Hamming entre les solutions x et y. Rappelons que pour les deux ensembles, et pour tout x, la perturbation appliquée pour obtenir y est de même intensité, celle considérée comme optimale pour cette instance (0.025).

— L'amélioration relative entre x et z (en ‰, notée rfi pour relative fitness improvement), c'est à dire l'amélioration empirique de la qualité de la solution attendue pour une itération de DRILS. En raison du fonctionnement du croisement par partition, cette valeur ne peut être que positive ou nulle.

Distance de Hamming après perturbation

Commençons par analyser la figure 6.2 qui correspond à la distance de Hamming entre le premier parent (x) et le deuxième (y). Rappelons que ce dernier est obtenu en perturbant 2.5% des variables de x, puis en utilisant un Hill Climber. Premièrement, on voit que la distance de Hamming décroît à mesure que la fitness de x augmente, et ce invariablement pour les solutions collectées à l'aide des deux algorithmes. On remarque également que bien que les courbes de DRILS+ et ILS soient très proches pour la première moitié du graphe (correspondant aux plus petites fitness), la distance de Hamming pour les solutions issues de DRILS+ se met brutalement à chuter beaucoup plus vite que celles

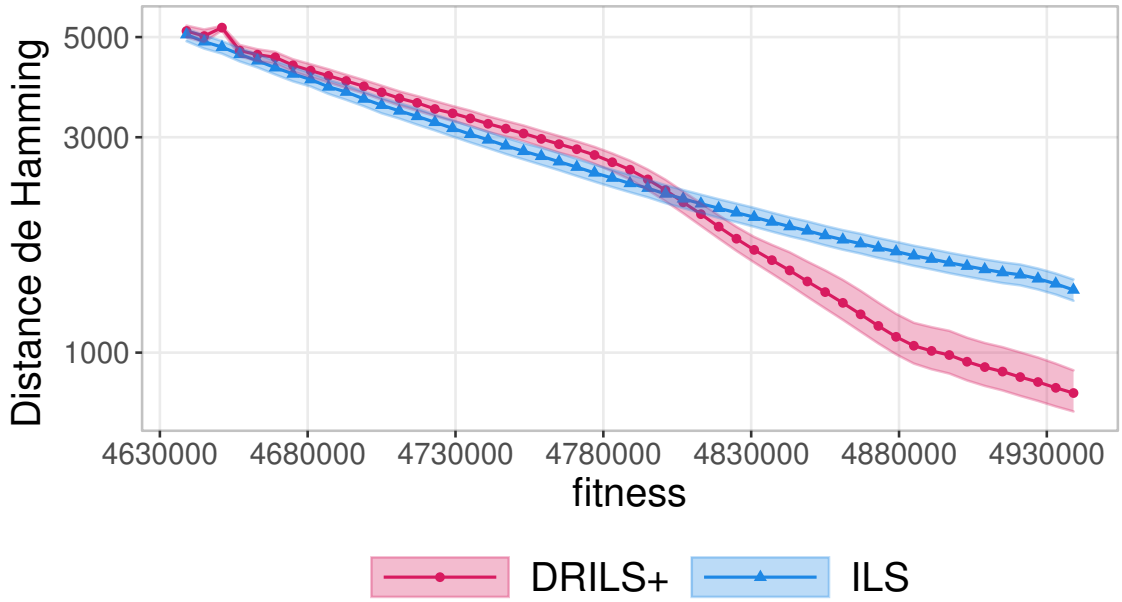

FIGURE 6.2 – *Distance de Hamming entre le premier parent et le second obtenu par une perturbation suivie d'une recherche locale. La distance est donnée en fonction de la fitness et de l'algorithme utilisé pour obtenir le premier parent. ($N = 100'000$, $K = 5$, $Q = 64$)*

issues d'ILS, qui continue à décroître au même rythme. Il est également intéressant de constater que ce changement à lieu plus ou moins au moment où la distance de Hamming atteint 2500 ce qui correspond exactement au nombre de variables qui ont été perturbées. D'ailleurs, on remarque qu'avant cela, le Hill Climber a davantage tendance à éloigner la solution y, avec 5% des variables qui ont changé d'affectation entre x et y ; tandis que pour les solutions de meilleures fitness, cette valeur est beaucoup plus faible, et même inférieur à 1% pour DRILS+. Ceci confirme l'hypothèse faite dans la partie précédente sur la dynamique d'intensification de DRILS+. En effet, DRILS+ fournit après quelques itérations des solutions qui sont dans des bassins d'attractions plus forts que les solutions fournies par ILS à qualité de fitness équivalente.

Une seconde question se pose donc maintenant. Si l'on se concentre sur le croisement par partition PX, la performance de ce dernier est fortement corrélée au nombre de partitions trouvées par l'opérateur de croisement. Ce nombre est lui aussi fortement influencé par la distance de Hamming entre les deux parents. Cependant, on a vu dans cette expérience qu'à intensité équivalente, la distance de Hamming pouvait varier du simple au quintuple en fonction de : (i) la fitness de la solution (ii) l'algorithme ayant découvert la solution. La question est donc de savoir si cet écart observé de la distance de Hamming à perturbation équivalente ne peut pas être néfaste pour l'opérateur de croisement ; dans la mesure où, la distance de Hamming entre les deux solutions à croiser

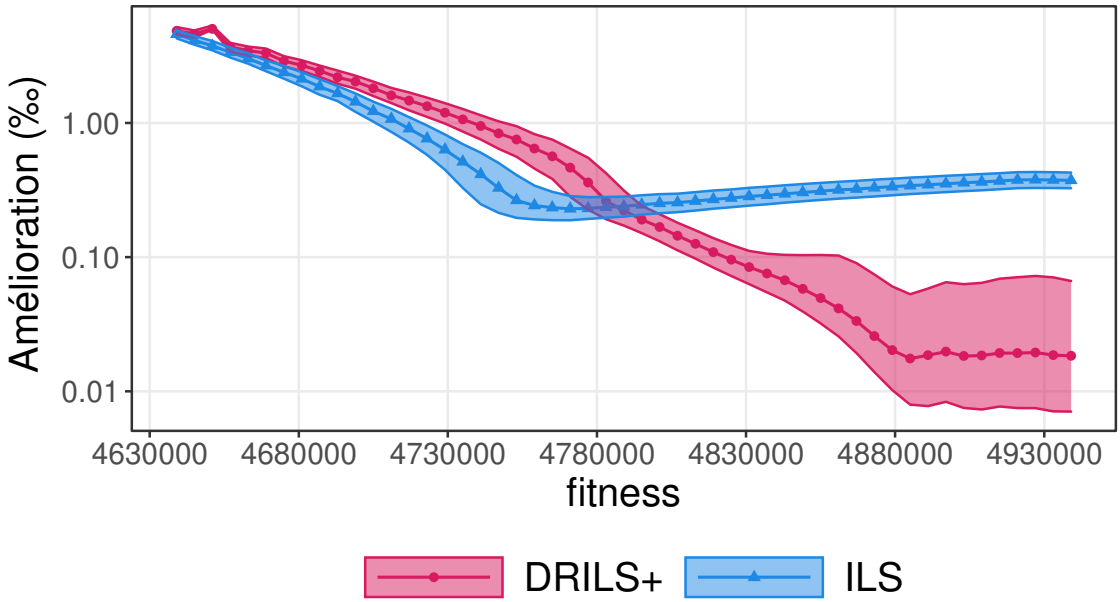

FIGURE 6.3 – *Amélioration empirique attendue (en ‰) pour l'éxécution d'une itération de DRILS, en fonction de la fitness et de l'algorithme utilisé pour obtenir le premier parent. ($N = 100000$, $K = 5$, $Q = 64$)*

x et y ne serait plus dans l'intervalle (théorique et inconnu) des valeurs optimales pour maximiser la probabilité de succès du croisement.

Pour répondre à cette question, nous analysons dans le paragraphe suivant la deuxième métrique considérée, à savoir l'efficacité empirique du croisement.

Amélioration relative empirique à l'issue d'une itération de DRILS

Dans la figure 6.3, nous pouvons observer l'amélioration relative (rfi) moyenne (en ‰) d'une itération de DRILS pour différentes solutions de départ (x) obtenues à l'aide de DRILS+ ou ILS, et ce, en fonction de la fitness de x. On peut remarquer que les deux courbes se croisent au même moment que pour la figure représentant les distances de Hamming entre les solutions parents. Avant ce croisement, on voit que la rfi décroît en fonction de la fitness pour les solutions des deux ensembles, à un rythme légèrement supérieur pour ILS comparé à DRILS+. Puis, légèrement avant le croisement, on peut clairement voir que la rfi des solutions de S_{ILS} se stabilise voir même croît très légèrement, et ce, jusqu'aux solutions ayant la meilleur fitness. D'un autre côté, la rfi des solutions de S_{DRILS+} décroît jusqu'à atteindre tardivement un plateau, et avec une rfi très inférieure (de plus d'un ordre de grandeur) comparé à ILS.

On voit donc qu'en utilisant exactement la même procédure, à savoir une itération de DRILS avec la même intensité de perturbation, l'amélioration obtenue dépend très fortement de l'algorithme ayant été exécuté pour obtenir la solution courante (utilisée en

début d'itération par DRILS). Intuitivement, on pourrait penser que la rfi dépend de la fitness, et que donc plus les solutions sont qualitatives, plus il est dur de les améliorer du même pourcentage. Cependant, il est assez surprenant de constater à travers notre expérience que cette hypothèse, pourtant souvent admise, n'est définitivement pas vérifiée.

De même que la distance de Hamming n'est pas suffisante pour expliquer ce phénomène puisque bien que l'on constate un lien pour les solutions de S_{DRILS+} entre la distance de Hamming et la rfi en fonction de la fitness, on voit clairement que la distance de Hamming dans le cas de S_{ILS} continue à décroître alors même que la rfi se stabilise voire augmente. Par exemple, pour une fitness de 4 830 000, S_{DRILS+} implique la même distance de Hamming que S_{ILS} à une fitness de 4 880 000. Mais si l'on regarde maintenant la rfi, on constate une différence d'un facteur 3 en faveur de S_{ILS} (0.1 vs 0.3) (une demi-unité en échelle logarithmique correspondant environ à 3/10 en échelle linéaire).

Discussion

Ces expériences préliminaires nous prouvent que l'algorithme DRILS+, bien que normalement plus orienté sur la diversification que DRILS (pour les instances considérées), introduit un biais d'intensification très important dans la recherche, en concentrant la recherche sur une zone relativement restreinte de l'espace. Ceci réduit ainsi fortement la capacité des futures itérations de DRILS à fournir des solutions de hautes qualités. Ce biais n'est pas présent pour les solutions obtenues par ILS. Ceci nous incite à considérer que commencer DRILS ou DRILS+ avec une solution aléatoire n'est peut-être pas optimal, dans le sens où, étant donné le biais d'intensification du croisement par partitions utilisé, il peut être primordiale de s'assurer que la solution initiale soit dans une zone prometteuse de l'espace de recherche. En effet, quitte à intensifier la recherche dans une zone restreinte, autant s'assurer que la qualité des solutions dans celle-ci soit la plus prometteuse que possible.

Il semble donc à la suite de ces expériences que ILS est un candidat prometteur pour remplir ce rôle, et de guider la trajectoire de recherche vers des zones qui sont plus intéressantes comparativement à ce que l'on peut obtenir avec seulement une initialisation aléatoire. Cependant, pour que cette approche fonctionne, il faut s'assurer que le temps passé dans cette phase d'exploration apporte un réel bénéfice comparé au retard accumulé en termes de qualité de la meilleure solution. Rappelons que la figure 6.3 ne prend pas en compte le temps de calcul. La différence de temps nécessaire pour obtenir les solutions de meilleures qualités est de plus d'un ordre de grandeur en défaveur de l'ensemble S_{ILS} comparé à S_{DRILS+}. Maintenant, que le contexte et les motivations sont posés, passons à la présentation de notre nouvelle variante de DRILS.

Algorithme 11 : PRILS

Input : α, β, γ, λ, τ

$F_{t-1} = 0$;
$F_t = 0$;
$t = 0$;
$x \leftarrow$ HBHC(random()) ;
/* Phase 1 : Exploration basée sur ILS */
repeat
  $t = t + 1$;
  $i = 0$;
  $F_t = 0$;
  repeat
    $x \leftarrow$ HBHC(Perturbation(x, γ)) ;
    $F_t = F_t + F(x)$;
    $i = i + 1$;
  until $time < (\tau \times (t/\lambda))$;
  /* Si la fitness moyenne par tranche de temps stagne, ILS a convergé donc on stoppe la phase d'exploration */
  if $F_t/i \geq F_{t-1}$ **then**
    $F_{t-1} \leftarrow F_t/i$;
  else
    break Repeat ;
  end
until $time < \tau$;
/* Phase 2 : DRILS+ pour le temps restant */
DRILS+(x, α, β, $\tau - time$) ;

6.3 PRILS : Ajout d'une phase exploratoire à DRILS

6.3.1 Description de l'algorithme

L'algorithme PRILS ('Pipelined Recombination and Iterated Local Search) est résumé dans le pseudo-code de l'algorithme 11. PRILS s'appuie sur les observations empiriques décrites dans la section précédente. Il repose sur une conception extrêmement simple. En effet, PRILS consiste essentiellement à enchaîner successivement (d'où le terme 'Pipeline') les deux phases suivantes :

— Une première phase qui consiste en l'exploration de l'espace des solutions à l'aide d'une recherche locale itérée simple avec un critère d'acceptation non-élitiste ;

— Une seconde phase qui consiste à exécuter l'algorithme DRILS+ introduit dans le chapitre précédent, en commençant avec la dernière solution obtenue à l'issue de la première phase.

Une fois ce concept général de PRILS introduit, il nous reste à préciser des points importants dans le but de concevoir un algorithme compétitif. Ces derniers seront abordés en même temps que la présentation plus détaillée de PRILS.

La première chose à considérer est la force des différentes perturbations. En effet, comme on a pu le voir dans le chapitre précèdent, la perturbation optimale pour ILS est plus faible que pour DRILS+. En ce sens, le plus simple est de considérer trois paramètres de perturbations différents, à savoir α et β pour DRILS+ (ligne 20) et le troisième que nous appellerons γ pour la perturbation de la première phase utilisant ILS (ligne 10).

Dans un second temps, étant donné que l'objectif de cet algorithme est de fonctionner pour un temps donné fixé à l'avance (τ), il nous faut trouver un moyen de répartir le temps de calcul entre les deux phases. Pour ce faire, nous proposons une méthode très simple, mais qui nous le verrons plus tard s'avère très efficace. Plus précisément, nous divisons le temps de calcul disponible en périodes en introduisant un nouveau paramètre λ. Pour chaque période, nous calculons la qualité moyenne des solutions obtenues par ILS (lignes 11). Lorsqu'à la fin de la période en cours (ligne 14), cette moyenne est inférieure ou égale à celle de la période précédente, alors on considère que ILS a convergé (ligne 17). On suppose donc qu'il n'est plus nécessaire d'explorer davantage l'espace de recherche à l'aide de cette stratégie, et que la zone dans laquelle on se trouve actuellement est suffisamment prometteuse pour commencer la deuxième phase de PRILS. Cette deuxième phase consiste simplement à exécuter l'algorithme DRILS+, pour le temps restant et en partant de la (dernière) solution courante de l'ILS en première phase.

6.3.2 Discussion

Notons que dans notre conception de PRILS, le critère de convergence que nous avons adopté, ainsi que la décision de passer d'une phase à l'autre est plutôt arbitraire, dans le sens où cela ne s'appuie pas sur une analyse empirique approfondie. Il existe très probablement des moyens plus efficaces de gérer cet équilibre entre les deux phases. Cette question requiert un travail approfondi et peut en elle-même être le fruit d'une contribution scientifique à part entière dans des travaux futurs. Notre objectif ici est de montrer que même avec une stratégie simple, il est possible d'améliorer significativement DRILS+ en lui fournissant une solution plus "prometteuse" que celle aléatoire utilisée habituellement.

Concernant les paramètres introduits dans PRILS, nous considérons une phase de calibrage en deux temps. Dans un premier temps, il s'agit de trouver le couple $\{\alpha, \beta\}$ optimal pour DRILS+. Une fois celui-ci trouvé et fixé, il s'agit de calibrer le couple restant

	N	10000		30000		100000		300000		1000000	
τ	K	3	5	3	5	3	5	3	5	3	5
1 m	▲	0	10	6	10	10	10	10	10	10	10
	≈	10	0	4	0	0	0	0	0	0	0
	▽	0	0	0	0	0	0	0	0	0	0
5 m	▲	4	10	3	8	10	10	10	10	10	10
	≈	6	0	7	2	0	0	0	0	0	0
	▽	0	0	0	0	0	0	0	0	0	0

TABLE 6.1 – *Nombre d'instances où PRILS est statistiquement meilleur (▲) ou pire (▽) que DRILS, (≈) indique que le test de Wilcoxon n'atteint pas le seuil de significativité fixé à 0.05.*

$\{\gamma, \lambda\}$ correspondant aux paramètres de la première phase. Ces paramètres feront l'objet d'une étude approfondie dans une prochaine section.

6.4 Analyse Expérimentale

6.4.1 Protocole

Afin d'étudier les performances de PRILS et de les comparer à celles de DRILS+, nous considérons des paysages NKQ avec des tailles et des degrés de non-linéarité différents. Ainsi, nous étudions les valeurs de N suivantes : {10000, 30000, 100000, 300000, 1000000}, pour K = 3 et 5. La valeur de Q sera fixée à 64. Enfin, nous utilisons 10 instances différentes par combinaison de N et de K. Pour chacune d'entre elles, nous effectuons 10 exécutions de chaque algorithmes avec les paramètres optimaux. Nous testons deux temps d'exécutions différents à savoir 1 minute et 5 minutes ; le but étant de savoir si la phase d'exploration initiale reste compétitive dans le cas de temps de calcul réduit. La phase de calibrage est effectuée pour chaque algorithme sur les couples N et K, ainsi que sur les deux temps d'exécutions. Pour cette phase, nous utilisons également 10 instances aléatoires et 10 exécutions. Comme expliqué précédemment, le calibrage de PRILS se fait en deux temps d'abord le couple $\{\alpha, \beta\}$ puis $\{\gamma, \lambda\}$. Les paramètres peuvent prendre les valeurs suivantes : {0.01, 0.025, 0.05 0.075, 0.1} pour α et β, {0.0005, 0.001, 0.005, 0.01, 0.025, 0.05} pour γ et enfin {60, 90, 120} pour λ.

6.4.2 Performance

Tests statistiques

Commençons par l'analyse statistique des performances en utilisant un test de Wilcoxon avec un seuil de significativité de 0.05. Ceci est résumé dans le tableau 6.1. Premièrement, on peut voir que PRILS n'est jamais statistiquement moins bon que DRILS+ peu importe les instances ou le temps d'exécution considéré. Pour les plus grandes instances, $N \geq 100\ 000$, PRILS est systématiquement meilleur que DRILS+, tandis que pour les instances les plus petites, on voit que la performance dépend de K. Pour $K = 3$, on voit que PRILS est meilleur sur 13 des 40 instances et pour $K = 5$ cela monte à 38 sur 40 des instances considérées. Cependant, le test de Wilcoxon ne permet pas de quantifier la différence de performances, mais uniquement la significativité. Ceci est étudié dans la suite.

Écart de performance entre PRILS et DRILS+

Afin de mieux appréhender l'écart de performances entre PRILS et DRILS+, référons-nous à la figure 6.4. Premièrement, on peut y voir que DRILS+ et PRILS obtiennent de meilleures solutions avec 5 minutes de temps de calcul comparé aux exécutions de 1 minute. Ceci est en somme assez logique et indique juste que les algorithmes ne convergent pas en 1 minute et continue d'améliorer la qualité de la meilleure solution trouvée. Ensuite, on peut voir que l'écart entre la rfd des solutions obtenues par PRILS et DRILS+ est très important pour les grandes instances et croît suivant la taille du problème, à une exception près pour les instances de taille 1 million avec 1 minute de temps de calcul. Cela s'explique par le fait que pour les instances de cette taille, l'itération de DRILS prend un temps significatif et PRILS commence juste à entrer dans sa deuxième phase, et n'a donc pas encore exploiter pleinement le potentiel de la solution fournie par la première phase.

Ces résultats indiquent aussi que même lorsque l'on dispose d'un temps de calcul restreint, il est quand même bénéfique de consacrer une portion de celui-ci à explorer l'espace de recherche durant la première phase. Le retard accumulé par PRILS par rapport à DRILS+ pendant ce temps sera ensuite rattrapé rapidement lors de la deuxième phase, et PRILS finira par obtenir de meilleures solutions avant la fin du temps alloué.

6.4.3 Comportement

Maintenant que nous avons prouvé empiriquement sur un grand nombre d'instances que PRILS est meilleur que DRILS+ (et dans le pire des cas non significativement différent de ce dernier), intéressons nous à sa dynamique de la recherche. En particulier, nous étudions l'évolution de la qualité de la meilleure solution trouvée par chaque algorithme

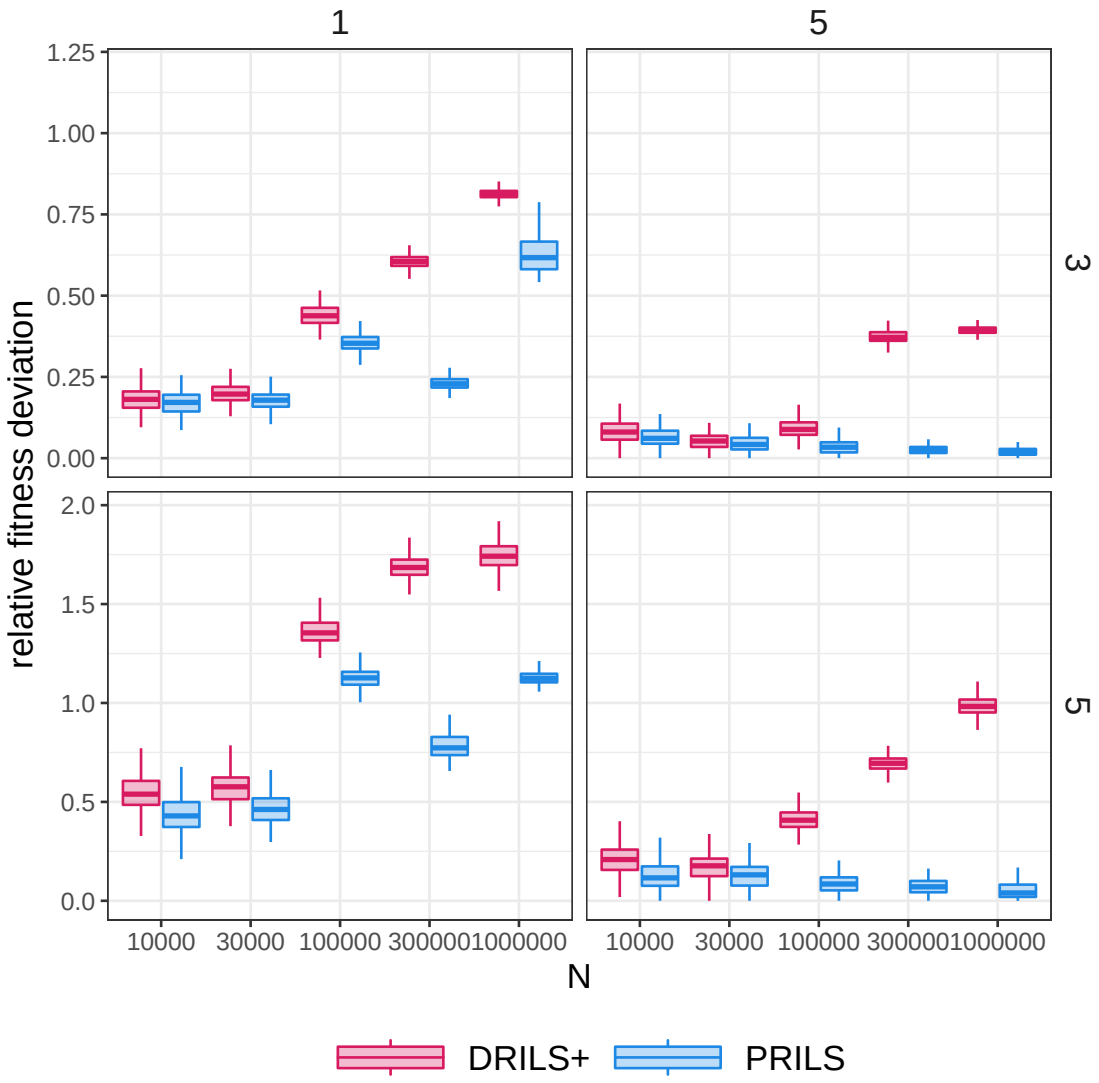

FIGURE 6.4 – *rfd de DRILS+ et PRILS pour des exécutions de 1 et 5 minutes (gauche et droite) et pour K = 3 et 5 (haut et bas) en fonction des valeurs de N croissantes.*

en fonction du temps. Ces valeurs sont regroupées dans la figure 6.5 en utilisant la rfd. On peut voir pour les instances de 100 000 variables le comportement des deux algorithmes en fonction du temps : 1 et 5 minutes sur les figures de gauche et droite respectivement ; et suivant la valeur de K : 3 et 5 de haut en bas.

On distingue clairement les deux phases de l'algorithme PRILS, et toutes les figures partagent les mêmes caractéristiques. Pour notre analyse, nous pouvons diviser le comportement global en trois parties. Dans les premiers instants de l'optimisation, PRILS est dans sa phase d'exploration à l'aide d'ILS. On remarque que la qualité de la meilleure solution obtenue par celui-ci est clairement inférieure à DRILS+. Ceci concorde avec le fait que DRILS+ tout seul est très nettement meilleur que ILS tout seul. Ensuite, on entre

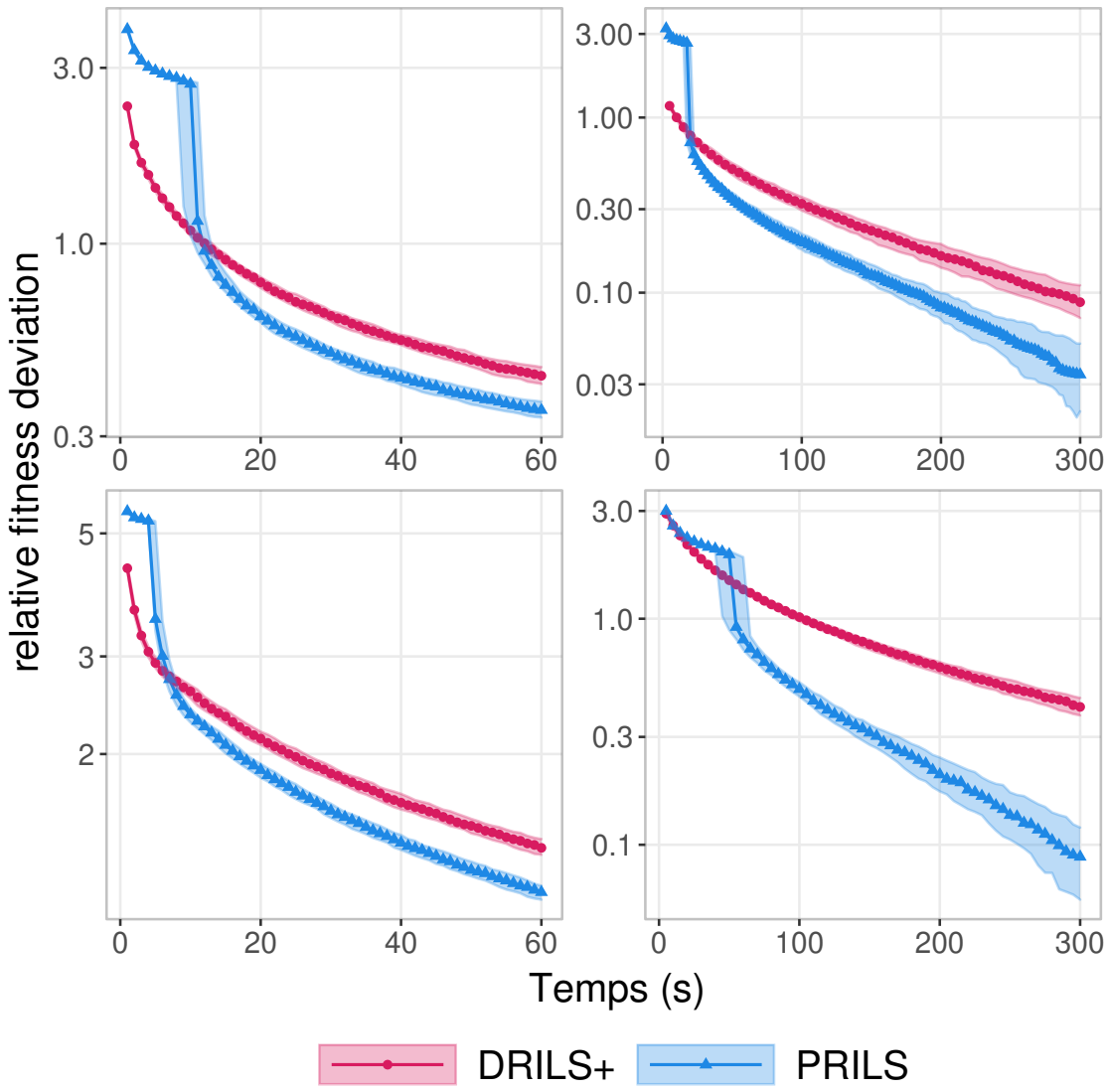

FIGURE 6.5 – *Exemple de la dynamique de recherche des différents algorithmes en fonction du temps pour $K = 3$ et 5 (haut et bas) et des temps d'exécution de 1 et 5 minutes (gauche et droite). $N = 100\,000$.*

dans la deuxième phase, où PRILS itère l'algorithme DRILS+. À partir de ce moment, les deux algorithmes utilisent uniquement DRILS+ avec les mêmes paramètres, et ce, jusqu'à la fin du temps imparti. La seule chose qui diffère à ce moment entre DRILS+ et PRILS est la solution de départ utilisée. On voit donc dans cette deuxième partie le très fort potentiel d'amélioration des solutions obtenues au cours de la première phase (ayant utilisé ILS). Ceci, rappelons le, était la principale motivation lors de la conception de PRILS. Cela rejoint donc le comportement observé au cours des expériences préliminaires réalisées dans la première section de ce chapitre. Une fois cette phase de croissance rapide terminée, on entre dans la troisième partie des graphiques où l'on remarque que les deux courbes ont le même comportement. Dans cette dernière, les deux algorithmes continuent

		Meilleurs paramètres				rfd PRILS			rfd DRILS+
N	K	α	β	γ	λ	$\lambda = 60$	$\lambda = 90$	$\lambda = 120$	
10000	3	0.075	0.025	0.0005	60	**0.069**	0.079	0.080	0.095
	5	0.025	0.025	0.01	120	0.161	0.158	**0.150**	0.243
30000	3	0.075	0.075	0.025	120	0.056	0.060	**0.055**	0.069
	5	0.025	0.05	0.01	90	0.197	**0.193**	0.201	0.233
100000	3	0.05	0.1	0.01	120	0.046	0.043	**0.042**	0.104
	5	0.025	0.05	0.01	60	**0.066**	0.093	0.093	0.386
300000	3	0.05	0.075	0.025	60	**0.032**	0.037	0.044	0.384
	5	0.01	0.05	0.01	60	**0.061**	0.086	0.100	0.692
1000000	3	0.05	0.05	0.025	60	**0.015**	0.020	0.032	0.386
	5	0.025	0.025	0.01	60	**0.038**	0.047	0.100	0.981

TABLE 6.2 – *Meilleures paramètres trouvées pour PRILS ($\tau = 5$ m), suivi de la rfd moyenne (en %) de PRILS avec différentes valeurs de $\lambda \in \{60, 90, 120\}$, ainsi que la meilleure rfd moyenne obtenue par DRILS+, (En gras la rfd moyenne la plus basse par instance).*

à améliorer la qualité des meilleures solutions trouvées au même rythme (suivant une échelle logarithmique). Il est intéressant de noter que DRILS+ ne parvient pas à rattraper son retard même après 5 minutes. Ceci est un argument supplémentaire en faveur de l'interprétation selon laquelle DRILS+ se concentre sur un espace restreint de l'espace de recherche. La différence étant que cet espace est composé de solutions plus qualitatives dans le cas de PRILS.

6.4.4 Analyse des paramètres

Terminons notre étude par l'analyse des paramètres optimaux et de la robustesse de PRILS par rapport aux nouveaux paramètres introduits (γ, λ). Étant donné la méthodologie utilisée pour calibrer α et β, nous ne reviendrons pas sur ces derniers, car il s'agit des meilleurs paramètres pour DRILS+ (étudié dans le chapitre précédent). La partie gauche du tableau 6.2 regroupe pour différentes valeurs de N et K les paramètres optimaux, tandis que la partie droite se concentre sur la performance relative de PRILS en fonction de différentes valeurs de λ, ainsi que celle de DRILS+ sur la dernière colonne.

Paramètres optimaux

Commençons cette analyse par la première partie du tableau, et plus particulièrement les deux dernières colonnes de celui-ci. Concernant le paramètre γ, la valeur optimale dépend de la rugosité de l'instance. Quand celle-ci est élevée, pour $K = 5$, la valeur est de 1% peu importe la taille de l'instance considérée. Tandis que pour les paysages moins

rugueux, soit K=3, on voit que la taille de l'instance influe sur la valeur de γ. Cependant, si l'on excepte la plus petite instance où cette valeur est de 0.05%, on remarque que cette dernière oscille entre 1% et 2.5%.

Passons maintenant au second paramètre introduit par PRILS, à savoir λ, sur la dernière colonne du tableau. Rappelons que ce paramètre correspond au nombre de périodes utilisées pour découper le temps d'exécution de l'algorithme et ainsi inférer le critère de convergence pour la première phase. La valeur la plus commune est 60, on voit en particulier que cette valeur est stable pour les deux plus grandes tailles d'instances, tandis que pour les autres, la valeur optimale de λ varie entre 60, 90 et 120.

Impact de λ sur la performance

Focalisons-nous désormais sur l'impact du paramètre λ sur la performance de PRILS (les trois premières colonnes de la seconde partie du tableau), et surtout , comparons cette dernière à celle de DRILS+ (dernière colonne). Cette performance est mesurée à l'aide de la rfd, ainsi, plus cette valeur est faible plus l'algorithme est efficace. Si l'on regarde les cinq premières instances, on voit que l'écart entre les rfd pour différentes valeurs de λ est relativement faible. L'impact de ce dernier paramètre est donc assez faible sur la qualité de la meilleure solution obtenue au final. Maintenant, si l'on passe aux cinq dernières instances, on voit que l'importance de λ augmente et l'écart entre les rfd augmente. On peut donc conclure que l'importance du calibrage de λ est globalement proportionnel à la fois à la taille et à la rugosité du problème.

Il est également intéressant de noter que même en considérant la pire des valeurs pour λ, les solutions obtenues par PRILS sont bien meilleurs que celles obtenues par DRILS+ ; avec parfois un écart plus que considérable. Par exemple, pour $N = 1$ million et $K = 5$, la rfd de DRILS+ est plus de 9.81 fois plus élevée que celle de PRILS avec $\lambda = 120$, 20.87 fois pour $\lambda = 90$ et 25.81 fois pour $\lambda = 60$. On peut donc, au vu de ces analyses, avancer le fait que PRILS surpasse aisément DRILS+ ; et ce même en utilisant une valeur générique de λ, par exemple 60.

6.5 Conclusion

Dans ce chapitre, nous avons montré empiriquement l'importance de la première solution utilisée par DRILS+. Nous avons également mis en lumière le biais d'intensification introduit par l'utilisation successive des opérateurs de croisements, et ce même en considérant la capacité de DRILS+ à apporter davantage de diversification de par son mécanisme d'échappement plus puissant que celui de DRILS. En comparant l'amélioration relative empirique d'une itération de DRILS avec une solution obtenue par DRILS+ ou par ILS, nous avons vu que ILS offrait davantage de potentielle d'amélioration à fitness

égale. De par sa capacité à "nourrir" DRILS+ avec des solutions prometteuses, ILS est une bonne alternative à l'utilisation d'une première solution aléatoire. Ainsi, nous avons proposé une nouvelle variante de DRILS appelé PRILS qui combine ILS (avec un critère d'acceptation non-élitiste) et DRILS+. L'idée est de découper le temps de calcul disponible en deux phases : la première explore l'espace de recherche à l'aide d'ILS et la deuxième exécute DRILS+ pour le temps restant avec la dernière solution obtenue lors de la première phase. Cette transition dépend d'un critère de convergence simple qui consiste à calculer la qualité moyenne des solutions obtenues par intervalle de temps. Cet intervalle étant fixé à l'aide d'un paramètre de l'algorithme, λ, le nombre de périodes. Nous avons ensuite montré empiriquement que cet algorithme est statistiquement meilleur que DRILS+ sur un grand nombre d'instances de tailles et de rugosités différentes, et ce, pour différents temps d'exécution.

Dans ce chapitre, ainsi que tous ceux qui le précèdent, nous avons uniquement considéré l'opérateur de croisement par partitions (PX). Il est donc intéressant d'étudier la combinaison des variantes de DRILS avec les derniers opérateurs de croisement proposés dans la littérature ; à savoir le croisement avec analyse des points d'articulation (APX) et le croisement avec programmation dynamique (DPX). Le dernier chapitre de cette thèse est consacré à cet aspect.

Chapitre 7

Étude des comportements des différents croisements boîte grise sur DRILS et ses variantes

Dans ce chapitre, nous comparerons les combinaisons possibles entre les différentes variantes de DRILS proposées précédemment et les différents croisements boîte grise de la littérature. Pour ce faire, nous utiliserons une grande variété d'instances NK de tailles et de rugosités différentes. Au vu des expériences que nous sommes amenées à effectuer, nous nous assisterons également d'une librairie de calibration des paramètres, à savoir irace [27], afin de rendre notre comparaison la plus juste possible, sans pour autant rentrer dans des campagnes d'expérimentations factorielles très coûteuses.

Ainsi, nous proposons dans ce chapitre une analyse détaillée du fonctionnement de DRILS et des croisements boîte grise. Le but étant d'améliorer notre compréhension de la dynamique de DRILS et de ces variantes DRILS+ et PRILS, ainsi que de tenter d'apporter des explications plus fondamentales des différences de performances observées.

7.1 Analyse globale de performance

Dans la première partie de ce chapitre, notre objectif est de comparer la performance des différentes combinaisons des variantes de DRILS et des croisements état de l'art existants. Ceci servira aussi à tirer des enseignements et des recommandations quant aux meilleures configurations d'algorithmes possibles pour un type d'instance donnée.

7.1.1 Protocole Expérimental

Combinaisons algorithme/croisement

Nous considérerons les trois algorithmes DRILS, DRILS+ et PRILS, ainsi que les trois opérateurs connus de croisements boite grise pour les fonctions pseudo-booléens considérés, à savoir, le croisement par partitions (PX), le croisement par partitions avec analyse des points d'articulation (APX) et pour finir le croisement par partitions avec programmation dynamique (DPX). Ces trois croisements ont été décrits auparavant dans la section 2.4. Rappelons que le croisement DPX est très gourmand en temps de calcul ainsi qu'en mémoire. De ce fait, pour les plus grandes tailles d'instances, à savoir 300 000 et 1 000 000, nous ne considérerons pas DPX mais seulement PX et APX. Nous avons ainsi respectivement 9 combinaisons/algorithmes possibles pour les plus petites instances et 6 pour les plus grandes.

Instances

Nous considérons des instances NKQ avec N dans $\{10, 30, 100, 300, 1000\}$ milliers de variables, et K dans $\{2, 3, 4, 5, 6\}$ interactions, ce qui représente 25 combinaisons possibles du couple (N,K). Nous utiliserons une valeur de Q = 10 000. Nous tenons à noter que, bien que nous avons également fait les mêmes expériences et obtenu des résultats très similaires pour une valeur de Q de 64, ces derniers ne seront pas montrés ; premièrement, pour des soucis de lisibilité et de concisions, et deuxièmement, car les instances avec cette valeur de Q font déjà l'objet d'études approfondies dans les chapitres précédents. Notons pour finir qu'un paysage NKQ avec Q = 10 000 est strictement équivalent à un paysage NK avec une précision des contributions des fonctions de 4 décimales.

Calibrage des paramètres

Pour réaliser nos expériences, nous suivrons le protocole suivant. Pour chacune des combinaisons de N et K, et pour chaque combinaison possible d'algorithme/croisement, nous calculons les paramètres optimaux en utilisant irace avec un budget de 1 000 itérations et des temps de calcul de 5 minutes. Ceci représente un total de 195 000 exécutions et un temps de calcul agrégé de 16 250 heures soit 677 jours.

Le nombre de paramètres à calibrer dépend de la combinaison algorithme/croisement et varie de 1 paramètre pour DRILS/PX à 5 pour PRILS/DPX. Une fois ces paramètres optimaux obtenus, 20 *nouvelles* instances aléatoires de paysage NK sont générées par valeur de N et K. Pour chacune d'entre elle, nous effectuons 20 exécutions indépendantes de 5 minutes pour chaque combinaison algorithme/croisement. Ceci représente un total de 78 000 exécutions pour un temps de calcul global de 6 500 heures.

Comparaison

Les résultats des différents algorithmes sont comparés en utilisant la valeur de fitness médiane obtenue après les 20 exécutions indépendantes, et ce, pour chaque instance. On utilise ensuite un test de Wilcoxon avec un seuil de significativité de 0.05 pour comparer les combinaisons algorithme/croisement entre elles. Le rang d'une combinaison algorithme/croisement est obtenu en comptant le nombre de fois où ce dernier est statistiquement moins bon qu'une autre combinaison. Ainsi, pour les plus petites instances, le rang peut varier de 0 dans le meilleur des cas, à 8 dans le cas où la combinaison algorithme/croisement est statistiquement inférieur à toutes les autres. Pour les instances les plus grandes, le rang varie de 0 à 5.

7.1.2 Résultats

Le tableau 7.1 affiche le rang obtenu par les différentes combinaisons étudiées sur des instances NK avec N allant de 10 000 à 1 000 000 et K allant de 2 à 6. Les meilleures combinaisons pour chaque couple (N,K) sont affichées en gras.

Vue générale

Dans l'ensemble, on remarque que le meilleur algorithme est toujours PRILS peu importe la taille et la rugosité des paysages considérés à l'exception des paysages avec N=30 000 et K=2 où DRILS+ est meilleur. On voit également que pour 3 autres instances PRILS et DRILS+ ont des résultats qui ne sont pas statistiquement différents, à savoir : N=10 000, K=3 et 6 ; et N=100 000, K=2. Concernant le meilleur croisement, ce dernier oscille entre APX et DPX ; PX n'étant meilleur que dans un seul cas pour N=300 000 et K=6. On peut voir que le meilleur croisement dépend principalement de la taille des instances considérées, DPX étant meilleur sur les petites instances et APX meilleur sur les grandes. Ceci est analysé plus en détail dans ce qui suit.

Paysages compris entre 10 000 et 100 000 variables

Pour les paysages composés de 10 000 variables, on voit que le choix de l'opérateur de croisement à un impact plus fort que celui de la variante de DRILS utilisée. En particulier, pour $K \geq 4$, DPX occupe les 3 premières places. On voit également que PX est le moins bon des croisements étudiés. PRILS est le meilleur algorithme, parfois égalé par DRILS+. Tandis que pour les plus petites valeurs de K, DRILS à tendance à offrir les moins bonnes performances, en particulier quand il est associé à PX ; on peut cependant remarquer que pour les plus grandes valeurs de K, DRILS est meilleur que DRILS+ lorsque les deux algorithmes sont utilisés avec PX.

N	$K=2$			$K=3$			$K=4$			$K=5$			$K=6$		
	Algorithme	X	*rang*	Algorithme	X	*rang*	Algorithme	X	*rang*	Algorithme	X	*rang*	Algorithme	X	*rang*
10'000	**PRILS**	**DPX**	0	**PRILS**	**DPX**	0	**PRILS**	**DPX**	0	**PRILS**	**DPX**	0	**PRILS**	**DPX**	0
	DRILS+	*DPX*	1	**DRILS+**	**DPX**	0	*DRILS+*	*DPX*	1	*DRILS+*	*DPX*	1	**DRILS+**	**DPX**	0
	PRILS	*APX*	2	*PRILS*	*APX*	2	*DRILS*	*DPX*	1	*DRILS*	*DPX*	2	*DRILS*	*DPX*	2
	DRILS+	*APX*	2	*DRILS+*	*APX*	2	*PRILS*	*APX*	3	*PRILS*	*APX*	3	*PRILS*	*APX*	3
	PRILS	*PX*	4	*DRILS*	*DPX*	4	*DRILS+*	*APX*	4	*DRILS+*	*APX*	4	*DRILS+*	*APX*	3
	DRILS+	*PX*	4	*PRILS*	*PX*	4	*DRILS*	*APX*	4	*DRILS*	*APX*	5	*PRILS*	*PX*	5
	DRILS	*DPX*	6	*DRILS+*	*PX*	5	*PRILS*	*PX*	6	*PRILS*	*PX*	6	*DRILS*	*APX*	6
	DRILS	*APX*	7	*DRILS*	*APX*	7	*DRILS+*	*PX*	7	*DRILS*	*PX*	7	*DRILS*	*PX*	7
	DRILS	*PX*	8	*DRILS*	*PX*	8	*DRILS*	*PX*	8	*DRILS+*	*PX*	8	*DRILS+*	*PX*	8
30'000	**DRILS+**	**DPX**	0	**PRILS**	**DPX**	0	**PRILS**	**DPX**	0	**PRILS**	**DPX**	0	**PRILS**	**DPX**	0
	PRILS	*DPX*	1	*DRILS+*	*DPX*	1	*PRILS*	*APX*	1	*PRILS*	*APX*	1	*PRILS*	*APX*	1
	PRILS	*APX*	2	*DRILS*	*DPX*	1	*DRILS+*	*DPX*	2	*DRILS+*	*DPX*	2	*DRILS+*	*DPX*	2
	DRILS+	*APX*	2	*PRILS*	*APX*	3	*DRILS+*	*APX*	3	*DRILS+*	*APX*	3	*DRILS+*	*APX*	3
	PRILS	*PX*	4	*DRILS+*	*APX*	4	*DRILS*	*DPX*	4	*PRILS*	*PX*	4	*PRILS*	*PX*	4
	DRILS+	*PX*	4	*DRILS*	*APX*	5	*PRILS*	*PX*	4	*DRILS+*	*PX*	5	*DRILS+*	*PX*	4
	DRILS	*DPX*	6	*PRILS*	*PX*	6	*DRILS*	*APX*	6	*DRILS*	*DPX*	6	*DRILS*	*DPX*	6
	DRILS	*APX*	7	*DRILS+*	*PX*	7	*DRILS+*	*PX*	7	*DRILS*	*PX*	7	*DRILS*	*APX*	7
	DRILS	*PX*	8	*DRILS*	*PX*	8	*DRILS*	*PX*	8	*DRILS*	*APX*	8	*DRILS*	*PX*	8
100'000	**PRILS**	**APX**	0	**PRILS**	**APX**	0	**PRILS**	**APX**	0	**PRILS**	**APX**	0	**PRILS**	**DPX**	0
	PRILS	**DPX**	0	*PRILS*	*DPX*	1	*PRILS*	*DPX*	1	*PRILS*	*DPX*	1	*PRILS*	*PX*	1
	DRILS+	**DPX**	0	*DRILS+*	*APX*	2	*PRILS*	*PX*	2	*PRILS*	*PX*	2	*PRILS*	*APX*	2
	DRILS+	*APX*	3	*DRILS*	*APX*	3	*DRILS+*	*APX*	3	*DRILS+*	*APX*	3	*DRILS+*	*APX*	3
	DRILS	*DPX*	4	*DRILS+*	*DPX*	4	*DRILS+*	*PX*	4	*DRILS+*	*PX*	4	*DRILS+*	*DPX*	4
	PRILS	*PX*	5	*PRILS*	*PX*	5	*DRILS+*	*DPX*	4	*DRILS+*	*DPX*	5	*DRILS+*	*PX*	4
	DRILS+	*PX*	5	*DRILS*	*DPX*	6	*DRILS*	*APX*	6	*DRILS*	*PX*	6	*DRILS*	*PX*	6
	DRILS	*APX*	7	*DRILS+*	*PX*	6	*DRILS*	*PX*	7	*DRILS*	*APX*	7	*DRILS*	*APX*	7
	DRILS	*PX*	8	*DRILS*	*PX*	8	*DRILS*	*DPX*	8	*DRILS*	*DPX*	8	*DRILS*	*DPX*	8
300'000	**PRILS**	**APX**	0	**PRILS**	**APX**	0	**PRILS**	**APX**	0	**PRILS**	**APX**	0	**PRILS**	**PX**	0
	DRILS+	*APX*	1	*PRILS*	*PX*	1	*PRILS*	*PX*	1	*PRILS*	*PX*	1	*PRILS*	*APX*	1
	DRILS	*APX*	1	*DRILS+*	*APX*	2	*DRILS+*	*APX*	2	*DRILS+*	*APX*	2	*DRILS+*	*APX*	2
	PRILS	*PX*	3	*DRILS+*	*PX*	3	*DRILS+*	*PX*	3	*DRILS+*	*PX*	3	*DRILS+*	*PX*	3
	DRILS+	*PX*	3	*DRILS*	*APX*	4	*DRILS*	*APX*	4	*DRILS*	*APX*	4	*DRILS*	*APX*	4
	DRILS	*PX*	5	*DRILS*	*PX*	5	*DRILS*	*PX*	5	*DRILS*	*PX*	5	*DRILS*	*PX*	5
1'000'000	**PRILS**	**APX**	0	**PRILS**	**APX**	0	**PRILS**	**APX**	0	**PRILS**	**APX**	0	**PRILS**	**APX**	0
	DRILS+	*APX*	1	*PRILS*	*PX*	1	*PRILS*	*PX*	1	*PRILS*	*PX*	1	*PRILS*	*PX*	1
	DRILS	*APX*	1	*DRILS+*	*APX*	2	*DRILS+*	*APX*	2	*DRILS+*	*APX*	2	*DRILS+*	*APX*	2
	PRILS	*PX*	3	*DRILS*	*PX*	3	*DRILS*	*APX*	2	*DRILS*	*APX*	3	*DRILS*	*APX*	2
	DRILS+	*PX*	4	*DRILS+*	*PX*	4	*DRILS+*	*PX*	4	*DRILS+*	*PX*	4	*DRILS+*	*PX*	4
	DRILS	*PX*	4	*DRILS*	*APX*	5	*DRILS*	*PX*	4	*DRILS*	*PX*	4	*DRILS*	*PX*	4

TABLE 7.1 – *Performances agrégées des combinaisons algorithmes/croisements en fonction de N et K. Les algorithmes sont classés par ordre croissant en utilisant le rang, qui correspond aux nombres de comparaisons statistiques défavorables. En gras les algorithmes qui ne sont jamais battus. On utilise un test de Wilcoxon avec un seuil de 0,05.*

Concernant les paysages de 30 000 variables, pour les valeurs de $K \geq 4$, on observe le même classement pour les 4 meilleures combinaisons à savoir PRILS avec DPX suivi de APX, puis DRILS+ avec DPX suivi de APX. Ceci indique que le choix de la variante de DRILS est ici plus important que celui du croisement. On note que DRILS est, pour toutes les valeurs de K, toujours le moins bon algorithme ; tandis que PRILS est le meilleur à l'exception du cas où K=2. C'est alors DRILS+ qui surpasse statistiquement PRILS, les deux variantes étant associées à DPX. On remarque pour finir qu'à algorithme constant, le classement des opérateurs de croisement est toujours le même, DPX surpassant APX, lui-même meilleur que PX.

Passons maintenant au cas des instances de taille N = 100 000. On peut voir que l'algorithme est d'une importance capitale puisque PRILS est toujours l'algorithme le plus efficace. De même, DRILS est systématiquement le moins bon des algorithmes considérés. Concernant les croisements, on peut voir que APX et DPX obtiennent de très bons résultats. APX étant meilleur pour K=3 à 5 suivi par DPX, tandis que les deux obtiennent des performances équivalentes pour K=2. Cependant, pour l'instance la plus rugueuse, on remarque que APX est moins bon que DPX qui occupe la meilleure place du classement, suivi par PX.

C'est également l'occasion de noter que les combinaisons des variantes de DRILS et des croisements changent complètement la dynamique et la qualité des algorithmes. Si l'on se concentre sur les instances N = 100'000 et K = 6, on peut voir que PRILS est toujours meilleur que DRILS+ qui lui-même est toujours meilleur que DRILS. Cependant, lorsque l'on prend en compte le croisement combiné à ces variantes de DRILS, on voit que le classement est différent pour chacun d'entre eux. Ainsi, avec PRILS, c'est APX qui obtient les moins bons résultats, tandis que le même opérateur de croisement sera celui qui obtient les meilleurs résultats si l'on considère DRILS+. Il est donc très difficile de prédire à l'avance si une combinaison algorithme/croisement sera meilleure qu'une autre.

Paysages de 300 000 variables et plus

Nous arrivons maintenant sur les instances les plus grandes pour lesquelles DPX n'est pas considéré en raison du temps de calcul et de la quantité de mémoire nécessaire pour son fonctionnement.

Pour 300 000 variables, on peut voir que pour K=2, le type croisement est plus important que la variante d'algorithme utilisée. Ainsi, APX surpasse toujours PX. Il est aussi intéressant de noter que le classement du meilleur algorithme est le même pour PX et APX ; avec PRILS ≥ DRILS+ ≥ DRILS. Pour les instances de plus grande rugosité, soit $K \geq 3$, c'est l'algorithme qui devient le plus important peu importe le croisement. Ainsi, PRILS (avec PX ou APX) est meilleur que DRILS+ (avec PX ou APX). On peut voir également

qu'à algorithme constant, APX est meilleur que PX, hormis pour le cas le plus rugueux, avec $K = 6$ où la meilleure combinaison est PRILS avec PX.

Pour les instances d'un million de variables, on remarque que les résultats sont très proches des instances avec 300 000 variables. Ceci à l'exception que pour un grand nombre d'instances, DRILS+ et DRILS avec le même croisement obtiennent des résultats similaires. On peut observer ceci notamment pour $K = 2, 4, 5, 6$. Cela s'explique par le fait que le critère d'acceptation qui différenci DRILS et DRILS+ n'intervient que quand le croisement échoue ; ce qui pour les instances de cette taille arrive plutôt rarement pendant les 5 minutes de temps de calcul allouées.

7.1.3 Discussion

L'analyse des différentes combinaisons possibles entre variantes de DRILS et croisements boîte grise sur un grand nombre de paysages NK de tailles et de rugosités différentes nous montre que PRILS est le meilleur algorithme dans 24 des 25 cas. Ce dernier est surpassé uniquement une fois par DRILS+. Ainsi, PRILS peut être considéré à juste titre comme le meilleur algorithme pour les paysages NK large échelle et devrait ainsi être le premier algorithme à considérer pour résoudre ces problèmes. Concernant le croisement optimal, une tendance se dégage à quelques exceptions près. APX est meilleur pour les problèmes de très grandes tailles, tandis que DPX performe mieux sur les instances relativement plus petites.

Durant nos expériences, nous avons remarqué que le temps de calcul joue un rôle important pour les instances avec 100 000 variables. En effet, pour ces dernières, la supériorité de APX sur DPX est principalement due au temps de calcul. Globalement, DPX est meilleur, mais requiert plus de temps. Ainsi, le temps de calcul de 5 minutes utilisé dans nos expériences favorise APX du fait de sa plus grande vitesse d'exécution. Cependant, pour des temps de calcul plus long, DPX peut finir par dépasser APX. Un tel comportement est intéressant, car il encourage à creuser la piste de l'utilisation de plusieurs opérateurs de croisements au cours d'une même exécution.

7.2 Étude de la dynamique de DRILS

Dans cette section, nous essayons de mieux comprendre et expliquer les performances des différentes variantes d'algorithmes considérées.

7.2.1 Chaine de DRILS

Avant de commencer la phase d'analyse expérimentale, nous introduisons la notion de *chaîne DRILS*. Prenons comme exemple illustratif le schéma de la figure 7.1. Considérons

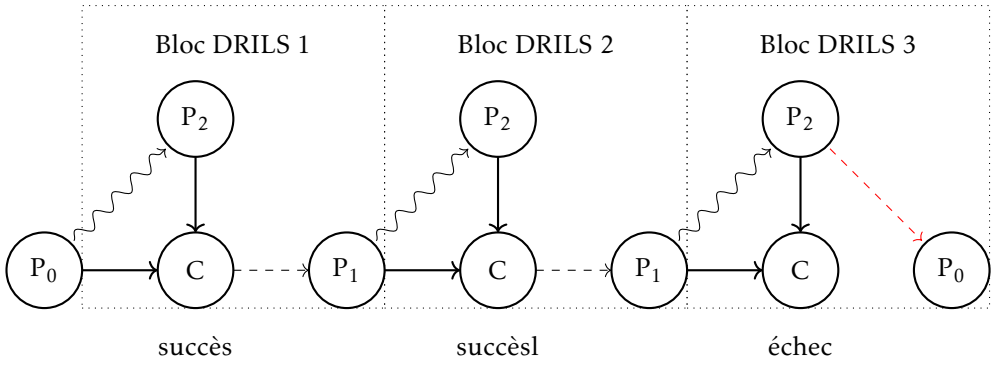

FIGURE 7.1 – *Exemple d'une chaine DRILS composée de 3 itérations ; la chaine se termine en raison de l'échec du croisement de la 3ème itération. La sortie finale de cette chaîne est le troisième optimum local C. Cependant, en raison du mécanisme d'échappement, si l'on démarre une nouvelle chaîne DRILS le P_0 final sera utilisé comme solution de départ. La premier Optima Local P_0 peut provenir d'une solution aléatoire suivie d'une Recherche Locale, d'une ILS ou même d'une autre chaîne DRILS ou DRILS+ tant qu'aucun bloc DRILS précédent n'a été appliqué à cette solution.*

une solution initiale P_0. Il est important pour notre analyse d'émettre l'hypothèse que cette dernière n'ait jamais été précédée d'une itération réussie de DRILS, le cas échéant, il faut que P_0 soit passé par le mécanisme d'échappement de DRILS ou DRILS+.

Cette solution est utilisée au début d'une itération de DRILS en tant que premier parent P_1. L'itération de DRILS effectue ensuite des croisements successifs entre les parents P_1 et P_2. Dans le cas où le croisement serait réussi, c'est-à-dire que l'enfant obtenu C est différent de P_1 et P_2, alors on considère que cette itération de DRILS est réussie. On continue alors avec C en guise de premier parent P_1 pour la prochaine itération. Ainsi, on enchaîne les itérations de DRILS jusqu'à ce qu'une itération ne parvienne pas à réussir son étape de recombinaison par croisement. Dans ce cas, l'enfant obtenu C est considéré comme la solution de fin de chaîne avec la garantie, en raison du comportement de DRILS, que C soit la meilleure solution de toute la chaîne, ou en tout cas, meilleur ou égale que toutes les autres solutions rencontrées.

Pour l'instant, le comportement attendu dans une telle chaîne est le même pour DRILS et DRILS+. En effet, l'exécution de la chaîne est exactement la même pour les deux algorithmes, et la seule chose qui les différencie est la stratégie d'échappement. Les deux algorithmes partagent donc le même comportement de recherche jusqu'à ce qu'une itération de DRILS soit infructueuse. Supposons qu'il s'agisse de la 3e itération d'une chaîne, comme illustrée dans notre exemple. Alors, suivant l'algorithme, DRILS ou DRILS+, la nouvelle solution P_0 de début de la chaîne suivante sera obtenue respectivement en copiant P_2 ou en générant P_0 en partant de P_1, auquel cas une perturbation β est appliqué, suivi d'une recherche locale. Ce nouveau P_0 est ensuite utilisé pour initier une nouvelle chaîne de DRILS, et ainsi de suite. Ainsi, la longueur d'une chaîne de DRILS correspond

au nombre d'itérations nécessaires pour qu'en partant d'une solution de départ P_0, DRILS finisse par échouer à une étape de croisement.

Dans la suite de cette section, nous utiliserons cette notion de chaîne afin de mieux étudier et de comprendre le comportement de DRILS, DRILS+ et PRILS. En particulier, nous verrons :

— Premièrement, que l'amélioration attendue en fin de chaîne est fortement dépendante de la solution de départ P_0 utilisée, et donc indirectement de l'algorithme exécuté pour l'obtenir, mais aussi de l'intensité de la perturbation β dans le cas de DRILS+ ;

— Deuxièmement, que l'amélioration apportée par une itération de DRILS (donc entre les solutions P_1 et C de cette itération) dépend davantage de l'algorithme utilisé pour générer la solution de départ P_0, et du nombre d'itérations précédentes dans la chaîne, plutôt que de la fitness de la solution P_1.

7.2.2 Protocole expérimentale

Afin de réaliser nos analyses, nous utiliserons des instances NKQ avec $N = 100\,000, K = 5$ et $Q = 64$. Les algorithmes suivants seront étudiés :

— ILS avec un facteur de perturbation de 1% ;

— DRILS avec $\alpha = 2.5\%$;

— DRILS+ avec un α de 2.5% et plusieurs valeurs de $\beta \in \{1\%, 5\%, 7.5\%\}$.

Pour chaque algorithme, nous effectuons plusieurs exécutions indépendantes. Pendant 5 minutes, nous enregistrons pour chaque chaîne de DRILS les informations suivantes : la solution P_0 de début de chaîne et le temps de calcul en début et fin de chaîne ; et pour la dernière itération en fin de chaîne : les solutions p_1, p_2 et C. Notons qu'étant donné que ILS n'utilise pas de croisement, nous sauvegarderons les valeurs de la solution courante (qu'on considère comme un P_0) toutes les 100 itérations, dues au fait qu'ILS génère beaucoup plus de données que les autres algorithmes.

Nous utilisons ensuite ces données pour effectuer 10 exécutions indépendantes de chaînes de DRILS (nouvelles/supplémentaires) avec un facteur *alpha* de 2.5%, en partant des P_0 ainsi collectés. Nous enregistrerons, pour chaque chaîne de DRILS ainsi exécutée, la fitness de la solution de départ P_0 ainsi que les données des premières, dixièmes, centièmes et dernières itérations de la chaîne. Ce sont donc essentiellement ces dernières données qui sont utilisées pour notre analyse.

Pour ces expériences, nous utilisons le croisement obtenant les meilleurs résultats sur les instances considérées, à savoir APX.

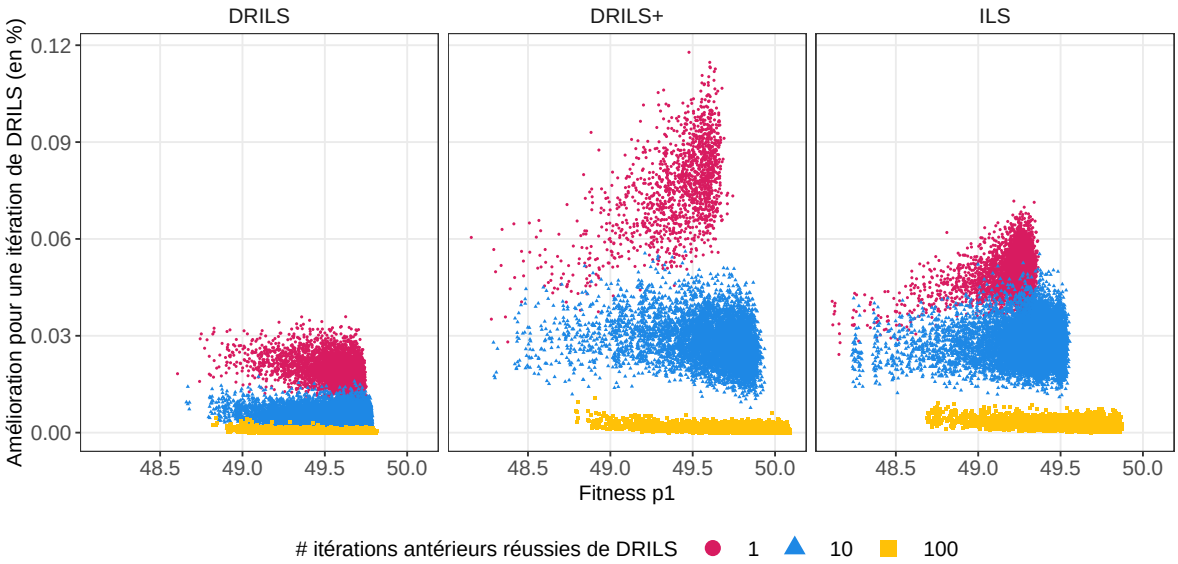

FIGURE 7.2 – *Amélioration en pourcentage attendue d'une itération de DRILS avec APX en fonction de la fitness du premier parent, du nombre d'itérations antérieurs réussies de DRILS et de l'algorithme utilisé pour obtenir la première solution pour commencer la chaîne de DRILS.*

7.2.3 Analyse des résultats

Amélioration en fonction du nombre d'itérations successives

Nous commençons par mettre en lumière le biais d'intensification introduit par les croisements successifs dans une chaîne de DRILS. La figure 7.2 montre l'amélioration en pourcentage apportée par une itération de DRILS : c'est-à-dire la différence entre les paires de solutions C et P_1. Et ceci, premièrement, en fonction de la fitness de P_1 représentée sur l'axe x. Et, deuxièmement, en fonction de l'algorithme, soit de gauche à droite DRILS, DRILS+ et ILS. Et enfin, troisièmement, en fonction de l'itération de la chaîne de DRILS dans laquelle on se trouve, représentée sur la figure par des couleurs différentes.

On peut voir que la fitness du premier parent P_1 n'a que peu d'effet sur l'amélioration attendue d'une itération de DRILS. Il est intéressant de noter que pour un même algorithme, et à fitness égale, l'amélioration attendue décroît fortement avec le nombre d'itérations précédentes de la chaîne de DRILS. Cet effet est particulièrement marqué pour DRILS+, et tout de même bien visible pour ILS et DRILS.

Si l'on compare maintenant les algorithmes entre eux, on peut voir que DRILS est l'algorithme qui propose les solutions avec le moins de potentiel. Dès le premier croisement, on peut voir que l'amélioration attendue est déjà équivalente aux améliorations les plus faibles obtenues par DRILS+ et ILS pour la dixième itération. Il est intéressant de noter que les dynamiques de recherche d'ILS et de DRILS+ sont différentes, dans le sens que DRILS+ tend à proposer des solutions avec un meilleur potentiel en début de chaîne, soit

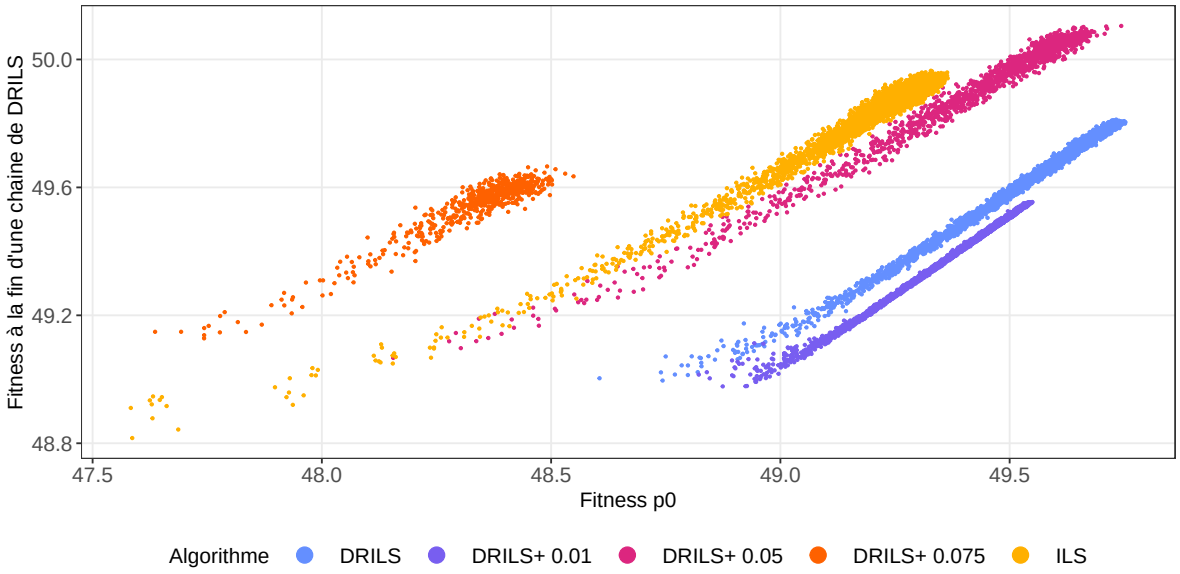

FIGURE 7.3 – *Amélioration attendue d'une chaîne de DRILS avec APX en fonction de la fitness du premier parent et de l'algorithme utilisé pour obtenir la première solution pour commencer la chaîne de DRILS. DRILS+ est donné avec plusieurs intensités de perturbations β*

la première itération ; tandis que l'amélioration est équivalente entre ILS et DRILS+ pour la dixième itération ; avant de finir légèrement en faveur d'ILS pour la centième.

Nous avons donc vu que l'amélioration en pourcentage apportée par une itération de DRILS décroît en fonction du nombre d'itérations successives réalisées. La valeur de départ ainsi que la vitesse de cette baisse dépend de l'algorithme ayant permis d'obtenir la première solution P_0, la valeur de la fitness initiale n'ayant que peu d'impact. Cependant, il est difficile, à partir de cette observation uniquement, de pouvoir conclure quant aux performances des différents algorithmes.

Amélioration empirique d'une chaîne de DRILS

Pour mieux comprendre l'impact de cette dynamique sur la performance des différentes variantes de DRILS, intéressons-nous à la fitness obtenue en fin de chaîne en fonction de l'algorithme et également en fonction de la qualité de la solution de départ P_0. Ceci est étudié à l'aide de la figure 7.3.

On peut notamment y observer le comportement de DRILS+ en fonction de l'intensité du mécanisme d'échappement, c'est-à-dire le paramètre β dont on peut retrouver les différentes valeurs après le nom de l'algorithme. Une fois de plus, on voit que l'amélioration dans une chaîne de DRILS est linéaire en fonction de la qualité de la première solution. Cependant, certains algorithmes offrent des solutions P_0 avec davantage de potentiel, ce qui se traduit sur la figure par des meilleures fitness de fin de chaîne à valeurs initiales de fitness égales.

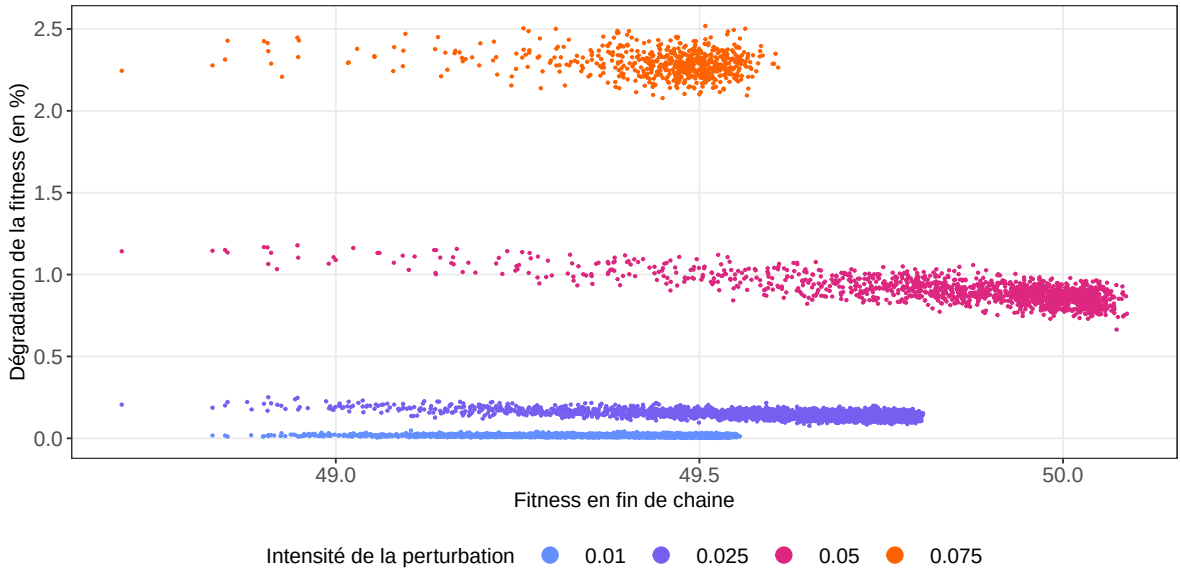

FIGURE 7.4 – *Dégradation de la fitness (en %) de la dernière solution d'une chaîne de DRILS lors de la phase d'échappement en fonction de la fitness et de l'intensité de la perturbation*

Ainsi, l'algorithme offrant les meilleures solutions initiales en termes de potentiel, et donc intrinsèquement les moins biaisées est DRILS+ avec $\beta = 0.075$, suivi par ILS, DRILS+ 0.05, DRILS et pour finir DRILS 0.01. Nous devons cependant noter deux aspects importants : premièrement, le temps de calcul n'est pas pris en compte ; et deuxièmement, les segments pour chaque algorithme s'arrêtent à des valeurs de P_0 différentes. Ainsi, bien que DRILS+ 0.075 soit en principe l'algorithme fournissant les solutions initiales avec le plus fort potentiel d'amélioration, on voit que cet algorithme n'est pas capable, dans le temps imparti, de fournir des solutions initiales de qualités suffisantes pour permettre d'obtenir des solutions finales de meilleures qualités que DRILS, ILS ou DRILS+ 0.05. Il faut donc trouver un compromis entre la qualité de la solution initiale et son potentiel d'amélioration.

Impact de la perturbation sur la dégradation de la fitness

Afin de mieux comprendre l'origine de la différence de potentiel d'amélioration observée entre les solutions initiales aux chaînes de DRILS, intéressons-nous à la perte de qualité induite par la perturbation ; et en particulier à l'impact de l'intensité de cette dernière. Pour ce faire, intéressons-nous dans la figure 7.4 à la dégradation de la fitness (en %) en fonction de la fitness de la dernière solution d'une chaîne de DRILS. Notons que les valeurs sont données pour 4 intensités de perturbations différentes allant de 1% à 7.5%.

On peut voir que l'intensité de la perturbation impacte fortement la dégradation de la solution. Sans surprise, plus l'intensité est élevée plus la solution est dégradée. On remarque également que la dégradation est très peu impactée par la qualité de la solution

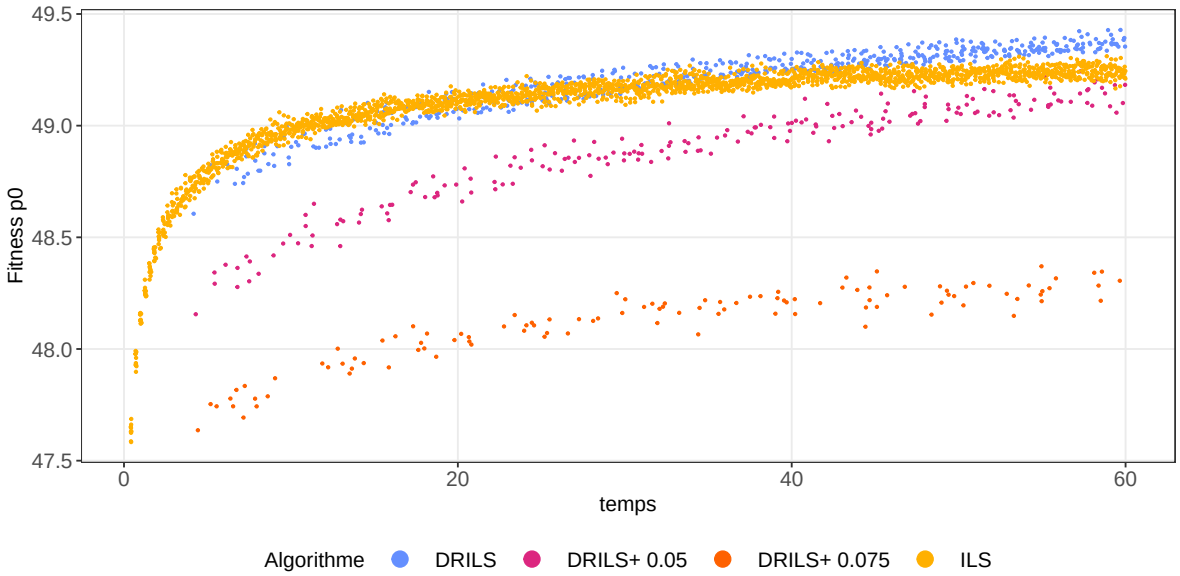

FIGURE 7.5 – *Fitness des solutions p0 obtenues en fonction du temps pour plusieurs algorithmes*

qu'on perturbe. Il est intéressant de noter que plus la solution perturbée est de bonne qualité, plus la dégradation observée est faible. Bien que cet effet soit relativement assez faible, on peut clairement l'observer pour une perturbation de 5% des variables.

En combinant ces observations, on voit que le compromis exploration/exploitation doit se penser également à l'échelle de la chaîne de DRILS. En effet, le potentiel des solutions dépend fortement de l'intensité de la dégradation appliquée à la dernière solution de la chaîne de DRILS précédente. Or, une perturbation plus importante ne va pas seulement augmenter le potentiel d'amélioration d'une solution, mais également réduire sa qualité. Il faut ainsi trouver un juste-milieu entre dégradation liée la perturbation et amélioration induite par la chaîne de DRILS.

Fitness des solutions de départ pour les chaînes de DRILS en fonction du temps

Dans le meilleur des cas, on voudrait des solutions de fort potentiel avec une fitness la plus élevée possible. De surcroît, on aimerait que ces dernières soient obtenues le plus rapidement possible. Comme on l'a vu précédemment, un bon choix des paramètres (en particulier de l'intensité de la perturbation β pour l'algorithme DRILS+) permet de maximiser ces objectifs. C'est notamment la raison pour laquelle l'algorithme PRILS se base sur cette variante de DRILS pour sa seconde phase.

Cependant, pour comprendre l'efficacité de PRILS, il nous faut comprendre pourquoi la première phase d'ILS bénéficie à la recherche globale. Comme nous l'avons vu dans la figure 7.3, ILS offre des solutions avec un très fort potentiel, supérieur même à celui de DRILS+ avec la meilleur valeur de β testée. La faiblesse d'ILS étant qu'il n'est pas capable de fournir des solutions initiales de qualité suffisante pour rivaliser avec DRILS+ en termes

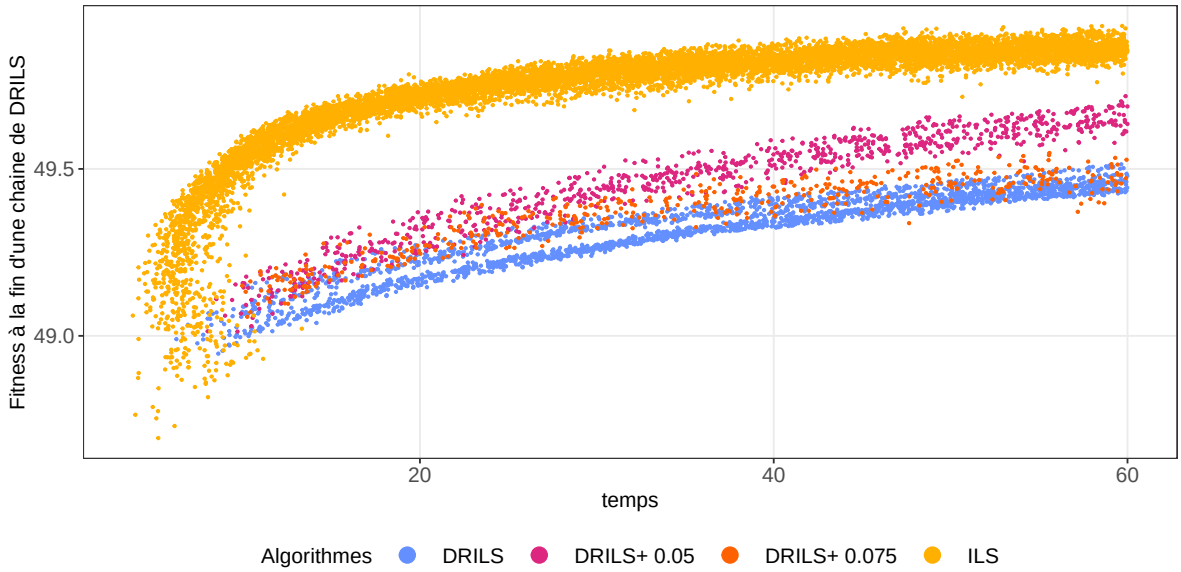

FIGURE 7.6 – *Fitness des solutions obtenues à la fin d'une chaîne de DRILS en fonction du temps et de différents algorithmes utilisés pour obtenir la première solution*

de fitness obtenue à la fin des chaînes. Cependant, rappelons que cette figure ne prenait pas en compte le temps de calcul.

Intéressons-nous maintenant à la figure 7.5 qui cette fois-ci prend en compte le temps de calcul. Sur cette figure, on peut voir la qualité des solutions P_0 trouvées par les différents algorithmes, à savoir : ILS, DRILS+ 0.05, DRILS+ 0.075 et DRILS, et ce, en fonction du temps. On voit ici clairement le double avantage d'ILS sur DRILS+ 0.05. En effet, non seulement ILS offre des solutions de meilleurs potentielles et de meilleurs qualités initiale, mais en plus, il les trouve plus rapidement que DRILS+ 0.05. Notons que DRILS obtient également des solutions de qualité, mais avec un potentiel très faible. À l'opposé, on trouve DRILS+, qui fournit des solutions de très faible qualité, mais avec un fort potentiel. Dans les deux cas, ces algorithmes offrent de faibles performances. On peut pour finir noter que les convergences d'ILS et DRILS+ 0.05 diffèrent. Le premier ayant convergé au bout d'une minute tandis que le second continue lentement de s'améliorer. La rapide convergence d'ILS explique la raison pour laquelle, dans PRILS, la transition de phase s'effectue relativement tôt, mais aussi pourquoi PRILS est également efficace dans le cas d'un temps d'exécution restreint.

Fitness des solutions à la fin des chaînes de DRILS en fonction du temps

Analysons maintenant la fitness des solutions obtenues à la fin d'une chaîne de DRILS au cours du temps, lorsque l'on utilise des solutions obtenues par différents algorithmes. Ceci est illustré dans la figure 7.6. En d'autres termes, si on prend l'exemple de la courbe correspondant à ILS, celle-ci nous montre la fitness finale d'une chaîne de DRILS appli-

quée aux solutions obtenues par ILS, prenant donc en compte le temps pour obtenir la solution ILS plus le temps pour effectuer la chaîne de DRILS. Comme exemple illustratif, considérons une solution X obtenue en 1 seconde par ILS. Supposons maintenant que si on applique une chaîne de DRILS à X, on obtient une solution Y après 3 secondes. Alors, on va représenter sur le graphe la qualité de la solution Y obtenue en 4 secondes (le temps pour obtenir X plus le temps de la chaîne de DRILS). Ceci constitue donc un moyen de représenter à la fois le potentiel des solutions, mais également la vitesse pour les obtenir.

On peut voir que ILS est très clairement meilleur que les trois autres algorithmes et en particulier DRILS+ 0.05. On a ici la preuve empirique du double avantage d'ILS mis en lumière dans le paragraphe précédent, à savoir la combinaison du haut potentiel et de la qualité initiale. On remarque également la convergence d'ILS, ce qui est cohérent avec le fait que la fitness P_0 stagne, comme vue dans la figure précédente, mais également que l'amélioration d'une chaîne de DRILS est linéaire en fonction de la qualité de P_0.

On voit donc que la première chaîne de DRILS effectuée par PRILS, c'est à dire la chaîne de DRILS qui a reçu une solution fournit par ILS après sa phase de recherche, offre une solution de très forte qualité comparée à ce que l'on peut obtenir avec d'autres algorithmes. Cette étape est en quelque sorte un raccourci, permettant très rapidement de lancer DRILS+ 0.05 avec des solutions de fortes qualités. En effet, après quelques itérations de chaîne de DRILS, les solutions P_0 de PRILS vont suivre le même comportement que l'on observe pour DRILS+ 0.05, avec cependant des fitness plus élevées, et donc des qualités de solutions en fin de chaîne également meilleures.

7.3 Conclusion

Dans la première partie de ce chapitre, nous avons dressé un état des lieux des meilleures combinaisons de variantes de DRILS et de croisements boîte grise. À notre connaissance, notre étude est la première de son genre qui permet de donner de façon compréhensive et exhaustive l'état de l'art en optimisation boîte grise pour les problèmes pseudo-booléens k-bornés considérés. Dans la deuxième partie du chapitre, l'introduction de la notion de chaîne de DRILS nous a permis de mieux comprendre la dynamique des différentes variantes de DRILS, et surtout d'expliquer leurs performances. On peut également apprécier l'impact sur la dynamique de recherche des différentes intensités des perturbations utilisées par le mécanisme d'échappement de DRILS+. Pour finir, on mesure tout le potentiel des solutions obtenues par l'usage exclusif d'ILS avec un critère d'acceptation non-élitiste, et surtout la vitesse à laquelle ces solutions peuvent être obtenues. On voit donc à travers cette contribution que les solutions à fitness égale ont des potentiels d'améliorations complètement différents lorsqu'elles servent à initialiser DRILS. Ainsi, la fitness n'est pas un bon moyen de regrouper les solutions si l'on souhaite

étudier les opérateurs de croisement. Il faut davantage s'intéresser à la façon dont celles-ci ont été obtenues, en particulier à l'algorithme (ici ILS, DRILS, DRILS+), mais également, dans le cas de DRILS+, à l'intensité de la perturbation du mécanisme d'échappement. Nous espérons que les notions et les études présentées dans cette dernière partie de ce chapitre permettront à la communauté de mieux comprendre la dynamique des opérateurs de croisement, et que cela mènera à la conception de nouveaux algorithmes encore plus efficaces.

Conclusion

Le paradigme boîte grise a permis à la communauté scientifique de concevoir des algorithmes très efficaces capables de traiter des problèmes de plus en plus grands. Dans ce manuscrit, nous nous sommes concentré sur les problèmes pseudo-booléens k-bornés tels que les paysages NK ou Mk, et en particulier sur des instances de très grandes dimensions allant de quelques milliers à un million de variables. L'état de l'art repose sur une recherche locale très efficace capable de trouver les mouvements améliorants en temps constant, et de sa combinaison avec un opérateur de croisement très efficace, le croisement par partition, au sein d'un algorithme générique s'inspirant d'une recherche locale itérée : l'algorithme DRILS. Ce dernier est intrinsèquement séquentiel, d'une part en raison de sa conception et de la dynamique de recherche souhaitée initialement, et d'autre part du fait des structures de données utilisées. Depuis la proposition de cet algorithme, les dernières avancées se sont concentrées exclusivement sur l'élaboration d'opérateurs de croisements de plus en plus complexes et efficaces, tels que les croisements avec points d'articulation ou utilisant la programmation dynamique sur des arbres de cliques.

Dans ce contexte, nos travaux se sont concentrés sur les deux axes complémentaires suivants. Premièrement, nous nous sommes intéressés à l'élaboration d'algorithmes parallèles. On peut notamment citer nos contributions sur la conception d'une recherche locale boîte grise parallèle, ainsi que le développement d'un framework coopératif asynchrone en environnement massivement parallèle basé sur DRILS, et capable de tirer profit de la puissance de calcul des derniers supercalculateurs les plus puissants, tel que le Fugaku. Deuxièmement, nous nous sommes intéressés à l'étude de la dynamique de DRILS et en particulier des mécanismes d'échappement, afin de contrebalancer la forte intensification induite par l'usage d'opérateur de croisements boîtes grises. Pour finir, nous avons conduit des études empiriques approfondies analysant la synergie entre les différentes variantes de DRILS et les derniers opérateurs de croisements boites grises, permettant ainsi de mieux appréhender la dynamique de recherche. Dans la suite, nous résumons nos différentes contributions et nous discutons de quelques perspectives de recherche.

Résumé de contributions

Hill Climber parallèle basée sur la coloration de graphe. Dans cette première contribution, présentée dans le chapitre 3, nous avons introduit un algorithme de recherche locale de type Hill Climber parallèle et fonctionnant en mémoire partagée. Cette parallélisation est rendue possible par l'utilisation d'un algorithme de coloration de graphe, afin de créer des ensembles de variables indépendantes. Ces variables indépendantes pouvant ensuite être modifiées en parallèle sans risque d'accès concurrents. En plus de cette coloration de graphe, nous avons proposé une stratégie permettant de réduire au maximum le nombre de calcul à effectuer pour tester si une variable est améliorante ou non. Celle-ci s'appuie sur les informations disponibles dans un contexte boîte grise, et plus particulièrement sur le graphe des interactions entre variables. L'idée est de considérer que si le score lié au changement d'une variable a été calculé et qu'il n'est pas améliorant, alors il n'est pas nécessaire de le recalculer à moins qu'une des variables interagissant avec cette même variable n'ait été changée dans la solution courante. Cet algorithme de recherche locale a ensuite été expérimenté sur un grand nombre d'instances de paysages NK et Mk de tailles et de rugosités différentes. Nos résultats ont montré que cette approche est compétitive en ce qui concerne la qualité des solutions trouvées. En termes de temps de calcul, notre approche est beaucoup plus rapide lorsqu'on s'attaque à des problèmes très grands, et offre dans ces cas-là une scalabilité substantielle jusqu'à une quarantaine de cœurs.

Approche coopérative massivement parallèle. Dans cette seconde contribution, détaillée dans le chapitre 4, nous nous sommes intéressés plus spécifiquement à des environnements de calcul massivement parallèle. Nous avons introduit un framework reposant sur DRILS et fonctionnant sur des supercalculateurs modernes, tels que le Fugaku. Ce travail s'inspire des modèles en îles afin d'effectuer des recherches locales itérées séquentielles sur plusieurs centaines de processeurs et de les faire communiquer entre elles. S'exécutant intégralement de façon asynchrone, cette approche propose plusieurs niveaux de parallélisme reposant à la fois sur des technologies d'échanges de messages et de mémoire partagée. Dans un premier temps, nous avons proposé des politiques avancées de communication permettant de réduire le volume des messages échangés, mais également d'introduire une nouvelle politique pour le partage de la meilleure solution trouvée. Ceci est montré important du point de vue du compromis exploration/exploitation. Dans un second temps, nous avons proposé de regrouper les threads en sous-groupes au sein de chaque processeur, et de faire exécuter les itérations de DRILS par ces threads de façon parallèle et coopérative. Plus précisément, chaque sous-groupe de threads dispose d'une solution courante, représentant le premier parent utilisée pour le croisement. Cette solution est partagée par tous les threads de ce sous-groupe, et son évolution au cours des itération parallèles de DRILS est donc gérée de façon concurrente au sein d'un même sous-groupe. En plus d'accélérer

sensiblement la vitesse d'exécution des itérations de DRILS, cette variante coopérative permet d'induire une dynamique de recherche différente, qui nous permet d'obtenir des solutions d'une qualité hors de portée des autres versions de DRILS. Nous avons ensuite démontré empiriquement que notre algorithme distribué, asynchrone et massivement parallèle, était capable, pour les problèmes les plus larges, de passer à l'échelle jusqu'à 12 288 cœurs, et ce, sans signe de stagnation.

Nouveaux mécanismes d'échappement pour DRILS. La troisième contribution, décrite dans le chapitre 5, propose de concevoir des nouveaux mécanismes d'échappement pour l'algorithme DRILS. Dans ce chapitre, nous avons mené une étude détaillée de l'importance combinée du critère d'acceptation, de la perturbation et de l'opérateur de croisement. Basées sur nos observations empiriques, nous avons ainsi pu proposer deux variantes d'algorithmes, appelées respectivement DRILS+ et DRILS++, améliorant la conception initiale de DRILS. La première variante, rétroactive, consiste à introduire un nouveau paramètre de perturbation permettant de découpler la perturbation utilisée pour générer le second parent nécessaire au croisement, de la perturbation utilisée en tant que mécanisme d'échappement. En effet, dans la version initiale de DRILS, la perturbation jouait de façon implicite ces deux rôles simultanément, ce qui s'est avéré être néfaste à la recherche. La deuxième variante est pro-active, c'est-à-dire que nous tentons d'éviter d'être bloqué de façon préventive en cours de recherche. L'idée est d'effectuer avec une certaine probabilité, soit une itération classique de DRILS, soit une itération d'un ILS classique sans utiliser de croisement. Nos nombreuses expérimentations montrent que ces deux stratégies améliorent significativement DRILS dans la majorité des instances de paysages NKQ considérées, avec un avantage pour DRILS+ sur l'ensemble.

Phase d'initialisation basée sur une approche explorative pour l'algorithme DRILS. Cette contribution, décrite dans le chapitre 6, concerne l'ajout d'une phase d'initialisation basée sur une approche explorative pour l'algorithme DRILS. En effet, à l'aide d'une série d'expériences, nous montrons d'abord l'importance de la première solution utilisée par DRILS sur la dynamique de recherche. Cette analyse s'avère extrêmement utile pour la conception de nouveaux algorithmes améliorés. On y voit notamment qu'une ILS simple, sans croisement et avec un critère d'acceptation non-élitiste, permet d'obtenir des solutions avec un très fort potentiel d'amélioration. En se basant sur ces observations, nous proposons un algorithme en deux phases que nous avons appelé PRILS, acronyme de l'anglais 'Pipelined Recombination and Iterated Local Search'. Cet algorithme s'appuie sur une conception simple qui divise le temps d'exécution disponible entre les 2 phases suivantes. On exécute d'abord l'algorithme ILS de base jusqu'à convergence ; puis on utilise la dernière solution trouvée pour lancer une exécution de DRILS+ pendant le temps restant. La phase initiale permet en quelque sorte d'explorer l'espace de recherche sans le biais

introduit par les croisements boîtes grises, puis le croisement est utilisé pour intensifier la recherche autour des zones prometteuses ainsi atteintes. Ce nouvel algorithme est expérimenté sur un grand nombre d'instances, obtenant des résultats supérieurs dans la majorité des cas ; en particulier pour les instances les plus difficiles. Nous nous sommes également intéressés au profil de convergence de PRILS, où l'on a clairement montré la transition entre les deux phases de l'algorithme.

Études des comportements des différents croisements boîtes grises sur DRILS et ses variantes. La dernière contribution, discutée dans le chapitre 7 de ce manuscrit, conclut nos différents développements en étudiant de façon systématique et exhaustive la synergie entre les différentes variantes de DRILS et les différents croisements boîte grise de la littérature, tels que le croisement par partition, celui avec analyse des points d'articulation, et celui avec programmation dynamique appliquée aux arbres de cliques. Nous avons ainsi pu démontrer de façon empirique que la variante de DRILS proposée dans le chapitre 6, à savoir PRILS, est bien meilleur dans l'ensemble que les autres versions de DRILS, tandis que le meilleur opérateur de croisement dépend fortement de la taille des instances considérées. Ce chapitre introduit ensuite de nouveaux concepts tels que la notion de chaîne de DRILS, nous permettant de mieux comprendre/expliquer la dynamique de DRILS ainsi que les écarts de performances entre ses différentes versions. Ce travail reflète à notre connaissance, et au moment de l'écriture de ce document, l'état de l'art actuel de l'optimisation boite grise pour les problèmes pseudo-booléens considérés, à savoir les paysages NK et MK.

Perspectives

Modèle en îles hétérogènes. Parmi nos contributions, l'une d'entre elles, présentée au chapitre 4, proposait une approche massivement parallèle et distribuée sous forme d'île. Dans sa version basique ou bien coopérative, tous les threads effectuaient en parallèle des itérations de DRILS en utilisant les mêmes paramètres et composantes algorithmiques. Bien que cette approche soit très efficace, et comme nous l'avons démontrée par la suite, il est possible d'améliorer l'algorithme DRILS séquentiel, en introduisant de nouveaux mécanismes d'échappements (chapitre 5), ou bien en explorant mieux l'espace de recherche par une phase initiale particulière (chapitre 6). En outre, comme étudiée dans le chapitre 7, mais également en se basant sur la littérature, les variantes du croisement par partition, comme le croisement avec analyse des points d'articulation [13] ou le croisement avec programmation dynamique sur les arbres de cliques [14], permettent dans certains cas d'obtenir de meilleurs résultats. On voit notamment que le croisement avec programmation dynamique obtient de très bons résultats, mais demande davantage de temps. Ainsi,

un axe de recherche prometteur est d'étudier la possibilité d'attribuer des rôles différents aux processeurs parallèles. Ceci consiste donc à considérer un modèle en île, non pas homogène, mais hétérogène. L'idée serait par exemple que certaines îles (ou threads pour une granularité à concevoir) se chargeraient de visiter l'espace de recherche à l'aide d'ILS ; alors que d'autres se chargeraient d'itérer DRILS, ou l'une de ses variantes, en parallèle sur les solutions ainsi trouvées. On peut aussi envisager d'autre type d'hétérogénéité entre îles en considérant d'intensifier plus ou moins fortement la recherche sur les meilleures solutions en poussant davantage la profondeur du croisement avec programmation dynamique par exemple, etc. Ces différents rôles algorithmiques étant attribués à différentes îles, ou à un ou plusieurs processeurs/threads, avec des règles de communications avancées. Cela permettraient notamment de concevoir des algorithmes capables de fournir les meilleures performances, en termes de qualité de la solution, *à tout moment* de l'exécution parallèle. Bien que cette idée soit très prometteuse conceptuellement parlant, car se basant sur des observations solides, il faudra aussi faire particulièrement attention à la façon avec laquelle un tel modèle en îles hétérogènes est déployé sur une vraie architecture parallèle, avec tout ce que cela représente comme défis en termes de paradigmes et de technologies de programmation parallèle et haute performance ; ceci afin d'obtenir des accélérations les plus intéressantes que possibles.

DRILS adaptatif. Continuons notre raisonnement dans le même esprit que la perspective discutée dans le paragraphe précédent, à savoir d'utiliser différentes variantes de DRILS et de croisements en parallèle, on peut aussi se pencher sur l'exploration d'une telle idée dans un contexte de calcul séquentiel. En effet, on a déjà vu dans le chapitre 6 que l'utilisation de deux phases, ILS suivi de DRILS+, permettait à PRILS d'améliorer significativement l'état de l'art. On peut ainsi légitimement considérer la perspective d'utiliser différentes variantes de DRILS ou de croisements au cours de la recherche, au sein d'un algorithme adaptatif. Un tel algorithme serait donc capable d'adapter la recherche, et ses composants et/ou paramètres, afin d'obtenir de meilleures performances. Pour ce faire, une meilleure compréhension des dynamiques de recherche des différentes variantes de DRILS est nécessaire. Dans ce sens, le chapitre 7 et les expériences qui y sont développées, constituent une première pierre à l'édifice. L'élaboration de critères et/ou de métriques capable de dicter les changements de comportements au cours de la recherche est également une piste à creuser davantage. À titre d'exemple, comme on l'a vu dans le chapitre 6 dans le cadre de notre algorithme PRILS, le critère de convergence permettant de passer de ILS à DRILS+ se révèle très efficace malgré sa simplicité. Une meilleure conception, plus fine, permettrait certainement d'améliorer davantage les performances de notre algorithme.

Généralisation à d'autres problèmes d'optimisation Les contributions présentées dans ce manuscrit ont été expérimentées pour des paysages Mk et NK. Cependant, nos contributions sous formes d'algorithmes pourraient être développées davantage dans le contexte d'autres problèmes. Par exemple, si l'on considère des problèmes MAX k-SAT, pouvant en théorie être modélisés par des paysages Mk, l'algorithme DRILS n'a pas été comparé de façon exhaustive à l'état de l'art pour ce problème. L'opérateur de croisement par partitions fut néanmoins utilisé dans la littérature pour améliorer des solveurs pour MAX k-SAT [9]. Nous pensons donc que les variantes de DRILS proposés dans ce manuscrit combinées aux derniers opérateurs de croisement boîte grise peuvent être une base solide pour concevoir de nouveaux algorithmes capable de rivaliser avec l'état de l'art pour d'autres types de problèmes ou d'instances. Il serait également intéressant de considérer des problèmes Mk multi-objectifs, pour lesquels une recherche locale boîte grise existe [11]. Cependant, il n'existe ni opérateurs de croisement, ni métaheuristiques telles que DRILS, spécifiquement adaptés aux problèmes multi-objectifs. Ainsi, de nombreuses opportunités de recherche existent pour le développement de tels algorithmes. Pour finir, on peut également considérer des problèmes à variables continues pour lesquelles un opérateur de croisement boîte grise fût développé récemment [37], ouvrant ainsi des perspectives intéressantes pour le développement de nouvelles méthodes boîte grise, dans un contexte de calcul séquentiel et parallèle.

Bibliographie

[1] T. Back. Selective pressure in evolutionary algorithms : a characterization of selection mechanisms. In *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*, pages 57–62 vol.1, 1994. doi : 10.1109/ICEC.1994.350042. 34

[2] Matthieu Basseur and Adrien Goeffon. Hill-climbing strategies on various landscapes : An empirical comparison. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, GECCO '13, page 479–486, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450319638. doi : 10.1145/2463372.2463439. URL https://doi.org/10.1145/2463372.2463439. 11

[3] Matthieu Basseur and Adrien Goëffon. On the efficiency of worst improvement for climbing nk-landscapes. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, GECCO '14, page 413–420, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450326629. doi : 10.1145/2576768.2598268. URL https://doi.org/10.1145/2576768.2598268. 11

[4] Matthieu Basseur and Adrien Goëffon. Climbing combinatorial fitness landscapes. *Applied Soft Computing*, 30 :688–704, 2015. ISSN 1568-4946. doi : https://doi.org/10.1016/j.asoc.2015.01.047. URL https://www.sciencedirect.com/science/article/pii/S156849461500068X. 11

[5] Hans L. Bodlaender. Discovering treewidth. In Peter Vojtáš, Mária Bieliková, Bernadette Charron-Bost, and Ondrej Sýkora, editors, *SOFSEM 2005 : Theory and Practice of Computer Science*, pages 1–16, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-30577-4. 31

[6] Erick Cantú-Paz. A survey of parallel genetic algorithms. 2000. URL https://api.semanticscholar.org/CorpusID:14264381. 68

[7] V. Černý. Thermodynamical approach to the traveling salesman problem : An efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45

(1) :41–51, Jan 1985. ISSN 1573-2878. doi : 10.1007/BF00940812. URL https://doi.org/10.1007/BF00940812. 11

[8] Wenxiang Chen, Darrell Whitley, Doug Hains, and Adele Howe. Second order partial derivatives for nk-landscapes. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, GECCO '13, page 503–510, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450319638. doi : 10.1145/2463372.2463437. URL https://doi.org/10.1145/2463372.2463437. 17

[9] Wenxiang Chen, Darrell Whitley, Renato Tinós, and Francisco Chicano. Tunneling between plateaus : improving on a state-of-the-art maxsat solver using partition crossover. pages 921–928, 07 2018. ISBN 9781450356183. doi : 10.1145/3205455.3205482. 148

[10] Francisco Chicano, Darrell Whitley, and Andrew M. Sutton. Efficient identification of improving moves in a ball for pseudo-boolean problems. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, GECCO '14, page 437–444, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450326629. doi : 10.1145/2576768.2598304. URL https://doi.org/10.1145/2576768.2598304. 17, 20, 21

[11] Francisco Chicano, Darrell Whitley, and Renato Tinos. Efficient hill climber for constrained pseudo-boolean optimization problems. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, GECCO '16, page 309–316, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342063. doi : 10.1145/2908812.2908869. URL https://doi.org/10.1145/2908812.2908869. 148

[12] Francisco Chicano, Darrell Whitley, Gabriela Ochoa, and Renato Tinós. Optimizing one million variable nk landscapes by hybridizing deterministic recombination and local search. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '17, page 753–760, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349208. doi : 10.1145/3071178.3071285. URL https://doi.org/10.1145/3071178.3071285. 35, 61, 95

[13] Francisco Chicano, Gabriela Ochoa, Darrell Whitley, and Renato Tinós. Enhancing partition crossover with articulation points analysis. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '18, page 269–276, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356183. doi : 10.1145/3205455.3205561. URL https://doi.org/10.1145/3205455.3205561. 27, 28, 95, 146

[14] Francisco Chicano, Gabriela Ochoa, L. Darrell Whitley, and Renato Tinós. Dynastic Potential Crossover Operator. *Evolutionary Computation*, 30(3) :409–446, 09 2022. ISSN 1063-6560. doi : 10.1162/evco_a_00305. URL https://doi.org/10.1162/evco_a_00305. 31, 32, 95, 146

[15] Philippe Codognet, Danny Munera, Daniel Diaz, and Salvador Abreu. *Parallel Local Search*, pages 381–417. Springer International Publishing, Cham, 2018. ISBN 978-3-319-63516-3. doi : 10.1007/978-3-319-63516-3_10. URL https://doi.org/10.1007/978-3-319-63516-3_10. 65

[16] G. A. Croes. A method for solving traveling-salesman problems. *Operations Research*, 6(6) :791–812, 1958. ISSN 0030364X, 15265463. URL http://www.jstor.org/stable/167074. 11

[17] David E. Goldberg and Kalyanmoy Deb. A comparative analysis of selection schemes used in genetic algorithms. volume 1 of *Foundations of Genetic Algorithms*, pages 69–93. Elsevier, 1991. doi : https://doi.org/10.1016/B978-0-08-050684-5.50008-2. URL https://www.sciencedirect.com/science/article/pii/B9780080506845500082. 34

[18] Tomohiro Harada and Enrique Alba. Parallel genetic algorithms : A useful survey. *ACM Comput. Surv.*, 53(4), aug 2020. ISSN 0360-0300. doi : 10.1145/3400031. URL https://doi.org/10.1145/3400031. 65

[19] John H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI, 1975. second edition, 1992. 34

[20] Stuart Kauffman and Simon Levin. Towards a general theory of adaptive walks on rugged landscapes. *Journal of Theoretical Biology*, 128(1) :11–45, 1987. ISSN 0022-5193. doi : https://doi.org/10.1016/S0022-5193(87)80029-2. URL https://www.sciencedirect.com/science/article/pii/S0022519387800292. 8

[21] Stuart A. Kauffman and Edward D. Weinberger. The nk model of rugged fitness landscapes and its application to maturation of the immune response. *Journal of Theoretical Biology*, 141(2) :211–245, 1989. ISSN 0022-5193. doi : https://doi.org/10.1016/S0022-5193(89)80019-0. URL https://www.sciencedirect.com/science/article/pii/S0022519389800190. 8

[22] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598) :671–680, 1983. doi : 10.1126/science.220.4598.671. URL https://www.science.org/doi/abs/10.1126/science.220.4598.671. 11

[23] A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3) :497–520, 1960. ISSN 00129682, 14680262. URL http://www.jstor.org/stable/1910129. 10

[24] John D. C. Little, Katta G. Murty, Dura W. Sweeney, and Caroline Karel. An algorithm for the traveling salesman problem. *Operations Research*, 11(6) :972–989, 1963. ISSN 0030364X, 15265463. URL http://www.jstor.org/stable/167836. 10

[25] Helena R. Lourenço, Olivier C. Martin, and Thomas Stützle. *Iterated Local Search*, pages 320–353. Springer US, Boston, MA, 2003. ISBN 978-0-306-48056-0. doi : 10.1007/0-306-48056-5_11. URL https://doi.org/10.1007/0-306-48056-5_11. 12

[26] Helena R. Lourenço, Olivier C. Martin, and Thomas Stützle. *Iterated Local Search : Framework and Applications*, pages 363–397. Springer US, Boston, MA, 2010. ISBN 978-1-4419-1665-5. doi : 10.1007/978-1-4419-1665-5_12. URL https://doi.org/10.1007/978-1-4419-1665-5_12. 12

[27] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari, and Thomas Stützle. The irace package : Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3 :43–58, 2016. ISSN 2214-7160. doi : https://doi.org/10.1016/j.orp.2016.09.002. URL https://www.sciencedirect.com/science/article/pii/S2214716015300270. 127

[28] N. Mladenović and P. Hansen. Variable neighborhood search. *Computers and Operations Research*, 24(11) :1097–1100, 1997. ISSN 0305-0548. doi : https://doi.org/10.1016/S0305-0548(97)00031-2. URL https://www.sciencedirect.com/science/article/pii/S0305054897000312. 11

[29] Ryohei Okazaki, Takekazu Tabata, Sota Sakashita, Kenichi Kitamura, Noriko Takagi, Hideki Sakata, Takeshi Ishibashi, Takeo Nakamura, and Yuichiro Ajima. Supercomputer fugaku CPU A64FX realizing high performance, high-density packaging, and low power consumption. Technical Report Fujitsu technical review, 2020. 72

[30] Martin Pelikan. Analysis of estimation of distribution algorithms and genetic algorithms on nk landscapes. In *Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation*, GECCO '08, page 1033–1040, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605581309. doi : 10.1145/1389095.1389287. URL https://doi.org/10.1145/1389095.1389287. 10

[31] Mitsuhisa Sato, Yutaka Ishikawa, Hirofumi Tomita, Yuetsu Kodama, Tetsuya Odajima, Miwako Tsuji, Hisashi Yashiro, Masaki Aoki, Naoyuki Shida, Ikuo Miyoshi, Kouichi

Hirai, Atsushi Furuya, Akira Asato, Kuniki Morita, and Toshiyuki Shimizu. Co-design for A64FX manycore processor and Fugaku. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2020. URL https://doi.org/10.1109/SC41405.2020.00051. 72

[32] Gilbert Sywerda. Uniform crossover in genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, page 2–9, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc. ISBN 1558600063. 34

[33] Sara Tari, Matthieu Basseur, and Adrien Goëffon. Worst improvement based iterated local search. In Arnaud Liefooghe and Manuel López-Ibáñez, editors, *Evolutionary Computation in Combinatorial Optimization*, pages 50–66, Cham, 2018. Springer International Publishing. ISBN 978-3-319-77449-7. 11

[34] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2) :146–160, 1972. doi : 10.1137/0201010. URL https://doi.org/10.1137/0201010. 28

[35] Robert E. Tarjan and Mihalis Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on Computing*, 13(3) :566–579, 1984. doi : 10.1137/0213035. URL https://doi.org/10.1137/0213035. 31

[36] Renato Tinós, Darrell Whitley, and Francisco Chicano. Partition crossover for pseudo-boolean optimization. In *Proceedings of the 2015 ACM Conference on Foundations of Genetic Algorithms XIII*, FOGA '15, page 137–149, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450334341. doi : 10.1145/2725494.2725497. URL https://doi.org/10.1145/2725494.2725497. 24, 25, 26, 27, 34

[37] Renato Tinós, Darrell Whitley, Francisco Chicano, and Gabriela Ochoa. Partition crossover for continuous optimization : Epx. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '21, page 627–635, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383509. doi : 10.1145/3449639.3459296. URL https://doi.org/10.1145/3449639.3459296. 148

[38] Darrell Whitley and Wenxiang Chen. Constant time steepest descent local search with lookahead for nk-landscapes and max-ksat. *GECCO'12 - Proceedings of the 14th International Conference on Genetic and Evolutionary Computation*, pages 1357–1364, 07 2012. doi : 10.1145/2330163.2330351. 17

[39] Darrell Whitley, Doug Hains, and Adele Howe. Tunneling between optima : Partition crossover for the traveling salesman problem. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, page 915–922, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605583259. doi : 10.1145/1569901.1570026. URL https://doi.org/10.1145/1569901.1570026. 23

[40] L Darrell Whitley, Francisco Chicano, and Brian W Goldman. Gray box optimization for mk landscapes (NK landscapes and MAX-kSAT). *Evol Comput*, 24(3) :491–519, April 2016. 9, 15

[41] Alden Wright, R.K. Thompson, and Jian Zhang. The computational complexity of n-k fitness functions. *Evolutionary Computation, IEEE Transactions on*, 4 :373–379, 11 2000. doi : 10.1109/4235.887236. 9