Contribution for solving combinatorial problems : sequential and parallel optimization

Sagvan Ali Saleh

Thèse de Doctorat

Spécialité Informatique

présentée à ***l'Ecole Doctorale en Sciences Technologie et Santé (ED 547)***

de l'Université de Picardie Jules Verne

par

Sagvan Ali SALEH

pour obtenir le grade de Docteur de l'Université de Picardie Jules Verne

Contribution à la résolution des problèmes combinatoires : optimisation séquentielle et parallèle

Soutenue le 15/06/2015, après avis des rapporteurs, devant le jury d'examen :

M. Haoxun CHEN,	**Professeur**	**Rapporteur**
M. Jin-Kao HAO,	**Professeur**	**Rapporteur**
M. Imed KACEM	**Professeur**	**Examinateur**
M. Toufik SAADI	**Maître de Conférences**	**Examinateur**
M. Mhand HIFI,	**Professeur,**	**Directeur**
M. Lei WU,	**Maître de Conférences**	**Co-directeur**

Preface

This PhD thesis has been prepared at the EPROAD[1] laboratory at the University of Picardie Jules Verne (UPJV), during the period October 2012 to June 2015. The work has been supervised by Professor Mhand Hifi[2] and Doctor Lei Wu[3].

The presentation includes a general introduction, a state of the art and three separate papers, considering a problem within the knapsack family, namely the disjunctively constrained knapsack problem. Chapters 4 has been published in an international conference with proceedings, the 3rd International Symposium on Combinatorial Optimization (cf., Hifi *et al.* [36]). Chapter 5 is accepted in May 2015 in the international journal *Cogent Engineering* with the paper: M. Hifi, S. Saleh, and L. Wu, "Hybrid guided neighborhood search for the disjunctively constrained knapsack problem". Chapter 7 has been published in an international conference with proceedings, the 28th IEEE International Parallel & Distributed Processing Symposium (cf., Hifi *et al.* [32]).

Acknowledgment

First of all, I would like to thank my advisers, Professor Mhand Hifi and Doctor Lei Wu, without them this work would not have been possible. I am very grateful for guiding me through these three years. Their supervision, steady support and inexhaustible ideas have been extremely useful in progressing in my research.

My deep gratitude goes to professors Haoxun CHEN[4] and Jin-Kao HAO[5] for accepting to review my PhD thesis and the interest that they have carried to my work. I would like to thank also the professor Imed KACEM[6] and Doctor Toufik SAADI[7] for accepting to participate to the jury of the thesis.

I'm happy to have spent my doctoral studies at the EPROAD laboratory. I would like to thank all members of this laboratory with whom I have shared lots of discussions and beautiful times.

The work on this thesis has been financially supported by research grants from ministry of higher education, Kurdistan regional government of Iraq. I would like to express my gratitude to them for financing my study.

Finally, I would like to express my deep gratitude to my family, especially my parents, my wife, and my daughter for their unconditional support and patience over the years. Thanks for motivating me the rest of the time and offering comfort when necessary.

[1] Eco-PRocédés, Optimization et Aide à la Décision
[2] Professor at the University of Picardie Jules Verne
[3] Associate professor at the University of Picardie Jules Verne
[4] Professor at the University of Troyes
[5] Professor at the University of Angers
[6] Professor at the University of Lorraine
[7] Associate professor at the University of Picardie Jules Verne

Résumé

Titre : contribution à la résolution des problemes combinatoires : optimisation séquentielle et paralléle

Les problèmes d'optimisation combinatoire sont d'un grand intérêt à la fois pour le monde scientifique et le monde industriel. La communauté scientifique a oeuvré pour la simplification de certains problèmes issus du monde industriel vers des modèles d'optimisation combinatoire. Parmi ces problèmes, on peut trouver des problèmes appartenant à la famille du problème du sac à dos (*knapsack*). Dans cette thèse, nous considérons une variante du problème du sac à dos : le *problème du sac à dos avec des contraintes disjonctives* (*Knapsack with Disjunctive Constraints*). En raison de la difficulté de cette problématique, nous nous sommes intéressés au développement de méthodes heuristiques produisant des solutions de bonne qualité en un temps de calcul modéré.

Nos travaux de recherche s'appuient sur le principe de la recherche par voisinage. Bien que cette façon de faire nous conduise vers des solutions approchées, leur utilisation ainsi que les résultats que nous avons obtenus restent intéressants tout en gardant un temps d'exécution raisonnable.

Afin de résoudre des instances de grande taille pour la problématique étudiée, nous avons proposé des méthodes séquentielles et parallèles. Ces deux techniques de résolution sont basées sur la recherche par voisinage. Dans un premier temps, une première méthode de recherche par voisinage aléatoire a été proposée. Elle s'appuie sur la combinaison de deux procédures : une première procédure qui cherche à construire une série de solutions partielles et une deuxième procédure qui complète chacune des solutions partielles courantes par une exploration de son voisinage. Ensuite, une deuxième méthode adaptative a été mise en place. Elle s'appuie sur un système d'optimisation par colonie de fourmis pour simuler une recherche guidée et une procédure de descente pour explorer au mieux les solutions produites au cours du processus de recherche. Finalement, une troisième méthode a été élaborée dans le but de faire évoluer la performance des méthodes de recherche par voisinage. Dans cette partie de nos travaux de recherche, nous avons proposé une recherche par voisinage aléatoire parallèle. Nous nous sommes principalement appuyés sur l'exploration simultanée de différents (sous)espaces de recherche par différents processeurs, où chaque processeur adopte sa propre stratégie aléatoire pour construire ses propres voisinages en fonction de ses informations internes récoltées.

Mots clés : Heuristique ; optimisation combinatoire ; parallélisme ; sac à dos ; voisinage.

Abstarct

Title : Contribution for solving combinatorial problems: sequential and parallel optimization

Combinatorial optimization problems are of high interest both for the scientific world and for the industrial world. The research community has simplified many practical situations as combinatorial optimization problems. Among these problems, we can find some problems belonging to the knapsack family. This thesis considers a particular problem belonging to the knapsack family, known as the disjunctively constrained knapsack problem. Because of the difficulty of this problem, we are searching for approximate solution techniques with fast solution times for its large scale instances.

A promising way to solve the disjunctively constrained knapsack problem is to consider some techniques based upon the principle of neighborhood search. Although such techniques produce approximate solution methods, they allow us to present fast algorithms that yield interesting solutions within a short average running time.

In order to tackle large scale instances of the disjunctively constrained knapsack problem, we present sequential and parallel algorithms based upon neighborhood search techniques. The first algorithm can be viewed as a random neighborhood search method. This algorithm uses a combination of neighborhood search techniques in order to randomly explore a series of sub-solution spaces, where each subspace is characterized by a neighborhood of a local optimum. The second algorithm is an adaptive neighborhood search that guides the search process in the feasible solution space towards high quality solutions. This algorithm uses an ant colony optimization system to simulate the guided search. The third and last algorithm is a parallel random neighborhood search method which exploits the parallelism for exploring simultaneously different sub-solution spaces by several processors. Each processor adopts its own random strategy to yield its own neighborhoods according to its internal information.

Keywords: Combinatorial optimization; heuristic; knapsack; neighborhood; parallelism.

Contents

List of Figures

List of Tables

Chapter 1

Introduction générale

1.1 Introduction

La plupart des problèmes pratiques peuvent être modélisés comme des problèmes d'optimisation combinatoire. Parmi ces problèmes, nous pouvons trouver des problèmes appartenant à la famille du célèbre problème du sac à dos. Afin de résoudre ce genre de problème, deux axes de recherche peuvent être suivis: (i) résoudre ces problèmes de façon optimale, en utilisant des méthodes exactes et (ii) chercher les solutions presque optimales, en utilisant les méthodes heuristiques. Ainsi, un algorithme exact essaye de trouver une ou un ensemble de solutions optimale(s) pour un problème donné. Pour les problèmes de type sac à dos, une solution optimale peut être trouvée en utilisant les méthodes comme: la procédure de séparation et évaluation, la procédure de séparation et coupe et/ou la programmation dynamique. Cependant, pour résoudre les problèmes de grande taille, une méthode exacte aura besoin d'un temps de résolution exponentiel. Ceci, rend les méthodes exactes moins intéressantes pour résoudre les cas pratiques ou industriels. Pour ces raisons, ces dernières décennies pour la résolution des problèmes de l'optimisation combinatoire. Les méthodes heuristiques ont reçu plus d'attention de la part des chercheurs. Néanmoins, nous notons que, il existe d'autres méthodes de résolution, appelées méthodes hydrides, car elles combinent les méthodes exactes et les méthodes heuristiques.

Dans cette thèse, les méthodes de recherche par voisinage sont considérées pour résoudre de façon approchée un cas particulier du problème du sac à dos, à savoir: le problème du sac à dos avec des contraintes disjonctives. Ce dernier, a d'importantes applications, particulièrement, dans le domaine de la logistique du transport, administration financière, etc. Ainsi, comme exemple d'application, considérons un cas pratique, rencontré dans le transport, relatif au déplacement des personnes malades, des personnes âgées et autres personnes ayant des demandes spécifiques. Sachant que, les personnes malades ont leurs propres exigences, la plupart de temps, elles ont besoin d'être servies dans le respect de quelques conditions spéciales. Par exemple, certains d'entre eux ne peuvent pas prendre le même bus. Autrement dit, le bus ne peut pas servir ces deux personnes incompatibles, en même temps. Formellement, ce problème peut être modélisé comme un problème de sac à dos, avec contraintes disjonctives.

Cette thèse présente, des algorithmes séquentiel et parallèle qui sont basés sur les techniques de recherche par voisinage. Ces approches sont adaptées pour résoudre le problème du sac à dos avec des contraintes disjonctives, même sur

les instances de grande taille. D'autre part, ces approches peuvent aussi, être utilisées pour résoudre d'autres problèmes de l'optimisation combinatoire avec des instances de grande taille.

Organisation de la thèse

Cette thèse comporte trois parties :

- La première partie présente un état de l'art sur le problème du sac à dos et ses variantes.
- La deuxième partie est consacrée à la proposition de méthodes heuristiques séquentielles pour le sac à dos disjonctif.
- La troisième artie contient une version parallèle de l'approche précédemment proposée dans la première partie.

Plus précisément, cette thèse est organisé comme suit. La première partie (chapitre 3) présente un état de l'art sur le problèmes de type sac à dos et certaines de ses variantes. Nous commençons par décrire le problème du sac à dos classique. Pour ce dernier, nous présentons une brève description de certaines méthodes de résolutions exactes et approchées bien connues dans la littérature. Finalement, le sac à dos avec contraintes disjonctifs (*Disjunctive Constrained Knapsac Probelm* − DCKP) est décrit puis suivi par la description de certaines méthodes préesentes dans la littérature.

La deuxième partie est comporte deux chapitres: chapitre 4 et chapitre 5 . Dans le chapitre 4, nous présentons le premier algorithme séquentiel pour les problèmes de l'optimisation combinatoire. Cet algorithme peut être vu comme une méthode de recherche aléatoire par voisinage. En effet, l'objectif principal de ce travail est de présenter un algorithme efficace qui permet de produire des solutions de bonne qualité en un temps d'exécution raisonnable. La méthode comporte deux étapes complémentaires: une première étape de destruction et une deuxième étape de reconstruction (ou re-optimisation). Ainsi, l'étape de re-optimisation est composée de deux procédures: une première procédure qui fournit une solution réalisable et une deuxième procédure qui s'intéresse à l'amélioration de la qualité de la solution courante. L'étape de destruction effectue une suppression aléatoire d'un nombre d'éléments appartenant à la solution dans le but de construire une nouvelle solution dans le voisinage. Notons que cette étape est considérée comme la plus importante de la méthode car elle permet d'élargir l'espace de recherche tout en gardant certains éléments de la salution courante. En plus, cette étape de la méthode a pour but d'éviter une série d'optima-locaux lors de la recherche. En effet, la solution produite par l'étape de destruction est traitée comme un problème réduit qui peut être optimiser par diverses approches.

Le chapitre 5, décrit la deuxième méthode heuristique séquentielle pour le DCKP. Le but de ce chapitre est de proposer un algorithme amélioré qui est capable de guider le processus de recherche vers de nouveaux espaces de recherche. La méthode combine le principe d'une recherche guidée et une stratégie de décomposition. D'abord, la stratégie de décomposition est utilisée afin de construire une série de sous-problèmes à résoudre. Ensuite, la méthode utilise une optimisation par colonie de fourmis afin de simuler la recherche guidée. dans ce cas, la stratégie de décomposition permet de construire deux sous-problèmes : le problème de la recherche d'un stable maximum pondérée dans un graphe et le problème du sac à dos classique. Le premier sous-problème est résolu d'une façon approchée par application de la méthode basée sur les colonies fourmis. Le deuxième sous-problème est résolu en utilisant un méthode exacte. Ainsi, la phase de décomposition donne une solution réalisable pour le DCKP. Une fois la solution réalisable est obtenue, une méthode de descente est appliquée pour tenter d'améliorer la solution courante. Par ailleurs, afin d'éviter la stagnation des solutions, une diversification est introduite. Cette dernière est basée sur une adaptation de la méthode d'optimisation par colonie de fourmis. Durant le processus de recherche, l'optimisation par colonie de fourmis tente de diversifier et d'améliorer les solutions en utilisant quelques informations récoltées des itérations précédentes. Ce processus est répété jusqu'à ce que le critère d'arrêt soit atteint.

La troisième partie est consacrée aux méthodes de résolution parallèle. Elle se compose de deux chapitres (chapitre 6 et 7).

Le chapitre 6 résume le principe du parallélisme. Il commence par introduire la programmation séquentielle et parallèle. Ensuite, il donne une brève description des principales architectures des machines parallèles tout en décrivant quelques approches parallèle qui nous seront utiles dans notre étude. Finalement, nous décrivons quelques librairies utilisées dans les méthodes parallèles.

Le chapitre 7 décrit l'algorithme parallèle considéré pour la résolution des instances de grande taille pour le DCKP. Il s'agit d'une méthode de recherche parallèle à voisinage large. Le but de cette étude est d'explorer les avantages du parallélisme pour accélérer la résolution et de tenter de construire des solutions de meilleures qualités. En effet, l'idée principale consiste à explorer différents voisinages en parallèle, où chaque processeur adopte sa propre stratégie dans la recherche de la meilleure solution dans ses propres voisinages successives. De plus, l'ensemble des processeurs travaillent pour bénéficier des meilleures solutions obtenues lors de la recherche parallèle. Cette approche utilise le modèle MPI[1], où les messages inter-processeurs s'appuient sur des opérations de coopération entre les processeurs.

Finalement, le chapitre 8 donne une conclusion générale, dans laquelle nous les résultats obtenus sont résumés ainsi qu'une orientations et perspectives pour des travaux futurs dans ce domaine.

[1]MPI signifie Message Passing Interface

1.2 Les problèmes du type sac à dos

Les problèmes du type sac à dos (noté: *Knapsack Problems*) ont été étudiés depuis plus d'un siècle (voir., Matheows [48]), et plus intensément après les travaux de Dantzig (voir., Dantzig [16]). Cela est dû au fait que non seulement ils possèdent de nombreuses applications pratiques importantes tant dans l'industrie que dans la gestion financière, mais aussi pour des raisons théoriques., En effet, ces pproblématiques se produisent fréquemment en tant que sous-problème dans les procédures de résolution de problèmes plus complexes (voir., Hifi *et al.* [35]; Pisinger [54]), ce qui rend leurs modèles théoriques très importants pour la communauté scientifique (voir., Pisinger *et al.* [57]).

Dans la littérature, il existe plusieurs variantes du problème du sac à dos, (voir la Figure 1.1). Parmi ces variantes, dans ce qui suit, nous citons la variante du problème du sac à dos que nous avons étudié dans cette thèse, à savoir le problème du sac à dos avec des contraintes disjonctives. Toutefois, cette famille a été largement étudiée (pour plus de détails voir, Martello *et al.* [46]; Dudzinski *et al.* [20]).

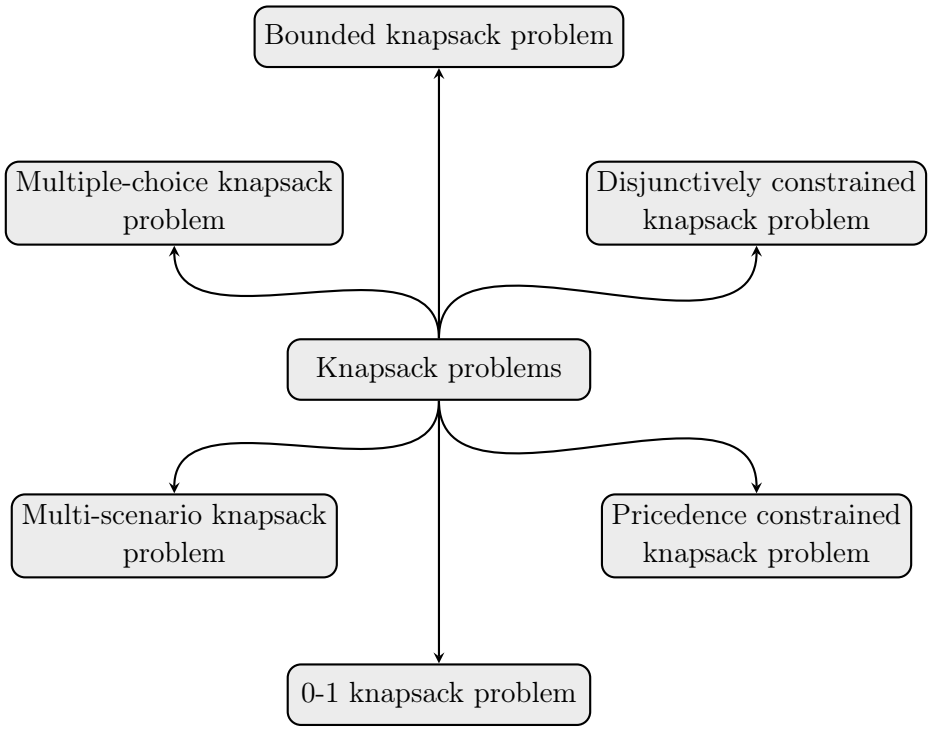


Figure 1.1: Quelques variantes du *problème du sac à dos.*

1.2.1 Le problème du sac à dos

Le problème du sac à dos (ou binary knapsack problem, noté 0-1 KP) est en particulier l'un des plus connu parmi les problèmes de l'optimisation combinatoire. La raison de cet intérêt s'explique principalement par trois faits (voir., Martello *et al.* [46]):

- Il a la représentation la plus simple dans le domaine de programmation linéaire en nombres entiers.
- Il apparaît souvent comme un sous-problème dans plusieurs problèmes plus complexes à résoudre.
- Il représente souvent beaucoup de situations pratiques.

Plusieurs applications peuvent être simulées au problème du sac à dos : par exemple le chargement de la cargaison, l'informatique (voir., Horowitz *et al.* [37]) et la gestion financière (voir., Pisinger [54]).

Dans cette partie, nous considérons juste un exemple simple issu de la gestion financière. Supposons que l'on dispose de n projets, où chaque projet j est caractérisé par un profit p_j, un coût c_j et une capacités c mesurée en dollar. Le problème est de déterminer l'ensemble des projets qu'on peut réaliser avec la somme limitée c. Ainsi, ce problème peut être représenté comme un probeème de sac à dos à variables binaires et peut être résolu par application de différentes méthodes dédiées connues dans la littérature (pour plus de details, voir Kellerer *et al.* [38]; Fayard *et al.* [21]; Balas *et al.* [7]; Martello *et al.* [45]; Martello *et al.* [47]).

1.2.1.1 Definition

Soit I un ensemble de n articles et c la capacité d'une contrainte knapsack. Chaque article i est caractérisé par un profit p_i et un poids w_i. L'objectif du problème est de sélectionner un sous-ensemble d'articles de sorte que la somme des profits des éléments choisis soit maximum, sans que la somme des poids ne dépasse la capacité c (la Figure 1.2 illustre un exemple traitant ce cas).

Considérons une instance du sac à dos représentée par I l'ensemble des article tel que $I = \{1, 2, 3, 4, 5\}$. Chacun de ces articles est caractérisé par un vecteur profit $p = \{p_1, p_2, p_3, p_4, p_5\} = \{12\ \$, 15\ \$, 17\ \$, 14\ \$, 10\ \$\}$, un vecteur poids $w = \{w_1, w_2, w_3, w_4, w_5\} = \{8\text{ kg}, 20\text{ kg}, 12\text{ kg}, 14\text{ kg}, 15\text{ kg}\}$, et une capacité $c = 25$ kg.

Une solution réalisable peut consister à placer les articles 1 et 5 avec une valeur de 22. En même temps, une autre solution réalisable consisterait à placer juste l'élément 2 avec une valeur totale de 15 (voir la Figure 1.2). Ainsi, tous les sous-ensembles d'articles dont le poids total n'excède pas la capacité c sont des solutions réalisables. Cependant, le nombre de solutions possibles augmente avec l'augmentation du nombre d'articles de l'ensemble I. Dans un tel cas, le but est

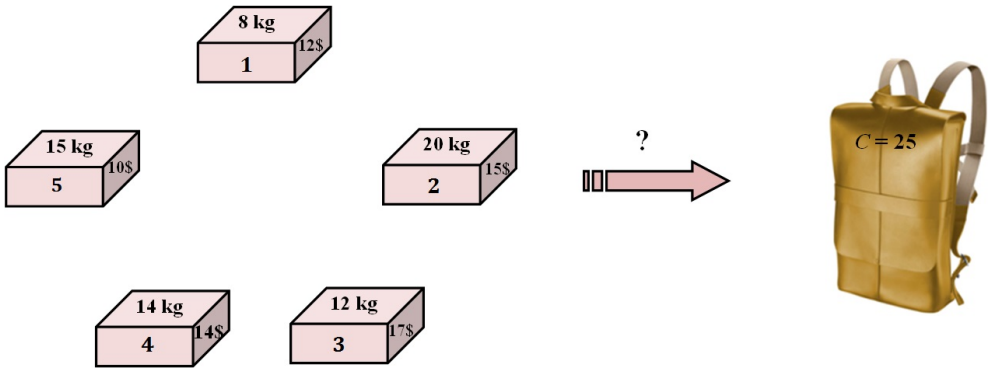


Figure 1.2: Le problème du sac à dos classique

de trouver parmi toutes les solutions possibles celle qui maximise le profit (c-à-d. une solution optimale). Si nous revenons sur l'exemple, il est clair la meilleure solution est celle qui réalise le meilleure gain: il s'agit de la solution contenant les articles 1 et 3 dont la valeur de la fonction objectif vaut 29 et dont le poids total est égal à 20 kg.

1.2.1.2 Modélisation du problème du sac à variables binaires

Le problème du sac à dos à variables binaires est souvent représenté par le modèle simple suivant :

$$Max: \quad f(x) = \sum_{i=1}^{n} p_i x_i \tag{1.1}$$

$$s.t. \quad \sum_{i=1}^{n} w_i x_i \leq c \tag{1.2}$$

$$x_i \in \{0,1\} \qquad \forall i \in I = \{1,...,n\} \tag{1.3}$$

où x_i représente la variable de décision telle que :

$$x_i = \begin{cases} 1 & \text{si l'objet } i \text{ est placé dans le sac} \\ 0 & \text{sinon} \end{cases}$$

Ce modèle est représenté par une fonction objectif (Eq. 1.1), une contrainte dite de *knapsack* ou de capacité (Eq. 1.2) et les contraintes d'intégralité sur les variables de décision (Eq. et 1.3). L'objectif du problème est de sélectionner un sous-ensemble d'éléments, permettant de maximiser la somme totale des profits tout en satisfaisant la contrainte de capacité. Notons qu'afin d'éviter les cas triviaux, on suppose que :

- Toutes les valuers $c, p_i, w_i, \forall i = 1, \ldots, n$, sont des entiers positifs.
- $\sum_{i \in n} w_i > c$, permettant d'éviter les solutions triviales.

1.2.1.3 Quelques méthodes de résolution pour le sac à dos

Dans cette section, nous rappelons brièvement quelques méthodes exactes et approchées qui permettent de résoudre, d'une manière générale, les problèmes de l'optimisation combinatoire et spécifiquement le problème du sac à dos classique.

1.2.1.4 Méthodes heuristiques

Parmi les méthodes heuristiques, nous citons ici une méthode basée sur le principe glouton (voir., Dasgupta *et al.* [17]). Les méthodes gloutonnes sont une classe de méthodes heuristiques qui construisent une solution en se concentrant entièrement sur une amélioration immédiate sans tenir compte de l'avant. En d'autres termes, l'algorithme choisit toujours un optimum local dans l'espoir de parvenir à une solution optimum global. Les algorithmes gloutons ne donnent pas forcement des solutions optimales. Néanmoins, ils sont très puissants et fonctionnent correctement pour plusieurs problèmes d'optimisation combinatoire. Depuis des décennies, ils sont la façon la plus immédiate pour déterminer une solution réalisable.

Dans ce qui suit, nous présentons, une méthode gloutonne basée sur les principes proposés par Dantzig (voir., Dantzig [16]) pro résoudre le 0-1 KP.

Premièrement, tous les éléments sont triés par ordre décroissant en fonction de leur rapport du profit sur le poids (p_i/w_i), tel que:

$$\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq ... \geq \frac{p_n}{w_n}$$

Ainsi, à chaque étape, sélectionner de façon gloutonne un article selon l'ordre défini précédemment. Si l'élément est recevable, cela veut dire si son poids ne dépasse pas la capacité du sac après fixation des autres éléments, alors, il est placé dans le sac à dos. Sinon, nous sélectionnons l'élément suivant qui peut être recevable, et ainsi de suite, jusqu'à épuisement des éléments qui pourraient être placés dans le sac à dos (voir., Algorithme 1).

Il convient de noter que, cette méthode n'est pas la seule méthode gloutonne pour le problème de sac à dos. Ainsi, il existe plusieurs autres versions et améliorations par rapport à cette simple méthode (pour plus de details, le lecteur peut se référé à Kellerer *et al.*[38]).

Input: Une instance de P_{0-1KP}.
Output: une solution faisable $\overline{x}$.

1: Set $\overline{c}$ as a residal knapsack capacity, where $\overline{c} \Longleftarrow c$;
2: **for** $i = 1$ to n **do**
3: **if** $w_i \leq \overline{c}$ **then**
4: $\overline{x}_j = 1$;
5: $\overline{c} = \overline{c} - w_i$;
6: **else**
7: $\overline{x}_j = 0$;
8: **end if**
9: **end for**
10: **return** $\overline{x}$;

Algorithm 1: Une méthode gloutonne pour 0-1 KP

1.2.1.5 Calcul des bornes

Le calcul des bornes supérieures et inférieures permet d'encadrer la valeur de la solution optimale d'un problème à résoudre (c'est à dire limiter la plage de valeurs qui contiennent la solution optimale). Ainsi, ils peuvent donner une indication sur la façon dont il faut aller sur la construction d'une solution améliorée ainsi que des conseils sur la façon dont nous sommes loin de trouver la solution optimale. En effet, ces bornes peuvent être utilisées pour le développement de méthodes de résolution exacte basées en général sur la procédure de dénombrement (méthodes de séparation et évaluation).

La première borne supérieure du 0-1 KP a été présentée par Dantzig (voir., Dantzig [16]). Il a donné une méthode simple pour déterminer la solution associée à la relaxation de la programmation linéaire en nombre entier 0-1 KP :

- Produire une relaxation linéaire du $(LP(0-1KP))$ en relaxant chaque variable x_i, for $i = 1, ..., n$, dans l'intervalle $[0, 1]$.
- Résoudre le $(LP(0-1KP))$ en utilisant la procedure gloutonne appelée: *greedy procedure*.

$$\text{LP(0-1KP)} \begin{cases} \max & f(x) = \sum_{i=1}^{n} p_i x_i \\ \text{s.t.} & \sum_{i=1}^{n} w_i x_i \leq c, \\ & 0 \leqslant x_i \leqslant 1 \quad i \in N = \{1, ..., n\}. \end{cases}$$

Supposons qu'en utilisant une procédure gloutonne, les articles sont consécutivement insérées dans le sac à dos (après avoir ordonné les articles selon certains

critères), l'un après l'autre, jusqu'à la saturation du sac à dos. Ainsi, nous identifions le premier élément x_s qui ne rentre pas dans le sac à dos. Nous appelons cet article comme un élément critique de l'instance:

$$s = \quad min \left\{ i : \sum_{j=1}^{i} w_j > c \right\}$$

Cet article critique peut être utilisé comme une variable de décision afin d'obtenir une borne supérieure pour le problème courant. Ainsi, il réalise:

$$\sum_{i=1}^{s-1} w_i \leq c < \sum_{i=1}^{s} w_i$$

Alors le *LP(0-1KP)* peut être résolu suivant la procédure Dantzig (voir., Dantzig [16]), et la solution peut être formellement représentée comme suit:

$$\overline{x}_i = \begin{cases} 1 & for\ i = 1, ..., s-1 \\ \frac{c - \sum_{i=1}^{s-1} w_i}{w_s} & for\ i = s. \\ 0 & for\ \ i = s+1, ..., n, \end{cases}$$

Depuis, l'intégralité de la variable x_i, *pour* $i = 1, ..., n$ du 0-1 KP (c'est à dire toutes les solutions possibles sont des entiers), les solutions entières du 0-1 KP sont toujours plus petite que la solution du $LP(0-1KP)$:

$$f_{LP(0-1KP)}(x) = UB \geqslant f_{0-1KP}(x)$$

Et parce que, toutes les solutions possibles du 0-1 KP sont aussi réalisables pour le $LP(0-1KP)$. Nous déduisons la borne supérieure suivante pour le 0-1 KP:

$$UB = \sum_{i=1}^{s-1} p_i + \lfloor \frac{c - \sum_{i=1}^{s-1} w_i}{w_s} p_s \rfloor.$$

Où $\lfloor x \rfloor$ représente le plus grand entier non supérieur à x.

Cette borne supérieure est connu comme borne de Dantzig (pour plus de details, le lecteur peut se référer à Martello et Toth [45]).

1.2.1.6 Méthodes de coupes

Les méthodes de coupes proposées par Gomory (voir., cf., Gomory *et al.* [28]), sont des outils efficaces pour trouver des solutions entières à la programmation linéaire. Le principe est d'affiner itérativement la fonction objectif en ajoutant des réductions (ou nouvelles contraintes). Une coupe est une nouvelle contrainte qui exclut une partie de l'espace de recherche (c'est à dire réduit l'espace de solution réalisable). Ceci vise à réduire le nombre de calcul pour la recherche d'une meilleure solution entière globale (la meilleure solution entière).

L'algorithme 2 introduit un algorithme de coupe afin de résoudre le 0-1 KP. L'algorithme applique de manière itérative trois étapes principales: d'abord résoudre la relaxation de la programmation linéaire du $LP(0-1KP)$ et obtenir une solution optimale $x^{'}$. Ensuite, si $x^{'}$ est un entier, arrêter avec une solution entière optimale pour 0-1 KP. Sinon, enfin, générer une nouvelle contrainte d'inégalité et l'ajouter au $LP(0-1KP)$ et aller à la première étape. L'algorithme se termine après un nombre fini d'itérations et/ou lorsqu'une solution entière optimale x^* est obtenue.

Input: Une instance de 0-1 KP.
Output: La meilleure solution entière x^*.

1: Obtain the relaxation of IP, let the resulting linear problem denoted as $LP(0-1KP)$;
2: Let $x^* = NULL$
3: **repeat**
4: Solve $LP(0-1KP)$ and let $x^{'}$ be the resulting optimal solution;
5: **if** $x^{'}$ is integer **then**
6: $x^* = x^{'}$;
7: Stop;
8: **else**
9: Generate a cut (new constraint);
10: Add this new constraint to the $LP(0-1KP)$.
11: **end if**
12: **until** finite number of iteration and/or x^* is optimum integer solution
13: **return** x^*: La meilleure solution entière;

Algorithm 2: un algorithme de coupe pour résoudre 0-1 KP.

1.2.1.7 Méthodes de résolution exacte

De nombreuses méthodes de résolution exactes ont été proposées pour la résolution des problèmes de l'optimisation combinatoire. Parmi ces méthodes, nous pouvons citer: *branch & bound, branch & cut et la programmation dynamique.*

1.2.1.8 La méthode du *branch* & *bound*

Le principe de cette méthode est basé sur l'énumération complète de l'espace de recherche d'un problème donné pour trouver la meilleure solution (voir., Land *et al.* [40]; Martello *et al.* [46]). L'énumération est une structure arborescente de la recherche de solution. Chaque noeud de l'arbre sépare l'espace de recherche en deux sous-espace de recherche, et ainsi de suite, jusqu'à l'exploration complète de l'espace de recherche (voir., Dasgupta *et al.* [17]).

Ainsi, il y'a trois strategies dans la méthode du *branch* & *bound* qui sont:

- Stratégie de séparation.
- Stratégie d'évaluation.
- Stratégie de sélection de noeud.

La technique de base du *Branch* & *Bound* est qu'il divise l'espace de recherche en de petits sous-espaces à explorer. La division appelée séparation[2] permet de créer de nouvelles branches dans l'arbre d'énumération. La stratégie de séparation est l'un des aspects les plus importants de la méthode du *Branch* & *Bound* car elle vérifie si le sous-espace contient une solution optimale. En d'autres termes, la stratégie de séparation pour un sous-espace est calculée et comparée à la meilleure solution trouvée jusque là. Pour les problèmes de maximisation, il faut surestimer la meilleure valeur de la solution. Sinon, le sous-espace ne peut pas contenir la solution et donc il est jeté. Le contrôle de la stratégie de séparation de noeud permet de contrôler comment choisir le noeud suivant dans l'arbre de recensement de ramification. Il existe plusieurs stratégies, les plus populaires pour la sélection de noeud sont: (a) en profondeur d'abord: choisi que parmi l'ensemble des nœuds venez de créer, et (b) meilleur premier: choisir le noeud ayant la meilleure valeur n'importe où sur l'arbre lié.

Le premier algorithme *Branch* & *Bound* pour le 0-1 KP a été proposé par Kolesar (voir., Kolesar [39]). Plusieurs autres développements ont été proposés plus tard (pour plus de détails, Horowitz *et al.* [37]; Martello *et al.* [46]).

1.2.1.9 La méthode du *Branch and cut*

La méthode du *Branch and cut* est aussi très intéressante pour la résolution de problèmes de l'optimisation combinatoire d'une manière générale. La méthode *branch* & *cut* est le résultat de l'intégration entre méthode de coupe et la méthode de *branch* &*bound* (voir., Resende *et al.* [60]). La méthode de coupe est un outil efficace qui conduit à une forte réduction de la taille de l'arbre de recherche pour l'approche de *branch* &*bound*. Ainsi, la méthode de *branch* & *bound* peut être accélérée par l'emploi d'un système de plan de coupe, (pour plus de details, le lecteur peut se référer à: Crowder *et al.* [14]; Balas *et al.* [6]; Balas *et al.* [5]).

[2] branching

1.2.1.10 La programmation dynamique

La technique de programmation dynamique est une approche générale qui apparaît comme un outil utile pour résoudre divers problèmes en optimisation combinatoire. L'idée de base qui se trouve derrière cette technique a été introduite par Bellman (voir., Bellman [9]). Cette approche consiste à: (i) décomposer un problème en sous-problèmes plus simples, (ii) résoudre ces sous-problèmes et (iii) combiner leurs solutions afin de trouver une solution globale. Pour plus de détails, nous recommandons le livre de Bellman [10] qui donne une introduction générale claire concernant la programmation dynamique.

1.2.1.11 L'algorithme exact le plus connu pour le 0-1 KP

Martello *et al.* [44] ont présenté une approche basée sur une méthode exacte, pour résoudre le 0-1 KP. A notre connaissance, cette approche est le meilleur algorithme connu surpassant toutes les méthodes exactes pour le 0-1 KP. Cet algorithme peut être considéré comme une combinaison de deux méthodes exactes: (i) la programmation dynamique et (ii) le *branch & bound*, avec l'utilisation d'une forte évaluation. L'algorithme s'est avéré efficace dans la résolution de toutes les instances de test classiques, avec un maximum de 10000 variables, dans un temps de calcul raisonnable.

1.2.2 Problème du sac à dos avec contraintes disjonctives

Le problème de sac à dos avec contraintes disjonctives plus connu sous sa nomination anglophone "*Disjunctively Constrained Knapsack Problem: DCKP*" est aussi appelé le problème du sac à dos avec le conflit de graphe (voir., Pferschy *et al.* [52]).

Le DCKP a été introduit par Yamada *et al.* (voir., Yamada *et al.* [69]) comme un problème dans lequel un élément peut être incompatible avec d'autres éléments. Il existe une grande variété d'applications qui peuvent être modalisées sous forme de DCKP. Ici, nous considérons deux exemples. Ainsi, le premier exemple est rencontré dans le domaine de la gestion financière: supposons que nous avons n projets disponibles. Chaque projet j est caractérisé par un profit p_j, un coût c_j, et que certains projets sont incompatibles avec d'autres projets et enfin, nous disposons de seulement c dollars. Le problème est de déterminer l'ensemble des projets compatibles qu'on peut réaliser avec la somme limitée c. Ici, ce problème peut être modélisé comme un problème d'optimisation combinatoire, en particulier le DCKP. Le deuxième exemple, apparaît dans le transport et concerne le déplacement des personnes handicapées, des personnes âgées, et d'autres citoyens ayant des besoins particuliers. Ainsi, les personnes handicapées ont besoin dans la plupart des cas d'être servi par rapport à des conditions particulières. Par exemple, certains d'entre eux ne peuvent pas prendre le même bus

avec d'autres. En d'autres termes, un bus ne peut pas servir les personnes incompatibles en même temps. Formellement, ce problème peut être simulé comme un DCKP. En effet, toute application qui peut être formulée sous la forme du 0-1 KP peut être simulée sous forme de DCKP s'il y'a des incompatibilités entre les problèmes.

1.2.2.1 Définition

Une instance de DCKP, est caractérisée par un ensemble I de n éléments, un sac à dos (knapsack) ayant une capacité fixe c. Chaque élément i est caractérisé par: un profit p_i et un poids w_i. En plus, certains couples d'éléments pourraient être incompatibles[3],c'est à dire que la sélection d'un élément du couple dans la solution exclut automatiquement l'autre élément. Nous avons noté l'ensemble de couples d'éléments incompatibles comme E, où $E \subseteq \{(i,j) \in I \times I, i < j\}$, tel que, $(i,j) \in E$. Les éléments i et j ne sont pas autorisés à être inclus simultanément dans le sac à dos.Ce rapport est supposé être réciproque, c'est à dire, $(i,j) \in E \Leftrightarrow (j,i) \in E$ (voir., Yamada *et al.* [69]).

Considérons l'exemple suivant (cf., Figure 1.3):

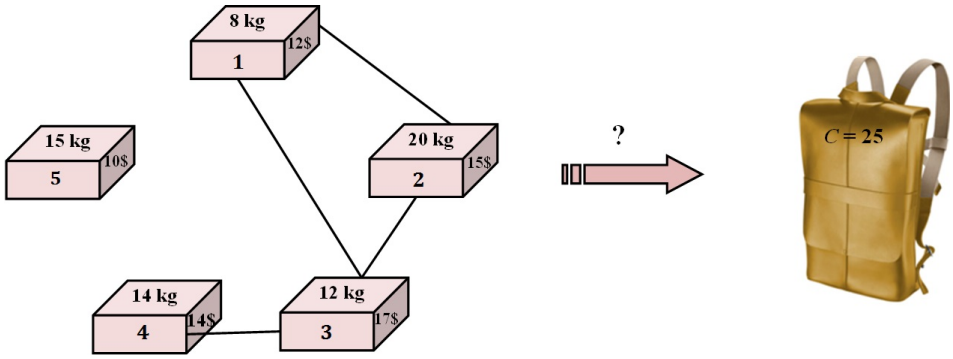


Figure 1.3: Le problème du sac à dos avec des contraintes disjonctives

Nous avons cinq éléments $I = \{1,2,3,4,5\}$. Chacuns de ces éléments a un profit respectivement, $p_1 = 12\$, p_2 = 15\$, p_3 = 17\$, p_4 = 14\$, p_5 = 10\$$, et un poids respectivement, $w_1 = 8$ kg, $w_2 = 15$ kg, $w_3 = 12$ kg, $w_4 = 14$ kg, $w_5 = 15$ kg. Sachant que la capacité du sac à dos est $c = 25kg$. En outre, nous avons quatre couples d'éléments incompatibles, où $E = \{(1,2),(2,3),(3,4),(1,3)\}$. Si nous plaçons l'élément 1 dans la solution, cela exclut les éléments 2 et 3. Cependant, nous pouvons placer l'élément 4 avec une valeur objective égale à 26.

En ce qui concerne notre exemple illustré dans la Figure 1.3, une solution optimale peut être trouvé en plaçant les éléments 1 et 4 dans le sac à dos de sorte que le profit total sera 26\$ avec un poids total de 22 kg.

[3]disjunctive

1.2.2.2 Modélisation du problème du sac à dos avec des contraintes disjonctives

Soit x_i la variable de décision associée à l'élément i tel que, x_i prend la valeur 1 si l'élément i est sélectionné et 0 sinon. Dans ce cas, nous pouvons formuler les contraintes disjonctives comme:

$$x_i + x_j \leq 1, \quad \forall (i,j) \in E$$

La contrainte du sac à dos est présentée comme suit:

$$\sum_{i=1}^{n} w_i x_i$$

L'objectif consiste à sélectionner un sous-ensemble d'éléments tels que en même temps, la capacité du sac à dos et les contraintes disjonctives soient satisfaites . Ceci en maximisant la somme totale des profits. Ensuite, la programmation linéaire en nombres entiers du DCKP peut être formulée comme suit:

$$Max: \quad f(x) = \sum_{i=1}^{n} p_i x_i \tag{1.4}$$

$$s.t. \quad \sum_{i=1}^{n} w_i x_i \leq c \tag{1.5}$$

$$x_i + x_j \leq 1, \qquad \forall (i,j) \in E \tag{1.6}$$

$$x_i \in \{0,1\} \qquad \forall i \in I = \{1,...,n\} \tag{1.7}$$

Dans cette programmation linéaire en nombres entiers, nous avons quatre types d'équations. La première équation (1.4) représente la fonction objective où le but est de maximiser autant que possible sa valeur sous les trois contraintes (équations 1.5, 1.6 et 1.7). La contrainte (1.5)) représente la capacité du sac à dos et impose que la somme totale des poids des éléments placés dans le sac à dos ne dépassent pas la capacité du sac à dos. La contrainte (1.6) assure la compatibilité de tous les éléments. La contrainte (1.7) impose d'inclure en totalité ou aucun un élément.

Sans perte de généralité, nous supposons que:

- Toutes les données à l'entrée $c, p_i, w_i, \forall i \in n$ sont des entiers strictement positifs.
- $\sum_{i \in n} w_i > c$ pour éviter des solutions triviales.

Nous pouvons noter que le DCKP est un problème de l'optimisation combinatoire (voir., Garey *et al.* [26]; Arora *et al.* [4]). Le DCKP devient un 0-1 KP,

lorsque, l'ensemble $E = \emptyset$, et il devient un problème dit: *weigthted independent set*, quand, la contrainte de capacité du sac à dos est omise.

1.2.2.3 DCKP et méthodes de résolution

Dans cette section, nous citons les méthodes de résolution utilisées pour résoudre le problème de sac à dos, avec contrainte disjonctive.

1.2.2.4 Les méthodes heuristiques

Yamada *et al* (voir., Yamada *et al.* [69]) ont proposé deux méthodes heuristiques: la première heuristique est une méthode gloutonne qui produit une solution réalisable pour le DCKP. La deuxième heuristique est basée sur une recherche locale qui améliore la qualité de la solution obtenue par la méthode gloutonne.

1.2.2.5 L'algorithme glouton ou *Greedy algorithm*

L'algorithme glouton proposé par Yamada *et al.* [69] est simple, il construit une solution initiale en n étapes basées sur le principe d'un choix optimal au niveau local. Ainsi, chaque étape i $i \leq n$ selon la fixation d'une variable de décision x_i. Notez que, les variables $x_1, ...x_{i-1}$ ont été déjà fixé auparavant. En effet, si le poids de l'élément i ne dépasse pas la capacité restante du sac à dos et que cet élément est compatible avec tous les éléments précédemment fixés, alors, l'algorithme place cet élément dans le sac à dos (i.e., fixer $x_i = 1$; Sinon $x_i = 0$). Les principales étapes de ce procédé sont présentées dans l'algorithme 3.

Input: Une instance de P_{DCKP}.
Output: une solution faisable $\overline{x}$.

1: Set $\overline{c}$ as a remainging knapsack capacity, where $\overline{c} \Longleftarrow c$;
2: **for** $i = 1$ to n **do**
3: **if** $w_i \leq \overline{c} and (i : i < j, x_i = 1), (i, j) \in E$ **then**
4: $\overline{x}_j = 1$;
5: $\overline{c} = \overline{c} - w_i$;
6: **else**
7: $\overline{x}_j = 0$;
8: **end if**
9: **end for**
10: **return** $\overline{x}$;

Algorithm 3: Algorithme glouton pour DCKP

1.2.2.6 L'algorithme de la recherche locale

La recherche locale proposée par Yamada *et al.* [69] applique la procedure 2-opt, afin d'améliorer la solution initiale obtenue à partir de l'algorithme glouton. Cette amélioration consiste à remplacer un élément par d'autres éléments. Cet échange est autorisé que si la solution obtenue est meilleure que l'actuelle. Soit $\overline{x}$ une solution réalisable du DCKP, et $I(\overline{x})$ est l'ensemble des variables de décision fixées à 1. Les auteurs définissent $N(\overline{x})$ comme le voisinage de la solution $\overline{x}$ qui est représenté par un ensemble de solutions obtenues en échangeant un certain nombre éléments de $I(\overline{x})$ avec d'autres éléments de l'instance. L'algorithme 4 illustre les principales étapes de cet algorithme.

Input: Une solution initiale $\overline{x}$.
Output: Une solution ameliorée ($x^{'}$).

1: Let NS be a neighbourhood solution.
2: Set $x^{'} \Longleftarrow \overline{x}$;
3: **while** $\exists\ NS\ \in N(\overline{x})$ **do**
4: **if** $NS > x^{'}$ **then**
5: $x^{'} = NS$;
6: **end if**
7: **end while**
8: **return** $x^{'}$;

Algorithm 4: 2-*opt* algorithme

Cette amélioration dans le voisinage $N(\overline{x})$ est effectuée comme suit: à partir de la solution obtenue par l'algorithme glouton. Le 2-opt améliore cette solution dans son voisinage $N(\overline{x})$ en appliquant, itérativement, deux étapes:

- Mettre $x_i = 0$ pour chaque $x_i \in I$, remplacé cet élément avec d'autres.
- Choisissez la meilleure solution trouvée jusqu'à présent.

Ensuite, mettre à jour la solution et répéter le processus avec la nouvelle solution. Ce processus s'arrête avec la meilleure solution locale trouvée.

1.2.2.7 Les métaheuristiques

Les métaheuristiques peuvent être considérées comme des heuristiques puissantes et avancées et peuvent être généralisées à de nombreux problèmes d'optimisation combinatoire. Elles permettent de guider la procédure de solution à explorer l'espace au-delà de la solution optimale locale, dans l'espoir d'échapper à une

série d'optima locaux afin de produire efficacement des solutions de haute qualité. Pour les dernières décennies, les métaheuristiques ont été largement développées et appliquées à une variété de problèmes d'optimisation combinatoire (voir., Blum *et al.* [11]; Osman *et al.* [49]; Vaessens *et al.* [67]).

Dans la littérature, plusieurs heuristiques et métaheuristiques ont été proposées. Ainsi, Hifi et Michrafy [30] ont proposé un algorithme réactif basé sur la recherche locale. L'algorithme proposé se compose en deux phases. La première phase donne une solution de départ. La deuxième phase tente d'améliorer la solution obtenue par la première phase en combinant la stratégie de dégradation avec la liste de mémoire. La stratégie de dégradation effectue un échange entre plusieurs éléments afin de diversifier la recherche. Ceci permet de changer la direction de la recherche afin d'explorer différentes régions. Pendant ce temps, la liste de mémoire est utilisée pour échapper aux optima locaux.

Akeb *et al.* [3], ont étudié l'utilisation d'une technique efficace de ramification locale (ou local branching technique). La ramification locale est une approche rentable, introduite par Fischetti et Lodi (voir., Fischetti *et al.* [23]), qui vise à trouver rapidement une solution presque optimale permettant d'accélérer le processus de recherche globale. Le principe de l'algorithme de branchement local peut être décrit comme suit: à partir d'une solution initiale, il explore le voisinage de la meilleure solution actuelle jusqu'à ce qu'une meilleure solution soit trouvée, puis réitérer le processus en utilisant la solution meilleure comme la solution initiale. Cela permet donc de chercher l'optimum dans la région réalisable inexploré. Ces processus sont réitérés jusqu'à la condition d'arrêt final. Enfin, l'algorithme retourne la meilleure solution trouvée. Toutefois, dans ce document, trois versions de l'algorithme sont considérées. La première version est basée sur la ramification locale qui utilise une solution de départ fournie par une procédure spéciale de solution d'arrondie. La deuxième version utilise une procédure de solution à deux phases qui peut être considérée comme une combinaison de la procédure d'arrondissement avec une procédure exacte tronquée. La troisième version est une adaptation du deuxième algorithme où une stratégie de diversification est considérée pour enrichir l'espace de recherche.

Hifi *et al.* [34], proposent une adaptation de la recherche dispersée (*Scatter Search: SS*) pour résoudre approximativement le DCKP. Le *Scatter Search: SS* est une métaheuristique, qui peut être considérée comme une méthode évolutive, (voir., Glover [27]) basée sur la combinaison des solutions utilisant des combinaisons linéaires pour atteindre de nouvelles solutions à travers des générations successives. L'approche fonctionne sur un ensemble de solutions données représentant l'ensemble de référence, en combinant ces solutions de manière systématique dans le but de créer de nouvelles solutions. En général, le mécanisme principal pour combiner des solutions est telle qu'une nouvelle solution est créé à partir de la combinaison linéaire des deux autres solutions et, dans certains cas, une telle nouvelle solution peut être atteinte en associant plus de deux solutions.

Dans [33], Hifi et al ont présenté une adaptation de la recherche dispersée proposé par Hifi *et al.* dans [34] pour résoudre approximativement le DCKP.

L'adaptation est basée sur l'application de cette approche à un problème équivalent au DCKP proposé par Hifi *et al.* dans [31] qui est renforcée en utilisant deux types de contraintes valides: le premier type de contrainte ajouter au modèle équivalent au DCKP est une borne sur la valeur de la fonction objective minimale, et le deuxième type de contrainte représente une contrainte de cardinalité valide. La borne exige que, la valeur de la fonction objectif soit au moins aussi grande que la limite inférieure qui est la valeur de la meilleure solution possible trouvée jusqu'ici. Les deux contraintes de cardinalité imposent que le nombre d'éléments inclus dans le sac à dos soit délimité par deux valeurs.

Finalement, Hifi in [29] ont proposé un algorithme de recherche itérative arrondi qui peut être considéré comme une alternative aux approches utilisées dans [34] et [3] pour résoudre approximativement le DCKP. En Effet, dans [3], les auteurs ont montré que la fixation d'une partie des variables de décision et la résolution du problème réduit avec ramification locale, permet d'améliorer la qualité des solutions atteinte, mais au détriment du temps d'execution. D'autre part, Hifi et Otmani dans [33] ont donné un modèle équivalent pour le DCKP avec des contraintes de cardinalité supplémentaires délimitées par les deux bornes inférieures et supérieures qui permet d'accélérer le processus de recherche et en même temps d'améliorer la qualité de certaines solutions. En conséquence, les auteurs ont présenté une méthode mixte combinant les deux approches et ont présenté un algorithme de recherche itérative d'arrondi qui combine: (i) la technique de la fixation de variable en utilisant le procédé d'arrondi appliqué à la relaxation linéaire, (ii) introduction de contraintes valides successives en délimitant la fonction objectif.

1.2.2.8 Les méthodes de résolution exactes

Du fait de la complexité du DCKP, très peu d'algorithmes de résolution exactes se sont attaqués au DCKP, la plupart des résultats sur ce dernier, sont basés sur des heuristiques (voir., Senisuka *et al.* [64]). Néanmoins, il existe des papiers proposant des méthodes exactes pour le DCKP. Parmi ces méthodes, nous citons par exemple, le papier de Yamade *et al.* [69]. Leurs algorithme est basé sur une recherche par énumération implicite, combinée avec une méthode de reduction d'intervalle. Cet algorithme utilise la solution obtenue par la méthode heuristique comme une borne inférieure initiale.

Hifi et Michrafy (voir., Hifi *et al.* [31]), ont proposé trois algorithmes exacts, dans lesquels, les stratégies de reduction, un modèle équivalent et la recherche dichotomique, coopèrent ensemble pour résoudre le DCKP. Le premier algorithme réduit la taille du problème original en commençant avec une borne inférieure et successivement, il résout les DCKPs relaxés. Le second algorithme combine une stratégie de reduction avec la recherche dichotomique afin d'accélérer le processus de recherche. Le troisième algorithme résout les instances avec un grad nombre de contraintes disjonctives.

1.3 Quelques problèmes de l'optimisation combinatoire

Il existe beaucoup d'autres problèmes d'optimisation combinatoire appartenant à la famille du problème de sac à dos. Dans cette section, nous citons certains d'entre eux.

1.3.1 Le problème du sac à dos en max-min avec multi-scenario

Le *Multi-Scenario max-min Knapsack Problem, noté: MSKP*, est une variante du problème de sac à dos classique (voir., Yu [71]; Pinto *et al.* [53]). Ce problème est caractérisé par un ensemble d'éléments i avec un profit (p_i) et le poids (w_i), et by la capacité limitée du sac à dos c. Ici, un scenario s est défini comme l'ensemble des profits qu'on applique à chaque élément i. Tel que, un profit p_i d'un élément i dépend du scenario s qui est considéré. Le *multi-scenario max-min knapsack problem* peut être exprimé comme suit:

$$max \quad \min_{s=1,...,S} \left\{ \sum_{i=1}^{n} p_i^s x_i \right\} \tag{1.8}$$

$$s.t. \quad \sum_{i=1}^{n} w_i x_i \leq c \tag{1.9}$$

$$x_i \in \{0,1\} \qquad \forall i = \{1,...,n\} \tag{1.10}$$

L'objectif est de trouver l'ensemble des éléments à placer dans le sac à dos et dont le profit total est maximisé sous le pire scénario possible.

1.3.2 Problème du sac à dos avec contraintes de précédences

Une variante du 0-1 KP classique, considère que, les éléments sont partiellement ordonnés par un ensemble de relations de précédences (noté: contraintes de précédences). Cette variation appelée comme le problème de sac à dos avec une contrainte de priorité (voir., Samphaiboon *et al.* [63]; You *et al.* [70]). Cependant, le problème peut être représenté mathématiquement comme un modèle de programmation binaire suivant:

$$Max: \quad f(x) = \sum_{i=1}^{n} p_i x_i \tag{1.11}$$

$$s.t. \quad \sum_{i=1}^{n} w_i x_i \leq c \tag{1.12}$$

$$x_i \geqslant x_j, \qquad \forall (i,j) \in E \tag{1.13}$$

$$x_i \in \{0,1\} \qquad \forall i \in I = \{1,...,n\} \tag{1.14}$$

Où, E représente les relations de précédence entre les éléments. Tel que, $(i, j) \in E$, l'élément j peut être considérée que lorsque l'élément i a déjà été placé dans le sac à dos.

1.3.3 Le problème du sac à dos borné

Le problème du sac à dos, borné, est une variante du 0-1 KP. Dans cette variante, il existe une quantité limitée m_i de chaque article de type i. Le problème du sac à dos, borné peut être exprimé comme:

$$Max: \quad f(x) = \sum_{i=1}^{n} p_i x_i \tag{1.15}$$

$$s.t. \quad \sum_{i=1}^{n} w_i x_i \leq c \tag{1.16}$$

$$x_i \in \{0, 1, ..., m_i\} \qquad \forall i = \{1, ..., n\} \tag{1.17}$$

où, x_i est la quantité de chaque type d'article à placer dans le sac à dos afin d'obtenir la meilleure valeur objective.

Il y'a plusieurs autres problèmes connexes au sein de la famille des problèmes de sac à dos: le problème de sac à dos de choix multiples and le multiple problème de sac à dos avec choix multiples , ...etc. Chaque variante est différente de l'autre par la répartition des éléments dans un ou plusieurs sac à dos (Pour plus de details voir., Martello *et al.* [46]).

Chapter 2

General introduction

Many practical situations can be modeled as combinatorial optimization problems. Among these problems, we can find some problems belonging to the knapsack family. In order to tackle such problems, two directions of research can be followed: (i) solving the given problem using exact methods and/or (ii) searching near optimal solutions using heuristic methods. An exact algorithm tries to find an optimal or a set of optimal solutions for a given problem. For the problems belonging to the knapsack family, an optimal solution can be found using branch and bound, branch and cut, and/or dynamic programming methods. Nevertheless, for large-scale problems, an exact method might need exponential computation time. This often leads to a solution time that is too high for the practical situation. Thus, the development of heuristic methods has received more attention in the last decades. Note that there are other solution methods that combine exact and heuristic methods. These methods, noted as hybrid methods, represent a powerful tool for solving combinatorial optimization problems.

Herein, the neighborhood search methods are considered for approximating a particular problem belonging to the knapsack family: the Disjunctively Constrained Knapsack Problem (abbreviated as $DCKP$). Such a problem has a large variety of applications, especially in transport logistics and financial management. As an example, suppose a real-life problem appears in transport with regards the movement of those with disabilities, seniors, and other citizens with unique needs. Since the disabled people have their own issues, in most cases, they need to be served with respect to special conditions. For example, some of them cannot take the same bus as some others. In other words, a bus cannot serve those incompatible persons at the same time. Formally, this problem can be modeled as the disjunctively constrained knapsack problem.

This thesis presents sequential and parallel algorithms based upon neighborhood search techniques. These approaches are tailored for optimizing large-scale instances of a combinatorial problem called the disjunctively constrained knapsack problem. This work may also benefit the development of optimization techniques that can be applied for tackling large-scale instances of other combinatorial optimization problems.

Thesis organization

The thesis contains three main parts. The first part discusses the state of the art. The second part is dedicated to the development of two sequential algorithms. The third part is devoted to a parallel solution approach.

More specifically, the thesis is organized as follows. The first part (Chapter 3) presents the state of the art of some problems belonging to the knapsack family. First, it begins by describing the classical knapsack problem. Second, a brief description of some well-known exact and approximate solution methods is presented. These methods can be used for solving a wide variety of combinatorial problems. Third and last, the $DCKP$ is illustrated along with some solution methods presented in the literature.

The second part contains two chapters (Chapter 4 and Chapter 5).

In Chapter 4, we present our first sequential algorithm tailored for solving large-scale combinatorial optimization problems. This algorithm can be viewed as a random neighborhood search method. In fact, the goal is to present a fast algorithm that yields interesting solutions within a short average running time. We use a combination of neighborhood search techniques in order to produce quick solutions with high quality. The proposed algorithm is composed of two main steps: a re-optimizing step and a destroying step. Indeed, the re-optimizing step consists of a two-phase procedure. The first phase serves to provide a feasible solution while the second phase serves to improve the quality of this solution. The destroying step performs a random destroying strategy in order to yield a new neighborhood solution. This step is considered as the most interesting step in the algorithm that controls its efficiency and responsible of the enlarging of the solution's search space. In addition, it helps to escape from a series of local optimum. In fact, the solution provided by the destroying step is treated as a reduced problem, which in turn will be subject to optimization.

Chapter 5 presents the second sequential algorithm for the DCKP. In fact, the goal is to present an adaptive algorithm that guides the search process in the feasible solution space towards high quality solutions. The algorithm combines both guided and decomposition strategies. First, the decomposition strategy is used in order to construct a series of sub-problems to solve. Second and last, the algorithm uses an ant colony optimization system to simulate the guided search. In fact, the decomposition strategy provides two sub-problems: the weighted independent set problem and the classical binary knapsack problem. The weighted independent set problem is approximately solved using an ant colony optimization system and the classical binary knapsack problem is solved using an exact algorithm. Then, the decomposition phase reaches a feasible solution for the DCKP. Once a feasible solution is obtained, a descent method is applied for improving the solution at hand. In order to escape from a local optimum, a diversification strategy is introduced which is based on a modified ant colony optimization system. During the search process, the ant colony optimization system tries to diversify and to enhance the solutions by using some information

collected from the previous iterations. This process is repeated until stopping criteria are performed.

The third part is devoted to the parallel implementation of an large neighborhood search approach. It contains two chapters (Chapter 6 and Chapter 7).

Chapter 6 presents the state of the art of the parallelism. First, it begins by introducing the sequential and parallel programming. Second, a brief description of some well-known parallel machine architectures is given. Third, some parallel approaches are discussed. Fifth and last, some well-known libraries used for building parallel programming are described.

Chapter 7 describes a parallel algorithm tailored for solving large-scale instances of DCKP. This algorithm is a parallel random neighborhood search method. Indeed, the goal is to exploit the parallelism for providing a fast algorithm that yields high quality solutions. The principal idea used by the parallel algorithm is based on exploring simultaneously different neighborhoods by several processors. Each processor adopts its own random strategy to yield its own neighborhoods according to its internal information. These processors share then the best solutions obtained in order to approach the most promising neighborhood through the search process. The proposed model designed using the MPI[1]. Messages are used to move data from the address space of one process to that of another process through cooperative operations on each process. The effectiveness of MPI allows us to build a flexible message passing parallel programming model.

Finally, Chapter 8 gives a general conclusion in which we summarize the obtained results and it gives some perspective and future works in this area.

[1]MPI stands for Message Passing Interface which is a message passing library standard for the message-passing parallel programming model.

Part I

Knapsack problems and solution methods

Chapter 3

State of the art

Contents

This chapter presents the state of the art of some combinatorial optimization problems belonging to the knapsack family. It is organized as follows. Section 3.1 presents an introduction to the knapsack family. In section 3.2, a brief reminder about the classical 0-1 knapsack problem is presented together with some of the well-known solution methods developed for solving some combinatorial optimization problems. Section 3.3 illustrates a variation of the 0-1 knapsack problem when treated with special constraints known as the disjunctive constraints. Section 3.4 introduces some other combinatorial optimization problems related to the knapsack family. Finally, section 3.5 summarizes the contents of the chapter.

3.1 Introduction

Knapsack problems (namely KPs) have been studied for more than a century (cf., Matheows [48]), and more intensively after the work of Dantzig (cf., Dantzig [16]). That is because they possess numerous important applications in various domains, especially in transport logistics, industry, financial management. In other words, they are faced by many organizations, from small to large. More specifically, many problems belonging to the knapsack family often appear as a reduced or sub-problem in the solution procedures of more complex problems (cf., Hifi *et al.* [35]; Pisinger [54]). Thus, this makes them theoretical models of particular importance (cf., Pisinger *et al.* [57]).

In the literature, there are various variants of problems of knapsack type (cf., Figure 3.1). Among of them, in what follows, we cite some problems including the 0-1 knapsack problem, and the disjunctively constrained knapsack problem (the problem studied in this thesis). However, this family has been considered widely and cannot be fully covered here (for more details the reader can be referred to Martello *et al.* [46]; Dudzinski *et al.* [20]).

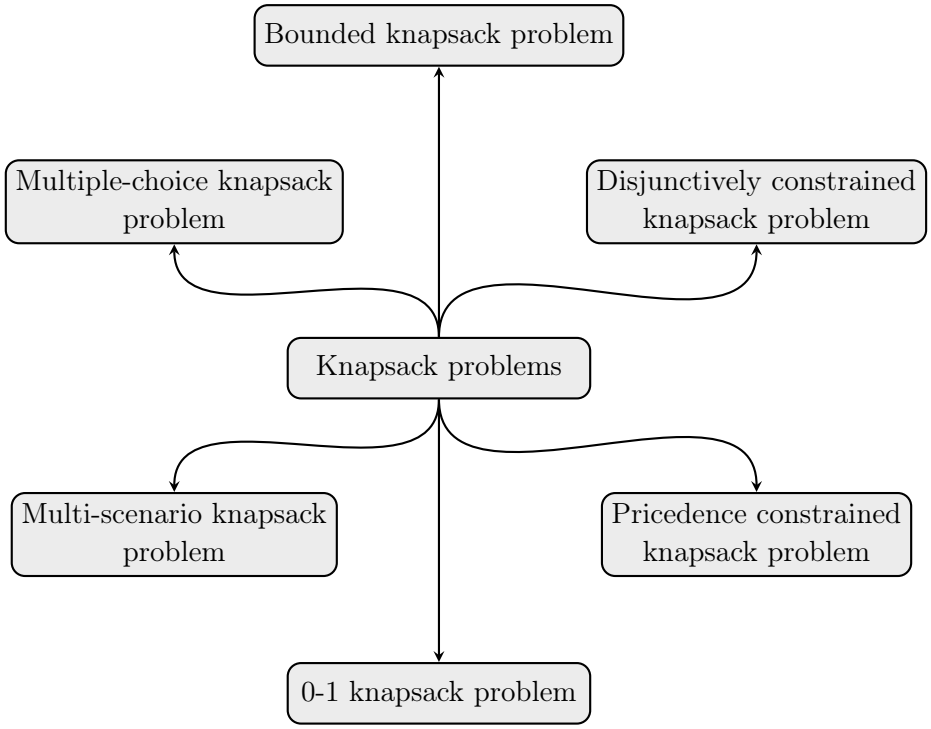


Figure 3.1: Some variants of *knapsack problems*.

3.2 0-1 Knapsack problem

The classical or binary knapsack problem (abbreviated as 0-1 KP) is one of the most well-known problems belonging to the combinatorial optimization family. The reasons for that may be explained by three facts (cf., Martello *et al.* [46]):

- It represents a simple integer linear programming model.
- It appears frequently as a sub-problem in more complex problems.
- It represents many practical situations.

There are a wide variety of applications that can be simulated as 0-1 KP in various domains, including: cargo loading, computer science (cf., Horowitz *et al.* [37]) and financial management (cf., Pisinger [54]). Herein, let's consider a simple example encountered in financial management: suppose that we have n projects and only c dollars are available. Each project j is characterized by a profit p_j and a cost c_j. The objective of the problem is to determine a set of projects to be performed with the limited finance c. Such a problem can be modeled as the 0-1 KP, which can be solved by using branch and bound, dynamic programming, hybrid algorithms (for more details the reader can be referred to Kellerer *et al.* [38]; Fayard *et al.* [21]; Balas *et al.* [7]; Martello *et al.* [45]; Martello *et al.* [47]).

3.2.1 Definition

Given a set I of n items and a knapsack with a fixed capacity c, where each item i is characterized by a profit p_i and a weight w_i, the objective of the problem is to select a subset of items so that the sum of the selected items' profits is maximized without exceeding the knapsack capacity c. For better understanding, consider the following example (cf., Figure 3.2):

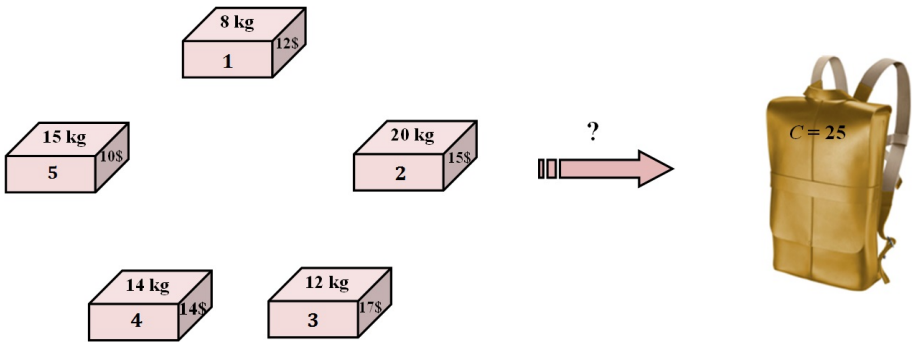


Figure 3.2: The classical knapsack problem

Let I be the set of 5 items such that $I = \{1, 2, 3, 4, 5\}$. The instance is characterized by a vector of profits $p = \{p_1, p_2, p_3, p_4, p_5\} = \{12\ \$, 15\ \$, 17\ \$, 14\ \$, 10\ \$\}$,

a vector of weights $w = \{w_1, w_2, w_3, w_4, w_5\} = \{8 \text{ kg}, 20 \text{ kg}, 12 \text{ kg}, 14 \text{ kg}, 15 \text{ kg}\}$, and a capacity $c =$ of 25 kg.

A feasible solution may consist of placing items 1 and 5 with the objective value equal to 22 \$. Another possible solution may consist of placing just item 2 with the objective value equal to 15 \$ or even just item 1 with the objective value equal to 12 \$. Thus, feasible solutions are those that are all subsets of items in which their total weight does not exceed the knapsack capacity c. One can observe that the number of possible solutions increased with the increasing of the items (I). In such a case, the aim is to choose the solution of maximum profit value among all feasible solutions (i.e. the optimal solution). Regarding the example, an optimal solution is that, containing items 1 and 3 with the objective value equal to 29 \$ and with a total weight equal to 20 kg.

3.2.2 Integer linear programming of 0-1 KP

The integer linear programming of the 0-1 KP can be stated as follows:

$$Max: \quad f(x) = \sum_{i=1}^{n} p_i x_i \tag{3.1}$$

$$s.t. \quad \sum_{i=1}^{n} w_i x_i \leq c \tag{3.2}$$

$$x_i \in \{0, 1\} \qquad \forall i \in I = \{1, ..., n\} \tag{3.3}$$

where x_i is a binary decision variable, such that:

$$x_i = \begin{cases} 1 & \text{if item } i \text{ is placed in the knapsack} \\ 0 & \text{otherwise.} \end{cases}$$

In this integer linear programming, we have three equations. The first equation (Equation 3.1) is the objective function where the goal is to maximize the value of total profit of items placed in the knapsack under two constraints (Equation 3.2 and Equation 3.3). The first constraint (Equation 3.2) is known as the capacity constraint, which imposes that the weights' sum of the selected items does not exceed the knapsack capacity. Meanwhile, the second constraint (Equation 3.3) is imposed on items that are to be included or not in the knapsack (it is not allowed to include a fractional item). In order to avoid trivial cases, it is assumed that:

- All input data $c, p_i, w_i, \forall i \in n$ are strictly positive integers.
- $\sum_{i \in n} w_i > c$ for avoiding trivial solutions.

3.2.3 0-1 KP and solution methods

In this section, we recall some approximate and exact solution methods dedicated to solve a large variety of combinatorial optimization problems including the 0-1 KP.

3.2.3.1 Heuristic method

Among heuristic methods, we cite here a method based on the principle of greed known as a greedy method (cf., Dasgupta *et al.* [17]). Greedy methods are a class of heuristic (or approximate) solution methods that construct a solution piece by piece focusing on an immediate improvement without considering the consequences. In other words, they make, always, a local optimum choice in the hope of reaching a global optimum solution. Greedy algorithms, normally, don't result in an optimal solution. Nevertheless, they work correctly for variant combinatorial problems since, they are the most immediate way to determine a feasible solution.

In what follows, we cite a simple greedy method based on the principles proposed by Dantzig (cf., Dantzig [16]) for tackling the 0-1 KP:

First, all items are sorted in decreasing order according to the ratio of profit over weights (p_i/w_i), such that:

$$\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq ... \geq \frac{p_n}{w_n}$$

Then, in each step, an item is selected greedily according to the previously defined order. If the item is admissible, meaning if its weight does not exceed the knapsack capacity after selection of the other items, then it is placed in the knapsack. Otherwise, we select the next item that may be admissible. This process is repeated until exhaustion of all the items (cf., Algorithm 5).

Input: An instance of 0-1 KP.
Output: A feasible solution $\overline{x}$.

1: Set $\overline{c}$ as a residal knapsack capacity, where $\overline{c} \Longleftarrow c$;
2: **for** $i = 1$ to n **do**
3: **if** $w_i \leq \overline{c}$ **then**
4: $\overline{x}_j = 1$;
5: $\overline{c} = \overline{c} - w_i$;
6: **else**
7: $\overline{x}_j = 0$;
8: **end if**
9: **end for**
10: **return** $\overline{x}$;

Algorithm 5: A Simple greedy procedure for 0-1 KP

It should be noted that there are several other improvements of this simple method (for more details the reader can be referred to Kellerer *et al.* [38]).

3.2.3.2 Bounds computing

The computation of the upper and lower bounds permits you to frame the value of the optimal solution of a problem being addressed (i.e. limits the range of values that contain the optimal solution). Thus, they can give an indication of how far away we are from the optimal solution and how one should proceed to construct an improved solution. Indeed, these bounds can be used for the development of exact solution methods based in general on an enumeration procedure.

The first upper bound for the 0-1 KP was presented by Dantzig (cf., Dantzig [16]). He gave a simple method to determine a solution to the relaxation of the integer linear programming of the 0-1 KP (cf., Section 3.2.2):

- Produce a linear programming relaxation (LP(0-1 KP)) by relaxing each decision variable x_i, for $i = 1, ..., n$, in the interval $[0, 1]$.
- Solve the LP(0-1 KP) using a greedy procedure.

The linear programming relaxation of the 0-1 KP can be stated as follows:

$$\text{LP(0-1 KP)} \begin{cases} \max & f(x) = \sum_{i=1}^{n} p_i x_i \\ \text{s.t.} & \sum_{i=1}^{n} w_i x_i \leq c, \\ & 0 \leqslant x_i \leqslant 1 \quad i \in N = \{1, ..., n\}. \end{cases}$$

Suppose that, by using a simple greedy procedure, the items are consecutively inserted into the knapsack one after the other (after ordering the items according to certain criteria), until its saturation. Then, we identify the first item x_s that does not fit in the knapsack. This item is named the critical item:

$$s = \quad min \left\{ i : \sum_{j=1}^{i} w_j > c \right\}$$

This critical item can be used as a decision variable in order to derive an upper bound for the current problem. Since, it realizes $\sum_{i=1}^{s-1} w_i \leq c < \sum_{i=1}^{s} w_i$.

Then, the LP(0-1 KP) can be solved following the Dantzig procedure (cf., Dantzig [16]), and the solution can be formally stated as follows:

$$\overline{x}_i = \begin{cases} 1 & \text{for } i = 1, ..., s-1 \\ \frac{c - \sum_{i=1}^{s-1} w_i}{w_s} & \text{for } i = s. \\ 0 & \text{for } \ i = s+1, ..., n, \end{cases}$$

Because of the integrality of the decision variables x_i of 0-1 KP, its integer solutions are always smaller than the optimal solution of the LP(0-1 KP):

$$f_{\text{LP(0-1KP)}}(x) = UB \geqslant f_{\text{0-1KP}}(x)$$

Also, because all feasible solutions of 0-1 KP are also feasible for LP(0-1 KP). We can derive the following upper bound for 0-1 KP:

$$UB = \sum_{i=1}^{s-1} p_i + \lfloor \frac{c - \sum_{i=1}^{s-1} w_i}{w_s} p_s \rfloor.$$

Where $\lfloor x \rfloor$ denotes the largest integer not greater than x. This upper bound is known as Dantzig bound (for more details the reader can be referred to Martello *et al.* [45]).

3.2.3.3 Cutting plane method

The cutting plane method, proposed by Gomory (cf., Gomory *et al.* [28]), is an effective tool for finding integer solutions to integer linear programming. The principle is to iteratively refine the objective function by adding cuts (or new constraints). A cut can be defined as a constraint that excludes a portion of the search space from consideration (i.e. reduces the space search of the feasible solution). This can reduce the computational efforts in the search process of finding a global optimum solution (the best integer solution).

Input: An instance of 0-1 KP.
Output: The best integer solution x^*.

1: Obtain the relaxation of IP, let the resulting linear problem denoted as LP(0-1 KP);
2: Let $x^* = NULL$
3: **repeat**
4: Solve LP(0-1 KP) and let $x^{'}$ be the resulting optimal solution;
5: **if** $x^{'}$ is integer **then**
6: $x^* = x^{'}$;
7: Stop;
8: **else**
9: Generate a cut (new constraint);
10: Add this new constraint to the LP(0-1 KP).
11: **end if**
12: **until** finite number of iteration and/or x^* is optimum integer solution
13: **return** x^*: the best integer solution;

Algorithm 6: A cutting plane algorithm

Algorithm 6 introduces a cutting plane algorithm for solving the 0-1 KP. The algorithm iteratively applies three main steps: first, solve the linear programming

relaxation LP(0-1 KP) and get an optimal solution $x^{'}$. Second, if $x^{'}$ is an integer, stop with an optimum integer solution for 0-1 KP. If not, the third and final step is to generate a new inequality constraint and add it to LP(0-1 KP) and go to the first step. The algorithm terminates after a finite number of iterations and/or an optimum integer solution is obtained x^{*}.

3.2.3.4 Exact solution methods

Many exact methods have been proposed for finding an optimal or a set of optimal solutions for a given problem. Among these methods, we can find branch and bound, branch and cut, and dynamic programming.

3.2.3.5 Branch and bound method

Branch and bound methods are based on the principle of enumerating the solution space of a given problem and then choosing the best solution (cf., Land *et al.* [40]; Martello *et al.* [46]). The enumeration has a tree structure. Each node of the tree separates the search space into two sub-spaces, until the complete exploration of the solution space (cf., Dasgupta *et al.* [17]).

However, there are three aspects in a branch and bound method. They are:

- Branching strategy.
- Bounding strategy.
- Node selection strategy.

The basic technique of the branch and bound method is that it divides the solution space into smaller subspaces to be investigated. The division is called branching as new branches are created in the enumeration tree. The bounding strategy is an important aspect, since it checks whether a subspace contains an optimal solution; meanwhile, it determines what prevents the search tree from growing too much. In other words, the bounding function for a subspace is computed and compared to the best solution found so far. It must be better, if not then, the subspace cannot contain the optimal solution, and this subspace is discarded. The node selection strategy controls how to choose the next node in the search tree for branching. There are several strategies, the most popular are: (a) depth first: choose only from among the set of nodes just created, and (b) best first: choose the node that has the best value of the bounding function from anywhere on the search tree.

The first branch and bound algorithm for the 0-1 KP was proposed by Kolesar (cf., Kolesar [39]). Several developments have been proposed later (for more details the reader can refer to Horowitz *et al.* [37]; Martello *et al.* [46]).

3.2.3.6 Branch and cut method

Branch and cut is a method of great interest for solving various combinatorial optimization problems. This method is a result of the integration between two methods: (i) cutting plane method, and (ii) branch-and-bound method (cf., Resende *et al.* [60]). The cutting planes lead to a great reduction in the size of the search tree of a pure branch and bound approach. Therefore, a pure branch and bound approach can be accelerated by the employment of a cutting plane scheme (for more details the reader can refer to Crowder *et al.* [14]; Balas *et al.* [6]; Balas *et al.* [5]).

3.2.3.7 Dynamic programming

The dynamic programming approach is a useful tool for solving some combinatorial optimization problems. The basic idea was first introduced by Bellman (cf., Bellman [9]). This approach consists of: (i) breaking a problem up into simpler sub-problems, (ii) solving these sub-problems, and (iii) combining the sub-solutions to reach the overall solution. For more details, Bellman's book (cf., Bellman [10]) gives a clear general introduction.

3.2.3.8 Best known exact algorithm for 0-1 KP

Martello *et al.* (cf., Martello *et al.* [44]) presented a powerful approach for solving the 0-1 KP using an exact method. To our knowledge, their algorithm is the best known algorithm and outperforms all others. This algorithm can be viewed as a combination of two exact methods: (i) dynamic programming method, and (ii) branch-and-bound method, with the use of a strong bound. The algorithm proved to be efficient in solving all classical test instances, with up to 10,000 variables, in a short solution time.

3.3 Disjunctively constrained knapsack problem

The Disjunctively constrained knapsack problem (abbreviated as DCKP) is a variant of the well-known classical knapsack problem with special disjunctive constraints. We can also find this problem, in the literature, as the classical knapsack problem with conflict graph (cf., Pferschy *et al.* [52]).

The DCKP was first introduced by Yamada *et al.* (cf., Yamada *et al.* [69]) as a problem in which an item may be incompatible with other items. There are a wide variety of applications that can be modeled as the DCKP. Herein, let's consider two examples. The first example is encountered in financial management: suppose that we have n projects and only c dollars are available. Each project j is characterized by a profit p_j and a cost c_j. Some projects are incompatible

with some others. The objective of the problem is to determine a set of compatible projects to be performed with the limited finance c. Such a problem can be modeled as the DCKP. The second example is encountered in transport with regards to the movement of disabled or those with disabilities, seniors, and other citizens with unique needs. Since the disabled people have their own issues, in most cases, they need to be served with respect to special conditions. For example, some of them cannot take the same bus with some others. In other words, a bus cannot serve those incompatible persons at the same time. Formally, this problem can be modeled as a DCKP.

3.3.1 Definition

An instance of DCKP is characterized by a set I containing n items, a set E of incompatible couples of items where $E \subseteq \{(i,j) \in I \times I, i < j\}$, and a knapsack of fixed capacity c. Each item i is characterized by a profit p_i and a weight w_i. Each $(i,j) \in E$ imposes that, item i and j are not allowed to be included together in the knapsack (i.e. the selection of one item in the solution excludes the other item). This relation is assumed to be reflective, i.e., $(i,j) \in E \Leftrightarrow (j,i) \in E$ (cf., Yamada *et al.* [69]). The objective of the problem is to select a subset of compatible items so that the sum of the selected items' profits is maximum without exceeding the knapsack capacity c. Consider the following example (cf., Figure 3.3):

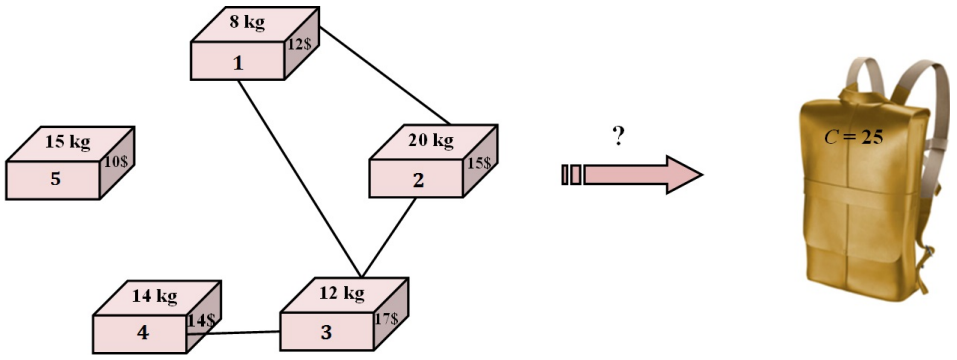


Figure 3.3: Disjunctively constrained knapsack problem

Let I be the set of 5 items such that $I = \{1,2,3,4,5\}$. The instance is characterized by a vector of profits $p = \{p_1, p_2, p_3, p_4, p_5\} = \{12\ \$, 15\ \$, 17\ \$, 14\ \$, 10\ \$\}$, a vector of weights $w = \{w_1, w_2, w_3, w_4, w_5\} = \{8\text{ kg}, 20\text{ kg}, 12\text{ kg}, 14\text{ kg}, 15\text{kg}\}$, and a capacity $c =$ of 25 kg. In addition we have four couples of incompatible items, where $E = \{(1,2), (2,3), (3,4), (1,3)\}$.

One can observe that if we consider item 1 in the solution, this excludes items 2 and 3, and if we consider item 4 this excludes items 3. A feasible solution may consist of placing items 1 and 5 with the objective value equal to 22 $. Another possible solution may consist of placing just item 3 with the objective value equal to 17 $. Thus, feasible solutions are all subsets of compatible items in which their total weight does not exceed the knapsack capacity c. In such a case, the aim is to choose the solution of maximum profit value among all feasible solutions (i.e. the optimal solution). Regarding the example, an optimal solution consists of selecting two items: item 1 and item 4 with the objective value equal to 26 $ and with a total weight equal to 22 kg.

3.3.2 Integer linear programming of DCKP

Let x_i be the decision variable associated to the item i such that, x_i takes the value 1 if the item i is selected, otherwise x_i takes the value 0. In this case, the disjunctive constraints can be stated as:

$$x_i + x_j \leq 1, \quad \forall (i,j) \in E$$

The knapsack constraint is represented as:

$$\sum_{i=1}^{n} w_i x_i$$

The objective consists of selecting a sub-set of items such that both the knapsack capacity constraint and the disjunctive constraints are satisfied, and the total sum of profits of items placed in the knapsack is the maximal. Then, the integer linear programming of the $DCKP$ can be formulated as follows:

$$Max: \quad f(x) = \sum_{i=1}^{n} p_i x_i \tag{3.4}$$

$$s.t. \quad \sum_{i=1}^{n} w_i x_i \leq c \tag{3.5}$$

$$x_i + x_j \leq 1, \qquad \forall (i,j) \in E \tag{3.6}$$

$$x_i \in \{0,1\} \qquad \forall i \in I = \{1,...,n\} \tag{3.7}$$

In this integer linear programming, we have four equations. The first equation (Equation 3.4) is the objective function where the goal is to maximize the value of total profit of items placed in the knapsack. The second equation (Equation 3.5)) is the knapsack capacity constraint which imposes that the weights'

sum of the selected items does not exceed the knapsack capacity. The third equation (Equation 3.6) represents the set of the disjunctive constraints which ensure the compatibility of all items. Meanwhile, the forth and last equation (Equation 3.7) is imposed on items that are to be included or not in the knapsack (i.e., it is not allowed to include a fractional item). In order to avoid trivial cases, it is assumed that:

- All input data $c, p_i, w_i, \forall i \in n$ are strictly positive integers.
- $\sum_{i \in n} w_i > c$ for avoiding trivial solutions.

The DCKP is a combinatorial optimization problem (cf., Arora *et al.* [4]). It reduces to the 0-1 KP when $E = \emptyset$ and the independent set problem when the knapsack capacity constraint is omitted (cf., for more details the reader can be referred to Garey *et al.* [26]).

3.3.3 DCKP and solution methods

Due to the complexity of the DCKP, most results on this topic are based on heuristics (cf., Senisuka *et al.* [64]). Even the Cplex solver which is one of the best known solvers, has difficulty in solving the problem within 2 or 3 hours. Therefore, we are motivated in developing efficient approaches for solving the considered problem that yields interesting solutions within a short average running time. However, in what follows, we cite some solution methods dedicated to solve the disjunctively constrained knapsack problem.

3.3.3.1 Heuristic solution methods

Yamada *et al.* (cf., Yamada *et al.* [69]) proposed two heuristic methods. The first heuristic method is a greedy method that yields a feasible solution for the DCKP. The second heuristic method is a local search method that improves the quality of the solution obtained from the greedy method.

3.3.3.2 A greedy algorithm

The greedy algorithm proposed by Yamada *et al.* (cf., Yamada *et al.* [69]) is simple, it builds an initial solution in n steps based on the principle of greed (i.e. make a local optimum choice without considering the consequences). In each step i (where $i \leq n$) a decision variable x_i is fixed to the values 1 (if it is considered in the solution) or 0 (if not). Note that the variables $x_1, x_2, ...x_{i-1}$ have been already fixed before. Indeed, if the weight of the item i does not exceed the remaining capacity of the knapsack and if it is compatible with all the previously fixed items, then, the algorithm considers this item in the solution (i.e., x_i is

fixed to the value 1). Othersiwe, it doesn't (i.e. x_i is fixed to the value 0). The main steps of this algorithm are illustrated in algorithm 7.

Input: An instance of P_{DCKP}.
Output: A feasible solution $\overline{x}$.

1: Set $\overline{c}$ as a remainging knapsack capacity, where $\overline{c} \Longleftarrow c$;
2: **for** $i = 1$ to n **do**
3: **if** $w_i \leq \overline{c} and (i : i < j, x_i = 1), (i, j) \in E$ **then**
4: $\overline{x}_j = 1$;
5: $\overline{c} = \overline{c} - w_i$;
6: **else**
7: $\overline{x}_j = 0$;
8: **end if**
9: **end for**
10: **return** $\overline{x}$;

Algorithm 7: A greedy algorithm for DCKP

3.3.3.3 A local search algorithm

The local search method proposed by Yamada *et al.* (cf., Yamada *et al.* [69]) applies a 2-*opt* procedure in order to improve the initial solution obtained from the greedy algorithm. The main idea consists of replacing an item by other items. This exchange is permitted if the obtained solution is better than the current one. Let $\overline{x}$ be a feasible solution of $DCKP$, and $I(\overline{x})$ is the set of the decision variables fixed to 1. The authors define $N(\overline{x})$ as the neighborhood of the solution $\overline{x}$ represented as a set of solutions obtained by exchanging some elements of $I(\overline{x})$ with other elements of the instance. Algorithm 8 illustrates the main steps of this algorithm.

This improvement in the neighborhood $N(\overline{x})$ is performed as follows: starts with the solution obtained from the greedy algorithm. The 2-*opt* improves this solution in its neighborhood $N(\overline{x})$ by applying, iteratively, two steps:

- Make $x_i = 0$ for each $x_i \in I$, replaced this object with others.
- Chose the best solution found so far.

Then, the solution is updated. This process is repeated with the new solution. This process is stopped with the best local optimum solution found.

Input: An initial solution $\overline{x}$.
Output: An improved solution ($x^{'}$).

1: Let NS be a neighborhood solution.
2: Set $x^{'} \Longleftarrow \overline{x}$;
3: **while** $\exists\ NS\ \in N(\overline{x})$ **do**
4: **if** $NS > x^{'}$ **then**
5: $x^{'} = NS$;
6: **end if**
7: **end while**
8: **return** $x^{'}$;

Algorithm 8: 2-*opt* algorithm

3.3.3.4 Meta-heuristic solution methods

Meta-heuristics are powerful and advanced heuristics. They guide the search process in the feasible solution space to explore different regions beyond local optimality, with the aim of escape from a series of local optimum and produce high-quality solutions. For the last decades, they have been widely developed and applied to a variety of combinatorial optimization problems (cf., Blum *et al.* [11]; Osman *et al.* [49]; Vaessens *et al.* [67]).

Several meta-heuristic approaches have been proposed. Hifi and Michrafy (cf., Hifi *et al.* [30]) proposed a three-step reactive local search. The first step of the algorithm determines an initial solution by using a greedy procedure that iteratively introduces an item into the knapsack until filling it, and sets it to the current solution. The second step is based on an intensification procedure in the neighborhood of the current solution where a neighboring solution is obtained by removing an item from the solution and inserting other ones. The third step diversifies the search process by accepting to temporarily degrade the quality of the solution in hope of escaping from local optima. The reactive local search repeats the last two steps until satisfactory solution is reached.

Akeb *et al.* (cf., Akeb *et al.* [3]) investigated the use of an effective local branching technique. The local branching is introduced by Fischetti and Lodi (cf., Fischetti *et al.* [23]), the goal is to provide quick suboptimal solution in order to accelerate the global search process of a bounding-based search scheme. The principle of the local branching algorithm may be described as follows: starting from an initial solution, it explores the neighborhood of the current best solution until a new best one is found, then re-iterates the process using the improved solution as the initial solution; otherwise, no new better solution is found; therefore, seeking the optimum inside the unexplored feasible region. These processes are iterated until they satisfy a stopping condition. Finally, the algorithm returns the best solution found.

Hifi *et al.* (cf., Hifi *et al.* [34]) proposed an adaptive scatter search for approximately solving the DCKP. The scatter search is a meta-heuristic, which can

be viewed as an evolutionary method (cf., Glover [27]). It is based upon the combination between some solutions in order to generate new trial solutions and so on. In general, the main mechanism for combining the solutions is such that a new solution is created from the linear combination of two other solutions and in some cases, such a new solution may be reached by combining more than two solutions.

In [33], Hifi *et al.* presented an adaptation to the scatter search approach proposed by Hifi *et al.* in [34] for approximating the DCKP. The adaptation is, mainly, based on applying such an approach to an equivalent problem of the DCKP proposed by Hifi *et al.* in [31]; which is re-enforced by using two types of valid constraints. The first is a bound on its minimal objective function value, and the second is two valid cardinality constraints. The bound requires the objective function value to be at least as large as a lower bound, which is the value of the best feasible solution found so far. The two cardinality constraints direct that the number of items included in the knapsack are bound by two values in order to accelerate the search process.

Finally, Hifi in [29] proposed an iterative rounding search algorithm, which can be viewed as an alternative to the approaches used in [33] and [3] for approximately solving the DCKP. Indeed, in [3] Akeb *et al.* showed that fixing a part of the decision variables and solving the reduced problem with local-branching, improved the quality of the solutions but at the expense of the running time. On the other hand, Hifi and Otmani in [33] noted that considering an equivalent model for the DCKP with adding the cardinality constraints bounded by both lower and upper values, accelerated the search process and at the same time improves the quality of some solutions. Accordingly, the authors presented an iterative rounding search algorithm. The algorithm combines two key features: (i) a rounding strategy applied to the fractional variables of a linear relaxation, and (ii) a local neighborhood search for improving the quality of the solutions at hand. Both strategies are iterated into a process based on adding a series of (i) valid cardinality constraints and (ii) lower bounds used for bounding the objective function.

3.3.3.5 Exacts solution methods

Among papers addressing the exact solution of DCKP, we found that of Yamada *et al.* (cf., Yamada *et al.* [69]) in which the problem was tackled with an exact method. The algorithm is based upon an implicit enumeration search combined with an interval reduction method. It uses a solution obtained from a heuristic method as an initial lower bound.

Hifi and Michrafy (cf., Hifi *et al.* [31]) proposed three exact algorithms in which reduction strategies, an equivalent model and a dichotomous search cooperate to solve the DCKP. The first algorithm reduces the size of the original problem by starting with a lower bound and successively solving relaxed DCKPs. The second algorithm combines a reduction strategy with a dichotomous search

in order to accelerate the search process. The third algorithm tackles instances with a large number of disjunctive constraints.

3.4 Some other combinatorial problems

There are many other combinatorial optimization problems belonging to the knapsack family. In this section, we cite some of them.

3.4.1 Multi-scenario max-min knapsack problem

Multi-scenario max-min knapsack problem (abbreviated as MSKP), is a variant of the classical knapsack problem (cf., Yu [71]; Pinto *et al.* [53]). This problem is characterized by a set of items i with a profit (p_i) and weights (w_i), and by a knapsack of a limited capacity c. Herein, a scenario s is defined as the set of profits that apply to each item i. Such that, a profit p_i of an item i depends on the scenario s that is considered. The multi-scenario max-min knapsack problem can be expressed as follows:

$$max \quad \min_{s=1,...,S} \left\{ \sum_{i=1}^{n} p_i^s x_i \right\} \tag{3.8}$$

$$s.t. \quad \sum_{i=1}^{n} w_i x_i \leq c \tag{3.9}$$

$$x_i \in \{0,1\} \qquad \forall i = \{1,...,n\} \tag{3.10}$$

The goal consists of finding the set of items placed in the knapsack whose total profit is maximized under the worst possible scenario.

3.4.2 Precedence constrained knapsack problem

A variant of the classical 0-1 KP considered that, the items are partially ordered through a set of precedence relations (noted as precedence constraints). This variation called as the precedence constrained knapsack problem (cf., Samphaiboon *et al.* [63]; You *et al.* [70]). However, the problem can be represented mathematically as following:

$$Max: \quad f(x) = \sum_{i=1}^{n} p_i x_i \tag{3.11}$$

$$s.t. \quad \sum_{i=1}^{n} w_i x_i \leq c \tag{3.12}$$

$$x_i \geqslant x_j, \qquad \forall (i,j) \in E \tag{3.13}$$

$$x_i \in \{0,1\} \qquad \forall i \in I = \{1,...,n\} \tag{3.14}$$

Where, E represents the precedence relations between the items. Such that $(i, j) \in E$ posses that item j can be considered only when item i has already been placed in the knapsack.

3.4.3 Bounded knapsack problem

The bounded knapsack problem is a variant of the 0-1 KP. In this variant, there is a bounded amount m_i of each item of type i. The bounded knapsack problem can be expressed as:

$$Max: \quad f(x) = \sum_{i=1}^{n} p_i x_i \tag{3.15}$$

$$s.t. \quad \sum_{i=1}^{n} w_i x_i \leq c \tag{3.16}$$

$$x_i \in \{0, 1, ..., m_i\} \qquad \forall i = \{1, ..., n\} \tag{3.17}$$

Where, x_i is the amount of each type of item to be placed in the knapsack in order to obtain the best objective value.

There are several other problems belonging to the knapsack family including: the multiple choice knapsack problem and the multiple choice multiple knapsack problem, ...etc. Each variation is different from others by the characterization of items and/or the distribution of the items in one or more knapsack (for more details the reader can be refereed to Martello *et al.* [46]).

3.5 Conclusions

This chapter presented some variants belonging to the knapsack family, which are known as being problems of combinatorial optimization. In order to tackle such problems, in this chapter, we presented two directions of research: exact methods and heuristic methods. An exact algorithm tries to find an optimal or a set of optimal solutions for a given problem while a heuristic algorithm is alternative procedure that permits you to solve large-scale instances by providing good solutions, not necessarily the optimal one but in an acceptable solution time.

In the following chapter, we present our first sequential solution method proposed for solving the disjunctively constrained knapsack problem.

Part II

Sequential solution methods

Chapter 4

A fast large neighborhood search for the disjunctively constrained knapsack problem

Contents

In this chapter, we propose a heuristic based upon large neighborhood search for the disjunctively constrained knapsack problem (DCKP). The proposed method combines a two-phase procedure and a large neighborhood search. First, the two-phase procedure is applied in order to provide a starting feasible solution for the large neighborhood search. The first phase serves to determine a feasible solution by successively solving two subproblems: the independent set and the classical binary knapsack. The second phase tries to improve the quality of the solutions by using a descent method which applies both degrading and re-optimizing strategies. Second, a large neighborhood search is introduced in order to diversify the search space. Finally, the performance of the proposed method is computationally analyzed on a set of benchmark instances of the literature where its provided results are compared to those reached by Cplex solver and some recent algorithms. The provided results show that the method is very competitive since it is able to reach new solutions within small runtimes.

4.1 Introduction

In this chapter we investigate the use of the large neighborhood search for solving the *Disjunctively Constrained Knapsack Problem* (DCKP). We recall that, the DCKP is characterized by a knapsack of fixed capacity c, a set I of n items, and a set E of incompatible couples of items, where $E \subseteq \Big\{(i,j) \in I \times I, \quad i < j\Big\}$. Each item $i \in I$ is represented by a nonnegative weight w_i and a profit p_i. The goal of the DCKP is to maximize the total profit of items that can be placed into the knapsack without exceeding its capacity, where all items included in the knapsack must be compatible. The mathimatical formulation of DCKP can be defined as follows:

$$
\begin{aligned}
(\mathrm{P}_{DCKP}) \quad \max \quad & \sum_{i \in I} p_i x_i && && (4.1)\\
\text{s.t.} \quad & \sum_{i \in I} w_i x_i \leq c && && (4.2)\\
& x_i + x_j \leq 1 && \forall\ (i,j) \in E && (4.3)\\
& x_i \in \{0,1\} && \forall\ i \in I, && (4.4)
\end{aligned}
$$

where $x_i, \forall\ i \in I$, is equal to 1 if the i-th item is included in the knapsack (solution); 0 otherwise. Inequality (4.1) represents the objective function. Inequality (4.2) denotes the knapsack constraint with capacity c and inequalities (4.3) represent the disjunctive constraints which ensure that all items belonging to a feasible solution must be compatible. We can observe that the solution domain of a knapsack problem can be characterized by (i) the inequality (4.2), (ii) the integral constraints $x_i \in \{0,1\}$, $\forall i \in I,$, and (iii) those corresponding to the independent set problem obtained by combining both inequalities (4.2) and (4.3) and by setting the profit of all items to one in the objective function (4.1). Without loss of generality, we assume that (i) all input data c, p_i, $w_i, \forall\ i \in I$, are nonnegative integers and (ii) $\sum_{i \in I} w_i > c$ for avoiding trivial solutions.

The DCKP is an NP-hard combinatorial optimization problem. It reduces to the *independent set* problem (cf., Garey and Johnson [26]) when the knapsack capacity constraint is omitted and to the *classic knapsack* problem when $E = \emptyset$. It is easy to show that DCKP is a more complex extension of the multiple choice knapsack problem which arises either as a stand alone problem or as a component of more difficult combinatorial optimization problems. Its induced structure in complex problems allows the computation of upper bounds and the design of heuristic and exact methods for these complex instances. For example, DCKP was used in Dantzig-Wolfe's decomposition formulation for the two-dimensional bin packing problem (cf., Pisinger and Sigurd [56]). It served as a local optimization subproblem for the pricing problem which consists in finding a feasible packing of a single bin verifying the smallest reduced cost. The same problem has been also used in Sadykov and Vanderbeck [62] as the pricing problem for solving the bin packing with conflicts.

This chapter is organized as follows. Section 4.2 discusses the neighborhood search method. Section 4.3 introduces a fast large neighborhood search-based heuristic for solving the DCKP. Section 4.4 evaluates the performance of the proposed method on the instances taken from the literature, and analyzes the obtained results. Finally, Section 4.5 summarizes the contents of the chapter.

4.2 Neighborhood search method

Neighborhood search is a wide class of improvement heuristics that have shown interesting results in solving various combinatorial optimization problems (cf., Aarts *et al.* [1]).

Suppose that, we have an instance I containing n items, such that $I = \{1, 2, ..., n\}$. Let F represent the subsets of feasible solutions of the set I, i.e., $F \subseteq 2^n$, where 2^n denotes the set of all the subsets of I. Let the objective function represented as f, where $f : F \rightarrow \Re$. Suppose that, the combinatorial optimization problem is a maximization problem, that, we want to find a solution x^* such that:

$$f(x^*) \geqslant f(x) \quad \forall x \in F$$

Each solution $x \in F$ has an associated subset $N(x) \subseteq F$. The set $N(x)$ is called the neighborhood of the solution x. That is, N is a function that maps a solution to a set of solutions. A solution x^* is said to be locally optimal with respect to a neighborhood function N if and only if $f(x^*) \geq f(x)$ for all $x \in N(x^*)$. The neighborhood $N(x)$ is usually said to be exponential if $|N(x)|$ grows exponentially as n increase. With these definitions it is possible to define, generally, a neighborhood search algorithm.

In general, a basic neighborhood search algorithm consists of three steps: first, the algorithm starts with an initial solution x as an input. Second, it computes $x^* = argmax_{x' \in N(x)}\{f(x')\}$, (i.e., finds the best solution x^* in the neighborhood of x). Third and last, performs the update $x = x^*$, if $f(x^*) > f(x)$. These steps are repeated until reaching to a local optimum and/or performing the stopping criteria. Then, the algorithm stops with the best solution found so far (cf., Ahuja *et al.* [2]). It can be observed that, a neighborhood search algorithm gradually improves a starting solution by exploring the neighborhoods of a current solution.

Among neighborhood search methods, we can find: (i) descent method, and (ii) large neighborhood search method. A descent algorithm stops with a local optimum solution. On the other hand, a large neighborhood search algorithm serves to explore several regions of the solution space aiming to escape from a series of local optimum solutions.

4.3 Large neighborhood search-based heuristic

Large Neighborhood Search (abbreviated as LNS) is a heuristic that has proven to be effective on wide range of combinatorial optimization problems. A simplest version of LNS has been presented in Shaw [65] for solving the vehicle routing problem (cf., also cf., Pisinger *et al.* [55]). LNS is based on the concepts of *building* and *exploring* a neighborhood; that is, a neighborhood defined implicitly by a *destroy* and a *repair* procedure. Unlike the descent methods, which may stagnates in local optimum, using large neighborhoods makes it possible to reach better solutions and explore a more promising search space.

This section presents an adaptive large neighborhood search for approximating the DCKP[1]. The proposed algorithm combines a two-phase solution procedure and a diversification procedure. Figure 4.1 illustrates the high block diagram of the proposed algorithm.

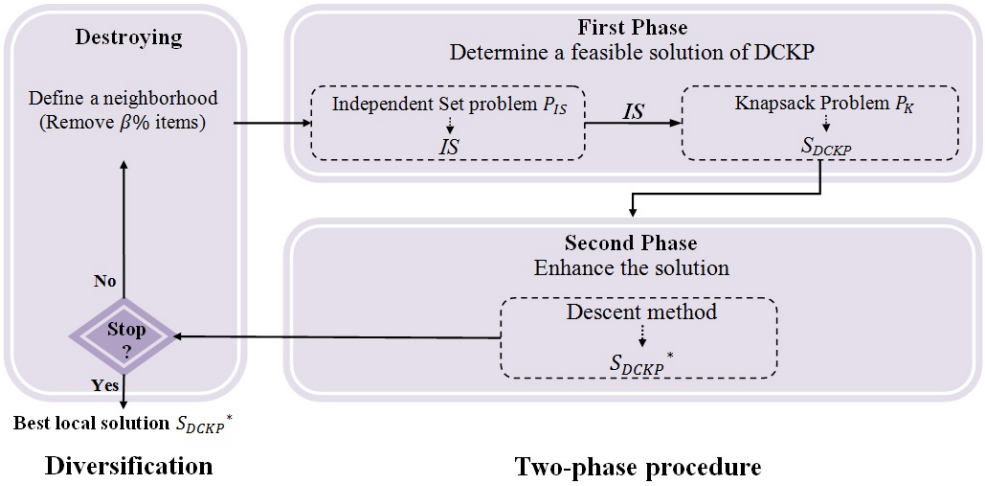


Figure 4.1: High block diagram of the proposed algorithm

4.3.1 Two-phase solution procedure

Herein, we describe an efficient procedure for approximating the DCKP by alternately running two phase solution procedures (cf., Figure 4.1). The first phase is applied in order to determine a feasible solution and then, the second one tries to improve the solution at hand. For the rest of the chapter, we assume that all items are ranked in decreasing order of their profits.

[1]This work is published in an international conference with proceedings *3rd International Symposium on Combinatorial Optimization* (cf., Hifi *et al.* [36])

4.3.1.1 The first phase

The first phase determines a feasible solution of the DCKP by solving two optimization problems (cf., Figure 4.2). The DCKP firstly is decomposed into two problems: an independent problem set and a classical knapsack problem. Both problems are solved in the following manner:

- A *independent set problem* (noted P_{IS}), extracted from P_{DCKP}, is first solved in order to determine an *Independent Set* solution, noted IS.
- A *classical binary knapsack problem* (noted P_K) associated to both IS and the corresponding capacity constraint (i.e., $\sum_{i \in IS} w_i x_i \leq c$) is solved in order to provide a feasible solution for the P_{DCKP}.

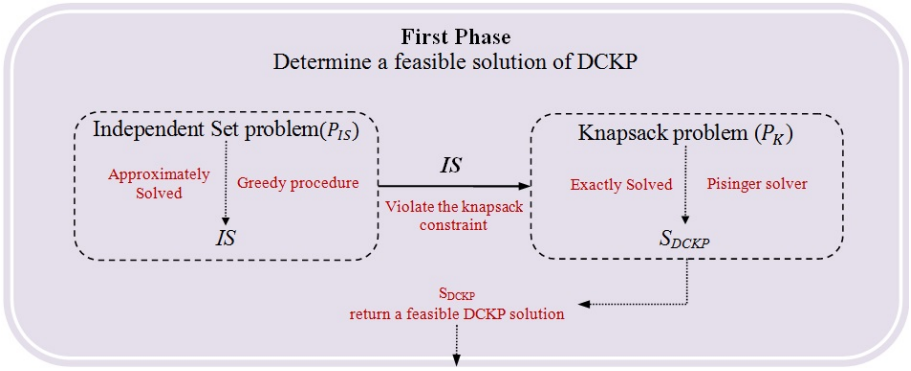


Figure 4.2: First phase: two optimization problems

Let $S_{IS} = (s_1, \ldots, s_n)$ be a feasible solution of P_{IS}, where s_i is the binary value assigned to x_i, $\forall\ i \in I$. Let $IS \subseteq I$ be the restricted set of items denoting items of S_{IS} whose values are fixed to 1. Then, linear programs referring to both P_{IS} and P_K may be defined as follow:

$$(\mathrm{P}_{IS}) \begin{cases} \max & \displaystyle\sum_{i \in I} x_i \\ \text{s.t.} & x_i + x_j \leq 1,\ \forall (i,j) \in E \\ & x_i \in \{0,1\},\ \ \forall\ i \in I, \end{cases}$$

$$(\mathrm{P}_K) \begin{cases} \max & \displaystyle\sum_{i \in IS} p_i x_i \\ \text{s.t.} & \displaystyle\sum_{i \in IS} w_i x_i \leq c, \\ & x_i \in \{0,1\},\ \forall\ i \in IS. \end{cases}$$

On the one hand, we can observe that the solution domain of P_{IS} includes the solution domain of P_{DCKP}. On the other hand, an optimal solution of P_{IS} is not necessary an optimal solution of P_{DCKP}. Therefore, in order to search a quick solution IS, the following procedure is applied (as described in Algorithm 9):

Input: An instance I of P_{DCKP}.
Output: A feasible solution (independent set) IS for P_{IS}.

1: **Initialization:**
   Set $IS = \emptyset$ and $I = \{1, \ldots, n\}$.
2: **while** $I \neq \emptyset$ **do**
3: Let $i = \text{argmax}\Big\{p_i \mid p_i \geq p_k,\ k \in I\Big\}$.
4: Set $IS = IS \cup \{i\}$.
5: Remove i and all items j such that $(i, j) \in E$ from I.
6: **end while**
7: **return** IS as a feasible solution of P_{IS}.

Algorithm 9: Compute an independent set (IS) solution for P_{IS}

Set IS to an empty set (a trivial solution of P_{IS}) and, fill iteratively the set IS with the items belonging to I. At each iteration (of Algorithm 9), the item realizing the greatest profit is selected and inserted into IS. The chosen item with its incompatible neighbors are then removed from I. Such a process is iterated and the algorithm stops and exits with a feasible solution IS for P_{IS} whenever no item can be added to the current solution IS.

Input: IS, an independent set of P_{IS}, and I, an instance of P_{DCKP}.
Output: S_{DCKP}, a DCKP's feasible solution.

1: **Initialization:**
   Let P_K be the resulting knapsack problem constructed according to items belonging to IS.
2: **if** IS satisfies the capacity constraint (4.2) of P_{DCKP} **then**
3: Set $S_{DCKP} = IS$;
4: **else**
5: Let S_{DCKP} be the resulting solution of P_K.
6: **end if**
7: **return** S_{DCKP} as a feasible solution of P_{DCKP}.

Algorithm 10: Compute a feasible for P_{DCKP}

As mentioned above, IS may violate the capacity constraint of P_{DCKP}. Then, in order to provide a feasible solution for P_{DCKP}, the knapsack problem P_K is solved. Herein, P_K is solved using the exact solver of Martello *et al.* [44]. Algorithm 10 describes the main steps used for determining a feasible solution of P_{DCKP}.

4.3.1.2 The second phase

In order to improve the quality of the solution provided by the first phase (i.e., the feasible solution S_{DCKP} returned by Algorithm 10), a local search is per-

formed. The used local search can be considered as a descent method that tends to improve a solution by alternatively calling two procedures: (i) a degrading procedure, and (2) re-optimizing procedure. The *degrading procedure* serves to build a k-neighborhood of S_{DCKP} by dropping k fixed items from S_{DCKP} while the *re-optimizing procedure* tries to determine an optimal solution regarding the current neighborhood. The descent procedure is stopped when no better solution can be reached.

Input: S_{DCKP}, a feasible solution of DCKP.
Output: $S^{\star}_{DCKP}$, a local optimal solution of DCKP.

1: Set $S^{\star}_{DCKP}$ as an initial feasible solution, where all variables are fixed to 0.
2: **while** S_{DCKP} is better than $S^{\star}_{DCKP}$ **do**
3: Update $S^{\star}_{DCKP}$ with S_{DCKP}.
4: Set $\alpha|I|$ fixed variables of S_{DCKP} as free.
5: Define the corresponding neighborhood of S_{DCKP}.
6: Determine the optimal solution in the current neighborhood and update S_{DCKP}.
7: **end while**
8: **return** $S^{\star}_{DCKP}$.

Algorithm 11: A descent method

Algorithm 11 shows how an improved solution can be computed by using a descent method. Let S_{DCKP} be the current feasible solution obtained at the first phase and α be a constant, where $\alpha \in [0, 1]$. Then, $\alpha|I|$ represents the number of the unassigned variables of S_{DCKP}. The main loop (cf., lines 2 - 7) of the descent method serves to provide a neighborhood of a local optimum and to explore it. At line 3, the best solution found so far, namely $S^{\star}_{DCKP}$, is updated with the solution S_{DCKP} reached in the last iteration. Line 4 determines the $\alpha|I|$ unassigned variables regarding the current solution S_{DCKP}, where items with highest degree (i.e., items with largest neighborhood), are favored. Let i be an item realizing the highest degree; that is, an item i whose variable x_i is fixed to 1 in S_{DCKP}. Then, set x_i to a free (unassigned) variable with its incompatible variables x_j, such that $(i, j) \in E$ and $(j, k) \notin E$, where k, $k \neq i$, corresponds to the index of the variables whose values are equal to 1 in S_{DCKP}. At line 6, S_{DCKP}, is replaced by the best solution found in the current neighborhood. Finally, the process is iterated until no better solution can be reached (in this case, Algorithm 11 exits with the best solution $S^{\star}_{DCKP}$).

Note that, on the one hand, the runtime of Algorithm 11 may increase when α tends to 100. Because dropping a large percentage of items involves that the reduced DCKP is closest to the original one. On the other hand, algorithm 11 is called at each iteration of the large neighborhood search (cf., Section 4.3.3),

a large size reduced DCKP can cause the large neighborhood search slow down. Therefore, we favor the achievement of a fast algorithm which is able to converge towards a good local optimum. This is why our choice is oriented to moderate the values of α, as shown in the experimental part (cf., Section 4.4).

4.3.2 Diversification procedure

For instance, the descent method discussed in the previous section (cf., Algorithm 11) may explore some regions and stagnates in a local optimum because either *degrading* or *re-optimizing* considers a mono-criterion. Thus, in order to (i) enlarge the chance of reaching a series of improved solutions, (ii) explore a more promising search space, and (iii) escape from a series of local optima, a random destroying strategy is performed. Algorithm 12 illustrates the main steps of the diversification procedure which remove $\beta\%$ of items, for determining a neighbourhood of the current solution.

Input: S_{DCKP}, a starting solution of P_{DCKP}.
Output: An independent set IS and a reduced instance I^r of P_{DCKP}.

1: Set $counter = 0$, $I^r = \emptyset$ and IS to the set of items whose decision variable are fixed to 1 in S_{DCKP}.
2: Range IS in non decreasing order of their profit per weight.
3: **while** $counter < \beta|I|$ **do**
4: Let r be a real number randomly generated in the interval $[0, 1]$ and $i = |IS| \times r^{\gamma}$.
5: Set $IS = IS \setminus \{i\}$, $I^r = I^r \cup i$ and increment $counter = counter + 1$.
6: **for** all items j such that $(i, j) \in E$ **do**
7: **if** item j is compatible with all items belong to IS **then**
8: Set $I^r = I^r \cup \{j\}$ and $counter = counter + 1$.
9: **end if**
10: **end for**
11: **end while**
12: **return** IS and I^r.

Algorithm 12: Remove $\beta|I|$ variables of S_{DCKP}

4.3.3 An overview of the LNSBH algorithm

Algorithm 13 summarizes the main steps of the proposed algorithm, namely Large Neighborhood Search-Based heuristic (abbreviated as LNSBH). The main loop consists of three steps. First, algorithm 12 is called in order to apply a random destroying strategy. Second, algorithms 9, 10 and 11 are used in order to re-optimize the reduced solution. Third and last, update the best local optimum

solution found so far.

Input: S_{DCKP}, a starting solution of P_{DCKP}.
Output: $S^{\star}_{DCKP}$, a local optimum of P_{DCKP}.

1: Set $S^{\star}_{DCKP}$ as a starting feasible solution, where all variables are assigned to 0.
2: **while** the time limit is not performed **do**
3: Call Algorithm 12 in order to find IS and I^r according to S_{DCKP}.
4: Call Algorithm 9 with an argument I^r to complete IS.
5: Call Algorithm 10 with an argument IS for reaching a new solution S_{DCKP}.
6: Improve S_{DCKP} by applying Algorithm 11.
7: Update $S^{\star}_{DCKP}$ with the best solution.
8: **end while**
9: **return** $S^{\star}_{DCKP}$.

Algorithm 13: A large neighborhood search-based heuristic

4.4 Computational results

This section evaluates the effectiveness of the proposed large neighborhood search-based heuristic on two groups of instances (taken from the literature [30] and generated following the schema used by Yamada *et al.* [69]). The first group contains twenty medium-size instances with 500 items, a capacity $c = 1800$ and different densities (assessed in terms of the number of disjunctive constraints). The second group contains thirty large-size instances, where each instance contains 1000 items, with c taken in the discrete interval $\{1800, 2000\}$ and with various densities. The proposed LNSBH was coded using C++ and run on a PC Intel Pentium Core i5-2500 with 3.3 Ghz.

The performance of LNSBH depends on certain parameters, like the percentage α of the unassigned items considered in the descent method, the percentage β of items to be removed when applying Algorithm 12 (according to the large neighborhood search diversification procedure), the constant γ used by Algorithm 12 and Algorithm 13's runtime limit t. In what follows, we show how the aforementioned parameters can be experimentally fixed in order to maintain the competitively of LNSBH.

4.4.1 Effect of both degrading and re-optimizing procedures

This section evaluates the effect of the descent method based upon degrading and re-optimizing procedures (as used in Algorithm 11) on the starting solution realized by Algorithm 10. We recall that the re-optimization procedure tries to

solve a reduced P_{DCKP} which contains a small number of items. These problems are optimally solved using the Cplex solver (version 12.4).

Table 4.1 shows the variation of Av_Sol, the average value of solutions provided by the considered algorithm over all treated instances and Av_time, the average runtime needed by each algorithm for reaching the results. One can observe that, for the descent method (cf., Algorithm 11), the results are obtained by performing Algorithm 11 with different settings of α, where the value of α is varied in the discrete interval $\{5, 10; 15; 20\}$.

		The descent method			
		$\alpha =$			
	Algo. 9-10	5%	10%	15%	20%
Av. Sol.	2014.62	2129.98	2188.80	2232.36	2217.04
Av. time	≈ 0.001	0.15	2.03	19.72	118.30

Table 4.1: Effect of the descent method on the starting DCKP's solution.

From Table 4.1, we can observe that the best average solution value is realized for the value $\alpha = 15\%$, but it needs an average runtime of 19.72 seconds. Note that LNSBH's runtime depends on the descent method's runtime. Therefore, according to the results displayed in Table 4.1, one can observe that the value of 5% favors a quick resolution (0.15 seconds) with an interesting average solution value of 2129.98. Because our aim is to propose a fast efficient LNSBH, we then set $\alpha = 5\%$ for the rest of the chapter.

4.4.2 Behavior of LNSBH on both groups of instances

Remark that, according to the results shown in Shaw [65], when γ varies over the range of the integer interval $[5, 20]$, the LNS works reasonably well. In our test, we set $\gamma = 20$ for Algorithm 12. Therefore, in order to evaluate the performance of LNSBH, we focus on both parameters β and t; that are used in Algorithm 13. The study is conducted by varying the value of β in the discrete interval $\{10, 15, 20, 25, 30\}$ and t in the interval $\{25, 50, 100, 150, 200\}$ (measured in seconds). Table 4.2 displays the average solution values realized by LNSBH using the different values of (β, t).

From Table 4.2, we observe what follows:

- Setting $\beta = 10\%$ provides the best average solution value. Further, the solution quality increases when the runtime limit is extended.
- All other variations of β induce smaller average values than those of $\beta = 10\%$ in 200 seconds.

According to the results displayed in Table 4.2, the objective value of the solutions determined by setting $(\beta, t) = (10\%, 100)$ and $(10\%, 200)$ are displayed

t \ β	Variation of β				
	10%	15%	20%	25%	30%
25	2395.38	2394.48	2392.5	2391.14	2389.06
50	2397.52	2397.44	2394.74	2393.34	2391.18
100	2399.16	2398.98	2396.36	2394.2	2393.9
150	2399.28	2399	2397.82	2395.62	2394.58
200	2400.22	2399.62	2398.76	2396.8	2395.74

Table 4.2: The quality of the average values when varying the values of the couple (β, t).

in Table 4.3. In our computational study, ten random trials of the LNSBH are performed on the fifty literature instances and each trial is stopped respectively after 100 and 200 seconds.

Table 4.3 shows objective values reached by LNSBH and Cplex when compared to the best solutions of the literature (taken from Hifi *et al.* [29, 34]). Column 1 of Table 4.3 displays the instance label, column 2 reports the values of the best solutions (denoted V_{Cplex}) reached by Cplex v12.4 after one hour of runtime and column 3 displays the solutions provided by the most recent algorithm of the literature, denoted V_{IRS}, after 1000 seconds of runtime. Finally, column 4 (resp. 5) reports Max.Sol. (resp. Av.Sol) denoting the maximum (average) solution value obtained by LNSBH over ten trials for the first runtime limit of 100 seconds for each trail and columns 6 and 7 display those of LNSBH for the second runtime limit of 200 seconds for each trail.

From Table 4.3, we observe what follows:

1. First, we can observe the inferiority of the Cplex solver because it realizes an average value of 2317.88 (within 3600s) compared to that realized by V_{IRS} (2390.40 within 1000s). In this case, Cplex matches 5 instances over 50, representing a percentage of 10% of the best solutions of the literature.

2. Second, for both runtime limits (100 and 200 seconds for each trail), the average objective values (Av.Sol) reached by LNSBH over ten trials are generally better than those reached by V_{IRS} within 1000 seconds.

3. Third and last, according to the best solutions realized by LNSBH on ten trials with the first runtime limit (100 seconds for each trail), one can observe that LNSBH is able to improve the best solutions of the literature on 30 cases, matches 14 instances and fails in 6 occasions. On the other hand, by extending the runtime limit to 200 seconds, the solutions reached by LNSBH become more interesting. Indeed, in this case, LNSBH realizes 33 new best solutions, matches 13 solutions and fails in 4 occasions.

			LNSBH			
	V_{Cplex}	V_{IRS}	$\beta = 10$ and $t = 100$		$\beta = 10$ and $t = 200$	
Instance	$3600s$	$1000s$	Max.Sol.	Av.Sol.	Max.Sol.	Av.Sol.
1I1	2567	2567	2567	2564.2	2567	2564.6
1I2	2594	2594	2594	2594	2594	2594
1I3	2320	2320	2320	2319	2320	2319
1I4	2298	2303	2310	2310	2310	2310
1I5	2310	2310	2330	2328	2330	2329
2I1	2080	2100	2117	2116.1	2118	2117
2I2	2070	2110	2110	2110	2110	2110
2I3	2098	2128	2119	2110.2	2132	2118.1
2I4	2070	2107	2109	2106.9	2109	2108.2
2I5	2090	2103	2110	2109.7	2114	2111.2
3I1	1667	1840	1845	1788.4	1845	1814
3I2	1681	1785	1779	1759.9	1779	1769.2
3I3	1461	1742	1774	1759.3	1774	1762.9
3I4	1567	1792	1792	1792	1792	1792
3I5	1563	1772	1775	1751.6	1775	1759.2
4I1	1053	1321	1330	1330	1330	1330
4I2	1199	1378	1378	1378	1378	1378
4I3	1212	1374	1374	1374	1374	1374
4I4	1066	1353	1353	1352.7	1353	1353
4I5	1229	1354	1354	1336.4	1354	1336.4
5I1	2680	2690	2690	2684	2690	2686
5I2	2690	2690	2690	2683.9	2690	2685.9
5I3	2670	2689	2680	2675.7	2690	2679.7
5I4	2680	2690	2698	2683.2	2698	2689.2
5I5	2660	2680	2670	2668	2670	2669.9
6I1	2820	2840	2850	2850	2850	2850
6I2	2800	2820	2830	2823.9	2830	2827.7
6I3	2790	2820	2830	2819.9	2830	2821.9
6I4	2790	2800	2820	2817	2822	2820.2
6I5	2800	2810	2830	2823.7	2830	2825.6
7I1	2700	2750	2780	2771.9	2780	2773
7I2	2720	2750	2770	2769	2770	2770
7I3	2718	2747	2760	2759	2760	2760
7I4	2728	2773	2800	2791	2800	2793
7I5	2730	2757	2770	2763	2770	2765
8I1	2638	2720	2720	2719.1	2720	2719.1
8I2	2659	2709	2720	2719	2720	2720
8I3	2664	2730	2740	2733	2740	2734
8I4	2620	2710	2710	2708.7	2719	2709.9
8I5	2644	2710	2710	2709	2710	2710
9I1	2589	2650	2676	2670.9	2677	2671.3
9I2	2580	2640	2665	2661.5	2665	2663
9I3	2580	2635	2670	2665.8	2670	2668.6
9I4	2540	2630	2660	2659.8	2660	2659.9
9I5	2594	2630	2669	2663.5	2670	2664.9
10I1	2500	2610	2620	2616.7	2620	2619.7
10I2	2549	2642	2630	2627.5	2630	2629.9
10I3	2527	2618	2620	2617.1	2627	2620.5
10I4	2509	2621	2620	2617	2620	2618.6
10I5	2530	2606	2620	2619.3	2625	2620.5
Av. Sol.	2317.88	2390.40	2399.16	2393.63	2400.22	2395.94

Table 4.3: Performance of LNSBH vs Cplex and IRS on the benchmark instances of the literature.

4.5 Conclusion

In this chapter, we proposed a fast large neighborhood search-based heuristic for solving the disjunctively constrained knapsack problem. The proposed method combines a two-phase procedure and a large neighborhood search. First, the two-phase procedure is applied in order to reach a starting feasible solution. This solution is obtained by combining the resolution of two complementary problems: the independent set and the classical binary knapsack. Second, a descent method, based upon degrading and re-optimization strategies, is applied in order to improve the solution provided by the first phase. Once a local optimal solution is reached, the large neighborhood search is used in order to diversify the search space. Finally, the computational results show that the proposed algorithm is very competitive when compared to both Cplex solver and one of the most recent algorithm of the literature. This work is published in an international conference with proceedings *3rd International Symposium on Combinatorial Optimization* (cf., Hifi *et al.* [36]).

In the following chapter, we present the second sequential algorithm for the DCKP. This algorithm is an adaptive algorithm that guides the search process in the feasible solution space towards high quality solutions.

Chapter 5

A guided neighborhood search for the disjunctively constrained knapsack problem

Contents

In this chapter we investigate the use of a guided neighborhood search for solving the disjunctively constrained knapsack problem. The studied problem may be viewed as a combination of two NP-hard combinatorial optimization problems: the weighted independent set and the classical binary knapsack. The proposed algorithm combines both determinist and random local searches. The determinist local search is based on a descent method, where both building and exploring procedures are alternatively used for improving the solution at hand. In order to escape from a local optima, a random local search strategy is introduced which is based on a modified ant colony optimization system. During the search process, the ant colony optimization system tries to diversify and to enhance the solutions by using some informations collected from the previous iterations. Finally, the proposed algorithm is computationally analyzed on a set of benchmark instances available in the literature. The provided results are compared to those realized by both the Cplex solver and a recent algorithm of the literature. The computational part shows that the obtained results improve most existing solution values.

5.1 Introduction

Integer and mixed-inter programming play a central role in modeling NP-hard combinatorial optimization problems. Such models generally serve as a guide to design effective exact methods. However, exact methods derived from these models are often discouraged, especially when tackling large-scale problems. In this case, the availability of effective approximative methods, like heuristics, metaheuristics and hybrid methods, are of paramount importance. In this chapter we investigate the use of an effective guided neighborhood search for approximately solving the DCKP.

The remainder of the chapter is organized as follows. Section 5.2 discusses ant colony optimization system. Section 5.3 presents the principle of the guided neighborhood search for approximately solving the DCKP. Section 5.4 evaluates the performance of the proposed guided neighborhood search on instances taken from the literature, and analyzes the obtained results. Finally, Section 5.5 summarizes the contents of the chapter.

5.2 Ant colony optimization

Ant colony optimization (abbreviated as ACO) is one of the recent techniques for optimization (cf., Maniezzo *et al.* [43]). It was first introduced by Dorigo et al (cf., Dorigo *et al.* [19]) in the early 1990s. In this section, we discuss the biological inspiration of ACO algorithms. We show how the foraging behavior of real ant colonies can be transferred into an algorithm for approximating combinatorial optimization problems.

ACO algorithms were inspired by the foraging behavior of real ant colonies. When they walk to search for food sources from their colony, they can find the shortest pathway to food sources and back. The core of this behavior is based on the indirect communication between the ants by means of a they chemical substance produce, named as pheromone trails. More specifically, while ants walk to search for food sources, they put the pheromone on the ground to mark their path. Ants smell the pheromone trails and they often track the path with the strong pheromone concentration (i.e., the probability with which an ant selects a path increases with the number of ants that previously selected the same path). This interacting among ants facilitates the parallelization of the computational effort in searching for food sources and enables them to find the shortest path between their colony and food sources. Ant colonies present a highly structured social organization that can accomplish complex tasks that are impossible for an individual ant to do (cf., Dorigo *et al.* [18]).

Consider the following example (cf., Figure 5.1): let F be a food source and A be the colony.

Suppose that some ants move from the food source F to the colony A. When

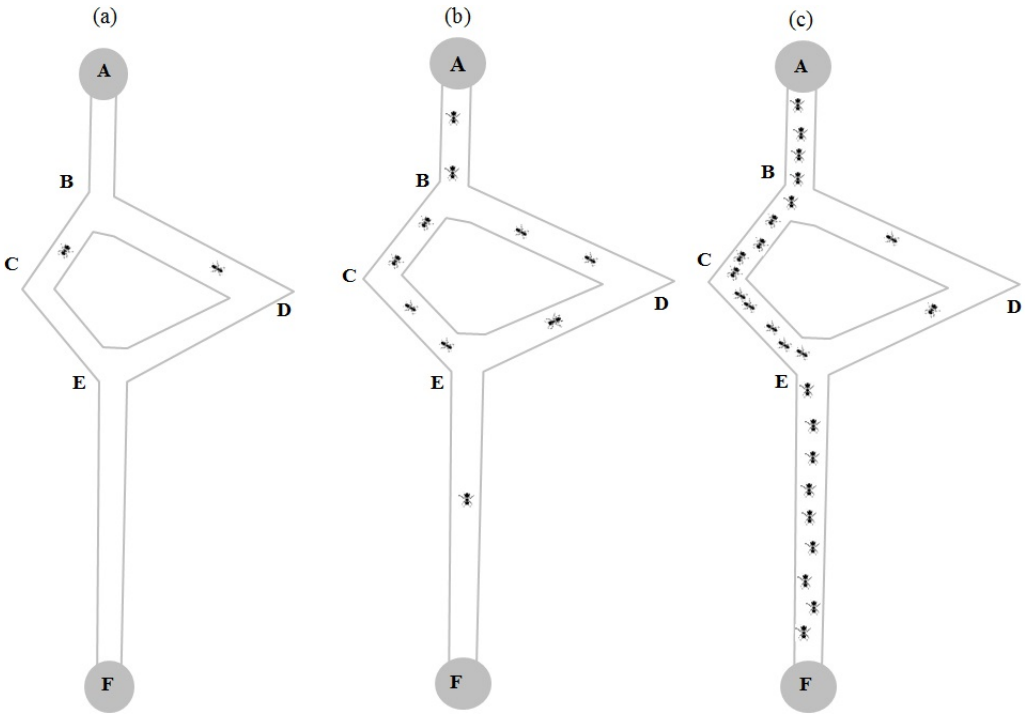


Figure 5.1: Foraging behavior of real ants

they reach point E, they must decide which path should be selected: ECB or EDB. In Figure 5.1 (a), as no pheromone was deposited previously at point E, the first ant reaching this point has the same probability going through either point C or D. Suppose that, the first ant followed path EDB and the second ant followed the path ECB. The second ant will reach point B earlier than the first ant, due to its shorter path. Consequently, an ant coming from colony A will select the path BCE due to the stronger trail. This increases the quantity of the pheromone on the shortest path than that on the longer one. Thus, the probability of selecting the shorter path by an ant is greater.

The behavior of real ants is exploited in artificial models. An analogy of the foraging behavior, the ACO algorithm is based on the indirect communication of a colony of agents, named as artificial ants; they interact with each other via distributed numerical information named as artificial pheromone trails. Ants use the pheromone trail for probabilistically constructing solutions to the problem being addressed, and updated, by the same ants, during the execution process.

However, it has been shown that ACO algorithms can efficiently approximate an important number of combinatorial optimization problems including: knapsack problems (cf., Zhao *et al.* [72]).

5.3 A guided neighborhood search

This section discusses the main principle of the proposed algorithm, which can be viewed as a guided neighborhood search approach[1]. Neighborhood search is an approximative approach that can be used to solve a wide range of combinatorial optimization problems. Generally, neighborhood search-based algorithms are composed of two complementary procedures: (i) a building procedure and (ii) an exploring procedure. The building procedure serves to yield a reduced solution space while the goal of the exploring procedure is to find a local optimum in the yielded space.

Moreover, two classes of neighborhood searches may be distinguished: (i) the descent methods and the random neighborhood searches. The descent methods are often recognized as the family containing the k-optimization, the local branching, the feasibility pump etc. Generally, such methods apply the deterministic strategies in order to build a neighborhood of a given local optimum. Next, the local optimum is improved by using enumerative methods. Then, both building and exploring procedures are successively employed until no further improvement occurs. However, the size of the neighborhood adopted in the descent method should not be large when the exploring procedure uses an enumerative method. Unlike the descent method, which often falls into a local optimum, a random neighborhood search method applies some nondeterministic strategies in order to yield a large size neighborhood. It also uses either exact methods or heuristics for exploring the resulting search space. Recently, such random approaches were widely applied for solving large-scale combinatorial optimization problems (cf., Pisinger *et al.* [55]).

In our study, the proposed algorithm is composed of three procedures: an *outer search procedure*, a *conversion procedure* and an *inner search procedure*. The outer search procedure is based on a modified ACO system (a random neighborhood search), which provides a series of independent sets. Through the execution of the outer search procedure, the conversion procedure is used in order to transform a current independent set to a feasible solution of DCKP while the inner search procedure uses a descent method for improving the DCKP's feasible solution. In what follows, we first introduce the conversion procedure used to compute a DCKP's solution from an independent set.

5.3.1 Building a DCKP's solution from an independent set

This section shows how a DCKP's feasible solution can be built from an independent set. In order to simplify mathematical expressions, all linear programming models will be illustrated as polytopes (or graphical representation with discrete points). We also adopt the following notations:

[1]This work is accepted in May 2015, in an international journal "Cogent Engineering" with the paper: M. Hifi, S. Saleh, and L. Wu. "Hybrid guided Neighborhood Search for the Disjunctively Constrained Knapsack Problem".

$Po(P)$: denotes the polytope of the problem P.

S_P : represents a feasible solution of P.

Opt_P : is an optimal solution of P.

IS^r : denotes an independent set which respects the disjunctive constraints (cf., Equation 4.3) of P_{DCKP}.

Indeed, the DCKP can be viewed as a combination of the *independent set problem* (noted P_{IS}) and the *classical binary knapsack problem* (noted P_K). P_{IS} is obtained from P_{DCKP} by removing the capacity constraint (cf., Equation 4.2) and by setting the profit of all items to one. Formally, the resulting problem can be rewritten as follows:

$$\begin{aligned} (P_{IS}) \quad \max \quad & \sum_{i \in I} x_i \\ \text{s.t.} \quad & x_i + x_j \leq 1 \quad \forall (i,j) \in E \\ & x_i \in \{0,1\} \quad \forall\, i \in I. \end{aligned}$$

Let $S_{IS} = (s_1, \ldots, s_n)$ be a feasible solution of P_{IS}, where s_i denotes the binary value assigned to x_i, $\forall\, i \in I$. Let $IS^r \subseteq I$ be a subset of S_{IS} such that, for all items included in IS^r, their decision variables are fixed to 1. However, S_{IS} may violate the capacity constraint (cf., Equation 4.2) of P_{DCKP}. In order to make it feasible for the DCKP, the following problem is solved :

$$\begin{aligned} (P_K^r) \quad u(x) = \max \quad & \sum_{i \in IS^r} p_i x_i \\ \text{s.t.} \quad & \sum_{i \in IS^r} w_i x_i \leq c, \\ & x_i \in \{0,1\}, \qquad \forall\, i \in IS^r. \end{aligned} \tag{5.1}$$

In order to explain the decomposition strategy, we represent all polytopes related to P_{DCKP}, P_{IS} and P_K^r by a graph, as illustrated in Figure 5.2: we can observe that all feasible solutions of P_{DCKP} belong to both $Po(K)$ and $Po(WIS)$, because $Po(K)$ is characterized by knapsack constraint (cf., Equation 4.2) and integrality constraint (cf., Equation 4.4) whereas $Po(WIS)$ is characterized by disjunctive constraints (cf., Equation 4.3) and integrality constraint (cf., Equation 4.4). Differently stated, P_K dominates P_{DCKP} and the optimal solution Opt_{DCKP} of P_{DCKP} is one of P_{IS}'s feasible solutions. Consequently, the convex hull of P_{DCKP} is the result of the intersection between the convex hull of both P_K and P_{IS}.

Algorithm 14 shows how a feasible DCKP solution can be computed by using IS^r (a feasible independent solution of P_{IS}). It starts by checking whether S_{IS} (an independent set) respects the capacity constraint (cf., Equation 4.2) of P_{DCKP} or not. If the capacity constraint is satisfied, S_{IS} is considered as a

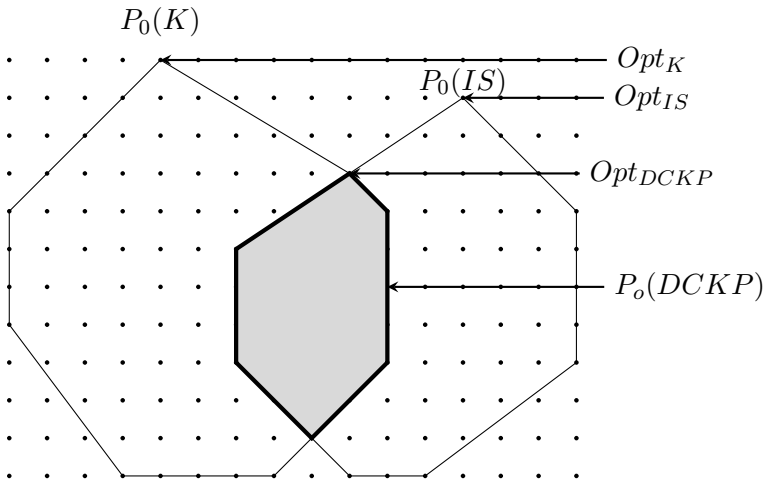


Figure 5.2: Representation of P_K's, P_{IS}'s and P_{DCKP}'s polytopes, and their corresponding optimal solutions.

Input: IS^r, an independent set of P_{IS}, and S_{IS}, a solution of P_{DCKP}.
Output: S_{DCKP}, a feasible DCKP solution.

1: **if** S_{IS} satisfies the capacity constraint (**??**) of P_{DCKP} **then**
2: Set $S_{DCKP} = S_{IS}$;
3: **else**
4: Let P_K^r be the resulting knapsack problem for IS^r;
5: Solve P_K^r by using an exact method and let S_K be the optimum solution;
6: Update S_{DCKP} according to S_K and S_{IS};
7: **end if**
8: **return** S_{DCKP} as a feasible solution of DCKP.

Algorithm 14: A feasible DCKP solution via a classical knapsack problem

feasible solution of P_{DCKP}. Otherwise, P_K^r is solved by using the exact algorithm proposed by Martello *et al.* [44]. Let S_K be a feasible solution of P_K^r: a DCKP's feasible solution can be obtained by setting the decision variables, whose values are fixed to 1 in S_{IS} but 0 in S_K, to 0. In other words, in order to make S_{IS} feasible, Algorithm 14 tries to remove certain items belonging to IS^r.

5.3.2 Using ACO to build a series of independent sets

It is well-known that P_{IS} is an NP-hard combinatorial optimization problem (cf., Garey and Johnson [26]). On the one hand, determining an optimal solution of such a problem may be time-consuming. On the other hand, our objective is to yield a feasible solution of P_{IS} (an independent set), which may contain a high quality solution of DCKP. So, instead of solving P_{IS} exactly, we apply an ACO system for approximately solving P_{IS}.

We recall that ACO is a simple and efficient population-based metaheuristic. It has been first introduced by Dorigo et al (cf., [19]) for approximately solving some large sized combinatorial optimization problems, like the traveling salesman problem. Its principle is based on the observation of real ants which are able to find the shortest path by using the pheromone trails deposited by other ants. In term of optimization, let P be a combinatorial optimization problem which is characterized by a set of decision variables. Assume that for each feasible solution of P there always exists a path for ants toward it. The main idea of ACO is to determine an auto-updating system that highlights the path so that the ants find the most interesting solution. As shown in Colorni *et al.* [13], after a sufficient number of iterations, the most of ants move onto the same path which represents the stability of the system.

An ACO-based algorithm is generally composed of two components: a *path building strategy* and a *pheromone updating strategy*. The path building strategy represents a solution as a path and finds a way of selecting arcs to build the path. The pheromone updating strategy contains two keys: the enhancement and the evaporation of pheromone. On the one hand, the enhancement ensures that the more interesting the path is, the more the ants tend to move onto such a path. On the other hand, in order to avoid that the search procedure converges quickly to a local optima, the pheromone trail evaporates with the time passing.

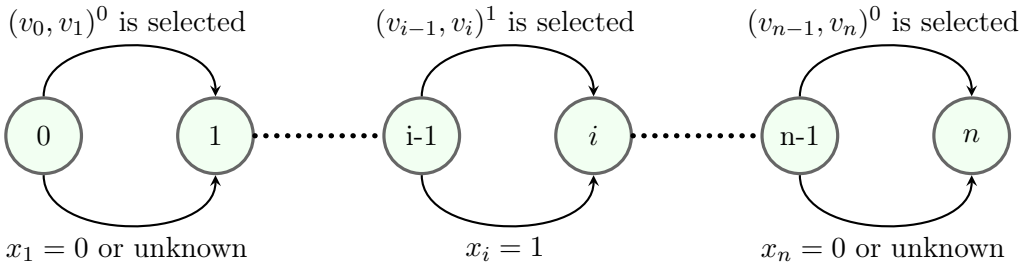


Figure 5.3: Representation of an independent set solution through the use of the path building strategy.

Following the mechanism used by Fidanova [22], a similar algorithm was introduced for solving P_{IS}. Indeed, Figure 5.3 illustrates the mechanism adopted: a directed graph $G = (V, A)$, where V denotes its vertex set (i.e., $V = \{v_0, \ldots, v_n\}$) and A is the set of arcs (i.e., $A = \{(v_i, v_{i+1})^k \mid \forall\ v_i \in V \text{ and } k \in \{0, 1\}\}$). For any travel from vertex i to $i + 1$, either $(v_i, v_{i+1})^0$ or $(v_i, v_{i+1})^1$ can be chosen, but not at the same time. For all i, $i \in I$, the vertex i indicates the state of the decision variable x_i of P_{IS}. In this case, x_i is fixed to 1 if $(v_{i-1}, v_i)^1$ is selected; x_i is unfixed otherwise. Further, x_i is fixed to 0 if and only if the item i is incompatible with one of the items whose decision variables have already fixed to 1. Once the vertex x_i is fixed to 0 (resp. 1), the ant must select the arc $(v_{i-1}, v_i)^0$

(resp. $(v_{i-1}, v_i)^1$) to follow. Such building continues when all decision variables of P_{IS} are fixed. Note that if the ant has arrived to the n-th vertex with some unfixed variables, then it restarts the selection process from the vertex 0.

Input: an instance I of P_{DCKP}.
Output: a feasible solution (independent set) IS^r for P_{IS}.

1: **Initialization:** set $IS^r = \emptyset$ and $S = \{1, \ldots, n\}$.
2: Rank all items of S following the non-increasing order of their reduced costs; set $i = 1$;
3: **while** $S \neq \emptyset$ **do**
4: Randomly select an arc $(v_{i-1}, v_i)^k$ $(k = 0, 1)$ related to item i;
5: **if** $(v_{i-1}, v_i)^1$ is selected **then**
6: Set $IS^r = IS^r \cup \{i\}$; remove i from S; remove all incompatible items with i from S;
7: **end if**
8: If $i = n$ and $S \neq \emptyset$, reset i as the first item included in S;
9: **end while**
10: **return** IS^r as a feasible solution of P_{IS}.

Algorithm 15: Determining of an independent set as a solution for P_{IS}

In our study, all items are ranked in non-increasing order of their reduced costs which can be obtained by solving the linear relaxation of P_{DCKP} with the simplex method. Instead of ranking items according to the profit per weight or the degree, we favor the reduced cost because it combines informations related to the objective function (cf., Equation 4.1), the capacity constraint (cf., Equation 4.2) and all disjunctive constraints (cf., Equation 4.3). Algorithm 15 starts by initializing IS^r to an empty set (a feasible solution of P_{IS}) and ranking all items of S in non-increasing order of their reduced costs. In the main loop, an ant selects an item with a certain probability, and removes the selected item i and all its incompatible items from S. The selection procedure iterates until no item can be added to the current independent set IS^r and the algorithm stops with a feasible solution IS^r of P_{IS}.

However, while building a solution or a path, the probability of the selection of an arc made by an ant depends on the density of pheromone deposited on the arc by the other ants. Herein, for the path building strategy, the following tunings are considered. First of all, the items are ranked in non-increasing order of their reduced costs. Let τ_{ik}^t be the amount of pheromone deposited on the arc $(v_{i-1}, v_i)^k$, $\forall\ i,\ i \in I$, and $k \in \{0, 1\}$) at the t-th iteration. The probability of selecting the arc $(v_{i-1}, v_i)^k$ at the t-th iteration can be defined as follows:

$$p_{ik}^t = \frac{\tau_{ik}^t}{\tau_{i0}^t + \tau_{i1}^t}.$$

Indeed, the ant randomly selects arcs to traverse the graph, but the probability of

selecting arcs is dynamically updated at the end of each iteration. The pheromone updating strategy generally depends on the local optima obtained from previous iterations and the number of iterations already completed. Formally, at the end of the t-th iteration, the amount of pheromone on the arc (i, j) is updated according to the following equality:

$$\tau_{ik}^{t+1} = (1 - \rho) \cdot \tau_{ik}^{t} + \Delta_{ik}^{t}, \quad \forall\ i \in I \text{ and } k \in \{0, 1\},$$

where $0 < \rho < 1$ is the evaporation parameter and Δ_{ik}^{t} denotes the reinforcement value on the arc $(v_{i-1}, v_i)^k$ at the t-th iteration. Note that the evaporation parameter ρ adjusts the diversity of solutions computed by ACO whereas the reinforcement value Δ_{ik}^{t} adjusts the convergence of ACO to a local optima. However, the right choice of both parameters can lead the ant system to good local optima. Unfortunately, the values of both parameters are experimentally determined. Our limited computational results showed that the following adjustments realized a good behavior for the convergence of ACO: $\rho = 0.999$. Indeed, when ρ is very close to 1, the evaporation is more important (this helps the ACO system to escape from a local optimum). The reinforcement value Δ_{ik}^{t}, $\forall\ i,\ i \in I$, and $k \in \{0, 1\}$:

$$\Delta_{ik}^{t} = \begin{cases} f(u^t) & \text{if } (v_{i-1}, v_i)^k \text{ is selected at the } t\text{-th iteration} \\ 0 & \text{otherwise,} \end{cases}$$

where u^t (resp. $u^\star$) is the current solution value (resp. the best solution value found). In order to have a good behavior of ACO system we adopt the following formula:

$$f(u^t) = \frac{e^{\,g(u^t)\cdot P}}{Q}, \quad \text{where} \quad g(u^t) = \frac{u^t}{u^\star + (u^\star - u^t)}. \tag{5.2}$$

where, $f(.)$ is a function of u^t.

Note that the parameter P of Formula (5.2) is used for adjusting the growth rate of u^t and so, that of $f(u^t)$. The parameter Q is used for adjusting the equilibrium between reinforcement and evaporation of pheromones. Differently stated, according to the parameters used in Formula (5.2), the better solution is, the more pheromones are deposited on the corresponding arcs.

5.3.3 The descent method as a local search

In order to improve a "local optimal" solution of P_{DCKP}, this section presents a descent method which is based on a deterministic building procedure and an exact exploring procedure. Figure 5.4 illustrates the main principle of the descent method that we applied to DCKP. The “dark area" of Figure 5.4 represents a neighborhood of a feasible DCKP solution. Such an area, a sub-solution space, can be obtained by removing a percentage of items whose decision variables have already been fixed in S_{DCKP} (i.e., the value of each variable is equal either to

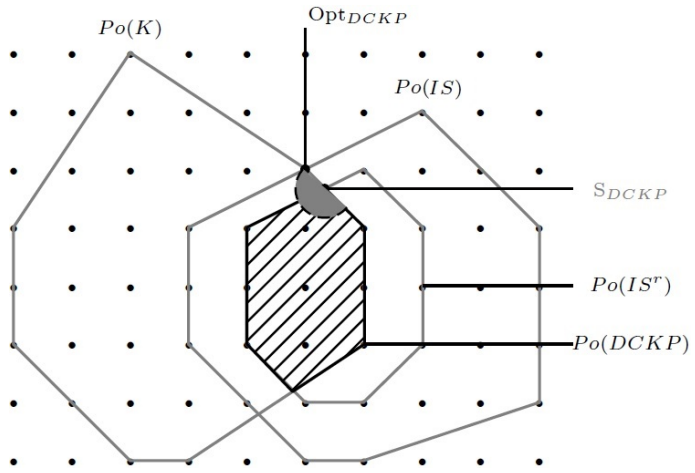


Figure 5.4: Illustration of the neighborhood of a feasible DCKP solution.

1 or 0). Therefore, the descent method (cf., Algorithm 16) consists in building and exploring a series of such areas until no further improvement occurs.

Let S_{DCKP} be a feasible DCKP solution provided by Algorithm 14 and α, where $\alpha \in [0, 100]$, be the percentage of the unassigned decision variables. Then, we consider $S^{(\alpha)}$ as the unassigned items set associated to S_{DCKP} and $\overline{S^{(\alpha)}}$ as the complementary set of $S^{(\alpha)}$ such that $S^{(\alpha)} \cup \overline{S^{(\alpha)}} = I$ (i.e., $\overline{S^{(\alpha)}}$ is the assigned items set associated to S_{DCKP}). Therefore, the descent method (cf., Algorithm 16) consists in building and solving alternatively the following linear program:

$$
\begin{aligned}
(P^{(\alpha)}_{DCKP}) \quad u(x) = \quad \text{Max} \quad & \sum_{i \in S^{(\alpha)}} p_i x_i \\
\text{s.t.} \quad & \sum_{i \in S^{(\alpha)}} w_i x_i \leq c^{(\alpha)} \\
& x_i + x_j \leq 1 \qquad \forall (i,j) \in E^{(\alpha)} \\
& x_i \in \{0,1\} \qquad \forall i \in S^{(\alpha)},
\end{aligned}
$$

where $E^{(\alpha)}$ denotes the subset of incompatible items belonging to $S^{(\alpha)}$ and $c^{(\alpha)}$ is the residual capacity such that $c^{(\alpha)} = c - \sum_{i \in \overline{S^{(\alpha)}}} x_i w_i$. From Figure 5.4, we can observe that a local optimal solution is used as an interior point of $Po(DCKP)$ to generate a $P^{(\alpha)}_{DCKP}$ by erasing the value of $\alpha\%$ of the variables. Further, a new local optimal solution can be reached by exactly solving the resulting problem $P^{(\alpha)}_{DCKP}$ (in dark area).

According to the presentation detailed above, we now describe the principle of the decent method (cf., Algorithm 16). The algorithm starts with a feasible

Input: S_{DCKP}, a starting solution of DCKP.
Output: $S^{\star}_{DCKP}$, an improved solution of DCKP.

1: Set $u(S^{\star}_{DCKP}) = -\infty$.
2: **while** ($u(S_{DCKP}) > u(S^{\star}_{DCKP})$) **do**
3: Update $S^{\star}_{DCKP}$ with S_{DCKP}.
4: Let $S^{(\alpha)}$ (resp. $\overline{S^{(\alpha)}}$) be the set of unassigned (resp. assigned) variables according to S_{DCKP}.
5: Let $P^{(\alpha)}_{DCKP}$ be a restricted subproblem according to $S^{(\alpha)}$.
6: Solve $P^{(\alpha)}_{DCKP}$ and let $S^{(\alpha)}_{DCKP}$ be its solution.
7: Update S_{DCKP} according to $\overline{S^{(\alpha)}}$ and $S^{(\alpha)}_{DCKP}$.
8: **end while**
9: **return** $S^{\star}_{DCKP}$.

Algorithm 16: Descent local search procedure

DCKP solution (noted S_{DCKP}) reached by applying Algorithm 14. The main loop (cf., lines 2-8) describes the main steps of the descent method. At line 3, $S^{\star}_{DCKP}$ is updated with S_{DCKP}, an improved DCKP solution provided in the last iteration. Line 4 determines a set $S^{(\alpha)}$ of $\alpha\%$ unassigned variables from the current solution S_{DCKP}, where items with highest degree (i.e., items having the greatest number of incompatible items) are favored. Note that, the incompatible items of the unassigned variables are then can be used for the next step in solving the reduced problem. Then, the reduced problem $P^{(\alpha)}_{DCKP}$ related to $S^{(\alpha)}$ is defined and solved (lines 5 and 6). We use $S^{(\alpha)}_{DCKP}$ to denote the optimal solution of $P^{(\alpha)}_{DCKP}$. At line 7, S_{DCKP} is updated according to the assigned items set $\overline{S^{(\alpha)}}$ and the optimal solution $S^{(\alpha)}_{DCKP}$ related to the unassigned items set $S^{(\alpha)}$. The main loop is iterated until no better solution is obtained. Finally, Algorithm 16 stops with the best solution $S^{\star}_{DCKP}$ of value $u(S^{\star}_{DCKP})$ found so far.

5.3.4 An overview of the guided neighborhood search

In this section, the main steps of the guided neighborhood search (noted GNS) are detailed. GNS can be considered as a hybrid neighborhood search which combines two complementary neighborhood strategies: (i) a random neighborhood search and (ii) a deterministic neighborhood search. In order to combine both random and deterministic strategies, we propose to decompose the DCKP into two complementary problems, i.e., the weighted independent set and the classical binary knapsack. A series of these complementary problems are iteratively solved for reaching a final solution for the DCKP. More precisely, the random strategy mimics an ACO system to provide a series of independent sets containing high quality solutions of DCKP. For each obtained independent set, a conversion procedure is used in order to convert an independent set to a feasible solution

of DCKP. Once a feasible solution is obtained, the deterministic neighborhood search applies a descent method for improving the current solution. Finally, the DCKP's solution provided by the descent method is used to update the ACO system.

Input: An instance I of DCKP.
Output: A near-optimal solution for the DCKP.

1: **Initialization:** set $S^{\star}_{DCKP} = \emptyset$.
2: **while** stopping criteria is not satisfied **do**
3: Set $i = 0$ and $S^{\circ}_{DCKP} = \emptyset$;
4: **while** $(i < m)$ **do**
5: Set $S^{local}_{DCKP} = \emptyset$;
6: The i-th ant applies Algorithm 15 to build the solution IS^r of P_{IS} according to the current state of the pheromones laid on the arcs.
7: Let S_{DCKP} be a new DCKP solution reached by Algorithm 14 according to IS^r.
8: **If**$(u(S_{DCKP}) > u(S^{local}_{DCKP}))$, **then** set $S^{local}_{DCKP} = S_{DCKP}$.
9: **end while**
10: Let S°_{DCKP} be the solution provided by Algorithm 16 when considering S^{local}_{DCKP}.
11: **If** $(u(S^{\circ}_{DCKP}) > u(S^{\star}_{DCKP}))$, **then** set $S^{\star}_{DCKP} = S^{\circ}_{DCKP}$.
12: According to the best local solution S^{local}_{DCKP}, determine a new independent set solution IS'.
13: Update the pheromones according to $u(S^{\star}_{DCKP})$, $u(S^{\circ}_{DCKP})$ and IS'.
14: **end while**
15: **return** $S^{\star}_{DCKP}$, the best solution found for the DCKP.

Algorithm 17: Guided neighborhood search for the DCKP: GNS

Algorithm 17 describes the main steps of GNS. At step 1, the best solution of DCKP is initialized to the empty set ($S^{\star}_{DCKP} = \emptyset$). Then, the main loop of GNS (steps 2-14) describes the principle of the proposed ACO system. It stops when the stopping criteria is performed. In the main loop, the inner loop (steps 4-9) is represented by m ants, where each of them applies Algorithm 15 to provide an independent set IS^r, and uses Algorithm 14 to compute a feasible solution of DCKP from IS^r. The best solution of DCKP realized by m ants is then saved in S^{local}_{DCKP}. Step 10 calls Algorithm 16 for improving the current local solution S^{local}_{DCKP} by using a descent local search (the best solution found so far is noted by S°_{DCKP}). Step 11 updates $S^{\star}_{DCKP}$ according to the current solution S°_{DCKP}. At step 12, we extend S°_{DCKP}, the best solution reached by the descent method, to an independent set IS' by adding the items which are compatible with the items belonging to S°_{DCKP}. Step 13 updates the pheromones used by the ACO system according to the current best objective value $u(S^{\star}_{DCKP})$, the current local

optimal objective value $u(S^{\circ}_{DCKP})$ and the independent set IS' related to the current optimal solution. Finally, the main loop is iterated until performing the stopping criteria and the algorithm stops with the best solution $S^{\star}_{DCKP}$ for the DCKP.

5.4 Computational results

This section investigates the effectiveness of the proposed *Guided Neighborhood Search* (GNS) on two groups of instances (taken from Hifi and Michrafy [31] and generated following Yamada *et al.*'s [69]). As detailed in Table 5.1, the first group, labeled from 1IAx to 4IAx with $1 \leq x \leq 5$, contains 20 medium instances with $n = 500$ items, a capacity $c = 1800$, and different densities ρ (assessed in terms of the number of disjunctive constraints $|E|$). The second group, labeled from 5IAx to 10IAx with $1 \leq x \leq 5$, contains 30 large instances, where each instance contains 1000 items, with $c = 1800$ or 2000 and with various densities. The algorithm GNS was coded in C++ and tested on an Intel Pentium Core i5-2500 with 3.3 Ghz.

# Inst.	n	c	ρ	$\|E\|$	# Inst.	n	c	ρ	$\|E\|$
1IAx	500	1800	0.10	12475	6IAx	1000	2000	0.06	29970
2IAx	500	1800	0.20	24950	7IAx	1000	2000	0.07	44955
3IAx	500	1800	0.30	37425	8IAx	1000	2000	0.08	39960
4IAx	500	1800	0.40	49900	9IAx	1000	2000	0.09	44955
5IAx	1000	1800	0.05	24975	10IAx	1000	2000	0.10	49950

Table 5.1: Characteristics of the medium and large scale instances.

5.4.1 Effect of the descent method

This section evaluates the effect of the descent local search (cf., Algorithm 16). We recall that the method begins with a starting solution and improves it by alternatively applying both building and exploring procedures. Herein, the reduced problems provided by the building procedure are solved by using the Cplex solver. In order to provide a starting solution, as an input for Algorithm 16, we first apply Algorithm 14 on the independent set built by applying the following simple greedy procedure: (i) from the current problem, select an unassigned item with the highest reduced cost, (ii) add the selected item to the current independent set, (iii) remove the selected item with its incompatible items to get a reduced problem and (iv) repeat steps from (i) to (iii) until reducing the problem to the empty set.

Table 5.2 shows the variation of Av_Sol (the average solution values provided by the considered algorithm over all instances), Av_Imp (the percentage improvement relative to the starting solution values), and Av_time (the average

		Algorithm 16: *the descent local search*			
		Variation of α			
Algorithm	Algorithm 14	5%	10%	15%	20%
Av. Sol.	1413.38	2129.98	2188.80	2232.36	2217.04
Av. Imp.		33.64	35.43	36.69	36.25
Av. time	0.32	0.47	2.33	20.02	118.6

Table 5.2: Effect of the descent local search.

runtime required by each algorithm for reaching the results). In order to evaluate the impact of α on the behavior of the descent method, Algorithm 16 is performed by varying the value of α in the interval $\{5\%, 10\%, 15\%, 20\%\}$. From Table 5.2, we observe what follows.

1. First, Algorithm 14 is able to provide the competitive results even if the average runtime remains rather small (0.32 sec).

2. Second, by setting $\alpha = 5\%$, Algorithm 16 is able to improve the quality of the solutions realized by Algorithm 14. Indeed, it obtains an average improvement of 33.64% whereas it consumes an average runtime of 0.47 *sec*; that is slightly larger than that of Algorithm 14.

3. Third, by increasing the value of α to either 5% or 10%, Algorithm 16 is able to provide more interesting results. Moreover, this adjustment consumes a longer runtime even if it can be considered as reasonably small (it needs an average of 2.33 sec).

4. Fourth and last, Algorithm 16 with both $\alpha = 15\%$ and $\alpha = 20\%$ realized better average solution quality, even though the results of the first assignment ($\alpha = 15\%$) dominate those of the second assignment ($\alpha = 20\%$). These results are achieved by consuming a much longer runtime. Indeed, Algorithm 16 with $\alpha = 15\%$ consumes an average runtime close to 20 seconds whereas with $\alpha = 20\%$ it needs an average runtime of 120 seconds.

Recall that Algorithm 16 is generally used in order to refine the solutions yielded by the ACO system. Therefore, it is interesting to find a judicious compromise between the quality of solutions reached by the descent method and the used runtime. Then, in order to maintain a reasonable runtime for GNS, the choice with α varying in the interval $[5\%, 10\%]$ is adopted for the remainder of this chapter.

5.4.2 Performance of GNS

In order to evaluate the performance of the proposed method, we first compare GNS with Cplex solver (version 12.4) and IRS −Iterative Rounding Search− proposed in Hifi [29] (one of the most recent methods in the literature). Because

of the high complexity of DCKP, two runtime limits of Cplex are considered: 3600 seconds and 10000 seconds. For GNS, two runtime limits are considered: $t_1 = 500$ and $t_2 = 1000$ (measured in seconds).

We recall that GNS is based on an ant colony optimization system which is performed with certain parameters (cf. Section 5.3.2). Herein, we tried to find a good tuning that ensures a balance between the convergence of ACO and its solutions' quality. Then, after different trials, two tunings are used. Indeed, for the first runtime t_1, ACO uses the following tunings: $\rho = 0.999$, $m = 10$, $P = 10$ and $Q = 500$. For the second runtime t_2, it uses the following values: $\rho = 0.9995$, $m = 10$, $P = 10$ and $Q = 500$. Further, for each considered runtime limit, GNS is performed with different values of α varying from 5% to 10%. Because of the stochastic aspect of the ACO system, ten independent trials of GNS were performed for all considered instances.

	V_{Cplex}		V_{IRS}	GNS: *variation of* α						
	3600	10000	$1000s$	cpu	10%	9%	8%	7%	6%	5%
nb_{best}	3	4	15	t_1	48	47	44	40	34	34
				t_2	48	47	47	47	46	43

Table 5.3: A comparative study between Cplex, IRS and GNS.

Table 5.3 exposes the number of the best solutions reached by each considered approach (cf. Table 5.4). Columns 2 and 3 (V_{Cplex}) (resp. column 4 (V_{IRS})) show nb_{best}, the number of the best solutions matched by the Cplex solver (resp. IRS). Column 5 displays the runtime limits used by GNS, i.e., with $t_1 = 500$ and $t_2 = 1000$ seconds. For each runtime limit, columns from 6 to 11 give the number of the best solutions realized by GNS when varying α. From Table 5.3, we observe what follows.

1. First, on the one hand, we can observe the inferiority of Cplex because it only matches three (resp. four) instances over 50 within 3600 (resp. 100000) seconds. It represents a percentage of 6% (resp. 8%) of the best solutions of the literature. On the other hand, the most recent method of the literature provides 15 best solutions out of 50 within 1000 seconds, representing a percentage of 30% of the best solutions. Note that among the best solutions, GNS matches 2 optimal solutions (as shown in Table 5.4: instances 1I1 and 1I2 have been optimally solved by the Cplex solver within 10000 seconds −solutions marked with the symbol "$\star$").

2. Second, for the first runtime $t_1 = 500$, the number of the best solutions reached by GNS increases. Indeed, for $\alpha = 10\%$ (resp. 9%) GNS reaches 48 (resp. 47) best solutions out of 50, representing a percentage of 96% (resp. 94%) of the best solutions.

3. Third, for the second runtime $t_2 = 1000$, the quality of the solutions provided by GNS becomes more interesting. Indeed, for $\alpha = 8\%$ (resp 7%, 6% and 5%), the percentage of nb_{best} increases from 88% to 94% (resp. from 80% to 94%, from 68% to 92% and, from 68% to 86%, respectively).

Because GNS is able to provide better solutions for $\alpha = 9\%$ and $\alpha = 10\%$, both values are adopted for the rest of the chapter.

Table 5.4 shows the solution values reached by Cplex, IRS and GNS. Column 1 displays the data labels and column 2 reports the best solutions found by the considered algorithms. Columns 3 (resp. column 4) reports the solution values (V_{Cplex}) reached by Cplex v12.4 within 3600 (resp. 10000) seconds. Column 5 displays the best known solution values available in the literature, noted by V_{IRS} within 1000 seconds. For the first runtime $t_1 = 500$, the best value $V_{\text{GNS}_{(10)}}$ (resp. $V_{\text{GNS}_{(9)}}$) realized by GNS with $\alpha = 10\%$ (resp. $\alpha = 9\%$), for ten independent trials, are displayed on column 6 (resp. 7). Similarly, columns 8 and 9 show the solution values reached by GNS for the second runtime $t_2 = 1000$. We note that the value in "boldface" means that the best solution value has been obtained by the considered algorithm, the value in italic indicates that GNS reaches a better solution value than IRS, and the value with the symbol " $\star$ " means that the method reaches the optimal solution value.

In order to evaluate the behavior of GNS, which includes a probabilistic procedure, ten trials of GNS were considered. According to these trials, Table 5.5 shows the average results of the solution values reached by these trials and those displayed in Table 5.4. From Table 5.5, we observe what follows:

1. The average value of 2400.86 confirms the superiority of GNS when both values t_2 and $\alpha = 9\%$ are used. The average solution values corresponding to the couple (cpu, α)=$(t_2, 10\%)$ remains very competitive because it realizes an average solution value of 2400.56 which is very close to the best average solution values, i.e., the value 2400.86 realized by the couple $(t_2, 9\%)$.

2. Both tunings $(t_1, 9)$ and $(t_1, 10)$ remain interesting when compared to the best average solution values realized by both Cplex (2317.88 with 3600 seconds and 2354.74 with 10000 seconds) and the best average solution values taken from the literature (2390.40 within 1000 seconds).

3. Over ten trials, with the first runtime limit t_1, GNS maintains its superiority over Cplex and the best solutions of the literature. Indeed, its average value is equal to 2391.67 for $\alpha = 10\%$ and increases slightly to 2392.99 for $\alpha = 9\%$.

4. Globally, by doubling the runtime limit (i.e., $t_2 = 1000$), the behavior of GNS becomes more interesting. Indeed, in this case, the proposed algorithm realizes an average value of 2393.64 for $\alpha = 10\%$ and 2396.74 for $\alpha = 9\%$.

		V_{Cplex}		V_{IRS}	Guided Neighborhood Search: V_{GNS}			
					First runtime t_1		Second runtime t_2	
Instance	Best	3600	10000	1000	10%	9%	10%	9%
1I1	2567*	**2567***	**2567***	**2567***	**2567***	**2567***	**2567***	**2567***
1I2	2594*	**2594***	**2594***	**2594***	**2594***	**2594***	**2594***	**2594***
1I3	2320	**2320**	**2320**	**2320**	**2320**	**2320**	**2320**	**2320**
1I4	2310	2298	2300	2303	**2310**	**2310**	**2310**	**2310**
1I5	2330	2310	2320	2310	**2330**	**2330**	**2330**	**2330**
2I1	2118	2080	2080	2100	**2118**	*2117*	**2118**	**2118**
2I2	2110	2070	2080	**2110**	**2110**	**2110**	**2110**	**2110**
2I3	2132	2098	2098	2128	**2132**	**2132**	**2132**	**2132**
2I4	2109	2070	2080	2107	**2109**	**2109**	**2109**	**2109**
2I5	2114	2090	2090	2103	*2110*	**2114**	*2110*	**2114**
3I1	1845	1667	**1845**	1840	**1845**	**1845**	**1845**	**1845**
3I2	1795	1681	1681	1785	**1795**	**1795**	**1795**	**1795**
3I3	1774	1461	1588	1742	**1774**	**1774**	**1774**	**1774**
3I4	1792	1567	1753	**1792**	**1792**	**1792**	**1792**	**1792**
3I5	1794	1563	1772	1772	**1794**	**1794**	**1794**	**1794**
4I1	1330	1053	1187	1321	**1330**	**1330**	**1330**	**1330**
4I2	1378	1199	1223	**1378**	**1378**	**1378**	**1378**	**1378**
4I3	1374	1212	1318	**1374**	**1374**	**1374**	**1374**	**1374**
4I4	1353	1066	1284	**1353**	**1353**	**1353**	**1353**	**1353**
4I5	1354	1229	1268	**1354**	**1354**	**1354**	**1354**	**1354**
5I1	2690	2680	2680	**2690**	**2690**	**2690**	**2690**	**2690**
5I2	2700	2690	2690	2690	**2700**	**2700**	**2700**	**2700**
5I3	2690	2670	2680	2689	**2690**	**2690**	**2690**	**2690**
5I4	2700	2680	2680	2690	**2700**	**2700**	**2700**	**2700**
5I5	2680	2660	2660	**2680**	**2680**	**2680**	**2680**	2670
6I1	2850	2820	2820	2840	**2850**	**2850**	**2850**	**2850**
6I2	2830	2800	2820	2820	2820	**2830**	**2830**	*2829*
6I3	2830	2790	2790	2820	2820	2820	**2830**	**2830**
6I4	2829	2790	2800	2800	*2820*	**2829**	*2828*	**2829**
6I5	2830	2800	2810	2810	2829	**2830**	**2830**	**2830**
7I1	2780	2700	2727	2750	**2780**	*2770*	**2780**	*2770*
7I2	2770	2720	2720	2750	**2770**	**2770**	**2770**	**2770**
7I3	2760	2718	2740	2747	**2760**	**2760**	**2760**	**2760**
7I4	2800	2728	2750	2773	*2790*	*2790*	*2790*	**2800**
7I5	2770	2730	2750	2757	*2760*	**2770**	*2760*	*2760*
8I1	2720	2638	2686	**2720**	**2720**	**2720**	**2720**	**2720**
8I2	2720	2659	2690	2709	**2720**	**2720**	**2720**	**2720**
8I3	2740	2664	2690	2730	2730	**2740**	**2740**	**2740**
8I4	2720	2620	2670	2710	*2719*	2710	*2719*	**2720**
8I5	2710	2644	2659	**2710**	**2710**	**2710**	**2710**	**2710**
9I1	2677	2589	2600	2650	**2677**	*2670*	**2677**	*2670*
9I2	2666	2580	2620	2640	*2660*	*2660*	*2660*	**2666**
9I3	2670	2580	2620	2635	*2660*	*2660*	*2669*	**2670**
9I4	2668	2540	2620	2630	*2659*	*2660*	*2660*	**2668**
9I5	2670	2594	2610	2630	*2660*	*2660*	*2660*	**2670**
10I1	2620	2500	2570	2610	2610	*2617*	*2618*	**2620**
10I2	2642	2549	2578	**2642**	2630	2620	2630	2630
10I3	2620	2527	2580	2618	**2620**	2610	**2620**	**2620**
10I4	2621	2509	2560	**2621**	2610	2620	2618	2618
10I5	2630	2530	2550	2606	**2630**	*2620*	**2630**	**2630**
Av. Sol.	2401.92	2317.88	2354.74	2390.40	2399.26	2399.36	2400.56	2400.86

Table 5.4: Performance of GNS vs Cplex and IRS on the benchmark instances of the literature. The symbol "$\star$" means that the used method reaches an optimal solution value.

	V_{Cplex}		V_{IRS}	Guided Neighborhood Search			
				First runtime limit t_1		Second runtime limit t_2	
	3600	10000	1000	10%	9%	10%	9%
Av. best Sol.	2317.88	2354.74	2390.40	2399.26	2399.36	2400.56	2400.86
Av. Sol. 10 trials				2391.67	2392.99	2393.64	2396.74

Table 5.5: Behavior of GNS vs Cplex and IRS.

5.4.3 Detailed results

This section provides a detailed study of the results reached by GNS over ten trials. Indeed, we have already mentioned that ten trials of the GNS have been performed for each of the fifty problem instances. These trials are performed for both $\alpha = 9\%$ and $\alpha = 10\%$ and each trial is stopped when the runtime limit t_1 (resp. t_2) is performed.

	t1		t2	
Trial	10	9	10	9
1	2391.48	2391.32	2394.58	2396.62
2	2390.66	2393.74	2391.82	2397.54
3	2390.80	2392.04	2392.28	2395.90
4	2392.92	2393.16	2394.08	2396.72
5	2391.52	2394.98	2394.14	2397.26
6	2392.20	2391.58	2393.72	2395.74
7	2391.80	2393.38	2393.22	2396.60
8	2392.30	2393.16	2393.66	2396.50
9	2392.14	2392.84	2394.66	2397.60
10	2390.92	2393.64	2394.20	2396.94
G. Av.	2391.67	2392.99	2393.64	2396.74

Table 5.6: Impact of the parameter used by GNS on the average solution quality.

Table 5.6 shows the average quality of the solutions reached by GNS over the ten trials and the detailed results are exposed in Tables 5.7-5.10.

Line 1 (resp. line 2) of Table 5.6 displays the runtime limits (resp. the value of the parameter α). Lines from 3 to 12 display the average solution quality over all instances, line 13 shows the average solution quality over the ten trials. The analysis of Table 5.6 shows what follows:

1. For the first couple of values $(t_1, 10)$, on the one hand, GNS realizes an average solution value varying from 2390.66 (trial 2) to 2392.92 (trial 4). On the other hand, the smallest average solution value realized by GNS is still greater than the best average value realized by IRS (that is equal to 2390.40 −cf., Table 5.4). The same observation holds for $(t_1, 9)$, where the minimum average solution value realized by GNS is equal to 2391.32 while

its maximum average solution value is equal to 2394.98.

2. By increasing the runtime limit (i.e., using t_2 instead of t_1), for either $\alpha = 10\%$ or $\alpha = 9\%$, one can observe that GNS realizes better average solution qualities. Indeed, for the parameter $\alpha = 10\%$ (resp. $\alpha = 9\%$), it reaches a minimum average solution value of 2391.82 (resp. 2395.74) and a maximum average solution value of 2394.66 (resp. 2397.60).

5.5 Conclusions

In this chapter, we proposed a guided neighborhood search for approximately solving the disjunctively constrained knapsack problem. The proposed algorithm applies an ant colony optimization (ACO) system to guide a neighborhood search procedure towards high quality solutions. In order to improve the quality of the solutions provided by ACO, a descent local search procedure is introduced. Both ACO and the descent method are based on building and exploring a series of neighborhoods related to a series of local optima. The performance of the proposed algorithm was computationally analyzed on a set of benchmark instances taken from the literature. The provided results were compared to those realized by both the Cplex solver and a recent algorithm available in the literature. The obtained results show that the guided neighborhood search was able to improve most existing solutions. This work is accepted in May 2015, in an international journal "Cogent Engineering" with the paper: M. Hifi, S. Saleh, and L. Wu. "Hybrid guided neighborhood search for the disjunctively constrained knapsack problem".

Inst.	1	2	3	4	5	6	7	8	9	10	AV. Sol.	Max
1I1	2567	2567	2567	2567	2567	2567	2567	2567	2567	2567	2567.00	2567
1I2	2594	2594	2594	2594	2594	2594	2594	2594	2594	2594	2594.00	2594
1I3	2320	2320	2320	2320	2320	2320	2320	2320	2320	2320	2320.00	2320
1I4	2310	2310	2310	2310	2310	2310	2306	2310	2310	2310	2309.60	2310
1I5	2330	2330	2330	2330	2330	2330	2330	2330	2330	2330	2330.00	2330
2I1	2115	2117	2110	2117	2114	2115	2117	2115	2118	2117	2115.50	2118
2I2	2110	2110	2110	2110	2110	2110	2110	2110	2110	2110	2110.00	2110
2I3	2115	2111	2111	2110	2111	2112	2108	2132	2118	2110	2113.80	2132
2I4	2100	2109	2109	2100	2100	2100	2100	2109	2105	2107	2103.90	2109
2I5	2110	2110	2110	2110	2110	2108	2108	2109	2108	2110	2109.30	2110
3I1	1743	1720	1714	1845	1845	1845	1778	1845	1845	1778	1795.80	1845
3I2	1779	1757	1795	1779	1779	1779	1795	1795	1757	1757	1777.20	1795
3I3	1774	1718	1712	1774	1774	1774	1774	1711	1774	1774	1755.90	1774
3I4	1792	1753	1792	1753	1753	1757	1792	1753	1792	1753	1769.00	1792
3I5	1775	1775	1748	1794	1721	1794	1735	1775	1777	1794	1768.80	1794
4I1	1330	1330	1330	1330	1330	1330	1330	1330	1330	1330	1330.00	1330
4I2	1378	1378	1378	1378	1378	1378	1378	1378	1378	1378	1378.00	1378
4I3	1374	1334	1374	1374	1354	1334	1334	1374	1318	1374	1354.40	1374
4I4	1353	1353	1353	1353	1353	1353	1353	1353	1353	1353	1353.00	1353
4I5	1332	1332	1354	1321	1354	1318	1354	1354	1330	1354	1340.30	1354
5I1	2680	2690	2690	2680	2690	2689	2680	2680	2690	2680	2684.90	2690
5I2	2700	2700	2700	2699	2700	2700	2700	2698	2700	2700	2699.70	2700
5I3	2690	2690	2690	2690	2690	2690	2690	2690	2690	2690	2690.00	2690
5I4	2700	2700	2700	2700	2700	2700	2690	2700	2690	2699	2697.90	2700
5I5	2669	2670	2670	2670	2670	2660	2680	2670	2680	2670	2670.90	2680
6I1	2840	2850	2850	2850	2850	2850	2850	2840	2840	2830	2845.00	2850
6I2	2810	2820	2820	2820	2820	2818	2820	2820	2810	2820	2817.80	2820
6I3	2810	2820	2820	2820	2820	2820	2819	2820	2810	2820	2817.90	2820
6I4	2820	2820	2820	2820	2819	2810	2820	2820	2820	2810	2817.90	2820
6I5	2820	2820	2820	2820	2820	2820	2820	2820	2829	2820	2820.90	2829
7I1	2769	2780	2770	2780	2769	2760	2770	2760	2768	2780	2770.60	2780
7I2	2764	2770	2769	2760	2760	2770	2770	2770	2760	2760	2765.30	2770
7I3	2740	2750	2750	2750	2750	2750	2760	2750	2758	2750	2750.80	2760
7I4	2780	2790	2780	2770	2770	2780	2790	2787	2780	2780	2780.70	2790
7I5	2760	2760	2760	2750	2760	2754	2750	2760	2750	2760	2756.40	2760
8I1	2720	2720	2700	2710	2710	2720	2709	2710	2720	2710	2712.90	2720
8I2	2709	2720	2710	2710	2720	2710	2720	2710	2710	2710	2712.90	2720
8I3	2720	2730	2730	2730	2730	2720	2729	2730	2730	2719	2726.80	2730
8I4	2719	2710	2700	2710	2707	2700	2710	2700	2700	2700	2705.60	2719
8I5	2690	2710	2710	2700	2700	2700	2710	2697	2695	2700	2701.20	2710
9I1	2659	2660	2660	2660	2660	2670	2670	2660	2677	2659	2663.50	2677
9I2	2657	2660	2650	2660	2660	2659	2660	2660	2656	2660	2658.20	2660
9I3	2660	2650	2660	2660	2660	2660	2650	2650	2660	2650	2656.00	2660
9I4	2650	2655	2650	2649	2650	2659	2650	2650	2640	2659	2651.20	2659
9I5	2650	2660	2660	2650	2650	2659	2650	2649	2650	2650	2652.80	2660
10I1	2610	2610	2610	2610	2600	2600	2600	2610	2600	2610	2606.00	2610
10I2	2630	2630	2620	2610	2610	2620	2620	2620	2610	2610	2618.00	2630
10I3	2620	2620	2610	2610	2614	2609	2600	2610	2620	2610	2612.30	2620
10I4	2610	2610	2610	2610	2600	2610	2610	2600	2600	2610	2607.00	2610
10I5	2617	2630	2630	2619	2610	2615	2610	2610	2630	2600	2617.10	2630
Average	2391.48	2390.66	2390.80	2392.92	2391.52	2392.20	2391.80	2392.30	2392.14	2390.92	2391.67	2399.26

Table 5.7: The quality of the solutions reached by the ten trials of GNS for $\alpha = 10$ and t_1.

Inst.	1	2	3	4	5	6	7	8	9	10	AV. Sol.	Max.
1I1	2567	2567	2567	2563	2567	2567	2567	2567	2567	2567	2566.60	2567
1I2	2594	2594	2594	2594	2594	2594	2594	2594	2594	2594	2594.00	2594
1I3	2320	2320	2320	2320	2320	2320	2320	2320	2320	2320	2320.00	2320
1I4	2310	2306	2310	2300	2310	2310	2310	2306	2310	2309	2308.10	2310
1I5	2330	2330	2330	2330	2330	2330	2330	2330	2330	2330	2330.00	2330
2I1	2115	2117	2115	2114	2117	2117	2115	2115	2110	2117	2115.20	2117
2I2	2110	2110	2110	2110	2110	2110	2110	2110	2110	2110	2110.00	2110
2I3	2119	2132	2108	2110	2115	2111	2115	2110	2122	2132	2117.40	2132
2I4	2100	2100	2100	2100	2100	2107	2100	2109	2100	2100	2101.60	2109
2I5	2110	2110	2110	2107	2109	2110	2110	2110	2114	2106	2109.60	2114
3I1	1845	1778	1845	1799	1845	1743	1845	1845	1799	1845	1818.90	1845
3I2	1779	1779	1779	1795	1779	1795	1795	1795	1795	1757	1784.80	1795
3I3	1750	1735	1774	1774	1774	1774	1755	1755	1774	1774	1763.90	1774
3I4	1734	1792	1689	1792	1792	1792	1792	1753	1757	1792	1768.50	1792
3I5	1748	1775	1748	1775	1794	1748	1794	1777	1775	1794	1772.80	1794
4I1	1330	1330	1330	1330	1330	1330	1330	1330	1330	1330	1330.00	1330
4I2	1378	1378	1378	1378	1378	1378	1378	1378	1378	1378	1378.00	1378
4I3	1374	1374	1374	1374	1374	1374	1334	1374	1374	1374	1370.00	1374
4I4	1353	1353	1353	1353	1353	1353	1353	1353	1353	1353	1353.00	1353
4I5	1321	1354	1354	1354	1332	1321	1330	1354	1332	1330	1338.20	1354
5I1	2689	2687	2680	2680	2689	2689	2680	2690	2680	2680	2684.40	2690
5I2	2690	2700	2690	2700	2690	2700	2690	2698	2700	2700	2695.80	2700
5I3	2690	2690	2690	2690	2690	2690	2690	2690	2690	2690	2690.00	2690
5I4	2690	2700	2700	2690	2700	2700	2690	2690	2690	2700	2695.00	2700
5I5	2680	2660	2670	2660	2677	2660	2670	2670	2670	2660	2667.70	2680
6I1	2850	2850	2850	2850	2840	2850	2850	2839	2850	2850	2847.90	2850
6I2	2820	2820	2810	2820	2820	2817	2810	2820	2830	2820	2818.70	2830
6I3	2820	2820	2820	2820	2820	2820	2820	2820	2820	2820	2820.00	2820
6I4	2829	2820	2820	2819	2820	2820	2820	2820	2820	2820	2820.80	2829
6I5	2820	2820	2820	2830	2820	2827	2820	2820	2820	2830	2822.70	2830
7I1	2770	2770	2770	2760	2770	2760	2770	2760	2750	2750	2763.00	2770
7I2	2760	2770	2760	2770	2770	2760	2760	2770	2750	2770	2764.00	2770
7I3	2750	2750	2750	2760	2750	2760	2750	2750	2760	2750	2753.00	2760
7I4	2780	2790	2788	2780	2780	2780	2790	2780	2780	2780	2782.80	2790
7I5	2750	2750	2750	2750	2770	2753	2760	2750	2750	2750	2753.30	2770
8I1	2710	2720	2709	2709	2710	2710	2710	2720	2710	2720	2712.80	2720
8I2	2710	2719	2720	2720	2720	2710	2720	2710	2720	2710	2715.90	2720
8I3	2720	2730	2730	2720	2730	2720	2720	2729	2740	2740	2727.90	2740
8I4	2700	2710	2710	2709	2710	2700	2710	2710	2710	2700	2706.90	2710
8I5	2700	2700	2710	2702	2700	2710	2709	2700	2701	2710	2704.20	2710
9I1	2660	2669	2670	2660	2660	2660	2670	2660	2669	2660	2663.80	2670
9I2	2650	2659	2660	2660	2650	2650	2660	2660	2660	2660	2656.90	2660
9I3	2650	2660	2660	2660	2660	2660	2649	2649	2660	2660	2656.80	2660
9I4	2660	2660	2659	2650	2650	2660	2659	2650	2650	2650	2654.80	2660
9I5	2660	2660	2650	2660	2660	2660	2660	2650	2660	2650	2657.00	2660
10I1	2614	2610	2610	2610	2610	2610	2617	2610	2610	2610	2611.10	2617
10I2	2617	2620	2619	2610	2620	2620	2620	2620	2610	2610	2616.60	2620
10I3	2610	2609	2609	2610	2610	2610	2600	2610	2608	2610	2608.60	2610
10I4	2620	2610	2610	2610	2610	2609	2608	2608	2620	2600	2610.50	2620
10I5	2610	2620	2620	2617	2620	2620	2610	2620	2610	2610	2615.70	2620
Average	2391.32	2393.74	2392.04	2393.16	2394.98	2391.58	2393.38	2393.16	2392.84	2393.64	2392.984	2399.36

Table 5.8: The quality of the solutions reached by the ten trials of GNS for $\alpha = 9$ and t_1.

Inst.	1	2	3	4	5	6	7	8	9	10	AV. Sol.	Max
1I1	2567	2567	2567	2567	2567	2567	2567	2567	2567	2567	2567.00	2567
1I2	2594	2594	2594	2594	2594	2594	2594	2594	2594	2594	2594.00	2594
1I3	2320	2320	2320	2320	2320	2320	2320	2320	2320	2320	2320.00	2320
1I4	2310	2310	2310	2310	2310	2310	2306	2310	2310	2310	2309.60	2310
1I5	2330	2330	2330	2330	2330	2330	2330	2330	2330	2330	2330.00	2330
2I1	2115	2117	2110	2117	2114	2115	2117	2115	2118	2117	2115.50	2118
2I2	2110	2110	2110	2110	2110	2110	2110	2110	2110	2110	2110.00	2110
2I3	2115	2111	2111	2110	2111	2112	2108	2132	2118	2110	2113.80	2132
2I4	2100	2109	2109	2100	2100	2100	2100	2109	2105	2107	2103.90	2109
2I5	2110	2110	2110	2110	2110	2108	2108	2109	2108	2110	2109.30	2110
3I1	1743	1720	1714	1845	1845	1845	1778	1845	1845	1845	1802.50	1845
3I2	1779	1757	1795	1779	1779	1779	1795	1795	1757	1757	1777.20	1795
3I3	1774	1718	1712	1774	1774	1774	1774	1711	1774	1774	1755.90	1774
3I4	1792	1753	1792	1753	1792	1757	1792	1753	1792	1753	1772.90	1792
3I5	1775	1775	1748	1794	1721	1794	1775	1775	1794	1794	1774.50	1794
4I1	1330	1330	1330	1330	1330	1330	1330	1330	1330	1330	1330.00	1330
4I2	1378	1378	1378	1378	1378	1378	1378	1378	1378	1378	1378.00	1378
4I3	1374	1334	1374	1374	1354	1334	1334	1374	1327	1374	1355.30	1374
4I4	1353	1353	1353	1353	1353	1353	1353	1353	1353	1353	1353.00	1353
4I5	1354	1354	1354	1330	1354	1332	1354	1354	1354	1354	1349.40	1354
5I1	2680	2690	2690	2680	2690	2689	2680	2680	2690	2680	2684.90	2690
5I2	2700	2700	2700	2699	2700	2700	2700	2698	2700	2700	2699.70	2700
5I3	2690	2690	2690	2690	2690	2690	2690	2690	2690	2690	2690.00	2690
5I4	2700	2700	2700	2700	2700	2700	2690	2700	2690	2699	2697.90	2700
5I5	2669	2670	2670	2670	2670	2660	2680	2670	2680	2670	2670.90	2680
6I1	2850	2850	2850	2850	2850	2850	2850	2840	2840	2850	2848.00	2850
6I2	2830	2820	2820	2829	2820	2820	2820	2830	2820	2820	2822.90	2830
6I3	2820	2820	2820	2820	2820	2830	2820	2820	2820	2820	2821.00	2830
6I4	2820	2820	2828	2820	2819	2820	2820	2820	2820	2820	2820.70	2828
6I5	2820	2820	2820	2820	2820	2820	2820	2830	2829	2820	2821.90	2830
7I1	2780	2780	2770	2780	2770	2760	2770	2768	2768	2780	2772.60	2780
7I2	2770	2770	2770	2760	2770	2770	2770	2770	2760	2770	2768.00	2770
7I3	2750	2750	2750	2750	2750	2760	2760	2750	2760	2750	2753.00	2760
7I4	2780	2790	2790	2770	2790	2790	2790	2789	2780	2780	2784.90	2790
7I5	2760	2760	2760	2750	2760	2754	2750	2760	2750	2760	2756.40	2760
8I1	2720	2720	2715	2720	2710	2720	2710	2710	2720	2710	2715.50	2720
8I2	2710	2720	2710	2710	2720	2710	2720	2719	2710	2710	2713.90	2720
8I3	2740	2740	2740	2730	2730	2720	2729	2730	2730	2720	2730.90	2740
8I4	2719	2710	2708	2710	2707	2700	2710	2700	2700	2700	2706.40	2719
8I5	2710	2710	2710	2700	2710	2700	2710	2700	2700	2700	2705.00	2710
9I1	2660	2663	2660	2670	2677	2670	2670	2663	2677	2666	2667.60	2677
9I2	2659	2660	2654	2660	2660	2660	2660	2660	2660	2660	2659.30	2660
9I3	2660	2660	2660	2660	2660	2660	2656	2653	2660	2669	2659.80	2669
9I4	2660	2660	2660	2650	2660	2659	2650	2650	2649	2659	2655.70	2660
9I5	2660	2660	2660	2659	2660	2659	2655	2659	2660	2650	2658.20	2660
10I1	2610	2618	2610	2610	2600	2618	2609	2610	2606	2610	2610.10	2618
10I2	2630	2630	2620	2620	2610	2620	2620	2620	2620	2610	2620.00	2630
10I3	2620	2620	2610	2610	2620	2610	2609	2610	2620	2610	2613.90	2620
10I4	2610	2610	2618	2610	2608	2610	2610	2610	2610	2610	2610.60	2618
10I5	2619	2630	2630	2619	2610	2615	2610	2610	2630	2630	2620.30	2630
Average	2394.58	2391.82	2392.28	2394.08	2394.14	2393.72	2393.22	2393.66	2394.66	2394.20	2393.64	2400.56

Table 5.9: The quality of the solutions realized by the ten trials of GNS for $\alpha = 10$ and t_2.

Inst.	1	2	3	4	5	6	7	8	9	10	AV. Sol.	Max
1I1	2567	2567	2567	2567	2567	2567	2567	2567	2567	2567	2567.00	2567
1I2	2594	2594	2594	2594	2594	2594	2594	2594	2594	2594	2594.00	2594
1I3	2320	2320	2320	2320	2320	2320	2320	2320	2320	2320	2320.00	2320
1I4	2310	2310	2310	2310	2310	2310	2310	2309	2300	2310	2308.90	2310
1I5	2330	2330	2330	2330	2330	2330	2330	2330	2330	2330	2330.00	2330
2I1	2118	2118	2118	2118	2118	2118	2118	2118	2118	2118	2118.00	2118
2I2	2110	2110	2110	2110	2110	2110	2110	2110	2110	2110	2110.00	2110
2I3	2119	2132	2132	2132	2119	2122	2132	2132	2122	2132	2127.40	2132
2I4	2109	2109	2109	2109	2109	2109	2109	2109	2109	2109	2109.00	2109
2I5	2114	2110	2110	2114	2114	2110	2110	2110	2114	2110	2111.60	2114
3I1	1845	1845	1845	1845	1845	1799	1845	1845	1845	1845	1840.40	1845
3I2	1795	1795	1779	1758	1795	1779	1795	1771	1795	1779	1784.10	1795
3I3	1774	1774	1774	1774	1774	1774	1774	1774	1774	1774	1774.00	1774
3I4	1792	1792	1792	1792	1792	1792	1792	1792	1792	1792	1792.00	1792
3I5	1775	1777	1794	1777	1794	1775	1794	1794	1775	1775	1783.00	1794
4I1	1330	1330	1330	1330	1330	1330	1330	1330	1330	1330	1330.00	1330
4I2	1378	1378	1378	1378	1378	1378	1378	1378	1378	1378	1378.00	1378
4I3	1374	1374	1334	1374	1374	1374	1374	1374	1374	1374	1370.00	1374
4I4	1353	1353	1353	1353	1353	1353	1353	1353	1353	1353	1353.00	1353
4I5	1332	1354	1332	1354	1332	1330	1332	1354	1354	1354	1342.80	1354
5I1	2680	2687	2680	2680	2689	2690	2680	2680	2690	2680	2683.60	2690
5I2	2690	2690	2698	2690	2700	2700	2690	2700	2690	2690	2693.80	2700
5I3	2690	2690	2690	2690	2690	2690	2690	2690	2690	2690	2690.00	2690
5I4	2700	2700	2700	2700	2700	2700	2700	2700	2700	2700	2700.00	2700
5I5	2670	2660	2670	2670	2660	2670	2660	2660	2660	2660	2664.00	2670
6I1	2850	2850	2850	2850	2850	2850	2850	2850	2850	2850	2850.00	2850
6I2	2829	2820	2820	2820	2820	2829	2820	2820	2820	2820	2821.80	2829
6I3	2820	2830	2820	2830	2829	2820	2829	2829	2820	2830	2825.70	2830
6I4	2820	2829	2820	2820	2820	2829	2820	2820	2829	2820	2822.70	2829
6I5	2820	2820	2820	2820	2829	2830	2829	2820	2820	2820	2822.80	2830
7I1	2760	2770	2770	2770	2770	2770	2760	2770	2770	2770	2768.00	2770
7I2	2770	2770	2770	2770	2770	2770	2770	2770	2770	2770	2770.00	2770
7I3	2760	2760	2750	2760	2760	2760	2759	2760	2760	2760	2758.90	2760
7I4	2790	2780	2800	2790	2790	2780	2780	2790	2790	2790	2788.00	2800
7I5	2760	2760	2759	2759	2750	2760	2760	2759	2760	2760	2758.70	2760
8I1	2710	2720	2710	2710	2720	2720	2710	2710	2720	2720	2715.00	2720
8I2	2720	2719	2720	2720	2720	2719	2719	2714	2710	2710	2717.10	2720
8I3	2728	2730	2719	2730	2728	2730	2730	2730	2740	2740	2730.50	2740
8I4	2720	2710	2710	2710	2710	2710	2710	2710	2710	2710	2711.00	2720
8I5	2700	2700	2700	2710	2700	2710	2710	2700	2710	2710	2705.00	2710
9I1	2670	2670	2670	2670	2670	2670	2670	2662	2660	2667	2667.90	2670
9I2	2660	2660	2660	2660	2660	2660	2660	2660	2660	2666	2660.60	2666
9I3	2670	2670	2660	2660	2670	2660	2660	2660	2669	2660	2663.90	2670
9I4	2658	2660	2658	2668	2660	2660	2660	2658	2658	2660	2660.00	2668
9I5	2660	2660	2660	2660	2660	2650	2660	2669	2670	2650	2659.90	2670
10I1	2610	2620	2620	2610	2610	2610	2610	2610	2620	2610	2613.00	2620
10I2	2620	2620	2620	2620	2620	2620	2620	2620	2630	2630	2622.00	2630
10I3	2619	2610	2620	2610	2610	2606	2620	2610	2610	2610	2612.50	2620
10I4	2618	2610	2610	2610	2610	2610	2607	2610	2610	2610	2610.50	2618
10I5	2620	2630	2630	2630	2630	2630	2620	2620	2630	2630	2627.00	2630
Average	2396.62	2397.54	2395.9	2396.72	2397.26	2395.74	2396.6	2396.5	2397.6	2396.94	2396.74	2400.86

Table 5.10: The quality of the solutions reached by the ten trials of GNS for $\alpha = 9$ and t_2.

Part III

Parallelism and optimization problems

Chapter 6

State of the art of the Parallelism

Contents

Recent technology allows emerging of new generation of computers developed with several CPU cores. With these resources, it has become more interesting than before to design algorithms that can be implemented effectively in a multiprocessor environment. In such environment, we have several processing units cooperate for solving the same problem. This permits a big progress in the solution of problems.

In this chapter, we start with a brief introduction about sequential and parallel programming, in section 6.1. Section 6.2, presents some well-known parallel machine architectures. In section 6.3, some parallel approaches are presented. Section 6.4 presents two well-known libraries used in parallel programming. Finally, this chapter is closed with a conclusion in section 6.5.

6.1 Introduction

The past decade has seen large advances in microprocessor technology, making the processors capable of handling multiple instructions concurrency. The role of concurrency accelerates tremendous the computing processes and this, reflected dramatically in the wide variety of applications of parallel computing. Thus, one option for increasing the performance of a solution procedure is to construct a computing system in which several processors work concurrently for solving a problem. In other words, distribute a problem over several processors cooperate with each other for solving the same problem; that is, the parallel programming (cf., Pacheco [50]; Wittwer [68]). This permits the solution to be found more efficiently.

6.1.1 Sequential programming

Sequential programming style is simple and natural. In such style, a program consists of a sequence of instructions executed one after the other in a sequential manner (cf., Figure 6.1).

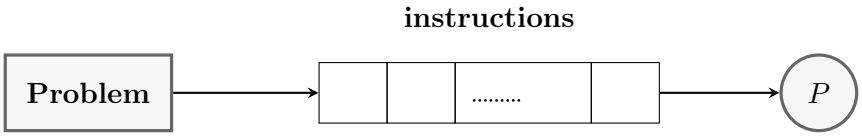


Figure 6.1: Structure of sequential programming

In general, the sequential paradigm has the following characteristics:

- The order of execution is deterministic by the textual order of the statements.
- Successive instructions must be executed one after the other, in the specified order, without any overlap in time.

It is clear that, the performance of sequential programs depends on the order of their statements. Thus, this style of programming is suitable for modeling solution procedures for problems in which the problem regards a sequence of statements.

6.1.2 Parallel programming

Parallel programming style involves the concurrent computation of processes or threads. Unlike sequential programming, where a problem is divided into a set of instruction executed one after the other in a succession fashion by a processor, parallel programming is a mean by which the processing is broken up into parts,

each of which can be executed concurrently on different processors at the same time (cf., Figure 6.2).

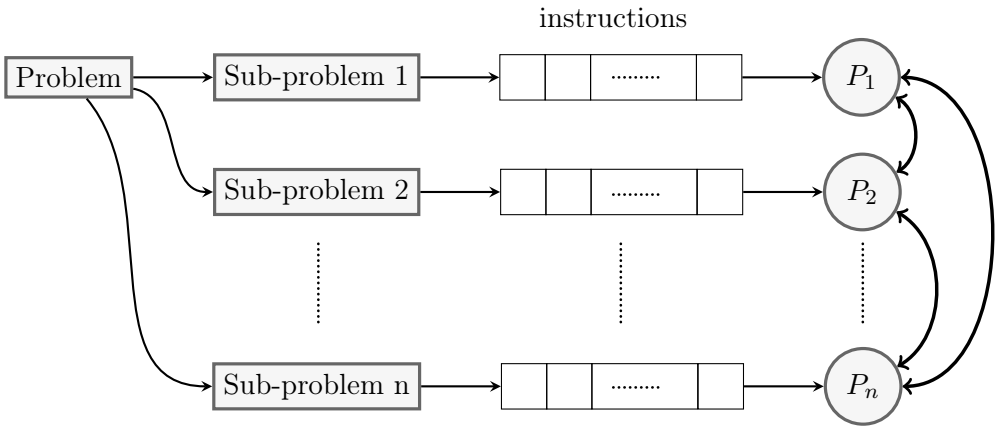


Figure 6.2: Structure of parallel programming

Certainly, parallel programming model imposes a multiprocessor environment. In this environment, we have multi processors located on a single machine, or in a set of distributed machines connected via a network. Fortunately, these days, multi-core personal computers are available makes them multiprocessor systems (cf., Rauber *et al.* [59]). This cheap computing power can be efficiently exploited in parallel programming and requires special adapted parallel software. Figure 6.2 introduces the strategy of parallel programming to solve a problem. First, it divides a problem into sub-problems. Second, solve them in parallel. Third and finally, combine the results in order to reach the final result.

6.2 Parallel machine architectures

A multiprocessor system (or parallel computer) is a collection of several interconnected nodes that cooperate and communicate with each other in order to solve complex problems by splitting them into parallel tasks. Each node corresponds to one or more processing element (or processor), memory system and communication assists. The computation of each task done on one processor.

There are a variety of ways that have been presented to classify parallel machines. Flynn (cf., Flynn [24]) classified parallel computers according to their operating structure into four main categories:

- One Instruction Single Data machine (SISD): a single processor executes one instruction operating on a single data.
- Single Instruction Multiple Data machine (SIMD): the processors, simultaneously execute the same instruction to multiple data.

- Multiple Instruction Single Data machine (MISD): several instructions, simultaneously executed on the same data.

- Multiple Instruction Multiple Data machine (MIMD): this structure allows different simultaneous operations on different data.

The multiple instruction multiple data machines are currently the most common than others and can be broadly divided according to the organization of the memory into three sub-classes: shared memory machines, distributed memory machines and mixed memory machines (cf., Rajaraman *et al.* [58]).

6.2.1 Shared memory architectures

Shared memory machines, as their name, suggest sharing a single memory that can be accessed by any of the processors. Specifically, the memory is addressed by a single range of addresses, such that each memory location is defined with a unique address. All processors can simultaneously access the same memory. Meanwhile, all performed operations on that memory is immediately available to all other processors (cf., Figure 6.3).

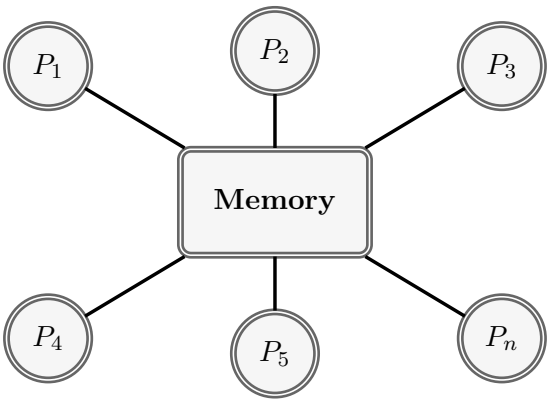


Figure 6.3: Shared memory machines

Shared memory is an efficient means of communications for passing data between processors. Generally, parallel programming with share memory is easier and more convenient than with the other types of memory architectures, since all processors can simultaneously access the same memory. However, in practice, this class is best suited to machines with a limited number of processors, because increasing the number of processors, may constitute a bottleneck with the access to the shared memory.

6.2.2 Distributed memory architectures

Distributed memory machines refer to a computer system in which each processor has its own memory space. Different processors are connected between them by an interconnected network and they communicate with each other by the exchanging of messages through this network (cf., Figure 6.4).

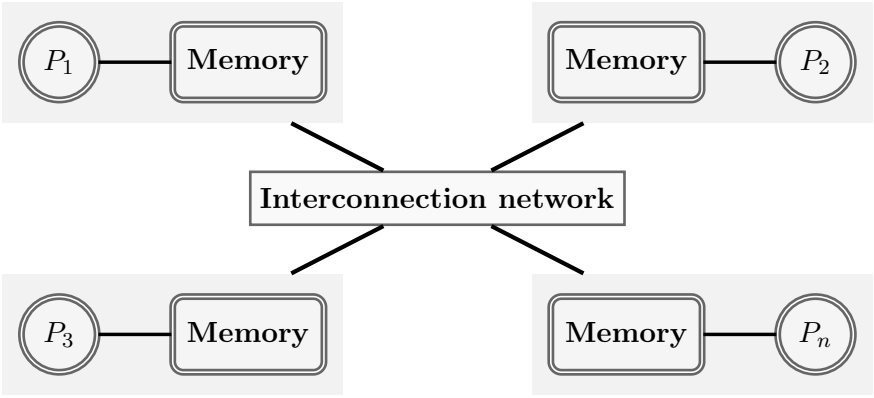


Figure 6.4: Distributed memory machines

Distributed memory machines can integrate a large number of processors (may be more than thousands). Parallel applications using this type of memory architecture can be scalable. The communication model can allow a considerable increase in speed, but its programming is difficult since the programmers have to handle all communication operations.

6.2.3 Mixed (or hybrid) memory architectures

The combination of both shared and distributed memory mechanisms (noted as mixed or hybrid memory architectures) provides a flexible means to adapt to various computing platforms (cf., Figure 6.5). This combination may increase scalability, increase performance computing, speed up computation, and permit to efficient utilization of the existing hardware capacities. However, this type of architecture combines advantages, but also may combine the disadvantages of the both architectures.

6.3 Problems and parallel approaches

Parallel approaches depend strongly on the nature of the problem being addressed. There is no standard parallel approach for an application. It is natural to model a suitable algorithm for an application according to its characteristics. However, with regard to the parallelization, it should identify the tasks that may be executed in parallel. It should be then, choose the order of their execution. It could be better to draw the precedence graph which is a directed graph of dependence between the tasks (cf., Saadi [61]).

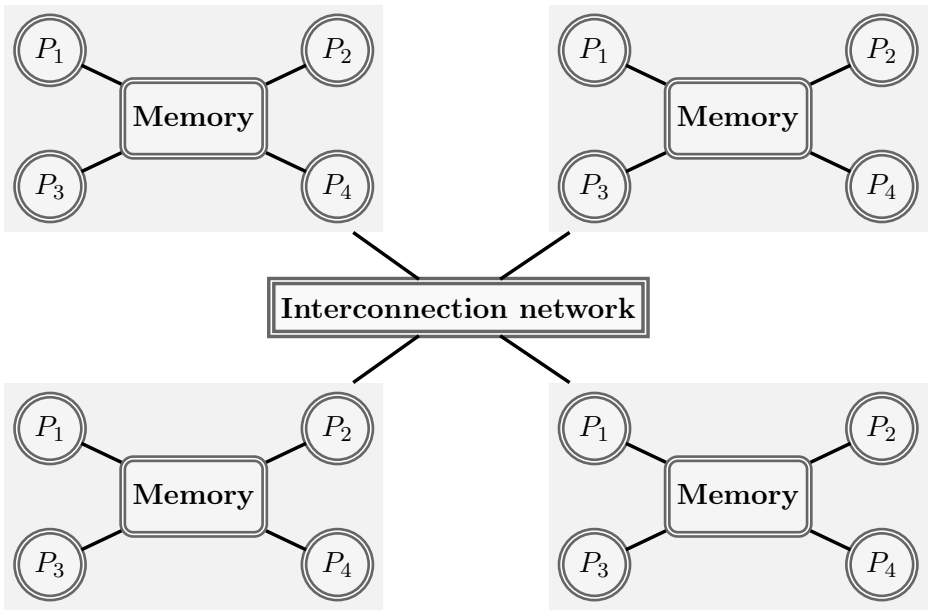


Figure 6.5: Mixed (hybrid) memory machines

6.3.1 Data parallelism

This approach consists of executing the same operation in parallel on a set of data. Each processor performs its task. In this parallelism paradigm, a parallel program can be viewed as a two phase: (1) calculation phase, and (2) communication phase. Calculation phase involves the necessary computation task while the communication phase permits to update the data and improves the overall operation.

6.3.2 Functional parallelism

The functional parallelism consists of dividing an application into several tasks, then executed them in parallel. In this parallelism paradigm, a parallel program can be illustrated as a directed graph of tasks. In such program, the sequence of execution is important, such that a task cannot be executed unless all its precedence tasks in the graph have already been executed.

6.3.3 Irregular parallel application

We can classify the parallel applications according to their behaviour into (i) regular and (ii) irregular parallel applications. A parallel application can be seen as regular, if we can predict in advance the tasks' graph and their duration. In the other hand, a parallel application can be seen as irregular, if we cannot predict in advance the behaviour of the problem instance being addressed. However, there are several reasons may cause this irregularity:

- The execution time of a task is unknown and differs from one task to another.
- We do not precisely know the dependencies between tasks. So we can not previously construct the task graph.
- The handled data are unknown in advance or its size varies greatly during the communication.

The parallelization of an irregular application facing difficulties in predicting, at least partially, the tasks' graph and the estimation cost of performing a task with regards to its input.

6.3.4 Scheduling and arranging

Task scheduling involves assigning a kind of arrangement to the tasks. This can be performed by attributing a starting time for the execution of each task. Scheduling, in general, depends on: (i) the number of processors used and (ii) the tasks precedence. For the regular applications, it is possible to schedule the tasks, because the precedence graph is predictable in advance. For the irregular applications, the precedence graph cannot be predicted in advance, although it can be known during the execution. In such cases, the scheduling is more difficult to implement.

6.4 Existing libraries

There are various software tools available to develop parallel applications, many of them are open-source libraries. Among these tools, we can find: (i) Open MultiProcessing (namely OpenMP) for shared memory architectures and (ii) the Message Passing Interface (namly MPI) library for distributed memory architectures (cf., Dagum *et al.* [15], Barney [8]). Both tools are available for all modern computers and programs can be written in C, C++, and FORTRAN.

6.4.1 Open Multi-Processing

Open Multi-Procesing (namely OpenMP) is an application programming interface that supports shard memory multiprocessing programming. It consists of a set of compiler directives, library routines, and environment variables for developing parallel software applications for different platforms ranging from desktop personal computers to supercomputers. (cf., Chapman *et al.* [12]).

6.4.2 Message Passing Interface

Message Passing Interface (noted as MPI) is a library interface specification (cf., Lastovetsky [42]). MPI addresses primarily the message-passing parallel programming model, in which data is moved from the address space of one processor

to that of another processor through cooperative operations on each processor. The main advantage of MPI is the ease of use, in addition, it enables us to build a portable, efficient, and flexible message passing model of parallel programming (cf., Langtangen *et al.* [41]; Message passing interface forum [25]).

6.5 Conclusion

This chapter presented some basic definitions and notions of the parallelism. We introduced two styles of programming: sequential and parallel. A sequential style represents a problem as a set of instruction executed one after the other in a sequential manner. Meanwhile, a parallel programming style divides a problem into several parts, each of them can be solved concurrently on different processors. This permits great progress to be made in problem solving.

The following chapter presents a parallel algorithm tailored for solving large scale instances of the DCKP. This algorithm exploits the parallelism for the aim of exploring simultaneously different neighborhoods.

Chapter 7

A parallel large neighborhood search-based heuristic for the disjunctively constrained knapsack problem

Contents

This chapter proposes a parallel large neighborhood search-based heuristic for solving the disjunctively constrained knapsack problem, which has an important impact on the transportation issues. The proposed approach is designed using Message Passing Interface (MPI). The effectiveness of MPI's allows us to build a flexible message passing model of parallel programming. Meanwhile, large neighborhood search heuristic is introduced in the model in order to propose an efficient resolution method yielding high quality solutions. The results provided by the proposed method are compared to those reached by the Cplex solver and to those obtained by one of the best methods of the literature. As shown from the experimental results, the proposed model is able to provide high quality solutions with fast runtime on most cases of the benchmark literature.

7.1 Introduction

Transport problems impact on many institutions from small companies to large multinational organization. It refers to the process of transport of peoples, products, or any items from one location to another. However, there are many applications regarding transport processes, among of them, one is regarded with the movement of disabled or disabilities, seniors, and other citizens with unique needs. Search the efficient models and algorithms for finding the high quality solutions of these problems attract the attention of the public. This chapter addresses an issue of transport problems, namely the disabled persons transport problems.

Since the disabled people have their own issues, in most of the cases they need to be served with respecting of some special conditions. As a consequence, someone of them cannot take the same bus as someone else. In the other words, a bus cannot serve those incompatible persons at the same time. Formally, the last problem can be simulated as a combinatorial optimization problem, namely the disjunctively constraint knapsack problem (DCKP), which is a variant of the well known NP-hard problem: knapsack problem. Furthermore, the disjunctive constraints appear in many other issues of transport, such as the bin-packing problem with conflicts (cf., Sadykov and Vanderbeck [62]).

In this chapter, we present an approach for solving the Disjunctively Constrained Knapsack Problem (DCKP). The DCKP is a variation of the classical knapsack problem. To our knowledge, all algorithms in the literature for solving DCKP are based on sequential procedure where the algorithm executed sequentially. As opposed to a traditional sequential algorithm, parallel algorithm can be executed a piece at a time, and then combined together again at the end to get the final result. In this chapter we investigate the use of large neighborhood search heuristic in parallel modeling algorithm for solving the DCKP.

The remainder of this chapter is organized as follows. Section 7.2 reviews the principles of the large neighborhood search. Section 7.3 describes the principle of the proposed approach. In Section 7.4, the performance of the proposed approach is evaluates on a number of instances of the literature, and analyzes the obtained results. Finally, Section 7.5 summarizes the contents of the chapter.

7.2 Large neighborhood search

Solving combinatorial optimization problems using exact methods becomes discouraged when the structure of problems is complex. Indeed, proving optimality of problems requires generally a huge computational resource (cf., Papadimitriou and Steiglitz [51]). Contrast to the fact that the exact methods aim at solving problems by proving their optimality, the approximative methods focus only on the computation of high quality solutions (near optimal) with reasonable computational effort. Large Neighborhood Search (LNS), proposed by Shaw [65], has

been considered as one of the most efficient algorithms for solving large-scale optimization problems (cf., Ahuja and Ergun [2]). Unlike the exact methods explore generally the whole solution space, LNS consists of finding high quality solutions by randomly exploring a series of sub-solution spaces, where each subspace is characterized by a neighborhood of a local optimum.

Generic schema of LNS consists of two strategies: *removing strategy* and *exploring strategy*. More precisely, the removing strategy is used to yield a sub-solution space (i.e., the neighborhood of the local optimum) while, the exploring strategy is then applied in order to improve the current local optimum in its neighborhood.

Input: An instance of P_{DCKP}.
Output: $S^{\star}$, a local optimum of P_{DCKP}.

1: Set $S^{\star}$ as a starting feasible solution, where all variables are assigned to 0;
2: **while** the iteration limit is not performed **do**
3: Apply a greedy procedure to compute a feasible solution of P_{DCKP};
4: Apply a removing procedure to yield an reduced instance of P_{DCKP} and a partial solution;
5: Apply the greedy procedure on the reduced instance in order to complete the partial solution;
6: Apply a local search procedure to improve the current solution;
7: Update $S^{\star}$ with the best solution at hand;
8: **end while**
9: **return** $S^{\star}$;

Algorithm 18: A large neighborhood search-based heuristic (LNSH)

Algorithm 18 summarizes the main steps of a generic schema of a large neighborhood search heuristic (noted as LNSH). Consider this

The main loop (step 2-8) of LNSH applies alternatively the removing strategy (step 4) and the exploring strategy (step 5) in order to improve the current local optimum. The removing procedure (step 4) servers to remove randomly a limited number of items, whose decision variables values have been determined, in order to yield a reduce instance of DCKP and a partial solution (i.e., with some decision variables have been set free). The exploring procedure (step 5) aims at improving the current local optimum by exploring the solution space yielded from step 4. LNSH (cf., Algorithm 18) exits with the best solution found so far when the iteration limit is reached.

7.3 Parallel large neighborhood search

Parallel programming is a form of computation in which numbers of calculations are carried out simultaneously. Operating on the principle that large problems can often be divided into sub-problems, which are then can be solved concurrently. Starting from this point of view, we propose a a parallel large neighborhood search algorithm for solving DCKP[1]. The proposed model is designed using Message Passing Interface (MPI) (cf., Snir *et al* [66]; Message passing interface forum [25]).

MPI is a message-passing library interface specification. MPI addresses primarily the message-passing parallel programming model, in which data is moved from the address space of one process to that of another process through cooperative operations on each process. The main advantages of MPI is the ease of use, in addition, it enables us to build a portable, efficient, and flexible message passing model of parallel programming.

7.3.1 MPI Communication and topology

The basic concept that stand behind MPI's design of parallel programming is the notion of a communicator (cf., Figure 7.1). A communicator is an ordered set of processes. Each process is associated with a unique integer rank. Rank values start at zero and go to $N-1$, where N is the number of processes in the communicator.

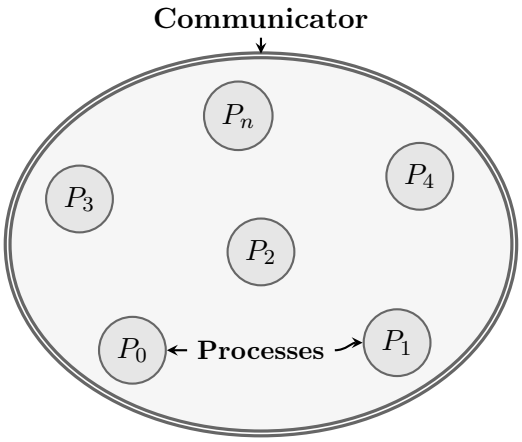


Figure 7.1: A communicator

A communicator encompasses a group of processes that may communicate

[1]This work is published in an international conference with proceedings *28th IEEE International Parallel & Distributed Processing Symposium* (cf., Hifi *et al* [32])

with each other by their ranks using point-to-point or collective communications operations. It provides a safe communications basis on a virtual topologies. A virtual topology means that, there may be no relation between the physical structure of the process topology. It describes a mapping/ordering of processes into a geometric shape. The two main types of topologies supported by MPI are Cartesian (grid) and Graph. The mapping of processes into an MPI virtual topology is dependent upon the algorithm implementation and the problem to solve.

In addition, the principle of the communication process is based on sending and receiving messages. A process may send a message to another process by providing the rank of the process and a unique tag to identify the message. The receiver can then post a receive for a message with a given tag (or it may not even care about the tag), and then handle the data accordingly. Communications which involve one sender and receiver are known as point-to-point communications. We note, MPI can handle a wide variety of collective communications where processes can communicate with each other.

7.3.1.1 Point to point communication

Point-to-point communication involves message passing between two, and only two, processes, one is called a sender process and the other is called a receiver process (cf., Figure 7.2).

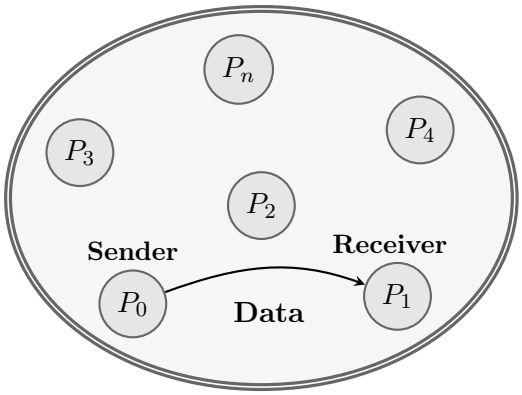


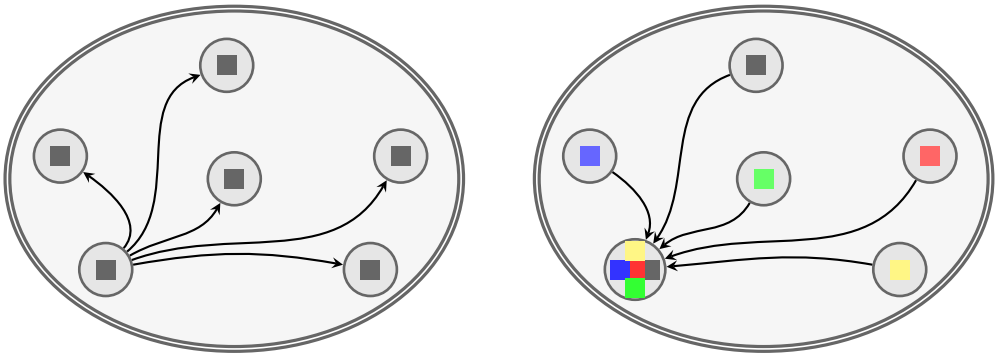
Figure 7.2: Point to point communication

The sender sends a message while the receiver receives this message (for more details the reader can be refereed to Snir *et al* [66], Message passing interface forum [25]).

7.3.1.2 Collective communication

Collective communication concerns all the processes within the scope of a communicator. It allows to make a series of point-to-point communications in a single call (cf., Figure 7.3). There are several types of collective communication, including:

- Broadcast: global distribution of data (i.e., take data from one process and send it to all other processes).
- Gather: collect distributed data (i.e., collect data from all processes and send it to one process).



(a) *Broadcast comunication* (a) *Gather comunication*

Figure 7.3: Collective communication

7.3.2 Parallel large neighborhood search Implementation

In this section, we describe the cooperation mechanism used by the proposed parallel algorithm (cf., Figure 7.4). Indeed, as previously indicated, the core of LNS is based on two principal strategies: *removing strategy* and *exploring strategy*. Further, an important element of the conception of a large neighborhood search is the structure of neighborhoods generated, like the size of the neighborhood, the local optimum from which the neighborhood has been generated, etc. The main concept of the proposed parallel algorithm consists of exploring simultaneously different neighborhoods (in parallel) by a number of processors, where each processor adopts its own random strategy to yield neighborhoods (cf., Figure 7.4 (a)). These processors share then the best solutions obtained in order to yield better neighborhoods through the large neighborhood search (cf., Figure 7.4 (b, and c)).

Algorithm 19 describes the main steps of the proposed parallel large neighborhood search-based heuristic (noted as PLNSH) for solving the DCKP.

Initially, in Step 1 the communicator is initialized, where n processors are activated and the processor 0 is considered as the root. Step 2 serves to find

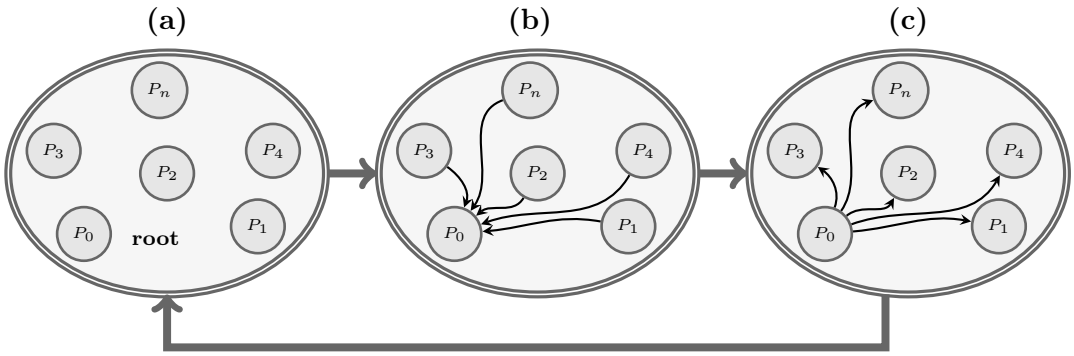


Figure 7.4: The structure of the proposed algorithm

Input: An instance of P_{DCKP}.
Output: $S^\star$, a local optimum of P_{DCKP}.

1: Call n processors and set processor 0 as the root;
2: On each processor i ($i = 1, \ldots, n-1$), apply LNSH to compute the local optimum and send to the root;
3: **while** the runtime limit is not performed **do**
4: From the solutions received, the root choses the best one and send to the processors i ($i = 1, \ldots, n-1$);
5: On each processor i ($i = 1, \ldots, n-1$), apply LNSH to improve the current received local optimum and send the best solution obtained to the root;
6: Update $S^\star$ with the best solution at hand;
7: **end while**
8: **return** $S^\star$;

Algorithm 19: A parallel large neighborhood search-based heuristic (PLNSH)

a starting solution by running LNSH (cf., Algorithm 18) on agents (i.e., the processors from 1 to $n-1$). In the main loop (Step 3-7), the $n-1$ agents apply their own random strategy to perform LNSH (i.e., using a random neighborhood size and a random selection of items to be removed). The best solution reached by each agent is then sent to the root. According to all received solutions, the root sends the best solution to each agent. The last received solution is then considered as the starting solution for running LNSH on each agent. Finally, PLNSH exits with the best solution found so far within a pre-fixed runtime limit.

7.4 Computational results

This section evaluates the effectiveness of the proposed *Parallel Large Neighborhood Search-Based Heuristic* (PLNSH) on two groups of benchmark instances generated by Hifi *et al.* [31] following the schema used by Yamada *et al.* [69]). The first group contains twenty medium instances with $n = 500$ items, a capacity $c = 1800$ and, the second group contains thirty large-scale instances, where each instance contains 1000 items, with c taken in the discrete interval $\{1800, 2000\}$. The proposed parallel algorithm was coded in C++ using the MPI library and run on a machine with Intel Xeon, 2×3.06 Ghz with 6-Core.

		LNSH		PLNSH			
	V_{Cplex}	Max	Mean	OV_5	RT_5	OV_{10}	RT_{10}
Inst.	7200	200		200		200	
1I	2418.2	2424.2	2423.3	2424.2	43.4	2424.2	22.7
2I	2083.4	2116.6	2112.9	2107.6	42.7	2107.6	22.6
3I	1650.4	1796.2	1780.4	1792	42.7	1777.8	22.1
4I	1217	1357.8	1354.3	1353.4	42.6	1353.4	22.0
5I	2676	2691.2	2683.1	2683.6	43.5	2680	23.8
6I	2804	2832.4	2829.1	2827.8	45.3	2827.6	24.3
7I	2735.4	2780	2773.4	2774	46.0	2772	27.0
8I	2679	2721.8	2718.6	2714	47.4	2718	27.9
9I	2603.6	2671.4	2666.5	2660	47.1	2661.8	26.7
10I	2541.8	2625.2	2622	2619.6	47.7	2615.8	30.7
Av.	2340.9	2401.7	2396.4	2395.6	44.8	2393.8	25.0

Table 7.1: Behavior of both LNSH and PLNSH when compared to the Cplex solver.

Table 7.1 reports the average objective values (i.e., on each class of the benchmark instances) related to the best solutions when applying the Cplex solver (version 12.5), LNSH (the sequential version of PLNSH) and PLNSH respectively. Column 1 displays the name of each class of the benchmark instances. The average objective values of the best solutions provided by Cplex within 7200 seconds are shown in column 2. As LNSH applies a random strategy to remove items, ten independent trials of LNSH were performed for each instance with a runtime limit fixed to 200 seconds. According to the numerical results, the number of items to be removed is experimentally limited to $10\% \times n$, where n is the total number of items of each instance. Column 3 and column 4 tally respectively the average objective values of the best solutions and the average solutions of the ten trials. First, in order to evaluate the performance of PLNSH, the total CPU runtime consumed by all active processors is limited to 200 seconds (i.e., the same value used by LNSH) whereas the maximum number of removed items performed by LNSH on each agent is randomly generated in discrete interval $[5\% \times n, 15\% \times n]$. Column 5 and column 6 shows respectively OV_5, the average objective values and RT_5, the real runtimes realized by PLNSH using 5 processors. Column 7 and column 8 shows respectively OV_{10}, the average objective values and RT_{10},

the real runtimes realized by PLNSH using 10 processors.

As shown in Table 7.1, the results provided by both LNSH and PLNSH outperform generally those provided by the Cplex solver. On the one hand, the best solutions of the ten independent trails are generally better than those realized by one trail of PLNSH. On the other hand, PLNSH matches the average values reached by LNSH. Moreover, in order to highlight the effectiveness of the proposed approach, we decided to compare the results provided by PLNSH to the best solutions of the literature (cf., Table 7.2). In this case, the number of active processors is fixed to 10 and the total CPU runtime consumed by all active processors is extended to 2000 seconds, where the corresponding real runtime is closed to 200 seconds.

Table 7.2 shows objective values reached by LNSH, PLNSH, and Cplex when compared to the best solutions of the literature (IRS proposed by Hifi [29]). Column 1 displays the instance lable, column 2 reports the values of the best solutions (denoted V_{cplex}) reached by Cplex solver within 7200s and column 3 displays the solutions provided by the the best solutions of the literature: IRS proposed by Hifi [29] within 1000s, denoted V_{IRS}. Column 4 (resp. 5) reports Max (resp. Mean) denoting the maximum (resp. average) solution value obtained by LNSH over ten trails with a runtime limit fixed to 200 seconds. Finally, column 6 (resp. 7) shows OV_{10} (resp. RT_{10}) average objective value (resp. average real runtime) realized by by PLNSH using 10 processors.

From Table 7.2, we can observe that:

- Both LNSH and PLNSH outperform the Cplex solver (in 50 cases) and one of the best methods of the literature: IRS proposed by Hifi [29] (in 48 cases). Indeed, Cplex solver (resp. IRS) realizes an average value of 2340.9 within 7200s (resp. 2390.4 within 1000s) compare to that realized by LNSH and PLNSH.

- By extending the runtime limit, the quality of the solutions provided by one trail PLNSH (an average value of 2400.0) is quite close to the best solution realized by the ten independent trails (an average solution value of 2401.7).

- The average solution value realized by one trail when running PLNSH (an average value of 2400.0) is better than the average solution value of all the ten independent trials of LNSH (it realizes an average solution value of 2396.4) and quite close to the best solution (an average solution value of 2401.7). One can observe that, the behavior of PLNSH is interesting.

7.5 Conclusion

In this chapter, we proposed a parallel algorithm based upon large neighborhood search for solving the disjunctively constrained knapsack problem. The proposed

			LNSH		PLNSH	
	V_{Cplex}	V_{IRS}	Max	Mean	OV_{10}	RT_{10}
Inst.	7200	1000	200			
1I1	2567	2567	2567	2564.6	2567*	202.82
1I2	2594	2594	2594	2594	2594*	203.01
1I3	2320	2320	2320	2319	2320*	204.21
1I4	2300	2303	2310	2310	2310*	203.2
1I5	2310	2310	2330	2329	2330*	204.4
2I1	2077	2100	2118*	2117	2110	203.24
2I2	2080	2110	2110	2110	2110*	204.7
2I3	2090	2128	2132	2118.1	2132*	202.47
2I4	2080	2107	2109	2108.2	2109*	203.28
2I5	2090	2103	2114	2111.2	2109	203.53
3I1	1667	1840	1845	1814	1845*	202.65
3I2	1681	1785	1795*	1774	1779	204.21
3I3	1588	1742	1774	1762.9	1774*	203.81
3I4	1753	1792	1792	1792	1792*	209.76
3I5	1563	1772	1775	1759.2	1794*	204.08
4I1	1174	1321	1330	1330	1330*	203.8
4I2	1210	1378	1378	1378	1378*	204.1
4I3	1212	1374	1374	1374	1374*	204.38
4I4	1221	1353	1353	1353	1353*	203.61
4I5	1268	1354	1354	1336.4	1354*	203.74
5I1	2680	2690	2690	2686	2690*	205.36
5I2	2690	2690	2698*	2686.7	2690	203.78
5I3	2670	2689	2690*	2679.7	2680	205.49
5I4	2680	2690	2698	2689.2	2700*	204.82
5I5	2660	2680	2680	2673.9	2689*	206.51
6I1	2820	2840	2850	2850	2850*	207.61
6I2	2800	2820	2830	2827.7	2830*	206.32
6I3	2790	2820	2830	2821.9	2830*	203.73
6I4	2800	2800	2822*	2820.2	2820	208.38
6I5	2810	2810	2830	2825.6	2830*	206.68
7I1	2727	2750	2780	2773	2780*	209.19
7I2	2720	2750	2780	2773	2780*	205.13
7I3	2740	2747	2770*	2763	2760	205.82
7I4	2740	2773	2800*	2793	2790	205.04
7I5	2750	2757	2770	2765	2770*	209.49
8I1	2686	2720	2720	2719.1	2720*	213.52
8I2	2690	2709	2720*	2720	2710	209.1
8I3	2690	2730	2740	2734	2740*	210.37
8I4	2670	2710	2719*	2709.9	2710	207.15
8I5	2659	2710	2710	2710	2710*	209.66
9I1	2589	2650	2677*	2671.3	2676	205.65
9I2	2600	2640	2670	2665.6	2670*	214.02
9I3	2620	2635	2670	2668.6	2670*	206.47
9I4	2609	2630	2670*	2661.9	2659	209.42
9I5	2600	2630	2670	2664.9	2670*	212.69
10I1	2550	2610	2624*	2620.5	2610	208.11
10I2	2550	2642*	2630	2629.9	2630	210.12
10I3	2530	2618	2627*	2620.5	2620	208.12
10I4	2549	2621*	2620	2618.6	2620	206.99
10I5	2530	2606	2625	2620.5	2630*	213.93
Av.	2340.9	2390.4	2401.7	2396.4	2400.0	206.4

Table 7.2: Performance of PLNSH on the standard benchmark instances of the literature.

approach was designed using message passing interface. Large neighborhood search-based heuristic was introduced in the parallel model in order to design an efficient resolution method yielding high quality solutions. The performance of the proposed method was evaluated on the set of the standard benchmark instances of the literature. Then, its provided results were compared to those reached by the Cplex solver and to those obtained by one of the best methods of the literature. As shown from the experimental results, the proposed model was able to provide high quality solutions within fast runtime. This work is published in an international conference with proceedings *28th IEEE International Parallel & Distributed Processing Symposium* (cf., Hifi *et al* [32]).

Chapter 8

General conclusion

In this general conclusion, we present a brief summary and outline only the principle contributions of this work, since the detailed discussion of each contribution is presented as a final section of the corresponding chapter. In addition, we draw some perspectives on future work.

At first, in order to draw some conclusions from the work presented in this thesis, it is necessary to draw attention to the primary goal that was considered when this research started. The primary goal was to develop solution approaches based upon neighborhood search techniques tailored for optimizing large size instances of combinatorial optimization problems. Among these problems, we considered a particular problem belonging to the knapsack family: the disjunctively constrained knapsack problem. In order to tackle such a problem, we found two directions of research: (i) solving this problem using exact methods, and/or (ii) searching near optimal solutions using heuristic methods. An exact algorithm tries to find an optimal solution. Indeed, due to the complexity of the considered problem, proving optimality requires a huge computational resource. In contrast to the exact algorithm, heuristic methods focus on the computation of high quality solutions with reasonable computational effort. For that, we considered in this work heuristic methods based upon neighborhood search techniques.

Suitability of neighborhood search techniques

The work realized here highlighted the effectiveness of neighborhood search techniques for developing heuristic methods tailored to optimize large-size instances of combinatorial optimization problems. Probably, neighborhood search methods are well-known mainly for two reasons:

- Their power to develop heuristic methods.
- Their simplicity in understanding and implementation.

The general outcome, according to the experimental analysis on benchmarks instances of the literature, is that neighborhood search solution techniques are successful in developing better algorithms to approximate large-size instances of the considered problem in this work.

Principle contributions

The research reported in the thesis has focused on the development of heuristic approaches to approximate large-size instances of NP-hard combinatorial optimization problems. Herein, we cited several sequential and parallel algorithms for approximating a particular problem belonging to the knapsack family: the disjunctively constrained knapsack problem:

- The first algorithm can be viewed as a random neighborhood search method. The motive was to present a fast algorithm that yields an interesting solution within a short average running time. Accordingly, a fast sequential algorithm (cf., Chapter 4) is presented that used a combination of neighborhood search techniques in order to produce a quick solution with high quality. In this algorithm a number of neighborhood search techniques are used which randomly explore a series of sub-solution spaces, where each subspace is characterized by a neighborhood of a local optimum.

- The second algorithm can be viewed as a guided neighborhood search method. The motive was to present an adaptive algorithm that guides the search process in the feasible solution space towards high quality solutions. Accordingly, an adaptive neighborhood search algorithm (cf., Chapter 5) is presented that used an ant colony optimization system to simulate the guided search.

- The third and last algorithm can be viewed as a parallel random neighborhood search method. The motive was to exploit the parallelism in developing a fast algorithm that yields high quality solutions within a short average running time. Accordingly, a parallel algorithm (cf., Chapter 7) is presented that used a random neighborhood search method in a parallel model. This algorithm consists of exploring simultaneously different sub-solution spaces by several processors. Each processor adopts its own random strategy to yield its own neighborhoods solution according to its internal information. These processors then share the best solutions obtained in order to approach the most promising neighborhood through the search process. The proposed algorithm is created using a message passing interface standard which enabled us to build a flexible message passing parallel programming model.

Perspective and future work

The work presented in this thesis could be adapted to deal with many various combinatorial optimization problems. Among these problems, we can cite: multi-scenario max-min knapsack problem, precedence constrained knapsack problem, k-clustering minimum bi-clique completion problem, and network design problem with relays. We strongly believe that the proposed methods can be beneficial for

developing heuristic procedures for such problems. On the other hand, it is also interesting to consider other methods in order to develop guided neighborhood search algorithms, such as tabu search, simulated annealing, and genetic algorithms. Another suggested work is to consider another paradigm of parallelism, such as a shared memory paradigm with OpenMP.

Bibliography

[1] Emile HL Aarts and Jan Karel Lenstra. *Local search in combinatorial optimization*. Princeton University Press, 2003. (Cited in page 49.)

[2] Ravindra K Ahuja, Özlem Ergun, James B Orlin, and Abraham P Punnen. A survey of very large-scale neighborhood search techniques. *Discrete Applied Mathematics*, 123(1):75–102, 2002. (Cited in pages 49 and 97.)

[3] Hakim Akeb, Mhand Hifi, and Mohamed Elhafedh Ould Ahmed Mounir. Local branching-based algorithms for the disjunctively constrained knapsack problem. *Computers & Industrial Engineering*, 60(4):811–820, 2011. (Cited in pages 17, 18, 40 and 41.)

[4] Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach*. Cambridge University Press, 2009. (Cited in pages 14 and 38.)

[5] Egon Balas, Sebastian Ceria, Gérard Cornuéjols, and N Natraj. Gomory cuts revisited. *Operations Research Letters*, 19(1):1–9, 1996. (Cited in pages 11 and 35.)

[6] Egon Balas and Jue Xue. Weighted and unweighted maximum clique algorithms with upper bounds from fractional coloring. *Algorithmica*, 15(5):397–412, 1996. (Cited in pages 11 and 35.)

[7] Egon Balas and Eitan Zemel. An algorithm for large zero-one knapsack problems. *operations Research*, 28(5):1130–1154, 1980. (Cited in pages 5 and 29.)

[8] Blaise Barney. Openmp. Website: https://computing. llnl. gov/tutorials/openMP. Accessed 15 Avril 2015. (Cited in page 93.)

[9] Richard Bellman. On the theory of dynamic programming. *Proceedings of the National Academy of Sciences of the United States of America*, 38(8):716, 1952. (Cited in pages 12 and 35.)

[10] Richard Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1 edition, 1957. (Cited in pages 12 and 35.)

[11] Christian Blum, Andrea Roli, and Michael Sampels. *Hybrid metaheuristics: an emerging approach to optimization*, volume 114. Springer, 2008. (Cited in pages 17 and 40.)

[12] Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas. *Using OpenMP: portable shared memory parallel programming*, volume 10. MIT press, 2008. (Cited in page 93.)

[13] Alberto Colorni, Marco Dorigo, and Vittorio Maniezzo. Distributed optimization by ant colonies. In *Proceedings of the first European conference on artificial life*, volume 142, pages 134–142. Paris, France, 1991. (Cited in page 67.)

[14] Harlan Crowder, Ellis L Johnson, and Manfred Padberg. Solving large-scale zero-one linear programming problems. *Operations Research*, 31(5):803–834, 1983. (Cited in pages 11 and 35.)

[15] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998. (Cited in page 93.)

[16] George B Dantzig. Discrete-variable extremum problems. *Operations research*, 5(2):266–288, 1957. (Cited in pages 4, 7, 8, 9, 28, 31 and 32.)

[17] Sanjoy Dasgupta, Christos H Papadimitriou, and Umesh Vazirani. *Algorithms*. McGraw-Hill, Inc., 2006. (Cited in pages 7, 11, 31 and 34.)

[18] Marco Dorigo and Christian Blum. Ant colony optimization theory: A survey. *Theoretical computer science*, 344(2):243–278, 2005. (Cited in page 62.)

[19] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. The ant system: An autocatalytic optimizing process. Technical report, 1991. (Cited in pages 62 and 67.)

[20] Krzysztof Dudziński and Stanisław Walukiewicz. Exact methods for the knapsack problem and its generalizations. *European Journal of Operational Research*, 28(1):3–21, 1987. (Cited in pages 4 and 28.)

[21] D Fayard and G Plateau. An algorithm for the solution of the 0–1 knapsack problem. *Computing*, 28(3):269–287, 1982. (Cited in pages 5 and 29.)

[22] Stefka Fidanova. Ant colony optimization for multiple knapsack problem and model bias. In *Numerical Analysis and its Applications*, pages 280–287. Springer, 2005. (Cited in page 67.)

[23] Matteo Fischetti and Andrea Lodi. Local branching. *Mathematical programming*, 98(1-3):23–47, 2003. (Cited in pages 17 and 40.)

[24] Michael Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, 100(9):948–960, 1972. (Cited in page 89.)

[25] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 3.0*. University of Tennessee, 2012. (Cited in pages 94, 98 and 99.)

[26] Michael Garey and David Johnson. *Computers and intractability*, volume 29. wh freeman, 2002. (Cited in pages 14, 38, 48 and 66.)

[27] Fred Glover. Heuristics for integer programming using surrogate constraints. *Decision Sciences*, 8(1):156–166, 1977. (Cited in pages 17 and 41.)

[28] Ralph Gomory. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Mathematical Society*, 64(5):275–278, 1958. (Cited in pages 10 and 33.)

[29] Mhand Hifi. An iterative rounding search-based algorithm for the disjunctively constrained knapsack problem. *Engineering Optimization*, 46(8):1109–1122, 2014. (Cited in pages 18, 41, 57, 74 and 103.)

[30] Mhand Hifi and Mustapha Michrafy. A reactive local search-based algorithm for the disjunctively constrained knapsack problem. *Journal of the Operational Research Society*, 57(6):718–726, 2006. (Cited in pages 17, 40 and 55.)

[31] Mhand Hifi and Mustapha Michrafy. Reduction strategies and exact algorithms for the disjunctively constrained knapsack problem. *Computers & operations research*, 34(9):2657–2673, 2007. (Cited in pages 18, 41, 73 and 102.)

[32] Mhand Hifi, Stephane Negre, Toufik Saadi, Sagvan Saleh, and Lei Wu. A parallel large neighborhood search-based heuristic for the disjunctively constrained knapsack problem. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 1547–1551. IEEE, 2014. (Cited in pages iii, 98 and 105.)

[33] Mhand Hifi and Nabil Otmani. A first level scatter search for disjunctively constrained knapsack problems. In *Communications, Computing and Control Applications (CCCA), 2011 International Conference on*, pages 1–6. IEEE, 2011. (Cited in pages 17, 18 and 41.)

[34] Mhand Hifi and Nabil Otmani. An algorithm for the disjunctively constrained knapsack problem. *International Journal of Operational Research*, 13(1):22–43, 2012. (Cited in pages 17, 18, 40, 41 and 57.)

[35] Mhand Hifi and Catherine Roucairol. Approximate and exact algorithms for constrained (un) weighted two-dimensional two-staged cutting stock problems. *Journal of combinatorial optimization*, 5(4):465–494, 2001. (Cited in pages 4 and 28.)

[36] Mhand Hifi, Sagvan Saleh, and Lei Wu. A fast large neighborhood search for disjunctively constrained knapsack problem. In *Combinatorial Optimization*, pages 396–407. Springer, 2014. (Cited in pages iii, 50 and 59.)

[37] Ellis Horowitz and Sartaj Sahni. Computing partitions with applications to the knapsack problem. *Journal of the ACM (JACM)*, 21(2):277–292, 1974. (Cited in pages 5, 11, 29 and 34.)

[38] Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Knapsack problems*. Springer, 2004. (Cited in pages 5, 7, 29 and 32.)

[39] Peter J Kolesar. A branch and bound algorithm for the knapsack problem. *Management Science*, 13(9):723–735, 1967. (Cited in pages 11 and 34.)

[40] Ailsa H Land and Alison G Doig. An automatic method of solving discrete programming problems. *Econometrica: Journal of the Econometric Society*, pages 497–520, 1960. (Cited in pages 11 and 34.)

[41] Hans Petter Langtangen and Aslak Tveito. *Advanced topics in computational partial differential equations: numerical methods and diffpack programming*, volume 33. Springer, 2003. (Cited in page 94.)

[42] Alexey Lastovetsky, Tahar Kechadi, and Jack Dongarra. *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 15th European PVM/MPI Users' Group Meeting, Dublin, Ireland, September 7-10, 2008, Proceedings*, volume 5205. Springer, 2008. (Cited in page 93.)

[43] V Maniezzo, LM Gambardella, and FD Luigi. Ant colony optimization (2004). (Cited in page 62.)

[44] Silvano Martello, David Pisinger, and Paolo Toth. Dynamic programming and strong bounds for the 0-1 knapsack problem. *Management Science*, 45(3):414–424, 1999. (Cited in pages 12, 35, 52 and 66.)

[45] Silvano Martello and Paolo Toth. An upper bound for the zero-one knapsack problem and a branch and bound algorithm. *European Journal of Operational Research*, 1(3):169–175, 1977. (Cited in pages 5, 9, 29 and 33.)

[46] Silvano Martello and Paolo Toth. *Knapsack problems*. Wiley New York, 1990. (Cited in pages 4, 5, 11, 20, 28, 29, 34 and 43.)

[47] Silvano Martello and Paolo Toth. Upper bounds and algorithms for hard 0-1 knapsack problems. *Operations Research*, 45(5):768–778, 1997. (Cited in pages 5 and 29.)

[48] GB Mathews. On the partition of numbers. *Proceedings of the London Mathematical Society*, 1(1):486–490, 1896. (Cited in pages 4 and 28.)

[49] Ibrahim H Osman and James P Kelly. *Meta-heuristics: theory and applications*. Springer, 1996. (Cited in pages 17 and 40.)

[50] Peter Pacheco. *An introduction to parallel programming*. Elsevier, 2011. (Cited in page 88.)

[51] Christos H Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity*. Courier Dover Publications, 1998. (Cited in page 96.)

[52] Ulrich Pferschy and Joachim Schauer. The knapsack problem with conflict graphs. *J. Graph Algorithms Appl.*, 13(2):233–249, 2009. (Cited in pages 12 and 35.)

[53] Telmo Pinto, Cláudio Alves, Raïd Mansi, and José Valério de Carvalho. Solving the multiscenario max-min knapsack problem exactly with column generation and branch-and-bound. *Mathematical Problems in Engineering*, 2015, 2015. (Cited in pages 19 and 42.)

[54] David Pisinger. *Algorithms for knapsack problems.* PhD thesis, University of Copenhagen, 1995. (Cited in pages 4, 5, 28 and 29.)

[55] David Pisinger and Stefan Ropke. Large neighborhood search. In *Handbook of metaheuristics*, pages 399–419. Springer, 2010. (Cited in pages 50 and 64.)

[56] David Pisinger and Mikkel Sigurd. Using decomposition techniques and constraint programming for solving the two-dimensional bin-packing problem. *INFORMS Journal on Computing*, 19(1):36–51, 2007. (Cited in page 48.)

[57] David Pisinger and Paolo Toth. Knapsack problems. In *Handbook of combinatorial optimization*, pages 299–428. Springer, 1999. (Cited in pages 4 and 28.)

[58] V Rajaraman and Siva Murthy. *Parallel computers: Architecture and programming.* Prentice-Hall of India Pvt.Ltd, 2004. (Cited in page 90.)

[59] Thomas Rauber and Gudula Rünger. *Parallel programming: For multicore and cluster systems.* Springer Science & Business, 2013. (Cited in page 89.)

[60] Mauricio Resende and PM Pardalos. *Handbook of applied optimization.* Oxford University Press, 2002. (Cited in pages 11 and 35.)

[61] Toufik Saadi. *Résolution séquentielles et parallèles des problèmes de découpe/placement.* PhD thesis, Université Panthéon-Sorbonne-Paris I, 2008. (Cited in page 91.)

[62] Ruslan Sadykov and François Vanderbeck. Bin packing with conflicts: a generic branch-and-price algorithm. *INFORMS Journal on Computing*, 25(2):244–255, 2013. (Cited in pages 48 and 96.)

[63] Natthawut Samphaiboon and Y Yamada. Heuristic and exact algorithms for the precedence-constrained knapsack problem. *Journal of optimization theory and applications*, 105(3):659–676, 2000. (Cited in pages 19 and 42.)

[64] Aminto Senisuka, Byungjun You, and Takeo Yamada. Reduction and exact algorithms for the disjunctively constrained knapsack problem. In *International Symposium, Operational Research Bremen*, 2005. (Cited in pages 18 and 38.)

[65] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *Principles and Practice of Constraint Programming,ÄîCP98*, pages 417–431. Springer, 1998. (Cited in pages 50, 56 and 96.)

[66] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference*, volume 1. MIT press, 1998. (Cited in pages 98 and 99.)

[67] Rob JM Vaessens, Emile HL Aarts, and Jan Karel Lenstra. A local search template. *Computers & Operations Research*, 25(11):969–979, 1998. (Cited in pages 17 and 40.)

[68] Tobias Wittwer. *Introduction to Parallel Programming*, volume 1. VSSD, 2006. (Cited in page 88.)

[69] Takeo Yamada, Seija Kataoka, and Kohtaro Watanabe. Heuristic and exact algorithms for the disjunctively constrained knapsack problem. *Information Processing Society of Japan Journal*, 43(9), 2002. (Cited in pages 12, 13, 15, 16, 18, 35, 36, 38, 39, 41, 55, 73 and 102.)

[70] Byungjun You and Takeo Yamada. A pegging approach to the precedence-constrained knapsack problem. *European journal of operational research*, 183(2):618–632, 2007. (Cited in pages 19 and 42.)

[71] Gang Yu. On the max-min 0-1 knapsack problem with robust optimization applications. *Operations Research*, 44(2):407–415, 1996. (Cited in pages 19 and 42.)

[72] Peiyi Zhao, Peixin Zhao, and Xin Zhang. A new ant colony optimization for the knapsack problem. In *Computer-Aided Industrial Design and Conceptual Design, 2006. CAIDCD'06. 7th International Conference on*, pages 1–3. IEEE, 2006. (Cited in page 63.)