UNIVERSITÉ DE MONTRÉAL

MÉTHODES DE DÉCOMPOSITION POUR LA PARALLÉLISATION DU SIMPLEXE EN NOMBRES ENTIERS

OMAR FOUTLANE
DÉPARTEMENT DE MATHÉMATIQUES ET DE GÉNIE INDUSTRIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE PHILOSOPHIÆ DOCTOR
(MATHÉMATIQUES DE L'INGÉNIEUR)
DÉCEMBRE 2018

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée :

MÉTHODES DE DÉCOMPOSITION POUR LA PARALLÉLISATION DU SIMPLEXE EN NOMBRES ENTIERS

présentée par : FOUTLANE Omar
en vue de l'obtention du diplôme de : Philosophiæ Doctor
a été dûment acceptée par le jury d'examen constitué de :

M. SOUMIS François, Ph. D., président
M. EL HALLAOUI Issmail, Ph. D., membre et directeur de recherche
M. HANSEN Pierre, Doct. Agr., membre et codirecteur de recherche
M. ALOISE Daniel, Ph. D., membre
M. RIDHA Mahjoub, Ph. D., membre externe

DÉDICACE

À mes parents Hammou et Itto
À mon oncle Ali et ma tante Mamma
À chaimae et Zakariya
À ma famille

REMERCIEMENTS

Je remercie très fortement M. Issmail El Hallaoui, directeur de recherche, pour son encadrement exceptionnel ainsi que son support continu et inconditionnel. Mes remerciements vont à M. Pierre Hansen, codirecteur de recherche, pour son accompagnement et son soutien.

Je remercie enfin tous ceux qui m'ont soutenu tout au long de mes études ainsi que ceux qui ont contribué de près ou de loin à l'accomplissement de ce travail.

RÉSUMÉ

Le SPP est un problème de la programmation linéaire en nombres entiers qui est utilisé pour modéliser des problèmes industriels dans de nombreux domaines comme la planification des horaires du personnel, la logistique et la reconnaissance de formes. Dans l'industrie de transport, il consiste à partitionner un ensemble de tâches (ex : vols d'avion, segments de trajet d'autobus...) en sous-ensembles (routes de véhicules ou rotations de personnel navigant) de sorte que les sous-ensembles sélectionnés aient un coût total minimal et que chaque tâche appartienne à un seul et unique sous-ensemble. Souvent, il est résolu par la méthode "branch and bound" ou ses variantes. Ces méthodes s'avèrent lentes dans le cas de problèmes denses de grande taille. Cependant, en industrie, il est apprécié d'avoir une solution rapidement et de tenir compte des informations disponibles telles que l'existence d'une solution initiale notamment lors de la ré-optimisation par exemple. Cet aspect est fourni aisément par les méthodes primales qui, à partir d'une solution initiale, produisent une suite de solutions à coûts décroissants qui converge vers une solution optimale. L'algorithme du simplexe en nombres entiers avec décomposition (ISUD) est une méthode primale qui, à chaque itération, décompose le problème original en deux sous-problèmes. Un premier sous-problème, appelé problème réduit, qui ne considère que les colonnes dites compatibles avec la solution courante, i.e., s'écrivant comme combinaison linéaire de colonnes/variables non dégénérées de la solution courante. Un deuxième sous-problème, appelé problème complémentaire, qui contient seulement les colonnes incompatibles avec la solution courante. Le problème complémentaire permet de trouver une direction de descente composée de plusieurs variables garantissant une solution meilleure, mais pas nécessairement entière. Dans le cas de solutions fractionnaires, un branchement en profondeur permet souvent d'aboutir rapidement à une solution entière. De nos jours, l'informatique connaît des évolutions frappantes. Les transformations que connaît le matériel informatique en termes de vitesse et de puissance sont impressionnantes : un ordinateur portable contemporain est l'équivalent des plus grosses machines des années 1970. Cette évolution induit une transformation profonde du logiciel et des algorithmes. Par conséquent, la tendance actuelle est de produire des processeurs multicœurs assimilables à des machines parallèles et de concevoir et implémenter des algorithmes parallèles.
L'objectif général de cette thèse est d'étudier les apports du parallélisme à l'algorithme d'ISUD. Le but est de proposer des implémentations parallèles d'ISUD afin d'améliorer ses performances et tirer profit des évolutions contemporaines de l'informatique. Pour concevoir ces algorithmes parallèles, nous avons exploité le parallélisme à l'intérieur d'ISUD et nous avons introduit des décompositions spécifiques au SPP.

Dans un premier temps, notre démarche est de grouper les colonnes de la solution courante en clusters afin de décomposer le problème initial en sous-problèmes indépendants. Ces derniers sont résolus en parallèle afin d'améliorer la solution courante par combinaison des solutions optimales des sous-problèmes. Pour cela, nous construisons un graphe dont les nœuds sont les colonnes de la solution courante. Nous attribuons aux arêtes des poids calculés par des fonctions de densité qui utilisent les informations issues du problème original comme le nombre de colonnes qui couvrent des tâches des colonnes A_i et A_j de la solution courante. Le graphe construit est scindé en sous-graphes et par la suite nous obtenons des clusters de la solution courante. Ainsi, nous avons ajouté une deuxième décomposition dynamique à celle qui est déjà intrinsèque à ISUD. Le résultat est un algorithme parallèle, le simplexe en nombres entiers avec double décomposition, baptisé ISU2D. Nous avons testé ce nouvel algorithme sur des instances d'horaires de chauffeurs d'autobus ayant 1600 contraintes et 570000 variables. L'algorithme réduit le temps d'exécution d'ISUD par un facteur de 3, voire 4 pour certaines instances. Il atteint la solution optimale, ou une solution assez proche, pour la majorité de ces instances en moins de 10 min alors que le solveur commercial CPLEX ne parvient pas à trouver une solution réalisable avec un gap moins de 10% après une durée de plus d'une heure d'exécution. L'algorithme ISU2D, dans sa première version, représente une première implémentation parallèle de l'algorithme du simplexe en nombres entiers. Cependant, ISU2D souffre encore de la limitation qui est l'utilisation d'une seule décomposition de la solution courante à la fois.

Dans un deuxième temps, nous améliorons ISU2D en généralisant certains aspects de son concept. Notre objectif dans cette étape est d'utiliser plusieurs décompositions dynamiques simultanément. Nous proposons un algorithme, nommé DISUD, distribué à base d'ISUD et du paradigme du système multi-agent (SMA). Chaque agent est une entité qui est, au moins partiellement, autonome et caractérisée par la décomposition dynamique de la solution courante qu'elle applique. Les agents peuvent être indépendants ou coopérants suivant la stratégie adoptée. Ainsi, nous augmentons les performances d'ISU2D et nous tirons profit d'avantage des nouveautés en matériel informatique. Les tests faits sur des instances issues de l'optimisation des horaires du personnel navigant de compagnies aériennes montrent que DISUD fonctionne mieux que DCPLEX, la version distribuée du solveur commercial de pointe CPLEX. Il atteint des solutions de qualité meilleure que le DCPLEX en réduisant le temps d'exécution par un facteur de 4 en moyenne. De plus, il a résolu des instances de grande taille que le DCPLEX n'a pu amélioré après une heure d'exécution.

Dans un troisième travail, nous réalisons l'objectif d'intégrer le DISUD dans un environnement de génération de colonnes (GC). Ce choix se justifie par le fait que l'intégration de la méthode de génération de colonnes avec les méthodes d'énumération telle le *"branch and*

price" est largement utilisé dans l'industrie. ISUD présente du potentiel pour remplacer les méthodes d'énumération usuelles pour résoudre le SPP. Par conséquent, il y a du potentiel à intégrer GC et DISUD pour traiter des problèmes de l'industrie. Nous développons donc DICG la version distribuée de génération de colonnes qui utilise DISUD. Les résultats que nous avons obtenus lors de nos tests ont montré que DICG permet d'avoir des solutions de bonne qualité et réduit le temps de calcul d'un facteur de 2 voire 4 par comparaison avec la DRMH, version distribuée de " Restricted Master Heuristic".

Avec ces trois travaux, nous pensons avoir réalisé des apports intéressants et amélioré les performances d'ISUD. En outre, nous ouvrons la voie pour des travaux futurs afin d'élargir les utilisations de la version distribuée d'ISUD comme par exemple, rendre les agents plus intelligents via des algorithmes d'apprentissage.

ABSTRACT

SPP is an integer linear programming problem that is used to model many industrial problems such as personnel scheduling, logistics and pattern recognition. In the transport industry, it consists of partitioning a set of tasks (plane flights, bus itinerary segments, ...) into subsets (rotation of navigating personnel) so that the selected subsets have a minimum total cost and each task belongs exactly to one subset. Usually, SPP is solved by the branch and bound method or its variants. These methods are known to be slow in the case of large and difficult problems. However, in industry it is appreciated to have a solution as quickly as possible and to consider available information such as the existence of an initial solution, especially in the re-optimization case. This aspect is easily provided by the primal methods which from an initial solution produce a sequence of decreasing cost solutions that converge towards an optimal or near optimal solution.

The Integral Simplex Using Decomposition, ISUD, is a primal method dedicated to solve SPP. At each iteration, it decomposes the original problem into two sub-problems. The first, called the reduced problem (RP), only considers the so-called compatible columns with the current solution. The second, called the complementary problem (CP), deals only with the columns that are incompatible with the current solution. The complementary problem makes it possible to find a descent direction composed of several variables that could be fractional or integer solution. In the case of fractional solutions, a branching often leads to an integer solution.

Nowadays, computing science evolves impressively. The transformation of computer hardware into speedy machines is spectacular : a current laptop is equivalent to the 1970s biggest machines. The current trend is to produce multi-core processors and to design and implement parallel computing techniques.

The general objective of this thesis is to study and apply parallel computing techniques to ISUD. We propose parallel implementations of ISUD in order to improve its performances and to profit from the contemporary evolution of the computer science. To design these algorithms we have exploited parallelism within ISUD and introduced specific decompositions.

At first, we group the columns of the current solution into clusters in order to decompose the initial problem into independent sub-problems. These will be solved in parallel to get an improving solution by combining the sub-problems optimal solutions. To do so, we construct a graph whose nodes are the current solution columns. The edge (i, j) weight is computed by weighting functions that use the information from the original problem such as the number of columns that span two columns A_i and A_j of the current solution. Then, the constructed

graph is split into sub graphs and as a result to a set of the current solution clusters. Thus, we add a second dynamic decomposition to the RP-CP one which is intrinsic to ISUD. We obtain a parallel algorithm, The Integral Simplex Using Double Decomposition, called ISU2D. We tested it on instances of bus drivers having up to 1600 constraints and 570000 variables. The ISU2D reduces the computing time of ISUD by a factor of 3, even 4 for some instances. It reaches an optimal or near optimal solution for the majority of these instances in less than 10 min while the commercial solver CPLEX cannot even find a feasible solution with a gap that is less than 10 % after a one-hour time limit. But, ISU2D suffers of the limitation which is the use of a single decomposition of the current solution at a time.

In a second step, we improve our algorithm by generalizing the second decomposition concept. Indeed, our goal is to use multiple dynamic decompositions simultaneously. We propose an algorithm, called DISUD, a distributed algorithm based on ISUD and the multi-agent system (MAS). Each agent is an entity that is, at least partially, autonomous and characterized by the dynamic decomposition that it applies. We implemented two variants where agents can be independent or cooperating according to the strategy adopted. Thus, we increase the performance of ISU2D and benefit more from computing hardware evolution. We tested DISUD on airplane flight scheduling problems. The obtained results show that DISUD is better than DCPLEX, the distributed version of the advanced CPLEX commercial solver on our test instances. It achieves better quality solutions than the DCPLEX and reduces the computing time by an average factor of 4 to 5 for some instances. In addition, it solved large instances that the DCPLEX could not improve after a one-hour time limit.

In a third work, we integrate the DISUD in a column generation context (GC). This choice is justified by the fact that the coupling of the method of generating columns with enumeration methods such as *branch and price* is widely used in industry. ISUD has potential to replace the usual enumeration methods to solve the SPP. As a result, there is potential to integrate GC and DISUD to address industry issues. We develop DICG a column generation algorithm which uses DISUD instead of enumeration methods. The results that we obtained during our tests showed that DICG solutions are of good quality (less than 1%). Moreover, it reduces the time of computation by a factor of 2 or even 4 compared to the DRMH, a distributed version of the Restricted Master heuristic.

Thus, we have contributed to ISUD evolution. In addition, we improved the performances of ISUD and reduced its computing time. Furthermore, we paved the way for future work to expand the uses of the distributed version of ISUD.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

SPP	Set Partitioning Problem
CG	Column Generation
IPS	Improved Primal Simplex
RP	Reduced Problem
CP	Complementary Problem
CPP	Crew Pairing Problem
VCSP	Vehicle and Crew Scheduling Problem
ISUD	Integral Simplex Using Decomposition
ISU2D	Integral Simplex Using Double Decomposition
DVD	Disjoint Vertical Decomposition
IVD	Incremental Vertical Decomposition
DISUD	Distributed Integral Simplex Using Decomposition
DISUD_{comp}	Competetive variant of DISUD
DISUD_{coop}	Cooperative variant of DISUD
DICG	Distributed Integral Column Generation
DICG_{comp}	Competetive variant of DICG
DICG_{coop}	Cooperative variant of DICG
MAS	Multi Agent System
SS	Set Of Solutions
CGSP	Column Generation Sub Problem
SP	Sub Problem
We	Weighting function
W	Weight vector
G(V,E)	Graph with vertices set V and Edges set E
RMH	Restricted Master Heuristic
DRMH	Distributed version of RMH
SPPRC	Shortest Path Problem with Resource Constraints

LISTE DES ANNEXES

CHAPITRE 1 INTRODUCTION

Le problème de partitionnement d'ensemble (SPP : *Set Partitioning Problem*) est un problème de la programmation linéaire en nombres entiers. Il a fait l'objet de nombreuses recherches et études. Il est très utilisé pour modéliser plusieurs situations dans l'industrie comme la planification des horaires d'équipages d'avions, de trains et horaires des chauffeurs d'autobus (voir Ribeiro & Soumis (1991), Desaulniers et al. (1994), Chu et al. (1997), Hoffman & Padberg (1993)).
La formulation simple du SPP est la suivante :

$$(SPP) \qquad \begin{aligned} min\ \ & c^t x \\ & Ax = e \\ & x_j\ \ binaire,\ \ j = 1, ..., n \end{aligned}$$

Où A est une matrice binaire (m, n) de plein rang, c est le vecteur coût, et e est le vecteur unitaire.

Afin de répondre aux besoins de l'industrie, des efforts considérables ont été dédiés au développement et l'amélioration des méthodes de résolution du SPP. Souvent, les méthodes de résolution du SPP résolvent la relaxation linéaire, où les variables binaires deviennent des réels positifs dans le problème relaxé, et trouvent une solution entière par énumération implicite. Les méthodes lesplus utilisées sont des variantes de la méthode d'énumération implicite *branch and bound* telle que la méthode *branch and cut* qui combine le *branch and bound* avec la méthode des coupes et la méthode *branch and price* qui combine la méthode *branch and bound* avec la méthode de génération de colonnes GC. Ces méthodes ne sont pas efficaces quand la taille du problème devient grande. Ceci est dû au fait que le SPP est un problème NP-Difficile (voir Garey & Johnson (1979)) connu pour sa dégénérescence. En fait, les besoins font que la taille des instances à résoudre grossit de jour en jour. Par exemple, dans le domaine de l'industrie aérienne, les fusions entre les compagnies aériennes et l'augmentation du trafic aérien font que les problèmes de planification des horaires du personnel sont devenus plus grands comparés à que ce qu'ils étaient auparavant.
Une nouvelle méthode de résolution (ISUD) dédiée aux problèmes SPP a été développée par Zaghrouti et al. (2014). Elle repose sur la méthode IPS, *Improved Primal Simplex*, introduite par El Hallaoui et al. (2011). Ils ont proposé un nouveau concept de décomposition intrinsèque aux problèmes dégénérés : *optimiser d'abord dans un espace réduit qui est le sous-espace vectoriel des variables (colonnes) de base non-dégénérées dites compatibles, chercher ensuite*

une ou plusieurs directions de descente (réalisables : menant à des solutions entières) dans le sous-espace complémentaire et réitérer jusqu'à l'optimalité. Étant donné une solution courante, un problème réduit (RP) est défini à partir des variables compatibles. Il est en fait le problème de partitionnement (SPP) restreint aux variables compatibles et aux contraintes non dégénérées. De façon similaire, le problème de partitionnement (SPP) restreint aux variables non compatibles définit le problème complémentaire (CP).

Depuis, des travaux ont été réalisés afin d'améliorer les performances de cette méthode prometteuse et d'élargir son domaine d'application. Ainsi, Rosat et al. (2016) ont comparé l'impact de différentes contraintes de normalisation du cône de directions utilisé par ISUD. En outre, Rosat et al. (2017a) ont étudié l'impact de l'ajout de coupes sur ISUD. Quant à Zaghrouti et al. (2013), ils ont développé l'algorithme Zoom basé sur ISUD qui explore un voisinage de la solution fractionnaire courante quand il est difficile de trouver une solution entière améliorée en utilisant la version originale d'ISUD.

La conception d'algorithmes parallèles a connu un essor important à nos jours avec l'apparition d'ordinateurs parallèles dans le monde du calcul scientifique (voir Pardalos et al. (1994)). En effet, le développement des architectures informatiques a fait que de plus en plus de ressources informatiques sont disponibles pour résoudre les problèmes de l'optimisation combinatoire. Cette classe de problèmes admet, pour les plus grandes instances, un grand nombre de solutions possibles et nécessite la conception de plates-formes logicielles efficaces. À titre d'exemple, la littérature comporte plusieurs travaux sur *le branch and bound* parallèle (Ralphs et al. (2018)). Cette tendance de production de méthodes de résolution parallèle se trouve renforcée par les aspects de parallélisme que l'on trouve lors de la résolution des problèmes d'optimisation. À titre d'exemple, pour les problèmes d'affectation des équipes (rotation) avec minimisation des coûts, il est clair que le facteur région est un instigateur du parallélisme. En effet, il n'est pas optimal d'affecter des équipes aux régions lointaines vu le coût induit. Ceci favorise une division du problème original en sous-problèmes par décomposition géographique des équipes et des affectations. Il en est de même pour l'affectation des équipages par intervalles de temps. Les équipes assurant les vols du début du mois ne peuvent accomplir dans le même pairing ceux de la fin du mois vu les contraintes de longueur sur les pairings. Ainsi, le parallélisme s'impose comme une bonne approche pour la production de méthodes efficaces de résolution des instances de très grande taille en termes de temps d'exécution et de la qualité de la solution obtenue.

Dans cette thèse, nous étudions les apports du calcul parallèle quant à l'amélioration des performances d'ISUD. Nous proposons de produire une méthode de résolution parallèle du SPP à base d'ISUD. Notre démarche est de décomposer les colonnes de la solution courante en clusters afin de décomposer le problème initial en sous-problèmes indépendants. Ces derniers

sont résolus en parallèle pour améliorer la solution courante par combinaison des solutions optimales des sous-problèmes. Notre approche utilise des fonctions de densité pour mesurer le couplage des colonnes de la solution courante et déduire par la suite la décomposition appropriée du problème original. Ainsi, nous avons introduit une deuxième décomposition dynamique à celle qui est déjà intrinsèque à ISUD. Le résultat est un algorithme parallèle baptisé ISU2D. Nous enrichissons ensuite notre méthode par une généralisation de ce concept de double décomposition dynamique. Nous proposons alors un algorithme, nommé DISUD, distribué à base d'ISUD et du paradigme du système multi-agent (SMA). Chaque agent est, au moins partiellement, autonome et caractérisé par la décomposition qu'il applique. Les agents peuvent être indépendants ou coopérants suivant la stratégie adoptée. Ainsi, nous augmentons les performances d'ISU2D et nous tirons profit davantage des progrès en informatique. Enfin, après avoir produit DISUD, nous l'intégrons dans un environnement de génération de colonnes (GC). Ce choix se justifie par le fait que le couplage de la méthode de génération de colonnes avec les méthodes d'énumération tel que le *branch and price* est largement utilisé dans l'industrie. En outre, les tests réalisés montre que ISUD présente du potentiel pour remplacer les méthodes d'énumération usuelles pour résoudre le SPP. Par conséquent, il y a du potentiel à intégrer GC et DISUD pour traiter des problèmes de partitionnement avec contraintes supplémentaires dans des travaux futurs, ce qui élargit le domaine d'utilisation d'ISUD pour résoudre la catégorie des problèmes apparentés au SPP.

Sur le plan pratique, nous considérons que les algorithmes que nous introduisons sont très compétitifs avec les méthodes les plus utilisées en industrie et qui sont déjà très matures. Nous effectuons nos tests sur des instances de rotations d'équipages aériens (2 000 vols et 1 000 000 pairings) ainsi que des instances de grande taille de problèmes d'horaires de chauffeurs d'autobus (1 600 tâches et 570 000 chemins). ISU2D arrive à réduire le temps d'exécution d'ISUD par un rapport de 3 en moyenne alors que le DISUD est compétitif avec la version distribuée du CPLEX et affiche un facteur de réduction de 4 en moyenne, voire plus. Ajoutons à cela le fait que la version distribuée de la génération de colonnes à base d'ISUD, (DICG), affiche un facteur de réduction allant de 2 à 5 par comparaison avec la DRMH. Cette dernière est la version distribuée de la méthode *" restricted master heuristic"* (RMH). Ces résultats sont encourageants vu que l'amélioration des temps d'exécution est toujours un besoin incessant surtout pour les traitements en temps réel et la ré-optimisation. Les résultats de cette recherche ont donné naissance à trois articles qui font l'objet des chapitres 4, 5 et 6. Le chapitre 3 donne une brève description de chaque article.

CHAPITRE 2 REVUE DE LITTÉRATURE

Notre revue de littérature rappelle des notions fondamentales de l'optimisation combinatoire en liaison avec notre sujet de thèse. Dans la section 2.1, nous présentons des rappels théoriques sur le SPP ainsi que les premières méthodes de type simplexe en nombres entiers dédiées à sa résolution. Dans la section 2.2, nous décrivons l'algorithme du simplex en nombres entiers utilisant la décomposition $ISUD$ et ses variantes qui constituent le fondement de base pour notre travail. Dans la section 2.3, nous discutons des notions du calcul parallèle et principalement de la parallélisation de la méthode de séparation et d'évaluation *branch and bound* ainsi que certaines applications de la parallélisation. Enfin dans la section 2.4, nous décrivons brièvement la méthode de la génération de colonnes.

2.1 Rappels Théoriques du PL et du SPP

2.1.1 Programmation Linéaire et Dégénérescence

Le simplexe a été introduit par George Dantzig en 1946. C'est un algorithme de résolution de problèmes d'optimisation linéaire. Il consiste à trouver une valeur optimale d'une fonction linéaire de n variables réelles $(x_1, x_2, ..., x_n)$ sur un ensemble défini au moyen de contraintes linéaires d'égalité. Par conséquent, l'ensemble des solutions admissibles au problème est donc un polyèdre convexe. La méthode du simplexe est une méthode itérative qui parcourt les sommets du polyèdre convexe jusqu'à atteindre un sommet optimal. Dans le cas d'un problème de minimisation, $\{\min cx\,;\, Ax = b,\, x >= 0\}$ où A est une matrice de taille $(m \times n)$ et de rang m. Soit donc, $(x_{j_1}, x_{j_2}, ..., x_{j_m})$, m variables correspondantes aux colonnes de base de A. Une solution admissible pour ce système est obtenue en posant les $(n - m)$ variables correspondantes aux colonnes libres égales à zéro. Ces variables sont appelées variables hors base. Ensuite, on résout le système pour les m variables de base où B= $(x_{j_1}, x_{j_2}, ..., x_{j_m})$. La solution obtenue est composée des variables de base et des variables hors base. Lors de la résolution ordinaire d'un problème PL, il arrive que l'une des variables de base B= $(x_{j_1}, x_{j_2}, ..., x_{j_m})$ soit nulle. On parle alors de solution de base dégénérée. Même si la base change, le simplexe ne change pas de point extrême et la valeur de la fonction objectif reste pareille lors de cette itération : on parle alors de phénomène de dégénérescence. La règle de Bland d'anticyclage assure l'arrêt de l'algorithme en un nombre fini d'itérations. Le SPP est un problème qui est connu pour sa forte dégénérescence.

2.1.2 Quasi Intégralité du SPP

Trubin (1969) a montré que toute arête de l'enveloppe convexe des solutions du problème de partitionnement est aussi une arête du polytope du problème relaxé pour le SPP. Cette propriété porte le nom de la quasi-intégralité. Elle révèle l'existence d'un chemin, ne contenant que des sommets entiers, entre toute paire de sommets entiers de l'enveloppe convexe du SPP. Tenant compte de cette propriété, plusieurs chercheurs ont proposé des variantes du simplexe pour résoudre le SPP et trouver les sommets qui mènent vers une solution optimale.

2.1.3 Méthodes Classiques à Pivots Entiers

Cette catégorie regroupe plusieurs méthodes de résolution dédiées au SPP dont les méthodes des suites intégrales, de base entière et du simplexe en nombres entiers.
Balas & Padberg (1975) ont repris les résultats de Trubin et ont montré que la borne supérieure sur le nombre de pivots nécessaires, pour passer d'un sommet entier à un sommet optimal est m, le rang de la matrice des contraintes A. En outre, ils ont prouvé l'existence d'une suite de solutions entières ayant des coûts décroissants qui converge vers une solution optimale. Ils ont proposé un mécanisme pour chercher les termes de la suite.
Haus et al. (2001) ont proposé une méthode pour la résolution des problèmes en nombres entiers dite "méthode de base entière". L'algorithme cherche, parmi les colonnes hors base, une colonne qui peut améliorer la solution courante. Ensuite, l'algorithme choisit une colonne et étudie la possibilité de la remplacer par une combinaison de colonnes pour améliorer la solution courante (voir Zaghrouti (2016) pour plus de détails).
Thompson (2002) a mis en œuvre la méthode du Simplexe en Nombres Entiers *(Integral Simplex Method)*. Celle-ci se déroule en deux phases : la première dite "méthode locale", et la deuxième nommée "méthode globale". La méthode locale consiste à examiner les possibilités pour effectuer des pivots sur des 1. La méthode globale construit un arbre de traitement dont chaque nœud représente un sous-problème pour chaque colonne ayant un coût réduit négatif. Ensuite, on résout avec la méthode locale les sous problèmes correspondants aux branches (nœuds parents et fils) de l'arbre construite. La solution retenue est la meilleure de toutes les solutions obtenues par exploration de l'arbre (voir Zaghrouti (2016) pour plus de détails).
Saxena (2003) a amélioré les performances de la méthode du Simplexe en Nombres Entiers en utilisant des règles d'anti-cyclage et des heuristiques. Ses apports permettent de réduire la taille de l'arbre de branchement de Thompson. En outre, il a démontré qu'une solution optimale peut être atteinte en n'utilisant que des pivots sur 1 (voir Zaghrouti (2016) pour plus de détails).

2.2 Integral Simplex Utilisant la Décomposition (ISUD)

Nous ne pouvons pas présenter ISUD sans mentionner son homologue le Simplexe Primal Amélioré (IPS) développé par El Hallaoui et al. (2011). Ce dernier est un algorithme réputé être efficace pour traiter les problèmes linéaires dégénérés. Aussi avons-nous jugé utile de parcourir les principes de celui-ci qui seront, par la suite, adaptés par ISUD pour la résolution du SPP.

2.2.1 Simplexe Primal Amélioré

L'idée de l'algorithme IPS repose sur le résultat connu des espaces vectoriels réels qui est le suivant : étant donné une famille de vecteurs indépendants F= (v_1, v_2, ...,v_l) de l'espace vectoriel $\mathbb{R}^m$, $vect(v_1, v_2, ..., v_l)$ est un sous espace vectoriel de $\mathbb{R}^m$ et nous avons :

$$\mathbb{R}^m = vect(v_1, v_2, ..., v_l) \cup (\mathbb{R}^m \setminus vect(v_1, v_2, ..., v_l))$$

Par conséquent, toute combinaison de colonnes qui améliore la solution courante $\bar{x}$, (direction de descente), est la réunion de deux familles de colonnes. La première contient des colonnes de l'espace vectoriel vect (supp($\bar{x}$)) où supp($\bar{x}$) est l'ensemble des indices des variables non nulles de la solution courante. La deuxième est formée par des colonnes appartenant à ($\mathbb{R}^m \setminus$ $vect(supp(\bar{x}))$. Ainsi, étant donné la solution courante $\bar{x}$, à chaque itération, IPS décompose le problème à traiter en deux sous-problèmes :

- Un problème réduit (PR) qui traite uniquement les colonnes appartenant à vect (supp($\bar{x}$)).
- Un problème complémentaire (PC) qui considère les colonnes appartenant à ($\mathbb{R}^m \setminus$ $vect(supp(\bar{x}))$.

Cette décomposition induit la notion suivante de *compatibilité* (voir El Hallaoui et al. (2011)). Par définition, une colonne est dite *compatible* si elle peut s'écrire comme combinaison linéaire des colonnes de la solution (variables non dégénérées) et donc appartient à vect (supp($\bar{x}$)). Elle est considérée incompatible dans le cas contraire. Ainsi, le problème réduit améliore la solution courante en échangeant une colonne compatible de moindre coût, une à la fois, avec certaines des colonnes de la solution, tandis que l'objectif du problème complémentaire est de trouver une combinaison de colonnes incompatibles qui devrait améliorer, après échange, le coût de la solution courante. Le processus s'arrête quand ni le problème réduit ni le problème complémentaire ne peuvent améliorer la solution courante.

La formulation originale du problème complémentaire est la suivante :

$$\begin{aligned} min \sum_{j \in I} c_j x_j - \sum_{l \in L} c_l \lambda_l, \\ \sum_{j \in I} x_j A_j = \sum_{l \in L} \lambda_l A_l, \\ \|x\|_1 \leq 1, \\ x \geq 0, \end{aligned}$$

où I est l'ensemble des indices des variables incompatibles et L l'ensemble des indices des variables non nulles de la solution courante, i.e., supp($\bar{x}$). Les variables x_j non nulles désignent les variables entrantes tandis que les λ_l non nulles désignent les variables sortantes. Évidemment, la différence des coûts entre les variables entrantes et sortantes (coût réduit) doit être négative pour un problème de minimisation afin de pouvoir améliorer la fonction objectif. Sinon, l'optimalité est atteinte. Un calcul matriciel permet aussi d'écrire le problème complémentaire sous la forme équivalente suivante :

$$\begin{aligned} min \quad & \bar{c}x, \\ & Mx = 0, \\ & \|x\|_1 \leq 1, \\ & x \geq 0 \end{aligned}$$

où $\bar{c}$ est le vecteur du coût réduit et M une matrice de projection sur le complémentaire de $vect(supp(\bar{x}))$, dite matrice de compatibilité.
Les tests ont montré que l'IPS permet d'augmenter considérablement l'efficacité de la méthode du simplexe envers la dégénérescence pour les problèmes linéaires. Le succès d'IPS a donné naissance à l'idée de l'adapter pour vaincre la dégénérescence rencontrée au niveau des problèmes SPP et par la suite à l'élaboration de l'algorithme ISUD.

2.2.2 Aperçu d'ISUD

Notons d'abord que le concept de compatibilité introduit par El Hallaoui et al. (2011) peut être énoncé dans le cas du SPP comme suit :

Definition 2.2.1. *Une combinaison de colonnes est dite compatible avec une solution $\bar{x}$ si elles peuvent remplacer certaines colonnes de son support supp($\bar{x}$) = $\{A_j$ tel que $x_j = 1\}$, pour donner une nouvelle solution réalisable $\bar{x}'$.*

L'ISUD, par construction, est un algorithme séquentiel à deux niveaux dédié à la résolution

de problèmes de partitionnement. En premier niveau, ISUD cherche à améliorer la solution courante en résolvant un problème réduit (PR) (voir section 2.2.3). En deuxième niveau, il consiste à résoudre un problème complémentaire (PC) (voir section 2.2.4). ISUD itère entre les deux étapes (voir algorithm 1) jusqu'à ce qu'il atteigne une solution optimale. (Zaghrouti et al. 2014) présentent le pseudo algorithme d'ISUD comme suit :

Algorithm 1 Pseudo code d'ISUD

Soit une solution entière initiale : $x_c = x_0$
niveau (PR) : Améliorer la solution courante en résolvant (PR).
niveau (PC) : Résoudre (PC) pour trouver une direction de descente d_{PC}.
Si d_{PC} est entière,
Mettre $x_c = x_c + d_{PC}$ et aller au niveau (PR)
Si d_{PC} est fractionnaire,
Brancher pour trouver direction de descente entière d'_{PC},
Mettre $x_c = x_c + d'_{PC}$ et aller au niveau (PR)
Si aucune direction de descente entière n'existe, x_c est optimale.
Fin Si

2.2.3 Problème Réduit d'ISUD

Soit $\bar{x}$ une solution entière à un problème de partitionnement donné $\{\min\ cx\,;\ Ax = e,\ x >= 0\}$ où A est une matrice de taille $(m \times n)$ et de rang m. Soit P l'ensemble d'indices des colonnes tel que $P = \text{supp}(\bar{x})$ et posons $N = (m$ - $|P|)$. Nous pouvons permuter les colonnes de la matrice de contraintes de manière à obtenir $Ax = e$ avec $x = [x_P, x_{n-|P|}]$ où $x_P \in \mathbb{R}^{|P|}$. Considérons B une base associée à $\bar{x}$, (voir Zaghrouti et al. (2014)), tel que :

$$B = \begin{bmatrix} I_P^P & 0 \\ A_P^N & I_N^N \end{bmatrix},\ \text{donc}\ B^{-1} = \begin{bmatrix} I_P^P & 0 \\ -A_P^N & I_N^N \end{bmatrix}\ \text{car}\ \begin{bmatrix} I_P^P & 0 \\ -A_P^N & I_N^N \end{bmatrix}\ \begin{bmatrix} I_P^P & 0 \\ A_P^N & I_N^N \end{bmatrix} = \begin{bmatrix} I_P^P & 0 \\ 0 & I_N^N \end{bmatrix}.$$

Où I_P^P et I_N^N sont les matrices identité d'ordre $|P|$ et $|N|$ respectivement. A_P^N est la sous matrice de taille $N \times p$ où les lignes sont indexées par N et les colonnes sont indexées par P. D'une façon générale, les exposants indiquent les lignes et les indices indiquent les variables. En multipliant par B^{-1}, l'équation des contraintes $Ax = e$ devient

$$\bar{A}x = \bar{e}\ \textit{ ou encore }\ \begin{bmatrix} \bar{A}^P \\ \bar{A}^N \end{bmatrix}\ \begin{bmatrix} x^P \\ x^N \end{bmatrix} = \begin{bmatrix} \bar{e}^P \\ \bar{e}^N \end{bmatrix},$$

Avec

$$\begin{bmatrix} \bar{A}_j^P \\ \bar{A}_j^N \end{bmatrix} = \begin{bmatrix} A_j^P & \\ -A_P^N & A_j^P + A_j^N \end{bmatrix}, \quad \text{et} \quad \begin{bmatrix} \bar{e}^P \\ \bar{e}^N \end{bmatrix} = \begin{bmatrix} I & 0 \\ -A_P^N & I \end{bmatrix} \begin{bmatrix} e^P \\ e^N \end{bmatrix} = \begin{bmatrix} e^p \\ 0 \end{bmatrix}.$$

Soient C et I les ensembles des indices des colonnes compatibles et incompatibles respectivement. De même, nous pouvons écrire $x = [x_C, x_I]$, $A = [A_C, A_I]$, et $c = [c_C, c_I]$. En utilisant ces vecteurs et matrices, Zaghrouti et al. (2014) définissent le problème réduit (PR) en imposant $x_I = 0$:

$$\begin{aligned} & & Minimize \quad & c_c x_c & (2.1) \\ (PR) & & subject\ to \quad & \bar{A}_C^P x_C = e^P & (2.2) \\ & & & x_C \geq 0. & (2.3) \end{aligned}$$

Notons que par construction, le problème (PR) dépend de la solution courante et change par conséquent au cours de la résolution.

2.2.4 Problème Complémentaire d'ISUD

Soit x_{PR} la solution entière obtenue au niveau du problème réduit (PR). ISUD résout un problème complémentaire (PC) pour chercher les colonnes incompatibles qui améliorent la solution courante x_{PR}. En d'autres termes, (PC) cherche une direction de descente d_{PC}. Zaghrouti et al. (2014) formulent (PC) comme suit :

$$\begin{aligned} & & Minimize \quad & \bar{c}_I \cdot x_I & (2.4) \\ (PC) & & subject\ to \quad & \bar{A}_I^N x_I = 0 & (2.5) \\ & & & e \cdot x_I = 1,\ x_I \geq 0. & (2.6) \end{aligned}$$

Où, $\bar{A}_I^N = -A_P^N A_I^P + A_I^N$, et $\bar{c}_I = c_I - A_I^P c_P$. Zaghrouti et al. (2014) ont montré que si x_C^* est une solution optimale du (PR) et (PC) est non réalisable ou $z^{PC} \geq 0$ alors $(x_C^*, 0)$ est une solution optimale du (SPP). Sinon, ISUD cherche une combinaison de colonnes disjointes qui améliore la solution courante du problème(PR). El Hallaoui et al. (2011) définissent la notion de colonnes disjointes comme suit :

Definition 2.2.2. *Deux colonnes A_j et A_i sont dites disjointes, également orthogonales, si les ensembles des tâches qu'elles couvrent sont disjoints i.e. $A_j.A_i = 0$. Une combinaison de colonnes est dite disjointe si toute paire de deux colonnes différentes de la combinaison est orthogonale.*

Ainsi, Zaghrouti et al. (2014) ont montré qu'un ensemble S de colonnes disjointes qui est solution du PC permet d'améliorer la solution courante $\bar{x}$ et obtenir une solution entière à moindre coût x^{**} moyennant la formule suivante :

$$x_j^{**} = \begin{cases} 1, & j \in S \cup (P - S^+) \; Avec \; S^+ = \{k \in P | \sum_{j \in S} \bar{a}_{kj} = 1\} \\ 0, & \text{Sinon} \end{cases}$$

Pour trouver des colonnes disjointes, ISUD implémente des techniques de séparation et construit un arbre de branchement pour éliminer les solutions non disjointes trouvées lors de la résolution du (PC). En outre, ISUD applique une stratégie de recherche en profondeur pour parcourir l'arbre de branchement obtenu. Les tests effectués montrent que ISUD favorise intrinsèquement l'intégralité des solutions produites. Il trouve facilement les combinaisons de colonnes recherchées par Balas et Padberg (voir Zaghrouti et al. 2014). Ces résultats prometteurs nous ont poussés à vouloir en profiter pour proposer nos méthodes de résolution parallèles à base d'ISUD.

2.2.5 Variantes d'ISUD

Depuis son introduction en 2014, ISUD a été l'objet de travaux de recherches pour améliorer ses performances et favoriser l'intégralité de la solution du problème complémentaire. Rosat et al. (2016) ont revu la contrainte 2.6 dite la contrainte de normalisation. Dans la version originale de l'algorithme, les coefficients de cette contrainte sont unitaires. Ils ont généralisé les coefficients de cette contrainte dont la direction de descente déterminée par le problème complémentaire en dépend fortement. Ils ont déterminé de nouvelles propriétés spécifiques à certains choix des coefficients de la contrainte de normalisation telle que le nombre de tâches dans le but de pénaliser les directions fractionnaires, i.e., menant à des solutions fractionnaires, et de favoriser des directions entières. En outre, Rosat et al. (2017a) ont discuté de l'adaptation des méthodes de plans coupants utilisés en programmation linéaire en nombres entiers au cas d'ISUD. Ils ont proposé des méthodes de séparation pour les coupes primales de cycle impair et de clique.

D'autre part, Zaghrouti et al. (2013) ont proposé une version "Zoom" qui propose d'éviter d'implémenter un branchement complexe et exhaustif en laissant au problème réduit le soin de trouver une solution entière au voisinage de la direction retournée par le problème complémentaire. Ainsi, un ensemble de colonnes compatibles avec la direction fractionnaire est résolu par le problème réduit. Dans un travail de recherche subséquent, Zaghrouti et al. (2018) ont modifié le problème complémentaire pour permettre de trouver des directions de descente simultanément. Ceci se fait par le biais de l'introduction d'une colonne artificielle

qui couvre toutes les tâches du problème et dont le coût est celui de la solution courante. Ce nouveau modèle permet de réduire le nombre d'itérations d'ISUD et par la suite le temps total de calcul.
Dans cette thèse, nous prenons en considération ces améliorations et nous les adaptons pour mener à bien notre objectif, qui est l'étude des apports du parallélisme sur ISUD.

2.3 Résolution Parallèle du SPP

Les besoins et les progrès industriels ont répandu l'utilisation des machines de calcul puissantes ayant des processeurs multicœurs. Ceci a encouragé la parallélisation des traitements et par la suite la production des algorithmes parallèles. Pour un problème donné, la conception d'un algorithme parallèle nécessite, parmi d'autres, le contrôle de la distribution des données, la répartition des traitements et la synchronisation entre les processeurs. Il y a deux façons pour concevoir un algorithme parallèle : l'une consiste à exploiter et introduire le parallélisme à l'intérieur des blocs d'un algorithme séquentiel déjà existant, l'autre conçoit complètement un nouvel algorithme parallèle. Pour ce qui suit, nous allons mettre l'accent sur la parallélisation de la méthode de séparation et d'évaluation *branch and bound* vu que ses variantes sont les plus utilisées pour la résolution du SPP. Ensuite, nous mentionnons et passons en revue certaines méthodologies de parallélisation utilisées dans l'optimisation combinatoire.

2.3.1 Résolution Parallèle à base de la Méthode de Séparation et Évaluation

La méthode séquentielle de séparation et d'évaluation énumère, via des contraintes, l'ensemble des solutions SS du problème d'optimisation à résoudre en examinant les sous-ensembles de SS. Le principe est de partitionner SS en sous-ensembles via l'étape de séparation "branch" qui ajoute de nouvelles contraintes au cours de la résolution. Elle construit un arbre dont les nœuds représentent les partitions obtenus par branchement tandis que les arcs reproduit l'ordre chronologique en liant les nœuds enfants aux nœuds parents. Au cours de la résolution, un nœud est sélectionné de la file des nœuds actifs puis évalué. Dans le cas d'un problème de minimisation, il est soit élagué si sa valeur est supérieure à la meilleure solution connue (borne supérieure), ou bien ajouté à la file des nœuds actifs dans le cas contraire. Le choix du nœud à traiter est régi par ce qu'on appelle une stratégie de recherche ou d'exploration de l'arbre. Par exemple, il y a la stratégie de recherche en profondeur d'abord où on priorise le nœud le plus récent. Le déroulement du traitement est défini principalement par l'ensemble des nœuds avec leurs priorités, la borne supérieure du problème, et la stratégie d'exploration. La littérature de la parallélisation de la méthode de séparation et d'évaluation est abondante

(voir Crainic et al. (2006), Trienekens & Bruin (1992), Gendron & Crainic (1994) et Ralphs et al. (2017)). Trienekens & Bruin (1992) identifient un algorithme parallèle de la méthode de séparation et d'évaluation en définissant deux niveaux de parallélisation :

- Le niveau bas : le déroulement de l'algorithme de séparation et d'évaluation parallèle est similaire à celui de l'algorithme séquentiel. Le parallélisme dans ce cas se limite uniquement à accélérer la résolution.
- Le niveau haut : les effets et les conséquences du parallélisme a de l'impact sur la logique et la chronologie des traitements de l'algorithme. Le déroulement de l'algorithme parallèle et son arbre de branchement ne sont pas les mêmes que ceux de l'algorithme séquentiel.

En pratique, les deux niveaux peuvent être appliqués aux branches de l'arbre de branchement pour concevoir une méthode parallèle complexe. Par exemple, l'algorithme peut adopter des explorations concurrentes, qui utilisent des stratégies d'exploration de l'arbre différentes. Un exemple de ce type de parallélisation est de construire des arbres de *branch and bound* avec des stratégies de branchement différentes (voir Miller & Pekny 1993). On peut aussi faire construire des arbres de "branch and bound" sur chaque processeur en utilisant des stratégies de parcours différentes (voir Janakiram et al. 1988). Ajoutons, à ce qui a été dit, le fait que différents modèles existent pour le contrôle des processeurs. Nous distinguons :

- Modèle maître-esclaves : le processus maître contrôle l'échange d'information, détermine la terminaison de la recherche et spécifie le travail que les processus esclaves doivent faire.
- Modèle point à point : tous les processus se transmettent les informations lors de la résolution et prennent les décisions concernant les traitements à faire sans passer par une entité centrale comme c'est le cas précédemment.

À l'échelle industrielle, plusieurs solveurs ont adapté leurs algorithmes au calcul parallèle tels que CPLEX et GUROBI. Ainsi, en plus du *multithreading* qui applique la parallélisation de bas niveau, les équipes de développement de CPLEX ont produit une version distribuée qui applique la parallélisation de haut niveau et adopte le modèle maître-esclaves. En effet, depuis 2013, avec la version 12.5.1, CPLEX a adapté ses algorithmes de programmation linéaire mixte en nombres entiers (MIP) aux architectures multi-cœurs. Par conséquent, il est possible d'utiliser plusieurs machines pour résoudre des SPP aussi bien que d'autres problèmes particuliers. Pour cela, CPLEX fournit deux approches algorithmiques :

- L'approche simultanée où le problème est résolu indépendamment sur chaque machine, mais avec des paramètres différents.
- L'approche distribuée qui est divisée en deux phases. Dans la première dite phase de *ramp up*, le problème est résolu sur chaque machine avec des paramètres différents. À

la fin de cette phase, l'optimisation simultanée est arrêtée sur toutes les machines et l'arborescence de recherche MIP de la machine qui a le meilleur résultat est répartie entre les machines. Ces dernières fonctionnent alors en collaboration pour résoudre le problème.

Cependant, l'approche parallèle de la méthode évaluation et séparation souffre de plusieurs problèmes (voir Ralphs et al. (2017)). Ils mentionnent que les temps passés dans les différents nœuds (nœud racine en particulier) sont disproportionnés, que l'arbre de recherche devienne non équilibrée avec le temps, que la construction dynamique de l'arbre ainsi que la génération dynamique d'informations utiles (coupures, bornes) nécessitent une synchronisation adéquate pour éviter de faire des traitements redondants. Ces problèmes entraînent généralement une dégradation des performances de l'algorithme.

2.3.2 Résolution Parallèle Heuristique

Le parallélisme est aussi présent dans le domaine des méthodes heuristiques qui sont utilisées souvent lors de la résolution de grandes instances. Ces méthodes donnent des solutions approchées dont la qualité permet de juger leur efficacité. Elles permettent de résoudre des problèmes d'optimisation issus du monde réel rapidement. Dans ce cadre, plusieurs auteurs ont proposé des méthodes pour décomposer le problème initial en sous-problèmes qui sont résolus par la suite en parallèle. Nous mentionnons ici que l'étape de décomposition se base sur divers critères que la pratique et l'expérience suggèrent et préconisent. Le savoir-faire acquis dans le domaine industriel aide à faire des choix de décompositions adéquats.

Topaloglu & Powell (2005) ont adopté un système de résolution multi-périodes des sous-problèmes pour résoudre un problème d'allocation dynamique de ressources. La décomposition est faite à base régionale. Au bout de chaque période, les informations d'impact sont échangées entre les sous-problèmes sous l'étiquette d'apprentissage. Ceci permet de corriger les décisions prises quant à l'allocation des ressources et par la suite sur la solution du problème.

Pour résoudre un problème d'affectation d'équipages, Abbink et al. (2007) ont procédé à des décompositions selon des paramètres différents comme la région et les journées de travail. Ils ont adopté l'idée d'une chaîne de décompositions où les résultats des décompositions précédentes sont pris en compte lors des décompositions subséquentes. Ils ont pour but d'améliorer la qualité de la solution atteinte en tenant compte de l'information acquise des décompositions précédentes.

Jütte & Thonemann (2012) ont proposé une décomposition par région du problème d'affectation des équipages pour un réseau de trains en sous-problèmes non indépendants. Ils ont permis un chevauchement entre les régions tout en ajoutant un facteur dans la fonction

objectif qui pénalise les stations qui ont été placées dans la mauvaise région. Leur méthode permet un ajustement de la décomposition au cours de la résolution. Les résultats qu'ils ont obtenus ont montré une amélioration du coût de la solution obtenue par leur méthode comparé à celui de la solution obtenue par la décomposition sans interaction entre les régions.

2.4 Génération de Colonnes

Pour un programme linéaire susceptible de posséder beaucoup de variables, la génération de colonnes est une méthode de résolution qui permet de surmonter cette difficulté. Elle résout un problème restreint à un ensemble limité de variables et génère des variables utiles au fur et à mesure que le processus de résolution avance jusqu'à obtenir une solution optimale de la relaxation continue (voir figure 2.1).

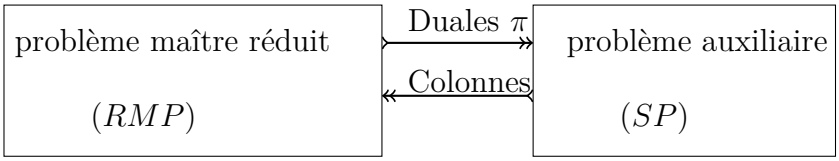


FIGURE 2.1 Méthode de génération de colonnes

La génération de colonnes repose sur la décomposition de Dantzig-Wolfe apparue en 1960 qui consiste à décomposer l'ensemble des contraintes en deux sous-ensembles. Le problème consistant à chercher la meilleure variable à ajouter au problème restreint est appelé sous-problème associé au problème maître. Il a comme objectif de trouver la variable de coût réduit minimum pour un problème de minimisation et donc la plus prometteuse pour améliorer la solution. Le coût réduit des variables est calculé par le biais des valeurs duales obtenues après la résolution du problème restreint. Le processus de génération-résolution continue jusqu'à ce que le sous-problème n'est plus capable de produire de variable (colonne) qui améliorerait la fonction objectif.

Cette technique est aussi reproductible dans le cas de programmation linéaire en nombres entiers et par la suite pour la résolution du SPP. Elle est largement utilisée pour l'optimisation discrète des problèmes industriels (voir Wilhelm 2001). Elle a fait l'objet de plusieurs travaux pertinents (voir Desrochers & Soumis (1989), Barnhart et al. (1998), Desaulniers et al. (1997), Gamache et al. (1999)). En général, elle est utilisée conjointement avec une méthode d'énumération implicite pour trouver une solution entière. Ce fait a produit des algorithmes dits de génération de colonnes et branchements "*branch and price*".

Dans cette thèse qui s'intéresse aux problèmes de tournées de véhicules et de rotations d'équi-

pages, le sous problème est un problème de plus court chemin avec contraintes de ressources dans un réseau espace-temps. Les chemins générés dans ce réseau correspondent aux colonnes de la matrice des contraintes dans la formulation du SPP. Nous avons intégré nos algorithmes parallèles avec la génération de colonnes pour étudier des alternatives aux algorithmes de branchement et ses variantes. Par ceci, nous voulons étendre l'utilisation d'ISUD et tirer profit des utilisations diverses de la génération de colonnes dans l'industrie.

CHAPITRE 3 ORGANISATION DE LA THÈSE

Cette thèse introduit des techniques de parallélisation pour les algorithmes basés sur la méthode de résolution ISUD. Le but est de tirer profit de l'évolution de l'informatique et par la suite d'améliorer les performances d'ISUD. Pour cela, nous utilisons l'information induite par la solution courante et des données du problème de partitionnement pour décomposer le problème initial en sous problèmes à traiter en parallèle. Dans ce qui suit, nous proposons une version à double décomposition ISU2D et une version distribuée DISUD basées sur la reformulation et la parallélisation du problème complémentaire. Ces versions permettent d'améliorer la qualité des solutions entières trouvées et de réduire le temps d'exécution. Ensuite, nous présentons une méthode qui étend le DISUD au contexte de la génération de colonnes. Les trois chapitres qui suivent présentent trois versions d'ISUD dont chacune traite un aspect particulier des améliorations de l'algorithme original d'ISUD.

Le simplexe intégral utilisant la double décomposition ISU2D. Au chapitre 4, nous proposons un simplexe intégral qui adopte une deuxième décomposition pour améliorer la solution courante. Il utilise une décomposition verticale disjointe innovante pour trouver en parallèle des directions de descente orthogonales conduisant à une solution optimale avec de grands pas. Chaque direction de descente est combinaison de directions minimales qui identifient un ensemble de variables entrantes avec de meilleurs coûts. Pour trouver ces directions, nous développons une approche de décomposition dynamique où l'inférence est basée sur les informations issues de la solution courante et des données du problème. L'idée de base est de décomposer les colonnes de la solution courante en clusters. Chaque cluster définit un sous problème complémentaire où une partie des colonnes de ce cluster va être remplacée par de meilleures colonnes, i.e, mettre ensemble des colonnes de la solution courante qui peuvent sortir ensemble. Notre principale innovation est l'utilisation de la solution courante pour la construction des sous-problèmes, ce qui la rend dynamique, car la décomposition change au cours du processus d'optimisation à mesure que la solution courante change. De plus, nous utilisons des stratégies d'accélération et utilisons des techniques de traitement parallèle. Les résultats des tests de différentes variantes d'ISU2D sur des instances des rotations du personnel de transport montrent le potentiel de notre algorithme notamment pour les grandes instances. En effet, ISU2D est 3 à 4 fois plus rapide que ISUD sur les grandes instances. Cet article est accepté sujet à des corrections dans le journal *Computers & Operations Research*.

Le simplexe intégral distribué pour le clustering. Au chapitre 5, nous examinons la généralisation de ISU2D vu que les résultats peuvent être encore meilleurs si nous pouvons utiliser plusieurs décompositions doubles simultanément. Nous proposons une version distri-

buée du simplex intégral, appelé DISUD, qui utilise le paradigme des agents multiples. Ainsi, chaque agent divise dynamiquement le problème complémentaire en sous-problèmes et les résout en parallèle. Le nouvel algorithme DISUD améliore à chaque itération la partition courante jusqu'à ce qu'une solution optimale (ou presque-optimale) soit atteinte. De plus, nous avons intégré dans cette version distribuée la version Zoom. Dans l'algorithme Zoom, le problème complémentaire s'occupe de trouver un voisinage potentiel dans lequel le problème réduit cherche à améliorer la solution courante. Nous proposons deux variantes de notre algorithme : Dans la première, les agents sont indépendants alors que dans la deuxième, les agents coopèrent et échangent de l'information. Les tests faits sur des instances d'optimisation des horaires du personnel navigant du transport aérien montrent que notre algorithme fonctionne mieux que DCPLEX, la version distribuée du solveur commercial de pointe CPLEX. Cet article est soumis pour publication dans le journal *Discrete Applied Mathematics*.

Le simplexe intégral distribué pour la génération de colonnes. Au chapitre 6, nous proposons d'intégrer la version DISUD au sein d'un environnement de génération de colonnes et ainsi de développer l'algorithme DICG. Pour ce faire, nous utilisons principalement le module de génération de colonne au sein des agents de DISUD. En outre, nous continuons d'améliorer l'état d'art d'ISUD et modifions par la suite le problème complémentaire pour prendre en considération la modification dynamique des coefficients de la contrainte de normalisation. Nous réalisons ainsi une plate-forme de base pour profiter de la méthode de génération de colonnes afin d'élargir le domaine des applications d'ISUD. Nous avons testé les variantes de notre algorithme sur des problèmes d'horaires de grande taille. Les résultats obtenus sont satisfaisants et encourageants. En effet, DICG fonctionne mieux que DRMH, la version distribuée de la méthode intitulée *Restricted Master Heuristic*. DICG est 2 à 5 fois plus rapide et fournit des solutions entières de bonne qualité (moins de 1%). Cet article est soumis pour publications dans le journal *RAIRO-Operations Research*

Au chapitre 7, une discussion générale étale les avantages et les limites de ce travail. Finalement, une conclusion générale est donnée au chapitre 8.

CHAPITRE 4 ARTICLE 1: INTEGRAL SIMPLEX USING DOUBLE DECOMPOSITION

Omar. Foutlane [a,b] Issmail. El Hallaoui [a,b] Pierre. Hansen [a,c]

Paper accepted in Computers and Operations Research subject to revisions

[a] *GERAD, Montréal (Québec), Canada, H3T 2A7*

[b] *Department of Mathematics and Industrial Engineering, Polytechnique Montréal (Québec) Canada, H3C 3A7*

[c] *Department of Decision Sciences, HEC Montréal, Montréal (Québec), Canada, H3T 2A7*

Abstract

The integral simplex using decomposition (ISUD) is a primal algorithm dedicated to solve set partitioning problems (SPP). Given an integer solution, the integral simplex using decomposition (ISUD) seeks a descent direction that leads to an improved adjacent integer solution. It uses a horizontal decomposition (of a linear transformation of the constraint matrix). We propose the integral simplex using double decomposition (ISU2D) which is parallel version of ISUD. It uses an innovative disjoint vertical decomposition to find in parallel orthogonal descent directions leading to an integer solution with a larger improvement. Each descent direction identifies a set of variables that will leave the current solution and a set of entering variables with better costs. To find these directions, we develop a dynamic decomposition approach that splits the original problem into subproblems that are then solved in parallel by ISUD. Our main innovation is the use of the current solution as a foundation for the construction of the set of subproblems ; the set changes during the optimization process as the current solution changes. In addition, we use bounding and pricing strategies and implement parallel processing techniques. We show that ISU2D is 3 to 4 times faster than ISUD on large instances.

Keywords : Set partitioning problems, integral simplex, parallel computing.

4.1 Introduction

Through this work, we are interested in solving large set partitioning problem (SPP) by a novel primal approach. The SPP is often used to model real-world combinatorial optimization problems including vehicle and crew scheduling. We use scheduling terminology to present the problem. A set partitioning constraint ensures that a *task* (for example, a flight leg or bus trip) is performed exactly once by a crew member (a pilot or bus driver). Let $T = \{1, 2, ..., m\}$ be the set of tasks and $J = \{1, 2, ..., n\}$ the set of feasible schedules. Here *feasible* means that the schedules satisfy all the safety and collective agreement rules limiting, for example, the maximum flying time during a working day and the maximum time away from the base. With each schedule j, we associate a variable x_j, a cost c_j, and a column $A_j = (a_{tj})_{t \in T}$ where a_{tj} is 1 if A_j covers task t and 0 otherwise. The matrix $A = [A_1, A_2, ..., A_n]$ is a binary matrix and assumed, without loss of generality, to be of full rank m. The SPP formulation is :

$$
\begin{aligned}
& & Minimize \quad & \sum_{j \in J} c_j x_j & (4.1)\\
(SPP) \quad & & subject\ to & & \\
& & & \sum_{j \in J} a_{tj} x_j = 1, \forall t \in T & (4.2)\\
& & & x_j \in \{0, 1\}, \forall j \in J & (4.3)
\end{aligned}
$$

The objective function (4.1) minimizes the total cost. The set partitioning constraints (4.2) ensure that each task is covered exactly once. Constraints set (4.3) imposes integrality on the x_j variables. The linear relaxation (LP) is obtained by replacing (4.3) by $x_j \geq 0, \forall j \in J$. The reduced cost of variable x_j, with respect to a dual vector dictated by the context, is denoted $\bar{c}_j$.

4.1.1 Literature review

The SPP is NP-hard (Garey & Johnson 1979). A partial list of its applications includes truck deliveries (Balinski & Quandt 1964), vehicle scheduling (Ribeiro & Soumis 1991), aircrew and bus driver scheduling (Desaulniers et al. 1994, Chu et al. 1997, Hoffman & Padberg 1993), and clustering and classification (Rao 1971). Many SPP algorithms have been developed. They can be classified into two main families *dual and primal methods*. Dual methods (called ***dual-fractional*** in Letchford & Lodi (2002)), including branch and cut (see, e.g., Hoffman & Padberg, 1993; Desaulniers et al., 1997), are efficient for small and medium problems but less efficient for large SPPs. They may take days to find good solutions for some aircrew scheduling problems. They do not take advantage of available primal information, as we

explain later.

Primal methods move from an integer solution to a better one. Many primal methods are based on the famous result (Balas & Padberg, 1975) that demonstrates the existence of a sequence of integer solutions with decreasing costs leading to an optimal one ; see Haus et al. (2001), Thompson (2002), Saxena (2003), and Rönnberg & Larsson (2009). Unfortunately, these algorithms suffer from degeneracy and are not efficient for large SPPs. Zaghrouti et al. (2014) develops the integral simplex using decomposition (ISUD), which is based on the improved primal simplex (IPS) decomposition introduced by El Hallaoui et al. (2011) to handle degeneracy. ISUD decomposes a linear transformation of the constraint matrix horizontally : The first group of constraints is handled in a reduced problem and the second group in the so-called complementary problem. ISUD handles degeneracy efficiently and is able to solve problems with up to 570000 columns and 1600 constraints.

Many authors have developed parallel algorithms based on the dual-fractional paradigm to take advantage of the availability of inexpensive parallel machines. Some of these explore the branching tree in parallel. For instance, Eso (1999) proposes a parallel branch-and-cut solver ; Klabjan et al. (2001) and Alefragis et al. (1999) presents parallel algorithms for crew scheduling problems ; and Linderoth et al. (2001) develops a parallel heuristic. Other parallel algorithms use domain decomposition techniques to split the original problem into subproblems. The subproblems are solved in parallel, and the partial solutions are merged to form a solution to the original problem. Topaloglu & Powell (2005) proposes a parallel heuristic with both time and space decomposition for a resource allocation problem. Abbink et al. (2007) studies various decompositions, such as geographical, weekday, and line-based decompositions, for a Netherlands railway crew scheduling problem. They implement a multi-stage method where at each stage they use a different decomposition. Jütte & Thonemann (2012) proposes a penalized-geographical decomposition for a railway crew scheduling problem. They decompose the problem into overlapping regions that are optimized in parallel. The objective function penalized misclassified railway stations to adjust the region boundaries during the optimization process.

All these methods are heuristic and based on prior knowledge the modeler has before dealing with the problem. They handle very large SPPs but do not guarantee optimality. In addition, they are generally static with no way to adjust the decomposition during the solution process. The exception is the penalized-geographical decomposition method (Jütte & Thonemann 2012), where boundaries change. Meanwhile the modeler must however identify the region boundaries at the beginning of the optimization process.

Each descent direction identifies a set of variables that will leave the current solution and a

set of entering variables with better costs.

4.1.2 Contributions and organization

To the best of our knowledge, there are currently no parallel integral simplex algorithms. We present the first parallel integral simplex algorithm called the *integral simplex using double decomposition* (ISU2D) that can solve large SPPs more efficiently. We list below the main contributions of this paper :

1. Unlike ISUD that finds one descent direction at a time using the traditional horizontal decomposition, ISU2D uses in addition an innovative disjoint vertical decomposition to find multiple orthogonal descent directions in parallel : we propose to split vertically, using a graph-based approach, the original complementary problem into smaller complementary subproblems. Each one targets a small subset of high potential columns that likely encompass a descent direction.
2. ISU2D also performs an incremental vertical decomposition to improve the integer solution obtained by the disjoint vertical decomposition. In the incremental decomposition, we identify a potential subset of variables using an LP dual solution and we increment this subset until the solution quality is satisfactory.
3. ISU2D uses generic decompositions (not problem specific) and thus minimizes the role of the modeler. It can theoretically be applied to a wide range of SPPs.
4. ISU2D also improves ISUD by measuring the solution quality : it calculates a lower bound in parallel (without penalizing the processing time). This permits to terminate the solution process when a specified quality is reached.
5. ISU2D reduces the computational time of ISUD by a factor of three on average.

The remainder of this paper is organized as follows. Section 2 presents the theoretical and computational aspects of ISUD. Section 3 discusses ISU2D, and Section 4 presents the computational results demonstrating the effectiveness of ISU2D. Section 5 provides concluding remarks and suggestions for future research.

4.2 Preliminaries

In this section, we describe the ISUD algorithm and explain its main components. We discuss its limitations and propose the enhancements that lead to ISU2D.

4.2.1 ISUD overview

Given an integer solution $\bar{x}$ to the SPP, let P be its support, i.e., the index set of its positive components. More formally, $P = supp(\bar{x}) = \{j \in J : \bar{x}_j > 1\}$, and $p = card(P)$. ISUD is a two-stage sequential algorithm that is specialized for the SPP. It is based on the concept of compatibility (El Hallaoui et al., 2011) :

Definition 4.2.1. *A subset S of J is said to be compatible with an integer solution $\bar{x}$, or simply compatible, if there exist two vectors $v \in \mathbb{R}_+^{|S|}$ and $\lambda \in \mathbb{R}^p$ such that $\sum_{j\in S} v_j A_j = \sum_{l\in P} \lambda_l A_l$. The columns/variables indexed by S and the combination $\sum_{j\in S} v_j A_j$ are also said to be compatible. S is said to be minimal if any strict subset of it is incompatible.*

At the first stage, ISUD seeks compatible columns to improve $\bar{x}$. This is done by solving a reduced problem (RP), as explained in Section 4.2.2.1. The second stage looks for a compatible combination of (incompatible) columns that improves the $\bar{x}$ obtained at the first stage. It solves a complementary problem (CP) to find an *integer* descent direction d_{cp}, i.e., leading to an **improved integer solution**. See Section 4.2.2 for the details of the decomposition. ISUD iterates between the two stages until it reaches an optimal solution. Algorithm 2 (Zaghrouti et al., 2014) outlines the procedure.

Algorithm 2 ISUD algorithm

Start with an initial integer solution x_0, set $\bar{x} = x_0$ and $k = 1$.
Stage RP : Improve the current solution $\bar{x}$ by solving RP.
Stage CP : Solve CP to get an *integer* descent direction d_{cp}.
Control : If $d_{cp} \neq 0$, i.e., CP improves the RP solution, then
set $\bar{x} = \bar{x} + d_{cp}$ and k = k+1. Go to Stage RP.
Else Stop : the integer solution $\bar{x}$ is **optimal**.
End if

4.2.2 ISUD decomposition

We use the notation of Zaghrouti et al. (2014). Let $K \subset J$ and $L \subset T$. Let v_K (v^L) be the subvector of a vector v with components indexed in K (L). Similarly, $A_K^L = (a_{lk})_{l\in L, k\in K}$ is the $|L| \times |K|$ submatrix of A with rows and columns indexed by L and K respectively. If $L = T$ or $K = J$, the superscripts (subscripts) are omitted ($A_J^T = A$ for instance). Finally, e is a vector of ones with dimension dictated by the context, and I_K^K is the identity matrix of dimension $|K| \times |K|$.

We can permute without loss of generality the columns of A in such a way that its first p columns are those indexed in P. Zaghrouti et al. (2014) associate with $\bar{x}$ a basis B where the

first p columns are those indexed in P and the remaining $|T| - p$ columns are artificial with a large cost. Let $N = T \setminus P$. We have

$$B = \begin{bmatrix} I_P^P & 0 \\ A_P^N & I_N^N \end{bmatrix} \text{ and } B^{-1} = \begin{bmatrix} I_P^P & 0 \\ -A_P^N & I_N^N \end{bmatrix}$$

because

$$\begin{bmatrix} I_P^P & 0 \\ -A_P^N & I_N^N \end{bmatrix} \begin{bmatrix} I_P^P & 0 \\ A_P^N & I_N^N \end{bmatrix} = \begin{bmatrix} I_P^P & 0 \\ 0 & I_N^N \end{bmatrix}.$$

When we multiply by B^{-1}, the constraint $Ax = e$ becomes

$$B^{-1}Ax = B^{-1}e \Leftrightarrow \bar{A}x = \bar{e} \Leftrightarrow \begin{bmatrix} \bar{A}^P \\ \bar{A}^N \end{bmatrix} \begin{bmatrix} x \end{bmatrix} = \begin{bmatrix} \bar{e}^P \\ \bar{e}^N \end{bmatrix},$$

where the j^{th} column $\bar{A}_j$ is

$$\begin{bmatrix} \bar{A}_j^P \\ \bar{A}_j^N \end{bmatrix} = \begin{bmatrix} A_j^P \\ -A_P^N A_j^P + A_j^N \end{bmatrix}, \quad \text{and} \quad \begin{bmatrix} \bar{e}^P \\ \bar{e}^N \end{bmatrix} = \begin{bmatrix} I_P^P & 0 \\ -A_P^N & I_N^N \end{bmatrix} \begin{bmatrix} e^P \\ e^N \end{bmatrix} = \begin{bmatrix} e^P \\ 0 \end{bmatrix}.$$

Actually, we have $\bar{e}^N = -A_P^N e^P + e^N = 0$ because $P = supp(\bar{x})$ and consequently each row of A_P^N contains only one "1 ", and "0"s elsewhere. So, $A_P^N e^P = e^N$ sums the values on each row. The sum is exactly equal to 1 on each of one of the $m - p$ rows, hence we obtain e^N.

El Hallaoui et al. (2011) show that a column A_j is compatible *iff* $\bar{A}_j^N = 0$. Let C and I be the index sets of the compatible and incompatible columns. Thus, the set of columns J is partitioned into C and I, and the set of constraints T is partitioned into P and N. Hence, we write $x = [x_C, x_I]$, $A = [A_C, A_I]$, and $c = [c_C, c_I]$. Using this partition, Zaghrouti et al. (2014) decompose the problem **horizontally** as explained in the following : "nondegenerate" constraints are handled in the reduced problem whereas the "degenerate" ones are handled in the complementary problem.

4.2.2.1 Reduced problem

The reduced problem RP is defined by imposing $x_I = 0$, i.e., including compatible columns only :

$$
\begin{aligned}
& & Minimize \quad & c_C \cdot x_C && (4.4)\\
(RP) \quad & & subject\ to \quad & \bar{A}^P_C x_C = e^P && (4.5)\\
& & & x_C \in \{0,1\}^{|C|} && (4.6)
\end{aligned}
$$

By definition, RP depends on $\bar{x}$. As we consider only compatible columns in RP, the rank of $\bar{A}^P_C$ is $p = card(P)$ because by definition, all compatible columns are linear combinations of the columns indexed by P..In addition, as $p < m$ (normally in practice), surely, we will have redundant constraints in RP that should be removed. Zaghrouti et al. (2014) show that a pivot on any compatible column with a negative reduced cost, of course in the linear relaxation of RP, leads to an improved integer solution. Let x^*_C be an optimal solution to RP. Note that $\bar{x} = (x^*_C, 0)$ is a solution to the SPP.

4.2.2.2 Complementary problem

Let $\bar{x}$ be the integer solution obtained by solving the RP. ISUD solves a complementary problem CP to find a set of incompatible columns that improve $\bar{x}$. If we pivot on the columns in this set in any order we obtain an improved integer solution. More precisely, we look for a set such that a (convex) combination of its columns is compatible and has a negative reduced cost. In other words, CP searches for a descent direction d_{cp} leading to an improved integer solution. Zaghrouti et al. (2014) formulate CP as follows :

$$
\begin{aligned}
& & Minimize \quad & \bar{c}_I \cdot x_I && (4.7)\\
(CP) \quad & & subject\ to \quad & \bar{A}^N_I x_I = 0 && (4.8)\\
& & & e \cdot x_I = 1 && (4.9)\\
& & & x_I \geq 0 && (4.10)
\end{aligned}
$$

where $\bar{A}^N_I = -A^N_P A^P_I + A^N_I$, and $\bar{c}_I = c_I - A^P_I c_P$. Zaghrouti et al. (2014) show that if CP is infeasible or $z^{CP} \geq 0$, i.e., the objective value of CP is non-negative, then $\bar{x}$ is an optimal solution to the SPP. Otherwise, CP finds a descent direction, i.e., $z^{CP} \geq 0$. Let $S = supp(x_I)$. If the columns A_j, $j \in S$ are pairwise row-disjoint, i.e., they do not cover the same constraints, we obtain an integer descent direction, i.e., leading to an improving integer

solution say x^*. More, S is shown to be minimal by El Hallaoui et al. (2011), equivalently said non-decomposable using the terminology of Balas & Padberg (1975), meaning that $\bar{x}$ and x^* are adjacent. Thus, Zaghrouti et al. (2014) show that S defines a descent direction $d_{cp} = x^* - \bar{x}$ where x^* can be obtained as follows :

$$x_j^* = \begin{cases} 1, & j \in S \cup (P \setminus S^-) \\ 0, & \text{otherwise} \end{cases}$$

where S^- is simply the index set of the leaving variables that cover the same tasks as the entering variables in S. Based on this, Zaghrouti et al. (2014) propose a branching technique to eliminate the non-disjoint solutions when solving CP. They use a deep search strategy to get a descent direction.

4.2.3 ISUD limitations

ISUD has four main limitations :

— The computational time increases quickly with problem size : when the number of constraints increases by a factor of 2, the solution time of ISUD (single-thread, i.e., sequential) increases by a factor of 250 or more (see Table 4.9).

— Since we move from an integer solution to an adjacent one at each iteration k, ISUD finds one descent direction at a time. It must solve many complementary problems to reach the final solution.

— CP often produces a small set of disjoint columns, i.e., $|S|$ is small, at each iteration k. Consequently, the computing time of ISUD increases.

— ISUD guarantees optimality, if full branching (i.e. including backtracking) is carried out during CP. It becomes heuristic for practical purposes by restricting branching. As it lacks a measure of assessing solution quality, because it does not calculate a lower bound, the stopping criterion in ISUD is to improve.

4.3 ISU2D approach

We now explain in more details how our double decomposition enables us to solve the SPP more efficiently and push back the limitations of ISUD. In Section 4.3.1 we introduce ISU2D and discuss its convergence. In addition to the horizontal decomposition of ISUD, we decompose the problem vertically to find orthogonal descent directions. We use both a disjoint vertical decomposition (DVD ; see Section 4.3.2) and an incremental vertical decomposition (IVD ; see Section 4.3.3). To ease the understanding of ISU2D, we consider the SPP problem

presented by Table 4.1. It consists of 10 columns and 8 tasks where the current solution non null variables are x_1, x_2, x_3 and x_4 (A_1, A_2, A_3 and A_4).

TABLE 4.1 Example with 8 tasks and 10 columns

	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	A_{10}
c_j	3	2	2	1	2	1	1	2	1	1
T_i/x_j	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**0**	**0**	**0**
1	1	0	0	0	0	0	0	0	1	0
2	1	0	0	0	0	0	0	0	1	0
3	0	1	0	0	1	0	0	0	1	0
4	0	1	0	0	0	1	0	0	0	0
5	1	0	0	0	0	1	0	0	0	0
6	0	0	0	1	0	0	1	0	0	0
7	0	0	1	0	1	0	1	0	0	0
8	0	0	1	0	0	0	0	1	0	1

4.3.1 General structure of ISU2D

Algorithm 3 presents ISU2D ; the IN-PARALLEL and END-PARALLEL terms are used to delimit multiple statements that are executed in parallel, z_{lb} and z_{ub} are the lower and upper bounds on the optimal (integer) value of SPP, and ϵ is a prespecified threshold. When the absolute gap of the ISU2D solution is less than ϵ, ISU2D stops the solution process. A brief description of ISU2D is given below ; the details are in the following subsections.

Algorithm 3 ISU2D pseudo code

Start with an integer solution $\bar{x} = x_0$, $z_{lb} = -\infty$, $z_{ub} = z_0 = c \cdot x_0$.
IN-PARALLEL
    Improve z_{lb} continuously and share it with the other threads.
    Improve $\bar{x}$ using DVD and update z_{ub} accordingly until either $z_{ub} - z_{lb} \leq \epsilon$ or no improvement is possible.
END-PARALLEL
If $z_{ub} - z_{lb} > \epsilon$, improve $\bar{x}$ using IVD.
Return $x^* = \bar{x}$.

ISU2D has two main phases. In the DVD phase, it improves the current solution $\bar{x}$ using DVD. Given parameter q, DVD builds q subproblems SP_k for $k \in \{1 \dots q\}$ by partitioning P and thus T into q clusters ; it solves them in parallel. We calculate a lower bound (z_{lb}) simultaneously.

In the IVD phase, ISU2D sequentially solves a set of q' suproblems SP_k to optimality or near-optimality. These subproblems are based on the dual information obtained from the computation of the lower bound. In both phases we use ISUD to solve the subproblems, which are significantly smaller than the original problem. ISU2D terminates when the solution quality is satisfactory.

4.3.2 DVD phase

4.3.2.1 Theoretical motivation

From any integer solution to the SPP, we need at most m orthogonal descent directions (see Proposition 4.3.1), which we can obtain in parallel, to reach an optimal solution. This result motivated us to decompose the problem further to improve the performance of ISUD.

Proposition 4.3.1. *From any integer solution $\bar{x}$ to the SPP, we can reach an optimal solution x^* via at most m minimal orthogonal descent directions.*

Démonstration. Let $\bar{x}$ be the current solution and x^* an optimal solution. Let $D^* = \{j \in J : x^*_j - \bar{x}_j \neq 0\}$, $D^*_1 = \{j \in J : x^*_j = 1 \text{ and } \bar{x}_j = 0\}$, i.e., the index set of variables that enter the basis (optimal solution), and $D^*_0 = \{j \in J : x^*_j = 0 \text{ and } \bar{x}_j = 1\}$ be the index set of the variables that will leave the basis (current solution). We have $D^* = D^*_1 \cup D^*_0$. Obviously, the tasks covered by the entering columns are the same as those covered by the leaving columns. Therefore,

$$\sum_{j \in D^*_1} A_j = \sum_{l \in D^*_0} A_l$$

Let us now define the set sequences D^k_0, D^k_1, H^k_0, and H^k_1 as follows :

$$D^0_1 = D^*_1, D^0_0 = D^*_0, H^0_1 = \emptyset, H^0_0 = \emptyset$$

$$D^k_1 = D^{k-1}_1 \setminus H^k_1, D^k_0 = D^{k-1}_0 \setminus H^k_0, \quad k \geq 1$$

where $H^k_1 \subset D^{k-1}_1$ is the smallest nonempty index subset of columns that could form a minimal compatible combination, and $H^k_0 \subset D^{k-1}_0$ is the index subset of columns covering the same tasks as those of H^k_1. Thus, we have

$$\sum_{j \in H^k_1} A_j = \sum_{l \in H^k_0} A_l$$

We define $k_{opt} = |\{k : H^k_1 \neq \emptyset\}|$, i.e., the number of compatible combinations formed from columns indexed by D^*_1. Clearly, $k_{opt} \leq |D^*_1|$. We have $|D^*_1| \leq |\{j : x^*_j = 1\}| \leq m$, so $k_{opt} \leq m$.

We define the sequence of descent directions $d^k \in \mathbb{R}^n$ as $d^k_j = 1$ if $j \in H^k_1$, $d^k_j = -1$ if $j \in H^k_0$, and $d^k_j = 0$ otherwise. By construction, the d^k are orthogonal ($d^l \cdot d^h = 0$ for every $h \neq l$) and $x^* = \bar{x} + \sum_{k=1}^{k_{opt}} d^k$.

Finally, it is easy to see that the sequence x^k defined by $x^0 = \bar{x}$, $x^k = x^{k-1} + d^k$ for $k \geq 1$ has nonincreasing cost. Suppose there exists k_1 such that $c \cdot x^{k_1+1} > c \cdot x^{k_1}$. Then

$$c \cdot x^{k_1+1} - c \cdot x^{k_1} = c \cdot d^{k_1} > 0$$

$$c \cdot (\bar{x} + \sum_{k=1}^{k_{opt}} d^k) > c \cdot (\bar{x} + \sum_{k=1}^{k_1-1} d^k + \sum_{k=k_1+1}^{k_{opt}} d^k)$$

$$c \cdot x^* > c \cdot (\bar{x} + \sum_{k=1}^{k_1-1} d^k + \sum_{k_1+1}^{k_{opt}} d^k),$$

which contradicts the fact that x^* is optimal because $\bar{x} + \sum_{k=1}^{k_1-1} d^k + \sum_{k_1+1}^{k_{opt}} d^k$ is a feasible solution. This completes the proof. □

We note that the sequence $c \cdot x^k$ is (strictly) decreasing because $c \cdot d^k < 0$ when H^k_1 is obtained by solving the CP. Although x^* is not known at the beginning of the optimization process, we develop a method to build good approximations of the subproblems to get the sequence sets H^k_1 and then d^k in parallel ; see Figure 4.1.

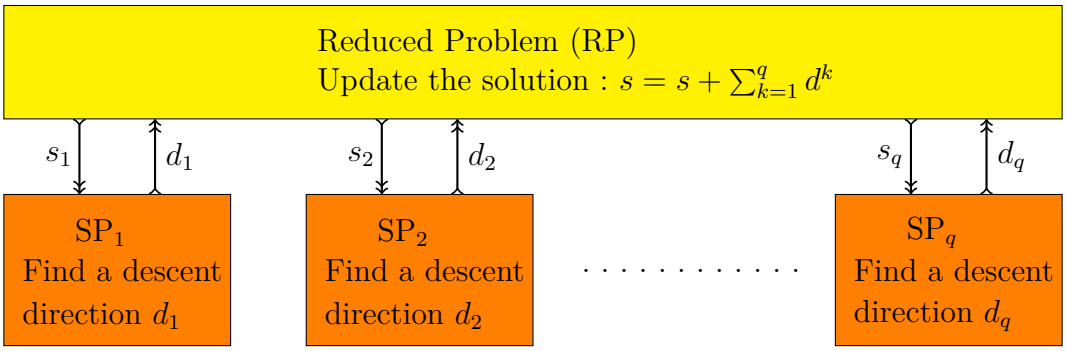


FIGURE 4.1 Decomposition and parallel descent directions.

To reduce communication between processes, a source of the overhead, we seek combined descent directions (see Corollary 4.3.2) that are actually combinations of minimal descent directions ; each subproblem finds at most one. To find these directions, ISU2D decomposes the original problem into subproblems using the splitting technique discussed in Section 4.3.2.2. The proof of the corollary below is omitted because it is straightforward : the result follows from orthogonality.

To ensure that the benefit of parallelization outweighs the overhead due mainly to communication between processes, one has to find the right number of processes to run in parallel by tuning well the granularity. We think that minimal descent directions have finer granularity and seeking them at each iteration may cause unnecessary overhead due to an increased communication effort. We propose reducing the overhead by seeking descent directions (see Corollary 4.3.2) that are combinations of minimal descent directions. The corollary just below outlines this fact.

Corollary 4.3.2. *Any combination of minimal orthogonal descent directions is a descent direction with cumulated improvements.*

Démonstration. The proof is straightforward. It results from orthogonality.

□

Remark 4.3.3. *Combined descent directions can be found by solving the complementary subproblem a certain number of times, combining the descent directions it finds, and finally return the combined one to the reduced problem. To find these directions in an equivalent and a simple manner, ISU2D decomposes the original SPP problem into smaller SPP subproblems using the splitting technique discussed in Section 4.3.2.2. Each SPP subproblem provides a descent direction that is exactly the final solution we find by solving it by ISUD minus the current solution.*

4.3.2.2 Splitting technique

The most novel feature of ISU2D is the use of the current solution to construct the subproblems $(SP_k)_{1\leq k\leq q}$. We partition P into q clusters where the columns indexed by cluster k cover a set of tasks T_k, i.e., $T = \cup(T_k)_{1\leq k\leq q}$. Let $J_k \subset J$ be the subset of columns that cover only tasks in T_k. We formulate subproblem (SP_k) as follows :

$$
\begin{aligned}
& & min \quad & \sum_{j\in J_k} c_j x_j & & \\
(SP_k) & & & \sum_{j\in J_k} a_{tj} x_j = 1 & \forall t \in T_k & \\
& & & x_j \in \{0,1\} & \forall j \in J_k &
\end{aligned}
$$

Hence, building the subproblems reduces to defining the task partition $\tau = (T_k)_{1\leq k\leq q}$. We define a graph $G(V, E)$ where each positive-valued variable in the current solution is repre-

sented by a vertex v of V. For simplicity, V is the index set of these variables. Figure 4.2 shows the graph of the current solution of our SPP example.

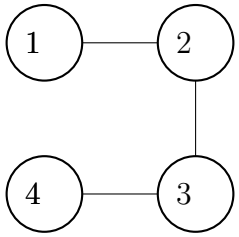


FIGURE 4.2 Example of a graph $G(V, E)$.

Let H_0^k and H_1^k be defined as above for $k \in 1..k_{opt}$. Note that $H_0^k \subseteq V$.

Proposition 4.3.4. *The subgraph induced by H_0^k, denoted $G(H_0^k)$, is a connected component of G.*

Démonstration. Suppose that $G(H_0^k)$ is not a connected component of G. Note that H_1^k is minimal by construction. Let $H_0' \subsetneq H_0^k$ be such that $G(H_0')$ is the smallest connected component of $G(H_0^k)$. Then there exists $H_1' \subset H_1^k$ such that

$$\sum_{j \in H_1'} A_j = \sum_{l \in H_0'} A_l$$

Therefore, H_1' is a compatible combination by Definition 6.2.1. This contradicts the fact that H_1^k is minimal since H_1' is a strict subset of H_1^k. □

Let $(v, v') \in V \times V$, $J_{vv'} = \{l \in J \ : \ A_v \cdot A_l \neq 0 \text{ and } A_{v'} \cdot A_l \neq 0\}$. We define E as the set $\{(v, v') : J_{vv'} \neq \emptyset\}$, i.e., v and v' cover some common task. We assign a weight $w_{vv'}$ to every edge $(v, v') \in E$ (see Section 4.3.2.3). Then, we partition G into q equally sized subgraphs $G_k = (E_k, V_k), 1 \leq k \leq q$ in such a way that the connected components characterized by Proposition 4.3.4 are likely to be (fully) included into these subgraphs. Indeed, we partition the graph so that the total weight of the cut (i.e., the edges having their ends in different subgraphs) is minimized. Figure 4.3 illustrates the splitting technique on our SPP example. Figure 4.3(a) shows the graph $G(V, E)$. The cut shown as a dashed line on Figure 4.3(b) gives two subproblems : (SP_1) with $T_1 = \{1, 2, 3, 4, 5\}$ and (SP_2) with $T_2 = \{6, 7, 8\}$. The index set of conflicting variables is $\{5\}$.

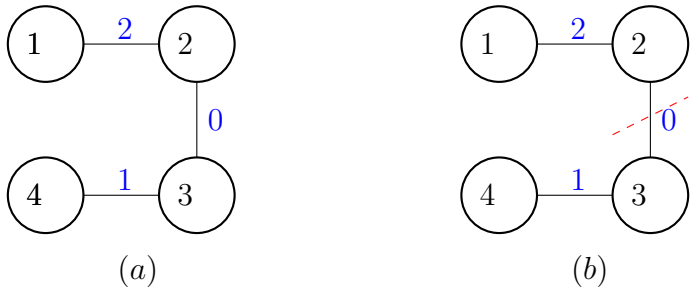


FIGURE 4.3 Example of ISU2D splitting.

The variables indexed by V_k cover a set of tasks T_k. Thus, we obtain the decomposition $\tau = \{T_k, 1 \leq k \leq q\}$, and the subproblems $(\text{SP}_k)_{1 \leq k \leq q}$ are built accordingly. Conflicting variables, i.e., those covering tasks in two different task clusters, are set to 0. The set of conflicting variables changes from iteration to iteration. The splitting is done in such a way that some potential conflicting variables, i.e, variables that could be part of a descent direction, become nonconflicting in the next iteration.

Proposition 4.3.5. *If the current solution $\bar{x}$ is not optimal, at least one conflicting variable will be positive in any improving solution.*

Démonstration. Let $\bar{x}^k$ be an optimal solution to SP_k. Then $(\bar{x}_{J_1}, ..., \bar{x}_{J_q})$ is an optimal solution to the SPP restricted to the variables indexed by $\cup_{k \in 1..q} J_k$, because the matrix of this restricted problem called SPP_R is block-angular. Suppose there exists an improving solution $\bar{x}'$ where all the conflicting variables are 0. Then $(\bar{x}'_{J_1}, ..., \bar{x}'_{J_q})$ is an improving solution to SPP_R, which contradicts the fact that $(\bar{x}_{J_1}, ..., \bar{x}_{J_q})$ is an optimal solution to SPP_R. □

The weight of the edge $(v, v') \in E$ measures the likelihood that the variables indexed by $J_{vv'}$ will improve the objective value. When the edge $(v,\ v')$ is not cut (e.g., edge $(1, 2)$ in Figure 4.3), A_v and $A_{v'}$ are grouped into a cluster and will be considered in the same subproblem. Thus, the variables indexed by v and v' could be part of a descent direction (as leaving variables). When the edge (v, v') is cut (e.g., edge (2,3) in Figure 4.3), it is not possible in the current iteration to improve the objective value with the variables indexed by $J_{vv'}$. Thus, a good splitting technique should avoid cutting edges (v, v') where at least one variable indexed in $J_{vv'}$ is a part of an optimal solution. Splitting depends heavily on the edge weights and consequently on the formulas used to calculate them, which we call weighting methods.

4.3.2.3 Weighting methods

Good weighting methods use the problem structure to decide which columns to group and which to separate. For this proof of concept, we tested some generic weighting methods and retained two promising ones :

$$w_1 : (v, v') \mapsto w_{vv'} = |\{j \in J_{vv'} \ : \ \bar{c}_j \leq 0\}|$$

$$w_2 : (v, v') \mapsto w_{vv'} = -\min(0, \min\{\bar{c}_j : \ j \in J_{vv'}\}).$$

We use reduced costs with respect to a dual vector α derived from the current solution $\bar{x}$ such that $\bar{c}_j = c_j - \alpha \cdot A_j = 0, \forall j \in supp(\bar{x})$, i.e., the reduced costs of these basic variables are null. An infinite number of vectors α satisfy this equation. A simple one is

$$\alpha_t = \sum_{j \in J} \frac{\bar{x}_j * c_j * a_{tj}}{n_j}, \ \ t \in T \tag{4.11}$$

where n_j is the number of tasks covered by A_j, $j \in P = supp(\bar{x})$. Equation 4.11 means that we associate with a task t a dual value that is the average cost per task. In the example above, $\alpha_t = 1$, $\forall t \in \{1 \dots 8\}$ and the reduced costs are given in Table 4.2. A more sophisticated option is $\alpha = (\alpha^{T_1}, \alpha^{T_2}, \dots, \alpha^{T_q})$ where α^k is the solution of the dual of the CP (4.7)–(4.10) when solving SP_k by ISUD.

TABLE 4.2 Example with 8 tasks and 10 columns : reduced costs

	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	A_{10}
$\bar{c}_j$	0	0	0	0	0	-1	-1	1	-2	0

The first weighting method w_1 stipulates that a descent direction is more likely to exist in regions where there are more variables with negative reduced costs. We therefore cut the edges with the smallest number of negative reduced costs. Based on this, w_1 associates with each edge (v, v') the number of negative reduced cost columns that $J_{vv'}$ contains. The second weighting method increases the chances of getting a large step (improvement in the objective value) provided that a descent direction exists. We assume that one of the entering variables has the smallest negative reduced cost and hope to realize a large improvement in the objective value. Consequently, w_2 associates with the edge (v, v') the absolute value of the smallest negative reduced cost column indexed by $J_{vv'}$. The weights computed with this second method are illustrated in Figure 4.3. We hence obtain two variants of ISU2D depending on the weighting method that we use. The decomposition is adjusted dynamically.

4.3.2.4 DVD algorithm

Algorithm 4 outlines the DVD procedure.

Algorithm 4 DVD pseudo code

$r = 1$.
Repeat
  Build τ^r and consequently SP_k, $k \in \{1 \dots q\}$ by splitting as in Section 4.3.2.2.
  For each $k \in 1, ..., q$, do
    IN-PARALLEL
     Solve SP_k using ISUD (each SP_k returns d_k).
    END-PARALLEL
  $\bar{x} = \bar{x} + \sum_{k=1}^{q} d^k$.
  $r = r + 1$.
  If $c \cdot \sum_{k=1}^{q} d^k = 0$ (i.e. no improvement), decrease q.
Until (q = 1)

Algorithm 4 is monotonic because $c \cdot (\sum_{k=1}^{q} d^k) \leq 0$. Actually, d^k is itself the sum of the descent directions (d_{cp}) obtained in the ISUD iterations and consequently $c \cdot d^k \leq 0, \forall k \in \{1 \dots q\}$. At each iteration, Algorithm 4 explores (see Proposition 4.3.6) a different neighborhood around the current solution $\bar{x}$ by using the relative dual information. Let τ^r and $\tau^{r'}$ be two task partitions that define the subproblems at iterations r and r' respectively (as explained in Section 4.3.2.2) .

Proposition 4.3.6. *We have* $\tau^r \neq \tau^{r+1}$.

Démonstration. If $r' = r + 1$, there are two cases : i) The solution is not improved : in this case, we decrease q. Consequently, $\tau^r \neq \tau^{r'}$ because $|\tau^r| \neq |\tau^{r'}|$. ii) The solution is improved : q remains the same but in the splitting at least one variable x_{j_0} previously conflicting with τ^r becomes nonconflicting with $\tau^{r'}$. Let $\tau^r = \{T_k, 1 \leq k \leq q\}$ and $\tau^{r'} = \{T'_k, 1 \leq k \leq q\}$, and let T_0 be the set of tasks that are covered by x_{j_0}. Then there exists $k \neq k'$ such that $T_0 \cap T_k \neq \emptyset$ and $T_0 \cap T_{k'} \neq \emptyset$. At the same time, there exists k'' such that $T_0 \subset T'_{k''}$. Consequently, $T'_{k''}$ intersects both T_k and $T_{k'}$ but is different from them. Clearly, $\tau^r \neq \tau^{r'}$. □

Corollary 4.3.7. *We have* $\tau^r \neq \tau^{r'}, \forall r \neq r'$.

Démonstration. Suppose without loss of generality that $r \leq r'$. We consider the case $r' - r \geq 2$; the other case is discussed in Proposition 4.3.6. Suppose that $\tau^r = \tau^{r'}$. Then the solution

did not change between iterations r and r' because the decomposition is the same and we assume that the subproblems are solved to optimality (in parallel). Thus, the solution is not improved between r and $r+1$. We have a contradiction because in this case $\tau^r \neq \tau^{r'}$, as shown by Proposition 4.3.6. □

4.3.3 IVD phase

IVD is an improvement strategy that explores near-optimal LP neighborhoods. The idea is to seek an optimal or near-optimal solution by solving q' subproblems based on the variables' reduced costs with respect to the duals for an improved lower bound. IVD starts from the solution of the DVD phase and so has good upper and lower bounds (and the related dual information). IVD uses the well-known fixation technique to reduce the problem size. Here, $\bar{c}_j$ is the dual reduced cost of variable x_j computed with the LP dual values μ (so $\bar{c}_j \geq 0$ for all variables), $z_{ub} = c \cdot \bar{x}$ is the upper bound, and z_{lb} is the lower bound. If $\bar{z}_{lb} + \bar{c}_j > z_{ub}$, we remove column A_j from the constraint matrix (fixing its variable x_j to 0). This is a well-known theoretical result. This decomposition becomes very useful as we approach optimality because the gap between the upper and lower bounds is small, so we apply this to complete the solution process. Algorithm 5 outlines the procedure; q' is a parameter tuned by experimentation.

Algorithm 5 IVD pseudo code

Price columns using μ (i.e., compute their reduced costs).
Sort the variables in increasing order of reduced cost and reindex them.
For $k = 1$ to q'
    For all j, if $\bar{z}_{lb} + \bar{c}_j > z_{ub}$, $J = J \setminus \{j\}$, i.e., $x_j = 0$.
    Build SP_k by considering the first $k\frac{|J|}{q'}$ variables.
    Solve SP_k with ISUD.
    Set $\bar{x} = \bar{x} + d^k$, Update $z_{ub} = c \cdot \bar{x}$.
    If $z_{ub} - z_{lb} \leq \epsilon$, terminate ISU2D.
End for.

Algorithm 5 is also monotonic because we locally improve a solution at each iteration. It is finite because the problem is bounded.

4.4 ISU2D convergence

ISU2D converges mainly because Algorithms 4 and 5 converge. We state this more formally in the proposition below.

Proposition 4.4.1. *ISU2D is a monotonic exact algorithm that converges in a finite time.*

The proof is simple. The time of the DVD phase is at most equal to the time of computing the lower bound. We improve this bound in a finite time by solving the LP (in polynomial time if an interior point method is used) and adding a finite number of facets. The IVD phase is an improvement on ISUD, which converges in a finite time (Zaghrouti et al., 2014).

4.5 Computational results

In this section, we compare single- and multi-thread versions of the standard methods, namely CPLEX (version 12.6.1 set to its default parameters) and ISUD to ISU2D with three different splitting methods : ISU2D$_1$ and ISU2D$_2$ use w_1 and w_2 respectively (see Section 4.3.2.3), and ISU2D$_3$ applies multistage splitting : w_2 followed by w_1. Our results show that ISU2D is faster and gives a better solution than the other methods. In order to highlight the contribution of the DVD phase, we show the results of two algorithms that implement only the IVD phase : the first, IVD$_1$, uses ISUD as an IP solver and the second, IVD$_2$, uses CPLEX as an IP solver.

We implemented ISU2D using C^{++} and the MPI (Message Passing Interface) library. The library provides parallelization and communication between processes. We use the bipartition algorithm (Kernighan & Lin, 1972) to partition the weighted graph G. The tests were performed on a Unix Dell Precision T1700 with a 3.30 GHz Intel Xeon E3-1226 V3 Quad-Core processor. We used the values $q = 2$ and $q' = 2$ as ISU2D parameters for DVD and IVD phases algorithms respectively. The computational times are in seconds.
During the realization of this work, other sequential versions of ISUD appeared, see (Rosat et al. 2016, 2017a, Zaghrouti et al. 2013). Our contributions concern the introduction of a new decomposition that helps parallelizing ISUD. Therefore, our improvements are orthogonal, i.e., the decomposition could be applied to any sequential version of ISUD. Thus, this proof of concept implements the most commonly used version of ISUD. We intend to use more recent versions of ISUD in future work.

4.5.1 Instances

We test ISU2D on the aircrew and bus driver scheduling instances used by Zaghrouti et al. (2014). Note that the time for generating the initial solution is not part of the computing time reported in this section. Each instance requires reoptimization after a perturbation because of unforeseen events. The reoptimized schedules usually have many components in common with the original schedules. There are generally penalties in the objective function to discourage changes, so many schedules are unchanged in the reoptimized solution. For a

given instance, we define the perturbation ratio ρ to be the percentage of columns of the reoptimized solution that are not present in the original solution. This is a good indicator of the difficulty of an instance : the larger ρ, the harder the instance.

We grouped the instances into three sets (small, medium, and large) and then into three subsets (easy, moderate, and hard) corresponding to $\rho = 50\%$, 65%, and 80% respectively. Each subset contains 10 instances. The aircrew scheduling instances are small, and we use the prefix AS. The bus driver scheduling instances are medium and large, with the prefixes BM and BL. We add ρ to the instance name to indicate the level of difficulty. We also add an incremental value to the name, e.g., BM80 indicates the set of hard medium bus driver scheduling instances, and BM80-2 indicates instance number 2 of the same set.

Table 4.3 lists the characteristics of the instances : the average number of rows and columns and the average density (number of nonzero elements per column). The bus driver instances have an average of 40 nonzero elements per column, which is large. This makes the problem difficult for a traditional method such as CPLEX because of the severe degeneracy and branching difficulties. The aircrew problems have low density (9 flights) and are easier to solve.

TABLE 4.3 Instance characteristics

Set	#Rows	#Columns	Density
Small aircrew scheduling (AS)	***803***	***8904***	***9***
Medium bus driver scheduling (BM)	***1200***	***130000***	***40***
Large bus driver scheduling (BL)	***1600***	***570000***	***40***

4.5.2 Testing methodology

We present aggregated results for the aircrew instances (Section 4.5.3) and then the bus driver instances (Section 4.5.4). The detailed results are in the Appendix. For each class of instances, we compare the three variants of ISU2D, and then we compare the best variant to CPLEX (multi-thread version), ISUD (single- and multi-thread versions), IVD_1 and IVD_2.

For each variant of ISU2D, we report information for each phase. For DVD, we report the number of iterations (#Itr), the time, and the improvement percentage (%Imp), i.e., $Imp = 100 * \frac{z_0 - z_f}{z_0 - z^*}$ where z_f is the final objective value obtained by DVD. For IVD, we report the percentage of columns (%Cols) and the time to obtain the optimal value. Note that the averages are computed by considering only the instances that ISUD solved to optimality.

We use the following ratios to compare ISU2D to CPLEX and ISUD :

— The time reduction ratio $T_a(b)$ between two algorithms a and b is defined as $T_a(b) = \frac{t(a)}{t(b)}$ where $t(a)$ and $t(b)$ are the computational times of a and b. When $T_a(b) > 1$ the

algorithm b is faster than algorithm a. We use C to indicate CPLEX, I_s for the sequential (i.e., single-thread) ISUD, I_p for the parallel (i.e., uses multi-threads CPLEX to solve complementary problems) ISUD, IV_1 for the IVD_1 and IV_2 for the IVD_2. For example, T_C is the time reduction factor of the current algorithm compared to CPLEX. Here, $a \in \{C, I_s, I_p\}$ and $b \in \{ISU2D_1, ISU2D_2, ISU2D_3\}$. For simplicity, we omit b because it can be deduced from the context (table).

— The gap between the (final) solution value $z(a)$ returned by algorithm a and the optimal value z^* is defined as $Gap(a) = 100 * \frac{z(a)-z^*}{z^*}$.

4.5.3 Aircrew scheduling results

Table 4.4 summarizes the results of the ISU2D variants on the aircrew instances ; the detailed results are in Table A.1. We observe that DVD has better performance in $ISU2D_2$ than in $ISU2D_1$: %Imp is significantly higher in $ISU2D_2$ (for a similar number of iterations and execution time). The number of instances solved to optimality (#Opt) by DVD is 17 out of 30 in $ISU2D_2$ and 11 out of 30 in $ISU2D_1$. The success rate is increased by 54%. This may be because w_2 minimizes the reduced cost value (the improvement is proportional to this value) whereas w_1 maximizes the number of negative reduced cost columns regardless of the reduced cost value.

For DVD in $ISU2D_3$ the success rate is increased to 22 out of 30. The percentage of columns needed in IVD to obtain an optimal solution is significantly lower in $ISU2D_3$. The success rate is 10 out of 10 in AS50 and decreases as the difficulty increases.

TABLE 4.4 Summary of ISU2D results for small aircrew scheduling instances

	$ISU2D_1$						$ISU2D_2$						$ISU2D_3$					
	DVD				IVD		DVD				IVD		DVD				IVD	
Set	#Itr	Time	%Imp	#Opt	Time	%Cols	#Itr	Time	%Imp	#Opt	Time	%Cols	#Itr	Time	%Imp	#Opt	Time	%Cols
AS80	4	2.4	72	2	4.0	77	5	2.1	68	2	4.9	63	6	2.5	80	4	2.6	47
AS65	5	2.5	93	5	2.4	47	5	2.2	94	7	1.2	20	5	2.3	97	8	0.5	13
AS50	4	2.1	94	4	2.1	57	5	2.3	99	8	0.1	20	5	2.4	100	10	-	-

Figure 4.4 shows the evolution of the objective value over time for CPLEX, ISUD, and the ISU2D variants on instance AS80-1. Indeed, Figure 4.4-a presents the entire solution process evolution while Figure 4.4-b focuses on the algorithms behavior at the end of the solution process. The algorithms start from the same initial solution with an objective value equal to 376243. This behavior is typical and representative of the other instances. We connect the points in this figure to improve its readability. The ISU2D curves decrease more sharply than the ISUD curves because ISU2D finds multiple descent directions in parallel. The ISUD

curves are almost constant at the end of the solution process because ISUD terminates when its branching does , i.e, when the last branching node is infeasible. ISU2D instead uses the LP value to terminate as soon as the objective value is satisfactory. ISU2D$_3$ improves the ISUD solution time by 40% on this instance. The CPLEX curve initially decreases rapidly, but the rate slows as it approaches an optimal solution. The ISU2D variants reach an optimal solution more quickly. The three variants share similar behavior. With respect to this instance results, we can globally rank the algorithms from best to worst by performance on this instance as follows : ISU2D$_3$ » ISU2D$_2$ » ISU2D$_1$ » Multi-thread ISUD » Single-thread ISUD » CPLEX.

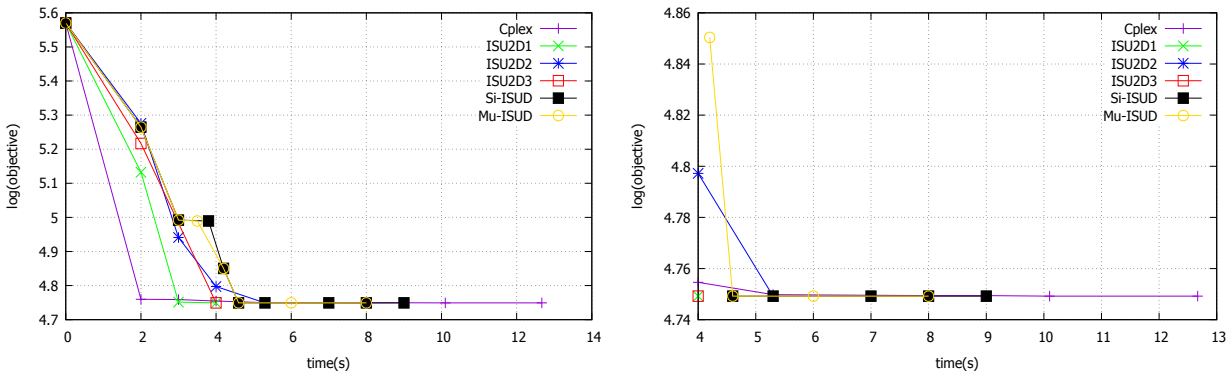


FIGURE 4.4 Evolution of objective value over time.

Table 4.5 shows the average solution times of CPLEX, ISUD and the two variants of IVD on the aircrew instances and compares them to the ISU2D$_3$ results. Table A.2 gives the detailed results (the CPLEX gap is 0 for all these instances, and we compute the reduction factor for only the instances where the gap is less than 1%). ISU2D$_3$ has the shortest computational time. It outperforms CPLEX : it is four times faster on easy instances and at least twice as fast on hard instances. As expected, ρ influences the ISU2D results : the lower the value of ρ, the better the performance of ISU2D.

ISU2D$_3$ performs well against ISUD : ISU2D$_3$ is almost twice as fast as Single-thread ISUD and almost one and a half times faster than Multi-thread ISUD. We note that Multi-thread ISUD is only slightly faster than Single-thread ISUD. The time reduction is around 10% because of the size of the instances and the high overhead. This shows that the time reduction of ISU2D comes from the decomposition methods and not from the parallelization.
Note that the performances of ISU2D$_3$ against IVD$_1$ and IVD$_2$ show clearly the good performance of DVD. That said, not to mention the number of instances solved "to optimality" using DVD phase only. This confirms the importance of this phase (DVD) in ISU2D algorithm.

TABLE 4.5 ISU2D vs. CPLEX, ISUD, IVD_1, and IVD_2 on aircrew instances

Set	C time	I_s time	I_p time	IV_1 time	IV_2 time	$ISU2D_3$ time	T_C	T_{I_s}	T_{I_p}	T_{IV_1}	T_{IV_2}
AS80	10.2	8.6	7.1	12.9	14.2	5.1	2.0	1.7	1.4	2.5	2.8
AS65	9.8	5.9	5.1	6.7	10.5	3.5	2.8	1.7	1.5	1.9	3.0
AS50	10.1	5.1	4.4	5.6	11.4	3.2	3.2	1.6	1.4	1.8	3.6

4.5.4 Bus driver scheduling instances

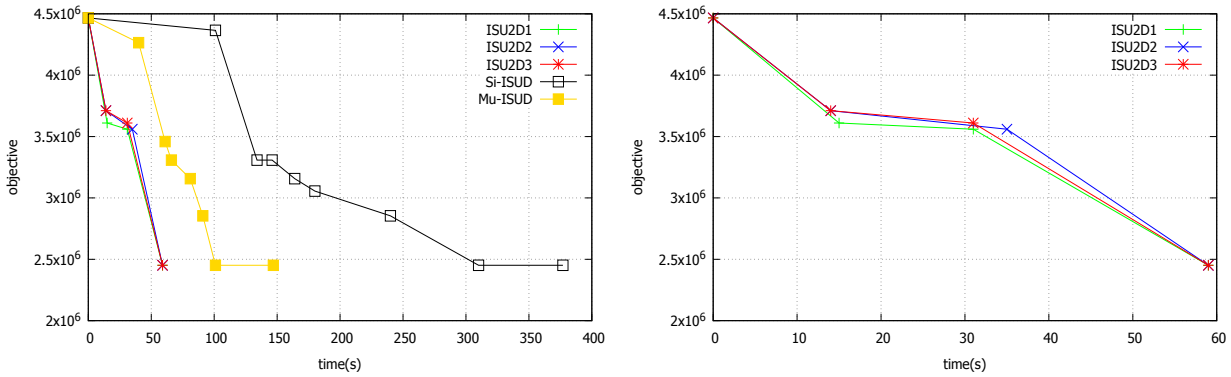
CPLEX was unable to neither obtain solutions with errors less than 10% for medium bus driver instances within a time limit of half an hour nor for large instances in a time limit of an hour. Note that CPLEX starts from the same initial solutions as ISUD and ISU2D. We instead focus on comparing ISU2D, ISUD and IVD phase only variants. Table 4.6 presents the same information for the bus driver instances that Table 4.4 presented for the aircrew instances ; the detailed results are in Tables B.1 and C.1. The number of instances solved to optimality (#Opt) by DVD is 6 in $ISU2D_1$, 14 in $ISU2D_2$ and 22 in $ISU2D_3$. The success rate is almost quadrupled. As explained in Section 4.5.3, $ISU2D_3$ has a better splitting approach. Again, the percentage of columns needed to obtain an optimal solution in IVD is significantly lower for $ISU2D_3$. $ISU2D_3$ solves 46% of the large instances to optimality during DVD ; this percentage decreases as the difficulty level increases.

TABLE 4.6 Summary of ISU2D results for bus driver scheduling instances

	$ISU2D_1$						$ISU2D_2$						$ISU2D_3$					
	DVD				IVD		DVD				IVD		DVD				IVD	
Set	#Itr	Time	%Imp	#Opt	Time	%Cols	#Itr	Time	%Imp	#Opt	Time	%Cols	#Itr	Time	%Imp	#Opt	Time	%Cols
MB-80	2	34	44	0	28	40	3	33	46	0	28	40	3	32	50	0	26	40
MB-65	2	33	58	0	48	41	3	33	58	0	34	42	3	34	60	1	36	38
MB-50	2	33	68	0	19	32	3	36	92	5	6	15	3	37	94	7	7	14
LB-80	3	206	41	0	244	43	3	198	41	0	189	43	4	200	44	0	164	43
LB-65	3	233	67	1	269	43	3	231	80	4	160	29	4	287	87	6	79.4	18
LB-50	3	218	88	5	139	25	3	206	88	5	108	26	3	225	95	8	38	10

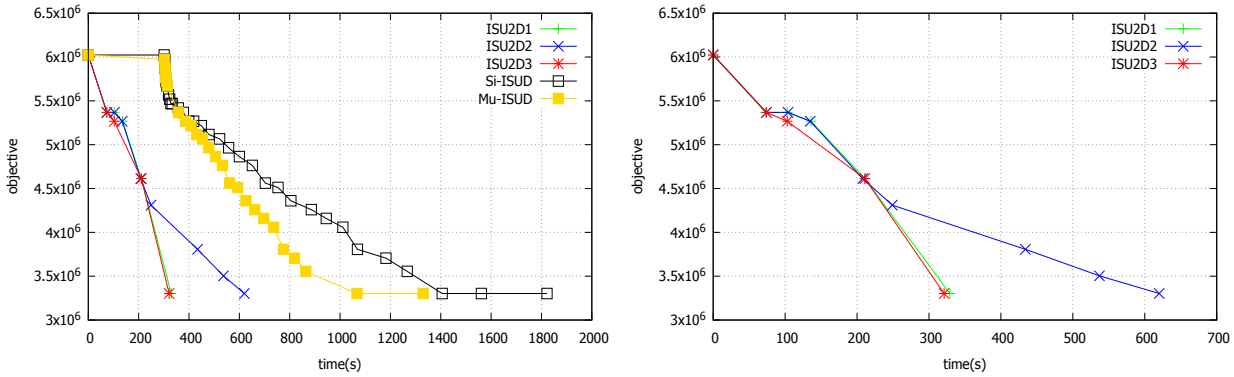
Figures 4.5-a and 4.6-a show the evolution of the objective value over time on MB80-1 and LB80-1 respectively using the following algorithms : Single-thread ISUD, Multi-thread ISUD, $ISU2D_1$, $ISU2D_2$, and $ISU2D_3$. $ISU2D_3$ is the fastest, yielding improvements of 60% and almost 75% over ISUD on these instances. The figures clearly show that the ISU2D variants outperform ISUD. The ISU2D curves decrease more sharply than the ISUD curves. Moreover, the ISUD curves are almost constant at the end of the solution process because there is no optimality proof, whereas the lower bound enables the ISU2D variants to stop

when the required solution quality is obtained. Figures 4.5-b and 4.6-b show the evolution of the objective value over time on MB80-1 and LB80-1 respectively using ISU2D variants. On LB80-1, the ISU2D$_2$ curve decreases more rapidly than the ISU2D$_1$ curve, because ISU2D$_2$ searches for the steepest descent direction at each iteration. ISU2D$_3$ is better than ISU2D$_2$. Finally, we can globally rank the algorithms from best to worst on these instances as follows : ISU2D$_3$ » ISU2D$_2$ » ISU2D$_1$ » Multi-thread ISUD » Single-thread ISUD.



(a) (b)

FIGURE 4.5 Evolution of Single-thread ISUD, Multi-thread ISUD, ISU2D$_1$, ISU2D$_2$, and ISU2D$_3$ on MB80-1.



(a) (b)

FIGURE 4.6 Evolution of Single-thread ISUD, Multi-thread ISUD, ISU2D$_1$, ISU2D$_2$, and ISU2D$_3$ on LB80-1.

Table 4.7 shows the average solution times of ISUD and IVD phase only algorithms for the bus driver instances and compares them to the ISU2D$_3$ results. Tables B.2 and C.2 give

the detailed results. We note that the percentages and their averages are computed for only the instances that were solved to optimality using ISUD. ISU2D$_3$ is again the best and outperforms ISUD : on the large instances, it is four times faster than the sequential ISUD and three times faster than the Multi-thread ISUD. On the medium instances, the reduction factor is lower because ISUD performs well. As the instance size increases, the reduction factor becomes more important because the size of the CP increases and consequently the solution time (e.g., for the simplex algorithm) increases much more (the observed complexity of the simplex algorithm is m^2n where m is the number of rows and n the number of columns). As expected, ρ influences the results of the ISU2D$_3$: the lower the value of ρ, the better the performance of ISU2D$_3$.

In contrast to the small instances, the Multi-thread ISUD is twice as fast as the Single-thread ISUD because the instances are large enough and the overhead is rather small. However, ISU2D$_3$ is much faster than the Multi-thread ISUD. The time reduction is much more important when the decomposition method is used.

TABLE 4.7 ISU2D$_3$ vs. ISUD, IVD$_1$, and IVD$_2$ on bus driver scheduling instances

	I_s	I_p	IV_1	IV_2	ISU2D$_3$				
Set	time	time	time	time	time	T_{I_s}	T_{I_p}	T_{IV_1}	T_{IV_2}
MB80	295	145	95	1800	58	5.1	2.5	1.6	-
MB65	241	109	92	1800	70	3.4	1.6	1.3	-
MB50	119	60	58	1800	44	2.7	1.4	1.4	-
LB80	2313	1203	690	3600	364	6.4	3.3	1.9	-
LB65	1806	1012	659	3600	367	4.9	2.8	1.8	-
LB50	1076	822	441	3600	263	4.1	3.1	1.7	-

Here again, The performance of ISU2D$_3$ against IVD$_1$ and IVD$_2$ highlights the essential contribution of the DVD in ISU2D$_3$ algorithm. This is in addition to the number of instances solved with DVD only. IVD$_2$ is the poorest algorithm as it is unable to improve instances solutions to less than 8% except for two medium instances where it needs more than five times computing time than ISU2D.

4.5.5 Parameters Influence

In this section, we study the influence of the ISU2D principal parameters, i.e., the number of the DVD complementary subproblems q, the number of the IVD subproblems q', and the initial solution. We present results of the influence of these parameters on the LB80-1 instance as it is sufficiently difficult and ISU2D is designed to solve large SPPs.
Table 4.8 gives results for different initial solutions for Single-thread ISUD, Multi-thread

ISUD and ISU2D. It presents both the computing time and the final gap for the three algorithms. The three methods converge towards an optimal solution despite the gap of the initial solution. They perform well for good initial solutions. Lower the gap of initial solution, the lower is the time of the solution process. In addition, we deduce that ISU2D performance increases as initial solution gap decreases.

TABLE 4.8 Influence of initial solutions on Single-ISUD, Multi-ISUD and ISU2D

initial solution		I_s		I_p		$ISU2D_3$	
num	gap_0	time	gap_f	time	gap_f	time	gap_f
1	82.0	1822	0	1335	0	320	0
2	62.5	1440	0	1303	0	332	0
3	59.5	1433	0	1301	0	330	0
4	39.6	1186	0	1095	0	325	0
5	30.5	1147	0	1044	0	328	0
6	9.2	420	0	390	0	250	0
7	3.1	308	0	320	0	227	0

Figure 4.7 shows the influence of the number of DVD complementary subproblems q, on the evolution of ISU2D during DVD phase. We choose the values of q so that they are a power of 2 (q = 2, 4, 8). This parameter controls the duration of the DVD phase and the DVD solution quality. We deduce that we get good performance for both q =2 or 4. This fact could be explained by the fact that for high values (q=8), we get poor performance as it becomes difficult to find descent direction (too many variables are ousted by the DVD decomposition process). In addition, there are more processes (8) than processors (4) which leads to the overload phenomenon. Remind that we used a computer with 4 cores for experimentation.

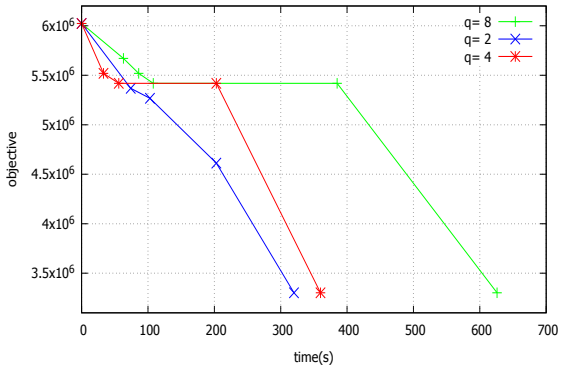


FIGURE 4.7 Evolution of objective value for different DVD parameter q values.

Figure 4.8 shows the influence of the number of IVD decompositions q', on the evolution of

ISU2D during IVD phase. We tested the following values of q' : $q' = 1, 2, 4, 8, 12$. We present both the evolution of objective value and the total computing time.

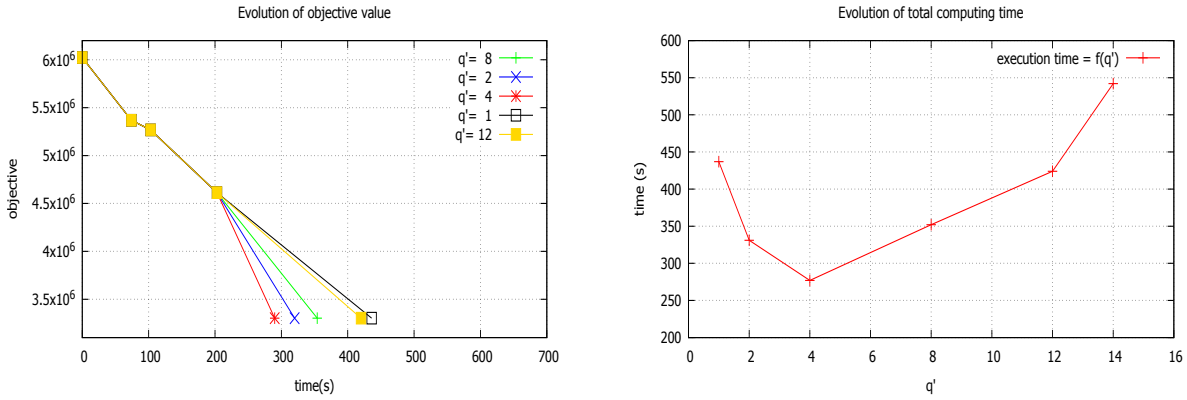


FIGURE 4.8 Influence of the IVD parameter q' on LB80-1 instance.

We deduce that we get good performance for both q' =2 or 4. This could be explained by the fact that for high values ($q' \geq 8$), we get poor performance because doing many and tiny divisions makes it difficult to have quickly the columns missing in an optimal solution. Indeed, the columns of an optimal solution would belong to different subproblems and consequently IVD phase needs more iterations to reach it. This explains the shape of the total computing time curve in function of q' : it increases for $q' \geq 5$.

4.5.6 Summary of Results

Table 4.9 summarizes the results. ISU2D$_3$ is almost twice as fast as ISUD and almost three times faster than CPLEX on the aircrew tests. It is almost three times faster than Single-thread ISUD and almost 1.5 times faster than Multi-thread ISUD on the medium bus driver tests. It is almost five times faster than Single-thread ISUD and three times faster than Multi-thread ISUD on the large bus driver tests. We can rank the ISUD and ISU2D variants from best to worst by performance average as follows : ISU2D$_3$ » ISU2D$_2$ » ISU2D$_1$ » Multi-thread ISUD » Single-thread ISUD.

TABLE 4.9 Summary of results

Set	Single ISUD	Multi ISUD	CPLEX	IV$_1$	ISU2D$_3$				
Name	Time	Time	Time	Time	Time	T_{I_s}	T_{I_p}	T_{I_C}	T_{IV_1}
AS : Small aircrew scheduling	***6.6***	***5.5***	***10.1***	***8.4***	***3.6***	***1.8***	***1.5***	***2.8***	***2.3***
MB : Medium bus driver scheduling	***163.1***	***77.9***	-	***81.4***	***56.7***	***2.9***	***1.4***	-	***1.4***
LB : Large bus driver scheduling	***1622.4***	***969.6***	-	***596.6***	***334.3***	***4.9***	***2.9***	-	***1.8***

4.6 Conclusion

We have introduced the ISU2D double decomposition to find, in parallel, descent directions leading to improved integer solutions. Our approach is applicable to vehicle and crew scheduling SPPs. It is especially beneficial for large problems. We implemented three ISU2D variants, using three different splitting methods, and discussed their performance. We demonstrated that the multistage splitting is the best. ISU2D was able to find optimal solutions for all the instances in less time than ISUD and CPLEX.

Future research on splitting methods should be done to further improve the ISU2D performance. Splitting methods with relevant weighting could be based on time-space decomposition and business rules that are specific to the application. Better lower bound dual values could be obtained by adding odd cycle cuts, clique cuts, or other facets to LP. This capability is available in commercial solvers such as CPLEX and GUROBI. In addition, combining ISU2D with heuristics that produce good initial solutions should significantly reduce the computational time for large SPPs.

CHAPITRE 5 ARTICLE 2: DISTRIBUTED INTEGRAL SIMPLEX FOR CLUSTERING

Omar. Foutlane [a,b] Issmail. El Hallaoui [a,b] Pierre. Hansen [a,c]

Paper submitted to Discrete Applied Mathematics.

[a] *GERAD, Montréal (Québec), Canada, H3T 2A7*

[b] *Department of Mathematics and Industrial Engineering, Polytechnique Montréal (Québec) Canada, H3C 3A7*

[c] *Department of Decision Sciences, HEC Montréal, Montréal (Québec), Canada, H3T 2A7*

Abstract

Clustering is the subject of active research in several fields such as operations research, statistics, pattern recognition, and machine learning. The range of applications is very wide : scheduling, vehicle routing, pattern recognition, etc. Depending on the specific needs of the community, the methods used to solve these problems vary from heuristics of rather primal nature (improving of clustering iteratively by relocation moves for instance) to exact methods of a rather dual nature where we generally solve the continuous relaxation by releasing the integrality constraints and restoring it by implicit enumeration (branch and cut or branch and cut and price). In this paper, we propose the integral simplex, an exact primal method that could be suitable for both major classes. More interestingly, it could be distributedly solved better than the dual approach. Consequently, this work aims to propose a distributed version of the integral simplex, called DISUD, using decompositions and multiple agents paradigm. Each agent dynamically splits the overall set partitioning (clustering) problem into sub-problems and solve them. The new algorithm DISUD improves at each iteration the current clustering until (near) optimality is reached. It works much better on real set partitioning instances from the airline industry than DCPLEX, the distributed version of the state of the art commercial solver CPLEX.

Keywords : Set partitioning problems, integral simplex using decomposition, multi-agents systems, distributed processing techniques.

5.1 Introduction

Clustering is the subject of active research in several fields such as operations research, statistics, pattern recognition, and machine learning. Set partitioning problem (SPP) is a combinatorial optimization problem that models well many interesting real-life clustering problems. The range of applications is very wide : scheduling, vehicle routing, pattern recognition, etc. Depending on the specific needs of the community, the methods used to solve these problems vary from heuristics of rather primal nature (improving of clustering iteratively, by relocation moves for example) to exact methods of a rather dual nature as classified by Letchford & Lodi 2002 (solving the continuous relaxation by releasing the integrality constraints and restoring it by implicit enumeration (branch and cut or branch and cut and price) when the clustering is too much constrained, like in aircrew scheduling. For instance, in the latter there are too many safety and collective agreement rules we have to respect to cluster flights (objects), i.e., flights scheduled for a pilot. In this paper, we propose the integral simplex, an exact primal method that could be suitable for both major classes. More interestingly, it could be distributedly solved better than the dual approach.

Despite our focus on the vehicle and crew scheduling applications (including but not limited to truck deliveries (see Balinski & Quandt 1964), vehicle scheduling (see Ribeiro & Soumis 1991), aircrew and bus driver scheduling (see Desaulniers et al. 1994, Chu et al. 1997, Hoffman & Padberg 1993)), the concepts and methods outlined in this paper are theoretically usable for the other application contexts.

SPP can be defined using the following crew scheduling applications terminology : a set partitioning constraint is associated with a *task* (for example, a flight leg or a bus trip to be accomplished by a *pilot or a bus driver*). Let $T = \{1, 2, ..., m\}$ be the set of tasks and $J = \{1, 2, ..., n\}$ the set of feasible schedules. With each schedule, we associate a variable x_j, a cost c_j and a column $Aj = (a_{tj})_{t \in T}$ where a_{tj} takes value 1 if Aj covers task t and 0 otherwise. The matrix $A = [A_1, A_2, ..., A_n]$ is a binary matrix. Then, the set partitioning problem formulation is :

$$
\begin{aligned}
& & Minimize \quad & \sum_{j \in J} c_j x_j & (5.1)\\
(SPP) \quad & subject\ to & & \\
& & & \sum_{j \in J} a_{tj} x_j = 1, \forall t \in T & (5.2)\\
& & & x_j \in \{0, 1\}, \forall j \in J & (5.3)
\end{aligned}
$$

The objective function (5.1) seeks to minimize the total cost. The set partitioning constraints (5.2)

ensure that each task is covered exactly once. Constraints set (5.3) imposes integrality on the x_j variables. The linear relaxation (LP) is obtained by replacing (5.3) by $x_j \geq 0, \forall j \in J$. An optimal solution of SPP consists of selecting a subset of schedules such as each task is done by one and only one schedule and the sum of the costs of the subset schedules is minimized.

The remainder of this paper is organized as follows. The literature review is presented in Section 5.2 and an overview of the contributions in Section . Section 5.4 presents briefly some useful preliminaries on the decomposition basics and the main parts of Zoom. Section 5.5 describes the new algorithm DISUD and provides a detailed algorithmic and theoretical analysis of its components. In Section 5.6, we discuss computational results and the effectiveness of our algorithm. Finally, we end this paper with some concluding remarks and suggestions for future research in Section 5.7.

5.2 Literature Review

SPP is NP-hard (see Garey & Johnson 1979). Given the wide use of SPP, there are many algorithms dedicated to its resolution. We focus on exact algorithms, using possibly heuristic stopping criterion that guarantees in practice a certain quality of the solution (optimality or near optimality). The most known method is the famous branch and cut (Hoffman & Padberg 1993, Desaulniers et al. 1997). However, this method becomes inefficient and takes huge time to reach an optimal solution for large instances due to degeneracy and the "explosion" of the branching tree.

SPP degeneracy complicates too much the solution of large $SPPs$. To deal with degeneracy in LP, El Hallaoui et al. 2011 presented the improved primal simplex (IPS) based on a decomposition processus. They decompose the problem into a non-degenerate easy to solve reduced problem and a complementary problem that finds descent directions to escape local optima of the reduced problem. Recently, Zaghrouti et al. 2014 developed the integral simplex using decomposition (ISUD), which is based on IPS, to solve SPP (with integrality constraints). Their results show that ISUD deals more efficiently with degeneracy and is able to solve large problems that are up to 570000 variables and 1600 constraints. Since then, other research works have been done by Rosat et al. 2016, 2017a and Zaghrouti et al. 2013 in order to improve ISUD performance. Rosat et al. 2017a studied the impact of adding cuts to ISUD and finds that this technique is costly in time computing for large instances. In addition, Rosat et al. 2016 compared different normalisation constraints of the cone of directions used by ISUD. As for Zaghrouti et al. 2013, they developed the Zoom algorithm based on ISUD. It explores a neighborhood of the current integer solution when it is not possible to find an improving "integer" direction, i.e., leading to an improving integer solution, using ISUD,

see Section 5.4.2 for more details. This neighborhood is constructed using the "fractional" direction returned by the complementary problem.

Nowadays, there is an increased interest to use parallel and a fortiori distributed algorithms especially with the advent of parallel computers in the world of scientific computing. The aim is to improve the solution time and to increase the size of the treated problems. Bürger et al. 2012 introduced an interesting distributed algorithm to solve degenerate linear problems. Their idea is to split the columns over a certain number of machines (agents). Each one solves the resulting reduced problem. The agents exchange their optimal bases and continue solving until the bases are all the same.

Except distributed Branch & Bound and Branch & Cut, distributed algorithms dedicated to primal integer programing and especially to SPP are to the best of our knowledge inexistant. Indeed, many authors have worked to develop distributed versions of Branch & Bound and Branch & Cut (see Eckstein 1993, Laursen 1993, Quinn 1990, Fischetti et al. 2018). They discuss issues such as architecture and communication. They apply distributed computing techniques to Branch & Bound and Branch & Cut mainly by distributing the computation of the branching tree of subproblems over multiple nodes (machines). Those research works have led to many applications. We refer the reader to an early survey by Gendron & Crainic 1994 and to the more recent one by Ralphs et al. 2017. In the latter, we report the main and classical issues that still persist : disproportionate amount of time spent in the shallowed nodes (root node particularly), unbalanced search tree, dynamic construction of the tree, dynamic generation of useful information (cuts, bounds), and consequently the need to some synchronization to avoid redundant work, which leads generally to a worse performance and scalability. CPLEX implements such a distributed mechanism using the Supervisor-Worker scheme, a kind of Master-Worker scheme where the master stores no data concerning the search tree. Its role only is to coordinate the load balancing.

The trend to develop distributed algorithms and the interest aroused by ISUD motivates us to look for a distributed version of it. The idea is to invest parallelizing a primal approach instead of a dual approach (the branch and bound for instance). This resolves the classical issues raised above. Foutlane et al. 2017 developed the integral simplex using double decomposition algorithm (ISU2D) which we generalize and improve in this paper. ISU2D is a parallel variant of the ISUD. ISU2D splits the original problem into small subproblems and solves them to get an improved solution at each iteration. The authors showed the existence of an optimal decomposition which leads to an optimal solution. The authors proposed an iterative procedure for finding such decomposition and showed the potential of such parallel methods.

5.3 Contributions Overview

In this paper, we use multi-agent system approach (MAS) to introduce a general framework for a distributed version of ISU2D called DISUD. We consider a network of worker agents, rather than a single agent such as in ISU2D (see Foutlane et al. 2017), that split SPP into a set of small subproblems, each of which containing a niche of potential improving columns. We summarize below the most important contributions of the paper :

— DISUD develops a procedure that can be seen as a parallel adaptation of *Zoom* to quickly improve the worker solution. Instead of zooming around one direction like in Zaghrouti et al. 2013, the procedure does a *multizoom* by zooming around a multitude of orthogonal directions.

— We present new theoretical results. We show that the proposed decomposition is, contrary to ISU2D, less sensitive to dual values ; it depends only on the costs of the current solution to improve. We also show that increasing the number of processors beyond a certain limit (the diameter of the polytope) is useless.

— We compare two versions : one competitive (a kind of racing implementation) and another cooperative that exploits the information gathered during the solution process. At each iteration, the two versions use dynamic decompositions simultaneously (in parallel) to guide the search to improved integer solutions.

— Tests on real crew pairing instances from the airline industry, with up to 1740 flights (per week) and more than one million of variables, show the effectiveness of DISUD compared to DCPLEX, the distributed version of CPLEX, which is the state of the art commercial solver. We succeed to compute better quality solutions (often optimal or near-optimal) in much less computational time for most instances.

5.4 Preliminaries

In this section, we provide the basic notions necessary to understand DISUD. We present the decomposition principles and the main parts of the Zoom algorithm. A parallel version of the latter is used by the agents of DISUD to solve their image of SPP. Zoom proved to be efficient in practice.

5.4.1 Decomposition Basics

Given an integer solution $\bar{x}$ to SPP, let P_{int} be the index set of its positive components, i.e., $P_{int} = supp(\bar{x}) = \{j \in J : \bar{x}_j = 1\}$, P an index set of some linearly independent columns containing at least P_{int}, and $p = card(P)$. SPP could be decomposed into a reduced problem

RP and a complementary problem using the following definition of compatibility (El Hallaoui et al., 2011) :

Definition 5.4.1. *A subset S of J is said to be compatible with P, or simply compatible, if there exist two vectors $v \in \mathbb{R}_+^{|S|}$ and $\lambda \in \mathbb{R}^p$ such that $\sum_{j \in S} v_j A_j = \sum_{l \in P} \lambda_l A_l$. The combination of columns, possibly a singleton, $\sum_{j \in S} v_j A_j$ is also said to be compatible. S is said to be minimal if any strict subset of it is incompatible.*

Let C and I be the index sets of the compatible and incompatible columns respectively. Thus, J is partitioned into C and I, i.e., $J = C \cup I, C \cap I = \emptyset$. RP is defined as a restriction of SPP to compatible columns only :

$$
\begin{aligned}
& & Minimize \quad & c_C \cdot x_C && (5.4)\\
(RP) \quad & & subject\ to \quad & A_C x_C = e && (5.5)\\
& & & x_C \in \{0,1\}^{|C|} && (5.6)
\end{aligned}
$$

As $p \leq m$, there could be some redundant constraints that we should remove from RP. When $P = P_{int}$, it is interesting to see that a pivot on any compatible column with a negative reduced cost leads to an improved integer solution. Let x_C^* be an optimal solution to RP. Observe that $\bar{x} = (x_C^*, 0)$ is a solution to SPP.

To improve $\bar{x}$, we use a complementary problem CP to find a set of incompatible columns to replace a subset of the current solution columns. More precisely, we look for a (convex) combination of incompatible columns that is compatible and has a negative reduced cost. CP can be formulated as :

$$
\begin{aligned}
& & Minimize \quad & \sum_{j \in I} c_j v_j - \sum_{l \in P} c_l \lambda_l && (5.7)\\
(CP) \quad & & s.t. \quad & \sum_{j \in I} A_j v_j - \sum_{l \in P} A_l \lambda_l = 0 && (5.8)\\
& & & e \cdot v_I = 1 && (5.9)\\
& & & v_j \geq 0\ , j \in I && (5.10)
\end{aligned}
$$

We can easily show that $d = (v, -\lambda, 0)$ defines a descent direction. Zaghrouti et al. (2013) show that if CP is infeasible or $z^{CP} \geq 0$, i.e., the objective value of CP is nonnegative, then $\bar{x}$ is an optimal solution to SPP. Otherwise, CP guarantees to find a descent direction leading to an improved integer solution. Let $S^+ = \{j \in I : v_j > 0\}$ and $S^- = \{l \in P, \lambda_l > 0\}$ be

the sets of entering and leaving variables respectively. If the columns A_j, $j \in S^+$ are pairwise row-disjoint, i.e., they do not cover the same constraints, and $S^- \subset P_{int}$, we obtain a descent direction leading to an improved integer solution. In this case, S^+ is shown to be minimal by El Hallaoui et al. (2011), i.e., nondecomposable using the terminology of Balas & Padberg (1975), meaning that pivoting on variables indexed by S^+ leads to an adjacent extreme point with improved cost.

On the other handLet $A_P = \begin{pmatrix} A_P^1 \\ A_P^2 \end{pmatrix}$ be a submatrix of A composed of columns indexed by P where A_P^1 is without loss of generality composed of the first $|P|$ linearly independent rows. A_P^2 is of course composed of dependent rows. Similarly, let $A_I = \begin{pmatrix} A_I^1 \\ A_I^2 \end{pmatrix} = (a_{ij})_{\substack{1 \leq i \leq m \\ j \in I}}$ be a submatrix of A composed of incompatible columns indexed by I with A_I^1 a $|P| \times |I|$ matrix. The variables λ can be eliminated by using the fact that the columns of A_P are linearly independent. We thus obtain an equivalent model involving only incompatible variables. In fact, constraint (5.8) could be written as :

$$\begin{pmatrix} A_I^1 \\ A_I^2 \end{pmatrix} v = \begin{pmatrix} A_P^1 \\ A_P^2 \end{pmatrix} \lambda$$

Observe that A_P^1 is invertible, so $\lambda = (A_P^1)^{-1} A_I^1 v$ and consequently could be replaced.

The incompatibility degree of a column A_j towards a given integer solution is a measure that represents a distance of the column from the solution. An example of this measure could be given by $\|MA_j\|_1$ where $M = (A_P^2 (A_P^1)^{-1}, -I_{|P|})$. $I_{|P|}$ is the $|P| \times |P|$ identity matrix. This measure is equal to 0 for compatible columns and positive for incompatible ones. The constraint can be rewritten simply as : $MA_I v = 0$; A_I is the submatrix of A containing only columns indexed by I .

When columns A_j, $j \in S^+$ are not pairwise row-disjoint, the direction d is said to be fractional. Instead of branching in CP that is a little bit complicate due to the structure of CP, Zaghrouti et al. (2013) proposed to zoom around this "fractional" direction. We discuss this in the next subsection.

5.4.2 Zoom Description

Zoom iterates between RP (compatible columns) and CP (incompatible columns) until it reaches an optimal solution. The main steps of Zoom are provided below.

Step 1 : *Find a good heuristic initial solution* x_0 *and set* $\bar{x} = x_0$, $P = P_{int}$, $d = 0$.

Step 2 : *Find a better integer solution around d :*

— *Increase P : set* $P = P \cup \{j : d_j > 0\}$.

— *Construct and solve RP.*

— *Update* $\bar{x}$ *and P : if* $\bar{x}$ *is improved, set* $P = P_{int}$.

Step 3 : *Get a descent direction d :*

— *Solve CP to get a descent direction d.*

— *If no descent direction can be found or* $|z^{CP}|$ *is small enough then stop : the current solution is optimal or near optimal.*

— *Otherwise, go to Step 2.*

We mention here that no branching is done in CP. Actually, if the direction is fractional we construct RP around this direction as explained above and solve it by a MIP solver. Zaghrouti et al. (2013) reported that this RP has good properties : small gap and density, good initial solution (that is the current integer solution) to start from, easy to solve, big chances to get an improved integer solution. We report in Zaghrouti et al. (2013) that in more than 80% of the cases, the directions found were integer. That means that no MIP was solved in Step 2 in these cases. We simply set $\bar{x} = \bar{x} + |S^+|d$ because when the direction is integer we can show that positive entries of d are all equal : $v_j = \lambda_l = \frac{1}{|S^+|}$. The CP favors integrality by its nature. We refer the reader to Zaghrouti et al. (2014, 2013) for more details.

5.5 DISUD Algorithm

As mentioned earlier, DISUD is a multi-agent algorithm where the master agent ensures, among others, the communication between other network agents called worker agents. These latter realize multiple decompositions of the problem and solve obtained sub-problems in parallel. We implemented DISUD in a such a way that each worker does not have to wait for other agents to end their iteration to start a new iteration. Consequently, DISUD reduces overhead due to communication synchronization. Rather, it exploits the available time to improve the current solution. DISUD stops when the master agent receives a satisfactory solution. In this section, we start by presenting the worker and master agents in detail in Sections 5.5.1 and 5.5.2 respectively. Also, we give a theoretical analysis of DISUD in Section 5.5.3. Finally, we note that throughout this paper we use superscript [i] to denote quantities belonging to the i^{th} agent.

5.5.1 Worker Agents

Worker agents realize multiple decompositions to increase the chance to get an improved solution. Each worker agent solves SPP using a specific decomposition either with DVD or

IVD mechanisms as described briefly in Sections 5.5.1.1 and 5.5.1.2 respectively. More details are in Foutlane et al. 2017. A worker agent reacts to the messages received from the master as indicated in the following Algorithm 6. We then, give an illustration using the MAS paradigm in Section 5.5.1.3.

Algorithm 6 Worker agent algorithm

```
Do
    Wait for a message from the master. In case of :
        msgSOL : Set x̄^[i] = x_b
        msgMODE-DVD : Call DVD algorihm (Algorithm 7) ;
        msgMODE-IVD : Call IVD algorihm (Algorithm 8) ;
        msgMODE-IDLE : Wait ;
        msgSTOP : Stop (do memory cleaning) ;
While (true)
```

5.5.1.1 DVD Mode

The idea of DVD is to split the original complementary problem into small "orthogonal" complementary subproblems (CSPs) that can be efficiently solved in parallel. In such subproblems, we look for replacing some of the variables from $supp(\bar{x}^{[i]})$ by some more interesting ones. So, during the DVD mode, the worker agent partitions $P_{int}^{[i]} = supp(\bar{x}^{[i]})$ into q clusters where the columns indexed by cluster k cover a set of tasks T_k, i.e., $T = \cup_{1 \leq k \leq q} T_k$. Let $I_k \subset I$ be the subset of incompatible columns that cover only tasks in T_k. We have $I^{[i]} = \cup_{1 \leq k \leq q} I_k$. The agent i constructs hence an RP and q complementary subproblems $(CSP_k^{[i]})_{1 \leq k \leq q}$ formulated as follows :

$$
\begin{aligned}
& & min \quad & \sum_{j \in J_k} \bar{c}_j v_j & & \\
CSP_k^{[i]} & & & MA_{I_k} v_k = 0 & & (5.11) \\
& & & \sum_{j \in I_k} v_j = 1 & & (5.12) \\
& & & v_j \in \{0,1\} \qquad \forall j \in I_k & & (5.13)
\end{aligned}
$$

To do so, each agent defines a weighted graph $G(V, E)$ where each column A_v, $v \in \text{supp}(\bar{x}^{[i]})$ is represented by a vertex $v \in V$. Let $(v, v') \in V^2$, $I_{vv'} = \{l \in I \ : \ A_v \cdot A_l \neq 0 \text{ and } A_{v'} \cdot A_l \neq 0\}$ and $T_{vv'}$ is the set of all tasks covered by either A_v or $A_{v'}$. We define E as the set $\{(v, v') \ : \ I_{vv'} \neq \emptyset\}$. It is obvious that if $E = \emptyset$ then the constraint matrix A is a bloc diagonal matrix and SPP is composed of independent set partitioning problems that can be solved in parallel. This is generally not the case in practice. Based on this, the weight of the edge $(v, v') \in E$ measures the "likelihood" that some of the variables indexed by $I_{vv'}$

could improve the objective value if entered into the basis. We partition the graph into q disjoint subgraphs using a min-cut algorithm (see Kernighan & Lin 1972). When the edge $(v,\ v')$ is not cut, A_v and $A_{v'}$ are grouped into a cluster and $\exists k\,;\ 1 \leq k \leq q$ where $T_{vv'} \subseteq T_k$. Thus, the variables indexed by v and v' could be part of a descent direction (as leaving variables). Obviously, when the edge (v, v') is cut, it is not possible in the current iteration of DVD to improve the objective value with the variables indexed by $I_{vv'}$. We proved in Foutlane et al. 2017 the existence of an optimal decomposition. But, as this latter is not known a priori, the worker agents implement different weighting methods simultaneously and consequently manage different weighted graphs $G(V, E)$ to increase the chance to find one rapidly. The efficiency of this depends heavily on the edge weights. Consequently, each agent of the network implements a different weighting method $we^{[i]}$ to calculate edge weights. We suggest computing $we^{[i]}$ as a function of the reduced cost, the incompatibility, the number of covered tasks (non zeros elements), and other relevant attributes of columns $A_j \in I_{vv'}$. In the following, we list four agents we used in this proof of concept. Two of them implement the two weighting methods that we studied in ISU2D. Of course, the list of agents is not exhaustive and other agents could be added easily using this framework. **Agent 1**. The first weighting method $we^{[1]}$ stipulates that a descent direction is more likely to exist in regions where there are more columns with negative reduced costs columns. We therefore cut the edges with the smallest number of negative reduced costs. Based on this, $we^{[1]}$ associates with each edge (v, v') the number of negative reduced cost columns that $I_{vv'}$ contains.

$$we^{[1]} : (v, v') \mapsto w_{vv'} = |\{j \in I_{vv'} \ : \ \bar{c}_j \leq 0\}|$$

Agent 2. The second weighting method is inspired by the simplex algorithm. It increases the chances of getting a large step (improvement in the objective value) provided that a descent direction exists. We assume that one of the entering variables has the smallest negative reduced cost. Therefore, $we^{[2]}$ associates with the edge (v, v') the absolute value of the smallest negative reduced cost column indexed by $I_{vv'}$.

$$we^{[2]} : (v, v') \mapsto w_{vv'} = -\min(0, \min\{\bar{c}_j : \ j \in I_{vv'}\}).$$

Agent 3. The third weighting method $we^{[3]}$ derives from $we^{[2]}$. We compute $we^{[3]}$ using the reduced cost and the the number n_j of tasks covered by A_j. We stipulate that a good entering variable should have the smallest average negative reduced cost. Thus, when two columns have the same reduced cost $\bar{c}_j$, we favor the one that covers fewer tasks.

$$we^{[3]} : (v, v') \mapsto w_{vv'} = -\min(0, \min\{\frac{\bar{c}_j}{n_j} : \ j \in I_{vv'}\}).$$

Agent 4. Given the importance of the degree of incompatibility in ISUD, we suggest to compute the fourth weighting method $we^{[4]}$ using the reduced cost and the incompatibility degree $k_j = \|MA_j\|_1$ of a column A_j. Indeed, the incompatibility degree can be seen as the distance from column A_j to the vector subspace generated by the columns of the current integer solution. It can be interpreted as the number of changes to the current solution. We compute $we^{[4]}$ then as :

$$we^{[4]} : (v, v') \mapsto w_{vv'} = -\min(0, \min\{\frac{\bar{c}_j}{k_j} : \ j \in I_{vv'}\}).$$

Observe that we thus favor solutions that are primally not too far from the current one because we suppose that the current one is good. Hence, building the subproblems reduces to defining the partition $\tau = (T_k)_{1 \leq k \leq q}$. DVD procedure can be interpreted as a parallel adaptation of *Zoom* to quickly improve the worker agent solution. Instead of zooming around one direction, DVD does a *multizoom* by zooming around a multitude of orthogonal directions. Algorithm 7 outlines the DVD procedure.

Algorithm 7 DVD pseudocode for agent i

Build $\tau^{[i]}$ and consequently $CSP_k^{[i]}$, $k \in \{1 \dots q\}$ using $we^{[i]}$.
Solve in parallel the $CSP_k^{[i]}$, $k \in \{1 \dots q\}$.
For $k = 1$ to q'
    IF d_k is integer (d^k is the direction returned by $CSP_k^{[i]}$) THEN
        Set $\bar{x}^{[i]} = \bar{x}^{[i]} + |S_k^+| d^k$.
    ELSE
        Set $P^{[i]} = P^{[i]} \cup \{j : d_j^k > 0\}$
        ENDIF
End for.
If some d_k is fractional, construct RP according to $P^{[i]}$ and solve it by a MIP solver (multizoom).
Send the resulting $x^{[i]}$ to the master agent.

As it can been seen, DISUD deals with many partitions rather than considering just one partition as it is the case in ISU2D. As a consequence, it is obvious that DISUD is a generalization of ISU2D.

5.5.1.2 IVD Mode

The IVD idea is to explore near-optimal "LP" neighborhoods incrementally. A worker agent starts by a neighborhood containing potential columns with good reduced costs (computed using the LP duals returned by the master agent) and increments the number of columns as needed. It solves in this neighborhood using Zoom starting with the best solution returned by its own DVD procedure in the competitive case and with the one returned by the master

agent in the cooperative mode. Algorithm 8 provides the pseudocode where q' is a parameter tuned by experimentation.

Algorithm 8 IVD pseudocode for agent i

Price the columns using μ (i.e., compute their reduced costs).
Sort the variables in an increasing order of their reduced costs and reindex them.
For $k = 1$ to q'
    For all j, if $\bar{z}_{lb} + \bar{c}_j > z_{ub}$, $J = J \setminus \{j\}$, i.e., $x_j = 0$.
    Build $SPP_k^{[i]}$ by considering the first $k\frac{|J|}{q'}$ variables.
    Solve $SPP_k^{[i]}$ with Zoom, set $\bar{x}^{[i]}$ to the obtained solution, and update z_{ub}.
    Send $\bar{x}^{[i]}$ to the master agent.
End for.

5.5.1.3 Illustration

We use the same terminology introduced by Notarstefano & Bullo 2011 to illustrate the mechanics of a worker agent. Like any distributed algorithm, we have :

1. Set of states W : At a time t, a worker agent is either in DVD or IVD mode improving a solution $\bar{x}$ or in an IDLE mode waiting for a message from the master. Thus, the set of the worker agent states W is the set of couples $(mode, \bar{x})$. In addition, we have the END state indicating that the worker agent is done.
2. Messaging function : The set of messages $\mathcal{A}$, called Alphabet, can be subdivided into two subsets. The first one concerns the solution transmission : the worker agent communicates its solution $\bar{x}^{[i]}$ to the master agent and, when worker agents cooperate, the master sends in its turn the best primal solution $\bar{x}_b$. The master also sends the dual solution μ to the worker agents when switching to IVD mode. In order to avoid communication overhead, the sending of a primal solution is accomplished by the transmission of its support only. The second subset contains other messages controlling the state of the worker agent : msgMODE-DVD and msgMODE-IVD to specify which type of decomposition to use, msgMODE-IDLE to stop temporarily a worker agent, msgSTOP to end DISUD.
3. State transition : Upon the reception of a message from the master, a worker agent updates its state as it is illustrated in Figure 5.1. The latter retraces the state transitions of the agent depending on its current state which would be one of three possible states : IDLE, DVD, or IVD.

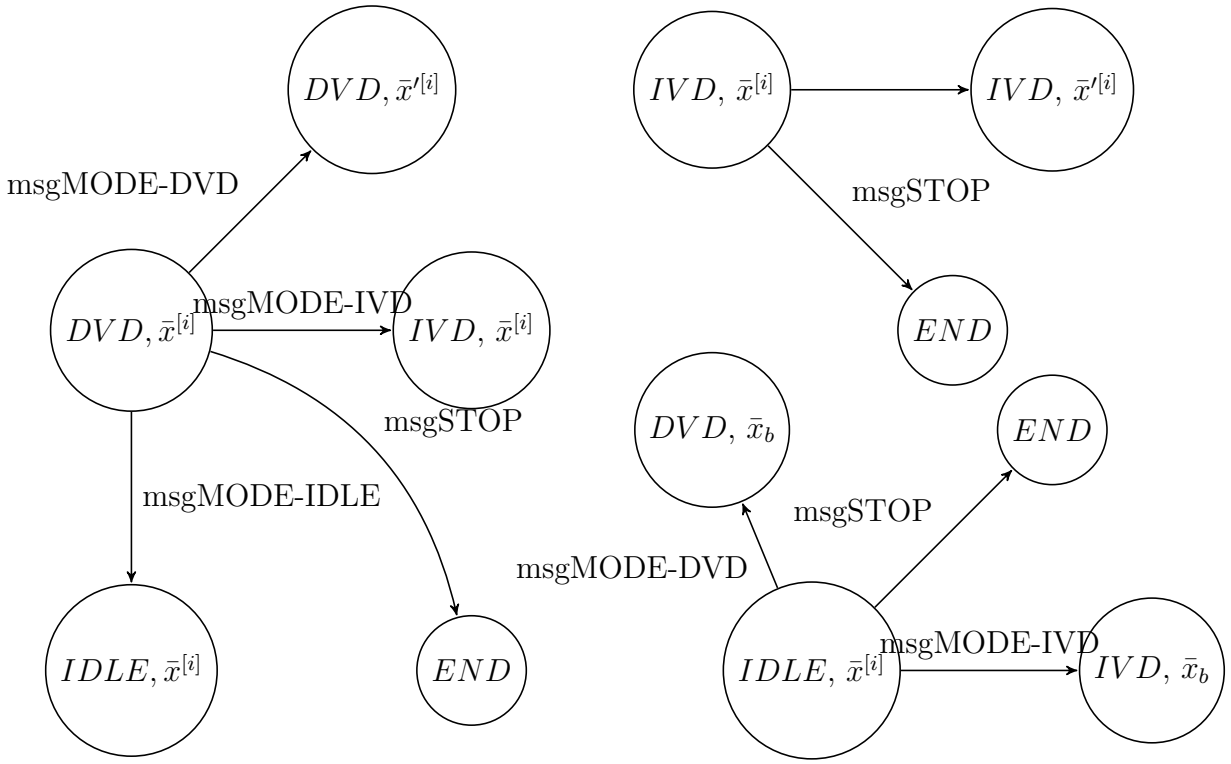


FIGURE 5.1 State transition of a worker agent

An illustration of a working agent states transition from the begining of DISUD to the end is given in Figure 5.2.

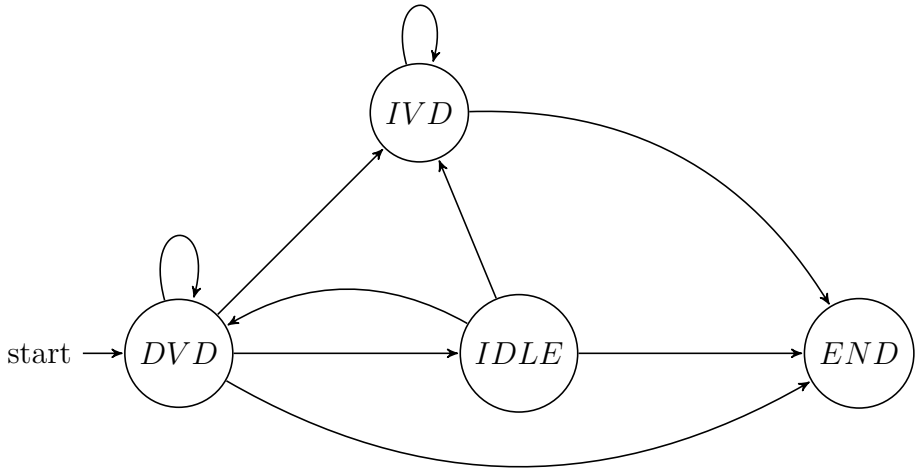


FIGURE 5.2 Worker agent basic working

5.5.2 Master Agent

Globally, the master agent calculates a lower bound, controls the execution of DISUD and ends it when a termination criterion is satisfied. We developed two variants of DISUD : the cooperative variant where worker agents cooperate and the competitive variant where worker agents work independently. Pseudocode of cooperative and competitive variants are provided in Sections 5.5.2.1 and 5.5.2.2 respectively. The IN-PARALLEL and END-PARALLEL terms are used to mention that the multiple statements in between are executed in parallel.
For both variants : the master starts with an initial primal solution x_0 of value z_0, sets the upper bound $z_{ub} = z_0$, sends supp(x_0) to all agents, begins to calculate in parallel a lower bound z_{lb} and waits for the solutions obtained by the worker agents. When the master receives a solution $\bar{x}^{[i]}$ from the i^{th} agent that improves the DISUD upper bound z_{ub}, DISUD updates z_{ub} and x_b, the best solution encountered, accordingly. Moreover, if its quality is satisfactory, then DISUD stops the solution process. In the other case, the master reacts according to which variant is activated. The master initializes a counter for each agent $nbrItr^{[i]}$ and increments it after each received solution from the worker agent i. The master agent uses this counter to tell the worker to stay in DVD mode or to move to IVD mode when it reaches a predefined value $IterMax$.

5.5.2.1 DISUD with Cooperative Agents

In the cooperative variant, the master intervenes frequently during the progress of the algorithm. This is shown clearly in Algorithm 9. We discuss below the most important issues. During the DVD mode, the communication between the master and the worker agent is bilateral : the worker agent sends its newly found solution to the master and waits for the solution x_b from which it starts and the OK to stay in DVD mode or any other decision from the master. The communication is rather unilateral between the master and a worker agent in the IVD mode : the worker agent sends its solution to the master and does not wait for a feedback from the master ; the latter just decides to continue or to stop the solution process depending on the quality of the solution it holds. The worker agent does not need x_b. Simply, it improves its $z_{ub}^{[i]}$ until reaching its best solution or get interrupted by the master. To start from different solutions, the master adjusts continously with the maximum number of iterations allowed in DVD mode. When an agent transits to IVD mode, it increment this counter to let other worker agents continue improving x_b in DVD mode. Finally, if a worker agent does not improve its own solution, the master makes it idle, i.e. waiting for an improved x_b to start from. If all agents are idle, the master let all of them to switch to IVD using different initial solutions.

Algorithm 9 Master cooperative pseudocode

```
Set z_ub = z_0, x_b = x_0 and for each agent i, set nbrItr^[i] = 0, mode^[i] = DVD
IN-PARALLEL
    Calculate z_lb, set μ to the obtained dual solution
    Send msgSOL, x_b, msgMODE-DVD to all worker agents
    Listen to worker agents
    On the reception of x̄^[i] from some agent i, DO (in sequential)
        Set nbrItr^[i] = nbrItr^[i] + 1
        If (c^t x̄^[i] − z_lb)/z_lb ≤ ε, Send msgSTOP to all worker agents; Stop DISUD
        IF mode^[i] = DVD THEN
            IF c^t x̄^[i] < z_ub THEN
                Set z_ub = c^t x̄^[i], z_ub^[i] = c^t x̄^[i], x_b = x^[i]
                Send msgSOL, x_b, msgMODE-DVD to agent i and idle agents
                and set their mode to DVD
            END IF
            IF z_ub <= c^t x̄^[i] < z_ub^[i] THEN
                Set z_ub^[i] = c^t x̄^[i]
                IF nbrItr^[i] = IterMax OR (z_ub − z_lb)/z_lb ≤ ε_dvd THEN
                    Send msgSOL, x_b, msgMODE-IVD to i
                    Set mode^[i] = IVD
                    Send μ to agent i
                    Increase IterMax by ΔIterMax if nbrItr^[i] = IterMax
                ELSE send msgSOL, x_b, msgMODE-DVD to agent i
                END IF
            END IF
            IF c^t x̄^[i] >= z_ub^[i] THEN
                Put the agent i in the idle queue and set mode^[i] = IDLE
                Send msgMODE-IDLE to agent i
            END IF
            IF all agents are idle THEN
                Send them msgSOL, x_b, msgMODE-IVD, μ
                Change their mode to IVD
            END IF
        END IF
    END DO
IN-PARALLEL
```

5.5.2.2 DISUD with Competitive Agents

Concerning the competitive variant, if the received solution improves the i^{th} agent's upper bound $z_{ub}^{[i]}$ only, then the master agent lets the i^{th} agent make another DVD decomposition, of course, if its iteration number does not exceed a predefined value IterMax and the DVD gap threshold ϵ_{dvd} is not reached yet. Otherwise, it sends to the i^{th} agent the message msgMODE-IVD in order to switch to IVD mode. In this case, it sends also the dual values obtained by solving to optimality the linear relaxation. Algorithm 10 presents the master competitive procedure. Note that the instructions comprised between DO and END DO are executed in sequential.

Algorithm 10 Master competitive pseudocode

Set $z_{ub} = z_0$, $x_b = x_0$ and for each agent i, set $nbrItr^{[i]} = 0$, $mode^{[i]} = DVD$
IN-PARALLEL
  Calculate LP-lower bound for SPP, , set μ to the obtained dual solution
  Send msgSOL, x_b, msgMODE-DVD to all worker agents
  Listen to worker agents
  On the reception of $\bar{x}^{[i]}$ from some agent i, DO (in sequential)
    Set $nbrItr^{[i]} = nbrItr^{[i]} + 1$
    IF $c^t\bar{x}^{[i]} < z_{ub}$ THEN
      Set $z_{ub} = c^t\bar{x}^{[i]}$ and $x_b = x^{[i]}$
      IF $\frac{z_{ub}-z_{lb}}{z_{lb}} \leq \epsilon$ THEN ; Send msgSTOP to all worker agents ; Stop DISUD.
    END IF.
    IF $mode^{[i]} = DVD$ THEN
      IF $c^t\bar{x}^{[i]} = z_{ub}^{[i]}$ OR $nbrItr^{[i]} \geq IterMax$ OR $\frac{z_{ub}-z_{lb}}{z_{lb}} \leq \epsilon_{dvd}$ THEN
        Send msgMODE-IVD to agent i and change its $mode^{[i]}$ to IVD
        Send LP dual values μ to agent i
      ELSE
        Send msgMODE-DVD to agent i
        Set $z_{ub}^{[i]} = c^t\bar{x}^{[i]}$
      END IF
    END IF
  END DO
END IN-PARALLEL

5.5.3 Theoretical Analysis

We discuss below that working with the partial or the standard reduced costs should give similar results. Let d be an integer descent direction, $S^+ = \{j : d_j > 0\}$, and $P^- = \{l : \lambda_l > 0\}$.

Proposition 5.5.1. *Let $\bar{\bar{c}}_j$ the reduced cost of variable j, computed with a dual solution α corresponding to the current integer solution. We have $\sum_{j\in S^+} \bar{\bar{c}}_j = \sum_{j\in S^+} \bar{c}_j$, $\forall\alpha$.*

To prove this, we need the following lemma that can easily be derived from Proposition 9 of Zaghrouti et al. 2014.

Lemma 5.5.2. *We have : $v_j = \lambda_l = \dfrac{1}{|S^+|}, \forall(j,l) \in S^+ \times P^-$.*

Démonstration. (of proposition 5.5.1) Let α be a corresponding dual solution and B the corresponding basis. We have : $\sum_{j\in S^+} A_j = \sum_{l\in P^-} A_l$. So, $\sum_{j\in S^+} c_B^T B^{-1} A_j = \sum_{l\in P^-} c_B^T B^{-1} A_l$. Meaning that $\sum_{j\in S^+}(c_j - \bar{\bar{c}}_j) = \sum_{l\in P^-} c_l$ and consequently $\sum_{j\in S^+}(\bar{\bar{c}}_j) = \sum_{j\in S^+} c_j - \sum_{l\in P^-} c_l$. On the other hand, from Lemma 5.5.2, (5.7), and (6.11), we obtain $\sum_{j\in S^+} \bar{c}_j = \sum_{j\in S^+} c_j - \sum_{l\in P^-} c_l$. This concludes the proof. □

The next corollary shows a weak "equivalence" between the partial and the standard reduced costs. We think that we do not need a stronger equivalence because theoretically, they behave the same in the worst case.

Corollary 5.5.3. *There exist necessarily $j, k \in S^+$ such that $\bar{c}_j < 0$ and $\bar{\bar{c}}_k < 0$.*

We can show that there exists a linear transformation such that $\forall j \in S^+, \bar{\bar{c}}_k < 0$. The next proposition indicates that increasing the number of needed processors (for solving subproblems) beyond a certain limit, that is the diameter of the polytope, is particularly "useless" in DVD mode.

Proposition 5.5.4. *Let q_{opt} be the number of CSPs that permit to get an optimal solution x^* in one DVD iteration where each CSP reveals exactly one descent direction. We have $q_{opt} = dist(x^*, \bar{x}) \leq d$ where d is the diameter of $conv(SPP)$.*

Démonstration. We can show that $supp(x^*) \setminus supp(\bar{x}) = (\cup_{1 \leq k \leq k'} S^+_{k'})$ where S^+_1, S^+_2, ... , $S^+_{k'}$ are minimal disjoint compatible subsets (solutions to the CSP), i.e., their corresponding task subsets T^1, T^2, ..., $T^{k'}$ are disjoint. By definition, the diameter of a polyhedron is the maximum distance between each two of its vertices ; note that the distance between two vertices is the minimum number of edges needed to reach the second one, starting from the first vertex. To move on an edge from a vertex to an adjacent one, recall that we need to pivot on a certain S^+_i, a minimal compatible subset of entering columns. Consequently, $q_{opt} \leq k'$. □

Remark 5.5.5. *We have two interesting facts :*

- *There exist an infinity of optimal weighting methods we^* such that the weighted graph $G(V, E)$ is a disconnected graph and the resulted complementary subproblems should provide directions leading to an optimal solution x^*.*
- *Let x_1, x_2 be two integer solutions and x_3 and x_4 their respective adjacent extreme points (also integer solutions). We may have $c \cdot x_3 < c \cdot x_4$ even though $c \cdot x_1 >> c \cdot x_2$.*

Proposition 5.5.6. *DISUD is a monotonic exact algorithm that converges in a finite time.*

Démonstration. DISUD is a multi-agent algorithm where every agent converges in a finite time since :

- During DVD phase, the partitions are different and the number of these different partitions is finite.
- During IVD phase, the number of iterations is finite because the number of column subsets is finite.

□

5.6 Computational Results

In this section, we present results of DISUD and discuss the effectiveness of our multi-agent algorithm. We show that DISUD results are interesting compared to DCPLEX.

5.6.1 Aircrew Instances

In aircrew scheduling, a pairing is a sequence of flights that start and end at the same airport. The crew pairing problem CPP consists of finding a set of pairings that covers all the scheduled flights at minimal cost over the planning horizon. Moreover, each flight has to be covered by a single pairing and therefore, CPP is modeled as a SPP. We tested DISUD on SPP derived from real-life CPP instances. The original datasets can be found in Kasirzadeh et al. 2017 and concern aircraft fleets D94, D95, 757, 319, and 320. We used GENCOL, a commercial software, to generate columns (a set of pairings) using different values for the dominance and incompatibility parameters. We extracted different tests at different phases of the process to build the set of our tests. Therefore, we obtained five groups of instances where each group contains six different instances. Furthermore, considering that DISUD needs an initial solution to start from, we choose a solution from those proposed by GENCOL. Table 5.1 presents the main characteristics of the instances. It reports for the set of tests the dataset name, the number of tasks m, the minimum number n_{min} and maximum n_{max} of columns before preprocessing. Then it prsents informations of the CPLEX reduced problem obtained after precossing. Indeed, it presents the minimum number m_{min} and maximum m_{max} of tasks and the minimum number n_{min} and maximum n_{max} of columns.

Table 5.1 Characteristics of instances

Serie	Before preprocessing			After preprocessing			
	Tasks	Variables		Tasks		Variables	
	m	n_{min}	n_{max}	m_{min}	m_{max}	n_{min}	n_{max}
D94	712	58144	110035	453	453	42433	80307
D95	2123	818686	1479515	1340	1344	503046	864673
757	2175	891269	1308428	1422	1423	396889	593987
319	2189	631300	831530	1418	1419	407249	513466
320	2931	689254	1313438	1924	1925	419280	674862

5.6.2 Testing Methodology

We implemented the two variants competitive ($DISUD_{comp}$) and cooperative ($DISUD_{coop}$) using C^{++} and the MPI (Message Passing Interface) library. This latter enables communication between the master and worker agents. The master run on a Linux PC with Quad-Core

processor of 3.30 GHz and the worker agents run on Linux PCs with 8 processors of 3.4 GHz each as shown on Figure 5.3.

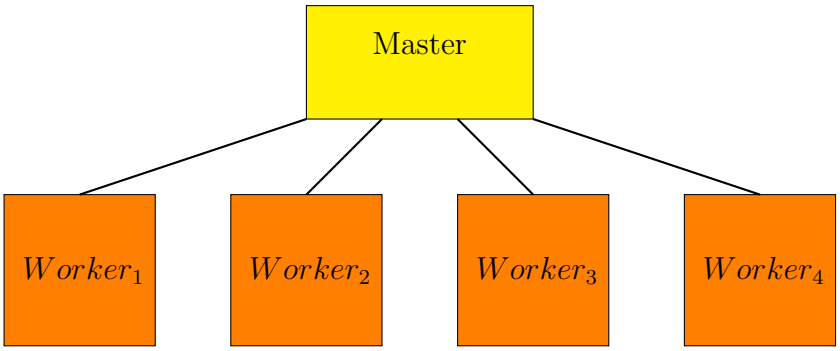


FIGURE 5.3 DISUD network architecture

The CPLEX version used here is IBM CPLEX 12.6.1. For each test, algorithms were run within a time limit of one hour except of D94 derived tests which were run during 4 min. DISUD parameters are : $\epsilon_{dvd} = 0.02$, q =4 , IterMax=10 and q'=2. The threshold of 2% comes from industrial observation : a solution with a gap of 1% is considered excellent, and one within 2% is acceptable as reported by Rosat et al. 2017b. First, we compare DISUD variants. Then we compare these variants to DCPLEX. The latter (see IBM Knowledge Center) is a distributed version of the well known branch and bound algorithm. It is dedicated to solve a MIP in an environment of distributed memory across multiple machines. It is based on a single master associated with multiple workers. DCPLEX presolves the problem on the master and sends the reduced model to each of the workers. Each of the workers then starts to solve the reduced model using its own parameter setting. This phase is known as ramp up. Then, the master selects the worker which performed the best and distributes its search tree over all workers : They work on the same search tree, with the master coordinating the search. We use the following ratios to compare DISUD to DCPLEX :

— The time efficiency ratio $T_a(b)$ between two algorithms a and b is defined as $T_a(b) = 100 * \frac{t(a)}{t(b)}$ where $t(a)$ and $t(b)$ are the computational times of a and b.

— The gap between the found solution value $z(a)$ returned by an algorithm a and the lower bound value z_{lb} is defined as $Gap(a) = 100 * \frac{z(a)-z_{lb}}{z_{lb}}$.

5.6.3 Cooperative vs Competitive Results

In this part, we show DISUD_{coop} and DISUD_{comp} results and discuss their performances on our set of tests. Figure 5.4 shows the evolution of the objective value over time for DISUD_{coop} on instance D95_1. It presents clearly the contribution of all agents during the process solution.

Indeed, it shows the strong point of DISUD : at any moment DISUD solution is the best solution realized by its agents. In addition, we can observe a rapid objective value decrease at the beginning of the solution process while the objective value decrease becomes slow at the end. In other words, the DVD phase (at the begining of the process) yields a large decrease than the IVD phase (at the end). This is explained by the fact that during DVD phase, DISUD combines orthogonal descent directions. This behavior is typical and representative of the other instances.

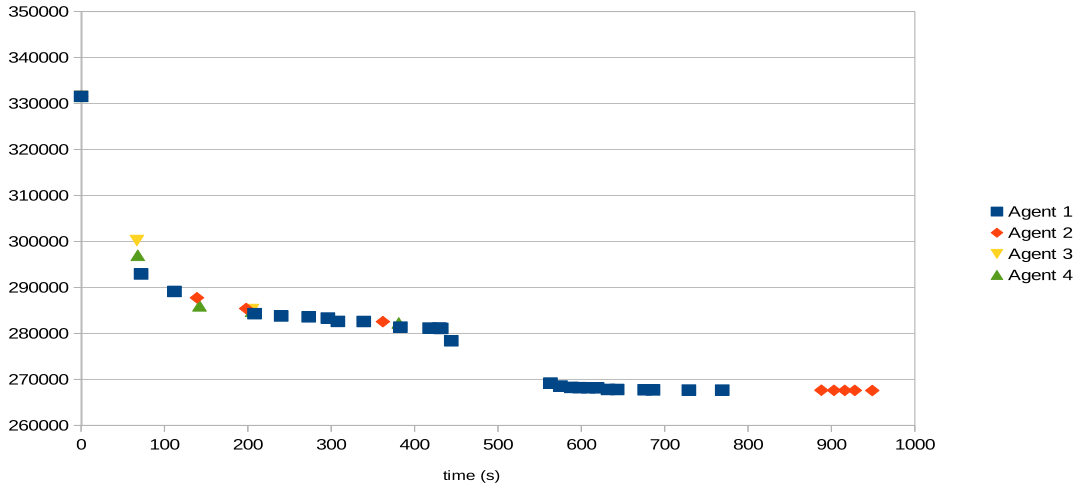


FIGURE 5.4 DISUD$_{comp}$ Evolution over time on the D95_1 instance

Figure 5.5 shows the evolution of DISUD$_{coop}$ and DISUD$_{comp}$ on the test 320_5. We connect the points to improve its readability. We note that the two curves present the same pace. DISUD$_{coop}$ is better in the middle of the process solution. This is due to the fact that DISUD$_{coop}$ embeded the spirit of the branch and bound depth first search strategy. DISUD$_{coop}$ uses all its agents to explore its best solution x_b (solution with the lowest cost) neighborhood.

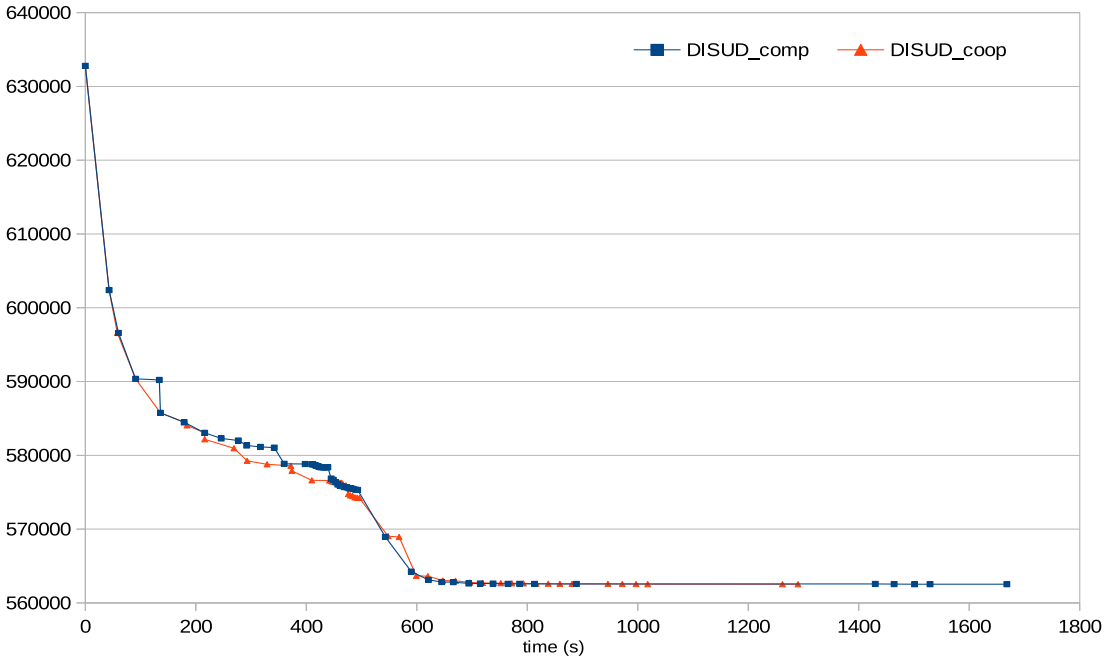


FIGURE 5.5 DISUD variants Evolution over time on 320-5 test

The set of tables 5.2-5.6 show results for each variant of DISUD. They report test name, number of columns, the gap value of the initial solution, gap values for DVD and IVD phases respectively, the number of times that the best agent invokes a mixed integer program during the SPP resolution and the time devoted to solving these MIP programs, the time to obtain the best solution t_{obj}, for DISUD$_{coop}$ the average time, t_{idle}, that an agent is in idle state and the agent identity Ag_b that reaches the best solution. Also, we have included average lines in bold to compare the average behavior of the two variants.

We observe that both DISUD variants solve all tests to near optimality within almost half an hour. Their solutions quality is less than 1% in all cases. Thus, the results show that the two variants of DISUD were approximately equal in terms of solution quality. Even if the found solutions are excellent, we can see that DISUD$_{comp}$ beats DISUD$_{coop}$ in 37% of the tests in terms of solution quality.

DISUD$_{comp}$ and DISUD$_{coop}$ solved 33% and 64% of tests to less than 2 % during DVD respectively. Furthermore, DISUD$_{coop}$ solved 83% of the 320 and 319 tests to less than 2 % during DVD. These results show that DISUD$_{coop}$ is better than DISUD$_{comp}$ as it is a variant that enables to increase the size of treated problems. Indeed, with large problems, DISUD coop has the potential to use only the DVD decomposition to get industrially acceptable solutions. In addition, the average idle time of an agent is small compared to computing time (less than 1%).

Let us compare DISUD variants computing time as all their solutions are excellent from industrial point of vue. As expected, DISUD$_{comp}$ is faster than DISUD$_{coop}$. The competitive variant beats the cooperative one in 60% of cases. DISUD$_{coop}$ is slower than DISUD$_{comp}$ because it forces worker agents to stay more in DVD phase instead of letting them switch to IVD phase. We would like to point out that the increase of the average DISUD$_{coop}$ computation time differs according to the dataset : it is 0%, 20.9%, 6.7%, 46.9% and -2.1% for D94, D95, 757, 319 and 320 tests respectively.

Concerning Zoom algorithm, we note that MIP influences the performances of DISUD variants : higher the value of MIP time, the lower is the time performance of DISUD. We note that the number of times that a mixed integer program was invoked is small and is approximatively similar for the two variants : they differ in average by one call. Meanwhile, the time reserved to solving these programs differ according to the dataset. Indeed, for DISUD$_{comp}$ the ratio is 1.1%, 35.7%, 8%, 14.2% and 6.8% of the solution process time in average for D94, D95, 757, 319, and 320 instances respectively. For DISUD$_{coop}$ the ratio is 2.1%, 39.9%, 7%, 23.3% and 6.3% of the process solution time in average for D94, D95, 757, 319, and 320 instances respectively. Despite the large proportion of time it could consume, this step has proven to be useful during the resolution : Mip is invoked for all large instances.

In general, DISUD variants perform well when they start with a good initial solution (low initial gap). They reproduce a known fact of primal algorithms which is sensitivity to the initial solution. Their DVD gap decreases with low initial solution gap value.

Finally, it is obvious that all agents contribute to the DISUD process solution as it is shown by the Ag_b column. Based on this, we deduce that considering many agents simultaneously is a better approach. But the agents contributions differ : the first and the second agents are the best for 83% of DISUD$_{comp}$ tests and 66% of DISUD$_{coop}$ tests.

From the aforementioned results, we conclude that DISUD variants yield better results. DISUD$_{coop}$ constitutes a good variant of DISUD that shows a good potentiel to treat larger SPP since its DVD results and the difficulties that may arise when managing the IVD phase. More, DISUD$_{coop}$ allows to manage multiple agents simultaneously in order to take advantage of their actions.

TABLE 5.2 DISUD results using D94 instances

			DISUD$_{comp}$						**DISUD$_{coop}$**						
Name	n	**gap**	**gap (%)**		**Mip**		**t$_{obj}$**	**Ag$_b$**	**gap (%)**		**Mip**		**t$_{obj}$**	**t$_{idle}$**	**Ag$_b$**
D94_		Ini	DVD	IVD	nb	**t**	**(s)**		DVD	IVD	nb	**t**	**(s)**	(s)	
D94_1	90637	10.05	4.02	0.15	1	1	23	1	3.84	0.08	0	0	33	2	1
D94_2	110035	5.23	1.55	0.47	0	0	19	2	1.07	0.62	0	0	17	0	4
D94_3	66072	4.50	1.35	0.13	0	0	12	2	0.77	0.17	1	1	10	0	4
D94_4	66952	5.24	1.27	0.40	0	0	7	1	1.27	0.50	0	0	7	0	1
D94_5	70216	10.08	3.15	0.13	0	0	22	2	1.74	0.43	0	0	17	0	1
D94_6	58144	3.00	0.91	0.15	0	0	8	2	0.91	0.14	1	1	9	0	4
Average			**2.04**	**0.24**	**0.17**	**0.17**	**15.17**		**1.60**	**0.32**	**0.33**	**0.33**	**15.50**	**0.33**	

TABLE 5.3 DISUD results using D95 instances

			DISUD$_{comp}$						**DISUD$_{coop}$**						
Name	n	**gap**	**gap (%)**		**Mip**		**t$_{obj}$**	**Ag$_b$**	**gap (%)**		**Mip**		**t$_{obj}$**	**t$_{idle}$**	**Ag$_b$**
D95_		Ini	DVD	IVD	nb	**t**	**(s)**		DVD	IVD	nb	**t**	**(s)**	(s)	
D95_1	1060464	24.1	5.85	0.149	7	229	815	1	5.19	0.148	10	92	949	16	2
D95_2	1478737	11.7	3.30	0.166	10	1281	2347	2	3.48	0.157	7	106	1239	51	2
D95_3	898518	24.6	4.07	0.143	7	262	1038	4	3.14	0.150	12	2296	2976	9	1
D95_4	1216846	11.7	3.08	0.140	8	440	1264	4	2.77	0.144	7	43	843	0	1
D95_5	817904	20.3	4.55	0.149	6	44	519	2	3.77	0.148	7	280	986	6	2
D95_6	1144156	9.7	3.14	0.146	7	281	1112	1	3.06	0.169	6	612	1587	0	4
Average			**4.00**	**0.15**	**7.50**	**422.83**	**1182.50**		**3.57**	**0.15**	**8.17**	**571.50**	**1430**	**13.67**	

TABLE 5.4 DISUD results using 757 instances

			DISUD$_{comp}$						**DISUD$_{coop}$**						
Name	n	**gap**	**gap (%)**		**Mip**		**t$_{obj}$**	**Ag$_b$**	**gap (%)**		**Mip**		**t$_{obj}$**	**t$_{idle}$**	**Ag$_b$**
757_		Ini	DVD	IVD	nb	**t**	**(s)**		DVD	IVD	nb	**t**	**(s)**	(s)	
757_1	924415	2331.7	2.47	0.008	6	90	868	1	1.87	0.009	4	86	859	0	3
757_2	1271490	2332.3	3.08	0.009	6	67	1105	3	1.88	0.009	4	47	1002	0	1
757_3	1001424	2331.7	1.89	0.008	5	67	900	1	2.22	0.008	6	61	942	6	1
757_4	1307682	2331.8	2.29	0.007	6	75	880	1	1.41	0.005	4	71	993	0	1
757_5	890523	4656.6	2.94	0.009	4	41	786	1	2.44	0.005	8	76	954	0	1
757_6	1139047	2331.8	1.94	0.007	5	79	686	1	1.74	0.006	5	36	825	0	1
Average			**2.44**	**0.01**	**5.33**	**69.83**	**870.83**		**1.93**	**0.01**	**5.17**	**62.83**	**929.17**	**1.00**	

TABLE 5.5 DISUD results using 319 instances

			DISUD$_{comp}$						**DISUD$_{coop}$**						
Name	n	**gap**	**gap (%)**		**Mip**		**t$_{obj}$**	**Ag$_b$**	**gap (%)**		**Mip**		**t$_{obj}$**	**t$_{idle}$**	**Ag$_b$**
319_		Ini	DVD	IVD	nb	t	**(s)**		DVD	IVD	nb	t	**(s)**	(s)	
319_1	830774	13.2	2.54	0.106	10	65	671	1	1.97	0.118	11	198	1123	0	2
319_2	786436	2681.9	2.03	0.100	8	101	636	4	1.83	0.099	8	223	819	0	4
319_3	668387	13.3	3.11	0.101	7	93	568	1	2.76	0.099	6	431	1115	8	2
319_4	654470	2682.2	2.16	0.096	7	135	691	2	1.92	0.095	5	100	698	0	4
319_5	630544	13.5	2.80	0.096	7	55	521	1	1.84	0.099	7	155	894	0	3
319_6	638308	8.8	1.84	0.097	7	63	511	1	1.51	0.097	12	124	636	5	2
Average			**2.41**	**0.10**	**7.67**	**85.33**	**599.67**		**1.97**	**0.10**	**8.17**	**205.17**	**880.83**	**2.17**	

TABLE 5.6 DISUD results using 320 instances

			DISUD$_{comp}$						**DISUD$_{coop}$**						
Name	n	**gap**	**gap (%)**		**Mip**		**t$_{obj}$**	**Ag$_b$**	**gap (%)**		**Mip**		**t$_{obj}$**	**t$_{idle}$**	**Ag$_b$**
320_		Ini	DVD	IVD	nb	**t**	**(s)**		DVD	IVD	nb	**t**	**(s)**	(s)	
320_1	1077443	11.1	2.94	0.026	5	67	2253	2	1.89	0.026	9	106	1847	0	3
320_2	1312452	3564.8	2.06	0.024	11	87	1919	1	1.92	0.016	8	67	1803	0	1
320_3	862177	12.9	2.55	0.018	10	88	1185	1	2.03	0.020	9	80	1718	0	2
320_4	848669	5.2	1.89	0.015	10	127	1439	2	1.80	0.012	13	202	1541	0	4
320_5	688268	12.5	2.71	0.016	15	234	1668	2	2.37	0.016	10	68	1290	14	2
320_6	791992	3564.5	2.38	0.025	10	75	1466	3	1.85	0.019	16	92	1519	0	1
Average			**2.42**	**0.02**	**10.17**	**113.00**	**1655.00**		**1.98**	**0.02**	**10.83**	**102.50**	**1619.67**	**2.33**	

5.6.4 Influence of the Parameters

In this section, we study the influence of the principal parameters, i.e., of the complementary subproblems number q and the maximum number of iterations during DVD phase. We also give insight into the influence of the initial solution. We present results of the influence of these parameters on the 320_1 test. This choice is directed by the fact that 320 derived instances are sufficiently difficult and DISUD is designed to solve large SPP.

Table 5.7 gives results for the influence of initial solution throughout its gap value. In general DISUD variants perform well for good initial solutions. Lower the gap of initial solution, the lower is the time of the solution process and the DVD gap. Hence we deduce that DISUD performance increases as initial solution gap decreases.

TABLE 5.7 DISUD results using different initial points for 320_1

		MISUD$_{comp}$					**MISUD$_{coop}$**					
Name	**gap**	**gap**		**Mip**		**t$_{obj}$**	**gap**		**Mip**		**t$_{obj}$**	**t$_{idle}$**
320_	Ini	DVD	IVD	nb	**t**	**(s)**	DVD	IVD	nb	**t**	**(s)**	(s)
320_1	1791.9	3.10	0.021	11	154	2347	1.91	0.010	10	225	2014	0
320_1	11.1	2.94	0.025	5	67	2253	1.64	0.011	9	359	2065	0
320_1	8.8	2.10	0.021	10	176	2072	1.91	0.010	8	311	2230	0
320_1	6.5	2.22	0.022	11	110	1783	1.76	0.010	5	184	1860	0
320_1	4.8	1.79	0.019	14	104	1536	1.60	0.009	7	201	1738	0

Table 5.8 gives results for the influence of the number of iterations, IterMax, during the DVD phase. This parameter controls the duration of the DVD phase. Higher the IterMax value, the higher is the DVD time. We deduce that the DISUD performance increses with the IterMax value. Indeed in general, DVD gap , IVD gap and computing time decrease with the IterMax value. This is explained by the fact that it is likely to get good DVD solution quality as the number of iterations during DVD phase increases.

TABLE 5.8 DISUD results using different iterMax for 320_1

		MISUD$_{comp}$						**MISUD$_{coop}$**						
IterMax	**gap$_0$**	**gap$_f$**		**Mip**		**t$_{obj}$**	**Ag$_b$**	**gap$_f$**		**Mip**		**t$_{obj}$**	**t$_{idle}$**	**Ag$_b$**
		DVD	IVD	nb	**t**	**(s)**		DVD	IVD	nb	**t**	**(s)**	(s)	
4	11.1	3.20	0.024	9	66	1147	3	1.75	0.008	6	74	1930	0	3
6		2.96	0.017	10	89	1418	3	1.86	0.009	7	204	1601	5	2
8		2.94	0.020	12	73	1315	2	1.76	0.009	10	143	1543	0	1
10		2.94	0.022	14	77	1266	1	1.88	0.001	11	362	2261	0	3
12		2.94	0.030	9	57	1273	3	1.64	0.007	6	65	1075	0	1

Figure 5.6 shows the influence of the number of complementary subproblems ,q, on the evolution of DISUD during DVD phase. We choose the values of q so that they are a power of 2 (q =1, 2,4, 8,16). We deduce that we got good performance for both q =4 or 8. This explained by the fact that for lower q (1 and 2) we still solve large complementary subproblems than those obtained with q=4 or 8. For high values (q=16) we got poor performance as it becomes difficult to find descent direction as more variables are ousted by the DVD decomposition process. In addition, there are more processes (16) than processors (8) which leads to the overload phenomen as we used computers with 8 processors.
Therefore, we used the values q =4 , iterMax=10 for the global results given in sections 5.6.3 and 5.6.5

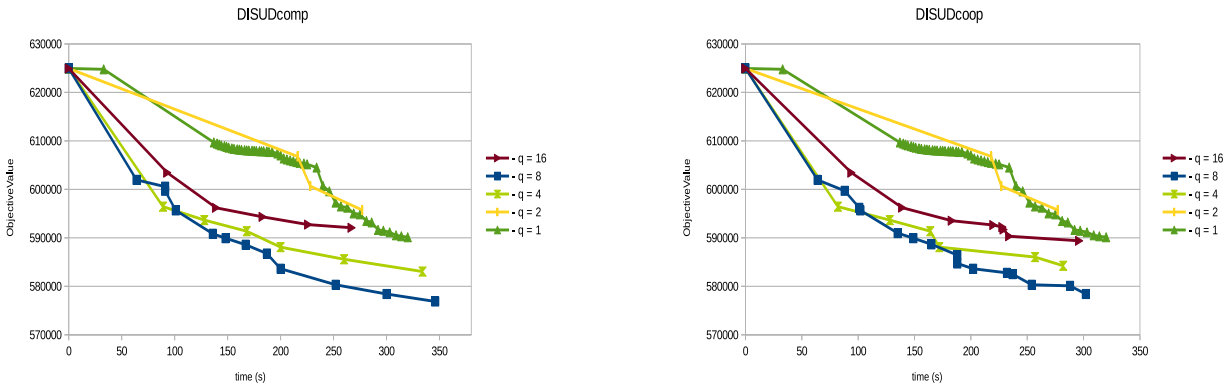


FIGURE 5.6 influence of q during DVD phase

5.6.5 DISUD vs DCPLEX Results

The goal in this part is to compare the performances of DISUD$_{coop}$, DISUD$_{comp}$ and DCPLEX using our set of tests. Figure 5.7 shows the gap value evolution over time for DISUD variants and DCPLEX on the D95_4 test. The figure clearly show that the DISUD variants outperform DCPLEX. The DISUD curves decrease more sharply than the DCPLEX one. DISUD$_{coop}$ is the fastest, yielding improvement of 280% over DCPLEX on this instance. In addition, the number of solutions found through the process resolution by DISUD is great compared to DCPLEX.

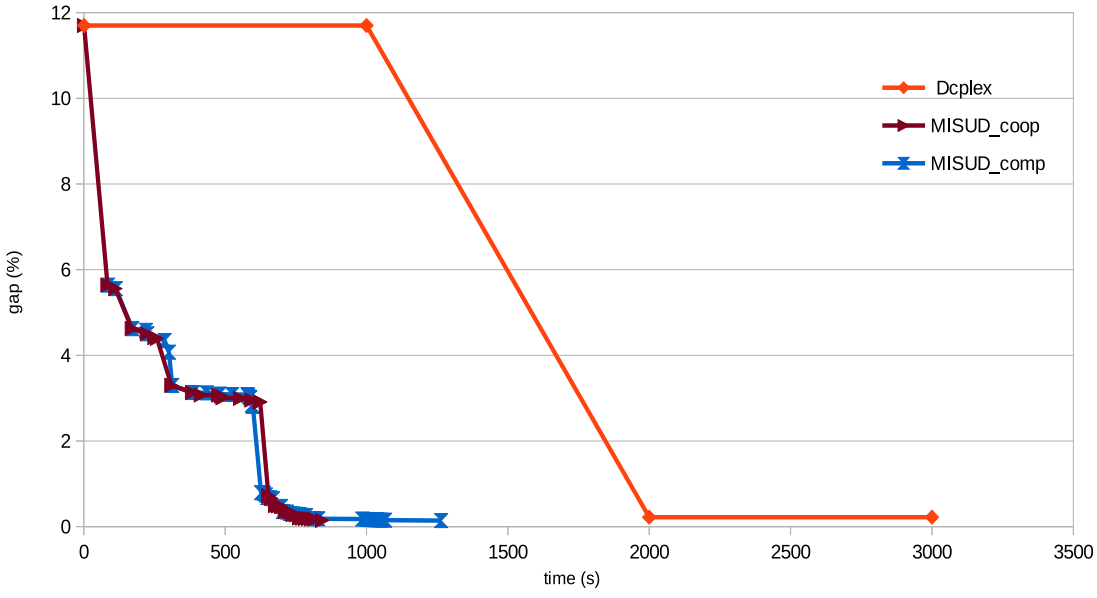


FIGURE 5.7 DISUD and DCPLEX Evolution over time on D95_4 instance

The set of tables 5.9- 5.12 shows DCPLEX results and performance comparison with DISUD variants. They report For DCPLEX, the time to obtain the best solution t_{obj}, the gap value and the number of solution found throughout the solution process. Then for each DISUD variant the time needed to outperform the last DCPLEX objective value : t_{cplex}^{DISUD}, the number of solution found during the resolution and improvement realized.
For D94 derived tests, we observe that DCPLEX was able to get good quality solution in less time than did DISUD.
For D95, 757,319 and 320 derived tests, we observe that DCPLEX was unable to improve the initial solution for 40% of instances within a time limit of an hour whereas bothDISUD variants solve 87% instances to near optimality (less than 1%) within half an hour and all instances within Three quarters of an hour. Furtehrmore DCPLEX was unable to improve any test of the 757 serie and 50% of the 319 derived tests.
We note also that DISUD algorithm find large number of different integer solutions than DCPLEX throughout the solution process : the ratio is between 5 and 19 times. We mention that this property is appreciable when solving the optimization problems because it permits to get an overview of the resolution process and to stop it once a satisfactory solution is found.
We conclude that based on our tests, DISUD outperforms DCPLEX in terms of the solution quality and the resolution time for large instances. On the other hand, DCPLEX is more efficient on small instances.

TABLE 5.9 DCPLEX and DISUD comparison on D94 instances

Instance		**DCPLEX**			**DISUD$_{comp}$**				**DISUD$_{coop}$**			
Name	**gap$_0$**	**t$_{obj}$**	**gap$_f$**	**nSol**	**gap$_f$**	**t$_{cplex}^{DISUD}$**	**nSol**	Imp	**gap$_f$**	**t$_{cplex}^{DISUD}$**	**nSol**	Imp
	(%)	**(s)**	**(%)**	**(%)**	**(%)**	**(s)**			**(%)**	**(s)**		**(%)**
D94_1	10.05	7	0.08	7	0.15	-	20	-	0.08	-	15	-
D94_2	5.23	13	0.10	2	0.47	-	11	-	0.62	-	12	-
D94_3	4.50	5	0.05	4	0.13	-	11	-	0.17	-	4	-
D94_4	5.24	7	0.09	9	0.40	-	9	-	0.50	-	5	-
D94_5	10.08	8	0.07	12	0.13	-	17	-	0.43	-	9	-
D94_6	3.00	5	0.06	5	0.15	-	14	-	0.14	-	8	-

TABLE 5.10 DCPLEX and DISUD comparison on D95 instances

Instance		**DCPLEX**			**DISUD**$_{comp}$				**DISUD**$_{coop}$			
Name	**gap**$_0$	$\mathbf{t}_{obj}$	**gap**$_f$	**nSol**	**gap**$_f$	$\mathbf{t}_{cplex}^{DISUD}$	**nSol**	Imp	**gap**$_f$	$\mathbf{t}_{cplex}^{DISUD}$	**nSol**	Imp
	(%)	(s)	(%)	(%)	(%)	(s)			(%)	(s)		(%)
D95_1	24.1	2600	0.20	6	0.149	623	34	417.34	0.148	686	40	379.01
D95_2	11.7	2890	0.27	5	0.166	1024	33	282.33	0.157	793	35	364.44
D95_3	24.6	2500	0.27	5	0.143	478	42	523.01	0.150	719	52	347.71
D95_4	11.7	2000	0.22	3	0.140	704	44	284.09	0.144	711	32	281.29
D95_5	20.3	1856	0.22	7	0.149	331	30	560.73	0.148	673	41	275.78
D95_6	9.7	3234	0.41	10	0.146	896	52	360.94	0.169	905	36	357.35

TABLE 5.11 DCPLEX and DISUD comparison on 319 instances

Instance		**DCPLEX**			**DISUD**$_{comp}$				**DISUD**$_{coop}$			
Name	**gap**$_0$	$\mathbf{t}_{obj}$	**gap**$_f$	**nSol**	**gap**$_f$	$\mathbf{t}_{cplex}^{DISUD}$	**nSol**	Imp	**gap**$_f$	$\mathbf{t}_{cplex}^{DISUD}$	**nSol**	Imp
	(%)	(s)	(%)	(%)	(%)	(s)			(%)	(s)		(%)
319_1	13.2	3019	0.17	6	0.106	550	35	548.91	0.118	678	55	445.28
319_2	2681.9	3247	0.20	5	0.100	421	22	771.26	0.099	544	57	596.88
319_3	13.3	3600	-	-	0.101	-	30	-	0.099	-	58	-
319_4	2682.2	3600	-	-	0.096	-	35	-	0.095	-	38	-
319_5	13.5	2030	0.25	4	0.096	312	48	650.64	0.099	437	51	464.53
319_6	8.8	3600	-	-	0.097	-	38	-	0.097	-	48	-

TABLE 5.12 DCPLEX and DISUD comparison on 320 instances

Instance		**DCPLEX**			**DISUD**$_{comp}$				**DISUD**$_{coop}$			
Name	**gap**$_0$	$\mathbf{t}_{obj}$	**gap**$_f$	**nSol**	**gap**$_f$	$\mathbf{t}_{cplex}^{DISUD}$	**nSol**	Imp	**gap**$_f$	$\mathbf{t}_{cplex}^{DISUD}$	**nSol**	Imp
	(%)	(s)	(%)	(%)	(%)	(s)			(%)	(s)		(%)
320_1	11.1	3600	-	-	0.026	-	43	-	0.026	-	67	-
320_2	3564.8	3243	1.534	5	0.024	1010	68	321	0.016	915	55	354
320_3	12.9	1887	0.025	6	0.018	1018	37	185.36	0.020	1408	66	134.02
320_4	5.2	2387	0.018	6	0.015	1305	54	182.91	0.012	1271	44	187.80
320_5	12.5	2323	0.016	6	0.016	1668	57	139.27	0.016	1290	54	180.07
320_6	3564	2053	0.036	10	0.025	1402	24	146.43	0.019	1170	71	175.47

5.7 Conclusion

We proposed a distributed version of ISUD. It is a multi-agent based algorithm dedicated to find descent directions leading to improved integer solutions. We presented and implemented two DISUD variants and discussed their performance. They differ in the strategy used to

manage network agents. We showed that our algorithm yields better results than the distributed version of CPLEX on a set of instances derived of industrial aircrew scheduling. Our tests set contains large-scale instances with up to almost 2,000 flights and 1,300,000 pairings. They are given as set-partitioning problems associated with initial solutions. We demonstrated that the cooperative strategy gives good results and shows good potentiel. DISUD was able to find near optimal solutions for all large instances in less time than that required by DCPLEX. DISUD realized a time efficiency ratio between 150% and 770%. More, DCPLEX is unable to produce solutions as good as those that DISUD produce within the same time limit.

Future research on network agents management strategies should be done to further improve the DISUD performance. In addition, other agents could be added also to the network. In addition, combining DISUD with heuristics that produce good initial solutions should significantly lead to obtain good DISUD performances.

CHAPITRE 6 ARTICLE 3: DISTRIBUTED INTEGRAL COLUMN GENERATION

Omar. Foutlane [a,b] Issmail. El Hallaoui [a,b] Pierre. Hansen [a,c]

Paper submitted to RAIRO-Operations Research

[a] *GERAD, Montréal (Québec), Canada, H3T 2A7*

[b] *Department of Mathematics and Industrial Engineering, Polytechnique Montréal (Québec) Canada, H3C 3A7*

[c] *Department of Decision Sciences, HEC Montréal, Montréal (Québec), Canada, H3T 2A7*

Abstract

The Integral Simplex Using Decomposition (ISUD) algorithm has been developed recently to solve large set partitioning problems (SPPs) in a primal way, i.e., moving from an integer solution to an improved adjacent one until optimality is reached. More recent works intended to enlarge its applications and to increase its performances. We cite namely the distribution version of ISUD called DISUD which implements the multi-agent system approach. In this work, we develop a distributed integral column generation (DICG) algorithm that extends DISUD to the column generation context in order to solve practical vehicle and crew scheduling problems. The computational tests on large bus drivers scheduling and aircrew pairing problems show that DICG gets good results and outperforms a distributed version of the well-known restricted master heuristic (RMH). DICG yields optimal or near optimal solutions in less than one hour.

Keywords : Set partitioning problems, Integral Simplex Using Decomposition, multi-agent systems, column generation.

6.1 Introduction

Column generation (CG) is closely connected to Dantzig–Wolfe decomposition which is introduced by Dantzig & Wolfe (1960). It is widely used to solve industrial optimization problems. It involves reformulating the problem as a restricted master problem RMP and one or more column generation subproblems CGSPs. The RMP has as few variables as possible. New variables are added to the RMP as long as the solution process continues. At each iteration, the RMP is solved to get a pair of primal and dual solutions. Then, duals are used in the CG subproblem to determine if there are any columns that can improve the RMP current solution as shown on Figure 6.1. The algorithm stops when no negative reduced-cost columns are generated and consequently the RMP solution is also optimal for the linear relaxation of the original problem. In integer optimization problems, column generation is usually embedded in a branch and bound procedure to get an integer solution. The resulting method is known as Branch and Price (see Desrochers & Soumis 1989, Barnhart et al. 1998, Desaulniers et al. 1997, Gamache et al. 1999).

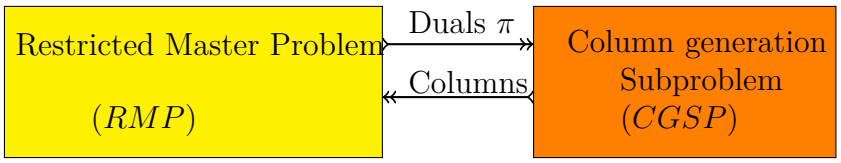


FIGURE 6.1 Column generation process

In vehicle routing and crew scheduling optimization problems, the RMP is often the well-known set partitioning problem SPP. The latter can be defined using the following scheduling terminology. A set partitioning constraint is associated with a *task* (for example, a flight leg or a bus trip to be accomplished by a *pilot or a bus driver*). Let $T = \{1, 2, ..., m\}$ be the set of tasks and $J = \{1, 2, ..., n\}$ the set of feasible schedules. With each schedule, we associate a variable x_j, a cost c_j and a column $Aj = (a_{tj})_{t \in T}$ where a_{tj} takes value 1 if Aj covers task t and 0 otherwise. The matrix $A = [A_1, A_2, ..., A_n]$ is a binary matrix. Then, the set partitioning problem formulation is :

$$(SPP) \quad \begin{aligned} &Minimize && \sum_{j \in J} c_j x_j && (6.1) \\ &subject\ to && && \\ & && \sum_{j \in J} a_{tj} x_j = 1, \forall t \in T && (6.2) \\ & && x_j \in \{0, 1\}, \forall j \in J && (6.3) \end{aligned}$$

The objective function (6.1) seeks to minimize the total cost. The set partitioning constraints (6.2) ensure that each task is covered exactly once. Constraints (6.3) impose integrality on the x_j variables. SPP is NP-hard (see Garey & Johnson 1979). There are many heuristic and exact algorithms devoted to solve it. The most used method is the famous branch and cut (Hoffman & Padberg 1993, Desaulniers et al. 1997). However, this method becomes inefficient and takes a huge time to reach an optimal solution for large instances due to degeneracy and the size of the branching tree.

Recently, Zaghrouti et al. (2014) proposed the Integral Simplex Using Decomposition algorithm (ISUD), which is based on the Improved Primal Simplex algorithm (IPS) (see El Hallaoui et al. 2011), to solve SPP. At each iteration ISUD decomposes the original problem into two subproblems. The first, called the reduced problem (RP), which only considers columns that are *compatible with the current solution, i.e., columns that belong to the vector subspace generated by the current solution columns. The second, called the Complementary Problem (CP), contains only the columns that are incompatible with the current solution. Its main role is to find a descent direction to improve the current solution of RP. ISUD stops when neither the reduced problem nor the complementary problem can improve the current solution.* Their results show that ISUD deals more efficiently with degeneracy and is able to solve large problems that are up to 570000 variables and 1600 constraints.

Since then, several improvements have been added to the initial version of ISUD namely those of Rosat et al. (2016, 2017a) and Zaghrouti et al. (2013). Rosat et al. (2016) studied the cone of the CP directions and proposed different formulas for the normalization constraint in order to favor the integrality of the descent direction found by the CP. In addition, Rosat et al. (2017a) proposed to add cuts to ISUD and concluded that this technique is costly in computing time especially for large instances. In the meanwhile, Zaghrouti et al. (2013) developed the Zoom algorithm which explores a neighborhood of the fractional solution, when it is not possible to find an improving "integer" direction, rather than exploring a branching tree as it is the case in the ISUD first version, see Section 6.2 for more details.

As the current trend in computer science is to produce multicore processors and to design parallel algorithms, Foutlane et al. (2017) developed the Integral Simplex Using Double Decomposition algorithm (ISU2D). It is a parallel algorithm based on ISUD. At each iteration, ISU2D groups the columns of the current solution into clusters in order to decompose the CP into independent complementary subproblems using the notion of compatibility defined just above. The set of complementary subproblems are solved in parallel to improve the current solution by combining the returned descent directions. ISU2D reduces the computing time of ISUD by a factor of 3 to 4 for the tested instances. ISU2D is then generalized and improved

in Foutlane et al. 2018. They use the multi-agent system approach (MAS) to introduce a general framework for a distributed version of ISU2D called DISUD. It enables to use different decompositions simultaneously. Tests of DISUD on aircrew scheduling problems show that DISUD is better than DCPLEX, the distributed version of the state of the art commercial solver CPLEX. DISUD achieves better quality solutions than DCPLEX and reduces the computing time by an average factor of 4 to 5 for test instances.

In this paper, we introduce a distributed integral column generation (DICG) that combines column generation and DISUD to solve very large scale practical SPP instances. We summarize below the most important contributions of the paper :

— DICG introduces a flexible framework using a multi-agent system to parallelize the integral column generation approach where, at each iteration, we improve the current integer solution until satisfaction. Each agent finds multiple descent directions in parallel and zooms around these directions to improve the current solution more significantly. This introduces a new level in the conventional column generation method.
— We compare two versions, one competitive and another cooperative, where agents compete or cooperate. In both, they exploit the information gathered during the solution process to improve the current integer solution.
— Tests on bus crew scheduling and real crew pairing instances from the transport industry, with up to 2000 tasks (bus trips, flights) and millions of variables, show the effectiveness of DICG. We succeed to compute excellent quality solutions (gap less than 1%) for all instances.

The remainder of this paper is organized as follows. Section 6.2 presents briefly some useful notions on the decomposition basics and the main parts of ISUD versions. Section 6.3 describes the new algorithm DICG and provides a detailed algorithmic and analysis of its components. In Section 6.4, we discuss computational results and the effectiveness of our algorithm. Finally, we end this paper with some concluding remarks and suggestions for future research in Section 6.5.

6.2 Preliminaries

In this section, we provide the basic notions of ISUD in addition to its main improved variants. These improvements are used in DICG to solve very large SPPs more efficiently.

6.2.1 Decomposition Basics

Given an integer solution $\bar{x}$ to SPP, let P_{int} be the index set of its positive components, i.e., $P_{int} = supp(\bar{x}) = \{j \in J : \bar{x}_j = 1\}$. SPP could be decomposed into a reduced problem RP and a complementary problem using the following definition of compatibility (El Hallaoui et al., 2011) :

Definition 6.2.1. *Given P an index set of some linearly independent columns containing at least P_{int}, a subset S of J is said to be compatible with P, or simply compatible, if there exist two vectors $v \in \mathbb{R}_+^{|S|}$ and $\lambda \in \mathbb{R}^p$ such that $\sum_{j\in S} v_j A_j = \sum_{l\in P} \lambda_l A_l$. The combination of columns, possibly a singleton, $\sum_{j\in S} v_j A_j$ is also said to be compatible. S is said to be minimal if any strict subset of it is incompatible.*

We note that the incompatibility degree of a column A_j towards a given integer solution is a measure that represents a distance of A_j from the current solution. Let C and I be the index sets of the compatible and incompatible columns respectively. They form a partition of J, i.e., $J = C \cup I, C \cap I = \emptyset$. The restriction of SPP to compatible columns only defines the reduced problem (RP) as follows :

$$
\begin{aligned}
& & Minimize \quad & c_C \cdot x_C && (6.4)\\
(RP) \quad & & subject\ to \quad & A_C x_C = e && (6.5)\\
& & & x_C \in \{0,1\}^{|C|} && (6.6)
\end{aligned}
$$

When $P = P_{int}$, a pivot on any compatible column with a negative reduced cost leads to an improved integer solution according to Zaghrouti et al. (2014). Moreover, if x_C^* is an optimal solution to RP, $\bar{x} = (x_C^*, 0)$ will be a solution to SPP.
Similarly, we define the Complementary Problem (CP) as follows :

$$
\begin{aligned}
& Minimize \quad & z^{CP} &= \sum_{j\in I} c_j v_j - \sum_{l\in P} c_l \lambda_l && (6.7)\\
(CP) \quad & s.t. \quad & \sum_{j\in I} A_j v_j - \sum_{l\in P} A_l \lambda_l &= 0 && (6.8)\\
& & e \cdot v_I &= 1 && (6.9)\\
& & v_j &\geq 0\ , j \in I && (6.10)
\end{aligned}
$$

In fact, the goal of the CP is to find a subset of incompatible columns to replace a subset of the current solution columns, i.e., from $supp(\bar{x})$. More precisely, we look for a convex

combination of incompatible columns that is compatible and has a negative reduced cost.

Zaghrouti et al. (2014) show that $\bar{x}$ is an optimal solution to SPP when the CP is infeasible or $z^{CP} \geq 0$, i.e., the objective value of the CP is nonnegative. In the other case, the CP returns a descent direction $d = (v, -\lambda, 0)$. In this case, let $S^+ = \{j \in I : v_j > 0\}$ and $S^- = \{l \in P, \lambda_l > 0\}$ be the sets of entering and leaving variables respectively. When the columns A_j, $j \in S^+$ are pairwise row-disjoint, i.e., they do not cover the same constraints, and $S^- \subset P_{int}$, we get an *integer* descent direction leading to an improved integer solution. Moreover, S^+ is shown to be minimal by El Hallaoui et al. (2011), i.e., non-decomposable using the terminology of Balas & Padberg (1975). This means that pivoting on variables indexed by S^+ leads to an adjacent extreme integer point with better cost value. The direction d is said to be fractional when columns A_j, $j \in S^+$ are not pairwise row-disjoint. In this case, Zaghrouti et al. (2014) proposed a branching scheme to eliminate the non-disjoint solutions when solving the CP using a diving branching strategy to get an integer descent direction.

In short, ISUD is a two-stage algorithm : at each iteration, it looks first for an improved integer solution by using the RP and second by solving the CP. The algorithm stops when both cannot improve the current solution. We discuss the improvements that have been made to ISUD in the next subsection.

6.2.2 ISUD Improvements and Versions

ISUD has been the subject of intensive research to improve it. Rosat et al. (2016) replaced the normalization constraint (6.9) by the constraint $w \cdot v_I = 1$ and studied the influence of the weight vector w on the integrality of the descent directions returned by the CP. It is obvious that when $w = e$, we obtain the classical normalization constraint (6.9). They proposed new formulas to compute the weight w_j based on the number of tasks covered by the column A_j and the degree of its incompatibility.

Zaghrouti et al. (2013) have proposed a "Zoom" version to avoid implementing a complex and exhaustive branching when the complementary problem returns a fractional descent direction. They proposed to zoom around this "fractional" direction instead of branching in the classical CP. Indeed, a set of columns compatible with the fractional direction is solved by the reduced problem to get an improved integer solution. The main steps of Zoom, as reported in Foutlane et al. (2018), are provided below :

Step 1 : *Find a good heuristic initial solution* x_0 *and set* $\bar{x} = x_0$, $P = P_{int}$, $d = 0$.

Step 2 : *Find a better integer solution around* d *:*

— *Increase* P *: set* $P = P \cup \{j : d_j > 0\}$.

— *Construct and solve RP.*

— *Update $\bar{x}$ and P : if $\bar{x}$ is improved, set $P = P_{int}$.*

Step 3 : *Get a descent direction d :*

— *Solve the CP to get a descent direction d.*

— *If no descent direction can be found or $|z^{CP}|$ is small enough then stop : the current solution is optimal or near optimal.*

— *Otherwise, go to Step 2.*

Thus, when the direction is fractional, they construct RP around this direction as explained above and solve it by a MIP solver.

In addition, there are other equivalent formulations of the complementary problem. Let $A_P = \begin{pmatrix} A_P^1 \\ A_P^2 \end{pmatrix}$ be a submatrix of A composed of columns indexed by P where A_P^1 is without loss of generality composed of the first $|P|$ linearly independent rows.

Similarly, let $A_I = \begin{pmatrix} A_I^1 \\ A_I^2 \end{pmatrix} = (a_{ij})_{\substack{1 \leq i \leq m \\ j \in I}}$ be a submatrix of A composed of incompatible columns indexed by I with A_I^1 a $|P| \times |I|$ matrix. We thus obtain an equivalent model involving only incompatible variables. In fact, constraint (6.8) could be written as :

$$\begin{pmatrix} A_I^1 \\ A_I^2 \end{pmatrix} v = \begin{pmatrix} A_P^1 \\ A_P^2 \end{pmatrix} \lambda$$

Observe that A_P^1 is invertible, so $\lambda = (A_P^1)^{-1} A_I^1 v$ and consequently the variables λ could be replaced. This results in the following CP formulation :

$$(CP) \qquad z_P^{CP} = \min_v \quad \left(c_I^\top - c_P^\top \left(A_P^1 \right)^{-1} A_I^1 \right) v \tag{6.11}$$

$$\text{s.t.} \qquad \left(A_P^2 \left(A_P^1 \right)^{-1} A_I^1 - A_I^2 \right) v = \mathbf{0} \tag{6.12}$$

$$w \cdot v = 1 \tag{6.13}$$

$$v \geq \mathbf{0}. \tag{6.14}$$

Consequently, we can use the matrix $M = (A_P^2 (A_P^1)^{-1}, -I_{|P|})$ to measure the incompatibility of A_j column by $\|MA_j\|_1$. We mention that $I_{|P|}$ is the $|P| \times |P|$ identity matrix. This measure is equal to 0 for compatible columns and positive for incompatible ones. The constraint (6.12) can be rewritten simply as : $MA_I v = 0$.

Foutlane et al. (2017) presented the principle of dynamic decomposition. They proposed

ISU2D which finds in parallel orthogonal descent directions leading to an integer solution with a larger improvement. The approach splits the complementary problem into a set of subproblems CSP_k, defined below, where $I_k \subset I$ and $\bar{c} = \left(c_I^\top - c_P^\top \left(A_P^1\right)^{-1} A_I^1\right)$. These subproblems are then solved in parallel.

$$\min_v \sum_{j \in I_k} \bar{c}_j v_j \tag{6.15}$$

$$CSP_k \qquad MA_{I_k} v_k = 0 \tag{6.16}$$

$$\sum_{j \in I_k} w_j v_j = 1 \tag{6.17}$$

$$v_j \in \{0,1\} \qquad \forall j \in I_k \tag{6.18}$$

To do so, they construct a graph G(V,E), where the columns of the current solution are represented by the vertices. Then, they use a scoring function to calculate the weight $w(v_1,v_2)$ for each edge $(v_1,v_2) \in$ E based on columns and problem information. They obtain a partition of the vertices of G into some clusters that minimizes the cut ; in other words, a partition of P into a certain number of subsets $P_k : P = \cup P_k$, $P_k \cap P'_k = \emptyset$ if $k \neq k'$ where P_k corresponds to the k^{th} cluster. CSP_k can actually be obtained from the CP by replacing P by P_k in (6.11) - (6.14) ; see Foutlane et al. (2017) for more details.

Foutlane et al. (2018) proposed the DISUD, a distributed version of ISUD using a multi-agent system approach. They consider a network of worker agents where each agent dynamically splits the original CP using its own scoring function. So, the agent i constructs hence an RP and q complementary subproblems $(CSP_k^{[i]})_{1 \leq k \leq q}$.

6.3 DICG Algorithm

DICG is a multi-agent algorithm where a master agent coordinates a set of worker agents. Using the set of worker agents, DICG realizes multiple column generations and solves the obtained restricted master problems in parallel to get an improved integer solution. We present the worker and master agents in Sections 6.3.1 and 6.3.2 respectively.

We implemented DICG as an asynchronous algorithm in such a way that each worker does not have to wait for other agents to end their iteration to start a new iteration. Worker agents exploit the available time to improve the current solution. DICG is designed to run on more than a single machine, thus making it possible to solve large problems. DICG stops when all the agents are idle or if it reaches a limit set by the user. Such limits include a time limit, a limit on the number of iterations, a limit on the number of solutions found, or other similar

criteria.

6.3.1 Worker Agents

Worker agents realize simultaneously multiple column generations and decompositions to get an improved solution. Each worker agent starts with a warm up phase where it generates, for a certain time, a set of columns with GEN procedure using the duals sent by master agent. Each worker agent generates columns with its own parameter setting, possibly different from the parameter settings of other workers, for a limited period of time or a limited number of iterations. In other words, this phase stops when the worker generates a sufficient number of columns. After this phase, a worker agent runs DISUD using DVD or IVD on the columns generated and generates new ones as needed using GEN until the master agent asserts that a good quality solution is found. GEN, DVD and IVD procedures are briefly described in Sections 6.3.1.1, 6.3.1.2 and 6.3.1.3 respectively. More details on DVD and IVD decompositions are in Foutlane et al. (2017). An illustration of a worker agent state mode transition is given in Figure 6.2. The nontrivial modes are described in the subsequent subsections.

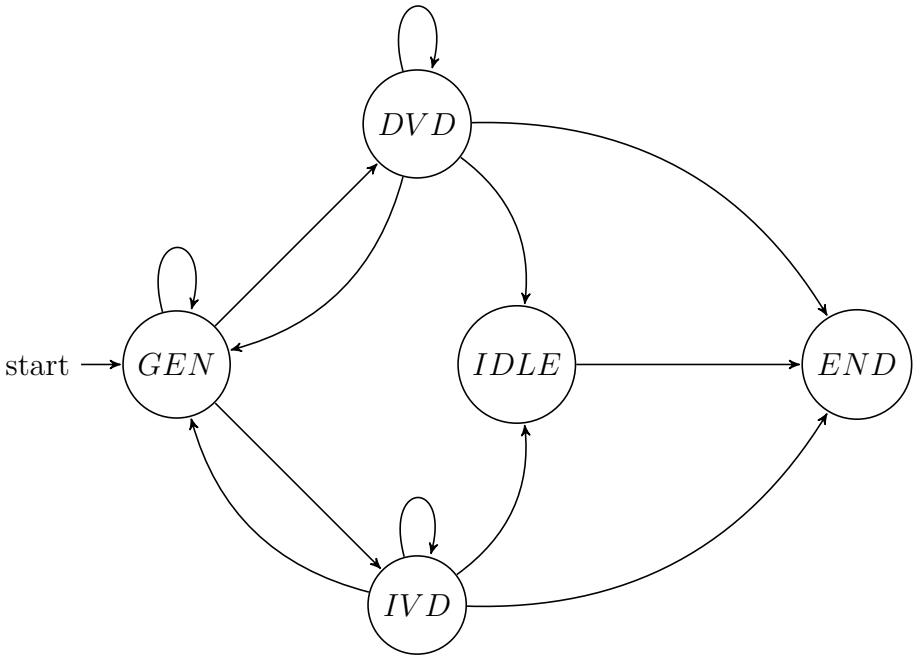


FIGURE 6.2 Worker agent basic state evolution

A worker agent behavior depends on the message received from the master as indicated in Algorithm 11 below.

Algorithm 11 Worker agent algorithm

Do
  Wait for a message from the master. In case of :
    msgGEN : Set $\bar{\pi}^{[i]} = \pi_b$, generate new columns using GEN (see Section 6.3.1.1) ;
    msgSOL : Set $\bar{x}^{[i]} = x_b$
    msgMODE-DVD : Call DVD algorihm (Algorithm 12) ;
    msgMODE-IVD : Call IVD algorihm (Algorithm 13) ;
    msgMODE-IDLE : Wait ;
    msgSTOP : Stop (do memory cleaning) ;
While (true)

We mention that throughout this paper, we use superscript [i] to denote quantities belonging to the i^{th} agent.

6.3.1.1 GEN Mode

In this mode, a worker agent generates potential columns (feasible schedules) with reduced cost sufficiently negative. The GEN procedure consists in solving the CGSP, that is actually the shortest path problem with resource constraints ($SPPRC$), using a label-setting algorithm (see Desaulniers et al. (2005)). It is basically a dynamic programming approach (generalized Dijkstra algorithm) where dominated labels are eliminated and nodes are sorted in a topological order (the networks of the CGSP used in this paper are acyclic). To generate columns, we use the dual vector π_b sent by the master agent to price out arcs of the CG networks. During the warm up phase, the dual vector is the one that the master agent finds after the solution of the linear relaxation of the restricted master problem at a given column generation iteration (see Section 6.3.2). The worker agents accumulate hence a certain number of columns to start up with in the next DVD phase. In the course of the DVD phase, π_b is exactly the one obtained by concatenating dual subvectors π_k returned by the CSP_k. After that, it is the CSP dual vector π in IVD phase. We note that π_b verifies $\bar{c}_j = c_j - \pi_b \cdot A_j = 0, \forall j \in supp(\bar{x})$, i.e., the reduced costs of these basic variables are null.

6.3.1.2 DVD Mode

During the DVD mode, the i^{th} worker agent partitions $P_{int}^{[i]} = supp(\bar{x}^{[i]})$ into q clusters, where the columns belonging to the cluster k cover a set of tasks T_k. Consequently, we have $T = \cup_{1 \leq k \leq q} T_k$ and decomposing the problem reduces to defining the partition $\tau = (T_k)_{1 \leq k \leq q}$. For the partitioning purpose, each agent constructs a weighted graph $G(V, E)$ where each column A_v, $v \in \text{supp}(\bar{x}^{[i]})$ is represented by a vertex $v \in V$. Let $(v, v') \in V^2$, $I_{vv'} = \{l \in I \ : \ A_v \cdot A_l \neq 0 \text{ and } A_{v'} \cdot A_l \neq 0\}$ and $T_{vv'}$ is the set of all tasks covered by either

A_v or $A_{v'}$. The weight of edge (v, v') measures the probability that some of the variables indexed by $I_{vv'}$ could improve the objective value if entered into the basis. The i^{th} agent uses a weighting method $we^{[i]}$ to score each edge $(v, v') \in E$. Then, it partitions the graph into q disjoint subgraphs using a min-cut algorithm (see Kernighan & Lin 1972). Finally, using its weight normalization vector $w^{[i]}$, the agent i constructs an RP and q complementary subproblems $(CSP_k^{[i]})_{1\leq k\leq q}$ formulated as follows :

$$
\begin{aligned}
& & min \quad & \sum_{j\in I_k^{[i]}} \bar{c}_j v_j & & \\
CSP_k^{[i]} & & MA_{I_k^{[i]}} v_k &= 0 & & (6.19)\\
& & \sum_{j\in I_k^{[i]}} w_j^{[i]} v_j &= 1 & & (6.20)\\
& & v_j &\in \{0,1\} \qquad & \forall j \in I_k^{[i]} \quad & (6.21)
\end{aligned}
$$

where $I_k^{[i]} \subset I$ is the subset of incompatible columns that cover only tasks in T_k. We have $I^{[i]} = \cup_{1\leq k\leq q} I_k^{[i]}$. Hence, we generalize the DVD concept mentioned in Foutlane et al. (2018) by introducing the weight normalization vector while defining the $(CSP_k^{[i]})_{1\leq k\leq q}$. Consequently, the i^{th}agent behavior is defined by the pair ($w^{[i]}$, $we^{[i]}$) as those parameters are used to construct the partition $(CSP_k^{[i]})_{1\leq k\leq q}$ and consequently the resulting pair ($\bar{x}^{[i]}$, $\pi^{[i]}$) after the resolution. As in Foutlane et al. (2018), we use the following weighting methods which propose to use the reduced cost $\bar{c}_j$, the number of tasks n_j and the incompatibility degree k_j of a column A_j :

- $we_1 : (v, v') \mapsto |\{j \in I_{vv'} \ : \ \bar{c}_j \leq 0\}|$ which scores each edge (v, v') with the number of negative reduced cost columns that $I_{vv'}$ contains. This means that it is likely to find a descent direction where there are more negative reduced cost columns.
- $we_2 : (v, v') \mapsto -\min(0, \min\{\bar{c}_j \ : \ j \in I_{vv'}\})$ that associates with the edge (v, v') the absolute value of the smallest negative reduced cost column from those indexed by $I_{vv'}$. It stipulates that a descent direction contains the least negative reduced cost column.
- $we_3 : (v, v') \mapsto w_{vv'} = -\min(0, \min\{\frac{\bar{c}_j}{n_j} \ : \ j \in I_{vv'}\})$ which takes into account the number of tasks of columns and stipulates that a good entering variable should have the smallest average negative reduced cost per task.
- $we_4 : (v, v') \mapsto w_{vv'} = -\min(0, \min\{\frac{\bar{c}_j}{k_j} \ : \ j \in I_{vv'}\})$ that scores the edge (v, v') with the reduced cost per incompatibility degree ($k_j = \|MA_j\|_1$) ratio. It is likely that small ratios would favor integrality of the descent direction.

In addition, we use the following weight vectors w_1 , w_2 and w_3 in the normalization constraint as defined in Rosat et al. (2016) :

— $w_{1j} = 1$ which was used in the first versions of ISUD and Zoom.
— $w_{2j} = k_j$, the incompatibility degree of column A_j. This favors the direction with columns having small incompatibility degree.
— $w_{3j} = n_j$, the number of tasks covered by A_j. This favors the direction with columns covering fewer tasks.

Worker agents take benefit from all of these improvements to get better solutions $\bar{x}^{[i]}$. Each of them explores a different region because it uses a different decomposition and normalization constraint. These latter impact the dual solution that in its turn impacts the columns generated. Indeed, while the classical Zoom algorithm zooms around one direction using the unit weight normalization vector, DICG looks for multiple descent directions around a multitude of orthogonal directions and using different weight normalization vectors. DICG can be hence interpreted as a multi-zooming algorithm. Algorithm 12 outlines the DVD procedure of a worker agent. An illustration is shown in Figure 6.3. Of course, the list of agents is not exhaustive and other agents could be added easily using this framework.

Algorithm 12 DVD pseudocode for agent i

Build $\tau^{[i]}$ and consequently $CSP_k^{[i]}$, $k \in \{1 \dots q\}$ using $we^{[i]}$ and $w^{[i]}$.
Solve in parallel the $CSP_k^{[i]}$, $k \in \{1 \dots q\}$.
For $k = 1$ to q
    IF d_k is integer (d^k is the direction returned by $CSP_k^{[i]}$) THEN
        Set $\bar{x}^{[i]} = \bar{x}^{[i]} + d^k$.
    ELSE
        Set $P^{[i]} = P^{[i]} \cup \{j : d_j^k > 0\}$
    ENDIF
End For.
IF some d_k is fractional, construct RP according to $P^{[i]}$ and solve it by a MIP solver.
Send the resulting $\bar{x}^{[i]}$ and duals $\pi_k^{[i]}$, $k \in 1..q$ to the master agent.

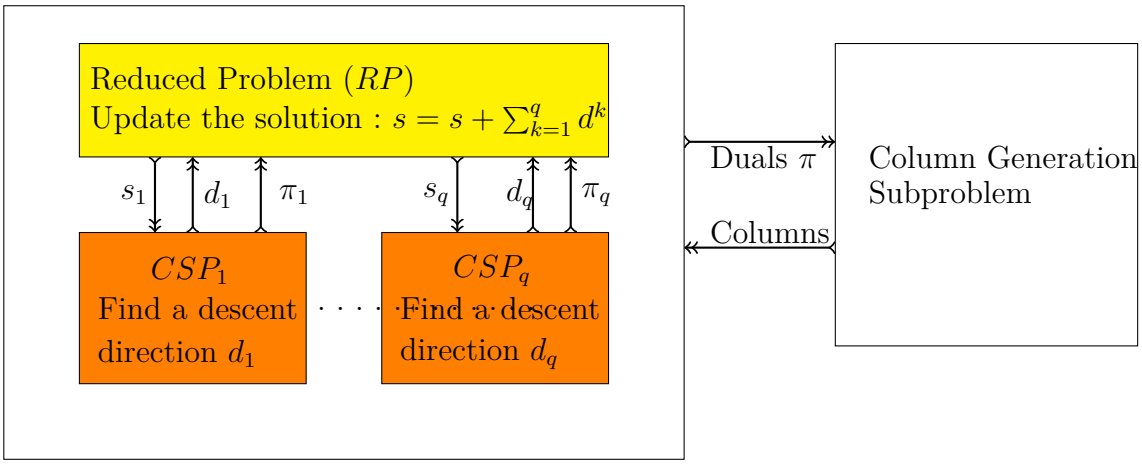


FIGURE 6.3 Column generation in DVD mode

6.3.1.3 IVD Mode

The IVD can be seen as a neighborhood exploration policy. The idea is to partition columns of the constraint matrix A into subsets. In Foutlane et al. (2017, 2018), we used reduced cost as a pricing criterion. One may use other criteria such as the incompatibility degree. Thus, a worker agent starts by a neighborhood containing potential columns according to some chosen criteria and explores the subsets of columns incrementally. More precisely, the worker agent runs Zoom on SPP_k, a restriction of SPP to the chosen subset of columns, starting with the best solution that is returned by the master agent. Algorithm 13 provides the pseudocode of the IVD phase ; q' is a parameter tuned by experimentation. An illustration is given in Figure 6.4

Algorithm 13 IVD pseudocode for agent i

Price out the columns using a distance metric to create neighborhoods.
Sort the variables in an increasing order of the distance and reindex them.
For $k = 1$ to q'
    Build $\text{SPP}_k^{[i]}$ by considering the first $k\frac{|J|}{q'}$ variables.
    Solve $\text{SPP}_k^{[i]}$ with Zoom, set $\bar{x}^{[i]}$ to the obtained solution, and update $z_{ub}^{[i]}$.
    Send $\bar{x}^{[i]}$, calculate a dual vector $\pi^{[i]}$ and send it to the master agent.
End for.

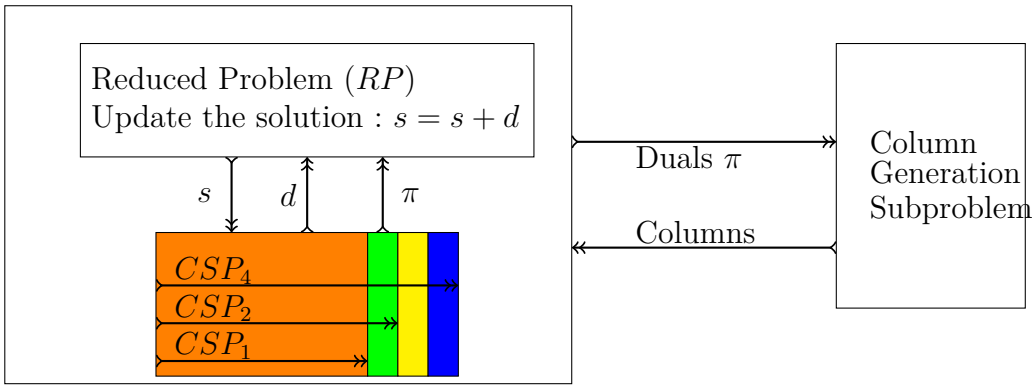


FIGURE 6.4 Column generation in IVD mode for q'=4

6.3.2 Master Agent

The master agent controls the progress of DICG. It ends it when a termination criterion is satisfied. Similarly to the work of Foutlane et al. (2018), we develop and study two variants of DICG : the cooperative variant where worker agents cooperate and the competitive variant where worker agents work independently. Pseudocode of cooperative and competitive variants are provided in Sections 6.3.2.2 and 6.3.2.1 respectively. The IN-PARALLEL and END-PARALLEL terms are used to mention that the multiple statements in between are executed in parallel. For both variants, the master agent starts with an initial primal solution x_0 of value z_0 and a dual vector π_0, sets the upper bound $z_{ub} = z_0$, sends supp(x_0) and π_0 to all agents and waits for the pairs (solution, dual) obtained by the worker agents.

The master agent solves in parallel the linear relaxation of SPP by column generation. During a certain number of iterations (IterWarm), it sends the dual solution π_b of its restricted master poblem to the workers for a warm up. After the warm up phase, when the master receives a solution $\bar{x}^{[i]}$ from the i^{th} agent that improves the DICG upper bound z_{ub}, the master agent updates z_{ub} and x_b, the best solution encountered. During the DICG execution, the master agent reacts according to which variant, competitive or cooperative, is activated. Finally, the master agent initializes the counter $nbrItr^{[i]}$ for each agent and increments it after receiving a solution from the worker agent i. The master agent uses this counter to tell the worker which mode to use, GEN, DVD or IVD mode, depending on the predefined values IterDVDMax, IterWarm and ϵ_{dvd}. In addition, this counter is used among others to stop the agent i when it reaches a predefined value $IterMax$.

6.3.2.1 DICG Competitive Agents

After the warm-up phase, which aims to generate a sufficient number of columns, the worker agents begin to solve their RMPs using DISUD. They send their solutions to the master. The latter updates the i^{th} agent's upper bound $z_{ub}^{[i]}$. The sender agent makes another DVD iteration if its iteration number does not exceed a predefined value IterDVDMax and the current solution quality is less than a predefined value ϵ_{dvd}. Otherwise, the master agent sends to the i^{th} agent the message msgMODE-IVD in order to switch to IVD iteration. An agent becomes idle when its iteration number exceeds $IterMax$. Finally, DICG stops when all worker agents become idle or some stopping criteria are met. As it can be seen, each agent works independently and does not share any information with other agents. Algorithm 14 presents the master competitive procedure.

Algorithm 14 Competitive pseudocode

Set $z_{ub} = z_0$, $x_b = x_0$, $\pi_b = \pi_0$ and for each agent i, set $nbrItr^{[i]}$ = IterWarm, $mode^{[i]} = GEN$
IN-PARALLEL
  Calculate a lower bound z_{lb} for SPP and update π_b consequently. During the first IterWarm iterations, send msgGEN and π_b to all worker agents.
  Listen to worker agents :
    On the reception of a solution $\bar{x}^{[i]}$ from some agent i, DO
      Set $nbrItr^{[i]} = nbrItr^{[i]} + 1$, $\pi_b = \pi^{[i]}$
      IF $c \cdot \bar{x}^{[i]} < z_{ub}$ THEN
        Set $z_{ub} = c \cdot \bar{x}^{[i]}$ and $x_b = \bar{x}^{[i]}$
      END IF
      IF $nbrItr^{[i]} = IterMAX$ THEN
        Send msgMODE-IDLE to agent i, set $mode^{[i]} = IDLE$
      ELSE IF $nbrItr^{[i]} \leq IterDVDMax$ AND $\frac{z_{ub} - z_{lb}}{z_{lb}} > \epsilon_{dvd}$ THEN
        Send msgGEN and π_b to agent i
        Send msgMODE-DVD to agent i
      ELSE
        Send msgGEN and π_b to agent i
        Send msgMODE-IVD to agent i
      END IF.
      IF all worker agents are IDLE or some stopping criteria are met THEN
        Send msgSTOP to all worker agents and return x_b
      END IF.
    END DO
END IN-PARALLEL

6.3.2.2 DICG Cooperative Agents

In the cooperative variant, the master agent intervenes more and changes the worker agents settings during the progress of the process as it is shown in Algorithm 15. We discuss below the most important issues. During the process execution, the communication between the master and the worker agents is bilateral : the worker agent sends its newly found solution pair (primal, dual) to the master and waits for the primal solution x_b from which it starts, the OK to stay in DVD mode or any other decision from the master. Here again, when an agent iteration number exceeds $IterMax$, its mode changes to IDLE. If all agents are idle, the master stops the process. We limit the cooperation to the exchange of the best integer solution found at the end of a column generation iteration between the worker agent and the master agent. But, we keep in mind that further cooperation policies and strategies can be made in future work to study more thoroughly this subject.

Algorithm 15 Cooperative pseudocode

Set $z_{ub} = z_0$, $x_b = x_0$, $\pi_b = \pi_0$ and for each agent i, set $nbrItr^{[i]}$ = IterWarm, $mode^{[i]} = GEN$
IN-PARALLEL
  Calculate a lower bound z_{lb} for SPP and update π_b consequently. During the first IterWarm iterations, send msgGEN and π_b to all worker agents.
  Listen to worker agents :
    On the reception of a solution $\bar{x}^{[i]}$ from some agent i, DO
      Set $nbrItr^{[i]} = nbrItr^{[i]} + 1$, $\pi_b = \pi^{[i]}$
      IF $c \cdot \bar{x}^{[i]} < z_{ub}$ THEN
        Set $z_{ub} = c \cdot \bar{x}^{[i]}$ and $x_b = \bar{x}^{[i]}$
      ELSE
        SEND msgSOL and supp(x_b) to agent i, set $x^{[i]} = x_b$
      END IF
      IF $nbrItr^{[i]} = IterMAX$ THEN
        Send msgMODE-IDLE to agent i, set $mode^{[i]} = IDLE$
      ELSE IF $nbrItr^{[i]} \leq IterDVDMax$ AND $\frac{z_{ub} - z_{lb}}{z_{lb}} > \epsilon_{dvd}$ THEN
        Send msgGEN, π_b and msgMODE-DVD to agent i
      ELSE
        Send msgGEN, π_b and msgMODE-IVD to agent i
      END IF
      IF all worker agents are IDLE or some stopping criteria are met THEN
        Send msgSTOP to all worker agents and return x_b
      END IF.
    END DO
END IN-PARALLEL

6.4 Computational Results

In this section, we present results of DICG and discuss its effectiveness. We tested DICG algorithm on crew pairing problem (CPP) and vehicle and crew scheduling problem (VCSP) instances. We compare DICG to DRMH, a distributed version of the well-known restricted master heuristic (RMH) (see Joncour et al. (2010)). DRMH consists in solving the linear relaxation of MP by column generation at the root node. Then, DCPLEX, the distributed version of CPLEX, solves the last RMP after adding integer constraints on the variables.

At the beginning, we go through characteristics of the instances composing our test benchmark. Then, we discuss the influence of the two parameters IterWarm and q (he number of complementary) on the performance of DICG. After this, we compare DICG_{comp} and DICG_{coop} performances when using the best values of these parameters. Finally, we compare the best variant of DICG to DRMH.

We implemented the two DICG variants (competitive (DICG_{comp}) and cooperative (DICG_{coop})) using C^{++} and the MPI (Message Passing Interface) library. This latter ensures communication between our agents. The master runs on a Linux PC with Quad-Core processor of 3.30 GHz and each worker agent runs on a Linux PC with 8 processors of 3.4 GHz each. Finally, we note that all induced optimization problems (CSP, RP, RMP ...) are solved using the commercial CPLEX solver version 12.6.1 while the SPPRC is solved using the Boost library version 1.55.

6.4.1 Instances Characteristics

The set of tests consists of CPP and VCSP instances, described respectively in Sections 6.4.1.1 and 6.4.1.2. Each subset contains medium and large instances. Furthermore, considering that DICG needs a pair (primal, dual) of solutions to start from, we construct an artificial initial primal solution where each task is covered by a single-task column with a large, big-M, cost and an initial dual solution where each dual value is set to this large cost value.

6.4.1.1 CPP Instances

In aircrew scheduling, a pairing is a sequence of flights that starts and ends at the same airport. CPP consists of finding a set of pairings that covers all the scheduled flights at minimum cost over the planning horizon. Moreover, each flight has to be covered by a single pairing and therefore, CPP is modeled as a SPP. In practice, the CPP is solved by branch & price method where the pairings are generated by solving subproblems modeled as SPPRCs

(see Saddoune et al. (2013)). For our tests, we use five instances derived from a real-life CPP of a major north American airline. The original datasets can be found in Kasirzadeh et al. (2017) (aircraft fleets concerned are D94, D95, 757, 319, and 320). We add the CPP prefix to the instance name to indicate that it belongs to the CPP subset. Table 6.1 presents the CPP instances characteristics. This table shows (from left to right) the name of the instance, the number of tasks (flights), the "density" (the rounded average number of flights per pairing (i.e., the number of nonzeros per column)), the (approximate) percentage of degeneracy in an optimal basis.

TABLE 6.1 Characteristics of the CPP instances

instance	nbrTasks	density	degeneracy
CPP_D94	424	8	87
CPP_D95	1255	9	88
CPP_757	1290	6	83
CPP_319	1293	7	85
CPP_320	1740	7	85

6.4.1.2 VCSP Instances

VCSP consists of assigning buses to bus trips and drivers to tasks which are defined by dividing each bus trip into segments. These latter link consecutive relief points where drivers exchange could occur. There is one task for each (bus trip) segment. We consider the single-depot homogenous-fleet variant addressed by Haase et al. (2001) and consider only set partitioning constraints. We use the random instance generator of Haase et al. (2001) to generate three test instances for the same pair (R, B) where R is the number of relief points on each bus trip and B is the number of bus trips. In fact, the size of an instance (number of tasks) is defined as the product $B \times (R + 1)$. Consequently, to name a VCSP instance, we use the acronym vcs_s_R_B. The prefix *vcs* indicates that it belongs to the VCSP subset and the incremental value x indicates the seed number of the instance. So, as an example, vcs_1_5_160 indicates that it is the second VCSP instance generated with the values s = 1, R = 5, and B =160. Table 6.2 presents the VCSP instances characteristics. The first column shows the name of the instance. Then, it reports the number of tasks (nbrTasks), the number of relief points (R), the number of bus trips (B), the density (density), and the percentage of degeneracy (degeneracy).

TABLE 6.2 Characteristics of the VCSP instances

instance	nbrTasks	R	B	density	degeneracy
vcs_s_9_80	800	10	80	12	91
vcs_s_5_160	960	6	160	9	89
vcs_s_9_160	1600	10	160	15	93
vcs_s_9_200	2000	10	200	14	93

6.4.2 Influence of the Parameters

In this section, we study the influence of the principal parameters, i.e., the number q of CSPs we solve in parallel and the number of column generation iterations (IterWarm) in the warm up phase. We present results of the influence of these parameters on the CPP_319 instance. The idea is to tune these parameters on a medium instance (leaning towards large ones) and use the tuned values for all the other instances.

First, as mentioned in Section 6.3.1.2, the behavior of an agent is defined by the scoring function we and the normalization weight vector w. The workers that we used during our tests are the following three agents :

— agent 1 defined by the pair (w_1, we_1),
— agent 2 defined by the pair (w_2, we_1), and
— agent 3 defined by the pair (w_3, we_1).

These agents use the same scoring function we_1 - as it seems to be the best for our tests - and different normalization weight vectors. They all look for descent directions in a region of the graph G where there are more negative reduced cost columns, but differ only on how the direction is composed. Indeed, the first agent favors directions having the minimum average reduced cost per entering column, the second favors entering columns with small incompatibility degree, while the third leans towards entering columns covering few tasks.

Figure 6.5 shows the influence of q on the evolution of the objective value that DICG finds over time during DVD phase. We choose the values of q so that they are power of 2 ($q = 2$, 4, 8). We deduce that we get good performance for both $q = 2$ and $q = 4$. This can be explained by the fact that for high values ($q = 8$), we get poor performance because it becomes difficult to find descent direction as more variables are ousted by the DVD decomposition process. In addition, when there are more processes, we face the overload phenomenon as we use computers with 8 processors.

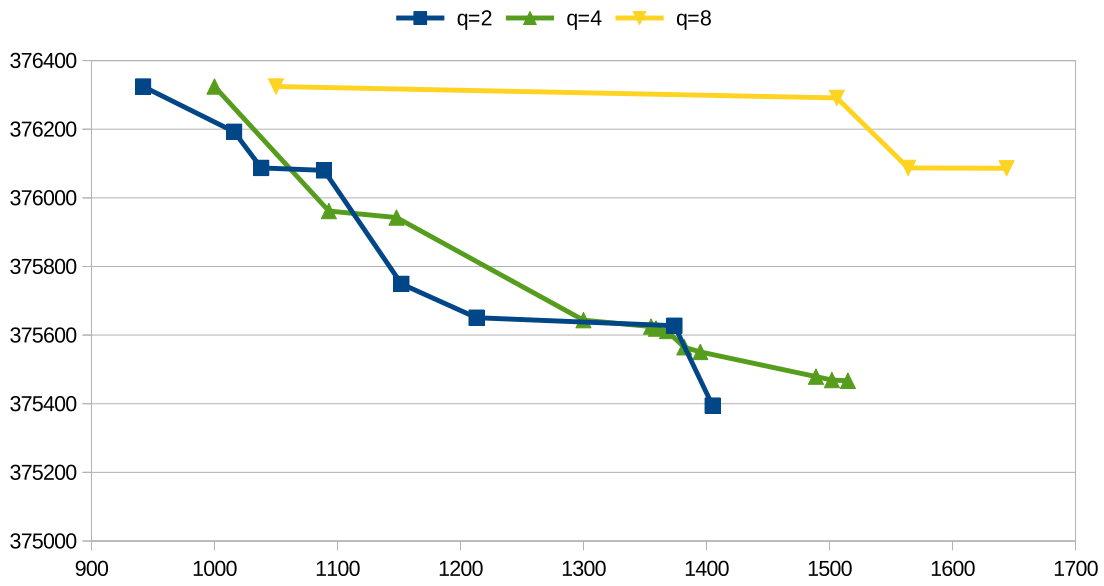


FIGURE 6.5 Influence of q on CPP_319 during the DVD phase

Similarly, Table 6.3 gives results for the influence of IterWarm. The first column shows the different tested values of IterWarm parameter. Then, for each DICG variant, it reports the total computational time in seconds (time), the optimality gap (in percentage) between the cost of the best solution found and the linear relaxation optimal value (gap), the number of column generation iterations (nbIter) and the agent that gets the best solution (Ag_b). The IterWarm parameter controls the duration of the warming phase. Indeed, higher is the IterWarm value, higher is the warming up time. From the results, we deduce that the DICG performance increases with IterWarm value. In general, DICG variants perform well. This is explained by the fact that ISUD and its variants perform better when they generate columns close to LP optimality and the integrality gap is small.

TABLE 6.3 Influence of IterWarm on CPP_319

	DICGcomp				DICGcoop			
IterWarm	time	gap (%)	nbIter	Ag_b	time	gap (%)	nbIter	Ag_b
5	5604	0.95	35	2	6786	0.62	44	3
10	3547	0.37	29	1	5399	0.29	16	2
15	2579	0.19	26	2	2879	0.23	25	1
20	5896	0.17	26	2	4148	0.20	41	3

Based on these results, we can see that we get good quality solutions in shorter computing times for IterWarm = 15. Therefore, we use the values $q = 4$ and IterWarm = 15 for the results given in sections 6.4.3 and 6.4.4. Moreover, we set $\epsilon_{dvd} = 2\%$; the threshold of 2% is inspired

by an industrial observation claiming that solutions within 2% gap are acceptable in practice (see Rosat et al. 2017b). Also, we set the other DICG parameters as follows : IterDVDMax = 20, IterMAX = 50 and the execution time limit to two hours for CPP instances and to a one hour for VCSP instances.

6.4.3 Cooperative vs Competitive Results

In this section, we show DICG_{coop} and DICG_{comp} results and discuss their performances on our set of tests. Figure 6.6 shows the evolution of the objective value over time for DICG_{comp} on the instance CPP_320, as it is the largest instance. It depicts the solutions found by each agent during the solution process.

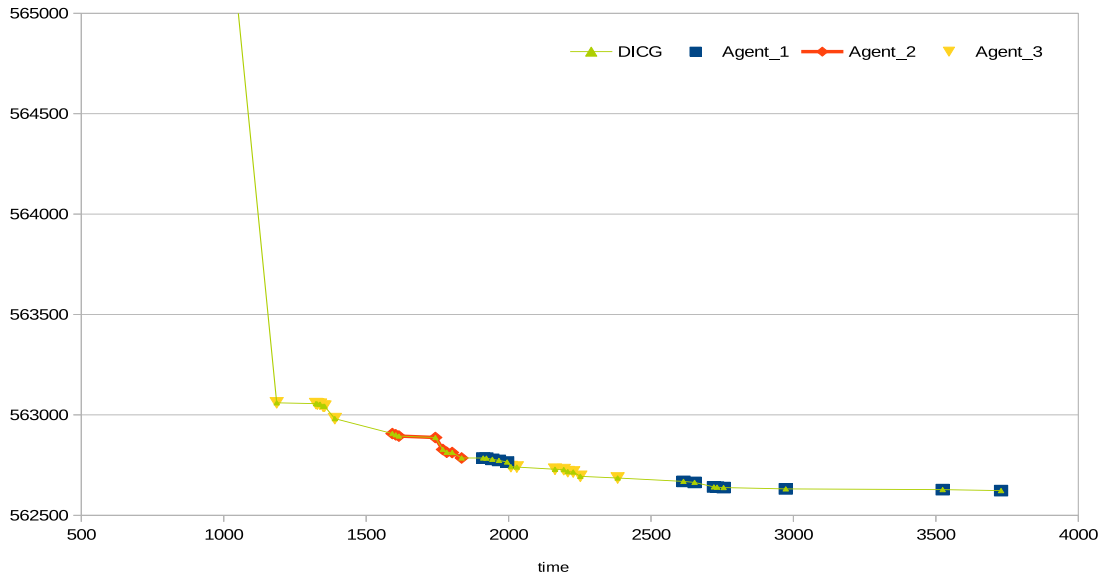


FIGURE 6.6 DICG_{comp} evolution over time on CPP_320

In addition, we can observe a rapid objective value decrease at the beginning of the solution process compared to the objective value decrease at the end. This is explained by the fact that like traditional column generation methods, it becomes difficult for DICG to generate improving directions when the solution process approaches the optimality. This behavior is typical and representative of the other instances.

Figure 6.7 shows the evolution of DICG_{coop} and DICG_{comp} on CPP_320. We connect the points to improve its readability. We note that the two curves present similar shape. DICG_{coop} is better in the middle of the solution process. This is due to the fact that DICG_{coop} embeds

the spirit of the depth first search strategy. DICG$_{coop}$ uses all its agents to explore its best solution x_b (solution with the lowest cost) neighborhood.

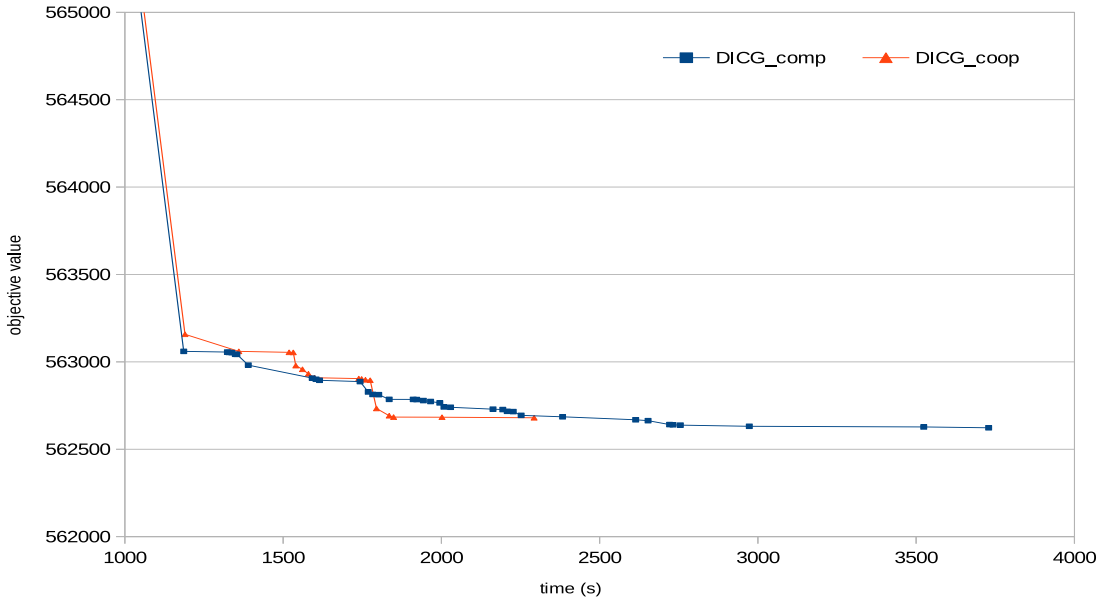


FIGURE 6.7 DICG variants evolution over time on CPP_320

Table 6.4 shows results for each variant of DICG. It reports the same information as in Table 6.3 for the columns having the same name. In addition, it presents the objective value (obj) and the number of improving integer solutions found during the solution process (nbSols).

TABLE 6.4 Results of DICG variants for the CPP instances

Instance	DICG$_{comp}$					DICG$_{coop}$				
Name	time	obj	nbIter	nbSols	Ag$_b$	time	obj	nbIter	nbSols	Ag$_b$
CPP_D94	59	109178	17	2	2	59	109178	17	2	2
CPP_D95	3296	268975	19	57	3	4580	268431	25	95	1
CPP_757	708	430807	16	5	3	1457	430671	14	19	1
CPP_319	2579	374859	26	35	3	2879	375015	25	46	1
CPP_320	3729	562623	25	36	1	2293	562680	18	17	1

We observe that both DICG variants solve all instances to near optimality within almost an hour. The differences between objective values of the two variants are less than 0.1% in all cases. Thus, the results show that both variants are excellent from industrial point of view. However, DICG_coop remains significantly better in terms of the number of integer solutions found and the overall objective value thanks to cooperation. On the other hand, concerning the computing time, DICG$_{comp}$ is faster than DICG$_{coop}$ because the marginal gain

in the last iterations is costly. In major complex applications, it worths it, especially when we have enough time for planning. Finally, it is obvious that all agents contribute to the DICG solution process as it is shown by the Ag_b column. Based on this, we deduce that considering many agents simultaneously is a better approach.

From the aforementioned results, we conclude that DICG variants yield excellent results. DICG_{coop} constitutes a good variant of DICG that shows a good potential since it allows to manage multiple agents simultaneously in order to take advantage of their cooperation and paves the way for implementing more sophisticated cooperation strategies.

6.4.4 DICG vs DRMH

The goal in this section is to compare the performance of the best variant of DICG against DRMH on CPP and VCSP instances. Figure 6.8 shows the gap value evolution over time for DICG variants and DRMH on CPP_320. The figure clearly shows that the DICG variants outperform DRMH in terms of computing time and solution quality.

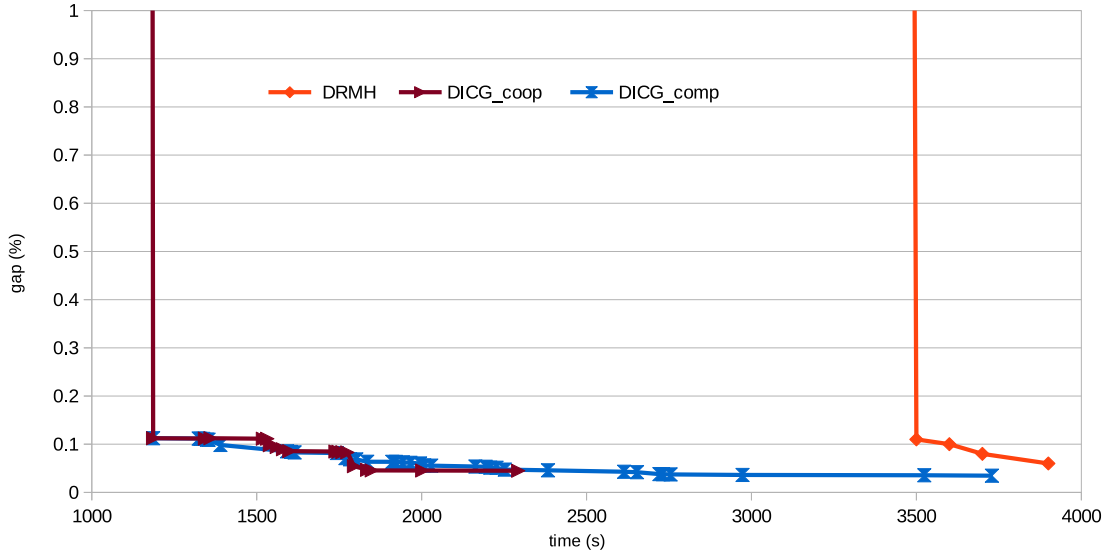


FIGURE 6.8 Gap evolution for DICG and DRMH for CPP_320

The CPP computational results are presented in table 6.5. We report (from the left to the right) the computational time, the solution gap, the total number of improving integer solutions and the best DICG variant. We observe that for the cited statistics, DICG is generally better than DRMH, especially for the number of improving integer solutions that is very desirable in practice. We would like to mention here that DRMH is good because the density is low which is an important fact.

TABLE 6.5 Results of DICG and DRMH for the CPP instances

Instance	DRMH				DIGC				
	time	gap (%)	nbrSol	nbIter	time	gap(%)	nbrSol	nbIter	bestVar
D94	81	0.07	9	22	59	0.35	2	17	coop
D95	4110	0.71	9	36	3296	0.48	95	25	coop
757	1392	0.05	4	30	1457	0.04	18	19	coop
319	6649	0.28	12	40	2879	0.23	46	25	coop
320	3950	0.05	6	42	3729	0.03	36	25	comp

The VCSP computational results are presented in Table 6.6. We report the computational time, the solution gap and the number of column generation iterations for both DRMH and DICG. We restrict the comparison of VCSP instances to the DICG_comp variant. This is dictated by the fact that the convergence of DICG is too fast that there is no need to any cooperation strategy between worker agents. For this reason again, we do not report results on VCSP instances in the previous section. For instances with medium difficulty, i.e., those with tasks' number less than 1000 and low density, DICG is two to five times faster than DRMH. While for instances with larger number of tasks' number (more than 1000) and higher density (greater than 13), DRMH is unable to find a good enough (less than 10% gap) feasible integer solution within one hour time limit.

Remark. This can be explained by the fact that in DRMH, columns that are good for the linear relaxation are not necessarily good for getting an optimal integer solution. In the opposite, we can show that at each CG iteration, DICG should succeed in generating one or more optimal columns that are missing in the current integer solution. This can also explain the fact that the DICG number of iterations is smaller than the DRMH number of iterations number.

TABLE 6.6 Results of DICG and DRMH on the VCSP instances

instance	DRMH			DICG			
	time	gap (%)	nbIter	time	gap (%)	nbIter	Ag_b
vcs_0_9_80	73	0	37	44	0	3	3
vcs_1_9_80	105	0	10	38	0	3	2
vcs_2_9_80	74	0	38	39	0	2	2
vcs_0_5_160	123	0	37	25	0	3	2
vcs_1_5_160	119	0	40	26	0	4	3
vcs_2_5_160	112	0	29	24	0	4	3
vcs_0_9_160	3600	-	-	163	0	4	3
vcs_1_9_160	3600	-	-	154	0	4	3
vcs_2_9_160	3600	-	-	151	0	4	1
vcs_0_9_200	3600	-	-	220	0	4	1
vcs_1_9_200	3600	-	-	220	0	5	1
vcs_2_9_200	3600	-	-	212	0	6	3

6.5 Conclusion

We proposed in this paper a new algorithm DICG, which is a distributed integral column generation algorithm. It is a multi-agent based algorithm dedicated to generate in parallel descent directions leading to an improved integer solution at each column generation iteration. We presented and implemented two DICG variants and discussed their performances. They differ in the strategy used to manage the worker agents. We showed that our algorithm yields good quality solutions (less than 1%), largely better than the distributed version of RMH on a set of vehicle and crew scheduling instances. Our tests set contains large-scale instances with up to almost 2000 tasks. DICG was able to find optimal or near optimal solutions for all instances in less time than DRMH, especially on hard VCSP instances.

Future research should be done to further improve the DICG performance. We believe that combining this primal algorithm DICG with meta/math/heuristics should produce better solutions in a drastically reduced time.

CHAPITRE 7 DISCUSSION GÉNÉRALE

Dans cette thèse, nous avons présenté trois travaux de recherche. L'objectif a été l'amélioration des performances d'ISUD et l'élargissement de son domaine d'application. Nous avons réalisé ceci moyennant de nouvelles décompositions intelligentes facilement parallélisables et son intégration dans le contexte de la génération de colonnes. La mise en œuvre de cette démarche a permis le développement de nouvelles méthodes parallèles à base d'ISUD et la réalisation d'une plate-forme distribuée de génération de colonnes dédiée aux problèmes de partitionnement. Ensuite, nous avons effectué nos tests sur des instances des problèmes de rotations d'équipages (chauffeurs de bus et pilotes d'avions). Les résultats obtenus montrent une nette amélioration des performances d'ISUD. Un sommaire des réalisations peut être étalé comme suit :

En premier lieu, nous avons produit l'algorithme parallèle ISU2D. Il introduit une deuxième décomposition de plus que celle qui est intrinsèque à ISUD. Nous rappelons ici que cette dernière utilise, itérativement, une décomposition du problème en deux : le problème réduit qui est non dégénéré et le problème complémentaire dont la vocation est de trouver des directions de descente. Cette décomposition s'avère être un outil efficace contre la dégénérescence qui reste un phénomène dont les méthodes de résolution du SPP souffrent. Par l'introduction d'une deuxième décomposition, ISU2D a amélioré l'efficacité d'ISUD et a diminué le temps de résolution par un facteur de 3 en moyenne.

Ensuite, nous avons développé la version distribuée, DISUD, qui généralise le principe des décompositions d'ISU2D pour permettre l'utilisation de plusieurs décompositions simultanément. En effet, l'application du système multi agents permet à DISUD de prospecter plusieurs régions de l'espace de solutions ayant du potentiel pour améliorer la solution courante. Chaque agent applique une stratégie de recherche dictée par une fonction de *scoring* qui lui permet de décomposer le problème à résoudre en une famille de sous-problèmes. L'idée est qu'il est plus probable de trouver dans l'ensemble des régions de l'espace de solutions des directions de descente qui permetteront d'obtenir une solution entière de meilleur coût.

En dernier lieu, nous effectuons l'intégration du DISUD au contexte de la génération de colonnes. Ainsi, nous obtenons un algorithme distribué de génération de colonnes (DICG). En effet, chacun des agents du DICG génère des colonnes dans une direction jugée propice. Ainsi, l'algorithme DICG génère de multiples directions de descente potentielles. Les informations échangées entre les agents et le master et les actions qui s'en suivent mettent en oeuvre des stratégies de coopération entre les agents. Cet algorithme permet d'aboutir à des solutions de bonne qualité.

Les tests que nous avons effectués sur des instances de grande taille de problèmes d'horaires de chauffeurs d'autobus et de pilotes d'avions confirment l'efficacité de ISU2D, DISUD et DICG. Pour ces instances, ISU2D réduit le temps d'exécution d'ISUD par un facteur de 3 en moyenne. En outre, il trouve une solution optimale dans quelques minutes alors que ce sont des instances considérées très difficiles pour CPLEX qui demande beaucoup plus de temps pour les résoudre. La version distribuée DISUD permet de réduire le temps de résolution de DCPLEX, la version distribuée du CPLEX, par un facteur de 5 pour certaines instances. De plus DISUD résout des instances que DCPLEX n'a pu trouver une solution entière après une heure d'exécution. Finalement, la version DICG a surpassé DRMH, la version distribuée de l'heuristique RMH, que ce soit par la rapidité ou encore la qualité des solutions trouvées. Le facteur de réduction du temps de résolution est entre 2 et 5.

Enfin, nous signalons que même si nous avons effectué nos tests sur des problèmes purs de partitionnement, notre approche devra aussi permettre, en principe, de résoudre les problèmes de partitionnement avec contraintes supplémentaires fréquemment rencontrés en industrie. Nous pensons, donc, avoir amélioré ISUD et proposé une voie pour tirer profit des nouveautés du calcul parallèle et des développements réalisés dans l'informatique comme la production de machines multiprocesseurs. Il nous reste à signaler qu'avec notre travail, plusieurs pistes de recherche sont ouvertes. En particulier, nous citons :

- l'étude des stratégies de coopération plus complexes au niveau du DISUD et DICG,
- l'introduction des méthodes d'apprentissage pour la détection des directions d'amélioration de la solution courante,
- l'élargissement des problèmes traités par ISUD pour résoudre les problèmes de partitionnement autres que les problèmes de tournées de véhicules et rotations d'équipages, comme par exemple les problèmes de classification plus généraux.

CHAPITRE 8 CONCLUSION ET RECOMMANDATIONS

Dans cette thèse, nous avons proposé des améliorations à ISUD, un nouvel algorithme efficace pour la résolution du problème de partitionnement. Ce dernier est parmi les plus étudiés dans la littérature et utilisés en pratique (tournées de véhicules/personnel, classification, ...). Nous avons présenté des algorithmes parallèles et distribués basés sur ISUD. Ainsi, nous avons profité des apports du calcul parallèle et des évolutions immenses que connaît l'informatique d'aujourd'hui. Nous avons utilisé aussi des améliorations entreprises sur ISUD comme la prise en compte des différentes formules de la contrainte de normalisation. Les algorithmes que nous avons développés ont permis d'augmenter les performances d'ISUD. Les tests que nous avons menés sur un ensemble d'instances provenant de l'industrie du transport aérien et de bus ont montré le potentiel de l'adaptation des techniques du calcul parallèle à ISUD. Les algorithmes développés permettent la résolution rapide des problèmes de planification des horaires du personnel. En effet, le temps d'exécution a été réduit d'un facteur de 3 voire 4 en moyenne sur les grandes instances. Ce fait permet de faire face aux urgences telle la ré-optimisation lors des avénements nécessitant la prise de décisions rapidement.

Notre travail ouvre la voie au développement de nouvelles versions parallèles et distribuées d'ISUD. En outre, la plate-forme distribuée issue de l'intégration de la version distribuée d'ISUD avec la génération de colonnes ouvre de nouvelles pistes pour des projets de recherche futurs afin d'incorporer d'autres paradigmes comme les algorithmes d'apprentissage dans ISUD.

RÉFÉRENCES

E. Abbink, J. van't Wout, & D. Huisman, "Solving large scale crew scheduling problems by using iterative partitioning", *Proceedings of the Seventh Workshop on Algorithmic Approaches for Transportation Modeling. Optimization and Systems Vol. 7*, 2007, pp. 96–106.

P. Alefragis, P. Sanders, T. Takkula, & D. Wedelin, "Parallel integer optimization for crew scheduling", *Annals of Operations Research*, vol. 1, pp. 141–166, 1999.

E. Balas & M. W. Padberg, "On the set-covering problem : II An algorithm for set partitioning", *Operations Research*, vol. 23, pp. 74–90, 1975.

M. L. Balinski & R. E. Quandt, "On an integer program for a delivery problem", *Operations Research*, vol. 12, pp. 300–304, 1964.

C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. Savelsbergh, & P. H. Vance, "Branch-and-price : Column generation for solving huge integer programs", *Operations Research*, vol. 46(3), p. 316–329, 1998.

M. Bürger, G. Notarstefano, F. Bullo, & F. Allgöwer, "Distributed abstract optimization via constraints consensus : theory and applications", *Automatica*, vol. 48, no. 1, pp. 2298–2304, 2012.

H. D. Chu, E. Gelman, & E. L. Johnson, "Solving large scale crew scheduling problems", *Interfaces in Computer Science and Operations Research*. Springer, 1997, pp. 183–194.

T. G. Crainic, B. L. Cun, & C. Roucairol, "Parallel branch-and-bound algorithms", *Parallel Combinatorial Optimization*, pp. 1–28, 2006.

G. Dantzig & P. Wolfe, "Decomposition principle for linear programs", *Operations Research*, vol. 8, p. 101–111, 1960.

G. Desaulniers, J. Desrosiers, M. M. Solomon, & F. Soumis, "Daily aircraft routing and scheduling", GERAD, Montreal, Canada, Rapp. tech., 1994, research report G-94-21.

G. Desaulniers, J. Desrosiers, Y. Dumas, S. Marc, B. Rioux, M. Solomon, & F. Soumis, "Crew pairing at Air France", *European Journal of Operational Research*, vol. 2, pp. 245–259, 1997.

G. Desaulniers, J. Desrosiers, & M. Solomon, “Column generation”, *Springer, New York,*, 2005.

M. Desrochers & F. Soumis, “A column generation approach to the urban transit crew scheduling problem”, *Transport Science*, vol. 23, pp. 1–13, 1989.

J. Eckstein, “Parallel branch-and-bound algorithms for general mixed integer programming”, *Thinking Machines Corporation Technical Report TMC*, vol. 257, 1993.

I. El Hallaoui, A. Metrane, F. Soumis, & G. Desaulniers, *An improved primal simplex algorithm for degenerate linear programs.* INFORMS Journal on Computing, 2011. DOI : 10.1287/ijoc.1100.0425

M. Eso, “Parallel branch and cut for set partitioning”, Thèse de doctorat, Cornell University, 1999.

M. Fischetti, M. Monaci, & D. Salvagnin, “Selfsplit parallelization for mixed-integer linear programming”, *Computers and O.R. (93), 101-112*, 2018.

O. Foutlane, I. El Hallaoui, & P. Hensen, “Integral simplex using double decomposition”, *Cahiers du Gerad, G-2017-73, HEC Montreal*, 2017.

——, “Distributed integral simplex for clustering”, *Cahiers du Gerad, G-2018-31, HEC Montreal*, 2018.

M. Gamache, F. Soumis, G. Marquis, & J. Desrosiers, “A column generation approach for large-scale aircrew rostering problems”, *Operations Research*, vol. 47(2), pp. 247–263, 1999.

M. R. Garey & D. Johnson, *Computers and Intractability : A Guide to the Theory of NP-Completeness.* San Francisco, USA : W.H. Freeman and Co., 1979.

B. Gendron & T. G. Crainic, “Parallel branch-and-branch algorithms : Survey and synthesis”, *Operations Research*, vol. 42 :6, pp. 1042–1066, 1994. DOI : 10.1287/opre.42.6.1042

K. Haase, G. Desaulniers, & J. Desrosiers, “Simultaneous vehicle and crew scheduling in urban mass transit systems”, *Transportation Science*, vol. 35(3), p. 286–303, 2001.

U. U. Haus, M. Koppe, & R. Weismantel, “The integral basis method for integer programming”, *Mathematical Methods of Operations Research*, vol. 53, no. 3, pp. 353–361, 2001.

K. L. Hoffman & M. Padberg, “Solving airline crew-scheduling problems by branch-and-cut”, *Management Science*, vol. 39, no. 6, pp. 657–682, 1993.

V. Janakiram, D. Agrawal, & R. Mehrotra, “A randomizedparallel branch-and-bound algorithm”, *The 1988 International Conference on Parallel Processing*, pp. 69–75, 1988.

C. Joncour, S. Michel, R. Sadykov, & F. Vanderbeck, “Column generation based primal heuristics”, *Electronic Notes in Discrete Mathematics*, vol. 36, p. 695–702, 2010.

S. Jütte & U. W. Thonemann, “Divide and price : A decomposition algorithm for solving large railway crew scheduling problems”, *European Journal of Operational Research*, vol. 219, pp. 214–223, 2012.

A. Kasirzadeh, M. Saddoune, & F. Soumis, “Airline crew scheduling : Models, algorithms, and data sets”, *EURO Journal on Transportation and Logistics*, vol. 6, no. 2, pp. 111–137, 2017.

B. W. Kernighan & S. Lin, “An efficient heuristic procedure for partitioning graphs”, *The Bell System Technical Journal*, vol. 49, pp. 291–307, 1972.

D. Klabjan, E. Johnson, G. Nemhauser, E. Gelman, & S. Ramaswamy, “Solving large airline crew scheduling problems : Random pairing generation and strong branching”, *Computational Optimization and Applications*, vol. 20, pp. 73–91, 2001.

P. Laursen, “Simple approaches to parallel branch and bound”, *Parallel Computing*, vol. 19, pp. 143–152, 1993.

A. N. Letchford & A. Lodi, “Primal cutting plane algorithms revisited”, *Mathematical Methods of Operations Research*, vol. 56, no. 1, pp. 67–81, 2002.

J. T. Linderoth, E. K. Lee, & M. W. P. Savelsbergh, “A parallel, linear programming-based heuristic for large-scale set partitioning problems”, *INFORMS Journal on Computing*, vol. 13, no. 3, pp. 191–209, 2001.

D. Miller & F. Pekny, “The role of performance metrics for parallel mathematical programming algorithms”, *ORSA J. Comput*, vol. 5, pp. 26–28, 1993.

G. Notarstefano & F. Bullo, “Distributed abstract optimization via constraints consensus : theory and applications”, *IEEE Transactions on Automatic Control*, vol. 56, no. 10, pp. 2247–2261, 2011.

P. M. Pardalos, M. G. C. Resende., & K. G. Ramakrishnan., “Parallel processing of discrete optimization problems”, *DIMACS workshop*, vol. 22, pp. 1052–1098, 1994.

M. Quinn, “Analysis and implementation of branch and bound algorithms on a hypercube multicomputer”, *IEEE Transactions on Computers*, vol. 39, pp. 384–387, 1990.

T. Ralphs, Y. Shinano, T. Berthold, & T. Koch, “Parallel solvers for mixed integer linear optimization”, *COR@L Technical Report 16T-014-R3 (ISE, Lehigh University) and Zuse Institute Berlin (ZIB) Technical Report 16-74.*, 2017.

——, “Parallel solvers for mixed integer linear optimization”, *Handbook of Parallel Constraint Reasoning*, pp. 283–336, 2018.

M. R. Rao, “Cluster analysis and mathematical programming”, *Journal of the American Statistical Association*, vol. 66.335, pp. 622–626, 1971.

C. Ribeiro & F. Soumis, “A column generation approach to the multiple depot vehicle scheduling problem”, *Operations Research*, vol. 42, pp. 41–52, 1991.

E. Rönnberg & T. Larsson, “Column generation in the integral simplex method”, *European Journal of Operations Research*, vol. 192, no. 1, pp. 333–342, 2009.

S. Rosat, I. Elhallaoui, F. Soumis, & D. Chakour, “Influence of the normalization constraint on the integral simplex using decomposition”, *Discrete Applied Mathematics*, vol. 217, no. 1, pp. 53–70, 2016.

S. Rosat, I. Elhallaoui, F. Soumis, D. Chakour, & A. Lodi, “Integral simplex using decomposition with primal cutting planes”, *Mathematical Programming, doi :10.1007/s10107-017-1123-x.*, 2017.

S. Rosat, F. Quesnel, I. Elhallaoui, & F. Soumis, “Dynamic penalization of fractional directions in the integral simplex using decomposition : Application to aircrew scheduling”, *European Journal of Operational Research, doi.org/10.1016/j.ejor.2017.05.047.*, vol. 263, pp. 1007–1018, 2017.

M. Saddoune, G. Desaulniers, & F. Soumis, “Aircrew pairings with possible repetitions of the same flight number”, *Computers & Operations Research*, vol. 40(3), p. 805–814, 2013.

A. Saxena, “Set-partitioning via integral simplex method”, Thèse de doctorat, Carnegie Mellon University, 2003.

G. L. Thompson, “An integral simplex algorithm for solving combinatorial optimization problems”, *Computational Optimization and Applications*, vol. 22, no. 3, pp. 351–367, 2002.

H. Topaloglu & W. Powell, “A distributed decision-making structure for dynamic resource allocation using nonlinear functional approximations”, *Operations Research*, vol. 53, pp. 281–297, 2005.

H. W. Trienekens & A. Bruin, “Towards a taxonomy of parallel branch and bound algorithms”, *Erasmus School of Economics (ESE)*, pp. 1042–1066, 1992.

V. Trubin, “On a method of solution of integer linear programming problems of a special kind”, *Soviet Mathematics Doklady*, vol. 10, p. 1544–1546, 1969.

W. Wilhelm, “A technical review of column generation in integer programming”, *Optimization and Engineering*, vol. 2, p. 159–200, 2001.

A. Zaghrouti, F. Soumis, & I. El Hallaoui, “Improving ilp solutions by zooming around an improving direction. cahiers”, *Cahiers du Gerad, G-2013-107, HEC Montreal*, 2013.

A. Zaghrouti, F. Soumis, & I. El Hallaoui, “Integral simplex using decomposition for the set partitioning problem”, *Operations Research*, vol. 62, pp. 435–449, 2014.

A. Zaghrouti, “Algorithme du simplexe en nombres entiers avec décomposition”, *PhD thesis, École Polytechnique de Montréal*, 2016.

A. Zaghrouti, I. E. Hallaoui, & F. Soumis, “Improved integral simplex using decomposition for the set partitioning problem”, *EURO J. Computational Optimization, doi : 10.1007/s13675-018-0098-6*, vol. 6, no. 2, pp. 185–206, 2018.

ANNEXE A ISU2D results on small instances

TABLE A.1 ISU2D results for small aircrew scheduling instances

	ISU2D$_1$						ISU2D$_2$						ISU2D$_3$					
	DVD			IVD		Final	DVD			IVD		Final	DVD			IVD		Final
Instance	#Itr	Time	%Imp	%Cols	Time	gap(%)	#Itr	Time	%Imp	%Cols	Time	gap(%)	#Itr	Time	%Imp	%Cols	Time	gap(%)
AS80-1	6	4	100	-	-	0	4	2	59	66	3	0	5	2	66	67	2	0
AS80-2	3	2	44	66	3	300	7	3	100	-	-	0	7	3	100	-	-	0
AS80-3	5	2	37	100	10	0	6	2	50	100	13	0	6	2	50	100	12	0
AS80-4	4	2	80	100	1	0	4	2	58	100	6	0	5	2	72	100	2	0
AS80-5	3	2	91	100	3	0	5	2	68	66	5	3	5	2	68	66	5	3
AS80-6	7	3	100	-	-	0	4	2	78	66	6	108	11	4	100	-	-	0.3
AS80-7	2	2	88	100	10	2	3	2	77	66	3	0	7	3	100	-	-	0
AS80-8	3	2	46	100	10	291	5	2	100	-	-	0	5	2	100	-	-	0
AS80-9	3	2	64	100	3	0	4	2	35	100	9	0	5	2	58	66	5	0
AS80-10	4	2	58	100	5	0	5	2	62	66	5	170	6	2	62	66	5	100
AS65-1	7	3	100	-	-	0	5	2	100	-	-	0	5	2	100	-	-	0
AS65-2	4	3	100	-	-	0	7	4	100	-	-	0	7	4	100	-	-	0
AS65-3	4	2	98	100	2	0	6	2	91	67	1	0	6	2	91	66	1	0
AS65-4	3	2	90	100	3	0	5	2	100	-	-	0	5	2	100	-	-	0
AS65-5	3	2	74	100	9	0	5	2	100	-	-	0	5	2	100	-	-	0
AS65-6	4	2	100	-	-	0	4	2	100	-	-	0	4	2	100	-	-	0
AS65-7	3	2	71	100	7	0	4	2	81	67	4	0	4	2	81	67	4	0
AS65-8	4	2	96	67	3	0	4	2	100	-	-	0	4	2	100	-	-	0
AS65-9	10	5	100	-	-	0	3	2	66	67	7	0	6	3	100	-	-	0
AS65-10	4	2	100	-	-	0	6	2	100	-	-	0	6	2	100	-	-	0
AS50-1	4	2	94	100	2	0	5	2	96	100	1	0	6	3	100	-	-	0
AS50-2	4	2	84	100	2	0	5	2	100	-	-	0	5	2	100	-	-	0
AS50-3	4	2	100	-	-	0	4	2	100	-	-	0	4	2	100	-	-	0
AS50-4	3	2	78	67	5	4	6	2	100	-	-	0	6	2	100	-	-	0
AS50-5	4	2	100	-	-	0	5	2	100	-	-	0	5	2	100	-	-	0
AS50-6	5	2	100	-	-	0	4	2	100	-	-	0	4	2	100	-	-	0
AS50-7	4	2	96	100	2	0	4	2	100	-	-	0	4	2	100	-	-	0
AS50-8	3	2	89	100	10	1	8	4	99	-	-	1	12	5	100	-	-	0
AS50-9	5	3	100	-	-	0	4	2	100	-	-	0	4	2	100	-	-	0
AS50-10	2	2	75	100	3	0	4	2	100	-	-	0	4	2	100	-	-	0

TABLE A.2 $ISU2D_3$ performances on small aircrew scheduling instances

Instance	C		Is		Ip		IV_1		IV_2		ISU2D$_3$					
	time	gap(%)	time	gap(%)	time	gap(%)	time	gap(%)	time	gap(%)	time	T_C	T_{I_S}	T_{I_P}	T_{IV_1}	T_{IV_2}
AS80-1	13	0	7	0	6	0	7	0	10	0	4	3.25	1.75	1.50	1.75	2.50
AS80-2	9	0	10	0	9	0	15	0	17	0	3	3.00	3.33	3.00	5.00	5.67
AS80-3	8	0	15	0	9	0	13	0	13	0	14	0.57	1.07	0.64	0.93	0.93
AS80-4	6	0	7	0	7	0	8	0	11	0	4	1.50	1.75	1.75	2.00	2.75
AS80-5	13	0	9	0	7	0	15	0	25	0	7	1.86	1.29	1.00	2.14	3.57
AS80-6	11	0	7	0	7	0	7	0	5	0	4	2.75	1.75	1.75	1.75	1.25
AS80-7	10	0	6	0	6	0	35	0	9	0	3	3.33	2.00	2.00	11.67	3.00
AS80-8	17	0	9	0	7	0	12	300	11	0	2	8.50	4.50	3.50	6.00	5.50
AS80-9	8	0	9	0	7	0	10	0	25	0	7	1.14	1.29	1.00	1.43	3.57
AS80-10	7	0	7	200	6	200	7	500	16	0	7	-	-	-	-	-
AS65-1	6	0	6	0	5	0	5	0	11	0	2	3.00	3.00	2.50	2.50	5.50
AS65-2	10	0	7	0	5	0	6	400	7	0	4	2.50	1.75	1.25	1.50	1.75
AS65-3	9	0	5	0	4	0	3	0	6	0	3	3.00	1.67	1.33	1.00	2.00
AS65-4	7	0	5	0	5	0	9	300	8	0	2	3.50	2.50	2.50	4.50	4.00
AS65-5	11	0	5	0	4	0	9	0	12	0	2	5.50	2.50	2.00	4.50	6.00
AS65-6	9	0	7	0	5	0	6	340	17	0	2	4.50	3.50	2.50	3.00	8.50
AS65-7	23	0	8	0	8	0	10	0	20	0	6	3.83	1.33	1.33	1.67	3.33
AS65-8	9	0	4	0	4	0	3	0	7	0	2	4.50	2.00	2.00	1.50	3.50
AS65-9	7	0	7	0	6	0	10	340	7	0	3	2.33	2.33	2.00	3.33	2.33
AS65-10	7	0	5	0	5	0	6	0	10	0	2	3.50	2.50	2.50	3.00	5.00
AS50-1	22	0	5	0	5	0	7	0	7	0	3	7.33	1.67	1.67	2.33	2.33
AS50-2	9	0	5	0	4	0	6	0	19	0	2	4.50	2.50	2.00	3.00	9.50
AS50-3	5	0	5	0	4	0	2	0	11	0	2	2.50	2.50	2.00	1.00	5.50
AS50-4	9	0	7	0	7	0	7	0	13	0	2	4.50	3.50	3.50	3.50	6.50
AS50-5	19	0	4	0	3	0	4	0	7	0	2	9.50	2.00	1.50	2.00	3.50
AS50-6	9	0	5	0	3	0	4	0	10	0	2	4.50	2.50	1.50	2.00	5.00
AS50-7	7	0	4	0	4	0	8	0	10	0	2	3.50	2.00	2.00	4.00	5.00
AS50-8	8	0	5	0	5	0	5	0	8	0	5	1.60	1.00	1.00	1.00	1.60
AS50-9	8	0	6	0	5	0	9	0	14	0	2	4.00	3.00	2.50	4.50	7.00
AS50-10	6	0	5	0	4	0	4	0	15	0	2	3.00	2.50	2.00	2.00	7.50

ANNEXE B ISU2D results on medium instances

TABLE B.1 ISU2D results for medium bus driver scheduling instances.

	ISU2D$_1$						ISU2D$_2$						ISU2D$_3$					
	DVD			IVD		Final	DVD			IVD		Final	DVD			IVD		Final
Instance	#Itr	Time	%Imp	%Cols	Time	gap(%)	#Itr	Time	%Imp	%Cols	Time	gap(%)	#Itr	Time	%Imp	%Cols	Time	gap(%)
MB80-1	3	32	45	33	28	0	2	33	37	33	27	0	3	32	42	33	28	0
MB80-2	2	34	50	34	22	0	3	33	55	34	23	0	4	32	55	33	22	0
MB80-3	3	32	57	32	19	0	3	32	67	32	29	0	3	32	67	32	28	0
MB80-4	2	35	55	33	26	0	3	34	40	33	39	0	4	33	40	33	45	0
MB80-5	3	33	35	50	33	0	3	33	30	50	31	0	3	33	40	50	31	0
MB80-6	2	36	20	65	34	0	3	35	15	32	23	0	4	34	20	32	20	0
MB80-7	2	32	55	33	69	0	3	32	85	33	15	0	2	32	85	33	15	0
MB80-8	2	34	42	33	29	0	3	33	35	33	26	0	3	32	35	33	27	0
MB80-9	1	33	42	50	43	0	3	33	65	50	24	0	3	32	65	50	25	0
MB80-10	2	34	37	34	23	0	2	32	30	34	42	0	3	31	50	34	22	0
MB65-1	3	33	44	40	62	0	3	33	50	40	40	0	3	32	50	40	33	0
MB65-2	2	32	72	36	32	0	3	32	72	36	8	0	6	54	100	-	-	0
MB65-3	1	32	59	64	118	0	3	32	50	50	42	0	3	32	50	50	34	0
MB65-4	1	34	51	36	69	0	3	33	64	36	44	0	3	32	64	36	52	0
MB65-5	2	32	47	36	40	0	3	32	50	36	37	0	3	32	50	36	37	0
MB65-6	3	34	70	32	23	0	3	32	67	32	29	0	2	32	67	33	32	0
MB65-7	1	33	48	50	44	0	2	32	39	50	84	0	2	31	39	50	83	0
MB65-8	3	33	66	35	26	0	3	34	59	35	4	0	3	33	59	33	5	0
MB65-9	1	34	47	30	40	0	3	33	44	50	34	0	3	32	44	50	36	0
MB65-10	2	33	79	50	24	0	2	33	82	50	22	0	2	32	82	50	23	0
MB50-1	2	34	69	50	11	0	3	35	100	-	-	0	3	33	100	-	-	0
MB50-2	1	33	72	34	19	4	3	33	92	34	16	0	2	32	92	34	16	0
MB50-3	2	30	85	10	2	0	3	29	94	10	2	6	5	47	100	-	-	0
MB50-4	3	33	85	35	11	0	4	52	100	-	-	0	4	52	100	-	-	0
MB50-5	2	31	52	35	28	0	3	32	100	-	-	0	3	31	100	-	-	0
MB50-6	3	32	84	33	13	0	3	32	100	-	-	0	3	32	100	-	-	0
MB50-7	3	33	64	33	14	0	2	33	76	40	21	0	2	32	76	40	22	0
MB50-8	1	33	80	20	50	0	3	53	100	-	-	0	3	52	100	-	-	0
MB50-9	2	34	72	34	33	0	2	33	84	34	15	0	2	33	84	34	14	0
MB50-10	2	33	92	34	13	0	3	32	76	34	5	0	6	44	100	-	-	0

TABLE B.2 ISU2D$_3$ performances on medium bus driver scheduling instances

Instance	Is		Ip		IV_1		IV_2		ISU2D3				
	time	gap(%)	time	gap(%)	time	gap(%)	time	gap(%)	time	T_{I_s}	T_{I_p}	T_{IV_1}	T_{IV_2}
MB80-1	379	0	147	0	120	0	1800	39	59	6.42	2.49	2.03	-
MB80-2	178	0	93	0	116	0	1800	42	54	3.30	1.72	2.15	-
MB80-3	139	0	70	0	68	0	1800	33	60	2.32	1.17	1.13	-
MB80-4	152	0	86	0	128	0	1800	42	78	1.95	1.10	1.64	-
MB80-5	145	22.6	90	22.6	76	22.6	1800	40	64	-	-	-	-
MB80-6	98	32.9	56	32.9	51	32.9	1800	43	54	-	-	-	-
MB80-7	371	0	147	0	102	0	1800	25	48	7.73	3.06	2.13	-
MB80-8	97	41.1	62	41.1	110	41.1	1800	38	59	-	-	-	-
MB80-9	156	0	76	0	91	0	1800	34	57	2.74	1.33	1.60	-
MB80-10	54	63.7	44	63.7	83	63.7	1800	40	53	-	-	-	-
MB65-1	214	0	93	0	114	0	1800	27	55	3.89	1.69	2.07	-
MB65-2	134	0	66	0	57	0	1800	11	54	2.48	1.22	1.06	-
MB65-3	131	10.3	69	10.3	96	10.3	1800	22	66	-	-	-	-
MB65-4	238	12.3	93	12.3	113	12.3	1800	21	84	-	-	-	-
MB65-5	324	0	124	0	114	0	1800	29	69	4.70	1.80	1.65	-
MB65-6	141	0	76	0	70	0	1800	25	64	2.20	1.19	1.09	-
MB65-7	288	0	131	0	112	0	1800	27	114	2.53	1.15	0.98	-
MB65-8	111	0	62	0	66	0	1800	15	38	2.92	1.63	1.74	-
MB65-9	171	0	78	0	85	0	1800	20	68	2.51	1.15	1.25	-
MB65-10	179	0	77	0	93	0	1800	29	55	3.25	1.40	1.69	-
MB50-1	54	0	44	0	33	0	1800	11	33	1.64	1.33	1.00	-
MB50-2	94	0	51	0	56	0	218	0	48	1.96	1.06	1.17	4.50
MB50-3	71	0	49	0	42	0	1800	17.52	47	1.51	1.04	0.89	-
MB50-4	83	0	51	0	50	0	1095	0	52	1.60	0.98	0.96	21.05
MB50-5	186	0	73	0	49	0	1800	9.32	31	6.00	2.35	1.58	-
MB50-6	57	0	44	0	51	0	1800	21.09	32	1.78	1.38	1.59	-
MB50-7	158	0	77	0	107	0	1800	23.56	54	2.93	1.43	1.98	-
MB50-8	155	0	71	0	82	0	1800	19.79	52	2.98	1.37	1.58	-
MB50-9	263	0	95	0	65	0	1800	21.09	47	5.60	2.02	1.38	-
MB50-10	73	0	44	0	42	0	1800	9.31	44	1.66	1.00	0.95	-

ANNEXE C ISU2D results on large instances

TABLE C.1 ISU2D results for large bus driver scheduling instances.

	ISU2D$_1$						ISU2D$_2$						ISU2D$_3$					
	DVD			IVD		Final	DVD			IVD		Final	DVD			IVD		Final
Instance	#Itr	Time	%Imp	%Cols	Time	gap(%)	#Itr	Time	%Imp	%Cols	Time	gap(%)	#Itr	Time	%Imp	%Cols	Time	gap(%)
LB80-1	3	209	52	40	127	0	3	190	28	40	120	0	3	192	28	40	129	0
LB80-2	4	217	32	38	120	0	2	206	21	38	123	0	5	210	34	38	115	0
LB80-3	3	201	62	44	461	0	4	201	70	44	209	0	4	202	70	44	212	0
LB80-4	4	203	55	37	108	0	4	196	57	37	108	0	4	196	57	37	109	0
LB80-5	2	201	17	48	172	0	4	188	32	48	251	0	4	189	32	48	275	0
LB80-6	2	202	42	47	153	0	4	194	49	47	140	0	4	194	49	47	151	0
LB80-7	3	213	26	48	154	0	2	207	13	48	457	0	5	212	30	48	147	0
LB80-8	3	200	34	42	150	0	3	194	28	42	136	0	3	195	28	42	141	0
LB80-9	2	204	28	50	889	0	4	209	66	50	150	0	4	207	66	50	152	0
LB80-10	3	212	66	40	106	0	4	200	49	40	197	0	4	203	49	40	213	0
LB65-1	2	213	100	-	-	0	3	231	100	-	-	0	3	205	100	-	-	0
LB65-2	3	214	58	44	283	0	7	213	60	44	312	0	7	524	100	-	-	0
LB65-3	2	203	81	40	158	0	4	243	100	-	-	0	4	246	100	-	-	0
LB65-4	2	187	53	44	311	0	2	190	81	44	189	0	2	189	81	44	195	0
LB65-5	3	213	49	48	374	0	3	209	60	48	130	0	3	215	72	48	146	0
LB65-6	3	215	70	60	281	0	3	216	77	60	395	0	6	487	100	-	-	0
LB65-7	3	196	61	45	453	0	2	192	56	45	247	0	3	199	70	45	232	0
LB65-8	3	207	77	50	384	0	3	236	100	-	-	0	3	208	100	-	-	0
LB65-9	3	210	53	48	248	0	4	386	100	-	-	0	4	387	100	-	-	0
LB65-10	3	201	63	46	207	0	3	196	28	46	332	0	4	201	47	46	221	0
LB50-1	2	233	100	-	-	0	3	193	88	47	98	0	4	193	100	-	-	0
LB50-2	3	202	88	53	108	0	3	202	100	-	-	0	3	201	100	-	-	0
LB50-3	2	220	100	-	-	0	3	210	100	-	-	0	3	207	100	-	-	0
LB50-4	3	211	61	46	346	0	2	203	52	46	418	0	4	214	67	46	226	0
LB50-5	2	261	100	-	-	0	2	208	76	53	213	0	5	394	100	-	-	0
LB50-6	2	212	100	-	-	0	3	212	100	-	-	0	3	209	100	-	-	0
LB50-7	3	210	79	56	150	0	3	210	79	56	151	0	3	208	79	56	154	0
LB50-8	2	211	100	-	-	0	3	204	100	-	-	0	2	200	100	-	-	0
LB50-9	2	204	68	55	469	0	3	203	85	55	203	0	4	201	100	-	-	0
LB50-10	2	222	85	42	318	0	3	222	100	-	-	0	3	223	100	-	-	0

TABLE C.2 ISU2D$_3$ performances on large bus driver scheduling instances

Instance	Is		Ip		IV_1		IV_2		ISU2D$_3$				
	time	gap(%)	time	gap(%)	time	gap(%)	time	gap(%)	time	T_{Is}	T_{Ip}	T_{IV_1}	T_{IV_2}
LB80-1	1822	0	1335	0	747	0	3600	44	322	5.66	4.15	2.32	-
LB80-2	2594	0	1199	0	664	0	3600	43	325	7.98	3.69	2.04	-
LB80-3	1485	0	1121	0	588	0	3600	43	414	3.59	2.71	1.42	-
LB80-4	2411	0	1060	0	679	0	3600	44	306	7.88	3.46	2.22	-
LB80-5	2867	0	1391	0	842	0	3600	45	464	6.18	3.00	1.81	-
LB80-6	1419	0	802	0	595	0	3600	42	345	4.11	2.32	1.72	-
LB80-7	4518	0	1321	0	282	56	3600	43	359	12.58	3.68	-	-
LB80-8	2544	0	1532	0	899	0	3600	44	336	7.57	4.56	2.68	-
LB80-9	1947	0	1067	0	599	0	3600	42	359	5.42	2.97	1.67	-
LB80-10	1522	0	1204	0	595	0	3600	43	416	3.66	2.89	1.43	-
LB65-1	1375	0	854	0	695	0	3600	35	205	6.71	4.17	3.39	-
LB65-2	935	0	666	0	812	0	3600	38	524	1.78	1.27	1.55	-
LB65-3	769	0	636	0	735	0	3600	35	246	3.13	2.59	2.99	-
LB65-4	1403	0	1106	0	534	0	3600	36	384	3.65	2.88	1.39	-
LB65-5	2064	0	959	0	687	15	3600	37	361	5.72	2.66	-	-
LB65-6	1842	0	1244	0	632	0	3600	35	487	3.78	2.55	1.30	-
LB65-7	3470	0	1187	0	484	19.6	3600	38	431	8.05	2.75	-	-
LB65-8	1657	0	1189	0	859	0	3600	38	208	7.97	5.72	4.13	-
LB65-9	2579	0	1213	0	708		3600	39	387	6.66	3.13	1.83	-
LB65-10	1965	0	1063	0	441	10.52	3600	34	422	4.66	2.52	-	-
LB50-1	833	0	602	0	389	0	3600	28	193	4.32	3.12	2.02	-
LB50-2	852	0	663	0	460	0	3600	27	201	4.24	3.30	2.29	-
LB50-3	367	0	359	0	258	0	3600	25	207	1.77	1.73	1.25	-
LB50-4	712	10.7	1970	10.7	440	10.7	3600	26	440	-	-	-	-
LB50-5	1215	0	951	0	713	0	3600	31	394	3.08	2.41	1.81	-
LB50-6	350	0	329	0	181	0	3600	22	209	1.67	1.57	0.87	-
LB50-7	3932	0	1040	0	722	0	3600	32	362	10.86	2.87	1.99	-
LB50-8	649	0	586	0	391	0	3600	23	200	3.25	2.93	1.96	-
LB50-9	577	0	520	0	389	0	3600	30	201	2.87	2.59	1.94	-
LB50-10	1269	0	1198	0	471	0	3600	31	223	5.69	5.37	2.11	-