Optimization algorithms for two knapsack problems

Zequn Wei

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

université angers

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE ANGERS
COMUE UNIVERSITÉ BRETAGNE LOIRE

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par
Zequn WEI

Optimization algorithms for two knapsack problems

Thèse présentée et soutenue à « Université d'Angers », le « 26 Mars 2021 »
Unité de recherche : LERIA
Thèse N° :

Rapporteurs avant soutenance :

M. Djamal HABET Professeur à Université d'Aix-Marseille
M. Mhand HIFI Professeur à Université de Picardie Jules Verne

Composition du Jury :

Président :		
Examinateurs :	M. Djamal HABET	Professeur à Université d'Aix-Marseille
	M. Mhand HIFI	Professeur à Université de Picardie Jules Verne
	Mme. Béatrice DUVAL	Professeur à Université d'Angers
	M. Jin-Kao HAO	Professeur à Université d'Angers
	M. Xiangjing LAI	Professeur à Nanjing University of Posts and Telecommunications
Dir. de thèse :	M. Jin-Kao HAO	Professeur à Université d'Angers

Table of Contents

General Introduction

Context

Many practical decision-making problems involve selecting a subset of objects from a set of candidate objects such that the selected objects optimize a given objective while satisfying some constraints. Knapsack problems (KP) [KPP04] are general models that allow such decision-making problems to be conveniently formulated. In this thesis, we focus on two representative knapsack problems: the set-union knapsack problem (SUKP) [GNY94] and the disjunctively constrained knapsack problem (DCKP) [YKW02]. As a variant of the popular KP, SUKP is to find a subset of candidate items (an item is composed of several distinct weighted elements) such that a profit function is maximized while a knapsack capacity constraint is satisfied. DCKP consists in packing a subset of pairwisely compatible items in a capacity-constrained knapsack in a way that the total profit of the selected items is maximized while satisfying the knapsack capacity. These two generalized knapsack problems can formulate additional relevant applications such as database partitioning [Nav+84], flexible manufacturing [GNY94], key-pose caching [LLD10], public key prototyping [Sch96], data allocating [WH20a] and public transportation [Hif+14].

Given their theoretical and practical significance, a number of solution methods have been developed including exact, approximation and heuristic algorithms and considerable progresses have been continually made since the introduction of these two problems. Meanwhile, given the $\mathcal{NP}$-hard nature of the problems, more powerful algorithms are always useful to push the limits of existing methods. In this work, we aim at advancing the state-of-the-art of solving these two problems effectively and robustly. From a perspective of performance assessment, we show the competitiveness of the proposed algorithms compared to the state-of-the-art algorithms on a variety of benchmark instances commonly used in the literature. We also perform additional experiments to shed lights on the roles of the key composing ingredients of the algorithms.

Objectives

This thesis is devoted to designing and implementing efficient heuristic and metaheuristic algorithms as well as verifying the effectiveness for solving the SUKP and the DCKP. The main objectives of this thesis can be summarized as follows.

— Study the specific features of these two problems and the weaknesses of the existing methods.
— Design discrete optimization approaches based on stochastic local search which directly operates in the binary search space.
— Evaluate the meaningfulness of the idea of kernel for solving difficult binary optimization problems.
— Investigate the parameter-free solution-based tabu search method to enhance a strongly intensified examination of the search.
— Integrate the threshold search technique with a population-based memetic framework to ensure a suitable balance between intensification and diversification.
— Evaluate the performance of the proposed algorithms on a wide range of benchmark instances and show computational results in comparison with state-of-the-art algorithms.
— Analyze the ingredients of the proposed methods to get useful insights about their impacts on the performances of the algorithms.

Contributions

The main contributions of this thesis are summarized below.

— For the SUKP, we have proposed three heuristic algorithms and achieved the following results:
  — First, we propose an iterated two-phase local search algorithm (I2PLS). We show for the first time that stochastic local search, which directly operates in the binary search space, can be a highly effective approach for solving the SUKP. We report improved best results for 18 large instances and equal best results for the 12 remaining instances. We also investigate for the first time the interest of the general mixed integer programming solver CPLEX for solving the SUKP. This work has been published in *Future Generation Computer Systems*.
  — Second, we present the kernel based tabu search algorithm (KBTS), which

combines for the first time the notion of kernel with the powerful tabu search method. Computational study performed on two sets of 60 benchmark instances indicated that the proposed algorithm dominates the current best SUKP algorithms in terms of solution quality, robustness and computation time. This work has been published in *Expert Systems with Applications*.

— Third, we investigate for the first time a multistart solution-based tabu search algorithm (MSBTS) for solving the SUKP. The proposed algorithm, which is parameter-free, combines a solution-based tabu search procedure with a multistart strategy to ensure an effective examination of candidate solutions. We demonstrate the interest of the MSBTS algorithm to deal with large instances and report new lower bounds for 7 large and difficult instances. This work has been published in *Applied Soft Computing*.

— For the DCKP, we introduce a threshold search based memetic algorithm (TSBMA) which combines the memetic framework with threshold search to find high quality solutions. Extensive computational assessments on two sets of 6340 benchmark instances in the literature demonstrate that the proposed algorithm is highly competitive compared to the state-of-the-art methods. This work is being revised for *Computers & Operations Research*.

Organization

The organization of the thesis is summarized as follows.

— In the first chapter, we start with the introduction of the general 0/1 knapsack problem and its variants. Then, we give the definitions of the two knapsack problems considered in this thesis and recall a number of real-life applications related to these two problems. We also provide an overview of the existing approaches in the literature for solving them, including exact algorithms, approximation algorithms, heuristic and metaheuristic algorithms. Moreover, we introduce the benchmark instances commonly tested in the literature.

— In the second chapter, we propose an effective iterated two-phase local search algorithm which relies on two innovative and complementary search components specially designed for the SUKP. Then we show the competitiveness of the proposed algorithm compared to the state-of-the-art algorithms on the set of 30 benchmark instances commonly used in the literature. We also show that the general mixed

integer programming solver can find some optimal solutions based on a simple 0/1 linear programming model. Finally, we perform an analysis of the parameters and the ingredients of the proposed algorithm to investigate their impacts on its performance.

— In the third chapter, we present another heuristic algorithm called kernel based tabu search algorithm to solve the SUKP. We detail the three complementary search components used to perform an effective examination of the search space. That is, a local search procedure is used to find various local optima, a kernel search method is employed to discover additional high-quality solutions within particular areas, and a non-kernel search method is applied to ensure a guided diversification. In the following, we perform an extensive evaluation of our algorithm and comparisons with state-of-the-art SUKP algorithms. Meanwhile, we analyze the parameters used in the proposed algorithm and the kernel based components.

— In the fourth chapter, we advance the state-of-the-art of solving the SUKP by proposing the first multistart solution-based tabu search algorithm. We first present its main scheme and then describe its components including the greedy randomized initialization procedure and the solution-based tabu search procedure. Then we carry out computational experiments and a time-to-target analysis to evaluate the performance of the proposed algorithm compared to the reference algorithms based on two sets of 60 benchmark instances. Finally, we provide additional analysis to investigate the influences of the main ingredients of our algorithm.

— In the fifth chapter, we investigate for the first time the population-based memetic framework to solve the DCKP and present an effective algorithm mixing threshold based local optimization and crossover based solution recombination. Then we show extensive computational results and comparisons with state-of-the-art DCKP algorithms based on two sets of 6340 benchmark instances. Finally, we analyze two essential components of the proposed algorithm: the importance of the threshold search and the contribution of the operation-prohibiting mechanism.

— In the last chapter, we summarize the contributions of this thesis and provide some perspectives for future research.

Part I

Introduction

Chapter 1

Introduction

1.1 Knapsack problems

Knapsack problems are very general and useful models able to formulate numerous real-world problems in a variety of fields. For instance, suppose that a firm has a fixed global budget envelope for project investment as well as a number of candidate projects. Suppose also that each candidate project requires a budget and its implementation implies a gain. One important decision problem is then to select a subset of projects from the candidate set such that the total gain of the retained projects is maximized and the total budget allocated to the retained projects is no more than the available budget envelope. This practical problem as well as many other similar problems can conveniently be formulated with the following general 0/1 knapsack problem (KP) [KPP04].

Given a knapsack with a positive weight capacity C, and a set $N = \{1, \ldots, n\}$ of items where each item $i = \{1, \ldots, n\}$ has a weight $w_i > 0$ and a profit $p_i > 0$. The KP involves selecting a subset $S \subseteq N$ of items in a way that the total profit of the selected items is maximized, while the weight sum of S does not exceed the knapsack capacity. Let x_i be a binary variable such that $x_i = 1$ if item i is selected, $x_i = 0$ otherwise. Formally, the KP can be stated as follows [KPP04].

$$(KP) \qquad \text{Maximize} \quad f(S) = \sum_{i=1}^{n} p_i x_i \tag{1.1}$$

$$\text{subject to} \quad W(S) = \sum_{i=1}^{n} w_i x_i \leq C, \; S \subseteq N, \tag{1.2}$$

$$x_i \in \{0, 1\}, i = 1, \ldots, n. \tag{1.3}$$

As indicated in [KPP04], the KP can be used to model many real-world decision-making problems such as selection of investments and portfolios, generating keys for cryptosystems, and finding the least wasteful way to cut raw materials.

The decision version of the KP is known to be $\mathcal{NP}$-complete in [Kar72]. As a basic model, the KP formulation has a variety of variations and extensions, such as:

— **Subset sum problem (SSP)**
Given a knapsack with a positive weight capacity C and a set $N = \{1, \ldots, n\}$ of items where each item $i = \{1, \ldots, n\}$ has a weight $w_i > 0$, the SSP is to select a subset of items S from N such that the total weight is maximized while satisfying the knapsack capacity C. By introducing a decision variable x_i to indicate whether

item i is selected, the SSP can be defined as follows.

$$(SPP) \qquad \text{Maximize} \quad f(S) = \sum_{i=1}^{n} w_i x_i \tag{1.4}$$

$$\text{subject to} \quad W(S) = \sum_{i=1}^{n} w_i x_i \leq C, \ S \subseteq N, \tag{1.5}$$

$$x_i \in \{0, 1\}, i = 1, \ldots, n. \tag{1.6}$$

It is easy to observe that the KP is equivalent to the $\mathcal{NP}$-hard SSP when the profit of each item is equal to the weight. The SSP model can be used to formulate practical applications, such as public-key cryptosystems [MH78] and scheduling problems [HOV94]. Moreover, the SSP is also closely related to other important problems [CS16], such as traveling salesman problem, satisfiability problem, factorization problem, integer programming problem. For solution methods of this problem, including dynamic programming algorithms, approximation algorithms and hybrid algorithms, see [KPP04; TM90].

— **Quadratic knapsack problem (QKP)**

Let C be the positive knapsack capacity and $N = \{1, \ldots, n\}$ be a set of items, where each item $i = \{1, \ldots, n\}$ has a profit $p_{ii} > 0$ and a weight $w_i > 0$. In addition, each pair of items i and j $(1 \leq i \neq j \leq n)$ has a pairwise profit p_{ij} if both of them are selected. Then QKP involves determining a subset of items $S \subseteq N$ to maximize the total profit of S while ensuring that the total weight of the items of S does not exceed the knapsack capacity C. Suppose a binary variable x_j is set to 1 if item j is selected, or $x_j = 0$ otherwise. Then the QKP can be written as follows.

$$(QKP) \qquad \text{Maximize} \quad f(S) = \sum_{i=1}^{n} \sum_{j=i}^{n} p_{ij} x_i x_j \tag{1.7}$$

$$\text{subject to} \quad W(S) = \sum_{j=1}^{n} w_j x_j \leq C, \ S \subseteq N, \tag{1.8}$$

$$x_j \in \{0, 1\}, j = 1, \ldots, n. \tag{1.9}$$

The QKP can be reduced to the KP by restricting all the pairwise profit p_{ij} to 0.

QKP is $\mathcal{NP}$-hard in the strong sense [CPT99] and thus is computationally difficult. As indicated in [KPP04], the QKP is a useful model for a number of real-world applications, such as selecting locations for satellite stations, airports and railway stations. Over the past decades, various solution methods have been proposed in the literature for solving the QKP [CH17; KPP04], including exact algorithms and heuristic algorithms.

— **Multiple knapsack problem (MKP)**

Let $N = \{1, \ldots, n\}$ be a set of items and $M = \{1, \ldots, m\}$ be a set of knapsacks. Each item $i = \{1, \ldots, n\}$ has a profit $p_i > 0$ and a weight $w_i > 0$. Each knapsack $j = \{1, \ldots, m\}$ has capacity C_j. Then the MKP aims to assign some items to the m knapsacks in a way that the overall profit of the assigned items is maximized while the capacity constraint of each knapsack is satisfied. Let $S \subseteq N$ be a subset of items, x_{ij} be a binary variable such that $x_{ij} = 1$ if item i is assigned to knapsack j, $x_{ij} = 0$ otherwise. Then the MKP can be formalized as follows.

$$(MKP) \qquad \text{Maximize} \quad f(S) = \sum_{i=1}^{n} \sum_{j=i}^{m} p_i x_{ij} \tag{1.10}$$

$$\text{subject to} \quad W(S) = \sum_{j=1}^{m} w_i x_{ij} \leq C_j, i = 1, \ldots, n, \tag{1.11}$$

$$\sum_{i=1}^{n} x_{ij} \leq 1, j = 1, \ldots, m, \tag{1.12}$$

$$x_{ij} \in \{0, 1\}, i = 1, \ldots, n, j = 1, \ldots, m. \tag{1.13}$$

It is obvious that the MKP reduces to the KP when the number of knapsacks is set to 1. As introduced in [EC71; FMW96; SAR17], the $\mathcal{NP}$-hard MKP has numerous applications, such as cargo loading, designing processors for mainframe computers, designing layout of electronic circuits, sugar cane alcohol production and self-sufficient system for military operations. The solution methods of the MKP are introduced in [Del+19; KPP04; TM90], including exact algorithms, approximation algorithms and heuristic algorithms.

— **Multiple-choice knapsack problem (MCKP)**

Given m disjoint sets $N_1, \ldots, N_m$ of items, where each item $j \subseteq N_i$ has a profit $p_{ij} > 0$ and a weight $w_{ij} > 0$. The MCKP is to pack exactly one item from each

item set N into a capacity (C) constrained knapsack to maximize the total profit of the selected items. By introducing a decision variable to indicate whether item j is inserted the knapsack, the MCKP can be defined as follows.

$$(MCKP) \qquad \text{Maximize} \quad f(S) = \sum_{i=1}^{m} \sum_{j \in N_i} p_{ij} x_{ij} \tag{1.14}$$

$$\text{subject to} \quad W(S) = \sum_{i=1}^{m} \sum_{j \in N_i} w_{ij} x_{ij} \leq C, \tag{1.15}$$

$$\sum_{j \in N_i} x_{ij} = 1, i = 1, \ldots, m, \tag{1.16}$$

$$x_{ij} \in \{0, 1\}, i = 1, \ldots, n, j \in N_i. \tag{1.17}$$

MCKP is a generalization of the conventional KP and has a variety of practical applications [Loj+20; Nau78; SZ79], such as capital budgeting, menu planning, product line pricing, and fault-tolerant system designing. The existing approaches for solving MCKP are introduced in [GL98; He+16; KPP04].

— **Multi-dimensional knapsack problem (MMKP)**

Given a set $M = \{1, \ldots, m\}$ of knapsacks, where each knapsack has a positive weight capacity C_j. Let $N = \{1, \ldots, n\}$ be a set of items, where each item i has a profit p_i and consumes a given weight w_{ij} for each knapsack j. Then the MMKP involves finding a subset of $S \subseteq N$ items such that the total profit of the selected items is maximized while their weights do not exceed the knapsack capacity. By introducing a binary decision variable x_i to indicate whether item j is packed in the knapsack, the MCKP can be defined as follows.

$$(MMKP) \qquad \text{Maximize} \quad f(S) = \sum_{i=1}^{n} p_i x_i \tag{1.18}$$

$$\text{subject to} \quad W(S) = \sum_{i=1}^{n} w_{ij} x_i \leq C_j, j = 1, \ldots, m, \tag{1.19}$$

$$x_i \in \{0, 1\}, i = 1, \ldots, n. \tag{1.20}$$

The MMKP is a useful model able to formulate a number of real-life applications [Fré04; Gav82; GG66; Shi79], including resource allocation, cargo loading, cutting

stock problem and capital budgeting. However, the MMKP is computationally difficult given that it belongs to the class of $\mathcal{NP}$-hard problems. Due to its relevance, a variety of solution methods have been devised for solving the MMKP [Fré04; FH05; KPP04], including exact algorithms, approximation algorithms and heuristic algorithms.

— **Set-union knapsack problem (SUKP)**
The SUKP is to find a subset of candidate items (an item is composed of several distinct weighted elements) such that a profit function is maximized while a knapsack capacity constraint is satisfied. Since this thesis is devoted to designing effective approaches for solving the SUKP, more details about the SUKP will be introduced in Section 1.2.

— **Disjunctively constrained knapsack problem (DCKP)**
The DCKP consists in packing a subset of pairwisely compatible items in a capacity-constrained knapsack such that the total profit of the selected items is maximized while satisfying the knapsack capacity. The introduction of the DCKP will also be given in Section 1.3.

1.2 Set-union knapsack problem

1.2.1 Problem introduction

Given a set of elements $U = \{1, \ldots, n\}$ and a set of items $V = \{1, \ldots, m\}$, each element has a weight $w_j > 0$ and each item has a profit $p_i > 0$. The items and elements are associated by a relation matrix $R_{ij}[m \times n]$ such that each item i corresponds to a subset of elements $U_i \subseteq U$. Let C be the capacity of a given knapsack. Then the set-union knapsack problem is to select a subset of items S from V such that the total profit of S is maximized, while the total weight of the covered elements does not exceed the knapsack capacity C. Formally, the SUKP can be stated as follows.

$$(SUKP) \qquad \text{Maximize} \quad f(S) = \sum_{i \in S} p_i \tag{1.21}$$

$$\text{subject to} \quad W(S) = \sum_{j \in \cup_{i \in S} U_i} w_j \leq C, \; S \subseteq V. \tag{1.22}$$

It is worth noting that for a given subset S of items, the weight w_j of an element j is

counted only once in $W(S)$ even if the element belongs to more than one selected items. One notices that the conventional knapsack problem is a special case of the SUKP. Indeed, the SUKP reduces to the KP when we set $m = n$ and $V = U$. The SUKP also generalizes the $\mathcal{NP}$-hard densest k-subhypergraph problem (DkSH) that aims to determine a set of k nodes of a hypergraph to maximize the number of hyperedges of the subhypergraph induced by the set of the selected nodes [Chl+18]. In fact, the SUKP reduces to the DkSH when we consider the elements and items as the nodes and hyperedges of a hypergraph respectively, with unit weights and unit profits as well as a capacity of k.

We also propose a 0/1 linear programming model that is solved by the general integer linear programming (ILP) solver CPLEX. Our model is based on the mathematical model of the SUKP introduced in [He+18] (see the detailed description of this model in Section 2, page 78 of this reference), which is, however, inapplicable by the CPLEX solver. We introduce below the modified 0/1 linear programming model that is suitable for the solver. For an arbitrary non-empty item set $S \subset V$ represented by its binary vector $S = (y_1, \ldots, y_m)$ such that $y_i = 1$ $(i = 1, \ldots, m)$ if item i is selected in S, and $y_i = 0$ otherwise. Let R be a $m \times n$ binary relation matrix such that $R_{ij} = 1$ if element j belongs to item i, and $R_{ij} = 0$ otherwise. Furthermore, for each element j $(j = 1, \ldots, n)$, define $L_j = \sum_{i=1}^{m} y_i R_{ij}$ that counts the number of appearances of element j in the items of S. Let x_j be a binary variable such that $x_j = 1$ if $L_j > 0$, and $x_j = 0$ otherwise, that is, x_j indicates whether element j is involved in calculating the total weight of S. Then our 0/1 linear programming model for the SUKP is defined as follows.

$$(SUKP) \quad \text{Maximize} \quad f(S) = \sum_{i=1}^{m} p_i y_i \tag{1.23}$$

$$\text{s.t.} \quad W(S) = \sum_{j=1}^{n} w_j x_j \leq C \tag{1.24}$$

$$x_j = \begin{cases} 1, & if\ L_j > 0; \\ 0, & \text{otherwise.} \end{cases} \tag{1.25}$$

$$L_j = \sum_{i=1}^{m} y_i R_{ij}, j = 1, \ldots, n \tag{1.26}$$

$$y_i \in \{0, 1\}, i = 1, \ldots, m. \tag{1.27}$$

Constraints 1.24–1.26 jointly ensure that the weight w_j of an element j is counted only once in $W(S)$ even if the element appears in more than one selected items and the capacity constraint is satisfied. Constraint 1.27 guarantees that each item is selected at most once. Equation (1.23) maximizes the total profit of the selected items.

1.2.2 Applications

Like other knapsack models, the SUKP has a number of practical applications. As an example, we consider the following decision-making problem to optimally allocate data in large cyber systems [TX16]. Given a centralized cyber system with a memory of fixed capacity holding a set of services (or requests) with profits, where each service contains a set of data objects. Each data object will consume a certain amount of memory when it is invoked, and multiple use of the same data object will not cause additional memory consumption. The goal is to select a subset of services, among the candidate services, such that the total profit of the selected services is maximized while the total memory consumed by the underlying data objects meets the memory capacity of the cyber system. This application can be conveniently formulated by the SUKP model where an item corresponds to a service with its profit and an element corresponds to a data object with its memory consumption (element weight). Then, solving the data allocation problem is equivalent to finding the optimal solution to the resulting SUKP problem.

The SUKP has other relevant applications related to decision-making and intelligent systems, including database partitioning [Nav+84], flexible manufacturing [GNY94], key-pose caching [LLD10], financial decision making [KPP04], and public key prototyping [Sch96].

1.2.3 Related work

In terms of computational complexity theory, the decision version of the SUKP is known to be $\mathcal{NP}$-complete [GNY94]. Therefore, from the perspective of solution methods, solving the SUKP is a highly challenging task. Given its practical and theoretical relevance, a number of algorithms for the SUKP have been introduced in the literature. The existing approaches for solving the SUKP can be classified into three categories as follows.

— **Exact and approximation algorithms**

These algorithms are theoretically able to find the optimal solutions or solutions of guaranteed quality.

[GNY94] introduced the SUKP for the first time and proved the SUKP is still $\mathcal{NP}$-hard even in very restricted cases. Based on the general dynamic programming method, an exact algorithm for solving the SUKP was devised, which is bounded by an exponential function corresponding to the cut width of the item-adjacency hypergraph. Sufficient conditions for the proposed methods to run in polynomial time was also presented.

[Aru14] proposed a greedy algorithm that is based on a previous approximation algorithm for the related budgeted maximum coverage problem. The algorithm provides a $(1 - e^{-\frac{1}{d}})$ approximation for the SUKP with the additional restriction that the number of items in which an element is present is bounded by a constant d.

[Tay16] designed an approximation algorithm using results of the related densest k-subhypergraph problem. The proposed algorithm is shown to achieve, for any given $\epsilon > 0$, an approximation ratio of at most $O(n^{\alpha_m+\epsilon})$ for $\alpha_m = \frac{2}{3}[m - 1 - \frac{2m-2}{m2+m-1}]$, where the subsets have at most m elements.

Focusing on theoretical aspects of the SUKP, these studies do not show computational results.

— **Population-based hybrid algorithms**

These algorithms are based on various bio-inspired metaheuristics operating with a population of solutions and associated search operators.

[He+18] devised the first binary artificial bee colony algorithm (BABC) for solving the SUKP. Since this approach inevitably generates infeasible solutions, a greedy repairing and optimization procedure (named S-GROA) is proposed to handle infeasible solutions. To assess the proposed algorithm, large scale experiments were performed based on a set of 30 new benchmark instances (with 85 to 500 items and elements). Comparisons with three other population-based algorithms (genetic algorithm, continuous artificial bee colony algorithm and differential evolution strategies) showed the competitiveness of the BABC algorithm.

[OB19] presented a binary swarm intelligence algorithm (gPSO) that combines the genetic algorithm with particle swarm optimization. The proposed algorithm employs a developed optional mutation operator that exponentially decreases the diversity of the population, which can avoid local optima at earlier iterations. The

gPSO algorithm has the advantage of requiring no transfer function. Computational results and statistical tests on 30 benchmark instances indicate that the proposed approach outperforms the previously reported algorithms.

[HW18] introduced a group theory-based optimization algorithm (GTOA) for knapsack problems including the SUKP. By applying the algebraic group operations and the greedy repairing and optimization procedure (S-GROA), both the feasible and infeasible search space is examined by the GTOA. The computational results on 30 benchmark instances demonstrate that the proposed GTOA algorithm performs better than the existing evolutionary algorithms such as genetic algorithm, binary particle swarm optimization, binary artificial bee colony, and their improved variations.

[BOS18] developed a modified weighted superposition attraction algorithm (WSA) for stationary binary optimization problems including the SUKP. With the help of the proposed modification, WSA does not require any transfer functions. The dedicated step sizing function is beneficial to avoid premature convergence and local optima traps.

[Ozs19] applied the learning mechanisms to find near-optima solutions of the SUKP. The proposed swarm-based optimization algorithm (intAgents) uses artificial search agents with individual cognitive intelligence to diversify the search. Each search agent is guided by the information-sharing techniques to explore new search regions. Extensive experiments on 30 benchmark instances show the effectiveness of the proposed algorithm.

[FAG19; FYW19] presented two versions of moth search algorithms (MS and EMS) for solving the SUKP. These algorithms adopt an empirical transfer function to map the continuous space to the discrete space and maintain both continuous and discrete solutions during the search. The MS algorithm employed twelve transfer functions to solve 15 SUKP benchmark instances and achieved good results. The EMS algorithm enhances the previous moth search algorithm by introducing an enhanced interaction operator (EIO) to replace the Lévy flight operator in the original MS and shows better performance than MS.

[WH20b] presented a hybrid Jaya algorithm (DHJaya) based on the differential evolution crossover operator and Cauchy mutation strategy. A double coding mechanism is introduced for the proposed Jaya algorithm and an improved repair-optimization strategy (MS-GROA) is employed to handle the infeasible solutions.

Experimental results on 30 benchmark instances show that the proposed DHJaya algorithm is superior to the original Jaya algorithm and the basic differential evolution algorithm.

[LH19] combined the estimation of distribution algorithm based on Lévy flight (LFEDA) with a quadratic greedy repair and optimization approach (Q-GROA). The LFEDA algorithm has the advantage of increasing the diversity of the population and escaping from the local optima trap. Computational testing on 30 benchmark instances shows that the proposed algorithm is more robust than the previous algorithms for solving the SUKP.

[GO20] designed a binary grey wolf optimization algorithm (GWO) which based on the warm intelligence framework. The GWO algorithm employs the evolutionary and adaptive inheritance mechanisms to operate in the binary spaces directly. A multi-parent crossover operation and an adaptive mutation are presented to avoid premature convergence. Evaluated on 30 benchmark instances, the proposed GWO algorithm is shown to be effective in finding high quality solutions.

In terms of computational performances, these approaches achieved interesting results. However, these algorithms are rather complex in design and most of them solve the binary SUKP problem indirectly by searching a continuous space.

— **Local search algorithms**

Contrary to the above population algorithms, local search algorithms solve the binary SUKP problem directly by examining candidate solutions in a discrete search space.

[Lin+19] applied a local search procedure (tabu search) into the binary particle swarm optimization framework (HBPSO/TS) to solve the SUKP. The proposed HBPSO/TS algorithm explores both the feasible and infeasible search space by using an adaptive penalty function. A tabu based mutation procedure is also employed to guide the search to promising regions. Experimental results on 30 benchmark instances indicate that HBPSO/TS performs much better than the previously reported algorithms according to the solution quality.

Computational results indicated that the local search approaches represent the current state-of-the-art in the literature. In this thesis, we propose three different local search algorithms to advance the state-of-the-art of solving the SUKP, which are proved to be highly effective in terms of both solution quality and computational efficiency.

1.2.4 Benchmarks

For the SUKP, the benchmark instances can be divided into two sets [1] as follows.

— **Set I (30 instances)**: Introduced in [He+18], this set of instances have 85 to 500 items and elements with the following features. For each instance with m items and n elements, the items and elements are associated by a $m \times n$ binary relation matrix R, where $R_{ij} = 1$ indicates that item i includes element j. Each instance is further characterized by two parameters: α represents the density of $R_{ij} = 1$ in the relation matrix R (i.e., $\alpha = (\sum_{i=1}^{m} \sum_{j=1}^{n} R_{ij})/(mn)$), β denotes the ratio of knapsack capacity C to the total weight of the elements (i.e., $\beta = C/\sum_{j=1}^{n} w_j$). Thus, each SUKP instance can be designated as $m\_n\_\alpha\_\beta$. These instances are widely tested in the literature.

— **Set II (30 instances)**: Introduced in [WH20a], this set of instances have the same characteristics as those of Set I, but are large in size with 585 to 1000 items and elements. Following [He+18], the profit and weight values of these instances are generated randomly in [1,500].

To facilitate the presentation of our computational results in the following chapters, these two sets of 60 benchmark instances are divided into three classes according to the relationship between the number of items and elements and denoted by F1−F20 ($m > n$), S1−S20 ($m = n$) and T1−T20 ($m < n$), respectively. The ID of each instance are shown in Table 1.1.

1.3 Disjunctively constrained knapsack problem

1.3.1 Problem introduction

Let $V = \{1, \ldots, n\}$ be a set of n items, where each item $i = \{1, \ldots, n\}$ has a profit $p_i > 0$ and a weight $w_i > 0$. Let $G = (V, E)$ be a conflict graph, where V is the set of n items and an edge $\{i, j\} \in E$ defines the incompatibility of items i and j. Let $C > 0$ be the capacity of a given knapsack. Then the DCKP involves finding a subset S of pairwisely compatible items of V to maximize the total profit of S while ensuring that the total weight of S does not surpass the knapsack capacity C. Let x_i be a binary variable such that $x_i = 1$ if item i is selected, $x_i = 0$ otherwise. Formally, the DCKP can be stated as

1. They are available at: http://www.info.univ-angers.fr/pub/hao/SUKP_KBTS.html.

Table 1.1 – Summary of main characteristics of the 100 SUKP instances of Set I.

Set	Instance	ID	Instance	ID	Instance	ID
I	100_85_0.10_0.75	$F1$	100_100_0.10_0.75	$S1$	85_100_0.10_0.75	$T1$
I	100_85_0.15_0.85	$F2$	100_100_0.15_0.85	$S2$	85_100_0.15_0.85	$T2$
I	200_185_0.10_0.75	$F3$	200_200_0.10_0.75	$S3$	185_200_0.10_0.75	$T3$
I	200_185_0.15_0.85	$F4$	200_200_0.15_0.85	$S4$	185_200_0.15_0.85	$T4$
I	300_285_0.10_0.75	$F5$	300_300_0.10_0.75	$S5$	285_300_0.10_0.75	$T5$
I	300_285_0.15_0.85	$F6$	300_300_0.15_0.85	$S6$	285_300_0.15_0.85	$T6$
I	400_385_0.10_0.75	$F7$	400_400_0.10_0.75	$S7$	385_400_0.10_0.75	$T7$
I	400_385_0.15_0.85	$F8$	400_400_0.15_0.85	$S8$	385_400_0.15_0.85	$T8$
I	500_485_0.10_0.75	$F9$	500_500_0.10_0.75	$S9$	485_500_0.10_0.75	$T9$
I	500_485_0.15_0.85	$F10$	500_500_0.15_0.85	$S10$	485_500_0.15_0.85	$T10$
II	600_585_0.10_0.75	$F11$	600_600_0.10_0.75	$S11$	585_600_0.10_0.75	$T11$
II	600_585_0.15_0.85	$F12$	600_600_0.15_0.85	$S12$	585_600_0.15_0.85	$T12$
II	700_685_0.10_0.75	$F13$	700_700_0.10_0.75	$S13$	685_700_0.10_0.75	$T13$
II	700_685_0.15_0.85	$F14$	700_700_0.15_0.85	$S14$	685_700_0.15_0.85	$T14$
II	800_785_0.10_0.75	$F15$	800_800_0.10_0.75	$S15$	785_800_0.10_0.75	$T15$
II	800_785_0.15_0.85	$F16$	800_800_0.15_0.85	$S16$	785_800_0.15_0.85	$T16$
II	900_885_0.10_0.75	$F17$	900_900_0.10_0.75	$S17$	885_900_0.10_0.75	$T17$
II	900_885_0.15_0.85	$F18$	900_900_0.15_0.85	$S18$	885_900_0.15_0.85	$T18$
II	1000_985_0.10_0.75	$F19$	1000_1000_0.10_0.75	$S19$	985_1000_0.10_0.75	$T19$
II	1000_985_0.15_0.85	$F20$	1000_1000_0.15_0.85	$S20$	985_1000_0.15_0.85	$T20$

follows.

$$(DCKP) \qquad \text{Maximize} \quad f(S) = \sum_{i=1}^{n} p_i x_i \tag{1.28}$$

$$\text{subject to} \quad W(S) = \sum_{i=1}^{n} w_i x_i \leq C, \ S \subseteq V, \tag{1.29}$$

$$x_i + x_j \leq 1, \forall (i,j) \in E, \tag{1.30}$$

$$x_i \in \{0,1\}, i = 1, \ldots, n. \tag{1.31}$$

Objective function (1.28) commits to maximize the total profit of the selected item set S. Constraint (1.29) ensures that the knapsack capacity constraint is satisfied. Constraints (1.30), called disjunctive constraints, guarantee that two incompatible items are never selected simultaneously. Constraint (1.31) forces that each item is selected at most once.

1.3.2 Applications

It is easy to observe that the DCKP reduces to the $\mathcal{NP}$-hard KP when G is an empty graph. The DCKP is equivalent to the $\mathcal{NP}$-hard maximum weighted independent set problem [GJ79] when the knapsack capacity is unbounded. Moreover, the DCKP is closely related to other combinatorial optimization problems, such as the multiple-choice knapsack problem [KPP04], and the bin packing problem with conflicts [Jan99].

In addition to its theoretical significance, the DCKP is a useful model for practical applications where the resources with conflicts cannot be used simultaneously while a given budget envelope cannot be surpassed. As an example, we consider the following practical project investment scenario. Given a set of projects where each project has a budget and a gain. The goal is to select a subset of projects in a way that the total gain is maximized, while the total budget does not surpass the global budget envelope. This problem can be conveniently formulated by the KP model, where a project corresponds to an item and the budget envelope corresponds to the knapsack capacity. However, the project investment problem may involve other constraints in real-life applications. A typical situation is that some projects can not be invested simultaneously due to the practical limits, such as locations, project lifecycle, facilities requirement, human resources, laws and regulations etc. Then the project investment problem is to find the optimal subset of projects while satisfying both the budget constraint and the disjunctive constraints, which can be conveniently formulated by the DCKP model.

As indicated in [Hif+14; QW17a], a number of practical applications can be formulated by the DCKP model, including resource allocation, loading of vehicles, public transportation, and scheduling problems.

1.3.3 Related work

Due to its relevance, the DCKP has received considerable attention in the past two decades. As the literature review shown in this section, considerable progresses have been continually made since the introduction of the problem. Existing solution methods can be roughly classified into two categories as follows.

— **Exact and approximation algorithms**

These algorithms are able to guarantee the quality of the solutions they find. [YKW02] introduced the DCKP for the first time and proposed an implicit enumeration algorithm to find upper bounds by relaxing the disjunctive constraints in

a Lagrangean way. The proposed algorithm is able to solve the DCKP instances up to 1000 items by integrating an interval reduction method with some pruning techniques. These instances are generated randomly with uncorrelated profits and weights (range from 1 to 100, independently) and very small conflict graph densities (range from 0.001 to 0.02).

[HM07] presented three versions of an exact algorithm based on a local reduction strategy. The proposed exact approach starts its search from a lower bound obtained by a reactive local search procedure, and then applies the reduction strategies to fix some decision variables to their optimum. Then the first version of the algorithm adopts an exact branch and bound algorithm to solve the reduced problem. The second version of the algorithm accelerates the search by combining a dichotomous search strategy with a reduction procedure. Based on a modified dichotomous search algorithm, the third version of the algorithm is introduced to solve the DCKP instances with large densities.

[PS09] devised a pseudo-polynomial time and space algorithm for solving three special cases of the DCKP, including trees, graphs with bounded treewidth and chordal graphs, and proved the DCKP is strongly $\mathcal{NP}$-hard on perfect graphs. Then the fully polynomial time approximation schemes (FPTAS) can be obtained by the proposed algorithm.

[Sal+18] divided the DCKP into two subproblems: binary knapsack problem and the independent set problem, and discussed the valid inequalities of these problems. Then a branch-and-cut algorithm is developed that combines a greedy clique generation procedure with a separation procedure. Experimental study is carried out to compare the proposed algorithm with the CPLEX solver.

[BCM17] presented a new branch-and-cut algorithm (BCM) to solve the DCKP optimally. The branching procedure solves the binary knapsack problem optimally by a dynamic programming algorithm while neglecting the disjunctive constraints. The upper bounding procedure considers both the knapsack constraint and the disjunctive constraints by using the weighted clique cover bound applied for the maximum weight stable set problem. Extensive experiments on 4800 benchmark instances indicate that BCM outperforms the previous algorithms for solving the DCKP, however, it is not particularly effective for solving the DCKP instances with small densities.

[PS17] applied the approximation methods of modular decompositions and clique

separators for solving the DCKP, and showed complexity results for the DCKP on special graph classes, including general graph, bounded treewidth graph, chordal graph, weakly chordal graph, planar graph and perfect graph. The existence of a polynomial time approximation scheme (PTAS) for H-minor free conflict graphs is proved.

[GR19] designed a dynamic programming algorithm that based on a tree-structure to represent the conflict graph. The pseudo-polynomial solutions of co-graphs were obtained and then extended to conflict graphs of bounded clique-width. Finally, the FPTAS can be achieved for the DCKP on conflict graphs of bounded clique-width.

[CFS21] presented a new and efficient branch-and-bound algorithm (CFS) based on an $n-$ary branching scheme and solved the integer linear programming formulation of the DCKP by using the CPLEX solver. Given the high pruning potential of CFS and the low computational effort required by branch-and-bound procedure, the proposed algorithm performs better than previous exact algorithms in terms of both solution quality and computational time for most of the 6240 instances tested.

— **Heuristic algorithms**

These algorithms aim to find good near-optimal solutions.

[YKW02] proposed a greedy algorithm to generate an initial solution with good quality and a 2-opt neighborhood search algorithm to improve the obtained solution. The proposed algorithm is able to obtain a lower bound within a reasonable time for instances of large size.

[HM06] reported a reactive local search algorithm (RLS) for the DCKP, which combines a complementary constructive procedure to improve the initial solution and a degrading procedure to diversify the search. A memory list (tabu list) is employed to avoid revisiting previous encountered solutions. Experimental results on a set of 50 new instances with 500 and 1000 items disclose that the proposed RLS algorithm is able to obtain some high-quality solutions within a reasonable time.

[AHM11] presented three versions of local algorithms based on the local branching techniques. The first version starts with a feasible solution provided by the basic rounding solution procedure (BRSP) and then uses the standard local branching technique to solve the DCKP. The second version applies a two-phase solution procedure (TPSP) including a rounding procedure to fix a subset of the items

and a truncated exact procedure to solve the reduced problem. The third version enhances the TPSP by introducing a diversification strategy.

[HO11] proposed a first level scatter search (SS) algorithm with the following procedures: 1) a starting solution-generator procedure to generate an initial solution; 2) a diversification generation procedure to generate diverse solutions; 3) an improvement procedure to improve or repair the current solution; 4) a reference update procedure to produce the reference set for the next step; 5) a subset generator procedure to generate groups of candidate solutions; 6) a solution combination procedure to produce offspring solutions. Computational comparisons indicate that the SS outperforms other previous algorithms as well as the CPLEX solver.

[HO12] reported another two versions of the SS algorithm based on the solution combination method. The first version of the SS employs a greedy combination method that takes into account both the structure and the relative profit per weight-degree values associated with each item. The second version of the SS adopts an alternative combination method that includes a variant of the 3-opt procedure. Computational experience discloses that the proposed algorithm is efficient for the DCKP instances of medium and large size.

[Hif14] devised an iterative rounding search-based algorithm (IRS) that uses a rounding strategy to perform a linear relaxation of the fractional variables and a neighborhood search procedure to improve the current solutions. Experimental results show that the proposed IRS algorithm outperforms the CPLEX solver and discovers new best-known results for most of the 50 instances tested.

[HSW14] proposed a fast large neighborhood search-based heuristic (LNSBH), which combines a two-phase procedure to generate an initial solution with good quality and a large neighborhood search procedure to diversify the search. The performance of the proposed algorithm is confirmed by experiments on 50 benchmark instances.

[Hif+14] introduced the first parallel algorithm (PLNSH) for the DCKP with a large neighborhood search heuristic (LNSH). The proposed PLNSH algorithm explores the neighborhoods simultaneously with 5 or 10 processors, where each processor applies a LNSH procedure to improve the current solution. Experimental results indicate that PLNSH is effective on most of the 50 instances tested.

[HSW15] presented a hybrid algorithm (HGNS) that combines the deterministic local search and the random local search. The deterministic local search improves

the current solution by alternating between a building procedure and an exploring procedure. The random local search diversifies the search by a modified ant colony optimization system. Extensive experiments disclose that the proposed HGNS algorithm is very efficient.

[Sal+17] designed a probabilistic tabu search algorithm (PTS) that operates with multiple neighborhoods. PTS starts from an initial solution obtained by a greedy procedure, and then it enters the tabu search procedure to explore different neighborhoods by using four types of candidate list strategies. Experimental evaluation on 50 benchmark instances demonstrates the effectiveness of the PTS algorithm.

[QW17a] devised a cooperative parallel adaptive neighborhood search algorithm (CPANS) which combines a cooperative algorithm (cooperative search stage) with a multi-neighborhood search procedure (individual search stage). The cooperative stage employs a team manager to record and share solutions, and a crowd of team members to explore the search space. The individual stage adopts a descent local search and an adaptive large neighborhood search to find local optimal solutions. Computational results on a set of 50 previous benchmark instances and a new set of 50 DCKP large instances with 1500 and 2000 items show that the proposed CPANS algorithm is highly competitive.

[QW17b] presented a parallel neighbor algorithm (PNS) for the DCKP, which is characterized by a random local search, a cooperation procedure, a tabu search procedure and an adaptive large neighborhood search procedure. The proposed PNS algorithm employs 10 to 400 processors to explore the search space and is able to achieve remarkable results on the 100 benchmark instances tested. Computational experience discloses that PNS performs better than the two linear programming solvers, i.e., CPLEX and GLPK.

According to the computational results reported in the literature, the parallel neighborhood search algorithm [QW17b], the cooperative parallel adaptive neighborhood search algorithm [QW17a], and the probabilistic tabu search algorithm [Sal+17] can be regarded as the state-of-the-art methods for the 100 instances of Set I (see Section 1.3.4). For the 6240 instances of Set II (see Section 1.3.4), the branch-and-bound algorithms presented in [BCM17; CFS21] and the integer linear programming formulations solved by the CPLEX solver [CFS21] showed the best performance.

1.3.4 Benchmarks

For the DCKP, the benchmark instances can be divided into two sets [2] as follows (see Tables 1.2 and 1.3 for the main characteristics of these instances).

— **Set I (100 instances)**: These instances are grouped into 20 classes (each with 5 instances) and named by xIy ($x = \{1, \ldots, 20\}$ and $y = \{1, \ldots, 5\}$). The first 50 instances ($1Iy$ to $10Iy$) were introduced in 2006 [HM06] and have the following features: number of items $n = 500$ or 1000, capacity $C = 1800$ or 2000, and density η going from 0.05 to 0.40. Note that the density is given by $2m/n(n-1)$, where m is the number of disjunctive constraints (i.e., the number of edges of the conflict graph). These instances have an item weight w_i uniformly distributed in $[1, 100]$ and a profit $p_i = w_i + 10$. For the instance classes $11Iy$ to $20Iy$ introduced in 2017 [QW17a], the number of items n is set to 1500 or 2000, the capacity C is set to 4000, and the density η ranges from 0.04 to 0.20. These instances have an item weight w_i uniformly distributed in $[1, 400]$ and a profit p_i equaling $w_i + 10$.

— **Set II (6240 instances)**: This set of instances was introduced in 2017 [BCM17] and expanded in 2020 [CFS21]. For the four correlated instance classes $C1$ to $C15$ (denoted by CC) and four random classes $R1$ to $R15$ (denoted by CR), the number of items n is from 60 to 1000, the capacity C is from 150 to 15000, and the density η is from 0.10 to 0.90. Each of these eight classes contains 720 instances. For the correlated instance class SC and the random instance class SR of the sparse graphs, the number of items n is from 500 to 1000, the capacity C is from 1000 to 2000, and the density η is from 0.001 to 0.05. Each of these two classes contains 240 DCKP instances. More details about this set of instances can be found in [CFS21].

1.4 Chapter conclusion

In this chapter, we presented a brief overview of the well-known knapsack problem and several common variants of the KP. We also introduced the two variants of the KP considered in this thesis and gave a number of applications related to these problems. Then we discussed the existing solution approaches for solving the SUKP and the DCKP, including exact algorithms, approximation algorithms, heuristic and metaheuristic algorithms. Finally, the benchmark instances tested in this thesis are given in the last section.

2. They are available at: http://www.info.univ-angers.fr/pub/hao/DCKP_TSBMA.html.

Table 1.2 – Summary of main characteristics of the 100 DCKP instances of Set I.

Class	Total	n	C	η	Class	Total	n	C	η
$1Iy$	5	500	1800	0.10	$11Iy$	5	1500	4000	0.04
$2Iy$	5	500	1800	0.20	$12Iy$	5	1500	4000	0.08
$3Iy$	5	500	1800	0.30	$13Iy$	5	1500	4000	0.12
$4Iy$	5	500	1800	0.40	$14Iy$	5	1500	4000	0.16
$5Iy$	5	1000	1800	0.05	$15Iy$	5	1500	4000	0.20
$6Iy$	5	1000	2000	0.06	$16Iy$	5	2000	4000	0.04
$7Iy$	5	1000	2000	0.07	$17Iy$	5	2000	4000	0.08
$8Iy$	5	1000	2000	0.08	$18Iy$	5	2000	4000	0.12
$9Iy$	5	1000	2000	0.09	$19Iy$	5	2000	4000	0.16
$10Iy$	5	1000	2000	0.10	$20Iy$	5	2000	4000	0.20

Table 1.3 – Summary of main characteristics of the 6240 DCKP instances of Set II.

Class	Total	n		C		η	
		Min	Max	Min	Max	Min	Max
$C1$	720	60	1000	150	1000	0.10	0.90
$C3$	720	60	1000	450	3000	0.10	0.90
$C10$	720	60	1000	1500	10000	0.10	0.90
$C15$	720	60	1000	15000	15000	0.10	0.90
$R1$	720	60	1000	150	1000	0.10	0.90
$R3$	720	60	1000	450	3000	0.10	0.90
$R10$	720	60	1000	1500	10000	0.10	0.90
$R15$	720	60	1000	15000	15000	0.10	0.90
SC	240	500	1000	1000	2000	0.001	0.05
SR	240	500	1000	1000	2000	0.001	0.05

Part II

Contributions

Chapter 2

Iterated two-phase local search for the set-union knapsack problem

In this chapter, we present an effective iterated two-phase local search algorithm for the SUKP. The proposed algorithm iterates through two complementary search phases: a local optima exploration phase to discover local optimal solutions, and a local optima escaping phase to drive the search to unexplored regions. We show the competitiveness of the algorithm compared to the state-of-the-art methods in the literature. Specifically, the algorithm discovers 18 improved best results (new lower bounds) for the 30 benchmark instances and matches the best-known results for the 12 remaining instances. We also report the first computational results with the general CPLEX solver, including 6 proven optimal solutions. Finally, we investigate the impacts of the key ingredients of the algorithm on its performance. The content of this chapter is based on an article published in *Future Generation Computer Systems*.

2.1 Introduction

Given its theoretical and practical significance, the SUKP has received more and more attention. As the review in Chapter 1.2.3 shows, various search methods have been proposed in the literature, including exact, approximation and metaheuristic algorithms. In particular, recent studies focused on metaheuristic algorithms which aim to find satisfactory solutions as fast as possible, without optimality guarantee of the attained solutions. These algorithms are especially useful to handle large and difficult problem instances when they cannot be solved by exact approaches. We observe that the state-of-the-art algorithms such as [FAG19; He+18; OB19] all adopted swam optimization metaheuristics. However, given that these methods are initially designed for solving continuous problems, the swam optimization based algorithms for the SUKP simulate discrete optimization via continuous search operators, instead of exploring the discrete space directly. As such, applying swam optimization to the SUKP requires various adaptations to cope with the binary feature of the SUKP. In particular, these algorithms must adopt an empirical transfer function to map the continuous space to the discrete space and maintain both continuous and discrete solutions during the search. Moreover, as indicated in [He+18], these approaches inevitably generate infeasible solutions, and therefore need a repairing procedure to handle these infeasible solutions.

In this chapter, we show for the first time that stochastic local search, which directly operates in the binary search space, can be a highly effective approach for solving the SUKP. The chapter is motivated by two considerations. First, stochastic local search has been quite successful in solving numerous challenging combinatorial problems [HS04], including several knapsack problems such as multidimensional knapsack problem [GK96; Lai+18a; VH01b], multidemand multidimensional knapsack problem [CT05; LHY19], multiple-choice multidimensional knapsack problem [CH14; HMS06], quadratic knapsack problem [CH17; YWC13], quadratic multiple knapsack problem [CH15; Pen+16] and generalized quadratic knapsack problem [AT17]. Second, given that the SUKP is basically a constrained subset selection problem with binary variables, it is natural to investigate solution methods that explore the binary search space and focus on feasible solutions. Indeed, as we show in this chapter, our discrete optimization approach based on stochastic local search is quite valuable for the SUKP.

The contributions of this chapter are summarized as follows.

— From a perspective of algorithm design, the proposed iterated two-phase local

search algorithm relies on two innovative and complementary search components specially designed for the SUKP. The intensification-oriented component (first phase) employs a combined neighborhood search strategy to discover local optimal solutions. The diversification-oriented component (second phase) helps the search process to explore unvisited regions. The combination of these two complementary search phases enables the algorithm to perform an effective examination of the search space.

— From a perspective of computational performance, we show the competitiveness of the proposed algorithm compared to the state-of-the-art algorithms on the set of 30 benchmark instances commonly used in the literature. In particular, we report improved best results for 18 large instances and equal best results for the 12 remaining instances. The improved best results (new lower bounds) are useful for future studies on the problem, e.g., they can serve as references for evaluating existing and new SUKP algorithms.

— Third, we investigate for the first time the interest of the general mixed integer programming solver CPLEX for solving the SUKP. We show that while CPLEX (version 12.8) can find the optimal solutions for the 6 small benchmark instances (with 85 to 100 items and elements) based on a simple 0/1 linear programming model, it fails to exactly solve the other 24 instances. These outcomes provide strong motivations for developing effective approximate algorithms to handle problem instances that cannot be solved exactly.

— This work demonstrates that the discrete optimization approach based on stochastic local search is quite valuable and effective for solving the SUKP. This work invites thus more investigations in this direction, in addition to the swarm optimization based approaches.

The remaining part of this chapter is organized as follows. In Section 2.2, we present the general framework of the proposed algorithm as well as its composing ingredients. Computational results and comparisons with the best-performing algorithms and CPLEX are reported in Section 2.3. In Section 2.4, we analyze the parameters and components of the algorithm and show their effects on its performance. In the last section, we summarize the present work and discuss future research directions.

2.2 Iterated two-phase local search for the SUKP

This section is dedicated to the presentation of the proposed iterated two-phase local search algorithm (I2PLS) for the SUKP. We first show its general scheme, and then explain the composing ingredients.

2.2.1 General algorithm

As shown in Algorithm 1, I2PLS is composed of two complementary search phases: a local optima exploration phase (Explore) to find new local optimal solutions of increasing quality and a local optima escaping phase (Escape) to displace the search to unexplored regions.

Algorithm 1 Iterated two-phase local search for the SUKP

1: **Input**: Instance I, cut-off time t_{max}, neighborhoods $N_1 - N_3$, exploration depth λ_{max}, sampling probability ρ, tabu search depth ω_{max}, perturbation strength η.
2: **Output**: The best solution found S^*.
3: /* Generate an initial solution S_0 in a greedy way, §2.2.3 */
$S_0 \leftarrow Greedy\_Initial\_Solution(I)$
4: $S^* \leftarrow S_0$ /* Record the overall best solution S^* found so far */
5: **while** $Time \leq t_{max}$ **do**
6: /* Local optima exploration phase using VND and TS, §2.2.4 */
$S_b \leftarrow$VND-TS$(S_0, N_1 - N_3, \lambda_{max}, \rho, \omega_{max})$
7: **if** $f(S_b) > f(S^*)$ **then**
8: $S^* \leftarrow S_b$ /* Update the best solution S^* found so far */
9: **end if**
10: /* Local optima escaping phase using frequency-based perturbation, §2.2.5 */
$S_0 \leftarrow$Frequency_Based_Local_Optima_Escaping(S_b, η)
11: **end while**
12: **return** S^*

The algorithm starts from a feasible initial solution (line 3, Alg. 1) that is obtained with a greedy construction procedure (Section 2.2.3). Then it enters the 'while' loop to iterate the 'Explore' phase and the 'Escape' phase (lines 5-11, Alg. 1) to seek solutions of improving quality. At each iteration, the 'Explore' phase (line 6, Alg. 1) first performs a variable neighborhood descent (VND) search to locate a new local optimal solution within two neighborhoods N_1 and N_2 and then runs a tabu search (TS) to explore additional local optima with a different neighborhood N_3 (Section 2.2.4). When the 'Explore' phase is exhausted, I2PLS switches to the 'Escape' phase (line 10, Alg. 1), which uses a frequency-based perturbation to displace the search to an unexplored region (Section 2.2.5). These two phases are iterated until a stopping condition (in our case, a given time limit t_{max}) is

reached. During the search process, the best solution found is recorded in S^* (lines 7-8, Alg. 1) and returned as the final output of the algorithm at the end of the algorithm.

One notices that the general scheme of the I2PLS algorithm for the SUKP shares ideas of breakout local search [BH13], three-phase local search [FH15] and iterated local search [LMS03]. Meanwhile, to ensure its effectiveness for solving the SUKP, the proposed algorithm integrates dedicated search components tailored for the considered problem, which are described below.

2.2.2 Solution representation, search space, and evaluation function

Given a SUKP instance composed of m items $V = \{1, \ldots, m\}$, n elements $U = \{1, \ldots, n\}$ and knapsack capacity C. The search space Ω includes all non-empty subsets of items such that the capacity constraint is satisfied, i.e., $\Omega = \{S \subset V : S \neq \emptyset, \sum_{j \in \cup_{i \in S} U_i} w_j \leq C\}$.

For any candidate solution S of Ω, its quality is assessed by the objective value $f(S) = \sum_{i \in S} p_i$ that corresponds to the total profit of the selected items.

Notice that a candidate solution S of Ω can be represented by $S =< A, \bar{A} >$ where A is the set of selected items and $\bar{A}$ are the non-selected items. Equivalently S can also be coded by a binary vector of length m where each binary variable corresponds to an item and its value indicates whether the item is selected or not selected.

The goal of our I2PLS algorithm is to find a solution $S \in \Omega$ with the objective value $f(S)$ as large as possible.

2.2.3 Initialization

The I2PLS algorithm starts its search with an initial solution, which is generated by a simple greedy procedure in three steps. First, we calculate the total weight w_i of each item i in $O(mn)$. Second, based on the given profit p_i of each item, we obtain the *profit ratio* r_i of each item by $r_i = p_i/w_i$ and sort all items in the descending order according to r_i in $O(log(m))$. Third, we add one by one the items to S by following this order until the capacity of the knapsack is reached in $O(m)$. The time complexity of the initialization procedure is thus $O(mn)$.

2.2.4 Local optima exploration phase

Algorithm 2 Local Optima Exploration Phase - VND-TS

1: **Input**: Starting solution S, neighborhoods $N_1 - N_3$, exploration depth λ_{max}, sampling probability ρ, tabu search depth ω_{max},
2: **Output**: The best solution S_b found by VND-TS.
3: $S_b \leftarrow S$ /*S_b records the best solution found so far during VND-TS */
4: $\lambda \leftarrow 0$ /*λ counts the number of consecutive non-improving rounds*/
5: **while** $\lambda < \lambda_{max}$ **do**
6: /* Attain a new local optimum S by VND with N_1 and N_2, see Alg. 3 */
$S \leftarrow$VND(S, N_1, N_2, ρ)
7: /* Explore nearby optima around the new S by TS with N_3, see Alg. 5 */
$(S_c, S) \leftarrow$TS(S, N_3, ω_{max}) /*S_c is the best solution found so far during TS */
8: **if** $f(S_c) > f(S_b)$ **then**
9: $S_b \leftarrow S_c$ /* Update the best solution S_b found so far */
10: $\lambda \leftarrow 0$
11: **else**
12: $\lambda \leftarrow \lambda + 1$
13: **end if**
14: **end while**
15: **return** S_b

From an initial solution, the 'Explore' phase (see Algorithm 2) aims to find new local optimal solutions of increasing quality. This is achieved by a combined strategy mixing a variable neighborhood descent (VND) procedure (line 6, Alg. 2, see Section 2.2.4) and a tabu search (TS) procedure (line 7, Alg. 2, see Section 2.2.4). For each VND-TS run (each 'while' iteration), the VND procedure exploits, with the best-improvement strategy, two neighborhoods N_1 and N_2 to locate a local optimal solution. Then from this solution, the TS procedure is triggered to examine additional local optimal solutions with another neighborhood N_3. At the end of TS, its best solution (S_c) is used to update the recorded best solution (S_b) found during the current VND-TS run, while its last solution (S) is used as the new starting point of the next iteration of the 'Explore' phase. The 'Explore' phase terminates when the best solution (S_b) found during this run cannot be updated during λ_{max} consecutive iterations (λ_{max} is a parameter called *exploration depth*).

Variable neighborhood descent search

Following the general variable neighborhood descent search [MH97], the VND procedure (Algorithm 3) relies on two neighborhoods (N_1 and N_2, see Sections 2.2.4) to explore the search space. Specifically, VND examines the neighborhood N_1 at first and iteratively identifies a best-improving neighbor solution in N_1 to replace the current solution. When

Algorithm 3 Variable Neighborhood Descent - VND

Input: Input solution S, neighborhoods N_1 and N_2, sampling probability ρ.
Output: The best solution S_b found during the VND search.
$S_b \leftarrow S$ /*S_b record the best solution found so far*/
$Improve \leftarrow True$
while $Improve$ **do**
  $S \leftarrow argmax\{f(S') : S' \in N_1(S)\}$
  if $f(S) > f(S_b)$ **then**
    $S_b \leftarrow S$ /*Update the best solution found so far*/
    $Improve = True$
  else
    $N_2^- \leftarrow Sampling(N_2, S, \rho)$
    $S \leftarrow argmax\{f(S') : S' \in N_2^-(S)\}$
    if $f(S) > f(S_b)$ **then**
      $S_b \leftarrow S$ /*Update the best solution found so far*/
      $Improve \leftarrow True$
    else
      $Improve = False$
    end if
  end if
end while
return S_b

a local optimal solution is reached within N_1, VND switches to the neighborhood N_2. As we explain in Section 2.2.4, given the large size of N_2, VND only examines a subset N_2^- which is composed of $\rho \times |N_2|$ randomly solutions of N_2 (ρ is a parameter called sampling probability and Algorithm 4 shows the sampling procedure where $random()$ is a random real number in [0,1]). If an improving neighbor solution is detected in N_2^-, VND switches back to N_1. VND terminates when no improving solution can be found within both neighborhoods. In Section 2.4.2, we study the influence of this sampling strategy.

Algorithm 4 Sampling Procedure

Input: Input solution S, neighborhood N_2, sampling probability ρ.
Output: Set N_2^- of sampled solutions from $N_2(S)$
$N_2^- \leftarrow \emptyset$
for each $S' \in N_2(S)$ **do**
  if $random() < \rho$ **then**
    $N_2^- \leftarrow N_2^- \cup \{S'\}$
  end if
end for
return N_2^-

Move operators, neighborhoods and VND exploration

To explore candidate solutions of the search space, the I2PLS algorithm employs the general *swap* operator to transform solutions. Specifically, let $S =< A, \bar{A} >$ be a given solution with A and $\bar{A}$ being the set of selected and non-selected items. Let $swap(q, p)$ denote the operation that deletes q items from A and adds p other items from $\bar{A}$ into A. By limiting q and p to specific values, we introduce two particular $swap(q, p)$ operators.

The first operator $swap_1(q, p)$ ($q \in \{0, 1\},\ p = 1$) includes two customary operations as described in the literature [LHY19; WH15; ZHG17]: the *Add* operator and the *Exchange* operator. Basically, $swap_1(q, p)$ either adds an item from $\bar{A}$ into A or exchanges one item in A with another item in $\bar{A}$ while keeping the capacity constraint satisfied.

The second operator $swap_2(q, p)$ ($3 \leq q + p \leq 4$) covers three different cases: delete two items from A and add one item from $\bar{A}$ into A; delete one item from A and add two items from $\bar{A}$ into A; exchanges two items of A against two items of $\bar{A}$. These three operations are subject to the capacity constraint.

On the basis of these two swap operators, we define the neighborhoods N_1^w and N_2^w induced by $swap_1$ and $swap_2$ as follows.

$$N_1^w(S) = \{S' : S' = S \oplus swap_1(q, p), q \in \{0, 1\},\ p = 1, \sum_{j \in \cup_{i \in S'} U_i} w_j \leq C\} \tag{2.1}$$

$$N_2^w(S) = \{S' : S' = S \oplus swap_2(q, p), 3 \leq q + p \leq 4, \sum_{j \in \cup_{i \in S'} U_i} w_j \leq C\} \tag{2.2}$$

where $S' = S \oplus swap_k(q, p)$ ($k = 1, 2$) is the neighbor solution of the incumbent solution S obtained by applying $swap_1(q, p)$ or $swap_2(q, p)$ to S.

N_1^w and N_2^w are bounded in size by $O(|A| \times |\bar{A}|)$ and $O(\binom{2}{|A|} \times \binom{2}{|\bar{A}|})$ respectively.

Given the large sizes of these neighborhoods, it is obvious that exploring all the neighbor solutions at each iteration will be very time consuming. To cope with this problem, we adopt the idea of a filtering strategy that excludes the non-promising neighbor solutions from consideration [LHY19]. Specifically, a neighbor solution S' qualifies as promising if $f(S') > f(S_b)$ holds, where S_b is the best solution found so far in Algorithm 3. Using this filtering strategy, we define the following reduced neighborhoods N_1 and N_2.

$$N_1(S) = \{S' \in N_1^w(S) : f(S') > f(S_b)\} \tag{2.3}$$

$$N_2(S) = \{S' \in N_2^w(S) : f(S') > f(S_b)\} \tag{2.4}$$

As explained in Section 2.2.4 and Algorithm 3, the VND procedure successively examines solutions of these two neighborhoods N_1 and N_2. Notice that $swap_2$ leads generally to a very large number of neighbor solutions such that even the reduced neighborhood N_2 can still be too large to be explored efficiently. For this reason, the VND procedure explores a sampled portion of N_2 at each iteration, according to the sampling procedure shown in Algorithm 4.

Tabu search

Algorithm 5 Tabu Search - TS

Input: Input solution S, Neighborhood N_3, tabu search depth ω_{max}
Output: The best solution S_b found during tabu search, the last solution S of tabu search.
$S_b \leftarrow S$ /*S_b records the best solution found so far*/
$\omega \leftarrow 0$ /*ω counts the number of consecutive non-improving iterations */
while $\omega < \omega_{max}$ **do**
  $S \leftarrow argmax\{f(S') : S' \in N_3(S)$ and S' is not forbidden by the $tabu\_list\}$
  if $f(S) > f(S_b)$ **then**
    $S_b \leftarrow S$ /* Update the best solution S_b found so far */
    $\omega \leftarrow 0$
  else
    $\omega \leftarrow \omega + 1$
  end if
  Update the $tabu\_list$
end while
return (S_b, S)

To discover still better solutions when the VND search terminates, we trigger the tabu search (TS) procedure (Algorithm 5) that is adapted from the general tabu search metaheuristic [GL97]. To explore candidate solutions, TS relies on the $swap_3(q, p)$ $(1 \leq p+q \leq 2)$ operator, which extends $swap_1$ used in VND by including the case $q = 1, p = 0$, which corresponds to the drop operation (i.e., deleting an item from A without adding any new item). One notices that $swap_3(1, 0)$ always leads to a neighbor solution of worse quality, which can be usefully selected for search diversification. We use N_3 to denote the neighborhood induced by $swap_3$.

$$N_3(S) = \{S' : S' = S \oplus swap_3(q, p), 1 \leq p + q \leq 2, \sum_{j \in \cup_{i \in S'} U_i} w_j \leq C\} \tag{2.5}$$

As shown Algorithm 5, the TS procedure iteratively makes transitions from the incumbent solution S to a selected neighbor solution S' in N_3. At each iteration, TS selects the best neighbor solution S' in N_3 (or one of the best ones if there are multiple best solutions) that is not forbidden by the so-called tabu list (*tabu_list*) (line 6, Alg. 5, see below). Notice that if no improving solution exists in $N_3(S)$, the selected neighbor solution S' is necessarily a worsening or equal-quality solution relative to S. It is this feature that allows TS to go beyond local optimal traps. To prevent the search from revisiting previously encountered solutions, the tabu list is used to record the items involved in the swap operation. And each item i of the tabu list is then forbidden to take part in any swap operation during the next T_i consecutive iterations where T_i is called the tabu tenure of item i and is empirically fixed as follows.

$$T_i = \begin{cases} 0.4 \times |A|, & \text{if } i \in A; \\ 0.2 \times |\bar{A}| \times (100/m), & \text{if } i \in \bar{A}. \end{cases} \tag{2.6}$$

TS terminates when its best solution cannot be further improved during ω_{max} consecutive iterations (ω_{max} is a parameter called the tabu search depth).

2.2.5 Frequency-based local optima escaping phase

The 'Explore' phase aims to diversify the search by exploring new search regions. For this purpose, the algorithm keeps track of the frequencies that each item has been displaced and uses the frequency information to modify (perturb) the incumbent solution. Particularly, we adopt an integer vector F of length m whose elements are initialized to zero. Each time an item i is displaced by a swap operation, F_i is increased by one. Thus, items with a low frequency are those that are not frequently moved during the 'Explore' phase. Then when the 'Explore' phase terminates and before the next round of the 'Explore' phase starts, we modify the best solution $S_b =< A_b, \bar{A}_b >$ as follows. We delete the top $\eta \times |A_b|$ least frequently moved items from A_b (η is a parameter called *perturbation strength* and adds to A_b randomly select items from $\bar{A}_b$ until the knapsack capacity is reached. This perturbed solution serves as the new starting solution S_0 of the next iteration of the algorithm (see line 10, Alg. 1). In Section 2.4.3, we study the usefulness of this perturbation strategy.

2.3 Experimental results and comparisons

This section presents a performance assessment of the I2PLS algorithm. We show computational results on the Set I of 30 benchmark instances (See Section 1.2.4) commonly used in the literature, in comparison with three state-of-the-art algorithms for the SUKP. We also present the first results from the CPLEX solver.

2.3.1 Experimental setting and reference algorithms

The proposed algorithm was implemented in C++ and compiled using the g++ compiler with the -O3 option. The experiments were carried on an Intel Xeon E5-2670 processor with 2.5 GHz and 2 GB RAM under the Linux operating system.

Table 2.1 – Settings of parameters.

Parameters	Sect.	Description	Value
λ_{max}	2	Exploration depth	2
ρ	2.2.4	Sampling probability for VND	5
ω_{max}	2.2.4	Tabu search depth	100
η	2.2.5	Perturbation strength in escaping phase	0.5

Table 2.1 shows the setting of parameters used in our algorithm, whose values were discussed in Section 2.4.1. Given the stochastic nature of the algorithm, we ran 100 times (like in [He+18; OB19]) with different random seeds to solve each instance, with a cut-off time of 500 seconds per run.

For the comparative studies, we use as reference algorithms the following three very recent algorithms: BABC (binary artificial bee colony algorithm) (2018), which is the best performing among five population-based algorithms tested in [He+18], gPSO (binary particle swarm optimization algorithm) (2019) [OB19] and MS (discrete moth search algorithm) (2019) [FAG19]. Among these reference algorithms, we obtained the code of BABC. So for BABC, we report both the results listed in [He+18] as well as the results obtained by running the BABC code on our computer under the same time limit of 500 seconds. For gPSO and MS, we cite the results reported in the corresponding papers. The results of these reference algorithms have been obtained on computing platforms with the following features: an Intel Core i5-3337u processor with 1.8 GHz and 4 GB RAM for BABC, an Intel Core i7-4790K 4.0 GHz processor with 32 GB RAM for gPSO, and an Intel Core i7-7500 processor with 2.90 GHz and 8.00 GB RAM for MS.

Additionally, we notice that until now, no result has been reported by using the general integer linear programming (ILP) approach to solve the SUKP. Therefore, we include in our experimental study the results achieved by the ILP CPLEX solver (version 12.8) under a time limit of 2 hours based on the 0/1 linear programming model presented in Section 1.2.1.

2.3.2 Computational results and comparisons

The computational results[1] of I2PLS on the three sets of benchmark instances are reported in Tables 2.2-2.4, together with the results of the reference algorithms (BABC [He+18], gPSO [OB19], MSO4 [FAG19]) where BABC* corresponds to the results by running the BABC code as explained in Section 2.3.1 and MSO4 is the best MS version among all twelve MS algorithms studied in [FAG19]. The first column of each table gives the name of each instance. Column 2 (Best_Known) indicates the best known value reported in the literature and compiled from [FAG19; He+18; OB19]. The best lower bound (LB) and upper bound (UB) achieved by the CPLEX solver are given in columns 3 and 4. Column 5 lists respectively the four performance indicators: best objective value (f_{best}), average objective value over 100 runs (f_{avg}), standard deviations over 100 runs (std), and average run times t_{avg} in seconds to reach the best objective value. Columns 6 to 9 present the computational statistics of the compared algorithms. The best values of f_{best} and f_{avg} among the results of the compared algorithms are highlighted in bold and the equal values are indicated in italic. Entries with "-" mean that the results are not available.

Given the fact that the compared algorithms were run on different computing platforms and they report solutions of various quality, it is not meaningful to compare the computation times. Therefore, the comparisons are mainly based on the quality, while run times (when they are available) are included only for indicative purposes.

Finally, Table 2.5 provides a summary of the compared algorithms on all 30 benchmark instances where rows $\#Better$, $\#Equal$ and $\#Worse$ indicate the number of instances for which each algorithm obtains a better, equal or worse f_{best} value compared to the best-known values in the literature (Best_Known). Moreover, to further analyze the performance of our I2PLS algorithm, we use the non-parametric Wilcoxon signed-rank test to check the statistical significance of the compared results between I2PLS and each reference

1. Our solution certificates and the code of I2PLS are available at: http://www.info.univ-angers.fr/pub/hao/SUKP_I2PLS.html.

algorithm in terms of f_{best} values. The outcomes of the Wilcoxon tests are shown in the last row of Table 2.5 where a *p-value* smaller than 0.05 implies a significant performance difference between I2PLS and its competitor.

From Tables 2.2 to 2.4, we observe that our I2PLS algorithm performs extremely well compared to the state-of-the-art results on the set of 30 benchmark instances. In particular, I2PLS improves on the best-known results of the literature for 18 out of 30 instances, while matching the best-known results for the remaining 12 instances. Notice that among these 12 instances, 6 instances with 85 and 100 items are solved to optimality by CPLEX (LB=UB), which are indeed not challenging for the other algorithms. Compared to the reference algorithms (BABC/BABC*, gPSO, MS), I2PLS reports better or equal f_{best} values for all the tested instances without exception. In terms of the average results (f_{avg}), I2PLS also performs very well by reporting better or equal f_{best} values for all instances except three cases (100_85_0.15_0.85, 100_100_0.15_0.85 and 85_100_0.15_0.85) for which BABC* has better values. Moreover, I2PLS has smaller standard deviations of its f_{best} values (f_{best} values often better than the compared results), suggesting that our algorithm is highly robust.

The small *p-values* (< 0.05) of Table 2.5 from the Wilcoxon signed-rank test (2.14e-4, 4.00e-6, 2.89e-5 and 1.43e-4) confirm that the results of our algorithm are significantly better than those of the compared results (best known in the literature, BABC, BABC* and gPSO).

Finally, we complete the above presentation by showing graphical comparisons of I2PLS against BABC, BABC*, and gPSO on the three sets of 30 instances. We ignore MS of [FAG19] since no result is available for half of the 30 instances. The plots of Fig. 2.1 concern the best and average objective values of the compared algorithms while the plots of Fig. 2.2 are based on the standard deviations. These figures clearly indicate the dominance of the proposed I2PLS algorithm over the reference algorithms in terms of the considered indicators.

2.4 Analysis and insights

In this section, we perform an analysis of the parameters and the ingredients of the algorithm to get useful insights about their impacts on its performance.

Table 2.2 – Computational results and comparison of the proposed I2PLS algorithm with the reference algorithms on the first set of instances ($m > n$).

Instance	Best_Known	LB	UB	Results	BABC	BABC*	gPSO	MSO4	I2PLS
100_85_0.10_0.75*	*13283*	*13283*	*13283*	f_{best}	13251	*13283*	*13283*	*13283*	*13283*
				f_{avg}	13208.5	*13283*	13050.53	13062	*13283*
				std	92.63	0	37.41	-	0
				t_{avg}	0.210	51.102	-	1.398	3.094
100_85_0.15_0.85*	12274	*12479*	*12479*	f_{best}	12238	*12479*	12274	-	*12479*
				f_{avg}	12155	**12479**	12084.82	-	12335.13
				std	53.29	0	95.38	-	98.78
				t_{avg}	0.223	24.032	-	-	103.757
200_185_0.10_0.75	*13521*	11585	27055.82	f_{best}	13241	13402	13405	*13521*	*13521*
				f_{avg}	13064.4	13260.16	13286.56	13193	**13521**
				std	99.57	38.98	93.18	-	0
				t_{avg}	1.562	253.693	-	7.901	71.984
200_185_0.15_0.85	14044	11017	29625.82	f_{best}	13829	*14215*	14044	-	*14215*
				f_{avg}	13359.2	14026.18	13492.60	-	**14031.28**
				std	234.99	151.55	328.72	-	131.46
				t_{avg}	1.729	241.932	-	-	180.809
300_285_0.10_0.75	11335	9028	43937.51	f_{best}	10428	10572	11335	11127	**11563**
				f_{avg}	9994.76	10466.45	10669.51	10302	**11562.02**
				std	154.03	61.94	227.85	-	3.94
				t_{avg}	5.281	315.240	-	24.912	181.248
300_285_0.15_0.85	12245	6889	53164.23	f_{best}	12012	12245	12245	-	**12607**
				f_{avg}	10902.9	12019.28	11607.10	-	**12364.55**
				std	449.45	85.76	477.80	-	83.03
				t_{avg}	5.673	226.818	-	-	240.333
400_385_0.10_0.75	*11484*	8993	66798.30	f_{best}	10766	11021	*11484*	11435	*11484*
				f_{avg}	10065.2	10608.91	10915.87	10411	**11484**
				std	241.45	138.07	367.75	-	0
				t_{avg}	12.976	293.560	-	56.838	31.801
400_385_0.15_0.85	10710	5179	77480.39	f_{best}	9649	9649	10710	-	**11209**
				f_{avg}	9135.98	9503.65	9864.55	-	**11157.26**
				std	151.90	94.69	315.38	-	87.29
				t_{avg}	13.359	270.813	-	-	141.525
500_485_0.10_0.75	11722	7202	86166.50	f_{best}	10784	10927	11722	11031	**11771**
				f_{avg}	10452.2	10628.31	11184.51	10716	**11729.76**
				std	114.35	70.31	322.98	-	6.59
				t_{avg}	25.372	486.210	-	124.378	349.545
500_485_0.15_0.85	10022	4762	97218.01	f_{best}	9090	9306	10022	-	**10238**
				f_{avg}	8857.89	9014.01	9299.56	-	**10133.94**
				std	94.55	64.06	277.62	-	94.72
				t_{avg}	26.874	482.740	-	-	369.375

Table 2.3 – Computational results and comparison of the proposed I2PLS algorithm with the reference algorithms on the second set of instances ($m = n$).

Instance	Best_Known	LB	UB	Results	BABC	BABC*	gPSO	MSO4	I2PLS
100_100_0.10_0.75*	*14044*	*14044*	*14044*	f_{best}	13860	*14044*	*14044*	*14044*	*14044*
				f_{avg}	13734.9	14040.87	13854.71	13649	**14044**
				std	70.76	11.51	96.23	-	0
				t_{avg}	0.213	169.848	-	1.646	38.245
100_100_0.15_0.85*	*13508*	*13508*	*13508*	f_{best}	*13508*	*13508*	*13508*	-	*13508*
				f_{avg}	13352.4	**13508**	13347.58	-	13451.50
				std	155.14	0	194.34	-	126.49
				t_{avg}	0.244	6.795	-	-	70.587
200_200_0.10_0.75	*12522*	11187	29394.32	f_{best}	11846	12350	*12522*	12350	*12522*
				f_{avg}	11194.3	11953.11	11898.73	11508	**12522**
				std	249.58	97.57	391.83	-	0
				t_{avg}	1.633	183.130	-	8.112	54.780
200_200_0.15_0.85	*12317*	9258	30610.99	f_{best}	11521	11929	*12317*	-	*12317*
				f_{avg}	10945	11695.21	11584.64	-	**12280.07**
				std	255.14	78.33	275.32	-	57.77
				t_{avg}	1.819	147.930	-	-	238.348
300_300_0.10_0.75	12736	11007	45191.75	f_{best}	12186	12304	12695	12598	**12817**
				f_{avg}	11945.8	12202.80	12411.27	11541	**12817**
				std	127.80	67.81	225.80	-	0
				t_{avg}	5.315	202.515	-	28.612	66.403
300_300_0.15_0.85	11425	7590	51891.53	f_{best}	10382	10857	11425	-	**11585**
				f_{avg}	9859.69	10383.64	10568.41	-	**11512.18**
				std	177.02	75.79	327.48	-	73.15
				t_{avg}	6.019	113.380	-	-	220.100
400_400_0.10_0.75	11531	7910	68137.98	f_{best}	10626	10869	11531	10727	**11665**
				f_{avg}	10101.1	10591.65	10958.96	10343	**11665**
				std	196.99	105.83	274.90	-	0
				t_{avg}	12.805	298.970	-	58.433	18.733
400_400_0.15_0.85	10927	4964	77719.78	f_{best}	9541	10048	10927	-	**11325**
				f_{avg}	9032.95	9602.13	9845.17	-	**11325**
				std	194.18	142.77	358.91	-	0
				t_{avg}	12.953	386.555	-	-	76.000
500_500_0.10_0.75	10888	7500	85184.48	f_{best}	10755	10755	10888	10355	**11249**
				f_{avg}	10328.5	10522.56	10681.46	9919	**11243.40**
				std	94.62	70.17	125.36	-	27.43
				t_{avg}	27.735	194.490	-	121.622	134.186
500_500_0.15_0.85	10194	3948	101964.36	f_{best}	9318	9601	10194	-	**10381**
				f_{avg}	9180.74	9334.52	9703.62	-	**10293.89**
				std	84.91	40.59	252.84	-	85.53
				t_{avg}	27.813	135.130	-	-	237.894

Table 2.4 – Computational results and comparison of the proposed I2PLS algorithm with the reference algorithms on the third set of instances ($m < n$).

Instance	Best_Known	LB	UB	Results	BABC	BABC*	gPSO	MSO4	I2PLS
85_100_0.10_0.75*	*12045*	*12045*	*12045*	f_{best}	11664	*12045*	*12045*	11735	*12045*
				f_{avg}	11182.7	11995.12	11486.95	11287	**12045**
				std	183.57	53.15	137.52	-	0
				t_{avg}	0.188	206.570	-	1.354	2.798
85_100_0.15_0.85*	*12369*	*12369*	*12369*	f_{best}	*12369*	*12369*	*12369*	-	*12369*
				f_{avg}	12081.6	**12369**	11994.36	-	12315.53
				std	193.79	0	436.81	-	62.60
				t_{avg}	0.217	0.531	-	-	17.47
185_200_0.10_0.75	*13696*	12264	25702.48	f_{best}	13047	13647	*13696*	13647	*13696*
				f_{avg}	12522.8	13179.14	13204.26	13000	**13695.60**
				std	201.35	100.78	366.56	-	3.68
				t_{avg}	1.502	202.560	-	7.642	124.136
185_200_0.15_0.85	11298	8608	26289.16	f_{best}	10602	10926	*11298*	-	*11298*
				f_{avg}	10150.6	10749.46	10801.41	-	**11276.17**
				std	152.91	97.24	205.76	-	83.78
				t_{avg}	1.948	259.050	-	-	139.865
285_300_0.10_0.75	11568	9421	44274.85	f_{best}	11158	11374	*11568*	11391	*11568*
				f_{avg}	10775.9	11143.69	11317.99	10816	**11568**
				std	116.80	76.90	182.82	-	0
				t_{avg}	5.450	426.680	-	24.539	25.128
285_300_0.15_0.85	11517	7634	51440.30	f_{best}	10528	10822	11517	-	**11802**
				f_{avg}	9897.92	10396.60	10899.20	-	**11790.43**
				std	186.53	128.63	300.36	-	27.51
				t_{avg}	5.571	192.575	-	-	206.422
385_400_0.10_0.75	10483	9591	59917.77	f_{best}	10085	10110	10483	9739	**10600**
				f_{avg}	9537.5	9926.18	10013.43	9240	**10536.53**
				std	184.62	87.43	202.40	-	56.08
				t_{avg}	13.012	203.870	-	57.000	234.475
385_400_0.15_0.85	10338	5810	73409.01	f_{best}	9456	9659	10338	-	**10506**
				f_{avg}	9090.03	9444.34	9524.98	-	**10502.64**
				std	156.69	46.40	286.16	-	23.52
				t_{avg}	13.724	177.910	-	-	129.505
485_500_0.10_0.75	11094	5940	84239.56	f_{best}	10823	10835	11094	10539	**11321**
				f_{avg}	10483.4	10789.57	10687.62	10190	**11306.47**
				std	228.34	27.29	168.06	-	36.00
				t_{avg}	27.227	299.260	-	114.066	207.118
485_500_0.15_0.85	10104	4325	100374.77	f_{best}	9333	9380	10104	-	**10220**
				f_{avg}	9085.57	9258.82	9383.28	-	**10179.45**
				std	115.62	58.72	241.01	-	46.97
				t_{avg}	28.493	49.170	-	-	238.630

Figure 2.1 – The best objective values (left) and mean objective values (right) of BABC, BABC*, gPSO and I2PLS for solving three sets of instances.

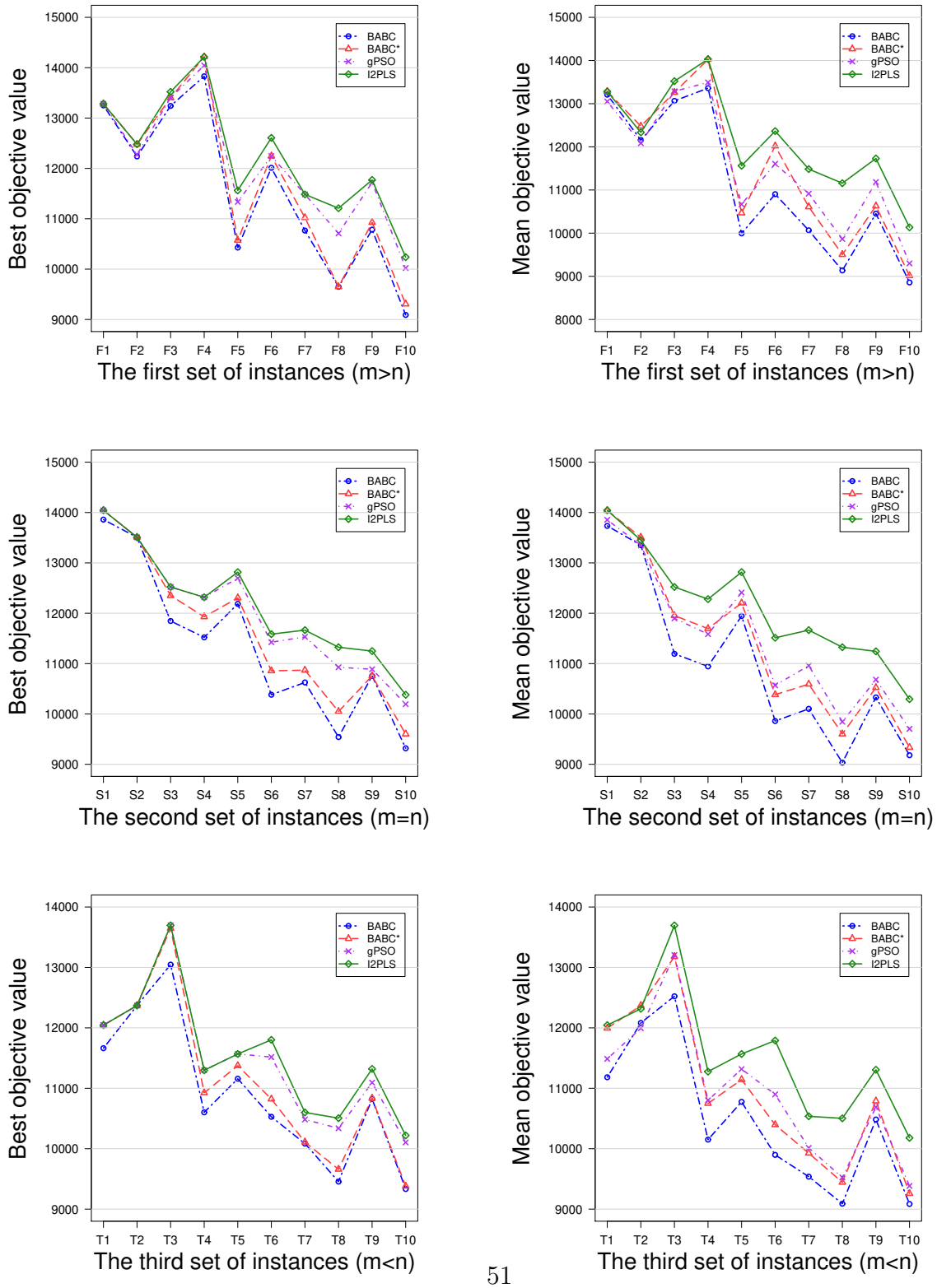


Table 2.5 – Summary of numbers of instances for which each algorithm reports a better, equal or worse f_{bst} value compared to the best-known value in the literature and *p-values* of the Wilcoxon singned-rank test on f_{best} values over all instances between I2PLS and each reference algorithm including the best-known values.

Instance	Best_Known	BABC	BABC*	gPSO	MSO4	I2PLS
# Better	-	0	2	0	0	18
# Equal	-	2	6	28	3	12
# Worse	-	28	22	2	12	0
p-value	2.14e-4	4.00e-6	2.89e-5	1.43e-4	2.52e-3	-

2.4.1 Analysis of parameters

As shown in Table 2.1, I2PLS requires four parameters: exploration depth λ_{max} (Section 2), neighborhood sampling probability ρ (Section 2.2.4), tabu search depth ω_{max} (Section 2.2.4), perturbation strength η (Section 2.2.5). To analyze the sensibility and tuning of the parameters, we select 8 out of the 30 benchmark instances, i.e., 185_200_0.15_0.85, 200_185_0.15_0.85, 200_200_0.15_0.85, 300_285_0.15_0.85, 400_385_0.15_0.85, 500_485_0.10_0.75, 500_485_0.15_0.85 and 500_500_0.15_0.85. According to Tables 2.2-2.4, the compared algorithms have a larger standard deviation for most of these instances than for other instances, implying that they are rather difficult to solve. We exclude the instances with 85 and 100 items since they can be solved exactly by the CPLEX and are thus too easy to be used for our analysis.

In this experiment, we studied each parameter independently by varying its value in a pre-determined range while fixing the other parameters to the default values shown in Table 2.1. We then ran I2PLS with each parameter setting 30 times to solve each of the 8 selected instances with the same cut-off time as in Section 2.3.2. Specifically, the exploration depth λ_{max} takes its values in $\{1, 2, \ldots, 10\}$ with a step size of 1, the sampling probability ρ varies from 0.01 to 0.10 with a step size of 0.01, the tabu search depth ω_{max} takes its values in $\{100, 200, \ldots, 1000\}$ with a step size of 100, and the perturbation strength η varies from 0.1 to 1 with a step size of 0.1. Fig. 2.3 shows the average of the best objective values (f_{best}) obtained by I2PLS with the four parameters on the 8 instances.

Fig. 2.3 indicates I2PLS achieves better results when $\lambda_{max} = 2, \rho = 0.05$ (the f_{avg} value is better when $\rho = 0.05$ than $\rho = 0.04$), $\omega_{max} = 100, \eta = 0.5$, respectively. This justifies the adopted settings of parameters as shown in Table 2.1. In addition, for each parameter, we used the non-parametric Friedman test to compare the f_{best} values reached with each

of the alternative parameter values. The resulting $p - value$ (> 0.05) of the parameters λ_{max} and ω_{max} show that the differences from alternative parameter settings are not statistically significant, implying that I2PLS is not sensitive to these two parameters.

2.4.2 Effectiveness of the variable neighborhood descent search strategy

The VND procedure explores two neighborhoods N_1 and N_2 with a sampling probability ρ applied to N_2. To investigate the impact of this sampling strategy, we performed an experiment by setting $\rho \in \{0.05, 0.0, 1.0\}$, where $\rho = 0.05$ is the adopted value as shown in Table 2.1, $\rho = 0.0$ indicates that only the neighborhood N_1 is used during the descent search while N_2 is disabled, and $\rho = 1.0$ indicates that the entire neighborhoods N_1 and N_2 are explored.

We denote these three VND variants by $\text{VND}_{0.05}$, $\text{VND}_{0.0}$ and $\text{VND}_{1.0}$ respectively. Recall that the VND procedure adopts the *best-improvement* strategy at each iteration. However, it is interesting to observe the effect of adopting the $first$-*improvement* strategy in N_2, So we included a fourth VND variant with the $first$-*improvement* strategy and $\rho = 1.0$ (denoted as $\text{VND}^{f}_{1.0}$). We ran these four VND variants to solve the 30 benchmark instances under the condition of Section 2.3.2 and report the results in terms of f_{best} in Table 2.6 (the best of the f_{best} values in bold). The rows $\#Better$, $\#Equal$ and $\#Worse$ respectively indicate the number of instances for which $\text{VND}_{0.0}$, $\text{VND}_{1.0}$ and $\text{VND}^{f}_{1.0}$ attain a better, equal and worse result compared to the result obtained by $\text{VND}_{0.05}$ (which is the default strategy of I2PLS).

Table 2.6 shows that $\text{VND}_{0.05}$ performs the best with the setting $\rho = 0.05$. Compared to $\text{VND}_{0.05}$, $\text{VND}_{0.0}$ obtains worse results on 3 instances, and equal results on the other 27 instances. $\text{VND}_{1.0}$ reaches the same results as $\text{VND}_{0.05}$ on 25 instances, and worse results on 5 instances. $\text{VND}_{1.0^f}$ obtains worse results on 4 instances, and equal results on the other 26 instances. Moreover, we observe that when exploring the whole neighborhood N_2, neither the *best-improvement* strategy nor the $first$-*improvement* strategy performs well. This can be explained by the fact that given the large size of N_2, a thorough examination of this neighborhood becomes very expensive. Within the cut-off time, the VND search cannot perform many iterations, decreasing its chance of encountering high-quality solutions. Finally, the $p - value$ of 4.18e-2 from the Friedman test indicates a significant difference among the compared VND strategies. This implies that the adopted VND strat-

egy and sampling technique of the I2PLS algorithm are relevant for its performance.

Table 2.6 – Influence of the VND search strategy on the performance of the I2PLS algorithm.

Instance/Setting	$\text{VND}_{0.05}$	$\text{VND}_{0.0}$	$\text{VND}_{1.0}$	$\text{VND}^{f}_{1.0}$
100_85_0.10_0.75	13283	13283	13283	13283
100_85_0.15_0.85	12479	12479	12479	12479
200_185_0.10_0.75	13521	13521	13521	13521
200_185_0.15_0.85	14215	14215	14215	14215
300_285_0.10_0.75	11563	11563	11563	11563
300_285_0.15_0.85	**12607**	12500	12332	12332
400_385_0.10_0.75	11484	11484	11484	11484
400_385_0.15_0.85	11209	11209	11209	11209
500_485_0.10_0.75	**11771**	11729	11746	11729
500_485_0.15_0.85	**10238**	10194	10194	10194
100_100_0.10_0.75	14044	14044	14044	14044
100_100_0.15_0.75	13508	13508	12238	13508
200_200_0.10_0.75	12522	12522	12522	12522
200_200_0.15_0.85	12317	12317	12317	12317
300_300_0.10_0.75	12817	12817	12817	12817
300_300_0.15_0.85	11585	11585	11502	11585
400_400_0.10_0.75	11665	11665	11665	11665
400_400_0.15_0.85	11325	11325	11325	11325
500_500_0.10_0.75	11249	11249	11249	11249
500_500_0.15_0.85	10381	10381	10381	10381
85_100_0.10_0.75	12045	12045	12045	12045
85_100_0.15_0.85	12369	12369	12369	12369
185_200_0.10_0.75	13696	13696	13696	13696
185_200_0.15_0.85	11298	11298	11298	11298
285_300_0.10_0.75	11568	11568	11568	11568
285_300_0.15_0.85	11802	11802	11802	11802
385_400_0.10_0.75	10600	10600	10600	10600
385_400_0.15_0.85	10506	10506	10506	10506
485_500_0.10_0.75	11321	11321	11321	11321
485_500_0.15_0.85	10220	10220	10220	10208
# Better	-	0	0	0
# Equal	-	27	25	26
# Worse	-	3	5	4

2.4.3 Effectiveness of the frequency-based local optima escaping strategy

The frequency-based local optima escaping strategy of I2PLS perturbs the locally best solution $S_b = (A, \bar{A})$ by replacing the first $\eta \times |A|$ (in I2PLS, η is set to 0.5) least frequently

Table 2.7 – Impact of the frequency-based local optima escaping strategy on the performance of the I2PLS algorithm.

Instance/Setting	I2PLS	I2PLS$_{random}$	I2PLS$_{strong}$
100_85_0.10_0.75	13283	13283	13283
100_85_0.15_0.85	12479	12479	12479
200_185_0.10_0.75	13521	13521	13521
200_185_0.15_0.85	14215	14215	14215
300_285_0.10_0.75	11563	11563	11563
300_285_0.15_0.85	12607	12607	12607
400_385_0.10_0.75	11484	11484	11484
400_385_0.15_0.85	11209	11209	11209
500_485_0.10_0.75	**11771**	11729	11729
500_485_0.15_0.85	**10238**	10194	10194
100_100_0.10_0.75	14044	14044	14044
100_100_0.15_0.75	13508	13508	13508
200_200_0.10_0.75	12522	12522	12522
200_200_0.15_0.85	12317	12317	12317
300_300_0.10_0.75	12817	12817	12817
300_300_0.15_0.85	11585	11585	11585
400_400_0.10_0.75	11665	11665	11665
400_400_0.15_0.85	11325	11325	11325
500_500_0.10_0.75	11249	11249	11249
500_500_0.15_0.85	10381	10381	10381
85_100_0.10_0.75	12045	12045	12045
85_100_0.15_0.85	12369	12369	12369
185_200_0.10_0.75	13696	13696	13696
185_200_0.15_0.85	11298	11298	11298
285_300_0.10_0.75	11568	11568	11568
285_300_0.15_0.85	11802	11802	11802
385_400_0.10_0.75	10600	10600	10600
385_400_0.15_0.85	10506	10506	10506
485_500_0.10_0.75	11321	11321	11321
485_500_0.15_0.85	10220	10220	10220
# Better	-	0	0
# Equal	-	28	28
# Worse	-	2	2

moved items of A with items that are randomly chosen from $\bar{A}$. In this experiment, we compared I2PLS against two variants with alternative perturbation strategies. In the first variant (denoted by I2PLS$_{random}$), we replace $0.5 \times |A|$ items randomly selected items of A while in the second variant (denoted by I2PLS$_{strong}$) and we perform a very strong perturbation by replacing all the items of A with items of $\bar{A}$ (i.e., setting η to 1). We ran I2PLS, I2PLS$_{random}$ and I2PLS$_{strong}$ 30 times to solve each of the 30 benchmark instances. The computational results of this experiment are shown in Table 2.7 where in addition to the best f_{best} values of each compared algorithm (the best of the f_{best} values in bold), the last three rows indicate the number of instances for which I2PLS$_{random}$ and I2PLS$_{strong}$ has a better, equal and worse result compared to that of I2PLS.

Table 2.7 shows that I2PLS with its frequency-based local optima escaping strategy performs slightly better than the two variants with alternative perturbation strategies. Indeed, even if the compared strategies lead to equal results for 28 instances, I2PLS achieves a better result on two of the most difficult instances (500_485_0.10_0.75 and 500_485_0.15_0.85). This experiment tends to indicate that the frequency-based local optima escaping strategy is particularly helpful for solving difficult instances. The $p-value$ of 1.35e-1 from the Friedman test indicates that the compared strategies differ only marginally.

2.5 Chapter conclusion

In this chapter, we introduce the first local search approach for solving the SUKP that directly operates in the discrete search space. The proposed algorithm combines a local optima exploration phase and a local optima escaping phase based on frequency information within the iterated local search framework.

The proposed algorithm has been tested on three sets of 30 benchmark instances commonly tested in the literature and showed a high competitive performance compared to the state-of-the-art SUKP algorithms. Specifically, our algorithm has improved on the best-known results (new lower bounds) for 18 out of the 30 benchmark instances, while matching the best-known results for the remaining 12 instances. Moreover, we have investigated for the first time the interest of the general mixed integer linear programming solver CPLEX for solving the SUKP, showing that the optimal solutions can be reached only for 6 small instances. Furthermore, we have analyzed the impacts of parameters and the main components of the algorithm on its performance.

In the next chapter, we will introduce a kernel based tabu search algorithm, which features original kernel-based search components and an effective local search procedure.

Figure 2.2 – The standard deviations of BABC, BABC*, gPSO and I2PLS for solving three sets of instances.

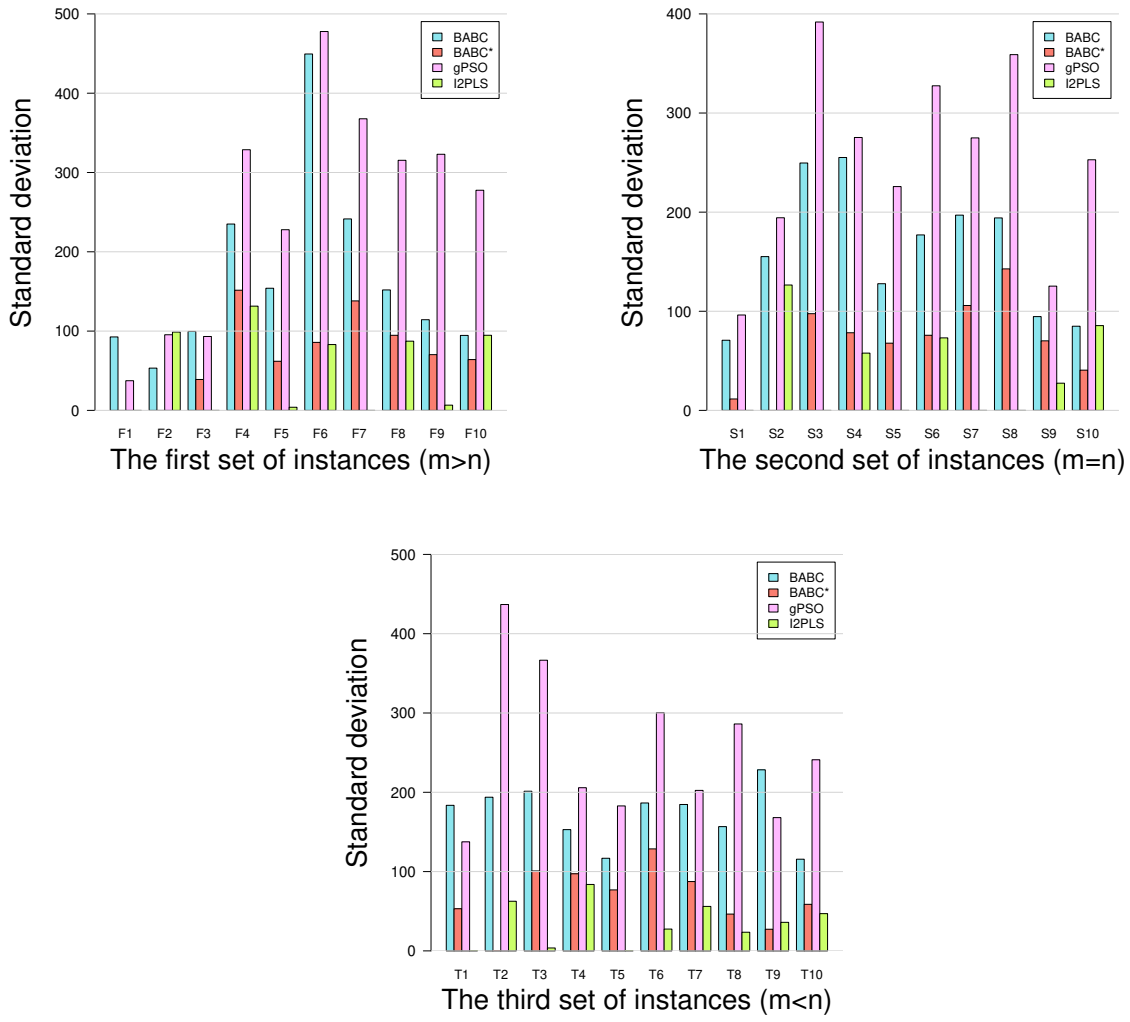
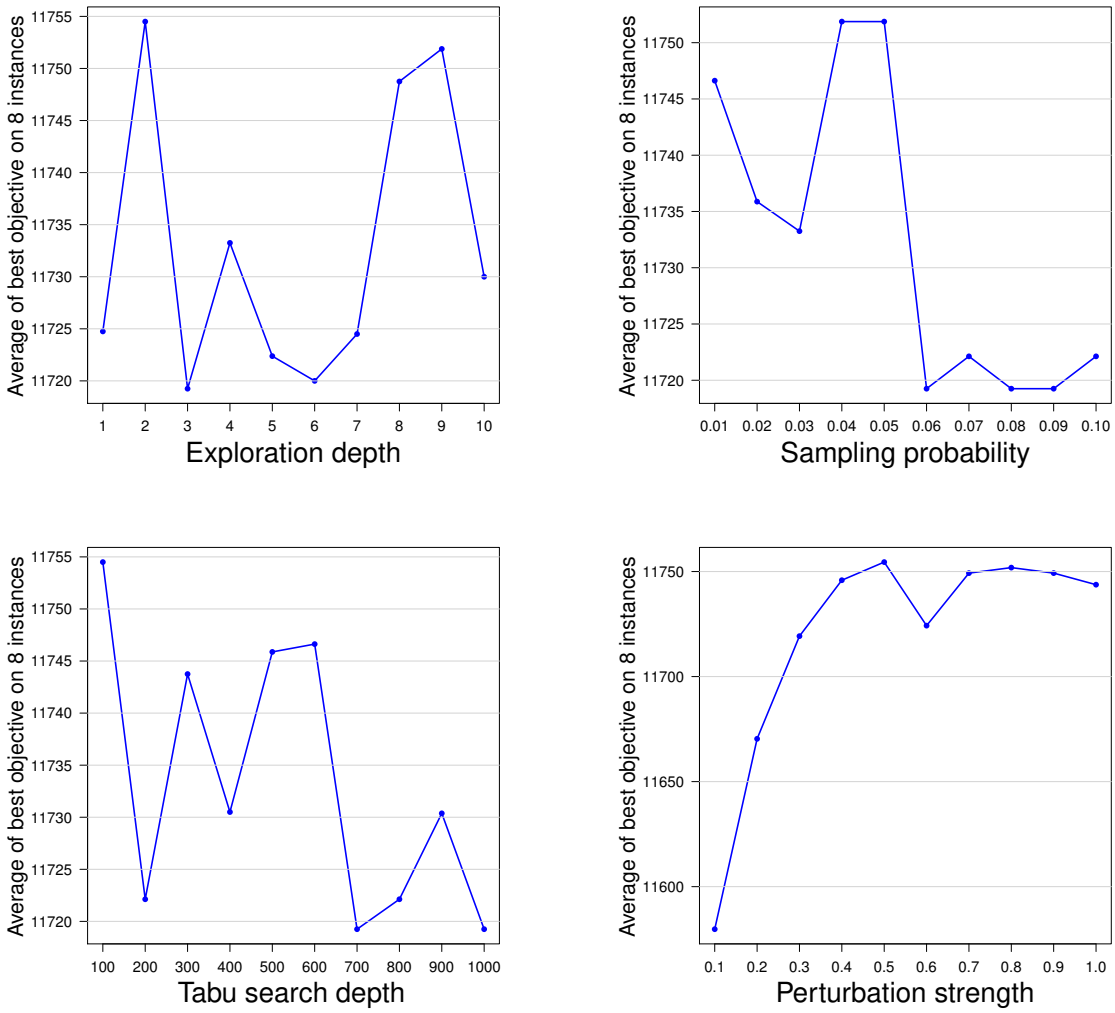


Figure 2.3 – Average of the best objective values (f_{best}) on the 8 instances obtained by executing I2PLS with different values of the four parameters.



Chapter 3

Kernel based tabu search for the set-union knapsack problem

In this chapter, we will introduce a competitive heuristic algorithm to advance the state-of-the-art for solving the SUKP. The proposed kernel based tabu search algorithm (KBTS) features original kernel-based search components and an effective local search procedure. Specifically, KBTS relies on a local search procedure to attain various local optima and a kernel search procedure to perform an additional exploration of promising search regions. Then, the non-kernel search procedure is employed to drive the search to a faraway new region. Extensive computational assessments on 60 benchmark instances demonstrate the high performance of the algorithm. We show different analyses to get insights into the influences of its algorithmic components. The content of this chapter is based on an article published in *Expert Systems with Applications*.

3.1 Introduction

The literature review (see Section 1.2.3) shows that the existing algorithms have a number of limitations. First, the performances of these algorithms lack stability and robustness (computational results with large standard deviations) even when solving small benchmark instances (with 85 to 100 items and elements). Second, their performances generally decrease when they are used to solve large instances (with at least 500 items and elements). Third, they consume a substantial amount of computation time to reach their reported results. Finally, most existing algorithms require a non-negligible number of parameters (e.g., 4 and 7 parameters for two leading algorithms I2PLS [WH19] and HBPSO/TS [Lin+19], respectively), making it difficult to control their performances and understand their behaviors.

In this chapter, we aim at advancing the state-of-the-art of solving the SUKP effectively and robustly in particular when large problem instances are considered. For this purpose, we investigate the first *kernel* based approach that overcomes the limitations mentioned above. This work is also motivated by another important consideration. In fact, the general idea of kernel has proved to be quite useful for several binary optimization problems (e.g., [VH01a; Wan+13; Zha04]). This work demonstrates for the first time its benefit for solving the SUKP, whose contributions are summarized as follows.

First, to evaluate the meaningfulness of the idea of kernel for solving the SUKP, we investigate the distribution of items among high-quality solutions. This investigation reveals the existence of kernels, which lays the basis for adopting the kernel concept to design our search algorithm. Indeed, the proposed kernel based tabu search algorithm (KBTS) integrates three complementary search components to perform an effective examination of the search space. That is, a local search procedure is used to find various local optima, a kernel search method is employed to discover additional high-quality solutions within particular areas, and a non-kernel search method is applied to ensure a guided diversification.

Second, we show the competitiveness of the proposed algorithm by comparing it with the state-of-the-art algorithms on 60 benchmark instances. We provide new lower bounds for several benchmark instances that can contribute to future research on the SUKP.

Third, we make the code of our KBTS algorithm publicly available, which can help researchers and practitioners to better solve various problems that can be formulated as the SUKP.

Finally, the kernel based search components of the proposed algorithm rely on general principals that can be advantageously adapted to other binary optimization problems.

The rest of the chapter is structured as follows. Section 3.2 presents the proposed algorithm as well as its components. Section 3.3 shows computational results and comparisons with the state-of-the-art algorithms. Section 3.4 shows several analyses to shed lights on the understanding of the key ingredients of the algorithm. Conclusions are provided in the last section.

3.2 Kernel based tabu search for the SUKP

In this section, we present the KBTS algorithm for solving the SUKP. We first present its main scheme and then describe its components.

3.2.1 Main scheme

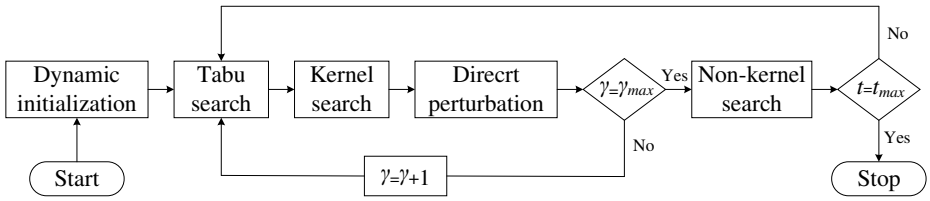


Figure 3.1 – Flow chart of the KBTS algorithm.

The KBTS algorithm follows the flow chart shown in Fig. 3.1 and is described in Algorithm 6.

The algorithm starts from a feasible initial solution generated by a *dynamic profit-ratio* mechanism (line 3, Alg. 6, and Section 3.2.3). Then it enters a 'while' loop to execute the main search process. Specifically, the input solution is improved by an iterative process (the 'repeat' loop), which includes a tabu search procedure, a kernel search procedure and a direct perturbation procedure. At each iteration of this process, the tabu search procedure (line 10, Alg. 6) is first invoked to obtain a high-quality solution with the neighborhood N_f (Section 3.2.4). During tabu search, a kernel solution (S_k) as well as a non-kernel solution ($\bar{S_k}$) are created using information from a frequency counter Φ. Then the kernel search procedure (line 11, Alg. 6, and Section 3.2.5) uses the neighborhood N_k to perform an intensified search around the kernel solution to seek other high-quality solutions. After that, the direct perturbation procedure (Section 3.2.6) is applied to modify

Algorithm 6 Kernel Based Tabu Search for the SUKP

Input: Instance I, cut-off time t_{max}, neighborhoods N_f, N_k, $\bar{N}_k$, *local search depth* γ_{max}, *kernel coefficient* ε, *direct perturbation strength* δ.
Output: The best solution found S^*.
$S \leftarrow$ Dynamic_Initialization(I) /* Generate an initial solution S, Sect. 3.2.3 */
$S^* \leftarrow S$ /* Record the overall best solution S^**/
while $Time \leq t_{max}$ **do**
  $\Phi \leftarrow$ Frequency_Initialization() /* Initialize frequency counter Φ to 0 */
  $S_b \leftarrow S$ /* Record the best solution S_b found so far */
  $\gamma \leftarrow 0$ /*γ counts the number of consecutive non-improving rounds*/
  repeat
    /* Record the local optimum S_l found by tabu search */
    $(S_l, S_k, \bar{S}_k) \leftarrow$ Tabu_Search $(S, N_f, \Phi, \varepsilon)$ /* Sect. 3.2.4 */
    $S_l \leftarrow$ Kernel_Search(S_k, S_l, N_k) /* Sect. 3.2.5 */
    $S \leftarrow$Direct_Perturbation(S_l, δ) /* Sect. 3.2.6 */
    if $f(S_l) > f(S_b)$ **then**
      $S_b \leftarrow S_l$ /* Update the local best solution S_b found so far */
      $\gamma \leftarrow 0$
    else
      $\gamma \leftarrow \gamma + 1$
    end if
  until $\gamma = \gamma_{max}$
  if $f(S_b) > f(S^*)$ **then**
    $S^* \leftarrow S_b$ /* Update the overall best solution S^* found so far */
  end if
  $S \leftarrow$ Non-Kernel_Search$(\bar{S}_k, \bar{N}_k)$ /* Sect. 3.2.7 */
end while
return S^*

the last local optimum found (controlled by the parameter δ), which is then used to start the next iteration of the process. This process ends when γ_{max} consecutive iterations are reached without further improving the local best solution S_b. At this point, the search is judged to be exhausted with the current search region and switches to the non-kernel search procedure (Section 3.2.7) to explore a distant and unexplored region. Finally, the whole algorithm terminates when the given time limit t_{max} is reached and returns the overall best solution S^* found during the search.

3.2.2 Solution representation, search space, and evaluation function

The search of the KBTS algorithm is limited to the feasible solution space Ω^F satisfying the knapsack constraint. By reference to the item set V with m items, a candidate solution S of Ω^F can be conveniently represented by $S = (y_1, \ldots, y_m)$ where each y_i is a binary

variable: $y_i = 1$ if item i is selected, $y_i = 0$ otherwise. A solution S can also be represented by $S =< A, \bar{A} >$ where $A \subseteq V$ is the set of selected items and $\bar{A} = V \setminus A$ is the set of the remaining items. The quality of S is measured by its objective value $f(S) = \sum_{i=1}^{m} p_i y_i$.

3.2.3 Dynamic initialization

The KBTS algorithm adopts an original initialization procedure using a *dynamic profit-ratio* of non-selected items. This procedure is based on the fact that for a given solution S, the weight of each element is counted only once. When a new item k is added to S, only the new elements of k that do not belong to the subset S will impact the total weight. Therefore, in our initialization procedure, the *profit-ratio* of non-selected items will be recalculated according to the elements belonging to the current solution S after adding a new item into S. The *dynamic profit-ratio* r_k^* of a non-selected item k is then given by $r_k^* = p_k / \sum_{j \in U_k \wedge j \notin \cup_{i \in S} U_i} w_j$.

From an empty subset S, the dynamic initialization procedure operates as follows. First, we calculate the *dynamic profit-ratio* r_k^* of non-selected items. Second, we identify the item k with the highest r_k^* value and add the item into S. We iterate these two steps until the knapsack constraint is reached.

Note that the *dynamic profit-ratio* refines the *static profit-ratio* used in [WH19] and generally leads to solutions of better quality.

3.2.4 Tabu search procedure

The KBTS algorithm adopts the well-known tabu search (TS) metaheuristic [GL97] to explore local optima within a restricted neighborhood. As a *general* search method, TS needs to be adequately adapted to the specific optimization problem under consideration. One notices that TS is quite successful to solve several knapsack problems (e.g., quadratic multiple knapsack [Qin+16], multidimensional knapsack [GK96; Lai+18a], set-union knapsack problem [Lin+19; WH19]) and other optimization problems (e.g., [Dıéa+17; LHG20]).

Our tabu search procedure is shown in algorithm 7, whose particular features tailored to the SUKP are discussed below. Given an input solution S, the TS procedure explores the neighborhood $N_f(S)$ induced by the *swap* operator (see Section 3.2.4) to make transitions from the current solution to neighbor solutions. Specifically, for each 'while' iteration (lines 5-11, Alg. 7), TS selects the best neighbor solution with the neighborhood search

Algorithm 7 Tabu Search

Input: Input solution S, neighborhood N_f, frequency counter Φ, *kernel coefficient* ε.
Output: Best solution S_l found during tabu search, kernel solution S_k, non-kernel solution $\bar{S_k}$.
$S_l \leftarrow S$ /* Record the best solution S_l found during tabu search */
$Continue \leftarrow True$
while $Continue$ **do**
  $(Continue, S) \leftarrow$ Neighborhood_Search(S, N_1, $Continue$) /* Algorithm 8 */
  if $f(S) > f(S_l)$ **then**
    $S_l \leftarrow S$ /* Update the best solution found during tabu search */
    $\Phi \leftarrow$ Update_Frequency(Φ)
  end if
end while
$S_k \leftarrow$ Create_Kernel(Φ, ε)
$\bar{S_k} \leftarrow$ Create_Non_Kernel(S_k)
return $(S_l, S_k, \bar{S_k})$

procedure, which is shown in Algorithm 8. If the new selected solution S is better than the best solution S_l found during tabu search, S_l is updated by S. Meanwhile, the frequency counter Φ_i of each selected item i in S is updated by $\Phi_i = \Phi_i + 1$, The main search ('while' loop) terminates when the neighborhood $N_f(S)$ becomes empty (see Algorithm 8). Then the kernel solution S_k and non-kernel solution $\bar{S_k}$ are created based on the frequency counter Φ, which will be presented in Sections 3.2.5 and 3.2.7.

Algorithm 8 Neighborhood Search

Input: Input solution S, flag $Continue$, neighborhood N.
Output: $Continue$, best solution S found.
Find admissible neighbor solutions $N(S)$
if $N(S) \neq \varnothing$ **then**
  $S \leftarrow argmax\{f(S') : S' \in N(S)\}$ /* Attain the best neighbor solution S */
  $Update\ tabu\_list$
  $Continue = True$
else
  $Continue = False$
end if
return $(Continue, S)$

Move operator and neighborhood structure

From the current solution, a neighbor solution is generated by applying the popular *swap* operator [WH19]. Specifically, given a solution $S =< A, \bar{A} >$ where $A \subseteq V$ is the set of selected items and $\bar{A} = V \setminus A$, a $swap(q, p)$ operation exchanges q items in A with p items in $\bar{A}$, leading to a neighbor solution designated by $S \oplus swap(q, p)$. Note

that q and p refer to the number of items involved in the *swap* operator. In our case, the candidate values for q and p are 0 or 1. Therefore, the *swap* operator includes three different operations: the *Add* operation with $q = 0$ and $p = 1$ (add one item from $\bar{A}$ into A), the *Delete* operation with $q = 1$ and $p = 0$ (delete one item from A) and the *Exchange* operation with $q = 1$ and $p = 1$ (exchange one item of A against one item of $\bar{A}$). Then the basic neighborhood induced by the *swap* operator includes all feasible solutions obtained by $S \oplus swap(q, p)$.

To enhance the computational efficiency of the KBTS algorithm, we define a restricted neighborhood by using a neighborhood filtering strategy [Lai+18a; WH19] to exclude unpromising neighbor solutions. With this strategy, only neighbor solutions S' of reasonable quality verifying $f(S') > f(S_b)$ are considered where S_b is the best solution found so far in the current tabu search run. Formally, the filter-based neighborhood $N_f(S)$ is defined as follows.

$$N_f(S) = \{S' : S' = S \oplus swap(q, p), q \in \{0, 1\}, p \in \{0, 1\}, f(S') > f(S_b)\} \tag{3.1}$$

Furthermore, to ensure the computational efficiency when evaluating a feasible neighbor solution, we adopt the so-called *gain updating* strategy [Lin+19; WH19]. Specifically, we use a vector G of length n where G_j ($G_j \in \{0, 1 \ldots, n\}$) records the number of appearances of element j in a solution S. Thus, only the elements that change values in G after performing $swap(q, p)$ will be considered when calculating the total weight of a new neighbor solution $S \oplus swap(q, p)$. That is, for each element j, if its G_j value changes from zero to non-zero, the total weight of the new solution is increased by w_j; if G_j changes from non-zero to zero, then total weight of the new solution is decreased by w_j. In other cases, the weight of the neighbor solution remains unchanged.

Tabu list management and aspiration criterion

Our TS procedure employs a tabu list to avoid revisiting previous encountered solutions. When a *swap* operation is performed, each item i involved in the swap is added in the tabu list and forbidden to move away from their respective item set for the next T_i consecutive iterations, where T_i is called the tabu tenure. Inspired by the tabu list management proposed in [VH01a], our tabu tenure T_i is set to the number of times item i is moved by the *swap* operation. As such, items with a high (low) move frequency will be forbidden for a longer (shorter) time. When no admissible move is available in the

neighborhood (i.e., $N_f(S) = \emptyset$), the TS procedure automatically stops.

During the tabu search, a best neighbor solution among those that are allowed by the tabu list is selected to replace the current solution. Notice that a neighbor solution is always selected if it is better than the best solution found during the TS procedure even if the solution is forbidden by the tabu list. This is the so-called *aspiration criterion* in tabu search [GL97].

3.2.5 Kernel search procedure

The tabu search procedure is able to explore different local optimal solutions with the help of the tabu list. Still, some interesting zones with better solutions may be overlooked. The kernel search procedure is introduced to perform an additional examination of particular regions identified by the so-called kernel solution.

Definition 1 *Let $\mathcal{S}$ be a set of feasible solutions, k an integer, and Φ_i the frequency of item i appearing in the solutions of $\mathcal{S}$, then the kernel solution (or simply kernel) S_k is the set of top k items with the highest frequencies such that $\Phi_i \geq \Phi_k$ and the total weight of S_k does not exceed the knapsack capacity.*

In the KBTS algorithm, we employ the frequency counter Φ_i to keep track of the number of times each item i appears in high-quality solutions. As mentioned in Section 3.2.4 (line 9, Alg. 7), each time a better solution is found during the tabu search procedure, the frequency counter Φ_i of the selected item i is updated by $\Phi_i = \Phi_i + 1$. Then at the end of the TS procedure, we generate the kernel S_k in two steps (line 12, Alg. 7). First, we sort all items in descending order according to the values of Φ. Second, we add the top $\varepsilon \times |S_l|$ most frequently appearing items to S_k, where ε is a parameter called *kernel coefficient* and $|S_l|$ is the number of the selected items in the best solution found during tabu search. Then S_k serves as the input solution S for the kernel search (KS) procedure shown in Algorithm 9.

The kernel search procedure shares the same framework with the TS procedure and employs the same neighborhood search procedure (see Algorithm 8), the same tabu list management and aspiration criterion. However, the KS procedure performs its search with the kernel based neighborhood $N_k(S)$ which is composed of neighbor solutions induced by the swap operator applied to the items of S excluding those of the kernel S_k. In other words, the items belonging to the kernel S_k remain fixed during the kernel search and

do not take part in any swap operation. By freezing the items of the kernel during the search, the KS procedure ensures a strongly intensified examination around the kernel.

The KS procedure ends if no admissible move is available in the kernel based neighborhood $N_k(S)$. At this point, the region around the kernel is considered to be sufficiently examined and the algorithm needs to move to a new region to continue its search. For this, we employ a direct perturbation strategy that is explained in the next section.

Algorithm 9 Kernel Search

Input: Input kernel solution S_k, attained local optimum S_l, neighborhood N_k.
Output: Best solution S_l during kernel search.
$S \leftarrow S_k$ /* Generate a new solution by S_k */
$Continue \leftarrow True$
while $Continue$ **do**
  $(Continue, S) \leftarrow$ Neighborhood_Search(S, N_k, $Continue$)
  if $f(S) > f(S_l)$ **then**
    $S_l \leftarrow S$ /* Update the best solution found during kernel search */
  end if
end while
return S_l

The kernel search procedure is inspired by the work presented in [VH01a] where the notion of kernel was introduced for solving a logic-constrained knapsack problem. The KS procedure is also related to the notion of *backbone* which was successfully applied to solve several binary optimization problems such as satisfiability [Zha04] and unconstrained binary quadratic programming [Wan+13]. This is the first application of this idea to the SUKP. Notice that given the particular feature of the SUKP, our way of defining (and identifying) kernels remains unique compared to previous studies.

3.2.6 Direct perturbation procedure

The direct perturbation procedure aims to diversify the TS-KS process, by modifying the input local optimum S_l to generate a new starting solution for the next round of the TS-KS process. Specifically, the perturbation performs δ random *swap(q,p)* ($q \in \{0,1\}$, $p \in \{0,1\}$, and excluding $swap(q,p)$ with $q = p = 0$) operations to transform the input solution while ensuring the feasibility of the resulting solution, where δ is a parameter called *direct perturbation strength*. It is clear that larger δ values lead to more important changes of the input solution.

3.2.7 Non-kernel search procedure

When the TS and KS procedures (lines 9-19, Alg. 6) terminate, we employ a global diversification strategy to definitively drive the search to a faraway new region. To identify this new region, we refer to the kernel solution $S_k = \{y_1, \ldots, y_m\}$ (described in Section 3.2.5) and define its opposite solution $\bar{S}_k = \{x_1, \ldots, x_m\}$ such that $x_i = 1 - y_i$ $(i = 1, \ldots, m)$. Then a feasible solution S is created from $\bar{S}_k$ and used as the input of the non-kernel search procedure. In order to obtain the feasible input solution S, we randomly select items from $\bar{S}_k$ and add them to S until the knapsack constraint is reached. The non-kernel search procedure follows the same search scheme (Algorithm 10) as TS and KS, but explores a different neighborhood $\bar{N}_k$ defined as follows. Specifically, during the non-kernel search, a swap operation is constrained to items that do not belong to the kernel S_k. In other words, items of S_k are never selected to become a part of a neighbor solution. As such, the non-kernel search has a strong diversification effect. The NKS procedure stops when the neighborhood becomes empty and the best solution found is used to initiate the next iteration of the whole KBTS algorithm.

Algorithm 10 Non-Kernel Search

```
1: Input: Input non-kernel solution S̄_k, neighborhood N̄_k.
2: Output: Best solution S_c found during non-kernel search.
3: S ← Random(S̄_k)                          /* Generate a feasible solution from S̄_k */
4: S_c ← S                                  /*S_c records the best solution found during non-kernel search */
5: Continue ← True
6: while Continue do
7:     (Continue, S) ← Neighborhood_Search(S, N̄_k, Continue)
8:     if f(S) > f(S_c) then
9:         S_c ← S                          /* Update the best solution found during non-kernel search */
10:    end if
11: end while
12: return S_c
```

3.2.8 Time complexity

We first consider the dynamic initialization procedure, which can be divided into two steps. The first step of updating *dynamic profit-ratio* can be achieved in $O(m^2n)$, and the second step of finding the non-selected item with the highest r_k^* value is bounded by $O(m^2)$, where m is the number of items and n is the number of elements. Thus the time complexity of the dynamic initialization procedure is $O(m^2n)$.

Now we evaluate one iteration of the main loop of the proposed algorithm. As shown in Algorithm 6, the tabu search procedure (TS), the kernel search procedure (KS) and the non-kernel search procedure (NKS) all adopt the Neighborhood_Search (NS) framework. Given the current solution $S =< A, \bar{A} >$ (see Section 3.2.4), the kernel solution S_k (see Section 3.2.5), and the non-kernel solution $\bar{S}_k$ (see Section 3.2.7), the corresponding complexity of one round of NS during the three procedures is $O([(m + |A| \times |\bar{A}|)] \times n)$, $O([(m-|S_k|)+(|A|-|S_k|)\times|\bar{A}|]\times n)$ and $O([|\bar{S}_k|+|A|\times(|\bar{S}_k|-|A|)]\times n)$. The complexity of the direct perturbation procedure is $O(1)$. Let R_{max} be the total maximum rounds of NS invoked by the TS, KS and NKS procedures. Then, the time complexity of one loop of KBTS is $O(m^2 n \times R_{max})$.

Let I_{max} be the maximum number of the iterations of the KBTS algorithm (which is determined by the cut-off time t_{max}). Then, the overall time complexity of KBTS is $O(m^2 n \times R_{max} \times I_{max})$. In Sections 3.3.2 and 3.4.4, we investigate the implications on the practical use of the above theoretical time complexity in terms of computational efficiency compared to existing SUKP algorithms.

3.2.9 Discussions

To highlight the novelties and contributions of the KBTS algorithm, we discuss below the main original features integrated in its search components.

First, the initialization procedure of Section 3.2.3 relies on an original *dynamic profit-ratio*. This strategy exploits the particular feature of SUKP that the elements of selected items can be reused regardless how many times they appear in the selected items of the current solution. The *dynamic profit-ratio* is thus a refined criterion compared to the *static profit-ratio* used in [WH19] and indeed favors the creation of high-quality initial solutions.

Second, the tabu search procedure of Section 3.2.4 has several special features that are different from other TS methods for the SUKP [Lin+19; WH19]. KBTS uses a parameter-free automatic tabu list strategy, while some parameters are required to control the tabu list and the tabu search termination in previous TS algorithms. Also, KBTS adopts an aspiration criterion to ensure that the best solution encountered is never overlooked, while no aspiration criterion is used in previous studies [Lin+19; WH19].

Third, although the general idea of *kernel* (or *backbone*) is known in the literature, we investigate for the first time the benefit of applying this idea to solve the SUKP and propose a new way of identifying and using the kernel with the KBTS algorithm. Specifically, we extract the most frequent items from a set of high-quality solutions and

use them to form a *kernel* solution (S_k). We additionally employ a parameter (*kernel coefficient*) to flexibly control the size of S_k within a proper range, which allows the kernel search procedure of Section 3.2.5 to intensively examine a given search region delimited by the kernel.

Fourth, the non-kernel search procedure of Section 3.2.7 relies on the opposite solution $\bar{S}_k$ of the kernel S_k. This is an original diversification strategy and has the advantage of diversifying the search in a guided manner. To our knowledge, such a strategy is not employed in the literature on the SUKP.

Finally, as we demonstrate in the next section, the KBTS algorithm equipped with these innovative features is able to compete very favorably with the current best algorithms for the SUKP in the literature.

3.3 Computational results and comparisons

This section is dedicated to an extensive evaluation of our KBTS algorithm and comparisons with state-of-the-art SUKP algorithms. We report computational results on two sets of 60 benchmark instances (see Section 1.2.4), available at `http://www.info.univ-angers.fr/pub/hao/SUKP_KBTS.html`.

3.3.1 Experimental protocol and reference algorithms

Computing platform. Our KBTS algorithm is programmed in C++ [1] and compiled with the g++ compiler with the -O3 option. To ensure a fair comparison, all the experiments mentioned in this work were performed on an Intel Xeon E5-2670 processor (2.5 GHz CPU and 2 GB RAM) running under the Linux operating system.

Parameter settings. The KBTS algorithm employs three parameters, whose descriptions and values are presented in Table 3.1. The effects and calibration of these parameters are presented in Section 3.4.1. The values of Table 3.1 can be considered to be the default setting and are used consistently to solve all 60 instances presented in Section 1.2.4 without any further fine-tuning.

Reference algorithms. We adopt three recent state-of-the-art algorithms: hybrid jaya algorithm (DHJaya) [WH20b], hybrid binary particle swarm optimization with tabu search (HBPSO/TS) [Lin+19] and iterated two-phase local search algorithm (I2PLS)

1. The code of our KBTS algorithm is available at: `http://www.info.univ-angers.fr/pub/hao/SUKP_KBTS.html`.

Table 3.1 – Parameters settings of KBTS.

Parameters	Section	Description	Value
γ_{max}	3.2.1	*local search depth*	3
ε	3.2.5	*kernel coefficient*	0.6
δ	3.2.6	*direct perturbation strength*	3

[WH19]. We also include the first binary artificial bee colony algorithm (BABC) [He+18] as a base reference. To ensure a fair comparison, we run the source codes of these algorithms (kindly provided by their authors) as well as our KBTS algorithm on our computing platform under the same stopping condition.

Stopping condition. Following [WH19], we run our KBTS algorithm and each reference algorithm to solve each of the 30 instances of Set I with a cut-off time of 500 seconds. For the 30 new large instances of Set II, the cut-off time is set to 1000 seconds. Given the stochastic nature of the compared algorithms, each instance is independently solved by each algorithm 100 times with different random seeds.

3.3.2 Computational results and comparisons

Tables 3.2 and 3.3 present the detailed computational results [2] of the compared algorithms achieved on the two sets of benchmark instances. Column 1 gives the names of the tested instances while the asterisk (*) indicates the optimal value that are proved by CPLEX and reported in [WH19]. The best objective value (f_{best}), the average objective value over 100 runs (f_{avg}), standard deviation over 100 runs (std) and the average run time (to reach the f_{best} value, denoted by t_{avg}) of each compared algorithm are reported in the remaining columns. In addition, the last row #Avg of Tables 3.2 and 3.3 indicates the average value of each column. Finally, dominating values of f_{best} and f_{avg} among the compared results are indicated in bold, and equal best values are shown in italic.

From the results of Table 3.2 on the instances of Set I, we observe that our KBTS algorithm is very competitive compared to the reference algorithms in terms of f_{best}, f_{avg} and std. Also, KBTS has a better average performance and very small standard deviations, indicating its high robustness. The high competitiveness of our KBTS algorithm becomes even more evident when we check the results of Table 3.3 for the 30 large instances of Set II. Indeed, KBTS dominates all the reference algorithms in all performance indicators.

2. Our solution certificates are available at: http://www.info.univ-angers.fr/pub/hao/SUKP_KBTS.html.

Moreover, KBTS requires less computation times to attain better solutions with small standard deviations, indicating its high computational efficiency and robustness.

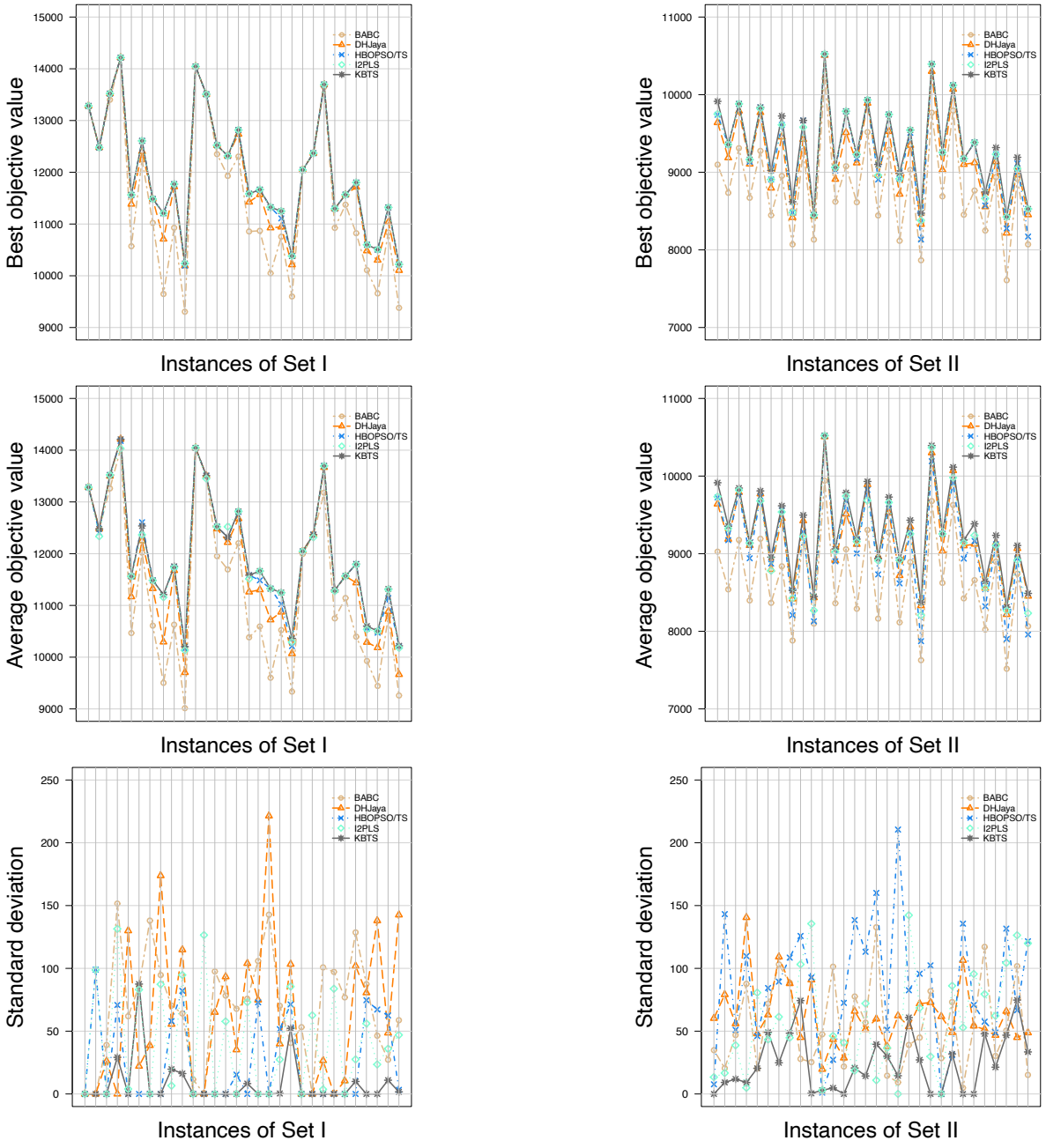


Figure 3.2 – Best objective values, average objective values and standard deviations of BABC, DHJaya, HBPSO/TS, I2PLS and KBTS on the 30 instances of Set I (left) and the 30 instances of Set II (right).

Fig. 3.2 additionally shows a graphical representation of the comparative results of

Table 3.2 – Computational results and comparison of the KBTS algorithm with the reference algorithms on the SUKP instances of Set I.

Instance	BABC				DHJaya				HBPSO/TS				I2PLS (Best_Known)				KBTS			
	f_{best}	f_{avg}	std	$t_{avg}(s)$	f_{best}	f_{avg}	std	$t_{avg}(s)$	f_{best}	f_{avg}	std	$t_{avg}(s)$	f_{best}	f_{avg}	std	$t_{avg}(s)$	f_{best}	f_{avg}	std	$t_{avg}(s)$
$F1^*$	*13283*	*13283*	0	51.102	*13283*	*13283*	0	9.477	*13283*	*13283*	0	0.098	*13283*	*13283*	0	3.094	*13283*	*13283*	0	4.082
$F2^*$	*12479*	*12479*	0	24.032	*12479*	*12479*	0	24.414	*12479*	12403.15	98.97	101.122	*12479*	12335.13	98.78	103.757	*12479*	*12479*	0	42.992
$F3$	13402	13260.16	38.98	253.693	*13521*	13498.22	26.10	258.213	*13521*	*13521*	0	0.490	*13521*	*13521*	0	71.984	*13521*	*13521*	0	6.988
$F4$	*14215*	14026.18	151.55	241.932	*14215*	**14215**	0	83.129	*14215*	14177.38	70.84	72.041	*14215*	14031.28	131.46	180.809	*14215*	14209.87	29.17	107.407
$F5$	10572	10466.45	61.94	315.240	11385	11167.77	129.98	174.335	*11563*	*11563*	0	38.355	*11563*	11562.02	3.94	181.248	*11563*	*11563*	0	28.841
$F6$	12245	12019.28	85.76	226.818	12402	12248.42	22.12	316.767	*12607*	**12607**	0	24.967	*12607*	12364.55	83.03	240.333	*12607*	12536.02	87.51	235.450
$F7$	11021	10608.91	138.07	293.560	*11484*	11325.88	38.65	229.370	*11484*	*11484*	0	10.870	*11484*	*11484*	0	31.801	*11484*	*11484*	0	0.296
$F8$	9649	9503.65	94.69	270.813	10710	10293.96	173.85	241.068	*11209*	*11209*	0	16.478	*11209*	11157.26	87.29	141.525	*11209*	*11209*	0	72.020
$F9$	10927	10628.31	70.31	486.210	11722	11675.51	55.53	226.604	*11771*	11746.19	57.98	293.514	*11771*	11729.76	66.59	349.545	*11771*	**11755.47**	19.74	206.199
$F10$	9306	9014.01	64.06	482.740	10194	9703.56	114.85	2383.021	10194	10163.76	82.11	92.121	*10238*	10133.94	94.72	369.375	*10238*	**10202.90**	16.25	293.140
$S1^*$	*14044*	14040.87	11.51	169.848	*14044*	*14044*	0	1.374	*14044*	*14044*	0	0.518	*14044*	*14044*	0	38.245	*14044*	*14044*	0	0.023
$S2^*$	*13508*	*13508*	0	6.795	*13508*	*13508*	0	1.572	*13508*	*13508*	0	2.923	*13508*	13451.50	126.49	70.587	*13508*	*13508*	0	33.403
$S3$	12350	11953.11	97.57	183.130	*12522*	12480.62	65.05	207.667	*12522*	*12522*	0	0.8125	*12522*	*12522*	0	54.780	*12522*	*12522*	0	48.206
$S4$	11929	11695.21	78.33	147.930	*12317*	12217.81	93.361	229.824	*12317*	*12317*	0	0.950	*12317*	12280.07	57.77	238.348	*12317*	*12317*	0	72.495
$S5$	12304	12202.80	67.81	202.515	12736	12676.78	35.20	241.774	*12817*	12806.44	15.39	29.074	*12817*	*12817*	0	66.403	*12817*	*12817*	0	74.247
$S6$	10857	10383.64	75.79	113.380	11425	11260.25	103.95	152.329	*11585*	**11585**	0	5.985	*11585*	11512.18	73.15	220.100	*11585*	11584.17	8.26	141.464
$S7$	10869	10591.65	105.83	298.970	11569	11301.56	74.88	322.143	*11665*	11484.20	72.95	45.025	*11665*	*11665*	0	18.733	*11665*	*11665*	0	64.126
$S8$	10048	9602.13	142.77	386.555	10927	10721.45	221.38	77.037	*11325*	*11325*	0	5.902	*11325*	*11325*	0	76.000	*11325*	*11325*	0	17.591
$S9$	10755	10522.56	70.17	194.490	10943	10871.22	39.93	41.383	11109	11026.24	51.62	340.958	*11249*	11243.40	27.43	134.186	*11249*	**11248.96**	0.40	146.040
$S10$	9601	9334.52	40.59	135.130	10214	10069.33	103.33	101.926	*10381*	10213.25	71.30	220.328	*10381*	10293.89	85.53	237.894	*10381*	**10362.63**	52.25	156.331
$T1^*$	*12045*	11995.12	53.15	206.570	*12045*	*12045*	0	17.199	*12045*	*12045*	0	0.056	*12045*	*12045*	0	2.798	*12045*	*12045*	0	0.075
$T2^*$	*12369*	*12369*	0	0.531	*12369*	*12369*	0	0.342	*12369*	*12369*	0	0.088	*12369*	12315.53	62.60	17.470	*12369*	*12369*	0	10.175
$T3$	13647	13179.14	100.78	202.560	*13696*	13667.63	26.56	244.205	*13696*	*13696*	0	0.489	*13696*	13695.60	3.68	124.136	*13696*	*13696*	0	5.851
$T4$	10926	10749.46	97.24	259.050	*11298*	*11298*	0	38.439	*11298*	*11298*	0	0.486	*11298*	11276.17	83.78	139.865	*11298*	*11298*	0	6.373
$T5$	11374	11143.69	76.90	426.680	*11568*	11563.80	10.41	203.874	*11568*	*11568*	0	13.630	*11568*	*11568*	0	25.128	*11568*	*11568*	0	30.618
$T6$	10822	10396.60	128.63	192.575	11714	11436.93	101.85	463.466	*11802*	**11802**	0	2.135	*11802*	11790.43	27.51	206.422	*11802*	11799.27	9.95	168.904
$T7$	10110	9926.18	87.43	203.870	10483	10287.36	80.61	53.459	*10600*	10552.73	74.68	100.155	*10600*	10536.53	56.08	234.475	*10600*	**10600**	0	73.087
$T8$	9659	9444.34	46.40	177.910	10302	10184.09	138.00	230.077	*10506*	10472.40	67.20	168.870	*10506*	10502.64	23.52	129.505	*10506*	**10506**	0	58.240
$T9$	10835	10789.57	27.29	299.260	11036	10883.19	48.58	66.029	*11321*	11142.27	62.51	223.387	*11321*	11306.47	36.00	207.118	*11321*	**11318.81**	10.95	121.494
$T10$	9380	9258.82	58.72	49.170	10104	9665.70	142.57	49.438	*10220*	10208.96	3.26	143.999	*10220*	10179.45	46.97	238.630	*10220*	**10219.76**	1.68	118.564
#Avg	11484.37	11279.18	69.08	216.769	11873.83	11748.07	61.56	156.332	11967.47	11938.10	24.29	65.194	*11973.60*	11932.39	40.54	138.476	*11973.60*	**11968.56**	7.87	78.16

Table 3.3 – Computational results and comparison of the KBTS algorithm with the reference algorithms on the SUKP instances of Set II.

Instance	BABC				DHJaya				HBPSO/TS				I2PLS				KBTS			
	f_{best}	f_{avg}	std	$t_{avg}(s)$	f_{best}	f_{avg}	std	$t_{avg}(s)$	f_{best}	f_{avg}	std	$t_{avg}(s)$	f_{best}	f_{avg}	std	$t_{avg}(s)$	f_{best}	f_{avg}	std	$t_{avg}(s)$
$F11$	9098	9026.05	34.87	498.591	9640	9449.97	60.22	690.489	9741	9724.60	7.68	576.260	9750	9734.74	13.39	479.356	**9914**	**9914**	0	209.679
$F12$	8736	8540.46	20.51	172.475	9187	8998.45	79.17	881.295	*9357*	9174.16	143.19	413.157	*9357*	9324.62	16.67	457.807	*9357*	**9354.52**	9.18	263.684
$F13$	9311	9176.28	46.93	363.381	9790	9602	55.96	543.236	*9881*	9792.23	51.06	881.999	*9881*	9819.24	38.74	363.945	*9881*	**9844.96**	11.88	455.713
$F14$	8671	8397.36	87.65	302.624	9106	8894.09	140.48	426.088	9135	8940.65	109.78	689.759	*9163*	9135.27	4.90	671.132	*9163*	**9138.36**	9.10	524.799
$F15$	9275	9192.36	20.27	253.268	9771	9540.08	47.95	637.331	*9837*	9736.89	46.11	777.755	9822	9678.89	80.67	719.986	*9837*	**9808.86**	20.42	483.384
$F16$	8447	8366.50	71.97	254.293	8797	8649	63.01	236.798	8907	8872.84	84.36	418.033	8907	8780.32	43.34	674.231	**9024**	**8955.29**	49.07	474.643
$F17$	8953	8837.18	103.15	471.428	9455	9249.53	109.14	687.150	9611	9560.93	89.43	514.922	9611	9537.61	61.42	511.245	**9725**	**9616.70**	24.85	609.811
$F18$	8072	7881.17	88.49	228.388	8418	8244.47	87.93	316.604	8481	8208.22	108.56	332.102	8481	8426.36	44.76	541.670	**8620**	**8526.55**	48.37	274.653
$F19$	9276	9254.19	27.89	640.529	9424	9306.86	45.01	309.873	*9668*	9278.50	125.80	620.436	9580	9221.23	103.18	329.743	*9668*	**9496.63**	74.35	487.925
$F20$	8133	8099.10	25.37	648.215	8433	8280.52	90.87	312.589	8448	8129.08	92.71	564.848	8448	8268.18	135.55	541.606	**8453**	**8448.05**	0.50	941.565
$S11$	10207	9939.38	47.52	66.660	10507	10504.25	19.67	321.196	10518	10517.89	1.09	60.254	*10524*	10520.70	2.99	513.537	*10524*	**10521.72**	2.91	404.697
$S12$	8621	8361.77	101.30	455.481	8910	8785.64	43.46	571.965	9024	8902.33	27.27	214.261	*9062*	9022.97	46.28	456.386	*9062*	**9061.16**	4.78	255.342
$S13$	9078	9056.52	21.89	224.370	9512	9409.01	28.70	809.836	*9786*	9679.56	72.51	215.910	*9786*	9742.73	40.87	383.700	*9786*	**9786**	0	97.316
$S14$	8614	8290.22	77.62	126.818	9121	8985.51	65.90	507.656	9177	9003.15	138.46	659.194	*9229*	9155.79	18.61	445.194	*9229*	**9187.55**	20.70	486.304
$S15$	9517	9305.40	56.76	418.476	9890	9656.38	51.42	567.090	*9932*	9823.17	113.20	607.506	*9932*	9685.79	72.06	868.227	*9932*	**9930.56**	14.33	214.286
$S16$	8444	8163.77	132.71	376.695	8961	8774.18	59.78	161.688	8907	8732.94	160.07	590.883	8961	8909.50	10.91	27.170	**9101**	**8936.12**	39.55	321.859
$S17$	9290	9272.99	14.56	460.026	9526	9462.86	37.83	670.990	*9745*	9639.60	51.13	598.520	*9745*	9660.12	36.68	341.110	*9745*	**9729.51**	30.06	368.807
$S18$	8118	8114.48	9.20	150.984	8718	8492.88	62.31	702.655	8916	8617.20	210.54	665.798	8916	8916	0	116.694	**8990**	**8918.96**	14.50	672.574
$S19$	9030	8891.34	39.01	657.972	9348	9250.80	53.65	542.187	9509	9273.64	82.57	802.652	*9544*	9255.73	142.33	876.669	*9544*	**9431.47**	60.84	510.660
$S20$	7867	7627.80	44.88	635.003	8330	8037.92	71.87	932.614	8134	7872.84	95.76	97.909	8379	8206.49	68.52	632.334	**8474**	**8376.20**	27.12	500.435
$T11$	9768	9677.80	81.90	535.874	10300	10161.45	72.81	98.186	*10393*	10191.01	102.35	729.422	*10393*	10366.15	29.83	499.311	*10393*	**10393**	0	89.785
$T12$	8689	8623.79	28.52	461.850	9031	8944.22	61.72	616.631	*9256*	*9256*	0	103.637	*9256*	*9256*	0	264.876	*9256*	*9256*	0	84.359
$T13$	9796	9627.40	73.18	248.733	10070	9953.55	49.02	430.180	*10121*	9909	30.82	123.012	*10121*	9979.70	86.13	540.289	*10121*	**10114.96**	31.87	230.918
$T14$	8453	8424.87	4.83	958.748	9102	8860.79	106.42	159.976	*9176*	8936.47	135.64	645.153	*9176*	9139.18	52.80	461.051	*9176*	**9176**	0	140.151
$T15$	8765	8658.45	54.33	869.031	9123	8885.09	54.14	316.494	*9384*	9163.90	70.91	339.415	*9384*	9236.10	95.56	576.738	*9384*	**9384**	0	136.173
$T16$	8249	8021.86	117.07	577.037	8556	8482.33	51.45	604.625	8572	8322.17	57.53	665.514	8663	8558.51	79.51	586.047	**8746**	**8643.93**	47.92	467.334
$T17$	8938	8897.58	30.23	587.200	9137	9079.09	46.70	590.376	9232	9121.24	48.92	455.104	9232	9106.31	62.28	452.360	**9318**	**9236.16**	21.32	281.632
$T18$	7610	7518.04	50.51	869.729	8217	7881.44	65.84	140.935	8277	7900.57	131.65	296.061	*8425*	8268	104.34	484.859	*8425*	**8311.68**	46.80	625.829
$T19$	8914	8741.25	101.76	739.861	9067	8994.48	44.99	313.094	9113	8938.38	66.64	967.315	9047	8917.48	126.37	89.760	**9193**	**9105.84**	74.76	319.356
$T20$	8071	8066.53	15.17	486.522	8453	8425.27	48.74	503.976	8172	7958.24	121.56	350.640	*8528*	8233.05	119.98	283.901	*8528*	**8488.13**	33.47	450.711
#Avg	8800.37	8668.40	54.33	458.009	9196.67	9041.40	62.54	482.096	9280.33	9105.91	85.91	499.248	9310.10	9202.09	57.96	473.031	**9352.30**	**9303.10**	23.95	379.479

the five competing algorithms on the two sets of instances in terms of the best objective values, the average objective values and the standard deviations. The X-axis in each sub-figure indicates the 30 instances of each set and the Y-axis gives the f_{best}, f_{avg} and *std* values of the compared algorithms. The plots of Fig. 3.2 clearly indicate the dominance of our KBTS algorithm over the reference algorithms and its particular advantage on the set of large instances.

Finally, Table 3.4 summarizes the comparative results between the KBTS algorithm and each reference algorithm. This table focuses on the f_{best} and f_{avg} indicators and shows the number of instances achieved by KBTS to obtain a better, an equal or a worse result (#Wins, #Ties and #Losses) compared to each reference algorithm. To verify the statistical significance of the comparisons of KBTS against the reference algorithms, the *p-values* from the non-parametric Wilcoxon signed-rank test are shown in the last column. And a *p-value* less than 0.05 implies a significant difference between KBTS and its competitor, while 'NA' means that the two sets of compared results are exactly the same. This summarized comparison clearly confirms the high performance of our KBTS algorithm. Indeed, for a majority of the tested instances, KBTS always reports better or equal results in terms of f_{best} and f_{avg}. Such a performance was never attained by any reference algorithm.

Table 3.4 – Summarized comparisons of the KBTS algorithm against each reference algorithm with the *p-values* of the Wilcoxon signed-rank test over the two sets of benchmark instances.

Algorithm pair	Instance set	Indicator	#Wins	#Ties	#Losses	*p-value*
KBTS vs. BABC	Set I (30)	f_{best}	23	7	0	2.70e-5
		f_{avg}	26	4	0	8.30e-6
	Set II (30)	f_{best}	30	0	0	1.73e-6
		f_{avg}	30	0	0	1.73e-6
KBTS vs. DHJaya	Set I (30)	f_{best}	16	14	0	4.38e-4
		f_{avg}	22	7	1	3.53e-5
	Set II (30)	f_{best}	30	0	0	1.73e-6
		f_{avg}	30	0	0	1.73e-6
KBTS vs. HBPSO/TS	Set I (30)	f_{best}	2	28	0	1.80e-1
		f_{avg}	12	15	3	7.60e-3
	Set II (30)	f_{best}	18	12	0	8.85e-5
		f_{avg}	29	1	0	2.56e-6
KBTS vs. I2PLS	Set I (30)	f_{best}	0	30	0	NA
		f_{avg}	20	10	0	1.51e-3
	Set II (30)	f_{best}	13	17	0	1.32e-4
		f_{avg}	29	1	0	2.56e-6

3.4 Analysis

In this section, we present an analysis of the parameters used in the proposed algorithm and the kernel based components.

3.4.1 Analysis of parameters

The proposed KBTS algorithm requires three parameters: *kernel coefficient* ε, *local search depth* γ_{max} and *direct perturbation strength* δ. We first carry out a factorial experiment [Mon17] to gain insights into the effect of parameters on the algorithm performance and then perform a one-at-a-time sensitivity analysis [Ham94] to calibrate the parameters. For these experiments, we select eight representative instances from Set II: 785_800_0.15_0.85, 800_785_0.15_0.85, 800_800_0.15_0.85, 885_900_0.15_0.85, 900_885_0.15_0.85, 985_1000 _0.10_0.75, 1000_985_0.10_0.75 and 1000_1000_0.10_0.75. These instances are difficult since the results reported by different algorithms (see Table 3.3) show large standard deviations.

Table 3.5 – Parameter levels for the 2-level full factorial experiment.

	Low level	High level
kernel coefficient ε	0.3	0.6
local search depth γ_{max}	3	6
direct perturbation strength δ	3	6

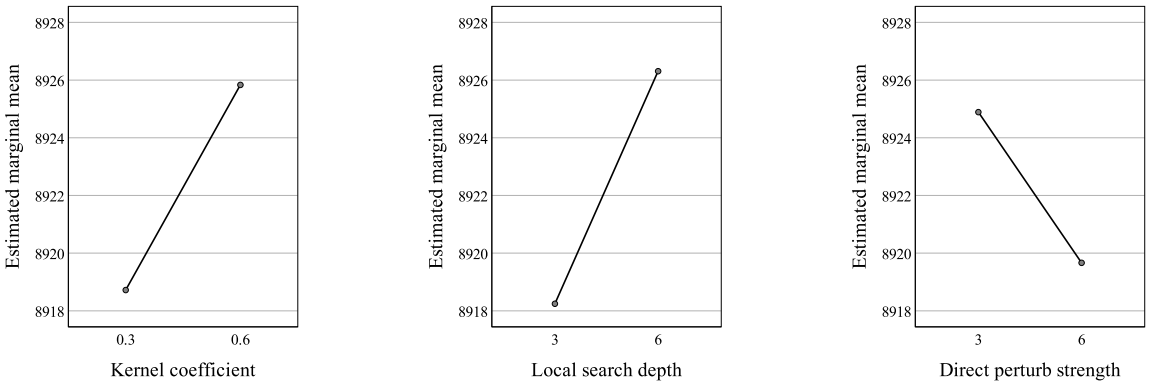


Figure 3.3 – Effects of the three parameters on the performance of the KBTS algorithm.

We employ a 2-level full factorial experiment to observe the interaction effects between the parameters. The levels of the three parameters are shown in Table 3.5. For this ex-

periment, each instance was independently solved 20 times with different combinations of parameters. Then we consider the average value of the best objective values (f_{best}) obtained on the eight instances for each parameter combination. We verify the normality of data distributions and the variance homogeneity. We show the main effects of the parameters in Fig. 3.3 and the analysis of the variances in Table 3.6.

Table 3.6 – *p-values* for the analysis of variances with the significance level 0.05.

Source of variation	ε	γ_{max}	δ	$\varepsilon * \gamma_{max}$	$\varepsilon * \delta$	$\gamma_{max} * \delta$	$\varepsilon * \gamma_{max} * \delta$
p-value	3.70e-2	1.80e-2	1.25e-1	3.90e-1	1.47e-1	1.92e-1	8.41e-1

From Fig. 3.3, we can observe that the effects of the parameter *kernel coefficient* and *local search depth* are positive, while the effect of direct perturbation strength is negative. The *p-values* (< 0.05) in columns 2-3 of Table 3.6 indicate that the performance of the algorithm is sensitive to the setting of *kernel coefficient* and *local search depth*. Moreover, it makes sense to check the interaction effects between the parameters. From Table 3.6, we can observe that the *p-values* of the last four columns are all greater than 0.05, which indicates that the interaction effects among the parameters are not statistically significant.

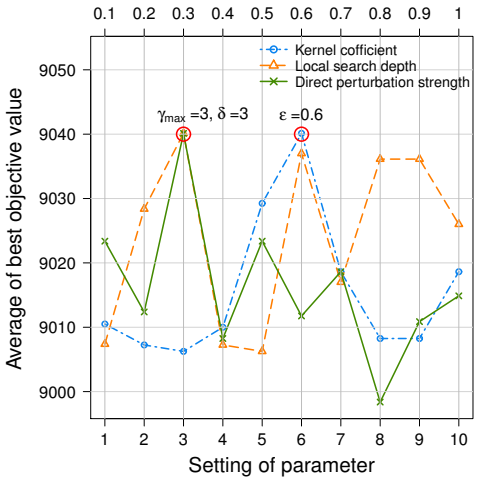


Figure 3.4 – Average of the best objective values (f_{best}) corresponding to different parameter settings obtained by the one-at-a-time sensitivity analysis.

Now we perform a one-at-a-time sensitivity analysis to determine a suitable value for each parameter. Based on a reasonable range of parameter values: $\varepsilon \in \{0.1, 0.2, ..., 1\}$, $\gamma_{max} \in \{1, 2, ..., 10\}$ and $\delta \in \{1, 2, ..., 10\}$, we test the values of each parameter independently while keeping the other parameters fixed to the values of Table 3.1. For this, we

run the algorithm with each parameter setting 30 times to solve each instance. Fig. 3.4 shows the average of the best objective values (f_{best}) attained by KBTS with different parameter settings. The X-axis indicates the ranges of the three parameters, i.e., 1 to 10 for γ_{max} and δ, 0.1 to 1 for ε. From Fig. 3.4, we observe that KBTS reaches its best performance with $\varepsilon = 0.6$, $\gamma_{max} = 3$ and $\delta = 3$. These values are thus used to define the default parameter setting shown in Table 3.1 of Section 3.3.1.

3.4.2 Impact of kernel search and non-kernel search

The proposed KBTS algorithm relies on the notion of kernel and the associated kernel search and non-kernel search procedures. To assess the usefulness of these components, we create a KBTS variant (denoted by KBTS$^-$) by disabling the kernel search procedure (i.e., removing line 11 in Alg. 1) and replacing the non-kernel search procedure with a random strategy (i.e., we generate randomly a feasible solution S of line 23 in Alg. 1). We run KBTS and KBTS$^-$ 30 times according to the experimental protocol given in Section 3.3.1 to solve each instance of Set II and report the results in Table 3.7. In this table, we show the f_{best}, f_{avg} and std values. The row #Avg indicates the average value of each column and the row #Best shows the number of instances for which an algorithm achieves the best results between the two set of results.

The results show that compared to KBTS, the KBTS$^-$ variant obtains worse f_{best} values for 7 instances, and worse f_{avg} values for 5 instances, leading to worse #Avg values of these performance indicators. Table 3.7 also indicates that KBTS$^-$ deteriorates the results of KBTS for the most difficult instances (with 785 to 1000 items and elements), which reveals that the kernel search procedure is particularly useful for solving difficult instances. Furthermore, the Wilcoxon signed-rank tests in terms of f_{best} ($p\text{-}value < 0.05$) confirm that the performance differences between KBTS and KBTS$^-$ are statistically significant.

3.4.3 Distribution of high-quality solutions and rationale of kernel search

To understand why the notion of kernel is pertinent, we present a study on distributions of items in high-quality solutions. This study is based on a selection of four representative instances: 500_485_0.15_0.85, 500_500_0.15_0.85, 1000_1000_0.10_0.75, 1000_1000_0.15_0.85. For each instance, we run KBTS 30 times to obtain 30 high-quality

Table 3.7 – Comparison between KBTS (with the kernel components) and KBTS$^-$ (without the kernel components) on the instances of Set II.

Instance/Setting	KBTS			KBTS$^-$		
	f_{best}	f_{avg}	std	f_{best}	f_{avg}	std
600_585_0.10_0.75	9914	9914	0	9914	9800.70	77.56
600_585_0.15_0.85	9357	9353.47	11.29	9357	9356.40	3.23
700_685_0.10_0.75	9881	9845	12	9881	9851.47	17.36
700_685_0.15_0.85	9163	9137.80	8.40	9163	9138.73	9.52
800_785_0.10_0.75	9837	9810.80	16.56	9829	9806.57	17.10
800_785_0.15_0.85	9024	8944	43.36	9024	8935.07	45.08
900_885_0.10_0.75	9725	9614.80	20.46	9725	9614.80	20.46
900_885_0.15_0.85	8620	8534.57	54.15	8588	8541.73	54.39
1000_985_0.10_0.75	9668	9512.13	74.70	9668	9477.40	56.68
1000_985_0.15_0.85	8448	8448	0	8448	8448	0
600_600_0.10_0.75	10524	10521.60	2.94	10524	10521.60	2.94
600_600_0.15_0.75	9062	9061.07	5.03	9062	9060.73	6.82
700_700_0.10_0.75	9786	9786	0	9786	9786	0
700_700_0.15_0.85	9229	9185.60	19.51	9177	9177	0
800_800_0.10_0.75	9932	9932	0	9932	9932	0
800_800_0.15_0.85	9101	8935.83	40.92	9101	8928.77	39.09
900_900_0.10_0.75	9745	9731.40	29.25	9745	9741.03	16.24
900_900_0.15_0.85	8990	8920.93	18.46	8916	8916	0
1000_1000_0.10_0.75	9544	9424	55.68	9544	9424.37	51.06
1000_1000_0.15_0.85	8474	8379.33	24.19	8438	8374.33	20.79
585_600_0.10_0.75	10393	10393	0	10393	10393	0
585_600_0.15_0.85	9256	9256	0	9256	9256	0
685_700_0.10_0.75	10121	10112.80	35.87	10121	10121	0
685_700_0.15_0.85	9176	9176	0	9176	9176	0
785_800_0.10_0.75	9384	9384	0	9384	9384	0
785_800_0.15_0.85	8746	8650.43	48.04	8663	8645.60	27.77
885_900_0.10_0.75	9318	9239.47	26.88	9318	9233.57	17.29
885_900_0.15_0.85	8425	8312.43	47.17	8425	8319.97	46.16
985_1000_0.10_0.75	9193	9086.07	77.58	9186	9083.90	69.38
985_1000_0.15_0.85	8528	8497.93	33.15	8528	8484.83	36.00
#Avg	**9352.13**	**9303.35**	23.52	9342.40	9297.69	21.16
#Best	**30**	**22**	-	23	17	-
p-value	-	-	-	1.80e-2	2.31e-1	-

solutions and then extract frequency statistics of selected items in these solutions, as shown in Fig. 3.5. The X-axis in each sub-figure indicates the number of selected items and the Y-axis refers to the frequency that one item appears in these solutions. We also present the number of items corresponding to each frequency on the right side of the Y-axis and the bottom value in this column corresponds to the number of items with a frequency of 0. Since this bottom value is much larger than the other values corresponding to the frequencies in the range $\{1, ..., 30\}$, we don't draw its corresponding plot for the convenience of observation.

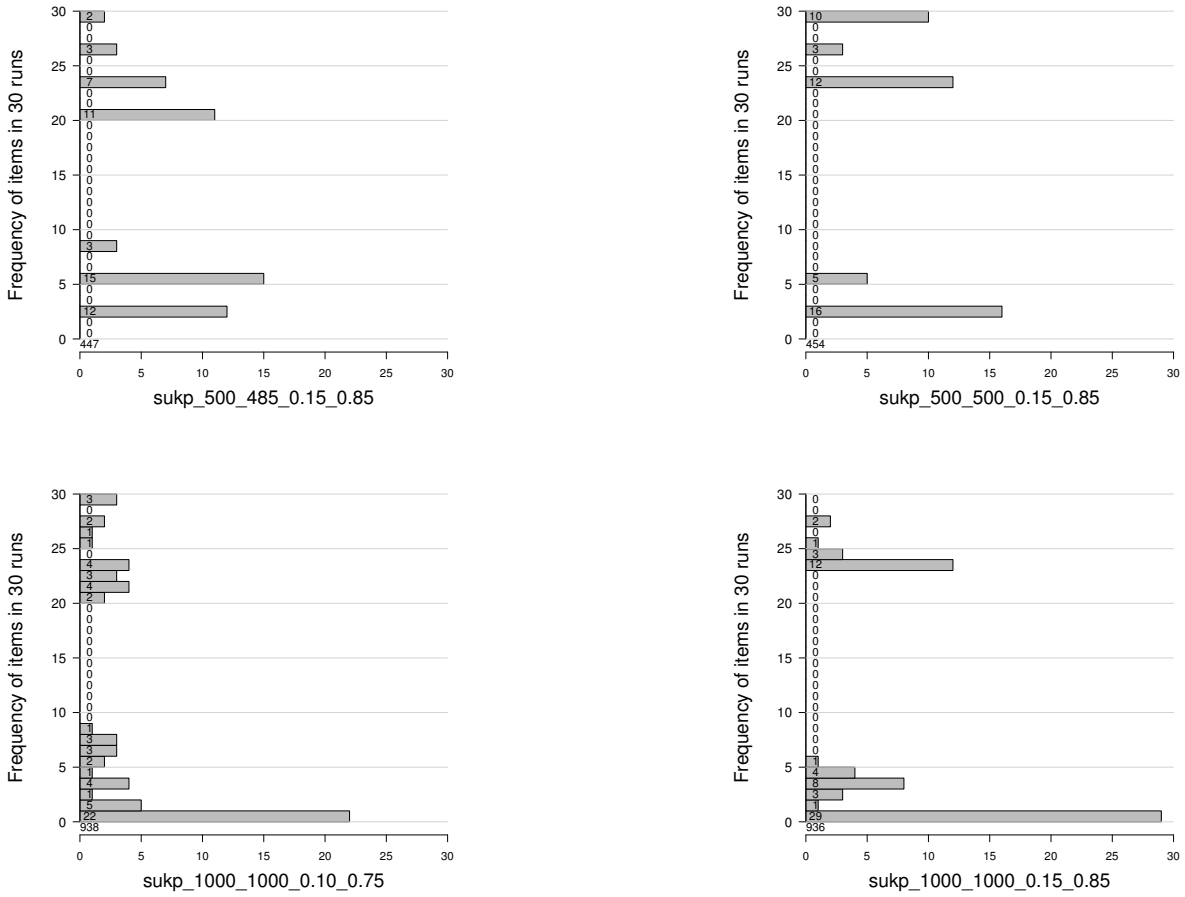


Figure 3.5 – Distributions of high-quality solutions corresponding to different item frequencies.

From Fig. 3.5, we observe that the frequency of most items being selected in a solution is polarized, that is, these items are either selected many times or are rarely selected. In particular, almost 90% of the items in each of these four instances never belong to a high-

quality solution. This experiment thus indicates that high-quality solutions often contain several identical items (which form a kernel), providing a supporting argument for the usefulness of the kernel based components of the KBTS algorithm.

3.4.4 Time-to-target analysis

To further assess the computational efficiency of the proposed KBTS algorithm with respect to the reference algorithms (BABC, DHJaya, HBPSO/TS, I2PLS, and KBTS), we present a time-to-target (TTT) analysis [ARR07; RRV12]. Basically, TTT shows the computation time required by an algorithm to attain a given target objective value. This analysis is based on four representative instances of Set II, i.e., 585_600_0.10_0.75, 600_600_0.15_0.85, 800_785_0.15_0.85, 1000_985_0.10_0.75. For each instance, we set the target value to be a value, which can be reached by all the compared algorithms (10000, 8800, 8700 and 9000, respectively) and record the time (over 100 runs) of each algorithm to reach a solution with an objective value at least as good as the given target value. The time-to-target plots are shown in Fig. 3.6, where the time required to achieve the target value and the corresponding cumulative probability are displayed on the X-axis and Y-axis, respectively.

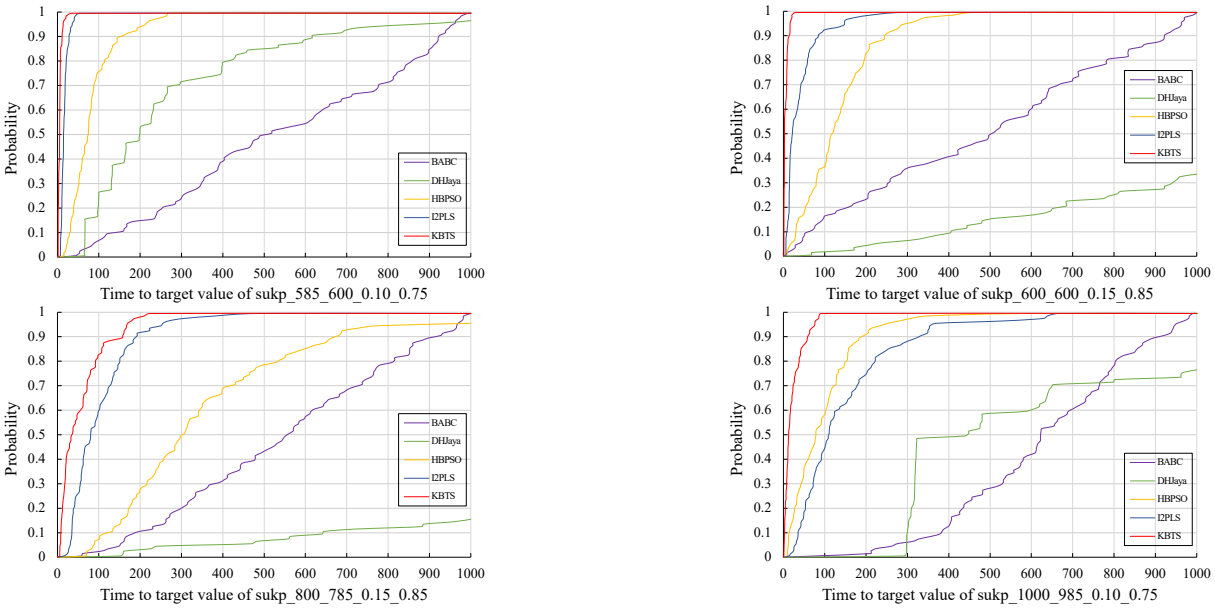


Figure 3.6 – Time-to-target plots of the compared algorithms on four SUKP instances.

From Fig. 3.6, we observe that our KBTS algorithm has a very high computational

efficiency, surpassing all the reference algorithms according to the cumulative probability. The lines of KBTS strictly runs above the lines of the reference algorithms, revealing that our algorithm has always a higher probability to reach the given target value.

3.5 Chapter conclusion

In this chapter, we presented the kernel based tabu search algorithm, which combines for the first time the notion of kernel with the powerful tabu search method. Our computational study performed on two sets of 60 benchmark instances indicated that the proposed algorithm dominates the current best SUKP algorithms in the literature in terms of solution quality, robustness and computation time. This dominance was particularly evidenced on large and difficult benchmark instances with at least 500 items and elements. Compared to the existing SUKP algorithms, the proposed algorithm requires only three parameters, making it more suitable to use in practice. Given that the SUKP has a number of interesting applications, the proposed algorithm provides a valuable tool for solving these real world problems. The availability of the source code of our algorithm and its high computational efficiency certainly facilitates such applications.

In the next chapter, we will carry on studying the SUKP and propose a multistart solution-based tabu search algorithm for solving the problem.

Chapter 4

Multistart solution-based tabu search for the set-union knapsack problem

In this chapter, we investigate for the first time a multistart solution-based tabu search algorithm for solving the problem. The proposed algorithm, which is parameter-free, combines a solution-based tabu search procedure with a multistart strategy to ensure an effective examination of candidate solutions. We report computational results on 60 benchmark instances from the literature, including new best results (improved lower bounds) for 7 large instances. We show additional experiments to shed lights on the roles of the key composing ingredients of the algorithm. The content of this chapter is based on an article published in *Applied Soft Computing*.

4.1 Introduction

The tabu search technology [GL97] has been successfully applied to solve many difficult optimization problems. Although most studies rely on the popular and well-known *attribute-based* tabu search (ABTS) as exemplified by the studies of [Lin+19; Lu+18; PB19; WH20a; Zho+20], recent studies indicated that the *solution-based* tabu search (SBTS) [CB96; WZ93] is a highly competitive approach for solving several notoriously difficult binary optimization problems such as 0/1 multidimensional knapsack [Lai+18a], multidemand multidimensional knapsack [LHY19], minimum differential dispersion [WWG17], and maximum min-sum dispersion [Lai+18b] and obnoxious p-median [Cha+21]. Compared to the ABTS method, SBTS has the advantage of avoiding the use of tabu tenure and simplifying the determination of tabu status. Moreover, the intensification ability of SBTS tends to be stronger than that of ABTS. In addition, the study reported in [LHY19] on SBTS and our study (see Section 4.4.3) reveal that SBTS is more suitable than ABTS for solving a number of binary optimization problems. However, SBTS requires more resources (to record all the encountered solutions) than ABTS. More information on the SBTS approach can be found in recent studies such as [Cha+21; LHY19; Lai+18b; WWG17], while some interesting studies using ABTS are provided in [Lu+18; NY19; PB19; SA21; Zho+20].

To the best of our knowledge, no study has been reported in the literature investigating the interest of the SBTS approach for solving the SUKP. In this work, we fill the gap by introducing the first multistart solution-based tabu search algorithm (MSBTS) for the SUKP and provide additional indications of the benefits of the SBTS approach for binary optimization. The main contributions of this work are summarized as follows.

First, the proposed MSBTS algorithm integrates a dedicated solution-based tabu search approach and a multistart mechanism to ensure an effective and efficient examination of candidate solutions. During the search, each visited solution is recorded in a tabu list implemented with the help of a hash function based method such that the tabu status of a candidate solution can be easily determined in constant time. The multistart mechanism is employed to escape local optima traps. The algorithm is simple in design and frees the user from the delicate task of calibrating parameters. Second, we report new best-known results (improved lower bounds) for 7 large instances, which are useful for future research on the SUKP. Third, we will make the code of our algorithm publicly available, which can be used by researchers and practitioners to solve various problems

that can be formulated by the SUKP model.

The rest of the paper is structured as follows. In Section 4.2, we describe the general solution approach of the proposed algorithm and its main components. Section 4.3 is devoted to the performance assessment and comparisons with state-of-the-art algorithms. We analyze in Section 4.4 the influences of important components of the algorithm, followed by conclusions in the last section.

4.2 Multistart solution-based tabu search for the SUKP

4.2.1 Search space, solution representation, and evaluation function

Given a SUKP instance composed of m items, n elements, and knapsack capacity C, the proposed MSBTS algorithm explores the feasible search space Ω^F which includes all feasible candidate solutions corresponding to non-empty subsets of items satisfying the knapsack constraint, i.e.,

$$\Omega^F = \{y \in \{0,1\}^m : \sum_{j \in U_i} w_j \leq C, U_i = \{i : y_i = 1\}, 1 \leq i \leq m, 1 \leq j \leq n\} \tag{4.1}$$

Thus, a candidate solution S in Ω^F can be expressed by a m-dimensional binary vector $S = (y_1, \ldots, y_m)$, where y_i takes 1 if item i is selected, and 0 otherwise. Let $A = \{q : y_q = 1 \; in \; S\}$ and $\bar{A} = \{p : y_p = 0 \; in \; S\}$, a candidate solution can be equivalently represented by $S =< A, \bar{A} >$.

Additionally, the quality of a candidate solution S is determined by the objective function value $f(S)$ (Equation 1.21) of SUKP. Since SUKP is a maximization problem, a larger f value indicates a better solution.

4.2.2 Main framework

The MSBTS algorithm follows the flow chart shown in Fig. 4.1 and is described in Algorithm 11.

The basic idea of the MSBTS algorithm (see Alg. 11) is to repeat a greedy randomized initialization procedure (Section 4.2.3) followed by a solution-based tabu search procedure

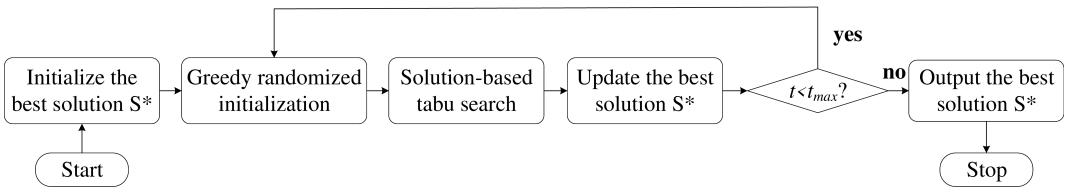


Figure 4.1 – Flow chart of the proposed MSBTS algorithm.

(Section 4.2.4). Specifically, after initializing the overall best solution S^* (line 3), the algorithm performs a 'while' loop (lines 4-10) to execute the main search process. At each round of this process, MSBTS first runs the greedy randomized procedure to generate a starting solution S (line 5), which is used as the input solution of the solution-based tabu search procedure. The solution-based tabu search procedure (line 6) iteratively improves the input solution S and returns the local best solution S_b encountered. After conditionally updating the overall best solution S^*, the algorithm moves to the next round of its search by re-starting the greedy randomized procedure. The main search process is terminated and returns the overall best solution S^*, when the cut-off time (t_{max}) is reached.

Algorithm 11 Multistart solution-based tabu search for the SUKP

1: **Input**: Instance I, cut-off time t_{max}, neighborhoods N, hash vectors H_1, H_2, H_3, length of hash vectors L, hash functions h_1 , h_2 , h_3.
2: **Output**: The best solution found S^*.
3: $S^* \leftarrow \emptyset$ /* Initialize the overall best solution S^* (i.e., $f(S^*) = 0$)*/
4: **while** $Time \leq t_{max}$ **do**
5: $S \leftarrow Greedy\_Randomized\_Initialization(I)$
6: /* Record the best solution S_b found during tabu search */
$S_b \leftarrow Solution\_Based\_Tabu\_search(S)$
7: **if** $f(S_b) > f(S^*)$ **then**
8: $S^* \leftarrow S_b$ /* Update the overall best solution S^* found so far */
9: **end if**
10: **end while**
11: **return** S^*

4.2.3 Greedy randomized initialization

The quality of initial solutions may impact the performance of the algorithm. In this work, we adopt a greedy randomized initialization procedure to generate initial solutions of good quality.

Let $W(S)$ be the total weight of the current solution S and W_k be the additional weight of a non-selected item k, where W_k is defined by $W_k = \sum_{j \in U_k \wedge j \notin \cup_{i \in S} U_i} w_j$. Then the

feasible non-selected items can be expressed by $R(x) = \{k \in \bar{A} : W_k + W(S) \leq C\}$, where $\bar{A}$ is the set of non-selected items. Following [CH17], we employ a restricted candidate list (denoted by RCL) to record rcl feasible non-selected items belonging to $R(x)$, where rcl is the maximum size of RCL. A too large rcl value will make many items to be recorded in RCL and thus result in an initial solution of poor quality, while a too small rcl value will limit the possible choices and lead to insufficient diversity of the initialization procedure. In our case, we set empirically $rcl = \sqrt{max\{m, n\}}$, where m and n are the number of items and elements respectively. Considering the fact that the number of items in $R(x)$ may be less than rcl, we finally set the size of RCL by $|RCL| = min\{rcl, |R(x)|\}$. Now, we build the restricted candidate list as follows. For each item k of $R(x)$, we calculate its dynamic profit ratio $r_k^* = p_k/W_k$. Then we identify the top $|RCL|$ items with the largest r^* values to form RCL. As the result, RCL contains the feasible non-selected items whose dynamic profit ratio is larger than the other non-selected items. Finally, each item k in RCL is selected with probability P_k, which is given by $P_k = r_k^* / \sum_{l=1}^{|RCL|} r_l^*$.

Algorithm 12 Greedy Randomized Initialization

```
1: Input: Instance I.
2: Output: The initial solution S.
3: /* Get the knapsack capacity C and restricted candidate list length rcl */
   (C, rcl) ← Read_instance(I)
4: W(S) ← 0                                   /* Initialize the total weight of S */
5: while W(S) ≤ C do
6:    Calculate additional weight W_k of each non-selected item k
7:    Add all items i with W_i = 0 into current solution S
8:    r* ← Calculate_dynamic_profit_ratio(W, |RCL|)
9:    P ← Calculate_probability(r*, |RCL|)
10:   S ← Add_one_item(P, S)
11: end while
12: return S
```

As shown in Algorithm 12, starting from an empty solution S, the initialization procedure randomly and adaptively adds feasible items k into S at each iteration of the 'while' loop (lines 5-11). Specifically, the initial solution is generated by four steps. First, we calculate the additional weight W_k of each non-selected item k (line 6), and add all items k with $W_k = 0$ into the current solution S, which means adding this item will not increase the total weight of S (lines 7). Second, we calculate the dynamic profit ratio r_k^* of each item k in $R(x)$ with $W_k \neq 0$ (line 8). Third, we calculate the selection probability P_k of each item k (line 9). Fourth, we randomly add one item from RCL into S according to P_k (line 10). These four steps are repeated until the knapsack capacity is reached.

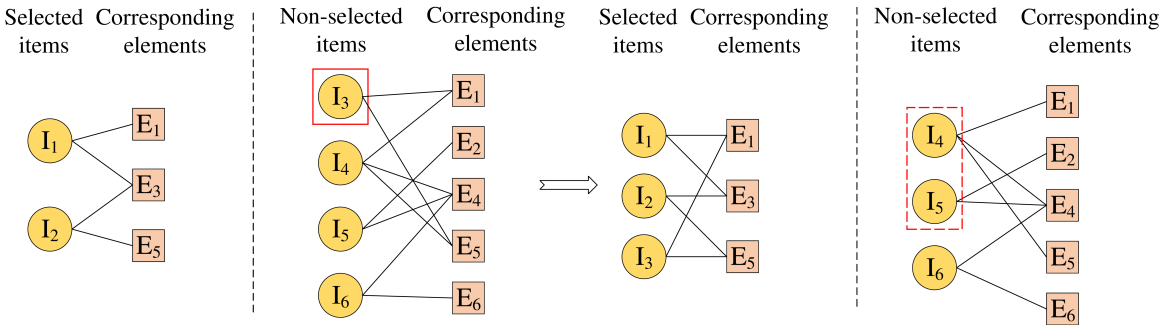


Figure 4.2 – An illustrative example of the main steps of the greedy randomized initialization procedure.

As shown in Fig. 4.2, we present a numerical example to illustrate the main steps of the greedy randomized initialization procedure. Given a set of six items (I_i, $i = 1, \ldots, 6$) with a profit of 1 to 6 respectively and a set of 6 elements (E_j, $j = 1, \ldots, 6$) with a weight of 1 to 6 respectively. Let the capacity of knapsack be equal to 16. At the step shown in the left figure, two items I_1 and I_2 are already added into the knapsack. We calculate additional weight W_i of each non-selected item i and find that $W_3 = 0$ (the elements E_1 and E_5 corresponding to item I_3 are already selected). Then we add the item I_3 into the knapsack and obtain the new solution shown in the right figure. Next, we calculate the dynamic profit ratio of the non-selected items and identify items I_4 and I_5 as belonging to RCL (in this case, $|RCL| = 2$). Finally, we add one of the two items into the knapsack according to the probability P_k.

4.2.4 Solution-based tabu search

Tabu search (TS) is a general and powerful metaheuristic for combinatorial optimization [GL97]. Typically, TS examines candidate solutions by iteratively transitioning from the current solution to a nearby (neighbor) solution by following a neighborhood. Each solution transition is performed by selecting the best admissible candidate among the neighboring solutions within the neighborhood. The key distinguishing feature of TS compared to other local optimization approaches is its tabu list strategy, which prevents the search from revisiting previously encountered solutions. With the so-called *solution-based* tabu search [CB96; WZ93], the tabu list is implemented with hash vectors and associated hash functions. Contrary to the popular *attribute-based* tabu search approach which typically needs some parameters for tabu list management, solution-based tabu search has the advantage of eliminating such parameters.

In the context of solving the SUKP, the best-performing algorithms are all based on the conventional attribute-based TS approach [Lin+19; WH19; WH20a]. This work adopts for the first time the solution-based tabu search approach for solving the SUKP, which leads to an effective algorithm while avoiding the difficulty of tuning parameters.

Algorithm 13 shows the general scheme of our solution-based tabu search (SBTS) procedure. After initializing the best solution found so far (line 3) and the associated hash vectors (i.e., tabu list, line 4), the SBTS procedure iteratively improves the current solution S (lines 6-20) until 1) no admissible neighboring solution (i.e., feasible and non-tabu neighboring solution) exists, or 2) the allowed cut-off time t_{max} is reached. Given the optimization function f, the neighborhood structure N (Section 4.2.4) and the tabu list management strategy (Section 4.2.4), the current solution S is replaced by a best admissible neighboring solution at each iteration of the SBTS procedure. And then the tabu list is updated with the newly obtained solution S. The best solution found during this procedure is recorded in S_b (lines 14-16) and returned as the output of SBTS. Note that the best admissible neighboring solution S is not necessarily better than S_b, but it will still be selected to replace the current solution S. In this way, the search can keep moving forward to discover better solutions without being trapped in local optima.

The SBTS procedure terminates under one of the two following conditions: (1) the overall cut-off time is reached; (2) no admissible neighboring solution can be found in the neighborhood, i.e., $N'(S) = \varnothing$ where $N'(S) \subseteq N(S)$ is the set of the admissible neighboring solutions not forbidden by the tabu list. Upon the termination of the SBTS procedure, two cases are considered: the overall cut-off time is reached and then the whole algorithm terminates. Otherwise, the algorithm re-starts its search by using the greedy randomized initialization procedure to creating a new starting solution, which is used to seed the next round of the SBTS procedure.

Next, we present the main ingredients of SBTS, including the move operator, the neighborhood structure and the tabu list strategy.

Move operator and neighborhood structure

Our SBTS procedure relies on two popular move operators, i.e., the *flip* operator and the *swap* operator to explore candidate solutions. Specifically, given a solution $S = (y_1, \ldots, y_m)$ as described in Section 4.2.1, the $flip(i)$ operator changes the value of a variable y_i to its opposite value $1 - y_i$. Similarly, given a solution $S =< A, \bar{A} >$, the $swap(q, p)$ operator exchanges one item in A against one item in $\bar{A}$, where q and p rep-

Algorithm 13 Solution-based tabu search

Input: Input solution S, neighborhood N, hash vectors H_1, H_2, H_3, hash functions h_1, h_2, h_3, cut-off time t_{max}, length of hash vectors L.
Output: Best solution S_b found during tabu search.
$S_b \leftarrow S$ /* Record the best solution S_b found during tabu search */
$(H_1, H_2, H_3) \leftarrow Initialize\_Hash\_Vectors(H_1, H_2, H_3, L)$ /* (i.e., tabu list) */
$Find \leftarrow True$ /* Track the admissible neighboring solution */
while $Find \wedge Time \leq t_{max}$ **do**
  Find admissible neighboring solutions $N'(S)$ in $N(S)$
  if $N'(S) \neq \varnothing$ **then**
    /* Attain the best admissible neighboring solution S */
    $S \leftarrow argmax\{f(S') : S' \in N'(S)\}$
    $Find \leftarrow True$
  else
    $Find \leftarrow False$
  end if
  if $f(S) > f(S_b)$ **then**
    $f(S_b) \leftarrow f(S)$ /* Update the best solution S_b found during tabu search */
  end if
  /* Update the hash vectors with S */
  $H_1[h_1(S)] \leftarrow 1$
  $H_2[h_2(S)] \leftarrow 1$
  $H_3[h_3(S)] \leftarrow 1$
end while
return S_b

resent items in sets A and $\bar{A}$ respectively. Meanwhile, a neighborhood filtering strategy [WH19; WH20a] is applied in both move operators to reduce the neighborhood size. So the neighborhoods $N_f(S)$ and $N_s(S)$ induced by $flip(i)$ and $swap(q, p)$ are defined as follows, respectively.

$$N_f(S) = \{S' : S' = S \oplus flip(i) : 1 \leq i \leq m, f(S') > f(S_b)\} \tag{4.2}$$

$$N_s(S) = \{S' : S' = S \oplus swap(q, p) : q \in A, p \in \bar{A}, f(S') > f(S_b)\} \tag{4.3}$$

In this work, we employ a union neighborhood that covers both neighborhoods $N_f(S)$ and $N_s(S)$, i.e., $N(S) = N_f(S) \cup N_s(S)$. Moreover, we also apply a streamlining *gain updating strategy* to quickly evaluate the weight of each neighboring solution (see [Lin+19; WH20a] for more details).

Tabu list management strategy using hash functions

During the SBTS procedure, the current solution S is iteratively replaced by the best admissible neighboring solution S', which is identified according to the objective function value and the tabu list strategy described in this section. Unlike the traditional attribute-based tabu search, where the tabu list records the performed moves, our solution-based tabu search uses hash vectors and hash functions to implement the tabu list.

Following previous studies [Lai+18a; LHY19; Lai+18b; WWG17], our tabu list management strategy relies multiple hash vectors and hash functions, which helps significantly reduce the probability of wrong identification of the tabu status. Specifically, we adopt three hash vectors H_v $(v = 1, 2, 3)$ of length L, where each position takes a binary value which contributes to the definition of the tabu status of candidate solutions. The hash vectors are initialized to 0, indicating that no candidate solution is classified as tabu. Once a candidate solution is selected to replace the current solution S, the corresponding positions in the three hash vectors will be set to 1 (i.e., $H_v[h_v(S)] \leftarrow 1$, $v = 1, 2, 3$).

Given a candidate solution $S = (y_1, \ldots, y_m)$ where $y_i = 1$ if item i is selected, and $y_i = 0$ otherwise, the hash values $h_v(S)$ $(v = 1, 2, 3)$ are calculated by

$$h_v(S) = (\sum_{i=1}^{m} \lfloor \mathcal{W}_i^v \times y_i \rfloor) \bmod L \tag{4.4}$$

where L is the length of the hash vectors and is set to 10^8. And $\mathcal{W}_i^v$ is a pre-computed weight that satisfies the following relation: $\mathcal{W}_i^v = i^{\gamma_v}$ $(v = 1, 2, 3$ and $i = 1, \ldots, m)$, where γ_v is a parameter that takes different values for the three hash functions $(\gamma_v = 1.2, 1.6, 2.0)$. To reduce the possible collisions that occur with hash functions, we randomly shuffle the order in the pre-computed weight vector $\mathcal{W}^v$ in order to ensure an extended distribution of hash values of the solutions. Fig. 4.3 shows an illustrative example of this shuffling operation with five items and γ_v being set to $1.2, 1.6, 2.0$, respectively. The left figure indicates the pre-computed weights $\mathcal{W}_i^v$ $(v = 1, 2, 3,$ and $i = 1, \ldots, 5)$. Then the order of each of the three weight vectors $\mathcal{W}^v$ is randomly shuffled to obtain a new weight vector shown in the right figure. Our preliminary experiment indicates that this random shuffling operation helps to reduce the error rates of the hash functions. We present the rationale for the setting of γ_v and an analysis of the hash functions in Section 4.4.1.

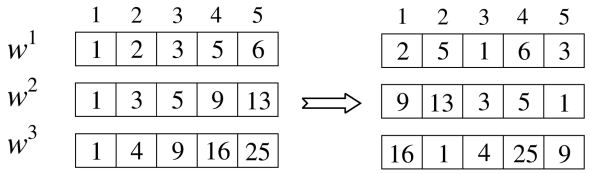


Figure 4.3 – An illustrative example of the random shuffling operation.

The hash-based tabu list management strategy works as follows. Given a candidate solution $S = (y_1, \ldots, y_m)$, we first calculate the three hash values $h_v(S)$ that are the indexes of the hash vectors. Then, the tabu status of solution S is determined according to the values of the hash vectors $H_v[h_v(S)]$. Specifically, S is determined as a forbidden solution (i.e., already visited) when $H_1[h_1(S)] \wedge H_2[h_2(S)] \wedge H_3[h_3(S)] = 1$. Otherwise, S is classified as an unforbidden solution that has not been visited by this round of SBTS and is eligible for solution transition. In this way, we can quickly determine the tabu status of a neighboring solution in $O(1)$, and this is the main advantage of the hash-based tabu list management strategy. For the illustrative example shown in Fig. 4.4, solution S is classified as tabu and thus is excluded for solution transition.

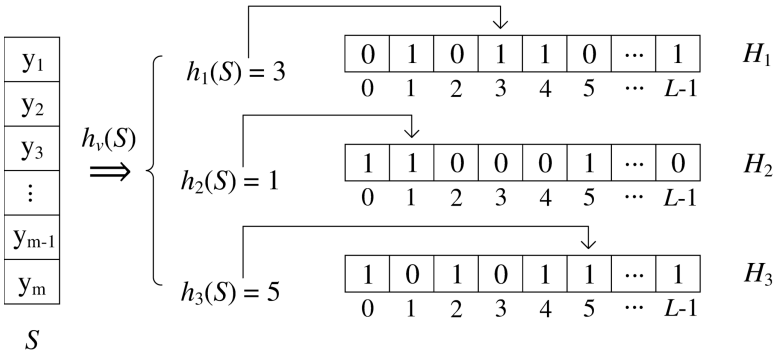


Figure 4.4 – An example of a solution forbidden by the hash functions and the associated hash vectors.

4.2.5 Computational complexity and discussion

From an empty subset S, the greedy randomized initialization procedure (Section 4.2.3 and Algorithm 12) creates a solution in four steps. The first step calculates additional weights in $O(m \times n)$, where m is the number of items and n is the number of elements. The second step calculates the dynamic profit ratio and identifies the top items with a complexity of $O(m \times log(m))$. The third step of calculating probability can be realized

in $O(|RCL|)$ and the fourth step of adding one item can be achieved in $O(1)$. Then the time complexity of the initialization procedure is $O(m \times n \times K_1)$, where K_1 is the maximum iterations of the initialization procedure. For the main solution-based tabu search procedure (Section 4.2.4 and Algorithm 13), we can evaluate its complexity as follows. Let $S =< A, \bar{A} >$ be a given input solution, the complexity of one iteration of the SBTS procedure is $O((m + |\bar{A}| \times |A|) \times n)$. Let K_2 be the maximum iterations of SBTS. Then the time complexity of SBTS is $O((m + |\bar{A}| \times |A|) \times n \times K_2)$.

Now we discuss the relations between our algorithm and the existing tabu search algorithms for the SUKP [Lin+19; WH19; WH20a]. First, MSBTS is the first solution-based tabu search algorithm for the SUKP, while the existing TS algorithms are based on the conventional attribute-based TS approach. Second, MSBTS employs a new tabu list management strategy that avoids tuning the tabu tenure. Third, unlike the previous TS algorithms that uses a perturbation procedure, MSBTS does not need such specific diversification strategies. Yet, it achieves remarkable results, as it is shown in Section 4.3.

Finally, it is worth mentioning that the solution-based tabu search approach has led to highly effective algorithms for several NP-hard binary problems such as 0-1 multidimensional knapsack [Lai+18a], multidemand multidimensional knapsack [LHY19], minimum differential dispersion [WWG17] and maximum min-sum dispersion [Lai+18b] and obnoxious p-median [Cha+21]. Our study of using solution-based tabu search for the SUKP further confirms the usefulness of this approach for binary optimization.

4.3 Computational results and comparisons

This section is devoted to a computational assessment of the proposed MSBTS algorithm, in comparison with three best-performing SUKP algorithms in the literature based on two sets of 60 benchmark instances available at `http://www.info.univ-angers.fr/pub/hao/SUKP_MSBTS.html`.

4.3.1 Benchmark instances

The SUKP benchmark instances adopted in our experiments were commonly tested in the literature, which can be divided into Set I and Set II. The Set I instances were proposed in [He+18] with 85 to 500 items and elements, while the Set II instances were introduced in [WH20a] with 585 to 1000 items and elements. These 60 instances share

the same characteristics. An instance is defined by m items, n elements and an associated binary relation matrix $R_{ij}[m \times n]$, where $R_{ij} = 1$ means that item i contains element j. Each instance is further characterized by two parameters: the density α of $R_{ij} = 1$ in the relation matrix R (i.e., $\alpha = (\sum_{i=1}^{m} \sum_{j=1}^{n} R_{ij})/(mn)$) and the ratio β of knapsack capacity C to the total weight of the elements (i.e., $\beta = C/\sum_{j=1}^{n} w_j$). As indicated in [He+18; WH20a], for the 60 instances tested in this study, α is equal to 0.10 or 0.15, while β is equal to 0.75 or 0.85.

4.3.2 Experimental settings

The proposed MSBTS algorithm was implemented in C++ and compiled using the g++ compiler with the -O3 option. All the experiments were carried out on an Intel Xeon E5-2670 processor (2.5 GHz CPU and 2 GB RAM) running under the Linux operating system. The MSBTS algorithm used the same stopping conditions for the reference algorithms (see below), i.e., 500 seconds for the Set I instances and 1000 seconds for the Set II instances. Each instance was solved 100 times independently with different random seeds. Note that contrary to the existing algorithms, our algorithm eliminates the need for tuning parameters.

Among the existing algorithms for the SUKP in the literature, we identify four best performing algorithms according to the reported computational results: hybrid jaya algorithm [WH20b] (DHJaya, 2019), hybrid binary particle swarm optimization with tabu search algorithm [Lin+19] (HBPSO/TS, 2019), iterated two-phase local search algorithm [WH19] (I2PLS, 2019) and the kernel based tabu search algorithm [WH20a] (KBTS, 2020). We thus use them as the reference algorithms for our comparative study. Since the results of these algorithms were obtained in [WH20a] on the same computing platform and under the same stopping condition as in this work, we directly adopt these results in our study.

4.3.3 Computational results

The computational results of our MSBTS algorithm and the reference algorithms on the SUKP instances of Set I and Set II are reported in Table 4.1 and 4.2, respectively [1]. The first column of these two tables gives the name of each instance, where the asterisk

1. Our solution certificates are available at: http://www.info.univ-angers.fr/pub/hao/SUKP_MSBTS.html.

(*) denotes that the optimal value proved by CPLEX [WH19]. The remaining columns report the following information: the best objective value (f_{best}), the average objective value (f_{avg}), the standard deviations over 100 runs (std) and the average run times t_{avg} (to obtain the f_{best} value) of each involved algorithm. The row #Avg shows the average value of each column. Furthermore, the **bold** entries highlight the dominating values among the compared results, while the *italic* entries indicate the equal best values.

Comparing the results of Table 4.1 leads to the following comments. First, in terms of the best performance indicator, MSBTS can attain all the best-known f_{best} results on all the 30 instances of Set I, thus dominating DHJaya and matching the performance of the best algorithms I2PLS and KBTS. Second, in terms of the average performance indicator, our MSBTS algorithm dominates DHJaya and competes favorably with HBPSO/TS and I2PLS, while performing marginally worse than KBTS even if MSBTS has better f_{avg} results on five large instances with 485 to 500 items and elements. It is difficult to further compare the competing algorithms on Set I, since the *p-values* in Table 4.3 from the non-parametric Wilcoxon signed-rank test don't show a statistical difference at 0.05 significance level between MSBTS and the reference algorithms except DHJaya. So we focus on Set II for a more detailed comparison.

Table 4.2 on the 30 instances of Set II discloses that our MSBTS algorithm outperforms the reference algorithms on large size instances. Specifically, MSBTS matches the best-known f_{best} values for the remaining 23 instances, and remarkably, finds 7 new best-known results (improved lower bounds). Most of these 7 instances have 985 to 1000 items, which demonstrates the advantage of our algorithm on the most difficult instances. When considering the average performance, MSBTS remains highly competitive compared to the reference algorithms. On the other hand, MSBTS has a zero std value on 20 instances while the reference algorithms achieve less zero std values (0 for DHJaya, 1 for HBPSO/TS, 2 for I2PLS and 6 for KBTS), which shows the robustness of our algorithm. Moreover, the smallest #Avg value of the corresponding t_{avg} entries obtained by MSBTS demonstrates that our algorithm is more computational efficient than the reference algorithms on this set of SUKP instances. We show a detailed time-efficiency comparison of our MSBTS algorithm with the reference algorithms in Section 4.3.4.

In order to better highlight the advantage of the proposed MSBTS algorithm, we summarize the comparative results between MSBTS and each reference algorithm in Table 4.3. The first two columns of the table give the pairs of two compared algorithms and the corresponding instance sets, respectively. Columns #Wins, #Ties and #Losses show the

number of instances for which MSBTS obtains a better, equal and worse result according to the f_{best} and f_{avg} indicators. The last column indicates the *p-values* from the Wilcoxon signed-rank test, where 'NA' implies that two underlying groups of results are exactly the same. From Table 4.3, we can observe that MSBTS achieves better or equal results in terms of f_{best} on all the tested instances, while being better in terms of f_{avg} on most instances. Note that KBTS reports more f_{avg} values better than MSBTS for Set I (13 vs 6). However, the Wilcoxon signed-rank test in Table 3 ($p\text{-}value$ = 9.10e-2 $>$ 0.05) indicates that there is no statistically significant difference. Furthermore, as shown in the last column, the *p-values* ($<$ 0.05) obtained between MSBTS and each compared algorithm on the instances of Set II confirm the statistically significant difference of the compared results.

4.3.4 Time-to-target analysis

We now present a time-to-target analysis (TTT) to evaluate the computational efficiency of the proposed MSBTS algorithm compared to the reference algorithms. For this, we compare the time required for each algorithm to obtain a solution at least as good as a given target value and measure the empirical probability distributions. More details about TTT can be found in [ARR07; RRV12].

Specifically, we run each compared algorithm 100 times to solve each instance of Set II with the setting shown in Section 4.3.2 and recorded the time to achieve an objective value at least as good as the given target value (the algorithm stops immediately when it reaches the target value). Then we sorted the times in increasing order and calculated the probability $\rho_i = (i - 0.5)/100$ with each time T_i, where T_i corresponds to the ith smallest time.

Table 4.4 shows the experimental results of DHJaya, HBPSO/TS, I2PLS, KBTS and MSBTS on the instances of Set II. The first two columns give the name of each instance and the corresponding target value, respectively. The remaining columns report the best time (T_{best}) in seconds to achieve the target value and the average time (T_{avg}) in seconds to reach the target value over 100 runs. The row (#Avg) indicates the average value of each column. And the row #Best shows the number of instances for which an algorithm obtains the smallest T_{best} value among the compared algorithms. Moreover, to check whether there exists a significant difference between the proposed MSBTS algorithm and the compared algorithms in terms of T_{best} and T_{avg}, we report the *p-values* from the Wilcoxon signed-rank test in the last row.

Table 4.1 – Computational results of the MSBTS algorithm and the reference algorithms on the 30 benchmark instances of Set I.

Instance	DHJaya [WH20b]				HBPSO/TS [Lin+19]				I2PLS [WH19]				KBTS [WH20a]				MSBTS			
	f_{best}	f_{avg}	*std*	$t_{avg}(s)$	f_{best}	f_{avg}	*std*	$t_{avg}(s)$	f_{best}	f_{avg}	*std*	$t_{avg}(s)$	f_{best}	f_{avg}	*std*	$t_{avg}(s)$	f_{best}	f_{avg}	*std*	$t_{avg}(s)$
F1*	*13283*	*13283*	0	9.477	*13283*	*13283*	0	0.098	*13283*	*13283*	0	3.094	*13283*	*13283*	0	4.082	*13283*	*13283*	0	12.770
F2*	*12479*	*12479*	0	24.414	*12479*	12403.15	98.97	101.122	*12479*	12335.13	98.78	103.757	*12479*	*12479*	0	42.992	*12479*	12413.78	79.79	184.323
F3	*13521*	13498.22	26.10	258.213	*13521*	*13521*	0	0.490	*13521*	*13521*	0	71.984	*13521*	*13521*	0	6.988	*13521*	*13521*	0	22.528
F4	*14215*	**14215**	0	83.129	*14215*	14177.38	70.84	72.041	*14215*	14031.28	131.46	180.809	*14215*	14209.87	29.17	107.407	*14215*	13946.15	153.67	258.541
F5	11385	11167.77	129.98	174.335	*11563*	*11563*	0	38.355	*11563*	11562.02	3.94	181.248	*11563*	*11563*	0	28.841	*11563*	*11563*	0	37.877
F6	12402	12248.42	22.12	316.767	*12607*	**12607**	0	24.967	*12607*	12364.55	83.03	240.333	*12607*	12536.02	87.51	235.450	*11563*	12430.51	73.86	216.465
F7	*11484*	11325.88	38.65	229.370	*11484*	*11484*	0	10.870	*11484*	*11484*	0	31.801	*11484*	*11484*	0	0.296	*11484*	*11484*	0	7.643
F8	10710	10293.96	173.85	241.068	*11209*	*11209*	0	16.478	*11209*	11157.26	87.29	141.525	*11209*	*11209*	0	72.020	*11209*	*11209*	0	46.800
F9	11722	11675.51	55.53	226.604	*11771*	11746.19	57.98	293.514	*11771*	11729.76	6.59	349.545	*11771*	11755.47	19.74	206.199	*11771*	**11771**	0	31.171
F10	10194	9703.56	114.85	2383.021	10194	10163.76	82.11	92.121	*10238*	10133.94	94.72	369.375	*10238*	10202.90	16.25	293.140	*10238*	**10205.62**	16.33	389.536
S1*	*14044*	*14044*	0	1.374	*14044*	*14044*	0	0.518	*14044*	*14044*	0	38.245	*14044*	*14044*	0	0.023	*14044*	*14044*	0	6.639
S2*	*13508*	*13508*	0	1.572	*13508*	*13508*	0	2.923	*13508*	13451.50	126.49	70.587	*13508*	*13508*	0	33.403	*13508*	*13508*	0	55.103
S3	*12522*	12480.62	65.05	207.667	*12522*	*12522*	0	0.8125	*12522*	*12522*	0	54.780	*12522*	*12522*	0	48.206	*12522*	12518.28	21.15	70.411
S4	*12317*	12217.81	93.361	229.824	*12317*	*12317*	0	0.950	*12317*	12280.07	57.77	238.348	*12317*	*12317*	0	72.495	*12317*	12316.21	7.86	92.155
S5	12736	12676.78	35.20	241.774	*12817*	12806.44	15.39	29.074	*12817*	*12817*	0	66.403	*12817*	*12817*	0	74.247	*12817*	12813.70	9.90	165.618
S6	11425	11260.25	103.95	152.329	*11585*	*11585*	0	5.985	*11585*	11512.18	73.15	220.100	*11585*	11584.17	8.26	141.464	*11585*	*11585*	0	156.333
S7	11569	11301.56	74.88	322.143	*11665*	11484.20	72.95	45.025	*11665*	*11665*	0	18.733	*11665*	*11665*	0	64.126	*11665*	11657.08	10.56	90.423
S8	10927	10721.45	221.38	77.037	*11325*	*11325*	0	5.902	*11325*	*11325*	0	76.000	*11325*	*11325*	0	17.591	*11325*	11309.20	112.46	125.950
S9	10943	10871.22	39.93	41.383	11109	11026.24	51.62	340.958	*11249*	11243.40	27.43	134.186	*11249*	11248.96	0.40	146.040	*11249*	**11249**	0	29.905
S10	10214	10069.33	103.33	101.926	*10381*	10213.25	71.30	220.328	*10381*	10293.89	85.53	237.894	*10381*	10362.63	52.25	156.331	*10381*	**10365.52**	49.41	169.084
T1*	*12045*	*12045*	0	17.199	*12045*	*12045*	0	0.056	*12045*	*12045*	0	2.798	*12045*	*12045*	0	0.075	*12045*	*12045*	0	3.117
T2*	*12369*	*12369*	0	0.342	*12369*	*12369*	0	0.088	*12369*	12315.53	62.60	17.470	*12369*	*12369*	0	10.175	*12369*	*12369*	0	26.240
T3	*13696*	13667.63	26.56	244.205	*13696*	*13696*	0	0.489	*13696*	13695.60	3.68	124.136	*13696*	*13696*	0	5.851	*13696*	*13696*	0	7.089
T4	*11298*	*11298*	0	38.439	*11298*	*11298*	0	0.486	*11298*	11276.17	83.78	139.865	*11298*	*11298*	0	6.373	*11298*	*11298*	0	30.689
T5	*11568*	11563.80	10.41	203.874	*11568*	*11568*	0	13.630	*11568*	*11568*	0	25.128	*11568*	*11568*	0	30.618	*11568*	11567.70	2.99	17.706
T6	11714	11436.93	101.85	463.466	*11802*	**11802**	0	2.135	*11802*	11790.43	27.51	206.422	*11802*	11799.27	9.95	168.904	*11802*	11798.88	10.58	186.685
T7	10483	10287.36	80.61	53.459	*10600*	10552.73	74.68	100.155	*10600*	10536.53	56.08	234.475	*10600*	**10600**	0	73.087	*10600*	10599.70	1.71	150.505
T8	10302	10184.09	138.00	230.077	*10506*	10472.40	67.20	168.870	*10506*	10502.64	23.52	129.505	*10506*	**10506**	0	58.240	*10506*	10504.23	16.08	133.340
T9	11036	10883.19	48.58	66.029	*11321*	11142.27	62.51	223.387	*11321*	11306.47	36.00	207.118	*11321*	11318.81	10.95	121.494	*11321*	**11321**	0	54.178
T10	10104	9665.70	142.57	49.438	*10220*	10208.96	3.26	143.999	*10220*	10179.45	46.97	238.630	*10220*	**10219.76**	1.68	118.564	*10220*	10219.04	3.26	123.052
#Avg	11873.83	11748.07	61.56	156.332	11967.47	11938.10	24.29	65.194	*11973.60*	11932.39	40.54	138.476	*11973.60*	**11968.56**	7.87	78.16	*11973.60*	11953.72	18.98	796.729

Table 4.2 – Computational results of the MSBTS algorithm and the reference algorithms on the 30 benchmark instances of Set II.

Instance	DHJaya [WH20b]				HBPSO/TS [Lin+19]				I2PLS [WH19]				KBTS [WH20a]				MSBTS			
	f_{best}	f_{avg}	std	$t_{avg}(s)$	f_{best}	f_{avg}	std	$t_{avg}(s)$	f_{best}	f_{avg}	std	$t_{avg}(s)$	f_{best}	f_{avg}	std	$t_{avg}(s)$	f_{best}	f_{avg}	std	$t_{avg}(s)$
F11	9640	9449.97	60.22	690.489	9741	9724.60	7.68	576.260	9750	9734.74	13.39	479.356	*9914*	*9914*	0	209.679	*9914*	*9914*	0	181.952
F12	9187	8998.45	79.17	881.295	*9357*	9174.16	143.19	413.157	*9357*	9324.62	16.67	457.807	*9357*	9354.52	9.18	263.684	*9357*	**9357**	0	59.382
F13	9790	9602	55.96	543.236	*9881*	9792.23	51.06	881.999	*9881*	9819.24	38.74	363.945	*9881*	9844.96	11.88	455.713	*9881*	**9881**	0	28.474
F14	9106	8894.09	140.48	426.088	9135	8940.65	109.78	689.759	*9163*	9135.27	4.90	671.132	*9163*	9138.36	9.10	524.799	*9163*	**9163**	0	102.379
F15	9771	9540.08	47.95	637.331	9837	9736.89	46.11	777.755	9822	9678.89	80.67	719.986	9837	9808.86	20.42	483.384	**9937**	**9937**	0	259.160
F16	8797	8649	63.01	236.798	8907	8872.84	84.36	418.033	8907	8780.32	43.34	674.231	*9024*	8955.29	49.07	474.643	*9024*	**8986.25**	25.38	486.666
F17	9455	9249.53	109.14	687.150	9611	9560.93	89.43	514.922	9611	9537.61	61.42	511.245	*9725*	9616.70	24.85	609.811	*9725*	**9725**	0	192.213
F18	8418	8244.47	87.93	316.604	8481	8208.22	108.56	332.102	8481	8426.36	44.76	541.670	*8620*	8526.55	48.37	274.653	*8620*	**8566.71**	31.18	978.573
F19	9424	9306.86	45.01	309.873	9668	9278.50	125.80	620.436	9580	9221.23	103.18	329.743	9668	9496.63	74.35	487.925	**9689**	**9632.59**	29.56	671.192
F20	8433	8280.52	90.87	312.589	8448	8129.08	92.71	564.848	8448	8268.18	135.55	541.606	8453	8448.05	0.50	941.565	**8455**	**8453.36**	0.77	634.006
S11	10507	10504.25	19.67	321.196	10518	10517.89	1.09	60.254	*10524*	10520.70	2.99	513.537	*10524*	10521.72	2.91	404.697	*10524*	**10524**	0	16.377
S12	8910	8785.64	43.46	571.965	9024	8902.33	27.27	214.261	*9062*	9022.97	46.28	456.386	*9062*	9061.16	4.78	255.342	*9062*	**9062**	0	224.626
S13	9512	9409.01	28.70	809.836	*9786*	9679.56	72.51	215.910	*9786*	9742.73	40.87	383.700	*9786*	*9786*	0	97.316	*9786*	*9786*	0	64.868
S14	9121	8985.51	65.90	507.656	9177	9003.15	138.46	659.194	*9229*	9155.79	18.61	445.194	*9229*	9187.55	20.70	486.304	*9229*	**9229**	0	96.472
S15	9890	9656.38	51.42	567.090	*9932*	9823.17	113.20	607.506	*9932*	9685.79	72.06	868.227	*9932*	9930.56	14.33	214.286	*9932*	**9932**	0	21.032
S16	8961	8774.18	59.78	161.688	8907	8732.94	160.07	590.883	8961	8909.50	10.91	27.170	*9101*	8936.12	39.55	321.859	*9101*	**9101**	0	129.395
S17	9526	9462.86	37.83	670.990	*9745*	9639.60	51.13	598.520	*9745*	9660.12	36.68	341.110	*9745*	9729.51	30.06	368.807	*9745*	**9745**	0	45.950
S18	8718	8492.88	62.31	702.655	8916	8617.20	210.54	665.798	8916	8916	0	116.694	*8990*	8918.96	14.50	672.574	*8990*	**8990**	0	237.865
S19	9348	9250.80	53.65	542.187	9509	9273.64	82.57	802.652	9544	9255.73	142.33	876.669	9544	9431.47	60.84	510.660	**9551**	**9551**	0	142.712
S20	8330	8037.92	71.87	932.614	8134	7872.84	95.76	97.909	8379	8206.49	68.52	632.334	8474	8376.20	27.12	500.435	**8538**	**8497.39**	28.46	505.954
S11	10300	10161.45	72.81	98.186	*10393*	10191.01	102.35	729.422	*10393*	10366.15	29.83	499.311	*10393*	*10393*	0	89.785	*10393*	*10393*	0	73.093
T12	9031	8944.22	61.72	616.631	*9256*	*9256*	0	103.637	*9256*	*9256*	0	264.876	*9256*	*9256*	0	84.359	*9256*	*9256*	0	99.163
T13	10070	9953.55	49.02	430.180	*10121*	9909	30.82	123.012	*10121*	9979.70	86.13	540.289	*10121*	10114.96	31.87	230.918	*10121*	**10121**	0	9.229
T14	9102	8860.79	106.42	159.976	*9176*	8936.47	135.64	645.153	*9176*	9139.18	52.80	461.051	*9176*	*9176*	0	140.151	*9176*	*9176*	0	96.859
T15	9123	8885.09	54.14	316.494	*9384*	9163.90	70.91	339.415	*9384*	9236.10	95.56	576.738	*9384*	**9384**	0	136.173	*9384*	9382.68	9.24	210.315
T16	8556	8482.33	51.45	604.625	8572	8322.17	57.53	665.514	8663	8558.51	79.51	586.047	*8746*	8643.93	47.92	467.334	*8746*	**8684.58**	36.41	720.765
T17	9137	9079.09	46.70	590.376	9232	9121.24	48.92	455.104	9232	9106.31	62.28	452.360	*9318*	9236.16	21.32	281.632	*9318*	**9318**	0	81.932
T18	8217	7881.44	65.84	140.935	8277	7900.57	131.65	296.061	*8425*	8268	104.34	484.859	*8425*	8311.68	46.80	625.829	*8425*	**8411.72**	9.88	573.526
T19	9067	8994.48	44.99	313.094	9113	8938.38	66.64	967.315	9047	8917.48	126.37	89.760	9193	9105.84	74.76	319.356	**9234**	**9193.15**	13.26	855.645
T20	8453	8425.27	48.74	503.976	8172	7958.24	121.56	350.640	8528	8233.05	119.98	283.901	8528	8488.13	33.47	450.711	**8612**	**8578.20**	32.47	628.435
#Avg	9196.67	9041.40	62.54	482.096	9280.33	9105.91	85.91	499.248	9310.10	9202.09	57.96	473.031	9352.30	9303.10	23.95	379.479	**9362.93**	**9351.59**	7.22	280.940

Table 4.3 – Summarized comparisons of the MSBTS algorithm against each reference algorithm over the two sets of benchmark instances.

Algorithm pair	Instance set	Indicator	#Wins	#Ties	#Losses	*p-value*
MSBTS vs. DHJaya [WH20b]	Set I (30)	f_{best}	16	14	0	4.82e-4
		f_{avg}	23	6	1	1.37e-4
	Set II (30)	f_{best}	30	0	0	1.82e-06
		f_{avg}	30	0	0	1.86e-09
MSBTS vs. HBPSO/TS [Lin+19]	Set I (30)	f_{best}	2	28	0	1.80e-1
		f_{avg}	11	12	7	1.33e-1
	Set II (30)	f_{best}	20	10	0	5.96e-5
		f_{avg}	29	1	0	2.56e-6
MSBTS vs. I2PLS [WH19]	Set I (30)	f_{best}	0	30	0	NA
		f_{avg}	19	5	6	2.64e-2
	Set II (30)	f_{best}	15	15	0	8.83e-5
		f_{avg}	29	1	0	2.56e-6
MSBTS vs. KBTS [WH20a]	Set I (30)	f_{best}	0	30	0	NA
		f_{avg}	6	11	13	9.10e-2
	Set II (30)	f_{best}	7	23	0	1.80e-2
		f_{avg}	24	5	1	1.57e-5

From Table 4.4, we observe that the proposed MSBTS algorithm is very competitive compared to the reference algorithms in terms of T_{best} and T_{avg}. In particular, MSBTS attains the smallest T_{best} values for 22 instances (out of 30) against 0 for DHJaya, HBPSO/TS, I2PLS and 8 for KBTS. Also, MSBTS has a better average performance according to the #Avg values in the last row. The *p-values* (< 0.05) from the Wilcoxon signed-rank test clearly indicate that differences between MSBTS and the compared algorithms are statistically significant.

To further illustrate the computational efficiency of MSBTS compared to the reference algorithms, we plot the points (T_i, ρ_i) based on two SUKP instances of Set II and show the time-to-target plots in Fig. 4.5. The X-axis in each sub-figure indicates the time to achieve the target value, and the Y-axis is the cumulative probability of reaching the given target value. We observe that the cumulative probability of each algorithm increases with the run-time. However, MSBTS (also KBTS) attains a high probability (over 90%) in a very short computation time (less than 20 seconds) on both instances, while the other algorithms perform poorly. Regarding MSBTS and KBTS, in order to attain a probability of 99.5% of reaching the target value, MSBTS requires about 12 seconds on both instances, while KBTS consumes around 42 seconds and 26 seconds. Note that DHJaya failed to obtain the probability of 99.5% within the time limit of 1000s on both instances. This experiment demonstrates the computational efficiency of the proposed MSBTS algorithm.

Table 4.4 – Time-to-target analysis on the SUKP instances of Set II.

Instance/Algorithm	Target	DHJaya		HBPSO/TS		I2PLS		KBTS		MSBTS	
		$T_{best}(s)$	$T_{avg}(s)$	$T_{best}(s)$	$T_{avg}(s)$	$T_{best}(s)$	$T_{avg}(s)$	$T_{best}(s)$	$T_{avg}(s)$	$T_{best}(s)$	$T_{avg}(s)$
600_585_0.10_0.75	9500	100.079	523.459	3.470	9.353	3.741	11.927	**0.213**	**0.618**	0.654	1.297
600_585_0.15_0.85	9100	65.826	566.872	67.883	382.176	6.599	59.784	3.486	12.187	**1.244**	**9.123**
700_685_0.10_0.75	9700	270.577	561.072	11.334	133.119	11.657	66.970	1.216	7.799	**0.858**	**5.310**
700_685_0.15_0.85	9100	106.614	427.274	123.888	526.406	9.663	178.041	1.647	42.561	**1.370**	**23.398**
800_785_0.10_0.75	9500	160.885	650.605	18.534	132.560	25.917	241.929	**1.272**	15.965	1.325	**7.600**
800_785_0.15_0.85	8700	151.590	516.174	68.445	323.062	15.246	102.963	5.448	55.774	**2.492**	**7.798**
900_885_0.10_0.75	9400	313.696	560.054	37.409	271.706	13.254	295.578	**1.530**	28.776	3.346	**8.834**
900_885_0.15_0.85	8400	221.799	400.128	499.176	652.865	13.318	459.241	2.592	60.691	**2.139**	**9.243**
1000_985_0.10_0.75	9000	291.897	421.090	9.114	97.051	13.008	150.602	1.061	21.855	**0.639**	**16.614**
1000_985_0.15_0.85	8300	293.618	574.331	678.089	820.440	530.745	530.745	**6.685**	116.893	10.870	**25.255**
600_600_0.10_0.75	10500	67.558	369.584	16.691	51.810	5.938	31.448	2.589	58.678	**1.041**	**5.271**
600_600_0.15_0.75	8800	68.179	560.449	6.067	131.515	5.670	40.425	1.170	**5.538**	**0.697**	5.450
700_700_0.10_0.75	9500	654.112	743.459	9.297	163.108	9.090	99.874	1.769	12.083	**0.721**	**4.817**
700_700_0.15_0.85	9100	105.922	521.651	111.166	690.543	23.306	265.033	4.807	29.800	**2.098**	**19.571**
800_800_0.10_0.75	9800	573.460	576.004	180.088	549.453	866.553	866.553	9.431	213.756	**6.618**	**21.098**
800_800_0.15_0.85	8800	162.727	575.454	114.424	508.682	15.821	131.655	**1.385**	27.487	2.459	**8.224**
900_900_0.10_0.75	9500	220.422	603.266	33.222	261.629	11.589	46.073	**1.275**	8.965	2.809	**6.297**
900_900_0.15_0.85	8600	235.578	459.369	50.142	554.410	12.601	84.906	1.033	10.397	**0.912**	**8.208**
1000_1000_0.10_0.75	9300	327.772	784.859	76.998	560.236	30.069	412.291	**2.125**	149.036	18.468	**43.389**
1000_1000_0.15_0.85	8000	294.562	530.699	76.860	548.614	25.684	225.339	2.132	24.634	**1.218**	**8.561**
585_600_0.10_0.75	10000	64.865	245.444	15.053	83.161	6.935	17.042	1.614	5.787	**0.746**	**2.510**
585_600_0.15_0.85	9000	65.337	528.534	8.954	63.449	7.319	96.275	1.033	21.288	**1.005**	**21.108**
685_700_0.10_0.75	10000	333.101	472.050	137.383	171.016	108.642	484.414	13.512	235.948	**2.818**	**5.230**
685_700_0.15_0.85	9000	154.648	514.173	189.391	531.964	19.709	299.990	3.127	45.425	**1.370**	**29.075**
785_800_0.10_0.75	8900	155.496	484.831	9.029	96.090	11.278	104.360	**0.756**	7.486	0.765	**2.847**
785_800_0.15_0.85	8500	150.938	607.258	679.254	679.254	27.872	358.703	10.115	155.450	**2.401**	**22.656**
885_900_0.10_0.75	9100	222.106	619.250	30.648	425.582	36.186	415.666	6.096	73.726	**4.159**	**17.378**
885_900_0.15_0.85	8000	346.018	631.780	228.520	564.195	28.099	209.716	1.941	17.235	**1.746**	**7.186**
985_1000_0.10_0.75	8900	300.232	540.491	278.500	651.225	36.254	428.971	12.316	113.230	**9.698**	**48.404**
985_1000_0.15_0.85	8100	281.460	437.529	109.148	276.985	47.763	287.634	1.088	12.129	**0.997**	**10.208**
#Avg	9073	225.369	533.573	129.272	363.722	65.984	233.472	3.482	53.040	**2.923**	**13.732**
#Best	-	0	0	0	0	0	0	8	2	**22**	**28**
#p-value	-	1.86e-09	1.86e-09	1.86e-09	1.86e-09	1.86e-09	1.86e-09	2.48e-2	9.31e-09	-	-

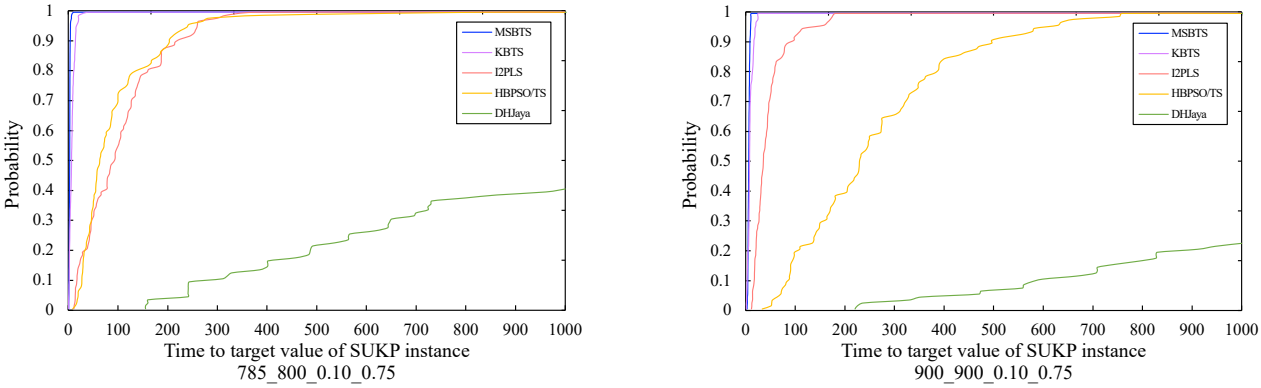


Figure 4.5 – Cumulative probability distributions for the time to reach a target value.

4.4 Analysis

In this section, we perform additional experiments to investigate the influences of the main ingredients of the MSBTS algorithm. Specifically, we study the effect of the parameter γ_v of the hash functions (Section 4.4.1), the error rates of hash functions (Section 4.4.2) and the benefit of the solution-based tabu search strategy (Section 4.4.3).

4.4.1 Sensitivity analysis of hash functions

Hash functions are the key ingredients of the MSBTS algorithm. Now, we analyze the influence of the parameter γ_v ($v = 1, 2, 3$) involved in the hash functions (see Section 4.2.4) on the performance of the MSBTS algorithm. As indicated in [WWG17], the proper settings of γ_v should satisfy two conditions: (1) the hash values of each candidate solution should be no more than the allowed maximum integer to avoid overflow; (2) the distribution of hash values of different candidate solutions should be wide enough to reduce possible collisions. We have carried out preliminary experiments for γ_v used in the hash functions. Experimental results show that a large γ_v value (> 2.8) will lead to integer overflow for instances with more than 985 items or elements. On the other hand, a small γ_v value (< 1.0) will lead to the same values of $\lfloor \mathcal{W}_i^v \rfloor$ ($\mathcal{W}_i^v = i^{\gamma_v}$) for adjacent items, increasing the probability of collisions. For example, assuming γ_v=0.9, the $\lfloor \mathcal{W}_i^v \rfloor$ values of the adjacent items 501 and 502 are both 269 ($\mathcal{W}_{501} = 501^{0.9} = 269.06$, $\mathcal{W}_{502} = 502^{0.9} = 269.55$). Given two neighboring solutions S_1 and $S_2 = S_1 \oplus swap(500, 501)$ where the *swap* operator was defined in Section 4.2.4, they will get the same hash value. As we focus on the ranges (1.0, 2.8) to analyze the influence of the parameter γ_v.

For this purpose, we tested 20 groups of parameters (γ_1, γ_2, γ_3) (see Table 4.5) on 10 representative SUKP instances, i.e., 785_800_0.15_0.85, 800_785_0.10_0.75, 800_785_0.15_0.85, 885_900_0.15_0.85, 900_885_0.15_0.85, 985_1000_ 0.10_0.75, 985_1000_0.15_0.85, 1000_985_0.10_0.75, 1000_985_0.15_0.85, 1000_1000_0.15_0.85. These 10 instances are denoted by the ID shown in Table 1.1, respectively. For the experiment, we performed 30 independent runs for each setting of parameters on each instance with the cut-off time of 1000 seconds, and recorded the average objective values (f_{avg}). In fact, we do not provide the best object values (f_{best}) here, since most of the f_{best} values obtained with different groups of parameters (γ_1, γ_2, γ_3) are exactly the same.

Table 4.5 displays the comparative results of this experiment, where the first row shows the label of each tested instance and the first column indicates the setting of the parameters (γ_1, γ_2, γ_3). The f_{avg} values of each group of γ_v are shown in rows 2 to 21, respectively. In addition, the last row #std gives the standard deviation of each column and the last column #Avg presents that the average values of each row.

Table 4.5 – Influence of the hash functions on the average performance of MSBTS algorithm.

(γ_1, γ_2, γ_3)/ Instance	T_{16}	F_{15}	F_{16}	T_{18}	F_{18}	T_{19}	T_{20}	F_{19}	F_{20}	S_{20}	#Avg
(1.1,1.3,1.5)	8665.77	9930.33	9004	8408.87	8578.23	9190.07	*8579.50*	**9647.67**	**8453.80**	8491.90	8895.01
(1.1,1.5,1.9)	8687.90	*9937*	8985.50	8411	8577.10	9191.87	8575.17	9627.50	8453.27	8490.07	8893.64
(1.2,1.4,1.8)	8687.90	*9937*	8983.67	8412.20	**8579.27**	9190.73	8583.83	9638.50	8453.60	8487.93	8895.46
(1.2,1.6,2.0)	*8693.43*	*9937*	8992.83	*8413.80*	8576.07	9192.53	*8579.50*	9631.60	8453.20	**8500.37**	**8897.03**
(1.3,1.5,1.7)	8671.30	*9937*	9000.17	8411.67	8581.43	**9192.93**	8577.33	9636.10	8453.43	8490.77	8895.21
(1.3,1.7,2.1)	8690.67	*9937*	8985.50	*8413.80*	8566.33	9189.03	8577.33	9631.60	8453.13	8491.70	8893.61
(1.4,1.6,2.0)	8687.90	9933.67	**8994.67**	8411.47	8582.53	9191.80	8573	9626.13	8453.27	8486.93	8894.14
(1.5,1.7,1.9)	8679.60	*9937*	8989.17	8412.20	8568.43	9189.73	8573	9631	8453.33	8496.10	8892.96
(1.5,1.9,2.3)	8685.13	9933.67	8983.67	8412.20	8571.67	9191.80	*8579.50*	9636.80	8453.40	8492.47	8894.03
(1.6,1.8,2.2)	8679.60	*9937*	8989.17	8412.20	8568.43	9189.73	8573	9631	8453.33	8496.10	8892.96
(1.7,1.9,2.1)	8685.13	9933.67	8985.50	8412.20	8573.90	9190.50	8573	9637.50	8453.40	8492.47	8893.73
(1.7,2.1,2.5)	8682.37	9933.67	8981.83	8412.20	8573.90	9191.57	8575.17	9637.50	8453.40	8492.47	8893.41
(1.8,2.0,2.4)	8685.13	9933.67	8983.67	8412.20	8571.67	9191.80	*8579.50*	9636.80	8453.40	8492.47	8894.03
(1.9,2.1,2.3)	8682.37	9933.67	8981.83	8412.20	8573.90	9191.57	8577.33	9637.20	8453.40	8492.47	8893.59
(1.9,2.3,2.7)	8682.37	9933.67	8981.83	8412.20	8573.90	9191.57	8577.33	9637.50	8453.40	8492.47	8893.62
(2.0,2.2,2.6)	8685.13	9933.67	8985.50	8412.20	8573.90	9190.50	8573	9637.50	8453.40	8492.47	8893.73
(1.1,1.2,2.7)	*8693.43*	*9937*	8983.67	8410.60	8568.40	9191.20	8573	9636.67	8453.40	8494.33	8894.17
(1.1,1.8,2.7)	8679.60	9933.67	8985.50	8412.73	8571.73	9191.57	8573	9643	8453.40	8490.33	8893.45
(1.1,2.0,2.7)	8676.83	9933.67	8981.83	8412.20	8573.90	9191.57	8575.17	9637.20	8453.40	8492.47	8892.82
(1.1,2.5,2.7)	8676.83	9933.67	8981.83	8412.20	8571.73	9191.57	8575.17	9637.20	8453.33	8492.47	8892.60
#std	6.93	1.96	6.30	1.04	4.35	0.99	3.10	4.94	0.14	2.88	-

From Table 4.5, we observe that the parameter γ_v is not sensitive for our algorithm.

First, the results obtained from different groups of parameters are very similar in terms of #Avg values. Specially, there are 12 out of 20 groups of parameters that obtained the same f_{avg} value on instance T_{18}. Second, the small #std values of each column indicate that the standard deviations of the results shown in the columns are relatively low. The *p-value* of 0.633 (> 0.05) from the Friedman statistical test again confirms that there are no significant differences among the tested results. This analysis indicated that any γ_v value in the interval (1.0, 2.8) is suitable for the proposed algorithm.

4.4.2 Error rates of hash functions

An error occurs when an unvisited solution is wrongly forbidden by the hash functions and the associated hash vectors. To calculate the error rates of hash functions, we ran our SBTS procedure for 10^4 iterations on two SUKP instances: 1000_1000_0.10_ 0.75, 1000_1000_0.15_0.85. During the search, each encountered solution is recorded in a pool POP. We use a counter c_1 to count the number of solutions forbidden (classified as tabu) by the hash functions. Another counter c_2 (error counter) will be incremented by 1 if the solution is not included in POP. Then the error rate is obtained by c_2/c_1. We perform additional experiments to investigate two factors that affect the error rates of the hash functions: 1) the length L of the hash functions, and 2) the number of the hash vectors.

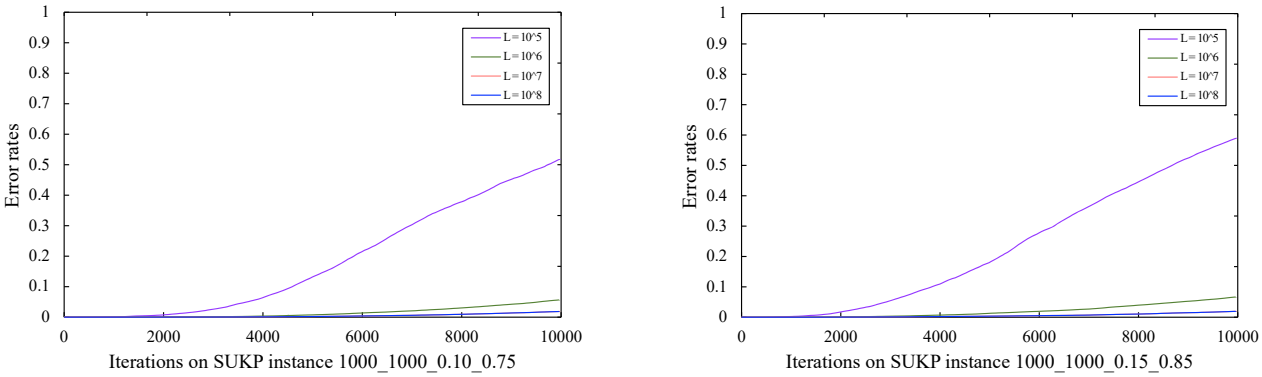


Figure 4.6 – Impact of the length L of hash vectors on the error rate of the solution-based tabu search procedure.

The role of the length L is to ensure that the hash vectors are long enough to be able to record the sampled solutions. A proper setting of L should not only avoid memory overflow, but also keep the error rates at a low level. The results of preliminary experiments indicate that a large L value ($> 10^8$) leads to memory overflow. Thus we carried out an

experiment to check the error rates of the hash vectors with L ranging from 10^5 to 10^8. The error rate plots are shown in Fig. 4.6, where the iterations of the SBTS procedure and the corresponding error rates are displayed on the X-axis and the Y-axis, respectively.

Fig. 4.6 shows that our algorithm can keep the error rate at a low level (< 0.07) with L ranging from 10^6 to 10^8. In particular, when the values of L are 10^7 and 10^8, the corresponding curves almost overlap and stay under 0.02. The error rates increase dramatically (more than 0.5) as the number of iterations increases for $L \leq 10^5$. Considering that the time complexity of evaluating a neighboring solution is $O(1)$, a large L value will not significantly affect the computation time. Thus any L value in the interval $[10^6, 10^8]$ is suitable for the proposed algorithm ($L = 10^8$ in the MSBTS algorithm).

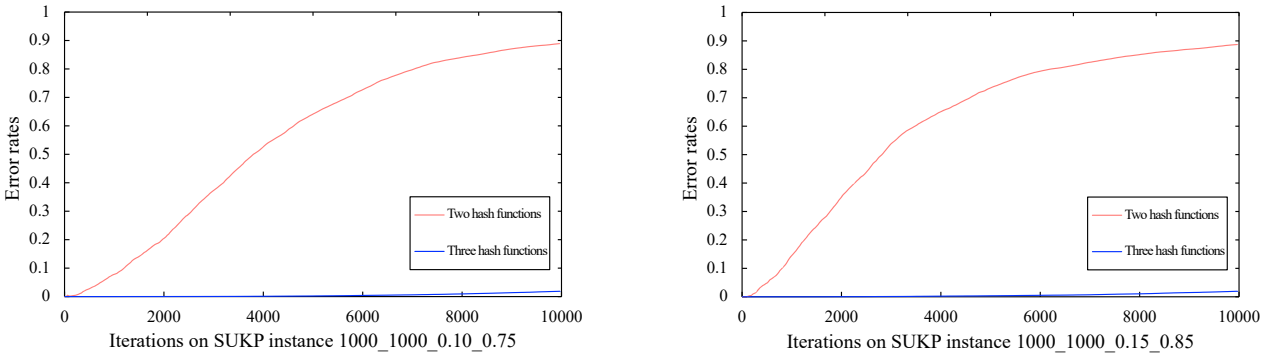


Figure 4.7 – Impact of the number of hash vectors on the error rate of the solution-based tabu search procedure.

The role of the hash vectors is to record the solutions encountered during the search, and the number of the hash vectors can significantly affect the error rates. We performed another experiment to analyze the error rates when using two or three hash vectors. As the error rate plots in Fig. 4.7 show, the SBTS procedure has an error rate of nearly 0.9 with two hash vectors over 10^4 iterations. The error rate with one hash vector will be naturally higher than that with two hash vectors for the same number of iterations. On the contrary, the error rate remains very low (< 0.02) with three hash vectors over 10^4 iterations. So three hash vectors can effectively identify the previously encountered solutions, which justifies the use of three hash vectors in MSBTS.

4.4.3 Analysis of solution-based tabu search

The solution-based tabu search strategy is the most innovative component of the MSBTS algorithm. To understand its influence on the algorithm, we created a MSBTS

variant; named MABTS where the solution-based tabu search procedure is replaced by an attribute-based tabu search procedure. For this experiment, we employed the attribute-based tabu search method introduced in [WH20a], which is one of the best SUKP algorithms. Thus, except the tabu search procedure, MABTS shares the other components of MSBTS.

Table 4.6 – Comparison between MSBTS and MABTS on the instances of Set II.

Instance/Setting	MSBTS			MABTS		
	f_{best}	f_{avg}	std	f_{best}	f_{avg}	std
600_585_0.10_0.75	*9914*	**9914**	0	*9914*	9801.57	72.65
600_585_0.15_0.85	*9357*	**9357**	0	*9357*	9329.40	23.76
700_685_0.10_0.75	**9881**	**9881**	0	9841	9814.37	34.83
700_685_0.15_0.85	**9163**	**9163**	0	9135	9126.67	14.16
800_785_0.10_0.75	**9937**	**9937**	0	9811	9679.73	61.37
800_785_0.15_0.85	*9024*	**8992.83**	27.25	*9024*	8892.53	51.21
900_885_0.10_0.75	**9725**	**9725**	0	9611	9503.63	53.57
900_885_0.15_0.85	**8620**	**8576.07**	27.14	8499	8459.87	26.51
1000_985_0.10_0.75	**9689**	**9631.60**	28.92	9580	9411.37	58.09
1000_985_0.15_0.85	**8455**	**8453.20**	0.60	8448	8359.30	106.74
600_600_0.10_0.75	*10524*	**10524**	0	*10524*	10519.67	3.54
600_600_0.15_0.75	*9062*	**9062**	0	*9062*	9058.20	11.40
700_700_0.10_0.75	*9786*	**9786**	0	*9786*	9770.20	37.93
700_700_0.15_0.85	**9229**	**9229**	0	9177	9145.20	30.65
800_800_0.10_0.75	*9932*	**9932**	0	*9932*	9734.87	64.12
800_800_0.15_0.85	**9101**	**9101**	0	8956	8907.10	14.88
900_900_0.10_0.75	**9745**	**9745**	0	9660	9629.20	36.02
900_900_0.15_0.85	**8990**	**8990**	0	8916	8911.03	17.49
1000_1000_0.10_0.75	**9551**	**9551**	0	9357	9269.87	92.10
1000_1000_0.15_0.85	**8538**	**8500.37**	28.65	8381	8282.20	73.08
585_600_0.10_0.75	*10393*	**10393**	0	*10393*	10325.43	34.75
585_600_0.15_0.85	*9256*	**9256**	0	*9256*	9256	0
685_700_0.10_0.75	*10121*	**10121**	0	*10121*	9944.10	59.12
685_700_0.15_0.85	*9176*	**9176**	0	*9176*	9144.97	31.29
785_800_0.10_0.75	*9384*	**9384**	0	*9384*	9229.37	93.68
785_800_0.15_0.85	**8746**	**8693.43**	40.00	8663	8526.57	59.71
885_900_0.10_0.75	**9318**	**9318**	0	9232	9158.57	40.38
885_900_0.15_0.85	*8425*	**8413.80**	7.33	*8425*	8276.07	42.39
985_1000_0.10_0.75	**9234**	**9192.53**	14.12	9193	9030.77	54.53
985_1000_0.15_0.85	**8612**	**8579.50**	32.50	8461	8384.43	75.03
#Avg	**9362.93**	**9352.61**	6.88	9309.17	9229.41	45.83
#Best	**30**	**30**	-	13	0	-
p-value	2.93e-4	2.563e-06	-	-	-	-

Considering that our algorithm mainly shows its superiority on the large instances, we carried out this experiment based on Set II, where each instance was independently

solved by each algorithm 30 times, each run being limited to 1000 seconds.

The experimental results are reported in Table 4.6. The first column shows the names of the instances. The results of the two compared algorithms are respectively presented in columns 2 to 7, including the best objective value (f_{best}), the average objective value (f_{avg}), the standard deviation over 30 runs (std). To facilitate the comparison, we also provide the similar #Avg, #Best and *p-values* as described in Section 4.3.4.

Table 4.6 shows that MSBTS significantly outperforms MABTS, achieving better f_{best} values (marked in bold) for 17 out of the 30 instances and equal results for the remaining 13 instances. When comparing the f_{avg} values, MSBTS again dominates MABTS for all the instances. Moreover, the std values of MSBTS are very small, indicating that MSBTS is highly robust. Furthermore, the small *p-values* (< 0.05) show that there is a significant difference between MSBTS and MABTS. This experiment confirms that the solution-based tabu search strategy constitutes one key ingredient of our algorithm.

4.5 Conclusions

The Set-Union Knapsack Problem attracts more and more attention in recent years due to its theoretical and practical interest. Inspired by the fact that the solution-based tabu search has been successfully applied to solve several difficult binary optimization problems, we devised the first multistart solution-based tabu search algorithm for solving the SUKP. The proposed MSBTS algorithm uses its solution-based tabu search procedure to find high-quality local optima and the multistart mechanism to overcome deep local optima traps. MSBTS has several desirable features such as simple design and implementation as well as absence of parameters.

We performed extensive experimental assessments of the proposed algorithm on two sets of 60 benchmark instances. The comparisons with the state-of-the-art algorithms demonstrated the high competitiveness of our algorithm in terms of solution quality, computational efficiency and robustness. In particular, we demonstrated the interest of the MSBTS algorithm to deal with large instances and reported new lower bounds for 7 large and difficult instances (with 585 to 1000 items and elements).

This work thus provides a useful tool for solving the general Set-Union Knapsack Problem. Moreover, since a number of real-world applications can be conveniently formulated by SUKP, the proposed algorithm can be hopefully applied to these practical problems. The availability of the code of the MSBTS algorithm will facilitate such applications.

In the next chapter, we will focus on the disjunctively constrained knapsack problem which is also a variant of the popular knapsack problem. A threshold search based memetic algorithm will be introduced for solving this problem.

Chapter 5

A THRESHOLD SEARCH BASED MEMETIC ALGORITHM FOR THE DISJUNCTIVELY CONSTRAINED KNAPSACK PROBLEM

In this chapter, we present a threshold search based memetic algorithm for solving the DCKP that combines the memetic framework with threshold search to find high quality solutions. Extensive computational assessments on two sets of 6340 benchmark instances in the literature demonstrate that the proposed algorithm is highly competitive compared to the state-of-the-art methods. In particular, we report 24 and 354 improved best-known results (new lower bounds) for Set I (100 instances) and for Set II (6240 instances), respectively. We analyze the key algorithmic components and shed lights on their roles for the performance of the algorithm. The content of this chapter is based on an article that is being revised for *Computers & Operations Research*.

5.1 Introduction

As the literature review shown in Section 1.3.3, existing studies have significantly contributed to better solving the DCKP. However, given the $\mathcal{NP}$-hard nature of the problem, more powerful algorithms are still needed to push the limits of existing methods.

In this chapter, we investigate for the first time the population-based memetic framework [Mos99] for solving the DCKP and design an effective algorithm mixing threshold based local optimization and crossover based solution recombination. The threshold search procedure ensures the main role of search intensification by finding high quality local optimal solutions. The specialized backbone crossover generates promising offspring solutions for search diversification. The algorithm uses also a distance-and-quality strategy for population management. The algorithm has the advantage of avoiding the difficult task of parameter tuning.

From a perspective of performance assessment, we apply the proposed algorithm to solve the two sets of DCKP benchmark instances in the literature. The results show that for the 100 instances of Set I (optimality still unknown) which were commonly tested by heuristic algorithms, our algorithm discovers 24 new best-known results (new lower bounds) and matches the best-known results for the 76 remaining instances. For the 6240 instances of Set II which were tested by exact algorithms, our algorithm finds 354 improved best lower bounds on the difficult instances whose optimal values are unknown and attains the known optimal results on most of the remaining instances.

The rest of the chapter is organized as follows. Section 5.2 presents the proposed algorithm. Section 5.3 shows computational results of our algorithm and provides comparisons with the state-of-the-art algorithms. Section 5.4 analyzes essential components of the algorithm. Finally, Section 5.5 summarizes the chapter.

5.2 Threshold search based memetic algorithm for the DCKP

Our threshold search based memetic algorithm (TSBMA) for the DCKP is a population-based algorithm combining evolutionary search and local optimization. In this section, we first present the general procedure of the algorithm and then describe its components.

5.2.1 General procedure

The TSBMA algorithm relies on the general memetic algorithm framework [Mos99] and follows the design principles recommended in [Hao12; Zho+20]. The flowchart of TSBMA and its pseudo-code are shown in Figure 5.1 and Algorithm 14, respectively.

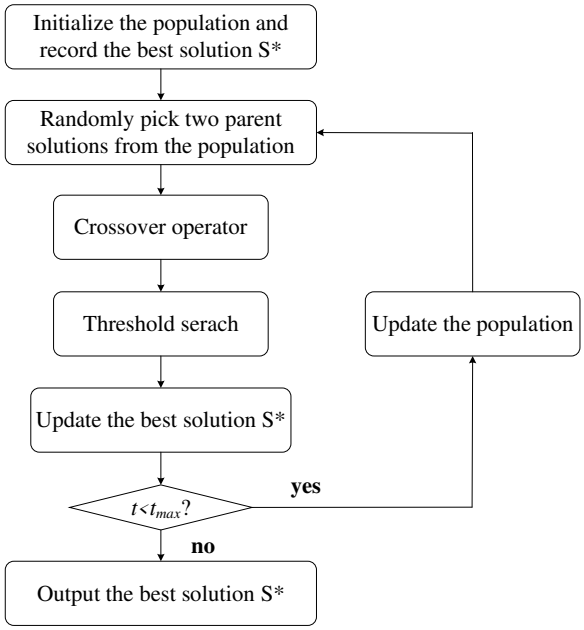


Figure 5.1 – Flowchart of the proposed TSBMA algorithm.

The algorithm starts from a set of feasible solutions of good quality that are generated by the population initialization procedure (line 4, Alg. 14, and Section 5.2.3). The best solution is identified and recorded as the overall best solution S^* (line 5, Alg. 14). Then the algorithm enters the main 'while' loop (lines 6-15, Alg. 14) to perform a number of generations. At each generation, two solutions are randomly picked and used by the crossover operator to create an offspring solution (line 7-8, Alg. 14, and Section 5.2.5). Afterwards, the threshold search procedure is triggered to perform local optimization with three neighborhoods N_1, N_2 and N_3 (line 9, Alg. 14, and Section 5.2.4). After conditionally updating the overall best solution S^* (lines 11-13, Alg. 14), the diversity-based pool updating procedure is applied to decide whether the best solution S_b found during the threshold search should be inserted into the population (line 14, Alg. 14, and Section 5.2.6). Finally, when the given time limit t_{max} is reached, the algorithm returns the overall best solution S^* found during the search and terminates.

Algorithm 14 Main framework of threshold search based memetic algorithm for the DCKP

Input: Instance I, cut-off time t_{max}, population P, the maximum number of iterations $IterMax$, neighborhoods N_1, N_2, N_3.
Output: The overall best solution S^* found.
$S^* \leftarrow \emptyset$ /* Initialize S^* (i.e., $f(S^*) = 0$)*/
$POP = \{S^1, \ldots, S^{|P|}\} \leftarrow Population\_Initialization(I)$ /* Section 5.2.3 */
$S^* \leftarrow argmax\{f(S^k)|k = 1, \ldots, p\}$
while $Time \leq t_{max}$ **do**
  Randomly pick two solutions S^i and S^j from the population POP
  $S^o \leftarrow Crossover\_Operator(S^i, S^j)$ /* Section 5.2.5 */
  $S_b \leftarrow Threshold\_Search(S^o, N_{1-3}, IterMax)$ /* Section 5.2.4 */
  /* Record the best solution S_b found during threshold search */
  if $f(S_b) > f(S^*)$ **then**
    $S^* \leftarrow S_b$ /* Update the overall best solution S^* found so far */
  end if
  $POP \leftarrow Pool\_Updating(S_b, POP)$ /* Section 5.2.6 */
end while
return S^*

5.2.2 Solution representation, search space, and evaluation function

The DCKP is a subset selection problem. Thus, a candidate solution for a set $V = \{1, \ldots, n\}$ of n items can be conveniently represented by a binary vector $S = (x_1, \ldots, x_n)$, such that $x_i = 1$ if item i is selected, and $x_i = 0$ otherwise. Equivalently, S can also be represented by $S =< A, \bar{A} >$ such that $A = \{q : x_q = 1 \text{ in } S\}$ and $\bar{A} = \{p : x_p = 0 \text{ in } S\}$.

Let $G = (V, E)$ be the given conflict graph and C be the knapsack capacity. Our TSBMA algorithm explores the following feasible search space Ω^F satisfying both the disjunctive constraints and the knapsack constraint.

$$\Omega^F = \{x \in \{0,1\}^n : \sum_{i=1}^{n} w_i x_i \leq C; x_i + x_j \leq 1, \forall \{i,j\} \in E, 1 \leq i,j \leq n, i \neq j\} \quad (5.1)$$

The quality of a solution S in Ω^F is determined by the objective value $f(S)$ of the DCKP (Equation 1.28).

5.2.3 Population initialization

The TSBMA algorithm builds each of the $|P|$ initial solutions of the population P in two steps. First, it randomly adds one by one non-selected items into an individual

solution S^i $(i = 1, \ldots, |P|)$ until the capacity of the knapsack is reached, while keeping the disjunctive constraints satisfied. Second, to obtain an initial population of reasonable quality, it improves the solution S^i by a short run of the threshold search procedure (Section 5.2.4) by setting $IterMax = 2n$.

It is worth mentioning that the population size $|P|$ is determined according to the number of candidate items n of the given instance, i.e., $|P| = n/100 + 5$. This strategy is based on two considerations. First, since the TSBMA algorithm is powerful enough to solve the instances of small size, a smaller population size can help to reduce the initialization time. Second, the instances of large size are more challenging, a larger population size helps to diversify the search.

5.2.4 Local optimization using threshold search

The local optimization procedure of the TSBMA algorithm relies on the threshold accepting method [GT90]. To explore a given neighborhood, the method accepts both improving and deteriorating neighbor solutions so long as the solution satisfies a quality threshold. One notices that this method has been successfully applied to solve several knapsack problems (e.g., quadratic multiple knapsack problem [CH15], multi-constraint knapsack problem [DW91] and multiple-choice knapsack problem [ZN08]) and other combinatorial optimization problems (e.g., [CS96; CH19; TKV04]). In this chapter, we adopt for the first time this method for solving the DCKP and devise a multiple neighborhood threshold search procedure reinforced by an operation-prohibiting mechanism.

Main scheme of the threshold search procedure

As shown in Algorithm 15, the threshold search procedure (TSP) starts its process from an input solution and three empty hash vectors (used for the operation-prohibiting mechanism, lines 3-5, Alg. 15). It then performs a number of iterations to explore three neighborhoods (Section 5.2.4) to improve the current solution S. Specifically, for each 'while' iteration (lines 9-25, Alg. 15), the TSP procedure explores the neighborhoods N_1, N_2 and N_3 in a deterministic way as explained in the next section. Any sampled non-prohibited neighbor solution S' (i.e., $H_1[h_1(S')] \wedge H_2[h_2(S')] \wedge H_3[h_3(S')] = 0$) is accepted immediately if the quality threshold T is satisfied (i.e., $f(S') \geq T$). Then the hash vectors are updated for solution prohibition and the best solution found during the TSP procedure is recorded in S_b (lines 18-20, Alg. 15). The main search ('while' loop) terminates when 1)

no admissible neighbor solution (i.e., non-prohibited and satisfying the quality threshold) exists in the neighborhoods N_1, N_2 and N_3, or 2) the best solution S_b cannot be further improved during $IterMax$ consecutive iterations. Specifically, the quality threshold T is determined adaptively by $f(S_b) - n/10$ (n is the number of items of each instance) while $IterMax$ is set to $(n/500 + 5) \times 10000$.

Algorithm 15 Threshold search procedure

Input: Input solution S^o, threshold T, the maximum number of iterations $IterMax$, hash vectors H_1, H_2, H_3, hash functions h_1, h_2, h_3, length of hash vectors L, neighborhoods N_1, N_2, N_3.
Output: The best feasible solution S_b found by threshold search procedure.
for $i \leftarrow 0$ **to** $L - 1$ **do**
    $H_1[i] \leftarrow 0$; $H_2[i] \leftarrow 0$; $H_3[i] \leftarrow 0$; /* Initialization of hash vectors */
end for
$S_b \leftarrow S^o$ /* S_b record the best solution found */
$S \leftarrow S^o$ /* S record the current solution */
$iter \leftarrow 0$
while $iter \leq IterMax$ **do**
    Examine the neighborhoods $N_1(S)$, $N_2(S)$, $N_3(S)$ in turn; /* Section 5.2.4 */
    /* Each non-prohibited neighbor solution S' satisfies $H_1[h_1(S')] \wedge H_1[h_1(S')] = 0 \wedge H_1[h_1(S')] = 0$ */
    for Each non-prohibited S' of $N_1(S)$ or $N_2(S)$ or $N_3(S)$ **do**
        if $f(S') \geq T$ **then**
            $S \leftarrow S'$
            /* Update the hash vectors with S, Section 5.2.4 */
            $H_1[h_1(S)] \leftarrow 1$; $H_2[h_2(S)] \leftarrow 1$; $H_3[h_3(S)] \leftarrow 1$
            break;
        end if
    end for
    if $f(S) > f(S_b)$ **then**
        $S_b \leftarrow S$ /* Update the best solution S_b found during threshold search */
        $iter \leftarrow 0$
    else
        $iter \leftarrow iter + 1$
    end if
end while
return S_b

Neighborhoods and their exploration

The TSP procedure examines candidate solutions by exploring three neighborhoods induced by the popular move operators: *add*, *swap* and *drop*. Let S be the current solution and mv is one of these operators. We use $S' = S \oplus mv$ to denote a feasible neighbor solution obtained by applying mv to S and N_x ($x = 1, 2, 3$) to represent the resulting neighborhoods. To avoid the examination of unpromising neighbor solutions, TSP employs

the following dynamic neighborhood filtering strategy inspired by [LHY19; WH19]. Let S' be a neighbor solution in the neighborhood currently under examination, and S_c be the best neighbor solution encountered during the current neighborhood examination. Then S' is excluded for consideration if it is no better than S_c (i.e., $f(S') \leq f(S_c)$). By eliminating the unpromising neighbor solutions, TSP increases the efficiency of its neighborhood search.

Specifically, the associated neighborhoods induced by *add*, *swap* and *drop* are defined as follows.

— $add(p)$: This move operator expands the selected item set A by one non-selected item p from the set $\bar{A}$ such that the resulting neighbor solution is feasible. This operator induces the neighborhood N_1.

$$N_1(S) = \{S' : S' = S \oplus add(p), p \in \bar{A}\} \tag{5.2}$$

— $swap(q, p)$: This move operator exchanges a pair of items (q, p) where item q belongs to the selected item set A and p belongs to the non-selected item set $\bar{A}$ such that the resulting neighbor solution is feasible. This operator induces the neighborhood N_2.

$$N_2(S) = \{S' : S' = S \oplus swap(q, p), q \in A, p \in \bar{A}, f(S') > f(S_c)\} \tag{5.3}$$

— $drop(q)$: This operator displaces one selected item q from the set A to the non-selected item set $\bar{A}$ and induces the neighborhood N_3.

$$N_3(S) = \{S' : S' = S \oplus drop(q), q \in A, f(S') > f(S_c)\} \tag{5.4}$$

One notices that the *add* operator always leads to a better current solution with an additional eligible item, and thus the neighborhood filtering is not needed for N_1. The *drop* operator always deteriorates the quality of the current solution, and the feasibility of a neighbor solution is always ensured. The *swap* operator may either increase or decrease the objective value and the feasibility of a neighbor solution needs to be verified. For N_2 and N_3, neighborhood filtering excludes uninteresting solutions that can in no way be accepted during the TSP process.

The TSP procedure examines the neighborhoods N_1, N_2, and N_3 in a token-ring way [DS06] to explore different local optimal solutions. For N_1, as long as there exists a non-

prohibited neighbor solution, TSP selects such a neighbor solution to replace the current solution (ties are broken randomly). Once N_1 becomes empty, TSP moves to N_2, if there exists a non-prohibited neighbor solution S' satisfying $f(S') \geq T$, TSP selects S' to become the current solution and immediately returns to the neighborhood N_1. When N_2 becomes empty, TSP continues its search with N_3 and explores N_3 exactly like with N_2. When N_3 becomes empty, TSP terminates its search and returns the best solution found S_b. TSP may also terminate if its best solution remains unchanged during $IterMax$ consecutive iterations.

Operation-prohibiting mechanism

During the TSP procedure, it is important to prevent the search from revisiting a previously encountered solution. For this purpose, TSP utilizes an operation-prohibiting (OP) mechanism that is based on the tabu list strategy [GL97]. To implement the operation-prohibiting (OP) mechanism, we adopt the solution-based tabu search technique [WZ93], which has shown its effectiveness on other decision-making problems [Lai+18a; LHY19; Lai+18b]. Specifically, we employ three hash vectors H_v $(v = 1, 2, 3)$ of length L ($|L| = 10^8$) to record previously visited solutions. Given a solution $S = (x_1, \ldots, x_n)$ $(x_i \in \{0, 1\})$, we pre-compute for each item i, the weight $\mathcal{W}_i^v = i^{\gamma_v}$ $(v = 1, 2, 3)$, where γ_v is equal to $1.2, 1.6, 2.0$, respectively. Then the hash functions h_v $(v = 1, 2, 3)$ are defined as follows.

$$h_v(S) = (\sum_{i=1}^{n} \lfloor \mathcal{W}_i^v \times x_i \rfloor) \bmod |L| \tag{5.5}$$

The hash value of a neighbor solution S' from the *add*, *swap* or *drop* operator can be efficiently computed as follows ($x \in A, y \in \bar{A}$, Section 5.2.2).

$$h_v(S') = \begin{cases} h_v(S) + \mathcal{W}_y, & \text{for the } add \text{ operator} \\ h_v(S) - \mathcal{W}_x + \mathcal{W}_y, & \text{for the } swap \text{ operator} \\ h_v(S) - \mathcal{W}_x, & \text{for the } drop \text{ operator} \end{cases} \tag{5.6}$$

Starting with the hash vectors set to 0, the corresponding positions in the three hash vectors H_v is updated by 1 whenever a new neighbor solution S' is accepted to replace the current solution S (lines 12-16, Alg. 15). For each candidate neighbor solution S', its hashing value $h_v(S')$ is calculated with Equation (5.6) in $O(1)$. Then, this neighbor solution S' is previously visited if $H_1[h_1(S')] \wedge H_2[h_2(S')] \wedge H_3[h_3(S')] = 1$ and is prohibited

from consideration by the TSP procedure.

5.2.5 Crossover operator

The crossover operator generally creates new solutions by recombining two existing solutions. For the DCKP, we adopt the idea of the double backbone-based crossover (DBC) operator [ZHG18] and adapt it to the problem.

Given two solutions S^i and S^j, we use them to divide the set of n items into three subsets: the common items set $X_1 = S^i \cap S^j$, the unique items set $X_2 = (S^i \cup S^j) \setminus (S^i \cap S^j)$ and the unrelated set $X_3 = V \setminus (S^i \cup S^j)$. The basic idea of the DBC operator is to generate an offspring solution S^o by selecting all items in set X_1 (the first backbone) and some items in set X_2 (the second backbone), while excluding items in set X_3.

As shown in Algorithm 16, from two randomly selected parent solutions S^i and S^j, the DBC operator generates S^o in three steps. First, we initialize S^o by setting all the variables x_a^o $(a = 1, \ldots, n)$ to 0 (line 3, Alg. 16). Second, we identify the common items set X_1 and the unique items set X_2 (line 4-10, Alg. 16). Third, we add all items belonging to X_1 into S^o and randomly add items from X_2 into S^o until the knapsack constraint is reached (line 11-17, Alg. 16). Note that the knapsack and disjunctive constraints are always satisfied during the crossover process.

Since the DCKP is a constrained problem, the DBC operator adopted for TSBMA has several special features to handle the constraints, which is different from the DBC operator introduced in [ZHG18]. First, we iteratively add an item into S^o by selecting one item from the unique items set X_2 randomly until the knapsack constraint is reached, while each item in X_2 is considered with a probability p_0 $(0 < p_0 < 1)$ in [ZHG18]. Second, unlike [ZHG18] where a repair operation is used to achieve a feasible offspring solution, our DBC operator ensures the satisfaction of the problem constraints during the offspring generation process.

5.2.6 Population updating

Once a new offspring solution is obtained by the DBC operator in the last section, it is further improved by the threshold search procedure presented in Section 5.2.4. Then we adopt a diversity-based population updating strategy [JH16; JH19; Lai+18a] to decide whether the improved offspring solution should replace an existing solution in the population. This strategy is beneficial to balance the quality of the offspring solution and

Algorithm 16 The double backbone-based crossover operator

Input: Two parent solutions $S^i = (x_1^i, x_2^i, \ldots, x_n^i)$ and $S^j = (x_1^j, x_2^j, \ldots, x_n^j)$.
Output: An offspring solution $S^o = (x_1^o, x_2^o, \ldots, x_n^o)$.
$S^o \leftarrow \emptyset$ /* Initialize S^o (i.e., $f(S^o) = 0$)*/
for $a \leftarrow 1$ **to** n **do**
  if $x_a^i = 1$ and $x_a^j = 1$ **then**
    $X_1 \leftarrow a$ /* X_1 is the common items set */
  else if $x_a^i = 1$ or $x_a^j = 1$ **then**
    $X_2 \leftarrow a$ /* X_2 is the unique items set */
  end if
end for
$S^o \leftarrow X_1$ /* Add all items belonging to X_1 into S^o */
Randomly shuffle all items in X_2;
for each $a \in X_2$ **do**
  if $S^o \cup (x_a^o = 1)$ is a feasible solution **then**
    $x_a^o \leftarrow 1$ /* The second backbone */
  end if
end for
return S^o

its distance from the population.

To accomplish this task, we temporarily insert the improved offspring solution into the population and compute the distance (Hamming distance) between any two solutions in the population. Then we obtain the goodness score of each solution in the same way as proposed in [Lai+18a]. Finally, the worst solution in the population is identified according to the goodness score and deleted from the population.

5.2.7 Time complexity

As shown in Section 5.2.3, the population initialization procedure includes two steps. Given a DCKP instance with n items, the first step of random selection takes time $O(n)$. Given an input solution $S =< A, \bar{A} >$ (see Section 5.2.2), the complexity of one iteration of the TSP procedure is $O((n + |A| \times |\bar{A}|))$. Then the second step of the initialization procedure can be realized in $O([(n + |A| \times |\bar{A}|)] \times IterMax)$, where $IterMax$ is set to $2n$ in the initialization procedure. The complexity of the population initialization procedure is $O(n^3)$.

Now we consider the four procedures in the main loop of the TSBMA algorithm: parent selection, crossover operator, the TSP procedure and population updating. The parent selection procedure is realized in $O(1)$. The crossover operator takes time $O(n)$. The complexity of the TSP procedure is $O([(n + |A| \times |\bar{A}|)] \times IterMax)$, where $IterMax$

is determined in Section 5.2.4. The population updating procedure can be achieved in $O(n|P|)$, where $|P|$ is the population size. Then, the complexity of one iteration of the main loop of the TSBMA algorithm is $O(n^2 \times IterMax)$.

5.3 Computational results and comparisons

In this section, we assess the proposed TSBMA algorithm by performing extensive experiments and making comparisons with state-of-the-art DCKP algorithms. The benchmark instances of the DCKP tested in our experiments were widely used in the literature, which can be divided into two sets (see Section 1.3.4 for the main characteristics of these instances). We report computational results on two sets of 6340 benchmark instances.

5.3.1 Experimental settings

Reference algorithms. For the 100 DCKP instances of Set I that were widely tested by heuristic algorithms, we adopt as our reference methods three state-of-the-art heuristic algorithms: parallel neighborhood search algorithm (PNS) [QW17b], cooperative parallel adaptive neighborhood search algorithm (CPANS) [QW17a], and probabilistic tabu search algorithm (PTS) [Sal+17]. Note that PTS only reported results of the 50 instances $1Iy$ to $10Iy$, since the other 50 instances of $11Iy$ to $20Iy$ were designed later. For the 6240 DCKP instances of Set II that were only tested by exact algorithms until now, we cite the results of three best performing methods: branch-and-bound algorithms BCM [BCM17] and CFS [CFS21]) as well as the integer linear programming formulations solved by the CPLEX solver (ILP) [CFS21].

Computing platform. The proposed TSBMA algorithm was written in C++ [1] and compiled using the g++ compiler with the -O3 option. All experiments were carried out on an Intel Xeon E5-2670 processor (2.5 GHz CPU and 2 GB RAM) under the Linux operating system. The results of the main reference algorithms have been obtained on computing platforms with the following features: an Intel Xeon processor with 2×3.06 GHz for CPANS and PNS, an Intel Pentium i5-6500 processor with 3.2 GHz and 4 GB RAM for PTS, and an Intel Xeon E5-2695 processor with 3.00GHz for CFS. Note that the parallel algorithms PNS and CPANS used 10 to 400 processors to obtain the results.

1. The code of our TSBMA algorithm will be available at: http://www.info.univ-angers.fr/pub/hao/DCKP_TSBMA.html.

Parameter settings. The TSBMA algorithm does not require parameter tuning (it is parameter-free). However, for the 6240 instances of Set II (with a wide range of densities and number of items), we adjusted the threshold T (see Section 5.2.4) to $T = MinP + rand(20)$, where $MinP$ is the minimum profit value for each instance tested.

Stopping condition. For the 100 DCKP instances of Set I, the TSBMA algorithm adopted the same cut-off time as the reference algorithms (PNS, CPANS and PTS), i.e., 1000 seconds. Note that for the instances $11Iy$ to $20Iy$, PNS used a much longer limit of 2000 seconds. Given its stochastic nature, TSBMA was performed 20 times independently with different random seeds to solve each instance. For the 6240 instances of Set II, the cut-off time was set to 600 seconds as in the CFS algorithm and the number of repeated runs was set to 10.

5.3.2 Computational results and comparisons

In this section, we first present summarized comparisons of the proposed TSBMA algorithm against each reference algorithm on the 100 instances of Set I, and then show the comparative results on the 6240 DCKP instances of Set II. The detailed computational results of our algorithm and the reference algorithms on the instances of Set I are shown in the Appendix, while our solution certificates for these 100 instances are available at the webpage indicated in footnote 1. For the 6240 instances of Set II, we report their objective values at the same website.

Comparative results on the 100 benchmark instances of Set I

The comparative results of the TSBMA algorithm and each reference algorithm are summarized in Table 5.1. Column 1 indicates the pairs of compared algorithms and column 2 gives the names of instance class. Column 3 shows the quality indicators: the best objective value (f_{best}) and the average objective value (f_{avg}) (when the average results are available in the literature). The following columns #Wins, #Ties and #Losses present the number of instances for which TSBMA achieves a better, equal and worse result according to the indicators. To further analyze the performance of our algorithm, we carried out the Wilcoxon signed-rank test to verify the statistical significance of the compared results between TSBMA and each compared algorithm in terms of the f_{best} and f_{avg} values (when the average results are available in the literature). The outcomes of the Wilcoxon tests are shown in the last column, where 'NA' means that the two sets of compared results

are exactly the same.

From Table 5.1, one observes that the TSBMA algorithm competes very favorably with all the reference algorithms by reporting improved or equal results on all the instances. Compared to the probabilistic tabu search algorithm (PTS) [Sal+17] which reported results only on the first 50 instances of classes $1Iy$ to $10Iy$, TSBMA finds 8 (45) better f_{best} (f_{avg}) values, while matching the remaining results. Compared to the two parallel algorithms (PNS) [QW17b] and (CPANS) [QW17a] that reported only the f_{best} values, TSBMA obtained 35 and 29 better f_{best} results, respectively. The small *p-values* (< 0.05) from the Wilcoxon tests between TSBMA and its competitors indicate that the performance differences are statistically significant. Finally, it is remarkable that our TSBMA algorithm discovered 24 new lower bounds on the instances $11Iy$ to $20Iy$ (see the detailed results shown in the Appendix).

Table 5.1 – Summarized comparisons of the TSBMA algorithm against each reference algorithm with the *p-values* of the Wilcoxon signed-rank test on the 100 DCKP instances of Set I.

Algorithm pair	Instance	Indicator	#Wins	#Ties	#Losses	*p-value*
TSBMA vs. PTS [Sal+17]	$1Iy - 10Iy$ (50)	f_{best}	8	42	0	1.40e-2
		f_{avg}	45	5	0	5.34e-9
TSBMA vs. PNS [QW17b]	$1Iy - 10Iy$ (50)	f_{best}	9	41	0	8.91e-3
	$11Iy - 20Iy$ (50)	f_{best}	26	24	0	8.25e-6
TSBMA vs. CPANS [QW17a]	$1Iy - 10Iy$ (50)	f_{best}	0	50	0	NA
	$11Iy - 20Iy$ (50)	f_{best}	29	21	0	2.59e-6

To complete the assessment, we provide the performance profiles [DM02] of the four compared algorithms on the 100 instances of Set I. Basically, the performance profile of an algorithm shows the cumulative distribution for a given performance metric, which reveals the overall performance of the algorithm on a set of instances. In our case, the plots concern the best objective values (f_{best}) of the compared algorithms since the average results of some reference algorithms are not available in the literature. Given a set of algorithms (solvers) $\mathcal{S}$ and an instance set $\mathcal{P}$, the performance ratio is given by $r_{p,s} = \frac{f_{p,s}}{min\{f_{p,s}:s\in\mathcal{S}\}}$, where $f_{p,s}$ is the f_{best} value of instance p of $\mathcal{P}$ obtained by algorithm s of $\mathcal{S}$. The performance profiles are shown in Figure 5.2, where the performance ratio and the percentage of instances solved by each compared algorithm are displayed on the *X-axis* and *Y-axis*, respectively. When the value of *X-axis* is 1, the corresponding value of *Y-axis* indicates the fraction of instances for which algorithm s can reach the best f_{best} value of the set $\mathcal{S}$ of the compared algorithms.

From Figure 5.2, we observe that our TSBMA algorithm has a very good performance on the 100 benchmark instances of Set I compared to the reference algorithms. For the 50 instances $1Iy$ to $10Iy$, TSBMA and CPANS are able to reach 100% best f_{best} values on these 50 instances, while PTS and PNS fail on around 15% of the instances. When considering the 50 instances $11Iy$ to $20Iy$, the plot of TSBMA strictly runs above the plots of PNS and CPANS, revealing that our algorithm dominates the reference algorithms on these 50 instances. These outcomes again confirm the high performance of our TSBMA algorithm.

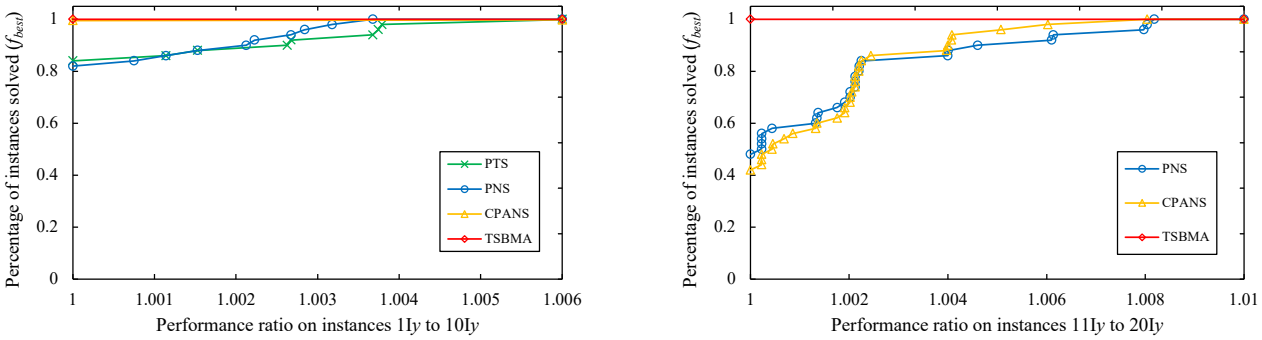


Figure 5.2 – Performance profiles of the compared algorithms on the 100 DCKP instances of Set I.

Comparative results on the 6240 benchmark instances of Set II

Table 5.2 summarizes the comparative results of our TSBMA algorithm on the 6240 instances of Set II, together with the three reference algorithms mentioned in 5.3.1. Note that three ILP formulations were studied in [CFS21], we extracted the best results of these formulations in Table 5.2, i.e., the results on instances CC and CR (conflict graph density from 0.10 to 0.90) with ILP_2 and the results on very sparse instances SC and SR (conflict graph density from 0.0001 to 0.005) with ILP_1. Columns 1 and 2 of Table 5.2 identify each instance class and the total number of instances of the class. Columns 3 to 5 indicate the number of instances solved to optimality by the three reference algorithms. Column 6 shows the number of instances for which our TSBMA algorithm reaches the optimal solution proved by exact algorithms. The number of new lower bounds (denoted by NEW LB in Table 5.2) found by TSBMA is provided in column 7. The best results of the compared algorithms are highlighted in bold. In order to further evaluate the performance of our algorithm, we summarize the available comparative results between

MSBTS and the main reference algorithm CFS in columns 8 to 10. The last three rows provide an additional summary of the results for each column.

From Table 5.2, we observe that TSBMA performs globally very well on the instances of Set II. For the 5760 CC and CR instances, TSBMA reaches most of the proved optimal solutions (5381 out of 5389) and discovers new lower bounds for 323 difficult instances whose optima are still unknown. For the 240 very sparse SC instances, TSBMA matches 195 out of 200 proved optimal solutions and finds 24 new lower bounds for the remaining instances. Although TSBMA successfully solves only 9 out of the 229 solved very sparse SR instances, it discovers 7 new lower bounds. The high performance of TSBMA is further evidenced with the comparison with the best exact algorithm CFS (last three columns).

Notice that the performance of CPLEX with ILP_1 is better than TSBMA as well as the two reference algorithms BCM and CFS on the two classes of very sparse instances (SC and SR). As analyzed in [CFS21], one of the main reasons is that the LP relaxation of ILP_1 provides a very strong upper bound, which makes the ILP_1 formulation very suitable for solving very sparse instances. The disjunctive constraints become very weak when the conflict graph is very sparse. For these two classes of instances, the pure branch-and-bound CFS algorithm is more effective on extremely sparse instances with densities up to 0.005. On the contrary, our TSBMA algorithm is more suitable for solving sparse instances with densities between 0.01 and 0.05. In fact, the new lower bounds found by TSBMA all concern instances with a density of 0.05. Finally, the TSBMA algorithm remains competitive on the 240 correlated sparse instances SC, even if the density is the smallest (0.001), which means that only the random sparse instance class SR is challenging for TSBMA.

In summary, our TSBMA algorithm is computational efficient on a majority of the 6240 benchmark instances of Set II and is able to discover new lower bounds on 354 difficult DCKP instances, whose optimal solutions are still unknown.

5.4 Analysis and discussions

In this section, we analyze two essential components of the TSBMA algorithm: the importance of the threshold search and the contribution of the operation-prohibiting mechanism. The studies in this section are based on the 50 benchmark instances $11Iy$ to $20Iy$ of Set I.

Table 5.2 – Summarized comparisons of the TSBMA algorithm against each reference algorithm on the 6240 DCKP instances of Set II.

Class	Total	$ILP_{1,2}$ [CFS21]	BCM [BCM17]	CFS [CFS21]	TSBMA		TSBMA vs. CFS		
		Solved	Solved	Solved	Solved	New LB	#Wins	#Ties	#Losses
$C1$	720	**720**	**720**	**720**	**720**	0	0	720	0
$C3$	720	584	**720**	**720**	716	0	0	716	4
$C10$	720	446	552	**617**	**617**	**91**	91	629	0
$C15$	720	428	550	**600**	**600**	**117**	117	603	0
$R1$	720	**720**	**720**	**720**	717	0	0	717	3
$R3$	720	680	**720**	**720**	**720**	0	0	720	0
$R10$	720	508	630	**670**	669	**37**	37	681	2
$R15$	720	483	590	**622**	**622**	**78**	78	641	1
SC	240	**200**	109	156	195	**24**	70	165	5
SR	240	229	154	176	9	**7**	43	8	189
Total on CC and CR	5760	4569	5201	5389	5381	323	323	5427	10
Total on SC and SR	480	429	263	332	204	31	113	173	194
Grand total	6240	4998	5424	5721	5585	354	436	5600	204

5.4.1 Importance of the threshold search

The threshold search procedure of the TSBMA algorithm is the first adaptation of the threshold accepting method to the DCKP. To assess the importance of this component, we compare TSBMA with two TSBMA variants by replacing the TSP procedure with the *first-improvement* descent procedure and *best-improvement* descent procedure. In other words, these variants (named as MA1 and MA2) use, in each iteration, the first and the best improving solution S' in the neighborhood to replace the current solution, respectively. We carried out an experiment by running the two variants to solve the 50 instances $11Iy$ to $20Iy$ with the same experimental settings of Section 5.3.1. The performance profiles of TSBMA and these TSBMA variants are shown in Figure 5.3 based on the best objective values (left sub-figure) and the average objective values (right sub-figure).

From Figure 5.3, we can clearly observe that TSBMA dominates MA1 and MA2 according to the cumulative probability obtained by the f_{best} and f_{avg} values. The plots of TSBMA strictly run above the plots of MA1 and MA2, indicating TSBMA performs always better than the two variants. This experiment implies that the adopted threshold search procedure of TSBMA is relevant for its performance.

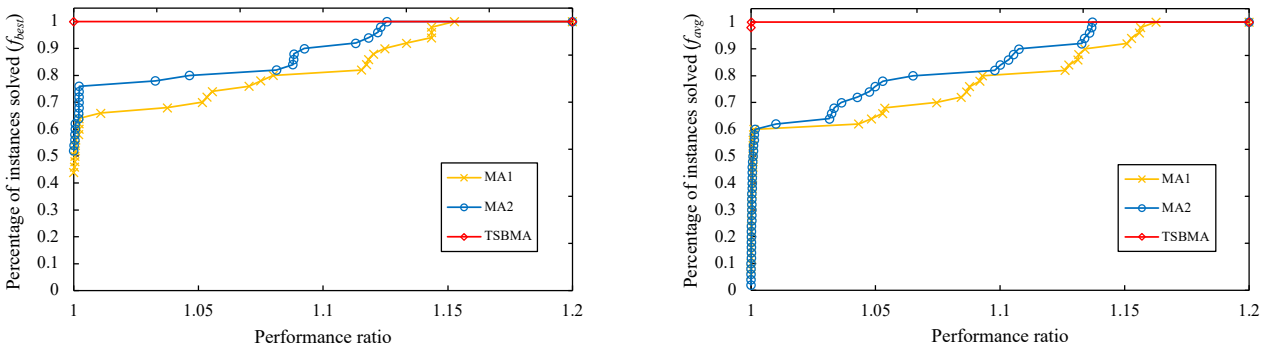


Figure 5.3 – Performance profiles of the compared algorithms on the 50 DCKP instances $11Iy$ to $20Iy$.

5.4.2 Contribution of the operation-prohibiting mechanism

TSBMA avoids revisiting previously encountered solutions with the OP mechanism introduced in Section 5.2.4. To assess the usefulness of the OP mechanism, we created a TSBMA variant (denoted by TSBMA$^-$) by disabling the OP component and keeping the other components unchanged. We ran TSBMA$^-$ to solve the 50 $11Iy$ to $20Iy$ instances according to experimental settings given in Section 5.3.1 and reported the results in Table 5.3. The first column gives the name of each instance and the remaining columns show the best objective values (f_{best}), the average objective values (f_{avg}) and the standard deviations (std). Row #Avg presents the average value of each column and row #Best indicates the number of instances for which an algorithm obtains the best values between the two sets of results. The last row shows the *p-values* from the Wilcoxon signed-rank test. The best results of the compared algorithms are highlighted in bold.

From Table 5.3, we observe that TSBMA$^-$ performs worse than TSMBA. TSBMA$^-$ obtains worse f_{best} values for 35 out of the 50 instances and worse f_{avg} values for 48 instances. Considering the std values, TSBMA$^-$ shows a much less stable performance than TSMBA. Moreover, the small *p-values* (< 0.05) from the Wilcoxon tests confirm the statistically significant difference between the results of TSMBA and TSBMA$^-$. This experiment demonstrates the effectiveness and robustness of the operation-prohibiting mechanism employed by the TSMBA algorithm.

Table 5.3 – Comparison between TSBMA$^-$ (without the OP mechanism) and TSBMA (with the OP mechanism) on the instances $11Iy$ to $20Iy$.

Instance	TSBMA$^-$			TSBMA		
	f_{best}	f_{avg}	std	f_{best}	f_{avg}	std
11*I*1	4960	4960	0.00	4960	4960	0.00
11*I*2	4940	4940	0.00	4940	4940	0.00
11*I*3	4950	4949.45	2.18	4950	**4950**	0.00
11*I*4	4930	4924	4.42	4930	**4930**	0.00
11*I*5	4920	4916.35	4.68	4920	**4920**	0.00
12*I*1	4685	4676.95	4.99	**4690**	**4687.65**	2.22
12*I*2	4670	4668.70	3.10	**4680**	**4680**	0.00
12*I*3	4690	4685.45	4.20	4690	**4690**	0.00
12*I*4	4680	4669.80	6.36	4680	**4679.50**	2.18
12*I*5	4670	4664.50	4.57	4670	**4670**	0.00
13*I*1	4525	4511.20	8.55	**4539**	**4534.80**	3.60
13*I*2	4521	4509.25	7.29	**4530**	**4528**	4.00
13*I*3	4520	4515.40	4.55	**4540**	**4531**	3.00
13*I*4	4520	4507.10	6.94	**4530**	**4529.15**	2.29
13*I*5	4530	4513.65	6.51	**4537**	**4534.20**	3.43
14*I*1	4429	4413.55	7.41	**4440**	**4440**	0.00
14*I*2	4420	4413.55	4.47	**4440**	**4439.40**	0.49
14*I*3	4420	4415.20	4.70	**4439**	**4439**	0.00
14*I*4	4420	4412.40	4.57	**4435**	**4431.50**	2.06
14*I*5	4420	4413.85	4.27	**4440**	**4440**	0.00
15*I*1	4359	4346.15	5.06	**4370**	**4369.95**	0.22
15*I*2	4359	4344.10	6.22	**4370**	**4370**	0.00
15*I*3	4359	4341.85	6.54	**4370**	**4369.25**	1.84
15*I*4	4350	4341.05	7.78	**4370**	**4369.85**	0.36
15*I*5	4360	4346.10	5.47	**4379**	**4373.15**	4.29
16*I*1	5020	5013.75	4.93	5020	**5020**	0.00
16*I*2	5010	5003.30	5.60	5010	**5010**	0.00
16*I*3	5020	5010.65	5.33	5020	**5020**	0.00
16*I*4	5020	5008.95	8.24	5020	**5020**	0.00
16*I*5	5060	5052.85	8.37	5060	**5060**	0.00
17*I*1	4730	4707.50	7.51	4730	**4729.70**	0.64
17*I*2	4716	4704.50	6.27	**4720**	**4719.50**	2.18
17*I*3	4720	4705.10	6.68	**4729**	**4723.60**	4.41
17*I*4	4722	4701.20	9.68	**4730**	**4730**	0.00
17*I*5	4720	4706.20	8.37	**4730**	**4726.85**	4.50
18*I*1	4555	4539.75	6.31	**4568**	**4565.80**	3.40
18*I*2	4540	4532.20	4.64	**4560**	**4551.40**	3.01
18*I*3	4570	4545.20	8.58	4570	**4569.40**	2.20
18*I*4	4550	4539.30	6.75	**4568**	**4565.20**	3.12
18*I*5	4550	4542.50	5.32	**4570**	**4567.95**	3.46
19*I*1	4432	4424.65	4.71	**4460**	**4456.65**	3.48
19*I*2	4443	4430.85	6.06	**4460**	**4453.25**	4.17
19*I*3	4440	4428.15	6.01	**4469**	**4462.05**	4.04
19*I*4	4450	4431.25	5.63	**4460**	**4453.20**	3.89
19*I*5	4449	4435.65	5.42	**4466**	**4460.75**	1.61
20*I*1	4364	4358.95	2.80	**4390**	**4383.20**	3.36
20*I*2	4360	4356.85	4.25	**4390**	**4381.80**	3.78
20*I*3	4370	4360.45	5.11	**4389**	**4387.90**	2.77
20*I*4	4370	4359.75	5.78	**4389**	**4380.40**	1.98
20*I*5	4366	4357.45	4.78	**4390**	**4386.40**	4.05
#Avg	4603.08	4593.13	5.56	**4614.14**	**4611.83**	1.80
#Best	15/50	2/50	-	50/50	50/50	-
p-values	2.51e-7	1.68e-9	-	-	-	-

5.5 Chapter conclusion

The disjunctively constrained knapsack problem is a well-known $\mathcal{NP}$-hard model. Given its practical significance and intrinsic difficulty, a variety of exact and heuristic algorithms have been designed for solving the problem. In this chapter, we proposed the threshold search based memetic algorithm that combines for the first time threshold search with the memetic framework.

Extensive evaluations on a large number of benchmark instances in the literature (6340 instances in total) showed that the algorithm performs competitively with respect to the state-of-the-art algorithms. Our approach is able to discover 24 new lower bounds out of the 100 instances of Set I and 354 new lower bounds out of the 6240 instances of Set II. These new lower bounds are useful for future studies on the DCKP. The algorithm also attains the best-known or known optimal results on most of the remaining instances. We carried out additional experiments to investigate the two essential ingredients of the algorithm (the threshold search technique and the operation-prohibiting mechanism).

Part III

Conclusions

Conclusions

This thesis focuses on developing effective approaches for solving two knapsack problems: the set-union knapsack problem and the disjunctively constrained knapsack problem, which have received increasing attention in recent years. As the literature review shown in **Chapter 1**, considerable progresses have been continually made since the introduction of these two problems. Meanwhile, given the $\mathcal{NP}$-hard nature and practical significance of these problems, more powerful algorithms are still needed to push the limits of existing methods. In this thesis, we aim at advancing the state-of-the-art of solving the SUKP and the DCKP effectively and robustly. Extensive experimental assessments on multiple sets of well-known benchmark instances commonly tested in the literature demonstrate that the proposed algorithms perform competitively with respect to the state-of-the-art algorithms.

In **Chapter 2**, we presented the first stochastic local search algorithm to directly operate the binary search space of the SUKP. The proposed iterated two-phase local search algorithm (I2PLS) adopts two complementary search components to achieve an appropriate balance between intensification and diversification. The local optima exploration phase (first phase) attains different local optimal solutions by performing a variable neighborhood descent search (VND) procedure and a tabu search procedure. The local optima escaping phase (second phase) examines the unexplored regions by employing a frequency-based perturbation procedure. Experimental assessments on the 30 benchmark instances confirmed the performance of the proposed I2PLS algorithm. Specifically, I2PLS is able to achieve improved best results (new lower bounds) for 18 instances and match the best-known results for the remaining 12 instances. The first computational results with the general CPLEX solver show that the optimal solutions can be reached only for 6 small instances. The VND search strategy and the frequency-based local optima escaping strategy are investigated to shed light on their influence on the performance of the proposed I2PLS algorithm.

In **Chapter 3**, after investigating the distribution of items among high-quality solutions, we observe that high-quality solutions often contain several identical items (kernel). For this reason, we designed a kernel based tabu search algorithm (KBTS) to perform

an effective examination of the search space, especially the space around the kernel solutions. The proposed KBTS algorithm integrates three complementary search components: a tabu search procedure to attain different local optimal solutions, a kernel search procedure performs an additional examination of promising regions around the local optima, a non-kernel search procedure to drive the search to a new and distant region. Evaluated on two sets of 60 benchmark instances, the proposed KBTS algorithm is demonstrated to be very competitive compared to the reference algorithms in terms of solution quality, robustness and computation time. In particular, for the large SUKP instances with at least 500 items and elements, KBTS dominated all the reference algorithms in all performance indicators. Furthermore, we also analyzed the influence of the parameters used in the proposed algorithm as well as the kernel based components to the performance of the proposed algorithm.

In **Chapter 4**, we introduced the solution-based tabu search approach to deal with the difficult binary optimization problems. The proposed multistart solution-based tabu search algorithm (MSBTS) algorithm relies on a dedicated solution-based tabu search procedure to discover high-quality solutions. MSBTS eliminates the need for tuning parameters for tabu list management by using three hash vectors and associated hash functions to record the previously encountered solutions. To escape from local optima traps, we employed a simple multistart mechanism based on a greedy randomized initialization procedure. We assessed the performance of the MSBTS algorithm in terms of solution quality, computational efficiency and robustness on the two sets of 60 benchmark instances. MSBTS performs well by finding 7 new best-known results (new lower bounds) and matching the best-known results for the remaining instances. In particular, most of the 7 instances are of large size (with 985 to 1000 items and elements), which reveals that TSBMA is effective for solving the most difficult SUKP instances. We also performed an additional time-to-target analysis (TTT) to confirm the high computational efficiency of the proposed algorithm.

In **Chapter 5**, we developed a threshold search based memetic algorithm (TSBMA) for the disjunctively constrained knapsack problem (DCKP). This is the first approach that combines the threshold based local optimization with crossover based solution recombination to solve the DCKP. The local optimization procedure relies on a multiple neighborhood threshold search procedure reinforced by an operation-prohibiting mechanism. Then the dedicated double backbone based crossover (DBC) operator is employed to generate promising offspring solutions. Extensive experimental assessments on two sets

of 6340 benchmark instances with a wide range of densities and number of items indicate that the proposed TSBMA algorithm was superior to the state-of-the-art algorithms. In particular, TSBMA is able to discover 24 new lower bounds for Set I (100 instances) and 354 new lower bounds for Set II (6240 instances), and match the best-known or known optimal results on most of the remaining instances. Furthermore, we studied the influence of the threshold search technique and the operation-prohibiting mechanism and verified the importance of these two essential ingredients to the performance of the proposed algorithm.

Perspectives

In this thesis, we proposed several effective heuristic algorithms for solving two knapsack problems. For future work, we identify the following perspectives.

For the SUKP, the following three aspects can be considered to improve the current work. First, even if the proposed algorithms apply the filtering mechanism or the sampling technique to reduce the neighborhoods, evaluating a given neighbor solution remains time-consuming, especially when the size of the instances increases. To speed up the search process, it is useful to seek some powerful techniques to reduce the complexity of neighborhood evaluation, for example streamlining techniques, new variable-fixing techniques, or pruning techniques. Second, considering the potential strong correlations of constituent elements between different items, a hybrid approach combining local search and population-based search could be helpful to break search barriers and traps. It would also be interesting to investigate mixed search strategies that explore both feasible and infeasible solutions. Third, for the KBTS algorithm proposed in Chapter 3, one can investigate other ways to obtain the kernel solution, e.g., by using frequent pattern mining technology.

For the DCKP, there are at least three possible directions for future work. First, TSBMA performed badly on most random sparse instances of SR. It would be interesting to improve the algorithm to better handle such instances. Second, the proposed approach could be further improved by designing more efficient crossover operators as well as other dedicated population updating strategies. Third, given the good performance of the adopted approach, it is worth investigating its underlying ideas to solve related problems (especially with disjunctive constraints) discussed in the introduction.

Finally, both the SUKP and the DCKP belong to the large family of knapsack prob-

lems, it would be interesting to investigate whether the proven techniques and strategies designed for these two problems remain useful for solving other variants of knapsack problem.

List of Figures

List of Tables

Part IV

Appendix

Chapter 6

Appendix

6.1 Computational results on the 100 DCKP instances of Set I

In this appendix, we report the detailed computational results of the TSBMA algorithm and the reference algorithms (PNS [QW17b], CPANS [QW17a] and PTS [Sal+17]) on the 100 DCKP instances of Set I (see Tables 6.1 and 6.2).

The first two columns of the two tables give the name of each instance and the best-known objective values (BKV) ever reported in the literature. We employ the following four performance indicators to present our results: best objective value (f_{best}), average objective value over 20 runs (f_{avg}), standard deviations over 20 runs (std), and average run time t_{avg} in seconds to reach the best objective value. However, some of the performance indicators of the reference algorithms are not available in the literature (i.e., f_{avg}, t_{avg} and std). Note that for [QW17b] (PNS) and [QW17a] (CPANS), the authors reported several groups of results obtained by using different numbers of processors (range from 10 to 400). To make a fair comparison, we take the best f_{best} value of each instance in these groups of results as the final result. We use the average of the t_{avg} values in these groups as the final average run time. The last row #Avg indicates the average value of each column. The 24 new lower bounds discovered by our TSBMA algorithm are highlighted in bold.

Table 6.1 – Computational results of the TSBMA algorithm with the reference algorithms on the 50 DCKP instances of Set I ($1Iy$ to $10Iy$).

Instance	BKV	PNS [QW17b]	CPANS [QW17a]		PTS [Sal+17]		TSBMA (this work)			
		f_{best}	f_{best}	$t_{avg}(s)$	f_{best}	f_{avg}	f_{best}	f_{avg}	std	$t_{avg}(s)$
$1I1$	2567	2567	2567	17.133	2567	2567	2567	2567	0.00	163.577
$1I2$	2594	2594	2594	12.623	2594	2594	2594	2594	0.00	19.322
$1I3$	2320	2320	2320	14.897	2320	2320	2320	2320	0.00	6.060
$1I4$	2310	2310	2310	13.063	2310	2310	2310	2310	0.00	10.969
$1I5$	2330	2330	2330	20.757	2330	2321	2330	2330	0.00	63.663
$2I1$	2118	2118	2118	21.710	2118	2115.2	2118	2117.70	0.46	330.797
$2I2$	2118	2112	2118	129.390	2110	2110	2118	2111.60	3.20	705.755
$2I3$	2132	2132	2132	23.820	2119	2112.4	2132	2132	0.00	210.108
$2I4$	2109	2109	2109	31.377	2109	2105.6	2109	2109	0.00	14.182
$2I5$	2114	2114	2114	20.040	2114	2110.4	2114	2114	0.00	99.133
$3I1$	1845	1845	1845	34.683	1845	1760.3	1845	1845	0.00	3.780
$3I2$	1795	1795	1795	107.993	1795	1767.5	1795	1795	0.00	3.029
$3I3$	1774	1774	1774	22.490	1774	1757	1774	1774	0.00	3.585
$3I4$	1792	1792	1792	27.953	1792	1767.4	1792	1792	0.00	3.275
$3I5$	1794	1794	1794	34.820	1794	1755.5	1794	1794	0.00	9.159
$4I1$	1330	1330	1330	37.307	1330	1329.1	1330	1330	0.00	1.967
$4I2$	1378	1378	1378	40.827	1378	1370.5	1378	1378	0.00	3.926
$4I3$	1374	1374	1374	100.183	1374	1370	1374	1374	0.00	2.431
$4I4$	1353	1353	1353	26.930	1353	1337.6	1353	1353	0.00	4.167
$4I5$	1354	1354	1354	81.113	1354	1333.2	1354	1354	0.00	6.196
$5I1$	2700	2694	2700	122.637	2700	2697.9	2700	2700	0.00	78.215
$5I2$	2700	2700	2700	111.160	2700	2699	2700	2700	0.00	57.300
$5I3$	2690	2690	2690	73.640	2690	2689	2690	2690	0.00	18.566
$5I4$	2700	2700	2700	130.913	2700	2699	2700	2700	0.00	52.807
$5I5$	2689	2689	2689	279.377	2689	2682.7	2689	2687.65	3.21	289.966
$6I1$	2850	2850	2850	104.623	2850	2843	2850	2850	0.00	57.997
$6I2$	2830	2830	2830	93.887	2830	2829	2830	2830	0.00	76.883
$6I3$	2830	2830	2830	203.677	2830	2830	2830	2830	0.00	157.597
$6I4$	2830	2824	2830	160.587	2830	2824.7	2830	2830	0.00	328.817
$6I5$	2840	2831	2840	112.947	2840	2825	2840	2833.10	4.22	378.393
$7I1$	2780	2780	2780	186.970	2780	2771	2780	2779.40	1.43	483.465
$7I2$	2780	2780	2780	161.117	2780	2769.8	2780	2775.50	4.97	372.935
$7I3$	2770	2770	2770	136.310	2770	2762	2770	2768.50	3.57	393.018
$7I4$	2800	2800	2800	123.957	2800	2791.9	2800	2795.50	4.97	162.060
$7I5$	2770	2770	2770	149.933	2770	2763.6	2770	2770	0.00	290.591
$8I1$	2730	2720	2730	472.153	2720	2718.9	2730	2724	4.90	484.264
$8I2$	2720	2720	2720	109.373	2720	2713.6	2720	2720	0.00	214.760
$8I3$	2740	2740	2740	112.847	2740	2731.5	2740	2739.55	1.96	207.311
$8I4$	2720	2720	2720	253.230	2720	2712	2720	2715.35	4.85	518.579
$8I5$	2710	2710	2710	115.777	2710	2705	2710	2710	0.00	67.003
$9I1$	2680	2678	2680	134.023	2670	2666.9	2680	2679.70	0.71	316.210
$9I2$	2670	2670	2670	158.397	2670	2661.7	2670	2669.90	0.44	238.149
$9I3$	2670	2670	2670	123.280	2670	2666.5	2670	2670	0.00	161.176
$9I4$	2670	2670	2670	137.690	2663	2657.3	2670	2668.90	2.49	522.294
$9I5$	2670	2670	2670	131.247	2670	2662	2670	2670	0.00	98.124
$10I1$	2624	2620	2624	244.020	2620	2613.7	2624	2621.45	1.72	348.617
$10I2$	2642	2630	2630	144.867	2630	2620.8	2630	2630	0.00	182.474
$10I3$	2627	2620	2627	198.050	2620	2614.5	2627	2621.40	2.80	326.099
$10I4$	2621	2620	2620	148.997	2620	2609.7	2620	2620	0.00	105.609
$10I5$	2630	2627	2630	170.620	2627	2617.6	2630	2629.50	2.18	307.851
#Avg	2403.68	2402.36	2403.42	112.508	2402.18	2393.26	2403.42	2402.47	0.96	179.244

Table 6.2 – Computational results and comparison of the TSBMA algorithm with the reference algorithms on the 50 DCKP instances of Set I ($11Iy$ to $20Iy$).

Instance	BKV	PNS [QW17b]	CPANS [QW17a]		TSBMA (this work)			
		f_{best}	f_{best}	$t_{avg}(s)$	f_{best}	f_{avg}	std	$t_{avg}(s)$
$11I1$	4950	4950	4950	333.435	**4960**	4960	0.00	4.594
$11I2$	4940	4940	4928	579.460	4940	4940	0.00	14.305
$11I3$	4925	4920	4925	178.400	**4950**	4950	0.00	69.236
$11I4$	4910	4890	4910	320.067	**4930**	4930	0.00	139.197
$11I5$	4900	4890	4900	222.053	**4920**	4920	0.00	100.178
$12I1$	4690	4690	4690	230.563	4690	4687.65	2.22	416.088
$12I2$	4680	4680	4680	502.600	4680	4680	0.00	224.000
$12I3$	4690	4690	4690	229.116	4690	4690	0.00	215.103
$12I4$	4680	4680	4676	367.330	4680	4679.50	2.18	256.300
$12I5$	4670	4670	4670	487.563	4670	4670	0.00	79.190
$13I1$	4533	4533	4533	395.985	**4539**	4534.80	3.60	415.880
$13I2$	4530	4530	4530	573.718	4530	4528	4.00	361.229
$13I3$	4540	4530	4540	901.620	4540	4531	3.00	498.622
$13I4$	4530	4530	4530	315.076	4530	4529.15	2.29	366.951
$13I5$	4537	4537	4537	343.240	4537	4534.20	3.43	425.064
$14I1$	4440	4440	4440	483.156	4440	4440	0.00	205.733
$14I2$	4440	4440	4440	735.505	4440	4439.40	0.49	438.190
$14I3$	4439	4439	4439	614.733	4439	4439	0.00	146.119
$14I4$	4435	4435	4434	533.908	4435	4431.50	2.06	106.389
$14I5$	4440	4440	4440	473.448	4440	4440	0.00	160.900
$15I1$	4370	4370	4370	797.125	4370	4369.95	0.22	321.296
$15I2$	4370	4370	4370	676.703	4370	4370	0.00	181.021
$15I3$	4370	4370	4370	612.792	4370	4369.25	1.84	315.575
$15I4$	4370	4370	4370	649.398	4370	4369.85	0.36	424.873
$15I5$	4379	4379	4379	678.354	4379	4373.15	4.29	359.003
$16I1$	4980	4980	4980	286.130	**5020**	5020	0.00	205.964
$16I2$	4990	4990	4980	232.825	**5010**	5010	0.00	342.824
$16I3$	5009	5000	5009	199.880	**5020**	5020	0.00	155.070
$16I4$	5000	4997	5000	831.750	**5020**	5020	0.00	86.324
$16I5$	5040	5020	5040	982.970	**5060**	5060	0.00	32.837
$17I1$	4730	4730	4721	422.640	4730	4729.70	0.64	388.541
$17I2$	4710	4710	4710	248.770	**4720**	4719.50	2.18	300.275
$17I3$	4720	4720	4720	454.317	**4729**	4723.60	4.41	343.016
$17I4$	4720	4720	4720	432.900	**4730**	4730	0.00	288.961
$17I5$	4720	4720	4720	102.468	**4730**	4726.85	4.50	366.752
$18I1$	4566	4566	4566	225.010	**4568**	4565.80	3.40	269.545
$18I2$	4550	4550	4550	288.862	**4560**	4551.40	3.01	13.884
$18I3$	4570	4570	4570	328.555	4570	4569.40	2.20	466.748
$18I4$	4560	4560	4560	511.527	**4568**	4565.20	3.12	264.931
$18I5$	4570	4570	4570	651.887	4570	4567.95	3.46	572.589
$19I1$	4460	4460	4460	506.945	4460	4456.65	3.48	459.570
$19I2$	4459	4459	4459	666.900	**4460**	4453.25	4.17	307.224
$19I3$	4460	4460	4460	608.913	**4469**	4462.05	4.04	485.550
$19I4$	4450	4450	4450	476.755	**4460**	4453.20	3.89	430.824
$19I5$	4460	4460	4460	508.730	**4466**	4460.75	1.61	40.752
$20I1$	4389	4389	4388	957.410	**4390**	4383.20	3.36	929.372
$20I2$	4390	4390	4387	756.908	4390	4381.80	3.78	299.673
$20I3$	4389	4383	4389	966.010	4389	4387.90	2.77	568.988
$20I4$	4388	4388	4380	993.630	**4389**	4380.40	1.98	657.694
$20I5$	4389	4389	4389	772.495	**4390**	4386.40	4.05	646.570
#Avg	4608.54	4606.88	4607.58	513.011	4614.14	4611.83	1.80	303.390

List of Publications

Published/accepted papers

— Zequn WEI, Jin-Kao HAO. Iterated two-phase local search for the set-union knapsack problem. *Future Generation Computer Systems* 101 (2019): 1005-1017.

— Zequn WEI, Jin-Kao HAO. Kernel based tabu search for the set-union knapsack problem. *Expert Systems with Applications* 165 (2021): 113802.

— Zequn WEI, Jin-Kao HAO. Multistart solution-based tabu search for the set-union knapsack problem. *Applied Soft Computing* 105 (2021): 107260.

Submitted papers

— Zequn WEI, Jin-Kao HAO. A threshold search based memetic algorithm for the disjunctively constrained knapsack problem. *Computers & Operations Research*, Revision, March, 2021.

Reference

[ARR07] Renata M. Aiex, Mauricio G.C. Resende, and Celso C. Ribeiro, « TTT plots: a perl program to create time-to-target plots », *in*: *Optimization Letters* 1.*4* (2007), pp. 355–366 (cit. on pp. 83, 98).

[AHM11] Hakim Akeb, Mhand Hifi, and Mohamed Elhafedh Ould Ahmed Mounir, « Local branching-based algorithms for the disjunctively constrained knapsack problem », *in*: *Computers & Industrial Engineering* 60.*4* (2011), pp. 811–820 (cit. on p. 28).

[Aru14] Ashwin Arulselvan, « A note on the set union knapsack problem », *in*: *Discrete Applied Mathematics* 169 (2014), pp. 214–218 (cit. on p. 21).

[AT17] Mustafa Avci and Seyda Topaloglu, « A multi-start iterated local search algorithm for the generalized quadratic multiple knapsack problem », *in*: *Computers & Operations Research* 83 (2017), pp. 54–65 (cit. on p. 36).

[BOS18] Adil Baykasoğlu, Fehmi Burcin Ozsoydan, and M. Emre Senol, « Weighted superposition attraction algorithm for binary optimization problems », *in*: *Operational Research* (2018), pp. 1–27 (cit. on p. 22).

[BH13] Una Benlic and Jin-Kao Hao, « Breakout local search for the quadratic assignment problem », *in*: *Applied Mathematics and Computation* 219.*9* (2013), pp. 4800–4815 (cit. on p. 39).

[BCM17] Andrea Bettinelli, Valentina Cacchiani, and Enrico Malaguti, « A branch-and-bound algorithm for the knapsack problem with conflict graph », *in*: *INFORMS Journal on Computing* 29.*3* (2017), pp. 457–473 (cit. on pp. 27, 30, 31, 121, 126).

[CT05] Paola Cappanera and Marco Trubian, « A local-search-based heuristic for the demand-constrained multidimensional knapsack problem », *in*: *INFORMS Journal on Computing* 17.*1* (2005), pp. 82–98 (cit. on p. 36).

[CPT99] Alberto Caprara, David Pisinger, and Paolo Toth, « Exact solution of the quadratic knapsack problem », *in*: *INFORMS Journal on Computing* 11.*2* (1999), pp. 125–137 (cit. on p. 16).

[CB96] William B. Carlton and J. Wesley Barnes, « A note on hashing functions and tabu search algorithms », *in*: *European Journal of Operational Research* 95.*1* (1996), pp. 237–239 (cit. on pp. 86, 90).

[CS96] Diane Castelino and Nelson Stephens, « Tabu thresholding for the frequency assignment problem », *in*: *Meta-Heuristics*, Springer, 1996, pp. 343–359 (cit. on p. 115).

[Cha+21] Jian Chang, Lifang Wang, Jin-Kao Hao, and Yang Wang, « Parallel iterative solution-based tabu search for the obnoxious p-median problem », *in*: *Computers & Operations Research* 127 (2021), p. 105155 (cit. on pp. 86, 95).

[CH14] Yuning Chen and Jin-Kao Hao, « A "reduce and solve"approach for the multiple-choice multidimensional knapsack problem », *in*: *European Journal of Operational Research* 239.*2* (2014), pp. 313–322 (cit. on p. 36).

[CH15] Yuning Chen and Jin-Kao Hao, « Iterated responsive threshold search for the quadratic multiple knapsack problem », *in*: *Annals of Operations Research* 226.*1* (2015), pp. 101–131 (cit. on pp. 36, 115).

[CH17] Yuning Chen and Jin-Kao Hao, « An iterated "hyperplane exploration" approach for the quadratic knapsack problem », *in*: *Computers & Operations Research* 77 (2017), pp. 226–239 (cit. on pp. 16, 36, 89).

[CH19] Yuning Chen and Jin-Kao Hao, « Dynamic thresholding search for minimum vertex cover in massive sparse graphs », *in*: *Engineering Applications of Artificial Intelligence* 82 (2019), pp. 76–84 (cit. on p. 115).

[Chl+18] Eden Chlamtác, Michael Dinitz, Christian Konrad, Guy Kortsarz, and George Rabanca, « The densest k-subhypergraph problem », *in*: *SIAM Journal on Discrete Mathematics* 32.*2* (2018), pp. 1458–1477 (cit. on p. 19).

[CFS21] Stefano Coniglio, Fabio Furini, and Pablo San Segundo, « A new combinatorial branch-and-bound algorithm for the Knapsack Problem with Conflicts », *in*: *European Journal of Operational Research* 289.*2* (2021), pp. 435–455 (cit. on pp. 28, 30, 31, 121, 124–126).

[CS16] Vitor Venceslau Curtis and Carlos Alberto Alonso Sanches, « An efficient solution to the subset-sum problem on GPU », *in*: *Concurrency and Computation: Practice and Experience* 28.*1* (2016), pp. 95–113 (cit. on p. 15).

[Del+19] Mauro Dell'Amico, Maxence Delorme, Manuel Iori, and Silvano Martello, « Mathematical models and decomposition methods for the multiple knapsack problem », *in*: *European Journal of Operational Research* 274.*3* (2019), pp. 886–899 (cit. on p. 16).

[DS06] Luca Di Gaspero and Andrea Schaerf, « Neighborhood portfolio approach for local search applied to timetabling problems », *in*: *Journal of Mathematical Modelling and Algorithms* 5.*1* (2006), pp. 65–89 (cit. on p. 117).

[Dıéa+17] Juan A. Dıéaz, Dolores E. Luna, José-Fernando Camacho-Vallejo, and Martha-Selene Casas-Ramıérez, « GRASP and hybrid GRASP-Tabu heuristics to solve a maximal covering location problem with customer preference ordering », *in*: *Expert Systems with Applications* 82 (2017), pp. 67–76 (cit. on p. 65).

[DM02] Elizabeth D. Dolan and Jorge J. Moré, « Benchmarking optimization software with performance profiles », *in*: *Mathematical Programming* 91.*2* (2002), pp. 201–213 (cit. on p. 123).

[DW91] Gunter Dueck and Jens Wirsching, « Threshold accepting algorithms for 0–1 knapsack problems », *in*: *Proceedings of the Fourth European Conference on Mathematics in Industry*, Springer, 1991, pp. 255–262 (cit. on p. 115).

[EC71] Samuel Eilon and Nicos Christofides, « The loading problem », *in*: *Management Science* 17.*5* (1971), pp. 259–268 (cit. on p. 16).

[FAG19] Yanhong Feng, Haizhong An, and Xiangyun Gao, « The importance of transfer function in solving set-union knapsack problem based on discrete moth search algorithm », *in*: *Mathematics* 7.*1* (2019), p. 17 (cit. on pp. 22, 36, 45–47).

[FYW19] Yanhong Feng, Jiao-Hong Yi, and Gai-Ge Wang, « Enhanced Moth Search Algorithm for the Set-Union Knapsack Problems », *in*: *IEEE Access* 7 (2019), pp. 173774–173785 (cit. on p. 22).

[FMW96] Carlos E. Ferreira, Alexander Martin, and Robert Weismantel, « Solving multiple knapsack problems by cutting planes », *in*: *SIAM Journal on Optimization* 6.*3* (1996), pp. 858–877 (cit. on p. 16).

[Fré04] Arnaud Fréville, « The multidimensional 0–1 knapsack problem: An overview », *in*: *European Journal of Operational Research* 155.*1* (2004), pp. 1–21 (cit. on pp. 17, 18).

[FH05] Arnaud Fréville and SaÏd Hanafi, « The multidimensional 0-1 knapsack problem—bounds and computational aspects », *in*: *Annals of Operations Research* 139.*1* (2005), p. 195 (cit. on p. 18).

[FH15] Zhang-Hua Fu and Jin-Kao Hao, « A three-phase search approach for the quadratic minimum spanning tree problem », *in*: *Engineering Applications of Artificial Intelligence* 46 (2015), pp. 113–130 (cit. on p. 39).

[GJ79] Michael R. Garey and David S. Johnson, *Computers and intractability: A guide to the theory of NP-completeness*, W.H. Freeman, 1979 (cit. on p. 26).

[Gav82] Bezalel Gavish, « Allocation of databases and processors in a distributed computing system », *in*: *Management of Distributed Data Processing* (1982), pp. 215–231 (cit. on p. 17).

[GL98] George Gens and Eugene Levner, « An approximate binary search algorithm for the multiple-choice knapsack problem », *in*: *Information Processing Letters* 67.*5* (1998), pp. 261–265 (cit. on p. 17).

[GG66] P.C. Gilmore and Ralph E. Gomory, « The theory and computation of knapsack functions », *in*: *Operations Research* 14.*6* (1966), pp. 1045–1074 (cit. on p. 17).

[GK96] Fred Glover and Gary A. Kochenberger, « Critical event tabu search for multidimensional knapsack problems », *in*: *Meta-heuristics*, Springer, 1996, pp. 407–427 (cit. on pp. 36, 65).

[GL97] Fred Glover and Manuel Laguna, *Tabu search*, Springer Science+Business Media New York, 1997 (cit. on pp. 43, 65, 68, 86, 90, 118).

[GO20] İlker Gölcük and Fehmi Burcin Ozsoydan, « Evolutionary and adaptive inheritance enhanced Grey Wolf Optimization algorithm for binary domains », *in*: *Knowledge-Based Systems* (2020), p. 105586 (cit. on p. 23).

[GNY94] Olivier Goldschmidt, David Nehme, and Gang Yu, « Note: On the set-union knapsack problem », *in*: *Naval Research Logistics (NRL)* 41.*6* (1994), pp. 833–842 (cit. on pp. 7, 20, 21).

[GT90] Dueck Gunter and Scheuer Tobias, « Threshold accepting: A general purpose optimization algorithm appearing superior to simulated annealing », *in*: *Journal of Computational Physics* 90.*1* (1990), pp. 161–175 (cit. on p. 115).

[GR19] Frank Gurski and Carolin Rehs, « Solutions for the knapsack problem with conflict and forcing graphs of bounded clique-width », *in*: *Mathematical Methods of Operations Research* 89.*3* (2019), pp. 411–432 (cit. on p. 28).

[Ham94] David M. Hamby, « A review of techniques for parameter sensitivity analysis of environmental models », *in*: *Environmental monitoring and assessment* 32.*2* (1994), pp. 135–154 (cit. on p. 78).

[Hao12] Jin-Kao Hao, « Memetic algorithms in discrete optimization », *in*: *Handbook of Memetic Algorithms*, Springer, 2012, pp. 73–94 (cit. on p. 113).

[He+16] Cheng He, Joseph Y.T. Leung, Kangbok Lee, and Michael L. Pinedo, « An improved binary search algorithm for the Multiple-Choice Knapsack Problem », *in*: *RAIRO-Operations Research* 50.*4-5* (2016), pp. 995–1001 (cit. on p. 17).

[HW18] Yichao He and Xizhao Wang, « Group theory-based optimization algorithm for solving knapsack problems », *in*: *Knowledge-Based Systems* (2018), p. 104445 (cit. on p. 22).

[He+18] Yichao He, Haoran Xie, Tak-Lam Wong, and Xizhao Wang, « A novel binary artificial bee colony algorithm for the set-union knapsack problem », *in*: *Future Generation Computer Systems* 78 (2018), pp. 77–86 (cit. on pp. 19, 21, 24, 36, 45, 46, 73, 95, 96).

[Hif14] Mhand Hifi, « An iterative rounding search-based algorithm for the disjunctively constrained knapsack problem », *in*: *Engineering Optimization* 46.*8* (2014), pp. 1109–1122 (cit. on p. 29).

[HM06] Mhand Hifi and Mustapha Michrafy, « A reactive local search-based algorithm for the disjunctively constrained knapsack problem », *in*: *Journal of the Operational Research Society* 57.*6* (2006), pp. 718–726 (cit. on pp. 28, 31).

[HM07] Mhand Hifi and Mustapha Michrafy, « Reduction strategies and exact algorithms for the disjunctively constrained knapsack problem », *in*: *Computers & Operations Research* 34.*9* (2007), pp. 2657–2673 (cit. on p. 27).

[HMS06] Mhand Hifi, Mustapha Michrafy, and Abdelkader Sbihi, « A reactive local search-based algorithm for the multiple-choice multi-dimensional knapsack problem », *in*: *Computational Optimization and Applications* 33.*2-3* (2006), pp. 271–285 (cit. on p. 36).

[Hif+14] 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*: *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, IEEE, 2014, pp. 1547–1551 (cit. on pp. 7, 26, 29).

[HO11] Mhand Hifi and Nabil Otmani, « A first level scatter search for disjunctively constrained knapsack problems », *in*: *2011 International Conference on Communications, Computing and Control Applications (CCCA)*, IEEE, 2011, pp. 1–6 (cit. on p. 29).

[HO12] Mhand Hifi and Nabil Otmani, « An algorithm for the disjunctively constrained knapsack problem », *in*: *International Journal of Operational Research* 13.*1* (2012), pp. 22–43 (cit. on p. 29).

[HSW14] Mhand Hifi, Sagvan Saleh, and Lei Wu, « A fast large neighborhood search for disjunctively constrained knapsack problems », *in*: *International Symposium on Combinatorial Optimization*, Springer, 2014, pp. 396–407 (cit. on p. 29).

[HSW15] Mhand Hifi, Sagvan Saleh, and Lei Wu, « A hybrid guided neighborhood search for the disjunctively constrained knapsack problem », *in*: *Cogent Engineering* 2.*1* (2015), p. 1068969 (cit. on p. 29).

[HOV94] Johannes Adzer Hoogeveen, Henricus Oosterhout, and S. L. Van de Velde, « New lower and upper bounds for scheduling around a small common due date », *in*: *Operations Research* 42.*1* (1994), pp. 102–110 (cit. on p. 15).

[HS04] Holger H. Hoos and Thomas Stützle, *Stochastic Local Search: Foundations & Applications*, Elsevier / Morgan Kaufmann, 2004 (cit. on p. 36).

[Jan99] Klaus Jansen, « An approximation scheme for bin packing with conflicts », *in*: *Journal of Combinatorial Optimization* 3.*4* (1999), pp. 363–377 (cit. on p. 26).

[JH16] Yan Jin and Jin-Kao Hao, « Hybrid evolutionary search for the minimum sum coloring problem of graphs », *in*: *Information Sciences* 352 (2016), pp. 15–34 (cit. on p. 119).

[JH19] Yan Jin and Jin-Kao Hao, « Solving the Latin square completion problem by memetic graph coloring », *in*: *IEEE Transactions on Evolutionary Computation* 23.*6* (2019), pp. 1015–1028 (cit. on p. 119).

[Kar72] Richard M. Karp, « Reducibility among combinatorial problems », *in*: *Complexity of computer computations*, Springer, 1972, pp. 85–103 (cit. on p. 14).

[KPP04] Hans Kellerer, Ulrich Pferschy, and David Pisinger, *Knapsack problems*, Springer, 2004 (cit. on pp. 7, 14–18, 20, 26).

[LHG20] Xiangjing Lai, Jin-Kao Hao, and Fred Glover, « A study of two evolutionary/tabu search approaches for the generalized max-mean dispersion problem », *in*: *Expert Systems with Applications* 139 (2020), p. 112856 (cit. on p. 65).

[Lai+18a] Xiangjing Lai, Jin-Kao Hao, Fred Glover, and Zhipeng Lü, « A two-phase tabu-evolutionary algorithm for the 0–1 multidimensional knapsack problem », *in*: *Information Sciences* 436 (2018), pp. 282–301 (cit. on pp. 36, 65, 67, 86, 93, 95, 118–120).

[LHY19] Xiangjing Lai, Jin-Kao Hao, and Dong Yue, « Two-stage solution-based tabu search for the multidemand multidimensional knapsack problem », *in*: *European Journal of Operational Research* 274.*1* (2019), pp. 35–48 (cit. on pp. 36, 42, 86, 93, 95, 117, 118).

[Lai+18b] Xiangjing Lai, Dong Yue, Jin-Kao Hao, and Fred Glover, « Solution-based tabu search for the maximum min-sum dispersion problem », *in*: *Information Sciences* 441 (2018), pp. 79–94 (cit. on pp. 86, 93, 95, 118).

[Lin+19] Geng Lin, Jian Guan, Zuoyong Li, and Huibin Feng, « A hybrid binary particle swarm optimization with tabu search for the set-union knapsack problem », *in*: *Expert Systems with Applications* 135 (2019), pp. 201–211 (cit. on pp. 23, 62, 65, 67, 71, 72, 86, 91, 92, 95, 96, 99–101).

[LLD10] Wayne Daniel Lister, R.G. Laycock, and A.M. Day, « A Key-Pose Caching System for Rendering an Animated Crowd in Real-Time », *in*: *Computer Graphics Forum* 29.*8* (2010), pp. 2304–2312 (cit. on pp. 7, 20).

[LH19] Xue-Jing Liu and Yi-Chao He, « Estimation of Distribution Algorithm Based on Lévy Flight for Solving the Set-Union Knapsack Problem », *in*: *IEEE Access* 7 (2019), pp. 132217–132227 (cit. on p. 23).

[Loj+20] Jakub Lojda, Jakub Podivinsky, Ondrej Cekan, Richard Panek, Martin Krcma, and Zdenek Kotasek, « Automatic Design of Reliable Systems Based on the Multiple-choice Knapsack Problem », *in*: *2020 23rd International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, IEEE, 2020, pp. 1–4 (cit. on p. 17).

[LMS03] Helena R. Lourenço, Olivier C. Martin, and Thomas Stützle, « Iterated local search », *in*: *Handbook of Metaheuristics*, Springer, 2003, pp. 320–353 (cit. on p. 39).

[Lu+18] Yinhao Lu, Buyang Cao, Cesar Rego, and Fred Glover, « A Tabu Search based clustering algorithm and its parallel implementation on Spark », *in*: *Applied Soft Computing* 63 (2018), pp. 97–109 (cit. on p. 86).

[MH78] Ralph Merkle and Martin Hellman, « Hiding information and signatures in trapdoor knapsacks », *in*: *IEEE Transactions on Information Theory* 24.*5* (1978), pp. 525–530 (cit. on p. 15).

[MH97] Nenad Mladenović and Pierre Hansen, « Variable neighborhood search », *in*: *Computers & Operations Research* 24.*11* (1997), pp. 1097–1100 (cit. on p. 40).

[Mon17] Douglas C. Montgomery, *Design and analysis of experiments*, John wiley & sons, 2017 (cit. on p. 78).

[Mos99] Pablo Moscato, « Memetic algorithms: A short introduction », *in*: *New Ideas in Optimization* (1999), pp. 219–234 (cit. on pp. 112, 113).

[Nau78] Robert M. Nauss, « The 0–1 knapsack problem with multiple choice constraints », *in*: *European Journal of Operational Research* 2.*2* (1978), pp. 125–131 (cit. on p. 17).

[Nav+84] Shamkant Navathe, Stefano Ceri, Gio Wiederhold, and Jinglie Dou, « Vertical partitioning algorithms for database design », *in*: *ACM Transactions on Database Systems* 9.*4* (1984), pp. 680–710 (cit. on pp. 7, 20).

[NY19] Elias David Nino-Ruiz and Xin-She Yang, « Improved Tabu Search and Simulated Annealing methods for nonlinear data assimilation », *in*: *Applied Soft Computing* 83 (2019), p. 105624 (cit. on p. 86).

[Ozs19] Fehmi Burcin Ozsoydan, « Artificial search agents with cognitive intelligence for binary optimization problems », *in*: *Computers & Industrial Engineering* 136 (2019), pp. 18–30 (cit. on p. 22).

[OB19] Fehmi Burcin Ozsoydan and Adil Baykasoglu, « A swarm intelligence-based algorithm for the set-union knapsack problem », *in*: *Future Generation Computer Systems* 93 (2019), pp. 560–569 (cit. on pp. 21, 36, 45, 46).

[Pen+16] Bo Peng, Mengqi Liu, Zhipeng Lü, Gary Kochengber, and Haibo Wang, « An ejection chain approach for the quadratic multiple knapsack problem », *in*: *European Journal of Operational Research* 253.*2* (2016), pp. 328–336 (cit. on p. 36).

[PS09] Ulrich Pferschy and Joachim Schauer, « The Knapsack Problem with Conflict Graphs », *in*: *Journal Graph Algorithms and Applications* 13.*2* (2009), pp. 233–249 (cit. on p. 27).

[PS17] Ulrich Pferschy and Joachim Schauer, « Approximation of knapsack problems with conflict and forcing graphs », *in*: *Journal of Combinatorial Optimization* 33.*4* (2017), pp. 1300–1323 (cit. on p. 27).

[PB19] Iwona Polak and Mariusz Boryczka, « Tabu Search in revealing the internal state of RC4+ cipher », *in*: *Applied Soft Computing* 77 (2019), pp. 509–519 (cit. on p. 86).

[Qin+16] Jin Qin, Xianhao Xu, Qinghua Wu, and T.C.E. Cheng, « Hybridization of tabu search with feasible and infeasible local searches for the quadratic multiple knapsack problem », *in*: *Computers & Operations Research* 66 (2016), pp. 199–214 (cit. on p. 65).

[QW17a] Zhe Quan and Lei Wu, « Cooperative parallel adaptive neighbourhood search for the disjunctively constrained knapsack problem », *in*: *Engineering Optimization* 49.*9* (2017), pp. 1541–1557 (cit. on pp. 26, 30, 31, 121, 123, 141–143).

[QW17b] Zhe Quan and Lei Wu, « Design and evaluation of a parallel neighbor algorithm for the disjunctively constrained knapsack problem », *in*: *Concurrency and Computation: Practice and Experience* 29.*20* (2017), e3848 (cit. on pp. 30, 121, 123, 141–143).

[RRV12] Celso C. Ribeiro, Isabel Rosseti, and Reinaldo Vallejos, « Exploiting run time distributions to compare sequential and parallel stochastic local search algorithms », *in*: *Journal of Global Optimization* 54.*2* (2012), pp. 405–429 (cit. on pp. 83, 98).

[Sal+17] Mariem Ben Salem, Saıd Hanafi, Raouia Taktak, and Hanêne Ben Abdallah, « Probabilistic Tabu search with multiple neighborhoods for the Disjunctively Constrained Knapsack Problem », *in*: *RAIRO-Operations Research* 51.*3* (2017), pp. 627–637 (cit. on pp. 30, 121, 123, 141, 142).

[Sal+18] Mariem Ben Salem, Raouia Taktak, A. Ridha Mahjoub, and Hanêne Ben-Abdallah, « Optimization algorithms for the disjunctively constrained knapsack problem », *in*: *Soft Computing* 22.*6* (2018), pp. 2025–2043 (cit. on p. 27).

[Sch96] Bruce Schneier, *Applied cryptography - protocols, algorithms, and source code in C, 2nd Edition*, John Wiley & Sons, 1996 (cit. on pp. 7, 20).

[SA21] Masoud Shahmanzari and Deniz Aksen, « A Multi-Start Granular Skewed Variable Neighborhood Tabu Search for the Roaming Salesman Problem », *in*: *Applied Soft Computing* 102 (2021), p. 107024 (cit. on p. 86).

[Shi79] Wei Shih, « A branch and bound method for the multiconstraint zero-one knapsack problem », *in*: *Journal of the Operational Research Society* 30.*4* (1979), pp. 369–378 (cit. on p. 17).

[SAR17] Jay Simon, Aruna Apte, and Eva Regnier, « An application of the multiple knapsack problem: The self-sufficient marine », *in*: *European Journal of Operational Research* 256.*3* (2017), pp. 868–876 (cit. on p. 16).

[SZ79] Prabhakant Sinha and Andris A. Zoltners, « The multiple-choice knapsack problem », *in*: *Operations Research* 27.*3* (1979), pp. 503–515 (cit. on p. 17).

[TKV04] Christos D. Tarantilis, Chris T. Kiranoudis, and Vassilios S. Vassiliadis, « A threshold accepting metaheuristic for the heterogeneous fixed fleet vehicle routing problem », *in*: *European Journal of Operational Research* 152.*1* (2004), pp. 148–158 (cit. on p. 115).

[Tay16] Richard Taylor, « Approximations of the densest k-subhypergraph and set union knapsack problems », *in*: *arXiv preprint arXiv:1610.04935* (2016) (cit. on p. 21).

[TM90] Paolo Toth and Silvano Martello, *Knapsack problems: algorithms and computer implementations*, John Wiley & Sons Ltd., 1990 (cit. on pp. 15, 16).

[TX16] Manghui Tu and Liangliang Xiao, « System resilience enhancement through modularization for large scale cyber systems », *in*: *2016 IEEE/CIC International Conference on Communications in China (ICCC Workshops)*, IEEE, 2016, pp. 1–6 (cit. on p. 20).

[VH01a] Michel Vasquez and Jin-Kao Hao, « A "logic-constrained" knapsack formulation and a tabu algorithm for the daily photograph scheduling of an earth observation satellite », *in*: *Computational Optimization and Applications* 20.*2* (2001), pp. 137–157 (cit. on pp. 62, 67, 69).

[VH01b] Michel Vasquez and Jin-Kao Hao, « A hybrid approach for the 0-1 multidimensional knapsack problem », *in*: *IJCAI*, 2001, pp. 328–333 (cit. on p. 36).

[Wan+13] Yang Wang, Zhipeng Lü, Fred Glover, and Jin-Kao Hao, « Backbone guided tabu search for solving the UBQP problem », *in*: *Journal of Heuristics* 19.*4* (2013), pp. 679–695 (cit. on pp. 62, 69).

[WWG17] Yang Wang, Qinghua Wu, and Fred Glover, « Effective metaheuristic algorithms for the minimum differential dispersion problem », *in*: *European Journal of Operational Research* 258.*3* (2017), pp. 829–843 (cit. on pp. 86, 93, 95, 103).

[WH19] Zequn Wei and Jin-Kao Hao, « Iterated two-phase local search for the Set-Union Knapsack Problem », *in*: *Future Generation Computer Systems* 101 (2019), pp. 1005–1017 (cit. on pp. 62, 65–67, 71, 73, 91, 92, 95–97, 99–101, 117).

[WH20a] Zequn Wei and Jin-Kao Hao, « Kernel based tabu search for the Set-union Knapsack Problem », *in*: *Expert Systems with Applications* 165 (2020), p. 113802 (cit. on pp. 7, 24, 86, 91, 92, 95, 96, 99–101, 107).

[WZ93] David L. Woodruff and Eitan Zemel, « Hashing vectors for tabu search », *in*: *Annals of Operations Research* 41.*2* (1993), pp. 123–137 (cit. on pp. 86, 90, 118).

[WH20b] Congcong Wu and Yichao He, « Solving the set-union knapsack problem by a novel hybrid Jaya algorithm », *in*: *Soft Computing* 24.*3* (2020), pp. 1883–1902 (cit. on pp. 22, 72, 96, 99–101).

[WH15] Qinghua Wu and Jin-Kao Hao, « A review on algorithms for maximum clique problems », *in*: *European Journal of Operational Research* 242.*3* (2015), pp. 693–709 (cit. on p. 42).

[YKW02] Takeo Yamada, Seija Kataoka, and Kohtaro Watanabe, « Heuristic and exact algorithms for the disjunctively constrained knapsack problem », *in*: *Journal of Information Processing Society of Japan* 43.*9* (2002) (cit. on pp. 7, 26, 28).

[YWC13] Zhen Yang, Guoqing Wang, and Feng Chu, « An effective grasp and tabu search for the 0–1 quadratic knapsack problem », *in*: *Computers & Operations Research* 40.*5* (2013), pp. 1176–1185 (cit. on p. 36).

[Zha04] Weixiong Zhang, « Configuration landscape analysis and backbone guided local search.: Part I: Satisfiability and maximum satisfiability », *in*: *Artificial Intelligence* 158.*1* (2004), pp. 1–26 (cit. on pp. 62, 69).

[Zho+20] Qing Zhou, Jin-Kao Hao, Zhe Sun, and Qinghua Wu, « Memetic search for composing medical crews with equity and efficiency », *in*: *Applied Soft Computing* (2020), p. 106440 (cit. on pp. 86, 113).

[ZHG18] Yangming Zhou, Jin-Kao Hao, and Fred Glover, « Memetic search for identifying critical nodes in sparse graphs », *in*: *IEEE Transactions on Cybernetics* 49.*10* (2018), pp. 3699–3712 (cit. on p. 119).

[ZHG17] Yi Zhou, Jin-Kao Hao, and Adrien Goëffon, « PUSH: A generalized operator for the maximum vertex weight clique problem », *in*: *European Journal of Operational Research* 257.*1* (2017), pp. 41–54 (cit. on p. 42).

[ZN08] Yunhong Zhou and Victor Naroditskiy, « Algorithm for stochastic multiple-choice knapsack problem and application to keywords bidding », *in*: *Proceedings of the 17th International Conference on World Wide Web*, 2008, pp. 1175–1176 (cit. on p. 115).

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

université angers

Titre : Algorithmes d'optimisation pour deux problèmes de sac à dos

Mot clés : Problème de sac à dos, Recherche locale, Métaheuristiques, Optimisation combinatoire.

Résumé : Cette thèse considère deux problèmes de sac à dos généralisés : le problème de sac à dos ensemble-union (SUKP) et le problème de sac à dos à contraintes disjonctives (DCKP). Ces deux problèmes sont un modèle utile pour formuler de nombreuses applications pratiques. Etant donné qu'ils appartiennent à la famille des problèmes $\mathcal{NP}$ - difficiles, il est difficile de les résoudre dans le cas général. Cette thèse est consacrée à l'avancement de l'état de l'art pour résoudre ces problèmes pertinents. Plus précisément, nous introduisons un algorithme de recherche locale en deux phases itéré, un algorithme de recherche tabou basé sur le noyau, un algorithme de recherche tabou basé sur une solution à redémarrages répétés pour résoudre le SUKP et un algorithme mémétique basé sur une recherche de seuil pour résoudre le DCKP. Des études expérimentales réalisées sur un large éventail d'instances de référence indiquent que toutes les approches proposées concurrencent favorablement les algorithmes de référence. En outre, les expériences supplémentaires montrent les rôles des ingrédients clés de nos algorithmes, y compris la stratégie d'échappement des optima locaux basée sur la fréquence, l'heuristique de recherche du noyau, la technique de recherche tabou basée sur la solution pour le SUKP et le méthode de recherche de seuil dédié pour le DCKP.

Title: Optimization algorithms for two knapsack problems

Keywords: Knapsack problems, Local search, Metaheuristics, Combinatorial optimization.

Abstract: This thesis considers two generalized knapsack problems: the set-union knapsack problem (SUKP) and the disjunctively constrained knapsack problem (DCKP). These two problems are useful models to formulate numerous practical applications. Given that they belong to the family of $\mathcal{NP}$-hard problems, it is computationally challenging to solve them in the general case. This thesis is devoted to advancing the state-of-the-art for solving these relevant problems. Specifically, we introduce an iterated two-phase local search algorithm, a kernel based tabu search algorithm, a multistart solution-based tabu search algorithm to solve the SUKP and a threshold search based memetic algorithm to solve the DCKP. Computational studies performed on a wide range of benchmark instances indicate that all the proposed approaches compete favourably with state-of-the-art algorithms. Additional experiments show the roles of the key composing ingredients of our algorithms, including the frequency-based local optima escaping strategy, the kernel search heuristic, the solution-based tabu search technique for the SUKP and the dedicated threshold search method for the DCKP.