An Approximate Dynamic Programming Approach to Discrete Optimization

by

Ramazan Demir

Submitted to the Department of Sloan School of Management
in partial fulfillment of the requirements for the degree of

PhD in Operations Research

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2000

Author .....
Department of Sloan School of Management
May 12, 2000

Certified by ....
Dimitris J. Bertsimas
Boeing Professor of Operations Research
Thesis Supervisor

Accepted by .....
Cynthia Barnhart
Associate Professor of Civil and Environmental Engineering
Co-director, Operations Research Center

An Approximate Dynamic Programming Approach to Discrete Optimization

by

Ramazan Demir

Submitted to the Department of Sloan School of Management
on May 12, 2000, in partial fulfillment of the
requirements for the degree of
PhD in Operations Research

Abstract

We develop Approximate Dynamic Programming (ADP) methods to integer programming problems. We describe and investigate parametric, nonparametric and base-heuristic learning approaches to approximate the value function in order to break the curse of dimensionality. Through an extensive computational study we illustrate that our ADP approach to integer programming competes successfully with existing methodologies including state of art commercial packages like CPLEX. Our benchmarks for comparion are solution quality, running time and robustness (i.e., small deviations in the computational resources such as running time for varying instances of same size). In this thesis, we particularly focus on knapsack problems and the binary integer programming problem. We explore an integrated approach to solve discrete optimization problems by unifying optimization techniques with statistical learning. Overall, this research illustrates that the ADP is a promising technique by providing near-optimal solutions within reasonable amount of computation time especially for large scale problems with thousands of variables and constraints. Thus, Approximate Dynamic Programming can be considered as a new alternative to existing approximate methods for discrete optimization problems.

Thesis Supervisor: Dimitris J. Bertsimas
Title: Boeing Professor of Operations Research

Acknowledgments

I would like to thank my advisor Professor Dimitris Bertsimas for his constant care and inspiration. His enthusiasm about research and his courage to tackle challenging problems made working with him a very valuable learning experience. I would also like to thank Professors Steve Graves and Tom Magnanti for serving on my committee and for providing me with very insightful comments. My sincere thanks to Professor Belá Vizvári who first introduced to me operations research field and motivated me to pursue a doctoral program.

We have a wonderful administrative team at ORC. I really appreciate their contributions to our lives. Thanks to Paulette, Laura and Danielle. I would like to thank our co-directors Professors Cynthia Barnhart and Jim Orlin for offering us such a nice quality of life at ORC. A non negligible part of the quality of my life goes to the tasty Turkish (sorry Greek) cuisine prepared by Georgia at Thanksgiving parties :-).

I would like to thank my friends for being so helpful and hospitable. My warm thanks go to Soulaymane, a great friend who constantly supported me and who added many dimensions to my life. Special thanks to Alp and Ozlem for our useful and insightful conversations at the most critical times. Thanks to Adam, Amy, Andy, Anurag, Aristide, Dessi, Dushyant, Eduardo, Fernando, Hernan, Jean-Paul, Jeremie, Jonathan C., Jonathan T., Mahesh, Marc, Marina, Martin, Nick, Omar and Sanne. I would also like to thank Beril for her valuable advises and firm support in the early phase of my doctoral program.

I am not going to forget the MIT Turkish gang for their valuable time and discussions which enlightened me constantly. It has been a pleasure to share the same apartment with Fatih for many years. Thanks to him for having a nice personality and for sharing his vision on future world. I am happy to know Aykut and I also thank him for our educative and precious conversations.

Although my parents did not have the opportunity of getting a good education, they dedicated their lives to offer me the chance of getting the best education. I am indebted to their constant support. This thesis goes to them. I would like to send

my special thanks to my brother Yusuf for his endorsement in all aspects of my life. I would also like to thank my eldest brother Bünyamin and his wife Günseli for visiting me and giving a moral support steadily. My final thanks go to my sister Hatice and her husband Mehmet for being so nice and sincere all the time. I have always enjoyed spending time with them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The aim of this thesis is to develop approximate dynamic programming methods for large scale integer programming problems. Discrete optimization problems such as resource allocation, scheduling, location and assignment problems arise in many practical domains. It is important to have methods to effectively tackle these problems. This chapter gives an overview of integer programming while providing a short description of existing methodologies. We briefly discuss dynamic programming and its limitations to motivate us for the approximate dynamic programming methodology development. We close the introduction with the thesis's contributions and its structure.

1.1 Integer Programming

In discrete optimization we are given a finite set E, and a set $\mathcal{I}$ of subsets of E and a function $c : E \to \Re$. For each set $F \subset E$ let $c(F) := \sum_{e \in F} c(e)$. Our objective is to find a set $I^* \in \mathcal{I}$ with

$$c(I^*) = \max\{c(I) | I \in \mathcal{I}\}.$$

We denote this discrete optimization problem by $(E, \mathcal{I}, c)$. Generally speaking, discrete optimization problems are all combinatorial in the sense that an optimal solution is some subset of a finite set. Thus, in principle these problems can be solved

by enumeration. However the number of feasible solutions can be very large, typically exponential in the size of the set E, hence enumerative methods are not practical.

In this thesis, our basic problem is the *binary integer program* (BIP):

$$\begin{aligned} \text{maximize} \quad & \sum_{j=1}^{n} c_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} a_{ij} x_j \leq b_i, \quad i = 1, \cdots, m, \\ & x_j \in \{0,1\}, \qquad j = 1, \cdots, n. \end{aligned}$$

The above problem has m constraints and n variables. The data is a_{ij}, b_i and c_j. We use the notation $\mathbf{A}_j = (a_{1j}, \cdots, a_{mj})'$ to denote the vector corresponding to the j-th variable. The column vector $\mathbf{b} = (b_1, \cdots, b_m)'$ whose i-th component is b_i is referred to as the *right-hand side vector* and $\mathbf{c} = (c_1, \cdots, c_n)'$. The vector $\mathbf{x}'$ is used for transpose of the vector $\mathbf{x}$. Then the problem can be written as follows:

$$\begin{aligned} \text{maximize} \quad & \mathbf{c}'\mathbf{x} \\ \text{subject to} \quad & \sum_{j=1}^{n} \mathbf{A}_j x_j \leq \mathbf{b} \\ & x_j \in \{0,1\}, \qquad j = 1, \cdots, n. \end{aligned}$$

An important special case when $a_{ij}, b_i, c_j \geq 0$ for $i = 1, \cdots, m$ and $j = 1, \cdots, n$ is called the multi-dimensional knapsack problem, or simply the binary knapsack problem if $m = 1$.

A wide variety of complex problems can be formulated as integer programs. As a result, there is a very large literature addressing both applications and solution techniques. Integer programming problems belong to family of NP-hard problems, meaning that it is very unlikely that we ever can devise polynomial algorithms for these problems. Combinatorial algorithms, enumerative algorithms (e.g. branch-and-bound and dynamic programming) and approximate algorithms have been utilized

to solve many of these problems (see Nemhauser and Wolsey [61], Wolsey [91] and Walukiewicz [87] for a thorough discussion). In almost all cases, successful solution strategies use structural properties of the problem at hand.

Due to the challenge of finding an optimal solution especially for large-scale discrete optimization problems, a large amount of work has been conducted on heuristic methods for solving these problems. These heuristics typically find good solutions at a reasonable computational cost without being able to guarantee optimality. Good feasible solutions are very valuable for practical problems. It is not an easy task to design good approximate algorithms. Most of the approximate algorithms can be classified as greedy, random search, relaxation, partitioning, partial enumeration, local search methods (see Ibaraki [39] for a nice discussion). On the other extreme, many heuristics are problem-specific, so that a method which behaves well for one problem cannot be utilized to solve a different one. Recently, meta-heuristics like genetic algorithms and tabu search heuristics have been in the center of interest to overcome some of the limitations of stand-alone heuristics (see Reeves [70] and Glover and Laguna [35] for a discussion and other references).

1.2 Curse of Dimensionality

Dynamic programming (DP) is an enumerative algorithm for discrete optimization problems. In almost all cases, when using dynamic programming (DP) to solve discrete optimization problems, the computational and data storage requirements are overwhelming. Generally, the number of states in this approach is exponential in the number of input parameters and a large memory space is necessary to store the associated information. This is known as the *curse of dimensionality* in dynamic programming methods (see Bellman [5], Bellman and Dreyfus [6], Nemhauser [60], Smith [79], Denardo [21], and Cooper and Cooper [17] for a discussion). Thus, generally speaking exact dynamic programming is not a practical methodology par-

ticularly for large-scale optimization problems.

Many ingenious approaches such as Lagrange multiplier, successive approximation, function approximation (e.g. neural networks, radial basis representation, polynomial representation) methods have been proposed to break the curse of dimensionality while contributing diverse approximate dynamic programming methodologies to the literature (see for example Bellman [5], Morin [57], and Cooper and Cooper [17]).

Motivated by the need to address the curse of dimensionality several researches have proposed approximations in dynamic programming. The collection of such methods is broadly known as Approximate Dynamic Programming (ADP). A breakthrough result that initiated many of the developments in ADP is the work of Tesauro [81, 82], who designed a *backgammon program* that is competitive at a grand-master level by using neural network methods. Functional approximation like neural networks ideas were used in Kleywegt et. al. [41] for inventory routing problems, and in Van Roy et. al. [71] for inventory management problems. Powell and Shapiro [69] introduced a dynamic programming reformulation for dynamic resource allocation for fleet management problems and used ADP methodology effectively to generate very good solutions. Bertsekas and Tsitsiklis [8] develop a framework of approximate dynamic programming primarily in the the context of stochastic optimization and include several case studies in various fields. Sarkar et. al. [74] used greedy or approximate algorithms for machine scheduling problems in a look-ahead search mechanism in order to alleviate the performance of the stand-alone greedy or approximate methodology while Bertsimas et. al. [11] applied a similar idea for location problems. See for other applications such as two-stage maintenance and repair problem in Bertsekas et. al. [9], stochastic scheduling problems in Bertsekas and Castanon [7], sequencing and stochastic vehicle routing in Secomandi [76], railroad scheduling in Christodouleas [14], deterministic supply chain problem in Wike [90], revenue management in Bertsimas and Popescu [10].

In this thesis, we develop an ADP approach to integer programming problems.

1.3 Contributions

We develop Approximate Dynamic Programming (ADP) methods to integer programming problems. We describe and investigate parametric, nonparametric and base-heuristic learning methods to approximate the value function in order to break the curse of dimensionality.

Base heuristic learning is a promising methodology for knapsack and binary integer programming problems, particularly randomly generated problems. It provides a flexible framework as it works with a given base-heuristic while improving its performance often significantly. Our computational evidence suggests that base-heuristic learning does have a robust performance as we observe small deviations in computation time, solution quality and storage requirements as instances change.

We propose a stand-alone adaptive fixing heuristic for the multi-dimensional knapsack problem. Adaptive fixing is computationally practical and generates good solutions compared to some heuristic methodologies from the literature. We observe that ADP with adaptive fixing heuristic is the most promising methodology for randomly generated multi-dimensional knapsack problems compared to existing methodologies and a state-of-art commercial package like CPLEX. It generates near-optimal solutions for large-scale multi-dimensional knapsack problems with thousands of variables and constraints.

We develop a pivot-fix-and-complement heuristic methodology for binary integer programming problems. In essence, it is as an enhancement to pivot-and-complement heuristic from literature. We introduce new pivoting rules, a new basic variable fixing and a new complementing scheme. Our computational evidence suggests that pivot-fix-and-complement is computationally practical for large randomly-generated

binary integer programming problems with thousands of variables. We also illustrate its effectiveness on the MIPLIB test problems.

Our computational study suggests that statistical learning is a computationally practical methodology for knapsack problems. It generates competitive solutions compared to some heuristic methodologies from the literature but in general it is weaker than the base-heuristic learning. It needs tuning in its selection of the sampling scheme, the functional form for parametric approximation, and the kernel and the bandwidth for nonparametric approximation.

In summary, through our extensive computational study we illustrate that our ADP approach to integer programming competes successfully with existing methodologies including state of art commercial package like CPLEX. Our benchmarks for comparison are solution quality, running time and robustness, judged by small deviations in computational resources and solution quality. Overall, this research illustrates that ADP is a promising technique by providing near-optimal solutions within reasonable amount of time especially for large scale problems with thousands of variables and constraints. Thus, Approximate Dynamic Programming can be considered as a new alternative to existing approximate methods for discrete optimization problems.

1.4 Thesis Structure

In Chapter 2 we reformulate the binary integer program as a dynamic program while introducing the basic definition and notations. After discussing the limitations of dynamic programming we present a survey on approximate methods. We close this chapter by providing our approximate dynamic programming methodology with both statistical and base-heuristic learning frameworks.

Chapter 3 customizes generic approximate dynamic programming methodolo-

gies to the binary knapsack problem. We briefly discuss existing approaches to this problem. After describing approximate dynamic programming, we provide extensive computational results to demonstrate the viability of our approach.

We address the multi-dimensional knapsack problem in Chapter 4 and the binary integer programming problem in Chapter 5. We offer our conclusions in Chapter 6 and present future research directions.

Chapter 2

Approximate Dynamic Programming

The aim of this chapter is to reformulate a binary integer program as dynamic program while introducing the basic definitions and notations that will be used in the following chapters. After introducing the basic ideas behind Approximate Dynamic Programming (ADP) we motivate the reader by describing generic algorithmic frameworks. We provide a detailed description of ADP frameworks while we will be focusing on specific applications in the next chapters.

2.1 The Dynamic Programming Formulation

We consider the following binary integer program $BIP = BIP(n, \mathbf{b}_0)$ with n variables and a right-hand-side $\mathbf{b}_0$:

$$\begin{aligned} z^* = \quad \text{maximize} \quad & \sum_{j=1}^{n} c_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} \mathbf{A}_j x_j \leq \mathbf{b}_0 \\ & x_j \in \{0,1\}. \end{aligned}$$

Let us denote by z^* the optimal solution value and $\mathbf{x}^*$ an optimal solution. In order to reformulate BIP as a dynamic program we consider the following subproblem $BIP(k, \mathbf{b})$ which includes the first k variables with the right-hand side $\mathbf{b}$:

$$\begin{aligned} F_k(\mathbf{b}) = \quad & \text{maximize} \quad \sum_{j=1}^{k} c_j x_j \\ & \text{subject to} \quad \sum_{j=1}^{k} \mathbf{A}_j x_j \leq \mathbf{b} \\ & \qquad\qquad x_j \in \{0, 1\}. \end{aligned}$$

Let $F_k(\mathbf{b})$ be the optimal solution value and $x_k(\mathbf{b})$ be an optimal solution for the subproblem $BIP(k, \mathbf{b})$ for k varying in $[1, n]$ and varying right-hand side $\mathbf{b}$. Using dynamic programming we have the following recursion:

- If subproblem $BIP(k, \mathbf{b})$ is *infeasible*, then we set $F_k(\mathbf{b}) = -\infty$.
- Initialization:

$$\begin{aligned} F_1(\mathbf{b}) = \quad & \text{maximize} \quad c_1 x_1 \\ & \text{subject to} \quad \mathbf{A}_1 x_1 \leq \mathbf{b} \\ & \qquad\qquad x_1 \in \{0, 1\}. \end{aligned}$$

- Dynamic Programming Recursion:

$$F_k(\mathbf{b}) = \max \{F_{k-1}(\mathbf{b}), F_{k-1}(\mathbf{b} - \mathbf{A}_k) + c_k\} \text{ for } k = 2, \cdots, n.$$

In the optimal solution $x_k(\mathbf{b})$ to $BIP(k, \mathbf{b})$ we either set $x_k = 0$ or $x_k = 1$. In the first case, the objective value is equal to the optimal value for $BIP(k-1, \mathbf{b})$. In the second case, the objective value is equal to the optimal value for the subproblem $BIP(k-1, \mathbf{b} - \mathbf{A}_k)$ plus the additional value c_k upon setting $x_k = 1$. Taking the maximum of the preceding objective values gives us the optimal value to the subproblem $BIP(k, \mathbf{b})$. Due to nature of dynamic programming recursion, optimal solutions $x_{k-1}(\mathbf{b})$ and $x_{k-1}(\mathbf{b} - \mathbf{A}_k)$ are available to the subproblems $BIP(k-1, \mathbf{b})$ and $BIP(k-1, \mathbf{b}-\mathbf{A}_k)$, respectively. We set $x_k(\mathbf{b})$ to $(x_{k-1}(\mathbf{b}), 0)$ or $(x_{k-1}(\mathbf{b}-\mathbf{A}_k), 1)$

depending on the value to which x_k is fixed in the dynamic programming recursion (i.e., $x_k(\mathbf{b}) = (x_{k-1}(\mathbf{b} - \mathbf{A}x_k^*), x_k^*)$ where $x_k^* = \text{argmax}_{x\in\{0,1\}}\{F_{k-1}(\mathbf{b} - \mathbf{A}_k x) + c_k x\}$). At the end of dynamic programming forward phase we obtain $x_n(\mathbf{b}_0)$ which is the optimal solution to the problem $BIP(n, \mathbf{b}_0)$. We can also construct an optimal solution once the optimal values $F_k(\mathbf{b})$ are known for the subproblems $BIP(k, \mathbf{b})$ for the $(k, \mathbf{b})$ combinations by a *Dynamic Programming Backward Phase* as described in Table 2.1.

DP Backward Phase
1: for $k = n, \cdots, 2$ do - $x_k^* = \text{argmax}_{x\in\{0,1\}}\{F_{k-1}(\mathbf{b} - \mathbf{A}_k x) + c_k x\}$ - $\mathbf{b} \leftarrow \mathbf{b} - \mathbf{A}_k x_k^*$ 2: x_1^* solves $BIP(1, \mathbf{b})$ 3: Output: $\mathbf{x}^*$

Table 2.1: Dynamic Programming Backward Phase.

Let us next analyze the computational complexity of the method to find the optimal value $F_n(\mathbf{b}_0)$ and an optimal solution $x_n(\mathbf{b}_0)$ for the problem $BIP(n, \mathbf{b}_0)$. We note that $z^* = F_n(\mathbf{b}_0)$ and $x^* = x_n(\mathbf{b}_0)$. Let $b_{l,i} = b_{0,i} - \sum_{a_{ij}>0} a_{ij}$, $b_{u,i} = b_{0,i} - \sum_{a_{ij}<0} a_{ij}$ and $\delta_i = b_{u,i} - b_{l,i}$. It is easy to see that exact dynamic programming requires $O(n(b^*)^m)$ calculations where $b^* = \max\{\delta_1, \cdots, \delta_m\}$.

2.2 Literature Review

Memory and computational requirements limit the practical use of dynamic programming even though in principle optimization problems can be solved by this methodology. Due to its flexible framework a number of ingenious approaches have been devised to address this practical limitation. We briefly describe various approaches

to overcome the curse of dimensionality for deterministic finite stage dynamic programs (see Puterman [57] and its references for further discussion).

Fathoming:

The basic idea is to eliminate states in dynamic programming by using bound pruning. State elimination, or *fathoming*, occurs if it is demonstrated that no completion of an optimal sub-policy for a state can lead an optimal policy. Memory requirements are reduced since fathomed states are not stored any more. In essence, this reduction is a procedure that eliminates the states which have no real influence on the optimal solution (see Marsten and Morin [48, 49] and Morin and Marsten [59]). Even though state elimination reduces memory requirements, we need to search whether a state is fathomed or not at the preceding stage of the dynamic programming recursion.

Reaching:

Denardo [21] introduced the *reaching* method in order to realize the full computational advantages of the state elimination method. Reaching finds optimal values of states that can be reached from *unfathomed* states by a label correcting algorithm. Thus, the optimal values for those states that can be reached only from *fathomed* states are not calculated. This reduces the computation requirements in comparison to conventional dynamic programming (see Denardo [21]).

Base Function Approximation:

Let $F : \Omega \rightarrow \Re$ be optimal value function where Ω is the state space. A classical technique can be utilized to approximate the optimal value function F by a linear combination of known functions $\phi_k : \Omega \rightarrow \Re$. For state $y \in \Omega$ we have

$$F(y) = \sum_{k=1}^{K} a_k \phi_k(y).$$

The main motivation is that if it is possible to find such functions $\phi_k,\ k = 1, \cdots, K$

then instead of storing $F(y)$ for each state y we only need to store the K parameters (i.e., $a_k,\ k = 1, \cdots, K$). Typically, Legendre polynomials and spline functions have been exploited as base functions (see Daniel [19], Morin [57] and Mond and Shisha [56]). However this method works best when the optimal value function does satisfy some smoothness properties. Hence, for discrete optimization problems the success might be very limited.

Multiplier Approach:

A variant of methods based on Lagrange multiplier, surrogate and dual methods have been utilized particularly for multi-dimensional state dynamic programs. The basic idea is to transform the multi-dimensional state to a one-dimensional state by pricing out the states using Lagrange multipliers or by aggregating using surrogate multipliers. It is hoped that the one-dimensional case generates a good solution. To improve the solution multipliers can be tuned iteratively. The success of such methods is problem specific and limited (see [57] for references).

Imbedded State Space:

Morin and Esogbue [58] introduced the *imbedded state space* trying to exploit the special structure of the optimal value function. For instance many resource allocation problems have an optimal value function which is a nondecreasing step function. By characterizing the domain set of its discontinuities, we can calculate optimal values by finding the corresponding region in the state space. This special structure might reduce the computational burden whereas it is impractical for large multi-dimensional states.

Other Approaches:

Many other approaches like successive approximation, decomposition techniques, nearest neighbor and state increment techniques have been utilized with some success. The successful methodologies use structural properties of the problem at hand. Even though these enhancements improve conventional dynamic programming, in

essence the successful applications assume usually one-dimensional state, smooth optimal value functions.

Our Approach:

We follow the basic idea to approximate value function in order to overcome the curse of dimensionality. We explored parametric, nonparametric and base-heuristic learning as value function approximation. We define by $H_k(\mathbf{b})$ an approximate value or estimate of the optimal value $F_k(\mathbf{b})$. Once we obtain estimates of optimal values through an approximation scheme we can utilize Dynamic Programming Backward Phase (see Table 2.1) to construct a suboptimal solution. We will describe how we calculate $H_k(\mathbf{b})$ in the following sections depending on the approximation scheme considered.

2.3 Base-Heuristic Learning

A suboptimal methodology that solves the subproblem $BIP(k, \mathbf{b})$ can be exploited to estimate the optimal value $F_k(\mathbf{b})$. We name this suboptimal methodology as *base-heuristic*. We define by $BH(k, \mathbf{b})$ a base-heuristic for the subproblem $BIP(k, \mathbf{b})$. Furthermore, let $x_{BH}(k, \mathbf{b})$ be the heuristic solution and $H_k(\mathbf{b})$ the heuristic value, i.e., an estimate of $F_k(\mathbf{b})$. If the subproblem $BIP(k, \mathbf{b})$ is infeasible, we set $H_k(\mathbf{b}) = -\infty$. We assign $H_1(\mathbf{b}) = F_1(\mathbf{b})$ since $BIP(1, \mathbf{b})$ is an one-dimensional problem, meaning that it is straightforward to optimize.

We construct a suboptimal solution as described in the *Base-heuristic Learning Backward Phase* (see Table 2.2). We calculate $H_k(\mathbf{b})$ by applying a base-heuristic $BH(k, \mathbf{b})$ to the subproblem $BIP(k, \mathbf{b})$ for varying $(k, \mathbf{b})$ combinations.

We provide several application specific enhancements such as problem reduction and variable fixing to the above generic base-heuristic learning framework while we will be focusing on the applications in the next chapters.

Base-heuristic Learning Framework
1: for $k = n, \cdots, 2$ do - $x_k^{adp} = \text{argmax}_{x \in \{0,1\}}\{H_{k-1}(\mathbf{b} - \mathbf{A}_k x) + c_k x\}$ - $\mathbf{b} \leftarrow \mathbf{b} - \mathbf{A}_k x_k^{adp}$ 2: x_1^{adp} solves $BIP(1, \mathbf{b})$ 3: Output: $\mathbf{x}^{adp}$

Table 2.2: Base-heuristic Learning Framework.

2.4 Statistical Learning

In this section, we describe a statistical learning framework for the knapsack problems.

2.4.1 Notations and Definitions

Let $\mathbf{b}_0$ be the *right-hand side* of the problem $BIP(n, \mathbf{b}_0)$. Let $b_{l,i} = b_{0,i} - \sum_{a_{ij}>0} a_{ij}$, $b_{u,i} = b_{0,i}$. We define by $\Omega = \{\mathbf{b} = (b_1, \cdots, b_m) | b_{l,i} \leq b_i \leq b_{u,i}\}$ the state space. Let $S = \{\mathbf{b}^i \in \Omega | i = 1, \cdots, s\}$ denote a sample of the state space Ω. Define $\Omega_k = \{(k, \mathbf{b}) | \mathbf{b} \in \Omega\}$ and $S_k = \{(k, \mathbf{b}) | \mathbf{b} \in S\}$ for k varying in $[1, n]$. We denote by $L(x, y)$ a loss function that measures the deviation between x and y. A typical example is $L(x, y) = (x - y)^2$. Let $d(\mathbf{x}, \mathbf{y})$ denote a distance function between vectors $\mathbf{x}, \mathbf{y} \in \Re^m$. A typical distance function is Euclidean distance $d_E(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{k=1}^{m}(x_k - y_k)^2} \in \Re$. As another example $d_I(\mathbf{x}, \mathbf{y}) = (|x_1 - y_1|, \cdots, |x_m - y_m|) \in \Re^m$ measures elementwise deviation. Let $K : \Re \rightarrow \Re$ be a one-dimensional kernel function, whereas $K_m : \Re^m \rightarrow \Re$ be an multi-dimensional kernel function.

2.4.2 Parametric Approximation

Let $F_k : \Omega_k \rightarrow \Re$ be the optimal value function where Ω_k is the state space. A classical technique can be utilized to approximate the optimal value function F_k by a known function $g : \Omega_k \rightarrow \Re$. For state $(k, \mathbf{b}) \in \Omega_k$ we approximate $F_k(\mathbf{b})$ by:

$$H_k(\mathbf{b}) = g(\boldsymbol{\theta}_k, \mathbf{b}).$$

The main motivation is that if it is possible to find such a function g then instead of storing $F_k(\mathbf{b})$ for each state $(k, \mathbf{b})$ we only need to store parameters $\boldsymbol{\theta}_k$ for k varying in $[1, n]$. We name this functional representation as *parametric approximation.*

The above model can be trained to minimize the following unweighted training criterion

$$C = \sum_{i=1}^{s} L(g(\boldsymbol{\theta}_k, \mathbf{b}^i), F_k(\mathbf{b}^i)), \tag{2.1}$$

where the $F_k(\mathbf{b}^i)$ are the output values corresponding to $\mathbf{b}^i \in S$, $\boldsymbol{\theta}_k$ is the parameter vector for the parametric model $H_k(\mathbf{b}) = g(\boldsymbol{\theta}_k, \mathbf{b})$ and $L(H_k(\mathbf{b}^i), F_k(\mathbf{b}^i))$ is a general loss function for predicting $H_k(\mathbf{b}^i)$ when the training data is $F_k(\mathbf{b}^i)$. Often the least squares criterion is used for the loss function $L(H_k(\mathbf{b}^i), F_k(\mathbf{b}^i)) = (H_k(\mathbf{b}^i) - F_k(\mathbf{b}^i))^2$ leading to the training criterion:

$$C = \sum_{i=1}^{s} (g(\boldsymbol{\theta}_k, \mathbf{b}^i) - F_k(\mathbf{b}^i))^2.$$

Typically $g(\boldsymbol{\theta}_k, \mathbf{b})$ is a linear function of the form $\boldsymbol{\alpha}_k' \mathbf{b} + \beta_k$ where $\boldsymbol{\theta}_k = (\boldsymbol{\alpha}_k, \beta_k)$ and $\boldsymbol{\alpha}_k = (\alpha_{k1}, \cdots, \alpha_{km})'$. We call the minimizer $\boldsymbol{\theta}_k^*$ of the above criterion in Equation (2.1) under a certain loss function as the *functional parameter* or as the *representative information.* Thus, for a query point $\mathbf{b}$, $F_k(\mathbf{b})$ is estimated by $H_k(\mathbf{b}) = g(\boldsymbol{\theta}_k^*, \mathbf{b})$.

2.4.3 Local Parametric Approximation

The parametric approximation tunes the model parameters by giving equal emphasis on the sample points. The sample set S can be tailored to a query point $\mathbf{b}$ by emphasizing nearby sample points in the training. We can do this by weighting the training criterion:

$$C = \sum_{i=1}^{s} L(g(\boldsymbol{\theta}_k, \mathbf{b}^i), F_k(\mathbf{b}^i)) w(\mathbf{b}, \mathbf{b}^i), \tag{2.2}$$

where $w(\mathbf{b}, \mathbf{b}^i)$ measures the emphasis of the sample point $\mathbf{b}^i$ to a query point $\mathbf{b}$. In our settings, we use $w(\mathbf{b}, \mathbf{b}^i) = K_m(d(\mathbf{b}, \mathbf{b}^i))$ for a distance function $d(\mathbf{b}, \mathbf{b}^i) = (|b_1 - b_1^i|, \cdots, |b_m - b_m^i|) \in \Re^m$ between the query point $\mathbf{b}$ and each sample point $\mathbf{b}^i$ and for a kernel function $K_m : \Re^m \rightarrow \Re$. In essence, the weighting scheme $w(\cdot)$ tailors the sample set S to a local sample set $S(\mathbf{b}) = \{\mathbf{b}^i \in S | w(\mathbf{b}, \mathbf{b}^i) \neq 0\}$. Thus our weighted training criterion can be written as:

$$C(\mathbf{b}) = \sum_{i \in S(\mathbf{b})} L(g(\boldsymbol{\theta}_k(\mathbf{b}), \mathbf{b}^i), F_k(\mathbf{b}^i)). \tag{2.3}$$

The model parameter $\boldsymbol{\theta}_k(\mathbf{b})$ is used instead $\boldsymbol{\theta}_k$ to emphasize the dependency on the query point $\mathbf{b}$. Using this training criterion, the parametric model $g(\boldsymbol{\theta}_k, \mathbf{b})$ becomes a local parametric model $g(\boldsymbol{\theta}_k(\mathbf{b}), \mathbf{b})$. Hence, we name this approximation scheme as the *local parametric approximation*. In the local parametric approximation scheme $F_k(\mathbf{b}^i)$, $\forall$ S are the *representative information* meaning that once we obtain them we find $\boldsymbol{\theta}_k^*(\mathbf{b})$ that minimizes $C(\mathbf{b})$ in Equation (2.3) and calculate the estimate of $F_k(\mathbf{b})$ by $g(\boldsymbol{\theta}_k^*(\mathbf{b}), \mathbf{b})$ for a query point $\mathbf{b}$.

2.4.4 Nonparametric Approximation

Suppose we have the optimal values $\{F_k(\mathbf{b}^1), \cdots, F_k(\mathbf{b}^s)\}$ for $\mathbf{b}^i \in S$. In *nonparametric approximation*, we approximate the optimal value $F_k(\mathbf{b})$ of a query point $\mathbf{b}$ by $H_k(\mathbf{b})$ as follows:

$$H_k(\mathbf{b}) = \frac{\sum_{i=1}^{s} w(\mathbf{b}, \mathbf{b}^i) F_k(\mathbf{b}^i)}{\sum_{i=1}^{s} w(\mathbf{b}, \mathbf{b}^i)}. \tag{2.4}$$

The weight $w(\mathbf{b}, \mathbf{b}^i) = K_m(d(\mathbf{b}, \mathbf{b}^i))$ measures the closeness between the query point $\mathbf{b}$ and the sample point $\mathbf{b}^i$. In this setting, we use $d(\cdot)$ as the distance function as described earlier. We employ the relationship $K_m(t_1, \cdots, t_m) = \sum_{i=1}^{m} K(t_i)$ to calculate $K_m(d(\mathbf{b}, \mathbf{b}^i)$. In this way, we can exploit one-dimensional kernel functions $K(\cdot)$. A typical one-dimensional kernel function is Gaussian $K(t) = e^{-t^2}$. In essence, nonparametric approximation emphasizes nearby sample points $\mathbf{b}^i \in S$ to the query point $\mathbf{b}$ via $w(\mathbf{b}, \mathbf{b}^i)$. From this perspective, it can be considered as a local learning which does not assume a functional form. In the nonparametric approximation scheme $F_k(\mathbf{b}^i)$, $\forall\ S$ are the *representative information* meaning that once we obtain them we calculate the estimate of $F_k(\mathbf{b})$ by using Equation (2.4). In reality, we use $H_k(\mathbf{b}^i)$, an approximate value of $F_k(\mathbf{b}^i)$, as the representative information because it is impractical to find and store $F_k(\mathbf{b}^i)$ for $\mathbf{b}^i \in S$. This fact puts another layer of difficulty in our desire to approximate the optimal values effectively.

2.4.5 The Statistical Learning Algorithm

The *Statistical learning algorithm* is an approximate dynamic programming methodology which exploits parametric, local parametric and nonparametric approximation schemes as value function approximation. Statistical learning algorithm consists of two phases, namely *forward* and *backward* phases. In the forward phase, under a certain approximation scheme we find and store corresponding representative information in a **look-up** table as described in Table 2.3. We introduced the parameters $\boldsymbol{\theta}_k^*$ as the representative information of the parametric approximation and $H_k(\mathbf{b}^i)$, $\mathbf{b}^i \in S$ as the representative information of both local parametric and nonparametric approximation scheme. Since local parametric uses the same representative information

as nonparametric approximation we describe only the nonparametric approximation scheme in the forward phase (see Table 2.3).

Thus, the following tables (Tables 2.4 and 2.5) are obtained at the end of forward phase.

In order to illustrate the functionality of look-up tables, let assume that we want to find an approximate value $H_k(\mathbf{b})$ to the optimal value $F_k(\mathbf{b})$ of an state $(k, \mathbf{b})$. Under the parametric case, we retrieve $\boldsymbol{\theta}_k^*$ from the k-th column of the parametric look-up table (Table 2.4) and calculate the estimate $H_k(\mathbf{b}) = g(\boldsymbol{\theta}_k^*, \mathbf{b})$. We construct a suboptimal solution in the backward phase as described in Table 2.6.

We apply statistical learning only to knapsack problems (KP). The underlying reason is that it is straightforward to detect whether a knapsack problem is feasible or not as will be described in the next chapters. Thus, if the knapsack problem $KP(k, \mathbf{b})$ with k variables and a right-hand-side $\mathbf{b}$ is infeasible, we set $H_k(\mathbf{b}) = -\infty$. The availability of *infeasibility detection* avoids misleading approximate values that can be obtained by statistical learning approaches. On the other extreme, the infeasibility detection is not straightforward for the binary integer programs. As a matter of fact, detecting an infeasibility is as hard as optimizing the binary integer programming problem. This unfortunate fact limits the capabilities of the statistical learning framework in the case of binary integer programming problems.

Statistical Learning: Forward Phase

1: Initialization:
Generate a sample $S = \{\mathbf{b}^1, \cdots, \mathbf{b}^s\}$

2: $k = 1$: $H_1(\mathbf{b}) = F_1(\mathbf{b})$

3: $k = 2$:

Parametric: Find $\boldsymbol{\theta}_2^*$

-$H_2(\mathbf{b}^i) \leftarrow F_2(\mathbf{b}^i),\ i \in [1, s]$

- $\boldsymbol{\theta}_2^* =$ argmin $[\sum_{i=1}^{s} L(g(\boldsymbol{\theta}_2, \mathbf{b}^i), H_2(\mathbf{b}^i))]$

Nonparametric: Find $H_2(\mathbf{b}^i),\ \mathbf{b}^i \in S$

- $H_2(\mathbf{b}^i) \leftarrow F_2(\mathbf{b}^i),\ i \in [1, s]$

4: for $k = 3, \cdots, n-1$ do

Parametric: Find $\boldsymbol{\theta}_k^*$

-$H_{k-1}(\mathbf{b}^i - \mathbf{A}_k x) \leftarrow g(\boldsymbol{\theta}_{k-1}^*, \mathbf{b}^i - \mathbf{A}_k x),\ x \in \{0,1\},\ i \in [1, s]$

-$H_k(\mathbf{b}^i) \leftarrow$ max $_{x \in \{0,1\}} \{H_{k-1}(\mathbf{b}^i - \mathbf{A}_k x) + c_k x\},\ i \in [1, s]$

- $\boldsymbol{\theta}_k^* =$ argmin $[\sum_{i=1}^{s} L(g(\boldsymbol{\theta}_k, \mathbf{b}^i), H_k(\mathbf{b}^i))]$

Nonparametric: Find $H_k(\mathbf{b}^i),\ b \in S$

$$\text{-}H_{k-1}(\mathbf{b}^i - \mathbf{A}_k x) \leftarrow \frac{\sum_{i=1}^{s} w(\mathbf{b}^i - \mathbf{A}_k x, \mathbf{b}^i) H_{k-1}(\mathbf{b}^i)}{\sum_{i=1}^{s} w(\mathbf{b}^i - \mathbf{A}_k x, \mathbf{b}^i)},\ x \in \{0,1\},\ i \in [1, s]$$

-$H_k(\mathbf{b}^i) \leftarrow$ max $_{x \in \{0,1\}} \{H_{k-1}(\mathbf{b}^i - \mathbf{A}_k x) + c_k x\},\ i \in [1, s]$

5: Output: **look-up table**

Parametric: $\boldsymbol{\theta}_k^*,\ k \in [1, n]$

Nonparametric $H_k(\mathbf{b}^i),\ \forall\ \mathbf{b}^i \in S,\ k \in [1, n]$

Table 2.3: Statistical Learning Framework: Forward Phase.

stage 1	stage 2	$\cdots$	stage k	$\cdots$	stage n-1	stage n
	$\boldsymbol{\theta}_2^*$	$\cdots$	$\boldsymbol{\theta}_k^*$	$\cdots$	$\boldsymbol{\theta}_{n-1}^*$	

Table 2.4: Parametric Look-up Table.

state	stage 1	stage 2	$\cdots$	stage k	$\cdots$	stage n-1	stage n
$\mathbf{b}^1$		$H_1(\mathbf{b}^1)$	$\cdots$	$H_k(\mathbf{b}^1)$	$\cdots$	$H_{n-1}(\mathbf{b}^1)$	
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	
$\mathbf{b}^s$		$H_1(\mathbf{b}^s)$	$\cdots$	$H_k(\mathbf{b}^s)$	$\cdots$	$H_{n-1}(\mathbf{b}^s)$	

Table 2.5: Nonparametric and Local Parametric Look-up Table.

Statistical Learning: Backward Phase

Input: g, L, w, S

1: $\mathbf{b} \leftarrow \mathbf{b}_0$

2: for $k = n, \cdots, 2$ do

- Find $H_{k-1}(\mathbf{b} - \mathbf{A}_k x)$ for $x \in \{0, 1\}$
- Infeasibility Detection:

  $(\mathbf{b} - \mathbf{A}_k x) \not\geq \mathbf{0} \rightarrow H_{k-1}(\mathbf{b} - \mathbf{A}_k x) = -\infty$

  Parametric:

  -Retrieve $\boldsymbol{\theta}_{k-1}$ from Table 2.4

  -$H_{k-1}(\mathbf{b} - \mathbf{A}_k x) \leftarrow g(\boldsymbol{\theta}^*_{k-1}, \mathbf{b} - \mathbf{A}_k x)$

  Local Parametric:

  -Retrieve $H_{k-1}(\mathbf{b}^i), \mathbf{b}^i \in S$ from Table 2.5

  -Find $\boldsymbol{\theta}_{k-1}(\mathbf{b} - \mathbf{A}_k x)$

  -$H_{k-1}(\mathbf{b} - \mathbf{A}_k x) \leftarrow g(\boldsymbol{\theta}^*_{k-1}(\mathbf{b} - \mathbf{A}_k x), \mathbf{b} - \mathbf{A}_k x)$

  Nonparametric:

  - Retrieve $H_{k-1}(\mathbf{b}^i), \mathbf{b}^i \in S$ from Table 2.5

  -$H_{k-1}(\mathbf{b} - \mathbf{A}_k x) \leftarrow \frac{\sum_{i=1}^{s} w(\mathbf{b} - \mathbf{A}_k x, \mathbf{b}^i) H_{k-1}(\mathbf{b}^i)}{\sum_{i=1}^{s} w(\mathbf{b} - \mathbf{A}_k x, \mathbf{b}^i)}$

- $x_k^{adp} \leftarrow$ argmax $_{x \in \{0,1\}} \{H_{k-1}(\mathbf{b} - \mathbf{A}_k x) + c_k x\}$
- $\mathbf{b} \leftarrow \mathbf{b} - \mathbf{A}_k x_k^{adp}$

3 : x_1^{adp} solves the knapsack problem $KP(1, \mathbf{b})$

Output: $\mathbf{x}^{adp} = (x_1^{adp}, \cdots, x_n^{adp})$

Table 2.6: Statistical Learning Framework: Backward Phase.

Chapter 3

The Binary Knapsack Problem

The aim of this chapter is to develop an approximate dynamic programming approach to the binary knapsack problem. The problem is as follows: Given a collection of n items where the objective is to select a subset of these given items so that the sum of the sizes in this subset does not exceed a given capacity b_0 and the sum of selected values is maximized. Assuming item j has a value c_j and a weight a_j, we can write the following mathematical programming model by introducing the binary decision variables x_j ($x_j = 1$ if the item j is selected).

(BKP)

$$\begin{aligned} \text{maximize} \quad & \sum_{j=1}^{n} c_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} a_j x_j \leq b_0 \\ & x_j \in \{0,1\},\ j \in \{1,\cdots,n\}. \end{aligned}$$

The binary knapsack problem assumes nonnegative data, i.e., $c_j, a_j \geq 0$. The chapter is structured as follows. We first provide a literature review. After reformulating the problem as a dynamic program, we customize statistical and base-heuristic learning methods to this problem. We finally offer computational results and conclusions.

3.1 Background

The binary knapsack problem has been intensively studied for both practical and theoretical reasons. On the practical side, many resource allocation and financial management problems can be formulated as binary knapsack problems. From a theoretical point of view, either a general problem is transformed to a binary knapsack problem or it is used as a subproblem of a complex problem. For instance, the binary knapsack problem is used as a subproblem when solving the Generalized Assignment Problem, as well as the Vehicle Routing Problems (see Laporte [42]). As a result there is a wide literature on the subject (see Dudzinski and Walukiewic [22] and Martello and Toth [54]).

The binary knapsack problem is NP-hard (see Garey and Johnson [30]) but it can be solved in pseudo-polynomial time by dynamic programming (see Papadimitriou and Steiglitz [62]). Based on the observations by Balas and Zemel [4], Pisinger [66] and Martello et. al. [50], the difficulty of a particular instance depends on two aspects: the degree of linearity between c_j and a_j, and the gap Δ between the linear programming relaxation value and optimal value. Computational experiments classifies randomly generated problem instances in one of the categories.

- *Uncorrelated data instances*: Coefficients c_j and a_j are not correlated.
- *Weakly correlated data instances*: The ratio c_j/a_j has a small variation. The gap Δ is usually relatively small.
- *Strongly correlated data instances*: We have $c_j = a_j + r$, where r is a constant. Frequently the gap Δ will be of same magnitude as r.

Available computational evidence suggests that the strongly correlated instances are among the most challenging types of the problem (see Martello and Toth [54], Martello et. al. [50] and Pisinger [66]). It is also observed that binary knapsack problems become increasingly more difficult if a_j values attain large values. Such

instances occur when transforming general integer programming problems to the binary knapsack problems, and are thus of great practical value as well as of theoretical interest. Where uncorrelated data instances of size up to $n = 100,000$ may be solved in seconds when a_j values are relatively small (see Balas and Zemel [4], Martello and Toth [53]), no algorithms in the literature are able to solve strongly correlated data instances of size larger than $n = 100$ when a_j values grow (see Martello et. al. [50]).

In the literature, *exact* algorithms for 0-1 knapsack problems are mainly based on branch-and-bound and dynamic programming. For successful branch-and-bound algorithms refer to Martello and Toth [51, 53] and Horowitz and Sahni [38]. It is observed that the key element in the success of branch-and-bound is finding good upper bounds (see Martello and Toth [54]).

Dynamic programming (DP) is used as an alternative to branch-and-bound. Toth [84] presents several dynamic programming algorithms and makes a comparative study between them. This specific paper also discusses how to eliminate non-promising states and some dominance rules. Recently, Pisinger [67, 68] generates a new class of problems for which the best known branch-and-bound algorithms behave poorly. Pisinger [68] proposes a new dynamic programming approach that solves large instances to optimality in a short time by utilizing a variant of the idea introduced by Balas and Zemel [4]. In a computational study, he demonstrates that dynamic programming algorithms behave better than branch-and-bound algorithms particularly for the hard instances whereas the former methodology does have a limitation due to large state space. Combining the advantageous aspects of DP and branch-and-bound, Martello and Toth [52] developed a hybrid approach for the subset problem indicating computational superiority of this method over existing ones.

A recent report by Martello et. al. [50] illustrates that state-of-art algorithms can effectively solve large and difficult instances of the 0-1 knapsack problem. It should be noted that these algorithms are very specialized and consist of hybrid approaches

in order to exploit effectively their advantageous aspects.

In contrast to *exact* algorithms many approximation schemes have been proposed by researchers facing difficult instances. Ibarra and Kim [40] presented the first fully polynomial approximation scheme for the binary knapsack problem. According to Martello and Toth [54] the performance of fully polynomial approximation schemes is considerably worse than heuristics based on partial enumeration. These heuristics cannot guarantee a worst-case performance, but they perform extremely well for several data instances. Partial enumeration techniques are based on an exact enumeration algorithm like branch-and-bound or dynamic programming, where enumeration is terminated according to some prespecified criterion before optimality has been found or proved. Greedy-type algorithms have also played a key role in obtaining good solutions in short times.

3.2 Dynamic Programming

Following the general guidelines developed in Chapter 2, we consider the following subproblem $BKP(k, b)$:

$$\begin{aligned} F_k(b) = \quad & \text{maximize} \quad \sum_{j=1}^{k} c_j x_j \\ & \text{subject to} \quad \sum_{j=1}^{k} a_j x_j \leq b \\ & \qquad\qquad x_j \in \{0,1\}, j = 1, \cdots, k. \end{aligned}$$

We use $F_k(b) = -\infty$ if $b < 0$. The dynamic programming recursion is, for $k = 2, \cdots, n$,

$$F_k(b) = \max \{F_{k-1}(b), F_{k-1}(b - a_k) + c_k\},$$

with the initial condition,

$$F_1(b) = \begin{cases} 0, & a_1 > b, \\ c_1, & a_1 \leq b. \end{cases}$$

We can construct the optimal solution as described in the dynamic programming backward phase (see Table 2.1). The worst case time complexity of the underlying dynamic programming for the problem $BKP(n, b_0)$ is $O(nb_0)$ where b_0 is the capacity. Dynamic programming based on the above recursion is generally not an effective way of solving binary knapsack problems since the space requirement is very large. Space requirements and value function computations become impractical for large b_0. The binary knapsack with large capacities arise in many applications. For instance, integer programs can be reduced to knapsack problems by some multiplier techniques (see Salkin and Mathur [73]) which results in large parameters and capacity. These reductions are inevitably very difficult to solve as the size of the problem increases.

3.3 Statistical Learning

In this section, we develop a statistical learning algorithm for the binary knapsack problem. We first introduce notations and definitions. We then adapt generic parametric, local parametric and nonparametric approximation schemes (see Chapter 2) to the binary knapsack problem. We provide motivating examples in order to illustrate how the proposed statistical learning algorithm is applied to a binary knapsack problem.

3.3.1 Notations and Definitions

Let b_0 be the right-hand side (capacity) of the problem $BKP(n, b_0)$. We define by $\Omega = \{b | 0 \leq b \leq b_0\}$ the state space. Let $S = \{b^i \in \Omega | i = 1, \cdots, s\}$ denote a sample of the state space Ω. Define $\Omega_k = \{(k, b) | b \in \Omega\}$ and $S_k = \{(k, b) | b \in S\}$ for k varying in $[1, n]$. Let us define by $N_h(b) = \{b^i \in S | b - h \leq b^i \leq b + h\}$ the *neighborhood set* of a query point b within the bandwidth h. Similarly, we let

$N^*(b) = \{b^k, b^{k+1} \in S | b^k \leq b \leq b^{k+1}\}$ the *nearest neighborhood set* of a query point b. Let $K : \Re \rightarrow \Re$ be a one-dimensional kernel function. Examples of kernel functions include:

1. Gaussian: $K(t) = e^{-t^2/2} I(|t| \leq 1)$.

2. Triangular: $K(t) = (1 - |t|) I(|t| \leq 1)$.

3. Beta family: $K(t) = [\ \max\ \{0, 1 - t^2\}]^\gamma I(|t| \leq 1),\ \gamma = 0, 1, 2, 3.$

where $I(\cdot)$ is an indicator function i.e., $I(|t| \leq 1) = 1$ (0) if $|t| \leq 1$ (otherwise).

3.3.2 Parametric Approximation

We approximate $F_k : \Omega_k \rightarrow \Re$ by a linear function $g : \Omega_k \rightarrow \Re$ of the form $g(\boldsymbol{\theta}_k, b) = \alpha_k b + \beta_k$, meaning that for a state $(k, b) \in \Omega_k$ we have $H_k(b) = g(\boldsymbol{\theta}_k, b)$. To tune the parameters we minimize the following unweighted training criterion

$$C = \sum_{i=1}^{s} L(g(\boldsymbol{\theta}_k, b^i), F_k(b^i))$$

where the $F_k(b^i)$ are the output values corresponding $b^i \in S$, $\boldsymbol{\theta}_k$ is the parameter vector for the parametric model $H_k(b) = g(\boldsymbol{\theta}_k, b)$ and $L(H_k(b^i), F_k(b^i))$ is a general loss function for predicting $\hat{F}_k(b^i)$ when the training data is $F_k(b^i)$. We use the least squares criterion for the loss function i.e., $L(H_k(b^i), F_k(b^i)) = (H_k(b^i) - F_k(b^i))^2$ leading to the training criterion:

$$C = \sum_{i=1}^{s} (\alpha_k b^i + \beta_k - F_k(b^i))^2.$$

We find α_k^*, β_k^* that minimizes the above criterion as follows:

$$\alpha_k^* = \frac{s \sum_i b^i F_k(b^i) - (\sum_i b^i)(\sum_i F_k(b^i))}{s(\sum_i (b^i)^2) - (\sum_i b^i)^2},$$

$$\beta_k^* = \frac{\sum_i F_k(b^i) - \alpha_k^* \sum_i b^i}{s}.$$

3.3.3 Local Parametric Approximation

Parametric approximation gives equal emphasis on every data point. We can tailor the data set to the query point by emphasizing nearby points in the regression. As discussed, we can do this by weighting the training criterion:

$$C = \sum_{i=1}^{s} L(g(\boldsymbol{\theta}_k, b^i), F_k(b^i))w(b, b^i)$$

where $w(b, b^i) = K(d(b^i, b)/h)$ for a kernel function $K(\cdot)$ and for a distance function $d(b^i, b)$ between the query point b and each data point input vector b^i. We use the distance function $d(b^i, b) = |b^i - b|$. The parameter h is called the bandwidth and plays an important role. As the bandwidth h increases then the influence of points b^i further from b decreases. Thus, h can be seen as a parameter for controlling the degree of localization of the approximation. We select least-squares criterion as our loss function.

Using this training criterion $g(\boldsymbol{\theta}_k(b), b) = \alpha(b)_k b + \beta(b)_k$ now becomes a *local* model, and can have a different set of parameter $\boldsymbol{\theta}_k(b)$ for each query point b. Thus our training criterion for query point b becomes:

$$C(b) = \sum_{i=1}^{s} (\alpha(b)_k b^i + \beta(b)_k - F_k(b^i))^2 w(b, b^i).$$

Under local parametric approximation $F_k(b)$ (see Fan [25]) is approximated by:

$$H_k(b) = \frac{\sum_{1}^{s} w_i F_k(b^i)}{\sum_{1}^{s} w_i}$$

where $w_i = w(b, b^i)\{S_{s,2} - (b^i - b)S_{s,1}\}$ and $S_{s,j} = \sum_1^s w(b, b^i)(b^i - b)^j, j = 1, 2.$

3.3.4 Nonparametric Approximation

Suppose we have calculated the value functions at points $b^i \in S$, i.e., $F_k(b^i)$ have been calculated. Under the nonparametric approximation, $F_k(b)$ is approximated by $H_k(b)$ as follows:

$$H_k(b) = \frac{\sum_{i=1}^{s} w(b, b^i) F_k(\mathbf{b^i})}{\sum_{i=1}^{s} w(b, b^i)}$$

where $w(b, b^i) = K(d(b^i, b)/h)$, $K(\cdot)$ is a kernel function, $d(b^i, b)$ is a distance function between the query point b and each data point input vector b^i, and h is the bandwidth. We use the distance function $d(b^i, b) = |b^i - b|$ under the nonparametric approximation.

As an example by selecting $K(t) = (1 - |t|)I(|t| \leq 1)$ as the kernel function, and the bandwidth h such that $N_h(b) = N^*(b)$ then we can obtain the approximation of $F_k(b)$ as a *linear interpolation.* To expedite the approximation we can use the following relationship. Let $N^*(b) = \{b_{i^*}, b_{i^*+1}\}$. Then under linear interpolation, $F_k(b)$ is approximated by $H_k(b)$ as follows:

$$H_k(b) = \frac{b - b_{i^*}}{b_{i^*+1} - b_{i^*}} F_k(b_{i^*+1}) + \frac{b_{i^*+1} - b}{b_{i^*+1} - b_{i^*}} F_k(b_{i^*}).$$

3.3.5 Motivating Example

We try to illustrate the ideas we introduced with an example. Consider the following binary knapsack problem (BKP):

$$\begin{aligned} \text{maximize} \quad & x_1 + x_2 + 3x_3 + 2x_4 \\ \text{subject to} \quad & 2x_1 + x_2 + 4x_3 + 3x_4 \leq 5 \\ & x_j \in \{0, 1\}, j = 1, \cdots, 4. \end{aligned}$$

The state-space of the problem is $\Omega = \{0, 1, 2, 3, 4, 5\}$. We calculate optimal values $F_k(b)$ for all $(k, b) \in \Omega_k$ for k varying in $[1, n]$ to construct the following tabular

information (see Table 3.1). The optimal value of the example $BKP(4,5)$ is 4 with an optimal solution $(0,1,1,0)$.

state b	$F_1(b)$	x_1^*	$F_2(b)$	x_2^*	$F_3(b)$	x_3^*	$F_4(b)$	x_4^*
< 0	$-\infty$		$-\infty$		$-\infty$		$-\infty$	
0	0	0	0	0	0	0	0	0
1	0	0	1	1	1	0	1	0
2	1	1	1	0	1	0	1	0
3	1	1	2	1	2	0	2	0
4	1	1	2	1	3	1	3	0
5	1	1	2	1	4	1	4	0

Table 3.1: Motivating Example: Optimal values and decisions.

Parametric Approximation:

The underlying assumption is optimal values $F_k(b)$ are approximated by $H_k(b) = g(\boldsymbol{\theta}_k, b) = \alpha_k b + \beta_k$. We use the least squares criterion to tune the parameters $\boldsymbol{\theta}_k = (\alpha_k, \beta_k)$. We use the forward phase of Statistical Learning Framework (see Table 2.3) to generate the *parametric look-up table* (see Table 3.3) . We present the algorithmic steps in Table 3.2.

We apply Backward Phase of Statistical Learning (see Table 2.6) to construct a suboptimal solution. We present the steps in Table 3.4. ADP generates a feasible solution of $\mathbf{x}^{adp} = (0,1,0,1)$ whose value is 3. Note that, this solution is a suboptimal solution since the optimal value is 4.

Motivating Example
Parametric : Forward Phase

1: Let the sample $S = \{0, 2, 5\}$

2: Stage $k = 1$: $H_1(b) = F_1(b) =$ max $\{x_1 | 2x_1 \leq b, x_1 = 0, 1\}$

3: Stage $k = 2$: Find $\boldsymbol{\theta}_2^*$

-Calculate $H_2(b), \forall b \in S$:

$H_2(b) = \max\{H_1(b), H_1(b - a_2) + c_2\}$.

-$\boldsymbol{\theta}_2^* =$ argmin $\{\sum_{b \in S}[H_2(b) - g(\boldsymbol{\theta}_2, b)]^2\}$

-Store $\boldsymbol{\theta}_2^* = (\alpha_2^*, \beta_2^*) = (0.395, 0.078)$ in Table 3.3.

4: Stage $k = 3$: Find $\boldsymbol{\theta}_3^*$

-Calculate $H_3(b), \forall b \in S$:

$H_3(b) =$ max $_{x \in \{0,1\}}\{H_2(b - a_3 x) + c_3 x\}$

$H_2(b - a_3 x) = g(\boldsymbol{\theta}_2^*, b - a_3 x),\ x \in \{0, 1\}$

Example: $H_3(5) =?,\ 5 \in S$

$H_3(5) =$ max$\{H_2(5), H_2(1) + c_3\}$

Retrieve $\boldsymbol{\theta}_2^*$ from Table 3.3

$H_2(5) = 0.395 \times 5 + 0.078 = 2.053$

$H_2(1) = 0.395 \times 1 + 0.078 = 0.473$

-$\boldsymbol{\theta}_3^* =$ argmin $\{\sum_{b \in S}[H_3(b) - g(\boldsymbol{\theta}_3, b)]^2\}$

-Store $\boldsymbol{\theta}_3^* = (\alpha_3^*, \beta_3^*) = (0.605, -0.078)$ in Table 3.3.

5: Output:

Parametric look-up table (Table 3.3)

Table 3.2: Motivating Example: Parametric Forward Phase.

$\boldsymbol{\theta}_k^*$	stage 1	stage 2	stage 3	stage 4
α_k	-	0.395	0.605	-
β_k	-	0.078	-0.078	-

Table 3.3: Motivating Example: Parametric Look-up Table.

Motivating Example
Parametric: Backward Phase

1: Initialization:

$b \leftarrow b_0 = 5$.

$\mathbf{c} \leftarrow (c_1, c_2, c_3, c_4) = (1, 1, 3, 2)$.

$\mathbf{a} \leftarrow (a_1, a_2, a_3, a_4) = (2, 1, 4, 3)$.

2: Find x_4^{adp}:

-$x_4^{adp} =$ argmax $_{x \in \{0,1\}} \{H_3(b - a_4 x) + c_4 x\}$

$H_3(b) = g(\boldsymbol{\theta}_3^*, b)$. Retrieve $\boldsymbol{\theta}_3^*$ from Table 3.3

$H_3(b) = 2.947,\ H_3(b - a_4) + c_4 = 3.132$

$x_4^{adp} = 1$

$b \leftarrow b - a_4 x_4^{adp} = 2$

3: Find x_3^{adp}:

-$x_3^{adp} =$ argmax $_{x \in \{0,1\}} \{H_2(b - a_3 x) + c_3 x\}$

$H_2(b) = g(\boldsymbol{\theta}_2^*, b)$. Retrieve $\boldsymbol{\theta}_2^*$ Table 3.3

$H_2(b) = 0.868,\ H_2(b - a_3) = -\infty$ (Infeasibility detection: $b - a_3 = -2 < 0$)

$x_3^{adp} = 0$

$b \leftarrow b - a_3 x_3^{adp} = 2$

4: Find x_2^{adp}:

-$x_2^{adp} =$ argmax $_{x \in \{0,1\}} \{H_1(b - a_2 x) + c_2 x\}$

$H_1(b) = F_1(b) = 1,\ H_1(b - a_2) + c_2 = F_1(b - a_2) + c_2 = 1$

$x_2^{adp} = 1$

$b \leftarrow b - a_2 x_2^{adp} = 1$

5: Find x_1^{adp}:

$x_1^{adp} = 0$ solves the reduced subproblem $BKP(1, 1)$, i.e., max $= \{x_1 | 2x_1 \leq 1, x_1 = 0, 1\}$.

6: We construct the suboptimal solution $\mathbf{x}^{adp} = (0, 1, 0, 1)$ whose objective value is 3.

The optimal value of the example $BKP(4, 5)$ is 4 with an optimal solution $(0, 1, 1, 0)$.

Table 3.4: Motivating Example: Parametric Backward Phase.

Nonparametric Approximation:

We describe the forward phase in Table 3.5 to create the nonparametric look-up table. In this example, we use triangular kernel function $K(t) = (1 - |t|)I(|t| \leq 1)$, we set the bandwidth parameter h to 2, and we utilize the distance function $d(x, y) = |x - y|$. Then, we calculate the weights as $w(x, y) = K(d(x, y)/h)$. In the backward phase which is described in Table 3.7 we construct the ADP solution. In this case, we find an optimal solution of $\mathbf{x}^{adp} = (0, 1, 1, 0)$ whose value is 4.

Motivating Example
Nonparametric: Forward Phase

1: Let $S = \{b^1, \cdots, b^s\} = \{0, 2, 5\},\ s = 3$

2: $k = 1$: $H_1(b) = F_1(b)$

3: $k = 2$: Find $H_2(b^i),\ b^i \in S$

-$H_2(b^i) \leftarrow F_2(b^i),\ i \in [1, s]$

-Store $H_2(b^i)$ in Table 3.6

4: $k = 3$: Find $H_3(b^i),\ b \in S$

$$\text{-}H_2(b^i - a_3 x) \leftarrow \frac{\sum_{i=1}^{s} w(b^i - a_3 x, b^i) H_2(b^i)}{\sum_{i=1}^{s} w(b^i - a_3 x, b^i)},\ x \in \{0, 1\},\ i \in [1, s]$$

$$\text{-}H_3(b^i) \leftarrow \max_{x \in \{0,1\}} \{H_2(b^i - a_3 x) + c_3 x\},\ i \in [1, s]$$

-Store $H_3(b^i)$ in Table 3.6

Example: $H_3(5) = ?,\ 5 \in S$

$H_3(5) = \max\{H_2(5), H_2(1) + c_3\}$

$H_2(1) = \frac{w(1,0)H_2(0) + w(1,2)H_2(2) + w(1,5)H_2(5)}{w(1,0) + w(1,2) + w(1,5)}$

$w(1, 0) = w(1, 2) = 1/2,\ w(1, 5) = 0 \rightarrow H_2(1) = 1$

$H_3(5) = \max\{2, 1 + c_3\} = 4$

5: Output:

Nonparametric look-up table (Table 3.6)

Table 3.5: Motivating Example: Nonparametric Forward Phase.

	$H_k(b)$			
$b \in S$	$k = 1$	$k = 2$	$k = 3$	$k = 4$
0	0	0	0	-
2	1	1	1	-
5	1	2	4	-

Table 3.6: Motivating Example: Nonparametric Look-up Table.

Motivating Example
Nonparametric: Backward Phase

1: Initialization:

$b \leftarrow b_0 = 5$.

$\mathbf{c} \leftarrow (c_1, c_2, c_3, c_4) = (1, 1, 3, 2)$.

$\mathbf{a} \leftarrow (a_1, a_2, a_3, a_4) = (2, 1, 4, 3)$.

2: Find x_4^{adp}:

-$x_4^{adp} =$ argmax $\{H_3(5), H_3(2) + 2\}$

$H_3(5) = 4,\ H_3(2) = 1$ from Table 3.6

$x_4^{adp} = 0$

$b \leftarrow b - a_4 x_4^{adp} = 5$

3: Find x_3^{adp}:

-$x_3^{adp} =$ argmax $\{H_2(5), H_2(1) + 3\}$

$H_2(5) = 2$ from Table 3.6

$H_2(1) = \frac{w(1,0)H_2(0)+w(1,2)H_2(2)+w(1,5)H_2(5)}{w(1,0)+w(1,2)+w(1,5)}$

$w(1,0) = w(1,2) = 1/2,\ w(1,5) = 0 \rightarrow H_2(1) = 1$

$x_3^{adp} = 1$

$b \leftarrow b - a_3 x_3^{adp} = 1$

4: Find x_2^{adp}:

-$x_2^{adp} =$ argmax $\{H_1(1), H_1(0) + 1\}$

$H_1(1) = F_1(1) = 1,\ H_1(0) = 0$

$x_2^{adp} = 1$ (or $x_2^{adp} = 0$)

$b \leftarrow b - a_2 x_2^{adp} = 0$

5: Find x_1^{adp}:

$x_1^{adp} = 0$ solves the reduced subproblem $BKP(1, 0)$, i.e., max $= \{x_1 | 2x_1 \leq 0,\ x_1 = 0, 1\}$.

6: We construct the solution $\mathbf{x}^{adp} = (0, 1, 1, 0)$ whose objective value is 4.

The optimal value of the example $BKP(4, 5)$ is 4 with an optimal solution $(0, 1, 1, 0)$.

Table 3.7: Motivating Example: Nonparametric Backward Phase.

3.4 Base-heuristic Learning

In this section, we develop a base-heuristic learning algorithm for the binary knapsack problem. We introduce the proposed base-heuristic in the next section. We conclude this section by providing a motivating example to illustrate the effectiveness of the base-heuristic learning framework.

3.4.1 Base-heuristic Selection

We consider the following greedy heuristic as our *base-heuristic* to the problem BKP (n, b_0). The items are ordered according to nonincreasing efficiencies $\frac{c_j}{a_j}$, thus

$$\frac{c_1}{a_1} \geq \frac{c_2}{a_2} \geq \cdots \geq \frac{c_n}{a_n}. \tag{3.1}$$

The greedy heuristic works as follows: Select items $1, \cdots, k$ such that

$$\sum_{i=1}^{k} a_i \leq b_0 < \sum_{i=1}^{k+1} a_i.$$

The solution value of the greedy heuristic is thus:

$$z_G = \sum_{i=1}^{k} c_j$$

The greedy heuristic is naturally connected with the LP relaxation of the knapsack problem as follows (see Dantzig [20]):

$$\begin{aligned} x_i^* &= 1 \text{ for } i = 1, \cdots, k, \\ x_{k+1}^* &= \frac{b_0 - \sum_{i=1}^{k} a_i}{a_{k+1}}, (\text{ note that } 0 \leq x_{k+1}^* \leq 1 \text{ by the construction of } k), \\ x_i^* &= 0 \text{ for } i = k+1, \cdots, n. \end{aligned}$$

The initial ordering (3.1) can be carried out in $O(n\ logn)$ time but Balas and Zemel [4] have shown that the linear programming relaxation of the binary knapsack can be solved in $O(n)$ without ordering as the problem can be solved as a weighted median. Another important empirically observed property is that, having solved the linear programming relaxation, generally only a few decision variables need to be changed in order to obtain the optimal integer solution. Most of the solution values are the same, whereas the differing variables generally are close to x_k. This behavior has been documented in several computational experiments and motivated Balas and Zemel [4] to propose that only a few variables around k are considered in order to solve the binary knapsack problem to optimality. This problem was denoted as the core problem and has been as an essential part of all efficient algorithms for knapsack problems (for further discussion see Pisinger [66] and Martello and Toth [54]).

The greedy algorithm is described in Table 3.8. We use the greedy algorithm as our base-heuristic for the binary knapsack problem. We employ Base-heuristic Learning Framework (Table 2.2) to construct a suboptimal solution. We illustrate the ideas with a motivating example in the next section.

3.4.2 Motivating Example

We try to illustrate the computations of the base-heuristic learning framework with an example. Consider the following binary knapsack problem $BKP = BKP(4, 4)$:

$$\begin{aligned} \text{maximize} \quad & 2x_1 + 4x_2 + x_3 + 3x_4 \\ \text{subject to} \quad & x_1 + 2x_2 + x_3 + 2x_4 \leq 4 \\ & x_j \in \{0, 1\}, j = 1, \cdots, 4. \end{aligned}$$

Following the notations introduced for the base-heuristic learning (see Chapter 2), we denote by $BH(k, b)$ the greedy heuristic as described in Table 3.8 applied to the subproblem $BKP(k, b)$ for varying k and b. Let $H_k(b)$ be the greedy heuris-

Greedy Algorithm- Binary Knapsack

Input: $BKP(n, b_0)$
Assumption:
$\frac{c_1}{a_1} \geq \frac{c_2}{a_2} \geq \cdots \geq \frac{c_n}{a_n}$
1: Initialization:
$k = 1,\ b = b_0,\ \mathbf{x}^G = \mathbf{0},\ z^G = 0$
2: for $k = 1, \cdots, n$ do
begin
if $(a_k \leq b)$ then
{
-$x_k \leftarrow 1$
-$z^G \leftarrow z^G + c_k$
- $b \leftarrow b - a_k$
}
end
3: Output: $\mathbf{x}^G, z^G$

Table 3.8: Base Heuristic for the Binary Knapsack Problem: Greedy Algorithm.

tic value and $\mathbf{x}_{BH}(k, b)$ be the greedy heuristic solution. The greedy solution is $\mathbf{x}_{BH}(4, 4) = (1, 1, 0, 0)$ with a value of $H_4(4) = 6$. The optimal value of the problem BKP is 7 with optimal solutions $(1, 1, 1, 0)$ and $(0, 1, 0, 1)$. We find the solution $\mathbf{x}^{adp} = (1, 1, 1, 0)$ as described in Table 3.9. As a matter of fact, this is one of the optimal solutions. This example demonstrates the effectiveness of the base-heuristic learning framework. ADP methodology generates an optimal solution by approximating the value functions by the greedy heuristic while the stand-alone application of the same greedy heuristic finds a suboptimal solution. Thus, ADP facilitates a framework to improve the performance of a base-heuristic from a solution quality perspective.

Motivating Example
Base-heuristic Learning Algorithm

1: Initialization:

$b \leftarrow b_0 = 4$.

$\mathbf{c} \leftarrow (c_1, c_2, c_3, c_4) = (2, 4, 1, 3)$.

$\mathbf{a} \leftarrow (a_1, a_2, a_3, a_4) = (1, 2, 1, 2)$.

2: Find x_4^{adp}:

-$x_4^{adp} =$ argmax $_{x \in \{0,1\}} \{H_3(b - a_4 x) + c_4 x\}$

-Apply greedy heuristic to $BKP(3, b)$ and $BKP(3, b - a_4)$

-Get heuristic values $H_3(b)$ and $H_3(b - a_4)$

$H_3(b) = 7,\ H_3(b - a_4) + c_4 = 5$

-$x_4^{adp} = 0$

-$b \leftarrow b - a_4 x_4^{adp} = 4$

3: Find x_3^{adp}:

-$x_3^{adp} =$ argmax $_{x \in \{0,1\}} \{H_2(b - a_3 x) + c_3 x\}$

-Apply greedy heuristic to $BKP(2, b)$ and $BKP(2, b - a_3)$

-Get heuristic values $H_2(b)$ and $H_2(b - a_3)$

$H_2(b) = 6,\ H_2(b - a_3) + c_3 = 7$

-$x_3^{adp} = 1$

-$b \leftarrow b - a_3 x_3^{adp} = 3$

4: Find x_2^{adp}:

-$x_2^{adp} =$ argmax $_{x \in \{0,1\}} \{H_1(b - a_2 x) + c_2 x\}$

-Apply greedy heuristic to $BKP(1, b)$ and $BKP(1, b - a_2)$

-Get heuristic values $H_1(b)$ and $H_1(b - a_2)$

$H_1(b) = 2,\ H_1(b - a_2) + c_2 = 6$

-$x_2^{adp} = 1$

-$b \leftarrow b - a_2 x_2^{adp} = 1$

5: Find x_1^{adp}:

$x_1^{adp} = 1$ solves the reduced subproblem $BKP(1, 1)$, i.e., max $= \{x_1 | 2x_1 \leq 1, x_1 = 0, 1\}$.

6: We construct the solution $\mathbf{x}^{adp} = (1, 1, 1, 0)$ whose objective value is 7.

Note that this solution is optimal.

Table 3.9: Motivating Example: Base-heuristic Learning Algorithm.

3.5 Computational Results

In this section, we provide computational results to the binary knapsack problem. Our basic aim is to demonstrate that approximate dynamic programming algorithms utilizing both the statistical and base-heuristic learning framework can generate good solutions for large and difficult instances in reasonably short computation times. We provide an extensive computational study to illustrate the performance of both statistical and base-heuristic learning framework. We will also be comparing the most promising ADP algorithm with CPLEX. Our criteria for comparing various approaches are solution quality, computation time, and robustness, i.e., the degree of deviation of the computational resources needed to solve the problems as the instances of the same size change.

We construct *uncorrelated, weakly* and *strongly correlated* random binary knapsack problems. Let $x \sim U(1, X)$ denote an integer number that is uniformly generated in $[1, X]$. In our computational study, we consider the following problem types following the guidelines in the literature (see Martello and Toth [54]):

Uncorrelated instances: There is no correlation between the value and resource usage of an item. **Uncorrelated (UC)**: $c_j \sim U(1, C)$ and $a_j \sim U(1, A)$, i.e., c_j, a_j are uniformly distributed in $[1, C], [1, A]$, respectively.

Weakly correlated instances: Value of an item is within a margin of its resource usage. For instance, in financial management it is realistic to assume that the return of an investment is within a probabilistic margin of the investment allocated from a given budget. **Weakly correlated (WC)**: $a_j \sim U(1, A)$ and $c_j = \max\{1, U(a_j - wc, a_j + wc)\}$, i.e., a_j are uniformly distributed in $[1, A]$.

Strongly correlated instances: These instances assume that the value of an item is linear function of the resource usage with some fixed deviation. **Strongly corre-**

lated (SC): $a_j \sim U(1, A)$ and $c_j = a_j + sc$, i.e., a_j are uniformly distributed in $[1, A]$.

Generally speaking, based on the available computational studies in the literature (see Martello and Toth [54] and Pisinger [67, 68]), the stronger the correlation the longer the computation time required to solve the corresponding binary knapsack problem.

Given the number of variables n we generate 5 test problems in each class (uncorrelated, weakly, and strongly correlated) with different parameters A, C, wc and sc. We set the right-hand-side or the capacity of the problem as $b_0 = \sum_{j=1}^{n} a_j/2$. All computational studies are done on a Dell Precision 410 with Linux operating system.

3.5.1 Algorithm Design

Approximate Dynamic Programming offers many algorithmic design choices based on the approximation scheme used. Under the base-heuristic learning framework we will use the greedy heuristic described in Table 3.8 in our computational study. Thus, we focus on design choices of statistical learning framework. Our goal is to find the best combinations given an approximation scheme. We summarize the design parameters and their values in Table 3.10.

In order to tune the design parameters of the statistical learning framework we make a *preliminary* computational study. Based on this computational study we fix the design parameters to such values so that the corresponding statistical learning algorithm is the most effective one compared to other possible choices of design parameters. We call the resultant statistical learning framework as the proposed statistical learning approach for the binary knapsack problem.

Sampling Type:

In our computational study we chose $S = \{b^1, \cdots, b^s\}$ with $b^i = i\lfloor b_0/s \rfloor$, $i =$

Approximation Scheme	Design Parameter	Alternatives
Parametric	Functional Form	Linear, Quadratic
	Parameter Learning	Least-squares, Least absolute deviation
	Sampling Type	Random, Uniform
Local Parametric	Functional Form	Linear, Quadratic
	Parameter Learning	Least-squares, Least absolute deviation
	Kernel Selection	Gaussian,Triangular, Beta
	Bandwidth	Search, Automated
	Sampling Type	Random, Uniform
Nonparametric	Kernel Selection	Gaussian,Triangular, Beta
	Bandwidth	Search, Automated
	Sampling Type	Random, Uniform

Table 3.10: Design parameters of the statistical learning framework for the binary knapsack problem.

$0, \cdots, s-1$ and $b^s = b_0$ where S is a sample of the state space $\Omega = \{0, 1, \cdots, b_0\}$. We name this sample as *uniform sample*. An alternative to selecting the set S is to choose the set S randomly over the state space Ω. We have found, however, that this choice is inferior to selecting the set S as outlined above. Thus, we fix *uniform sampling* as our sampling scheme. Furthermore, we use the same sample over the stages, meaning that $S_k = \{(k, b) | b \in S\}$ for varying k in $[1, n]$.

Bandwidth Selection:

We introduced the notation $N_h(b)$ as the neighborhood set for a query point b with the bandwidth h. Thus, through h we determine the neighbor sample points for a query point b. Bandwidth selection procedures (e.g., search methods) developed in the statistics literature have been effectively used mainly within local learning framework to approximate *smooth* functions (see Fan [25] for a nice discussion). In our computational study, we automate the selection of h by the following relationship $h = n_p \delta_q + \delta_r$ where $\delta_q = \lfloor b_0/s \rfloor$ and $\delta_r = b_0 - s\delta_q$ through specifying the *size* parameter $n_p = \{1, \cdots, s-1\}$. We call this bandwidth selection procedure as the *automated bandwidth* selection procedure.

Let us illustrate sampling and bandwidth selection by an example. Consider a

binary knapsack problem with the capacity $b_0 = 50$. Let the sample size $s = 5$. Then, $\delta_q = \lfloor b_0/s \rfloor = 10$ and $\delta_r = 0$. Thus, our sample space $S = \{0, 10, 20, 30, 40, 50\}$ according to the above construction. If we set *size* parameter $n_p = 1$ then $h = n_p\delta_q + \delta_r = 10$. As an example, for the query point $b = 15$ the neighborhood set $N_{10}(b) = \{10, 20\}$. If we set $n_p = 2$ then $h = 20$. Then, $N_{20}(15) = \{0, 10, 20, 30\}$. We note that $N_{20}(15)$ is a larger set than $N_{10}(15)$ due to a larger value of bandwidth h. Generally speaking, the larger the bandwidth the more sample points included in the neighborhood set for a given query point.

On the parametric approximation side, our early computational study suggests that quadratic approximation effectively turns into a linear approximation meaning that quadratic terms become zero in the learning phase. As a result of this computational evidence, we consider only *linear* approximation in the parametric and local parametric approximations. Finally, we fix least-absolute deviation as our parameter learning scheme due to its robust performance.

We use the following notations to present the computational study.

- Let ADP-GL be the approximate dynamic programming with parametric approximation.
- Let ADP-LL be the approximate dynamic programming with local parametric approximation.
- Let ADP-KS be the approximate dynamic programming with nonparametric approximation.
- Let ADP-LI be the approximate dynamic programming with linear interpolation.
- Let ADP-H be the approximate dynamic programming with base-heuristic learning where H is a given base-heuristic, e.g., H can be the proposed greedy heuristic.

- Let v(X) be the objective value of the solution obtained by the methodology X, e.g., v(ADP-H) is the objective value of the solution obtained by ADP-H.
- Let PE(X) be the percentage deviation of the v(X) from the LP value v(LP), i.e., PE(X) $= \frac{v(LP)-v(X)}{v(LP)} \times 100$.
- Let T(X) be the computation time of the methodology X.
- Let s be the sample size, i.e., the cardinality of sample S.
- Let n_p be the *size* parameter, i.e., bandwidth parameter h is implicitly determined through the proposed automated bandwidth selection procedure.

3.5.2 Preliminary Computational Results

The goal of this *preliminary* computational study is to tune the design parameters like the kernel function, the bandwidth for the nonparametric and local parametric approximations. As pointed out earlier in section 3.5.1, for the parametric approximation, we use linear as the functional form, least-absolute deviation criterion as the parameter learning scheme. In addition, we select sample points out of state space through uniform sampling as described in section 3.5.1. For the locally weighted regression, in order to exploit a closed form expression from the literature (see Fan [25]), we use the least-squares criterion as the parameter learning scheme.

We consider the following choices regarding kernel functions:

1. Gaussian: $K(t) = e^{-t^2/2} I(|t| \leq 1)$.
2. Triangular: $K(t) = (1 - |t|) I(|t| \leq 1)$.
3. Beta family: $K(t) = [\ \max\ \{0, 1 - t^2\}]^{\gamma} I(|t| \leq 1),\ \gamma = 0, 1.$

where $I(\cdot)$ is an indicator function, i.e., $I(|t| \leq 1) = 1$ (0) if $|t| \leq 1$ (otherwise). The bandwidth h is tuned through the size parameter n_p as described in section 3.5.1. In this preliminary computation phase, we consider size parameter values of 1, 2 and 3,

i.e., $n_p \in \{1, 2, 3\}$. Once we fix the design parameters (e.g., $K(t) = (1 - |t|)I(|t| \leq 1)$ and $n_p = 1$), we construct problem instances of size 100 and 500 from all types (i.e., uncorrelated, weakly and strongly correlated). For a certain problem instance, we report the computation time and the percentage deviation of the corresponding statistical learning algorithm. For uncorrelated problem instances, we set $A = C = 1,000$ to generate $c_j \sim U(1, C)$ and $a_j \sim U(1, A)$ for all $j \in \{1, \cdots, n\}$. For weakly correlated problem instances, we set $A = 1,000$ and $wc = 100$ to generate $a_j \sim U(1, A)$ and $c_j = \max\{1, U(a_j - wc, a_j + wc)\}$ for all $j \in \{1, \cdots, n\}$. For strongly correlated problem instances, we set $A = 1,000$ and $sc = 100$ to generate $a_j \sim U(1, A)$ and $c_j = a_j + sc$ for all $j \in \{1, \cdots, n\}$. Finally, we set $b_0 = 0.5 \sum_{j=1}^{n} a_j$.

Tables 3.11 and 3.15 illustrate the effectiveness of the *Gaussian* kernel function. These tables suggest that ADP-KS does not generate competitively good quality of solutions compared to those obtained by ADP-LL and ADP-LI. As an example, for the problem instances of size $n = 100$ with varying s and n_p, average percentage deviations of ADP-KS, ADP-LL and ADP-LI are 10.57, 1.24 and 0.52, respectively. We observe that ADP-LL with $n_p > 1$ generates inferior quality of solutions compared to those obtained by ADP-LL with $n_p = 1$. For instance, for the problem instances of size $n = 100$ with varying sample sizes s, average percentage deviations of ADP-LL with $n_p = 2$ and ADP-LL with $n_p = 1$ are 1.26 and 0.39, respectively. Both ADP-LL with $n_p = 1$ and ADP-LI generate reasonably good quality of solutions while ADP-LL consumes longer computation time. For instance, for the problem instances of size $n = 100$ with varying sample size s, average percentage deviations of ADP-LL with $n_p = 1$ and ADP-LI are 0.39 and 0.52, respectively. For the same instances, average computation times of ADP-LL with $n_p = 1$ and ADP-LI are 7.43 and 0.005, respectively. Based on our comparison criteria discussed earlier, we favor ADP-LI over ADP-KS, and ADP-LL once the kernel function is set to the Gaussian.

Tables 3.12 and 3.16 illustrate the effectiveness of the *Triangular* kernel function. For certain values of n_p and s, ADP-KS with Triangular kernel function generates

higher quality of solutions compared to those obtained by ADP-KS with Gaussian kernel function (see Tables 3.11 and 3.15). As an example, for the problem instances of size $n = 100$ under $n_p = 1$, average percentage deviations of ADP-KS with Gaussian kernel function and ADP-KS with Triangular kernel function are 14.28 and 1.15, respectively. This empirical observation suggests the importance of kernel function selection. ADP-LL with Triangular kernel generates similar quality of solutions compared to those produced by ADP-LL with Gaussian kernel. Indeed, average percentage deviations of ADP-LL with Triangular kernel and ADP-LL with Gaussian kernel are 1.42 and 1.24, respectively. We continue to observe that ADP-KS (ADP-LL) with $n_p = 1$ generates superior quality of solutions compared to those obtained by ADP-KS (ADP-LL) with $n_p > 1$. As an example, let us focus on the problem instances of size $n = 500$ with varying sample sizes s, we note that average percentage deviations of ADP-KS (ADP-LL) with $n_p = 2$ and ADP-KS (ADP-LL) with $n_p = 1$ are 3.31 (1.88) and 0.51 (0.50), respectively. On average, ADP-LL consumes longer computation time than ADP-KS. For instance, for the problems of size $n = 500$, average computation times of ADP-KS and ADP-LL are 0.16 and 19.08, respectively. Even though ADP-KS and ADP-LL become better in terms of both solution quality and computation time, ADP-LI continues to be the leading approach for the problems of size $n = 500$. ADP-LI generated solutions with an average percentage deviation of 0.41 in an average computation time of 0.028. The Tables 3.12 and 3.16 illustrate the importance of kernel and bandwidth selection. We observe that, for a certain size parameter value, even though ADP-KS with Gaussian kernel was inferior to ADP-LL and ADP-LI in terms of solution quality, ADP-KS with Triangular kernel could generate competitively good quality of solutions in comparison to those obtained by both ADP-LL and ADP-LI.

Tables 3.13 and 3.17 examine the effectiveness of the kernel function *Beta* with $\delta = 0$. ADP-KS with Gaussian kernel generated better quality of solutions compared to those produced by ADP-KS with Beta ($\delta = 0$). For the problem instances of size $n = 100$ ($n = 500$), average percentage deviations of ADP-KS with Gaussian kernel

and ADP-KS with Beta ($\delta = 0$) are 10.77 (48.38) and 14.11 (57.54), respectively (summary statistics from Tables 3.11 and 3.15). The same tendency is observed in the case of ADP-LL, that is, ADP-LL with Gaussian kernel generated better quality of solutions compared to those obtained by ADP-LL with Beta ($\delta = 0$). For the problem instances of size $n = 100$ ($n = 500$), average percentage deviations of ADP-LL with Gaussian kernel and ADP-LL with Beta ($\delta = 0$) are 1.24 (3.78) and 1.71 (1.79), respectively (summary statistics from Tables 3.11 and 3.15). Based on our comparison of ADP-KS (ADP-LL) with Triangular and ADP-KS (ADP-LL) with Gaussian, we can conclude that ADP-KS (ADP-LL) with Triangular generates better quality of solutions compared to those produced by ADP-KS (ADP-LL) with Beta ($\delta = 0$).

Tables 3.14 and 3.18 examine the effectiveness of kernel selection *Beta* with $\delta = 1$. Our available computational evidence suggests that ADP-KS (ADP-LL) with Gaussian dominates ADP-KS (ADP-LL) with Beta ($\delta = 0$). Thus, we need to compare ADP-KS (ADP-LL) with Beta ($\delta = 1$) to ADP-KS (ADP-LL) with Gaussian. ADP-KS with Gaussian kernel generated inferior quality of solutions compared to those obtained by ADP-KS with Beta ($\delta = 1$). For the problem instances of size $n = 100$ ($n = 500$), average percentage deviations of ADP-KS with Gaussian kernel and ADP-KS with Beta ($\delta = 1$) are 10.77 (48.38) and 3.12 (4.88), respectively (summary statistics from Tables 3.11 and 3.15). We also compare ADP-LL with Gaussian with ADP-LL with Beta ($\delta = 1$). For the problem instances of size $n = 100$, average percentage deviations of ADP-LL with Gaussian kernel and ADP-LL with Beta ($\delta = 1$) are 1.24 and 1.43, respectively (summary statistics from Tables 3.11 and 3.15). For the problem instances of size $n = 500$, average percentage deviations of ADP-LL with Gaussian kernel and ADP-LL with Beta ($\delta = 1$) are 1.71 and 1.51, respectively (summary statistics from Tables 3.11 and 3.15). In contrast to the case of ADP-KS, we do not have a clear-cut winner in terms of solution quality in the case of ADP-LL once we compare ADP-LL with Gaussian with ADP-LL with Beta ($\delta = 1$). Average percentage deviations of ADP-KS with Triangular are 2.24 and 3.33 for the problems of size of $n = 100$ and $n = 500$, respectively, in comparison to the

average percentage deviation values of 3.12 and 4.88 of ADP-KS with Beta ($\delta = 1$) for the corresponding instances, respectively. Average percentage deviations of ADP-LL with Triangular are 1.42 and 1.41 for the problems of size $n = 100$ and $n = 500$, respectively, in comparison to the average percentage deviation values of 1.43 and 1.51 for ADP-LL with Beta ($\delta = 1$) for the corresponding instances, respectively. Overall, on average, ADP-KS (ADP-LL) with Triangular generates better quality of solutions compared to ADP-KS (ADP-LL) with Beta ($\delta = 1$). ADP-LI continues to be the leading approach among ADP-KS with Beta ($\delta = 1$) and ADP-LL with Beta ($\delta = 1$). Average percentage deviations of ADP-LI are 0.52 and 0.41 for the problems of size $n = 100$ and $n = 500$, respectively, while the closest values are 1.24 and 1.51 obtained by ADP-LL with Gaussian kernel and ADP-LL with Beta ($\delta = 1$) for the corresponding instances, respectively.

s	n_p	T(ADP-KS)	PE(ADP-KS)	T(ADP-LL)	PE(ADP-LL)	T(ADP-LI)	PE(ADP-LI)
10	1	0.015	35.97	0.39	0.66	0.000	0.65
	2	0.015	19.32	0.39	2.32		
	3	0.015	15.17	0.39	3.18		
20	1	0.045	13.01	2.59	0.37	0.005	0.43
	2	0.050	8.68	2.58	1.12		
	3	0.050	9.08	2.59	1.99		
30	1	0.10	4.06	8.16	0.12	0.005	0.62
	2	0.10	4.49	8.14	0.65		
	3	0.10	4.81	8.13	1.62		
40	1	0.18	4.09	18.59	0.39	0.01	0.39
	2	0.18	5.36	18.48	0.96		
	3	0.17	5.19	18.47	1.46		

Table 3.11: Preliminary Results: $n = 100$, Kernel: Gaussian.

s	n_p	T(ADP-KS)	PE(ADP-KS)	T(ADP-LL)	PE(ADP-LL)
10	1	0.01	0.68	0.21	1.48
	2	0.01	3.49	0.21	1.64
	3	0.01	6.10	0.21	2.87
20	1	0.02	1.62	1.35	1.09
	2	0.02	2.66	1.34	1.58
	3	0.02	3.85	1.34	1.26
30	1	0.04	1.08	4.23	0.63
	2	0.04	1.43	4.22	1.31
	3	0.04	2.02	4.23	1.51
40	1	0.07	1.22	9.58	0.9
	2	0.07	1.10	9.58	1.61
	3	0.07	1.63	9.58	1.21

Table 3.12: Preliminary Results: $n = 100$, Kernel: Triangular.

s	n_p	T(ADP-KS)	PE(ADP-KS)	T(ADP-LL)	PE(ADP-LL)
10	1	0.00	41.53	0.20	1.48
	2	0.01	24.80	0.21	3.03
	3	0.00	19.55	0.21	3.13
20	1	0.02	16.51	1.32	1.48
	2	0.02	17.44	1.32	1.00
	3	0.02	11.02	1.33	2.06
30	1	0.04	5.40	4.16	0.63
	2	0.04	5.01	4.16	1.47
	3	0.04	6.25	4.15	1.90
40	1	0.07	6.33	9.48	0.9
	2	0.07	7.08	9.40	0.91
	3	0.07	8.44	9.41	1.60

Table 3.13: Preliminary Results: $n = 100$, Kernel: Beta with $\delta = 0$.

s	n_p	T(ADP-KS)	PE(ADP-KS)	T(ADP-LL)	PE(ADP-LL)
10	1	0.00	3.62	0.21	1.48
	2	0.01	4.27	0.21	2.38
	3	0.00	6.92	0.21	3.04
20	1	0.02	3.12	1.35	1.09
	2	0.02	3.47	1.38	2.25
	3	0.02	4.71	1.36	1.46
30	1	0.04	1.44	4.22	0.63
	2	0.05	2.01	4.23	0.85
	3	0.04	2.32	4.22	1.48
40	1	0.07	2.26	9.56	0.90
	2	0.07	1.41	9.54	0.95
	3	0.07	1.90	9.60	0.68

Table 3.14: Preliminary Results: $n = 100$, Kernel: Beta with $\delta = 1$.

s	n_p	T(ADP-KS)	PE(ADP-KS)	T(ADP-LL)	PE(ADP-LL)	T(ADP-LI)	PE(ADP-LI)
10	1	0.07	74.22	1.98	0.76	0.01	0.76
	2	0.07	57.98	1.95	3.88		
	3	0.07	37.52	1.96	4.53		
20	1	0.22	64.20	12.91	0.51	0.02	0.47
	2	0.23	53.52	12.87	1.93		
	3	0.23	39.25	12.86	3.04		
30	1	0.48	55.02	40.64	0.19	0.03	0.19
	2	0.48	47.77	40.51	1.32		
	3	0.48	38.03	40.50	1.95		
40	1	0.83	43.91	93.29	0.17	0.05	0.21
	2	0.82	37.79	92.75	0.74		
	3	0.83	31.51	92.75	1.44		

Table 3.15: Preliminary Results: $n = 500$, Kernel: Gaussian.

s	n_p	T(ADP-KS)	PE(ADP-KS)	T(ADP-LL)	PE(ADP-LL)
10	1	0.03	0.77	1.03	0.90
	2	0.03	7.86	1.02	3.31
	3	0.03	13.54	1.02	4.12
20	1	0.09	0.81	6.66	0.48
	2	0.09	2.73	6.66	1.48
	3	0.09	6.41	6.66	2.10
30	1	0.19	0.21	20.93	0.34
	2	0.19	1.46	20.88	0.78
	3	0.19	3.04	20.91	1.60
40	1	0.32	0.23	47.76	0.27
	2	0.32	1.19	47.76	0.46
	3	0.32	1.74	47.76	1.07

Table 3.16: Preliminary Results: $n = 500$, Kernel: Triangular.

s	n_p	T(ADP-KS)	PE(ADP-KS)	T(ADP-LL)	PE(ADP-LL)
10	1	0.03	80.10	1.00	0.90
	2	0.03	64.41	1.00	3.68
	3	0.03	46.79	1.00	4.53
20	1	0.09	68.58	6.54	0.48
	2	0.09	62.29	6.51	2.18
	3	0.09	48.86	6.56	3.25
30	1	0.17	65.77	20.52	0.34
	2	0.17	57.85	20.47	1.41
	3	0.17	50.59	20.52	2.00
40	1	0.29	53.18	46.88	0.27
	2	0.29	49.51	46.79	0.77
	3	0.29	42.51	47.01	1.69

Table 3.17: Preliminary Results: $n = 500$, Kernel: Beta with $\delta = 0$.

s	n_p	T(ADP-KS)	PE(ADP-KS)	T(ADP-LL)	PE(ADP-LL)
10	1	0.03	3.83	1.03	0.90
	2	0.03	8.90	1.03	3.60
	3	0.03	14.89	1.03	4.11
20	1	0.09	3.21	6.67	0.48
	2	0.09	3.30	6.65	1.59
	3	0.09	7.47	6.68	2.55
30	1	0.18	2.98	20.94	0.34
	2	0.18	2.47	20.88	0.88
	3	0.18	3.60	20.93	1.71
40	1	0.30	3.40	47.74	0.27
	2	0.31	1.87	47.81	0.61
	3	0.31	2.63	47.76	1.12

Table 3.18: Preliminary Results: $n = 500$, Kernel: Beta with $\delta = 1$.

This preliminary computational study emphasizes the selection of kernel and bandwidth in the case of nonparametric approximation. Once we set the kernel function to Triangular and set size parameter n_p to 1, both ADP-KS and ADP-LL generate better quality of solutions compared to those obtained by ADP-KS and ADP-LL with varying kernel functions and bandwidth selections. Thus, we suggest to use nonparametric approximation that utilizes Triangular kernel function with the smallest size parameter, i.e., $n_p = 1$. Overall, ADP-LI is the leading methodology in terms of both solution quality and computation time. An increase in the sample size does not necessarily reflect an improvement in the quality of solutions obtained by ADP-LI. As a first example, for the problems of size $n = 100$, average percentage deviations of ADP-LI are 0.43 and 0.62 for the sample sizes of 20 and 30, respectively. As a second example, for the problems of size $n = 500$, average percentage deviations of ADP-LI are 0.19 and 0.21 for the sample sizes of 30 and 40, respectively. Even though, we have these counter examples, we observe that the quality of solutions obtained by ADP-LI tends to be improving as the sample size increases. We can illustrate this tendency for the problem instances of size $n = 500$ by noting that average percentage deviations of ADP-LI are 0.76, 0.47, 0.19 and 0.21 for the sample sizes of 10, 20, 30 and 40, respectively. We emphasize that ADP-LI is a *special* case of ADP-KS with Triangular kernel function and nearest neighborhood selection as introduced in section 3.3.4. In our computations, we use the linear interpolation algorithm as described in section 3.3.4.

3.5.3 Detailed Computational Results

In this section, our aim is to compare the effectiveness of ADP methodologies utilizing both statistical and base-heuristic learning for a larger set of binary knapsack problem instances. Regarding statistical learning framework, we consider both nonparametric and parametric approximation. As pointed earlier, our preliminary computational study suggests to use nonparametric approximation through setting the kernel function to Triangular and size parameter to 1. Thus, in this section, ADP-KS

means approximate dynamic programming approach under nonparametric approximation where Triangular kernel is selected as kernel function and the bandwidth h is calculated by setting the size parameter to 1, i.e., $n_p = 1$. Our observation that ADP-LL consumes a large amount of computation time in the preliminary computational study suggests that this methodology might not become practical for larger problem instances. So, we exclude ADP-LL from a further comparison. Regarding the parametric approximation, we use linear function as the functional form and least-absolute deviation criterion as the parameter learning scheme. We also include ADP-LI, the leading methodology in our preliminary computational study, in order to illustrate its effectiveness for a larger set of problem instances. In this section, we will also delineate the effectiveness of base-heuristic learning. We use the greedy heuristic described in Table 3.8 within the base-heuristic learning methodology.

Given the number of variables n we generate 5 test problems in each class (i.e., uncorrelated, weakly or strongly correlated) with different parameters A, C, wc, sc as provided in Table 3.19. We set $b_0 = 0.5 \sum_{j=1}^{n} a_j$. All computational studies are done on a Dell Precision 410 with Linux operating system.

UC		WC		SC	
A	C	A	wc	A	sc
1,000	1,000	1,000	100	1,000	100
2,000	1,000	2,000	500	2,000	500
10,000	5,000	10,000	3,000	10,000	3,000
500	700	500	100	500	100
500	300	500	10	500	10

Table 3.19: Parameters for the uncorrelated, weakly and strongly correlated type instances.

3.5.3.1 Computational results for the statistical learning framework

In this section, our aim is to compare ADP-KS, ADP-GL and ADP-LI under the proposed algorithmic design (i.e., kernel function=Triangular, sampling scheme=uniform

sampling etc.) in terms of solution quality and computation time for a larger set of binary knapsack problems.

Tables 3.20, 3.21 and 3.22 present the computational results for uncorrelated type binary knapsack problems. We observe that ADP-KS and ADP-LI generate better quality solutions than those produced by ADP-GL. In fact, ADP-GL generates not strong quality of solutions. Let us first summarize our computational findings. For the problems of size $n = 1,000$, average percentage deviations of ADP-KS, ADP-GL and ADP-LI are 0.48, 9.83 and 0.14, respectively, while their corresponding average computation times are 2.55, 2.09 and 0.76, respectively. For the problems of size $n = 5,000$, average percentage deviations of ADP-KS, ADP-GL and ADP-LI are 0.30, 8.36 and 0.09, respectively, while their corresponding average computation times are 13.07, 8.68 and 3.89, respectively. For the problems of size $n = 10,000$, average percentage deviations of ADP-KS, ADP-GL and ADP-LI are 0.81, 8.12 and 0.07, respectively, while their corresponding average computation times are 26.83, 17.42 and 7.81, respectively. Overall, ADP-LI is the leading algorithm in terms of both solution quality and computation time. The next best statistical learning approach for the uncorrelated type instances is obviously ADP-KS. We summarize our findings regarding the sample size as follows. An increase in the sample size does not necessarily mean an improvement in the quality of the solutions generated by an approach. As an example, for problems of size $n = 1,000$, ADP-KS achieves an average percentage deviation of 0.34 for a sample size of 100, while it achieves a worse average percentage deviation of 0.46 for a larger sample size of 150. Indeed, under ADP-GL, an increase in the sample size did not change the quality of the solutions produced. In contrast to ADP-KS and ADP-GL, the average quality of solutions either improved or stayed same under ADP-LI. As an example, for the problem instances of size $n = 1,000$, average percentage deviations of ADP-LI are 0.20, 0.17, 0.13, 0.10 and 0.10 for the sample sizes of 50, 100, 150, 200 and 250, respectively.

Tables 3.23, 3.24 and 3.25 present the computational results for weakly correlated

s	T(ADP-KS)	PE(ADP-KS)	T(ADP-GL)	PE(ADP-GL)	T(ADP-LI)	PE(ADP-LI)
50	0.59	0.22	0.78	9.83	0.12	0.20
100	1.28	0.34	1.31	9.83	0.34	0.17
150	2.27	0.46	2.03	9.83	0.66	0.13
200	3.55	0.37	2.78	9.83	1.09	0.10
250	5.06	1.00	3.54	9.83	1.61	0.10

Table 3.20: Comparison of ADP-KS, ADP-GL, and ADP-LI: Uncorrelated type instances, $n = 1,000$.

s	T(ADP-KS)	PE(ADP-KS)	T(ADP-GL)	PE(ADP-GL)	T(ADP-LI)	PE(ADP-LI)
50	3.12	0.15	3.60	8.36	0.61	0.16
100	6.57	0.08	5.80	8.36	1.73	0.08
150	11.75	0.15	8.44	8.36	3.38	0.09
200	18.16	0.73	11.24	8.36	5.53	0.06
250	25.73	0.37	14.33	8.36	8.18	0.05

Table 3.21: Comparison of ADP-KS, ADP-GL, and ADP-LI: Uncorrelated type instances, $n = 5,000$.

type instances. We observe that ADP-KS and ADP-LI continue to generate higher quality solutions than those obtained by ADP-GL. We first summarize our computational results as follows. For the problems of size $n = 1,000$, average percentage deviations of ADP-KS, ADP-GL and ADP-LI are 1.82, 12.43 and 0.14, respectively, while their corresponding average computation times are 2.55, 1.90 and 0.75, respectively. For the problems of size $n = 5,000$, average percentage deviations of ADP-KS, ADP-GL and ADP-LI are 1.58, 12.64 and 0.16, respectively, while their corresponding average computation times are 13.12, 8.08 and 3.88, respectively. For the problems of size $n = 10,000$, average percentage deviations of ADP-KS, ADP-GL and ADP-LI are 2.87, 12.44 and 0.17, respectively, while their corresponding average computation times are 26.03, 17.48 and 7.79, respectively. Overall, ADP-LI remains to be the leading algorithm in terms of both solution quality and computation time. ADP-KS remains to be the second best statistical learning approach. Finally, ADP-GL still generates worst quality of solutions compared to those obtained by both ADP-KS and ADP-LI. Our findings about sensitivity of sample size remain to be similar as in the case of uncorrelated type instances. In words, an increase in the sample size

s	T(ADP-KS)	PE(ADP-KS)	T(ADP-GL)	PE(ADP-GL)	T(ADP-LI)	PE(ADP-LI)
50	6.68	0.16	7.33	8.12	1.23	0.15
100	13.63	0.24	11.55	8.12	3.48	0.08
150	24.27	0.67	16.94	8.12	6.77	0.05
200	37.12	0.97	22.79	8.12	11.11	0.04
250	52.44	2.02	28.49	8.12	16.46	0.04

Table 3.22: Comparison of ADP-KS, ADP-GL and ADP-LI: Uncorrelated type instances, $n = 10,000$.

does not necessarily mean an improvement in the quality of the solutions generated by an approach. As an example, for problems of size $n = 5,000$, ADP-KS achieves an average percentage deviation of 0.57 for a sample size of 100, while it achieves a worse average percentage deviation of 1.65 for a larger sample size of 150. We again observe that an increase in the sample size did not change the quality of the solutions produced by ADP-GL. In contrast to ADP-KS and ADP-GL, we continue to observe that the average quality of solutions either improved or stayed same under ADP-LI. As an example, for the problem instances of size $n = 5,000$, average percentage deviations of ADP-LI are 0.36, 0.17, 0.10, 0.09 and 0.07 for the sample sizes of 50, 100, 150, 200 and 250, respectively.

s	T(ADP-KS)	PE(ADP-KS)	T(ADP-GL)	PE(ADP-GL)	T(ADP-LI)	PE(ADP-LI)
50	0.59	0.73	0.69	12.43	0.12	0.24
100	1.28	1.74	1.18	12.43	0.34	0.14
150	2.28	2.30	1.81	12.43	0.67	0.10
200	3.54	1.89	2.54	12.43	1.10	0.13
250	5.07	2.44	3.26	12.43	1.51	0.11

Table 3.23: Comparison of ADP-KS, ADP-GL and ADP-LI: Weakly-correlated type instances, $n = 1,000$.

Tables 3.26, 3.27 and 3.28 present the computational results for strongly correlated type instances. We observe that both ADP-KS and ADP-LI continue to generate higher quality solutions than those obtained by ADP-GL. Let us first summarize our computational findings as follow. For the problems of size $n = 1,000$, average percentage deviations of ADP-KS, ADP-GL and ADP-LI are 3.61, 8.87 and 0.09,

s	T(ADP-KS)	PE(ADP-KS)	T(ADP-GL)	PE(ADP-GL)	T(ADP-LI)	PE(ADP-LI)
50	3.08	0.82	3.11	12.64	0.61	0.36
100	6.62	0.57	5.36	12.64	1.72	0.17
150	11.83	1.65	7.92	12.64	3.38	0.10
200	18.22	2.35	10.61	12.64	5.52	0.09
250	25.84	2.52	13.42	12.64	8.19	0.07

Table 3.24: Comparison of ADP-KS, ADP-GL and ADP-LI: Weakly-correlated type instances, $n = 5,000$.

s	T(ADP-KS)	PE(ADP-KS)	T(ADP-GL)	PE(ADP-GL)	T(ADP-LI)	PE(ADP-LI)
50	6.24	1.12	7.30	12.44	1.23	0.38
100	13.15	2.66	11.75	12.44	3.47	0.18
150	23.40	3.41	17.07	12.44	6.76	0.12
200	36.18	3.82	22.71	12.44	11.07	0.09
250	51.20	3.36	28.56	12.44	16.42	0.07

Table 3.25: Comparison of ADP-KS, ADP-GL and ADP-LI: Weakly-correlated type instances, $n = 10,000$.

respectively, while their corresponding average computation times are 2.52, 1.89 and 0.77, respectively. For the problems of size $n = 5,000$, average percentage deviations of ADP-KS, ADP-GL and ADP-LI are 2.83, 8.76 and 0.11, respectively, while their corresponding average computation times are 12.75, 8.05 and 3.81, respectively. For the problems of size $n = 10,000$, average percentage deviations of ADP-KS, ADP-GL and ADP-LI are 3.83, 8.87 and 0.12, respectively, while their corresponding average computation times are 26.66, 16.50 and 7.78, respectively. Overall, ADP-LI remains to be the leading algorithm in terms of both solution quality and computation time. ADP-KS and ADP-GL remain to be the second and third best statistical learning approach, respectively. Our findings about sensitivity of sample size remain to be similar as in the case of uncorrelated and weakly correlated type instances, i.e., an increase in the sample size does not necessarily mean an improvement in the quality of the solutions produced by an approach. As an example, for problems of size $n = 10,000$, ADP-KS achieves an average percentage deviation of 2.01 for a sample size of 50, while it achieves a worse average percentage deviation of 3.84 for a larger sample size of 100. We again observe that an increase in the sample size did not change the

quality of the solutions produced by ADP-GL. For the first time, an increase in the sample size did not improve the quality of solutions obtained by ADP-LI. Indeed, ADP-LI achieves an average percentage deviation of 0.05 (0.07) for a sample size of 150 (200), while it achieves a worse average percentage deviation of 0.07 (0.09) for a larger sample size of 200 (250). Except these instances, we continue to observe that the average quality of solutions either improved or stayed same under ADP-LI for the problem instances of size $n = 5,000$ and $n = 10,000$. As an example, for the problem instances of size $n = 5,000$, average percentage deviations of ADP-LI are 0.26, 0.11, 0.07, 0.05 and 0.04 for the sample sizes of 50, 100, 150, 200 and 250, respectively.

s	T(ADP-KS)	PE(ADP-KS)	T(ADP-GL)	PE(ADP-GL)	T(ADP-LI)	PE(ADP-LI)
50	0.58	0.59	0.68	8.87	0.12	0.17
100	1.26	3.99	1.16	8.87	0.35	0.10
150	2.26	4.33	1.81	8.87	0.67	0.05
200	3.50	5.07	2.51	8.87	1.09	0.07
250	4.99	4.08	3.28	8.87	1.63	0.09

Table 3.26: Comparison of ADP-KS, ADP-GL and ADP-LI: Strongly-correlated type instances, $n = 1,000$.

s	T(ADP-KS)	PE(ADP-KS)	T(ADP-GL)	PE(ADP-GL)	T(ADP-LI)	PE(ADP-LI)
50	2.91	0.77	3.26	8.76	0.61	0.26
100	6.45	0.63	5.34	8.76	1.73	0.11
150	11.45	3.88	7.82	8.76	3.39	0.07
200	17.69	3.28	10.54	8.76	5.58	0.05
250	25.24	6.11	13.28	8.76	8.22	0.04

Table 3.27: Comparison of ADP-KS, ADP-GL and ADP-LI: Strongly-correlated type instances, $n = 5,000$.

We observe that the quality of solutions generated by ADP-KS degrades for the problem instances with stronger correlation. Average percentage deviations of ADP-KS are 0.53, 2.09 and 3.42 for all uncorrelated, weakly and strongly correlated type problem instances, respectively. For a certain number of variables n, average computation times of ADP-KS for different types of problem instances stay close to one

s	T(ADP-KS)	PE(ADP-KS)	T(ADP-GL)	PE(ADP-GL)	T(ADP-LI)	PE(ADP-LI)
50	6.32	2.01	6.70	8.87	1.22	0.28
100	13.62	3.84	11.06	8.87	3.47	0.13
150	24.01	4.05	16.18	8.87	6.74	0.08
200	36.94	4.61	21.45	8.87	11.04	0.06
250	52.39	4.65	27.13	8.87	16.41	0.05

Table 3.28: Comparison of ADP-KS, ADP-GL and ADP-LI: Strongly-correlated type instances, $n = 10,000$.

another. As an example, let us focus on the problem instances of size $n = 1,000$. We observe that, average computation times of ADP-KS are 2.55, 2.55 and 2.52 for uncorrelated, weakly and strongly correlated type problem instances, respectively. Thus, from a computation time point of view, ADP-KS is a robust methodology across different types of instances. For the same size problem instances, we observe that average percentage deviations of ADP-KS are 0.48, 1.82 and 3.61 for uncorrelated, weakly and strongly correlated type problem instances, respectively. Based on this result, we can not say that ADP-KS is a robust methodology from a percentage point of view across different types of instances. Overall, ADP-GL generated poor quality of solutions compared to those obtained both by ADP-KS and ADP-LI. Indeed, average percentage deviations of ADP-GL are 8.77, 12.50 nd 8.83 for all uncorrelated, weakly and strongly correlated type problem instances, respectively. In comparison to ADP-GL, ADP-KS (ADP-LI) achieves average percentage deviations of 0.53 (0.10), 2.09 (0.16) and 3.42 (0.11) for all uncorrelated, weakly and strongly correlated type problem instances, respectively. Overall, ADP-LI is the leading algorithm among the proposed statistical learning approaches. It achieves very good quality of solutions within short times. ADP-LI attains average percentage deviations of 0.10, 0.16 and 0.11 for all uncorrelated, weakly and strongly correlated type problem instances, respectively, while its corresponding average computation times are 4.15, 4.14 and 4.12 for all uncorrelated, weakly and strongly correlated type problem instances, respectively. These results suggest that ADP-LI is robust across different types of instances. Let us illustrate this observation by an example. For problem instances of size $n = 1,000$, average percentage deviations of ADP-LI are 0.14, 0.14 and 0.09 for

uncorrelated, weakly and strongly correlated type instances, respectively. This observation indicates that the quality of solutions stayed quite stable, hence ADP-LI can be considered as a robust methodology in terms of solution quality across different types of instances. For the same size problem instances, average computation times of ADP-LI are 0.76, 0.75 and 0.77 for uncorrelated, weakly and strongly correlated type instances, respectively, illustrating that computation times are almost same for different type of instances. Hence, ADP-LI is robust in terms of computation time.

3.5.3.2 Computational results for the base-heuristic learning framework

In this section, our basic aim is to illustrate the effectiveness of the approximate dynamic programming with base-heuristic learning (ADP-H) on the same randomly generated problem instances. We exploit the proposed greedy heuristic (described in Table 3.8 on page 52) as a base-heuristic for the binary knapsack problem. We also compare ADP-H with one of the state-of-art commercial packages like CPLEX.

In Table 3.29 we report the computational results of the base-heuristic learning for uncorrelated type instances. We observe that ADP-H achieves near-optimal solutions in short computation times. As an example, for problems of size $n = 10,000$, ADP-H attains an average percentage deviation of 0.0017 and an average computation time of 4.84. This table also suggests that ADP-H is robust in terms of both solution quality and computation time for problem instances of certain number of variables. For instance, for the problems of size $n = 5,000$, minimum, average and maximum computation times of ADP-H are 1.36, 2.54 and 3.39, respectively, illustrating that computation times stay quite stable for different instances of the same size. In addition, minimum, average and maximum percentage deviation values of ADP-H are 0.0016, 0.0038 and 0.0068, respectively, depicting that percentage deviation values are close to one another. Hence, ADP-H is robust in terms of solution quality. We also compare ADP-H and CPLEX. In order to make a legitimate comparison, for a given problem instance wet set the parameter `CPX-PARAM-EPGAP`, relative

tolerance on the gap between the best integer objective v(B) and the objective of the best node remaining v(BN), to PE(ADP-H)/100. When the relative tolerance $\frac{|v(BN)-v(B)|}{|1.0+v(BN)|}$ falls below CPX-PARAM-EPAGAP, CPLEX stops the optimization. We retrieve the necessary statistics such as objective value of the solution by CPLEX, number of nodes required by CPLEX etc. from the CPLEX environment. This table suggests that CPLEX requires considerably longer average computation times than ADP-H to achieve the same level of percentage deviation. As a summary statistics, average computation time of CPLEX is 334.15 CPU seconds while ADP-H achieves the same quality of solutions within 2.51 CPU seconds.

Tables 3.30 and 3.31 continue to suggest ADP-H produces near-optimal solutions in very short times. In summary, average percentage deviation and computation time of ADP-H are 0.0204 and 3.09 for weakly correlated type instances while the corresponding values are 0.0193 and 1.15 for strongly correlated type instances. In contrast to ADP-H, average computation time of CPLEX are 416.50 and 276.26 for weakly and strongly correlated type instances, respectively. Thus, CPLEX consumes significantly longer computation time to achieve the same level of percentage deviation, or the same quality of solutions, obtained by ADP-H. ADP-H continues to demonstrate a robust performance for a certain number of variables n. As an example, for strongly correlated type instances of size $n = 10,000$, minimum, average and maximum computation times of ADP-H are 1.19, 2.77 and 5.19 CPU seconds, respectively, illustrating that computation times stay close to one another. Furthermore, minimum, average and maximum percentage deviation values of ADP-H are 0.0004, 0.0025 and 0.0045, respectively, suggesting robustness in terms of solution quality.

An important observation is that ADP-H improves significantly over the base-heuristic H, namely the greedy-heuristic, meaning that ADP-H generates higher quality of solutions compared to those produced by H. In order to delineate this observation we calculate *percentage improvement* PI(H) of ADP-H over H. PI(H) is calculated by $\frac{PE(H)-PE(ADP-H)}{PE(H)} \times 100$, where PE(H) is the percentage deviation of the heuris-

tic value v(H) from the linear programming value v(LP), i.e, $\frac{v(LP)-v(H)}{v(LP)} \times 100$ and PE(ADP-H) is the percentage deviation of the ADP-H value, v(ADP-H), from the LP value v(LP), i.e., $\frac{v(LP)-v(ADP-H)}{v(LP)} \times 100$. Our available computational study suggests that the greedy heuristic H generates good solutions in very short times. Through Table 3.32, we illustrate that ADP-H provides a framework that generates better quality of solutions than those obtained by H. Average percentage improvements of H are 50.85, 37.78 and 81.51 for uncorrelated, weakly and strongly correlated type instances, respectively. This observation suggests that, on average, greedy heuristic constructs better quality of solutions for uncorrelated and weakly correlated problem instances than strongly correlated ones since average percentage improvements for uncorrelated and weakly correlated instances are higher than those for strongly correlated instances.

	CPLEX		T(ADP-H)			PE(ADP-H)		
n	avg. time	avg. no nodes	min	avg	max	min	avg	max
1,000	8.73	5,321.20	0.06	0.15	0.22	0.003	0.0304	0.0689
5,000	272.86	30,413.20	1.36	2.54	3.39	0.0016	0.0038	0.0068
10,000	720.87	31,846.40	2.67	4.84	6.44	0.0001	0.0017	0.0061

Table 3.29: Comparison of ADP-H with CPLEX: Uncorraleted type instances, H: Greedy Heuristic.

	CPLEX		T(ADP-H)			PE(ADP-H)		
n	avg. time	avg. no nodes	min	avg	max	min	avg	max
1,000	41.62	25,401.60	0.19	0.26	0.29	0.0038	0.0294	0.0621
5,000	363.58	43,701.60	1.61	4.91	14.27	0.0014	0.0119	0.0233
10,000	842.95	42,629.80	1.16	4.09	10.52	0.0007	0.0020	0.0040

Table 3.30: Comparison of ADP-H with CPLEX: Weakly-correlated type instances, H: Greedy Heuristic.

Motivated by the challenge that binary knapsack problems with large coefficients a_j, b_0 become intractable and challenging, we conduct an additional computational

	CPLEX		T(ADP-H)			PE(ADP-H)		
n	avg time	avg no nodes	min	avg	max	min	avg	max
1,000	189.55	134,417.40	0.10	0.26	0.61	0.0069	0.0282	0.0470
5,000	272.54	31,909.00	0.33	0.73	1.21	0.0009	0.0051	0.0106
10,000	366.70	17,769.20	1.19	2.77	5.19	0.0004	0.0025	0.0045

Table 3.31: Comparison of ADP-H with CPLEX: Strongly-correlated type instances, H: Greedy Heuristic.

	PI(H)		
n	UC	WC	SC
1,000	46.17	30.14	81.31
5,000	35.08	56.72	81.61
10,000	71.30	26.49	81.60
average	50.85	37.78	81.51

Table 3.32: Average percentage improvement of ADP-H over H: Uncorrelated, Weakly-correlated and Strongly-correlated type instances, H: Greedy Heuristic.

study. We know that a binary integer programming problem can be represented as a binary knapsack problem. Even though, the transformation is straightforward the corresponding binary knapsack problems have *large* coefficients a_j, b_0. Computational experience in the literature suggests that these problems are difficult to optimize (see Martello et. al. [50], Pisinger [66] and Chvatal [16]).

In the following computational study, given the number of variables n we generate 5 test problems in each class (i.e., uncorrelated, weakly or strongly correlated) with different parameters A, C, wc, sc as provided in Table 3.33. Let $x \sim U(1, X)$ denote an integer number that is uniformly generated in $[1, X]$. Let us first explain how we generate problem instances with large coefficients. For each uncorrelated type instance, we use A, C from Table 3.33 to generate $c_j \sim U(1, C)$ and $a_j \sim U(1, A)$ for all $j \in \{1, \cdots, n\}$. For each weakly correlated instance, we use A, wc from Table 3.33 to generate $a_j \sim U(1, A)$ and $c_j = \max\{1, U(a_j - wc, a_j + wc)\}$ for all $j \in \{1, \cdots, n\}$. For each strongly correlated instance, we use A, sc from Table 3.33

to generate $a_j \sim U(1, A)$ and $c_j = a_j + sc$ for all $j \in \{1, \cdots, n\}$. Finally, we set $b_0 = 0.5 \sum_{j=1}^{n} a_j$. All computational studies are done on a Dell Precision 410 with Linux operating system.

UC		WC		SC	
A	C	A	wc	A	sc
10,000	10,000	10,000	1,000	10,000	100
100,000	10,000	100,000	5,000	100,000	1,000
100,000	50,000	100,000	10,000	100,000	5,000
100,000	100,000	100,000	25,000	100,000	1,000
1,000,000	100,000	1,000,000	100,000	1,000,000	10,000

Table 3.33: Parameters for the uncorrelated, weakly and strongly correlated type instances.

Tables 3.34, 3.35 and 3.36 illustrate that ADP-H generates near-optimal solutions in very short times. Let us first summarize our computational results as follows. For uncorrelated type instances, average percentage deviation and average computation time of ADP-H are 0.0124 and 2.59, respectively. For weakly correlated type instances, average percentage deviation and average computation time of ADP-H are 0.0228 and 3.21, respectively. For strongly correlated type instances, average percentage deviation and average computation time of ADP-H are 0.0043 and 6.17, respectively. This suggests that ADP-H is robust across different types of instances. Indeed, for problems of size $n = 1,000$, average computation times are 0.20, 0.40 and 0.68 for uncorrelated, weakly and strongly correlated type problem instances, respectively. This result illustrates that ADP-H is robust in terms of computation time across different type of instances. In addition, for problems of size $n = 1,000$, average percentage deviations are 0.0243, 0.0566 and 0.0097 for uncorrelated, weakly and strongly correlated type problem instances, respectively, indicating that quality of solutions stayed close to one another. This result supports that ADP-H is robust in terms of solution quality across different types of problem instances. In contrast to being robust across different types of instances, ADP-H is also robust for a cer-

tain number of variables under a given type (i.e, uncorrelated, weakly or strongly correlated types). Let us illustrate this kind of robustness by an example. Let us focus on the weakly correlated instances of size $n = 5,000$. Minimum, average and maximum computation times of ADP-H are 1.40, 2.70 and 4.09, respectively, while minimum, average and maximum percentage deviations of ADP-H are 0.0031, 0.0085 and 0.0192, respectively. We observe small deviations both in solution quality and computation time which implies that ADP-H is robust in terms of both solution quality and computation time for a certain problem size. Tables 3.34, 3.35 and 3.36 also compare ADP-H with CPLEX. As described earlier, we set `CPX-PARAM-EPGAP` to PE(ADP-H)/100. We retrieve the necessary statistics such as objective value of the solution obtained by CPLEX, number of nodes of CPLEX etc. from the CPLEX environment. These tables suggest that CPLEX requires *significantly* longer computation times than ADP-H to achieve the same level of percentage deviation, or the same level of quality of solutions. Average computation times of CPLEX are 543.10, 658.70 and 826.09 for uncorrelated, weakly and strongly correlated type instances with large coefficients, respectively. We note that CPLEX consumes longer computation time for the problem instances with large coefficients than the ones with small coefficients. Indeed, average computation times of CPLEX are 334.15, 416.50 and 276.26 for uncorrelated, weakly and strongly correlated type instances with small coefficients, respectively. This signifies the difficulty of a binary knapsack problem instances with large coefficients against a binary knapsack problem instances with small coefficients.

	CPLEX		T(ADP-H)			PE(ADP-H)			PI(H)
n	avg time	avg no nodes	min	avg	max	min	avg	max	
1,000	13.39	8,620.00	0.13	0.20	0.28	0.0013	0.0243	0.0826	30.34
5,000	708.74	8,0571.60	0.50	1.41	3.55	0.0015	0.0081	0.0166	57.27
10,000	907.16	42,392.20	4.01	6.15	8.50	0.0016	0.0047	0.0101	52.63

Table 3.34: Comparison of ADP-H with CPLEX: Uncorrelated type problems with large coefficients, H: Greedy Heuristic.

	CPLEX		T(ADP-H)			PE(ADP-H)			PI(H)
n	avg time	avg no nodes	min	avg	max	min	avg	max	
1,000	12.50	7,804.40	0.12	0.40	0.58	0.0245	0.0566	0.0973	68.75
5,000	703.58	80,776.60	1.40	2.70	4.09	0.0031	0.0085	0.0192	51.10
10,000	1260.02	61,957.20	3.05	6.52	9.83	0.0018	0.0032	0.0040	59.07

Table 3.35: Comparison of ADP-H with CPLEX: Weakly-correlated type problems with large coefficients, H: Greedy Heuristic.

	CPLEX		T(ADP-H)			PE(ADP-H)			PI(H)
n	avg time	avg no nodes	min	avg	max	min	avg	max	
1,000	454.57	312,198.60	0.13	0.68	1.63	0.0035	0.0097	0.0263	92.87
5,000	825.14	92,107.60	1.14	4.80	14.20	0.0001	0.0022	0.0041	92.98
10,000	1198.55	73,053.00	2.91	13.03	32.62	0.0002	0.0009	0.0019	93.16

Table 3.36: Comparison of ADP-H with CPLEX: Strongly-correlated type problems with large coefficients, H: Greedy Heuristic.

Table 3.37 gives an important summary of the computational study. We observe that ADP-H continues to improve the performance of the greedy heuristic H substantially, meaning that ADP-H produces significantly better quality of solutions than those obtained by H. Tables 3.32 and 3.37 indicate that the heuristic H produced lower quality of solutions for problems with large coefficients against those with small coefficients. This observation is based on the following summary statistics. For small coefficient instances, average percentage improvements of ADP-H over H are 50.85, 37.78 and 81.51 for uncorrelated, weakly and strongly correlated type instances, respectively. On the other extreme, for large coefficient instances, average percentage improvements of ADP-H over H are 46.74, 59.64 and 93.00 for uncorrelated, weakly and strongly correlated type instances, respectively. Thus, on average, ADP-H achieves a larger improvement of H, for the instances with large coefficient than those with small coefficients. This implies, implicitly, the performance of H is worse for the large coefficient instances than the ones with small coefficients. Even though, H could generate worse quality of solutions for large coefficient instances than the small ones, ADP-H enhances the capability of H in a way that the resultant

average percentage deviations are comparably close for both the problems with large coefficients and the ones with small coefficients. Let us confirm this observation by the following computational evidence. We note that, percentage deviations of ADP-H are 0.0119, 0.0144 and 0.0193 for the small coefficient uncorrelated, weakly and strongly correlated type instances, respectively, while percentage deviations of ADP-H are 0.0371, 0.0228 and 0.0043 for the large coefficient uncorrelated, weakly and strongly correlated type instances, respectively. This computational evidence signifies that base-heuristic learning can have a larger marginal improvement of a given base-heuristic for those problems for which the heuristic generate worse solutions. As a result of this enhancement, ADP-H can exploit heuristics that are not competitive to other methodologies in order to come up with a competitive methodology.

	PI(H)		
n	UC	WC	SC
1,000	30.34	68.75	92.87
5,000	57.27	51.10	92.98
10,000	52.63	59.07	93.16
average	46.74	59.64	93.00

Table 3.37: Average percentage improvement of ADP-H over H: Uncorrelated, Weakly-correlated and Strongly-correlated type problems with large coefficients, H: Greedy Heuristic.

3.6 Conclusions

We develop and investigate an approximate dynamic programming methodology for the binary knapsack problem. We describe both statistical and base-heuristic learning. The statistical learning framework generated good quality of solutions but weaker than those obtained by the base-heuristic learning framework. Our overall computational evidence suggests that base-heuristic learning framework generates near-optimal solutions with short computation times and robust performances. Another interesting observation is that base-heuristic learning provides a framework that im-

proves over a given base-heuristic. This enhancement capability is important because base-heuristic learning framework can be used to other challenging optimization problems to improve over stand-alone heuristics, i.e., to attain better quality of solutions compared to those generated by the base-heuristic. In summary, our extensive computational study supports our belief that approximate dynamic programming can be considered as a practical alternative to existing methodologies.

Chapter 4

The Multi-Dimensional Knapsack Problem

This chapter addresses the multi-dimensional knapsack problem. In Section 4.1, we motivate the problem while providing a brief literature review. In Section 4.2, we reformulate the multi-dimensional knapsack problem as a dynamic program. In Section 4.3, we develop an approximate dynamic programming algorithm using statistical learning framework while we focus on base-heuristic learning framework in Section 4.4. We offer an extensive computational study in section 4.5. Section 4.6 concludes this chapter by providing a summary of insights obtained from the proposed approximate dynamic programming methodologies.

4.1 Background

The multi-dimensional knapsack problem (MKP) is a binary integer programming problem with nonnegative data which can be formulated as follows:

$$\begin{aligned} z_{MKP} = \quad & \text{maximize} \quad \sum_{j=1}^{n} c_j x_j \\ & \text{subject to} \quad \sum_{j=1}^{n} \mathbf{A}_j x_j \leq \mathbf{b}_0 \\ & \qquad\qquad x_j \in \{0,1\}, \end{aligned}$$

with the underlying assumption that $\mathbf{c}, \mathbf{A}, \mathbf{b}_0 \geq \mathbf{0}$ where z_{MKP} denotes the optimal value.

The multi-dimensional knapsack problem is known to be much more difficult than the binary knapsack problem. Complexity theory provides some insight to the increase in difficulty from changing a single constraint in the BKP to multiple constraints in the MKP. For the MKP no pseudo-polynomial algorithm can exist unless P=NP, since the MKP can be proved to be NP-hard in the strong sense (see Martello and Toth [54]). For the binary knapsack problems both polynomial time (Sahni [72]) and fully polynomial time approximation schemes (Ibarra and Kim [40]) have been developed. For the MKP, no fully polynomial-time approximation scheme can exist (Magazine and Chern [46]), unless P=NP, since this would imply the existence of a pseudo-polynomial algorithm for their optimal solution (which is impossible, these being NP-hard problems in the strong sense).

Many other well-known optimization problems such as packing and covering problems can be modeled as a multi-dimensional knapsack problems. Given a set $T = \{1, \cdots, t\}$, let $L = \{1, \cdots, l\}$ be an index of subsets $L_j \subset T$, $j \in L$. We associate a value c_j with a member of L_j of L. Let $x_j = 1$ (0) if L_j is (is not) selected, and let $a_{ij} = 1$ (0) if $i \in L_j$ ($i \notin L_j$) for $i = 1, \cdots, m$ and $j = 1, \cdots, n$.

In the *set packing problem* the objective is to find the maximum value collection of sets so that no two overlap (i.e., every $i \in T$ is contained in at most one of the selected members of L). We can formulate the set packing problem as follows:

$$\text{maximize } \{\mathbf{c}'\mathbf{x} | \mathbf{A} \cdot \mathbf{x} \leq \mathbf{e}, x_j \in \{0,1\}\},$$

where $\mathbf{e}$ is a column vector of ones. Note that this is a special case of 0-1 multi-dimensional knapsack problem since $a_{ij} \in \{0,1\}$, $\mathbf{b} = \mathbf{e}$.

In the *set covering problem* the objective is to find that collection of sets so that every element of the set T is in at least one of the selected members of L. Thus the integer programming formulation becomes:

$$\text{minimize } \{\mathbf{c}'\mathbf{x} | \mathbf{A} \cdot \mathbf{x} \geq \mathbf{e}, x_j \in \{0,1\}\}.$$

By complementing the variables, that is, letting $y_j = 1 - x_j$, we can convert a set covering problem into a multidimensional knapsack problem with right hand side $\mathbf{b} = \sum_j \mathbf{A}_j - \mathbf{e}$, that is,

$$\text{maximize } \{\mathbf{c}'\mathbf{y} | \mathbf{A} \cdot \mathbf{y} \leq \mathbf{b}, y_j \in \{0,1\}\}.$$

Multi-dimensional knapsack problems are typically encountered in resource allocation problems. As one of the early references Gilmore and Gomory [32] outline a dynamic programming algorithm. Weingartner [88] also analyzed its economic applications within a dynamic programming framework. Multi-dimensional knapsack problems have been used also for project selection (see Peterson [63]), cutting stock (see Gilmore and Gomory [32]), and loading problems (see Shih [78]). In addition to these specific applications, many complex problems can be transformed to multi-dimensional knapsack problems by some relaxation methodologies. Thus, it is an important class of optimization problem to be addressed.

Most of the algorithms in the literature scale the problem by the capacity vector (right hand side) in order to have a uniform capacity vector. Starting with the original multi-dimensional knapsack problem:

$$\begin{aligned} z_{MKP} = \quad & \text{maximize} \quad \sum_{j=1}^{n} c_j x_j \\ & \text{subject to} \quad \sum_{j=1}^{n} a_{ij} x_j \leq b_{0,i} \quad i = 1, \cdots, m \\ & \qquad\qquad x_j \in \{0,1\}, \qquad j = 1, \cdots, n, \end{aligned}$$

we transform the problem as follows:

TMKP:

$$\begin{aligned} z_{MKP} = \quad & \text{maximize} \quad \sum_{j=1}^{n} c_j x_j \\ & \text{subject to} \quad \sum_{j=1}^{n} h_{ij} x_j \leq 1 \quad i = 1, \cdots, m \\ & \qquad\qquad x_j \in \{0,1\}, \qquad j = 1, \cdots, n. \end{aligned}$$

where $h_{ij} = a_{ij}/b_{0,i}$, $i = 1, \cdots, m$, $j = 1, \cdots, n$ and $\mathbf{H}_j = (h_{1j}, \cdots, h_{mj})'$ the resource consumption vector for the variable j after the transformation. Thus, the right hand side becomes $\mathbf{e} = (1, \cdots, 1)'$. We next introduce some notation. Let $N = \{1, \cdots, n\}$; X_1= set of variables assigned to one; X_0= set of variables assigned to zero; $\mathbf{R}_u$= used resource vector, i.e., $\mathbf{R}_u = \sum_{j \in X_1} \mathbf{A}_j$, and $\mathbf{R}_a = \mathbf{e} - \mathbf{R}_u$ = available resource vector.

We classify the existing methodologies as *greedy*, *aggregate*, *enumerative* types while we refer to tabu search and genetic algorithms as an integrated framework that exploits one of the above methodologies. We describe briefly some of the proposed methodologies in the literature below.

Greedy type methods

Generally speaking, greedy type algorithms calculate score values $s_j,\ j = 1, \cdots, n$ and iteratively assigns $x_j = 1$ or $x_j = 0$ based on some order of the corresponding score values.

Senju and Toyoda [77] developed a *dual gradient method* (ST-DG) to find a sub-optimal solution to the MKP. The dual gradient heuristic starts from an infeasible solution ($x_j = 1,\ j = 1, \cdots, n$) and constructs a feasible solution by setting variables to zero with the lowest gradient. The *gradient* G_j of the variable j is $\frac{c_j}{\mathbf{H}_j'\mathbf{S}}$ where $\mathbf{S} = (S_1, \cdots, S_m)$ and $S_i = \sum_{j \in X_1} h_{ij} - 1$ (0) if $\sum_{j \in X_1} h_{ij} \geq 1$ (otherwise). In their original work, Senju and Toyoda provided a limited number of computational experiments with promising results.

Toyoda [85] developed a *primal gradient method* (T-PG) which constructs a feasible solution from the all-zero solution ($x_j = 0,\ j = 1, \cdots, n$). This heuristic constructs a feasible solution by setting variables to one with the largest gradient. The *gradient* G_j of the variable j is $\frac{c_j}{\mathbf{H}_j'\mathbf{R}_u}$ where $\mathbf{R}_u = \sum_{j \in X_1} \mathbf{H}_j$. Toyoda [85] compared T-PG with ST-DG on the problem instances presented in Senju and Toyoda [77]. This very limited comparison indicates that T-PG gives better solutions than ST-DG.

Loulou and Michaelides [45] developed a *greedy-like heuristic* (LM-GL) by incorporating the basic ideas of Toyoda's primal gradient T-PG and the greedy method applied to the binary knapsack problem. Starting from the all-zero solution, LM-GL constructs a feasible solution by setting variables to one with the largest gradient. The *gradient* G_j of the variable j is $\frac{c_j}{P_j}$ where $P_j = \max\limits_{i=1,\cdots,m} \left\{ \frac{(R_{a,i}+h_{ij})(\sum_{k \in X_c} h_{ik} - h_{ij})}{(1-R_{a,i}-h_{ij})} \right\}$ and $\mathbf{R}_a = \mathbf{e} - \mathbf{R}_u$. Loulou and Michaelides proposed different penalty schemes P_j for a variable j to capture (a) the total consumption of resource if an item is added to current solution, (b) the amount of resource remaining if an item is added to the current solution and (c) the future potential demand for a resource if an item is added

to current solution. Their computational study indicates that, on the average, LM-GL generates better solutions than T-PG. However, the computational requirements for LM-GL considerably exceed T-PG due to the elaborate penalty calculation so it might not be practical for large-scale problems.

Kochenberger et. al. [29] proposed another *primal gradient method* (K-PG) to obtain good feasible solutions to general integer programming problems. The basic idea of this paper was originally stimulated by the method ST-DG by Senju and Toyoda [77]. K-PG begins with the all-zero solution and increments variables with the largest gradient iteratively. The *gradient* G_j of the variable j is $\frac{c_j}{\sum_{i=1}^{m} \overline{a}_{ij}}$ where $\overline{a}_{ij} = \frac{a_{ij}}{\overline{b}_i}$, (0) if $\overline{b}_i > 0$, (otherwise) and $\overline{\mathbf{b}} = \mathbf{b} - \sum_{j \in X_1} \mathbf{A}_j$. Zanakis [92] compared ST-DG with K-PG and with another heuristic algorithm by Hillier [37] (H-H). H-H is the slowest one, yet it produces better solutions, on average, than both ST-DG and K-PG. K-PG consumes the least computation time for tightly constrained MKP and yields the best solution quality for loosely constrained MKP. On average, ST-DG consumes less computation time than the other two heuristics, but it seems to generate lower quality solutions.

Magazine and Oguz [47] combine the ideas of Senju and Toyoda [77] ST-DG with Everett's Generalized Lagrange Multipliers approach [24] to develop a suboptimal method (MKNAP) to the multi-dimensional knapsack problem. It starts from an infeasible solution (e.g., $x_j = 1,\ j = \{1, \cdots, n\}$) and sets the variable with the smallest gradient to zero. The gradient G_j of the variable j is $\frac{c_j}{\mathbf{u}' \mathbf{A}_j}$ where $\mathbf{u} \in \Re_+^m$ is the multiplier vector calculated by their proposed technique. A computational study is done to compare MKNAP with ST-DG by Senju and Toyoda [77] and K-PG by Kochenberger et. al. [29]. K-PG performs remarkably better than ST-DG and MKNAP in terms of solution quality at a cost of larger computational time. ST-DG and MKNAP perform about the same, whereas MKNAP requires slightly more computational time than ST-DG. Although MKNAP consumes slightly more computational time, it outperforms ST-DG for larger problems (e.g., problems of size $m = 60, n = 500$ and

$m = 20, n = 1,000$). K-PG is not competitive in this category as it often performs about 50 times slower.

Lee and Guignard [43] developed a two phase *parametric approximate algorithm* (LG-P). The algorithm finds a feasible solution, fixes variables and complements certain sets of variables. In phase I, LG-P attempts to find a good feasible solution by applying a modified faster version of Toyoda's method (T-PG) iteratively. In phase II, LG-P tries to improve the current solution by a complementing scheme introduced by Balas and Martin's pivot-and-complement heuristic BM-PC (see Balas and Martin [3]). LG-P is compared with MKNAP, T-PG and BM-PC. According to their computational study LG-P performed better in terms of both solution quality and computation time than MKNAP and T-PG. The results of the comparison with BM-PC indicates that BM-PC provided better solution at the expense of longer computation times.

Aggregation type methods

Freville and Plateau [26, 27] describes a pool of solution methodologies called AGNES. The AGNES method is based on the idea of surrogate relaxation (a heuristic approach proposed by Glover [33] to integer programming problems). AGNES provides problem reduction techniques that fixes variables and removes redundant constraints in advance. AGNES is compared with other surrogate relaxation type and greedy type heuristics while being competitive in terms of both solution quality and time.

Gavish and Pirkul [31] demonstrated that better bounds can be generated by surrogate and composite relaxation. In order to utilize these better bounds, Gavish and Pirkul [31] proposed a branch-and-bound framework. Due to excessive computation times for large-scale problems, they devised schemes to terminate the branch-and-bound algorithm early without proving optimality. Their findings assert that surrogate relaxation degrades its effectiveness for the problems with large number of

constraints. In connection to the previous work, Pirkul [65] uses surrogate multipliers to develop a greedy-type heuristics with some enumeration (P-SU). P-SU is compared with LM-GL and BM-PC. P-SU provides better solutions than LM-GL at the expense of longer computation time. The comparison with BM-PC does not have an absolute winner in terms of both solution time and solution quality. Computational experience in the literature suggests that surrogate relaxation type algorithms work well for small m.

Enumerative type methods

Exact methodologies based on enumerative methods reported limited success (see Thesen [83], Shih [78], and Weingartner and Ness [89]).

Tabu search

Hanafi and Freville [36] describes an efficient tabu search approach for multi-dimensional knapsack problem while providing a substantial literature of this methodology (e.g. see also Glover and Kochenberger [34], Aboudi and Jornsten [1], and Lokketangen and Glover [44]).

Genetic algorithms

Chu and Beasley [15] proposed a recent genetic algorithm (CB-GA) for the multi-dimensional knapsack problem while providing a good survey on meta-heuristics. CB-GA is capable of obtaining high-quality solutions for problems with various characteristics, while requiring only a modest amount of computational effort. CB-GA is also compared to some of the well-known heuristics from the literature while Chu and Beasley focus on a comparison only based on quality of solutions generated by different approaches. To our knowledge, CB-GA tries to solve the largest problem instances that have been ever attempted. We also compare the performance of approximate dynamic programming approach with their test bed.

4.2 Dynamic Programming

Following the general guidelines developed in Chapter 2, we consider the following subproblem $MKP(k, \mathbf{b})$ with the first k variables and a right-hand-side $\mathbf{b}$:

$$\begin{aligned} F_k(\mathbf{b}) = \quad & \text{maximize} \quad \sum_{j=1}^{k} c_j x_j \\ & \text{subject to} \quad \sum_{j=1}^{k} \mathbf{A}_j x_j \leq \mathbf{b} \\ & \qquad\qquad x_j \in \{0,1\}. \end{aligned}$$

The dynamic programming recursion is:

$$F_k(\mathbf{b}) = \max \{F_{k-1}(\mathbf{b}), F_{k-1}(\mathbf{b} - \mathbf{A}_k) + c_k\}, \ k = 2, \cdots, n,$$

with the initial condition,

$$\begin{aligned} F_1(\mathbf{b}) = \quad & \max \quad c_1 x_1 \\ & \qquad \mathbf{A}_1 x_1 \leq \mathbf{b} \\ & \qquad x_1 \in \{0,1\}. \end{aligned}$$

We can construct an optimal solution to the $MKP(n, \mathbf{b}_0)$ by dynamic programming backward phase as described in Table 2.1. We set $F_k(\mathbf{b}) = -\infty$ if the subproblem $MKP(k, \mathbf{b})$ is infeasible. It is straightforward to detect whether the subproblem $MKP(k, \mathbf{b})$ is *feasible* or not. If there exists an i such that $b_i < 0$ it is obvious to see that the problem is infeasible due to nonnegative data assumption, i.e., $a_{ij} \geq 0$. The worst case time complexity of the underlying dynamic programming for the problem $MKP(n, \mathbf{b}_0)$ is $O(n(b^*)^m)$ where $b^* = \max\{b_{0,1}, \cdots, b_{0,m}\}$. Space requirements and value function computations become impractical for large $\mathbf{b}_0$, hence for large b^*. We provide an approximate dynamic programming algorithm based on both statistical and base-heuristic learning framework in the next sections to overcome the curse of dimensionality.

4.3 Statistical Learning

This section customizes the statistical learning framework introduced earlier in Chapter 2 for the multi-dimensional knapsack problem.

4.3.1 Notations and Definitions

Let $\mathbf{b}_0$ be the right-hand side (capacity) of the problem $MKP(n, \mathbf{b}_0)$. Let $\Omega = \{\mathbf{b}|\mathbf{0} \leq \mathbf{b} \leq \mathbf{b}_0\}$ be the state space. Let $S = \{\mathbf{b}^i \in \Omega | i = 1, \cdots, s\}$ denote a sample of the state space Ω. Define $\Omega_k = \{(k, \mathbf{b})|\mathbf{b} \in \Omega\}$ and $S_k = \{(k, \mathbf{b})|\mathbf{b} \in S\}$ for k varying in $[1, n]$. Let $K : \Re \rightarrow \Re$ be an one-dimensional kernel function (see Chapter 3 for examples of kernel functions). We denote by $K_m : \Re^m \rightarrow \Re$ a multi-dimensional kernel function. A typical example is $K_m(\mathbf{t}) = e^{-\mathbf{t}'\mathbf{t}}$ where $\mathbf{t} = (t_1, \cdots, t_m)$.

4.3.2 Parametric Approximation

We motivate the parametric approximation method using research results from the probabilistic analysis of the multi-dimensional knapsack problem (see Frieze and Clarke [28], Meanti et. al. [55], Dyer and Frieze [23], Szkatula [80], Piersma [64]). Let $\mathbf{A}_j = (a_{1,j}, \cdots, a_{m,j})'$ be nonnegative i.i.d. random vectors and suppose that c_j are i.i.d. random variables, with $j \in \{1, \cdots, n\}$, that are independent of the $\mathbf{A}_j$. Let $b_i = n\tau_i$ for some fixed m-vector $\boldsymbol{\tau} = (\tau_1, \cdots, \tau_m)$. The solution value of the MKP, defined by the random variables $\mathbf{A}_j$ and c_j and by the constants $b_i \geq 0$ is denoted by Z_n^{MKP}. Let us denote by Z_n^{LP} the solution value of the LP relaxation of the MKP. Piersma [64] and Meanti et. al. [55] study the limiting behavior of the random variable Z_n^{LP}/n as $n \rightarrow \infty$ and m remains fixed. It is demonstrated that there exists a constant ρ depending on the parameters b_i and on the expected values $Ea_{i,1}$ and Ec_1 such that, w.p. 1, Z_n^{LP}/n converges to ρ. Since the solution value of the

MKP (Z_n^{MKP}) is close to Z_n^{LP} (see Meanti et.al. [55]), the random variable Z_n^{MKP}/n can be shown to converge, w.p. 1, to the same constant ρ where $\rho = \min_{\boldsymbol{\lambda} \geq 0} Ef_{\boldsymbol{\lambda}}(c_1, \mathbf{A}_1)$ and $f_{\boldsymbol{\lambda}}(t_0, \mathbf{t}) = \sum_{i=1}^{m} \lambda_i \tau_i + \max(0, t_0 - \sum_{i=1}^{m} \lambda_i t_i)$ for a fixed $\boldsymbol{\lambda} \geq \mathbf{0}$. Thus, the previous results suggest that Z_n^{MKP} can be represented asymptotically as a linear function of the capacity vector $\mathbf{b}$ with some additional terms. This fact motivates us to approximate the optimal value function by a linear function as will be described below. We emphasize that our problem generation schemes, particularly weakly and strongly correlated type problems, do not satisfy the assumption of the probabilistic analysis which holds for uncorrelated type problems with a certain right-hand-side generation scheme (i.e., $b_i = n\tau_i$ for some fixed m-vector $\boldsymbol{\tau}$) and with very large number of variables for a fixed m.

We define by $H_k(\mathbf{b})$ an approximate value to the optimal value $F_k(\mathbf{b})$. We approximate $F_k : \Omega_k \rightarrow \Re$ by a linear function $g : \Omega_k \rightarrow \Re$ of the form $g(\boldsymbol{\theta}_k, \mathbf{b}) = \boldsymbol{\alpha}_k' \mathbf{b} + \beta_k$, meaning that for a state $(k, \mathbf{b}) \in \Omega_k$ we have $H_k(\mathbf{b}) = g(\boldsymbol{\theta}_k, \mathbf{b})$. To tune the parameters we minimize the following unweighted training criterion

$$C = \sum_{i=1}^{s} L(g(\boldsymbol{\theta}_k, \mathbf{b}^i), F_k(\mathbf{b}^i)),$$

where the $F_k(\mathbf{b}^i)$ are the sample output values corresponding $\mathbf{b}^i \in S$, $\boldsymbol{\theta}_k$ is the parameter vector for the parametric model $H_k(\mathbf{b}) = g(\boldsymbol{\theta}_k, \mathbf{b})$ and $L(H_k(\mathbf{b}^i), F_k(\mathbf{b}^i))$ is a general loss function for predicting $H_k(\mathbf{b}^i)$ when the training data is $F_k(\mathbf{b}^i)$. We use the least absolute deviation criterion for the loss function, i.e, $L(H_k(\mathbf{b}^i), F_k(\mathbf{b}^i)) = |H_k(\mathbf{b}^i) - F_k(\mathbf{b}^i)|$ leading to the training criterion:

$$C = \sum_{i=1}^{s} |\boldsymbol{\alpha}_k' \mathbf{b}^i + \beta_k - F_k(\mathbf{b}^i)|.$$

We calculate model parameters $\boldsymbol{\alpha}_k^*, \beta_k^*$ by solving the following optimization prob-

lem

$$\underset{\alpha_k,\beta_k}{\text{minimize}} \quad \sum_{i=1}^{s} |\alpha_k' \mathbf{b}^i + \beta_k - F_k(\mathbf{b}^i)|.$$

We reformulate the above optimization problem as a linear program (see Bertsimas and Tsitsiklis [12]). In our computations, we use the commercial package CPLEX to solve the corresponding linear program in order to compute the parameters α_k^* and β_k^*.

4.3.3 Nonparametric Approximation

Suppose we have calculated the value functions at points $\mathbf{b}^i \in S$, i.e., $F_k(\mathbf{b}^i)$, $i = 1, \cdots, s$ have been calculated. Under the nonparametric approximation $F_k(\mathbf{b})$ is approximated by $H_k(\mathbf{b})$ as follows:

$$H_k(\mathbf{b}) = \frac{\sum_{i=1}^{s} w(\mathbf{b}, \mathbf{b}^i) F_k(\mathbf{b^i})}{\sum_{i=1}^{s} w(\mathbf{b}, \mathbf{b}^i)},$$

where $w(\mathbf{b}, \mathbf{b}^i)$ is a local weight assigned to the sample point $F_k(\mathbf{b}^i)$. In the multi-dimensional setting, we use

$$w(\mathbf{b}, \mathbf{b}^i) = K_m \left(\tfrac{|b_1 - b_1^1|}{h_1}, \cdots, \tfrac{|b_m - b_m^s|}{h_m} \right)$$

where K_m is a multi-dimensional kernel function and $\mathbf{h} = (h_1, \cdots, h_m)$ is the bandwidth vector that will be described in section 4.3.4 on page 96.

4.3.4 Algorithm Design

The statistical learning framework for the multi-dimensional knapsack problem offers several algorithmic design choices based on the approximation scheme used. We describe the *sampling schemes* for both parametric and nonparametric approximation below. For the nonparametric approximation we provide details of the *bandwidth parameter* and *kernel function* selection.

Sampling Type:

Sampling is a central issue for both the parametric and nonparametric approximation. In our computational study we chose $S = \{\mathbf{b}^1, \cdots, \mathbf{b}^s\}$ with

$$\mathbf{b}^i = (i\lfloor b_{0,1}/s \rfloor, \cdots, i\lfloor b_{0,m}/s \rfloor),$$

and $\mathbf{b}^s = \mathbf{b}_0$ where S is a sample of the state space Ω. We call the sample S as the *uniform sample.* We extend the uniform sample by taking additional sample points $\mathbf{b} = (b_1, \cdots, b_m)$ such that $\mathbf{b}^k \leq \mathbf{b} \leq \mathbf{b}^{k+1}$, where $b_i \sim U(b_i^k, b_i^{k+1})$ (i.e. uniformly distributed between b_i^k and b_i^{k+1}) and $\mathbf{b}^k, \mathbf{b}^{k+1} \in S$. We name this extended sample as the *stratified sample.* In our computational study we found that the stratified sample does not necessarily generate better solutions than the uniform sample. An alternative to selecting the set S is to choose the set S randomly over the state space Ω. We have found, however, that this choice is inferior to selecting the set S as outlined above. Thus, we fix *uniform sampling* as our sampling scheme. Furthermore, we use the same sample over the stages, meaning that $S_k = \{(k, \mathbf{b}) | \mathbf{b} \in S\}$ for varying k in $[1, n]$.

Bandwidth selection:

We introduce the notation $N_{\mathbf{h}}(\mathbf{b}) = \{\mathbf{b}^i \in S | \mathbf{b} - \mathbf{h} \leq \mathbf{b}^i \leq \mathbf{b} + \mathbf{h}\}$ as the neighborhood set for a query point $\mathbf{b}$ with the bandwidth $\mathbf{h}$. Thus, through $\mathbf{h}$ we determine the neighbor sample points for a query point $\mathbf{b}$. In the computational study for the binary knapsack problem we observed that bandwidth is a very important parameter. We automate the selection of $\mathbf{h}$ by the following relationship $\mathbf{h} = n_p \boldsymbol{\delta}_q + \boldsymbol{\delta}_r$ where $\boldsymbol{\delta}_q = (\lfloor b_{0,1}/s \rfloor, \cdots, \lfloor b_{0,m}/s \rfloor)$ and $\boldsymbol{\delta}_r = \mathbf{b}_0 - s\boldsymbol{\delta}_q$ through specifying the *size* parameter $n_p \in \{1, \cdots, s-1\}$. Based on our previous computational study on the binary knapsack problem, we set $n_p = 1$.

Let us illustrate the sampling and bandwidth selection by an example. Consider

a multi-dimensional knapsack problem $MKP(\cdot, \mathbf{b}_0)$ with the capacity $\mathbf{b}_0 = (50, 100)$. Let the sample size $s = 5$. Then $\boldsymbol{\delta}_q = (\lfloor b_{0,1}/s \rfloor, \lfloor b_{0,m}/s \rfloor) = (10, 20)$ and $\boldsymbol{\delta}_r = \mathbf{0}$. Thus, the uniform sample becomes $S = \{(0,0), (10,20), (20,40), (30,60), (40,80), (50,100)\}$. If we set *size* parameter $n_p = 1$ then $\mathbf{h} = (10, 20)$. As an example, for the query point $\mathbf{b} = (15, 25)$ the neighborhood set is $N_{\mathbf{h}}(\mathbf{b}) = \{(10,20), (20,40)\}$. If we set $n_p = 2$ then $\mathbf{h} = (20, 40)$. For the same query point $\mathbf{b} = (15, 25)$, the neighborhood set becomes $N_{\mathbf{h}}(\mathbf{b}) = \{(0,0)(10,20), (20,40), (30,60)\}$. Thus, for a certain query point $\mathbf{b}$, $N_{\mathbf{h}}(\mathbf{b})$ includes more sample points as the bandwidth parameter $\mathbf{h}$ increases.

Kernel function selection:

We need a multi-dimensional kernel function $K_m(\cdot) : \Re^m \rightarrow \Re$ to measure the closeness of two multi-dimensional vectors, e.g., a query point $\mathbf{b}$ and a sample point $\mathbf{b}^i \in S$. Instead, we use the transformation $K_m(t_1, \cdots, t_m) = \sum_{i=1}^{m} K(t_i)$ for a given one-dimensional kernel function $K(\cdot) : \Re \rightarrow \Re$. Our computational evidence for the binary knapsack problem suggests that the triangular kernel is the most promising kernel function within the nonparametric approximation. Based on this observation, we use the triangular kernel function $K(t) = (1 - |t|)I(|t| \leq 1)$ where $I(\cdot)$ is an indicator function, i.e., $I(|t| \leq 1) = 1$ (0) if $|t| \leq 1$ (otherwise).

Once statistical learning generates a feasible solution $\mathbf{x}^{adp}$ with X_0= set of variables assigned to zero in the ADP solution, X_1= set of variables assigned to one in the ADP solution, we calculate the slack vector $\mathbf{s} = \mathbf{b}_0 - \sum_{j \in X_1} \mathbf{A}_j$. We iteratively set a variable $x_k, k \in X_0$ to 1 if $\mathbf{A}_k \leq \mathbf{s}$ while updating $\mathbf{s}$ accordingly (i.e., $\mathbf{s} = \mathbf{s} - \mathbf{A}_k$ once x_k is set to 1). This final step can be considered as a *local* improvement to the solution obtained by the statistical learning methodology.

4.4 Base-Heuristic Learning

This section introduces a *base-heuristic learning algorithm* for the multi-dimensional knapsack problem. We describe some of the existing methodologies fully while introducing a new heuristic, which we name *adaptive fixing*, as a base-heuristic. These heuristics will be used to approximate the value function as described in Chapter 2.

4.4.1 A Base-Heuristic Learning Algorithm

We adapt and enhance the generic base-heuristic learning framework described in Table 2.2 (see Section 2.2 on page 28) to the multi-dimensional knapsack problem. Let $BH(k, \mathbf{b})$ be a base-heuristic for the multi-dimensional knapsack subproblem $MKP(k, \mathbf{b})$ with k variables and a right-hand-side $\mathbf{b}$. Let $x_{BH}(k, \mathbf{b})$ be the corresponding heuristic solution and $H_k(\mathbf{b})$ be the heuristic value, i.e., an estimate of the optimal value $F_k(\mathbf{b})$. We denote by $LP(k, \mathbf{b})$ the linear programming relaxation of the problem $MKP(k, \mathbf{b})$. Let $z^{LP(k,\mathbf{b})}$ be the corresponding optimal LP value. Let $U_k(\mathbf{b})$ be an upper bound to the optimal value $F_k(\mathbf{b})$ (e.g. $U_k(\mathbf{b}) = \lfloor z^{LP(k,\mathbf{b})} \rfloor$). We denote by $p^f(k, \mathbf{b})$ whether the subproblem $MKP(k, \mathbf{b})$ is *feasible* or not. We set $p^f(k, \mathbf{b}) = 1$ (0) if the problem is feasible (or not). Let us denote by $s^o(k, \mathbf{b})$ whether $x_{BH}(k, \mathbf{b})$ is *optimal* solution or not to the subproblem $MKP(k, \mathbf{b})$ (e.g. $x_{BH}(k, \mathbf{b})$ is optimal if $U_k(\mathbf{b}) \leq H_k(\mathbf{b})$). We set $s^o(k, \mathbf{b}) = 1$ (0) if the heuristic solution is optimal (not optimal).

The *base-heuristic learning algorithm* consists of two phases. In the first phase, we apply $BH(n, \mathbf{b}_0)$ to the problem $MKP(n, \mathbf{b}_0)$ and get $x_{BH}(n, \mathbf{b}_0), p^f(n, \mathbf{b}_0), s^o(n, \mathbf{b}_0)$. If the solution $x_{BH}(n, \mathbf{b}_0)$ is optimal, i.e., $s^o(n, \mathbf{b}_0) = 1$, the algorithm terminates with an optimal solution. If the problem is infeasible, i.e., $p^f(n, \mathbf{b}_0) = 0$, the algorithm terminates without solution. Under the assumption that the problem is feasible we set *best-solution* known $\mathbf{x}^{best} = x_{BH}(n, \mathbf{b}_0)$ and *best value* known $z^{best} = H_n(\mathbf{b}_0)$. We proceed by applying *reduced cost fixing* as described below to fix some of the variables to

the corresponding values (0 or 1) in an *optimal* solution to the problem $MKP(n, \mathbf{b}_0)$. Reduced cost fixing might effectively reduce the number of variables in advance. To illustrate this, we assume for example that we apply reduced cost fixing to the problem $MKP(n, \mathbf{b}_0)$ and obtain $F = \{k, k+1, \cdots, n\}$ the set of fixed variables, i.e., we know the values (0 or 1) of $x_{k+1}, \cdots, x_n$ in an optimal solution. This implies that solving $MKP(n, \mathbf{b}_0)$ is equivalent to solving $MKP(k-1, \mathbf{b}_0 - \sum_{j \in F} x_j)$.

Reduced-cost fixing:

At optimality of the linear programming relaxation (LP) of the problem MKP $(n, \mathbf{b}_0)$, the objective function can be written as

$$z^{LP} = z_0 + \sum_{j \in L} \overline{c}_j x_j + \sum_{j \in U} \overline{c}_j x_j,$$

where z^{LP} is the LP solution value, L denotes the set of non-basic variables at their lower bound, U denotes the set of non-basic variables at their upper bounds and $\overline{c}_j$ denotes the reduced cost of the variable $j \in \{1, \cdots, n\}$. Perturbing a non-basic variable k by δ changes the objective by $\delta \overline{c}_k$. Let us denote by $z^{current}$ a lower bound or a known solution value to the problem $MKP(n, \mathbf{b}_0)$ (e.g., heuristic value $H_k(\mathbf{b})$). In reduced-cost fixing we consider the following cases (i) If x_k is a non-basic variable at its lower bound, and $z^{LP} + \overline{c}_k \leq z^{current}$, then we should fix variable x_k at its lower bound in any optimal solution, i.e., $x_k = 0$. (ii) If x_k is a non-basic variable at its upper bound, and $z^{LP} - \overline{c}_k \leq z^{current}$, then we should fix variable x_k at its upper bound in any optimal solution, i.e., $x_k = 1$. Thus, for a non-basic variable k if $|\overline{c}_k| > z^{LP} - z^{current}$ then we fix the variable x_k at its LP value, i.e. $x_k = x_k^{LP}$ (see Proposition 2.1 on page 389 in Nemhauser and Wolsey [61]). Once reduced-cost fixing is applied, we update the fixed variable set F accordingly. We use $\mathbf{x}^f = (x_1^f, \cdots, x_n^f)$ to describe the status of the variables, i.e, fixed or not. We call a variable j is *fixed* if $x_j^f = 0$ or 1.

In the second phase, we construct the ADP solution $\mathbf{x}^{adp}$ and update the best

solution $\mathbf{x}^{best}$ and its value z^{best} accordingly. We iteratively assign x_j^{adp}, $j \notin F$ according to the *policy assignment* scheme as described below.

Policy assignment:

We can set x_k^{adp}, $k \notin F$ to 0 or 1 by,

$$x_k^{adp} = \underset{x \in \{0,1\}}{\text{argmax}}\{H_{k-1}(\mathbf{b} - \mathbf{A}_k x) + c_k x\},$$

in a straightforward manner which will be called the *standard policy scheme.* In order to enhance the optimal value approximation, we exploit some performance statistics of base-heuristics $BH(k-1, \mathbf{b} - \mathbf{A}_k x)$ to the subproblems $MKP(k-1, \mathbf{b} - \mathbf{A}_k x)$ to recalculate the estimates $H_{k-1}(\mathbf{b} - \mathbf{A}_k x)$ for $x = 0, 1$. A typical performance statistic for the base-heuristic $BH(k, \mathbf{b})$ would be the percentage deviation of $H_k(\mathbf{b})$ from the upper bound $U_k(\mathbf{b})$ of the subproblem $MKP(k, \mathbf{b})$. In our computations, we use $U_k(\mathbf{b}) = \lfloor z^{LP(k,\mathbf{b})} \rfloor$. The expectation that the base heuristic should have a similar percentage deviation from the upper bound motivates us to recalculate the estimate of the optimal values $F_{k-1}(\mathbf{b} - \mathbf{A}_k x)$. Let $\epsilon_x = \frac{U_{k-1}(\mathbf{b}-\mathbf{A}_k x) - H_{k-1}(\mathbf{b}-\mathbf{A}_k x)}{U_{k-1}(\mathbf{b}-\mathbf{A}_k x)}$ denote the relative error of $H_{k-1}(\mathbf{b} - \mathbf{A}_k x)$ with respect to $U_{k-1}(\mathbf{b} - \mathbf{A}_k x)$ for $x = 0, 1$. Our expectation is that base-heuristics $BH(k-1, \mathbf{b} - \mathbf{A}_k x)$ have the same level of relative error meaning that we calculate the minimum relative error $\epsilon^* = \min_{x \in \{0,1\}}\{\epsilon_x\}$ and recalculate the estimates of the optimal values as $H_{k-1}^u(\mathbf{b} - \mathbf{A}_k x) = (1 - \epsilon^*) U_{k-1}(\mathbf{b} - \mathbf{A}_k x)$ for $x = 0, 1$. Then we set x_k^{adp}, $k \notin F$ to 0 or 1 by,

$$x_k^{adp} = \underset{x \in \{0,1\}}{\text{argmax}}\{H_{k-1}^u(\mathbf{b} - \mathbf{A}_k x) + c_k x\}.$$

We call this policy scheme as the *inferential policy scheme.* In our computations for the multi-dimensional knapsack problem, we use the inferential policy scheme to calculate policies.

Update best known solution and value:

Once we set x_k^{adp}, $k \notin F$ to 0 or 1, we update the best solution $\mathbf{x}^{best}$ and value z^{best}. Let $x^* = x_k^{adp}$. Due to the nature of policy construction in the backward phase, a solution $x^{current} = (\mathbf{x}_{BH}(\mathbf{b} - \mathbf{A}_k x^*), x^*, x_{k+1}^{adp}, \cdots, x_n^{adp})$ with a value $z^{current} = H_{k-1}(\mathbf{b} - \mathbf{A}_k x^*) + c_k x^* + \sum_{j=k+1}^{n} c_j x_j^{adp}$ is available. We call $x^{current}$ the *current solution* and $z^{current}$ the *current solution value.* If $z^{current}$ is larger than z^{best} we update z^{best}, $\mathbf{x}^{best}$ as described in Table 4.1 below.

Update Best Solution
1: Input: $x_k^{adp}, k \notin F$ (as an output of policy assignment) $x_{BH}(k, \mathbf{b} - \mathbf{A}_k x), H_k(\mathbf{b} - \mathbf{A}_k x), x = 0, 1,\ z^{best},\ \mathbf{x}^{adp}$ $\mathbf{x}^{adp}, \mathbf{x}^{best}, z^{best}$
2: $x^* \leftarrow x_k^{adp}$
3: $z^{current} = H_{k-1}(\mathbf{b} - \mathbf{A}_k x^*) + c_k x^* + \sum_{j=k+1}^{n} c_j x_j^{adp}$

4: If $(z^{current} > z^{best})$ then
begin
$z^{best} \leftarrow z^{current}$
$x_j^{best} = x_{BH}(k, b - A_k x^*)_j,\ j = 1, \cdots, k-1$
$x_j^{best} = x^*,\ j = k$
$x_j^{best} = x_j^{adp},\ j = k+1, \cdots, n$
end
else
do not update the best solution

5: Ouput: z^{best}, x^{best}.

Table 4.1: Description of Update Best Solution.

We have several enhancements to speed up the approximate dynamic program-

ming algorithm: an early termination of algorithm through an *ADP stopping condition* and variable fixing through a *lag policy assignment* scheme, both of which are described next.

ADP stopping condition:

Approximate dynamic programming can be terminated once we find an optimal solution to $MKP(k, \mathbf{b})$ in the backward phase. While setting x_k^{adp}, $k \notin F$ in the policy assignment we consider the subproblems $MKP(k-1, \mathbf{b}-A_k x)$, $x = 0, 1$ and we apply $BH(k-1, \mathbf{b}-\mathbf{A}_k x)$, $x = 0, 1$ and get $p^f(k-1, \mathbf{b}-\mathbf{A}_k x), s^o(k-1, \mathbf{b}-\mathbf{A}_k x)$, $x = 0, 1$. We stop ADP if $s^o(k-1, \mathbf{b}-\mathbf{A}_k x) = 1$ (i.e., $\mathbf{x}_k(k-1, \mathbf{b}-\mathbf{A}_k x)$ is an optimal solution to $MKP(k-1, \mathbf{b}-A_k x)$) for both $x = 0$ and $x = 1$, meaning that we find an optimal solution to $MKP(k, \mathbf{b})$. For a certain $x \in \{0, 1\}$, we say $MKP(k, \mathbf{b}-\mathbf{A}_k x)$ dominates $MKP(k, \mathbf{b}-\mathbf{A}_k(1-x))$ if $H_k(\mathbf{b}-\mathbf{A}_k x) > U_k(\mathbf{b}-\mathbf{A}_k(1-x))+c_k(1-x)$. We set the parameter $d(k-1, \mathbf{b}-\mathbf{A}_k x) = 1$ if $MKP(k, \mathbf{b}-\mathbf{A}_k x)$ dominates $MKP(k, \mathbf{b}-\mathbf{A}_k(1-x))$. Thus, ADP can be terminated if $d(k-1, \mathbf{b}-\mathbf{A}_k x) = 1$ and $s^o(k-1, \mathbf{b}-\mathbf{A}_k x) = 1$ for $x = 0$, or 1. The parameter $stop - adp$ described in Table 4.2 indicates whether the base-heuristic learning algorithm can be terminated or not, i.e., $stop - adp = 1$ (0) means *early termination* is satisfied (not satisfied). We describe the algorithmic version of ADP stopping criterion in Table 4.2.

Lag policy assignment scheme:

As introduced earlier, we use $\mathbf{x}^f = (x_1^f, \cdots, x_n^f)$ to describe the status of the variables, i.e., fixed or not. We call a variable j *fixed* if $x_j^f = 0$ or 1. Let F denote the set of indices of the fixed-variables. For a certain l value, lag policy assignment fixes variables $x_{k-l}, \cdots, x_{k-1}$ (assuming they have not been fixed so far) once x_k^{adp} is assigned to a value 0 or 1 as described in the policy assignment. Under the *lag policy assignment scheme* variable fixing is done through updating the status vector $\mathbf{x}^f$ by setting $x_j^f = x_{BH}(k-1, \mathbf{b}-\mathbf{A}_k x^*)_j$ for $j \in [k-l, k)$ and $j \notin F$ where $x^* = x_k^{adp}$ and l is the `lag-size` parameter. l is determined by $l = \lfloor \frac{k}{\texttt{lag-time}} \rfloor$, where `lag-time` is a user-specified parameter. Lag-size parameter l specifies the number of

ADP Stopping Condition
$stop - adp = 0$
Case 1: if $p^f(k-1, \mathbf{b} - \mathbf{A}_k x) = 1,\ x = 0, 1$ then
begin
-$s^o(k-1, \mathbf{b}) = 1$ and $s^o(k-1, \mathbf{b} - \mathbf{A}_k) = 1 \rightarrow stop - adp = 1$
-$s^o(k-1, \mathbf{b}) = 1$ and $d(k-1, \mathbf{b}) = 1 \rightarrow stop - adp = 1$
-$s^o(k-1, \mathbf{b} - \mathbf{A}_k) = 1$ and $d(k-1, \mathbf{b} - \mathbf{A}_k) = 1 \rightarrow stop - adp = 1$
end
Case 2: if $p^f(k-1, \mathbf{b}) = 1$ then
begin
-$s^o(k-1, \mathbf{b}) = 1$ and $d(k-1, \mathbf{b}) = 1 \rightarrow stop - adp = 1$
end
Case 3: if $p^f(k-1, \mathbf{b} - \mathbf{A}_k) = 1$ then
begin
-$s^o(k-1, \mathbf{b} - \mathbf{A}_k) = 1$ and $d(k-1, \mathbf{b} - \mathbf{A}_k) = 1 \rightarrow stop - adp = 1$
end
Output: $stop - adp = 0$ or 1

Table 4.2: Description of the ADP Stopping Condition.

variables to be fixed once x_k^{adp} is determined in the policy assignment. So, the larger the value of the `lag-size` parameter l, the more variables are fixed every time the policy assignment is called. Hence, the lag policy assignment effectively reduces the computational burden. Due to our construction $l = \lfloor \frac{k}{\text{lag-time}} \rfloor$, for a certain `lag-time` value, we dynamically update the value of `lag-size` parameter l for varying k in $[1, n]$ in the backward phase. In our computational study for the multi-dimensional knapsack problem, we use varying `lag-time` values.

Let us illustrate the lag policy assignment by an example. Assume that $k = 50$ (i.e., the algorithm is at stage $k = 50$ in the backward phase and $F \supseteq \{k+1, \cdots, n\}$) and `lag-time`=10. While setting x_k^{adp}, $k \notin F$ we consider the subproblems

$MKP(k-1, \mathbf{b}-A_k x)$, $x = 0, 1$ and we apply $BH(k-1, \mathbf{b}-\mathbf{A}_k x)$, $x = 0, 1$ and get $x_{BH}(k, \mathbf{b}-\mathbf{A}_k x)$, $x = 0, 1$. Lag policy fixes additional l variables at once aiming to expedite solution construction. In this example, $l = 5$ and according to proposed scheme we set $(x^f_{k-5}, \cdots, x^f_{k-1}) = (x_{BH}(k-1, \mathbf{b}-\mathbf{A}_k x^*)_{k-5}, \cdots, x_{BH}(k-1, \mathbf{b}-\mathbf{A}_k x^*)_{k-1})$ where $x^* = x^{adp}_k$. We update F by $F \cup \{k-5, \cdots, k-1\}$. Updating $\mathbf{x}^f$ and F means *fixing* the variables $x_{k-5}, \cdots, x_{k-1}$ (i.e., $(x^{adp}_{k-5}, \cdots, x^{adp}_{k-1}) = (x^f_{k-5}, \cdots, x^f_{k-1})$) automatically in step 4 of base-heuristic learning algorithm in Table 4.3.

We describe the base heuristic learning algorithm to solve multi-dimensional knapsack problems in Table 4.3.

The Base-heuristic Learning Algorithm

1: Initialization:
   $k = n,\ \mathbf{b} = \mathbf{b}_0,\ \mathbf{x}^f = \mathbf{0}$
   $stop - adp = 0$
2: Apply $BH(k, \mathbf{b})$
   Output: $x_{BH}(k, \mathbf{b}),\ H_k(\mathbf{b}),\ p^f(k, \mathbf{b}),\ s^o(k, \mathbf{b})$
   $p^f(k, \mathbf{b}) = 0 \rightarrow$ exit (no feasible solution)
   $s^o(k, \mathbf{b}) = 1 \rightarrow$ exit (optimal solution)
   $\mathbf{x}^{best} \leftarrow x_{BH}(k, \mathbf{b}),\ z^{best} \leftarrow H_k(\mathbf{b})$
3: Apply *reduced cost fixing*
   Output: $\mathbf{x}^f$
4: while $(k > 2)$ and $(stop - adp = 0)$ do
   begin (while)
      if $(x_k^f = 0$ or $x_k^f = 1)$ then
      {
        -$x_k^{adp} \leftarrow x_k^f$
      }
      else
      {
        -*policy-assignment*
          Output: x_k^{adp}, (0 or 1)
        -*update best solution*
          Output: $\mathbf{x}^{best}, z^{best}$
        -*stopping condition*
          Output: $stop - adp$, (0 or 1)
        -*lag policy assignment*
          Output: $\mathbf{x}^f$
      }
      $\mathbf{b} \leftarrow \mathbf{b} - \mathbf{A}_k x_k^{adp}$
      $k \leftarrow k - 1$
   end (while)
5: x_1^{adp} solves $MKP(1, \mathbf{b})$
6: Output: $\mathbf{x}^{best}$ and z^{best}.

Table 4.3: The Base Heuristic Learning Algorithm for the MKP problem.

4.4.2 Base-Heuristic Selection

The selection of the base-heuristic is very important in the proposed base-heuristic learning framework. In essence, any methodology that is capable of solving the multi-dimensional knapsack problem can be considered as one of the alternatives for the base-heuristic selection. We employ a certain base-heuristic many times within the base-heuristic learning framework. Thus, we focus on heuristic methodologies with small computation time requirements. Based on our survey and available computational experience provided in the literature we consider the following heuristic methodologies as our base-heuristic for the multi-dimensional knapsack problem: Senju and Toyoda's *Dual Gradient Algorithm* [77], Toyoda's *Primal Gradient Algorithm* [85], Loulou and Michaelides' *Greedy-like Heuristics* [45], and Kochenberger et. al.'s *Incremental Heuristic* [29]. We also propose a new heuristic, namely *adaptive fixing*. We will be describing the details of these base-heuristic in the following sections.

4.4.2.1 Base-Heuristics from the literature

In this section, we describe heuristics from the literature in a greater detail. See Table 4.4 for the *Dual Gradient* , Table 4.5 for the *Primal Gradient*, Table 4.6 for the *Greedy-like*, Table 4.7 for the *Incremental* heuristic methodology.

Senju and Toyoda's *Dual Gradient Algorithm* [77]

1: initialization:

$$X_0 = \emptyset,\ X_1 = N,\ \mathbf{R}_u = \sum_{j \in X_1} \mathbf{H}_j,\ \mathbf{R}_a = \mathbf{e} - \mathbf{R}_u$$

2: calculate direction vector $\mathbf{S} = (S_1, \cdots, S_m)$:

$$S_i = \sum_{j \in X_1} h_{ij} - 1,\ (0) \text{ if } \sum_{j \in X_1} h_{ij} \geq 1,\ (\text{otherwise})$$

3: while ($\exists\ i$ such that $S_i > 0$) do

begin

penalty calculation: $P_j = \mathbf{H}_j' \mathbf{S}$

gradient calculation: $G_j = c_j / P_j$

variable selection: $j^* = \text{argmin}\{G_j | j \in X_1\}$

update information:

-$X_1 \leftarrow X_1 \setminus \{j^*\}$
-$X_0 \leftarrow X_0 \cup \{j^*\}$
-$\mathbf{S} \leftarrow \mathbf{S} - \mathbf{H}_{j^*}$
-$\mathbf{R}_u \leftarrow \mathbf{R}_u - \mathbf{H}_{j^*}$
-$\mathbf{R}_a \leftarrow \mathbf{R}_a + \mathbf{H}_{j^*}$

end

4: candidate list:

$$X_c = \{j | j \in X_0, \mathbf{H}_j \leq \mathbf{R}_a\}$$

5: while ($X_c \neq \emptyset$) do

begin

$j^* = \text{argmax}\{c_j | j \in X_c\}$
$X_1 \leftarrow X_1 \cup \{j^*\}$
$X_0 \leftarrow X_0 \setminus \{j^*\}$
$X_c = \{j | j \in X_0, \mathbf{H}_j \leq \mathbf{R}_a\}$
$\mathbf{R}_a \leftarrow \mathbf{R}_a + \mathbf{H}_{j^*}$

end

6: output: $\mathbf{x}^H, z^H$

$$x_j^H = 0,\ j \in X_0,\ x_j^H = 1,\ j \in X_1$$

$$Z^H = \sum_{j=1}^{n} c_j x_j^H$$

Table 4.4: Description of the *Dual Gradient Algorithm.*

Toyoda's *Primal Gradient Algorithm* [85]

1: initialization:

$$X_1 = \emptyset,\ X_0 = N,\ \mathbf{R}_u = \sum_{j \in X_1} \mathbf{H}_j,\ \mathbf{R}_a = \mathbf{e} - \mathbf{R}_u$$

2: candidate List:

$$X_c = \{j | j \in X_0, \mathbf{H}_j \leq \mathbf{R}_a\}$$

3: while $(X_c \neq \emptyset)$ do

begin

penalty calculation:

$P_j = \mathbf{H}_j'\mathbf{e},\ (\mathbf{H}_j'\mathbf{R}_u)$ if $\mathbf{R}_u = \mathbf{0}$, (otherwise)

gradient calculation: $G_j = c_j / P_j$

variable selection: $j^* =$ argmax$\{G_j | j \in X_1\}$

update information

-$X_1 \leftarrow X_1 \cup \{j^*\}$
-$X_0 \leftarrow X_0 \setminus \{j^*\}$
-$\mathbf{R}_u \leftarrow \mathbf{R}_u + \mathbf{H}_{j^*}$
-$\mathbf{R}_a \leftarrow \mathbf{R}_a - \mathbf{H}_{j^*}$
-$X_c = \{j | j \in X_0, \mathbf{H}_j \leq \mathbf{R}_a\}$

end

6: output: $\mathbf{x}^H, z^H$

$$x_j^H = 0,\ j \in X_0,\ x_j^H = 1,\ j \in X_1$$

$$Z^H = \sum_{j=1}^{n} c_j x_j^H$$

Table 4.5: Description of the *Primal Gradient Algorithm.*

Loulou and Michaelides' *Greedy-like Heuristics* [45]

1: initialization:

$$X_1 = \emptyset,\ X_0 = N,\ \mathbf{R}_u = \sum_{j \in X_1} \mathbf{H}_j,\ \mathbf{R}_a = \mathbf{e} - \mathbf{R}_u$$

2: candidate List:

$$X_c = \{j | j \in X_0, \mathbf{H}_j \leq \mathbf{R}_a\}$$

3: while $(X_c \neq \emptyset)$ do

begin

penalty calculation:

alternative 1

$$P_j = \max_{i=1,\cdots,m} \{ \frac{(R_{a,i}+h_{ij})(\sum_{k \in X_c} h_{ik} - h_{ij})}{(1-R_{a,i}-h_{ij})} \}$$

alternative 2

$$P_j = \max_{i=1,\cdots,m} \{ \frac{(R_{a,i}+h_{ij})(\sum_{k \in X_c} h_{ik} - h_{ij})^{1/2}}{(1-R_{a,i}-h_{ij})^{1/2}} \}$$

alternative 3

$$P_j = \sum_{i=1}^{m} \{ \frac{(R_{a,i}+h_{ij})(\sum_{k \in X_c} h_{ik} - h_{ij})^{1/2}}{(1-R_{a,i}-h_{ij})^{1/2}} \}$$

gradient calculation: $G_j = c_j / P_j$

variable selection: $j^* = \text{argmax}\{G_j | j \in X_1\}$

update information

-$X_1 \leftarrow X_1 \cup \{j^*\}$
-$X_0 \leftarrow X_0 \setminus \{j^*\}$
-$\mathbf{R}_u \leftarrow \mathbf{R}_u + \mathbf{H}_{j^*}$
-$\mathbf{R}_a \leftarrow \mathbf{R}_a - \mathbf{H}_{j^*}$
-$X_c = \{j | j \in X_0, \mathbf{H}_j \leq \mathbf{R}_a\}$

end

6: output: $\mathbf{x}^H, z^H$

$$x_j^H = 0,\ j \in X_0,\ x_j^H = 1,\ j \in X_1$$

$$Z^H = \sum_{j=1}^{n} c_j x_j^H$$

Table 4.6: Description of the *Greedy-like Heuristic.*

Kochenberger et. al.'s *Incremental Heuristic* [29]

1: initialization:

$$X_1 = \emptyset,\ X_0 = N,\ \mathbf{R}_u = \sum_{j \in X_1} \mathbf{H}_j,\ \mathbf{R}_a = \mathbf{e} - \mathbf{R}_u$$

2: candidate List:

$$X_c = \{j | j \in X_0,\ \overline{\mathbf{b}} - \mathbf{A}_j \geq \mathbf{0}\}$$

3: while $(X_c \neq \emptyset)$ do

begin

penalty calculation:

$\overline{a}_{ij} = a_{ij}/\overline{b}_i$, (0) if $\overline{b}_i > 0$, (otherwise)
$P_j = \sum_{i=1}^{m} \overline{a}_{ij}$

gradient calculation: $G_j = c_j/P_j$

variable selection: $j^* =$ argmax$\{G_j | j \in X_1\}$

update information

-$X_1 \leftarrow X_1 \cup \{j^*\}$
-$X_0 \leftarrow X_0 \setminus \{j^*\}$
-$\overline{\mathbf{b}} = \mathbf{b} - \sum_{j \in X_1} \mathbf{A}_j$
-$X_c = \{j | j \in X_0,\ \overline{\mathbf{b}} - \mathbf{A}_j \geq \mathbf{0}\}$

end

6: output: $\mathbf{x}^H, z^H$

$$x_j^H = 0,\ j \in X_0,\ x_j^H = 1,\ j \in X_1$$

$$Z^H = \sum_{j=1}^{n} c_j x_j^H$$

Table 4.7: Description of the *Incremental Heuristic.*

4.4.2.2 A New Base-heuristic: Adaptive Fixing

In this section we propose a new heuristic algorithm based on the solution of the linear programming relaxation, namely the *adaptive fixing heuristic*. Let us define by $(LP(k, \mathbf{b})|X_0, X_1)$ the linear programming relaxation of the subproblem $MKP(k, \mathbf{b})$ with the additional constraints of $x_j = 0,\ j \in X_0$ and $x_j = 1,\ j \in X_1$.

The adaptive fixing heuristic consists of two phases. In the first phase, we assume X_0 and X_1 to be empty sets which means none of the variables have been fixed to a value. Once the linear programming solution $\mathbf{x}^{LP}$ is obtained to $(LP(k, \mathbf{b})|\emptyset, \emptyset)$ we set $x_j = 0$ if $0 \leq x_j^{LP} < \gamma$ and $x_j = 1$ if $x_j^{LP} = 1$. While setting variables to 0 or 1, we dynamically update X_0 and X_1.

In the second phase, once the linear programming solution $\mathbf{x}^{LP}$ is obtained to $(LP(k, \mathbf{b})|\ X_0, X_1)$ we find the lowest fractional variable x_j^{LP} and set it to 0. We set $x_j = 0$ if $x_j^{LP} = 0$ and $x_j = 1$ if $x_j^{LP} = 1$. We repeat this fixing process until we reach an integral solution, i.e., $x_j = 0$ or $1\ \forall j \in \{1, \cdots, n\}$. In other words, we do fixing until $X_0 \cup X_1 = \{1, \cdots, n\}$. See Table 4.8 for an algorithmic description of the adaptive fixing heuristic.

As described above the user specifies the parameter γ. Our preliminary computational study suggests adaptive fixing with low γ (e.g., $\gamma = 0.05, 0.10$) values generates usually better solutions. On the other hand, it uses more computation time. Thus, in order to balance the trade-off we set $\gamma = 0.25$ in our computational study because the base-heuristic learning algorithm described in Table 4.3 suggests many calls for the adaptive-fixing heuristic.

The *adaptive fixing* heuristic

1: input: γ (e.g, $\gamma = 0.25$).

2: initialization:

$k = n,\ \mathbf{b} = \mathbf{b}_0,\ X_1 = \emptyset,\ X_0 = \emptyset$

3: candidate list:

$X_c \leftarrow \{j | j \in N \setminus \{X_0, X_1\}\}$

4: solve $(LP(k, \mathbf{b})|X_0, X_1)$ and get $\mathbf{x}^{LP}$

5: update information:

$X_0 \leftarrow X_0 \cup \{j | j \in X_c,\ x_j^{LP} = 0\}$

$X_0 \leftarrow X_0 \cup \{j | j \in X_c,\ 0 < x_j^{LP} < \gamma\}$

$X_1 \leftarrow X_1 \cup \{j | j \in X_c,\ x_j^{LP} = 1\}$

$X_c \leftarrow \{j | j \in N \setminus \{X_0, X_1\}\}$

6: **while** $(X_c \neq \emptyset)$ **do**

begin

Solve $(LP(k, \mathbf{b})|X_0, X_1)$ and get $\mathbf{x}^{LP}$

$X_0 \leftarrow X_0 \cup \{j | j \in X_c,\ x_j^{LP} = 0\}$

$X_1 \leftarrow X_1 \cup \{j | j \in X_c,\ x_j^{LP} = 1\}$

$j^* = \underset{0 < x_j^{LP} < 1}{\operatorname{argmin}} \{x_j^{LP}\}$

$X_0 \leftarrow X_0 \cup \{j^*\}$

$X_c \leftarrow N \setminus \{j^*\}$

end

7: output: $\mathbf{x}^H, z^H$

$x_j^H = 0,\ j \in X_0,\ x_j^H = 1,\ j \in X_1$

$$Z^H = \sum_{j=1}^{n} c_j x_j^H$$

Table 4.8: Description of the *adaptive fixing heuristic.*

Truncation Heuristic:

We consider the special case $\gamma = 1$ of the adaptive fixing heuristic as the *linear programming based truncation heuristic*, or concisely *truncation heuristic*. In other words, this heuristic first obtains the LP solution $\mathbf{x}^{LP}$. It then sets $x_j = 0$ if $0 \leq x_j^{LP} < 1$ and $x_j = 1$ if $x_j^{LP} = 1$. Thus, the truncation heuristic can be seen as adaptive fixing with the first phase and $\gamma = 1$ as described in Table 4.9.

The *truncation* heuristic
1: initialization: $k = n,\ \mathbf{b} = \mathbf{b}_0,\ X_1 = \emptyset,\ X_0 = \emptyset$ 2: candidate list: $X_c \leftarrow \{j \mid j \in N \setminus \{X_0, X_1\}\}$ 3: solve $(LP(k, \mathbf{b}) \mid X_0, X_1)$ and get $\mathbf{x}^{LP}$ 4: update information: $X_0 \leftarrow X_0 \cup \{j \mid j \in X_c,\ x_j^{LP} = 0\}$ $X_0 \leftarrow X_0 \cup \{j \mid j \in X_c,\ 0 < x_j^{LP} < 1\}$ $X_1 \leftarrow X_1 \cup \{j \mid j \in X_c,\ x_j^{LP} = 1\}$ 5: output: $\mathbf{x}^H, z^H$ $x_j^H = 0,\ j \in X_0,\ x_j^H = 1,\ j \in X_1$ $Z^H = \sum_{j=1}^{n} c_j x_j^H$

Table 4.9: Description of the *truncation heuristic.*

4.5 Computational Results

In this section, we provide computational results to the multi-dimensional knapsack problem. Our basic aim is to demonstrate that approximate dynamic programming algorithms utilizing both the statistical and base-heuristic learning framework can generate good solutions for large and difficult instances in reasonably short computation times. We compare heuristics from the literature with our proposed heuristics *adaptive fixing* and *truncation* heuristics. We provide an extensive computational study to illustrate the performance of both statistical and base-heuristic learning framework. We will also be comparing the most promising ADP algorithm with CPLEX and Chu and Beasley's genetic algorithm (see Chu and Beasley [15]). Our criteria for comparing various approaches are solution quality, computation time, and robustness, i.e., the degree of deviation of the computational resources needed to solve the problems as the instances of the same size change.

We construct *uncorrelated, weakly* and *strongly correlated* random multi dimensional knapsack problems. Computational experience in the literature suggests that strongly correlated multi-dimensional problems are the most difficult ones (see Chu and Beasley [15]).

Let $x \sim U(1, X)$ denote an integer number that is uniformly generated in $[1, X]$. In our computational study, we consider the following problem types:

- **Uncorrelated (UC)**: $c_j \sim U(1, C)$ and $a_{ij} \sim U(1, A)$, i.e., c_j, a_{ij} are uniformly distributed in $[1, C], [1, A]$ respectively.
- **Weakly correlated (WC)**: $a_{ij} \sim U(1, A)$ and $c_j = \max\{1, U(\sum_i a_{ij}/m - wc, \sum_i a_{ij}/m + wc)\}$.
- **Strongly correlated (SC)**: $a_{ij} \sim U(1, A)$ and $c_j = \sum_i a_{ij}/m + sc$.

Given the number of constraints m and the number of variables n we generate 10 test problems in each class (uncorrelated, weakly, and strongly correlated) with

different parameters C, A, wc, sc as provided in Table 4.10. We set $b_{0,i} = 0.5 \sum_{j=1}^{n} a_{ij}$. All computational studies are done on a Dell Precision 410 with Linux operating system. We use the following notation to present the computational study.

- H1: Adaptive Fixing Heuristic (Table 4.8 on page 112).
- H2: Truncation Heuristic (Table 4.9 on page 113).
- H3: Primal Gradient Heuristic (Table 4.5 on page 108).
- H4: Dual Gradient Heuristic (Table 4.4 on page 107).
- H5: Greedy-like Heuristic (Table 4.6 on page 109).
- H6: Incremental Heuristic (Table 4.7 on page 110).
- ADP-X: Base heuristic learning algorithm with the base-heuristic X (e.g., ADP-H1 means base-heuristic learning algorithm with the base-heuristic H1, namely *adaptive fixing*).
- ADP-P: ADP with parametric approximation.
- ADP-N: ADP with nonparametric approximation.
- ADP-B: ADP with CPLEX based branch-and-bound, i.e., we use CPLEX based branch-and-bound as the base-heuristic with a user-specified CPLEX tolerance parameter `CPX-PARAM-EPAGAP`.
- Let v(X) be the objective value of the solution obtained by the methodology X, e.g., v(ADP-H) is the objective value of the solution obtained by the methodology ADP-H.
- Let PE(X) be the percentage deviation of v(X) from the LP objective value v(LP), i.e., PE(X) $= \frac{v(LP)-v(X)}{v(LP)} \times 100$.
- PI(X): Percentage Improvement of ADP-X over X, i.e., $\frac{PE(X)-PE(ADP-X)}{PE(X)} \times 100$
- T(X): Computation time of the methodology X in CPU seconds.

UC		WC		SC	
A	C	A	wc	A	sc
100	100	100	10	100	10
500	100	500	50	500	50
500	1,000	1,000	100	1,000	100
1,000	500	1,000	500	1,000	500
1,000	1,000	3,000	300	2,000	500
1,000	2,000	4,000	500	3,000	100
5,000	1,000	5,000	500	5,000	500
5,000	500	10,000	1,000	5,000	1,000
10,000	1,000	10,000	2,000	10,000	2,000
10,000	5,000	15,000	3,000	10,000	5,000

Table 4.10: Parameters for the uncorrelated, weakly and strongly correlated type problems.

4.5.1 Comparison of base-heuristics

In this section we compare the proposed stand-alone heuristics, namely truncation heuristic H1 and adaptive fixing heuristic H2, with some heuristics from the literature, namely primal gradient heuristic H3 (Table 4.5), dual gradient heuristic H4 (Table 4.4), greedy-like heuristic H5 (Table 4.6), incremental heuristic H6 (Table 4.7). We will be comparing these heuristics on uncorrelated, weakly correlated and strongly correlated type problem instances as described earlier.

Table 4.11 presents the computational results for uncorrelated type multi dimensional knapsack problems. We observe both *truncation heuristic* H1 and *adaptive fixing heuristic* H2 generate good solutions in short times. Heuristics H1 and H2 have average percentage errors of 4.45 and 1.14, respectively. The Table 4.11 suggests that the heuristics proposed in the literature do not generate strong solutions while their computation times are within acceptable limits. Heuristics H3, H4, H5 and H6 have average percentage errors of 34.18, 36.78, 34.73 and 35.59, respectively, that are significantly larger than the average percentage deviations of both H1 and H2. We observe that H5 is the heuristic that requires the largest computation time. This table

also illustrates that heuristics H3, H4 and H6 are competitively similar for uncorrelated type problems in terms of both solution quality and computation time. Overall, for uncorrelated type problems, adaptive fixing heuristic H2 can be considered as the leading base-heuristic based on both solution quality and computation time criterion.

		H1		H2		H3		H4		H5		H6	
m	n	T	PE	T	PE	T	PE	T	PE	T	PE	T	PE
10	100	0.01	3.92	0.01	1.69	0.00	35.12	0.01	38.56	0.11	34.49	0.02	34.52
50	100	0.02	9.59	0.07	2.13	0.03	31.77	0.02	37.79	0.76	33.67	0.07	30.96
100	500	0.36	3.25	0.93	0.45	2.21	35.31	1.51	36.05	209.17	36.09	3.46	34.59
100	1000	1.04	1.94	2.69	0.28	10.44	34.51	6.78	34.71	2,015.61	34.68	15.45	34.28

Table 4.11: Performance of base-heuristics: Uncorrelated type instances.

Table 4.12 presents the computational results for weakly correlated type multi dimensional knapsack problems. We observe that quality of solutions generated by truncation heuristic H1 degrades considerably in comparison to its performance for the uncorrelated type instances. We also note that heuristics H3, H5 and H6 achieve better average percentage deviations than H1 for the problems of size $m = 50, n = 100$. Adaptive fixing heuristic H2 still generates good solutions with an average percentage deviation of 1.81 in reasonably short but larger computation times compared to uncorrelated type instances. This is mainly because of larger number of iterations required to obtain an optimal LP solution for problems with stronger correlation (e.g., weakly correlated vs. uncorrelated type problems). Table 4.12 suggests that the heuristics H3, H4, H5 and H6 performed better for the weakly correlated type problem instances compared to uncorrelated type ones (see Table 4.11) while their computation times stayed stable. Heuristics H3, H4, H5 and H6 have average percentage errors of 17.20, 19.14, 17.66 and 16.59, respectively. We note again that H5 consumes the largest computation time. Based on solution quality and computation time criterion, heuristics H3, H4 and H6 perform competitively similar for weakly correlated type problems. Overall, adaptive fixing heuristic H2 is the leading heuristic for weakly correlated type instances in terms of both solution quality and computa-

tion time.

		H1		H2		H3		H4		H5		H6	
m	n	T	PE	T	PE	T	PE	T	PE	T	PE	T	PE
10	100	0.01	8.08	0.02	2.36	0.01	18.52	0.00	20.98	0.22	19.70	0.01	16.86
50	100	0.05	26.85	0.20	3.58	0.04	17.38	0.02	21.65	1.07	17.56	0.06	16.88
100	500	0.97	10.29	3.25	0.85	2.42	16.97	1.28	17.25	215.00	17.67	3.41	16.88
100	1000	2.31	6.29	7.13	0.46	10.68	15.92	5.58	16.68	1,995.10	15.72	14.93	15.74

Table 4.12: Performance of base-heuristics: Weakly-correlated type instances.

Table 4.13 presents the computational results for strongly correlated type multi dimensional knapsack problems. We observe that quality of solutions generated by truncation heuristic H1 degrades *significantly* in comparison to its performance for the uncorrelated and weakly correlated type instances. We also note that heuristics H3, H5 and H6 achieve better average percentage deviations than H2 for all the problem instances except the ones of size $m = 10, n = 100$. Adaptive fixing heuristic H2 still generates good solutions with an average percentage deviation of 2.50 in reasonably short but longer computation times compared to uncorrelated and weakly correlated type instances. As argued, this is mainly because of larger number of iterations required to obtain an optimal LP solution for problems with stronger correlation (e.g., strongly correlated vs. uncorrelated and weakly correlated type problems). The quality of solutions obtained by heuristics H3, H4, H5 and H6 improved significantly with average percentage deviations of 7.75, 10.44, 8.40 and 7.96, respectively. H5 requires the largest computation time while heuristics H3 and H6 perform better in terms of both solution quality and computation time. In general, the computation times of heuristics H3, H4, H5 and H6 stayed stable. Overall, adaptive fixing heuristic H2 is still the leading heuristic for strongly correlated type problems based on both solution quality and computation time criterion.

Our computational study summarizes that the proposed adaptive fixing heuristic H2 is a practical heuristic methodology to multi-dimensional knapsack problems that

		H1		H2		H3		H4		H5		H6	
m	n	T	PE	T	PE	T	PE	T	PE	T	PE	T	PE
10	100	0.02	9.81	0.02	3.84	0.01	8.83	0.01	13.83	0.21	10.72	0.01	10.11
50	100	0.13	50.45	0.60	4.26	0.04	10.84	0.03	14.47	1.07	11.23	0.06	10.31
100	500	6.93	19.87	14.92	1.30	2.27	6.50	1.57	7.58	201.82	6.58	3.52	6.40
100	1000	8.42	10.00	18.67	0.59	10.79	4.84	5.69	5.89	1,986.85	5.07	14.95	5.03

Table 4.13: Performance of base-heuristics: Strongly-correlated type instances.

generates good solutions for different types of problem instances. Average percentage deviations of H2 are 1.14, 1.81 and 2.50 for uncorrelated, weakly correlated and strongly correlated type instances, respectively. We observe that under a certain m, n average percentage deviation does not deviate dramatically for different types of problem instances. For instance, for the problems of size $m = 100, n = 1,000$, average percentage deviations of H2 (i.e., average of PE(H2) values) are 0.28, 0.46 and 0.59 for uncorrelated, weakly correlated and strongly correlated instances, respectively. Thus, from a solution quality perspective we consider adaptive fixing heuristic H2 as a robust methodology. On the other extreme, we can not say that H2 is robust from a computation time perspective since for the same problems average computation times of H2 (i.e., average of T(H2) values) are 2.69, 7.13 and 18.67 for uncorrelated, weakly correlated and strongly correlated instances, respectively.

4.5.2 Computational results for the statistical learning framework

In this section, our objective is to demonstrate the performance of the statistical learning framework on the same uncorrelated, weakly correlated and strongly correlated type problem instances. We also compare the statistical learning framework with the proposed heuristics, namely H1 and H2, and heuristics from the literature, namely H3, H4, H5 and H6.

The statistical learning framework generates a solution under the proposed algorithmic design (i.e., the proposed kernel function, sampling type selection etc.) as

described in section 4.3.4 for a certain sample size s. In other words, once a sample size s is given the approximate dynamic programming algorithm is fixed to the ADP with parametric or nonparametric as described in section 4.3.4. Let ADP-P denote the approximate dynamic programming with parametric approximation and ADP-N denote the approximate dynamic programming with nonparametric approximation. Thus, for a certain problem instance of size m, n, once the sample size s is given, we can use ADP-P or ADP-N as a heuristic methodology to obtain a feasible solution. In our computational study, we take sample size s values from the sets $G_1 = \{5, 10, 15, 20, 25\}$ and $G_2 = \{50, 100, 150, 200, 250\}$. We use the notation G to denote either G_1 or G_2. For an m, n combination (e.g., $m = 10, n = 100$) we apply statistical learning (ADP-P or ADP-N) with varying sample sizes $s \in G$ (e.g., $G = G_1$). We obtain performance statistics such as solution quality, computation time etc. for each $s \in G$. We report the *best solution* and its corresponding statistics out of all solutions generated for varying $s \in G$. Thus, we construct tables for each type of problem instance for both $s \in G_1$ and $s \in G_2$.

Tables 4.14 and 4.15 present computational results of ADP-P and ADP-N for uncorrelated type problems with $s \in G_1$ and $s \in G_2$, respectively. These tables indicate that ADP-P and ADP-N generated better solutions than H3, H4, H5, H6 in shorter computation times. In comparison to H1 and H2, ADP-P and ADP-N are inferior in terms of both solution quality and computation time. Average percentage deviations of ADP-N (i.e., average of PE(ADP-N) values from tables 4.14 and 4.15) are 11.12 and 8.81 for $s \in G_1$ and $s \in G_2$, respectively. This illustrates that, on average, the larger the sample sizes the better the solutions obtained by ADP-N. As the sample size increases the computation time of both ADP-N and ADP-P increase. For instance, the average computation times of ADP-N (i.e., average of T(ADP-N) values from tables 4.14 and 4.15) are 1.31 and 12.02 for $s \in G_1$ and $s \in G_2$, respectively. For a certain m, n and G, we observe that quality of solutions differ *substantially* from one another under both ADP schemes, namely ADP-P and ADP-N. As an illustrative example, let us focus on the statistics of ADP-P for the

problems of size $m = 10, n = 100$ with $G = G_1$ as presented in Table 4.14. Even though, the minimum (0.03), average (0.06) and maximum (0.10) computation times are close to each other, the minimum (8.0118), average (16.8413), and maximum (25.0646) percentage deviations differ considerably from one another. Thus, we can not consider both ADP-P and ADP-N as robust methodologies from a solution quality perspective. This implies that ADP-P and ADP-N need extra tuning to determine the best sample size. On the other extreme, our computation time statistics suggests that both ADP-P and ADP-N are robust from a computation time point of view.

		T(ADP-P)			PE(ADP-P)			T(ADP-N)			PE(ADP-N)		
m	n	min	avg	max	min	avg	max	min	avg	max	min	avg	max
10	100	0.03	0.06	0.10	8.0118	16.8413	25.0646	0.02	0.05	0.07	3.7448	10.5054	21.9889
50	100	0.10	0.19	0.26	10.1345	19.0510	30.6011	0.09	0.15	0.23	4.9710	14.4108	26.8070
100	500	0.95	1.61	2.25	4.4152	11.4099	21.3784	0.96	1.62	2.33	2.5496	10.6635	21.8752
100	1000	1.98	3.50	4.96	4.1738	7.0525	11.5795	1.98	3.42	4.85	3.4661	8.8820	22.9663

Table 4.14: ADP-P, ADP-N: Uncorrelated type instances, $s \in G_1$.

		T(ADP-P)			PE(ADP-P)			T(ADP-N)			PE(ADP-N)		
m	n	min	avg	max	min	avg	max	min	avg	max	min	avg	max
10	100	0.21	0.78	1.49	8.0118	16.8413	25.0646	0.16	0.57	1.06	3.3962	9.1635	16.1160
50	100	0.59	1.85	3.51	10.1345	19.0510	30.6011	0.54	1.54	2.79	4.3317	11.6657	19.7060
100	500	5.32	14.49	26.49	4.4152	11.4099	21.3784	5.54	14.90	25.63	3.3983	7.4069	18.7507
100	1000	11.54	29.61	51.10	4.1738	7.0525	11.5795	11.70	31.06	53.76	2.7383	6.6097	14.7399

Table 4.15: ADP-P, ADP-N: Uncorrelated type instances, $s \in G_2$.

Tables 4.16 and 4.17 present computational results of ADP-P and ADP-N for weakly correlated type problems with $s \in G_1$ and $s \in G_2$, respectively. ADP-P and ADP-N continue to achieve better quality of solutions compared to those obtained by heuristics H3, H4, H5 and H6. For instance, average percentage deviations of ADP-P and ADP-N with $s \in G_1$ are 10.72 and 10.04, respectively, while average percentage deviations of heuristics H3, H4, H5 and H6 are 17.20, 19.14, 17.90 and 16.59, respectively. H2 continues to dominate both ADP-P and ADP-N in terms of solution quality and computation time. H1 generated inferior solutions with an average percentage deviation of 12.88 in comparison to ADP-P and ADP-N. An increase in the

sample size does not necessarily improve the quality of solutions. For instance, for the problem instances of size $m = 10, n = 100$ ADP-N with $s \in G_1$ and $s \in G_2$ have average percentage deviations of 11.87 and 12.42, respectively. Average percentage deviations of ADP-N (i.e., average of PE(ADP-N) values from tables 4.16 and 4.17) are 10.72 and 9.52 for $s \in G_1$ and $s \in G_2$, respectively. This illustrates that, on average, the larger the sample sizes the better the solutions obtained by ADP-N. As the sample size increases the computation time of both ADP-N and ADP-P increases. For instance, the average computation times of ADP-N (i.e., average of T(ADP-N) values from tables 4.16 and 4.17) are 1.45 and 12.34 for $s \in G_1$ and $s \in G_2$, respectively. We observe that both ADP-P and ADP-N achieve similar statistics for uncorrelated and weakly correlated type instances. For instance, for $s \in G_1$, the average percentage deviation and computation time of ADP-N are 11.12 and 1.31, respectively, for uncorrelated type problems while the average percentage deviation and computation time of ADP-N are 11.87 and 1.45, respectively, for weakly correlated type problems. This indicates that statistical learning is *robust across* different types of problem instances. For a certain m, n and G, we observe that quality of solutions differ *substantially* from one another under both ADP schemes, namely ADP-P and ADP-N. As an illustrative example, let us focus on the statistics of ADP-N for the problems of size $m = 10, n = 100$ with $G = G_1$ as presented in Table 4.16. Even though, the minimum (0.02), average (0.05) and maximum (0.07) computation times are close to each other, the minimum (6.0482), average (11.8696), and maximum (19.7634) percentage deviations differ considerably from one another. Thus, for the weakly correlated type instances, we can not consider both ADP-P and ADP-N as robust methodologies from a solution quality perspective. This implies that ADP-P and ADP-N need extra tuning to determine the best sample size. We observe that both ADP-P and ADP-N are robust in terms of computation time.

Tables 4.18 and 4.19 present computational results of ADP-P and ADP-N for strongly correlated type problems with $s \in G_1$ and $s \in G_2$, respectively. Heuristics H3, H4, H5 and H6 perform quite competitively compared to ADP-P and ADP-N.

		T(ADP-P)			PE(ADP-P)			T(ADP-N)			PE(ADP-N)		
m	n	min	avg	max	min	avg	max	min	avg	max	min	avg	max
10	100	0.03	0.06	0.09	9.7910	14.0499	22.6718	0.02	0.05	0.07	6.0482	11.8696	19.7634
50	100	0.09	0.19	0.27	10.4399	14.0215	26.2094	0.09	0.16	0.24	9.0369	12.9657	23.3960
100	500	0.96	1.83	2.88	5.1505	7.9797	10.7776	0.95	1.74	2.62	6.0106	8.4587	22.4693
100	1000	1.99	3.76	6.13	4.1417	6.8218	8.7893	2.00	3.83	6.03	3.6354	6.8907	17.0984

Table 4.16: ADP-P, ADP-N: Weakly-correlated type instances, $s \in G_1$.

		T(ADP-P)			PE(ADP-P)			T(ADP-N)			PE(ADP-N)		
m	n	min	avg	max	min	avg	max	min	avg	max	min	avg	max
10	100	0.21	0.74	1.41	9.7910	14.0499	22.6718	0.17	0.56	1.04	7.1990	12.4185	17.1103
50	100	0.62	1.94	3.66	10.4399	14.0215	26.2094	0.55	1.59	2.82	8.0563	12.0423	16.4591
100	500	5.81	15.42	27.07	5.1505	7.9797	10.7776	5.80	15.54	27.30	4.5702	7.2631	10.7053
100	1000	12.47	30.10	51.63	4.1417	6.8218	8.7893	12.16	31.67	54.26	2.3647	6.3751	10.9660

Table 4.17: ADP-P, ADP-N: Weakly-correlated type instances, $s \in G_2$.

For instance, average percentage deviations of H3 (i.e., average of PE(H3) values from Table 4.13), ADP-P, ADP-N are (8.83,10.84,6.50,4.84), (8.63,10.32,5.38,3.87) and (8.73,10.47,5.32,4.00), respectively, for the problem instances of size $(m, n) \in \{(10, 100), (50, 100), (100, 500), (100, 1000)\}$. This example illustrates a close competition in terms of solution quality between ADP-P, ADP-N and H3. H2 continues to dominate both ADP-P and ADP-N in terms of both solution quality and computation time. H1 generated significantly inferior solutions, with an average percentage deviation of 22.46, compared to those obtained by ADP-P and ADP-N with average percentage deviations of 7.05 and 7.13, respectively. The quality of solutions of ADP-P and ADP-N with $s \in G_2$ are inferior compared to those obtained by ADP-P and ADP-N with $s \in G_1$. This observation, being unique only for strongly correlated type instances, suggests, on average, an increase in the sample size does not improve the solution quality of the corresponding approach. We observe a smaller deviation in percentage deviations for the strongly correlated type instances compared to uncorrelated and weakly correlated ones. As an example, let us focus on the statistics of ADP-N for the problems of size $m = 100, n = 1,000$ with $s \in G_1$ as presented in Table 4.18. Even though, the minimum (2.00), average (4.80) and maximum (10.96) computation times deviate modestly, the minimum (2.9938), average (4.0020), and

maximum (5.5049) percentage deviations did not deviate *considerably* from one another. This observation still suggests that ADP-P and ADP-N need extra tuning to determine the best sample size. We continue to observe robustness across different types of problem instances. For instance, for $s \in G_1$, the average percentage deviations of ADP-N (i.e., average of PE(ADP-N) values from Tables 4.14, 4.16 and 4.18) are 11.12, 11.87 and 7.13 for uncorrelated, weakly correlated and strongly correlated type instances, respectively. Similarly, for $s \in G_1$, the average computation times of ADP-N (i.e., average of T(ADP-N) values from Tables 4.14, 4.16 and 4.18) are 1.31, 1.45 and 2.00 for uncorrelated, weakly correlated and strongly correlated type instances, respectively.

		T(ADP-P)			PE(ADP-P)			T(ADP-N)			PE(ADP-N)		
m	n	min	avg	max	min	avg	max	min	avg	max	min	avg	max
10	100	0.03	0.06	0.11	5.8559	8.6297	11.7666	0.02	0.05	0.008	5.0795	8.7346	13.5535
50	100	0.10	0.19	0.30	7.5812	10.3166	12.0993	0.09	0.18	0.28	7.6882	10.4663	13.6747
100	500	0.95	2.91	8.40	4.4200	5.3768	6.4244	0.96	2.95	8.47	4.1320	5.3237	6.6767
100	1000	1.99	4.92	12.11	2.8619	3.8747	5.3549	2.00	4.80	10.96	2.9938	4.0020	5.5049

Table 4.18: ADP-P, ADP-N: Strongly-correlated type instances, $s \in G_1$.

		T(ADP-P)			PE(ADP-P)			T(ADP-N)			PE(ADP-N)		
m	n	min	avg	max	min	avg	max	min	avg	max	min	avg	max
10	100	0.22	0.79	1.50	5.8559	8.6297	11.7666	0.17	0.58	1.05	5.8589	10.1842	13.5970
50	100	0.68	1.91	3.55	7.5812	10.3166	12.0993	0.66	1.62	2.87	8.3427	11.3239	14.8340
100	500	8.65	16.44	26.15	4.4200	5.3768	6.4244	9.67	17.17	27.39	4.3071	5.5721	7.4115
100	1000	18.25	31.57	51.35	2.8619	3.8747	5.3549	19.74	33.38	54.15	2.8814	4.0093	5.3960

Table 4.19: ADP-P, ADP-N: Strongly-correlated type instances, $s \in G_2$.

In summary, our computational study suggests that both ADP-P and ADP-N can be considered as alternative heuristics to some heuristic methodologies from the literature. Generally speaking, both ADP-P and ADP-N are capable of generating solutions at least as good as the ones obtained by heuristics H3, H4, H4 and H6. We note that ADP-P and ADP-N approaches need extra tuning in oder to determine the best sample size because the quality of solutions deviates substantially for varying

sample sizes. We also observe that an increase in the sample size does not reflect an improvement in the solution quality for both ADP-P and ADP-N. Since the larger the sample size the larger the computation time required by ADP-P and ADP-N, we should take into account the trade-off between solution quality and computation time once we use these approaches as heuristics for the multi-dimensional knapsack problems. As an interesting observation, average percentage deviations and computation times of ADP-P and ADP-N stayed stable across different types of instances which illustrates robustness across uncorrelated, weakly and strongly correlated type problem instances. Overall, adaptive fixing heuristic H2 is the leading heuristic compared to other heuristics, namely H1, H3, H4, H5, H6, ADP-P and ADP-N in terms of both solution quality and computation time.

4.5.3 Computational results for the base-heuristic learning framework

In this section, our basic aim is to illustrate the effectiveness of the approximate dynamic algorithm (ADP) with base-heuristic learning. We consider H1, H2, H3, H4, H5, H6 and B (CPLEX based branch-and-bound) as base-heuristics. We set `CPX-PARAM-EPAGAP` to 0.25 meaning that CPLEX terminates once the relative error, between the best solution and the best bound of the branch-and-bound tree, is below 0.25. We apply the base-heuristic learning framework to the same uncorrelated, weakly correlated and strongly correlated problem instances.

In Table 4.20 we report the computational results of the base-heuristic learning for uncorrelated type problems. This table suggests ADP with a certain base heuristic (ADP-H) improves the performance of the base-heuristic (H) meaning that ADP-H generates better quality of solutions than those obtained by H. As an example, for the problems of size $(m, n) \in \{(10, 100), (50, 100), (100, 500), (100, 1000)\}$, the average percentage deviations of H1 (i.e., PE(H1) values from Table 4.11) are 3.92,

9.59, 3.25 and 1.94, respectively, while the average percentage deviations of ADP-H1 (i.e., PE(ADP-H1) values from Table 4.20) are 1.63, 3.05, 0.70 and 0.40, respectively. This example illustrates that base-heuristic learning improves the solution quality of a base-heuristic significantly. Tables 4.11 and 4.20 indicate that ADP-H consumes longer computation time than H. This is mainly due to nature of the base-heuristic learning algorithm which calls a base-heuristic many times to approximate the optimal values. For instance, average computation times of H1 and ADP-H1 are 0.36 and 21.40, respectively. In order to delineate the observation that ADP-H improves the performance of a base-heuristic H, we present percentage improvements as presented in Table 4.21. ADP improves the performance of the proposed heuristics H1 and H2 substantially. As an example, let us focus on the problem instances of size $m = 50, n = 100$, we note that average percentage deviations of H1 and ADP-H1 are 9.59 and 3.05, respectively, which yields an percentage improvement of 65.22. We also observe that ADP improves the performance of heuristics H3, H4, H5 and H6, particularly for the problem instances of size $(m, n) \in \{(10, 100), (50, 100)\}$.

		ADP-H1		ADP-H2		ADP-H3		ADP-H4		ADP-H5		ADP-H6	
m	n	T	PE	T	PE	T	PE	T	PE	T	PE	T	PE
10	100	0.47	1.63	0.74	0.89	0.51	20.95	0.27	21.14	5.88	21.29	0.94	20.54
50	100	2.56	3.05	6.12	1.33	2.07	18.80	1.14	20.45	38.17	21.75	4.31	18.60
100	500	23.88	0.70	58.23	0.32	20.14	32.73	13.94	33.00	1,151.20	33.74	32.26	32.14
100	1000	58.70	0.40	163.11	0.17	81.62	33.14	63.21	33.27	10,540.60	33.26	122.56	32.98

Table 4.20: ADP base-heuristics: Uncorrelated type instances.

		PI(H)					
m	n	H1	H2	H3	H4	H5	H6
10	100	58.26	34.00	40.23	44.63	36.82	39.82
50	100	65.22	31.08	39.12	45.26	32.27	37.06
100	500	77.53	24.71	7.04	8.21	6.38	6.83
100	1000	79.39	32.32	3.89	4.03	4.06	3.78

Table 4.21: Percentage improvements : Uncorrelated type instances.

Table 4.22 presents results for CPLEX based branch-and-bound heuristic B and ADP-B. ADP-B attains better quality of solutions than B at an additional computation time. Overall, heuristic B generated superior quality of solutions compared to those obtained by ADP-P, ADP-N, H3, H4, H5 and H6 in modest computation times. Heuristics H1 and H2 compete successfully with B and ADP-B in terms of both solution quality and computation time. For instance, average percentage deviation of H2 is 1.14 with an average computation time of 0.93 while ADP-B attains an average percentage deviation of 3.42 with an average computation time of 150.23. Tables 4.20 and 4.22 illustrate that both ADP-H1 and ADP-H2 generate higher quality of solutions in shorter computation time than ADP-B.

		B		ADP-B		PI(B)
m	n	T	PE	T	PE	
10	100	0.06	9.4885	7.90	2.2830	64.51
50	100	0.19	12.8548	23.60	4.6875	60.36
100	500	5.59	5.3854	128.10	3.3762	33.27
100	1000	24.34	4.2522	441.30	3.3500	18.73

Table 4.22: ADP-B: Uncorrelated type instances, (CPX-PARAM-EPAGAP=0.25).

Overall, for uncorrelated type problem instances of varying sizes, ADP with adaptive fixing heuristic (ADP-H2) generated the best quality of solutions in modest computation times.

In Table 4.23 we report the computational results of the base-heuristic learning for weakly correlated type problems. ADP with a certain base heuristic (ADP-H) continues to improve the performance of the base-heuristic (H) with an additional computation time. We observe that quality of solutions generated by ADP-H1 degrades for weakly correlated type instances in comparison to uncorrelated type ones. The average percentage deviations are 1.45 and 5.49 for uncorrelated and weakly correlated type instances, respectively. This result partially depends on the performance

of the base-heuristic H1. As observed earlier, H1 generated considerably worse quality of solutions to weakly correlated type problems. ADP-H2 still generates good solutions with an average percentage deviation of 1.11 in reasonably modest but longer computation times compared to uncorrelated type instances. This is due to the nature of base-heuristic learning (i.e., many calls for the underlying base-heuristic H2) and longer computation times needed to obtain a solution by H2 for the weakly correlated type instances. Overall, both ADP-H1 and ADP-H2 generate higher quality of solutions than those obtained by ADP-H3, ADP-H4, ADP-H5 and ADP-H6. Average of percentage deviations of ADP-H3, ADP-H4, ADP-H5 and ADP-H6 are 13.26, 14.40, 13.42 and 13.43, respectively. We observe that computation times of ADP-H3, ADP-H4, ADP-H5 and ADP-H6 stayed reasonably stable. Table 4.24 suggests that, through percentage improvement statistics, ADP improves significantly over a base-heuristic. We observe that percentage improvement of heuristics are similar both for the uncorrelated type problems and the weakly correlated type problems as we compare Tables 4.21 and 4.24. Indeed, for uncorrelated type problems, average percentage improvements of heuristics H1, H2, H3, H4, H5 and H6 are 70.01, 30.52, 22.57, 25.53, 19.88 and 21.87, respectively, while for weakly correlated type problems, average percentage improvements of heuristics H1, H2, H3, H4, H5 and H6 are 60.59, 31.46, 20.98, 22.28, 20.91 and 17.41, respectively.

		ADP-H1		ADP-H2		ADP-H3		ADP-H4		ADP-H5		ADP-H6	
m	n	T	PE	T	PE	T	PE	T	PE	T	PE	T	PE
10	100	0.67	4.90	1.25	1.19	0.60	10.64	0.40	12.00	6.23	10.57	0.83	11.65
50	100	7.54	12.88	27.38	2.29	2.63	12.36	1.81	14.07	39.28	12.29	3.84	12.02
100	500	60.65	2.71	239.67	0.61	17.71	15.18	12.07	15.95	901.60	15.88	24.72	15.42
100	1000	118.00	1.48	513.88	0.34	70.86	14.85	46.56	15.59	8,805.60	14.94	102.11	14.64

Table 4.23: ADP base-heuristics: Weakly-correlated type instances.

Table 4.25 presents results for CPLEX based branch-and-bound heuristic B and ADP-B. ADP-B continues to improve the performance of B at an additional computation time. Overall, heuristic B attains superior quality of solutions compared to those obtained by ADP-P, ADP-N, H3, H4, H5 and H6. In comparison

		PI(H)					
m	n	H1	H2	H3	H4	H5	H6
10	100	41.06	37.31	39.38	39.84	44.32	29.73
50	100	51.00	34.64	28.09	35.10	26.04	26.82
100	500	73.29	27.82	9.84	7.12	9.32	7.06
100	1000	77.02	26.07	6.62	7.09	3.94	6.02

Table 4.24: Percentage improvements: Weakly-correlated type

of H1 to B, B generated higher quality of solutions except for the problem instances of size $m = 10, n = 100$. ADP-H1 obtains higher quality of solutions with shorter computation times than those obtained by ADP-B for the problem instances of size $(m, n) \in \{(100, 500), (100, 1,000)\}$ while for the problem instances of size $(m, n) \in \{(10, 100), (50, 100)\}$ we observe even though ADP-B dominates in terms of solution quality it consumes considerably longer computation time than ADP-H1. Adaptive fixing heuristic H2 continues to compete successfully with B and ADP-B in terms of both solution quality and computation time (see Tables 4.12 and 4.25). For instance, average percentage deviation of H2 is 1.81 with an average computation time of 2.65 while ADP-B attains an average percentage deviation of 4.21 with an average computation time of 472.55. Tables 4.23 and 4.25 illustrate that ADP-H2 generates significantly better quality of solutions, with an average percentage deviation of 0.89, in shorter times, with an average computation time of 195.55, than ADP-B.

		B		ADP-B		PI(B)
m	n	T	PE	T	PE	
10	100	0.10	12.6130	8.10	3.6248	66.38
50	100	0.36	10.9259	36.00	6.2559	38.20
100	500	18.96	4.2001	391.30	3.5820	13.54
100	1000	118.84	4.0331	1,454.80	3.3793	15.50

Table 4.25: ADP-B: Weakly-correlated type (CPX-PARAM-EPAGAP=0.25).

Overall, for weakly correlated type problems, ADP with adaptive fixing heuris-

tic (ADP-H2) remains to be the leading methodology in comparison to other ADP methodologies in terms both solution quality and computation time.

In Table 4.26 we report the computational results of the base-heuristic learning for strongly correlated type problems. We observe that ADP with a certain base heuristic (ADP-H) continues to improve the performance of the base-heuristic (H) with an additional computation time. ADP-H1 generated similar quality of solutions for strongly correlated type instances compared to weakly correlated instances. The average percentage deviations of ADP-H1 are 5.49 and 5.40 for weakly and strongly correlated type instances, respectively. ADP-H2 generates good solutions with an average percentage deviation of 1.51. Both ADP-H1 and ADP-H2 require longer computation time due to the nature of base-heuristic learning (i.e., many calls for the underlying base-heuristic, H1 or H2) and longer computation times needed for problem instances with higher correlation (i.e., strongly correlated vs. uncorrelated or weakly correlated). For instance, average computation times of ADP-H2 are 57.05, 195.42 and 538.54 for uncorrelated, weakly and strongly correlated type instances, respectively. Overall, both ADP-H1 and ADP-H2 generate higher quality of solutions than those obtained by ADP-H3, ADP-H4, ADP-H5 and ADP-H6. Indeed, average of percentage deviations of ADP-H3, ADP-H4, ADP-H5 and ADP-H6 are 5.69, 6.95, 6.02 and 5.78, respectively, while these ADP approaches achieved their best performance in terms of solution quality. We continue to observe that average computation times of ADP-H3, ADP-H4, ADP-H5 and ADP-H6 stayed reasonably stable. Table 4.27 continues to suggest that ADP improves often significantly over a base-heuristic. Average percentage improvements of heuristics H1, H2, H3, H4, H5 and H6 are 71.15, 29.73, 23.76, 26.07, 23.22 and 20.83, respectively. Indeed, for uncorrelated type problems, average percentage improvements of heuristics H1, H2, H3, H4, H5 and H6 are 70.01, 30.52, 22.57, 25.53, 19.88 and 21.87, respectively, while for weakly correlated type problems, average percentage improvements of heuristics H1, H2, H3, H4, H5 and H6 are 60.59, 31.46, 20.98, 22.28, 20.91 and 17.41, respectively.

		ADP-H1		ADP-H2		ADP-H3		ADP-H4		ADP-H5		ADP-H6	
m	n	T	PE	T	PE	T	PE	T	PE	T	PE	T	PE
10	100	0.80	5.75	2.25	1.48	0.61	5.28	0.38	6.98	6.15	5.93	0.78	5.72
50	100	10.37	10.98	69.20	3.13	2.59	7.83	1.70	8.85	39.11	7.99	3.21	7.79
100	500	110.59	3.21	814.32	0.95	23.60	5.41	18.21	6.63	1,267.80	5.69	30.10	5.32
100	1000	169.31	1.67	1,268.40	0.47	83.65	4.27	50.12	5.35	8,311.20	4.49	106.39	4.30

Table 4.26: ADP base-heuristics: Strongly-correlated type

		PI(H)					
m	n	H1	H2	H3	H4	H5	H6
10	100	39.65	49.21	39.92	47.39	40.27	39.89
50	100	77.82	24.76	27.23	35.87	28.20	23.18
100	500	83.81	24.86	16.47	12.08	13.05	6.31
100	1000	83.30	20.09	11.40	8.94	11.36	13.93

Table 4.27: Percentage improvements: Strongly-correlated type

Table 4.28 presents results for CPLEX based branch-and-bound heuristic B and ADP-B. ADP-B continues to improve the performance of B at an additional computation time. Overall, heuristic B attains superior quality of solutions compared to those obtained by ADP-P, ADP-N, H3, H4, H5 and H6. In comparison of H1 to B, B generated higher quality of solutions for all the problem instances. Average percentages of H1 and B are 22.53 and 6.19, respectively, while corresponding average computation times are 3.89 and 85.59, respectively. ADP-H1 obtains an average percentage deviation of 5.40 with an average computation time of 72.77. This is an interesting observation, since base-heuristic learning improves the performance of a base-heuristic (e.g., H1) so that it becomes competitively comparable to another heuristic (e.g., B) which originally generates better quality of solutions than the corresponding base-heuristic (i.e., B generates higher quality of solutions than those obtained by H1). In comparison of ADP-H1 to ADP-B, ADP-H1 obtains higher quality of solutions with shorter computation times than those obtained by ADP-B for the problem instances of size $(m, n) \in \{(100, 500), (100, 1,000)\}$, while for the problem instances of size $(m, n) \in \{(10, 100), (50, 100)\}$, we observe that, even though ADP-B dominates in terms of solution quality it consumes significantly longer computation

time than ADP-H1. Average computation times of ADP-H1 and ADP-B are 72.77 and 901.35, respectively (summary statistics from Tables 4.28 and 4.28). Adaptive fixing heuristic H2 continues to compete successfully with B and ADP-B in terms of both solution quality and computation time (see Tables 4.13 and 4.28). For instance, average percentage deviation of H2 is 2.50 with an average computation time of 8.55 while ADP-B attains an average percentage deviation of 3.31 with an average computation time of 901.35. Tables 4.26 and 4.28 illustrate that ADP-H2 generates better quality of solutions, with an average percentage deviation of 1.51, in shorter times, with an average computation time of 538.54, than ADP-B.

		B		ADP-B		PI(B)
m	n	T	PE	T	PE	
10	100	0.10	7.3215	8.25	3.5326	45.13
50	100	0.63	9.6962	47.21	5.2003	42.16
100	500	65.27	4.3011	652.95	2.5641	34.97
100	1000	276.36	3.4242	2,897.00	1.9601	38.25

Table 4.28: ADP-B: Strongly-correlated type. (CPX-PARAM-EPAGAP=0.25)

Overall, for strongly correlated type problems, ADP with adaptive fixing heuristic (ADP-H2) remains to be the leading methodology in comparison to other ADP approaches.

Our important observation is that ADP with base-heuristic learning (ADP-H) improves the performance of a given base-heuristic (H) often significantly. Base-heuristic learning provides a framework that exploits a certain base-heuristic to generate higher quality of solutions than the quality of solutions obtained by the corresponding base-heuristic. We measure the corresponding enhancement in the quality of solutions through percentage improvements as provided in Tables 4.21, 4.24 and 4.27 for uncorrelated, weakly and strongly correlated type instances, respectively. Our available computational study illustrates that ADP with adaptive fixing heuristic ADP-H2 is the most promising methodology in terms both solution quality and computa-

tion time. Interestingly enough, ADP-H2 is a robust methodology based on solution quality criterion across different types of problem instances meaning that quality of solutions obtained by ADP-H2 are competitively comparable for uncorrelated, weakly and strongly correlated type instances, respectively. As an example, for the problems of size $m = 100, n = 1,000$, average percentage deviations of ADP-H2 are 0.17, 0.34 and 0.47 for uncorrelated, weakly and strongly correlated type problem instances, respectively.

4.5.4 Test problems from the literature

In this section, our aim is to compare the most promising ADP methodology with one of the best heuristics in the recent literature for the multi-dimensional knapsack problems. Our previous computational study suggests that ADP-H2, i.e., base-heuristic learning with *adaptive fixing heuristic*, is the best algorithm in terms of solution quality and computation time. In recent research, Chu and Beasley [15] developed a genetic algorithm (GA) to solve multi-dimensional knapsack problems. Their computational study shows that GA provides very good solutions at modest computation time to some large multi-dimensional knapsack problems. In addition, to our knowledge, they attempted to solve the largest instances in the literature. Their problem generation scheme is as follows: $a_{ij} \sim U(1, 1,000)$, $c_j = \sum_i a_{ij}/m + 500q_j$ where $q_j \sim U(0, 1)$. For each $(m, n) \in \{(30, 250), (30, 500)\}$ combination, the right hand side coefficients (b_i's) were set using $b_i = \tau \sum_{j=1}^{n} a_{ij}$ where τ is a tightness ratio. They use $\tau = 0.25, 0.50,$ and 0.75. Their computational study is done on a Silicon Indigo workstation which is supposed to be two times slower than our machine Dell Precision 410. Thus, in order to make a legitimate comparison we scale their computation times by dividing the original times presented in their paper by two.

Table 4.29 compares ADP-H2 with the genetic algorithm proposed by Chu and Beasley [15] on their test bed, i.e, on their problem instances. This table illustrates H2 generated good solutions while ADP-H2 enhanced its performance to obtain very

good solutions at an additional computation time. We observe that even though GA provides *slightly* better solutions than ADP-H2, its average computational time requirement is substantially larger than the average computation time of ADP-H2. Indeed, average computation times of GA and ADP-H2 are 1,287.15 and 156.48, respectively, which says that, on average, GA is approximately 8 times slower than ADP-H2 for the same multi-dimensional knapsack problem instances.

			H2		ADP-H2		PI(H2)	GA	
m	n	τ	T	PE	T	PE		T	PE
30	250	0.25	0.21	2.73	69.64	1.61	39.39	749.75	1.19
		0.50	0.19	1.31	63.62	0.69	44.59	990.03	0.53
		0.75	0.19	0.76	55.51	0.58	22.15	1,220.70	0.61
30	500	0.25	0.40	1.39	284.73	0.98	25.50	1,218.85	0.61
		0.50	0.41	0.65	241.47	0.43	28.41	1,599.45	0.26
		0.75	0.39	0.35	223.90	0.29	15.07	1,944.10	0.17

Table 4.29: Comparison of ADP-H2 with Chu and Beasley's data and genetic algorithm.

Table 4.30 reports computational results for ADP-H2 with lag policy assignment. ADP with lag policy assignment is an ADP methodology where solution construction is expedited as described in Section 4.4 through `lag-time` parameter. In this computational study, we set `lag-time`=100. Table 4.30 supports our expectation so that ADP-H2 with lag policy finds solutions at shorter times than ADP-H2 with a *small decline* in the solution quality. Indeed average computation time and percentage deviation of ADP-H2 with lag policy (summary statistics from Table 4.30) are 44.77 and 0.80, respectively. In comparison to this, average computation time and percentage deviation of ADP-H2 (summary statistics from Table 4.29) are 156.48 and 0.76, respectively. Thus, ADP-H2 with lag policy runs approximately 4 times faster than ADP-H2 to get comparably high quality of solutions for the test problems.

We pointed out in section 4.4 the larger the `lag-time` values the better the solutions obtained by an ADP methodology with a lag policy assignment. In order

			H2		ADP-H2		PI(H2)	GA	
m	n	τ	T	PE	T	PE		T	PE
30	250	0.25	0.21	2.73	25.80	1.74	34.87	749.75	1.19
		0.50	0.19	1.32	23.44	0.82	35.05	990.03	0.53
		0.75	0.20	0.76	20.39	0.53	26.98	1,220.70	0.61
30	500	0.25	0.39	1.39	74.23	0.97	25.03	1,218.85	0.61
		0.50	0.40	0.65	63.30	0.46	24.19	1,599.45	0.26
		0.75	0.39	0.35	61.48	0.30	14.60	1,944.10	0.17

Table 4.30: Comparison of ADP-H2 with lag policy (lag-time=100) with Chu and Beasley's GA.

to illustrate this tendency we set `lag-time`=200. Table 4.31 supports our expectations because ADP (`lag-time`=200) generated better quality of solutions than those obtained by ADP (`lag-time`=100). Indeed, average percentage deviations of ADP (`lag-time`=200) and ADP (`lag-time`=100) are 0.74 and 0.80, respectively. ADP (`lag-time`=200) achieves higher quality of solutions in an average of 59.92 CPU seconds compared to the average time 44.77 CPU seconds of ADP (`lag-time`=100). As an interesting observation, ADP (`lag-time`=200) achieves, on average, higher quality of solutions in shorter times than ADP without a lag policy whose average percentage deviation and computation time are 0.76 and 156.48, respectively. This observation indicates that ADP without lag policy does not necessarily generate higher quality of solutions. On the other extreme, we know that ADP without lag policy consumes more computation time compared to ADP with lag policy.

Chu and Beasley [15] compared their GA methodology with the heuristic of Magazine and Oguz (MKNAP) [47] (as described briefly in Section 4.1), the heuristic of Volgenant and Zoon (VZ) [86] and the heuristic of Pirkul (P-SU) [65] (as described briefly in section 4.1) on their test bed. For all these algorithms, since the original codes were not available to them, they coded the algorithms themselves based on the descriptions of the methods outlined in the corresponding papers. We provide the corresponding comparative study in Table 4.32 (information extracted from Table 3

			H2		ADP-H2		PI(H2)	GA	
m	n	τ	T	PE	T	PE		T	PE
30	250	0.25	0.20	2.73	31.68	1.64	38.51	749.75	1.19
		0.50	0.19	1.32	29.71	0.72	42.22	990.03	0.53
		0.75	0.20	0.76	26.20	0.54	26.62	1,220.70	0.61
30	500	0.25	0.40	1.39	99.90	0.81	37.45	1,218.85	0.61
		0.50	0.40	0.65	87.97	0.45	25.77	1,599.45	0.26
		0.75	0.39	0.35	84.08	0.29	15.09	1,944.10	0.17

Table 4.31: Comparison of ADP-H2 with lag policy (lag-time=200) with Chu and Beasley's GA.

on page 19 in the paper by Chu and Beasley [15]). This table clearly indicates that the superiority of their GA over those other heuristic methods in terms of the quality of the solutions obtained. Chu and Beasley did not report computation time of those heuristics.

			PE(H)			
m	n	τ	MKNAP	VZ	P-SU	GA
30	250	0.25	13.54	12.41	3.70	1.19
		0.50	8.64	7.12	1.53	0.53
		0.75	4.49	3.91	0.84	0.31
30	500	0.25	9.84	9.62	1.89	0.61
		0.50	7.10	5.71	0.73	0.26
		0.75	3.72	3.51	0.48	0.17

Table 4.32: Comparison of GA with other heuristic methods from the literature.

We can make an *indirect* comparison between our proposed methodologies and MKNAP, VZ, P-SU. Tables 4.29 suggests that the proposed stand-alone heuristic, namely *adaptive fixing* heuristic, generates higher quality of solutions than those obtained by MKNAP, VZ and P-SU. Average percentage deviations of MKNAP, VZ and P-SU are 7.89, 7.05 and 1.53, respectively, while H2 attains an average percentage deviation of 0.87. We also observe that, for certain m, n, τ values, H2 achieved smaller average percentage deviation than the heuristics MKNAP, VZ and P-SU. We

already observed that both ADP-H2 and ADP-H2 with lag policy generated higher quality of solutions than H2 (see Tables 4.29, 4.30 and 4.31). Thus, both ADP-H2 and ADP-H2 with lag policy automatically achieve better quality of solutions than those obtained by the heuristics MKNAP, VZ and P-SU.

Overall, the proposed stand-alone *adaptive fixing* heuristic (H2) and ADP with adaptive fixing (ADP-H2) generated very good solutions to some of the test problems from the literature. Through our direct comparison, we illustrate that ADP-H2 is a practical methodology that generates very comparable solutions to the GA methodology but much faster. Through our indirect comparison, we partially demonstrate that both H2 and ADP-H2 obtain better quality of solutions compared to those obtained by some of well-known heuristics in the literature.

4.5.5 Comparison of the most promising ADP algorithm with CPLEX

In this section, our aim is to make a comparative study between the most promising ADP algorithm (ADP-H2), namely ADP with *adaptive fixing heuristic*, and one of state-of-art commercial packages like CPLEX on the same randomly generated problem instances.

In Tables 4.33, 4.34 and 4.35 we compare ADP-H2 with CPLEX. In order to make a legitimate comparison we set CPLEX tolerance `CPX-PARAM-EPGAP` to PE(ADP-H2)/100 where PE(ADP-H2) denotes the percentage deviation of the objective value of the solution obtained by ADP-H2 from the LP objective value. We also set tree memory limit and node file size limit of CPLEX to 250 MB. In addition, we set CPLEX time limit to 6,000 CPU seconds. CPLEX terminates when one of the binding parameters is satisfied. Once CPLEX terminates, we retrieve its solution and calculate the percentage deviation of the objective value of the corresponding solution from the LP objective value. We also report its average CPU time and number

of nodes to analyze the performance of CPLEX. Overall, these tables suggest that CPLEX could not find a feasible solution that matches PE(ADP-H2) within the time-limit of 6,000 CPU seconds and memory limit of 250 MB except for the uncorrelated type problem instances of size $m = 10, n = 100$.

Tables 4.33, 4.34 and 4.35 illustrate that ADP-H2 does have a robust performance. For a certain m, n combination, we observe that both the minimum, average, and maximum computation times and the minimum, average, and maximum percentage deviations have stable values signifying that ADP-H2 is robust in terms of both solution quality and computation time. As an example, let us focus on strongly correlated type problem instances of size $m = 100, n = 1,000$ (Table 4.35), we note that the minimum (1,086.36), average (1,222.69) and maximum (1,323.54) computation times are close to one another and also the minimum (0.4357), average (0.4651) and maximum (0.4953) percentage deviations have near values from one another.

		CPLEX			T(ADP-H2)			PE(ADP-H2)		
m	n	avg. time	avg. no nodes	PE	min	avg	max	min	avg	max
10	100	12.20	13,722.40	0.5796	0.16	0.72	1.21	0.4804	0.8947	1.1321
50	100	100.40	53,161.90	1.6264	0.93	5.99	10.23	0.8528	1.3285	1.8455
100	500	time-limit	480,048.00	3.3196	13.64	58.98	89.99	0.2101	0.3215	0.4880
100	1000	time-limit	195,550.80	3.6750	115.45	154.482	192.74	0.1424	0.1710	0.2166

Table 4.33: Comparison of ADP-H2 with CPLEX. Unorrelated type problem instances.

		CPLEX			T(ADP-H2)			PE(ADP-H2)		
m	n	avg. time	avg. no nodes	PE	min	avg	max	min	avg	max
10	100	1,394.80	1,312,296.80	1.7911	0.07	1.26	2.49	0.7064	1.1919	1.5768
50	100	4,324.30	977,778.50	3.2313	4.91	27.37	36.68	1.2837	2.2910	3.1162
100	500	time-limit	141,415.30	3.9888	56.09	238.79	335.12	0.3138	0.6091	0.7491
100	1000	time-limit	50,164.90	3.9381	137.03	520.54	732.07	0.1431	0.3361	0.4143

Table 4.34: Comparison of ADP-H2 with CPLEX. Weakly-correlated type problem instances.

		CPLEX			T(ADP-H2)			PE(ADP-H2)		
m	n	avg. time	avg. no nodes	PE	min	avg	max	min	avg	max
10	100	1,320.00	1,123,245.80	1.7442	0.04	2.27	2.73	1.0694	1.4825	2.0281
50	100	time-limit	819,612.50	5.7340	62.98	69.74	76.18	2.9389	3.1329	3.4859
100	500	time-limit	46,111.10	4.3003	692.50	774.21	891.98	0.8633	0.9513	1.0342
100	1000	time-limit	20,090.40	2.8485	1,086.36	1,222.69	1,323.54	0.4357	0.4651	0.4953

Table 4.35: Comparison of ADP-H2 with CPLEX. Strongly-correlated type problem instances.

4.6 Conclusions

We developed an approximate dynamic programming methodology for the multi-dimensional knapsack problem. We describe both statistical and base-heuristic learning. The statistical learning framework produced good quality of solutions but weaker than those achieved by the base-heuristic learning framework. Our overall computational evidence suggests that base-heuristic learning framework generates near-optimal solutions with short computation times and robust performances. Another interesting observation is that base-heuristic learning provides (ADP-H) a framework that improves the performance of a given base-heuristic (H) meaning that ADP-H generates better quality of solutions than H. In summary, our extensive computational study supports our belief that approximate dynamic programming can be considered as an attractive alternative to existing methodologies.

Chapter 5

Binary Integer Programming

This chapter addresses the binary integer programming problem. In Section 5.1, we provide a brief literature review while giving emphasis on heuristic methodologies. In Section 5.2, we customize the base-heuristic learning framework to the binary integer programming problem. In Section 5.3, we present an extensive computational study and summarize the insights we obtained.

5.1 Background

Many questions in practice can be formulated as integer programming (IP) problems, in which we are looking for a vector $\mathbf{x}^* = (x_1^*, \cdots, x_n^*)$ such that an objective function

$$z = \sum_{j=1}^{n} c_j x_j$$

takes the maximal value for $\mathbf{x} = \mathbf{x}^*$ subject to constraints

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_{0,i}, \ i = 1, \cdots, m \tag{5.1}$$

$$x_j \in Z^+, \ j = 1, \cdots, n.$$

If all variables are binary (0-1), Problem (5.1) is called binary (0-1) integer programming problem. Binary integer programming is perhaps the most important subclass of integer programming problems. Any bounded integer program can be transformed to binary integer program. Thus, in theory, a methodology that is capable of solving binary integer programs can be utilized to solve generic bounded integer programs. We formulate the binary integer programming problem (BIP) with n variables and a right-hand-side $\mathbf{b}_0$ as follows follows:

$$
\begin{aligned}
z_{BIP} = \quad & \text{maximize} \quad \sum_{j=1}^{n} c_j x_j \\
& \text{subject to} \quad \sum_{j=1}^{n} \mathbf{A}_j x_j \leq \mathbf{b}_0 \\
& \qquad\qquad x_j \in \{0,1\}, \qquad j = 1, \cdots, n,
\end{aligned}
$$

assuming that $\mathbf{A}_j = (a_{1j}, \cdots, a_{mj})'$ denotes the column vector of the variable j and $\mathbf{b}_0 = (b_{0,1}, \cdots, b_{0,m})'$ denotes the right-hand-side of the problem. We define by z_{BIP} the optimal value of the binary integer programming problem.

Since many real-world problems can be represented as integer programs, the literature on integer programming is very large. Several cutting plane, branch-and-bound, branch-and-cut algorithms have been developed both as special algorithms for specific IPs as well as general-purpose algorithms. Nemhauser and Wolsey [61], Salkin and Mathur [73], Wolsey [91], Walukiewicz [87], Schrijver [75] provide an extensive review of the developments in the field.

Due to the challenge of finding an optimal solution especially for large-scale binary integer programming problems, heuristic methodologies have been offered to solve these problems. Most heuristics for integer programming are dedicated to a *specific* problem (e.g. set covering, scheduling, location etc.) and often succeed in finding high quality solutions fast. Perhaps surprisingly, only few efforts have been

made to devise heuristics that target a wider range of discrete optimization problems. We review these methods next.

5.1.1 General purpose heuristic methods

Balas and Martin [3] devised a linear programming based heuristic for pure binary integer programming problems. The proposed heuristic exploits the fact that an optimal solution to a binary integer programming problem can be found at one of the extreme points of the linear programming relaxation of the corresponding problem. Suppose the linear programming relaxation of BIP is solved by an upper-bounded simplex algorithm (see Bertsimas and Tsitsiklis [12]). If all structural variables are nonbasic at optimality or, equivalently, all slack variables are in the basis, then the LP solution is integral. In essence, their proposed heuristic tries to *pivot in* all slack variables into basis at a minimal change in objective function value of the linear programming relaxation of the BIP. Once all the slack variables are in the basis, the heuristic stops pivoting meaning that a feasible solution is available. The heuristic then tries to improve the current feasible solution by some *complementing* techniques. Balas and Martin called this linear programming based heuristic as *pivot-and-complement* (BM-PC) mainly due to *pivoting* and *complementing* operations. Their computational study suggests that BM-PC generates good solutions for randomly generated problem instances with small coefficients. BM-PC is tested on real-world problems particularly on set-covering problems with promising results. Aboudi and Jornsten [1] embedded the pivot-and-complement heuristic into a tabu search framework in the hope that this framework improves the performance of the stand-alone application of the pivot-and-complement heuristic. They provide a computational study on multi-dimensional knapsack problems, on special set-covering problems from MIPLIB test problems (Bixby et. al. [13]) and problems from Crowder et. al. [18]. Their computational evidence suggests that the success of their methodology depends mainly on the performance of Balas and Martin's pivot-and-complement heuristic.

Balas et. al. [2] proposed a heuristic, called OCTANE, for pure binary programming problems, which finds feasible solutions by enumerating extended facets of the octahedron, the outer polar of the unit hypercube. Their computational study on MIPLIB test problems (Bixby et. al. [13]) suggests that OCTANE is an efficient and robust heuristic for a variety of pure binary integer programs with different structures. Løkketangen and Glover [44] describe a tabu search approach for solving general zero-one integer programming problems that exploits the extreme point property of zero-one solutions meaning that an optimal solution for the binary integer programming problem may be found at an extreme point of the linear programming feasible set. Although their approach is designed to solve thoroughly general 0-1 integer programming problems they only provide computational results on a portfolio of multi-dimensional knapsack problems with promising results. Thus, it is hard to assess the success of this methodology for the binary integer programming problems.

We should note that any *exact* methodology that is devised to solve binary integer programming problems can be exploited as a heuristic methodology once some of the early termination criteria are satisfied. For instance, a branch-and-bound methodology can be terminated as soon as it attains a feasible solution.

5.2 Base-heuristic Learning

In this section we focus on base-heuristic learning framework for the binary integer programming problems. We first introduce a *base-heuristic learning algorithm* for the binary integer programming problem. We then describe base-heuristics that will be used to approximate the optimal values.

5.2.1 A Base-Heuristic Learning Algorithm

We adapt and enhance the generic base-heuristic learning framework described in Table 2.2 to the binary integer programming problem as in the case of the multi-dimensional knapsack problem (see Section 4.4 on page 98). Let $BH(k, \mathbf{b})$ be a base-heuristic for the binary integer programming subproblem $BIP(k, \mathbf{b})$ with k variables and a right-hand-side $\mathbf{b}$. We denote by $F_k(\mathbf{b})$ the optimal solution of the $BIP(k, \mathbf{b})$. Let $x_{BH}(k, \mathbf{b})$ be the corresponding heuristic solution and $H_k(\mathbf{b})$ be the heuristic value, i.e., an estimate of optimal value $F_k(\mathbf{b})$. We denote by $LP(k, \mathbf{b})$ the linear programming relaxation of the problem $BIP(k, \mathbf{b})$. Let $z^{LP(k,\mathbf{b})}$ be the corresponding optimal LP value. Let $U_k(\mathbf{b})$ be an upper bound to the optimal value $F_k(\mathbf{b})$ (e.g. $U_k(\mathbf{b}) = \lfloor z^{LP(k,\mathbf{b})} \rfloor$). We denote by $p^f(k, \mathbf{b})$ whether the subproblem $BIP(k, \mathbf{b})$ is *feasible* or not. We set $p^f(k, \mathbf{b}) = 1$ (0) if the problem is feasible (or not). Let us denote by $s^o(k, \mathbf{b})$ whether $x_{BH}(k, \mathbf{b})$ is *optimal* solution or not to the subproblem $BIP(k, \mathbf{b})$ (e.g. $x_{BH}(k, \mathbf{b})$ is optimal if $U_k(\mathbf{b}) \leq H_k(\mathbf{b})$). We set $s^o(k, \mathbf{b}) = 1$ (0) if the heuristic solution is optimal (not optimal).

Base-heuristic learning algorithm consists of two phases. In the first phase, we apply $BH(n, \mathbf{b}_0)$ to the problem $BIP(n, \mathbf{b}_0)$ and get $x_{BH}(n, \mathbf{b}_0), p^f(n, \mathbf{b}_0), s^o(n, \mathbf{b}_0)$. If the solution $x_{BH}(n, \mathbf{b}_0)$ is optimal, i.e., $s^o(n, \mathbf{b}_0) = 1$, the algorithm terminates with an optimal solution. If the problem is infeasible, i.e., $p^f(n, \mathbf{b}_0) = 0$, the algorithm terminates without a solution. Under the assumption that the problem is feasible we set *best-solution* known $\mathbf{x}^{best} = x_{BH}(n, \mathbf{b}_0)$ and *best value* known $z^{best} = H_n(\mathbf{b}_0)$. We proceed by applying *reduced cost fixing* as described below to fix some of the variables to the corresponding values (0 or 1) in an *optimal* solution to the problem $BIP(n, \mathbf{b}_0)$. Let us define by F the set of fixed variables. Reduced cost fixing might effectively reduce the number of variables in advance. To illustrate this, we assume for example that we apply reduced cost fixing to the problem $BIP(n, \mathbf{b}_0)$ and obtain $F = \{k, k+1, \cdots, n\}$ the set of fixed variables, i.e., we know the values (0 or 1) of $x_{k+1}, \cdots, x_n$ in an optimal solution. This implies that solving $BIP(n, \mathbf{b}_0)$ is equiva-

lent to solving $BIP(k-1, \mathbf{b}_0 - \sum_{j \in F} x_j)$.

Reduced-cost fixing:

We obtain linear programming relaxation (LP) statistics of the problem $BIP(n, \mathbf{b}_0)$ such as the optimal LP solution $\mathbf{x}^{LP}$, the optimal LP objective value z^{LP} and the reduced cost vector $\overline{\mathbf{c}} = (\overline{c}_1, \cdots, \overline{c}_n)$ where $\overline{c}_j$ denotes the reduced cost of the variable $j \in \{1, \cdots, n\}$ in the LP solution. Let us denote by $z^{current}$ a lower bound or a known solution value to the problem $BIP(n, \mathbf{b}_0)$ (e.g. heuristic value $H_n(\mathbf{b}_0)$). Reduced-cost fixing fixes nonbasic variable x_k at its LP value, i.e. $x_k = x_k^{LP}$, if $|\overline{c}_k| > z^{LP} - z^{current}$ (see Section 4.4.1 for a detailed description on page 99 or see Proposition 2.1 on page 389 in Nemhauser and Wolsey [61]). Once reduced-cost fixing is applied we update the fixed variable set F accordingly. We use $\mathbf{x}^f = (x_1^f, \cdots, x_n^f)$ to describe the status of the variables, i.e, fixed or not. We call a variable j is *fixed* if $x_j^f = 0$ or 1.

In the second phase, we construct an ADP solution $\mathbf{x}^{adp}$ and update the best solution $\mathbf{x}^{best}$ and its value z^{best} accordingly. We iteratively assign x_j^{adp}, $j \notin F$ according to the *policy assignment* scheme as described below.

Policy assignment:

We can set x_k^{adp}, $k \notin F$ to 0 or 1 by,

$$x_k^{adp} = \text{ argmax }_{x \in \{0,1\}} \{H_{k-1}(\mathbf{b} - \mathbf{A}_k x) + c_k x\},$$

in a straightforward manner which will be called as the *standard policy scheme.* In order to enhance optimal value approximation, we exploit some performance statistics of base-heuristics $BH(k-1, \mathbf{b} - \mathbf{A}_k x)$ to the subproblems $BIP(k-1, \mathbf{b} - \mathbf{A}_k x)$ to recalculate the estimates $H_{k-1}(\mathbf{b} - \mathbf{A}_k x)$ for $x = 0, 1$. A typical performance statistic for the base-heuristic $BH(k, \mathbf{b})$ would be the percentage deviation of $H_k(\mathbf{b})$ from the upper bound $U_k(\mathbf{b})$ of the subproblem $BIP(k, \mathbf{b})$. In our computations, we use $U_k(\mathbf{b}) = \lfloor z^{LP(k,\mathbf{b})} \rfloor$. The expectation that the base heuristic should have a similar

percentage deviation from the upper bound motivates us to recalculate the estimate of the optimal values $F_{k-1}(\mathbf{b} - \mathbf{A}_k x)$. Let $\epsilon_x = \frac{U_{k-1}(\mathbf{b}-\mathbf{A}_k x)-H_{k-1}(\mathbf{b}-\mathbf{A}_k x)}{U_{k-1}(\mathbf{b}-\mathbf{A}_k x)}$ denote the relative error of $H_{k-1}(\mathbf{b} - \mathbf{A}_k x)$ with respect to $U_{k-1}(\mathbf{b} - \mathbf{A}_k x)$ for $x = 0, 1$. Our expectation is that base-heuristics $BH(k-1, \mathbf{b} - \mathbf{A}_k x)$ have the same level of relative error meaning that we calculate the minimum relative error $\epsilon^* = \min_{x\in\{0,1\}}\{\epsilon_x\}$ and recalculate the estimates of the optimal values as $H^u_{k-1}(\mathbf{b} - \mathbf{A}_k x) = (1 - \epsilon^*)U_{k-1}(\mathbf{b} - \mathbf{A}_k x)$ for $x = 0, 1$. Then we set x_k^{adp}, $k \notin F$ to 0 or 1 by,

$$x_k^{adp} = \text{ argmax }_{x\in\{0,1\}}\{H^u_{k-1}(\mathbf{b} - \mathbf{A}_k x) + c_k x\}.$$

We call this policy scheme as the *inferential policy scheme.* In contrast to multi-dimensional knapsack problems our preliminary computational study suggest using the standard policy scheme to calculate policies for the binary integer programming problems.

Update best known solution and value:

Once we set x_k^{adp}, $k \notin F$ to 0 or 1 we update the best solution $\mathbf{x}^{best}$ and value z^{best}. Let $x^* = x_k^{adp}$. Due to the nature of policy construction in the backward phase, a solution $x^{current} = (\mathbf{x}_{BH}(\mathbf{b}-\mathbf{A}_k x^*), x^*, x_{k+1}^{adp}, \cdots, x_n^{adp})$ with a value $z^{current} = H_{k-1}(\mathbf{b}-\mathbf{A}_k x^*) + c_k x^* + \sum_{k+1}^{n} c_j x_j^{adp}$ is available. We call $x^{current}$ as the *current solution* and $z^{current}$ as the *current solution value.* If $z^{current}$ is larger than z^{best} we update z^{best}, $\mathbf{x}^{best}$ as described in Table 4.1 on page 101.

We have several enhancements to speed up the approximate dynamic programming algorithm as introduced in Section 4.4: an early termination of algorithm through *ADP stopping condition* and variable fixing through *lag policy assignment* scheme. We adapt those enhancements to binary integer programming problems as follows.

ADP stopping condition:

Approximate dynamic programming can be terminated once we find an optimal solution to $BIP(k, \mathbf{b})$ in the backward phase. While setting x_k^{adp}, $k \notin F$ in the policy assignment we consider the subproblems $BIP(k-1, \mathbf{b} - A_k x)$, $x = 0, 1$ and we apply $BH(k-1, \mathbf{b}-\mathbf{A}_k x)$, $x = 0, 1$ and get $p^f(k-1, \mathbf{b}-\mathbf{A}_k x)$, $s^o(k-1, \mathbf{b}-\mathbf{A}_k x)$, $x = 0, 1$. We stop ADP if $s^o(k-1, \mathbf{b}-\mathbf{A}_k x) = 1$ for both $x = 0$ and $x = 1$ (i.e., $\mathbf{x}_{BH}(k-1, \mathbf{b}-\mathbf{A}_k x)$ is optimal for $x = 0, 1$ meaning that we find an optimal solution to $BIP(k, \mathbf{b})$). For a certain value $x \in \{0, 1\}$, we say that $BIP(k, \mathbf{b} - \mathbf{A}_k x)$ dominates $BIP(k, \mathbf{b} - \mathbf{A}_k(1-x))$ if $H_k(\mathbf{b} - \mathbf{A}_k x) > U_k(\mathbf{b} - \mathbf{A}_k(1-x)) + c_k(1-x)$. We set the parameter $d(k-1, \mathbf{b}-\mathbf{A}_k x) = 1$, (0) if $BIP(k, \mathbf{b}-\mathbf{A}_k x)$ dominates $BIP(k, \mathbf{b}-\mathbf{A}_k(1-x))$, (otherwise). Thus, ADP can be terminated if $d(k-1, \mathbf{b}-\mathbf{A}_k x) = 1$ and $s^o(k-1, \mathbf{b}-\mathbf{A}_k x) = 1$ for $x = 0$, or 1. The parameter $stop - adp$ described in Table 4.2 indicates whether the base-heuristic learning algorithm can be terminated or not, i.e., $stop-adp = 1$ (0) means *early termination* is satisfied (not satisfied). We describe the algorithmic version of ADP stopping criterion in Table 4.2 on page 103.

Lag policy assignment scheme:

As introduced earlier, we use $\mathbf{x}^f = (x_1^f, \cdots, x_n^f)$ to describe the status of the variables, i.e., fixed or not. We call a variable j is *fixed* if $x_j^f = 0$ or 1. Let F denote the set of indices of the fixed-variables. Under the *lag policy assignment scheme* once x_k^{adp} is assigned to a value 0 or 1 as described in the policy assignment we update the status vector $\mathbf{x}^f$ by setting $x_j^f = x_{BH}(k-1, \mathbf{b} - \mathbf{A}_k x^*)_j$ for $j \in [k-l, k)$ and $j \notin F$ where $x^* = x_k^{adp}$ and l is the `lag-size` parameter. l is determined by $l = \lfloor \frac{k}{\text{lag-time}} \rfloor$, where `lag-time` is a user-specified parameter. Lag-size parameter l specifies the number of variables to be fixed once x_k^{adp} is determined in the policy assignment. Lag policy fixes variables $x_{k-l}, \cdots, x_{k-1}$ assuming they have not been fixed. So, the larger the values of the `lag-size` parameter l, the more variables are fixed everytime the policy assignment is called. Due to our construction $l = \lfloor \frac{k}{\text{lag-time}} \rfloor$, for a certain `lag-time` value, we dynamically update the value of `lag-size` parameter l for varying k in $[1, n]$ in the backward phase. We provide an illustrative example how the lag policy assignment is applied within the base-heuristic learning framework in Section 4.4.1

on page 102. In our computations, we use varying `lag-time` values. In summary, we follow the base heuristic learning algorithm described in Table 5.1 to solve binary integer programming problems.

Base-heuristic Learning Algorithm
Binary Integer Programming

1: Initialization:
$k = n,\ \mathbf{b} = \mathbf{b}_0,\ \mathbf{x}^f = \mathbf{0}$
$stop - adp = 0$

2: Apply $BH(k, \mathbf{b})$
Output: $x_{BH}(k, \mathbf{b}),\ H_k(\mathbf{b}),\ p^f(k, \mathbf{b}),\ s^o(k, \mathbf{b})$
$p^f(k, \mathbf{b}) = 0 \rightarrow$ exit (no feasible solution)
$s^o(k, \mathbf{b}) = 1 \rightarrow$ exit (optimal solution)
$\mathbf{x}^{best} \leftarrow x_{BH}(k, \mathbf{b}),\ z^{best} \leftarrow H_k(\mathbf{b})$

3: Apply *reduced cost fixing*
Output: $\mathbf{x}^f$

4: while $(k > 2)$ and $(stop - adp = 0)$ do
begin (while)
if $(x_k^f = 0$ or $x_k^f = 1)$ then
{
$-x_k^{adp} \leftarrow x_k^f$
}
else
{
-policy-assignment
Output: x_k^{adp}, (0 or 1)
-update best solution
Output: $\mathbf{x}^{best}, z^{best}$
-stopping condition
Output: $stop - adp$, (0 or 1)
-lag policy assignment
Output: $\mathbf{x}^f$
}
$\mathbf{b} \leftarrow \mathbf{b} - \mathbf{A}_k x_k^{adp}$
$k \leftarrow k - 1$
end (while)

5: x_1^{adp} solves $BIP(1, \mathbf{b})$

6: Output: $\mathbf{x}^{best}$ and z^{best}.

Table 5.1: Base Heuristic Learning Algorithm for the BIP problem.

5.2.2 Base-heuristic selection

We emphasize that base-heuristic selection is very important in the base-heuristic learning framework. In essence, any methodology that is capable of solving the binary integer programming problem can be considered as one of the alternatives for the base-heuristic selection. Generally speaking, for binary integer programs finding a feasible solution is as difficult as solving the problem. Thus, it is quite challenging even to come up with a practical suboptimal methodology. As a matter of fact, as we pointed out in the background section only few efforts have been made to devise heuristics that target generic binary integer programming problems. In our computations, we use CPLEX based branch-and-bound and an enhancement of pivot-and-complement heuristic proposed by Balas and Martin [3], called *pivot-fix-and-complement heuristic*. We will be describing the details of these heuristics below.

5.2.2.1 CPLEX branch-and-bound based heuristic

The CPLEX based branch-and-bound heuristic (B) works as follows: The CPLEX tolerance parameter (`CPX- PARAM-EPAGAP`), which is a relative tolerance on the gap between the best integer objective and the objective of the best node remaining in the branch-and-bound tree, is set to a user-specified value and CPLEX is started to optimize the problem. Once CPLEX finds a feasible solution, whose value is referred as *best integer solution*, such that the value

$$\frac{|\text{best node value -best integer value}|}{|\text{best node value}|}$$

falls below the `CPX-PARAM-EPAGAP` CPLEX will be stopped where *best node value* is the tightest upper bound for the problem. We retrieve the necessary statistics such as solution, the objective value of the corresponding solution etc. from the CPLEX environment. For instance, if the value `CPX-PARAM-EPAGAP` is set to 0.25, CPLEX is required to find a feasible solution within 25% of an upper bound, e.g., truncated linear programming value.

5.2.2.2 Pivot-fix-and-complement heuristic

Pivot-fix-and-complement heuristic is an enhancement to pivot-and-complement heuristic proposed by Balas and Martin [3]. In essence, pivot-and-complement is a simplex-based heuristic methodology for binary integer programming problems motivated by the proposition below. We propose new pivot selection rules, a new basic variable fixing, and a new complementing scheme as will be described later in the section while exploiting the main ideas behind the pivot-and-complement heuristic methodology.

Let us consider the following binary integer program BIP_0:

BIP_0:

$$\begin{aligned} \text{maximize} \quad & \sum_{j=1}^{n} c_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} a_{ij} x_j \leq b_i \quad i = 1, \cdots, m \\ & x_j \in \{0,1\}, \qquad j = 1, \cdots, n. \end{aligned}$$

The linear programming relaxation yields an important information for BIPs because of the following proposition (see Proposition 2.1 in Section II.6, Nemhauser and Wolsey [61]).

Proposition 1 *Every feasible solution to binary integer programming problem is an extreme point of linear programming relaxation* $P = \{\mathbf{x} \in \Re_+^n : \sum_{j=1}^{n} \mathbf{A}_j x_j \leq \mathbf{b},\ \mathbf{x} \leq \mathbf{e}\}$.

This result motivates a methodology that systematically searches the integral extreme points of P in the neighborhood of an optimal solution to the linear programming relaxation for good feasible solutions to BIP_0. Suppose we solve the linear programming relaxation of BIP by an upper-bounded simplex algorithm (see Bertsimas and Tsitsiklis [12]). If all structural variables are nonbasic at optimality or,

equivalently, all slack variables are in the basis, then the LP solution is integral. Thus, we can transform BIP_0 to an equivalent problem BIP_1 by forcing all of the slack variables to be in the basis (i.e., all of the slack variables needs to be basic) as follows:

BIP_1:

$$
\begin{array}{lll}
\text{maximize} & \sum_{j=1}^{n} c_j x_j & \\
\text{subject to} & \sum_{j=1}^{n} a_{ij} x_j + y_i = b_i & i = 1, \cdots, m \\
& 0 \leq x_j \leq 1 & j = 1, \cdots, n \\
& y_i \geq 0 \;\; i = \{1, \cdots, m\} & \\
& y_i, i \in \{1, \cdots, m\} \text{ basic variable.} &
\end{array}
$$

The basic motivation of *pivot-and-complement* is to exploit this equivalence in an approximate way meaning that it tries to put all of the slack variables into the linear programming basis at a minimal change in the objective value of the linear programming relaxation of the BIP_0. The fact that all of the slack variables are in the basis implies that a feasible solution is available to the problem. Let $N = \{1, \cdots, n\}$ be the index set of the structural (original x_j) variables and $M = \{1, \cdots, m\}$ be the index set of constraints (rows).

Pivot-fix-and-complement (PFC) consists of two phases. In the first phase, it tries to generate a feasible solution by pivoting, basic variable fixing, and complementing as will be described below. PFC starts the first phase with solving the linear programming relaxation. Let I be the index set for basic variables and J be the index set for non-basic variables. Under the assumption that upper-bounded simplex method is exploited, we have the following tabular information at optimality.

$$x_i + \sum_{j \in J} y_{ij} x_j = \bar{b}_i$$

If the linear programming solution is integral (i.e., $x_j^{LP} = 0$ or 1) then the algorithm terminates with an optimal solution. Otherwise, the pivot-fix-and-complement performs *pivoting* as long as it finds pivots as described below.

Pivoting:

Type 1 Pivot: Under this type of pivoting we select a column q corresponding to a nonbasic slack variable. Column q will enter the basis. By performing a *min ratio test*, we will attempt to maintain feasibility and force one of the basic original variables out of the basis. Having selected the column to enter the basis, the index of the corresponding basic structural variable to exit the basis will be selected by computing

$$\delta_q = \min \left(\min \left\{ \frac{\bar{b}_i}{y_{iq}} : i \in I, y_{iq} > 0 \right\}, \min \left\{ \frac{1 - \bar{b}_i}{-y_{iq}} : i \in I \cap N, -y_{iq} > 0 \right\} \right).$$

Let p the index that achieves the minimum. Note that the change in the objective function value will be $\bar{c}_q \delta_q < 0$. Let P_1 denote the index set of type 1 pivots. We propose to select the column k to enter the basis by $k = \text{argmin}_{q \in P_1} |\bar{c}_q| \delta_q$, i.e., we attempt to decrease the objective function value as small as possible. Type 1 pivoting reduces the number of structural basic variables in the basis.

Type 2 Pivot: A pivot of type 2 is one that maintains primal feasibility and leaves unchanged the number of basic structural variables (i.e., exchanges a nonbasic slack for a basic slack or a nonbasic structural or basic structural variable). Let $INF = \sum_{i \in I \cap N} \min \{\bar{b}_i, 1 - \bar{b}_i\}$ be the value of integer infeasibility of the current LP solution. Let INF_q be the *resulting* integer infeasibility of the corresponding LP solution once the nonbasic variable q is pivoted in the LP basis. Let P_2 denote the index set of type 2 pivots. We propose to select variable k to enter the basis

by the *minimum integer infeasibility pivoting rule*, i.e., $k = \text{argmin}_{q \in P_2} INF_q$. This type 2 pivoting scheme will be used in place of Balas and Martin's suggestion that type 2 pivoting will be done if the integer infeasibility is reduced by some value $\Delta > 0$.

We perform pivoting until there exists no type 1 pivots and the value of integer infeasibility INF of the corresponding LP solution is not decreasing once only type 2 pivots exist (i.e, $P_1 = \emptyset$ and $P_2 \neq \emptyset$). At the end of this pivoting phase, either a feasible solution is available meaning that all nonbasic slack variables reside in the current basis or there are structural variables in the current basis. We obtain the current LP solution $\mathbf{x}^{LP}$ and check whether all of the basic structural variables have values of 0 or 1 implying that we have an integral solution so that we proceed with the second phase of pivot-fix-and-complement. Under the assumption that we have fractional basic structural variables we apply *basic variable fixing* as described below:

Basic variable fixing:

Let us define by X_0 the set of structural 0-1 variables at zero, i.e., $x_j^{LP} = 0$, and X_1 the set of structural variables at one, i.e., $x_j^{LP} = 1$. Then $X_b = N \setminus \{X_0, X_1\}$ denotes the fractional basic structural variables, i.e., $0 < x_j^{LP} < 1$. Let $\mathbf{s} = \mathbf{b} - \sum_{j \in X_0} \mathbf{A}_j x_j^{LP} - \sum_{j \in X_b} \mathbf{A}_j x_j^{LP} - \sum_{j \in X_1} \mathbf{A}_j x_j^{LP}$ be the slack vector for the current LP solution $\mathbf{x}^{LP}$. We denote by C^L the $'\leq'$ constraints. Similarly, let C^E (C^G) denote $'='$ ($'\geq'$) constraints. We know that $s_i \geq 0$ for all $i \in C^L$, $s_i \leq 0$ for all $i \in C^G$ and $s_i = 0$ for all $i \in C^E$ of a feasible slack vector $\mathbf{s}$. We measure the infeasibility for a constraint $i \in C^L$ by $\max\{0, -s_i\}$ which has a positive value if $s_i < 0$. Similarly, we introduce the infeasibility measure for a constraint $i \in C^G$ by $\max\{0, s_i\}$ which has a positive value if $s_i > 0$ and for a constraint $i \in C^E$ by $|s_i|$ which has a positive value if $s_i \neq 0$. Thus, we can use the following *slack infeasibility* measure $SINF(\mathbf{s})$ for a given slack vector $\mathbf{s}$:

$$SINF(\mathbf{s}) = \sum_{i \in C^L} \max\{0, -s_i\} + \sum_{i \in C^G} \max\{0, s_i\} + \sum_{i \in C^E} |s_i|$$

We do basic variable fixing for a fractional structural variable x_k as follows. We first set the variable x_k to 0 and calculate its corresponding slack vector $\mathbf{s}^0 = \mathbf{s} + \mathbf{A}_k x_k^{LP}$ and resultant slack infeasibility $SINF(\mathbf{s}^0)$. We then set the variable x_k to 1 and calculate its corresponding slack vector $\mathbf{s}^1 = \mathbf{s} + \mathbf{A}_k(1 - x_k^{LP})$ and resultant slack infeasibility $SINF(\mathbf{s}^1)$. We fix the variable x_k to a value (0 or 1) so that the resultant slack infeasibility is smallest (e.g., if $SINF(\mathbf{s}^0) < SINF(\mathbf{s}^1)$, then we fix x_k to 0). Once we fix the variable x_k to a value (e.g., $x_k = 0$), we update $\mathbf{s}$ and $SINF(\mathbf{s})$ accordingly (e.g., $\mathbf{s} = \mathbf{s}_0$ and $SINF(\mathbf{s}) = SINF(\mathbf{s}^0)$). We perform the described basic variable fixing for all of the fractional structural variables. We describe the details of the *basic variable fixing* in Table 5.2 as follows:

Once basic variable fixing is complete, a binary solution is available which is not necessarily feasible (i.e., the slack infeasibility $SINF(\mathbf{s}) \neq 0$ for the current slack vector $\mathbf{s}$). Let us denote this solution by $\mathbf{x}^H$ where $X_0 = \{j | x_j^H = 0\}$ and $X_1 = \{j | x_j^H = 1\}$. We complement variables until slack infeasibility $SINF(\mathbf{s})$ becomes zero meaning that a feasible solution is attained. By *complementing* a variable x_j we mean moving x_j from one of its bounds to the opposite bound, and substituting $x_j = 1 - y_j$. We describe the complementing in the search phase in Table 5.3.:

In summary, pivot-fix-and-complement tries to find a feasible solution in the first phase by pivoting, basic variable fixing and complementing that aims to reduce the slack infeasibility as described in Table 5.4. Let $P1$ denote the index set of type 1 pivots, and $P2$ denote the index set of type 2 pivots. We denote by $INF(current)$ the value of integer infeasibility of the current LP solution and by $INF(new)$ the value of resulting integer infeasibility of the corresponding LP solution once a type 2 pivoting is performed.

Basic variable fixing

Input: $X_0, X_1, \mathbf{x}^{LP}$

1: $\mathbf{s} = \mathbf{b} - \sum_{j=1}^{n} \mathbf{A}_j x_j^{LP}$

2: $X_b \leftarrow N \setminus \{X_0, X_1\}$

3: while $(X_b \neq \emptyset)$ do

begin (while)

-select $k \in X_b$

$\mathbf{s}^0 = \mathbf{s} + \mathbf{A}_k x_k^{LP}$

$SINF(\mathbf{s}^0) = \sum_{i \in C^L} \max\{0, -s_i^0\} + \sum_{i \in C^G} \max\{0, s_i^0\} + \sum_{i \in C^E} |s_i^0|$

$\mathbf{s}^1 = \mathbf{s} + \mathbf{A}_k (1 - x_k^{LP})$

$SINF(\mathbf{s}^1) = \sum_{i \in C^L} \max\{0, -s_i^1\} + \sum_{i \in C^G} \max\{0, s_i^1\} + \sum_{i \in C^E} |s_i^1|$

-if $SINF(\mathbf{s}^0) < SINF(\mathbf{s}^1)$ then {

$X_0 \leftarrow X_0 \cup \{k\}$

$\mathbf{s} = \mathbf{s} + \mathbf{A}_k x_k^{LP}$

}

else {

$X_1 \leftarrow X_1 \cup \{k\}$

$\mathbf{s} = \mathbf{s} + \mathbf{A}_k (1 - x_k^{LP})$

}

end (while)

4: Output: $\mathbf{s}, X_0, X_1$

Table 5.2: Description of basic variable fixing.

Complementing: Search Phase

1: Input: $X_0,\ X_1$

2: $\mathbf{s} \leftarrow \mathbf{b} - \sum_{j \in X_1} \mathbf{A}_j$

$SINF(\mathbf{s}) = \sum_{i \in C^L} \max\{0, -s_i\} + \sum_{i \in C^G} \max\{0, s_i\} + \sum_{i \in C^E} |s_i|$

3: while ($SINF(\mathbf{s}) > 0$) do

begin (while)

Find $O \subset X_0$ and $I \subset X_1$

$\mathbf{s}' = \mathbf{s} + \sum_{j \in O} \mathbf{A}_j - \sum_{j \in I} \mathbf{A}_j$

if $SINF(\mathbf{s}') < SINF(\mathbf{s})$ then

{

$\mathbf{s} \leftarrow \mathbf{s}'$

$X_0 \leftarrow (X_0 \setminus O) \cup I$

$X_1 \leftarrow (X_1 \cup O) \setminus I$

}

end (while)

3: Output: feasible solution (or not due to an early termination criterion)

Table 5.3: Description of complementing: search phase.

Pivot-fix-and-complement: Search Phase

Initialization $X_0 = X_1 = \emptyset$

1: solve LP and get $\mathbf{x}^{LP}$

LP is infeasible ? $\rightarrow$ exit (problem is infeasible)

$\mathbf{x}^{LP}$ integral? $\rightarrow$ exit (optimal solution)

2: search for pivots (type 1 and 2)

output: $P1,\ P2$

if ($P1 = \emptyset$ and $P2 \neq \emptyset$) then

calculate $INF(current)$

3: while ($P1 \neq \emptyset$ or $P2 \neq \emptyset$) do

begin (while)

if $P1 \neq \emptyset$ {

- perform type 1 pivoting

- update the simplex tabular information

}

else $P2 \neq \emptyset$ {

- perform type 2 pivoting

- update the simplex tabular information

}

get $\mathbf{x}^{LP}$

$\mathbf{x}^{LP}$ integral? exit (feasible solution)

search for pivots (type 1 and 2)

output: $P1,\ P2$

if ($P1 = \emptyset$ and $P2 \neq \emptyset$) then

{

calculate $INF(new)$

if $INF(new) > INF(current)$ then exit while loop

}

end (while)

4: $X_0 \leftarrow \{j \in N | x_j^{LP} = 0\}$ and $X_1 \leftarrow \{j \in N | x_j^{LP} = 1\}$

5: perform basic variable fixing (Table 5.2)

output: X_0, X_1

6: $x_j^H = 0,\ j \in X_0$ and $x_j^H = 1,\ j \in X_1$

7: $\mathbf{x}^H$ feasible? exit (feasible solution)

8: perform complementing (Table 5.3)

output: $\mathbf{x}^H$

9: Output: $\mathbf{x}^H$ feasible or not

Table 5.4: Description of pivot-fix-and-complement: search phase.

Second Phase of pivot-fix-and-complement:

Once the first phase of pivot-fix-and-complement finishes either a feasible solution is attained or the pivot-fix-and-complement terminates unsuccessfully meaning that no feasible solution is found by this methodology. Under the assumption that a feasible solution is attained, the second phase of pivot-fix-and-complement tries to improve the current solution through a complementing mechanism. Let $\mathbf{x}^H$ be the solution attained at the end of first phase. Let $X_0 = \{j | x_j^H = 0\}$ and $X_1 = \{j | x_j^H = 1\}$. Let $O \subset X_0$ and $I \subset X_1$ such that $\sum_{l \in O} c_l > \sum_{k \in I} c_k$. We complement the set $S = O \cup I$ in order to *strictly* improve the objective value of the current solution as long as the resultant slack vector $\mathbf{s}' = \mathbf{s} + \sum_{j \in O} \mathbf{A}_j - \sum_{j \in I} \mathbf{A}_j$ is feasible (i.e., $SINF(\mathbf{s}') = 0$).

5.3 Computational Study

In this section, we provide computational results to the binary integer programming problems. Our basic aim is to demonstrate that approximate dynamic programming with base-heuristic learning can generate near optimal solutions for large and difficult instances in reasonable amount of computation time. We consider the proposed base-heuristics, namely the pivot-fix-and-complement and CPLEX based branch-and-bound heuristics, as described in Section 5.2.2. We apply our approximate dynamic programming methodologies on both randomly generated problem instances and MIPLIB test problems (Bixby et. al. [13]). We will also be comparing ADP algorithm with CPLEX. Our criteria for comparing various approaches are solution quality, computation time, and robustness, i.e. the degree of deviation of the computational resources needed to solve the problems as the instances of the same size change.

5.3.1 Randomly generated problems

In this section our objective is to analyze the performance of ADP with the proposed base-heuristic pivot-fix-and-complement on randomly-generated problems and make a comparative study with one of the state-of-art commercial packages like CPLEX.

We construct *uncorrelated, weakly and strongly correlated* binary integer programming problems. Let $x \sim U(1, X)$ be a *integer* random number uniformly generated in $[1, X]$. We consider the following problem types:

- **Uncorrelated (UC):** $c_j \sim U(1, C)$ and $a_{ij} \sim U(-A, A)$, i.e, c_j, a_{ij} are uniformly distributed in $[1, C], [-A, A]$ respectively.
- **Weakly correlated (WC):** $a_{ij} \sim U(-A, A)$ and $c_j = \max\{1, U(\sum_i a_{ij}/m - wc, \sum_i a_{ij}/m + wc)\}$.
- **Strongly correlated (SC):** $a_{ij} \sim U(-A, A)$ and $c_j = \sum_i a_{ij}/m + sc$.

Given the number of constraints m and the number of variables n we generate 5 test problems in each class (uncorrelated, weakly, and strongly correlated) with different parameters C and A as given in Table 5.5. We set $b_{0,i} = 0.5 \sum_{j=1}^{n} \mathbf{A}_j$. All computational studies are done on a Dell Precision 410 with Linux operating system.

UC		WC		SC	
A	C	A	wc	A	sc
100	100	100	10	50	5
1,000	100	1,000	50	100	10
5,000	1,000	1,000	100	1,000	150
5,000	2,500	5,000	500	5,000	500
10,000	5,000	10,000	1,000	10,000	1,000

Table 5.5: Parameters for the Uncorrelated, Weakly and Strongly correlated type problems.

We use the proposed *pivot-fix-and-complement* (PFC) as the base-heuristic within

the base-heuristic learning framework. We call the corresponding approximate dynamic algorithm as ADP-PFC. We set ADP-PFC's time-limit to 900 CPU seconds meaning that the algorithm will be stopped once the time-limit is reached. We use ADP with lag policy and we set `lag-time` to 10. We compare ADP-PFC with the commercial package CPLEX. In order to make a legitimate comparison we set CPLEX tolerance parameter `CPX-PARAM-EPGAP` to PE(ADP-PFC)/100 (i.e., for a problem instance, we run ADP-PFC and get a suboptimal solution and we calculate the percentage deviation from the linear programming solution value, PE(ADP-PFC)). We also set the tree memory limit and node file size limit in CPLEX to 250 MB. In addition, we set CPLEX time limit to 2,700 CPU seconds. CPLEX terminates when one of the binding parameters is satisfied. Once CPLEX terminates, we retrieve its solution and calculate the percentage deviation from the linear programming value. We also report its average CPU time and number of nodes to analyze the performance of CPLEX.

We use the following notation to present the computational study for the binary integer programming problems.

- PFC: Pivot-fix-and-complement heuristic.
- ADP-PFC: Base Heuristic Learning with the base-heuristic PFC.
- Let v(X) be the objective value of the solution obtained by methodology X, e.g., v(ADP-PFC) is the solution value obtained by the methodology ADP-PFC.
- Let PE(X) be the percentage deviation of the solution value of the methodology X from the linear programming (LP) value i.e. $\text{PE(X)} = \frac{v(LP)-v(X)}{v(LP)} \times 100$.
- PI(X): Percentage Improvement of ADP-X over X, i.e., $\frac{PE(X)-PE(ADP-X)}{PE(X)} \times 100$
- T(X): Computation time of the methodology X in CPU seconds.
- NF: number of problems that a methodology finds feasible a solution (maximum value of NF is 5)

Tables 5.6, 5.7 and 5.8 present the results for the proposed base-heuristic, namely pivot-fix-and-complement. We observe that PFC finds good solutions at short computation time even though the solution quality of PFC degrades as m and n increases. PFC could not find a feasible solution for one instance of weakly correlated type problems of size $m = 100, n = 1,000$ and for two instances of strongly correlated type problems of size $m = 100, n = 1000$ within the PFC time-limit of 900 CPU seconds. This observation suggests that difficulty of problem instances increase in the direction of stronger correlation. Tables 5.6, 5.7 and 5.8 report summary statistics for those problem instances where PFC finds feasible solutions. Overall, PFC generated reasonably good solutions in short times even for large problems with thousands of variables. We observe that the solution quality of PFC degrades as the problems have stronger correlation. These tables also indicate that the stronger the correlation the larger the computation time required by the pivot-fix-and-complement heuristic.

		NF	T(PFC)			PE(PFC)		
m	n		min	avg	max	min	avg	max
5	100	5	0.00	0.01	0.01	0.0244	0.3636	0.7885
10	500	5	0.08	0.09	0.10	0.0007	0.1386	0.4823
10	1,000	5	0.18	0.19	0.21	0.0045	0.1643	0.3890
50	1,000	5	1.88	2.19	2.33	0.6887	1.3536	2.0244
100	1,000	5	10.63	19.72	51.40	2.0924	3.1881	5.0504

Table 5.6: Performance of PFC. Uncorrelated type problems.

		NF	T(PFC)			PE(PFC)		
m	n		min	avg	max	min	avg	max
5	100	5	0.00	0.01	0.01	0.0000	3.9377	7.6990
10	500	5	0.09	0.09	0.09	0.0003	1.9335	2.9356
10	1,000	5	0.21	0.22	0.22	0.0004	1.0528	1.6536
50	1,000	5	2.18	2.40	2.76	4.2503	5.4426	6.6725
100	1,000	4	9.76	20.27	34.13	6.9465	7.9970	10.1096

Table 5.7: Performance of PFC. Weakly correlated type problems.

		NF	T(PFC)			PE(PFC)		
m	n		min	avg	max	min	avg	max
5	100	5	0.01	0.01	0.01	0.0000	2.8599	6.7949
10	500	5	0.09	0.10	0.10	0.0371	1.6598	2.8032
10	1,000	5	0.21	0.22	0.24	0.0000	0.8221	1.0846
50	1,000	5	2.18	2.35	2.53	2.7982	3.4922	4.3115
100	1,000	3	8.90	15.64	25.71	4.2310	6.5870	8.1456

Table 5.8: Performance of PFC. Strongly correlated type problems.

Tables 5.9, 5.10 and 5.11 present computational results for ADP-PFC. These tables suggest that ADP-PFC improves the performance of PFC to attain better solutions within the time-limit of 900 CPU seconds. ADP-PFC could not find a feasible solution for one instance of weakly correlated type problems of size $m = 100, n = 1,000$ and for two instances of strongly correlated type problems of size $m = 100, n = 1000$ within the time-limit of 900 CPU seconds. In fact, due to the nature of base-heuristic learning framework once the base-heuristic can not find a feasible solution ADP algorithm also terminates unsuccessfully with no feasible solution (see Table 5.4). These tables report summary statistics for those problem instances where ADP-PFC finds feasible solutions. Average percentage deviation values of PFC (i.e., average of PE(PFC) values in Tables 5.6, 5.7 and 5.8) are 1.0416, 4.0728, and 3.0842 for uncorrelated, weakly correlated and strongly correlated type problems respectively. In comparison to these values, ADP-PFC attains smaller average percentage deviation values (i.e., average of PE(ADP-PFC) values in Tables 5.9, 5.10 and 5.11), namely 0.6798, 2.8890, 2.4710 for uncorrelated, weakly correlated and strongly correlated type problems respectively. We observe that the solution quality of ADP-PFC degrades as the problems have stronger correlation and as m and n increases. For a certain m, n, ADP-PFC demonstrated a robust performance meaning that there exists small deviation in both solution quality and computation time. As an example, let us focus on uncorrelated type problems of size $m = 10, n = 1,000$ (see Table 5.9). We observe that the minimum (0.51), average (1.65) and maximum (2.44) computation times are quite close to one another. Thus, ADP-PFC can be considered robust

from a computation time perspective. Similarly, Table 5.9 illustrates that the minimum (0.0042), average (0.0871) and maximum (0.1621) percentage deviations are quite close to one another indicating that ADP-PFC is robust from a solution quality perspective. Overall, based on our performance criteria ADP-PFC can be considered as a robust methodology.

As described, we also apply CPLEX to these problem instances. Overall, these tables suggest that CPLEX could not find a feasible solution that matches the percentage deviation of ADP-PFC, PE(ADP-PFC), within the time-limit of 2,700 CPU seconds and memory limit of 250 MB. In fact, CPLEX could not find feasible solutions at all for the uncorrelated type problems of size $m = 100, n = 1,000$, the weakly-correlated type problems of size $m = 50, n = 1,000$ and $m = 100, n = 1,000$, and the strongly-correlated type problems of size $m = 50, n = 1,000$ and $m = 100, n = 1,000$.

		CPLEX				T(ADP-PFC)			PE(ADP-PFC)		
m	n	avg. time	avg. no nodes	NF	PE	min	avg	max	min	avg	max
5	100	0.00	59.60	0.1899	5	0.01	0.03	0.07	0.0244	0.2593	0.4452
10	500	14.00	3,991.80	0.1107	5	0.11	0.44	0.79	0.0007	0.1139	0.3587
10	1,000	1,382.80	187,986.80	0.2147	5	0.51	1.65	2.44	0.0042	0.0871	0.1621
50	1,000	time-limit	109,938.60	6.0833	5	15.47	26.60	47.54	0.4762	0.7059	1.2931
100	1,000	time-limit	NA	NA	0	900.88	901.74	902.42	1.6998	2.2328	2.8505

Table 5.9: Comparison of ADP-PFC with CPLEX. Uncorrelated type problems.

		CPLEX				T(ADP-PFC)			PE(ADP-PFC)		
m	n	avg. time	avg. no nodes	NF	PE	min	avg	max	min	avg	max
5	100	7.20	8,666.00	5	2.065	0.01	0.07	0.11	0.0000	2.3142	5.1248
10	500	2043.20	472,531.20	5	4.5745	1.08	1.14	1.21	0.0003	1.2450	1.8676
10	1,000	2,160.40	239,108.40	5	5.4042	2.15	2.37	2.46	0.0001	0.7043	1.2381
50	1,000	time-limit	NA	0	NA	202.34	522.81	900.56	1.8817	3.6311	4.8159
100	1,000	time-limit	NA	0	NA	901.36	902.79	904.42	5.2859	6.5505	7.7867

Table 5.10: Comparison of ADP-PFC with CPLEX. Weakly correlated type problems.

In order to illustrate that ADP with a base-heuristic (H) improves the performance of the base-heuristic (H) we calculate percentage improvement statistics. Table 5.12

		CPLEX				T(ADP-PFC)			PE(ADP-PFC)		
m	n	avg. time	avg. no nodes	NF	PE	min	avg	max	min	avg	max
5	100	0.80	1,010.20	5	1.4103	0.00	0.07	0.12	0.0000	2.0138	4.0486
10	500	1,533.80	319,981.00	5	1.2010	0.17	1.01	1.29	0.0002	1.0024	1.6174
10	1,000	1,236.80	117,194	5	0.9564	0.21	2.20	2.79	0.0000	0.6388	0.8897
50	1,000	time-limit	NA	0	NA	2.86	5.34	15.29	2.6575	3.0991	3.4908
100	1,000	time-limit	NA	0	NA	13.05	159.13	450.05	4.2310	5.6010	6.5039

Table 5.11: Comparison of ADP-PFC with CPLEX. Strongly correlated type problems.

demonstrates that ADP-PFC improves over the base-heuristic PFC, meaning that ADP-PFC generates better solutions than the stand-alone heuristic PFC. The percentage improvement statistics of PFC, PI(PFC), in Table 5.12 is the average of percentage improvements for those problems for which both PFC and ADP-PFC find feasible solutions.

		PI(PFC)		
m	n	UC	WC	SC
5	100	15.63	23.80	30.69
10	500	5.12	25.26	47.92
10	1,000	33.78	38.43	17.78
50	1,000	43.03	34.59	9.45
100	1,000	25.93	17.48	12.48

Table 5.12: Average percentage improvement of ADP-PFC over PFC

Overall, ADP-PFC is a practical methodology for randomly generated binary integer programming problems. It finds good solutions in a reasonable amount of computation time with a robust performance. It successfully competes with CPLEX both from a solution quality and computation time perspective. In fact, our computational evidence demonstrated that CPLEX could not find even feasible solutions for the problems with large number of variables. We also observe that base-heuristic learning framework improves the performance of the proposed base-heuristic pivot-fix-and-complement heuristic at an additional computation time which supports our

consistent obervation throughout the thesis.

5.3.2 MIPLIB Test Problems

In this section, our aim is to illustrate the effectiveness of base-heuristic learning for test problems from the literature, namely MIPLIB test problems (see Bixby et. al. [13]). MIPLIB is an electronically available library of both pure and mixed integer programs, most of which arise from real-world applications. MIPLIB is generally considered a standard test set for comparing different integer programming approaches. We consider only pure binary integer programming problems out of all MIPLIB test problems. Typical examples of real-world pure binary integer programs are set partitioning problems arising in airline industry and set covering problems arising in railways scheduling.

Let B be the CPLEX based branch-and-bound heuristic as described in Section 5.2.2. In summary, CPLEX terminates with solution statistics once it attains the relative performance of best node value and best integer value below the tolerance parameter `CPX-PARAM-EPAGAP`. Let PFC be the proposed pivot-fix-and-complement heuristic. Then, ADP-B is the approximate dynamic programming algorithm with the base-heuristic B and ADP-PFC is the the approximate dynamic programming algorithm with the base-heuristic PFC. Let T(X) denote the computation time in CPU seconds of the methology X, and V(X) is the objective value of the solution generated by methodology X.

Tables 5.13 illustrates the effectiveness of the base-heuristic learning on MIPLIB test problems. We apply ADP-B with lag policy with varying `lag-time` values (e.g., 5, 10 and 50) depending on the problem. We report the best solution value of ADP-B and its corresponding computation time for varying `lag-time` values. We observe from these tables ADP-B improves the performance of the base-heuristic B often significantly. In order to illustrate this observation, we summarize the results as follows. We put different number of bullets • next to the solution value of ADP-B to

demonstrate different case:

(i) (•): denotes the case where V(ADP-B) is better than V(B), and V(ADP-B) is not the best known solution.

(ii) (• •): denotes the case where both V(ADP-B) and V(B) attain the best-known solution value.

(iii) (• • •): denotes the case *only* V(ADP-B) attains the best-known solution, i.e., ADP-B improved the solution value of B to obtain the best-known solution value.

In case (• •) we note that ADP-B can not improve over B if the best-known solution value is indeed the optimal value to the corresponding problem. Case (• • •) is the most interesting case since ADP-B significantly improves the performance of the given base-heuristic B. Another interesting observation is that as the underlying base-heuristic B becomes stronger (by decreasing the `CPLEX-PARAM-EPGAP`) ADP-B also improves, although the relative improvement is smaller.

We apply the proposed pivot-fix-and-complement heuristic (PFC) and ADP-PFC to the MIPLIB test problems as provided in Table 5.14. We apply ADP-PFC with lag policy with a `lag-time` value of 10. We report summary statistics only for those problems where PFC and ADP-PFC find a feasible solution within the time-limit of 1,800 CPU seconds. We observe that ADP-PFC improves the performance of base-heuristic PFC which supports the tendency of base-heuristic learning framework observed so far. Both PFC and ADP-PFC find very good solutions to cap6000 problem for which CPLEX could not find a feasible solution within the time-limit of 1,800 CPU seconds. ADP-PFC attains the best-known solution value to the problem stein27 which is an instance for the case categorized by • • •.

To our knowledge, the original pivot-and-complement heuristic is applied to some of the MIPLIB problems within a tabu search framework (see Aboudi and Jornsten

[1]). Their computational study provide results only for the problems p0033, stein27 and stein 45 out of MIPLIB test problems. Table 5.14 illustrates that our proposed pivot-fix-and-complement heuristic can find good solutions to a broader problem instances out of MIPLIB test problems indicating that our enhancement to the original pivot-and-complement heuristic is practically important. Generally speaking, MIPLIB test problems contain problems with different types of constraints particularly equality constraints. This reality hinders the success of PFC which is suitable mainly for constraints of inequality type.

name	m	n	best-known	CPX-PARAM-EPAGAP	T(B)	V(B)	T(ADP-B)	V(ADP-B)
air03	124	10575	340160	0.5	2.88	340160	17.42	340160
air04	823	8904	56137	0.5	time-limit	NA	NA	NA
air05	426	7195	26374	0.5	time-limit	NA	NA	NA
cap6000 (max)	2176	6000	2451377	0.5	time-limit	NA	NA	NA
enigma	21	100	0.0	0.5	0.82	0.0	0.82	0.0 ••
fast0507	507	63009	174	0.5	770.10	179	853.11	179
harp2 (max)	112	2993	73899798	0.7	15.87	68282610	16.00	68282610
			73899798	0.05	52.37	72991951	455.25	73205884•
l152lav	97	1989	4722	0.5	1.35	5267	7.09	4937•
				0.2	1.37	5267	6.37	4942 •
				0.1	3.46	4755	7.41	4755
				0.05	3.54	4755	89.30	4726•
lseu	28	89	1120	0.5	0.11	1154	3.21	1154
				0.2	0.10	1154	5.72	1120 •••
mitre	2054	10724	115155	0.5	time-limit	NA	NA	NA
mod008	6	319	307	0.05	0.37	307	4.59	307 ••
mod010	146	2655	6549	0.8	1.00	6549	2.98	6549 ••
mod010	146	2655	6549	0.5	1.01	6549	3.04	6549 ••
nw04	36	87482	16862	0.5	time-limit	NA	NA	NA
p0033	16	33	3089	0.9	0.02	3089	0.29	3089 ••
	16	33	3089	0.5	0.02	3089	0.29	3089
p0201	133	201	7615	0.9	0.16	8175	6.20	7855•
			7615	0.1	0.93	7795	10.04	7675 •
	133	201	7615	0.05	1.27	7795	11.11	7615•••
p0282	241	282	258411	0.5	0.2	361225	6.68	266259•
			258411	0.2	0.48	263522	10.43	259542•
			258411	0.1	0.48	263522	18.86	259542•
			258411	0.05	0.48	263522	11.86	258411•••
p0548	176	548	8691	0.5	time-imit	NA	NA	NA
p2756	755	2756	3124	0.5	time-limit	NA	NA	NA
seymour	4944	1372	423	0.5	320.01	436	928.42	436
stein27	118	27	18	0.5	0.47	19	1.00	18 •••
stein45	331	45	30	0.5	0.11	32	4.67	31 ••

Table 5.13: Comparison of CPLEX and ADP-B on MIPLIB test problems

name	m	n	best-known	T(PFC)	V(PFC)	T(ADP-PFC)	V(ADP-PFC)
cap6000 (max)	2176	6000	2451377	14.41	2431973	453.00	2450402
fast0507	507	63009	174	922.65	216	1800.0	204
mod008	6	319	307	0.01	370	0.20	329
p0033	16	33	3089	0.01	3661	0.79	3095
p0282	241	282	258411	0.15	347334	1.03	340947
seymour	4944	1372	423	256.04	452	1800.00	447
stein27	118	27	18	0.02	27	0.30	18 • • •
stein45	331	45	30	0.03	45	2.00	33

Table 5.14: Performance of PFC and ADP-PFC on MIPLIB test problems

In the following tables, we compare ADP-B with CPLEX on test problems from the literature. ADP-B is the base-heuristic learning with the CPLEX based branch-and-bound B as described in Section 5.2.2. We set the tree memory limit and node file size limit in CPLEX to 250 MB. In addition, we set heuristic time limit to 600 CPU seconds. As an input to heuristic B, we need to specify the CPLEX tolerance parameter `CPX-PARAM-EPGAP`. In the following computations we use varying `CPX-PARAM-EPGAP` values of 0.25, 0.50 and 0.75. We use different `lag-time` values of 5 and 10. Thus, for a certain `CPX-PARAM-EPGAP` and `lag-time` we apply ADP-B to a test problem and obtain the objective value of the solution obtained by ADP-P and the corresponding computation time. In order to make a legitimate comparison we set CPLEX tolerance parameter `CPX-PARAM-EPGAP` to PE(ADP-B)/100. We also set the tree memory limit and node file size limit in CPLEX to 250 MB. In addition, we set CPLEX time limit to 1,800 CPU seconds. CPLEX terminates when one of the binding parameters is satisfied. Once CPLEX terminates, we obtain the solution value and computation time of CPLEX.

Table 5.15 compares CPLEX and ADP-B on air03, a set partitioning problem. For all `CPX-PARAM-EPGAP` and `lag-time` combinations, ADP-B achieves the best-known solution (340160) by improving solution value of B (360386) at an additional computation time. We note that CPLEX finds the best-known solution in a shorter time than ADP-B. For instance, in the case of `CPX-PARAM-EPGAP`=0.25 and `lag-time`=5, the computation time of ADP-B is 7.71 while CPLEX requires only 2.09 CPU seconds.

best-known	T(CPX)	V(CPX)	lag-time	CPX-PARAM-EPAGAP	T(B)	V(B)	T(ADP-B)	V(ADP-B)
340160	2.09	340160	5	0.25	3.79	360386	7.71	340160
	2.09	340160		0.50	3.78	360386	7.62	340160
	2.09	340160		0.75	3.85	360386	7.72	340160
	2.10	340160	10	0.25	3.83	360386	11.49	340160
	2.09	340160		0.50	3.79	360386	11.40	340160
	2.07	340160		0.75	3.79	360386	11.48	340160

Table 5.15: Comparison of CPLEX and ADP-B on air03, $m = 124, n = 10575$

Table 5.16 compares CPLEX and ADP-B on air04, a set partitioning problem. We observe that ADP-B improves the solution value of B. We note that CPLEX finds a better solution than ADP-B in shorter time. For instance, in the case of `CPX-PARAM-EPGAP`=0.25 and `lag-time`=10, the computation time and solution value of ADP-B are 1,274.53 and 58416, respectively. CPLEX achieves a solution value of 56327 in 847.65 CPU seconds. Hence, it continues to beat ADP-P on the test problems.

best-known	T(CPX)	V(CPX)	lag-time	CPX-PARAM-EPAGAP	T(B)	V(B)	T(ADP-B)	V(ADP-B)
56137	837.84	56327	5	0.25	198.65	59546	497.98	59063
	855.12	56327		0.50	213.50	59546	529.02	59063
	854.96	56327		0.75	211.92	59546	530.24	59063
	847.65	56327	10	0.25	214.80	59546	1274.53	58416
	847.75	56327		0.50	210.89	59546	1264.95	58416
	865.30	56327		0.75	213.25	59546	1274.15	58416

Table 5.16: Comparison of CPLEX and ADP-B on air04, $m = 823, n = 8904$

Table 5.17 compares CPLEX and ADP-B on air05, another set partitioning problem. We observe that ADP-B improves the solution value of B. We note that CPLEX finds a better solution than ADP-B in shorter time. For instance, in the case of `CPX-PARAM-EPGAP`=0.25 and `lag-time`=5, ADP-B produces a solution value of 26787 in 281.50 CPU seconds. On the other extreme, CPLEX achieves a solution value of 26720 in a shorter time of 128.20 CPU seconds. Hence, it remains to be the leading methodology compared to ADP-P on the test problems.

best-known	T(CPX)	V(CPX)	lag-time	CPX-PARAM-EPAGAP	T(B)	V(B)	T(ADP-B)	V(ADP-B)
26374	128.20	26720	5	0.25	27.77	27013	281.50	26787
	126.93	26720		0.50	27.60	27013	266.69	26787
	126.14	26720		0.75	27.31	27013	267.23	26787
	123.91	26720	10	0.25	26.77	27013	392.15	26662
	126.57	26720		0.50	27.23	27013	378.49	26662
	123.62	26720		0.75	27.07	27013	378.77	26662

Table 5.17: Comparison of CPLEX and ADP-B on air05, $m = 426, n = 7195$

Table 5.18 compares CPLEX and ADP-B on cap6000. Both ADP-B and CPLEX

best-known	T(CPX)	V(CPX)	lag-time	CPX-PARAM-EPAGAP	T(B)	V(B)	T(ADP-B)	V(ADP-B)
2451377	1802.50	NA	5	0.25	602.31	NA	602.32	NA
	1802.94	NA		0.50	601.43	NA	601.43	NA
	1802.33	NA		0.75	601.69	NA	601.70	NA
	1801.74	NA	10	0.25	601.65	NA	601.66	NA
	1803.13	NA		0.50	601.58	NA	601.58	NA
	1801.76	NA		0.75	601.35	NA	601.35	NA

Table 5.18: Comparison of CPLEX and ADP-B on cap6000, $m = 2176, n = 6000$

could not find feasible solutions within time-limits of 600 and 2,700 CPU seconds, respectively.

Table 5.19 compares CPLEX and ADP-B on enigma. We observe that both B and ADP-B find an optimal solution in 0.83 CPU seconds. CPLEX also finds the optimal solution within that time.

best-known	T(CPX)	V(CPX)	lag-time	CPX-PARAM-EPAGAP	T(B)	V(B)	T(ADP-B)	V(ADP-B)
0.0	0.85	0.0	5	0.25	0.83	0.0	0.83	0.0
	0.86	0.0		0.50	0.83	0.0	0.83	0.0
	0.84	0.0		0.75	0.83	0.0	0.83	0.0
	0.84	0.0	10	0.25	0.84	0.0	0.84	0.0
	0.83	0.0		0.50	0.84	0.0	0.84	0.0
	0.85	0.0		0.75	0.83	0.0	0.83	0.0

Table 5.19: Comparison of CPLEX and ADP-B on enigma, $m = 21, n = 100$

Table 5.17 compares CPLEX and ADP-B on fast0507, a set covering problem arides in railway scheduling. ADP-B could not find a feasible solution within the time limit of 600 CPU seconds. For instance, in the case of `CPX-PARAM-EPGAP`=0.25 and `lag-time`=5, ADP-P actually consumed 765.55 CPU seconds and could not get a feasible solution. On the other extreme, CPLEX achieves a very good solution in 694.98 CPU seconds. This observation signifies CPLEX without a tolerance setting and CPLEX with tolerance setting have obviously different search mechanisms in the branch-and-bound tree.

The Tables from 5.20 to 5.35 continue to suggest the same tendency meaning that

best-known	T(CPX)	V(CPX)	lag-time	CPX-PARAM-EPAGAP	T(B)	V(B)	T(ADP-B)	V(ADP-B)
174	694.98	179	5	0.25	765.55	NA	765.61	NA
	702.39	179		0.50	765.74	NA	765.80	NA
	704.74	179		0.75	767.15	NA	767.21	NA
	696.35	179	10	0.25	762.34	NA	762.40	NA
	699.62	179		0.50	764.91	NA	764.96	NA
	703.80	179		0.75	766.10	NA	766.15	NA

Table 5.20: Comparison of CPLEX and ADP-B on fast0507, $m = 507, n = 63009$

CPLEX achieves a better solution value in shorter time than ADP-P. We persistintly observe that ADP-B improves the solution quality of B at an additional computation time. This tendency can be explained as follows. Due to the nature of B, heuristic B depends on CPLEX performance. In the base-heuristic learning, we call B many times to estimate the optimal values. Once B terminates, CPLEX destroys all the information regarding the branch-and-bound. Hence, whenever B is called, CPLEX starts from scratch. On the other extreme, CPLEX constructs a branch-and-bound tree and uses the same tree to obtain a feasible solution.

best-known	T(CPX)	V(CPX)	lag-time	CPX-PARAM-EPAGAP	T(B)	V(B)	T(ADP-B)	V(ADP-B)
73899798	51.89	72991951	5	0.25	15.95	68282610	58.44	71877908
	52.05	72991951		0.50	15.92	68282610	58.24	71877908
	52.10	72991951		0.75	16.06	68282610	58.15	71877908
	52.01	72991951	10	0.25	16.00	68282610	116.45	71388169
	52.63	72991951		0.50	15.83	68282610	116.93	71388169
	51.78	72991951		0.75	15.82	68282610	116.05	71388169

Table 5.21: Comparison of CPLEX and ADP-B on harp2, $m = 112, n = 2993$

best-known	T(CPX)	V(CPX)	lag-time	CPX-PARAM-EPAGAP	T(B)	V(B)	T(ADP-B)	V(ADP-B)
4722	3.20	4755	5	0.25	1.44	5267	4.70	4942
	3.20	4755		0.50	1.41	5267	4.66	4942
	3.34	4755		0.75	1.39	5267	4.63	4942
	3.17	4755	10	0.25	1.38	5267	6.43	4942
	3.26	4755		0.50	1.40	5267	6.43	4942
	3.22	4755		0.75	1.41	5267	6.45	4942

Table 5.22: Comparison of CPLEX and ADP-B on l152lav, $m = 97, n = 1989$

best-known	T(CPX)	V(CPX)	lag-time	CPX-PARAM-EPAGAP	T(B)	V(B)	T(ADP-B)	V(ADP-B)
1120	0.10	1154	5	0.25	0.11	1154	0.66	1149
	0.10	1154		0.50	0.11	1154	0.46	1154
	0.10	1154		0.75	0.11	1154	0.46	1154
	0.10	1154	10	0.25	0.11	1154	0.92	1149
	0.11	1154		0.50	0.11	1154	0.84	1154
	0.10	1154		0.75	0.11	1154	0.69	1154

Table 5.23: Comparison of CPLEX and ADP-B on lseu, $m = 28, n = 89$

best-known	T(CPX)	V(CPX)	lag-time	CPX-PARAM-EPAGAP	T(B)	V(B)	T(ADP-B)	V(ADP-B)
115155	34.50	116025	5	0.25	35.39	116025	299.63	116025
	35.04	116025		0.50	36.33	116025	295.42	116025
	34.79	116025		0.75	35.23	116025	291.38	116025
	34.94	116025	10	0.25	35.26	116025	579.91	115845
	34.22	116025		0.50	34.96	116025	578.99	115845
	33.88	116025		0.75	35.23	116025	585.50	115845

Table 5.24: Comparison of CPLEX and ADP-B on mitre, $m = 2054, n = 10724$

best-known	T(CPX)	V(CPX)	lag-time	CPX-PARAM-EPAGAP	T(B)	V(B)	T(ADP-B)	V(ADP-B)
307	0.33	364	5	0.25	0.33	364	1.84	364
	0.33	364		0.50	0.03	387	0.60	367
	0.34	364		0.75	0.00	387	0.61	367
	0.38	307	10	0.25	0.34	364	2.18	333
	0.37	307		0.50	0.03	387	1.00	333
	0.37	307		0.75		387	1.00	333

Table 5.25: Comparison of CPLEX and ADP-B on mod008, $m = 6, n = 319$

best-known	T(CPX)	V(CPX)	lag-time	CPX-PARAM-EPAGAP	T(B)	V(B)	T(ADP-B)	V(ADP-B)
6549	0.60	6549	5	0.25	1.05	6549	2.38	6549
	0.60	6549		0.50	1.05	6549	2.38	6549
	0.60	6549		0.75	1.04	6549	2.38	6549
	0.60	6549	10	0.25	1.04	6549	3.13	6549
	0.59	6549		0.50	1.05	6549	3.14	6549
	0.60	6549		0.75	1.04	6549	3.12	6549

Table 5.26: Comparison of CPLEX and ADP-B on mod010, $m = 146, n = 2655$

best-known	T(CPX)	V(CPX)	lag-time	CPX-PARAM-EPAGAP	T(B)	V(B)	T(ADP-B)	V(ADP-B)
16862	17.70	16942	5	0.25	15.18	17514	59.44	16976
	17.60	16942		0.50	15.21	17514	59.59	16976
	17.57	16942		0.75	15.24	17514	59.51	16976
	17.05	17004	10	0.25	15.19	17514	107.33	17040
	17.04	17040		0.50	15.18	17514	107.09	17040
	17.10	17040		0.75	15.20	17514	107.16	17040

Table 5.27: Comparison of CPLEX and ADP-B on nw04, $m = 36, n = 87482$

best-known	T(CPX)	V(CPX)	lag-time	CPX-PARAM-EPAGAP	T(B)	V(B)	T(ADP-B)	V(ADP-B)
3089	0.01	3089	5	0.25	0.02	3089	0.11	3089
	0.02	3089		0.50	0.02	3089	0.10	3089
	0.02	3089		0.75	0.02	3089	0.10	3089
	0.02	3089	10	0.25	0.02	3089	0.14	3089
	0.02	3089		0.50	0.02	3089	0.14	3089
	0.02	3089		0.75	0.02	3089	0.14	3089

Table 5.28: Comparison of CPLEX and ADP-B on p0033, $m = 16, n = 33$

best-known	T(CPX)	V(CPX)	lag-time	CPX-PARAM-EPAGAP	T(B)	V(B)	T(ADP-B)	V(ADP-B)
7615	0.13	8175	5	0.25	0.16	8175	1.79	7915
	0.13	8175		0.50	0.16	8175	1.00	8175
	0.13	8175		0.75	0.17	8175	1.03	8175
	0.13	8175	10	0.25	0.16	8175	2.01	7915
	0.13	8175		0.50	0.16	8175	1.35	7915
	0.13	8175		0.75	0.1	8175	1.35	7915

Table 5.29: Comparison of CPLEX and ADP-B on p0201, $m = 133, n = 201$

best-known	T(CPX)	V(CPX)	lag-time	CPX-PARAM-EPAGAP	T(B)	V(B)	T(ADP-B)	V(ADP-B)
258411	0.20	361226	5	0.25	0.48	263524	5.11	-263524
	0.20	361226		0.50	0.21	361226	2.17	295649
	0.47	263524		0.75	0.20	360885	1.38	361226
	0.19	361226	10	0.25	0.49	263524	6.22	263524
	0.20	361226		0.50	0.22	361226	3.05	262319
	0.47	263524		0.75	0.20	361226	2.59	358917

Table 5.30: Comparison of CPLEX and ADP-B on p0282, $m = 241, n = 282$

best-known	T(CPX)	V(CPX)	lag-time	CPX-PARAM-EPAGAP	T(B)	V(B)	T(ADP-B)	V(ADP-B)
8691	1812.46	11727	5	0.25	733.66	11727	2246.00	9842
	1812.50	11727		0.50	602.56	11727	1811.74	11727
	1809.70	11727		0.75	3.06	11727	239.86	11727
	1831.43	11727	10	0.25	605.02	11727	1823.37	9820
	1832.14	11727		0.50	609.38	11727	1849.79	10392
	1892.86	11727		0.75	3.07	11727	241.90	10392

Table 5.31: Comparison of CPLEX and ADP-B on p0548, $m = 176, n = 548$

best-known	T(CPX)	V(CPX)	lag-time	CPX-PARAM-EPAGAP	T(B)	V(B)	T(ADP-B)	V(ADP-B)
3124	24.96	3413	5	0.25	24.93	3413	65.48	3413
	24.69	3413		0.50	25.03	3413	65.78	3413
	24.74	3413		0.75	24.73	3413	65.03	3413
	24.97	3413	10	0.25	25.08	3413	79.58	3374
	24.74	3413		0.50	25.09	3413	79.46	3374
	24.74	3413		0.75	24.91	3413	79.02	3374

Table 5.32: Comparison of CPLEX and ADP-B on p2756, $m = 755, n = 2756$

best-known	T(CPX)	V(CPX)	lag-time	CPX-PARAM-EPAGAP	T(B)	V(B)	T(ADP-B)	V(ADP-B)
423	307.09	436	5	0.25	322.05	436	1137.92	435
	306.28	436		0.50	322.81	436	1136.74	435
	304.68	436		0.75	326.36	436	1141.32	435
	303.03	436	10	0.25	321.74	436	1484.77	435
	302.11	436		0.50	318.49	436	1479.29	435
	305.39	436		0.75	322.40	436	1492.09	435

Table 5.33: Comparison of CPLEX and ADP-B on seymour, $m = 4944, n = 1372$

best-known	T(CPX)	V(CPX)	lag-time	CPX-PARAM-EPAGAP	T(B)	V(B)	T(ADP-B)	V(ADP-B)
18	0.07	18	5	0.25	0.25	18	0.64	18
	0.06	18		0.50	0.02	19	0.19	18
	0.06	18		0.75	0.02	19	0.18	18
	0.06	18	10	0.25	0.25	18	0.75	18
	0.06	18		0.50	0.03	19	0.22	18
	0.06	18		0.75	0.02	19	0.22	18

Table 5.34: Comparison of CPLEX and ADP-B on stein27, $m = 118, n = 27$

best-known	T(CPX)	V(CPX)	lag-time	CPX-PARAM-EPAGAP	T(B)	V(B)	T(ADP-B)	V(ADP-B)
30	0.97	32	5	0.25	6.08	31	13.30	31
	0.96	32		0.50	0.11	32	0.92	31
	0.97	32		0.75	0.11	32	0.92	31
	0.98	32	10	0.25	6.05	31	13.82	31
	0.10	32		0.50	0.11	32	1.22	32
	0.10	32		0.75	0.11	32	1.20	32

Table 5.35: Comparison of CPLEX and ADP-B on stein45, $m = 331, n = 45$

5.4 Conclusions

Realizing that there are few generic heuristic methodologies for binary integer programming problems, we can consider both PFC and ADP-PFC as important contributions to the integer programming literature. We demonstrated that PFC generates good solutions to randomly generated problems and some of the MIPLIB test problems. Our computational evidence with ADP-PFC and PFC supports the observation that ADP-H improves the performance of H meaning that ADP-H generates better solutions than H. For the MIPLIB test problems, we find CPLEX based branch-and-bound heuristic B quite effective. We oberve that ADP-B improves over B significantly in some instances so that the value of ADP-B attains the best-known solution value.

Chapter 6

Conclusions

We have developed an Approximate Dynamic Programming (ADP) methodology to integer programming problems. We describe and investigate parametric, nonparametric and base-heuristic learning methods in order to approximate the value function in order to break the curse of dimensionality.

Base heuristic learning is a promising methodology for knapsack and binary integer programming problems, particularly randomly generated problems. It provides a flexible framework as it works with a given base-heuristic while improving its performance often significantly as illustrated through percentage improvement. Our computational evidence suggests that base-heuristic learning does have a robust performance as we observe small deviations in computation time, solution quality and storage requirements.

We have proposed a stand-alone adaptive fixing heuristic for the multi-dimensional knapsack problem. Adaptive fixing is computationally practical and generates good solutions compared to some heuristic methodologies from the literature. We observe that ADP with adaptive fixing heuristic is the most promising methodology for multi-dimensional knapsack problem compared to existing methodologies and a state-of-art commercial package like CPLEX. It generates near-optimal solutions for large-scale multi-dimensional knapsack problems with thousands of variables and constraints.

We have developed a pivot-fix-and-complement heuristic methodology for the binary integer programming problems, as an enhancement to pivot-and-complement heuristic from the literature. Our computational evidence suggests that pivot-fix-and-complement is computationally practical for large randomly-generated binary integer programming problems with thousands of variables. We also illustrate its effectiveness on the MIPLIB test problems. It produces good solutions for some of the test problems ranging in size.

Our computational study suggests that statistical learning is a computationally practical methodology for knapsack problems. It generates competitive solutions compared to some heuristic methodologies from the literature but in general it is weaker than the base-heuristic learning. It needs tuning in selection of sampling scheme, sample size, functional form for parametric approximation, and kernel and bandwidth for nonparametric approximation.

In summary, through our extensive computational study we illustrate that our ADP approach to integer programming competes successfully with existing methodologies including state of art commercial package like CPLEX. Our benchmarks for comparison are solution quality, running time and robustness, judged by small deviations in computational resources and solution quality. We explore an integrated approach to solve discrete optimization problems by unifying optimization techniques with statistical learning. Overall, this research illustrates that the ADP is a promising technique by providing near-optimal solutions within reasonable amount of time especially for large scale problems with thousands of variables and constraints. Thus, Approximate Dynamic Programming can be considered as a new attractive alternative to existing approximate methods for discrete optimization problems.

Bibliography

[1] R. Aboudi and K. Jörnsten. Tabu search for general zero-one integer programs using the pivot and complement heuristic. *ORSA Journal on Computing*, 6:82–93, 1984.

[2] E. Balas, S. Ceria, M. Dawande, F. Margot, and G. Pataki. Octane: A new heuristic for pure 0-1 programs. Technical report, GSIA, Carnegie Mellon University, 1997.

[3] E. Balas and C. H. Martin. Pivot and complement - a heuristic for 0-1 programming. *Management Science*, 26:86–96, 1980.

[4] E. Balas and E. Zemel. An algorithm for large zero-one knapsack problems. *Operations Research*, 28:1130–1154, 1980.

[5] R. E. Bellman. *Dynamic Programming.* Princeton University Press, Princeton, N.J., 1957.

[6] R. E. Bellman and S. E. Dreyfus. *Applied Dynamic Programming.* Princeton University Press, Princeton, N.J., 1962.

[7] D. P. Bertsekas and D. A. Castañon. Rollout algorithms for stochastic scheduling problems. Technical Report LIDS-P-2413, MIT, 1997.

[8] D. P. Bertsekas and J. N. Tsitsiklis. *Neuro-Dynamic Programming.* Athena-Scientific, Belmont, 1996.

[9] D. P. Bertsekas, J. N. Tsitsiklis, and C. Wu. Rollout algorithms for combinatorial optimization. *Journal of Heuristics*, 3:245–262, 1997.

[10] D. J. Bertsimas and I. Popescu. Revnenue management in a dynamic network environment. Technical report, MIT, ORC, 2000.

[11] D. J. Bertsimas, C. P. Teo, and R. Vohra. Greedy, randomized and approximate dynamic programming algorithms for facility location problems. Technical report, MIT, ORC, 1998.

[12] D. J. Bertsimas and J. Tsitsiklis. *Introduction to Linear Optimization.* Athena-Scientific, Belmont, 1997.

[13] R. E. Bixby, S. Ceria, C. M. McZeal, and M. P. Savelsbergh. An updated mixed integer programming library miplib 3. Technical Report TR98-03, Rice University, 1998.

[14] J. D. Christodouleas. *Solution Methods for Multiprocessor Network Scheduling Problems with application to railroad operations.* PhD thesis, MIT, ORC, 1997.

[15] P. C. Chu and J. E. Beasley. A genetic algorithm for the multidimensional knapsack problem. *Journal of Heuristics*, 4:63–86, 1998.

[16] V. Chvatal. Hard knapsack problems. *Operaions Research*, 28:1402–1411, 1980.

[17] L. Cooper and M. W. Cooper. *Introduction to Dynamic Programming.* Pergamon Press, Elmsford, New York, 1981.

[18] H. Crowder, E. L. Johnson, and M. W. Padberg. Solving large scale zero-one linear programming problems. *Operations Research*, 31:803–834, 1983.

[19] J. W. Daniel. Splines and efficiency in dynamic programming. *J. Math. Anal. and Appl.*, 54:402–407, 1976.

[20] G. B. Dantzig. Discrete variable extremum problems. *Operation Research*, 5:266–277, 1957.

[21] E. V. Denardo. *Dynamic Programming: Models and Applications.* Printice-Hall, Englewood Cliffs, N.J., 1982.

[22] K. Dudzinski and S. Walukiewicz. Exact methods for the knapsack problem and its generalization. *European Journal of Operational Research*, 28:3–21, 1987.

[23] M. E. Dyer and A. M. Frieze. Probabilistic analysis of the multidimensional knapsack problem. *Mathematics of Operations Research*, 14:162–176, 1989.

[24] H. Everett. Generalized lagrange multiplier method for solving problems of optimum allocation of resources. *Operations Research*, 2:399–417, 1963.

[25] J. Fan and I. Gijbels. *Local Polynomial Modeling and Its Applications.* Chapman & Hall, London, UK, 1996.

[26] A. Freville and G. Plateau. Heuristics and reduction methods for multiple constraints 0-1 linear rpogramming problems. *European Journal of Operational Research*, 24:206–215, 1986.

[27] A. Freville and G. Plateau. An efficient preprocessing procedure for the multidimensional 0-1 knapsack problem. *Discrete Applied Mathematics*, 48:189–212, 1994.

[28] A. M. Frieze and M. R. B. Clarke. Approximation algorithms for the m-dimensional 0-1 knapsack problem: Worst case and probabilistic analyses. *European Journal of Operational Research*, 15:100–109, 1984.

[29] B. A. McCarl G. A. Kochenberger and F. P. Wymann. A heuristic for general integer programming. *Decision Sciences*, 5:36–44, 1974.

[30] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, San Francisco, 1979.

[31] B. Gavish and H. Pirkul. Efficient algorithms for solving multiconstraint zero-one knapsack problems to optimality. *Mathematical Programming*, 31:78–105, 1985.

[32] P. C. Gilmore and R. E. Gomory. The theory and computation of knapsack functions. *Operations Research*, 14:1045–1075, 1966.

[33] F. W. Glover. Heuristics for integer programming using surrogate constraints. *Decision Sciences*, 8:156–166, 1977.

[34] F. W. Glover and G. A. Kochenberger. Critical event tabu search for multidimensional knapsack problems. In *Meta-Heuristics: Theory and Applications*, pages 407–427. Kluwer Academic Publishers, Boston, 1996.

[35] F. W. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, Boston, 1998.

[36] S. Hanafi and A. Freville. An efficient tabu search approach for the 0-1 multidimensional knapsack problem. *European Journal of Operational Research*, 106:659–675, 1998.

[37] F. S. Hillier. Efficient heuristics procedures for integer linear programming with an interior. *Operations Research*, 17:600–637, 1969.

[38] E. Horowitz and S. Sahni. Computing partitions with applications to the knapsack problem. *Journal of ACM*, 21:277–292, 1974.

[39] T. Ibaraki. *Annals of Opeartions Research*, volume 11. J.C. Baltzer AG, Basel-Switzerland, 1987.

[40] O. H. Ibarra and C. E. Kim. Fast approximation algorithms for the knapsack ans sum of subset problem. *Journal of ACM*, 22:463–468, 1975.

[41] A. J. Kleywegt, V. S. Nori, and M. W. P. Savelsbergh. A computational approach for the inventory routing problem. Technical report, School of Industrial and Systems Engineering, Georgia Institute of Technology, 1998.

[42] G. Laporte. The vehicle routing problem: An overview of exact and approximate algorithms. *European Journal of Operational Research*, 59:345–358, 1992.

[43] J. S. Lee and M. Guignard. An approximate algorithm for multidimensional zero-one knapsack problems- a parametric approach. *Management Science*, 34:402–410, 1988.

[44] A. Løkketangen and F. W. Glover. Solving zero-one mixed integer programming problems using tabu search. *European Journal of Operational Research*, 106:624–658, 1998.

[45] R. Loulou and E. Michaelides. New greedy-like heuristics for the multidimensional 0-1 knapsack problem. *Operations Research*, 27:1101–1114, 1979.

[46] M. J. Magazine and M. S. Chern. A fully polynomial approximation schemes for multidimensional knapsack problem. *Mathematics of Operations Research*, 9:244–247, 1984.

[47] M. J. Magazine and O. Oguz. A heuristic algorithm for the multidimensional zero-one knapsack problem. *European Journal of Operational Research*, 16:319–326, 1984.

[48] R. E. Marsten and T. L. Morin. Optimal solutions found for senju and toyoda's 0-1 integer programming problems. *Management Science*, 24:1364–1365, 1977.

[49] R. E. Marsten and T. L. Morin. A hybrid approach to discrete mathematical programming. *Mathematical Programming*, 14:21–40, 1978.

[50] S. Martello, D. Pisinger, and P. Toth. New trends in exact algorithms for the 0-1 knapsack problem. *European Journal of Operational Research*, 123:325–332, 2000.

[51] S. Martello and P. Toth. An upper bound for the zero-one knapsack problem and a branch and bound algorithm. *European Journal of Operational Research*, 1:169–175, 1977.

[52] S. Martello and P .Toth. A mixture of dynamic programming and branch-and-bound for the subset-sum problem. *Management Science*, 30:275–288, 1984.

[53] S. Martello and P. Toth. A new algorithm for the 0-1 knapsack problem. *Management Science*, 34:663–644, 1988.

[54] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementation.* Wiley, Chichester, England, 1990.

[55] M. Meanti, A. H. G. Rinnooy Kan, L. Stougie, and C. Vercellis. A probabilistic analysis of the multiknapsack value function. *Mathematical Programming*, 46:237–247, 1990.

[56] B. Mond and O. Shisha. On the approximation of functions of several variables. *J. Res. Nat. Bur. Stand. Ser. B. Math and Phy.*, 70B:211–218, 1966.

[57] T. L. Morin. *Dynamic Programming and Its Applications*, chapter in Computational Advances in Dynamic Programming edited by M. L. Puterman, pages 53–90. Academic Press, New York, 1978.

[58] T. L. Morin and A. O. Esogbue. The imbedded state space approach to reducing dimensionality in dynamic programs of higher dimensions. *J. Math. Anal. and Appl.*, 48:801–810, 1974.

[59] T. L. Morin and R. E. Marsten. An algorithm for nonlinear knapsack problems. *Management Science*, 22:1147–1158, 1976.

[60] G. L. Nemhauser. *Introduction to Dynamic Programming.* Wiley, New York, 1966.

[61] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization.* Wiley, New York, 1988.

[62] C. D. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, Englewood Cliffs, N.J., 1982.

[63] C. C. Peterson. Computational experience with variants of the Balas algorithm applied to the selection of research and development projects. *Management Science*, 13:736–750, 1967.

[64] N. Piersma. *Combinatoriol Optimization and Empirical Processes.* PhD thesis, The Tinbergen Institute, 1993.

[65] H. Pirkul. A heuristic solution procedure for the multiconstraint zero-one knapsackproblem. *Naval Research Logistics*, 34:161–172, 1987.

[66] D. Pisinger. *Algorithms for Knapsack Problems.* PhD thesis, Dept. of Computer Science, University of Copenhagen, 1995.

[67] D. Pisinger. An expanding-core algorithm for the exact 0-1 knapsack problem. *European Journal of Operational Research*, 87:175–187, 1995.

[68] D. Pisinger. A minimal algorithm for the 0-1 knapsack problem. *Operations Research*, 45:758–767, 1997.

[69] W. P. Powell and J. A. Shapiro. A dynamic programming approximation for ultra large-scale dynamic resource allocation problems. Technical Report SOR-96-06, Statistics and Operations Research, Princeton University, 1996.

[70] C. R. Reeves. *Modern Heuristic Techniques for Combinatorial Problems.* Blackwell Science, Oxford, UK, 1993.

[71] B. Van Roy, D. Bertsekas, Y. Lee, and J. N. Tsitsiklis. A neuro-dynamic programming approach to retailer inventory management. *IEEE Transactions on Control*, 16:1–23, 1998.

[72] S. Sahni. Approximate algorithms for the 0-1 knapsack problem. *Journal of ACM*, 22:115–124, 1976.

[73] H. M. Salkin and K. Mathur. *Foundations of integer programming.* North-Holland, New York, 1989.

[74] U. K. Sarkar, P. P. Chakrabarti, S. Ghose, and S. C. DeSarkar. Improving greedy algorithms by lookahead-search. *Journal of Algorithms*, 16:1–23, 1994.

[75] A. Schrijver. *Linear and Integer Programming.* Wiley, New York, 1986.

[76] N. Secomandi. *Exact and heuristic dynamic programming algorithms for the vehicle routing problem with stochastic demand.* PhD thesis, Faculty of the College of Business Administration, University of Houston, 1998.

[77] S. Senju and Y. Toyoda. An approach to linear programming with 0-1 linear variables. *Management Science*, 15:196–207, 1968.

[78] W. Shih. A branch and bound method for the multiconstraint zero-one knapsack problem. *Journal of the Operational Research Society*, 30:369–378, 1979.

[79] D. K. Smith. *Dynamic Programming: A Practical Introduction.* Ellis Horwood, New York, 1991.

[80] K. Szkatuła. The growth of multi-constraint random knapsacks with various right-hand sides of the constraints. *European Journal of Operational Research*, 73:199–204, 1994.

[81] G. J. Tesauro. Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6:215–219, 1994.

[82] G. J. Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 33:58–68, 1995.

[83] A. Thesen. A recursive branch and bound algorithm for multidimensional knapsack problem. *Naval Research Logistics*, 22:341–353, 1975.

[84] P. Toth. Dynamic programming algorithms for the zero-one knapsack problems. *Computing*, 25:29–45, 1980.

[85] Y. Toyoda. A simplified algorithm for obtaining approximate solutions to zer-one programming problems. *Management Science*, 21:1417–1427, 1975.

[86] A. Volgenant and J. A. Zoon. An improved heuristic for the multi-dimensional 0-1 knapsack problems. *Journal of the Operational Research Society*, 41:963–970, 1990.

[87] S. Walukiewicz. *Integer Programming.* Kluwer Acadamic Publishers, Boston, 1990.

[88] H. M. Weingartner. Capital budgeting of interrelated projects: survy and synthesis. *Operations Research*, 12:485–516, 1966.

[89] H. M. Weingartner and D. N. Ness. Methods for the solution of the multidimensional 0/1 knapsack problem. *Operation Research*, 15:83–103, 1967.

[90] E. Wike. Supply chain optimization: Formulations and algorithms. Master's thesis, MIT, ORC, 1998.

[91] L. A. Wolsey. *Integer Programming.* J. Wiley, New York, 1998.

[92] S. H. Zanakis. Heuristic 0-1 linear programming: an experimental comparison of three methods. *Management Science*, 24:91–104, 1977.