UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Mathématiques – Informatique**

Arrêté ministérial : 7 août 2006

Présentée par

Valentin Weber

Thèse dirigée par **Nadia Brauner**
et codirigée par **Yann Kieffer**

préparée au sein du **Laboratoire G-SCOP**
et de l'**École Doctorale MSTII**

Caractérisation des instances difficiles de problèmes d'optimisation *NP*-difficiles

Thèse soutenue publiquement le **8 juillet 2013**,
devant le jury composé de :

Mme. Clarisse Dhaenens
Professeur des Universités, Université Lille 1, Présidente et Rapporteur
M. Christian Artigues
Directeur de Recherche, LAAS-CNRS, Rapporteur
M. David Coudert
Chargé de Recherche, INRIA Sophia Antipolis, Examinateur
M. Mathieu Liedloff
Maître de Conférences, Université d'Orléans, Examinateur
Mme. Nadia Brauner
Professeur des Universités, Université Grenoble 1, Directeur de Thèse
M. Yann Kieffer
Maître de Conférences, Grenoble INP, Co-Directeur de Thèse

À mon amour

Remerciements

Je remercie en premier lieu mes directeurs de thèse, Nadia Brauner et Yann Kieffer, qui ont toujours su être disponibles et à mon écoute. Il s'est crée une franche complicité qui nous a permis de travailler efficacement et d'aboutir aux résultats présentés dans ce manuscrit. Merci à Nadia de m'avoir encouragé à faire une thèse et merci à Yann de m'avoir proposé un sujet aussi intéressant.

J'adresse de sincères remerciements aux membres de mon jury de thèse, Clarisse Dhaenens, présidente et rapporteur, Christian Artigues, rapporteur, David Coudert, examinateur, et Mathieu Liedloff, examinateur, pour leur participation à ma soutenance de thèse et leurs retours sur mes travaux.

Je remercie chaleureusement tous les membres du laboratoire G-SCOP, en particulier son directeur, Yannick Frein, et le personnel administratif, pour leur accueil. Sans chercher à faire une liste exhaustive, je remercie l'ensemble des personnes passées et présentes qui ont participé à la vie du laboratoire pendant ma thèse et ont permis d'en faire un lieu agréable à vivre : les membres de l'A-DOC, les participants aux midi-jeux et aux soirées jeux, les joueurs de backgammon, les participants aux 24h de l'Innovation, l'équipe du Challenge, mes collègues de bureau. Un grand merci à ceux qui ont bien voulu m'héberger lors de mes derniers passages à Grenoble.

J'en profite aussi pour remercier les enseignants en informatique de Phelma avec lesquels j'ai travaillé. Merci à Michel Desvignes d'avoir été mon tuteur.

J'adresse un grand merci à toute l'équipe Mascotte/Coati d'Inria Sophia Antipolis, qui m'a accueilli pendant plus d'une année et m'a permis de m'y intégrer comme un membre à part entière. Je remercie tout particulièrement Frédéric Havet, qui est à l'origine de cette fructueuse collaboration.

Enfin je remercie ma famille et ma belle-famille pour leur soutien et leurs encouragements. Une pensée particulière pour mes grands-parents. Un immense merci à ma femme, qui était toujours à mes côtés pour me redonner le sourire et qui m'a aidé à conclure en beauté cette thèse.

Résumé

L'étude expérimentale d'algorithmes est un sujet crucial dans la conception de nouveaux algorithmes puisqu'elle est inévitablement influencée par le contexte d'évaluation. Le sujet particulier qui nous intéresse dans cette problématique est la pertinence des instances choisies pour servir de base de test à l'expérimentation. Nous formalisons cette notion par une définition de la *difficulté d'instance* qui dépend des performances de méthodes de résolution. Nous présentons une synthèse des différentes approches abordant la notion dans la littérature. Il en ressort un modèle pour classifier les utilisations de la difficulté d'instance.

Le cœur de la thèse porte sur un outil pour évaluer empiriquement la difficulté d'instance. L'approche proposée repose sur une méthode de *benchmarking d'instances*. Contrairement au concept habituel de benchmarking, qui porte sur les algorithmes, nous comparons le comportement de classes d'instances par rapport à un jeu de test composé d'algorithmes. La définition d'un contexte d'étude nous permet d'appliquer un cadre expérimental rigoureux pour appuyer la validité des résultats obtenus. Nous décrivons étape par étape le déroulement de la méthode expérimentale proposée. La comparaison de classes d'instances permet ensuite, par exemple, de valider leur utilisation pour l'expérimentation algorithmique, ou d'analyser des sources de difficulté du problème. Nous illustrons la méthode à travers l'étude de plusieurs classes d'instances pour le problème du voyageur de commerce.

Nous explorons également la génération d'instances, avec la notion de *modification d'instances*. Les opérations que nous considérons, modifient les instances, mais permettent de retrouver facilement une solution optimale, d'une instance à l'autre. Nous étudions une modification pour le problème du voyageur de commerce qui modifie les distances entre les sommets. L'opération permet de modifier artificiellement les performances expérimentales de type ratio à l'optimum, pour les méthodes approchées. Grâce à cette observation, nous déduisons une méthode de génération d'instances difficiles, qui repose sur la solution duale du problème de couplage fractionnaire parfait de poids minimum. Nous donnons une caractérisation de ces solutions dans le cas général et dans le cas où l'instance est métrique après modification. L'impact de la modification est illustré par une étude expérimentale des performances d'algorithmes approchés sur des instances classiques de la littérature avant et après modification.

La modification d'instances est liée à la notion de nucléarisation propre aux problèmes FPT. Nous présentons une définition pratique de la *réduction d'instances* dans le but de mener des études expérimentales sur la nucléarisation. D'une part, nous étudions des opérations de pré-traitement pour le problème d'indépendant maximum et leur impact lorsqu'on les applique au problème du transversal minimum. D'autre part, nous prouvons que le problème du nombre enveloppe appartient à la classe FPT pour le paramètre de diversité de voisinage. Nous montrons que le noyau est linéaire grâce à des opérations de réduction pour des configurations de sommets partageant le même voisinage.

Table des matières

Introduction

Contexte

La recherche opérationnelle

La recherche opérationnelle (RO) est un domaine relativement récent, datant d'après-guerre, qui a vu son éclosion suivre le développement de l'informatique. Aussi appelée aide à la décision, la RO permet de résoudre des problèmes aux applications variées, en particulier dans les domaines industriels. Selon Jean-Paul Hamon [1], vice-président exécutif R&D d'Amadeus, "l'utilité et les champs d'exploitation [de la RO] n'ont fait que croître au fil des années". Il a été rédigé un *Livre Blanc de la Recherche Opérationnelle* [1] qui contient plusieurs exemples d'applications au sein de sociétés françaises. Les principaux domaines d'application sont les transports, la logistique, la planification et l'informatique. La RO est présente à tous les niveaux de décision du stratégique à l'opérationnel. Par exemple, le *Challenge ROADEF/EURO 2010* [2], proposé par EDF, traite du problème de la gestion de la production nationale de l'énergie électrique avec prise en compte de la planification des arrêts pour maintenance des centrales nucléaires, sur un horizon de plusieurs années. Pour résoudre les problèmes réels, la RO permet une modélisation des problèmes sous forme d'énoncés mathématiques. À partir de ces modèles, elle propose différentes approches de résolution selon la nature du problème.

Les méthodes de résolution

La théorie de la complexité fournit une classification des problèmes selon leur difficulté. Les deux principales familles de problèmes sont la classe P des problèmes pour lesquels il existe une méthode de résolution rapide, et la classe des problèmes NP-complets, dont on ne connaît pas de méthode de résolution rapide. La distinction entre les deux classes n'est aujourd'hui pas prouvée et se ramène au problème ouvert

1. `http://www.roadef.org/content/roadef/Libro.htm`
2. `http://challenge.roadef.org/2010/fr/` Participation avec J. Darlay, L. Esperet, Y. Kieffer et G. Naves (annexe B).

$P = NP$. Ce problème appartient à la liste des problèmes du millénaire promue par le *Clay Mathematical Institute*[1], qui offre un million de dollar pour leur résolution. L'opinion la plus courante chez les chercheurs en RO est que $P \neq NP$. Sous cette condition, on peut penser qu'il est peine perdue de vouloir résoudre des problèmes *NP*-difficiles de manière efficace. Bien qu'il n'existe pas de méthodes rapides pour les résoudre de manière exacte, il existe cependant des méthodes alternatives pour traiter ces problèmes. D'une part, il y a les méthodes de résolution exacte, mais dont la durée d'exécution peut être exponentielle, comme les méthodes de type *branch-and-bound*. D'autre part, il existe des méthodes plus rapides, mais dont le résultat n'a pas la garantie d'être optimum, voire peut être parfois arbitrairement mauvais. La conception de méthodes de résolution est généralement un compromis entre le temps de calcul et la qualité des solutions. Se pose alors le problème de comparer les méthodes de résolution, selon leur capacité à résoudre le problème.

Nous différencions deux types d'approches selon leur aspect théorique ou empirique. Les études théoriques permettent de calculer les bornes sur les performances des méthodes de résolution. Elles peuvent porter sur la qualité des solutions avec les algorithmes d'approximation à facteur constant, ou sur le temps d'exécution avec la complexité des algorithmes. Les études empiriques permettent d'évaluer le comportement pratique des méthodes de résolution. Toutefois, en RO, le cadre d'application d'approches expérimentales n'est pas toujours aussi formel que les approches théoriques.

La science de l'expérimentation

L'expérimentation est au cœur de la démarche scientifique dans de nombreux domaines. Elle permet souvent de percevoir la réalité, en observant par exemple les phénomènes naturels. Les données ainsi relevées servent de base pour concevoir des modèles et énoncer des concepts scientifiques. L'expérience scientifique repose également sur l'expérimentation, qui consiste à valider l'hypothèse par des expériences répétées selon un protocole expérimental. Les disciplines où elle est couramment appliquée sont la physique, la chimie, la médecine et la biologie.

En algorithmique pour la RO, le rôle de l'expérimentation est souvent assez marginal et la porté des résultats peut être perçue assez limitée. En effet, dans les sciences naturelles, les systèmes étudiés sont réels et leurs mécanismes ne sont pas pleinement définis. Or, dans notre cas, les systèmes étudiés sont complètement connus. Cela ne semble donc pas justifier l'utilisation d'études expérimentales perçues comme approximatives, alors qu'une étude explicite et exacte pourrait être menée. Toutefois, les systèmes étudiés deviennent de plus en plus complexes et, bien qu'on ait une définition exacte de tous les éléments, il peut être très compliqué d'étudier leur comportement précis lorsqu'ils interagissent entre eux. Ainsi, il

1. `http://www.claymath.org/`

existe de nombreux algorithmes dont on ne connaît pas la complexité exacte. On peut seulement en donner une borne supérieure, mais sans exemple qui l'atteigne effectivement.

Des études expérimentales sont donc de plus en plus présentes dans les publications, pour présenter le comportement pratique des algorithmes. Toutefois, ces études ont souvent plus la forme d'observations pour donner des tendances que des études qui suivent un protocole expérimental dont on pourrait déduire des résultats fiables. Plusieurs chercheurs militent pour une approche plus rigoureuse de l'expérimentation d'algorithmes. Hooker (1994) propose de développer une science empirique des algorithmes pour formaliser cette approche. Ces travaux ont été suivis par plusieurs papiers, sous forme de guides décrivant les "bonnes" méthodes pour mener des études expérimentales, que ce soit au niveau du plan d'expérimentation, de la description des expériences ou de la publication des résultats (McGeoch 1996,Rardin and Uzsoy 2001, Johnson 2002, Bartz-Beielstein et al. 2010, McGeoch 2012).

Motivations

Le but de nos travaux est de mettre les instances au cœur de l'étude et répondre aux besoins de la science de l'expérimentation. Nous allons aborder, selon Rardin and Uzsoy (2001), "les défis techniques les plus difficiles [qui] sont de trouver (ou générer) des instances de test convenables".

Les instances actuellement utilisées dans les études expérimentales sont issues soit de librairies d'instances, soit de générateurs aléatoires. Toutefois, leur utilisation n'a été sujette à aucune validation scientifique. Pire, elles sont même souvent critiquées pour leurs mauvaises propriétés. Certains générateurs d'instances ont des comportements asymptotiques prévisibles tels que l'on peut en déduire la valeur des solutions optimales, comme les instances *Random Euclidean* pour le problème de voyageur de commerce (TSP), décrites dans Johnson and McGeoch (1997), dont le ratio entre la valeur optimale et $\sqrt{n}$ tend vers une constante quand la taille n de l'instance tend vers l'infini (Beardwood et al. 1959). De même, les instances issues de librairies sont critiquées par Hooker (1995). Il affirme que le processus pour ajouter de nouvelles instances est biaisé, car les publications présentant de nouveaux algorithmes privilégient les résultats sur les instances pour lesquelles les algorithmes sont performants. Ainsi, les nouvelles instances intégrées aux librairies sont déjà résolues efficacement par au moins une méthode de résolution, ce qui semble remettre en cause leur pertinence. De manière générale, il manque des outils pour étudier l'utilisation d'instances pour l'expérimentation.

Dans cette thèse, nous proposons une réflexion sur la pertinence du choix d'instances pour la pratique expérimentale. Cette pertinence dépend donc des résultats empiriques des méthodes de résolution. Identifier des instances pertinentes permettra de valider scientifiquement la sélection d'instances de référence pour évaluer les mé-

thodes de résolution. Mais cela ouvrira aussi d'autres perspectives comme permettre la génération de nouvelles instances et, plus généralement, de mieux comprendre les problèmes, si on arrive à caractériser la source de leur difficulté à travers l'étude des instances.

Plan de la thèse

Tout au long de la thèse, nous avons centré nos travaux sur les instances. Nous abordons la notion sous diverses formes, en commençant par des approches expérimentales pour finir avec des études purement théoriques. Nous détaillons les différents sujets développés dans cette thèse et organisés sous forme de chapitres.

État de l'art

Le chapitre 1 présente la description faite dans la littérature des notions d'*instance*, de *classe d'instances* et de *difficulté d'instance*. Ce travail de synthèse est, à notre connaissance, la première étude qui présente une unification de la notion de difficulté d'instance, au croisement de plusieurs communautés : recherche opérationnelle, intelligence artificielle, théorie de la complexité, algorithmes évolutionnistes. Il s'appuie sur un modèle pour classifier les études portant sur la difficulté d'instance, selon la définition, la mesure et l'utilisation qui en est faite. De nombreux domaines d'application sont identifiés : la transition de phase, l'évolution d'instances, l'analyse de paysage, les portfolii d'algorithmes, la complexité d'instance et d'autres domaines aussi bien théoriques que pratiques.

Mesure de la difficulté d'instance

Nous précisons dans le chapitre 2 les définitions adoptées dans la thèse des notions présentées dans l'état de l'art, en particulier celle de la difficulté d'instance. Nous présentons ensuite une méthodologie pour évaluer la difficulté d'instance pour un ensemble d'algorithmes, qui repose sur la comparaison de deux classes d'instances. La méthode est décrite précisément étape par étape. Elle repose sur une analyse de la variabilité des performances des méthodes de résolution et des classes d'instances pour déduire le nombre d'exécutions et d'instances nécessaires pour mener une étude fiable.

Difficulté d'instance appliquée au TSP

Dans le chapitre 3, nous appliquons la méthode d'évaluation de la difficulté d'instance présentée au chapitre précédent. Trois études différentes sont menées, chacune reflétant une utilisation de la méthodologie. La première porte sur la pertinence de la classe d'instances *Random Clustered* par rapport à *Random Euclidean* sur un grand nombre de méthodes de résolution. Les études suivantes portent sur des algorithmes gloutons. Le but de la deuxième est la validation d'une nouvelle classe d'instances *Squared Norm* en remplacement de *Random Distance*. La dernière étude s'articule autour de l'influence de critères d'instance sur la difficulté du problème pour le TSP.

Modification d'instances pour le TSP

Le chapitre 4 porte sur une modification d'instances du TSP qui conserve la solution optimale. À partir de cette opération, on formalise un problème combinatoire dont la solution permet de générer des instances difficiles lorsqu'on mesure les performances par ratio. Le dual de ce problème est le problème de couplage fractionnaire parfait de poids minimum. On montre plusieurs résultats théoriques sur ces problèmes et on étudie l'impact empirique des instances difficiles sur plusieurs algorithmes.

Réductions d'instances et problèmes FPT

Nous présentons dans le chapitre 5 deux études théoriques sur deux problèmes *Fixed-Parameter Tractable* (FPT) : le problème du transversal minimum et le problème du nombre enveloppe. Nous étudions de nouvelles modifications d'instances pour ces problèmes et donnons une preuve que le second problème est FPT pour le paramètre de diversité de voisinage.

Chapitre 1

État de l'art

Nous présentons dans ce chapitre les deux notions clefs de nos travaux : instances et difficulté. Nous en donnons un premier aperçu à travers leur place dans la littérature existante.

1 Instances et classes d'instances

Papadimitriou (1994) présente les *instances* comme des objets mathématiques, sur lesquels on pose une *question* et on attend une réponse. Cette question caractérise le *problème* auquel appartient l'instance. Les problèmes que nous étudions sont algorithmiques, c'est-à-dire qu'on peut les résoudre avec un algorithme. Un *algorithme* est une méthode de résolution générale au problème, qui s'exécute étape par étape. Il prend en entrée une instance du problème et retourne en sortie la *solution*, qui est la réponse à la question du problème appliquée à l'instance.

Plusieurs algorithmes peuvent résoudre le même problème. Pour les comparer, on étudie leur *complexité*, qui représente le nombre d'étapes de leur exécution dans la pire situation. Une telle étude dépend de la taille des instances. La *taille* d'une instance représente la quantité de données nécessaires pour la décrire. Il peut toutefois exister plusieurs représentations de ces données. En théorie de la complexité, selon Garey and Johnson (1979), pour que la taille des représentations diffèrent au pire polynomialement, les représentations utilisées doivent être au moins en base 2 et sans données inutiles. On parle de *codage compact* des instances.

L'ensemble des instances d'un problème est appelé l'*espace d'instances* (Corne and Reynolds 2010,Smith-Miles and Lopes 2012). Pour les problèmes algorithmiques, la taille de leurs espaces d'instances est infinie. En restreignant l'espace d'instances, on peut définir de nouveaux problèmes, qui posent la même question. On parle dans ce cas de *sous-problèmes*. Pour une même question, l'espace d'instances d'un sous-problème est un sous-ensemble de l'espace d'instances du problème initial. Un

algorithme qui résout le problème initial, résout alors aussi le sous-problème.

Les instances sont également des objets pratiques. Alors qu'un problème est un énoncé formel, une instance en est une configuration précise. Rardin and Uzsoy (2001) définissent une instance comme une application numérique du problème. Pour tester empiriquement un algorithme, il faut le tester sur des instances du problème. Nous détaillerons plus loin les origines possibles des instances, mais nous pouvons déjà anticiper la présentation de méthodes algorithmiques, appelées *générateurs d'instances*. Ahammed and Moscato (2011) et Osorio and Cuaya (2005) définissent un générateur comme une fonction qui prend en entrée un ensemble de paramètres et qui retourne en sortie une instance de l'espace d'instances du problème. Or cette fonction n'est pas toujours injective, car il peut exister des paramètres qui génèrent la même instance. Ainsi parler des instances obtenues par un générateur simplement comme un ensemble d'instances est réducteur, car il manque la répétition éventuelle de certaines instances.

Pour traduire ce comportement, Birattari (2004b) définit la notion de *classe d'instances*, appelée aussi *classe de problème* par Vandegriend and Culberson (1998). Une *classe d'instances* est une description probabiliste d'un ensemble d'instances. Elle est définie par l'espace d'instances du problème et une distribution de probabilité sur les instances. Avec cette définition probabiliste, Birattari et al. (2006) font remarquer qu'on peut parler indifféremment d'une classe d'instances d'un problème ou de la *mesure de probabilité* des instances. En pratique, cette définition est assez originale et n'est pas toujours explicitement présentée ainsi, car la loi de distribution des instances n'est généralement pas connue.

En effet, les procédures de génération utilisent souvent des sources d'aléa. Par exemple, pour le problème du voyageur de commerce, une classe usuelle d'instances aléatoires consiste à tirer uniformément la distance entre chaque paire de sommets. Comme dans l'exemple, la mesure de probabilité des instances peut être complexe à calculer. Malgré cela, Birattari et al. (2006) font remarquer que la présence de la mesure de probabilité est implicite dans le processus expérimental, à travers des mesures moyennes sur les instances ou des tests de signification statistique. De plus, ils mettent en garde de ne pas confondre la mesure de probabilité des instances avec les lois de distribution internes aux générateurs aléatoires.

Hall and Posner (2001) présentent ce que devrait être un "bon" générateur d'instances. Ils énoncent quatre principes généraux, indépendants du problème ou de l'expérience, indispensables à suivre pour obtenir des instances de qualité : le *but* de l'étude, la *comparabilité* des résultats, la *conformité* des données, la *reproductibilité* de l'expérience. En particulier, la conformité des données peut concerner les instances dont la construction a engendré un biais, une propriété non voulue ; par exemple, certaines instances dont l'optimum est connu par construction peuvent avoir une structure particulière, non souhaitée pour l'étude (Rardin and Uzsoy 2001).

La définition par mesure de probabilité permet de définir n'importe quel sous-

ensemble d'instances comme une classe. Il existe toutefois certaines règles pratiques qui s'appliquent aux classes d'instances dans le cadre d'études empiriques. La première hypothèse concerne la taille des classes qui est supposée *infinie*. Selon Birattari et al. (2006), un nombre restreint d'instances risque d'engendrer des biais dans les études. En effet, pour la théorie de la complexité, un ensemble fini d'instances peut être résolu en temps polynomial par un algorithme ad hoc qui reconnaît les instances. Pour la pratique, concevoir une méthode de résolution sur un nombre restreint d'instances risque de trop la spécialiser au détriment du problème plus général. L'hypothèse suivante porte sur la cohérence de la classe d'instances. Pour utiliser des classes d'instances dans des études expérimentales, comme proposé par Ridge and Kudenko (2008), on doit pouvoir choisir des *instances représentatives*, telles que les conclusions soient généralisables à l'ensemble de la classe (Johnson 2002). Dans ce sens, Birattari (2004b) propose des classes d'instances minimales, telles qu'on ne puisse pas en extraire des sous-classes avec des descriptions propres.

Il existe plusieurs grandeurs, dépendant du problème, qui permettent de décrire les classes d'instances. Smith-Miles and Lopes (2012) parlent de *mesures*, ou Santana et al. (2012) et Smith-Miles and van Hemert (2011) de *caractéristiques*. Ces valeurs peuvent être des paramètres de la génération d'instances. On peut citer, comme caractéristiques d'instance, le ratio du nombre de clauses par variable pour le problème SAT ou l'écart-type entre les distances pour le problème du voyageur de commerce. Les mesures jouent souvent un rôle important dans la phase d'analyse des études empiriques pour évaluer, par exemple, le comportement de méthodes de résolution lorsque leur valeur varie.

On dénombre différents types d'instances et de classes d'instances. Greenberg (1990), Hall and Posner (2001) et McGeoch (2012) proposent une classification qui dépend de leurs origines et de l'utilisation qui en est faite. Les principaux types sont les instances aléatoires, les instances structurées et celles compilées dans des librairies. Les instances *aléatoires* sont obtenues par des générateurs aléatoires, comme pour l'exemple précédent du voyageur de commerce. On considère qu'elles n'ont pas de structure a priori et ne dépendent que d'une génération aléatoire brute des données de l'instance. Les instances *structurées* sont aussi obtenues par des générateurs, avec ou sans une part d'aléa, mais les instances générées possèdent des propriétés particulières, telles les instances fractales ou des propriétés d'applications réelles. Les *librairies* sont des compilations d'instances, généralement issues de divers applications réelles. Elles peuvent permettre de pallier le manque de données réelles sur un problème. Leur diffusion en font des bases de test de référence pour comparer les performances des méthodes de résolution. Or la taille limitée du jeu d'instances et les motivations parfois floues sur leur construction, comme critique Hooker (1995), demandent une utilisation précautionneuse. Les autres types d'instances, moins communs, sont les cas réels, les cas pathologiques et les pires cas. Les instances *réelles* sont celles issues directement du problème pratique que l'on cherche à résoudre. Or ces instances sont généralement très difficiles à obtenir et en petit nombre, souvent pour des raisons de confidentialité ou un manque d'historique des données. Les *cas*

pathologiques ont pour but de tester la robustesse de l'implémentation de l'algorithme. Ils sont de faible intérêt pour éprouver la résolution, car les solutions sont généralement triviales, mais ils ont un rôle dans le développement logiciel pour tester le bon fonctionnement des programmes. Les *pires cas* sont des résultats théoriques de l'analyse de complexité des algorithmes. Ils représentent le pire déroulement d'un algorithme donné. Nous ne ferons pas une étude plus approfondie des motivations autour de chaque type d'instances. Ce qu'il faut retenir est que pour chaque type, il existe un contexte d'application qui a motivé la construction de ces instances.

2 La notion de difficulté d'instance

La notion de difficulté d'instance est sous-jacente à différents domaines de recherche. Elle peut prendre plusieurs formes et elle est généralement assez mal identifiée comme telle. À notre connaissance, il n'y a pas de modèle cherchant à unifier et partager cette notion. Certains auteurs, comme Hooker (1994) ou Rardin and Uzsoy (2001), essaient de sensibiliser la communauté sur les problématiques liées à l'expérimentation. À défaut de proposer un modèle unifié et des solutions claires, ces études ont ouvert une porte vers un nouveau domaine de recherche sur la difficulté d'instance.

La première étude reconnaissant la notion de difficulté d'instance en tant que telle est, à notre connaissance, la revue de littérature de Smith-Miles and Lopes (2012). Elle présente une liste de travaux qui étudient le lien entre des caractéristiques d'instances et la performance des algorithmes pour plusieurs problèmes d'optimisation combinatoire : problèmes d'affectation, du voyageur de commerce, de découpe et de conditionnement, dans des graphes, d'emplois du temps, et de satisfaction de contraintes. Malgré le titre *"Measuring Instance Difficulty for Combinatorial Optimization Problems"*, semblant s'intéresser à la difficulté d'instance en elle-même, on déplore l'absence de définition formelle de cette notion. Le message de cette revue de littérature se focalise principalement sur l'appréhension des instances pour les portfolii d'algorithmes. La motivation des auteurs consiste à évaluer et reconnaître les instances afin de leur appliquer l'algorithme le plus adapté. Cet article est une source majeure d'information pour nos travaux, avec ses nombreuses références bibliographiques. Bien que nous considérions cette approche comme importante pour la difficulté d'instance, elle nous semble toutefois trop restreinte au regard des perspectives existantes autour de la notion. En effet, la motivation principale de notre approche consiste à améliorer la qualité des études expérimentales. Enrichir nos connaissances sur les instances doit permettre de proposer des bases de tests pertinentes pour le benchmarking.

De manière générale, nous voulons donner une présentation plus globale et unifiée de la notion. Pour cela, nous présentons dans la suite les différentes notions de difficulté d'instance rencontrées dans la littérature. Nous proposons une définition

de la difficulté (section 2.1), puis détaillons les outils utilisés dans la littérature pour la mesurer (section 2.2) et enfin décrivons l'utilisation qui en est faite (section 2.3). Il s'en dégage des critères de classification des différentes notions de difficulté d'instance. Nous les utilisons dans la section 3 pour décrire les différents types d'études.

2.1 Définition de la difficulté

La notion de difficulté d'une instance représente sa capacité à éprouver sa résolution. Cette définition va dans le même sens que les principes de la théorie de la complexité, qui classe les problèmes selon la capacité existante à les résoudre. La difficulté considérée n'est pas toujours la même entre les études. En nous appuyant sur des modèles de la littérature, nous proposons une définition de cette notion, qui dépend de deux critères que nous appelons *référentiel* et *portée* de la difficulté, et qui sont présentés dans la suite.

Modèle de Rice

Smith-Miles and van Hemert (2011), en reprenant les travaux de Rice (1976), présentent un modèle qui sert de contexte pour le *problème de sélection d'algorithme*, consistant à choisir, pour résoudre une instance, l'algorithme qui minimise sa difficulté. Le modèle possède quatre composantes essentielles.

- l'ensemble P des instances du problème ;
- l'espace F des caractéristiques d'instance ;
- l'ensemble A des algorithmes de résolution ;
- l'espace Y des performances qui représente les vecteurs des performances de chaque algorithme pour chaque instance.

La sélection d'algorithme repose sur une fonction d'application $S : F \to A$. Pour une instance donnée $x \in P$, avec des caractéristiques $f(x) \in F$, la fonction $S(f(x))$ retourne l'algorithme $\alpha \in A$ qui est le plus adapté pour résoudre x, *i.e.* tel que la mesure de performance $\|y(\alpha, x)\|$ de l'algorithme α, avec $y(\alpha, x) \in Y$, soit maximum.

À travers ce modèle formalisé, l'auteur présente implicitement une notion de difficulté d'instance. La sélection d'algorithme consiste à proposer l'algorithme le plus adéquat pour résoudre l'instance proposée. Du point de vue de l'instance, c'est l'algorithme pour lequel la difficulté de l'instance est minimale. La difficulté dépend alors des performances d'une unique méthode de résolution.

Référentiel de difficulté

Le premier élément de classification de la difficulté porte sur les méthodes de résolution qui servent de base à la mesure de la difficulté. Nous appelons cela le

référentiel de l'étude. D'après Vandegriend and Culberson (1998), la difficulté d'instance est relative aux performances d'une méthode de résolution. Ainsi les résultats sur la difficulté d'instance dépendent de l'algorithme considéré. Toutefois, l'étude peut aussi porter sur plusieurs algorithmes. On différencie donc les études selon la nature et le nombre de méthodes considérées.

Dans le modèle de Rice, la difficulté d'instance est évaluée vis à vis d'un seul algorithme donné. Cette situation est la plus fréquemment rencontrée dans la littérature. Toutefois il existe des cas où la difficulté porte sur des *classes d'algorithmes*, i.e. des ensembles d'algorithmes qui partagent des propriétés communes, par exemple les algorithmes par recherche locale (Kochetov and Ivanenko 2005, Papadimitriou and Steiglitz 1978) ou les algorithmes de type Branch-and-Bound (Chvátal 1980, Pisinger 2005). Plus généralement encore, la complexité d'instance, présentée par Schöning (1990), considère la difficulté d'instance pour toute méthode de résolution.

Portée de la difficulté

D'autre part, la portée de la difficulté diffère selon les études, entre vision relative ou absolue. Dans beaucoup d'études, on justifie la difficulté d'une instance en comparaison à d'autres instances. Mais il existe aussi une difficulté intrinsèque suffisante pour qualifier des instances de difficile.

La comparaison est souvent relative. Les nouvelles instances sont comparées aux instances classiques de la littérature issues de librairies d'instances (Ridge and Kudenko 2008) ou de générateurs (Arthur and Frendewey 1988, Fischer et al. 2005, Osorio and Pinto 2003). Ces études cherchent ainsi à montrer que les nouvelles instances qu'elles proposent sont au moins aussi difficiles que les instances usuelles, c'est-à-dire que la résolution par les algorithmes considérés n'est pas plus facile.

Certaines études étendent cette comparaison à des ensembles plus larges d'instances. Elles considèrent que les instances étudiées sont au moins aussi difficiles que toute autre instance (Cheeseman et al. 1991, Cotta and Moscato 2003). Dans ce cas, on peut parler de difficulté absolue. En pratique, l'ensemble d'instances considéré est généralement restreint pour permettre une comparaison pertinente. Par exemple, si le taille des instances n'est pas fixée, les instances les plus grandes seront généralement les plus difficiles. Cela n'empêche pas de mener des études en faisant varier la taille des instances mais en comparant à taille fixée.

On identifie une autre situation de difficulté absolue, qui est cette fois indépendante d'autres instances. La difficulté ne dépend que de la mesure donnée par l'algorithme de résolution sur la famille infinie d'instances évaluée. Il n'est plus question de comparaison avec d'autres résultats. Certaines études, comme Lawler et al. (1985), proposent des instances dédiées à un algorithme précis qui traduisent le pire déroulement de l'algorithme asymptotiquement, à un facteur constant près. Ces instances difficiles sont les *pires cas* de l'algorithme. D'autres études s'appuient

sur l'étude du comportement asymptotique des algorithmes, mais en relâchant la recherche du pire cas à des cas qui induisent plus généralement un mauvais comportement de l'algorithme. Pour Chvátal (1980), qui considère comme mesure de difficulté le temps d'exécution de méthodes exactes, une instance difficile demande au moins un temps de résolution qui est exponentiel avec $\sqrt{t}$, où t est la taille de l'instance. Dans ce cas plus général, on peut parler de recherche de *mauvais cas*.

En résumé, la difficulté d'instance représente la capacité à éprouver la résolution d'algorithmes. Elle dépend donc logiquement des méthodes de résolution considérées, mais aussi de leur nombre : une seule, un ensemble ou toutes. De plus, la portée de la difficulté peut être relative (comparaison avec un ensemble d'instances) ou absolue (soit aussi difficile que toute autre instance, soit un mauvais cas pour la résolution des algorithmes).

2.2 Mesure de la difficulté

La difficulté d'instance se mesure à travers le comportement d'une méthode de résolution. On différencie les études où le résultat de l'algorithme est obtenu théoriquement, par étude analytique de son déroulement (Chvátal 1980, Lawler et al. 1985, Papadimitriou and Steiglitz 1978), à celles où le résultat est obtenu en pratique par l'exécution de l'algorithme sous forme de programme informatique (Fischer et al. 2005, Ridge and Kudenko 2008).

Quelle que soit la méthode employée, il n'y a que deux types de mesure : soit sur le *temps d'exécution* de l'algorithme, soit sur la *qualité de la solution* obtenue. Par contre, pour chaque type, il existe de nombreux *critères de mesure*. Les principaux d'entre eux, avec quelques exemples d'application, sont la durée (Fischer et al. 2005, Pisinger 2005) ou le nombre d'itérations d'une sous-routine (Chvátal 1980, van Hemert and Urquhart 2004) pour le temps d'exécution et l'écart à la valeur optimale (Julstrom 2009, Ridge and Kudenko 2008) ou à une borne (Kosoresow and Johnson 2002, Schiavinotto and Stützle 2004) pour la qualité de la solution.

Le type de mesure dépend de la méthode de résolution. Pour les méthodes exactes, seule la mesure du temps de résolution a un sens. Pour les algorithmes approchés, on trouve les deux types de mesure.

2.3 Utilisation de la difficulté

La difficulté d'instance est rencontrée dans différentes études. On y observe différentes utilisations. Nous en avons identifié trois principales.

Valider la difficulté d'instance

Plusieurs travaux proposent de nouvelles instances comme jeu d'instances. Avant de les utiliser comme tel, elles sont soumises à une étude pour valider la pertinence de leur utilisation. Celle-ci consiste à tester si les instances sont suffisamment difficiles. Dans ce but, Ridge and Kudenko (2008) proposent une méthodologie sous forme d'un plan d'expérimentation générique pour comparer la difficulté de classes d'instances vis à vis d'une méthode de résolution. De précédents travaux, sans référence méthodologique, utilisaient déjà un schéma similaire pour valider leurs classes d'instances (Arthur and Frendewey 1988, Osorio and Cuaya 2005, Osorio and Pinto 2003, Schiavinotto and Stützle 2004).

Générer des instances difficiles

Certaines études s'appuient sur la notion de difficulté d'instance pour générer des instances difficiles. La difficulté effective des instances dépend de la méthode de génération. On distingue deux approches. Dans l'une, on génère expérimentalement des instances de telle sorte qu'on obtienne des instances difficiles. La principale méthode ainsi appliquée s'appelle l'évolution d'instances (Cotta and Moscato 2003, Kosoresow and Johnson 2002, van Hemert 2003) ; nous la présentons dans la suite. Dans l'autre, une méthode ad hoc de construction des instances garantit théoriquement que les instances générées possèdent des propriétés qui les rendront difficiles pour les algorithmes considérés (Chvátal 1980, Drezner et al. 2005, Kochetov and Ivanenko 2005, Papadimitriou and Steiglitz 1978, Pisinger 2005).

Caractériser la source de difficulté

Pour comprendre la difficulté d'instance et pouvoir générer facilement des instances difficiles, certains auteurs cherchent des critères d'instances qui caractérisent la difficulté d'instance. Ces critères sont généralement des métriques mesurées sur l'instance, par exemple pour le problème du voyageur de commerce, la déviation standard des distances ou le taux de regroupement des villes (Smith-Miles et al. 2010). On distingue deux approches. À partir d'instances difficiles, Cotta and Moscato (2003), Corne and Reynolds (2010), Santana et al. (2012) et Smith-Miles and van Hemert (2011) cherchent des propriétés à des instances qu'ils savent difficiles pour les caractériser et ainsi identifier des critères de difficulté pour les instances. À l'inverse, Fischer et al. (2005), Ridge and Kudenko (2008) et Stützle and Fernandes (2004) partent d'instances avec des critères identifiés et étudient leur difficulté pour valider ou non l'impact de ces critères dans la difficulté d'instance.

3 La place de la difficulté d'instance

Nous allons maintenant présenter de manière plus précise les différentes études qui s'appuient sur la difficulté d'instance. D'après la présentation qui a été faite précédemment, nous allons pouvoir classifier les études selon le référentiel, la portée, la mesure et leurs utilisations de la difficulté d'instance.

3.1 Transition de phase

Le phénomène de transition de phase peut s'observer en optimisation combinatoire lorsqu'on étudie des instances aléatoires. Cheeseman et al. (1991) conjecturent l'équivalence qu'un problème soit *NP*-difficile et qu'il existe une transition de phase. Les premières études portèrent sur le problème SAT. Goldberg (1979) observe que les instances générées aléatoirement sont en moyenne faciles. En mesurant par exemple le ratio du nombre de clauses par variable, Monasson et al. (1999) constatent que pour un ratio petit, le problème est presque toujours satisfiable et, lorsque le ratio est grand, il est insatisfiable. Le phénomène qui nous intéresse apparaît lorsque l'on passe de l'un à l'autre. Il s'avère que la probabilité qu'une instance soit presque certainement satisfiable lorsque l'on augmente le ratio, passe brusquement de 1 à 0, car le problème passe de sous-contraint à sur-contraint (figure 1.1). En référence aux observations faites en chimie, le passage rapide d'un état à un autre est ce qu'on appelle la *transition de phase*.

De manière générale, la transition de phase dépend de la caractéristique d'instance que l'on fait varier et qui s'appelle, dans ce contexte, le *paramètre d'ordre*. Dans le cas de notre exemple, le paramètre d'ordre est le ratio du nombre de clauses par variables. La transition de phase s'observe autour d'une valeur de ce paramètre, qui peut être estimée empiriquement et parfois calculée théoriquement. Toujours dans notre exemple, selon Selman et al. (1996), la valeur estimée du seuil de transition pour le problème 3-SAT tend vers 4,25 quand la taille de l'instance croît.

La transition de phase implique la présence d'instances difficiles autour du seuil de transition. Lorsque l'on applique une méthode de résolution sur les instances d'un problème où l'on observe une transition de phase, les performances de la méthode présenteront un pic de difficulté au niveau de ce seuil (figure 1.2). Williams and Hogg (1992) décrivent cette valeur comme celle où la "fluctuation de la difficulté est la plus grande". Intuitivement, à la frontière entre les deux états, il est plus difficile de décider de la satisfiabilité d'une instance.

On peut examiner la difficulté d'instance rencontrée dans l'étude de la transition de phase à partir de la classification présentée dans la section 2. Le référentiel de la difficulté repose sur une méthode de résolution précise et un critère de performance associé. Toutefois Selman et al. (1996) suspectent une généralisation possible du phénomène à toute méthode qui résout le problème. Hayes (1997) va plus loin puis-

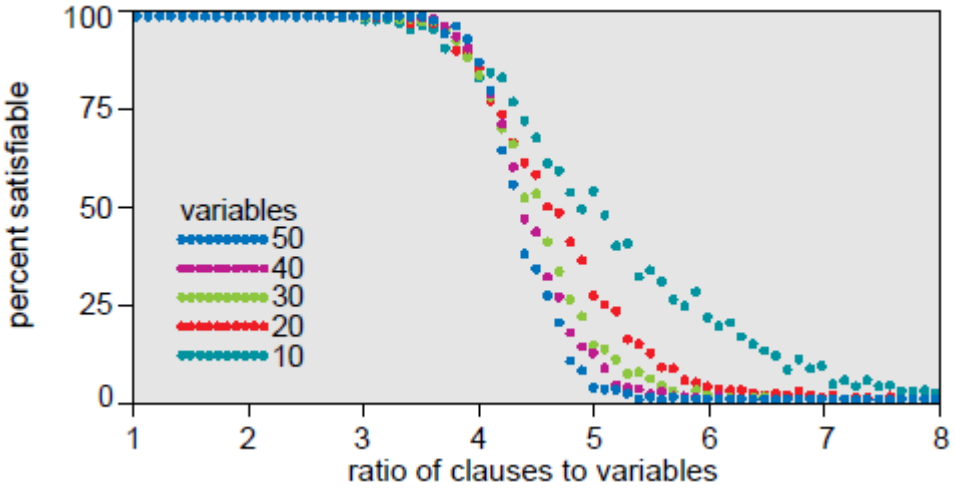

FIGURE 1.1: Évolution de la satisfiabilité pour différentes tailles d'instances de 3-SAT lorsque le nombre de clauses par variable varie (Hayes 1997).

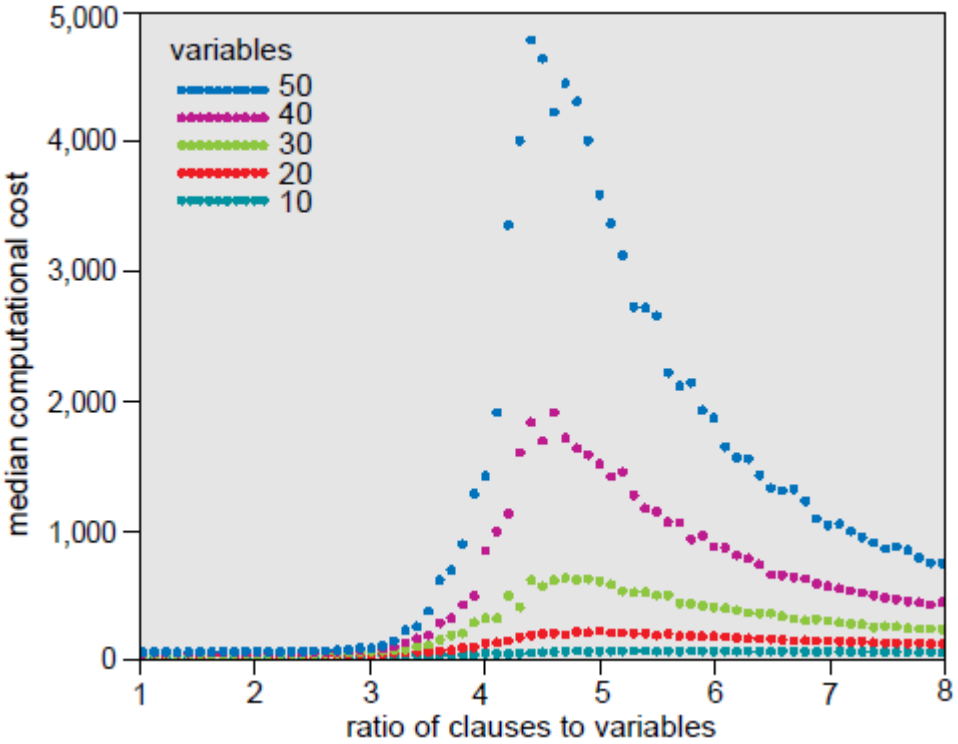

FIGURE 1.2: Évolution de la difficulté dans la résolution de différentes tailles d'instances de 3-SAT lorsque le nombre de clauses par variable varie (Hayes 1997).

qu'il considère qu'il est "maintenant bien établi que la transition de phase pour le problème SAT est intrinsèque au problème". Mais nous n'avons pas trouvé d'études qui valident cette déclaration.

L'approche employée laisse penser à une portée absolue. On identifie les instances les plus éprouvantes. Mais il faut relativiser cette portée, car on se place sur des classes d'instances particulières, qui sont des instances aléatoires.

Le but des études qui traitent de la transition de phase est de générer de nouvelles instances difficiles pour le benchmarking d'algorithmes. L'utilisation d'instances aléatoires est assez courante dans cette pratique. Or l'étude de la transition de phase montre clairement que la plupart des instances aléatoires sont faciles à résoudre, si la valeur de leur paramètre d'ordre n'est pas proche de la valeur du seuil de transition. Ces études permettent aussi de caractériser des instances difficiles à travers la valeur du seuil de transition. Toutefois il faut garder à l'esprit que l'étude se place dans le cadre d'instances aléatoires. On ne peut donc pas généraliser cette observation à n'importe quelle instance. Gent and Walsh (1994) présentent des instances difficiles pour des valeurs du paramètre d'ordre différentes du seuil. Inversement certaines instances avec le paramètre égal à la valeur de seuil peuvent être faciles, comme illustré par l'étude de Vandegriend and Culberson (1998).

3.2 Évolution d'instances

Les premiers travaux utilisant cette méthode concernent la génération de vecteurs de test pour la vérification de fautes dans les circuits intégrés. L'objectif est de trouver des séquences de valeurs à charger dans les circuits pour couvrir l'ensemble des fautes en un minimum de temps. Pour générer ces séquences, Saab et al. (1992), Srinivas and Patnaik (1993) et Rudnick et al. (1994) ont proposé une méthode originale qui consiste à faire évoluer, grâce à un algorithme génétique, les séquences afin de trouver la plus adéquate pour l'exercice. Par ailleurs, Wegener et al. (1997) appliquent une méthode similaire pour faire évoluer des jeux de tests pour la validation de logiciel.

Ces études ont en commun la volonté de générer des tests qui éprouvent le système étudié, ce qui rappelle la notion d'instance difficile précédemment donnée. Les systèmes qui nous intéressent sont les algorithmes. À notre connaissance, les premières utilisations d'une telle approche sur des méthodes de résolution pour des problèmes combinatoires remontent à Johnson (2001). Cette étude porte sur le problème de couplage maximum dans un graphe biparti, qui est un problème polynomial. De même, Cotta and Moscato (2003) ont testé des algorithmes de tri. Dans ce contexte, les auteurs essaient d'obtenir des instances difficiles pour évaluer la complexité de leurs algorithmes.

Kosoresow and Johnson (2002) sont les premiers à appliquer la méthode d'*évolution d'instances* à un problème *NP*-difficile, en l'occurence le *Hard Planar Taxicab*

Problem. Cette étude sera suivie par plusieurs autres sur différents problèmes *NP*-difficiles : satisfaction de contraintes (van Hemert 2003), voyageur de commerce (van Hemert and Urquhart 2004), SAT (van Hemert 2006), sac à dos (Julstrom 2009), ordonnancement (Branke and Pickardt 2011), programmation quadratique (Porta and Julstrom 2012).

L'approche employée se rapproche de la résolution d'un problème d'optimisation. Pour une méthode de résolution donnée d'un problème, on cherche des instances du problème qui minimisent les performances considérées de la méthode de résolution. La recherche de solution se fait alors grâce à des méta-heuristiques classiques : algorithme génétique (Kosoresow and Johnson 2002) ou recherche locale (Ahammed and Moscato 2011). Les solutions sont ici les instances et leur valeur objectif est donnée à travers l'exécution de la méthode de résolution étudiée. Les opérations d'évolution (croisement, mutation, voisinage) portent sur les instances, généralement en laissant invariante leur taille. Dans van Hemert (2005), le croisement d'instances du problème du voyageur de commerce euclidien est obtenu en choisissant pour chaque ville, de manière équiprobable, soit les coordonnées de la mère, soit celles du père. La définition de ces opérations semble être la partie originale et délicate de cette approche, car elles sont déterminantes dans l'évolution des instances, mais il existe peu de travaux qui traitent du sujet.

Dans ces études, le référentiel de la difficulté d'instance repose sur une unique méthode de résolution et un critère de performance associé, que l'évolution d'instances cherche à maximiser. La vision idéale du concept correspond à une portée absolue. Une instance sera difficile car elle sera la solution optimale du problème d'optimisation consistant à trouver une instance plus difficile que toutes les autres. Mais en pratique, la portée reste plus relative, car les méthodes de génération utilisées ne garantissent pas l'optimalité de la solution retournée ; elles ne retournent que des optimaux locaux. La mesure de la difficulté se fait empiriquement et représente la valeur de l'instance. Selon les méthodes de résolution, le critère de mesure peut être le temps de résolution (van Hemert 2003) ou la qualité de la solution (Kosoresow and Johnson 2002).

L'objectif principal est la génération d'instances difficiles, dont la difficulté est garantie par la méthode de génération. Ceci permet d'approcher les pires cas d'exécution des méthodes de résolution. Johnson (2001), Kosoresow and Johnson (2002) et Branke and Pickardt (2011) utilisent cette approche pour évaluer la complexité d'algorithmes et les classer selon leur efficacité. Le second objectif s'appuie sur le premier. À partir des instances difficiles générées, plusieurs études essaient de les caractériser en s'appuyant sur des critères d'instances. Smith-Miles et al. (2010) en donnent une liste pour le problème du voyageur de commerce. Pour le même problème, van Hemert (2005) montre que certaines instances éprouvantes pour l'algorithme de Lin-Kernighan, générées par l'évolution d'instances, sont fortement clusterisées, i.e. les villes sont réparties en plusieurs groupes denses.

3.3 Analyse de paysage

La notion de paysage a été à l'origine développée pour le domaine de la biologie (Wright 1932). Elle permettait de modéliser le concept d'évolution des espèces pour essayer de la comprendre. Ce n'est que plus tard que la méthode a été transposée au domaine de l'optimisation combinatoire (Brady 1985).

L'*analyse de paysage* est un outil pour mieux comprendre la structure des problèmes pour les méthodes de résolution de type recherche locale, qui partent d'une solution réalisable et l'améliorent par modifications successives. Ces modifications se traduisent par une relation de voisinage ν, qui associe à chaque solution réalisable son ensemble de solutions réalisables voisines.

L'analyse porte sur l'étude de l'ensemble des solutions réalisables Ω pour une instance du problème traité. Le problème d'optimisation dépend de la fonction objectif f, qui évalue la qualité des solutions réalisables.

Un paysage est défini par un triplet (Ω, ν, f) (Marmion 2011). Il représente l'espace de recherche pour résoudre le problème traité avec la relation de voisinage étudiée. L'ensemble des solutions réalisables sont distribuées selon leurs distances de voisinage et leur altitude dans le paysage est donnée par la fonction objectif. On peut illustrer l'approche par une métaphore géographique. Dans le cas d'un problème de minimisation, le problème peut être vu comme trouver la vallée la plus basse dans une chaîne montagneuse, en parcourant le paysage. Or la multiplication de vallées (optima locaux) à visiter ou des vallées trop vastes (bassin d'attraction) peuvent décourager les marcheurs (méthodes de résolution).

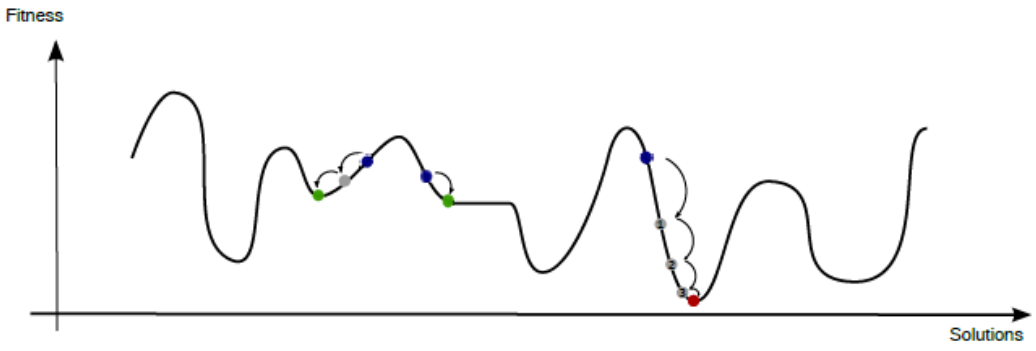

FIGURE 1.3: Exemple de paysage et comportement de plusieurs appels à un algorithme de descente (Marmion 2011).

Pour les problèmes *NP*-difficiles, le nombre de solutions réalisables croît exponentiellement avec la taille de l'instance. Une approche exhaustive pour décrire le paysage est donc proscrite. Pour y pallier, des méthodes approchées ont été imaginées. Pitzer and Affenzeller (2012) présentent différentes méthodes pour échantillonner les paysages afin de les décrire. La description peut ensuite se faire grâce à divers indicateurs. Par exemple, Weinberger (1990) présente une mesure de la rugosité du

paysage, qui décrit le nombre d'optima locaux et qui est une des premières mesures de difficulté. Bien d'autres indicateurs ont ensuite été proposés comme décrits par Marmion (2011).

La notion de difficulté d'instance présente dans l'analyse de paysage est expliquée par Weise et al. (2009). Une instance difficile rend plus pénible l'exécution des méthodes de résolution. Le référentiel de la difficulté est l'ensemble des algorithmes qui utilisent la relation de voisinage ν étudiée. La portée de la difficulté concerne en pratique une seule instance. L'étude porte de manière intensive sur cette seule instance. Toutefois, les études essaient de généraliser ces résultats à une classe d'instances dont l'instance étudiée serait caractéristique. La force de ces études repose sur les nombreuses mesures de la difficulté. Ces mesures ont inspiré d'autres types d'études comme Smith-Miles and van Hemert (2011) avec le portfolio d'algorithmes, qui évaluent a priori la difficulté des instances grâce à ces mesures pour trouver la meilleure méthode de résolution à appliquer.

Le principal objectif de ces études est de comprendre ce qui perturbe le bon fonctionnement des méthodes de résolution, en décrivant les instances "piégeuses". Vis à vis de la difficulté d'instance, cela peut être vu comme une caractérisation des instances difficiles. Les nombreux indicateurs en permettent une description précise. Santana et al. (2012) ont utilisé ces outils pour caractériser des instances difficiles obtenues par évolution d'instances, ou Schiavinotto and Stützle (2004) pour étudier la difficulté de classes d'instances.

3.4 Portfolii d'algorithmes

Les *portfolii d'algorithmes* peuvent être vus comme des méthodes de résolution. Huberman et al. (1997) proposent de s'appuyer sur une collection d'algorithmes pour les combiner et proposer une méthode plus efficace que chaque algorithme individuellement. On peut distinguer deux applications différentes de ce concept.

La première approche a pour but de rendre les algorithmes stochastiques plus robustes. Les algorithmes randomisés peuvent donner de très bons résultats. Mais leur déroulement reste variable d'une exécution à l'autre et donc les performances peuvent aussi être très mauvaises. Pour limiter la variabilité et donc le risque de mauvais résulats, Gomes and Selman (1997) proposent de combiner l'exécution simultanée de plusieurs algorithmes, en parallèle ou en séquentiel. Ce n'est toutefois pas cette approche qui nous intéresse dans le cadre de la difficulté d'instance. Pour plus d'informations se référer à Gomes and Selman (2001).

La seconde application propose une stratégie plus fine pour choisir les algorithmes à exécuter. La stratégie classiquement employée consiste à comparer les performances de différentes méthodes de résolution et à choisir la méthode qui répond le mieux au problème. Or il est très rare qu'une méthode supplante toutes les autres pour toute instance du problème. La "bonne" méthode de résolution est alors

celle qui globalement donne les meilleurs résultats. Ce choix est donc un compromis, en sachant que la méthode choisie peut être surpassée par d'autres méthodes sur certaines instances. Le principe des portfolii d'algorithmes proposé par Leyton-Brown et al. (2003) est précisément de ne pas faire ce compromis et d'appliquer à chaque instance le meilleure algorithme connu pour cette instance. Une des applications les plus remarquables est sans doute le portfolio d'algorithmes SATzilla, présenté par Xu et al. (2008), qui s'est illustrée à de nombreuses reprises lors des compétitions annuelles sur le problème SAT.

Ces portfolii partagent des techniques utilisées sur le *problème de sélection d'algorithmes*, formalisé par le modèle de Rice (1976), et présenté dans la section 2.1. Ce problème consiste à trouver l'algorithme à exécuter qui minimise une performance, par exemple le temps d'exécution. Pour le résoudre, il est nécessaire d'être capable de prédire la performance d'un algorithme sans l'exécuter. Dans ce but, on mesure la performance de l'algorithme et les caractéristiques d'instances d'un échantillon de l'espace d'instances. À partir de ces données, on crée un modèle par régression qui prédit la performance de l'algorithme pour toute instance du problème. Ce modèle s'appuie sur des corrélations entre des caractéristiques et la difficulté d'instance ; pour plus de détails sur les techniques employées pour trouver ces corrélations, se référer à Brown et al. (2009). Grâce à cet estimateur, Goldberg (1999) propose de comparer les algorithmes entre eux pour essayer de choisir lequel est le plus approprié à la résolution.

Les portfolii utilisent aussi l'estimateur pour comparer les méthodes de résolution, mais il en découle une fonction qui couvre l'ensemble des instances et associe à chacune d'entre elles l'algorithme le plus efficace. La distribution est organisée en familles d'instances qui sont définies par des caractéristiques d'instances. Smith-Miles and van Hemert (2011) donnent un exemple de cette classification pour le problème du voyageur de commerce. Les portfolii reposent sur une première phase d'identification de l'instance à résoudre pour trouver à quelle famille elle appartient. Puis on assigne à l'instance l'algorithme associé à la famille, dont il a été prédit qu'il serait le plus performant.

Bien que la motivation des portfolii d'algorithmes soit la résolution du problème, la difficulté d'instance y est très présente. Sélectionner le bon algorithme pour résoudre une instance consiste à trouver l'algorithme qui minimise la difficulté de l'instance. Toutefois, dans cette approche, les instances difficiles ne sont pas l'objet de l'étude. On cherche au contraire à les éviter. Il n'en demeure pas moins que l'analyse de la difficulté d'instance vis à vis du portfolio donne une approche intéressante de la notion. Le référentiel de cette difficulté d'instance est l'ensemble des méthodes de résolution connues, plus précisément la plus performante d'entre elles sur l'instance testée, pour un critère de performance donné (généralement le temps d'exécution). De plus, les outils pour caractériser les familles d'instances peuvent être appliqués, comme dans Smith-Miles and van Hemert (2011), pour caractériser des familles d'instances difficiles.

3.5 Cœurs de complexité et complexité d'instance

Ces deux notions sont des approches théoriques de la difficulté d'instance. Bien qu'elles n'offrent pas de méthodes pour générer des instances difficiles, elles permettent de prouver leur existence. Elles s'appuient sur une notion de difficulté un peu différente de celle considérée jusqu'à présent.

La notion de *cœurs de complexité* (*complexity cores*) a été introduite par Lynch (1975). Un cœur de complexité est un ensemble d'instances qui sont fondamentalement difficiles pour toute méthode de résolution. Il en découle que de tels ensembles sont infinis. En effet, un ensemble fini serait facilement résolu par un algorithme ad hoc qui reconnait les instances et retourne la réponse associée qui aura été préalablement calculée et stockée.

Formellement, Schöning (1990) définit la notion comme suit. Il note par Σ^* l'ensemble des instances du problème étudié. L'implémentation d'un algorithme sur un ordinateur peut être vue comme une machine de Turing. Tout d'abord, l'auteur définit un ensemble d'instances difficiles pour une machine M, en respectant une limite sur le temps d'exécution $t : \mathbb{N} \to \mathbb{N}$.

$$\text{Hard}(M, t) = \{x \in \Sigma^* | T_M(x) > t(|x|)\},$$

où $T_M(x)$ représente le nombre d'étapes sur la machine M pour résoudre x.

Un cœur de complexité pour un ensemble d'instances A est un ensemble infini X d'instances tel que pour toute machine M qui résout l'ensemble A, et pour tout polynôme p,

$$X \subseteq \text{Hard}(M, p) \text{ } p.p. \text{ (presque partout)},$$

ce qui signifie que seul un nombre fini d'instances de X ne sont pas dans $\text{Hard}(M, p)$. Si $P \neq NP$, tout problème NP-complet a un cœur de complexité (Schöning 1990).

La notion de cœur de complexité fait partie du domaine de la théorie de la complexité asymptotique. Cette théorie repose sur la complexité au pire cas. Ainsi toutes les instances ne sont pas autant difficiles à résoudre. Malheureusement cette approche ne nous permet pas de mesurer une complexité individuelle des instances.

Ceci est précisément le but de la *complexité d'instance* (*instance complexity*), notion due à Hartmanis (1983). Elle permet d'évaluer la complexité pour résoudre des instances individuelles. Comme expliqué précédemment, un nombre fini d'instances, même pour un problème NP-complet, peut être résolu facilement par un algorithme ad hoc qui reconnait les instances. La complexité d'instance s'appuie sur la description de l'instance à reconnaître.

Orponen et al. (1994), à partir de précédents travaux (Ko et al. 1986, Orponen 1990), donnent une définition formelle de la complexité d'instance pour une instance x en respectant un ensemble A et une limite t sur le temps d'exécution. Noté $ic^t(x : A)$, c'est la taille du plus petit algorithme qui décide x en moins de $t(x)$

tout en répondant sans erreur à toutes les instances de A (une réponse indécise est autorisée). Si $P \neq NP$ et A est NP-complet, alors pour tout polynôme t et pour toute constante c, il existe une infinité d'instances x telles que $ic^t(x : A) \geq c \log |x|$ (Orponen et al. 1994). Cela signifie que le plus petit algorithme pour un nombre infini d'instances consiste à reconnaître l'instance x, dont la taille du codage vaut $\log |x|$.

Le codage minimum d'une instance, qui peut être vu comme une chaîne de caractères, rappelle naturellement la notion de complexité de Kolmogorov (Kolmogorov 1965, Li and Vitányi 1997). Orponen et al. (1994) ont montré la relation suivante entre les deux notions. Une instance est difficile pour un problème si sa complexité d'instance est similaire à sa complexité de Kolmogorov. Si *EXPTIME* $\neq$ *NEXPTIME*, tout problème *NP*-complet a des instances difficiles (Schöning 1990).

Le référentiel ambitieux de ces études concerne l'ensemble des méthodes de résolution. Pour un problème de décision donné, convient tout algorithme qui permet de décider l'ensemble des instances du problème, sachant que les réponses indécises sont autorisées. Mais cet algorithme doit répondre en un temps d'exécution borné.

Les instances issues d'un cœur de complexité sont difficiles car elles éprouvent suffisamment les méthodes de résolution. Du point de vue de la complexité d'instance, une instance est difficile car sa reconnaissance est complexe et la taille de tout algorithme qui la résout dépend de la taille de l'instance. La portée de ces notions de difficulté peuvent ainsi être vue comme absolue.

Comme les méthodes de résolution employées sont exactes, le critère naturel de mesure est la durée d'exécution. Toutefois ici, ce critère n'est pas un quantificateur pour la mesure de la difficulté, mais il est contraint pour définir le domaine d'étude. Le temps d'exécution des algorithmes est borné par un polynôme qui dépend de la taille de l'instance.

Ces études sont purement théoriques. Elles permettent de prouver l'existence d'instances difficiles mais elle ne proposent pas, à notre connaissance, de méthode pour construire de telles instances. De plus, avec une définition de la difficulté d'instance différente des autres études rencontrées sur le sujet, ces travaux ne donnent a priori pas de piste pour l'étude empirique de la difficulté d'instance.

3.6 Autres études importantes

La notion de difficulté est abordée dans d'autres études, qui ne rentrent pas directement dans la classification précédente. Nous décrivons plus ou moins en détail les études selon leur originalité et l'importance qu'elles ont eu dans nos travaux.

Études théoriques

Les études de Papadimitriou and Steiglitz (1978) et Chvátal (1980) sont, à notre connaissance, les premières qui abordent la notion de difficulté d'instance pour un ensemble d'algorithmes. De précédentes études (Rosenkrantz et al. 1977) présentent déjà des instances particulières qui caractérisent le pire déroulement d'un seul algorithme donné. L'originalité des études de Papadimitriou and Steiglitz (1978) et Chvátal (1980) est de proposer des instances difficiles pour une classe d'algorithmes, d'un point de vue théorique. Pour cela, ils énoncent des propriétés éprouvantes pour la résolution des algorithmes qu'ils considèrent.

Papadimitriou and Steiglitz (1978) étudient les méthodes par recherche locale pour le problème du voyageur de commerce. Ils considèrent qu'une instance est difficile si elle a un grand nombre d'optima locaux pour la règle de voisinage étudiée et une solution optimale globale qui est bien meilleure. Une instance est alors globalement difficile si elle est difficile pour tout voisinage qui échange k arêtes, avec k assez grand, comparable à la taille de l'instance.

Chvátal (1980) étudie le problème du sac à dos pour les algorithmes exacts de type branchement ou programmation dynamique. Il énonce quatre propriétés concernant les instances pour qu'elles soient difficiles et montre que le temps de résolution de telles instances croît exponentiellement avec la racine carrée de la taille de l'instance.

Par la suite, d'autres études ont utilisé des approches analogues pour proposer de nouvelles instances difficiles pour des classes d'algorithmes. Par exemple, Kochetov and Ivanenko (2005) considèrent pour un algorithme de branchement qu'une mauvaise instance aura un large écart entre la valeur optimale et la valeur duale. En effet, cette valeur, qui est une borne de l'optimum, est généralement utilisée pour évaluer les nœuds dans l'arbre de branchement.

Dans ces études, le référentiel de la difficulté d'instance est une classe d'algorithmes et la portée est globale puisque les instances proposées représentent des situations où les algorithmes sont éprouvés. Le but de ces études est de caractériser la difficulté d'instance et de proposer de nouvelles instances qui sont difficiles.

Études empiriques

Plusieurs études proposent de nouvelles instances pour enrichir les bases de tests qui permettent d'évaluer les algorithmes. Ces études comparent souvent leurs nouvelles instances aux instances classiques. Bien que le cadre méthodologique ne soit pas directement explicité, le principe de comparer empiriquement des instances en utilisant des méthodes de résolution est commun à ces études. Nous décrivons succinctement ces études en précisant les idées qui nous ont semblé originales.

Drezner et al. (2005) présentent des classes d'instances pour le problème d'affectation quadratique et les testent avec plusieurs méthodes de résolution. Ils ont observé que des instances difficiles pour des métaheustiques parmi les plus performantes connues se résolvent assez facilement avec des méthodes exactes. Pisinger (2005) étudie les instances du problème du sac à dos, qui est souvent perçu comme un problème *NP*-difficile simple à résoudre. Il présente des instances difficiles, grâce à deux propriétés : de larges coefficients pour ralentir la programmation dynamique, ou de mauvaises bornes supérieures pour perturber les algorithmes de branchement.

Plusieurs études évaluent la difficulté d'instances aléatoires, en variant les lois de distribution. Classiquement, les données sont générées de manière uniforme. Pour changer, Osorio and Pinto (2003) utilisent une loi exponentielle pour le problème du sac à dos multidimensionnel, et Osorio and Cuaya (2005) pour le problème du voyageur de commerce. Ces deux études mesurent la difficulté d'instance grâce aux performances d'algorithmes exacts, représentées par le nombre de nœuds parcourus dans l'arbre de branchement. Dans Ridge and Kudenko (2007) et Ridge and Kudenko (2008), la loi de distribution étudiée est la loi log-normale pour le problème du voyageur de commerce, car elle s'approche plus des instances réelles. Les auteurs observent une corrélation entre l'écart-type des distances et la difficulté des instances. Leur étude présente un cadre expérimental intéressant pour évaluer les performances des algorithmes, que nous pourrons adapter à l'évaluation des instances.

Stützle and Fernandes (2004) proposent aussi de nouvelles classes d'instances aléatoires, dans le cadre du problème d'affection quadratique, mais avec des propriétés sur la structure. Ces instances permettent d'observer des liens entre ces propriétés et la difficulté des instances pour les algorithmes. Les propriétés étudiées proviennent en partie du domaine de l'analyse de paysage (section 3.3). Dans cette étude, la mesure de performance d'un algorithme correspond à son rang dans le classement des résultats retournés par l'ensemble des algorithmes. Fischer et al. (2005) étudient également la relation entre la structure et la difficulté des instances, pour le problème du voyageur de commerce. Ils partent d'instances fractales, décrites par Norman and Moscato (1995), ou de grilles, qui sont très typées mais assez simple à résoudre. Ils les perturbent aléatoirement en supprimant ou déplaçant certains sommets pour obtenir une instance plus difficile. La difficulté est mesurée par les performances d'une méthode exacte ou d'une méthode approchée très efficace, l'algorithme de Lin-Kernighan-Helsgaun. Pour ce dernier, comme il trouve souvent la solution optimale, la mesure de performance proposée par Fischer et al. (2005) est le pire temps d'exécution sur plusieurs appels, pondéré par la proportion de solutions non optimales obtenues.

Arthur and Frendewey (1988) valident empiriquement la difficulté de leur classe d'instances pour le problème du voyageur de commerce dont la solution optimale est connue par construction. Ils la comparent à une classe d'instances aléatoires. Leur mesure originale de la performance est donnée par des heuristiques gloutonnes. Pour chaque instance, elle correspond à la plus grande proportion d'arêtes en commun

entre la solution de l'heuristique et une solution optimale.

Dans ces études empiriques, le référentiel de la difficulté d'instance varie entre un ou plusieurs algorithmes. La portée toutefois est toujours relative, car les classes d'instances étudiées sont comparées à des instances classiques pour évaluer leur difficulté. Les mesures de difficulté sont données par les performances des algorithmes considérés. On retrouve les mesures habituelles, mais aussi des mesures plus originales comme le classement de l'algorithme ou la similarité de la solution avec l'optimum. Ces performances sont obtenues empiriquement, par l'exécution des algorithmes. La difficulté d'instance permet ainsi de valider l'utilisation des nouvelles classes d'instances proposées dans les études, car pertinentes pour la pratique expérimentale.

Conclusion

Dans ce chapitre, nous avons montré globalement la manière dont la difficulté d'instance est présente dans la littérature. Nous avons commencé par le concept d'instance et son utilisation dans les études empiriques. La manière dont les instances sont considérées est primordiale pour parler de difficulté d'instance. Dans ce contexte, on a vu qu'il est assez délicat d'étudier des instances individuelles. Les études font généralement plutôt référence à des classes d'instances. Une classe d'instances correspond à une mesure de probabilité sur l'ensemble des instances. Certaines hypothèses implicites accompagnent cette notion : une classe est un ensemble infini d'instances, mais dont on peut extraire un nombre fini d'instances représentatives pour généraliser leurs comportements à la classe. En pratique, pour obtenir de telles instances, il est nécessaire de définir un générateur qui caractérise la classe. Les instances ont des origines diverses, qui correspondent à des buts différents. Toutes ne sont donc pas appropriées pour l'étude empirique de la difficulté.

La notion de difficulté d'instance s'exprime différemment selon les études, mais elle représente toujours la capacité à éprouver la résolution. Nous avons donné un cadre pour décrire les études qui est composé du référentiel, de la portée, de la mesure et de l'utilisation de la difficulté dans l'étude. À partir de cela, nous avons présenté les différents types d'études qui utilisent la difficulté d'instance en les confrontant à notre approche. Malgré un domaine assez récent et encore peu unifié, nous apprécions la quantité de travaux existant autour de cette notion. Les propositions de Hooker (1995) et Rardin and Uzsoy (2001) sur la science de l'expérimentation semblent avoir reçu un écho à travers de plus en plus de travaux qui placent les instances au cœur de l'étude, pour justifier de leur utilisation. Cependant ces initiatives restent selon nous encore trop isolées et devraient s'inscrire davantage dans une théorie commune. Dans ce sens, nous présentons dans le chapitre suivant une méthodologie sur l'étude empirique de la difficulté d'instance.

Chapitre 2

Mesure de la difficulté d'instance

Nous présentons dans cette partie le cœur de nos travaux sur la difficulté d'instance. En nous appuyant sur les études détaillées dans l'état de l'art, nous définissons les notions clefs dont dépendent nos travaux. Nous expliquons ensuite notre méthodologie expérimentale pour étudier des classes d'instances de problèmes algorithmiques, qui repose sur la comparaison empirique de deux classes d'instances. Nous décrivons étape par étape la démarche à suivre.

1 Définitions

Dans l'état de l'art, nous avons identifié plusieurs notions importantes autour des instances et de la difficulté. Nous avons donné les représentations qui en sont faites dans la littérature, qui peuvent être assez différentes selon le domaine de recherche et le but de l'étude. Pour notre étude, nous nous plaçons dans le cadre d'études empiriques des instances dont le but est d'améliorer la pratique du benchmarking, en portant un regard critique sur le choix des instances employées. Nous donnons maintenant les définitions des notions telles qu'elles seront considérées dans la suite de la thèse.

1.1 Classe et générateur d'instances

La notion d'instance est bien définie pour des problèmes algorithmiques et est sujette à peu d'interprétations différentes. Une instance peut être vue comme une application numérique du problème pour laquelle un algorithme de résolution retourne une solution. Un problème contient une infinité d'instances.

Pour étudier des sous-ensembles d'instances, on utilise la notion de classe d'instances telle que l'a définie Birattari (2004b). Une classe d'instances est définie par

l'espace d'instances du problème muni d'une loi de probabilité. On peut donc voir une classe d'instances comme un sous-ensemble d'instances auxquelles sont associés des poids. Pour les classes d'instances utilisées en pratique, la loi de distribution des instances n'est généralement pas explicitement connue. De plus, cette formulation ne permet de représenter que des sous-ensembles d'instances dénombrables, ce qui n'est pas le cas, par exemple, de l'espace d'instances.

Cette définition probabiliste d'une classe d'instances répond au contexte pratique autour de l'utilisation d'instances qui repose sur des générateurs d'instances. On définit un générateur comme une procédure paramétrée qui retourne une instance. La construction de l'instance se fait suivant une méthode algorithmique qui dépend des paramètres d'entrée du générateur ou d'une source d'aléa. Par exemple, les graphes d'Erdös-Rényi, notés $G(n, p)$, sont des graphes de n sommets où, pour toute paire de sommets, l'arête est présente avec une probabilité p. Les instances obtenues pour un générateur d'instances forment une classe d'instances. Et, comme les instances utilisées en pratique dépendent d'un générateur, on considère qu'on peut aussi toujours associer un générateur à une classe d'instances. Le générateur est l'implémentation pratique de la classe.

Les paramètres des générateurs sont des grandeurs qui influent sur les instances générées. Ces valeurs peuvent porter sur les caractéristiques d'instances que nous avons présentées dans la section 1 du chapitre 1. Toutefois, nous identifions deux paramètres particuliers qui se différencient des autres par leur rôle. En premier, dans le cas de la génération aléatoire, pour des soucis de reproductibilité, les sources d'aléa des générateurs sont initialisées par une valeur arbitraire, appelée *graine*. Contrairement aux paramètres classiques, ce paramètre n'est pas corrélé à une caractéristique d'instance et sa valeur n'a pas de signification vis à vis du problème. Un autre paramètre particulier, et primordial, est la taille de l'instance. C'est une valeur intrinsèque à chaque générateur, qui représente la dimension des instances. Il s'agit de la quantité d'un des objets constituant l'instance. Pour l'exemple précédent des graphes d'Erdös-Rényi, d'après la définition du générateur, la taille est le nombre de sommets n du graphe. Il semble acquis dans le domaine que la taille est un critère de difficulté pour les méthodes de résolution dont la complexité est mesurée en fonction de la taille. Pour la pratique du benchmarking, il est donc primordial de pouvoir générer des instances de taille variable. On suppose que pour tout générateur, la taille est un paramètre et qu'on peut générer des instances d'une taille arbitrairement grande.

Birattari et al. (2006) ont énoncé des hypothèses sur les classes d'instances que nous supposons valables dans notre cadre d'étude. D'une part, le nombre d'instances dans une classe est supposé infini. Dans le cas contraire, un algorithme ad hoc pourrait résoudre toutes les instances de la classe grâce à une table de correspondance entre l'ensemble fini d'instances et leurs solutions. En pratique, on considère qu'une classe est infinie à partir du moment où le nombre d'instances est suffisamment grand pour ne pas être exprimable exhaustivement. D'autre part, on suppose que

les caractéristiques des instances d'une classe sont relativement semblables et que leur comportement est similaire. Ainsi, toute instance obtenue par le générateur est un représentant valide de la classe correspondante. On parle dans ce cas de classe *homogène*.

1.2 Critères de performance

Les méthodes de résolution sont évaluées par leurs performances. La performance est une mesure qui caractérise l'efficacité d'un algorithme sur une instance. Si les performances d'un algorithme sont hautes, cela signifie que l'algorithme est efficace. Et à l'inverse, des performances basses traduisent des difficultés de résolution.

Les performances sont mesurées par des *critères*. On obtient la valeur de ces mesures par l'étude du déroulement des algorithmes. Pour certaines méthodes de résolution, la performance peut être calculée théoriquement. En effet, dans la théorie de la complexité, on analyse les performances d'après le comportement des algorithmes dans le pire cas, c'est-à-dire quand la performance est la plus mauvaise. Mais pour analyser le comportement moyen d'un algorithme, on mesure généralement plutôt ces critères de manière empirique, en exécutant l'algorithme sous la forme d'un programme informatique et en observant son déroulement ou en analysant son résultat.

Il existe principalement deux types de critères de performance pour étudier les algorithmes. Ils portent soit sur le temps d'exécution de la méthode de résolution, soit sur la qualité des solutions retournées par la méthode. Les critères applicables dépendent de l'algorithme étudié. Dans la résolution de problèmes *NP*-difficiles, pour laquelle il n'existe pas à ce jour de méthode rapide et optimale, ce sont deux grandeurs généralement complémentaires. En améliorer une se fait au détriment de l'autre. De la même manière, on peut mentionner le compromis entre temps et espace, mais la mesure de l'espace n'est encore que peu étudiée dans notre domaine, car elle sort du spectre des problèmes *NP*.

La mesure du temps d'exécution est applicable à toute les méthodes de résolution, mais elle est particulièrement pertinente pour les méthodes exactes ou les méthodes approchées dont la durée d'exécution n'est pas bornée polynomialement avec la taille de l'instance, comme la recherche tabou ou le recuit simulé. La mesure porte sur le déroulement de l'algorithme. Les éléments mesurés sont le plus souvent la durée chronométrée de l'exécution ou le nombre d'appels à une sous-routine de l'algorithme. Dans ce cas, une mesure élevée du temps d'exécution représente une performance basse, car on attend d'un algorithme efficace qu'il résout rapidement le problème.

Dans le cas de la mesure de la qualité des solutions, on s'intéresse au résultat retourné par l'algorithme. Cette mesure est appliquée spécifiquement sur les méthodes approchées, pour lesquelles la solution obtenue peut être sous-optimale. On compare alors généralement la solution retournée à la solution optimale. La mesure classique

consiste à comparer la valeur des solutions. D'autres mesures peuvent également être employées, comme la similitude de la solution à une solution optimale. Aussi, la solution optimale n'est pas toujours connue. Dans ce cas, la valeur de la solution de l'algorithme peut être comparée à une borne du problème ou à la meilleure solution connue pour l'instance testée. Quand on compare des solutions, en règle générale, une mesure proche de zéro pour une différence (ou de un pour un ratio) est une bonne performance pour l'algorithme, et inversement des mesures éloignées de zéro (ou un) reflètent de mauvaises performances.

1.3 Difficulté d'instance

La difficulté d'instance est la notion centrale de nos travaux. Elle représente la capacité d'une instance à éprouver les algorithmes dans sa résolution. Nous nous appuyons sur la théorie de la complexité (Garey and Johnson 1979) pour en donner une première définition théorique, que nous adaptons ensuite pour utiliser la notion dans des études empiriques.

La théorie de la complexité repose sur l'étude au pire cas des méthodes de résolution. Ainsi, la complexité (difficulté) d'un algorithme est mesurée pour l'instance qui donne la plus mauvaise performance de l'algorithme. La conception d'algorithmes peut alors être vue comme la recherche d'un algorithme dont les performances dans le pire cas sont maximales.

Là où nous cherchions la meilleure méthode de résolution, nous cherchons, avec l'étude de la difficulté d'instance, la pire instance à résoudre. Ce changement de point de vue implique maintenant une étude au meilleur cas. La difficulté d'une classe d'instances est donnée par les performances de l'algorithme qui donne les meilleures performances pour la classe. Une classe d'instances est difficile si les performances du meilleur algorithme sont minimales.

Cependant ces définitions restent d'ordre théorique car elles ne sont pas applicables en pratique dans l'état. D'une part, la théorie de la complexité repose sur l'ensemble des instances d'un problème qui est fini pour une taille donnée [1] et dont on sait construire tous les éléments pour trouver le pire cas. Quant à la difficulté d'instance, elle repose sur le meilleur cas de l'ensemble des algorithmes de résolution du problème. Or cet ensemble n'est absolument pas caractérisé. D'autre part, la complexité d'un algorithme est donnée asymptotiquement avec la taille. Cela explique que l'étude de la difficulté d'instance porte sur des classes dont la taille des instances ne doit pas être bornée. Toutefois, il n'est pas possible en pratique de mesurer des performances asymptotiquement. En effet, même en multipliant les ressources matérielles, en une période donnée de calcul, le nombre d'expériences que l'on peut faire est toujours fini.

1. Dans le contexte de la théorie de la complexité, on considère la taille de l'encodage d'une instance. Si cette valeur est fixée, le nombre d'instances qu'on peut coder est borné.

De même que les algorithmes peuvent être évalués empiriquement, nous proposons une approche similaire pour les instances. L'étude empirique de méthodes de résolution consiste à appliquer les algorithmes sur un ensemble d'instances et à comparer leurs résultats. On appelle cette méthode le *benchmarking*. Pour évaluer empiriquement la difficulté d'instance, nous étudions les classes d'instances sur un ensemble d'algorithmes dont nous comparons les performances entre les classes d'instances.

Dans l'état de l'art, nous avons identifié un modèle pour classer les notions de la difficulté d'instance que nous avons rencontrées. Nous appliquons maintenant ce modèle pour décrire notre approche. Nous définissons la difficulté d'instance comme une mesure empirique qui représente la capacité d'une instance à éprouver les algorithmes dans sa résolution. Le référentiel de la difficulté est composé d'un ensemble de couples contenant un algorithme et un critère de performance, qui sert de base de test pour évaluer les instances. La portée de notre approche est relative, car la difficulté d'une classe d'instances est évaluée en comparaison à d'autres classes d'instances. En effet, sans élément de comparaison, il est délicat de juger de la facilité ou de la difficulté juste avec une simple mesure de performance. Enfin, l'étude d'une classe d'instances ne peut se faire asymptotiquement en pratique, car l'expérimentation porte sur un nombre fini d'instances de la classe qui sont des configurations précises du problème, avec une taille donnée. L'étude porte donc sur des classes dont la taille des instances est bornée. On appelle l'*intervalle d'étude* la plage des tailles d'instances testées.

Nous précisons dans les parties suivantes la mise en œuvre de notre approche et les utilisations qui peuvent en être faites.

2 Buts de l'étude

La méthodologie que nous proposons permet de mesurer la difficulté d'instance. Comme expliqué dans notre définition de la notion, la difficulté d'instance considérée porte sur des classes d'instances et repose sur une comparaison de classes d'après les performances d'un ensemble d'algorithmes de résolution. Étant donné que les algorithmes sont évalués empiriquement par la pratique du benchmarking, en testant les algorithmes sur un ensemble d'instances de test, notre approche peut être définie comme du benchmarking d'instances.

Le but de l'étude est de comparer deux classes d'instances entre elles pour identifier la plus difficile des deux. Cette approche semble assez réductrice dans l'étude de la difficulté d'instance, mais elle peut être vue comme une brique de base pour mener des études plus complexes. Nous présentons, dans le chapitre 3, des applications de cette méthode avec des buts plus généraux que comparer deux classes d'instances.

Parmi les études qui s'appuient sur cette méthodologie, on peut citer la validation

de l'utilisation d'une classe d'instances pour la pratique du benchmarking. Cette étude est très proche de l'approche de base. Elle consiste à comparer la classe de l'étude à la classe habituellement utilisée pour le benchmarking. Pour que la nouvelle classe soit pertinente, il ne faut pas qu'elle soit plus facile que la classe usuelle. Dans le cas d'un problème sans classes d'instances attitrées pour les études expérimentales, on peut valider l'utilisation d'une classe si elle est plus difficile que les autres classes d'instances connues pour le problème.

Une autre étude, plus complexe, consiste à caractériser la difficulté d'instance d'un problème. On s'inspire des protocoles expérimentaux des sciences de l'expérimentation, comme la biologie. Pour identifier les causes d'un phénomène, on ne fait varier qu'un seul paramètre pour évaluer son impact sur le phénomène étudié. Dans notre cas, l'hypothèse porte sur l'influence d'une caractéristique d'instance. On forme des classes d'instances pour différentes valeurs de la caractéristique, que l'on compare ensuite, grâce à notre méthode, leur difficulté d'instance.

3 Description de la méthode

Nous détaillons étape par étape la méthode pour comparer deux classes d'instances. Nous commençons par donner le déroulement global, puis nous expliquons chaque étape.

Méthode de comparaison de 2 classes d'instances

1. Choix des 2 classes d'instances à comparer
2. Définition du référentiel de l'étude
3. Précision de l'intervalle d'étude
4. Étude de la variabilité des mesures
5. Échantillonnage des classes d'instances
6. Exécutions et collecte des mesures d'évaluation
7. Compilation des résultats par classe d'instances
8. Agrégation des mesures des algorithmes
9. Visualisation des performances
10. Analyse des résultats

3.1 Classes d'instances de l'étude

La méthode que nous proposons permet de mener une étude sur deux classes d'instances. Son but, comme expliqué dans la section 2, est de comparer les deux

classes entre elles. D'après la définition donnée dans la section 1.1, les classes reposent sur un générateur d'instances afin de pouvoir générer autant d'instances que nécessaires pour l'étude. Pour être capable de comparer les instances issues des deux générateurs, on suppose qu'ils permettent de générer des instances de même taille.

3.2 Référentiel de l'étude

On définit le référentiel de l'étude qui permet de mesurer la difficulté des instances. Pour tester les instances, on s'appuie sur une base de test composée d'un ensemble d'algorithmes de résolution avec un critère de performance associé à chacun. Dans le cas de différents critères pour un même algorithme, chaque paire (algorithme, critère) compte pour un élément de l'ensemble.

Les algorithmes choisis ont un rôle crucial sur la pertinence de l'étude. Ce choix est à la charge de l'expérimentateur, selon le but de son étude. Si l'étude porte sur la difficulté d'instance pour un type d'algorithmes de résolution, l'expérimentateur sera avisé de choisir plusieurs algorithmes de ce type. On peut également utiliser cette méthode en ne l'appliquant qu'à un seul algorithme.

3.3 Intervalle d'étude

Comme vu dans la section 1.3, on ne peut pas mener, en pratique, d'études asymptotiques sur la taille des instances. La taille de la plus grande instance est toujours bornée. On définit l'intervalle d'étude sur la taille des instances, mais en spécifiant également une borne inférieure. Les bornes de l'étude doivent prendre en compte la réalisabilité de l'expérience.

Une borne inférieure trop petite risque de ne pas permettre une comparaison pertinente des classes d'instances, car les performances ne seraient pas significatives. Si la mesure porte sur le temps d'exécution, de petites instances risquent d'être résolues trop rapidement. Si la mesure porte sur la qualité de la solution, la résolution risque d'être trop évidente et de retourner toujours la solution optimale.

Une borne supérieure trop grande peut ne pas être viable pour l'étude expérimentale qui demanderait des efforts de calcul trop conséquents pour être applicables en temps raisonnable. Or la durée d'exécution pratique d'un algorithme sous forme de programme dépend de la plateforme sur laquelle il est exécutée. Les contraintes matérielles peuvent également être prises en compte au niveau du stockage de données. Une taille trop grande peut engendrer des instances difficilement manipulables en mémoire.

L'intervalle d'étude doit donc être précisé après la définition du référentiel. En effet, les bornes, dont la valeur choisie dépend des performances des algorithmes, doivent être ajustées selon les méthodes de résolution du référentiel. On observe aussi

que l'intervalle, surtout la borne supérieure, dépend des ressources informatiques de l'expérimentation, qui ne cessent de se perfectionner. Ainsi, l'intérêt d'une étude dépend de l'époque où elle a été menée car, avec les avancées technologiques, les résultats risquent de ne plus refléter la réalité des études futures.

À moins que l'intervalle d'étude ne se limite qu'à une seule valeur, il faut aussi spécifier des valeurs intermédiaires pour couvrir l'intervalle. Le nombre et le choix des tailles intermédiaires dépendent de l'étude et de la taille de l'intervalle. Pour avoir une bonne couverture de l'intervalle, on prend des valeurs à une distance régulière. Si les bornes sont du même ordre de grandeur, on choisit les valeurs intermédiaires uniformément. Si les bornes sont très éloignées, il peut être judicieux de prendre des valeurs suivant un facteur exponentiel, comme le propose McGeoch (2012) en doublant la taille des instances.

3.4 Étude de la variabilité

Il existe deux sources possibles de variabilité dans notre étude : le déroulement des algorithmes de résolution pour ceux qui contiennent une part d'aléa ; les générateurs d'instances qui caractérisent les classes d'instances. Dans les deux cas, la variabilité n'est généralement pas connue théoriquement. Il faut donc mener une étude empirique pour évaluer la manière de la prendre en compte dans l'étude.

Ces sources d'aléa peuvent entraîner une instabilité des résultats. Pour y remédier, il faut s'assurer que le nombre d'échantillons de test de l'étude soit suffisant pour bien estimer les performances de l'algorithme sur la classe d'instances : le nombre d'exécutions d'un algorithme de résolution et le nombre d'instances représentant la classe d'instances.

D'après Birattari (2004a), pour le benchmarking d'algorithmes, il est plus efficace de multiplier le nombre d'instances testées que le nombre d'exécutions par instance. En effet, comme on suppose que les classes d'instances sont homogènes, les sources d'aléa sont indépendantes. Ainsi, pour un nombre fixé d'exécutions d'un algorithme, augmenter la taille de l'échantillon d'instances donne un estimateur de même qualité pour les performances de l'algorithme, mais il permet d'améliorer l'estimateur pour la classe d'instances. Suivant le même raisonnement pour le benchmarking d'instances, il doit être plus efficace de multiplier le nombre d'instances testées que le nombre d'algorithmes exécutés par instance. En combinant les deux résultats, on obtient qu'il est préférable de générer une instance propre à chaque exécution d'un algorithme du référentiel. Toutefois, cette méthode pose des problèmes d'ordre pratique lorsque le critère de performance dépend de la solution optimale, qui peut être difficile à calculer. On préfère alors mutualiser l'effort du calcul de l'optimum pour comparer les solutions de plusieurs algorithmes. On étudie ainsi la variabilité de chaque algorithme sur les même échantillons de classe d'instances.

L'étude de la variabilité qu'on propose se fait comme suit. On considère un

algorithme de résolution qu'on applique sur une classe d'instances pour une taille donnée et on cherche le nombre d'instances à générer pour que les résultats de l'étude soient fiables à $X\%$ près. On observe l'impact d'ajouts de nouvelles instances sur la variabilité de l'échantillon. Si la nouvelle instance a des performances similaires aux précédentes instances, la variabilité devrait peu changer. Pour nous assurer de la stabilité de l'échantillon, on demande que la mesure soit stable pour plusieurs ajouts d'instances. En pratique, on génère de nouvelles instances de la classe, sur lesquelles on applique l'algorithme de résolution, et on mesure l'écart-type relatif de la mesure du critère de performance sur les instances générées. On arrête la génération lorsque, pour 3 nouvelles instances consécutives, l'écart-type relatif varie de moins de $X\%$. Quand la variation de l'écart-type est stabilisé, on compte le nombre d'instances générées pour connaître la taille de l'échantillon nécessaire à la mesure des performances de l'algorithme de résolution sur la classe d'instances.

Pour notre approche, on applique cette étude de variabilité à chaque algorithme du référentiel pour chaque classe d'instances. Pour ne pas appliquer l'étude sur chaque taille étudiée des classes d'instances, on suggère de ne faire l'étude que sur la borne supérieure de l'intervalle de l'étude. En effet, on peut s'attendre à ce que la variabilité augmente avec la taille des instances. Avec la borne supérieure, on se place alors dans le pire cas de l'étude.

3.5 Échantillons de test

L'étude empirique d'une classe d'instances repose sur des instances concrètes issues de cette classe, qu'on appelle *représentants* de la classe. Étant données les hypothèses d'homogénéité des classes d'instances, on peut appliquer la base de test aux représentants et généraliser les résultats obtenus à la classe qu'ils représentent.

Les tailles des instances générées correspondent à l'intervalle de l'étude spécifié précédemment. Pour chaque taille (bornes et valeurs intermédiaires) de l'intervalle, on génère le nombre d'instances maximum donné par l'étude de variabilité sur les algorithmes du référentiel pour la classe d'instances. On applique à cette étape la génération des instances pour les deux classes d'instances étudiées.

3.6 Exécutions et collecte des mesures

Les mesures qu'on analyse sont obtenues expérimentalement. À cette étape, on exécute les algorithmes sur les représentants des classes d'instances.

On applique les algorithmes de résolution sur les instances générées à l'étape précédentes, qui sont les représentants des classes d'instances. Comme la variabilité des algorithmes est prise en compte dans les échantillons, on n'applique qu'une seule fois chaque algorithme par représentant de classe. Le nombre de représentants résolus

par algorithme est donné par son étude de la variabilité sur la classe d'instances concernée. Pour chaque exécution d'un algorithme de résolution, on relève le (ou les) critère(s) de performance associé(s) dans le référentiel de l'étude.

Lorsque le critère de performance est la qualité de la solution, les exécutions peuvent se faire de manière distribuée sur différentes plateformes. Toutefois, dans le cas où le critère est le temps d'exécution chronométré, tous les appels d'un même algorithme doivent se faire sur des machines identiques (même configuration matérielle et logicielle) pour que les performances soient comparables.

Dans le cadre de la thèse, nous avons développé un logiciel qui permet de gérer plusieurs exécutions d'algorithmes sur différentes instances, avec prise en charge de la sauvegarde des résultats (annexe B). Ce logiciel s'appuie sur un fichier de description de l'expérimentation, qui permet une meilleure clarté des études expérimentales, en particulier pour la reproductibilité.

3.7 Compilation par classe d'instances

Les données brutes après expérimentation dépendent de quatre paramètres : la classe d'instance, la taille d'instance, le représentant de la classe et l'algorithme appliqué. Pour analyser les résultats de l'expérience, on regroupe les données pour obtenir une mesure de performance par taille et par classe d'instances.

La première étape consiste à abstraire les résultats expérimentaux obtenus par chaque algorithme sur les représentants pour se ramener à une mesure de l'algorithme pour la classe et la taille. Pour chaque algorithme de résolution et chaque classe d'instances, on fait la moyenne des mesures collectées sur les représentants de classe de même taille.

3.8 Agrégation des algorithmes de résolution

La seconde étape sur les données consiste à obtenir la *mesure de difficulté* d'instance de chaque classe d'instances pour chaque taille. Pour cela, on caractérise les mesures de performance des algorithmes par taille et par classe d'instances, afin d'avoir une mesure globale représentative du référentiel de l'étude.

Pour combiner les performances entre les algorithmes, qui peuvent avoir différents critères de performance ou des mesures avec des ordres de grandeur différents, on normalise les résultats de chaque algorithme, à taille fixée, par sa performance moyenne sur l'espace d'étude. En pratique, pour chaque algorithme et chaque classe, on divise la mesure de performance de l'algorithme sur la classe par la moyenne des mesures sur les deux classes. On appelle cette mesure la *mesure normalisée* de l'algorithme sur la classe. Une valeur de 1 traduit ainsi un comportement moyen.

On caractérise ensuite la mesure de difficulté d'instance par classe et par taille. Pour cela, on s'appuie sur les mesures normalisées des algorithmes, sur lesquelles on applique différentes mesures statistiques. En règle générale, on suggère de prendre les mesures classiques suivantes : la moyenne et l'écart-type pour donner une valeur représentative avec sa précision ; les valeurs minimales et maximales pour étudier l'existence d'un ordre strict de difficulté entre les classes d'instances. Mais, selon l'étude, d'autres valeurs peuvent être significatives, comme présenté dans la section suivante.

3.9 Visualisation des performances

Pour analyser les données, avoir une représentation visuelle est un support non négligeable. On différencie deux applications qui dépendent de l'intervalle d'étude et des tailles des instances de l'échantillon.

Tout d'abord, dans le cas d'un intervalle d'étude avec des valeurs intermédiaires régulières, on peut étudier l'évolution de la mesure de difficulté d'instance avec la taille. On présente alors les résultats dans un graphique avec une courbe pour chaque classe d'instances qui donne les mesures de difficulté de la classe en fonction de la taille d'instance. Généralement, on se contente d'afficher la moyenne des performances normalisées.

Autrement, on peut visualiser les résultats d'une classe d'instances pour une taille donnée. Cette représentation est plutôt adaptée lorsque l'intervalle ne contient qu'une seule valeur de taille. On utilise à ce moment, pour une taille d'instance donnée, le diagramme de Tukey (1977), appelé *boîtes à moustaches*. Elle permet de donner une représentation synthétique riche de la mesure de difficulté d'une classe. Il existe plusieurs variantes de ce diagramme. La figure 2.1 présente un exemple de boîte à moustaches.

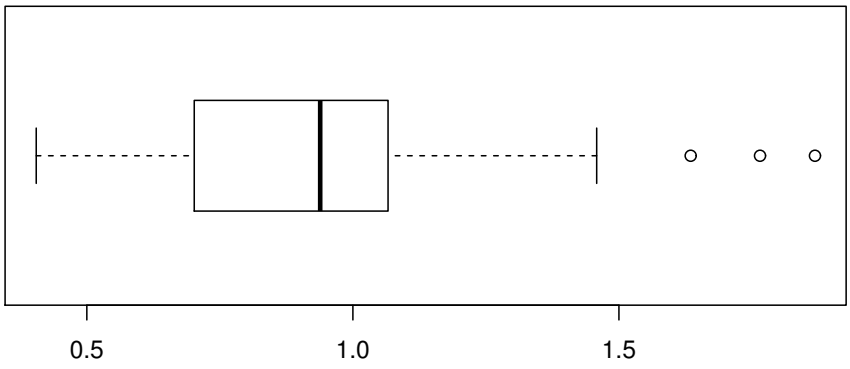

FIGURE 2.1: Exemple d'une boîte à moustaches avec des valeurs anormales.

Dans la représentation que nous utilisons des boîtes à moustaches, les limites de la boîte correspondent aux 1^{er} et $3^{ème}$ quartiles. La marque dans la boite caractérise la valeur médiane. Le bout des moustaches de part et d'autre de la boîte correspond au minimum (resp. maximum) entre la valeur maximale (resp. minimale) des mesures et $1,5$ fois l'écart interquartile (écart entre le 1^{er} et le $3^{ème}$ quartile) à partir de la boîte. Si c'est la seconde valeur qui est considérée, on représente par des cercles toutes les mesures de l'échantillon qui sont au-delà des moustaches.

3.10 Analyse des résultats

La dernière étape de la méthode consiste à répondre au but de l'étude, qui est de déterminer la classe d'instances la plus difficile.

Les performances agrégées représentent la difficulté des classes d'instances par rapport à l'étude menée. Étant donnée la manière dont la mesure a été construite, sa somme sur les deux classes est égale à 2 (ou la moyenne vaut 1). Ainsi des performances agrégées inférieures à 1 caractérisent une classe d'instances plus faciles, au contraire d'instances aux performances supérieures à 1 qui sont plus difficiles. La représentation graphique en fonction de la taille, de l'étude de deux classes d'instances donnera donc des courbes symétriques par rapport à l'axe horizontal d'ordonnée égale à 1.

À partir de la visualisation des performances, on peut comparer la difficulté d'une classe par rapport à l'autre dans le contexte d'étude. Des performances plus mauvaises caractérisent dans l'étude une instance plus difficile. S'il ne se dégage pas une tendance claire sur tout l'intervalle d'étude, on analyse l'intervalle par morceaux, voire à taille fixée.

Cependant, la mesure de difficulté n'a pas de signification en-dehors du contexte d'étude. Elle représente l'écart de performance des algorithmes du référentiel par rapport à leurs performances moyennes sur les classes de l'étude. Ainsi la mesure de comparaison est très dépendante de l'étude. Le rapport des mesures entre les classes donne simplement un ordre de grandeur de la différence de difficulté, mais qui est suffisant pour comparer les classes de l'étude entre elles.

Conclusion

Nous avons donné dans ce chapitre des définitions de plusieurs notions importantes pour nos travaux, en nous appuyant sur diverses études de la littérature, en particulier celles de Birattari (2004b). La notion probabiliste de classe d'instances permet de considérer les ensembles d'instances obtenus par des générateurs d'instances. On peut raisonnablement supposer qu'une classe d'instances a une cardinalité infinie et qu'il existe des instances de taille arbitrairement grande.

À partir de cette notion, nous avons défini la notion, centrale dans la thèse, de difficulté d'instance, qui mesure la capacité d'une instance à éprouver sa résolution. Nous avons donné une première définition théorique idéale, que nous avons ensuite adaptée pour en faire une mesure empirique qui repose sur les performances pratiques de méthodes de résolution.

Puis, nous avons présenté une méthode expérimentale pour mesurer cette difficulté d'instance en comparant deux classes d'instances par rapport à un ensemble d'algorithmes. Cette approche de benchmarking d'instances est expliquée en détail à travers les différentes étapes pour sa mise en application, avec, en particulier, des précisions sur les choix techniques pour prendre en compte la variabilité présente dans l'expérience. Les contraintes matérielles liées aux expérimentations font que l'intérêt des résultats est rattaché à l'époque où l'étude a été menée.

Cette méthode est une brique de base pour mener des études plus générales sur les classes d'instances. Nous avons identifié deux applications qui sont la validation de classes d'instances pour la pratique du benchmarking et l'identification de sources de difficulté grâce à un processus expérimental. Nous présentons dans le chapitre suivant des exemples de telles applications.

Chapitre 3

Difficulté d'instance pour le TSP

Nous présentons trois exemples d'application de la méthode expérimentale décrite dans le chapitre 2 pour évaluer des classes d'instances du problème de voyageur de commerce. Nous commençons par une application simple, qui consiste à valider l'utilisation d'une classe d'instances de la littérature sur une large base de méthodes de résolution. Ensuite, nous évaluons une nouvelle classe d'instances que nous comparons à une classe d'instances habituelle pour voir si on peut la lui substituer. Enfin, nous menons une étude plus poussée sur des sources possibles de difficulté en comparant des classes d'instances qui ont des lois de distribution différentes pour générer les distances et la présence ou non d'une structure.

1 Validation de la classe *Random Clustered*

Nous étudions une classe d'instances du TSP symétrique parfois utilisée dans la pratique du benchmarking pour générer des instances euclidiennes. Nous voulons vérifier la validité de cette classe par rapport à une autre classe d'instances usuelle, qui est aussi euclidienne. La particularité de l'étude est d'exploiter des résultats expérimentaux publiés dans une autre étude.

1.1 Classes d'instances de l'étude

Les classes d'instances que nous étudions représentent des instances euclidiennes, c'est-à-dire qu'on peut projeter le graphe dans un plan de telle sorte que le coût sur les arêtes soit donné par la norme euclidienne. Ces instances sont généralement décrites par la liste des coordonnées des sommets. Les deux classes étudiées sont présentées dans une revue de la littérature de Johnson and McGeoch (2002), théorique et empirique, sur les méthodes de résolution approchée pour le TSP, dans le

cadre du 8^{th} *DIMACS Implementation Challenge*[1]. Le générateur des instances des deux classes étudiées est disponibles sur le site du Challenge.

Random Clustered **(RC)**

Les instances de la classe sont *clusterisées*, c'est-à-dire que les sommets de l'instance sont regroupés en plusieurs paquets. Dans notre étude, nous considérons que les paquets ont une taille moyenne de 100 sommets. Les coordonnées des centres des paquets sont choisies uniformément dans l'intervalle $[0; 10^6]$. Les sommets sont placés à une distance aléatoire du centre de leur paquet suivant une loi normale de paramètre $\frac{10^6}{\sqrt{n}}$, avec n la taille de l'instance.

Random Euclidean **(RE)**

C'est sans doute la classe d'instances aléatoires euclidiennes la plus couramment utilisée. Les coordonnées des sommets sont choisies selon une loi uniforme. Dans cette étude, on les choisit sur l'intervalle $[0; 10^6]$.

1.2 Algorithmes du référentiel de l'étude

Le but du 8^{th} *DIMACS Implementation Challenge* était de présenter un état de l'art de toutes les méthodes de résolution existantes pour le problème du TSP, avec des résultats sur leur comportement en pratique. Les organisateurs ont appelé la communauté à leur fournir des algorithmes implémentés. Ils ont compilé les résultats de tous ces algorithmes sur un large jeu d'instances pour en faire une base de données consultable sur le site du Challenge. L'étude a été clôturée en 2002, mais reste une référence pour le domaine.

Le code des algorithmes de l'étude ne sont pas libres d'accès, mais on peut consulter leurs performances sur l'ensemble des instances de l'étude. Nous avons donc pu utiliser l'ensemble des algorithmes présentés dans la revue de la littérature. Nous avons comptabilisé 122 algorithmes approchés dont le comportement diffère sur les classes d'instances considérées. Nous n'allons pas les détailler ici. Il y a plusieurs types d'algorithmes différents : glouton, recherche locale, recuit simulé, recherche tabou, variantes de Lin-Kernighan. Johnson and McGeoch (2002) en donnent plus de détails dans leur article. Pour toutes ces méthodes qui sont approchées, nous prenons la qualité des solutions comme critère de performance. Plus précisément, nous considérons le ratio à l'optimum de la différence entre l'optimum et la solution de l'algorithme.

1. `http://www2.research.att.com/~dsj/chtsp/index.html`

1.3 Échantillons de l'étude et collecte des données

Étant donnée l'origine de nos données expérimentales, nous avons peu de marge de manœuvre pour faire varier la taille des instances testées, car nous devons nous adapter aux résultats disponibles dans la base de données. Notre étude se fait donc à taille fixée. Nous avons choisi une taille de 1000 sommets, qui correspond au plus grand échantillon d'instances testées pour les deux classes.

Dans le cas particulier de cette étude, nous menons une étude de la variabilité a posteriori. À partir des 10 instances testées, nous étudions la variabilité. Le nombre important d'instances testées rend difficile de déduire une mesure stable en prenant chaque algorithme individuellement, car certains sont très variables. On peut toutefois obtenir une mesure stable avec peu d'instances lorsqu'on considère la mesure agrégée qui est la moyenne des performances normalisées des algorithmes et qui est la mesure que nous utiliserons pour évaluer les classes d'instances.

On mesure l'évolution de la variabilité de la performance agrégée lorsqu'on ajoute la dixième instance. On obtient une variation de l'écart-type relatif d'environ 4,9% pour RE et de 5,5% pour RC. Si l'on regarde les trois derniers ajouts d'instance, la variabilité est assez stable pour RE, car l'écart-type relatif varie au plus de 6,2%. Mais c'est moins vrai pour RC, qui varie jusqu'à 12% sur les trois derniers ajouts. D'après la définition que nous avons donnée, l'étude est donc fiable à 12% près.

Nous allons utiliser toutes les données disponibles dans l'étude. Elles ont été obtenues par l'exécution des algorithmes sur les 10 représentants de chaque classe d'instances. La mesure de performance est l'écart à l'optimum de la valeur de la solution retournée par l'algorithme, normalisé par l'optimum.

1.4 Résultats et analyse

Nous suivons les étapes 7 et 8 de la méthode de comparaison pour obtenir la mesure de difficulté des deux classes pour l'étude. Pour chaque classe, nous faisons la moyenne des mesures de performances des algorithmes sur l'ensemble des représentants de la classe. Pour chaque algorithme, nous normalisons ces valeurs par la mesure moyenne de l'algorithme sur les deux classes RC et RE. À partir des performances normalisée, nous synthétisons la mesure de difficulté d'instance des classes. La mesure est résumée dans la table 3.1. Nous caractérisons aussi la difficulté des classes par la répartition des performances normalisées de chaque classe d'instances grâce à une boîte à moustaches (figure 3.1).

La moyenne des mesures normalisées de RC est égale à 1,07, soit 15% supérieure à celle de RE, avec un écart-type de 0,3. La plus petite mesure normalisée est pour RC et donc la plus grande pour RE. Les distributions des mesures normalisées couvrent approximativement le même intervalle, qui est très large, mais d'avantage

TABLE 3.1: Statistiques décrivant la difficulté des classes RC et RE à 1000 sommets pour l'ensemble des algorithmes présents dans la base de données du Challenge DIMACS.

	RC	RE
moyenne	1,07	0,93
écart-type	0,30	0,30
min	0,13	0,28
max	1,72	1,87

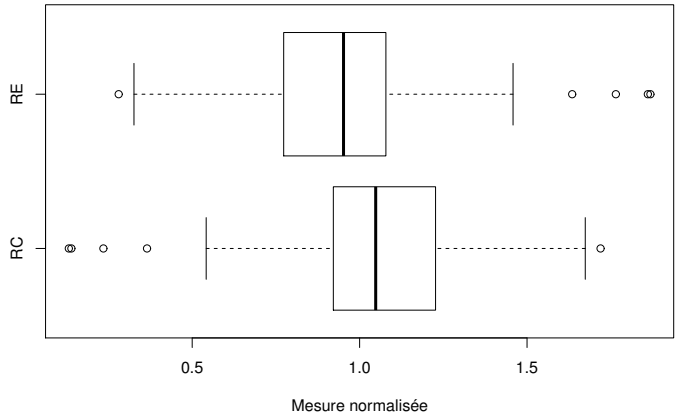

FIGURE 3.1: Boîtes à moustaches caractérisant la difficulté des classes RC et RE à 1000 sommets pour l'ensemble des algorithmes présents dans la base de données du Challenge DIMACS.

de mesures sont supérieures à 1 pour RC.

Les résultats indiquent des performances du référentiel en moyenne moins bonnes pour RC, mais avec une variabilité est assez forte. La répartition des performances confirme l'impression globale que RC est plus difficile que RE pour une majorité d'algorithmes. Toutefois les valeurs extrémales montrent qu'il n'y a pas d'ordre strict entre les classes.

Sur un référentiel composé d'une bonne part des algorithmes de la littérature, nous pouvons déduire que les deux classes d'instances RE et RC sont environ du même niveau de difficulté pour une taille de 1000 sommets. Elles peuvent toutefois donner des performances assez différentes pour un algorithme donné. On en déduit que la classe RC est un bon substitut à la classe RE comme base de test pour le benchmarking d'algorithmes, pour la taille d'instance étudiée.

2 Nouvelle classe d'instances : *Squared Norm*

Nous présentons une nouvelle classe d'instances pour le TSP symétrique dans le but de combler des lacunes dans le benchmarking concernant les instances non métriques.

2.1 Classes d'instances de l'étude

Les instances généralement utilisées dans le benchmarking sont des instances euclidiennes, décrites par les coordonnées des sommets et dont les distances sont calculées par la norme euclidienne. La seule classe d'instances usuelle à ne pas être euclidienne est *Random Distance* (RD), dont les instances sont décrites par une matrice des coûts, qui sont générés uniformément. Ainsi, les instances ne sont pas métriques.

Or les instances de RD nécessitent un codage dont la taille est quadratique avec le nombre de sommets, tandis que les instances euclidiennes, avec une liste de coordonnées, ont une taille linéaire avec le nombre de sommets. Dans la pratique, il en découle des limitations techniques qui ne permettent pas de générer des instances non métriques aussi grandes que des instances euclidiennes. Par exemple, des instances de 30 000 sommets font 3,7Go pour RD, contre 618Ko pour RE. Ceci empêche donc de tester des algorithmes classiques sur des instances non métriques assez grandes, qui ne tiendraient pas dans la mémoire vive, tel qu'observé dans l'étude de Johnson and McGeoch (1997). Or, du point de vue du temps d'exécution, il est possible de résoudre des instances de taille plus grande, comme c'est le cas pour les instances euclidiennes. Le facteur limitant n'est donc que le problème d'espace mémoire.

Pour remédier à ce problème, nous proposons une nouvelle classe d'instances dont le codage est linéaire avec le nombre de sommets, mais qui ne sont pas métriques. Pour cela, on décrit l'instance, comme pour les instances euclidiennes, par des coordonnées associées aux sommets. Cependant les distances entre les sommets sont maintenant données par une fonction dépendant des coordonnées des sommets mais qui ne vérifie pas l'inégalité triangulaire. Dans le cas de la nouvelle classe d'instances proposée, on prend le carré de la norme euclidienne, d'où le nom de la classe *Squared Norm* (SN). Cette classe permet donc de générer des instances non métriques beaucoup plus grandes que RD. Pour l'utiliser à la place de RD, nous devons vérifier si elle est au moins aussi difficile. Pour cela, nous allons appliquer la méthode pour comparer deux classes d'instances, présentée dans le chapitre précédent.

2.2 Algorithmes du référentiel de l'étude

La base de test est composé d'un ensemble d'algorithmes gloutons et la mesure de performance est la qualité des solutions. Nous prenons les algorithmes gloutons de l'étude menée par Johnson and McGeoch (1997) que nous citons précédemment. De plus, nous ajoutons des algorithmes gloutons de type insertion de sommets. Nous présentons dans les grandes lignes les algorithmes employés.

Nearest Neighbor (NN)

Cet algorithme a été proposé par Bellmore and Nemhauser (1968). Il construit une solution en parcourant le graphe, sommet après sommet. À chaque sommet, il suit l'arête de coût minimum qui mène vers un sommet qui n'a pas encore été visité, jusqu'à fermer le tour.

Greedy (GRE)

L'algorithme construit la solution en choisissant les arêtes une par une, suivant l'ordre croissant de leurs coûts. On s'assure que les arêtes choisies incidentes à un même sommet sont au plus 2 et qu'elles ne forment pas de sous-tour, jusqu'à prendre la dernière arête de la solution qui ferme le tour. En d'autre terme, on applique l'agorithme de Kruskal pour obtenir un arbre couvrant de poids minimum en ajoutant une contrainte sur l'arité des sommets qui peut valoir au plus 2, puis on ferme le tour avec l'arête restante.

Clarke-Wright (CW)

Cet algorithme a été originellement proposé par Clarke and Wright (1964) pour le problème de tournées de véhicules. On suppose qu'à partir d'un sommet d'origine o, on visite tous les autres sommets de l'instance. On cherche alors le moyen le moins coûteux pour passer d'un sommet à un autre sans repasser par le sommet d'origine. On calcule pour toute paire de sommets (i, j) l'économie qui peut être faite en passant directement de i à j : $c_{io} + c_{jo} - c_{ij}$. On applique ensuite la même méthode de construction que pour GRE, qui consiste à ajouter à la solution l'arête qui maximise l'économie sous condition de ne pas former de sous-tour, jusqu'à obtenir un tour complet en ajoutant le sommet d'origine.

Nearest Insertion (NI)

Cet algorithme a été proposé par Rosenkrantz et al. (1977). Le mécanisme général des méthodes par insertion de sommets est de partir d'un sous-tour restreint à un

seul sommet, puis à insérer un par un les autres sommets dans le sous-tour jusqu'à obtenir un tour complet. Lorsqu'on insère un sommet, on le place à la position qui minimise la longueur du nouveau sous-tour. Le choix du sommet inséré dépend de l'algorithme. Dans le cas de NI, on insère, à chaque étape, le sommet qui est le plus proche d'un sommet déjà dans le sous-tour.

***Farthest Insertion* (FI)**

L'algorithme est une méthode par insertion de sommets. Le sommet inséré est celui qui est le plus loin des sommets déjà insérés.

***Random Insertion* (RI)**

L'algorithme est aussi une méthode par insertion de sommets. Le sommet inséré est choisi arbitrairement parmi ceux qui ne sont pas encore dans le sous-tour.

Pour les algorithmes gloutons, le temps d'exécution est borné selon la taille de l'instance, puisqu'on construit la solution sans remettre en cause les choix précédents. Le critère de performance le plus pertinent pour mesurer l'efficacité des algorithmes est donc la qualité des solutions.

2.3 Échantillons de l'étude

L'intervalle de notre étude porte sur des tailles d'instance comprises entre 500 et 3000 sommets, avec un pas de 500. De telles instances en codage matriciel font déjà 38Mo et le temps de résolution optimale avec Concorde [1] peut être de plusieurs heures.

TABLE 3.2: Résultats des études de variabilité à 5% près pour les algorithmes gloutons sur les classes RD et SN.

	NN	GRE	CW	NI	FI	RI
RD	14	12	12	12	19	10
SN	11	21	14	14	12	13

Pour connaître le nombre d'instances à générer dans l'échantillon de chaque classe d'instances, nous menons une étude de variabilité avec une garantie de fiabilité à 5% près. Nous appliquons l'étude à chaque algorithme pour chaque classe d'instances

1. Concorde est une librairie de résolution pour le TSP symétrique (Applegate et al. 2007). Le cœur de l'outil repose sur une méthode exacte de résolution de type PLNE. Plusieurs méthodes de résolution approchée y sont aussi implémentées. `http://www.tsp.gatech.edu/concorde.html`

sur la borne supérieure de l'intervalle d'étude. La table 3.2 récapitule le nombre d'instances qui ont été générées pour obtenir une variabilité stable à moins de 5%.

Chaque résultat nous donne le nombre nécessaire d'exécutions d'un algorithme sur une classe d'instances pour obtenir des résultats fiables. Pour connaître le nombre d'instances à générer pour chaque classe d'instances, il faut donc prendre le maximum entre les algorithmes.

2.4 Exécutions

Nous appliquons chaque algorithme sur le nombre de représentants de classe donné par l'étude de la variabilité. Le critère de performance considéré dans l'étude pour les algorithmes approchés est la qualité des solutions. On mesure le ratio entre l'écart de la solution à l'optimum, et l'optimum.

Étant donné que nous ne mesurons que la qualité des solutions, il n'est pas utile de spécifier les caractéristiques matérielles des plateformes de calcul employées. Nous avons utilisé l'algorithme NN proposé par Concorde et l'algorithme GRE du Challenge DIMACS. Les autres algorithmes ont été développés par nous-même. L'optimum est obtenu grâce à la méthode exacte proposée par Concorde.

2.5 Collecte des données et résultats

Nous compilons les résultats obtenus pour chaque algorithme par la moyenne sur les représentants de chaque classe et de même taille. Nous normalisons ensuite cette valeur par la moyenne des valeurs sur les deux classes. Puis, nous faisons la moyenne des mesures normalisées entre les algorithmes pour obtenir une mesure de classe pour chaque taille. Nous présentons l'évolution de la mesure des classes avec la taille dans la figure 3.2.

La mesure de difficulté est assez stable. Elle vaut environ 1,5 pour RD et 0,5 pour SN. Le ratio entre RD et SN varie de $2,7$ à $3,3$, globalement en augmentant avec la taille sur l'intervalle étudié.

2.6 Analyse

L'étude montre clairement que RD est plus difficile que SN pour les algorithmes gloutons sur l'intervalle entre 500 et 3000 sommets. De plus, la tendance ne semble pas indiquer le contraire au-delà de l'intervalle, car le ratio entre RD et SN tend à croître avec la taille.

Notre étude ne permet donc pas de valider l'utilisation de la nouvelle classe SN pour l'application à des algorithmes gloutons. Toutefois, en calculant la valeur op-

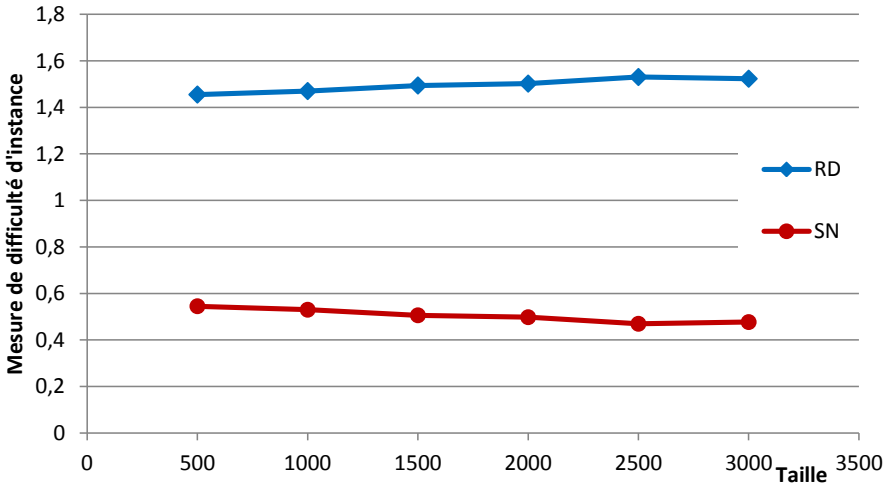

FIGURE 3.2: Évolution avec la taille de la difficulté d'instance des classes SN et RD pour des algorithmes gloutons sur l'intervalle entre 500 et 3000 sommets.

timale, nous avons observé un temps de résolution bien plus long pour SN que pour RD. Nous pourrions donc approfondir l'étude pour valider l'utilisation de SN dans ce référentiel ou avec d'autres types d'algorithmes approchés.

3 Études de sources de difficulté

Nous avons observé dans l'étude précédente une différence de difficulté entre RD et SN. Nous cherchons maintenant la source de cette difficulté. Les critères que nous avons identifiés et que nous allons étudier sont la loi de distribution des distances et la structure de l'instance, c'est-à-dire si les distances sont corrélées entre elles. Nous allons mener trois études comparatives. La première porte sur la loi de distribution, où nous allons comparer trois classes d'instances avec des lois différentes et sans structure. Les deux autres études portent sur la comparaison de deux paires de classes d'instances. Pour chaque paire, les classes ont la même distribution mais l'une a une structure et pas l'autre.

3.1 Classes d'instances étudiées

Nous utilisons les classes déjà présentées précédemment : RD, RE et SN. Mais nous proposons aussi de nouvelles classes pour satisfaire les besoins de l'étude.

On choisit les intervalles des lois de probabilité de telle sorte que les valeurs des solutions optimales sont du même ordre de grandeur entre les classes d'instances. Pour RD, les distances sont choisies uniformément sur l'intervalle $[0; 4.10^6]$. Les paramètres pour RE et SN sont les mêmes que pour les classes sans structure correspondantes.

RD avec la distribution de RE (reRD)

Pour chaque paire de sommets, nous exprimons le coût de l'arête, suivant la loi de distribution des distances de RE. Pour définir une valeur, on tire uniformément les coordonnées de deux points sur l'intervalle $[0; 4.10^6]$ et on calcule la distance euclidienne qui les sépare. On affecte cette valeur à l'arête.

Cette classe d'instances a donc bien la même loi de distribution des distances que RE, mais sans la structure, car les distances sont choisies indépendamment.

RD avec la distribution de SN (snRD)

On applique la même méthode que pour reRD, mais en suivant la loi de distribution de SN. Pour cela, on tire uniformément les coordonnées de deux points sur l'intervalle $[0; 10^4]$ et on calcule le carré de la distance euclidienne qui les sépare. On affecte cette valeur à l'arête. On obtient donc une classe d'instances avec la même loi de distribution des distances que SN, mais sans la structure.

3.2 Référentiel des études

Nous utilisons les même algorithmes que dans l'étude précédente, pour mesurer la difficulté d'instance vis à vis des algorithmes gloutons. La base de test contient les algorithmes *Nearest Neighbor* (NN), *Greedy* (GRE), *Clarke-Wright* (CW), *Nearest Insertion* (NI), *Farthest Insertion* (FI) et *Random Insertion* (RI).

Le critère de performance le plus pertinent pour mesurer l'efficacité des algorithmes est la qualité des solutions.

3.3 Échantillons des études

Comme pour l'étude précédente, nous prenons un intervalle compris entre 500 et 3000 sommets, avec un pas de 500.

On étend l'étude de la variabilité mené précédemment aux nouvelles classes d'instances considérées. La table 3.3 récapitule le nombre d'instances qui ont été générées pour obtenir une fiabilité à 5% près.

TABLE 3.3: Résultats des études de variabilité à 5% près pour les algorithmes gloutons sur les classes RD, RE, reRD, SN et snRD

	NN	GRE	CW	NI	FI	RI
RD	14	12	12	12	19	10
RE	16	13	13	14	14	11
reRD	16	11	17	13	13	13
SN	11	21	14	14	12	13
snRD	11	14	12	12	14	11

3.4 Exécutions et collecte des données

Nous appliquons les algorithmes sur les classes d'instances. Le nombre d'instances testées par chaque algorithme est donné par l'étude de la variabilité (table 3.3). Nous mesurons l'écart normalisé entre l'optimum et la valeur de la solution retournée par l'algorithme.

Nous agrégeons les résultats de chaque algorithme par classe d'instances, en faisant la moyenne sur les représentants de chaque classe. Ensuite, la mesure de difficulté de la classe dépend de l'étude qui est menée.

Dans le cas des deux études sur la structure, on est dans le cadre usuel de la méthode où on compare deux classes d'instances entre elles. On peut donc normaliser les performances de chaque algorithme, pour une taille donnée, par la moyenne des performances de l'algorithme sur les deux classes. La difficulté de chaque classe pour chaque taille est ensuite obtenue par la moyenne des performances normalisée.

Dans le cas de l'étude sur la loi de distribution, plutôt que de comparer les trois classes d'instances deux à deux, on peut les comparer toutes les trois simultanément. Pour cela, on normalise simplement la performances des algorithmes par leur performance moyenne sur les trois classes. La difficulté de chaque classe pour chaque taille est ensuite obtenue par la moyenne des performances normalisée.

3.5 Résultats

La première étude compare SN et snRD. L'évolution des performances est représentée dans la figure 3.3. Les mesures de difficulté sont relativement stables avec 1,6 pour snRD et 0,4 pour SN. Le ratio entre snRD et SN vaut entre $3,6$ et 4.

La seconde étude compare RE et reRD. L'évolution des performances est représentée dans la figure 3.4. Les mesures de difficulté valent environ 1,6 pour reRD et 0,4 pour RE. Le ratio entre reRD et RE varie entre 4 et $4,8$, en augmentant avec la taille.

La dernière étude compare RD, reRD et snRD. L'évolution des performances

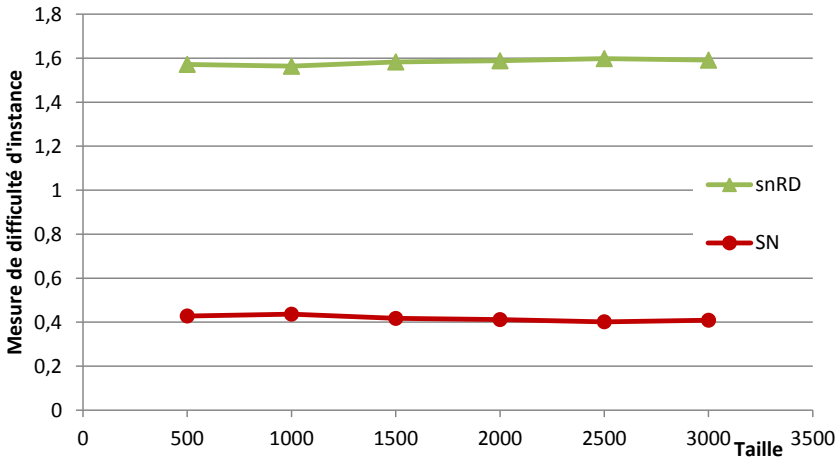

FIGURE 3.3: Évolution avec la taille de la difficulté d'instance des classes SN et snRD pour des algorithmes gloutons sur l'intervalle entre 500 et 3000 sommets.

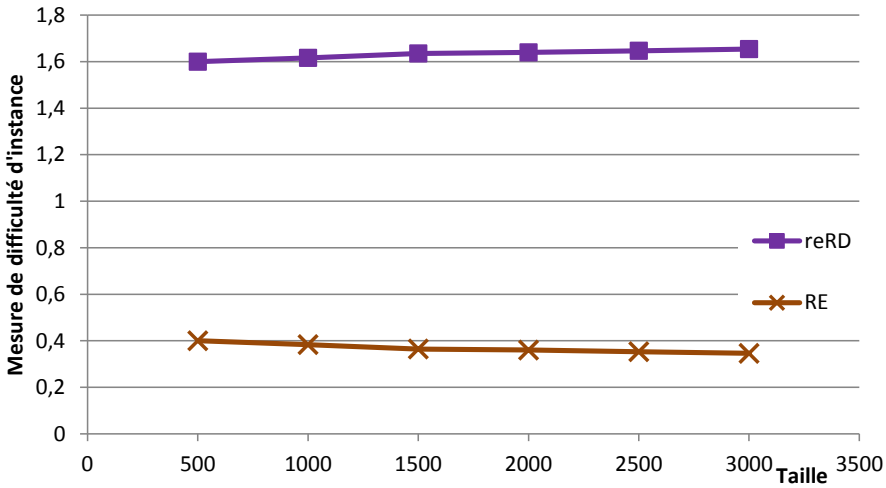

FIGURE 3.4: Évolution avec la taille de la difficulté d'instance des classes RE et reRD pour des algorithmes gloutons sur l'intervalle entre 500 et 3000 sommets.

est représentée dans la figure 3.5. Les mesures de difficulté valent environ 0,2 pour reRD, 1,2 pour RD et 1,6 pour snRD. Le ratio entre snRD et RD varie de 1,4 à 1,2 et celui entre snRD et reRD de 6 à 8. Entre snRD et RD, les performances tendent à se rapprocher avec la taille, au contraire des deux classes avec reRD, pour lesquels les ratios augmentent.

3.6 Analyse

Les résultats obtenus nous indiquent que les deux critères étudiés ont un impact non négligeable sur la difficulté des instances, pour le référentiel des algorithmes gloutons. Les résultats sont relativement stables sur l'intervalle d'étude, ce qui laisse supposer des tendances similaires pour des tailles d'instance plus grandes.

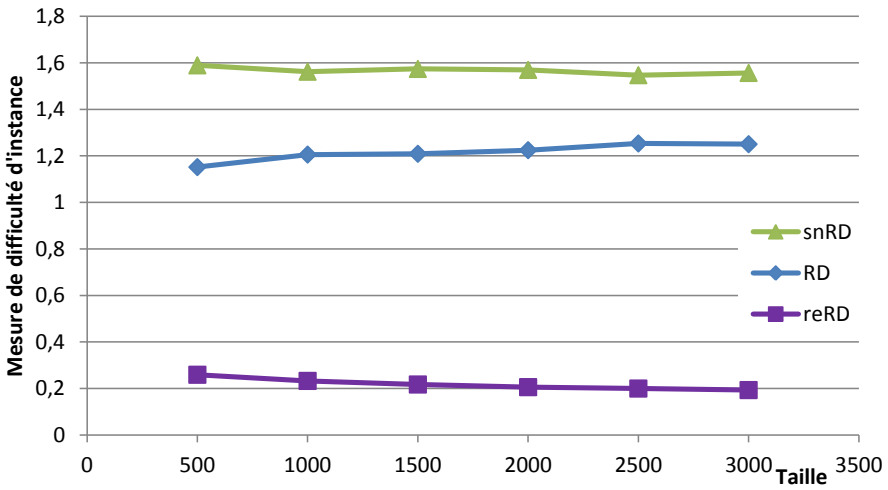

FIGURE 3.5: Évolution avec la taille de la difficulté d'instance des classes RD, reRD et snRD pour des algorithmes gloutons sur l'intervalle entre 500 et 3000 sommets.

L'absence de structure est une source de difficulté. Pour les distributions étudiées, on observe une différence de performance importante entre les classes structurées et leurs pendants sans structure. Étant donné le paradigme de conception des algorithmes gloutons, ce résultat est logique. Ces algorithmes sont généralement conçus plutôt pour des instances euclidiennes, exploitant la propriété que deux sommets proches d'un troisième sont aussi proches entre eux. Cette propriété repose sur l'inégalité triangulaire pour RE. Il est cependant intéressant de voir que pour SN, où l'inégalité triangulaire n'est pas respectée, les algorithmes gloutons donnent de bonnes performances, comme si la relation entre les distances leur suffit pour être efficaces.

Le loi de distribution des distances peut être une source de difficulté. On observe des écarts de performance entre les classes selon la loi de distribution des distances. L'étude nous a donné un classement entre des lois. La distribution selon RE donne une classe beaucoup plus facile que celles des deux autres distributions, parmi lesquelles la distribution selon SN est un peu plus difficile que la loi uniforme.

Il peut être intéressant d'approfondir l'étude en ajoutant plus de critères. Le calcul de distance, comme proposé pour SN, peut reposer sur d'autres règles. On peut utiliser d'autres fonctions simples comme la racine carrée ou l'inverse. Cela permettrait de vérifier s'il existe différentes formes de structure, mais aussi tester si d'autres distributions de distances peuvent être plus difficiles que celle de SN.

Conclusion

Ce chapitre nous a permis d'illustrer, par plusieurs applications, l'utilisation de la méthode d'évaluation de classes d'instances présentée dans le chapitre 2.

Tout d'abord, nous avons comparé deux classes d'instances présentes dans la littérature sur un large référentiel d'étude (section 1). Nous en avons déduit une utilisation sensiblement équivalente entre les classes *Random Clustered* et *Random Euclidean*, avec une difficulté un peu supérieure pour la première, pour un large ensemble d'algorithmes approchés avec le critère de qualité des solutions.

Nous avons ensuite mené une étude de validation d'une nouvelle classe d'instances pour représenter des instances non métriques dans un codage linéaire avec la taille (section 2). En la comparant à la classe usuelle *Random Distance*, nous avons constaté que *Squared Norm* est moins difficile pour les algorithmes gloutons en terme de qualité des solutions, ce qui ne nous a pas permis de valider son utilisation dans le contexte de l'étude.

Enfin, nous avons étudié la source de difficulté pour les algorithmes gloutons à travers plusieurs classes d'instances (section 3). Pour cela, nous avons comparé des classes d'instances telles qu'un seul paramètre varie entre elles. À travers trois comparaisons, nous avons déduit, pour l'intervalle étudié, que l'absence de structure dans les instances, c'est-à-dire l'absence de relation entre les distances d'arêtes adjacentes, est un critère majeur de difficulté pour les algorithmes gloutons. La loi de distribution des distances joue également un rôle non négligeable. Selon les lois de distributions comparées, on observe des impacts parfois plus importants que l'absence de structure. Il en ressort que la classe la plus difficile, dans le contexte de l'étude, est la classe de type *Random Distance*, avec la loi de distribution de *Squared Norm*.

Même si *Squared Norm* ne s'est pas avérée être un bon candidat pour les algorithmes gloutons, d'autres résultats nous laissent penser qu'elle est éprouvante pour les méthodes de résolution exacte comme Concorde ou des algorithmes approchés très efficaces comme Lin-Kernighan. Il serait donc intéressant d'étendre l'étude à d'autres algorithmes. De plus, en conservant le principe du codage par des coordonnées associées aux sommets, on pourrait proposer d'autres fonctions de calcul des distances pour tester de nouvelles classes d'instances non métriques ayant un codage linéaire.

Chapitre 4

Modification d'instances

La modification d'instances est une opération sur une instance qui renvoie une nouvelle instance, qui possède plusieurs éléments en commun avec l'instance d'origine. Par exemple, pour les problèmes sur les graphes, des modifications possibles sont des ajouts/suppressions de certains sommets ou arêtes (Fischer et al. 2005), ou la modification de poids sur le graphe (Ahammed and Moscato 2011). De manière générale, les modifications portent sur certaines données de l'instance, en conservant les autres identiques.

Ces modifications peuvent être classées selon trois objectifs. Tout d'abord, la principale motivation est généralement d'étayer une base de test en rajoutant de nouvelles instances pour le benchmarking. Les modifications qui dépendent d'un paramètre ou incluent de l'aléa sont particulièrement adaptées à cet objectif, car elles permettent de générer un ensemble infini d'instances. Rardin and Uzsoy (2001) proposent une méthode pour créer de nouvelles instances à partir d'instances existantes, dans le but d'enrichir les jeux de test issus du monde réel. Pour ce faire, ils suggèrent d'ajouter de petites variations dans les instances existantes afin de conserver la structure de l'instance initiale, mais en modifiant certaines valeurs numériques. C'est l'idée suivie par Greenberg (1991) pour perturber aléatoirement un programme linéaire tout en conservant la structure du modèle réel. Une autre application où la modification d'instances permet d'obtenir de nouvelles instances intéressantes pour le benchmarking concerne l'évolution d'instances. Les instances ainsi générées sont difficiles pour une méthode de résolution. Par exemple, Ahammed and Moscato (2011) modifient récursivement des instances du TSP en perturbant la position des villes, dans l'optique de faire évoluer les instances jusqu'à trouver des instances plus éprouvantes pour une méthode de résolution donnée.

La modification d'instances peut également aider à la résolution de problèmes. Pour certains problèmes, les instances peuvent être réduites par des pré-traitements en écartant les configurations triviales de la solution. On peut ainsi se ramener à une version épurée de l'instance qui représente le noyau difficile du problème à résoudre

(Cheeseman et al. 1991). Certaines réductions permettent même de se ramener à une nouvelle instance dont la taille est bornée indépendamment de la taille de l'instance de départ. La taille de l'instance réduite est alors bornée par une fonction qui peut dépendre d'un paramètre différent selon le problème. Dans ce cas-là, il s'agit de la classe particulière des problèmes, appelée *Fixed-Parameter Tractable* (Downey and Fellows 1999). Nous présentons plus en détail cette notion, ainsi que des travaux sur cette problématique dans le chapitre 5.

Enfin, générer de nouvelles instances peut permettre de tester la robustesse des algorithmes et de leurs solutions. Idée déjà présente chez Rardin and Uzsoy (2001), la modification des instances permet de perturber les instances de manière contrôlée pour évaluer l'évolution de la qualité des solutions obtenues face aux changements. Pour des applications précises, le paramétrage des algorithmes est amené à être optimisé empiriquement sur les jeux de données courants. La modification d'instances peut ainsi simuler une évolution des données pour évaluer la robustesse des solutions. Ce point ne sera pas davantage détaillé dans notre étude.

Dans nos travaux, nous considérons des modifications qui laissent invariante la structure de la solution optimale du problème initial ou, plus généralement, telles que l'on puisse rapidement calculer la solution de l'instance modifiée à partir de celle de l'instance initiale. D'une part, ces modifications permettent d'épurer les instances des configurations triviales pour se ramener au cœur de la difficulté du problème. D'autre part, ces modifications sont une source inépuisable pour générer de nouvelles instances. On peut en générer un nombre infini, en itérant si besoin. L'intérêt majeur d'une telle méthode consiste, à partir d'une instance dont la solution optimale est connue, de déduire simplement les solutions optimales des instances modifiées. La famille d'instances ainsi obtenue devient une large base de benchmarking sur laquelle les performances des méthodes de résolution pourront se comparer à la solution exacte, qui n'aura nécessité l'effort que d'une résolution optimale.

Dans la littérature, la modification d'instances est généralement associée à d'autres problématiques dont nos travaux ne découlent pas forcément. Un large pan de recherche, présenté par Liberatore (2004), existe autour de la résolution d'instances modifiées en s'appuyant sur la solution de l'instance initiale et la modification apportée. C'est une problématique naturellement rencontrée en pratique pour réajuster les solutions face aux aléas. Il y est généralement fait référence sous le terme de *ré-optimisation*. Pour de plus amples informations, nous redirigeons le lecteur vers les travaux de thèse de Berg (2008). Dans notre approche, les problèmes de ré-optimisation qui peuvent nous intéresser sont ceux qui sont faciles. En effet, dans ce cas, il existe une méthode rapide pour connaitre la solution du problème modifié. Ce peut être une piste pour élargir les champs d'application de notre approche.

Nous étudions, dans la suite, une modification d'instances pour le problème du voyageur de commerce, avec diverses applications. Nous commençons par présenter le problème avec ses sous-cas les plus courants, puis une modification d'instances pour ce problème qui laisse invariante la structure des solutions optimales entre les

instances. Nous verrons comment utiliser cette modification pour améliorer l'évaluation des performances de méthodes de résolution à travers un nouveau problème d'optimisation dont le but est de maximiser la somme des modifications appliquées sur une instance. Nous présentons ensuite certaines propriétés, en particulier vis à vis de la métricité de ces instances obtenues après plusieurs modifications. Enfin nous observerons l'impact de ces instances modifiées pour quelques méthodes de résolution.

1 Problème du voyageur de commerce

Le problème du voyageur de commerce ou *Traveling Salesman Problem* (TSP) est un problème classique en optimisation combinatoire. Il fait partie des premiers problèmes qui ont été montrés *NP*-difficiles par Garey and Johnson (1979).

Le problème est le suivant. Dans un digraphe complet $G = (V, E)$ à N sommets, avec des coûts c_{ij} positifs ou nuls [1] sur les arcs $ij \in E$, trouver un circuit hamiltonien de coût minimum. Il en existe plusieurs formulations sous forme de PLNE. Voici une formulation possible, où les variables $x_{ij} = 1$ si le tour contient l'arc entre les sommets i et j :

$$\begin{aligned} &\text{minimiser} \sum_{i \neq j \in V} c_{ij} \cdot x_{ij} \\ &\sum_{j \in V, j \neq i} x_{ij} = 1, \forall i \in V \\ &\sum_{i \in V, i \neq j} x_{ij} = 1, \forall j \in V \\ &\sum_{i \neq j \in S} x_{ij} \leq |S| - 1, \forall S \neq \emptyset, S \subsetneq V \\ &x_{ij} \in \{0, 1\}, \forall i \neq j \in V \end{aligned} \tag{4.1}$$

L'avant-dernière contrainte empêche des sous-tours dans tout sous-ensemble propre et non vide de sommets de V. Avec cette formulation, lorsqu'on relaxe les contraintes sur les sous-tours, en ajoutant des coût c_{ii} arbitrairement grand par rapport aux autres coûts, on se ramène au problème d'affectation simple (ou couplage parfait maximum) dans un graphe biparti complet. On construit le graphe orienté biparti complet $H = (A, B; E)$, où les sous-ensembles A et B représentent les sommets V de G, en mettant le coûts c_{ij} de G sur l'arc de H orienté de $i \in A$ vers $j \in B$. La valeur optimale du problème d'affectation donne une borne inférieure pour les solutions du TSP.

On observe différentes variantes du TSP. Nous en détaillons quelques unes. Si $\forall i, j \in V, c_{ij} = c_{ji}$, on parle de *TSP symétrique*, sinon de *TSP asymétrique*. Dans le cas du TSP symétrique, on se place alors dans un graphe non orienté. Dans le cas où les c_{ij} respectent l'inégalité triangulaire, i.e. $\forall i, j, k \in V, c_{ij} \leq c_{ik} + c_{kj}$, on parle

1. Pour le TSP, nous parlons indifféremment de *coût*, de *poids* ou de *distance*.

de *TSP métrique*.

La modification d'instances que nous présentons dans la suite s'applique localement à un sommet et modifie le coût des arcs/arêtes incidentes à ce sommet. Nous présentons une modification pour le problème asymétrique et symétrique, mais nous nous focaliserons davantage sur cette dernière version.

2 Modification d'instances *par offset*

Dans la pratique du benchmarking, on compare, si c'est possible, les performances des méthodes de résolution étudiées aux solutions optimales des instances de test. Des instances pratiques sont donc celles dont on obtient facilement la solution optimale ou, à défaut, le coût optimal (Arthur and Frendewey 1988). En effet, cela permet d'évaluer précisément et simplement la qualité des heuristiques testées. Avec les mêmes motivations, nous avons cherché à générer de nouvelles instances dont nous pourrions facilement déduire le tour optimal et son coût. Toutefois, ces instances sont souvent assez faciles à résoudre. Par exemple, les instances fractales de Moscato and Norman (1998) sont résolues optimalement par des méthodes gloutonnes. Pour obtenir des instances difficiles, nous proposons de faire l'effort de calculer la solution optimale d'une instance de départ, puis de la modifier sans changer le tour optimal. Dans ce cas, on dit que les instances sont équivalentes, car elles sont résolues par les mêmes tours optimaux.

Il est connu dans la littérature que les instances du TSP sont équivalentes lorsque l'on applique une fonction affine à toutes les distances entre les sommets (Birattari 2004b, Ridge and Kudenko 2008). Toutefois, nous n'en avons pas trouvé trace d'application. Nous avons adapté cette modification pour une application locale.

Dans le cas du TSP symétrique, on peut modifier localement les arêtes d'une instance autour d'un sommet v en conservant le même tour optimal. On soustrait une valeur α, positive ou négative, que nous appelons *offset*, au coût de toutes les arêtes incidentes au sommet v. Tout tour voit alors sa longueur diminuer de 2α, car, pour atteindre le sommet v, il est obligatoire de passer par deux arêtes incidentes au sommet.

À partir de cette modification locale, on peut généraliser la modification *par offset*, en l'appliquant à tous les sommets du graphe. L'instance initiale $I = (V, E)$ est un graphe complet non orienté avec des coûts c_{ij} sur les arêtes entre les sommets i et j, et α_i des poids associés aux sommets. On obtient l'instance modifiée telle que les coûts sur les arêtes ij valent $(c_{ij} - \alpha_i - \alpha_j)$. Le tour optimal est donc identique entre les deux instances. Toutefois sa longueur sur l'instance modifiée est diminuée de $2 \times \sum_{i \in V} \alpha_i$. L'opération est illustrée par un exemple présenté dans la figure 4.1.

On peut étendre la modification au TSP asymétrique. On considère pour chaque

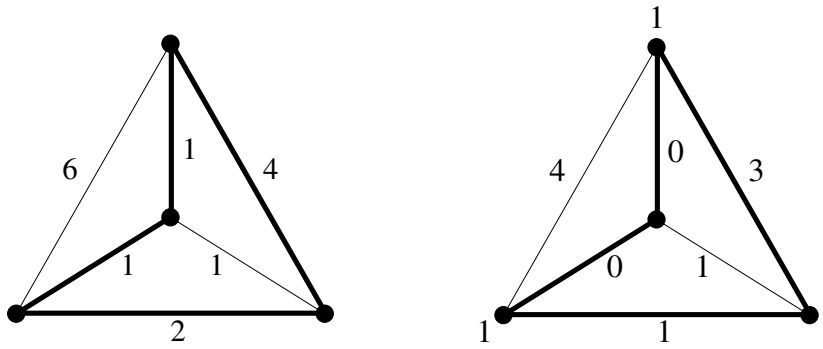

FIGURE 4.1: Exemple d'une modification d'instances : l'instance initiale (à gauche) est modifiée par deux offsets de valeur 1 (sommet d'en-haut et sommet de gauche) pour obtenir l'instance de droite. Le tour optimal, symbolisé par les arêtes en gras, est le même sur les deux instances, mais le coût change de 8 (initiale) à 4 (modifiée), soit une différence de deux fois la somme des offsets : $8 = 4 + 2 \times (1 + 1)$.

sommet une paire de poids (α_i^+, α_i^-), qui sont respectivement la valeur de l'offset pour les arcs sortants et entrants. Les coûts sur les arcs de l'instance modifiée deviennent alors $(c_{ij} - \alpha_i^+ - \alpha_j^-)$. Son tour optimal reste également inchangé et sa valeur est diminuée de $\sum_{i \in V} (\alpha_i^+ + \alpha_i^-)$.

Cette méthode permet, à partir d'un effort pour calculer la solution optimale de l'instance de départ, de générer autant d'instances que souhaitées, dont on connaît par construction le tour optimal et sa valeur. Toutefois, cette méthode de génération connaît des limites pour tester certaines classes d'algorithmes. En effet, la modification par offset conserve l'ordre entre les tours. Or certaines méthodes de résolution construisent des tours indépendamment de la structure de l'instance, puis comparent la qualité des tours obtenus pour conserver le meilleur tour, comme pour le parcours de voisinages successifs dans les algorithmes par recherche locale. Le déroulement de ces méthodes sera donc identique dans le parcours des tours, et la solution obtenue sera la même pour toutes les instances générées par modification d'une même instance initiale, sous réserve d'initialiser la recherche par la même solution.

3 Réduction d'instances

Nous avons vu que le déroulement et la solution retournée, pour certaines méthodes de résolution, sont invariants lorsque l'on modifie par offset des instances. Or ces modifications entraînent aussi une modification du coût de la solution optimale, qui peut avoir un impact sur l'évaluation des performances des méthodes de résolution. Ceci soulève donc un certain nombre de questions concernant la pertinence des outils de mesure de la difficulté employés. La performance des méthodes de résolution approchée est généralement mesurée par l'écart de la valeur de leurs

solutions à celle de la solution optimale. Pour comparer les résultats obtenus sur différentes instances, il est courant de normaliser cet écart. L'observation qui suit est vraie pour toute mesure qui est normalisée, *i.e.* où l'on divise par la longueur d'un tour. On définit un critère de mesure pour l'instance I par :

$$\epsilon(I) = \frac{SOL(I)}{OPT(I)}$$

avec $SOL(I)$ la longueur du tour obtenu par la méthode de résolution évaluée et $OPT(I)$ la solution optimale de I.

Or le comportement de certaines méthodes de résolution reste identique entre les instances modifiées et l'instance d'origine. Ainsi la valeur des solutions obtenues sur les instances modifiées évolue de la même manière que sur la solution optimale. On se retrouve avec un ratio qui varie selon la valeur des offsets appliqués :

$$\epsilon'(I, \alpha) = \frac{SOL(I) - \alpha}{OPT(I) - \alpha}$$

En particulier, avec de grands offsets négatifs, on peut faire tendre le ratio vers 1. Cette mesure que l'on peut faire varier artificiellement est problématique si on veut en faire un critère de difficulté.

Afin de remédier à cette variabilité artificielle, Zemel (1981) propose un autre critère de performance qui est invariant par modification affine des distances. Ce critère est l'écart à l'optimum normalisé par rapport à l'écart entre la pire et la meilleure solution :

$$\epsilon_Z(I, \alpha) = \frac{SOL(I) - OPT(I)}{WORST(I) - OPT(I)}$$

avec $WORST(I)$ la longueur maximum d'un tour de I. Or cette valeur n'est pas facile à obtenir, car le problème *Maximum TSP* est aussi difficile que *Minimum TSP* (Barvinok et al. 2004).

Birattari (2004b) a adapté cette méthode, pour rendre la mesure plus simple, en normalisant par l'écart moyen à l'optimum :

$$\epsilon_B(I, \alpha) = \frac{SOL(I) - OPT(I)}{RAND(I) - OPT(I)}$$

avec $RAND(I)$ la longueur moyenne d'un tour de I. Cette démarche est plus précise, puisqu'on connaît précisément la longueur moyenne d'un tour : $RAND(I) = \frac{2 \cdot \Sigma_{e \in E} c_e}{n-1}$.

Nous proposons un nouveau critère de mesure de la difficulté qui soit aussi stable vis à vis des modifications par offset. On cherche une modification d'instances par offset qui retourne une instance I_R remarquable telle que son ratio $\epsilon(I_R)$ soit maximal. Cela revient à minimiser la longueur du tour optimal de l'instance modifiée. La figure 4.2 présente un tel exemple pour le TSP symétrique. La méthode pour obtenir une telle instance modifiée est donnée dans la suite.

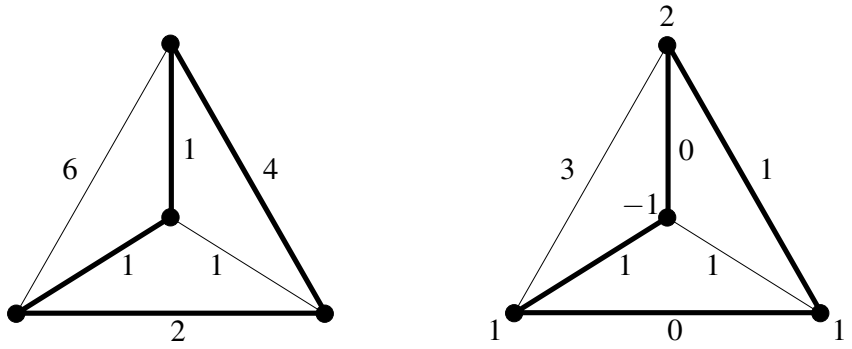

FIGURE 4.2: Exemple d'une modification d'instances par offset (sur les sommets) de l'instance initiale (à gauche) qui minimise la longueur du tour optimal (en gras) de l'instance modifiée (à droite).

On rappelle les conditions du TSP sur les coûts qui doivent être positifs ou nuls. Cela évite des solutions triviales consistant à soustraire le coût d'un tour quelconque, ce qui rendrait le coût du tour nul dans l'instance modifiée et le ratio maximum. Toutefois, une solution optimale de coût nul peut être rencontrée sur certaines instances après modification. Ce sont des instances pour lesquelles la valeur optimale est égale à la borne inférieure donnée par la solution du problème d'affectation associé. Or cette solution ne donne pas forcément une solution optimale du TSP, car la solution peut toujours contenir des sous-tours.

Comme chaque offset diminue la longueur du tour optimal, évalué sur l'instance modifiée, du double de la valeur de l'offset, minimiser la longueur du tour optimal revient à trouver la plus grande somme des offsets à appliquer à l'instance initiale pour obtenir une instance modifiée avec des coûts positifs sur les arêtes. Nous appelons ce problème la *réduction d'instances*. Nous en donnons la formulation dans les cas du TSP asymétrique, puis symétrique.

3.1 Pour le TSP asymétrique

Le problème de réduction d'instances pour le TSP asymétrique se ramène exactement au problème dual du problème d'affectation simple :

$$\begin{array}{c} \text{maximiser } \sum_{i \in V} (\alpha_i^+ + \alpha_i^-) \\ \alpha_i^+ + \alpha_j^- \leq c_{ij}, \forall i \neq j \in V \\ \alpha_i^+, \alpha_i^- \in \mathbb{R}, \forall i \in V \end{array} \tag{4.2}$$

En écrivant le dual, on retrouve la formulation primale du problème d'affectation :

$$\begin{gathered}\text{minimiser} \sum_{ij \in E} c_{ij} \cdot x_e \\ \sum_{j \in V, j \neq i} x_{ij} = 1, \forall i \in V \\ \sum_{i \in V, i \neq j} x_{ij} = 1, \forall j \in V \\ x_{ij} \geq 0, \forall i \neq j \in V\end{gathered} \tag{4.3}$$

Le problème de réduction d'instances revient alors à résoudre le problème dual du couplage parfait de poids minimum dans le graphe orienté biparti complet équivalent $H = (A, B; E)$ avec E orienté de A vers B. Pour le résoudre, nous avons une méthode algorithmique en temps polynomial qui est la méthode hongroise présentée par Kuhn (1955). Cette méthode qui s'appuie sur une approche primale-duale, résout les deux problèmes simultanément. Arbitrairement, on associe les poids α_i^+ à A et α_i^- à B, pour $i \in V$. L'étape finale de l'algorithme donne ainsi directement l'instance réduite.

3.2 Pour le TSP symétrique

Pour le TSP symétrique, le problème de réduction d'instances est plus contraint que pour le TSP asymétrique. En effet, les modifications successives doivent conserver la symétrie des instances. Ainsi, on ne peut se ramener au problème d'affectation. On applique la définition du problème de réduction d'instances au graphe complet (V, E), avec des coûts $(c_{ij})_{ij \in E}$ positifs ou nuls, de l'instance d'origine du TSP symétrique. Sa formulation sous forme de programme linéaire est la suivante :

$$\begin{gathered}\text{maximiser} \sum_{i \in V} \alpha_i \\ \alpha_i + \alpha_j \leq c_{ij}, \forall ij \in E \\ \alpha_i \in \mathbb{R}, \forall i \in V\end{gathered} \tag{4.4}$$

Le problème dual donne :

$$\begin{gathered}\text{minimiser} \sum_{ij \in E} c_{ij} \cdot x_{ij} \\ \sum_{ij \in E} x_{ij} = 1, \forall i \in V \\ x_{ij} \geq 0, \forall ij \in E\end{gathered} \tag{4.5}$$

Ce problème, modélisable sous forme de programme linéaire, peut évidemment se résoudre grâce au simplexe. Or nous reconnaissons dans le dual un problème remarquable qui est le problème du *couplage fractionnaire parfait de poids minimum.* Toutefois ce problème en tant que tel est à notre connaissance peu traité dans la littérature. Les seuls travaux que nous avons trouvés sont dus à Derigs and Metz

(1986), qui proposent une méthode de résolution. Ce problème est équivalent à chercher un couplage parfait de poids minimum dans un graphe biparti complet. Pour cela, ils proposent de transformer le graphe comme suit : diviser chaque sommet $v \in V$ en deux sommets v' et v'' ; diviser chaque arête $vw \in E$ en deux arêtes $v'w''$ et $v''w'$. Les coûts des arêtes du nouveau graphe sont donnés par $c_{v'w''} = c_{v''w'} = \frac{c_{vw}}{2}$. La solution du problème d'origine est alors donnée par $x_{vw} = \frac{(x_{v'w''}+x_{v''w'})}{2}$.

Ainsi, nous avons une méthode pour résoudre le problème dual. Toutefois, ce qui nous intéresse pour réduire une instance est la solution du problème primal. D'après le théorème de la dualité forte, on sait que si le problème dual admet une solution bornée, il existe une solution primale de même valeur. Le problème consiste alors, à partir d'une solution optimale duale, à déduire la solution optimale primale.

D'autres travaux portent sur le polytope associé à (4.5), qui est plus généralement appelé le polytope du *couplage fractionnaire parfait*. Ces études nous donnent des informations sur la forme des solutions de (4.5). Le principal résultat existant porte sur la propriété des sommets du polytope qui sont demi-entiers. Nous en donnons notre propre preuve, inspirée d'un résultat de Balinski (1965), rappelé dans Schrijver (2003), sur les sommets demi-entiers du polytope du couplage fractionnaire.

Théorème 4.1 *Tout sommet du polytope associé à* (4.5) *est demi-entier.*

Preuve : Soit x un sommet du polytope associé à (4.5). Les composantes entières de x satisfont déjà la proposition. Ainsi on se restreint à l'étude des valeurs non entières. Soit G_R le graphe réduit aux arêtes e telles que $0 < x_e < 1$ et aux sommets qui leur sont incidents. Pour que les contraintes soient vérifiées, les sommets de ce graphe sont de degré au moins 2.

Le nombre de variables non nulles est inférieur au nombre de contraintes, car les composantes non nulles de x sont des variables de base. Il y a $|E(G_R)|$ variables non nulles, par construction, et $|V(G_R)|$ contraintes. On en déduit que $|E(G_R)| \leq |V(G_R)|$. Or les sommets sont de degré au moins 2. Donc $|E(G_R)| = |V(G_R)|$ et G_R est 2-régulier.

Un tel graphe est une union de cycles disjoints. Comme x est un sommet du polytope, il ne peut y avoir de cycles pairs. En effet, un vecteur caractérisant un cycle pair avec des valeurs non entières n'est pas un sommet du polytope, car il peut s'exprimer comme une combinaison convexe de deux sommets du polytope, chacun caractérisant un cycle alterné de 0 et de 1 et complémentaires. Dans le cas des cycles impairs, on peut alterner les valeurs p et $1 - p$ $(0 < p < 1)$ sur le cycle. Mais on obtient forcément deux arêtes avec la même valeur incidente à un sommet commun, ce qui implique que $p = \frac{1}{2}$ et que les composantes non entières sont demi-entières. ■

Il en découle une caractérisation de l'existence de solutions pour le problème du couplage fractionnaire parfait, donnée par Scheinerman and Ullman (1997, Chapitre

2) et présentée dans le corollaire suivant.

Corollaire 4.2 *Un graphe G admet un couplage parfait fractionnaire, si et seulement si, il existe une partition $\{V_1, V_2, \ldots, V_k\}$ des sommets de $V(G)$, telle que, pour tout i, le graphe induit par V_i est soit K_2, soit un graphe hamiltonien avec un nombre impair de sommets.*

Ce résultat implique que si le graphe G admet un couplage parfait fractionnaire, il existe une solution x à tout problème de couplage parfait fractionnaire telle que le graphe obtenu par les arêtes $\{e \in E \mid x_e > 0\}$ soit une union disjointe d'arêtes isolées, données par $\{e \in E \mid x_e = 1\}$, et de cycles disjoints, donnés par $\{e \in E \mid x_e = \frac{1}{2}\}$, qui couvre tous les sommets de V.

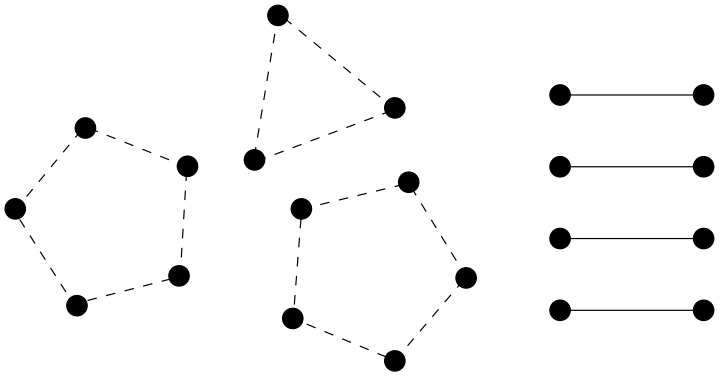

FIGURE 4.3: Caractérisation d'un couplage fractionnaire parfait : les arêtes en pointillé, qui valent $\frac{1}{2}$, forment des cycles impairs disjoints, et les arêtes continues, qui valent 1, sont des K_2 disjoints.

3.3 Propriétés des instances réduites pour le TSP symétrique

Une *instance réduite* est obtenue par une solution au problème de réduction d'instances, en appliquant les valeurs optimales des offsets à l'instance initiale. On dit qu'une instance réduite est la *réduction* de l'instance initiale. On en déduit qu'une instance est réduite si la valeur de la solution au problème de réduction associé est nulle. L'ensemble des instances réduites est noté $\mathcal{I}_R$.

Nous pouvons maintenant définir un critère de mesure stable d'une instance I :

$$\epsilon_R(I) = \frac{SOL(reduction(I))}{OPT(reduction(I))}$$

avec $reduction(I)$ une instance obtenue par réduction de l'instance I. Des instances déjà réduites seraient donc de bonnes candidates pour la pratique du benchmarking. On présente une caractérisation de telles instances.

Chaque sommet d'une instance réduite appartient à au moins une arête dont le coût est nul, c'est-à-dire :

Propriété 4.3 *Soit $I = (V, E)$ un graphe non orienté complet avec des coûts c_e sur les arêtes. $I \in \mathcal{I}_R \Rightarrow \forall i \in V, \exists j \in V, c_{ij} = 0$*

Preuve : On démontre le résultat par contraposée. Si un sommet de I n'a aucune arête incidente de coût nul, alors on peut soustraire un offset, correspondant au coût de la plus petite arête incidente, à toutes ses arêtes incidentes. Ceci permet de réduire la longueur du tour optimal. Ainsi I n'est pas une instance réduite. ■

On remarque que la réciproque est fausse. Une réduction minimale n'est pas une réduction minimum. En réappliquant des offsets négatifs sur certains sommets, on peut améliorer la valeur de réduction totale. La figure 4.4 présente un contre-exemple de la réciproque.

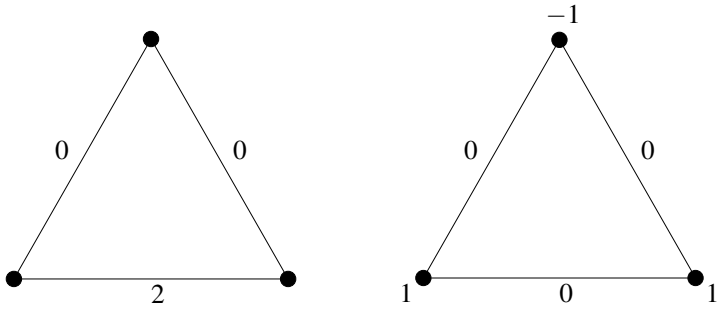

FIGURE 4.4: Exemple d'une instance (à gauche) où tous les sommets appartiennent à au moins une arête de coût nul, mais qui n'est pas une instance réduite (à droite).

On peut obtenir un résultat similaire pour le TSP asymétrique, où chaque sommet est incident à au moins une arête entrante et une arête sortante de coût nul.

Grâce au corollaire 4.2, on déduit une propriété plus générale qui caractérise la forme d'une instance réduite.

Propriété 4.4 *Soit $I = (V, E)$ un graphe non orienté complet avec des coûts c_e sur les arêtes. I est une instance réduite, si et seulement si le graphe induit par $\{e \in E \mid c_e = 0\}$ possède une partition des sommets de V en sous-ensembles qui induisent des K_2 ou des graphes hamiltoniens avec un nombre impair de sommets.*

Preuve : Montrons tout d'abord qu'une instance réduite est de la forme donnée dans la propriété. Une instance réduite est obtenue en résolvant le problème (4.4), dual de (4.5). Comme le graphe complet est hamiltonien, il admet toujours un

couplage parfait fractionnaire et on peut appliquer le corollaire 4.2 pour obtenir une solution du problème dual. En appliquant le théorème des écarts complémentaires, on en déduit que toute arête appartenant au couplage a un coût nul dans l'instance réduite, d'où la partition des arêtes de coût nul de l'instance réduite, donnée dans la propriété. De plus, comme le couplage parfait est incident à tous les sommets, on en déduit une autre preuve pour la propriété 4.3.

Pour la réciproque, on montre qu'en appliquant le problème de réduction à une instance de ce type, on trouve une solution de valeur zéro, qui est par définition une instance réduite. Soit $\{C_1, \ldots, C_k, e_1, \ldots, e_l\}$ une partition des sommets de G en cycles impairs (C_i) et en arêtes isolées (e_i), dont les arêtes ont un coût nul. Or, quand on somme les contraintes du programme linéaire (4.4) qui portent sur les arêtes d'un cycle C_i, on obtient $\sum_{j \in C_i} x_j = 0$, et pour les arêtes isolées e_i, $\sum_{j \in e_i} x_j = 0$. Comme une partition est un ensemble de sous-ensembles disjoints dont l'union est V, $\sum_{i \in V} x_i = 0$. ■

4 Instances modifiées et métricité

Dans cette partie, nous considérons des instances du TSP symétriques. La métricité est un critère fort pour la résolution du TSP. En effet, dans ce cas, il existe des algorithmes d'approximation qu'on ne trouve pas dans le cas général. Nous présentons des relations entre la modification par offset et la métricité des instances. Tout d'abord, nous étudions les conditions sur les offsets pour rendre une instance métrique, ce qui nous permet de proposer un nouvel algorithme d'approximation pour le TSP, dont les garanties sont indépendantes du nombre de sommets. Puis nous énonçons une caractérisation des instances réduites qui sont métriques.

4.1 Modification par offset et métricité des instances

La modification par offset peut impacter la métricité d'une instances. Écrivons l'inégalité triangulaire sur le triangle (i, j, k) :

$$c_{ij} \leq c_{ik} + c_{kj}$$

On applique un offset α sur un sommet. Les différents cas sont présentés dans la figure 4.5.

Soit le sommet concerné est sur la base du triangle (i ou j), alors l'inégalité devient après modification :

$$c_{ij} - \alpha \leq c_{ik} + c_{kj} - \alpha$$

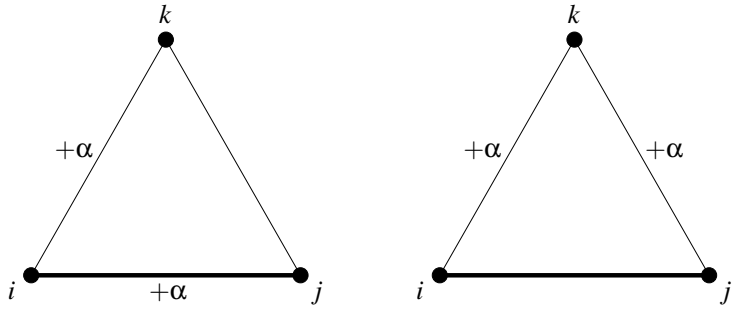

FIGURE 4.5: À gauche, modification par un offset (α) sur un sommet de la base (en gras) du triangle ; à droite, modification sur le sommet en-dehors de la base.

Dans ce cas-ci, quel que soit l'offset, l'inégalité est toujours respectée.

Soit le sommet concerné est le sommet k du triangle et alors l'inégalité devient après modification :

$$c_{ij} \leq c_{ik} + c_{kj} - 2 \cdot \alpha$$

L'inégalité reste vraie après modification par tout offset inférieur à $\frac{c_{ik}+c_{kj}-c_{ij}}{2}$, en particulier s'il est négatif. L'instance reste métrique si cette condition est vérifiée pour chaque triplet de sommets et elle ne l'est plus si, pour un triplet de sommets (i, j, k), on applique un offset supérieur à $\frac{c_{ik}+c_{kj}-c_{ij}}{2}$ au sommet k.

De manière générale, toute instance métrique reste métrique par modification par offset négatif, tandis que toute instance non métrique reste non métrique par offset positif. Toutefois, il est possible, pour toute instance qui n'est pas métrique, de la rendre métrique avec une modification par offset adéquate.

Dans le cas du TSP métrique, c'est-à-dire où l'inégalité triangulaire est respectée, il existe des algorithmes à garantie constante de performance. Les deux principales méthodes sont la 2-approximation en doublant l'arbre couvrant de poids minimum et la $\frac{3}{2}$-approximation de l'algorithme de Christofides (1976). Toutefois, le cas général du TSP n'est pas approximable à une constante près. Nous proposons une garantie de performance indépendante du nombre de sommets de l'instance, mais qui dépend d'autres grandeurs de l'instance, comme le coût maximum $c_{max} = \max\{c_{ij} \mid ij \in E\}$ et le coût minimum $c_{min} = \min\{c_{ij} \mid ij \in E\}$. On suppose ici que $\forall ij \in E, c_{ij} > 0$, pour être sûrs de ne jamais diviser par zéro.

Théorème 4.5 *Soit un algorithme d'approximation de facteur ρ pour le TSP métrique. Le TSP symétrique est alors $(1 + (\rho - 1) \cdot (\frac{c_{max}}{c_{min}} - 1))$-approximable.*

Preuve : Cette affirmation est démontrée par un algorithme qui garantit la valeur de la solution qu'il fournit. Il consiste tout d'abord à modifier par offset l'instance en une instance métrique, puis à appliquer l'algorithme d'approximation pour les instances métriques.

Pour tout $i, j, k \in V$ tel que $c_{ij} > c_{ik} + c_{kj}$, il faut au moins appliquer un offset de $\frac{1}{2} \cdot (c_{ik} + c_{kj} - c_{ij})$ aux trois sommets pour que l'inégalité triangulaire soit respectée entre eux dans l'instance modifiée. Or $\forall ij \in E, c_{min} \leq c_{ij} \leq c_{max}$. Pour tous les sommets, on peut minorer l'offset par $\frac{2 \cdot c_{min} - c_{max}}{2}$. Si l'instance n'est pas métrique, on applique un offset de cette valeur strictement négative à tous les sommets. La solution optimale (l'ordre des sommets) pour cette instance transformée reste la même que celle de l'instance de départ, mais sa valeur est augmentée de $n \times (2 \cdot c_{min} - c_{max})$.

On appelle I l'instance initiale et J l'instance modifiée métrique. Soit $OPT(I)$, resp. $OPT(J)$, la longueur du tour optimal pour l'instance I, resp. J. Soit $APP(J)$, la longueur du tour obtenu par l'algorithme d'approximation pour l'instance J. Soit $ALG(I)$, la longueur du même tour mais évalué pour l'instance I.

$$OPT(J) = OPT(I) + n \times (c_{max} - 2 \cdot c_{min})$$
$$APP(J) = ALG(I) + n \times (c_{max} - 2 \cdot c_{min})$$

D'après la garantie de performance de l'algorithme d'approximation, on sait que $\frac{APP(J)}{OPT(J)} \leq \rho$. On en déduit :

$$ALG(I) \leq \rho \cdot OPT(I) + n \times (\rho - 1) \cdot (c_{max} - 2 \cdot c_{min})$$

Or $OPT(I) \geq n \cdot c_{min}$. Donc :

$$\frac{ALG(I)}{OPT(I)} \leq 1 + (\rho - 1) \cdot \left(\frac{c_{max}}{c_{min}} - 1\right)$$

■

Sans améliorer la garantie de performance, car le pire cas reste le même, on peut peaufiner l'algorithme pour obtenir en moyenne de meilleurs résultats. Plutôt que d'ajouter sur tous les triangles un offset qui résout le pire cas, on peut appliquer des offsets différents sur chaque sommet de telle sorte que l'instance devienne métrique et que la somme des offsets appliqués soit minimum. On peut pousser le perfectionnisme jusqu'à ajouter des offsets négatifs quand l'inégalité est large pour se ramener à une instance métrique dont le tour est minimum par modification par offset, ce qui permet encore d'améliorer les performances de l'algorithme d'approximation.

En appliquant le théorème 4.5 avec l'algorithme de Christofides, qui a la meilleure garantie de performance connue pour le TSP métrique, où $\rho = \frac{3}{2}$, on obtient :

Corollaire 4.6 *Le TSP symétrique est* $\left(\frac{1}{2} \cdot (1 + \frac{c_{max}}{c_{min}})\right)$*-approximable.*

Les résultats donnés ne remettent pas en cause la non-approximabilité du TSP dans le cas général. Un problème est considéré comme approximable s'il existe un

algorithme avec un facteur d'approximation constant entre la valeur retournée et l'optimum. Or, même si, dans notre cas, la garantie de performance est indépendante du nombre de sommets, elle dépend de données du problème avec les coûts maximum et minimum sur les arêtes.

4.2 Instances réduites métriques

Nous avons vu comment rendre une instance métrique en augmentant les distances. On étudie maintenant l'évolution de la métricité lorsque l'on diminue les distances. Les instances réduites sont telles que la somme des offsets appliqués à l'instance initiale est la plus grande possible. Or cette somme est toujours positive. Les instances métriques n'ont donc aucune garantie de rester métrique après réduction. Malgré cela, existe-t-il des instances réduites qui sont métriques et si oui, quelle est leur forme ?

Nous utiliserons les notations suivantes. Soient $I = (V, E)$ un graphe complet non orienté avec des coûts positifs c_e sur les arêtes,

$$E^0 = \{e \in E \mid c_e = 0\} \text{ et } I^0 = (V, E^0),$$
$$E^* = \{e \in E \mid c_e > 0\} \text{ et } I^* = (V, E^*).$$

On remarque que I^0 et I^* sont complémentaires.

Théorème 4.7 *L'instance I est réduite et métrique si et seulement si toutes les propositions suivantes sont vraies.*

1. *I^0 est une union de cliques disjointes de taille au moins deux,*
2. *les arêtes incidentes à deux mêmes cliques de I^0 ont le même coût,*
3. *l'inégalité triangulaire est respectée pour tout triplet de sommets pris dans trois cliques distinctes de I^0.*

Preuve : Nous allons faire la démonstration en cinq étapes : d'abord montrer que les arêtes de coût nul couvrent les sommets de l'instance réduite par des sous-cliques disjointes de taille au moins deux, puis que les arêtes restantes, celles de coût non nul, forment une seule composante connexe, qu'elles ont le même coût entre deux mêmes parties et que l'inégalité triangulaire est respectée entre plusieurs parties, et enfin l'équivalence.

D'après la propriété 4.3, tous les sommets de I sont incidents à une arête de coût nul, donc I^0 est le graphe induit par E^0. De plus, d'après l'inégalité triangulaire, si deux arêtes d'un triangle sont de coût nul, la troisième l'est aussi. Ainsi tout sous-ensemble de sommets connexe dans I^0 forme une clique. On en déduit que E^0 couvre V par un ensemble de cliques disjointes (*point 1*).

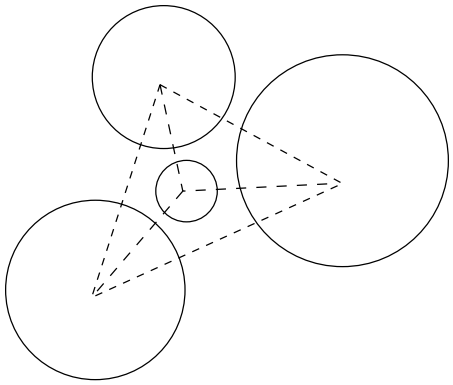

FIGURE 4.6: Caractérisation d'une instance réduite métrique. Les cercles représentant des cliques de coût nul partitionnent les sommets du graphe. Toutes les arêtes entre deux cliques, représentées par une arête en pointillé, sont de même coût. Le graphe formé par les cercles et les arêtes en pointillé est métrique.

Le résultat précédent implique que le graphe I^*, restreint aux arêtes de coût non nul, est connexe ou sans arêtes. En effet, s'il n'est pas connexe, alors le graphe complémentaire, qui est I^0, contient une coupe du graphe complet I, c'est-à-dire toutes les arêtes entre deux ensembles partitionnant les sommets. Ceci représente un ensemble connexe contenant tous les sommets de I. Or toute composante connexe de I^0 est une clique. Dans ce cas, I^0 est alors tout le graphe I et I^* n'a pas d'arêtes.

D'après la propriété 4.3, toute arête de coût non nul est adjacente à au moins une arête de coût nul. Or, d'après l'inégalité triangulaire, dans un triangle, si une arête a un coût nul, le coût des deux autres arêtes est égal. Ainsi, toutes les arêtes entre deux parties ont le même coût. De proche en proche, entre deux sous-cliques de coût nul, les arêtes ont le même coût non nul, qu'on note c_{ij} entre les sous-cliques i et j (*point 2*).

Par ailleurs, s'il y a au moins trois sous-cliques de coût nul, il apparaît des triangles sans arêtes de coût nul, lorsqu'on prend chaque sommet dans une clique disjointe. Dans ce cas-là, il faut que l'inégalité triangulaire y soit respectée (*point 3*).

Pour prouver l'équivalence, il faut montrer que la caractérisation donnée est une instance métrique réduite. Il y a deux types de coûts d'arêtes : 0 et c_{ij} strictement positif. Avec ces coûts, deux configurations violent l'inégalité triangulaire. Soit le triangle a une arête de coût positif et deux de coût nul. Or cette configuration est proscrite, car les arêtes de coût nul forment des sous-cliques disjointes et donc tout triangle dont deux arêtes ont un coût nul appartient à une telle clique. Soit il existe un triangle sans arêtes de coût nul tel que $c_{ij} + c_{jk} < c_{ik}$, qui n'est possible qu'entre des sommets appartenant à des cliques disjointes. Or l'inégalité triangulaire est bien respectée entre les parties, ce qui rend aussi cette configuration impossible.

Enfin, pour vérifier que la caractérisation représente une instance réduite, on montre que la valeur de la solution de son problème de réduction d'instances associé

est nulle. Tout sommet appartient à une unique clique. Or, quand on additionne les contraintes du programme linéaire (4.4) associées aux arêtes d'une clique de coût nul, on obtient que la somme des offsets sur la clique est nulle. En considérant toutes les cliques disjointes et de taille au moins deux, couvrant les sommets du graphe, on en déduit que la solution optimale est de coût nul. ■

Ce théorème est une caractérisation des instances réduites métriques, schématisée par la figure 4.6. Cette description donne une méthode pour diminuer la taille de l'instance lorsqu'on cherche à la résoudre. En effet, on peut décomposer le tour optimale. Celui-ci est composé de chemins hamiltoniens dans chaque composante connexe du sous-graphe I^0 restreint aux arêtes de coût nul, qui sont des cliques disjointes couvrant le graphe. Ces chemins sont ensuite reliés entre eux chacun par une arête de I^*. Cela revient à trouver un cycle hamiltonien dans le graphe obtenu par contraction des composantes connexes de I^0, qui est une clique dont la taille correspond au nombre de composantes connexes de I^0 et les coûts sur les arêtes sont non nuls de coût c_{ij} entre les sommets représentant les contractions des sous-cliques de coût nul i et j. La taille de la nouvelle instance ainsi obtenue est au moins divisée par deux. Toutefois il n'y a aucune garantie de pouvoir itérer le processus, car la réduction d'une instance métrique ne reste pas forcément métrique.

Si l'on souhaite rester sur l'ensemble des instances métriques, on peut poser le problème de réduction d'instances sous contrainte de rester métrique. Il en découle également une nouvelle mesure de performance stable en évaluant les algorithmes sur ces instances réduites. On suppose qu'une caractérisation des telles instances pourrait être développée en s'appuyant sur les travaux déjà fait pour la réduction non métrique. Ces travaux pourrait avoir des répercussion dans l'étude de l'approximation du TSP métrique, pour lequel est conjecturé l'existence d'un algorithme de facteur $\frac{4}{3}$, en donnant des instances où la mesure de performance est maximum.

5 Études expérimentales

Ces travaux évaluent l'impact de la réduction d'instances pour des études expérimentales. Nous mesurons les variations de performances des méthodes de résolution entre les instances réduites et initiales. Nous commençons par étudier un algorithme de recherche locale, invariant par modification par offset, pour mesurer l'écart avec les mesures habituelles. Puis nous appliquons l'étude à des méthodes gloutonnes, dont le comportement est sensible aux modifications par offset, pour observer l'impact sur ces méthodes.

5.1 Impact de la réduction d'instances sur la performance

Dans la section 3, nous avons présenté un nouveau critère de mesure pour remédier au manque de stabilité des mesures classiques par ratio vis à vis des modifications par offset des distances. Dans cette partie, nous allons comparer cette nouvelle mesure à une mesure habituellement utilisée dans l'étude de méthodes de résolution. Les mesures de performance sont obtenues par un algorithme de recherche locale de type *2-opt*, pour lequel les solutions sont invariantes avec les modifications par offset.

Pour mesurer la performance de l'algorithme sur l'instance I, nous utiliserons le ratio suivant, qui est l'écart de la solution à l'optimum normalisé par la valeur optimale :

$$\gamma(I) = \frac{SOL(I) - OPT(I)}{OPT(I)}$$

Notre nouvelle mesure sera ce ratio appliqué à l'instance réduite de I :

$$\gamma_R(I) = \frac{SOL(reduction(I)) - OPT(reduction(I))}{OPT(reduction(I))}$$

De bonnes performances sont données par une mesure petite et une amélioration des performances correspond donc à une diminution de la mesure.

Pour obtenir l'instance réduite, il faut résoudre le problème formulé par le programme linéaire (4.4). La solution représente la valeur des offsets à appliquer à chaque sommet pour obtenir une réduction de l'instance initiale. Dans notre étude expérimentale, nous avons appliqué le simplexe pour résoudre le problème de réduction. Dans les résultats suivants, nous comparons les résultats pour les méthodes de résolution avec l'impact de la diminution de la valeur optimale sur l'instance réduite.

Notre étude porte sur plusieurs jeux d'instances classiques dans la pratique du benchmarking pour le TSP. Nous étudions la TSPLIB, fameuse librairie d'instances dédiée au problème et initiée par Reinelt (1991). Pour des raisons de taille mémoire, nous restreignons l'étude aux instances dont la taille est inférieure à 3000 sommets. Certaines instances sont des matrices qui donnent exhaustivement les distances entre les sommets. D'autres instances sont des listes de coordonnées correspondant aux sommets, dont les distances entre sommets sont données par la distance euclidienne. Nous étudions aussi des classes d'instances aléatoires : *Random Distance* (RD) et *Random Euclidean* (RE). Les instances de RD sont des matrices explicites dont les distances sont choisies suivant une loi uniforme. Les instances de RE sont des listes de coordonnées obtenues suivant une loi uniforme et les distances sont données par la norme euclidienne. Les distances considérées sont toujours entières. Pour les instances aléatoires, nous générons pour chaque taille 10 instances avec 200, 1000 et 3000 sommets.

La méthode de résolution employée pour illustrer l'étude est *2-opt*, un algorithme de recherche locale, proposé par Croes (1958). Nous avons utilisé le programme

1. Pour avoir les résultats détaillés de chaque instances de la TSPLIB, se référer à l'annexe A.

TABLE 4.1: Impact de la réduction d'instance pour l'algorithme *2-opt* sur les instances de la TSPLIB au format en coordonnées (68 instances).

	$\frac{OPT(reduction(I))}{OPT(I)}$	γ	γ_R	$\frac{\gamma_R}{\gamma}$
moyenne	0,21	0,13	1,7	14,1
écart-type	0,11	0,05	7,6	64
min	$1,9.10^{-3}$	0,02	0,07	1,5
max	0,66	0,28	63	539

implémenté dans le logiciel Concorde (Applegate et al. 2007). Mais cette version n'est applicable qu'aux instances décrites par la liste des coordonnées des sommets. Or les instances réduites sont sous la forme d'une matrice exhaustive. Pour remédier à ce problème, la solution obtenue sur l'instance d'origine par la recherche locale, qui est invariante par modification par offset de l'instance, est ensuite évaluée sur l'instance réduite. La table 4.1 récapitule les résultats obtenus sur les différents jeux d'instances au format en coordonnées, soit 68 instances [1]. Dans cette table, comme dans les suivantes, nous avons reporté les valeurs moyennes, l'écart-type, les valeurs minimum et maximum de chaque valeur mesurée pour l'ensemble d'instances considéré.

Pour mener l'étude sur les instances qui ne sont pas décrites par une liste de sommets, nous nous sommes appuyés sur le constat suivant. Le rapport entre le ratio sur l'instance réduite et le ratio sur l'instance d'origine est indépendant de la méthode de résolution. En effet, étant donné que $SOL(reduction(I)) - OPT(reduction(I)) = SOL(I) - OPT(I)$,

$$\frac{\gamma_R(I)}{\gamma(I)} = \frac{OPT(I)}{OPT(reduction(I))}$$

Sans appliquer la méthode de résolution à l'instance réduite, on peut donc déduire la variation entre les critères de performance, car la solution de l'algorithme de référence est invariante par modification. Il nous suffit de connaître les valeurs des tours optimaux des instances initiales et réduites. La table 4.2 récapitule les résultats obtenus sur les différents jeux d'instances au format en matrice, soit 29 instances [1].

TABLE 4.2: Impact de la réduction d'instances pour l'algorithme *2-opt* sur les instances de la TSPLIB au format matriciel (29 instances).

	$\frac{OPT(reduction(I))}{OPT(I)}$	$\frac{\gamma_R}{\gamma}$
moyenne	0,19	8,1
écart-type	0,17	8,3
min	0,02	1,0
max	1,0	47

L'écart entre les deux critères d'évaluation γ et γ_R est important. Il y a en

1. Pour avoir les résultats détaillés de chaque instance de la TSPLIB, se référer à l'annexe A.

moyenne un facteur de $12,3$ entre les deux valeurs sur les instances de la TSPLIB, différents formats confondus. Toutes les instances ne se comportent cependant pas de la même manière. On peut remarquer quelques instances particulières dans le jeu d'instances. Presque toutes les instances ont une valeur optimale pour l'instance réduite qui est diminuée d'au moins $\frac{1}{3}$, voire bien plus en moyenne avec $\frac{4}{5}$. Mais l'instance *brg180* fait exception, car l'écart entre les solutions initiales et réduites est nulle. La solution du problème de réduction sur cette instance est donc nulle, car cette instance est déjà réduite. À l'inverse, certaines instances voient leur valeur optimale fortement diminuée avec la réduction (*si1032*, *u1432*, *si175*, *a280*). Le cas le plus marquant est l'instance *u2319*, dont la solution optimale de l'instance réduite ne représente que 2.10^{-4} de celle de l'instance initiale, ce qui correspond à une multiplication par $538,5$ du critère de mesure.

TABLE 4.3: Impact de la réduction d'instances pour l'algorithme *2-opt* sur 10 instances de la classe RE avec 200 sommets.

	$\frac{OPT(reduction(I))}{OPT(I)}$	γ	γ_R	$\frac{\gamma_R}{\gamma}$
moyenne	0,19	0,11	0,59	5,2
écart-type	0,02	0,03	0,17	0,56
min	0,16	0,06	0,38	4,5
max	0,22	0,15	0,87	6,2

TABLE 4.4: Impact de la réduction d'instances pour l'algorithme *2-opt* sur 10 instances de la classe RE avec 1000 sommets.

	$\frac{OPT(reduction(I))}{OPT(I)}$	γ	γ_R	$\frac{\gamma_R}{\gamma}$
moyenne	0,18	0,13	0,73	5,6
écart-type	0,01	0,02	0,09	0,28
min	0,16	0,09	0,55	5,2
max	0,19	0,16	0,82	6,1

TABLE 4.5: Impact de la réduction d'instances pour l'algorithme *2-opt* sur 10 instances de la classe RE avec 3000 sommets.

	$\frac{OPT(reduction(I))}{OPT(I)}$	γ	γ_R	$\frac{\gamma_R}{\gamma}$
moyenne	0,18	0,14	0,79	5,7
écart-type	$3,9.10^{-3}$	0,01	0,06	0,12
min	0,17	0,12	0,70	5,4
max	0,19	0,16	0,90	5,8

Les tables 4.3, 4.4, 4.5, 4.6, 4.7 et 4.8 récapitulent les résultats obtenus sur les instances aléatoires, avec 10 instances générées pour des tailles de 200, 1000 et 3000 sommets. Les instances aléatoires sont assez proches du comportement moyen

TABLE 4.6: Impact de la réduction d'instances pour l'algorithme *2-opt* sur 10 instances de la classe RD avec 200 sommets.

	$\frac{OPT(reduction(I))}{OPT(I)}$	$\frac{\gamma_R}{\gamma}$
moyenne	0,20	5,1
écart-type	0,03	0,90
min	0,15	4,0
max	0,25	6,9

TABLE 4.7: Impact de la réduction d'instances pour l'algorithme *2-opt* sur 10 instances de la classe RD avec 1000 sommets.

	$\frac{OPT(reduction(I))}{OPT(I)}$	$\frac{\gamma_R}{\gamma}$
moyenne	0,20	4,9
écart-type	0,01	0,22
min	0,19	4,6
max	0,22	5,4

TABLE 4.8: Impact de la réduction d'instances pour l'algorithme *2-opt* sur 10 instances de la classe RD avec 3000 sommets.

	$\frac{OPT(reduction(I))}{OPT(I)}$	$\frac{\gamma_R}{\gamma}$
moyenne	0,20	5,1
écart-type	0,01	0,16
min	0,19	4,9
max	0,21	5,3

des instances de la TSPLIB avec une diminution du tour optimal après réduction d'environ $\frac{4}{5}$, et avec une très faible variabilité, ce qui donne un facteur proche de 5 entre les critères d'évaluation. On note toutefois, pour les instances de RE, un impact de la réduction un peu plus marqué, avec des résultats légèrement supérieurs à $\frac{4}{5}$. De plus, cette tendance semble s'accentuer lorsque la taille des instances de RE augmente, alors que les résultats sont plutôt stables avec la taille pour les instances de RD.

Cette étude est une première application de notre mesure de performance sur les instances réduites. On constate en testant sur les jeux d'instances classiques que la différence avec la mesure de performance habituelle est importante. Alors qu'une méthode de résolution comme *2-opt* donne des solutions au pire à 28% de l'optimum sur les instances euclidiennes, en réduisant on obtient des performances bien plus mauvaises, qui vont jusqu'à 63 fois l'optimum, pour retourner la même solution.

Ces résultats devraient remettre en cause les critères usuels pour mesurer l'efficacité des algorithmes. Les performances sur une instance quelconque peuvent faci-

lement être améliorées par du "bruit" qui camoufle la vraie difficulté de l'instance. Ceci soulève également des questions sur le choix d'instances pertinentes pour l'étude empirique. Dans le cadre d'études d'algorithmes dont le déroulement est invariant vis à vis des modifications par offset, comme les méthodes locales, mais aussi les algorithmes génétiques ou toute méthode qui compare des solutions, nous recommandons de ne travailler plus qu'avec des instances réduites.

5.2 Difficulté des instances réduites pour des algorithmes gloutons

Nous avons pu observer l'impact de la réduction d'instances sur l'étude des méthodes de résolution qui reposent sur la comparaison des solutions. Mais qu'en est-il des autres méthodes, comme les algorithmes gloutons, qui reposent sur la comparaison des distances et dont le déroulement est sensible à des modifications par offset ?

Nous avons étudié le comportement de deux algorithmes gloutons classiques pour résoudre le TSP. Le premier est l'algorithme *Greedy* (GRE), qui ajoute à la solution les arêtes dans l'ordre croissant de leur coût, à condition de ne pas former de sous-tours. Le second algorithme est *Nearest Neighbor* (NN), qui construit la solution en parcourant les sommets en choisissant l'arête de plus petit coût, à condition de ne pas fermer le tour avant le dernier sommet.

Nous appliquons ces algorithmes sur les instances précédemment utilisées et leurs réductions : les instances de la TSPLIB et des instances aléatoires de type *Random Distance* et *Random Euclidean*. La mesure de performance utilisée est l'écart normalisé de la solution à l'optimum.

Pour évaluer l'impact de la réduction de ces instances sur les algorithmes, nous comparons leurs performances entre les instances initiales (γ) et les instances réduites (γ_R). Les tables 4.9, 4.10, 4.11, 4.12, 4.13 et 4.14 regroupent la moyenne, l'écart-type, les valeurs minimum et maximum des différentes mesures obtenues grâce à ces expérimentations. Pour les instances aléatoires, nous avons compilé les résultats de 10 instances à 200, 1000 et 3000 sommets.

TABLE 4.9: Impact de la réduction d'instances pour l'algorithme *Greedy* sur les instances de la TSPLIB (97 instances).

	$\frac{OPT(I)}{OPT(reduction(I))}$	γ	γ_R	$\frac{\gamma_R}{\gamma}$	δ	β
moyenne	12	0,26	3,2	27	15	1,3
écart-type	55	0,81	13	114	70	5,9
min	1,0	$4,7.10^{-3}$	0,20	2,3	-4,3	-1,6
max	539	8,2	127	972	541	57

Les performances obtenues sur nos jeux d'instances sont toujours moins bonnes lorsque l'instance est réduite. Au minimum, le facteur est de $1,25$ avec NN pour

TABLE 4.10: Impact de la réduction d'instances pour l'algorithme *Nearest Neighbor* sur les instances de la TSPLIB (97 instances).

	$\frac{OPT(I)}{OPT(reduction(I))}$	γ	γ_R	$\frac{\gamma_R}{\gamma}$	δ	β
moyenne	12	0,33	3,9	19	6,8	1,2
écart-type	55	0,81	18	89	36	6,8
min	1,0	0,02	0,22	1,2	-4, 0	-1,1
max	539	8,2	177	871	332	67

TABLE 4.11: Impact de la réduction d'instances pour l'algorithme *Greedy* sur la classe RD avec 200, 1000 et 3000 sommets (10 instances par taille).

	$\frac{OPT(I)}{OPT(reduction(I))}$	γ	γ_R	$\frac{\gamma_R}{\gamma}$	δ	β
moyenne	5,0	1,7	7,0	4,1	-0,92	-1,6
écart-type	0,53	0,38	1,8	0,84	0,87	1,6
min	4,0	1,0	3,5	2,2	-2,8	-5,3
max	6,9	2,6	10	5,43	0,53	0,92

TABLE 4.12: Impact de la réduction d'instances pour l'algorithme *Nearest Neighbor* sur la classe RD avec 200, 1000 et 3000 sommets (10 instances par taille).

	$\frac{OPT(I)}{OPT(reduction(I))}$	γ	γ_R	$\frac{\gamma_R}{\gamma}$	δ	β
moyenne	5,0	2,4	10	4,3	-0,78	-1,9
écart-type	0,53	0,55	3,1	0,78	0,96	2,0
min	4,0	1,5	5,2	2,6	-3,5	-6,1
max	6,9	3,4	16	6,1	1,1	2,2

TABLE 4.13: Impact de la réduction d'instances pour l'algorithme *Greedy* sur la classe RE avec 200, 1000 et 3000 sommets (10 instances par taille).

	$\frac{OPT(I)}{OPT(reduction(I))}$	γ	γ_R	$\frac{\gamma_R}{\gamma}$	δ	β
moyenne	5,5	0,16	1,4	8,6	3,1	0,48
écart-type	0,41	0,02	0,17	1,5	1,5	0,20
min	4,5	0,12	0,87	5,7	-0,20	-0,04
max	6,2	0,21	1,7	12	6,7	0,78

l'instance *brg180* et va jusqu'à 972 avec GRE pour *u2319*. L'instance *brg180* a déjà été mentionnée dans la précédente étude pour sa particularité à être initialement réduite. Toutefois, les performances des heuristiques changent comme si l'instance avait été modifiée. En effet, bien que la solution du problème de réduction ait une valeur nulle pour les instances réduite, les valeurs des variables de la solution peuvent ne pas être nulle, à condition que la somme le soit.

Globalement, on remarque que les performances empirent lorsque $\frac{OPT(I)}{OPT(reduction(I))}$

TABLE 4.14: Impact de la réduction d'instances pour l'algorithme *Nearest Neighbor* sur la classe RE avec 200, 1000 et 3000 sommets (10 instances par taille).

	$\frac{OPT(I)}{OPT(reduction(I))}$	γ	γ_R	$\frac{\gamma_R}{\gamma}$	δ	β
moyenne	5,5	0,25	2,1	8,9	3,4	0,79
écart-type	0,41	0,03	0,26	1,5	1,4	0,32
min	4,5	0,19	1,5	4,0	-1,0	-0,39
max	6,2	0,37	2,4	12	6,8	1,3

est grand. En effet, bien que les solutions des algorithmes gloutons ne soient pas invariantes par réduction, elles n'en sont pas moins influencées par la diminution de la valeur optimale. Il semble donc assez logique que les performances des heuristiques sur les instances réduites augmentent, comme pour toute méthode de résolution. Pour évaluer si les instances réduites sont plus difficiles pour les heuristiques, nous proposons l'étude de deux critères.

Tout d'abord, on compare l'évolution des performances des heuristiques par rapport à l'évolution causée par la diminution de la valeur optimale. Comme vu précédemment, dans le cas d'une méthode qui est invariante par réduction, le ratio des performances entre l'instance initiale et l'instance réduite est égal à $\frac{OPT(I)}{OPT(reduction(I))}$. Dans le cas des algorithmes gloutons, cette égalité n'est pas vérifiée. Ainsi nous utilisons leur différence entre les deux valeurs comme premier critère, que nous notons δ. Le signe du critère indique si l'instance réduite est plus difficile (positif) ou plus facile (négatif) à résoudre.

$$\delta(I) = \frac{\gamma_R(I)}{\gamma(I)} - \frac{OPT(I)}{OPT(reduction(I))}$$

Avec ce critère, on constate que la majorité des instances réduites de la TSPLIB et celles de RE sont plus difficiles, contrairement à celles de RD qui sont principalement plus faciles. Toutefois, les mesures extrêmes sont bien moindre du côté de la facilité ($-4,29$ avec GRE pour *eil51*) que de la difficulté (541 avec GRE pour *si1032*) des instances réduites.

Le second critère, noté β, repose sur l'écart entre les performances des instances initiales et de celles réduites par rapport à leur optimum respectif. Pour avoir un ordre de grandeur comparable entre les instances, on normalise cet écart par la valeur optimale de l'instance réduite, qui est une mesure stable.

$$\beta(I) = \frac{(SOL(Reduction(I)) - OPT(Reduction(I))) - (SOL(I) - OPT(I))}{OPT(Reduction(I))}$$

Si ce critère est nul, les performances de l'heuristique sont équivalentes entre l'instance initiale et réduite, comme pour les méthodes par recherche locale. S'il est négatif, les performances sont meilleures pour les instances réduites, qui sont alors plus faciles. Sinon, les instances réduites sont plus difficiles.

On remarque que les résultats de difficulté des instances réduites pour ce critère sont corrélés avec ceux du critère δ, car ils sont toujours du même signe. En effet, on peut montrer que $\delta(I) = \beta(I) \times \gamma(I)$, ce qui confirme nos observations. Même si la différence d'ordre de grandeur des extrêmes entre facilité et difficulté des instances réduites persiste pour ce critère, l'écart est moindre : entre $-6,06$ avec NN pour l'instance *RD_0200_2* et $67,5$ avec NN pour l'instance *u2319*.

Cette étude permet d'observer l'impact de la réduction d'instances pour des méthodes de résolution dont le comportement n'est pas invariant par cette modification. Globalement, on constate que les performances de ces méthodes de résolution sont dégradées sur les instances réduites. Toutefois, cette dégradation est toujours influencée par la diminution de la valeur optimale de l'instance réduite. Lorsqu'on fait abstraction de ce phénomène, on observe des performances sur les instances réduites plus variables : plutôt pires pour la TSPLIB et RE, plutôt meilleures pour RD.

À nouveau, les résultats obtenus dans cette étude nous mettent en garde sur l'utilisation d'instances non réduites. Les performances peuvent être arbitrairement bonnes pour toute méthode de résolution dont on mesure la qualité de la solution. Dans le cas d'algorithmes sensibles aux modifications d'instances par offset, nous suggérons tout de même de ramener la mesure de performance des instances au problème réduit. Pour évaluer le fonctionnement de l'algorithme sur les instances initiales, on présente un critère de mesure qui évalue la solution obtenue pour l'instance initiale sur l'instance réduite.

$$\gamma'_R = \frac{SOL(I) - OPT(I)}{OPT(reduction(I))}$$

On obtient la performance de l'algorithme sur l'instance réduite, comme si son comportement n'était pas modifié. On observe que $\gamma'_R(I) = \beta(I) - \gamma_R(I)$.

Lors de l'étude, nous avons toutefois observé, indépendamment de la diminution de la valeur optimale, des performances assez variables, mais qui semblent tendre globalement vers une hausse de la difficulté après réduction. Il pourrait donc être intéressant d'approfondir la recherche dans cette direction et essayer de générer des instances difficiles grâce à la modification par offset pour les méthodes de résolution qui y sont sensibles.

Conclusion

Ce chapitre porte sur une opération qui modifie les instances du problème du voyageur de commerce. Cette modification présente l'intérêt majeur de pouvoir trouver facilement la solution optimale de la nouvelle instance à partir de la solution de l'instance initiale. Ainsi, en faisant l'effort de résoudre une instance, on peut générer un nombre quelconque d'instances dont on connait la solution optimale. En particulier, cette opération pourrait être combinée à l'approche par évolution d'instances qui

repose sur la recherche locale, comme proposée par Ahammed and Moscato (2011). L'évaluation des instances parcourues rend la méthode assez fastidieuse, car elle repose sur la performance d'un algorithme par rapport à la valeur optimale, longue à calculer. Si la règle de voisinage dépend de la modification d'instances, le calcul de la solution optimale serait immédiat. La méthode deviendrait alors beaucoup plus efficace et permettrait d'étudier des instances de taille bien plus grande.

Nous avons remarqué que cette modification peut influencer la valeur des performances de certains algorithmes de résolution, comme les méthodes de type recherche locale, lorsque l'on mesure le rapport à la valeur optimale. En appliquant judicieusement les modifications, nous avons montré qu'on peut obtenir une instance réduite qui maximise la valeur de la performance. Des travaux théoriques nous ont permis de caractériser la forme de ces instances. Nous avons ainsi proposé, avec les instances réduites, une nouvelle classe d'instances qui permet de mesurer des performances plus représentatives des algorithmes.

Nous avons mené des études expérimentales sur les classes d'instances usuelles pour évaluer l'impact de la réduction d'instances. Le résultat est impressionnant par des écarts très importants, qui peuvent donner une autre interprétation des études sur les méthodes de résolution. Nous proposons d'utiliser une nouvelle mesure de performance qui soit stable pour ces modifications :

$$\gamma'_R = \frac{SOL(I) - OPT(I)}{OPT(reduction(I))}$$

Nous avons aussi commencé des travaux sur la métricité en caractérisant les instances réduites métriques et en présentant un algorithme d'approximation avec une garantie de performance indépendante du nombre de sommets du problème, mais fonction des coûts minimum et maximum. Nous avons proposé des pistes d'approfondissement avec l'étude d'une nouvelle réduction qui conserverait, voire imposerait, la métricité.

Au final, cette étude sur une modification d'instances pour le TSP a abouti à des résultats intéressants avec des répercutions sur les études expérimentales. Nous ne pouvons donc qu'encourager pareilles études pour d'autres problèmes. Une piste pour trouver de bons candidats serait d'étudier les problèmes de ré-optimisation dont la résolution est facile, qui donneraient une modification d'instances dont on connaît l'impact sur la solution optimale.

Chapitre 5

Réductions d'instances et problèmes FPT

Ce chapitre est dans la continuité du précédent, car nous y présentons des modifications sur les instances. Toutefois, nous y abordons d'autres problèmes et les travaux s'inscrivent davantage dans le cadre de travaux théoriques sur la complexité des problèmes.

Le lien avec la difficulté d'instance est toujours présent. Nous étudions des modifications qui réduisent la taille des instances en résolvant les parts faciles de l'instance. Elles "grignotent" les instances pour ne laisser que le cœur difficile de l'instance. Cette idée est présente dans les problèmes *Fixed-Parameter Tractable* (FPT) avec la notion de nucléarisation. Nous détaillons dans un premier temps ces deux notions. Puis nous présentons des modifications dans deux études sur des problèmes combinatoires.

Les travaux présentés dans la section 3 font partie d'une étude menée au sein du projet MASCOTTE d'INRIA-Sophia Antipolis, en collaboration avec J. Araujo, G. Morel, L. Sampaio et R. Soares (Araujo et al. 2012). D'autres résultats y figurent mais ne sont pas abordés ici, car il ne s'inscrivent pas dans le thème de la thèse (annexe D). Ils ont été acceptés à la conférence *VII Latin-American Algorithms, Graphs and Optimization Symposium (LAGOS '13)*.

1 Définitions

Nous définissons les différentes notions de problème rencontrées en théorie de la complexité. Puis, nous présentons la définitions de *Fixed-Parameter Tractability* (FPT). Enfin nous donnons une nouvelle définition de réduction d'instance, qui complète la définition que nous avons donnée dans le chapitre 4.

1.1 Problèmes de décision paramétrés

Nous avons abordé la notion de problème dans la section 1 du chapitre 1. Nous allons en donner une définition plus précise, sans pour autant utiliser dans le formalisme de la théorie de la complexité.

Dans les chapitres précédents, nous avons étudié le problème du voyageur de commerce. Tel que nous l'avons traité, nous le considérions comme un problème d'optimisation, c'est-à-dire que la réponse attendue était la valeur d'une solution au problème qui minimise l'objectif. Plus formellement, un problème d'optimisation se décrit comme suit. On note $co\hat{u}t(I, y)$ la fonction qui donne le coût de la solution y pour l'instance I.

Problème d'optimisation
INSTANCE : I
QUESTION : Trouver $\min\{co\hat{u}t(I, y) \mid y \in sol(I)\}$

La classification des problèmes dans la théorie de la complexité, repose sur la notion de problème de décision. Nous avons vu qu'un problème est composé d'un ensemble infini d'instances et d'une question. Pour un problème de décision, la réponse attendue est "oui" ou "non". Le problème de décision associé à un problème d'optimisation se décrit comme suit.

Problème de décision
INSTANCE : I et un réel (ou entier) r
QUESTION : Existe-t-il $y \in sol(I)$ tel que $co\hat{u}t(I, y) \leq r$?

Le problème de décision pour le problème du voyageur de commerce revient à répondre à la question "Existe-t-il un cycle hamiltonien de longueur inférieure à r ?". Les problèmes de décision et d'optimisation sont évidemment intimement liés. Une recherche par dichotomie sur la longueur du tour permet de trouver la solution optimale en un nombre logarithmique de résolutions du problème de décision.

On peut dériver de ces problèmes des versions paramétrées. Le problème contient maintenant une donnée supplémentaire qui représente la valeur d'un paramètre.

Problème de décision paramétré
INSTANCE : I et un réel (ou entier) r
PARAMÈTRE : k
QUESTION : Existe-t-il $y \in sol(I)$ tel que $co\hat{u}t(I, y) \leq r$?

Le paramétrage naturel pour un problème de décision associé à un problème d'optimisation est de prendre la valeur de la solution ($k = r$). Dans ce cas, la valeur de la solution devient une donnée fixe du problème, contrairement au cas du problème de décision où elle est une entrée variable. Le paramètre peut aussi porter sur une

grandeur différente de la valeur de la solution, comme une caractéristique de l'instance. Pour le problème de décision k-SAT, par exemple, le paramètre k représente le nombre de variables par clause.

1.2 FPT et nucléarisation

Les problèmes *NP*-complets sont des problèmes de décision pour lesquels il n'existe pas d'algorithme de résolution en temps polynomial avec la taille de l'instance, si $P \neq NP$. Toutefois, la version paramétrée d'un problèmes de décision *NP*-complet peut être facile pour un k fixé, comme le problème de k-coloration lorsque $k = 2$. Plus généralement, certains problèmes paramétrés sont faciles pour toute valeur fixée du paramètre. Par exemple, trouver un indépendant de taille au moins k dans une instance de taille n est polynomial en la taille de l'instance. En effet, il suffit de tester tous les sous-ensembles de sommets de taille k, dont le nombre est polynomial avec n, pour k fixé.

Un algorithme est *Fixed-Parameter Tractable* (FPT) selon k s'il peut résoudre toute instance (I, k) d'un problème paramétré en temps

$$\mathcal{O}(f(k) \cdot n^{\mathcal{O}(1)}),$$

avec n la taille de l'instance I et $f : \mathbb{N} \to \mathbb{N}$ une fonction calculable. On dit qu'un problème paramétré par k est FPT s'il existe un algorithme FPT selon k pour le résoudre. Les problèmes FPT sont donc des problèmes de décision paramétrés qu'on sait résoudre polynomialement selon la taille de l'instance à valeur fixée du paramètre. Ce n'est toutefois pas suffisant pour être FPT, comme dans le cas du problème d'indépendant maximum.

Une *nucléarisation* est un algorithme polynomial qui réduit une instance (I, k) d'un problème paramétré en une instance (I', k') du même problème, dont la taille est bornée par une fonction de k, et $k' \leq k$ [1]. Ces algorithmes sont directement liés aux problèmes FPT.

Théorème 5.1 *(Niedermeier 2006) Un problème est FPT, si et seulement si, le problème admet une nucléarisation.*

Preuve : On démontre l'équivalence en traitant chaque sens d'implication.

S'il existe un algorithme de nucléarisation, on peut réduire l'instance à une instance de taille bornée en un nombre d'opérations polynomial avec la taille, puis y appliquer une méthode de résolution par recherche exhaustive.

1. Théoriquement, $k' \leq h(k)$, avec $h : \mathbb{N} \to \mathbb{N}$ une fonction calculable, mais en pratique, pour aucun algorithme de nucléarisation connu $k' > k$

Si le problème est FPT en k, on suppose qu'il existe un algorithme A qui résout en temps $f(k) \cdot n^c$, avec f une fonction calculable quelconque et c une constante. Soit une instance (I, k) du problème de taille n. Si $f(k) \leq n$, on applique A qui résout I en un temps polynomial $f(k) \cdot n^c \leq n^{c+1}$ et selon la solution, on retourne une instance trivialement vraie ou fausse, qui est dans le noyau. Si $f(k) > n$, I est déjà dans le noyau. ■

Les implications de ce résultat restent toutefois mitigées en pratique. En effet, la taille du noyau est bornée par une fonction quelconque, qui peut être exponentielle. Dans un tel cas, l'application pratique de la recherche exhaustive est rapidement limitée lorsque la valeur du paramètre devient trop grande. Il y a tout de même des problèmes pour lesquels la fonction est polynomiale (resp. linéaire). On parle alors de noyau polynomial (resp. linéaire). Dans ce cas, l'algorithme de nucléarisation est souvent une succession de règles de réduction.

L'exemple couramment employé est le problème du transversal minimum. On résout le problème de décision de savoir s'il existe un ensemble (transversal) de sommets de taille au plus r, tel que toute arête a au moins une extrémité dans l'ensemble. L'algorithme de nucléarisation proposé par Buss considère r comme paramètre et possède deux règles de réduction. D'une part, tous les sommets de degré supérieur à r sont forcément dans le transversal. D'autre part, les sommets de degré 0 ne sont pas dans le transversal. Les sommets d'un graphe qu'on ne peut pas réduire, ont des degrés compris entre 1 et r. S'il existe un transversal de taille inférieure ou égale à r, le graphe a au plus $r + r^2$ sommets : r sommets de degré r et r^2 de degré 1. Le problème admet donc un noyau quadratique. Avec d'autres règles de réduction, on peut obtenir un noyau linéaire pour le même problème. Pour plus d'informations sur cet exemple et de manière générale sur les problèmes FPT, nous renvoyons le lecteur à des ouvrages de référence (Downey and Fellows 1999, Flum and Grohe 2006, Niedermeier 2006).

1.3 Réduction

Nous définissons une nouvelle notion de *réduction*, qui englobe les précédentes notions rencontrées et contient une dimension pratique. Une réduction est une transformation polynomiale d'une instance d'un problème algorithmique dans le but de faciliter sa résolution par une méthode adéquate. Elle peut être assimilée à du prétraitement. Dans un premier temps, on ne considère que les problèmes paramétrés. Nous verrons ensuite comment généraliser aux problèmes d'optimisation.

On différencie deux conditions pour qu'une modification d'instances soit considérée comme une réduction.

Définition 5.2 *Soit un problème de décision paramétré par $k \geq 0$. Une réduction d'instance est une modification d'instance : $(I, k) \to (I', k')$, polynomiale en la taille*

de l'instance, qui vérifie au moins un des deux points suivants :

1. $|I'| < |I|$ *et* $k' \leq f(k)$*, avec* f *quelconque ;*
2. $k' < k$ *et* $|I'| \leq g(|I|)$*, avec* g *polynomial.*

On appelle *algorithme de réduction*, un algorithme qui itère une (ou plusieurs) réduction(s) tant que les conditions pour réduire sont vérifiées. Si l'algorithme contient plusieurs règles de réduction, soit $f(k) \leq k$ pour toute réduction respectant le cas 1, soit $g(|I|) \leq |I|$ pour toute réduction respectant le cas 2 et $f(k) = k + c$ avec $c \in \mathbb{R}$ pour toute réduction respectant le cas 1.

Théorème 5.3 *Un algorithme de réduction est en temps polynomial.*

Preuve : On commence par étudier le cas où toutes les réductions de l'algorithme respectent le même cas. Si la taille de l'instance diminue strictement (cas 1), le nombre de réductions sera au plus égal à $|I|$, ce qui assure que l'algorithme termine en temps polynomial. Si la valeur du paramètre diminue strictement (cas 2), le nombre de réductions est aussi borné, car zéro est une borne inférieure de la valeur du paramètre. La taille des instances après chaque réduction est toujours bornée par $\max_{k \in [0,k_0]} g^k(|I_0|)$, où (I_0, k_0) est l'instance initiale et $g^i = g \circ g^{i-1}$ avec $g^1 = g$. L'algorithme de réduction est donc polynomial.

Si l'algorithme contient des règles qui respectent les deux cas, d'après la définition d'un algorithme de réduction, soit la taille, soit le paramètre n'augmente jamais. Supposons que $f(k) \leq k$ pour toute réduction respectant le cas 1. Alors le paramètre du problème n'augmente jamais. Montrons que le nombre de réductions est borné. D'après l'hypothèse, on déduit qu'il y a au plus k_0 réductions de cas 2. Soit la fonction $G(n) = \max_{x \in [0,n]} g(x)$. La valeur du paramètre est toujours bornée par $G^{k_0}(|I_0|)$. Même en alternant les réductions de cas 1 avec au plus k_0 réductions de cas 2, le nombre de réductions de cas 1 est borné par $k_0 \cdot G^{k_0}(|I_0|)$ qui est un polynôme en $|I_0|$ et donc l'algorithme est en temps polynomial avec la taille de l'instance.

Supposons maintenant que $g(|I|) \leq |I|$ pour toute réduction respectant le cas 2. La taille de l'instance n'augmente jamais. Par un raisonnement analogue, on montre que le nombre de réductions de cas 2 est borné par $|I_0| \cdot (k_0 + |I_0| \cdot c)$. L'algorithme de réduction est donc en temps polynomial avec la taille de l'instance. ■

Cette définition de la réduction est une généralisation de la notion vue dans le chapitre 4. La réduction donnée sur les instances du TSP diminue la valeur objectif, ce qui se ramène au cas 2. La définition peut aussi être vue comme une expression pratique des règles de réduction d'un algorithme de nucléarisation. La seule condition manquante est celle qui borne la taille de l'instance finale en fonction du paramètre de l'instance initiale. Mais cette définition pratique peut précisément permettre de mener des études dans ce sens et évaluer empiriquement si les instances obtenues après réductions sont de taille bornée et donc si elles forment un noyau.

Les définitions données dans ce chapitre portent jusqu'ici sur les problèmes de décision. Nous montrons maintenant comment cette définition de réduction peut être applicable aux problèmes d'optimisation. La seule condition concerne la nature des réductions, qui ne doivent pas dépendre du paramètre du problème. Les réductions qui modifient la valeur du paramètre peuvent alors être vues comme une modification de la valeur de la solution optimale (de $k - k'$) entre I et I'. Cette valeur, à partir d'une solution de l'instance réduite, permet de remonter à la solution de l'instance de départ.

1.4 Notations

Nous donnons quelques autres définitions qui seront utilisées dans la suite. On note $\overline{G}$ le graphe complémentaire de G. On note K_l la clique à l sommets. Le complémentaire de K_l est un graphe à l sommets sans arêtes, qu'on appelle *anti-*K_l. Si $l = 3$, on parle d'*anti-triangle*. On note $N(v) = \{u \in V \mid uv \in E\}$ le *voisinage (ouvert)* du sommet v et $N[v] = N(v) \cup \{v\}$ le *voisinage fermé* de v. Si le voisinage d'un sommet est une clique, on dit que le sommet est *simplicial*. Pour le graphe $G = (V, E)$ et le sous-ensemble de sommets $S \subseteq V$, on note $G\langle S\rangle = (S, \{ij \in E \mid \{i, j\} \subseteq S\})$ le sous-graphe de G *induit* par S.

2 Transversal minimum et indépendant maximum

Nous présentons les deux problèmes de l'étude, puis deux opérations de réduction avec quelques résultats de leurs impacts sur les solutions.

2.1 Présentation des problèmes

Le *problème du transversal minimum* est un problème d'optimisation classique en optimisation combinatoire, connu pour être *NP*-difficile (Garey and Johnson 1979). Le problème est le suivant. Dans un graphe simple $G = (V, E)$, trouver un sous-ensemble (*transversal*) de sommets $S \subseteq V$ minimum tel que toute arête $e \in E$ ait au moins une extrémité dans S. La *taille du transversal minimum* de G est notée $\tau(G)$.

Problème du transversal minimum
INSTANCE : un graphe $G = (V, E)$
QUESTION : Trouver la taille d'un transversal minimum de G

Le *problème d'indépendant maximum* est donc aussi *NP*-difficile. Ce problème consiste à trouver le sous-ensemble (*indépendant*) de sommets maximum tel que le

sous-graphe induit n'ait aucune arête. La taille de l'indépendant maximum de G est notée $\alpha(G)$.

Problème d'indépendant maximum
INSTANCE : un graphe $G = (V, E)$
QUESTION : Trouver la taille d'un l'indépendant maximum de G

Le deux problèmes sont très liés, car le complémentaire d'un transversal est un indépendant. Il en découle l'identité suivante de Gallai (1959).

Propriété 5.4 *Soit un graphe G=(V,E),*

$$\alpha(G) + \tau(G) = |V|$$

Nous présentons d'abord des réductions pour le problème d'indépendant maximum. Puis nous étudions leurs impacts lorsqu'on les applique au problème du transversal minimum.

2.2 Pliage

Pour le problème d'indépendant, Fomin et al. (2006) présentent une transformation du graphe, appelée *pliage* ou *folding*, qui diminue la taille de l'indépendant maximum de 1. Cette transformation s'applique localement à un sommet et à son voisinage. Pour être éligible à cette transformation, le voisinage du sommet ne doit pas contenir d'anti-triangles. On parle alors de sommet *pliable*.

Définition 5.5 *Voici les étapes du pliage d'un sommet v pliable.*

1. *Ajouter un nouveau sommet $u_{ij} (i > j)$ pour chaque non-arête $u_i u_j$ de N(v) ;*
2. *Ajouter des arêtes entre u_{ij} et les sommets de $N(u_i) \cup N(u_j)$;*
3. *Ajouter une arête entre chaque paire de nouveaux sommets u_{ij} ;*
4. *Retirer v et $N(v)$.*

Le nouveau graphe ainsi obtenu est noté $\tilde{G}(v)$. Un exemple de pliage est donné figure 5.1. v et son voisinage sont transformés en 3 sommets qui correspondent aux 3 non-arêtes du voisinage de v.

Théorème 5.6 *(Fomin et al. 2006) Pour un graphe G et un sommet v pliable,*

$$\alpha(\tilde{G}(v)) = \alpha(G) - 1.$$

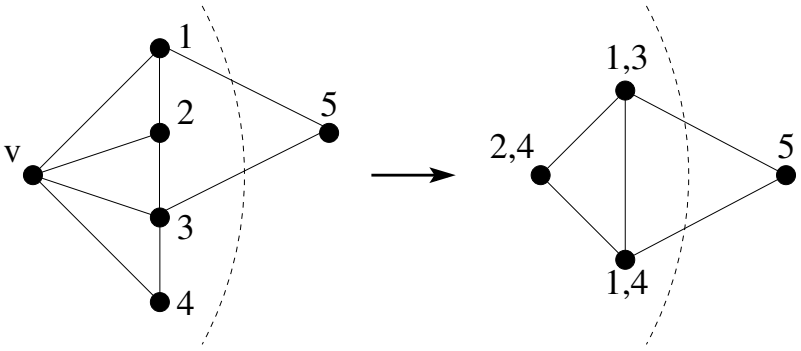

FIGURE 5.1: Exemple de pliage pour le sommet pliable v. Le sous-graphe induit à droite des pointillés est invariant. (2,4-5) est l'indépendant maximum dans le graphe modifié, ce qui donne (2-4-5) comme indépendant maximum du graphe initial.

2.3 Pliage généralisé

Nous avons généralisé la transformation de pliage afin de l'appliquer à tout voisinage. Il s'avère que cette transformation est déjà connue sous le nom de *struction*, qui est la contraction de *stability reduction*. Nous en présentons une preuve personnelle. Ensuite nous étudions son utilisation pour réduire les instances et son application au problème du transversal maximum.

Nous proposons une transformation pour étendre le pliage à tout sommet, que nous appelons *pliage généralisé*. La méthode est similaire, à la différence que les nouveaux sommets ajoutés ne forment plus nécessairement une clique (étape 3). Pour chaque anti-sous-clique de $N(v)$ correspond une anti-sous-clique de taille juste inférieure dans l'ensemble des nouveaux sommets. Cette transformation a été proposée avant nous par Ebenegger et al. (1984).

Définition 5.7 *Voici les étapes du pliage généralisé d'un sommet v quelconque. On suppose un ordre total sur les sommets du voisinage de v.*

1. *Ajouter un nouveau sommet $u_{ij} (i > j)$ pour chaque non-arête $u_i u_j$ de N(v) ;*
2. *Ajouter des arêtes entre u_{ij} et les sommets de $N(u_i) \cup N(u_j)$;*
3. *Ajouter une arête, pour chaque sommet $u_i \in N(v)$,*
   (a) *entre u_{ij} et u_{kl}, si $i > j, k, l$;*
   (b) *entre u_{ij} et u_{il}, si $i > j, l$ et $u_j u_l \in E$;*
4. *Retirer v et $N(v)$.*

Le nouveau graphe ainsi obtenu est noté $\mathring{G}(v)$. Un exemple de pliage généralisé est donné figure 5.2.

La construction peut être vue de manière itérative en parcourant les sommets du voisinage de v suivant l'ordre totale. Toute non-arête entre le sommet sélectionné

et les sommets déjà parcourus donne un nouveau sommet (étape 1). On relie les nouveaux sommets aux voisinages des sommets dont ils sont issus (étape 2). On relie les nouveaux sommets aux précédents nouveaux sommets (étape 3a). On relie les nouveaux sommets entre eux si les sommets déjà parcourus dont ils sont issus sont adjacents dans G (étape 3b). On supprime v et son voisinage, ainsi que toutes les arêtes auxquelles ils sont incidents (étape 4).

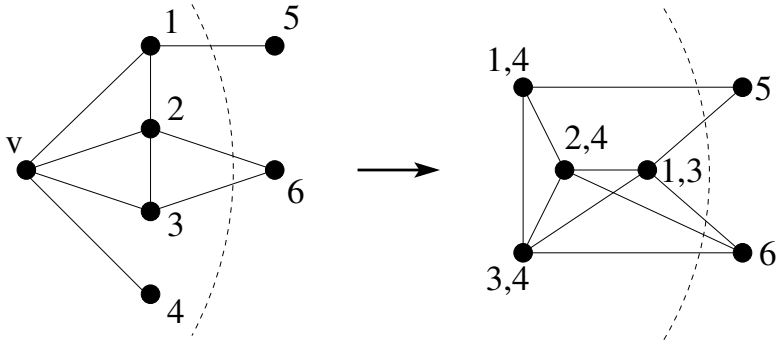

FIGURE 5.2: Exemple de pliage généralisé pour le sommet v, dont le voisinage contient un anti-triangle (1-3-4). Le sous-graphe induit à droite des pointillés est invariant. (5-6) est un indépendant maximum dans le graphe modifié, ce qui donne (v-5-6) comme un indépendant maximum pour le graphe initial.

Cette nouvelle transformation est bien une généralisation du pliage. En effet, si le voisinage de v ne possède pas d'anti-triangles, l'étape 3 relie tous les nouveaux sommets entre eux, et on se ramène à la définition 5.5

Théorème 5.8 *(Ebenegger et al. 1984) Pour un graphe $G = (V, E)$ et un sommet $v \in V$,*

$$\alpha(\mathring{G}(v)) = \alpha(G) - 1.$$

Preuve : Nous allons montrer qu'on peut construire, à partir d'un indépendant maximum de G, un indépendant de $\mathring{G}(v)$ dont la taille est diminuée de 1. Puis nous allons montrer qu'à partir d'un indépendant maximum de $\mathring{G}(v)$, on peut obtenir un indépendant pour G de taille strictement supérieure.

Soit S un indépendant maximum de G. Si $v \in S$, $S\backslash\{v\}$ est un indépendant de $\mathring{G}(v)$. Autrement, S contient au moins un sommet de $N(v)$, car S est de cardinalité maximum. Dans le cas où S ne contient qu'un sommet de $N(v)$: $N(v) \cap S = \{u\}$, $S\backslash\{u\}$ est un indépendant de $\mathring{G}(v)$. Dans le cas de plusieurs sommets : $N(v) \cap S = \{u_1, u_2, \ldots, u_l\}$, $G\langle\{u_1, u_2, \ldots, u_l\}\rangle$ est un anti-K_l. D'après l'étape 3b du pliage généralisé, un anti-K_l est transformé en un anti-K_{l-1} entre $\{u_{l1}, u_{l2}, \ldots, u_{l(l-1)}\}$. Alors $S \cup \{u_{l1}, u_{l2}, \ldots, u_{l(l-1)}\}\backslash\{u_1, u_2, \ldots, u_l\}$ est un indépendant de $\mathring{G}(v)$. À partir de tous les cas listés, on en déduit que $\alpha(G) \leq 1 + \alpha(\mathring{G}(v)$.

Un argument similaire permet de montrer la réciproque. Soit S un indépendant maximum de $\mathring{G}(v)$. Si $S \cap \{u_{ij} | 1 \leq j < i \leq d(v)\} = \{u_{i_1 j_1}, u_{i_2 j_2}, \ldots, u_{i_l j_l}\}$, le graphe induit est un anti-K_l. Or, d'après l'étape 3b, il ne peut y avoir des anti-arêtes qu'entre des sommets $(u_{i_0 j})_{0 \leq j < i_0}$ pour un i_0 fixé, donc $i_1 = i_2 = \ldots = i_l$. Ceci implique que $\{u_{i_1}, u_{j_1}, u_{j_2}, \ldots, u_{j_l}\}$ forment un anti-K_{l+1}. On en déduit que $S \cup \{u_{i_1}, u_{j_1}, u_{j_2}, \ldots, u_{j_l}\} \setminus \{u_{i_1 j_1}, u_{i_1 j_2}, \ldots, u_{i_1 j_l}\}$ est un indépendant dans G. Il s'en suit que $\alpha(G) \geq 1 + \alpha(\mathring{G}(v)$. ■

Corollaire 5.9 *Pour tout sommet v tel que $|N[v]| \geq |E(\overline{G\langle N(v)\rangle})|$, le pliage généralisé de v est une réduction du problème d'indépendant maximum.*

Cette réduction n'augmente pas la taille de l'instance et diminue toujours la taille de la solution, d'après le théorème 5.8. En effet, le pliage généralisé de v transforme chaque non-arête du voisinage de v en une nouvelle arête, puis supprime v et son voisinage. $|E(G)| = |E(\mathring{G}(v))| - |E(\overline{G\langle N(v)\rangle})| + |N[v]|$. Pour que le nouveau graphe soit plus petit (en nombre de sommets), il faut donc que, dans le voisinage fermé de v, le nombre de non-arêtes soit inférieur au nombre de sommets.

2.4 Application au problème du transversal minimum

Comme l'indépendant maximum est le complémentaire du transversal minimum (Gallai 1959), on peut appliquer les résultats précédents à cet autre problème.

Théorème 5.10 *Pour un graphe $G = (V, E)$ et un sommet $v \in V$,*

$$\tau(G) = \tau(\mathring{G}(v)) + |N(v)| - |E(\overline{G\langle N(v)\rangle})|$$

On déduit ce théorème du théorème 5.8, en y appliquant l'identité de la propriété 5.4.

Corollaire 5.11 *Pour tout sommet v tel que $|N(v)| \geq |E(\overline{G\langle N(v)\rangle})|$, le pliage généralisé de v est une réduction du problème du transversal minimum.*

Cette réduction diminue toujours la taille de l'instance. Comme la taille du transversal varie toujours d'un de plus que la taille de l'instance, elle n'augmente jamais après réduction. Cette réduction est un peu plus stricte que celle pour l'indépendant maximum dans le corollaire 5.9 qui porte sur le voisinage fermé de v. La réduction pour le transversal minimum porte donc sur un peu moins de sommets que celle pour l'indépendant maximum.

2.5 Bilan et perspectives

En généralisant une transformation de graphe pour diminuer la taille de l'indépendant maximum, nous avons retrouvé une transformation connue qui est la *struction*. Nous en avons déduit une réduction qui diminue strictement la taille de la solution, sans augmenter celle de l'instance, pour le problème d'indépendant maximum.

En transposant ces résultats au problème complémentaire, qui est le transversal minimum, nous avons trouvé une réduction qui diminue strictement la taille de l'instance. Et, cette réduction diminue ou laisse invariante la taille de la solution.

L'étape suivante serait maintenant d'appliquer ces réductions. Fomin et al. (2006) présentent un algorithme exponentiel pour le problème d'indépendant maximum, en s'appuyant sur le pliage. L'étude pourrait être adaptée pour proposer un nouvel algorithme s'appuyant sur le pliage généralisé.

Pour le problème du transversal minimum, on pourrait évaluer l'impact de cette réduction dans la résolution du problème. Théoriquement, on pourrait l'associer à la réduction de Buss, sur les sommets de degré supérieur à k, pour obtenir un algorithme FPT. Avec des règles de réduction supplémentaires, on pourrait espérer obtenir un noyau polynomial plus petit que l'algorithme classique de Buss. Empiriquement, on pourrait mesurer l'influence de cette réduction en pré-traitement d'autres méthodes de résolution.

3 Problème du nombre enveloppe

Le problème du nombre enveloppe est un problème combinatoire sur les graphes proposé par Everett and Seidman (1985). Il repose sur une notion de convexité dans les graphes. Nous allons présenter ce problème à travers plusieurs définitions et donner quelques résultats de la littérature. Puis nous allons présenter nos travaux sur des opérations de réduction d'instances et des applications.

3.1 Présentation du problème

Le *nombre enveloppe* (géodésique) d'un graphe G est le nombre de sommets du plus petit sous-graphe de G dont G est l'enveloppe convexe. En géométrie, on peut faire l'analogie avec le nombre de sommets d'un polytope. À partir de ces éléments, on peut reconstruire l'ensemble convexe. Tout segment dont les extrémités sont dans l'ensemble convexe appartient aussi à l'ensemble. Dans notre définition de la convexité dans les graphes, le segment est représenté par le plus court chemin entre deux sommets.

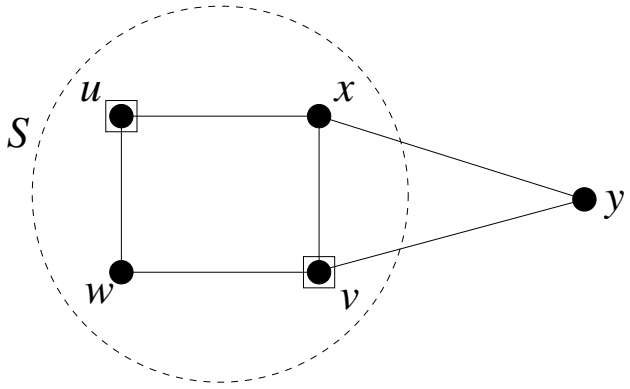

FIGURE 5.3: Exemple d'un sous-ensemble connexe S qui est l'intervalle fermé $I[u, v]$.

Soit un graphe connexe non orienté $G = (V, E)$. Pour deux sommets $u, v \in V$, l'*intervalle fermé* $I[u, v]$ de u et v est l'ensemble des sommets qui appartiennent à un plus court chemin entre u et v dans G. Si on considère l'intervalle fermé de u et v dans le sous-graphe H de G, on note $I_H[u, v]$.

Pour tout sous-ensemble $S \subseteq V$, on note la fermeture $I[S] = \bigcup_{u,v \in S} I[u, v]$. On dit qu'un sous-ensemble $S \subseteq V$ est (géodésiquement) *convexe* si $I[S] = S$ (cf. figure 5.3). On peut remarquer qu'on a toujours $S \subseteq I[S]$. Quand on se restreint au sous-graphe H, avec $S \subseteq V(H)$, on note $I_H[S]$.

Pour un sous-ensemble $S \subseteq V$, l'*enveloppe convexe* $I_{env}[S]$ de S est le plus petit ensemble convexe qui contient S. On dit qu'un sommet v est *généré* par l'ensemble S si $v \in I_{env}[S]$. En partant de S, une itération de fermetures peut être nécessaire pour générer l'ensemble convexe $I_{env}[S]$ en s'appuyant sur les sommets générés par les fermetures précédentes. Le nombre de sommets générés est strictement croissant jusqu'à ce que tous les sommets soient générés. On note $I^i[S]$ le graphe obtenu après i fermetures, avec $I^i[S] = I[I^{i-1}[S]]$ et $I^0[S] = S$. On dit qu'un sommet v est généré à la fermeture $i \geq 1$ si $v \in I^i[S]$ et $v \notin I^{i-1}[S]$. Le nombre de fermetures pour obtenir $I_{env}[S]$ est borné par $|V(S)|$.

Le sous-ensemble $S \subseteq V$ est appelé *ensemble enveloppe* de G si $I_{env}[S] = V$. La taille d'un ensemble enveloppe minimum de G est le *nombre enveloppe* de G, noté $hn(G)$.

Problème du nombre enveloppe
INSTANCE : un graphe $G = (V, E)$
QUESTION : Trouver la taille d'un ensemble enveloppe minimum de G

Ce problème est *NP*-difficile d'après Araujo et al. (2013), qui montrent en particulier que le problème est déjà *NP*-difficile pour les graphes bipartis.

L'exemple d'un ensemble enveloppe minimum et des fermetures successives pour générer les autres sommets du graphe est donné figure 5.4.

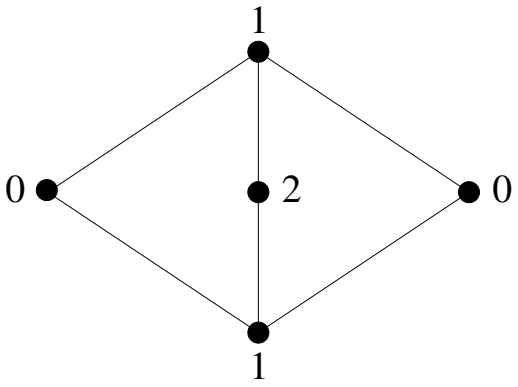

FIGURE 5.4: Exemple d'un graphe avec un nombre enveloppe de taille 2. Les sommets numérotés 0 forment un ensemble enveloppe. Les sommets numérotés 1 sont générés par la première fermeture et le sommet numéroté 2 par la seconde.

3.2 Quelques résultats connus

Voici des résultats de base tirés de la littérature, sur lesquels nous nous appuyons dans la suite. Nous commençons par préciser quelques définitions.

La *distance* $d_G(u, v)$ entre u et v dans le graphe G est la longueur en nombre d'arêtes du plus court chemin entre u et v. Un sous-graphe $H \subseteq G$ est *isométrique* si, pour tout $u, v \in V(H)$, $d_H(u, v) = d_G(u, v)$.

Propriété 5.12 *(Everett and Seidman 1985) Tout ensemble enveloppe de G contient tous les sommets simpliciaux de G.*

Propriété 5.13 *(Dourado et al. 2009) Soient un graphe G, un sous-graphe H isométrique de G et un ensemble enveloppe S de H. Alors, l'enveloppe convexe de S dans G contient $V(H)$.*

3.3 Réduction d'instances

Nous proposons trois règles pour réduire une instance du problème du nombre enveloppe, qui portent sur des sommets avec le même voisinage. Les sommets u et v sont *jumeaux* si $N(u)\backslash\{v\} = N(v)\backslash\{u\}$. Si u et v sont adjacents, on parle de *vrais* jumeaux, sinon de *faux* jumeaux.

Ces réductions reposent sur la transformation d'*identification* (ou de contraction) de deux sommets $u, v \in V$. Celle-ci peut être vue comme la fusion des deux sommets en un nouveau qui hérite de l'union de leurs voisinages.

Définition 5.14 *Voici les étapes de l'identification de u et v.*

1. *Ajouter un sommet* s_{uv} *;*
2. *Ajouter les arêtes* $\{s_{uv}w \mid w \in N(u) \cup N(v)\}$ *;*
3. *Retirer* u *et* v*.*

Dans les cas suivants de notre étude, les sommets identifiés sont toujours jumeaux. Ainsi la transformation revient juste à retirer un des deux sommets. On considère, sans perte de généralité, que c'est v et donc que $s_{uv} = u$. On déduit aussi la propriété suivante sur le résultat d'une identification, que nous utilisons par la suite dans les démonstrations des réductions.

Propriété 5.15 *Soient un graphe* G *et* $u, v \in V$ *des sommets jumeaux. Soit* H *le résultat de l'identification de* u *et* v *dans* G*. Alors* H *est isométrique de* G*.*

Preuve : H est isomorphe à $V(G)\backslash\{v\}$. Pour tout plus court chemin de G contenant v, remplacer v par u donne aussi un plus court chemin de G. Ainsi les distances entre deux sommets de $V(G)\backslash\{v\}$ sont les mêmes dans G et H. ■

Avec ces pré-requis, nous pouvons maintenant énoncer les résultats de notre étude. Les transformations suivantes sont toutes des réductions, car la taille de l'instance diminue strictement et la taille de la solution décroît ou reste constante.

Lemme 5.16 *Soient un graphe* G *et* $u, v \in V$ *des sommets non simpliciaux et jumeaux. Soit* H *le résultat de l'identification de* u *et* v *dans* G *(cf. figure 5.5). Alors* $hn(G) = hn(H)$*.*

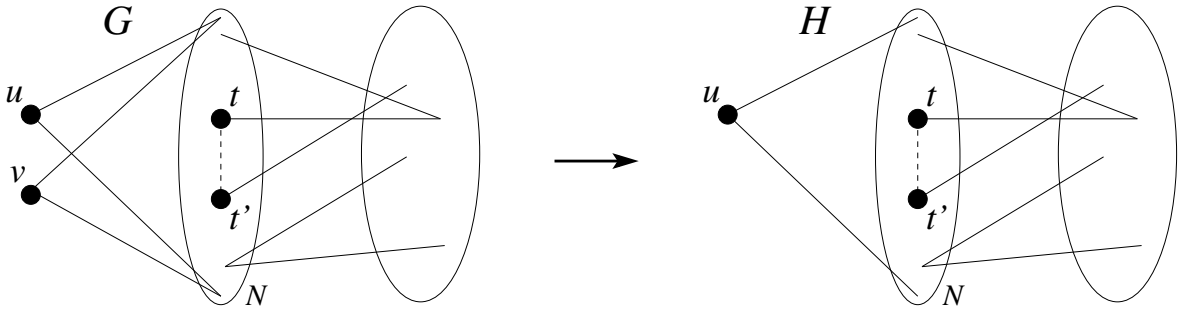

FIGURE 5.5: Identification de deux sommets (u, v) non simpliciaux et jumeaux. N est le voisinage des sommets u et v ; tt' est une non arête.

Preuve : Soient $t, t' \in V(G)$ des sommets non adjacents appartenant au voisinage de u et de v jumeaux non simpliciaux.

Montrons tout d'abord que $hn(G) \leq hn(H)$. Soit S un ensemble enveloppe minimum de H. Comme H est un sous-graphe isométrique de G, on déduit d'après

la propriété 5.13 que $V(G)\backslash\{v\}$ est généré par S. Or $v \in I_G[t,t']$, donc v est aussi généré par S au pire avec une fermeture supplémentaire à celles nécessaires pour générer t et t'. Ainsi S est un ensemble enveloppe de G.

Montrons ensuite que $hn(G) \geq hn(H)$. Soit S un ensemble enveloppe minimum de G. On montre que v a le même comportement que u pour les fermetures successives de S dans G, pour en déduire que S est un ensemble enveloppe de H. On peut admettre que S ne contient pas u et v en même temps. En effet, s'il existe un ensemble enveloppe minimum qui contient u et v, alors il en existe aussi un qui contient t et t' à la place. En effet, $u, v \in I_G[t,t']$, puis u et v pourront éventuellement générer les autres sommets lors des fermetures suivantes.

D'une part, on suppose que $u, v \notin S$. Soient $x, y \in V$ tels que de $\{x,y\} \neq \{u,v\}$ et P un plus court chemin entre x et y. On observe que P ne peut pas contenir à la fois u et v. Dans le cas où P contient u (resp. v), en le remplaçant par v (resp. u), on obtient un autre plus court chemin, car u et v ont le même voisinage. En particulier, cela implique que u et v sont générés par la même fermeture k et que pour tout $i \leq k$, $I^i_H[S] = I^i_G[S]$. Mais aussi, cela implique que les plus courts chemins dont une seule extrémité est u ou v, génèrent les mêmes autres sommets qu'eux-mêmes dans G. Ainsi, pour tout $i > k$, $I^i_H[S] = I^i_G[S]\backslash\{v\}$. Par conséquent, S est un ensemble enveloppe de H.

D'autre part, on suppose qu'un seul sommet entre u et v appartient à S. S'il existe un ensemble enveloppe minimum qui contient l'un, il existe aussi un ensemble qui contient l'autre. On choisit $u \in S$. Par le même argument qu'à la fin du paragraphe précédent, on en déduit que pour tout $i \in [\![1, |V(G)|]\!]$, $I^i_H[S] = I^i_G[S]\backslash\{v\}$ et donc, dans ce cas aussi, S est un ensemble enveloppe de H. ■

Lemme 5.17 *Soient un graphe G et $u, v, w \in V$ des sommets simpliciaux et deux à deux faux jumeaux. Soit H le résultat de l'identification de u et w dans G (cf. figure 5.6). Alors $hn(G) = hn(H) + 1$.*

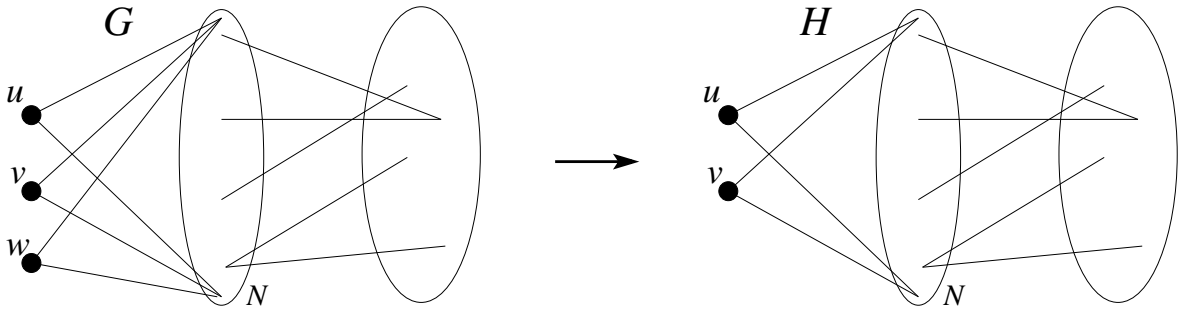

FIGURE 5.6: Identification de deux sommets (u, w) simpliciaux et faux jumeaux. N est le voisinage des sommets u, v et w.

Preuve : Pour montrer que $hn(G) \leq hn(H)+1$, on utilise que H est un sous-graphe isométrique de G. Par conséquent, pour tout ensemble enveloppe S de H, d'après la propriété 5.13, $I_H[S] = V(G)\backslash\{w\}$. Donc $S \cup \{w\}$ est un ensemble enveloppe de G.

Montrons maintenant que $hn(G) \geq hn(H) + 1$. Soit S un ensemble enveloppe minimum de G. D'après la propriété 5.12, les sommets simpliciaux $\{u, v, w\} \subseteq S$. On détaille les cas où w appartient à un plus court chemin de G pour monter que u et v sont suffisants pour générer les sommets appartenant à ces plus courts chemins. Un sommet simplicial n'appartient à un plus court chemin que s'il en est une extrémité. Dans G, tout plus court chemin entre w et $x \notin \{v, w\}$ reste un plus court chemin si on remplace w par v. Ainsi $I[w, x] = I[v, x]$ pour $x \in V\backslash\{v, w\}$. Dans le cas d'un plus court chemin entre w et v, remplacer w par u donne encore un plus court chemin et $I[w, v] = I[u, v]$. Par conséquent, $I_{env}(S\backslash\{w\}) = I_{env}(S)\backslash\{w\}$ et $S\backslash\{w\}$ est un ensemble enveloppe de H. ■

Lemme 5.18 *Soient un graphe G et $u, v \in V$ des sommets simpliciaux et vrais jumeaux. Soit H le résultat de l'identification de u et v dans G (cf. figure 5.7). Alors $hn(G) = hn(H) + 1$.*

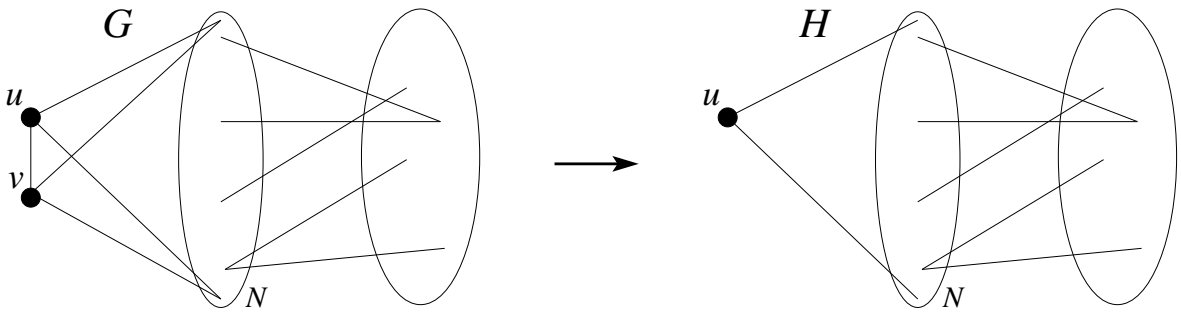

FIGURE 5.7: Identification de deux sommets (u, v) simpliciaux et vrais jumeaux. N est le voisinage des sommets u et v.

Preuve : Comme dans la preuve du lemme 5.17, $hn(G) \leq hn(H) + 1$.

Montrons ensuite que $hn(G) \geq hn(H) + 1$. Soit S un ensemble enveloppe minimum de G. D'après la propriété 5.12, les sommets simpliciaux $\{u, v\} \subseteq S$. Comme u et v sont adjacents et qu'ils ont le même voisinage, pour tout $x \in V(H)$, $I_G[v, x]\backslash\{v\} \subseteq I_H[u, x]$. Donc, $S\backslash\{v\}$ est un ensemble enveloppe de H. ■

Le lemme 5.17 ne peut pas être simplifié pour réduire une paire de faux jumeaux simpliciaux. En effet, c'est la seule des configurations étudiées où deux jumeaux sont nécessaires dans l'ensemble enveloppe minimum pour générer d'autres sommets du graphe. Dans le cas de deux vrais jumeaux simpliciaux, ils ne génèrent qu'eux-même, et dans le cas de deux jumeaux non simpliciaux, ils peuvent être remplacés par deux sommets non adjacents de leur voisinage.

On peut illustrer cette remarque avec l'exemple d'un graphe complet avec au moins 3 sommets, dont une arête uv est retirée. On constate que u et v sont des faux jumeaux simpliciaux et qu'ils sont les seuls sommets de l'ensemble enveloppe minimum. On ne peut donc pas réduire ce graphe.

3.4 Applications des réductions

Grâce à ces réductions, nous pouvons montrer que le problème est FPT pour le paramètre de diversité de voisinage, et que les réductions permettent de caractériser le problème pour le produit lexicographe de graphes.

Lampis (2012) définit la *diversité de voisinage* d'un graphe. Elle vaut k si les sommets du graphe peuvent être partitionnés en k sous-ensembles $S_1, S_2, \ldots, S_k$, tels que toute paire de sommets d'un même sous-ensemble forme des jumeaux.

Théorème 5.19 *Soit un graphe G dont la diversité de voisinage vaut au plus k. Il existe alors un algorithme FPT pour calculer $hn(G)$ de complexité $\mathcal{O}(4^k poly(|V(G)|))$.*

Preuve : Lampis (2012) prouve que la diversité de voisinage de G peut être calculée en $\mathcal{O}(poly(|V(G)|))$. Par définition, chaque élément de la partition est un ensemble de jumeaux, soit un indépendant de faux jumeaux, soit une clique de vrais jumeaux. On peut appliquer les lemmes 5.16, 5.17 et 5.18 pour réduire chaque élément de la partition qui aura au plus deux sommets.

Tout d'abord, on réduit les ensembles de sommets non simpliciaux par identification des sommets jusqu'à ce qu'il n'en reste qu'un, pour obtenir un graphe F. D'après le lemme 5.16, $hn(G) = hn(F)$.

Si un élément de la partition de G contient un sommet simplicial, alors tous les sommets de l'élément sont simpliciaux. On réduit ensuite les ensembles avec au moins trois faux jumeaux simpliciaux en appliquant le lemme 5.17 jusqu'à ce qu'il ne reste que deux sommets dans chaque élément de la partition. Si c identifications ont été appliquées sur F pour obtenir ce graphe F', $hn(F') = hn(F) + c$.

Enfin, on réduit les ensembles composés de vrais jumeaux simpliciaux en un sommet par identification, pour obtenir un graphe F''. D'après le lemme 5.18, $hn(F'') = hn(F') + c'$, où c' est le nombre d'identifications qui ont été appliquées sur F'.

On déduit que F'' a au plus $2k$ sommets, car au plus deux sommets par élément de la partition, dont le nombre est donné par la valeur de la diversité de voisinage. On peut à ce niveau énumérer tous les sous-ensembles de $V(F'')$, soit au plus 2^{2k}, et tester pour chacun d'eux s'il est un ensemble enveloppe. La taille du plus petit ensemble nous donne $hn(F'')$ et on en déduit $hn(G) = hn(F'') + c' + c''$.

Cette preuve traduit un algorithme de nucléarisation avec un noyau dont la taille est linéaire et vaut $2k$. ■

Le *produit lexicographique* de deux graphes G et H est le graphe dont les sommets sont $V(G \circ H) = V(G) \times V(H)$ et deux sommets (g_1, h_1) et (g_2, h_2) sont adjacents si, et seulement si, $g_1 g_2 \in E(G)$ ou si $g_1 = g_2$ et $h_1 h_2 \in E(G)$. On appelle la *H-strate* de $g \in V(G)$ dans $G \circ H$, l'ensemble $H(g) = \{(g, h) \in V(G) \circ H) \mid h \in V(H)\}$. Soit $S(G)$ l'ensemble des sommets simpliciaux de G.

Si G n'a qu'un seul sommet, alors $hn(G \circ H) = hn(H)$. Sinon on a le résultat suivant.

Théorème 5.20 *Soient un graphe G connexe avec $|V(G)| \geq 2$ et un graphe H quelconque,*

$$hn(G \circ H) = \begin{cases} 2, & \text{si } H \text{ n'est pas complet ;} \\ (|V(H)| - 1)|S(G)| + hn(G), & \text{sinon.} \end{cases}$$

Preuve : Si H n'est pas complet, comme G est connexe et $|V(G)| \geq 2$, deux sommets non adjacents dans la même H-strate suffisent pour générer tous les sommets de $G \circ H$.

On suppose maintenant que H est un graphe complet à k sommets. Les sommets d'une même H-strate sont soit tous simpliciaux soit aucun. De plus, un sommet est simplicial dans G si, et seulement si, la H-strate correspondante dans $G \circ H$ est composée de sommets simpliciaux.

Nous réduisons d'abord chaque H-strate de $G \circ H$ composée de sommets non simpliciaux, en un unique sommet, pour obtenir un graphe F. D'après le lemme 5.16, $hn(G \circ H) = hn(F)$. Nous réduisons ensuite les H-strates restantes, qui sont composées de vrais jumeaux simpliciaux, en un unique sommet, pour obtenir un graphe F'. Il y a $|V(H)||S(G)|$ sommets simpliciaux dans $G \circ H$. Donc F est réduit par $(|V(H)| - 1)|S(G)|$ identifications. D'après le lemme 5.18, $hn(F) = hn(F') + (|V(H)| - 1)|S(G)|$. Comme toutes les H-strates ont été réduites en un unique sommet, F' est isomorphe à G et donc $hn(G \circ H) = hn(G) + (|V(H)| - 1)|S(G)|$. ■

3.5 Bilan et perspectives

Ces résultats ont été soumis dans Araujo et al. (2012). Y figurent également des résultats de complexité pour la classe des graphes sans P_5 et sans triangle, pour laquelle nous avons proposé un algorithme polynomial.

Nous avons présenté trois règles de réduction pour le problème du nombre enveloppe. Pour être des réductions, celles-ci diminuent strictement la taille de l'instance.

De plus, elles diminuent ou laissent constante la taille de la solution. Du point de vue théorique, elles nous ont permis d'aboutir à des résultats sur la complexité du problème qui est FPT pour le paramètre de diversité de voisinage, qui plus est avec un noyau linéaire. Ces résultats permettent également d'en déduire un algorithme FPT selon la taille du transversal minimum, qui donne une borne sur la taille de la diversité de voisinage. Il est encore ouvert de savoir si le problème est FPT selon la taille de la solution.

Les réductions portent sur des sommets jumeaux, c'est-à-dire avec le même voisinage. Nous pourrions essayer de généraliser les réductions dans le cas de domination entre sommets, c'est-à-dire que le voisinage d'un sommet est inclus dans celui d'un autre.

Même en sachant que certains sommets doivent appartenir à l'ensemble enveloppe minimum, comme pour le lemme 5.17, il n'est pas possible de réduire d'avantage l'instance, car leur présence est nécessaire pour générer d'autres sommets. Pour palier cette limitation, on pourrait utiliser la variante enrichie du problème généralisé, proposée par Araujo et al. (2013), en ajoutant une liste de sommets qui sont déjà dans l'ensemble enveloppe, ce qui permettrait de pousser plus loin la réduction.

Conclusion

Dans ce chapitre, nous avons présenté des résultats en lien avec des problèmes théoriques sur la complexité autour de la notion de FPT. Nous avons donné une définition de la réduction d'un point de vue pratique, qui englobe la notion présentée dans le chapitre 4. Ces réductions permettent de réduire les instances pour se ramener au cœur du problème. Selon Niedermeier (2006), "le leitmotiv de la conception et de l'analyse d'algorithmes paramétrés est de chercher une meilleure explication de ce qui cause la difficulté des problèmes *NP*-complets".

La première étude sur les problèmes d'indépendant minimum et du transversal maximum présente des résultats pour définir une nouvelle réduction. Nous avons donné une étude théorique qui mériterait maintenant d'être appliquée à d'autres études. D'une part, en s'inspirant des travaux de Fomin et al. (2006) qui utilisent le pliage, on peut intégrer la version généralisée au modèle et étudier les performances de l'algorithme exponentiel pour le problème d'indépendant minimal. D'autre part, le pliage généralisé pour le problème du transversal maximum, qui est FPT, peut être une piste intéressante pour concevoir un nouvel algorithme de nucléarisation. Les motivations sous-jacentes sont d'évaluer l'impact de la réduction par rapport aux autres méthodes déjà connues.

La seconde étude sur le problème du nombre enveloppe a permis d'identifier des règles de réduction qui sont devenues des outils pour en déduire des résultats plus généraux sur la résolution du problème. Ainsi nous avons conçu un algorithme FPT

selon le paramètre de diversité de voisinage. La question de savoir si le problème est FPT selon la taille de l'instance est par contre toujours ouvert. Nous pensons que les réductions peuvent être généralisées à d'autres configurations de sommets, comme lorsque le voisinage d'un sommet est inclus dans celui d'un autre, ce qui permettrait peut-être de résoudre de nouvelles classes du problème.

Le rôle pratique des réductions peut permettre d'expérimenter leurs impacts sur les instances. Par exemple, Fleischer et al. (2009) présentent une étude expérimentale d'une réduction pour le problème de *Directed Feedback Vertex Set* inspirée d'un algorithme de nucléarisation polynomial pour le problème non orienté et en déduisent empiriquement qu'elle ne permet pas d'obtenir les mêmes résultats. Cette approche expérimentale peut donc permettre une étude exploratrice sur des règles de réduction pour orienter une éventuelle étude théorique ultérieure.

Conclusion et perspectives

Le sujet central de la thèse est la notion d'**instance**. Le but est de mieux comprendre la difficulté des problèmes *NP*-difficiles à travers l'étude de leurs instances. Pour cela, nous avons défini la notion de **difficulté** d'instances, qui repose sur la comparaison entre classes d'instances.

Les études que nous avons menées sur les instances peuvent être classées selon deux types : l'**évaluation** d'instances et la **manipulation** d'instances. Nous présentons une méthode pour comparer des classes d'instances entre elles à partir d'un jeu de test composé d'algorithmes. Cette méthode a des applications pour évaluer empiriquement la difficulté d'instance selon le contexte d'étude. Notre étude porte aussi sur la pertinence de critères de mesure, en particulier la mesure de la qualité de la solution par ratio à l'optimum. La manipulation d'instances se fait par le biais de modifications d'instances. Elle permet de générer des instances difficiles ou d'aider à la résolution des problèmes.

Principaux résultats

Dans la thèse, nous avons jonglé entre approches pratiques et réflexions théoriques. Nous y avons abordé plusieurs problèmes : voyageur de commerce, transversal minimum, indépendant maximum, nombre enveloppe, couplage fractionnaire parfait. Nous en rappelons les principaux résultats.

Unification de la littérature sur la difficulté d'instance

Dans la littérature, la notion de difficulté d'instance est rarement explicite et peut prendre différentes formes. Nous avons dressé un portrait de la notion telle qu'elle peut être rencontrée dans la littérature, à travers la classification suivante, que nous avons proposée. La difficulté d'instance dépend du référentiel composé de méthodes de résolution, de la portée de la difficulté pour le problème, de la mesure des performances et du contexte d'utilisation. Nous avons identifié trois contextes d'utilisation : la validation de la difficulté d'instance, la génération d'instances diffi-

ciles et la caractérisation de l'origine de la difficulté. Nous avons décrit les différents domaines où la difficulté d'instance est rencontrée et comment elle s'exprime : transition de phase, évolution d'instances, analyse de paysage, algorithmes portfolii, complexité d'instance...

Méthode de comparaison de classes d'instances

Nous avons présenté une méthode pratique pour évaluer les classes d'instances. Cette approche repose sur la comparaison empirique de deux classes. Elle dépend de méthodes de résolution auxquelles sont associés des critères de performance. La méthode se ramène à du benchmarking d'instances avec un jeu de test composé d'algorithmes. Le déroulement de la méthode est décrite avec précision et rigueur. La méthode prend en compte la variabilité des classes et des algorithmes employés. À travers des instances représentatives, elle évalue le comportement des classes d'instances pour le contexte d'étude. La comparaison entre les deux classes s'appuie sur des représentations graphiques de la mesure de difficulté d'instance.

Cette approche est une brique de base pour des études plus poussées, telles qu'illustrées dans le chapitre 3 avec des études pour le problème du voyageur de commerce. D'une part, elle apporte un cadre formalisé pour les études expérimentales, en validant le choix des instances utilisées comme base de test. D'autre part, elle permet de mener, à travers un protocole expérimental, des études sur les sources de la difficulté des problèmes pour les méthodes de résolution de l'étude. Ainsi, pour le problème du voyageur de commerce, nous avons observé que les performances des algorithmes gloutons dépendent de la structure des instances et de la loi de distribution des distances.

Modification d'instances pour le TSP

Nous avons mené une étude théorique sur les pratiques empiriques en ciblant un critère de performance usuel pour mesurer la qualité des solutions. Pour une méthode de résolution, la mesure du ratio entre la valeur de la solution retournée et la valeur optimale n'est pas une mesure robuste. Nous avons vu que des instances équivalentes au niveau de la difficulté de résolution peuvent donner des performances très différentes suivant ce critère. Appliquée aux instances habituellement utilisées dans la pratique du benchmarking, cette observation remet en cause les performances pratiques d'algorithmes classiques pour le problème du voyageur de commerce et donc aussi leur efficacité.

L'invariance de la solution optimale par la modification d'instances par offset nous a permis de définir une méthode pour générer des instances difficiles, dans le sens où la mesure de performance y est maximale par la modification. Cette méthode repose sur la solution duale du problème de couplage fractionnaire parfait de poids

minimum. Nous en avons donné une caractérisation des solutions dans le cas général et dans le cas où l'instance modifiée est métrique.

La modification d'instances étudiée a également des propriétés intéressantes sur la métricité des instances après modification. Grâce à celles-ci, nous avons présenté pour le problème du voyageur de commerce non métrique un algorithme avec une garantie d'approximation qui est fonction du coût maximum et du coût minimum, et indépendante du nombre de sommets.

Pour remédier à l'instabilité du critère de performance par ratio et conserver les jeux d'instances usuels, nous proposons un nouveau critère de mesure. Ce critère est le ratio de l'écart entre la solution de l'algorithme et l'optimum, par l'optimum de l'instance réduite, qui est une valeur stable par rapport à la modification par offset.

Réductions d'instances

La modification d'instances nous a amené à définir la notion de réduction, qui est une définition pratique de l'opération de "grignotage" d'instances qui consiste à enlever des instances les configurations simples à traiter. Nous avons mené plusieurs études théoriques sur cette notion pour les problèmes du voyageur de commerce, du transversal minimum, d'indépendant maximum et du nombre enveloppe.

Nous avons présenté les conditions pour que le pliage généralisé (*struction*) soit une réduction pour le problème d'indépendant maximum. Nous avons ensuite étudié l'opération comme réduction pour le problème du transversal minimum.

Nous avons aussi présenté trois nouvelles réductions pour le problème du nombre enveloppe. À partir de ces opérations, nous avons prouvé que le problème est FPT selon la diversité de voisinage avec un noyau linéaire et nous avons résolu le cas du produit lexicographique de deux graphes.

Perspectives

Cette thèse sur la notion d'instance a ouvert de nouvelles pistes de recherche. Certains domaines sont déjà explorés comme la réduction d'instances liée aux problèmes FPT. Mais plusieurs questions sur les études expérimentales attendent encore des réponses. Notre méthode est une réponse parmi d'autres possibles qui permet d'évaluer la pertinence de classes d'instances. En particulier, elle laisse encore en suspens le problème de la génération d'instances difficiles, même si la réduction d'instances peut être une réponse partielle.

Nous avons appliqué notre méthodologie d'évaluation des classes d'instances à quelques exemples du problème du voyageur de commerce. Cette méthode générique

mériterait maintenant d'être appliquée à d'autres problèmes. Cela passe par une diffusion de la méthode et une sensibilisation de la communauté à son utilité et à l'intérêt de la développer. Pour faciliter sa mise en œuvre, on peut prévoir une solution logicielle, en enrichissant par exemple l'outil pour le benchmarking que nous avons conçu pendant la thèse (annexe B).

Nous avons présenté une réduction d'instances pour le problème du voyageur de commerce qui permet de se ramener à une mesure stable pour évaluer la difficulté d'instance. Or cette opération ne conserve pas la métricité qui permettrait la comparaison avec des algorithmes à garantie de performance pour le problème métrique. Ainsi, on suggère de développer une nouvelle réduction d'instances telle que les instances obtenues soient métriques et le ratio de performance maximum. Si les résultats sont similaires à ceux obtenus pour la réduction étudiée, ils permettraient de caractériser les instances ainsi réduites, ce qui apporterait de nouveaux éléments pour l'étude des algorithmes d'approximation pour le problème métrique.

Plus globalement, l'instabilité de la mesure peut être rencontrée dans d'autres problèmes. Tout problème pour lequel il existe une réduction d'instances y est sensible, car l'opération permet de modifier artificiellement les critères de performance de type ratio à l'optimum en modifiant la valeur de la solution optimale. Pour tous ces problèmes, les mesures de performance devraient donc prendre en compte cette instabilité, en s'appuyant sur une instance réduite dont la solution optimale est minimale par modification d'instances. Cela ouvre la porte à de nouveaux problèmes qui modélisent la manière d'appliquer les modifications pour trouver une telle instance réduite.

Notre dernière proposition concerne les algorithmes de nucléarisation et plus généralement les réductions pratiques telles que définies dans le chapitre 5. Les études actuelles sont surtout théoriques. On préconise une plus grande place à l'étude empirique de l'impact des réductions : d'une part en prospection, pour orienter les études théoriques ; d'autre part pour évaluer l'intérêt pratique des réductions en pré-traitement d'autres méthodes de résolution.

Annexe A

Impact de la réduction sur les instances de la TSPLIB

On détaille les résultats obtenus pour chaque instance de la TSPLIB avec moins de 3000 sommets. Le premier tableau (sur 2 pages) décrit les instances euclidienne, sur lesquelles nous avons appliqué un algorithme *2-opt*, dont les performances sont données par γ pour l'instance initiale et par γ_R pour l'instance réduite. Le critère considéré est l'écart à l'optimum normalisé par la valeur optimale.

Nous rappelons que nous avons montré : $\frac{OPT(reduction(I))}{OPT(I)} = \frac{\gamma}{\gamma_R}$

nom de l'instance	$\frac{OPT(reduction(I))}{OPT(I)}$	γ	γ_R	$\frac{\gamma_R}{\gamma}$
berlin52.tsp	0,17	0,06	0,35	6,01
bier127.tsp	0,19	0,22	1,14	5,26
ch130.tsp	0,28	0,07	0,24	3,53
ch150.tsp	0,15	0,07	0,47	6,73
d1291.tsp	0,20	0,12	0,62	5,03
d1655.tsp	0,14	0,14	0,97	7,18
d198.tsp	0,33	0,16	0,48	3,05
d2103.tsp	0,08	0,05	0,60	12,09
d493.tsp	0,13	0,13	0,97	7,42
d657.tsp	0,17	0,17	1,00	5,86
dsj1000.tsp	0,21	0,19	0,91	4,85
eil101.tsp	0,09	0,11	1,21	10,84
eil51.tsp	0,12	0,13	1,12	8,52
eil76.tsp	0,10	0,13	1,30	9,96
fl1400.tsp	0,40	0,15	0,36	2,47
fl1577.tsp	0,25	0,14	0,54	3,98
fl417.tsp	0,37	0,11	0,28	2,67
gil262.tsp	0,19	0,17	0,89	5,21

nom de l'instance	$\frac{OPT(reduction(I))}{OPT(I)}$	γ	γ_R	$\frac{\gamma_R}{\gamma}$
kroA100.tsp	0,20	0,07	0,35	5,07
kroA150.tsp	0,19	0,14	0,73	5,30
kroA200.tsp	0,21	0,11	0,53	4,68
kroB100.tsp	0,24	0,06	0,26	4,14
kroB150.tsp	0,22	0,06	0,28	4,63
kroB200.tsp	0,20	0,13	0,62	4,88
kroC100.tsp	0,19	0,18	0,94	5,17
kroD100.tsp	0,22	0,13	0,58	4,48
kroE100.tsp	0,24	0,14	0,56	4,10
lin105.tsp	0,38	0,18	0,49	2,65
lin318.tsp	0,35	0,11	0,33	2,85
nrw1379.tsp	0,12	0,16	1,33	8,42
p654.tsp	0,32	0,27	0,85	3,11
pcb1173.tsp	0,10	0,16	1,54	9,82
pcb442.tsp	0,08	0,12	1,60	12,86
pr1002.tsp	0,17	0,12	0,68	5,75
pr107.tsp	0,45	0,03	0,07	2,20
pr124.tsp	0,34	0,03	0,10	2,94
pr136.tsp	0,12	0,15	1,31	8,62
pr144.tsp	0,66	0,25	0,38	1,52
pr152.tsp	0,42	0,05	0,13	2,40
pr226.tsp	0,38	0,03	0,07	2,64
pr2392.tsp	0,16	0,16	1,01	6,41
pr264.tsp	0,33	0,11	0,32	3,05
pr299.tsp	0,17	0,28	1,64	5,80
pr439.tsp	0,28	0,12	0,41	3,54
pr76.tsp	0,29	0,06	0,21	3,48
rat195.tsp	0,10	0,10	1,05	10,19
rat575.tsp	0,11	0,13	1,18	8,83
rat783.tsp	0,16	0,15	0,94	6,41
rat99.tsp	0,10	0,13	1,30	9,93
rd100.tsp	0,17	0,13	0,75	5,85
rd400.tsp	0,19	0,14	0,74	5,23
rl1304.tsp	0,25	0,14	0,58	4,01
rl1323.tsp	0,24	0,14	0,59	4,22
rl1889.tsp	0,23	0,19	0,82	4,31
st70.tsp	0,23	0,04	0,18	4,33
ts225.tsp	0,09	0,02	0,25	11,47
tsp225.tsp	0,13	0,19	1,47	7,86
u1060.tsp	0,18	0,15	0,83	5,50
u1432.tsp	0,04	0,17	4,89	28,22
u159.tsp	0,18	0,17	0,99	5,66
u1817.tsp	0,14	0,11	0,76	6,94
u2152.tsp	0,12	0,18	1,45	8,04
u2319.tsp	$1,9.10^{-3}$	0,12	63,15	538,52
u574.tsp	0,21	0,13	0,63	4,73
u724.tsp	0,15	0,11	0,76	6,85
vm1084.tsp	0,20	0,11	0,55	4,99
vm1748.tsp	0,17	0,16	0,92	5,80

Le second tableau décrit les conséquences de la réduction sur les instances non euclidiennes de la TSPLIB. Le ratio entre les optima permet de déduire l'impact sur les performances de méthodes de résolution stable à la modification par offset.

nom de l'instance	$\frac{OPT(reduction(I))}{OPT(I)}$	$\frac{\gamma_R}{\gamma}$
ali535.tsp	0,23	4,38
att48.tsp	0,21	4,83
att532.tsp	0,18	5,65
bayg29.tsp	0,11	9,47
bays29.tsp	0,13	7,89
brazil58.tsp	0,35	2,88
brg180.tsp	1,00	1,00
dantzig42.tsp	0,24	4,19
fri26.tsp	0,11	9,01
gr120.tsp	0,16	6,44
gr137.tsp	0,18	5,46
gr17.tsp	0,21	4,82
gr202.tsp	0,14	7,18
gr21.tsp	0,11	9,43
gr229.tsp	0,18	5,49
gr24.tsp	0,17	5,78
gr431.tsp	0,16	6,11
gr48.tsp	0,18	5,55
gr666.tsp	0,14	7,19
gr96.tsp	0,17	5,93
hk48.tsp	0,14	7,20
pa561.tsp	0,14	7,23
si1032.tsp	0,02	47,46
si175.tsp	0,05	18,39
si535.tsp	0,07	14,93
swiss42.tsp	0,21	4,82
ulysses16.tsp	0,18	5,44
ulysses22.tsp	0,25	4,07

Annexe B

PARROT : Outil pour le benchmarking

C'est un travail en collaboration avec J. Darlay et Y. Kieffer et initié par l'encadrement du stage de Master 1 de A. Noizillier et du stage de Licence 3 de C. Vial.

Nous avons conçu un outil générique pour automatiser l'expérimentation. Il s'appuie sur un fichier décrivant entièrement l'expérience. Nous avons aussi intégré une gestion intelligente de la sauvegarde des résultats.

Le document suivant, soumis au *13^e Congrès Annuel de la Recherche Opérationnelle et d'Aide à la Décision (ROADEF '12)*, présente plus en détail le fonctionnement de l'outil.

Parrot : un Outil pour le Benchmarking Automatisé

Yann Kieffer, Valentin Weber, Julien Darlay

Grenoble-INP / UJF-Grenoble 1 / CNRS, G-SCOP UMR5272
46, avenue Félix Viallet, F-38031 Grenoble, France
{yann.kieffer,valentin.weber,julien.darlay}@g-scop.grenoble-inp.fr

Mots-clés : *benchmarking, automatisation des expériences, généricité*

1 Motivations

Le benchmarking est un aspect important de la recherche opérationnelle. Dès lors qu'on veut évaluer en pratique des algorithmes, il est nécessaire de les exécuter sur de nombreuses données. Voici quelques cas d'utilisation classique, libre à chacun d'y ajouter les siens : comparaison de méthodes de résolution, paramétrage d'algorithmes, data-mining, simulation, etc. Tous ces cas d'utilisation rencontrent les mêmes problématiques, qui apparaissent concernant l'aspect technique de la démarche du benchmarking.

D'une part, la répétitivité de l'exécution intensive d'algorithmes demande généralement la mise en place d'une structure permettant d'automatiser les exécutions, le stockage des résultats et leur exploitation. Cela permet d'augmenter également la robustesse du procédé, en limitant les erreurs de manipulation. Quelques outils logiciels existent, mais dans des cadres d'utilisation assez précis. Malheureusement, il n'existe pas à notre connaissance d'outils logiciels dédiés au lancement multiple d'exécutions dans un contexte générique. La solution qui semble couramment utilisée est un ensemble spécifique de scripts, souvent faiblement documentés, et difficiles à maintenir ou à étendre lors de nouvelles expérimentations.

D'autre part, comme le suggère Johnson [2], la reproductibilité des expériences est un élément clef pour valoriser les résultats expérimentaux. En effet, pour être confronté à d'autres résultats, toute la démarche expérimentale doit être décrite et accessible pour permettre à chacun de la renouveler dans son propre environnement de travail ; en particulier, pour comparer les temps d'exécution des programmes qui dépendent directement des machines où ils sont exécutés.

Il apparaît donc un besoin, celui d'un outil offrant la possibilité de lancer des exécutions nombreuses. Dans la continuité des travaux de Bentley [1], nous proposons un nouvel outil qui répond aux problématiques du benchmarking actuel. Nous encodons de manière synthétique le plan d'expérimentation en décrivant les variations entre les différentes exécutions. Outre la disponibilité qu'apporte une telle solution, elle permet d'envisager des fonctionnalités étendues, difficilement imaginables dans un système de scripts ad-hoc. Un exemple typique est la reprise d'une longue série d'exécutions après un arrêt, quel qu'en soit la cause : s'il est possible de ne reprendre que les calculs qui n'ont pas abouti, le gain de temps peut être très appréciable.

2 Fonctionnalités et contraintes

Dans le logiciel que nous avons imaginé, certaines caractéristiques ont vite paru souhaitables.

Tout d'abord, nous visons la généricité. L'outil est d'autant plus pertinent qu'il est largement utilisable. C'est le même souci qui a mené aux impératifs du caractère multi-plateforme, à envisager une diffusion sous forme de logiciel libre, et à assurer une bonne mesure d'extensibilité et de simplicité.

Ces contraintes étant posées, la première fonctionnalité visée est la possibilité de lancer des exécutions variées en créant des lignes de commandes, une par appel. Seuls les paramètres de

ligne de commande varient d'un appel à l'autre. Quitte à encapsuler l'applicatif dans une boîte noire qui manipule plus avant son environnement, cela permet aussi de mettre en place des variables d'environnement ou des fichiers de configuration.

Afin de décrire les variations des paramètres, un type de fichier XML a été défini. Ce fichier permet de décrire complètement l'ensemble du plan d'expériences. A partir de cela, il est possible de reproduire la même expérience dans différents environnements d'exécution.

Pour décrire l'expérience, nous avons développé un langage simple de description des lignes de commande. Ce langage permet l'assemblage de lignes de commande avec des éléments fixes et variables, que l'on peut grouper, rendre optionnels, et accoler. Les séquences arithmétiques et géométriques d'entiers sont fournies, ainsi que les énumérations de chaînes. Enfin, il est possible de faire référence à un élément de la ligne, de manière à réutiliser la valeur qu'il a reçu dans la ligne courante. Ce langage couvre ainsi un large spectre de cas d'utilisation.

La seconde fonctionnalité est l'archivage des résultats. Cela suppose un nommage unique des expériences, et le choix d'un mode de sortie : conservation des fichiers dans des répertoires ; agrégation de valeurs dans une table ou une base de données.

La troisième fonctionnalité est la gestion des exécutions individuelles. Chaque exécution a lieu dans un répertoire créé pour cette exécution. Pour ne pas limiter les usages possibles, l'utilisateur peut spécifier d'une part les différents fichiers nécessaires à l'exécution, et d'autre part les fichiers à conserver.

La quatrième fonctionnalité est la reprise des exécutions en cas d'échec. Pour cela, chaque exécution réussie est marquée comme telle ; si l'outil est lancé en mode reprise, il ne relance pas les exécutions réussies.

3 Conclusion et perspectives

Nous avons identifié des besoins récurrents à toute pratique du benchmarking, en particulier en recherche opérationnelle, et avons essayé d'y répondre à travers notre outil Parrot. La première version de notre logiciel n'apporte pour l'instant que quelques fonctionnalités innovantes. Elle permet principalement de lancer de manière automatique et séquentielle un grand nombre d'exécutions. Mais elle simplifie déjà la pratique d'expérimentation intensive, par un gain de temps, et en s'adaptant, par sa généricité, à un large spectre de problèmes. Son utilisation a déjà été éprouvée avec succès dans de précédents travaux [3].

De plus, cette approche offre une description de l'expérimentation qui assure sa reproductibilité, ce qui est un critère important pour valoriser les résultats empiriques.

Cependant ceci n'est qu'une première étape dans notre projet et nous souhaitons maintenant l'enrichir par diverses fonctionnalités avancées qui simplifieraient encore d'avantage la pratique du benchmarking. Dans les grandes lignes, nos principales idées d'améliorations sont la parallélisation des exécutions, l'exploitation des résultats, une interface graphique pour faciliter l'utilisation.

Nous en profitons donc pour appeler la communauté à nous faire part de nouveaux besoins et cas d'utilisation afin d'améliorer les capacités de Parrot pour le benchmarking et de garantir sa généricité, tout en conservant sa simplicité.

Références

[1] J.L. Bentley. Tools for Experiments on Algorithms. *CMU Computer Science : A 25th Anniversary Commemorative*, 99–123, 1991.

[2] D.S. Johnson. A Theoreticians Guide for Experimental Analysis of Algorithms. *Proceedings of the 5th and 6th DIMACS Implementation Challenges*, 215–250, 2002.

[3] V. Weber et Y. Kieffer. Étude d'Instances pour les Problèmes NP-Difficiles. *JGA 2011*, Lyon, 2011.

Annexe C

Challenge ROADEF/EURO 2010

J'ai eu l'occasion de participer au *Challenge ROADEF/EURO 2010* avec J. Darlay, L. Esperet, Y. Kieffer et G. Naves.

Le sujet, proposé par EDF, portait sur la gestion de la production nationale de l'énergie électrique avec prise en compte de la planification des arrêts pour maintenance des centrales nucléaires, sur un horizon de plusieurs années.

En particulier, j'ai pu mettre à profit ma réflexion sur les instances pour étoffer le jeu d'instances proposé en générant de nouvelles instances. Ce procédé nous a permis d'améliorer la robustesse de l'algorithme, ce qui nous a certainement aidé à nous classer troisième de la compétition.

Nous joignons un article qui décrit plus en détail notre approche.

A direct approach for an energy planning problem

Julien Darlay[*], Louis Esperet[*], Yann Kieffer[*],
Guyslain Naves[†], Valentin Weber[*]

March 24, 2011

Abstract

1 Introduction and main motivation

In July 2009, the French electricity provider EDF offered a formalized optimization problem of energy management for the first EURO/ROADEF challenge, a competition featuring optimization in limited computation time [5]. The problem to be tackled includes the planning of production and stops of nuclear reactors, and production levels of thermal reactors, on a 5-year horizon, divided into short-term time steps (less than a day).

One particularly challenging aspect of this problem was the size of the input data (up to 2.5 GB of RAM).

Our approach to the problem is based on two premises. First, no high-performance multi-purpose black-box was to be used, like Integer Linear Programming, or Constraint Programming solvers. This means that ad-hoc algorithms would be devised for identified subproblems, yielding a completely hand-written program for the global optimization. Second, in support of this first guideline, a high-level programming language should be used. We chose Ocaml for this task.

This paper is both a report on the exact procedure we ended up using to algorithmically solve the problem; and a report on the experience of solving real-world, computationally-demanding optimization problems using higher-level languages.

After reviewing the state of the art, we will argue about the choice of programming languages for computationally intensive tasks, and that functionaly programming languages could be a valuable alternative to imperative style programming languages.

We then give a description of the problem that we solve, and then describe our algorithmic method to solve this problem. We then give numerical results to illustrate the roles of the different steps of our approach, and end up with a conclusion and perspectives of our work.

[*]Laboratoire G-SCOP, Grenoble INP, Université Joseph Fourier, CNRS, Grenoble, France.
[†]McGill University, Montréal, Québec, Canada.

2 State of the art

To review the litterature for our energy-planning problem, we refer to the closest theorical problem, which is called the Unit Commitment Problem (UCP). Considering a set of production units, the goal is to find their optimal turn-on and turn-off schedules and output levels over a given time horizon to satisfy a certain power production requirement. The objective is to minimise the power generation costs [1].

The problems has many extensions. For a globall overview, we suggest those surveys [4] [6]. In particular, we can emphasize some attributes. First the UCP generally refers to short time horizon. It covers some weeks or few months of production to schedule. Long term UCP implicates additionnal constraintes such as maintenance time or price increases. Secondly to take the forecasting uncertainty into account, models can include stochastic datas or multiple scenarios of prevision [3] [7]. Lastly many approachs to solve the UCP are presented in the literature, lilke heuristics, mixed integer programming, Benders decomposition, dynamic programming or Lagrangian relaxation [8].

The most approaching work is [2]. It seems to describe the current model solved by EDF tools, which is an version of long-term UCP. The main approach of resolution is based on constraint programming, to find a feaisble solution, and then local search, for the improvement. Moreover, the handling of the problem is also split between two subproblems : the outage scheduling and the production planning.

Our model is more complex. In particular, it includes a part of uncertainty, through multiple demand scenarios. Also some refueling management is added to the model. Our proposition can be seen as following the same general lines as that of [2]. We decompose the problem in its natural stages: stops scheduling, refueling, production planning. And we also first aim at quickly finding a feasible solution with a constraint-programming-like approach; then we improve this solution with two further improvement steps.

3 Functional programming languages

When discussing real-world problem solving, with real computers, and real-time computing limits, it is worthwhile to take into account two aspects that are seldom analyzed: actual data-structures used, and the choice of the programming language.

These two design choices are related, since some programming styles allow for some kinds of data structures that some others forbid. The choice of the programming language has an impact both in the actual execution time of the computing program, but also on the time and effort devoted to its development.

The usual trade-off is between computing efficiency and program development efficiency. Since computing efficiency is regarded as foremost, imperative languages like C or C++ are usually used. This represents a serious unbalance towards program efficiency that is seldom questioned.

In the case of a time-limited competition where software development occurs outside contractual working time, the need for program development efficiency is great. Actually,

it might also be the best testing ground for testing new development methods, since bad results have no other bad consequences than spent time and learned lessons.

Therefore, we tried to disturb the usual unbalance from computation efficiency to programming efficiency, ease of programming and program flexibility.

We now review the characteristics of modern (meaning statically typed) functional programming languages - such as Ocaml and Haskell, as compared to more common imperative programming languages. This review is synthetic, providing ground only for the discussion of the benefits and losses associated to the choice of the programming language paradigm. We will discuss data structures further in the next section.

A dramatic difference between functional programming languages and imperative languages lies in what basic objects are manipulated. Functional programming languages talk mostly about values, while imperative languages handle both values and states. The very word "Imperative" denotes state handling, and operation sequencing. In a functional programming language, expressions are written by the programmer, and the order of operations for their evaluation is inferred by the compiler, alleviating the need to express precise sequencing for operations.

Modern functional programming languages like Ocaml and Haskell provide static typing analysis, unlike their predecessors Lisp and Scheme. In fact, each subexpression of any expression has a type, and the system can infer types automatically; this means that type checking is a lot stronger than for imperative languages, where commands are sequenced without any type dependence between them. Function definitions themselves are automatically inferred by the compiler from their codes; their types are the equivalent of function prototypes in imperative languages

Typing differences extend further to type definitions and values handling. Type constructors include function definition (lambda abstraction), giving the very name to the paradigm, "functional programming"; and also product types (records) and sum types (union), providing enumerations as a special case. All these type constructors allow type creation *on the fly*, for example in a function definition, freeing the user in his use of more elaborated types. As a comparison, some uses of complex types in an imperative languages might be inhibited by the very cost of a type definition, especially in the case of one-time use. Hence, functional languages support abstract programming, and programming experimentation.

Union types are discriminated through the use of pattern matching, meaning that the compiler lends support and type checking for union values usage. This is in stark contrast with C and its awkward, unchecked union types, and alleviates the need for many uses of object-oriented inheritance.

A foremost parameter is efficiency, meaning how quickly a task can be performed in each language. Efficiency is first linked with the fact that the language is compiled or interpreted. Like C and C++, functional languages are compiled, and state-of-the-art compilers are available for them, with powerful optimizing code generators. It is commonly accepted that the slowdown when going from imperative to functional languages is a small factor - depending on both the application, and the programmer proficiency with the language.

4 How functional programming might help for optimization computations

We now present the expected benefits to be had from using the Ocaml language for solving a real-world optimization problem. Since to the best of our knowledge, functional programming languages are seldom used for heavy-duty computations, this article will serve as a preliminary report on the suitability of functional programming languages for serious computations.

In the previous section, a quick review was given of the characteristics of functional programming languages. We rely on specific elements from this review to explain in which ways functional programming can support the programming activity. Benefits can be categorized into two classes: helping the programmer in his day-to-day task of writing programs; and enlarging his possibilities in program-writing.

With richer and more flexible types, code length reduces when going from the imperative to the functional style. Hence typing time reduces, but further than that, what the programmer writes now looks more like what he is thinking: the program looks closer to the mathematical models used to write it.

This makes programming faster and easier. It also makes it more enjoyable: programming is less of a hurdle in the task of implementing abstract algorithms.

The new possibilities offered by functional programming are themselves of two kinds: those that were available in imperative programming, but rather costly; and those that were were very difficult or impossible to implement in imperative programming languages.

In the first class falls all uses of involved constructed types, as explained in the previous section. Code experimentation in general is easier when program expression is more straightforward.

In the second class, we find all kinds of constructs and use cases that are so badly supported in imperative languages that programming styles based on them developed only on the functional side. Some examples are lambda abstraction and garbage collecting. Lambda abstraction means that function are values like others, that can be created, passed to and returned from functions, and composed in powerful ways.

The availability of higher-order function types allow for synthetic interfaces for modules. These interfaces can be made to be exact equivalent of their abstract, mathematical counterpart. This means that design choices do not have to be made when defining modules and their interfaces; only when implementation starts does one commit to particular choices. If they are found unsuitable, they can be changed easily without compromising the whole design of the main program.

As a summary, functional programming languages offer a different experience of programming, which is more conceptual, and that may be better suited for implementing programs of a mathematical nature. Our bet was that the ease of programming would compensate the loss in computation efficiency. We leave it to the reader to appreciate whether this was a reasonable bet.

5 The problem

The problem can be summed up as the need to define the production planning for two kinds of power plants: non-nuclear thermal (called *type I* in the following), and nuclear (*type II*). The type II plants have to be stopped on a regular basis both to replenish nuclear fuel, and also for maintenance operations. These stops will be referred to as *outages*. The time of producion between two outages is called a *production compaign*. Campaigns are numbered for each type II plant, so that constraints for different outages and for different compaigns can be conveniently referred to.

The precise amount of electricity produced by each plant is to be computed for a large number of scenarios, representing different profiles of electricity demand, and the final objective depends equally on the results of all of them.

A solution To get a finer look, we will go over the exact quantities that have to be determined. First, outage dates have to be chosen for each type II plant, with a rough estimate of one outage per year. All instances offered featured a 5-year horizon. Each time a type II plant is stopped, it is to be refueled. Refuel quantities is the second kind of datas to be determined. An important aspect of the problem is that outage dates and refuel amounts have to be the same in each scenario. Finally, one has to give for each scenario, and for each timestep, how much electricity each plant should produce (and consequently, how much fuel is left in each nuclear plant). Notice that the amount of fuel in a given type II plant at a given timestep is not required to be the same in all scenario. In practice, it highly depends on the production strategy that is used.

The amount of electricity produced by a type II power plant at some instant is seen as a portion of the maximum amount of electricity that can be produced by the plant. The discrepancy between the amount of energy produced and the maximum that can be produced is call the *modulation*. One particular constraint of our problem is that this modulation cannot exceed a limit set for each production campaign. Therefore, having too many plants on-line during a low demand period yields a risk of overproduction.

Another important aspect of the problem is that the time scales are different for the outage scheduling and for the production planning: each timestep represents a week in the first case, while it represents 8 hours in the second case. This already means that these two subproblems will need to be considered with different approaches, although they are strongly related.

The constraints The constraints that define a feasible solution come in different kinds. First, outages are limited by some purely combinatorial constraints. For an outage a (identified by a pair (i, k) where i is a power plant of type II and k is a campaign), the interval of time where a is allowed to start is denoted by $I(a)$. Assume we set the first week of the outage a to $ha(a)$. It is naturally required that $\forall a \in A$, $ha(a) \in I(a)$, where A is the set of all outages. Another set of constraints can be expressed in the following way:

$$\forall a, a' \in A,\ (ha(a) - ha(a') \geq C(a, a')) \text{ or } (ha(a') - ha(a) \geq C(a', a)),$$

where $C(a, a')$ is the minimum spacing (or maximum overlapping) between a and a'. Note that $C(a, a')$ need not be the same as $C(a', a)$. This corresponds to the fact that some equipment is necessary for the maintenance of a power plant, and that it is not available in every plant (and rather has to be shared between different plants), so they cannot be in outage at the same time. Therefore, $C(a, a')$ is for instance the length of a plus the time it takes to transport the resource from the power plant associated with a to the power plant associated with a'.

Besides these binary constraints, the fact that it is preferable to avoid having to many plants offline during the same week w can be roughly expressed with the following constraints:

$$\forall w, \sum_{a \text{ with } 0 \leq w - ha(a) \leq DA(a)} P(a, w) \leq Q(w)$$

where $DA(a)$ is the length of the outage a, $P(a, w)$ is the capacity during the week w of the power plant corresponding to a, and $Q(w)$ is the upper bound on maximum off-line capacity during w.

These two types of constraints are the most important ones for the outage scheduling. A remarkable feature of the binary constraints is that the connected components of the graph of constraints are small. As explained above, these components can correspond to (geographically) close plants, or even different reactors of the same plant. It means that placing the outage of a plant only influences a limited number of other plants (with these specific constraints). This will be used in the scheduling algorithm described in Section 6.

The constraints on fuel level in type II plants during production reflect the conservative transformation of fuel to electricity, and limit the production levels allowed when fuel gets low: in such a case, the production has to follow a fixed pattern. This set of constraints links type II fuel levels and production levels.

Another set of constraints expresses the rules surrounding refueling. There are lower and upper bounds on levels of fuel in the plant before and after refueling; and there are bounds on the very amount of fuel that is added. These link production-time type II fuel levels with refuel amounts.

Constraints are given respecting the possibility of not using type II plants at their full power – expressed in terms of total slack in a production period. These, together with capacity bounds on the production of type II plants, link outage dates with type II production levels.

The cost of a solution Finally, to meet the demand exactly, one has to use a combination of type I and type II plants for production; this makes the link with type I production levels.

The cost function to be optimized is the monetary cost of the production suggested to meet the demand. It has to be averaged over scenarios, which amounts to minimizing the

sum of the cost over all scenarios, since the number of scenarios is fixed for each instance of the problem. The cost has two components: cost of production by type I plants, which varies with each plant, each timestep and each scenario, and is assumed proportional to the power produced during that time step; and the cost of type II fuel. There is no cost associated (in this model) with running a type II plant, besides the fuel cost.

6 Algorithms

6.1 General structure

The algorithm can be separated into two main steps. The first step finds a feasible solution under basic assumptions while the second step improves this basic solution. Finding a first, realizable solution is decomposed in several subproblems, each one being tackled by a specific procedure. The decomposition is depicted in Figure 1.

The *scheduler* procedure finds a realizable schedule of power plant outages according to some global parameter $m \in [0, 1]$. It basically corresponds to the duration of the production period, the time between two outages must be large enough to consume the minimum amount of fuel added during the outage plus m times the maximum amount of modulation during the campaign. From this schedule, the first refueling procedure called *RefuelerMin* adds the minimum amount of fuel to each power plant during its outage, then the procedure *Production* is called for the first time. It returns a realizable production plan on each scenario according to the schedule and the refueled quantities. From this realizable solution, the second refueling procedure (*Refueler*) computes a new amount of fuel to be added during the outages. Finally the procedure *Production* is called a second time with the new refueled quantities.

The second step is essentially a local improvement procedure called *Improver*. Its objective is to remove inconsistencies due to the basic assumptions made during the first step.

Finally, this general procedure is embedded in a robustness framework. The procedures *Scheduler* and *Production* may not find a solution because of the value of m. More precisely, *Scheduler* may fail to find a realizable schedule if m is too large (meaning a long production) whereas *Production* may be unable to return a realizable production if m is too small. In the latter case the time between two outages is not long enough to consume all the fuel and to allow some modulation. If an error is raised by *Scheduler* then the entire process is restarted with a smaller value of m. On the other hand, if *Production* fails then the value of m is increased and the process is restarted.

6.2 Scheduler

The objective of the scheduling procedure is to find a realizable schedule for a list of outages. It takes as parameters a list of outages to schedule, and a real number m in $[0, 1]$ indicating how much spare time should be left between outages to allow for modulation.

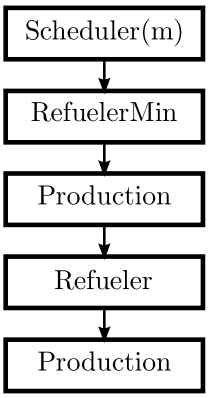

Figure 1: General structure of the first step

This parameter is common to all power plants and campaigns. The scheduling algorithm is based on a simple backtracking procedure, it has to assign a starting week to each outage according to a list of possible values. When an outage is fixed, the possible values for the remaining outages are reduced according to various constraints, as described below. If an outage cannot be scheduled, the algorithm removes the last assignment and tries its next possible value.

This procedure may take a very long time to find a realizable solution. Thus it is important to schedule the outages in a clever order to be tractable in practice. We first divide the horizon in periods of four weeks and try to find a realizable schedule in each period starting from the beginning of the horizon to the end. This relies on the reasonable assumption that two outages separated by several weeks cannot be directly in conflict. Then inside a period we first consider the outages that belong to the largest number of constraints in order to solve local conflicts with the minimum number of backtracks. For this sake we use the topology of the graph of constraints mentioned in Section 5.

The second important point in our procedure is to assign "good" values to each outage in order to avoid conflicts. Since our procedure scans outages in chronological order, we chose to examine possible outage start dates from earliest to latest. We have tested a number of orderings for these affectations, and this particular one was the best overall; others would take huge computing times on at least some of the instances.

Finally, when the procedure assigns a week to the beginning of an outage, we reduce the list of possible values for the remaining outages. This reduction is based on scheduling constraints and on production considerations. The procedure gives enough time to a power plant to consume the minimum amount of fuel added during the outage plus a given percentage of total modulation (parameter m).

This procedure finds a realizable schedule within a reasonable amount of time but it does not take into account the peaks of high demand. We address this issue by considering a black list of weeks to start an outage. This list is based on the K weeks of maximum demand among all scenarios (where K in the number of campaigns). Theses weeks are initially removed from the considered values, and added back one by one if the scheduling

procedure fails to find a solution.

The scheduling procedure is called to plan both compulsory outages and optional outages. In the latter case, the procedure is exactly the same except for the backtracking. If an outage cannot be scheduled it is simply ignored and the power plant continues to produce until it runs out of fuel. Later production campaigns for that plant are cancelled.

6.3 Refueler

The refueling procedure considers a first valid production with the minimum amount of fuel added during the outages. It computes for each outage and each scenario the amount of fuel required to produce at its maximum power (except for some imposed decreasing profile) before the beginning of the next outage. It returns for each outage the minimum amount computed over all the scenarios.

This procedure may lead to infeasible production because of some periods of low demand when modulation is imposed. In this case, we slightly increase the global parameter m and start again the whole procedure with this new value.

6.4 Production

For a given outages scheduling and refueling plan, we can compute the production plan. Production is computed independently for each scenario. The applied algorithm is a greedy list method. Nevertheless we focus on its robustness to return a feasible solution.

As a first step, all type II plants are set to produce at their maximum power level. Then the remaining demand is completed by type I production, in order of increasing price. Sometimes type II production can however be higher than demand. This situation is called overproduction.

Overproduction can be avoided by reducing the production level of type II plants. This loss of power is restricted by two conditions. First, the modulation, presented in Section 5, which represents the discrepancy between the effective and the maximum production cannot be too high. Second, a loss of power would reduce the fuel consumption. But fuel levels are restricted before and after outages. Stocks of fuel can also become too high and fuel solution infeasible. We compute the maximum amount of power that can be lost during each campaign for each type II power plant. This measure is called regulation quantity.

The algorithm of production is based on this measure. For each timestep of overproduction, we order the type II power plants according to the due date of their current campaign. Following this order, we reduce greedily the production of the type II power plants, until we reach feasibility. This method tries to preserve a maximum of flexibility in the production of type II power plants.

However it can happen that the regulation quantities are not enough to avoid the overproduction. In this case, we consider that we cannot find a feasible production solution for the current situation, and then we reconsider the scheduling plan.

6.5 Improver

As mentioned before, the first step of our algorithm may find solutions with some inconsistencies. They can be easily detected with a graphic analysis of the solutions. We developed a small solution viewer and found two types of local improvement that can be done by a greedy procedure.

The first type of inconsistency occurs at the end of the horizon. The last outage is generally optional and because of the assumption of minimal refuel during the scheduling part, it is scheduled a few weeks before the end of the horizon. It can be removed from the solution by adding more fuel during the previous outage as shown in Figure 2.

Figure 2: First type of inconsistency. The horizontal axis is the time (in weeks) and each line on the vertical axis corresponds to a power plant. Outages are represented by black rectangles and production level are in gray. Inside an outage the first bar represents the level of fuel added during the outage, the second and third ones correspond to the bound on the fuel level. The rightmost outage may be removed from the solution by adding more fuel in the previous outage.

The second inconsistency occurs when the time between two outages is too large in the schedule. This leads to several weeks without production. Since the outages are scheduled using an earliest week first policy, we may be able to delay the first outage. This situation is illustrated in Figure 3 where the first outage of the first power plant may be delayed to avoid weeks without production.

The *improver* procedure starts from the end of the timeline and addresses inconsistencies in a greedy way. From this new schedule, the functions *refueler* and *production* are called to produce a new solution.

Figure 3: Second type of inconsistency in the first solution. The leftmost outage may be delayed to avoid weeks without production.

7 Experimental results

We now give empirical results for key steps of our algorithms when run on the various instances proposed by EDF*. A brief description of the instances is given Table 1. The various solutions found at each step of our algorithm are presented in Table 2. As expected the solution after the procedure *Refueler* is better expect for some instances where the refueling values are imposed (*e.g.* instances 8 and 13). The average improvement after the *Improver* procedure is about $2.4 * 10^4$.

Instance	Timesteps	Weeks	Scenarios	PP1	PP2	C13
1	1750	250	10	11	10	46
2	1750	250	20	21	18	84
3	1750	250	20	21	18	80
4	1750	250	30	31	30	122
5	1750	250	30	31	28	120
6	5817	277	50	25	50	222
7	5565	265	50	27	48	192
8	5817	277	121	19	56	114
9	5817	277	121	19	56	114
10	5565	265	121	19	56	235
11	5817	277	50	25	50	239
12	5523	263	50	27	48	207
13	5817	277	121	19	56	260
14	5817	277	121	19	56	256
15	5523	263	121	19	56	245

Table 1: Parameters of the 15 instances proposed by EDF

References

[1] R.H. Kerr, J.L. Scheidt, A.J. Fontanna, and J.K. Wiley. Unit commitment. *Power Apparatus and Systems, IEEE Transactions on*, PAS-85(5):417 –421, May 1966.

[2] M.O.I. Khemmoudj, M. Porcheron, and H. Bennaceur. When constraint programming and local search solve the scheduling problem of electricité de france nuclear power plant outages. In F. Benhamou, editor, *CP*, volume 4204 of *Lecture Notes in Computer Science*, pages 271–283. Springer, 2006.

[3] U.A. Ozturk. *The Stochastic Unit Commitment Problem : A Chance Constrained Programming Approach Considering Extreme Multivariate Tail Probabilities*. PhD thesis, University of Pittsburgh, 2003.

*http://challenge.roadef.org/2010/en/instances.html

Instance	First solution (rmin)	Solution (refueler)	Final solution
1	1.7168	1.7010	1.7006
2	1.4938	1.4708	1.4692
3	1.6100	1.5623	1.5603
4	1.2223	1.1549	1.1505
5	1.3466	1.3089	1.3036
6	1.3741	0.9675	0.9306
7	2.2014	0.9213	0.8869
8	0.9435	0.9435	0.9268
9	1.1749	1.0386	1.0285
10	2.7185	0.9163	0.8761
11	1.1901	0.8999	0.8732
12	1.6840	0.8954	0.8622
13	0.8840	0.8840	0.8627
14	1.3639	0.9692	0.8991
15	2.6995	0.9200	0.8628

Table 2: Cost of solutions found at each step of our algorithm ($*10^6$)

[4] N.P. Padhy. Unit commitment-a bibliographical survey. *Power Systems, IEEE Transactions on*, 19(2):1196 – 1205, May 2004.

[5] M. Porcheron, A. Gorge, O. Juan, T. Simovic, and G. Dereu. Challenge roadef/euro 2010 : A large-scale energy management problem with varied constraints.

[6] I.J. Raglend and N.P. Padhy. Comparison of practical unit commitment problem solutions. *Electric Power Components and Systems*, 36(8):844 – 863, 2008.

[7] P.A. Ruiz, E. Zak, C.R. Philbrick, and K.W. Cheung. Modeling the unit commitment problem with uncertainty. In *INFORMS Annual Meeting, Seattle, WA*, 2007.

[8] S. Salam. Unit commitment solution methods. In *Proceedings of World Academy of Science, Engineering and Technology*, volume 26, pages 600–605, 2007.

Annexe D

Problème du nombre enveloppe

C'est une étude menée au sein du projet MASCOTTE d'INRIA-Sophia Antipolis, où j'ai été accueilli régulièrement pendant ma dernière année de thèse. Elle est le fruit d'une collaboration avec J. Araujo, G. Morel, L. Sampaio et R. Soares.

L'article qui suit a été accepté à la conférence *VII Latin-American Algorithms, Graphs and Optimization Symposium (LAGOS '13)*. Pour une version plus détaillée, contenant les démonstrations des résultats, nous avons mis en ligne un rapport de recherche (Araujo et al. 2012).

En plus des résultats présentés dans le chapitre 5 sur les réductions d'instances et l'algorithme FPT selon le paramètre de diversité de voisinage, nous décrivons un algorithme polynomial pour résoudre le problème du nombre enveloppe dans les graphes sans P_5 induit et sans triangle.

Hull number: P_5-free graphs and reduction rules [1]

J. Araujo [a,b,2,3], G. Morel [a,2,3], L. Sampaio [a,2,3], R. Soares [a,b,2,3] and V. Weber [c,4]

[a] *COATI Project, I3S (CNRS & UNS) and INRIA, INRIA Sophia Antipolis, 2004 route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex France.*

[b] *ParGO Research Group, Universidade Federal do Ceará, Campus do Pici, Bloco 910. 60455-760, Fortaleza, Ceará, Brazil.*

[c] *Grenoble-INP / UJF-Grenoble 1 / CNRS, G-SCOP UMR5272 Grenoble, F-38031, France.*

Abstract

In this paper, we study the (geodesic) hull number of graphs. For any two vertices $u, v \in V$ of a connected undirected graph $G = (V, E)$, the closed interval $I[u, v]$ of u and v is the set of vertices that belong to some shortest (u, v)-path. For any $S \subseteq V$, let $I[S] = \bigcup_{u,v \in S} I[u, v]$. A subset $S \subseteq V$ is (geodesically) convex if $I[S] = S$. Given a subset $S \subseteq V$, the convex hull $I_h[S]$ of S is the smallest convex set that contains S. We say that S is a hull set of G if $I_h[S] = V$. The size of a minimum hull set of G is the hull number of G, denoted by $hn(G)$.

First, we show a polynomial-time algorithm to compute the hull number of any P_5-free triangle-free graph. Then, we present four reduction rules based on vertices with the same neighborhood. We use these reduction rules to propose a fixed parameter tractable algorithm to compute the hull number of any graph G, where the parameter can be the size of a vertex cover of G or, more generally, its neighborhood diversity, and we also use these reductions to characterize the hull number of the lexicographic product of any two graphs.

Keywords: Graph Convexity, Hull Number, Geodesic Convexity, P_5-free Graphs, Lexicographic Product, Parameterized Complexity, Neighborhood Diversity.

[1] Partly supported by ANR Blanc AGAPE ANR-09-BLAN-0159, PRONEM/FUNCAP, CNPq Universal 483387/2011-8 and the INRIA/FUNCAP exchange program.

[2] J. Araujo is supported by CNPq/Brazil PDE 202049/2012-4, G. Morel, L. Sampaio and R. Soares by ANR Blanc AGAPE and L. Sampaio by CAPES/Brazil.

[3] {julio.araujo, gregory.morel, leonardo.sampaio_rocha, ronan.pardo_soares}@inria.fr

[4] valentin.weber@g-scop.grenoble-inp.fr

1 Introduction

All graphs in this work are undirected, simple and loop-less. Given a connected graph $G = (V, E)$, the closed interval $I[u, v]$ of any two vertices $u, v \in V$ is the set of vertices that belong to some u-v geodesic of G, i.e. some shortest (u, v)-path. For any $S \subseteq V$, let $I[S] = \bigcup_{u,v \in S} I[u, v]$. A subset $S \subseteq V$ is *(geodesically) convex* if $I[S] = S$. Given a subset $S \subseteq V$, the *convex hull* $I_h[S]$ of S is the smallest convex set that contains S. We say that a vertex v is *generated* by a set of vertices S if $v \in I_h[S]$. We say that S is a *hull set* of G if $I_h[S] = V$. The size of a minimum hull set of G is the *hull number* of G, denoted by $hn(G)$ [9].

It is known that computing $hn(G)$ is an NP-hard problem for bipartite graphs [3]. Several bounds on the hull number of triangle-free graphs are presented in [8]. In [7], the authors show, among other results, that the hull number of any P_4-free graph, i.e. any graph without induced path with four vertices, can be computed in polynomial time. In Section 3, we show a linear-time algorithm to compute the hull number of any P_5-free triangle-free graph.

In Section 4, we show four reduction rules to obtain, from a graph G, another graph G^* that has one vertex less than G and which satisfies either $hn(G) = hn(G^*)$ or $hn(G) = hn(G^*) + 1$, according to the used rule. We then first use these rules to obtain a fixed parameter tractable (FPT) algorithm, where the parameter is the neighborhood diversity of the input graph. For definitions on Parameterized Complexity we refer to [10]. Given a graph G and vertices $u, v \in V(G)$, we say that u and v are *twins* (a.k.a. of the *same type*) if $N(v) \setminus \{u\} = N(u) \setminus \{v\}$. The *neighborhood diversity* of a graph is k, if its vertex set can be partitioned into k sets $S_1, \ldots, S_k$, such that any pair of vertices $u, v \in S_i$ are twins. This parameter was proposed by Lampis [12], motivated by the fact that a graph of bounded vertex cover also has bounded neighborhood diversity, and therefore the later parameter can be used to obtain more general results. Many problems have been show to be FPT when the parameter is the neighborhood diversity [11].

Finally, we use these rules to characterize the hull number of the lexicographic product of any two graphs. Given two graphs G and H, the *lexicographic product* $G \circ H$ is the graph whose vertex set is $V(G \circ H) = V(G) \times V(H)$ and such that two vertices (g_1, h_1) and (g_2, h_2) are adjacent if, and only if, either $g_1 g_2 \in E(G)$ or we have that both $g_1 = g_2$ and $h_1 h_2 \in E(G)$.

It is known in the literature a characterization of the (geodesic) convex sets in the lexicographic product of two graphs [1] and a study of the pre-hull number for this product [13]. There are also some results concerning the hull number of the Cartesian and strong products of graphs [5, 6].

2 Preliminaries

Let us recall some definitions and lemmas that we use in the sequel.

We denote by $N_G(v)$ (or simply $N(v)$) the neighborhood of a vertex. A vertex v is *simplicial* (resp. *universal*) if $N(v)$ is a clique (resp. is equal to $V(G) \setminus \{v\}$). Let $d_G(u, v)$ denote the *distance* between u and v, i.e. the length of a shortest (u, v)-path. A subgraph $H \subseteq G$ is *isometric* if, for each $u, v \in V(H)$, $d_H(u, v) = d_G(u, v)$. A P_k (resp. C_k) in a graph G denotes an induced path (resp. cycle) on k vertices. Given a graph H, we say that a graph G is H-free if G does not contain H as an induced subgraph. Moreover, we consider that all the graphs in this work are connected. Indeed, if a graph G is not connected, its hull number can be computed by the sum of the hull numbers of its connected components, as observed by Dourado et al. [7].

Lemma 1. *[9] For any hull set S of a graph G, S contains all simplicial vertices of G.*

Lemma 2. *[7] Let G be a graph which is not complete. No hull set of G with cardinality $hn(G)$ contains a universal vertex.*

Lemma 3. *[7] Let G be a graph, H be an isometric subgraph of G and S be any hull set of H. Then, the convex hull of S in G contains $V(H)$.*

Lemma 4. *[7] Let G be a graph and S a proper and non-empty subset of $V(G)$. If $V(G) \setminus S$ is convex, then every hull set of G contains at least one vertex of S.*

3 Hull number of P_5-free triangle-free graphs

In this section, we present a linear-time algorithm to compute $hn(G)$, for any P_5-free triangle-free graph G. To prove the correctness of this algorithm, we need to recall some definitions and previous results:

Definition 1. *Given a graph $G = (V, E)$, we say that $S \subseteq V$ is a* dominating set *if every vertex $v \in V \setminus S$ has a neighbor in S.*

It is well known that:

Theorem 1. *[4] G is P_5-free if, and only if, for every induced subgraph $H \subseteq G$ either $V(H)$ contains a dominating C_5 or a dominating clique.*

As a consequence, we have that:

Corollary 1. *If G is a connected P_5-free bipartite graph, then there exists a dominating edge in G.*

Theorem 2. *The hull number of a P_5-free bipartite graph $G = (A \cup B, E)$ can be computed in linear time.*

For the next result, recall that the complexity of finding the convex hull of

a set of vertices $S \subseteq V(G)$ of a graph G is $\mathcal{O}(|S||E(G)|)$, as described in [7]. We can relax the constraint of G being bipartite to obtain the following:

Corollary 2. *If G is a P_5-free triangle-free graph, then $hn(G)$ can be computed in polynomial time.*

4 Neighborhood Diversity and Lexicographic Product

In this section, we present four reduction rules to compute the hull number of a graph. We need to introduce some definitions.

Given a set S, let $I^0[S] = S$ and $I^k[S] = I[I^{k-1}[S]]$, for $k > 0$. We say that v is generated by S *at step* $t \geq 1$, if $v \in I^t[S]$ and $v \notin I^{t-1}[S]$. Observe that the convex hull $I_h(S)$ of a given set of vertices S is equal to $I^{|V(G)|}[S]$.

Given a graph G, we say that two vertices v and v' are *twins* if $N(v)\setminus\{v'\} = N(v') \setminus \{v\}$. If v and v' are adjacent, we call them *true* twins, otherwise we say that they are *false* twins.

Let G be a graph and v and v' be two of its vertices. The *identification* of v' into v is the operation that produces a graph G' such that $V(G') = V(G)\setminus\{v'\}$ and $E(G') = (E(G) \setminus \{v'w \mid w \in N_G(v')\}) \cup \{vw \mid v'w \in E(G) \text{ and } w \neq v\}$.

Lemma 5. *Let G be a graph and v and v' be non-simplicial and twin vertices. Let G' be obtained from G by the identification of v' into v. Then, $hn(G) = hn(G')$.*

Lemma 6. *Let G be a graph and v, v', v'' be simplicial and pairwise false twin vertices. Let G' be obtained from G by the identification of v'' into v. Then, $hn(G) = hn(G') + 1$.*

Observe that we cannot simplify the statement of Lemma 6 to consider any pair of simplicial false twin vertices instead of triples. As an example, consider the graph obtained by removing an edge uv from a complete graph with more than 3 vertices.

Lemma 7. *Let G be a graph and v, v' be simplicial and true twin vertices. Let G' be obtained from G by the identification of v' into v. Then, $hn(G) = hn(G') + 1$.*

Recall that the *neighborhood diversity* of a graph is k, if its vertex set can be partitioned into k sets $S_1, \ldots, S_k$, such that any pair of vertices $u, v \in S_i$ are twins. Now, we use this partition to obtain the following result:

Theorem 3. *Let G be a graph whose neighborhood diversity is at most k. Then, there exists an FPT algorithm to compute $hn(G)$ in $\mathcal{O}(4^k poly(|V(G)|))$-time.*

As pointed before, a graph of bounded vertex cover size has also bounded neighborhood diversity, therefore the previous result also holds for this parameter.

Now, we use Lemma 5 and Lemma 7 to determine the lexicographic product of two graphs. Let $S(G)$ denote the set of simplicial vertices of G.

Observe that if G has a single vertex, then $hn(G \circ H) = hn(H)$. Else, we have that:

Theorem 4. *Let G be a connected graph, such that $|V(G)| \geq 2$, and let H be an arbitrary graph. Then,*

$$hn(G \circ H) = \begin{cases} 2, & \text{if } H \text{ is not complete;} \\ (|V(H)| - 1)|S(G)| + hn(G), & \text{otherwise.} \end{cases}$$

5 Conclusions

In this work, we first presented a linear-time algorithm to compute the hull number of any P_5-free triangle-free graph. However, the computational complexity of determining the hull number of a P_5-free graph is still unknown. More generally, we propose the following open question:

Question 1. *For a fixed k, what is the computational complexity of determining $hn(G)$, for a P_k-free graph G?*

In the second part of this paper, we introduced four reduction rules that we use to present an FPT algorithm to compute the hull number of any graph, where the parameter is its neighborhood diversity, and a characterization of the lexicographic product of any two graphs. It is already known in the literature another FPT algorithm to compute the hull number of any graph, where the parameter is the number of its induced P_4's [2]. To the best our knowledge, the following is also open:

Question 2. *Given a graph G, is there an FPT algorithm to determine whether $hn(G) \leq k$, for a fixed k?*

References

[1] Anand, B., M. Changat, S. Klavžar and I. Peterin, *Convex sets in lexicographic products of graphs*, Graphs and Combinatorics (2011), pp. 1–8.

[2] Araujo, J., V. Campos, F. Giroire, N. Nisse, L. Sampaio and R. Soares, *On the hull number of some graph classes*, Technical Report RR-7567, INRIA (2011). URL http://hal.inria.fr/inria-00576581/en/

[3] Araujo, J., V. Campos, F. Giroire, L. Sampaio and R. Soares, *On the hull number of some graph classes*, Electronic Notes in Discrete Mathematics **38** (2011), pp. 49 – 55, the Sixth European Conference on Combinatorics, Graph Theory and Applications, EuroComb 2011.

[4] Bacsó, G. and Z. Tuza, *Dominating cliques in p_5-free graphs*, Periodica Mathematica Hungarica **21** (1990), pp. 303–308.

[5] Cáceres, J., C. Hernando, M. Mora, I. Pelayo and M. Puertas, *On the geodetic and the hull numbers in strong product graphs*, Computers and Mathematics with Applications **60** (2010), pp. 3020 – 3031.

[6] Cagaanan, G. B., S. R. Canoy and Jr., *On the hull sets and hull number of the cartesian product of graphs*, Discrete Mathematics **287** (2004), pp. 141 – 144.

[7] Dourado, M. C., J. G. Gimbel, J. Kratochvíl, F. Protti and J. L. Szwarcfiter, *On the computation of the hull number of a graph*, Discrete Mathematics **309** (2009), pp. 5668 – 5674, combinatorics 2006, A Meeting in Celebration of Pavol Hell's 60th Birthday (May 1-5, 2006).

[8] Dourado, M. C., F. Protti, D. Rautenbach and J. L. Szwarcfiter, *On the hull number of triangle-free graphs*, SIAM J. Discret. Math. **23** (2010), pp. 2163–2172.

[9] Everett, M. G. and S. B. Seidman, *The hull number of a graph*, Discrete Mathematics **57** (1985), pp. 217 – 223.

[10] Flum, J. and M. Grohe, "Parameterized Complexity Theory," Texts in Theoretical Computer Science. An EATCS Series, Springer, 2006, 1 edition.

[11] Ganian, R., *Using neighborhood diversity to solve hard problems* (2012). URL http://arxiv.org/abs/1201.3091

[12] Lampis, M., *Algorithmic meta-theorems for restrictions of treewidth*, Algorithmica **64** (2012), pp. 19–37.

[13] Peterin, I., *The pre-hull number and lexicographic product*, Discrete Mathematics **312** (2012), pp. 2153 – 2157, special Issue: The Sixth Cracow Conference on Graph Theory, Zgorzelisko 2010.

Références

Ahammed, F. and P. Moscato (2011). Evolving L-systems as an intelligent design approach to find classes of difficult-to-solve traveling salesman problem instances. In C. D. Chio, S. Cagnoni, C. Cotta, M. Ebner, A. Ekárt, A. Esparcia-Alcázar, J. J. M. Guervós, F. Neri, M. Preuss, H. Richter, J. Togelius, and G. N. Yannakakis (Eds.), *EvoApplications (1)*, Volume 6624 of *Lecture Notes in Computer Science*, Berlin, Heidelberg, pp. 1–11. Springer-Verlag.

Applegate, D. L., R. E. Bixby, V. Chvátal, and W. J. Cook (2007). *The Traveling Salesman Problem : A Computational Study*. Princeton Series in Applied Mathematics. Princeton, NJ, USA : Princeton University Press.

Araujo, J., V. Campos, F. Giroire, N. Nisse, L. Sampaio, and R. Soares (2013). On the hull number of some graph classes. *Theoretical Computer Science*.

Araujo, J., G. Morel, L. Sampaio, R. Soares, and V. Weber (2012). Hull number : P_5-free graphs and reduction rules. Technical Report RR-8045, INRIA.

Arthur, J. L. and J. O. Frendewey (1988). Generating travelling-salesman problems with known optimal tours. *Journal of the Operational Research Society 39*(2), 153–159.

Balinski, M. (1965). Integer programming : Methods, uses, computation. *Management Science Series A 12*(3), 253 – 313.

Bartz-Beielstein, T., M. Chiarandini, L. Paquete, and M. Preuss (Eds.) (2010). *Experimental Methods for the Analysis of Optimization Algorithms*. Berlin Heidelberg : Springer-Verlag.

Barvinok, A., E. Gimadi, and A. Serdyukov (2004). The maximum TSP. In G. Gutin, A. Punnen, D.-Z. Du, and P. M. Pardalos (Eds.), *The Traveling Salesman Problem and Its Variations*, Volume 12 of *Combinatorial Optimization*, pp. 585–607. Springer US.

Beardwood, J., J. H. Halton, and J. M. Hammersley (1959). The shortest path through many points. *Mathematical Proceedings of the Cambridge Philosophical Society 55*, 299–327.

Bellmore, M. and G. L. Nemhauser (1968). The traveling salesman problem : A survey. *Operations Research 16*(3), 538–558.

Berg, T. (2008). *On the Complexity of Modified Instances*. Ph. D. thesis, Friedrich Schiller University of Jena.

Birattari, M. (2004a). On the estimation of the expected performance of a metaheuristic on a class of instances. How many instances, how many runs ? Tech-

nical Report TR/IRIDIA/2004-001, IRIDIA, Université Libre de Bruxelles, Brussels, Belgium.

Birattari, M. (2004b). *The Problem of Tuning Metaheuristics as Seen from a Machine Learning Perspective*. Ph. D. thesis, Université Libre de Bruxelles.

Birattari, M., M. Zlochin, and M. Dorigo (2006). Towards a theory of practice in metaheuristics design : A machine learning perspective. *RAIRO - Theoretical Informatics and Applications 40*, 353–369.

Brady, R. M. (1985). Optimization strategies gleaned from biological evolution. *Nature 317*, 804–806.

Branke, J. and C. W. Pickardt (2011). Evolutionary search for difficult problem instances to support the design of job shop dispatching rules. *European Journal of Operational Research 212*(1), 22–32.

Brown, K. L., E. Nudelman, and Y. Shoham (2009). Empirical hardness models : Methodology and a case study on combinatorial auctions. *Journal of the Association for Computing Machinery 56*(4), 1–52.

Cheeseman, P., B. Kanefsky, and W. M. Taylor (1991). Where the really hard problems are. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'91, San Francisco, CA, USA, pp. 331–337. Morgan Kaufmann Publishers Inc.

Christofides, N. (1976). Worst-case analysis of a new heuristic for the travelling salesman problem. Technical Report 388, Graduate School of Industrial Administration, Carnegie Mellon University.

Chvátal, V. (1980). Hard knapsack problems. *Operations Research 28*(6), 1402–1411.

Clarke, G. and J. W. Wright (1964). Scheduling of vehicles from a central depot to a number of delivery points. *Operations Research 12*(4), 568–581.

Corne, D. W. and A. P. Reynolds (2010). Optimisation and generalisation : Footprints in instance space. In R. Schaefer, C. Cotta, J. Kolodziej, and G. Rudolph (Eds.), *Parallel Problem Solving from Nature, PPSN XI*, Volume 6238 of *Lecture Notes in Computer Science*, pp. 22–31. Springer Berlin Heidelberg.

Cotta, C. and P. Moscato (2003). A mixed evolutionary-statistical analysis of an algorithm's complexity. *Applied Mathematics Letters 16*(1), 41–47.

Croes, G. A. (1958). A method for solving traveling-salesman problems. *Operations Research 6*(6), 791–812.

Derigs, U. and A. Metz (1986). On the use of optimal fractional matchings for solving the (integer) matching problem. *Computing 36*, 263–270.

Dourado, M. C., J. G. Gimbel, J. Kratochvíl, F. Protti, and J. L. Szwarcfiter (2009). On the computation of the hull number of a graph. *Discrete Mathematics 309*(18), 5668–5674.

Downey, R. G. and M. R. Fellows (1999). *Parameterized Complexity*. Springer-Verlag.

Drezner, Z., P. Hahn, and E. Taillard (2005). Recent advances for the quadratic assignment problem with special emphasis on instances that are difficult for meta-heuristic methods. *Annals of Operations Research 139*(1), 65–94.

Ebenegger, C., P. L. Hammer, and D. de Werra (1984). Pseudo-boolean functions and stability of graphs. In R. C.-G. R.E. Burkard and U. Zimmermann (Eds.), *Algebraic and Combinatorial Methods in Operations Research Proceedings of the Workshop on Algebraic Structures in Operations Research*, Volume 95 of *North-Holland Mathematics Studies*, pp. 83–97. North-Holland.

Everett, M. G. and S. B. Seidman (1985). The hull number of a graph. *Discrete Mathematics 57*(3), 217–223.

Fischer, T., T. Stützle, H. Hoos, and P. Merz (2005). An analysis of the hardness of TSP instances for two high-performance algorithms. In *MIC'2005 - 6th Metaheuristics International Conference*, Vienna, Austria.

Fleischer, R., X. Wu, and L. Yuan (2009). Experimental study of FPT algorithms for the directed feedback vertex set problem. In A. Fiat and P. Sanders (Eds.), *Algorithms - ESA 2009*, Volume 5757 of *Lecture Notes in Computer Science*, pp. 611–622. Springer Berlin Heidelberg.

Flum, J. and M. Grohe (2006). *Parameterized Complexity Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer.

Fomin, F. V., F. Grandoni, and D. Kratsch (2006). Measure and conquer : A simple $\mathcal{O}(2^{0.288n})$ independent set algorithm. In *Proceedings of the 17th annual ACM-SIAM Symposium on Discrete Algorithm*, SODA '06, New York, NY, USA, pp. 18–25. ACM.

Gallai, T. (1959). Über extreme Punkt- und Kantenmengen. *Ann. Univ. Sci. Budap. Rolando Eötvös, Sect. Math. 2*, 133–138.

Garey, M. R. and D. S. Johnson (1979). *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co Ltd.

Gent, I. P. and T. Walsh (1994). Easy problems are sometimes hard. *Artificial Intelligence 70*(1-2), 335–345.

Goldberg, A. (1999). Selecting problems for algorithm evaluation. In J. Vitter and C. Zaroliagis (Eds.), *Algorithm Engineering*, Volume 1668 of *Lecture Notes in Computer Science*, pp. 1–11. Springer Berlin / Heidelberg.

Goldberg, A. T. (1979). *On the Complexity of the Satisfiability Problem*. Ph. D. thesis, New York University, New York, NY, USA.

Gomes, C. P. and B. Selman (1997). Algorithm portfolio design : Theory vs. practice. In D. Geiger and P. P. Shenoy (Eds.), *Proceedings of the 13th Conference Annual Conference on Uncertainty in Artificial Intelligence*, pp. 190–197. Morgan Kaufmann.

Gomes, C. P. and B. Selman (2001). Algorithm portfolios. *Artificial Intelligence 126*(1-2), 43–62.

Greenberg, H. J. (1990). Computational testing : Why, how and how much. *ORSA Journal on Computing 2*(1), 94–97.

Greenberg, H. J. (1991). RANDMOD : A system for randomizing modifications to an instance of a linear program. *ORSA Journal on Computing 3*(2), 173–175.

Hall, N. G. and M. E. Posner (2001). Generating experimental data for computational testing with machine scheduling applications. *Operations Research 49*(6), 854–865.

Hartmanis, J. (1983). On sparse sets in NP-P. *Information Processing Letters 16*(2), 55–60.

Hayes, B. P. (1997). Can't get no satisfaction. *American Scientist 85*(2), 108–112.

Hooker, J. N. (1994). Needed : An empirical science of algorithms. *Operations Research 42*(2), 201–212.

Hooker, J. N. (1995). Testing heuristics : We have it all wrong. *Journal of Heuristics 1*, 33–42.

Huberman, B. A., R. M. Lukose, and T. Hogg (1997). An economics approach to hard computational problems. *Science 275*, 51–54.

Johnson, D. S. (2002). A theoretician's guide to the experimental analysis of algorithms. In D. S. Johnson and C. C. McGeoch (Eds.), *Data Structures, Near Neighbor Searches, and Methodology : Proceedings of the 5th and 6th DIMACS Challenge Workshop*. American Mathematical Society, Providence, RI.

Johnson, D. S. and L. A. McGeoch (1997). *Local Search in Combinatorial Optimization*, Chapter The Traveling Salesman Problem : A Case Study in Local Optimization, pp. 215–310. John Wiley and Sons, Ltd.

Johnson, D. S. and L. A. McGeoch (2002). Experimental analysis of heuristics for the STSP. In G. Gutin, A. Punnen, D.-Z. Du, and P. M. Pardalos (Eds.), *The Traveling Salesman Problem and Its Variations*, Volume 12 of *Combinatorial Optimization*. Springer US.

Johnson, E. L. (2001). Genetic algorithms as algorithm adversaries. In L. Spector, E. D. Goodman, A. Wu, W. B. Langdon, H.-M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. H. Garzon, and E. Burke (Eds.), *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '01, San Francisco, California, USA, pp. 1314–1321. Morgan Kaufmann.

Julstrom, B. A. (2009). Evolving heuristically difficult instances of combinatorial problems. In F. Rothlauf (Ed.), *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, New York, NY, USA, pp. 279–286. ACM.

Ko, K.-I., P. Orponen, U. Schöning, and O. Watanabe (1986). What is a hard instance of a computational problem ? In *Proceedings of the Conference on Structure in Complexity Theory*, London, UK, UK, pp. 197–217. Springer-Verlag.

Kochetov, Y. and D. Ivanenko (2005). Computationally difficult instances for the uncapacitated facility location problem. In T. Ibaraki, K. Nonobe, and M. Yagiura (Eds.), *Metaheuristics : Progress as Real Problem Solvers*, Volume 32 of

Operations Research/Computer Science Interfaces Series, pp. 351–367. Springer US.

Kolmogorov, A. N. (1965). Three approaches to the quantitative definition of information. *Problems of Information Transmission 1*(1), 1–7.

Kosoresow, A. P. and M. P. Johnson (2002). Finding worst-case instances of, and lower bounds for, online algorithms using genetic algorithms. In B. McKay and J. Slaney (Eds.), *AI 2002 : Advances in Artificial Intelligence*, Volume 2557 of *Lecture Notes in Computer Science*, pp. 344–355. Springer Berlin / Heidelberg.

Kuhn, H. W. (1955). The hungarian method for the assignment problem. *Naval Research Logistic Quarterly 2*, 83–97.

Lampis, M. (2012). Algorithmic meta-theorems for restrictions of treewidth. *Algorithmica 64*(1), 19–37.

Lawler, E. L., J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys (Eds.) (1985). *The Traveling Salesman Problem*. Chichester : John Wiley & Sons Ltd.

Leyton-Brown, K., E. Nudelman, G. Andrew, J. McFadden, and Y. Shoham (2003). A portfolio approach to algorithm selection. In G. Gottlob and T. Walsh (Eds.), *Proceedings of the 8th International Joint Conference on Artificial Intelligence*, pp. 1542–1542. Morgan Kaufmann.

Li, M. and P. Vitányi (1997). *An Introduction to Kolmogorov Complexity and Its Applications*. Springer.

Liberatore, P. (2004). The complexity of modified instances. *CoRR cs.CC/0402053*.

Lynch, N. (1975). On reducibility to complex or sparse sets. *Journal of the Association for Computing Machinery 22*(3), 341–345.

Marmion, M.-E. (2011). *Recherche Locale et Optimisation Combinatoire : de l'Analyse Structurelle d'un Problème à la Conception d'Algorithmes Efficaces*. Ph. D. thesis, Université des Sciences et Technologie de Lille - Lille I.

McGeoch, C. C. (1996). Toward an experimental method for algorithm simulation. *INFORMS Journal on Computing 8*(1), 1–15.

McGeoch, C. C. (2012). *A Guide to Experimental Algorithmics*. Cambridge University Press.

Monasson, R., R. Zecchina, S. Kirkpatrick, B. Selman, and L. Troyansky (1999). Determining computational complexity from characteristic 'phase transitions'. *Nature 400*, 133–137.

Moscato, P. and M. G. Norman (1998). On the performance of heuristics on finite and infinite fractal instances of the euclidean traveling salesman problem. *INFORMS Journal on Computing 10*(2), 121–132.

Niedermeier, R. (2006). *Invitation to Fixed-Parameter Algorithms*. Oxford University Press.

Norman, M. G. and P. Moscato (1995). The euclidean traveling salesman problem and a space-filling curve. *Chaos, Solitons & Fractals 6*, 389–397.

Orponen, P. (1990). On the instance complexity of NP-hard problems. In *Structure in Complexity Theory Conference*, pp. 20–27.

Orponen, P., K.-i. Ko, U. Schöning, and O. Watanabe (1994). Instance complexity. *Journal of the Association for Computing Machinery 41*(1), 96–121.

Osorio, M. A. and G. Cuaya (2005). Hard problem generation for MKP. In *Proceedings of the 6th Mexican International Conference on Computer Science*, ENC '05, Washington, DC, USA, pp. 290–297. IEEE Computer Society.

Osorio, M. A. and D. Pinto (2003). Hard and easy to solve TSP instances. In *XXX Aniversario del Programa Educativo de Computación, BUAP*, pp. 26–30.

Papadimitriou, C. H. (1994). *Computational Complexity*. Addison-Wesley.

Papadimitriou, C. H. and K. Steiglitz (1978). Some examples of difficult traveling salesman problems. *Operations Research 26*(3), 434–443.

Pisinger, D. (2005). Where are the hard knapsack problems ? *Computers and Operations Research 32*(9), 2271–2284.

Pitzer, E. and M. Affenzeller (2012). A comprehensive survey on fitness landscape analysis. In J. Fodor, R. Klempous, and C. P. Suárez Araujo (Eds.), *Recent Advances in Intelligent Engineering Systems*, Volume 378 of *Studies in Computational Intelligence*. Springer Berlin Heidelberg.

Porta, M. and B. A. Julstrom (2012). Evolving instances of unconstrained binary quadratic programming that challenge a tabu search heuristic. In *Proceedings of the 14th International Conference on Genetic and Evolutionary Computation Companion*, GECCO Companion '12, New York, NY, USA, pp. 639–640. ACM.

Rardin, R. L. and R. Uzsoy (2001). Experimental evaluation of heuristic optimization algorithms : A tutorial. *Journal of Heuristics 7*(3), 261–304.

Reinelt, G. (1991). TSPLIB : A traveling salesman problem library. *INFORMS Journal on Computing 3*(4), 376–384.

Rice, J. R. (1976). The algorithm selection problem. *Advances in Computers 15*, 65–118.

Ridge, E. and D. Kudenko (2007). An analysis of problem difficulty for a class of optimisation heuristics. In *Proceedings of the 7th European Conference on Evolutionary Computation in Combinatorial Optimization*, EvoCOP'07, Berlin, Heidelberg, pp. 198–209. Springer-Verlag.

Ridge, E. and D. Kudenko (2008). Determining whether a problem characteristic affects heuristic performance. In C. Cotta and J. van Hemert (Eds.), *Recent Advances in Evolutionary Computation for Combinatorial Optimization*, Volume 153 of *Studies in Computational Intelligence*, pp. 21–35. Springer Berlin / Heidelberg.

Rosenkrantz, D. J., R. E. Stearns, and P. M. Lewis II (1977). An analysis of several heuristics for the traveling salesman problem. *SIAM Journal on Computing 6*(3), 563–581.

Rudnick, E. M., J. H. Patel, G. S. Greenstein, and T. M. Niermann (1994). Sequential circuit test generation in a genetic algorithm framework. In *Proceedings*

of the 31st annual Design Automation Conference, DAC '94, New York, NY, USA, pp. 698–704. ACM.

Saab, D. G., Y. G. Saab, and J. A. Abraham (1992). Cris : A test cultivation program for sequential VLSI circuits. In *Proceedings of the 1992 IEEE/ACM international conference on Computer-aided design*, ICCAD '92, Los Alamitos, CA, USA, pp. 216–219. IEEE Computer Society Press.

Santana, R., A. Mendiburu, and J. A. Lozano (2012). Using network mesures to test evolved NK-landscapes. Technical Report EHU-KZAA-TR ;2012-03, University of the Basque Country.

Scheinerman, E. R. and D. H. Ullman (1997). *Fractional Graph Theory : A Rational Approach to the Theory of Graphs.* Wiley : John Wiley & Sons.

Schiavinotto, T. and T. Stützle (2004). The linear ordering problem : Instances, search space analysis and algorithms. *Journal of Mathematical Modelling and Algorithms 3*, 367–402.

Schöning, U. (1990). Complexity cores and hard problem instances. In *Proceedings of the International Symposium on Algorithms*, SIGAL '90, London, UK, UK, pp. 232–240. Springer-Verlag.

Schrijver, A. (2003). *Combinatorial Optimization : Polyhedra and Efficiency.* Springer.

Selman, B., D. G. Mitchell, and H. J. Levesque (1996). Generating hard satisfiability problems. *Artificial Intelligence 81*(1-2), 17–29.

Smith-Miles, K. and L. Lopes (2012). Measuring instance difficulty for combinatorial optimization problems. *Computers and Operations Research 39*(5), 875–889.

Smith-Miles, K. and J. I. van Hemert (2011). Discovering the suitability of optimisation algorithms by learning from evolved instances. *Annals of Mathematics and Artificial Intelligence 61*, 87–104.

Smith-Miles, K., J. I. van Hemert, and X. Y. Lim (2010). Understanding TSP difficulty by learning from evolved instances. In C. Blum and R. Battiti (Eds.), *Learning and Intelligent Optimization*, Volume 6073 of *Lecture Notes in Computer Science*, pp. 266–280. Springer Berlin / Heidelberg.

Srinivas, M. and L. M. Patnaik (1993). A simulation-based test generation scheme using genetic algorithms. In *VLSI Design'93*, pp. 132–135.

Stützle, T. and S. Fernandes (2004). New benchmark instances for the QAP and the experimental analysis of algorithms. In J. Gottlieb and G. Raidl (Eds.), *Evolutionary Computation in Combinatorial Optimization*, Volume 3004 of *Lecture Notes in Computer Science*, pp. 199–209. Springer Berlin Heidelberg.

Tukey, J. W. (1977). *Exploratory Data Analysis.* Addison-Wesley.

van Hemert, J. I. (2003). Evolving binary constraint satisfaction problem instances that are difficult to solve. In *CEC '03*, Volume 2, pp. 1267–1273.

van Hemert, J. I. (2005). Property analysis of symmetric travelling salesman problem instances acquired through evolution. In *Proceedings of the 5th European*

conference on Evolutionary Computation in Combinatorial Optimization, EvoCOP'05, Berlin, Heidelberg, pp. 122–131. Springer-Verlag.

van Hemert, J. I. (2006). Evolving combinatorial problem instances that are difficult to solve. *Evolutionary Computation 14*(4), 433–462.

van Hemert, J. I. and N. B. Urquhart (2004). Phase transition properties of clustered travelling salesman problem instances generated with evolutionary computation. In X. Yao, E. K. Burke, J. A. Lozano, J. Smith, J. J. Merelo-Guervós, J. A. Bullinaria, J. E. Rowe, P. Tino, A. Kabán, and H.-P. Schwefel (Eds.), *Parallel Problem Solving from Nature - PPSN VIII*, Volume 3242 of *Lecture Notes in Computer Science*, pp. 151–160. Springer Berlin Heidelberg.

Vandegriend, B. and J. C. Culberson (1998). The $G_{n,m}$ phase transition is not hard for the hamiltonian cycle problem. *Journal of Artificial Intelligence Research 9*, 219–245.

Wegener, J., H. Sthamer, B. F. Jones, and D. E. Eyres (1997). Testing real-time systems using genetic algorithms. *Software Quality Journal 6*, 127–135.

Weinberger, E. (1990). Correlated and uncorrelated fitness landscapes and how to tell the difference. *Biological Cybernetics 63*(5), 325–336.

Weise, T., M. Zapf, R. Chiong, and A. J. Nebro (2009). Why is optimization difficult ? In R. Chiong (Ed.), *Nature-Inspired Algorithms for Optimisation*, Volume 193/2009 of *Studies in Computational Intelligence*, Chapter 1, pp. 1–50. Springer-Verlag : Berlin/Heidelberg.

Williams, C. P. and T. Hogg (1992). Using deep structure to locate hard problems. In *Proceedings of the 10th National Conference on Artificial intelligence*, AAAI'92, pp. 472–477. AAAI Press.

Wright, S. (1932). The roles of mutation, inbreeding, crossbreeding, and selection in evolution. In *Proceedings of the 6th International Congress of Genetics*, Volume 1, pp. 356–366.

Xu, L., F. Hutter, H. H. Hoos, and K. Leyton-Brown (2008). SATzilla : Portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research 32*(1), 565–606.

Zemel, E. (1981). Functions for measuring the quality of approximate solutions to zero-one programming problems. *Mathematics of Operations Research 6*(3), 319–332.

Caractérisation des instances difficiles de problèmes d'optimisation *NP*-difficiles

Résumé L'étude expérimentale d'algorithmes est un sujet crucial dans la conception de nouveaux algorithmes, puisque le contexte d'évaluation influence inévitablement la mesure de la qualité des algorithmes. Le sujet particulier qui nous intéresse dans l'étude expérimentale est la pertinence des instances choisies pour servir de base de test à l'expérimentation. Nous formalisons ce critère par la notion de *difficulté d'instance* qui dépend des performances pratiques de méthodes de résolution. Le cœur de la thèse porte sur un outil pour évaluer empiriquement la difficulté d'instance. L'approche proposée présente une méthode de benchmarking d'instances sur des jeux de test d'algorithmes. Nous illustrons cette méthode expérimentale pour évaluer des classes d'instances à travers plusieurs exemples d'applications sur le problème du voyageur de commerce. Nous présentons ensuite une approche pour générer des instances difficiles. Elle repose sur des opérations qui modifient les instances, mais qui permettent de retrouver facilement une solution optimale, d'une instance à l'autre. Nous étudions théoriquement et expérimentalement son impact sur les performances de méthodes de résolution.

Mot-clefs Difficulté d'instance, Modification d'instances, Science expérimentale des algorithmes, Fixed-parameter tractability (FPT), Problème du voyageur de commerce.

Characterization of hard instances for *NP*-hard problems

Abstract The empirical study of algorithms is a crucial topic in the design of new algorithms because the context of evaluation inevitably influences the measure of the quality of algorithms. In this topic, we particularly focus on the relevance of instances forming testbeds. We formalize this criterion with the notion of *instance hardness* that depends on practical performance of some resolution methods. The aim of the thesis is to introduce a tool to evaluate instance hardness. The approach uses benchmarking of instances against a testbed of algorithms. We illustrate our experimental methodology to evaluate instance classes through several applications to the traveling salesman problem. We also suggest possibilities to generate hard instances. They rely on operations that modify instances but that allow to easily find the optimal solution of one instance from the other. We theoretically and empirically study their impact on the performance of some resolution methods.

Keywords Instance hardness, Instance modification, Empirical science of algorithms, Fixed-parameter tractability (FPT), Traveling salesman problem (TSP).