Isto Aho

Interactive Knapsacks: Theory and Applications

ACADEMIC DISSERTATION

To be presented, with the permission of the Faculty of Information Sciences of the University of Tampere, for public discussion in the Paavo Koli Auditorium of the University on November 22nd, 2002, at 12 noon.

DEPARTMENT OF COMPUTER AND INFORMATION SCIENCES
UNIVERSITY OF TAMPERE

A-2002-13
TAMPERE 2002

Supervisor: Professor Erkki Mäkinen
Department of Computer and Information Sciences
University of Tampere

Opponent: Professor Martti Penttonen
Department of Computer Science and Applied Mathematics
University of Kuopio

Reviewers: Professor Ville Leppänen
Department of Computer Science
University of Turku

Professor Jorma Tarhio
Laboratory of Information Processing Science
Helsinki University of Technology

Department of Computer and Information Sciences
FIN-33014 UNIVERSITY OF TAMPERE
Finland

Electronic dissertation
Acta Electronica Universitatis Tamperensis 217
ISBN 951-44-5516-9
ISSN 1456-964X
http://acta.uta.fi

ISBN 951-44-5494-4
ISSN 1457-2060

Tampereen yliopistopaino Oy
Tampere 2002

Abstract

The interactive knapsack problems are generalizations of the classical knapsack problem. Three different new NP-complete decision problems, interactive knapsack heuristic decision (IKHD), interactive knapsack desicion (IKD), and multi-dimensional interactive knapsack (MDIK), are presented for the interactive knapsack model. The interactions occur between knapsacks when an item is put into a knapsack. We identify several natural interaction types. Interactive knapsacks with one item are closely related to the 0–1 multi-dimensional knapsack problem.

By using interactive knapsacks we model various planning and scheduling problems in an innovative way. We show interactive knapsacks to have several applications, for example, in electricity management, single and multiprocessor scheduling, and packing of two, three and n-dimensional items to different knapsacks. Many natural problems related to interactive knapsacks are NP-complete. IKD and MDIK are shown to be strongly NP-complete.

IKHO and IKO are introduced as optimization versions of IKHD and IKD, respectively. IKHO and IKO are shown to be APX-hard. Further, we describe special cases of IKHO and IKO solvable in polynomial time; given an instance parameterized by k, the solution can be found in polynomial time, where the polynomial has degree k. A similar construction solves a special case of the 0–1 multi-dimensional knapsack and the 0–1 linear integer programming problems in polynomial time. We extend the 0–1 multi-dimensional knapsack solution to 0–n multi-dimensional knapsack problems and to 0–n integer programming problems. Our algorithms are based on the resource bounded shortest path search: we represent restrictions efficiently in a form of a graph such that each feasible solution has a path between given source and target vertices.

We apply interactive knapsacks to load clipping used in electricity management. Specifically, we implement several heuristic methods, dynamic programming, enumerative, and genetic algorithms for solving direct load control problem. The enumerative method and dynamic programming are slow while the heuristics and genetic algorithms are faster. The dynamic programming gives best results in reasonable time. Heuristics, however, are several times faster than the other methods.

Acknowledgements

The most important person regarding my studies is Professor Erkki Mäkinen, my supervisor. He got me interested in algorithms near 94, guided me on my Master's thesis 95–96, arranged funding at the University of Tampere after I finished working at the Tampere University of Technology, encouraged me to write Licentiate's thesis 96–98, gave a hint of assistant's position at the university, and pushed me to complete PhD. thesis 99–02.

Working under Erkki's guidance has been rewarding, educative, and interesting. He has quickly and without saving his efforts read and commented my work and papers, thus improving the quality a lot. Without him, this dissertation would not exist. I hope to achieve a similar patience some day. Thank you Erkki!

Reviewers, professors Ville Leppänen and Jorma Tarhio, richly deserve my gratitude. Their notes and comments have given me alternative and valuable views. For example, I was rather surprised when I had to change my view on the relative performance of the algorithms.

Professor Jukka Saarinen and Dr. Harri Klapuri guided me during my time at Tampere University of Technology. They introduced me to Kari Keränen and Juha Räty of Enermet Oy for which some parts of the work were initially done. I want to thank all of them. Their fruitful problem still gives new ideas to me.

Our department has had four heads during these years: professors Kari-Jouko Räihä, Pertti Järvinen, Seppo Visala, and Jyrki Nummenmaa. I am thankful to all of them for providing a stimulating working environment and for supporting important and relaxing personnel free-time activities. I feel immeasurable gratitude for Teppo Kuusisto, Toni Pakkanen, Arto Viitanen, Tuula Moisio, Minna Mäkinen, Minna Parviainen, Marja-Liisa Nurmi, and Taru Koskinen for their continuing technical and administrative assistance and support.

There are several friends and colleagues to whom I am grateful. I wish to mension by name Antti Aaltonen, Anne Aula, Jan-Erik Granbacka, Petri Granroth, Jaakko Hakulinen, Tero Haapakoski, Miroslav Hudec, Pasi Kellokoski, Harri Kemppainen, Jari Kitinoja, Päivi Majaranta, Jouni Mykkänen, Mika Mäenpää, Petri Mäki-Fränti, Erno Mäkinen, Tapio Niemi, Vik Nuckchady, Toni Pakkanen, Timo Poranen, Annukka Ruokolainen, Timo Tossavainen, Kati Viikki, and Arto Viitanen. I wish to thank them since they have occasionally shared their thoughts and time with me. Further, I would like to thank many others not mentioned here, people who I have met and who I meet more or

less regularly in the floorball games and in some other events and occasions. Particular thanks go to Timo Poranen and to Jouni Mykkänen. It has been easy to talk about writing and studying with them.

Finally, I wish to express my thanks to my parents and brother. And to Anna! She has been encouraging and forebearing throughout. I dare not to think the gloomy darkness if it were the case that she were not present.

Academy of Finland and Tampere Graduate School in Information Science and Engineering (TISE) have financially supported the work.

Tampere, October 2002
Isto Aho

Contents

1 Introduction **1**

2 Preliminaries **9**

2.1 Complexity analysis and NP-completeness 9
2.2 Other complexity issues 14
2.3 Knapsack problems 16
2.4 Resource bounded longest path problem 21

3 The model of interactive knapsacks **27**

4 IK problems and their complexities **33**

4.1 Deriving the basic problem 33
4.2 Heuristic approach to IK problems 38
4.3 IKD is NP-complete 42
4.4 On the approximability and fixed tractability 45
4.5 A transformation from GAP into 0–1 MDKP 50

5 Polynomial time instances **53**

5.1 IKHO instances without radiation 53
5.2 IKHO in polynomial time with radiation 55
5.3 IKO in polynomial time without radiation 63
5.4 IKO in polynomial time with radiation 67
5.5 MDKP and IP in polynomial time 72

6 Applications of the IK model **77**

6.1 Applications in scheduling 77
6.2 Other applications 81

7 Multi-dimensional interactive knapsacks **85**

7.1 0–1 MDIK is strongly NP-complete 85
7.2 Applications of 0–1 MDIK 90

8 Application in load clipping **95**
8.1 Introduction to load clipping 95
8.2 Basic model 97
8.3 Payback, restrictions, and goal function 101

9 Methods for load clipping **105**
9.1 Integer composition 105
9.2 Dynamic programming 112
9.3 Properties of dynamic programming 119
9.4 Heuristics 121
9.5 Genetic algorithms 128

10 Experiments **133**
10.1 Instances good to heuristics 137
10.2 Difficult instances 141
10.3 Normal instances 144
10.4 DP state structure 148
10.5 The validity of hypotheses 155

11 Conclusion **157**

A Referenced problems **163**
A.1 Single machine scheduling 163
A.2 Multi-period variable-task-duration assignment problem 164
A.3 Multiple translational containment 165
A.4 Some known NP-complete problems 166

B IK problem repository **169**
B.1 Fixed clone and radiation lengths 169
B.2 Clone lengths depending on item and knapsack 171
B.3 Variable clone lengths 172

C Material for load clipping **173**
C.1 Web resources on energy management 173
C.2 Data for the first test set 173
C.3 Data for the hard instances 177
C.4 Data for normal instances 178

Bibliography **181**

Chapter 1

Introduction

The *interactive knapsack (IK) problem* is a generalization of the classical knapsack problem: an instance of the interactive knapsack problem consists of several knapsacks connected somehow together. These connections are implemented by the various kinds of interactions we shall study. We mainly use two of them, *cloning* and *radiation*, since they appear in several applications. In applications the knapsacks often model discrete time, cloning models some control or task, and radiation models the additional activity caused by the control or task.

In the IK model we have an ordered group of knapsacks, also called a *knapsack array*. By cloning we mean a situation where inserting an item into a knapsack causes the item to be cloned or copied to some near laying knapsacks (in the standard interpretation, clones are identical to the inserted item). Radiation, in turn, does not necessarily make identical copies, but rather takes some proportion of the item to nearby knapsacks.

In other words, when an item j is put in a knapsack i, it is copied into knapsacks $i+1, \ldots, i+c$ ($c \geq 0$). The inserted item and its copies together are called a *clone*. When an item radiates, some portions of it are copied around the item and its copies into knapsacks $i-u, \ldots, i-1$ and $i+c+1, \ldots, i+c+u$ ($u \geq 0$). This behavior (copying to form clone and radiation) can model, for example, the controls made in the electricity management, like in load clipping [1, 2, 3, 6]. We allow the same item to be inserted several times in the knapsack array but not into the same knapsack. In load clipping, this corresponds the situation where we can make several controls with the same utility.

It is not crucial that the copies of items are identical. The main distinction between cloning and radiation is in the restrictions of the optimization problems. Some (real life) problems need both of these interaction types, while some other problems modellable by interactive knapsacks do not conventionally use

both of them. Interactions can, however, bring new insights to these problems and the ways they can be applied.

Most of this work is based on [1, 2, 3, 4, 5, 6]. We introduce the IK model, problems defined in it, complexity analysis of problems and the current boundary between instances solvable in deterministic polynomial time and the NP-complete cases. Further, we study, how the defined problems are related to existing well-known problems, and how we can apply the IK model.

Interactive knapsacks were introduced in [1, 2]. The main motivation was (and it still is) to study the computational properties of load clipping, which has a number of real life restrictions. By simplifying the load clipping problem (and model), we obtain a problem (and model), which is easier to analyze. By adding new features to the simple problems, we approach the load clipping problem. At the same time, the simple problems and their extensions enable us to connect our results to existing combinatorial optimization theory.

The knapsack problem and its variants are much studied combinatorial optimization problems with numerous applications sharing both practical and theoretical interest. We study, how different IK problems are related to other knapsack problems. For instance, both multi-dimensional and interactive knapsack problems involve n items and m knapsacks (which are filled differently). The multi-dimensional knapsack problem (MDKP) is closely related to a 1-item IK optimization problem (that is, to IKHO, interactive knapsack heuristic optimization). MDKP has many applications; for example in situations where different periodic resource requirements must be satisfied over several periods (see [68]), or in query optimization (see [46], and the references of [22]). Lin [68] presents a survey of some well-known nonstandard knapsack problems including MDKP. Many integer and combinatorial optimization problems are formulated as general linear integer programming problems. There exist several monographs devoted to (linear) integer programming (IP) and they describe several applications. For a general introduction, see for example [79].

The model can and should also be presented in stochastic framework because random variables and processes are used in many applications, like stochastic scheduling [78, 88]. Stochastic interactive knapsacks, however, are not covered in this work.

We assume that the readers are familiar with algorithms and complexity analysis. Basic notions and definitions with a few pointers to literature are covered in Chapter 2. Introduction to NP-notation and concepts of complexity analysis is given in Sections 2.1 and 2.2. We follow generally accepted terminology about knapsacks and related problems. This terminology with

literature pointers is recalled in Section 2.3. Especially, we define and discuss the basic properties of several knapsack problems and other problems related to interactive knapsack problems. Section 2.4 gives an introduction to the longest and shortest path problems.

The model of interactive knapsacks along with different types of interactions is introduced in Chapter 3. This is the main innovation of the work. We also discuss some variants of the model. Many problems connected to interactive knapsacks are NP-complete.

Chapter 4 studies the complexity of different interactive knapsack problems. First, we present a basic interactive knapsack problem and discuss about the number of solutions. Next, we present decision problem IKD (interactive knapsack decision), which is strongly NP-complete. After introducing the decision problem, we introduce the corresponding optimization version, namely IKO.

One possible approach to avoid computational difficulties is to restrict the problems to use one item at a time leading to IKHD (– heuristic decision) and IKHO problems, which are also shown to be NP-complete. IKHO can be seen as a special case of 0–1 MDKP (more on MDKP in Section 2.3), although IKHO equals 0–1 MDKP [2] (see Section 4.4). In Section 4.4 we show that IKO is APX-hard. IKD is later generalized to the multi-dimensional interactive knapsack decision (MDIK) problem, which is also strongly NP-complete (see Chapter 7).

After giving the hardness results, we study some special cases in Chapter 5. We show that a number of instances can be solved in polynomial time by giving constructive proofs. Our methods are based on extensive use of the resource bounded longest paths. Our method codes some of the restrictions into the sets of vertices, and some into the weights of the edges, while the profits are coded into the lengths of the edges. The graphs to be constructed are directed and acyclic. These longest path instances can be solved efficiently with the methods developed for the shortest path problem (see Section 2.4).

The reason for using the longest path problems is that their algorithms are easier to apply than to construct the dynamic programming solutions directly to IKHO and IKO. We use the term longest path instead of the shortest, because the use of the term "longest" is more natural in the context of maximizing. We apply the one resource bounded longest path problem in Sections 5.1 and 5.2, and the several resource case in Sections 5.3 and 5.4, while we need them both in Section 5.5.

Section 5.1 gives polynomial time instances to IKHO, where radiation is not used. Section 5.2, in turn, handles polynomial time instances for IKHO, where both the radiation and clone part have nonzero length. The instances

are connected to the cases, where the length of clone and radiation is at most logarithmic with respect to the length of the knapsack array. In load clipping, this corresponds to the case, where the control length is at most logarithmic with respect to the optimization time interval.

After one item case, we turn to IKO in Sections 5.3 and 5.4. First we consider the case with no radiation, and then we handle the restricted length of radiation. The results in this section are generalizations of the corresponding results obtained for IKHO.

The presented techniques are applied to 0–1 MDKP (and 0–1 IP) in Section 5.5. We obtain polynomial time algorithms for the cases, where the restriction matrix is a band matrix and the bandwidth of the matrix is at most logarithmic with respect to the size of the matrix. We also show, how to efficiently solve instances with variables ranging between 0 and $2^n - 1$, where n is fixed, thus giving algorithms for 0–n MDKP (and 0–n IP).

We neglect the question of the sizes of instance parameters (that is, the sizes of weights, profits and knapsack sizes); in the case of load clipping, for instance, we may assume that every parameter including the optimal solution fits into a single memory word. However, the detailed analysis for the MDKP and IP instances should be made. We have checked that the results will not change tremendously, and leave the exact running times open.

In a way, these logarithmic size bounds on structures (the problem still being polynomial time solvable) are the best we can hope, since now some of the parameters of the problems can grow with the other instance parameters. Note that we describe instances for 0–1 MDKP, where both the number of items and the number of knapsacks grow, which, in general, is an APX-hard case. From the practical point of view, the situation is similar to 0–1 IP: if we have an instance with $m \times m$ bandwidth restriction matrix of fixed width w (that is, $w = O(1)$), we can solve the instances in time bounded linearly by $m \log(m)$ and $w2^{2w}$ by using the methods to be presented. However, by applying the dynamic programming solution presented in [96, pp. 264–265], or in [85] to these instances, we end up with much larger graphs and slower running times.

Dynamic programming solution to the 0–1 knapsack problem has been interpreted as a shortest path problem [96, pp. 261-263], or as a maximum-weight path problem [79, pp. 420–422]. The shortest paths have been applied to IP problems as well [96, pp. 264-265]. See also [104] and the graphs therein. The novelty in our results is that we give an efficient transformation of the described instances to a simpler graph problems.

Figure 1.1 summarizes some of the results obtained. For instance, it shows

how IKHO and IKO are related to 0–1 MDKP and generalized assignment problem (GAP). A link shows the restriction, if the number of items n or the number of knapsacks m is set to one, or if the problem changes to other (GAP is obtained from IKO by setting $c = u = 0$). Otherwise, the essential restrictions are in the boxes containing the problem names. Note that MDKP is otherwise the same as IP but in MDKP all parameters are nonnegative.

Chapter 6 demonstrates that some known problems are closely related to interactive knapsacks. Since IKHO and IKO are general problem formulations, they can express several other problems, like 0–1 MDKP and GAP do. In some cases, the methods of Chapter 5 may give efficient solutions to the closely related problems. For example, knapsack arrays can simulate time intervals of any length partitioned into slots of appropriate length [65, 95]. Many scheduling and planning problems involve decisions in this kind of domain (see Chapter 6), like load clipping (see Chapter 8). Moreover, some assignment problems can be presented in a space formed by interactive knapsacks (see Chapter 7). We give several examples on how to apply the interactive knapsacks.

Application of interactive knapsacks to load clipping is presented in detail. This contains the application of the model, algorithms, test runs and comparisons of the algorithms (Chapters 8–10). The implemented algorithms include dynamic programming, several heuristics, enumerative method, and genetic algorithms.

Electricity suppliers can use load clipping in load management. There is a need to limit the load, when the electricity consumption rises above the level the supplier can produce. The supplier has another option: to buy the needed extra electricity from an outer supplier, but this is expensive [6]. The supplier can form *controllable groups* or *loads* from its customers that behave in a similar way when the electricity supply is limited. By controlling these groups, the supplier can lower the peak load for a period, and the objective is to minimize the losses caused by buying electricity from other suppliers, see [6, 27, 102]. Each control corresponds to a clone. There are also other problems in load management that can be modeled with interactive knapsacks. (See Chapter 8.)

Payback (also called *strikeback*) is a phenomenon caused by the consumption peak after the control of customers devices, like electricity heaters or air conditioners [29, 102]. At the control time, the houses cool down (or warm up in the case of air conditioners). The payback means the house temperature stabilization: heaters use more energy right after the control period. The energy storage capability [91] is similar to payback, but it occurs before the control.

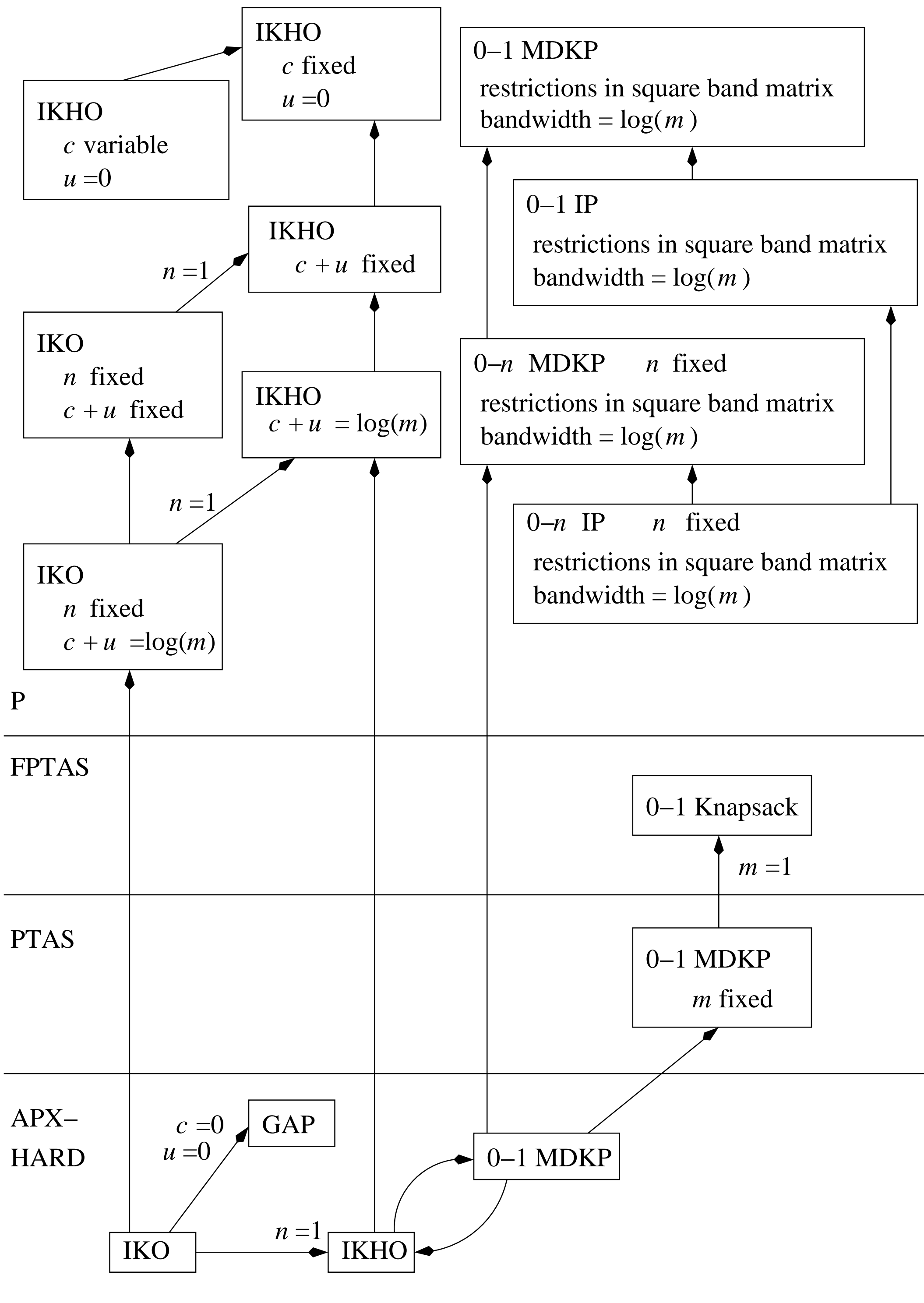

Figure 1.1: Some relationships between the problems considered.

The paybacks and energy storage capabilities are modeled with radiation.

The objective of the supplier is to put items (to decrease consumption by making controls) into the knapsack array (given the energy demand) and take into account the radiation (payback phenomenon of customer devices) in order to appropriately fill the knapsacks (to make the electricity stand while maximizing the profits).

Figure 1.2 depicts a sample control of a controlling utility in realistic and theoretical situations. The control period is seen as a decreased load corresponding to the clone, and after the control period, consumption increases for awhile. This corresponds to the radiation (and to the payback).

Chapter 9 contains different methods for load clipping. Load clipping and some other scheduling and planning problems are often used as "almost real time software". By using previous results calculated, say, ten minutes ago, we can save calculation time with genetic algorithms. Dynamic programming and some other methods, like the heuristics presented in this work, or branch and bound methods, cannot easily take into account the "time window" in which we operate (make control plans). However, genetic algorithms can use earlier outputs given by the algorithm: we just drop away genes describing the solution of the time slots passed out from window, move genes as much as time window moves, and make new random genes for the added time slots (see Chapter 9 and Section 9.5). Tests of the methods are presented in Chapter 10.

To summarize, we will give reasons with this work that it is arguable to study IK model, define optimization problems for it and study their complexity properties, and to find out efficient algorithms and applications for the problems.

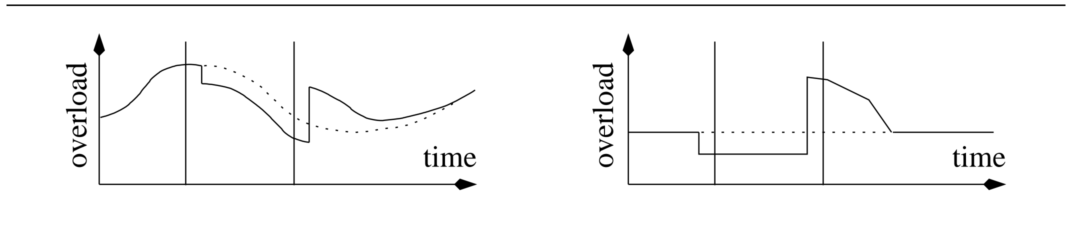

Figure 1.2: A control made in realistic and theoretical load clipping situations.

Chapter 2

Preliminaries

In this chapter we shortly introduce the concept of NP-completeness (Section 2.1), approximability and fixed parameter tractability (Section 2.2), basic knapsack problems and their relatives (Section 2.3), and the resource bounded longest path problem (Section 2.4). An introduction to design and analysis of algorithms can be found, for instance, in [30], and to complexity analysis in [30, 45]. Martello and Toth [72] study extensively different knapsack problems and we follow mainly their notations when describing optimization problems. When we are talking about decision problems, we mainly follow the notations given in [45] (names of decision problems are written in upper case).

$\mathbb{N}$ denotes natural numbers $\{0, 1, 2, \ldots\}$, and $\mathbb{Z}$, $\mathbb{Q}$ and $\mathbb{R}$ denote the sets of integers, rationals and reals, respectively. Empty set is denoted by $\emptyset$, and the cardinality of set N is denoted by $|N|$.

2.1 Complexity analysis and NP-completeness

We usually analyze time consumption as a function of the input size. We define the *time complexity function* to give the maximum time needed by the algorithm to solve a problem instance of certain size. First we define asymptotic notions to give upper and lower bounds for the worst case running times of our algorithms.

Consider functions $f : \mathbb{N} \to \mathbb{R}$ and $g : \mathbb{N} \to \mathbb{R}$. We say that function f is $O(g)$, if there exist positive constants c and n_0 such that $|f(n)| \leq c|g(n)|$, when $n \geq n_0$. Further, f is $\Omega(g)$, if there exist positive constants c and n_0 such that $|f(n)| \geq c|g(n)|$, when $n \geq n_0$.

If $f(n)$ is $O(n^k)$ ($k \in \mathbb{N}$) we say that f is a *polynomial* function. If $f(n)$ is not $O(n^k)$, for any $k \in \mathbb{N}$, we say that f is *exponential* (this definition includes, for instance, $n^{\log n}$, which is not normally regarded as an exponential function).

If the time complexity function of an algorithm is of order $O(p(n))$, where p is a polynomial function, then the algorithm is a *polynomial time algorithm*. The class of problems solvable in polynomial time is denoted by P. There are, however, inherently more difficult problems than the problems contained in P.

If the time complexity of an algorithm cannot be bound by any polynomial $p(n)$, that is, the time complexity is not of order $O(p(n))$ for any polynomial $p(n)$, then the algorithm is an *exponential time algorithm*.

We have problems for which we do not know any deterministic polynomial time algorithm which would solve the problem. Some of these problems constitute a subclass consisting of problems for which we have nondeterministic polynomial time algorithms. This subclass of problems is denoted by NP. Assuming that $P \neq NP$, we call a problem *NP-hard*, if we can transform a problem in $NP \setminus P$ to it in polynomial time. For these problems, we may not know any nondeterministic polynomial time algorithm.

Further, we call a problem *NP-complete*, if it belongs to NP and if it is one-to-one transformable to an NP-complete problem in polynomial time. If we can solve one problem in that class in polynomial time, then we can solve all the other problems in polynomial time by using the transformations.

Efficient solutions to NP-complete problems do not necessarily help us to solve efficiently the NP-hard problem at hand. The other way works: knowing how to solve an NP-hard problem efficiently would give us a way to solve the NP-complete problems efficiently.

Every problem with a polynomial time algorithm belongs to the class of NP-problems. The question about the inequality $P \neq NP$ is an open problem. As an example of NP-complete problems, consider the PARTITION problem [45], which is needed later in this section.

PARTITION
Instance: A finite set N and a size $s(n) \in \mathbb{Z}^+$ for each $n \in N$.
Question: Is there a subset $N' \subseteq N$ such that

$$\sum_{n \in N'} s(n) = \sum_{n \in N \setminus N'} s(n)?$$

To show that a problem Π is NP-complete, we have to show that $\Pi \in NP$, to select a known NP-complete problem $\Pi' \in NP$, and to construct a polynomial transformation f from Π' to Π. There are mainly three techniques to show that a polynomial transformation exists between problems [45]. These are *restriction*, *local replacement* and *component design*.

In restriction we show that a problem contains a known NP-complete problem as a special case. In local replacement we pick some basic units from a

known NP-complete problem and obtain the corresponding instance of the target problem by replacing the basic units with appropriate structure of the target. In component design we use constitutes of the target problem instance to design "components" that can be combined to "realize" instances of a known NP-complete problem. (See [45].)

We use restriction and local replacement. As an example of restriction, suppose that PARTITION is the known NP-complete problem and that we are to show that the SUBSET SUM problem is NP-complete ([45]). First we define the SUBSET SUM problem:

SUBSET SUM
Instance: A finite set N, a size $s(n) \in \mathbb{Z}^+$ for each $n \in N$ and a positive integer B.
Question: Is there a subset $N' \subseteq N$ such that the sum of the sizes of the elements in N' is exactly B?

SUBSET SUM can be seen as a restricted knapsack problem (see Section 2.3). To show that SUBSET SUM is NP-complete by restricting it to PARTITION, we have to show first that SUBSET SUM belongs to NP. We do that by constructing a nondeterministic polynomial time algorithm, which guesses the answer and verifies it, both in polynomial time (see Figure 2.1).

Next, we have to add some restrictions to SUBSET SUM so that the resulting problem is identical to PARTITION, or, that there is an "obvious" one-to-one correspondence between the problems [45]. This correspondence is enough to provide the transformation from PARTITION to SUBSET SUM. We restrict SUBSET SUM to instances where $B = \sum_{n \in N} s(n)/2$. Hence, in SUBSET SUM we have to find a subset N' such that $\sum_{n \in N'} s(n) = B$. However, $B = \sum_{n \in N} s(n)/2 = \sum_{n \in N \setminus N'} s(n)$. Hence, if we can solve SUBSET SUM with the specified restriction in polynomial time we have also a polynomial time solution for PARTITION.

PARTITION and SUBSET SUM as given above are *decision problems*. We can state these problems also as *optimization problems*. For example, the

Input: A SUBSET SUM instance (N, $s(n) \in \mathbb{Z}^+$ for each $n \in N$, and $B \in \mathbb{Z}^+$)
Output: A subset N' solving the instance, if such a subset exists
(1) Guess $N' \subseteq N$
(2) Verify that $\sum_{n \in N'} s(n) = B$ and return N' if it is correct

Figure 2.1: A nondeterministic algorithm solving the SUBSET SUM problem.

SUBSET SUM optimization problem has the same instance as the corresponding decision problem but now we maximize the sum of the chosen elements without exceeding B, that is, we

$$\begin{aligned}\max_{N' \subseteq N} \quad & \sum_{n \in N'} s(n) \\ \text{subject to} \quad & \sum_{n \in N'} s(n) \leq B.\end{aligned}$$

If we could solve this optimization problem in polynomial time, it would give us a polynomial time method to find the answer to the decision version of the SUBSET SUM problem. Hence, the optimization version of SUBSET SUM is NP-hard.

We can exactly solve the SUBSET SUM optimization problem with dynamic programming in $O(B|N|)$ time [72]. Thus, SUBSET SUM is solvable in polynomial time of two variables B and $|N|$. The exponential instances occur when B is much greater than $|N|$.

We also establish results concerning NP-completeness "in the strong sense". Before defining it, we need the concepts of a pseudo-polynomial time algorithm and a number problem. First we define two functions, Length : $D_\Pi \to \mathbb{Z}^+$ and Max : $D_\Pi \to \mathbb{Z}^+$, where D_Π is the set of all instances of problem Π. The function Length corresponds to the number of symbols used to describe $I \in D_\Pi$ under some reasonable encoding scheme for problem Π. The function Max, in turn, is intended to map instance I to an integer corresponding to the magnitude of the largest number occurring in I. (See [45, pp. 92–95].)

For example, in SUBSET SUM we may set $\text{Length}[I] = |N| + \log B + \sum \log s(n)$ and $\text{Max}[I] = B$. We may drop items larger than B from each instance, in which case B is polynomially related to $\max\{B, s(n) : n \in N\}$.

Definition 2.1. *Let I be an instance of* Π. *An algorithm is* pseudo-polynomial time algorithm *if its time complexity function is bounded above by a polynomial function of the two variables* Length$[I]$ *and* Max$[I]$.

Each polynomial time algorithm is also a pseudo-polynomial time algorithm: the running time is polynomially bounded by Length$[I]$ alone. An algorithm runs in pseudo-polynomial time, if its running time is a polynomial function of the length of the data encoded in unary (a one-symbol alphabet) [79, p. 137]. The dynamic programming for the SUBSET SUM optimization problem is a pseudo-polynomial time algorithm, because its time complexity function is polynomially bounded above by B and $|N|$ (as already said, we may remove large items from a problem instance so that B is the largest number) [45].

Definition 2.2. *A decision problem* Π *is a* number problem *if there exists no polynomial* p *such that* $\text{Max}[I] \leq p(\text{Length}[I])$ *for all* $I \in D_\Pi$.

Let Π_p denote the subproblem of Π obtained by restricting Π to only those instances I that satisfy $\text{Max}[I] \leq p(\text{Length}[I])$ and hence, Π_p is not a number problem. If we can solve Π by a pseudo-polynomial time algorithm, then Π_p is solvable by a polynomial time algorithm.

SUBSET SUM is a number problem, because Max can be exponential over Length by increasing B. Moreover, all instances having $B \leq p(|N|)$ are solvable in polynomial time, because now $B|N| \leq p(|N|)|N|$ is polynomially related to $|N|$ and hence, the time complexity of dynamic programming $O(B|N|) \leq O(p(|N|)|N|)$ gives a polynomial time (in $|N|$) upper bound.

Definition 2.3. *A decision problem* Π *is* NP-complete in the strong sense *if* Π *belongs to* NP *and there exists a polynomial* p *over the integers for which* Π_p *is* NP*-complete.*

If Π is NP-complete in the strong sense, it cannot be solved by a pseudo-polynomial time algorithm unless $P = NP$. Intuitively, a problem is strongly NP-complete, "if a problem remains NP-complete even if any instance of length n is restricted to contain integers of size at most $p(n)$, a polynomial" [86, p. 204].

Next we give the sequencing with release times and deadlines (sequencing within intervals) problem, which is NP-complete in the strong sense [45].

SEQUENCING WITH RELEASE TIMES AND DEADLINES
Instance: Set T of tasks and, for each task $t \in T$, a length $l(t) \in \mathbb{Z}^+$, a release time $r(t) \in \mathbb{Z}_0^+$, and a deadline $d(t) \in \mathbb{Z}^+$.
Question: Is there a one-processor schedule for T that satisfies the release time constraints and meets all the deadlines, i.e., a one-to-one function $\sigma : T \rightarrow \mathbb{Z}_0^+$, with $\sigma(t) > \sigma(t')$ implying $\sigma(t) \geq \sigma(t') + l(t')$, such that, for all $t \in T$, $\sigma(t) \geq r(t)$ and $\sigma(t) + l(t) \leq d(t)$?

In the above definition we require that t is executed after $r(t)$, stops before $d(t)$, and does not overlap with other task $t' \in T$. We use this sequencing problem to show that IKD is strongly NP-complete in Section 4.3.

To show that a problem $\Pi \in$ NP is NP-complete in the strong sense, we have to construct a *pseudo-polynomial transformation* from a known strongly NP-complete problem Π' to Π. Suppose that Max and Length are defined in Π. A pseudo-polynomial transformation is a function $f : D_{\Pi'} \rightarrow D_\Pi$ such that

1. for all instances I of Π', f preserves the "yes" answer,
2. f can be computed in polynomial time in the two variables $\text{Max}'[I]$ and $\text{Length}'[I]$,
3. there exists a polynomial q_1 such that, for all instances $I \in D_{\Pi'}$,
$$q_1(\text{Length}[f(I)]) \geq \text{Length}'[I],$$
4. there exists a two-variable polynomial q_2 such that, for all instances $I \in D_{\Pi'}$,
$$\text{Max}[f(I)] \leq q_2(\text{Max}'[I], \text{Length}'[I]).$$

Intuitively, if Π is strongly NP-complete, that is, Π_p is NP-complete for some polynomial p, we can construct a polynomial p' by using the polynomials q_1 and q_2 and properties 3 and 4 giving $\text{Max}'[f(I)] \leq p'(\text{Length}'[f(I)])$. The first and second properties with Π_p imply that f is a polynomial transformation. Thus $\Pi_{p'}$ is strongly NP-complete. (See [45, p. 101].)

2.2 Other complexity issues

If a problem is NP-complete, we often want to know, whether we can approximate the problem. Another question of interest is, if the problem is fixed parameter tractable. A solution given by algorithm A for an instance I is denoted $A(I)$.

Definition 2.4. *A polynomial time algorithm A is said to be an ε-approximation algorithm if for every maximization problem instance I with an optimal solution value $OPT(I)$,*
$$A(I) \geq OPT(I)/\varepsilon.$$

Definition 2.5. *A NP-complete problem belongs to the class APX, if it has an ε-approximation algorithm, where ε is constant.*

Note that only optimization problems can have an ε-approximation algorithm. The approximation ratio is bigger than or equal to 1 by definition. For example, the optimization version of SUBSET SUM has 2-approximation algorithm, in which we order the elements and then choose items in order, the largest first [72]. Hence, SUBSET SUM belongs to APX.

Definition 2.6. *Given an instance I and error bound $\varepsilon \geq 1$, a polynomial time approximation scheme (PTAS) is an ε-approximation algorithm A such that the running time of A is bounded by a polynomial in* $\text{Length}(I)$.

SUBSET SUM is in PTAS [72]. In the next definition we require the algorithm to be bounded by a polynomial $1/(\varepsilon - 1)$ (that is, one has more chances to tune the error).

Definition 2.7. *Given an instance I and error bound $\varepsilon \geq 1$, a fully polynomial time approximation scheme (FPTAS) is an ε-approximation algorithm A such that the running time of A is bounded above by a polynomial in* Length(I) *and* $1/(\varepsilon - 1)$.

SUBSET SUM is also in FPTAS [72]. There are problems that do not belong to PTAS nor FPTAS. Problems that are NP-complete (or NP-hard) in the strong sense may belong to PTAS but do not belong to FPTAS.

Theorem 2.8. *[45, pp. 140–141]. A strongly NP-complete problem does not belong to FPTAS.*

Approximation classes are considered in greater detail for instance in [9, 32]. Arora and Lund [8] review techniques for deriving lower bounds on approximations and give inapproximability results for several classes of problems. (See also [7, 31].)

Definition 2.9. *Let Π and Π' be two maximization problems. A gap-preserving reduction from Π to Π' with parameters (c, p) and (c', p') is a polynomial-time algorithm f. For each instance I of Π, algorithm f produces an instance $I' = f(I)$ of Π'. The optima of I and I', say $OPT(I)$ and $OPT(I')$, respectively, satisfy the following properties:*

$$OPT(I) \geq c \;\Rightarrow\; OPT(I') \geq c',$$
$$OPT(I) < \frac{c}{p} \;\Rightarrow\; OPT(I') < \frac{c'}{p'}.$$

We can show inapproximability results by composing reductions [8]. Suppose that we have a polynomial time reduction g from an NP-complete problem, say PARTITION, to Π that ensures, for every instance I that

$$I \in \text{PARTITION} \;\Rightarrow\; OPT(g(I)) \geq c,$$
$$I \notin \text{PARTITION} \;\Rightarrow\; OPT(g(I)) < \frac{c}{p}.$$

If we want to prove the inapproximability of Π', composing g with the reduction of Definition 2.9 gives a reduction $f \circ g$ from PARTITION to Π' ensuring

$$I \in \text{PARTITION} \;\Rightarrow\; OPT'(f(g(I))) \geq c',$$
$$I \notin \text{PARTITION} \;\Rightarrow\; OPT'(f(g(I))) < \frac{c'}{p'}.$$

In other words, $f \circ g$ shows that achieving an approximation ratio p' for Π' is NP-hard. (See [8].)

Next we define the concept of fixed parameter tractable problem [37, 41]. Let $F(I)$ denote the set of feasible solutions of an instance I. Suppose also that a parameter k is independent of Length(I).

Definition 2.10. *Maximization problem Π is fixed parameter tractable, if there is some deterministic algorithm A, given I, such that*

$$A(I) = OPT(I),$$

if $F(I) \neq \emptyset$; otherwise, the algorithm may loop forever. The running time of A should be bounded by $f(k)\, p(\text{Length}(I))$, where f is some arbitrary function and p is some polynomial.

Function f is typically some exponential function. Idea behind the above definition is that some problems have natural instances, where we can assume k to be small, so that $f(k)$ will be a small constant. For examples, see [37, 42].

Fixed parameter tractable problems constitute class FPT. Downey and Fellows [37] define a problem classes $W[1]$ and conjecture that the containment

$$FPT \subseteq W[1]$$

is proper. We do not define $W[1]$ here, but only note that the question $FPT = W[1]$ is somewhat similar to the question $P = NP$; there are several problems that are hard for $W[1]$, and which are used to show other problems to be hard for $W[1]$. (See [37].)

For instance, SIZED SUBSET SUM is the SUBSET SUM problem with the additional requirement that the sublist L that we are looking for is of size k. Downey and Fellows [37] show that SIZED SUBSET SUM is not fixed parameter tractable, unless $FPT = W[1]$. We say that SIZED SUBSET SUM is $W[1]$-hard, or, that SIZED SUBSET SUM does not belong to FPT.

2.3 Knapsack problems

The *knapsack problem* (KP) of size n can be described with the size b of a knapsack and three sets of variables related to the objects: decision variables $x_1, \ldots, x_n$, positive integers $w_1, \ldots, w_n$, and $p_1, \ldots, p_n$, where, for each $i = 1, \ldots, n$, x_i is either 1 or 0, integer w_i is the *weight* of i and p_i is the *profit* of

i. In the *single knapsack problem* the objective is to

$$\max \quad \sum_{j=1}^{n} p_j x_j \tag{2.1}$$

$$\text{subject to} \quad \sum_{j=1}^{n} w_j x_j \leq b, \tag{2.2}$$

$$x_j = 0 \text{ or } 1, \quad j = 1, \ldots, n. \tag{2.3}$$

This problem is also known as the *0–1 knapsack problem* (or 0–1 KP). As an example, consider items given in Table 2.1. If $b = 30$, we can choose items 1, 5 and 7 and the total weight is 30 and the corresponding value is 45. We achieve a better solution by choosing items 1, 2, 3 and 4, which gives total weight 27 and value 46. If we replace item 4 by item 7, the total weight is 29 and value 51.

The 0–1 knapsack problem above is in the optimization form, and the NP-completeness of the corresponding decision version can be shown as in the example concerning SUBSET SUM in Section 2.1. Optimization version of SUBSET SUM, also called the value independent knapsack problem [72] (because $p_i = w_i$), is an NP-hard special case of 0–1 KP, which is enough to show that 0–1 KP is NP-hard. The other knapsack problems to be introduced in this section are also NP-hard.

Martello and Toth [72] give an account of different exact and approximation methods to 0–1 KP before 1990. Methods include among others, (greedy) heuristics, branch and bound, and dynamic programming. New efficient methods are given in [89]. See also [71].

Suppose the set of objects $\{1, \ldots, n\}$ is partitioned into subsets. If we impose an additional constraint that at most one object per subset is selected, we have the *multiple-choice knapsack problem*. By assuming that several objects of type j can be selected, we have the *bounded* (if restricted by some constant) or *unbounded* (no restriction) *knapsack problem*.

In the *0–1 multiple knapsack problem* we have m knapsacks of capacity

Table 2.1: Example of 0–1 knapsack problem with $n = 7$.

item	1	2	3	4	5	6	7
w_i	1	5	8	13	14	14	15
p_i	1	10	10	25	14	19	30

b_i $(i = 1, \ldots, m)$ with decision variables x_{ij}, where x_{ij} is 1 if the object j is selected for knapsack i and 0 otherwise. Further, if the profit and weight of each object can vary depending on the knapsack for which they are selected, we have the *generalized assignment problem* (GAP)

$$\max \quad \sum_{i=1}^{m}\sum_{j=1}^{n} p_{ij}x_{ij} \tag{2.4}$$

$$\text{subject to} \quad \sum_{j=1}^{n} w_{ij}x_{ij} \leq b_i, \qquad i = 1, \ldots, m, \tag{2.5}$$

$$\sum_{i=1}^{m} x_{ij} \leq 1, \qquad j = 1, \ldots, n, \tag{2.6}$$

$$x_{ij} = 0 \text{ or } 1, \qquad i = 1, \ldots, m, \quad j = 1, \ldots, n. \tag{2.7}$$

Here p_{ij} and w_{ij} are the profit and weight of object j if inserted in knapsack i. In other words, the profit and weight of an item depend on the knapsack, and an item is selected to a knapsack. This problem is generally considered as assigning n jobs to m machines, where p_{ij} is profit, if machine i having b_i resource available is assigned a job j requiring resource w_{ij}. Notice that all values p_{ij}, w_{ij} and b_i are positive integers.

For a survey of GAP, see, for example, [72, 73]. Contrast to our approach, most of the literature handling GAP assume that restriction (2.6) is equality: $\sum_{i=1}^{m} x_{ij} = 1$. The problem with inequality is called LEGAP in [72] and GAP in [23].

Table 2.2 contains an example of GAP. There are tables for profits p_{ij}, weights w_{ij} and knapsacks b_i, where $i = 1, 2, 3$ and $j = 1, \ldots, 5$. Here $n = 5$, $m = 3$ and hence, we have 15 decision variables. By (2.6), we can choose j to belong to any of the knapsacks i only once. This means that each column contains only one nonnull variable x_{ij}.

We cannot put all items in the same knapsack, because the total weights 52, 50 and 58 will exceed the given bounds 35, 20 and 25, respectively. If we

Table 2.2: Example of GAP with $m = 3$ and $n = 5$.

p_{ij}	j					w_{ij}	j					b_i
	18	27	13	4	15		11	8	10	14	9	35
i	21	12	15	2	16	i	6	15	10	4	15	20
	30	8	20	14	17		20	5	14	9	10	25

put items 1, 2, and 3 to the first knapsack (profit 58 and weight 29) and items 4 and 5 to the second knapsack (profit 18 and weight 19), our total profit is 76 and we have fulfilled the restrictions (2.5)–(2.7). This solution is not optimal. If we change the first item from the first knapsack to the third knapsack, our profit will be 88. (We do not present the optimum.)

In the *multi-dimensional knapsack problem* (MDKP) [80, 103], selection of an item means that we add an amount to every knapsack, as opposite to GAP, where the contents of items are inserted in separate knapsacks. In 0–1 MDKP we are to

$$\max \quad \sum_{j=1}^{n} p_j x_j \tag{2.8}$$

$$\text{subject to} \quad \sum_{j=1}^{n} w_{ij} x_j \leq b_i, \qquad i = 1, \ldots, m, \tag{2.9}$$

$$x_j = 0 \text{ or } 1, \qquad j = 1, \ldots, n. \tag{2.10}$$

The parameters are nonnegative integers. In 0–k MDKP equation (2.10) is replaced with $0 \leq x_j \leq k$ and with the requirement that x_j is an integer.

Figure 2.2 has a sample 0–1 MDKP (a) and GAP (b) instances. Selected items bear some weight to every knapsack in 0–1 MDKP, while in GAP, we insert an item in a specific knapsack. In this example, we have $x_1 = x_2 = 1$ in 0–1 MDKP, and $x_{4,1} = x_{9,1} = 1$ in GAP.

The formulation of linear integer programming (IP) is otherwise the same as MDKP, but now we allow negative parameters as well. Thus, p_j, w_{ij}, and $b_i \in \mathbb{Z}$. If we use restriction $x_j = 0$ or 1 for each item, we call the problem 0–1 IP.

MDKP and IP (and 0–1 MDKP and 0–1 IP) have several exact and heuristic algorithms, see [79, 96]; dynamic programming [80, 103] and genetic algorithms

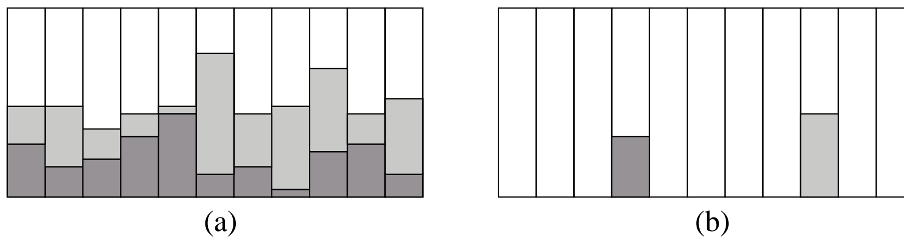

Figure 2.2: Two items selected in 0–1 MDKP (a) and two items inserted in two knapsacks in GAP (b).

[56], to name few. Osorio *et al.* [84] give an overview of MDKP referencing several paradigms, including branch and bound, enumeration, different relaxation techniques with branch and bound, like Lagrangean, surrogate, composite and linear programming, and different heuristics like tabu search and genetic algorithms, to solve the problem. Other extensive surveys can be found in [68], [17] and [26]. Population heuristics include some of the above methods [11]. Hill climbing is considered in [57]. Further, Dyer *et al.* [39] consider the number of solutions to MDKP. Thesen [100] studies 0–1 MDKP with constraint (2.11). Note that 0–1 MDKP and 0–1 IP have enumerative pseudo-polynomial time algorithms running in $O(mn2^n)$ time (see [79, p. 125], and [85]). (The dynamic programming algorithms are the most relevant to our work.)

A fixed parameter algorithm for the classical 0–1 knapsack problem has been considered in [41]. Based on the fact that SIZED SUBSET SUM is not in FPT, Caprara *et al.* [18] argue that a k-item KP is not fixed-parameter tractable. A *k-item knapsack problem* is (2.1)–(2.3) with an additional constraint on the number of items:

$$\sum_{j=1}^{n} x_j \leq k. \tag{2.11}$$

We use (2.1)–(2.3) with (2.11) in Section 4.4 to show that IKHO and IKO are not fixed parameter tractable. Note that k-item KP is a 0–1 MDKP, where $m = 2$, and hence, 0–1 MDKP is not in FPT [18].

If m is fixed, 0–1 MDKP is in PTAS [21, 44, 69]. Chekuri and Khanna [22] show that 0–1 MDKP is hard to approximate within a factor of $\Omega(m^{(1/(\lfloor B \rfloor+1))-\varepsilon})$, for every fixed ε, where $B = \min_i b_i$. Srinivasan [98] shows how to obtain $\Omega(t^{B/(B-1)})$ solutions in polynomial time, where $t = \Omega(z^*/m^{1/B})$ and z^* is the optimal solution (and $B > 1$). GAP is APX-hard, but it can be 2-approximated [23, 97]. The APX-hardness is shown for a very restricted case of GAP [23]. We show that IKHO and IKO are APX-hard in Section 4.4.

An extension of the knapsack problem is the 0–1 *collapsing knapsack problem*, where we allow the capacity of the knapsack to depend on the number of items it contains [87]. Restriction (2.2) has now the form

$$\sum_{j=1}^{n} w_j x_j \leq b\Big(\sum_{j=1}^{n} x_j\Big).$$

We can extend the objective function of the knapsack problem similarly and define the *added value knapsack problem* by changing objective (2.1) to be

$$\max \sum_{i=1}^{m} p_i\Big(\sum_{j=1}^{n} x_{ij}\Big).$$

Here we let the profit of the items in knapsack i to depend on the number of the items. Still another possibility is to define

$$\max \sum_{i=1}^{m} p_i\Big(\sum_{j=1}^{n} w_{ij}x_{ij}\Big), \tag{2.12}$$

so that the profit of knapsack i depends on the total weight of the items it contains. We call also (2.12) added value knapsack objective: in this work there is no need to distinguish between these two objectives. The author is not aware of any references in the knapsack literature for the two added value objectives given above. If they are new, at least the latter is not artificial, because it is used in load clipping.

2.4 Resource bounded longest path problem

Most of the polynomial time instances we describe in Chapter 5 are based on the longest path methods. Let $G = (V, E)$ be a graph, where each edge $e \in E$ has length $l_e \in \mathbb{N}$ and weight $w_e \in \mathbb{N}$. The *longest weight-constrained path problem* is to find the longest path P between given vertices such that the weight constraint is not violated: $\sum_{p \in P} w_p \leq K$. Note that the longest and shortest weight-constrained path problems are not equivalent in general; for example, while they both are NP-complete, they do not have similar special cases solvable in polynomial time [2], which is demonstrated at the end of this section.

Fortunately, the graphs that we consider are acyclic and directed. Hence, the *longest weight-constrained path* problem can use the methods of *resource bounded shortest path* problem, which is an extensively studied problem. Neglecting the resource constraints, the shortest path is always well-defined for acyclic directed graphs as they will not contain any negative cycles [30, p. 515].

Let $l_e < 0$ be the minimum edge length in a graph G. We can form a new instance by adding $|l_e|$ to each length. Now the instance has nonnegative edge lengths and the original resource bounded shortest path instance corresponds to this new one. They will give the same path as the result. To see this, consider $\min(M + l)x = \min(Mx + lx)$, where x denotes the 0–1 vector of selected edges, M the added length and l the vector of lengths. Now the Mx part does not contribute to the choice of x. If the original problem was to find the longest weight-constrained path, we can multiply every length by -1 and then use the above transformation and a resource bounded shortest path method. We do not have to change the weights.

Other names used in the literature for the resource bounded shortest path problem are the *restricted shortest paths* and the *shortest weight-constrained paths*. The shortest weight-constrained path problem (with one weight constraint) is in FPTAS [53]. Lagrangean relaxation, k-shortest paths, and dynamic programming have also been applied, see [38] for an overview. Ziegelmann [108] gives a survey of the constrained shortest paths. We start with the following restricted special case.

Theorem 2.11. *[45, p. 214]. Consider a graph $G = (V, E)$ with each edge e having a length l_e and a weight w_e, where $l_e, w_e \in \mathbb{N}$. If the weights are equal and the graph is acyclic and directed, the longest weight-constrained path can be found in polynomial time.*

We can relax the assumption of equal weights when we construct algorithms. (And as already said above, since we suppose that the graphs are acyclic and directed, we can use the shortest weight-constrained paths methods.)

By redesigning the dynamic programming solution of the knapsack problem, we can solve the longest weight-constrained path problem in $O(K|V|^2)$ time. We advise the reader to see [96, pp. 261–263] and [79, pp. 420–422]. There the graphs consist of one parameter edges (length) and the second parameter (weight) is coded into the labels of the vertices. Other possibility is to directly form graphs, whose edges have both length and weight, which is demonstrated below.

However, the shortest weight-constrained path can be solved in $O(K|E|)$ time, if the weights are positive (see [38] and the references therein). This holds also for acyclic directed graphs, and therefore the time bound is valid for the longest weight-constrained paths in acyclic directed graphs. We will use the $O(K|E|)$ bound later. Note that since the number of edges $|E|$ can be as large as $|V|(|V|-1)$ in a directed acyclic graph, in general, the space consumption and running time are $O(K|V|^2)$ for the longest weight-constrained path problem in acyclic directed graphs. Hence, we obtain the following result.

Theorem 2.12. *Consider an instance of the longest weight-constrained path problem with a graph $G = (V, E)$, where each edge e has length l_e and weight w_e, where $l_e, w_e \in \mathbb{N}$. Suppose further that the weight-constraint K is polynomial on the number of vertices $|V|$. If G is acyclic and directed, the longest weight-constrained path between two vertices can be found in polynomial time.*

We also need the *longest r-weight-constrained path problem*. Let $G = (V, E)$ be a graph, where each edge $e \in E$ has length $l_e \in \mathbb{N}$ and r separate weights $w_{e1}, \ldots, w_{er}$. The longest r-weight constrained path problem is to find

the longest path P between given vertices such that the weight constraints are not violated: $\sum_{e \in P} w_{ej} \leq K_j$, where $j = 1, \ldots, r$.

Because our graphs are directed and acyclic, we can use the shortest r-weight constrained path problem, which can be solved in $O(|V||E|d)$ time with dynamic programming [12], where d is the number of distinct resource vectors. The problem handled by Beasley and Christofides in [12] is slightly more general than ours (they have resource bounds in vertices as well). Further, we have $d = K^n$, or $d = K_1 \cdots K_n$, if we use separate bounds for inclusion times for the items, which means that d is bounded if n is fixed (in our case each K_i will be polynomial on the number of vertices).

We write down a similar folklore result about the running time, which suits our purposes better, since $|V||E|$ can be larger than $|V|^2$ in the graphs we construct. Further, as will be discussed in Section 5.5, 0–1 MDKP can be solved by the dynamic programming presented below.

Lemma 2.13. *Let $G = (V, E)$ be a graph with weight constraints $K_1, \ldots, K_r \in \mathbb{N}$, and with each edge e having length $l_e \in \mathbb{N}$ and weights $w_{e1}, \ldots, w_{er} \in \mathbb{N}$. If G is directed and acyclic, the longest r-weight constrained path problem between given vertices can be solved in $O(K_1 \cdots K_r |V|^2)$ time and space.*

Proof. We apply dynamic programming closely following the proof given by Bertsekas [13, p. 53]. Let $\{1, \ldots, |V| - 1, t\}$ be the set of vertices, and let t be the target vertex. If vertices i and j are not incident, then $l_e = w_{ek} = \infty$, where $k = 1, \ldots, r$.

Let $f_p(i, k_1, \ldots, k_r)$ be the minimum length from vertex i to vertex t in $|V| - p$ moves with total weights $k_1, \ldots, k_r$. The optimal path having weights $k_1, \ldots, k_r$ from vertex i to t is $f_1(i, k_1, \ldots, k_r)$. Each edge has at most $K_1 \cdots K_r$ different weight combinations.

Initialize $f_{|V|-1}(i, w_{e,1}, \ldots, w_{er}) = l_e$, where $i = 1, \ldots, |V| - 1$ and $e = (i, t)$. If there is no $e = (i, t)$ having weights $k_1, \ldots, k_r$, then $f_{|V|-1}(i, k_1, \ldots, k_r) = \infty$. The shortest distance from i to t in $|V| - p$ moves is

$$f_p(i, k_1, \ldots, k_r) = \min_{j=1,\ldots,|V|-1} \{ l_e + f_{p+1}\big(j, k_1 + w_{e,1}, k_2 + w_{e,2}, \cdots, k_r + w_{er}\big) \},$$

where $p = 1, \ldots, |V| - 2$ and $e = (i, j)$.

The optimal choice when at vertex i after p moves is to continue to a vertex j^* that minimizes $l_e + f_{p+1}(j, k_1 + w_{e,1}, k_2 + w_{e,2}, \cdots, k_r + w_{er})$, where $e = (i, j)$, over all $j = 1, \ldots, |V| - 1$. The above dynamic programming algorithm solves the shortest r-weight-constrained path problem. We can ignore the dynamic program tabulations $f_p(i, k_1, \ldots, k_r)$, where $k_q > K_q$, for some $q = 1, \ldots, r$.

Next we consider the running time. Because G is acyclic and directed, the graph has at most $|V|(|V|-1)/2$ edges. This is also the maximum length of a path in G.

To find the longest r-weight constrained path, we can store the longest path p of each r-weight combination so far found in each vertex. The number of these combinations is $(K_1+1)\cdots(K_r+1)$ and we can store the combinations in a $(K_1+1)\cdots(K_r+1)$ dimensional table contained in each vertex.

If there are two different paths having different lengths but both having the same r-weight combination, we store the longer one into the table. Hence, the space consumption is $O(K_1\cdots K_r|V|^2)$. We start from the source vertices and proceed towards the sink vertices. In each vertex, we check every incoming edge. There are at most $|V|^2$ edges, and thus, the running time is $O(K_1\cdots K_r|V|^2)$. □

At the end of this section, we demonstrate that the longest weight-constrained path instances, in general, cannot use the algorithms of the shortest weight-constrained path problem. The longest weight-constrained path problem is defined in decision form as follows.

LONGEST WEIGHT-CONSTRAINED PATH (LWCP)
Instance: Graph $G=(V,E)$, length $l(e)\in\mathbb{N}$, and weight $w(e)\in\mathbb{N}$ for each $e\in E$, specified vertices $s,t\in V$, positive integers K,W.
Question: Is there a simple path in G from s to t with total weight W or less and total length K or more?

LWCP is NP-complete (also for directed acyclic graphs), which can be shown easily with the knapsack problem. Let p_i and w_i be the profits and weights and K and W be positive integers. The capacity of the knapsack is W and we are asking whether there is a set of items fitting in W while the sum of profits is K or more. We construct a graph containing two set of edges. Each lower edge between consecutive vertices has length 0 and weight 0, while the upper set of edges has lengths p_i and weights w_i. Figure 2.3 shows an example of the graph. If we are able to find the path between vertex 0 and m with length K or more and weight W or less, we have solved the knapsack

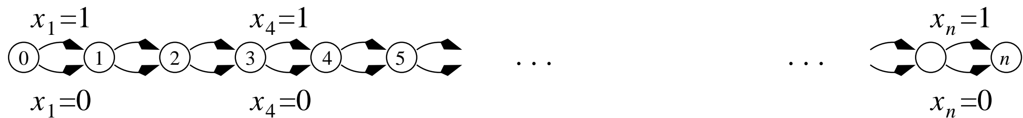

Figure 2.3: Graph construction for 0–1 KP and for 0–1 MDKP.

problem. This graph can be used to solve both the longest weight-constrained path problem as well as the longest r-weight-constrained path problem (in the case of 0–1 MDKP). In the latter case the weights w_i are r-vectors. Note that if we are to solve knapsack type problems, the graph can be constructed differently (as described in the discussion preceding Theorem 2.12).

The following two special cases are solvable in polynomial time for the shortest weight-constrained path problem [45], while for the longest paths they are not. Hence, the problems with the shortest and longest paths are essentially different, in general.

Let w be the common weight of all edges of graph G. Set $W' = \lceil W/w \rceil$ and $w' = 1$. Values w and W will give the same answer as values w' and W' for the longest weight-constrained path problem. Hence, we are to find a simple path of length K or more consisting of at most W' edges. If W' equals to the number of vertices plus one, this problem is same as the longest path problem and hence cannot be solved in polynomial time in general.

If, on the other hand, all lengths are equal, we have to find a simple path having at least $K' = \lceil K/l(e) \rceil$ edges and having total weight at most W. If K' is the number of vertices plus one, this is the Hamiltonian path problem meaning that this subproblem is not solvable in polynomial time.

Chapter 3

The model of interactive knapsacks

In this chapter we describe the model of interactive knapsacks. We discuss different alternatives to build up the model. The actual choice is made in the next chapter, when we define problems for the model.

In order to have any interaction between knapsacks we have to have at least two knapsacks. We suppose that there are m knapsacks. Let the knapsacks be ordered so that knapsack i is before knapsack $i+1$ and $i+1$ is after i. We also say that i is left from $i+1$ and $i+1$ is right from i. A set of knapsacks $i, i+1, \ldots, j$ (in order) is denoted by interval $[i, j]$ of *length* $j-i+1$. Hence, knapsack i corresponds to interval $[i, i]$. The ordered set of all knapsacks is also called a *knapsack array*.

The idea of interaction can be applied in many ways. The knapsacks can have some restrictions which make them interactive. Also the application of decision variables for choosing items into the knapsacks can imply interaction. In this work we mainly study the former type of interaction. Sometimes, however, we cannot clearly distinguish different forms of interaction from each other. Yet, there is a third kind of interaction type: knapsacks can be grouped into knapsack sets and knapsacks in one set may have identical weights and profits for an item while different knapsack sets may differ from each other. In each group, we are interested in the average filling, and calculate the average profits and weights for the group. An example of the averaging interaction can be found in Chapter 8.

If we select object j for knapsack i we consume the capacity b_i of knapsack i as in the knapsack problems. The interaction of the selection is a function $I_{ij} : \{1, \ldots, m\} \to \mathbb{Q}$. Selecting j for i has an effect $I_{ij}(k)$ to other knapsacks $k \neq i$ and the effect also depends on the "target" of the interaction. Usually

$I_{ij}(k)$ gives an amount, either positive or negative, to be added in the knapsack k. Typically, that amount is some proportion of item j added to knapsack i affecting knapsack k, that is, we add to the profit and weight of knapsack k values

$$p_{kj} I_{ij}(k) \quad \text{and} \quad w_{kj} I_{ij}(k), \tag{3.1}$$

respectively. Other possibility is to add values

$$p_{ij} I_{ij}(k) \quad \text{and} \quad w_{ij} I_{ij}(k), \tag{3.2}$$

respectively, or some mixture of the above values. Note that in applications we usually have $I_{ij}(i) = 1$.

Next we give a sufficient condition, when we can form profits or weights (3.2) from (3.1), and vice verse. Note that an interaction function I for item j can be presented in matrix form. Without loss of generality, we assume that I_i, p_i (corresponds to the values of (3.1)) and I', p'_i (corresponds to the values of (3.2)) are defined for item j.

Theorem 3.1. *Assume that $\sum I'_i(k) \neq 0$, for $i = 1, \ldots, m$, and that matrix I representing interactions of item j is invertible. The profit indexing* (3.1) *$p_k I_i(k)$ can be obtained from the profit indexing* (3.2) *$p'_i I'_i(k)$, and vice verse.*

Proof. First, sum over (3.2) is $\sum_{k=1}^{m} p'_i I'_i(k) = p'_i \sum I'_i(k)$, for $i = 1, \ldots, m$. Let $q_i = \sum_{k=1}^{m} I_i(k)$, for $i = 1, \ldots, m$. Now, the m sums in vector notation is $p'q$, where $p' = (p'_1 \ \cdots \ p'_m)$ and $q = (q_1 \ \cdots \ q_m)$.

Sum (3.1) $\sum_{k=1}^{m} p_k I_i(k)$, for $i = 1, \ldots, m$, is Ip in vector notation, where I is $m \times m$-matrix and $p = (p_1 \ \cdots \ p'_m)$.

Now, we may set $Ip = p'q$ giving $p'_i = [Ip]_i / q_i$. Hence, we obtain profits from (3.1) for (3.2), if $q_i \neq 0$. If I is invertible, we have $p = I^{-1} p'q$, and thus, we obtain profits from (3.2) for (3.1). □

Theorem 3.2. *Similarly, with the assumptions that $\sum I'_i(k) \neq 0$, for $i = 1, \ldots, m$, and that matrix I representing interactions of item j is invertible, the weights in* (3.1) *and in* (3.2) *can be obtained from each other, while keeping the interactions unchanged.*

If interaction function I for item j is appropriate, by Theorems 3.1 and 3.2, we may use the indexing best suiting for our purposes (including mixed indexings).

In some applications interaction I_{ij} is a function of the distance between the knapsacks i and k. For instance, we may have $I_{ij}(k) = 0$ when $|k - i| > u$, where u is the interaction distance.

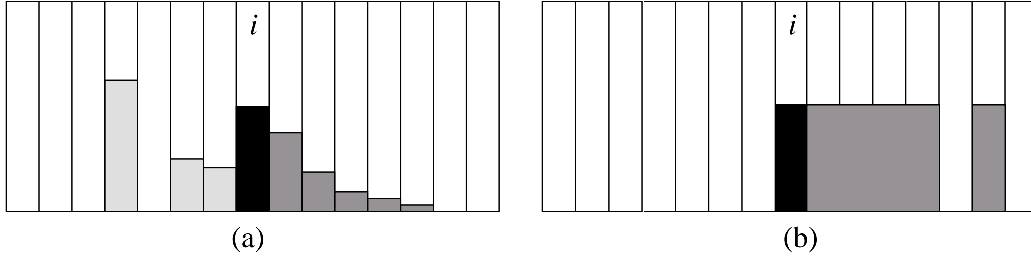

Figure 3.1: Example of radiation (a) and cloning (b).

Figure 3.1 (a) illustrates the case where j (in black) is inserted to i. The gray areas describe the extent of the interaction. Hence, the insertion of an object affects other knapsacks as well. We denote this interaction as a function $r_{ij}(k) : \{1, \ldots, m\} \to \mathbb{Q}$. Here $r_{ij}(k) = 0$, when $|k - i| > 5$, and hence $u = 5$. The amount to be added (gray areas) to profits is $p_{kj}r_{ij}(k)$ and to knapsack contents $w_{kj}r_{ij}(k)$ when $|k - i| \leq 5$. We can also use $p_{ij}r_{ij}(k)$ and $w_{ij}r_{ij}(k)$ or a mixture $p_{kj}r_{ij}(k)$ and $w_{ij}r_{ij}(k)$, which are used later. The shape of an interaction can be arbitrary, like in the left side of i (in light gray). However, in practice the shape usually follows some regular pattern like in the right side of knapsack i (in dark gray).

Figure 3.1 (a) also suggests that we should not apply this interaction scheme recursively. For example, consider the effect $r_{i-1,r_{ij}(i-1)}(k)$, where $k \neq i-1$, of knapsack $i - 1$ after the inclusion related to $r_{ij}(i - 1)$. The interaction could be defined in this way but known applications do not derive advantage from recursive interactions.

The interaction r described above is called *radiation*. The radiation at the left side of (knapsack) i (in light gray) is called the *left radiation* and at the right side (in dark gray) the *right radiation*. The knapsacks involved in radiation caused by object j at knapsack i are denoted by $R_{ij} = [i - u, i - 1] \cup [i + 1, i + u]$. Note that there is no reason why we could not use negative radiation; indeed, in applications we need negative radiation. For radiation $r_{ij}(k)$, we typically have $-2 < r_{ij}(k) < 2$.

Next we describe *copying*, the other basic form of interaction. If we put item j into knapsack i, the restrictions between knapsacks may imply that we also have to put j into some other knapsacks. In Figure 3.1 (b) we have an example of this kind of interaction. By inserting j (in black) into knapsack i, we also have to *copy* j into knapsacks $i + 1, \ldots, i + 4$ and $i + 6$.

Mathematically, this interaction is similar to I. This time I is replaced with a function $c_{ij}(k) : \{1, \ldots, m\} \to \{0, 1\}$. The interaction distance is denoted by

c. We could define this in the same manner as we defined radiation, introducing the left and right copies. However, in applications we mainly need right copies. If the weight of item j inserted into knapsack i is w_{ij}, we have $c_{ij}(k) = 1$, when $i+1 \leq k \leq i+c$, otherwise $c_{ij}(k) = 0$. This means that we add the copies to knapsacks k, for $i+1 \leq k \leq i+c$, which we see as $p_{kj}c_{ij}(k) = p_{kj}$ and $w_{kj}c_{ij}(k) = w_{kj}$ or as $p_{ij}c_{ij}(k) = p_{ij}$ and $w_{ij}c_{ij}(k) = w_{ij}$. The other knapsacks are kept unchanged.

Usually we are more interested in copying patterns without holes, unlike in Figure 3.1 (b). Reasons for this become clear in the following chapters. Thus, it is also sufficient to tell the insertion of j for the leftmost knapsack i and the number of copies $c \in \mathbb{N}$. Hence, we only have to copy j for knapsacks $i+1, \ldots, i+c$. We say that the whole set of copies with the inserted item contained in the knapsacks $i, \ldots, i+c$ is a *clone*. Clone refers also to the knapsacks $i, \ldots, i+c$. Several clones form a *clone family*. The knapsacks involved in copying object j starting at i are denoted by $C_{ij} = [i, i+c]_j$. Clone C_{ij} also contains the knapsack into which the insertion was made. A clone family is denoted by C meaning

$$C = \bigcup_{i,j} C_{ij}.$$

For example, $C = \{[1,5]_1, [10,15]_1, [1,15]_2\}$ is a clone family containing three clones, of which two are of item type 1 ($C_{1,1}$ and $C_{10,1}$) and one is of type 2 ($C_{1,2}$).

Hybrid models with both radiation and copying are usual. This time we assume that radiation spreads around clone C_{ij} instead of knapsack i. This means that copying is not involved with radiation itself and that we do not separate the first knapsack i, into which the insertion is made, from the other items taking part in copying by applying the radiation only around i. Hence, the interaction in hybrid models involves knapsacks

$$I_{ij} = C_{ij} \cup R_{ij} = [i-u, i-1]_j \cup [i, i+c]_j \cup [i+c+1, i+c+u]_j.$$

We define the described hybrid model to be the *0–1 interactive knapsacks model* or the 0–1 IK model shortly.

Figure 3.2 has item j inserted into knapsack i, for which we have both a clone (in black) and radiation (in gray). There is also item k in knapsack $i-4$ involving clone $[i-4, i-1]_k$ (boxed) of length 4 which does not have any radiation. The early radiation of length 3 (in light gray) of item j is negative: it has cut down the contents of knapsacks $[i-3, i-1]$. The back radiation of length 5 of item j is positive and affects knapsacks $[i+4+1, i+4+5]$.

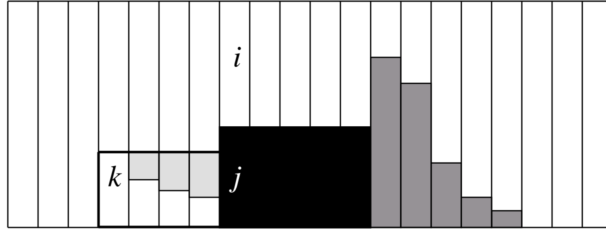

Figure 3.2: Radiation and cloning together.

Function r_{ij} is approximately -0.4, -0.5 and -0.55 for $r_{ij}(i-3)$, $r_{ij}(i-2)$ and $r_{ij}(i-1)$, and 1.5, 1.4, 0.7, 0.4 and 0.3 for $r_{ij}(i+5), \ldots, r_{ij}(i+9)$, respectively. We apply radiation with rule $w_{ij}r_{ij}(\cdot)$. The length of clone $C_{ij} = [i, i+4]_j$ is 5. Here the copying is applied for the resources with rule $w_{ij}c_{ij}(k) = w_{ij}$, when $k \in C_{ij} = [i, i+4]_j$.

Clearly, radiation is more general than copying. The approach to divide interaction to copying and radiation comes from load clipping application (see Chapter 8), where copying corresponds to the control and radiation to the payback. This can be seen in the decision and optimization problem formulations in the next chapter; copy part of the clone has its own restrictions. For example, two copy parts are not allowed to overlap while the radiation parts are.

The model of the 0–1 interactive knapsacks can be generalized to the 0–n interactive knapsacks model meaning that the same object j may be copied into the same knapsack at most n times. Here we have two options for handling the copies of a clone. We can allow the clones connected to object j to have different lengths, like in Figure 3.3 (a). Other possibility is that the clones have equal length as in Figure 3.3 (b). The dotted line separates items from their copies.

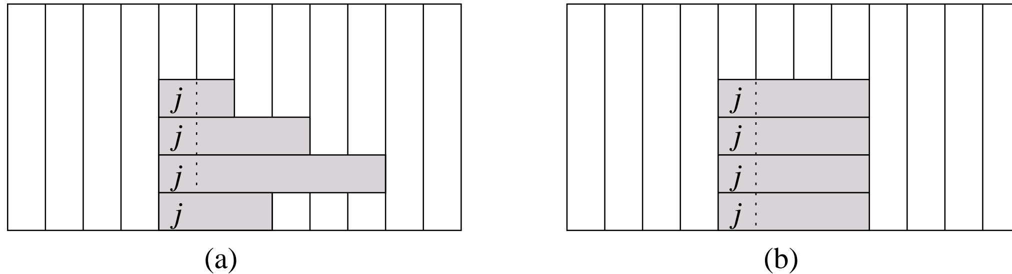

Figure 3.3: Examples of 0–n interactive knapsacks.

Moreover, we may partition the set of objects into subsets and restrict the selections so that at most one object is selected from any subset. Hence, we have the *interactive multiple-choice knapsacks model*.

The added value knapsack objective (2.12) is demonstrated in Figure 3.4. It contains two versions of the knapsack arrays. Both of these use cloning and radiation just like knapsack arrays (darker triangles in (a) depicts short radiation).

We give desired filling rate for knapsacks. The objective level B can be same for all (a) or it can depend on a knapsack (b), in which case we need values B_i, where $i = 1, \ldots, m$.

If the predefined filling rate is exceeded (b), some penalty is given. Otherwise, like in (a), the profits are calculated normally. The penalty can be formalized in several ways. For instance, let a be a positive constant scaling the disadvantage caused by exceeding B. Now the objective may use profits

$$p_i(x) = \begin{cases} x, & \text{when } x \leq B, \\ x + a(B - x), & \text{otherwise.} \end{cases}$$

Input of function p_i usually depends on the items put into knapsack i. We may use the weights ($\sum w_{ij} x_{ij}$, as in Figure 3.4 (a) and (b)) or the number of items put ($\sum x_{ij}$).

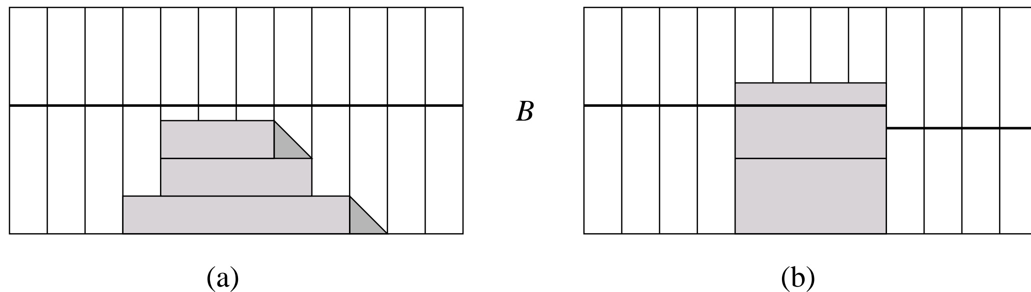

Figure 3.4: Example of added value objective. We have inserted three items in (a) and two items in (b).

Chapter 4

IK problems and their complexities

We start this chapter with a short description of different kinds of IK problems in Section 4.1. We restrict ourselves to the basic problems and some of their variants. At the end of Section 4.1 lower bounds for the number of solutions is roughly estimated for one item problem. Sections 4.2 and 4.3 show that decision problems IKHD and IKD are NP-complete. The corresponding optimization problems IKHO and IKO are introduced at the same time, and their approximability and fixed parameter tractability are considered in Section 4.4. The last section in this chapter studies the relationship of GAP and 0–1 MDKP.

4.1 Deriving the basic problem

Assume that the selection of object j $(1 \leq j \leq n)$ for knapsack i $(1 \leq i \leq m)$ employs c copies to the right from i. Hence, the selection makes $c+1$ copies from i to $i+c$, that is, clone $C_{ij} = [i, i+c]_j$. Suppose also that radiation has the maximum length of u, that is, $R_{ij} = [i-u, i-1]_j \cup [i+c+1, i+c+u]_j$. Thus, the interactions occur in knapsacks $i-u$ to $i+c+u$. The use of equation

$$I_{ij}(k) = \begin{cases} r_{ij}(k), & \text{when } k \in [i-u, i-1]_j \cup [i+c+1, i+c+u]_j, \\ 1, & \text{when } k \in [i, i+c]_j, \\ 0, & \text{otherwise} \end{cases} \tag{4.1}$$

makes the following material more readable. The left and right radiation are given in the first line, and the clone part in the second (radiation is put around the clone). Analogically, we define I_{ij} $(i = 1, \ldots, m$ and $j = 1, \ldots, n)$ to be the corresponding intervals.

Our basic problem for m knapsacks and n items is to

$$\max \quad \sum_{i=1}^{m}\sum_{j=1}^{n} x_{ij} \sum_{k=i-u}^{i+c+u} I_{ij}(k) p_{kj} \tag{4.2}$$

$$\text{subject to} \quad \sum_{i=1}^{m}\sum_{j=1}^{n} x_{ij} w_{ij} I_{ij}(l) \le b_l, \qquad l = 1,\dots,m, \tag{4.3}$$

$$x_{ij} = 0 \text{ or } 1, \qquad i = 1,\dots,m \quad j = 1,\dots,n, \tag{4.4}$$

$$\sum_{i=1}^{m} x_{ij} \le 1, \qquad j = 1,\dots,n. \tag{4.5}$$

Sum (4.2) counts the profit of selections x_{ij}. It takes into account the effect of the left and right radiations and the clones by using $I_{ij}(k)$ given in (4.1). Inequality (4.3) tells that one cannot overload the knapsacks: it takes into account the effects of the clones and radiations as in (4.2).

Note that this is a mixture interaction type discussed earlier: we use profits p_{kj} and weights w_{ij}. Some of the results we are going to give may not change, even if we used for instance w_{lj} in (4.3). This is not surprising in the light of Theorem 3.1. Reason for choosing p_{kj} and w_{ij} comes from load clipping, where the profits vary on the target moment (knapsack) k, while the weights do not vary.

Condition (4.4) ensures that each object can be selected only once for a knapsack. Inequality (4.5), in turn, ensures that each object is selected at most once for all knapsacks $i = 1,\dots,m$. The problem setting (4.2)–(4.5) is called the *basic 0–1 interactive knapsack problem* (basic 0–1 IK for short). By changing condition (4.4) to

$$x_{ij} \in \mathbb{N} \tag{4.6}$$

(condition (4.5) has to be changed accordingly) we have a prototype for the *0–n interactive knapsack problem* (0–n IK). By defining 0–n IK in this way we, at the same time, fix that the clones and radiations of n objects of type j are identical, like in Figure 3.3 (b). To implement 0–n IK as in Figure 3.3 (a), we need more decision variables. We need more restrictions, if we want to select each item once, like (4.5), but still choose the amount of the item when the item is selected, like in (4.6).

We can also extend the problem setting (4.2)–(4.5) by modifying (4.5) to

$$\sum_{i=1}^{m} x_{ij} \le K, \tag{4.7}$$

and by adding an extra restriction

$$x_{kj} = 0, \text{ for } i < k \le i + c, \text{ when } x_{ij} = 1 \tag{4.8}$$

obtaining the *0–1 interactive knapsack problem*. Inequality (4.7) allows at most K clones in a knapsack array. Condition (4.8) is used to prevent the selection of object j for a knapsack which already has a copy or a part of some other clone of type j. This means that we do not accept overlapping clones for the same object in this problem setting. Figure 4.1 on page 41 shows an example of the case (4.8), where the clones (black boxes) do not overlap.

Basic 0–1 IK has an important special case: $u = 0$, that is, the case with no radiation. This special case has applications in scheduling to be described later. If we further assume that $c = 0$ (each clone contains one member, the object itself, in one knapsack), we obtain GAP.

Theorem 4.1. *If $c = u = 0$, the 0–1 IK problem is equivalent to GAP.*

Proof. Conditions (4.4)–(4.5) are the same for both problems. Neither of them contain c or u. Substitution $c = u = 0$ to equation (4.1) gives

$$I_{ij}(k) = \begin{cases} 1, & \text{when } k \in [i, i+c]_j = i, \\ 0, & \text{otherwise,} \end{cases}$$

because $[i, i+c]_j = [i,i]_j = i$, and because $[i-u, i-1]_j = [i, i-1]_j = [i+c+1, i+c+u]_j = [i+1, i]_j = \emptyset$. Hence, sum (4.2) is equal to

$$\sum_{i=1}^{m}\sum_{j=1}^{n} x_{ij} \sum_{k=i-u}^{i+c+u} I_{ij}(k) p_{kj} = \sum_{i=1}^{m}\sum_{j=1}^{n} x_{ij} \sum_{k=i}^{i} I_{ij}(k) p_{kj} = \sum_{i=1}^{m}\sum_{j=1}^{n} x_{ij} p_{ij}.$$

Note that $I_{ij} = [i,i]_j = i$ and $l \in I_{ij}$ implies $l = i$. Now, in (4.3) index j runs the objects from 1 to n. This follows from the fact that the length of a clone is one. Thus, (4.3) turns out to be

$$\sum_{j=1}^{n} w_{ij} x_{ij} \leq b_i$$

($i = 1, \ldots, m$). Hence, we have shown both (4.2) and (4.3) to be equivalent with their counterparts in GAP. □

We can extend IK problem settings easily by assuming that the lengths of a clone and radiation depend on j. So, we introduce $c_j \in \mathbb{N}$ and $u_j \in \mathbb{N}$ for $j = 1, \ldots, n$. Or, equivalently, we can introduce $c_{ij} \in \mathbb{N}$ and $u_{ij} \in \mathbb{N}$, for $i = 1, \ldots, m$ and $= 1, \ldots, n$. Further, we may use them as decision variables among the variables x_{ij}, that is, the length of a clone is a decision variable. With these assumptions we may expose other optimization problems that can be modeled with the 0–1 interactive knapsacks (and also with other models).

Appendix B contains examples of optimization problems with variable length clones.

If we interpret m knapsacks as *time line* $[1, m]$, we can think that a clone of object j from i to $i + c_j$ is a *decision* (or act, task) j *executed* (actualized, realized) on the time interval $[i, i+c_j]$ of *length* c_j+1 (we assume that $[i, i+c_j] \subseteq [1, m]$). Now we can optimize the length, position and the number of decisions with different criteria. Restrictions concerning these variables are also possible. Radiation with time has application specific interpretations: the decision j may have some consequences to the profits and resources before or after (or both) the specific decision j is executed at period $[i, i + c_j]$. For example, in load clipping we may store energy (corresponds to left radiation), before the control occurs. We could think that the event starts, when energy storing starts, but the start of the control period is the reason for storing energy (see Chapter 8).

Next we show a result concerning the number of clone families in a knapsack array. We count here the families consisting of one type. We use the basic problem that is extended with conditions (4.7)–(4.8). First we recall some useful combinatorial tools.

Lemma 4.2. *(Binomial theorem [52, p. 162].) For $a, b \in \mathbb{R}$ and $n \geq 0$,*

$$(a + b)^n = \sum_{k=0}^{n} \binom{n}{k} a^k b^{n-k}.$$

Putting $k = m = n$ in Vandermonde's identity

$$\binom{m+n}{k} = \sum_{i=0}^{k} \binom{m}{i} \binom{n}{k-i}$$

(see [52, pp. 169–170]), we obtain identity

$$\binom{2n}{n} = \sum_{i=0}^{n} \binom{n}{i}^2.$$

Note that

$$\binom{2n}{n} = \sum_{i=0}^{n} \binom{n}{i}^2 > \sum_{i=0}^{n} \binom{n}{i} = 2^n \tag{4.9}$$

by Lemma (4.2). Now we have machinery to show the next theorem.

Theorem 4.3. *Assume the basic problem with conditions* (4.7) *and* (4.8)*. The total number of clone families consisting of constant length clones in a knapsack array of length m is exponential on m.*

Proof. Let c be an arbitrary positive integer and set $d = c + 1$ (length of a clone). First consider the amount of k clones put in the knapsack array. Let $v_0, v_1, \ldots, v_k$ be nonnegative integers. Integer v_i (except the last one) gives the number of knapsacks having no clones (or part of any clone) before the $(i + 1)$th clone. Hence, we can model k clones in the knapsack array as a sum $v_0 + \sum_{i=1}^{k}(d + v_i)$ provided that the sum is equal to m. First integer v_0 equals the number of knapsacks before the first clone and the last integer v_k equals the number of knapsacks after the last clone. This is the composition of m into $2k + 1$ pieces. Thus, $\sum_{i=0}^{k} v_i = m - dk$. Next we count all possible combinations for integers $v_0, \ldots, v_k$ which sum up to $m - dk$. This gives the number of compositions of $m - dk$ into $k + 1$ parts. It is the number of clone families having k clones of length d in the knapsack array of length m.

By [93, p. 190] the number of compositions is

$$\binom{(m - dk) + (k + 1) - 1}{(k + 1) - 1} = \binom{m - kd + k}{k}.$$

So, the number of all clone families is

$$\sum_{k=0}^{\lfloor m/d \rfloor} \binom{m - kd + k}{k}. \tag{4.10}$$

The largest k is $\lfloor m/d \rfloor$ (we consider only families that fully fit in the knapsack array) and the smallest is zero. In between we find $k = m/2d$, and by considering the corresponding term in (4.10), we obtain

$$\binom{m - kd + k}{k} = \binom{m - md/2d + m/2d}{m/2d} = \binom{(md + m)/2d}{m/2d}.$$

Since $d \geq 1$, we have $md + m > 2m$, and

$$\binom{2(m/2d)}{m/2d} > 2^{m/2d}$$

by (4.9). □

Corollary 4.4. *When the length of a clone is one ($c = 0$ or $d = 1$), the number of different clone families in the knapsack array is 2^m, and when the length is two ($c = 1$ or $d = 2$), the number of clone families is F_{m+1}, the $(m + 1)$th Fibonacci number.*

Proof. When $c = 0$, sum (4.10) is of the form

$$\sum_{k=0}^{m} \binom{m - k + k}{k} = \sum_{k=0}^{m} \binom{m}{k} = 2^m$$

by Lemma 4.2. When $c = 1$ ($d = 2$), we have

$$\binom{m-2k+k}{k} = \binom{m-k}{k}.$$

Identity

$$\sum_{k=0}^{m} \binom{m-k}{k} = F_{m+1}$$

by [52, p. 289], shows the latter results, because

$$\sum_{k=0}^{m} \binom{m-k}{k} = \sum_{k=0}^{\lfloor m/2 \rfloor} \binom{m-k}{k},$$

where the equality holds, since $\binom{m-k}{k} = 0$, when $k > m/2$. □

Theorem 4.3 and Corollary 4.4 do not tell much about the growth rate of the number of clone families as a function of the length of the knapsack array. The strength of Theorem 4.3 is in its simplicity. As Corollary 4.4 suggests, it is intuitively clear that the number of clone families decreases as the length of a clone increases, as there will be less room to make different compositions. In applications we often have some fixed minimum value (> 2) for c.

One should also note that in Theorem 4.3 we have only constant length clones. If we allow variable length clones (by using variables c_j), the number of clone families explodes. We have to count the number of clone families for each combination, or clone families containing different length clones instead of constant length clones.

These remarks imply that the enumerative solution to be presented in Section 9.1 is not fast enough. Indeed, in practical applications n can grow to rule out the enumerative (sub)solution, even with effective branch and bound methods. An approach, where the places of clone types are fixed one at a time is used also in other methods, like dynamic programming [3, 6] (see Sections 9.1 and 9.2). It also turns out that this approach is NP-complete, in general, as will be shown in the next section.

4.2 Heuristic approach to IK problems

We show that an important heuristic approach using one item needs exponential time. The 1-item version is considered because it is much simpler than the several item version of the problem. Even thought this approach is NP-complete, all versions of it are solvable in pseudo-polynomial time. Hence,

there exists polynomial time algorithms when we fix K, one of its parameters. Some special cases can be solved in polynomial time (see the next chapter).

Let our problem consist of

m,	the number of knapsacks,
$x_i \in \{0,1\}$,	the decision variables ($i = 1, \dots, m$),
$p_i \in \mathbb{N}$,	the profits of item ($i = 1, \dots, m$),
$w_i \in \mathbb{N}$,	the weights of item ($i = 1, \dots, m$),
c,	the number of copies ($c + 1$ is the length of a clone),
u,	the length of radiations,
I_i,	interaction functions that map the distance from i to a rational number ($i = 1, \dots, m$),
$b_i \in \mathbb{N}$,	capacities of the knapsacks ($i = 1, \dots, m$),
K,	a positive integer that gives the upper bound for the times we can insert an item into the knapsack array.

By using c and u we (usually) mean that $I_i : \{1, \dots, m\} \to \mathbb{Q}$ is defined like in (4.1), that is, I_i equals one, for knapsacks $i, \dots, i+c$, and is arbitrary for knapsacks $i-u, \dots, i-1$ and $i+c+1, \dots, i+c+u$. For all other knapsacks, I_i is zero. This interpretation is used most of the time, and the exceptions are pointed out. The main reason for using c and u is restriction (4.13) below, also shown in Figure 4.1 as overlapping radiations and nonoverlapping clones. Recall that in load clipping, the controls cannot overlap, while overlapping is allowed for some of the effects of controls.

The decision problem (interactive knapsacks heuristic decision, IKHD) is

IKHD
Instance: $x_i \in \{0,1\}$, profits p_i, weights w_i, length of clones c, length of radiations u and functions $I_i : \{1, \dots, m\} \to \mathbb{Q}$, for knapsacks $i = 1, \dots, m$, with capacities b_i, and two positive integers P and K.
Question: Is there a distribution of values x_i fulfilling the requirements below such that

$$\sum_{i=1}^{m} x_i \sum_{k=i-u}^{i+c+u} I_i(k) p_k \geq P? \tag{4.11}$$

We require that the knapsacks are not overfilled (4.12), that the clones are separate (4.13), that item is selected at most once to a knapsack (4.14), and that there are at most K clones (that is, the maximum number of times an

item can be put) in the knapsack array (4.15):

$$\sum_{i=1}^{m} x_i w_i I_i(l) \leq b_l, \qquad l = 1, \ldots, m, \tag{4.12}$$

$$x_k = 0, \text{ for } i < k \leq i + c, \text{ when } x_i = 1, \qquad i = 1, \ldots, m, \tag{4.13}$$

$$x_i = 0 \text{ or } 1, \text{ and} \qquad i = 1, \ldots, m, \tag{4.14}$$

$$\sum_{i=1}^{m} x_i \leq K, \qquad i = 1, \ldots, m. \tag{4.15}$$

The optimization version of IKHD is called IKHO (IK heuristic optimization) problem. In IKHO we are to

$$\max \sum_{i=1}^{m} x_i \sum_{k=i-u}^{i+c+u} I_i(k) p_k \tag{4.16}$$

subject to (4.12)–(4.15).

The above formulation of IKHO is closely related to 0–1 MDKP. In this form, IKHO can be transformed into 0–1 MDKP, and 0–1 MDKP can be transformed into IKHO [4] (which is also demonstrated in Section 4.4). Hence, why to study IKHO problem at all? The answer is that IKHO (4.16), (4.12)–(4.15), among many other problems, can be seen as a special case of 0–1 MDKP. Normally, an IKHO instance uses restrictions (4.13) and (4.15) that take a special form in 0–1 MDKP. If we transform an 0–1 MDKP instance to IKHO, these restrictions are not normally used: we will have $c = 0$ and $K = m$.

Another motivation for studying IKHO is that (4.16), (4.12)–(4.15) is the simplest formulation of the problems in a larger family of IKHO-type optimization problems of practical interest. Other family members use c and u differently: they can depend on the knapsack or c can be a variable for each inserted clone (and the value of u depends on the value chosen for c). There are also other variants, which can be motivated by the load clipping application.

Figure 4.1 reflects both IKHD and IKHO problems. We have inserted an item into knapsack i. The same item is also inserted into knapsack $i+10$. The length of clone is $c + 1 = 5$ and the length of radiation is 5. In the methods presented in Section 5.2, we distinguish between left and right radiation: they are denoted with u_l and u_r, respectively. In the case of Figure 4.1, we have $u_l = 3$ and $u_r = 5$. Define ℓ to be the total length of the clone, and left and right radiations together. Hence, the total length in the example of Figure 4.1 is $\ell = 3 + 5 + 5 = 13$.

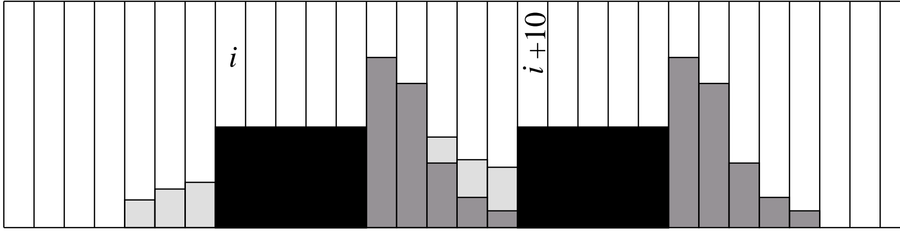

Figure 4.1: A sample clone inserted in knapsacks i and $i + 10$.

If the knapsacks were larger, we could insert the item third time into knapsack $i + 5$: restriction (4.13) says that the parts drawn in black cannot overlap while the radiation (parts drawn in grey in Figure 4.1) can overlap.

When we take into account the radiation, IKHD turns out to be NP-complete, in general. Hence, the radiation makes IKHD hard to solve. Unfortunately, radiation is needed in many applications.

Theorem 4.5. IKHD *is* NP-*complete, when* $u > 0$ *and* $c = 0$.

Proof. Let $u = m$, that is, the radiation spreads to the last knapsack from each knapsack. Note that restriction (4.13) is only for clones and not for radiation. Knapsack m will have radiation $\sum_{i=1}^{m-1} x_i w_i I(m)$, which is restricted above by b_m (restriction (4.12)). Let each p_i be positive. Hence, every knapsack will have item if the radiations fit into the knapsack m. With objective (4.11) we have the normal 0–1 knapsack problem in knapsack m, consisting of m quantities ($m - 1$ quantities from radiation and one from the item which we insert into the last knapsack). The 0–1 knapsack problem is NP-complete [72]. ☐

In Section 4.4 we show, how 0–1 MDKP can be transformed into IKHO problem.

Corollary 4.6. IKHD *is* NP-*complete (1) for variable* u *and* $c = 0$*, (2) for variable* u *and* c*, and (3) for* $u > 0$ *and* $c > 0$ *(depending on* m*).*

Proof. These problems contain instances handled in Theorem 4.5 as special cases. In (3), $c > 0$ decreases the number of items k ($< m$) in the obtained 0–1 KP instance. However, k can still depend on m. ☐

If in Theorem 4.5 and in Corollary 4.6 we have a fixed K, IKHD can be solved in polynomial time by enumerative methods. To see this, consider the

proof of Theorem 4.3. The running time of this enumerative method is at most

$$\begin{aligned}
& \sum_{k=0}^{K} \binom{m-kc+c}{k} \le \sum_{k=0}^{K} \binom{m}{k} \\
= & \sum_{k=0}^{K} \frac{m(m-1)\cdots(m-k+1)}{k!} \le \sum_{k=0}^{K} m^k/k! \\
\le & \sum_{k=0}^{K} m^k = O(m^K),
\end{aligned}$$

where $K \ge 0$ is fixed, as each step in the enumeration can be done in polynomial time. Hence we have a pseudo-polynomial time algorithm for IKHD. Note also that

$$\begin{aligned}
& \sum_{k=0}^{K} \binom{m-kc+c}{k} \ge \sum_{k=0}^{K} \binom{m-Kc}{k} \\
= & \sum_{k=0}^{K} \frac{(m-Kc)(m-Kc-1)\cdots(m-Kc-k+1)}{k!} \\
\ge & \sum_{k=0}^{K} \big(m-K(c+1)\big)^k/K! \\
= & \Omega(m^K),
\end{aligned}$$

holds, when K is fixed. Although it is not very practical, the enumerative method will be given in Section 9.1 (in load clipping there are instances, where K is not fixed, and instances where K is large).

In practice we need variable c_i's and sometimes even variable u_i's. Variable clone lengths as well as radiation itself may give new directions for scheduling problems, and at least in load clipping the variable clone lengths are essential.

4.3 IKD is NP-complete

IKHD discussed in the previous section is enough for showing that IKD is also NP-complete. In IKD we have many items to be handled at the same time and IKD is a direct generalization of IKHD. IKO, the optimization version of IKD is otherwise the same as IKHO, but with n items.

In Theorem 4.1 we established a link between basic 0–1 IK and GAP, which is NP-complete in the strong sense [72]. IKD is almost the same as basic 0–1 IK, and hence, reduction to GAP is also possible. In this section we first give

the corresponding decision problem IKD, and then some results concerning special cases not allowing GAP to occur. We suppose that items have their own clone lengths c_j.

IKD
Instance: $x_{ij} \in \{0,1\}$, profits p_{ij}, weights w_{ij}, lengths of clones c_j, lengths of radiations u_j and functions $I_{ij} : \{1,\ldots,m\} \to \mathbb{Q}$, for items $j = 1,\ldots,n$, and knapsacks $i = 1,\ldots,m$ with capacities b_i and two positive integers P and K.
Question: Is there a distribution of values x_{ij} fulfilling the requirements below such that

$$\sum_{i=1}^{m}\sum_{j=1}^{n} x_{ij} \sum_{k=i-u_j}^{i+c_j+u_j} I_{ij}(k) p_{kj} \geq P? \tag{4.17}$$

We require that the knapsacks are not overfilled (4.18), that the clones are separate (4.19), that an item is selected at most once to a knapsack (4.20) and that there are at most K clones in the knapsack array (4.21) (that is, K is the maximum number of times an item can be put in the knapsack array):

$$\sum_{i=1}^{m}\sum_{j=1}^{n} x_{ij} w_{ij} I_{ij}(l) \leq b_l, \tag{4.18}$$

$$x_{kj} = 0, \text{ for } i < k \leq i + c_j, \text{ when } x_{ij} = 1 \tag{4.19}$$

$$x_{ij} = 0 \text{ or } 1, \tag{4.20}$$

$$\sum_{i=1}^{m} x_{ij} \leq K, \tag{4.21}$$

where $l = 1,\ldots,m$ in (4.18), $i = 1,\ldots,m$ in (4.19)–(4.20), and $j = 1,\ldots,n$ in (4.19)–(4.21).

In IKO our aim is to

$$\max \quad \sum_{i=1}^{m}\sum_{j=1}^{n} x_{ij} \sum_{k=i-u_j}^{i+c_j+u_j} I_{ij}(k) p_{kj} \tag{4.22}$$

subject to (4.18)–(4.21) with appropriate bounds on indices.

Both IKHO and IKO have (practical) versions, where the length of clone is variable and the length of radiation depends of the length of clone. Usually, we refer to the problems (4.16), (4.12)–(4.15) and (4.22), (4.18)–(4.21), but when talking about variable clone lengths, the corresponding problem should be adjusted. (See Appendix B for examples.)

We already know that IKHD, a special case having only one clone type ($n = 1$) is NP-complete. Similarly, if there is only one knapsack ($m = 1$)

implying that $c_j = u_j = 0$ for the items considered, IKD reduces to the 0–1 knapsack problem. Thus, IKD is NP-complete also in this special case.

Theorem 4.7. IKD *with fixed* $c_j > 0$*, and* IKD *with variable* c_j*'s (and variable* u_j*'s) are* NP*-complete problems in the strong sense.*

Proof. We show that SEQUENCING WITH RELEASE TIMES AND DEADLINES reduces to IKD with fixed $c_{ij} > 0$. The variable clones contain the fixed length clones as special case.

Let our sequencing problem consist of T tasks, with release times $r(t)$, deadlines $d(t)$, and lengths $l(t)$, where $t \in T$. Our problem is to find a feasible schedule for T. In other words, t is executed after $r(t)$, stops before $d(t)$ and does not overlap with other task $t' \in T$.

The knapsacks simulate time such that the first knapsack is moment 1 and the last one is moment m, where $m = \max r(t) + \sum l(t)$ (so that every task can be positioned into the knapsack array). The set of tasks T corresponds to the set of items, which are numbered $j = 1, \ldots, n$ (and hence, $|T| = n$). Weights for each item are 1, and each knapsack capacity b_i is 1. This entails that only one item can be put into a knapsack at a time (restriction (4.18)). At the same time this means with restriction (4.19) that no item will overlap and that in the sequencing problem two tasks will not overlap.

We set $c_j = l(j)$ and $u_j = 0$, for $j = 1, \ldots, n$. Further, we set $p_{ij} = 0$, if $i < r(j)$, that is, the profits are 0 for items which are scheduled before their release time. Similarly, set $p_{ij} = 0$, if $i > d(j)$, and $p_{ij} = 1$, for $r(j) \leq i \leq d(j)$. Moreover, we have (the standard interpretation) $I_{ij}(l) = 1$, when $l = i, \ldots, i + c_j$.

If item j is put (task is scheduled) inside its feasible region, the profits will be $\sum_{k=i}^{i+c_j} I_{ij}(k)p_{kj} = \sum_{k=i}^{i+c_j} 1 = l(j)$. If, on the other hand, item j is put (task is scheduled) partially or totally outside of its feasible region, the profits will be less than $l(j)$ for j.

Let $P = \sum_{t \in T} l(t)$. Now we ask, whether we can put items into the knapsack array such that the total profit is P or more and such that restrictions are not violated. If the tasks are feasibly scheduled, the profit is exactly P.

It is clear that this transformation fulfills the requirements for pseudo-polynomial transformation. The case of variable c_j's is contained in the above case, and hence the claim. □

Figure 4.2 shows two tasks put into the knapsack array. The first item gives profit of 4, but the second only 1. Thus, this solution does not give a solution to the sequencing within intervals problems.

GAP is NP-complete in the strong sense and IKD is a generalization of GAP. However, in applications we are given some minimum for integers $c_j > 0$ $(j = 1 \ldots, n)$ and GAP does not cover that case, neither the case of variable c_j's. The above proof also illustrates the close relationship between sequencing problems and interactive knapsacks.

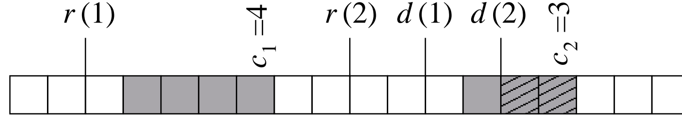

Figure 4.2: Two tasks put into the knapsack array. Task 1 is feasibly scheduled but task 2 continues after its deadline.

4.4 On the approximability and fixed tractability

Since IKO is a generalization of GAP, it is also APX-hard and is not included in PTAS. IKHO is closely related to 0–1 MDKP, where m is not fixed, and hence, IKHO is also APX-hard.

We reduce a highly restricted version of GAP (RGAP) to IKHO. As shown in [23], RGAP is APX-hard even on the instances of the following form, for all positive δ:

- $p_{ij} = 1$, for all j and i,
- $w_{ij} = 1$ or $w_{ij} = 1 + \delta$, for all j and i, and
- $b_i = 3$, for all i.

The APX-hardness of RGAP means that there exists $\varepsilon > 0$ such that it is NP-hard to decide whether an instance has an assignment of $3m$ items or if each assignment has value at most $3m(1 - \varepsilon)$ (see [23] and Definition 2.9). In the sequel, we suppose that $u = m$.

Theorem 4.8. *IKHO is APX-hard.*

Proof. Given an instance of RGAP (having m knapsacks and n items), we create an instance of IKHO as follows.

If $x_{ij} = 1$ in RGAP, then $x'_{j_i} = 1$ in IKHO. In IKHO we have knapsacks $j_1, \ldots, j_m$ for each item j, and knapsacks $b'_1, \ldots, b'_m$ similar to the knapsacks in RGAP.

Interaction I is defined in such a way that knapsacks $j_1, \ldots, j_i, \ldots, j_m$ will be full when item j_i is chosen (that is, when we set $x'_{j_i} = 1$). Hence, item j can be put at most once into the knapsack array of IKHO. Moreover, interaction for knapsack b'_i is such that the weight w_{ij} of RGAP will be put into knapsack b'_i. Other knapsacks are left intact. We need only radiation (that is, $c = 0$) defined as:

$$I_{j_i}(k) = \begin{cases} w_{ij} & \text{when } k = b'_i, \\ 1 & \text{when } k \in [j_1, j_m], \\ 0 & \text{otherwise.} \end{cases} \tag{4.23}$$

The capacity of knapsacks $b'_1, \ldots, b'_m$ is 3, while the capacity of knapsacks $j_1, \ldots, j_m$ is 1, for each item j. We set $p_{j_1} = 1$, for each j, and $p_{j_i} = 0$, for each j and $i \neq 1$, and $p_{b'_i} = 0$, for each i, because a nonzero interaction involves $m + 1$ knapsacks. Hence, the profit given by radiation equals 1. Further, we set $w_{j_i} = 1$, for each item and knapsack $1_1, \ldots, n_m$. For knapsacks $b'_1, \ldots, b'_m$ the weight is 4, to prevent $x_{b'_i} = 1$, for all i.

Three items can be put into a knapsack i in RGAP if and only if three items can be fitted into a knapsack b'_i through interaction. If RGAP has a full assignment of value $3m$, the corresponding IKHO instance has the same value and knapsacks b'_i are full. Otherwise, in RGAP the value is at most $3m(1 - \varepsilon)$ as well as in IKHO. Each item in RGAP can be assigned only once and IKHO behaves similarly. Hence, we have constructed a gap-preserving reduction complying with Definition 2.9. □

Hence, a polynomial time approximation scheme for IKHO would contradict the fact that RGAP is APX-hard. Figure 4.3 shows the knapsack structure used in Theorem 4.8. It corresponds to a situation where we set $x_{11} = 1$ in RGAP, that is, we put the first item into the first knapsack. In IKHO we set $x_{1_1} = 1$ and radiation takes care that knapsacks $1_2, \ldots, 1_m$ are full so that the first item of RGAP cannot be put into any other knapsack any more. Moreover, to handle the normal RGAP knapsack restriction, the interaction puts to the first corresponding knapsack b_1 in IKHO the same amount than in RGAP. In this example, weight w_{11} in RGAP is $1 + \delta$.

Theorem 4.8 considers the special case of IKHO where $c = 0$, $u = m$ and $K = m$. We can also reformulate IKHO so that it equals a special case of 0–1 MDKP. Thus, we can use the algorithms designed for 0–1 MDKP [44, 98]. Assume, for a moment that $c = 0$. The case $c > 0$ is handled below.

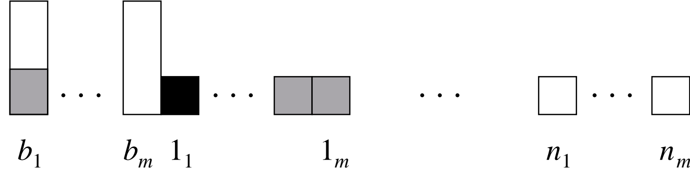

Figure 4.3: RGAP with n items and m knapsacks is mapped into IKHO, which will have $m + nm$ knapsacks.

First, define $p'_i = \sum_{k=i-u}^{i+c+u} I_i(k)p_k$ $(i = 1, \ldots, m)$. This changes our objective function (4.16) to be $\sum_{i=1}^{m} x_i p'_i$. Second, define $w'_{ik} = w_i I_i(k)$, for $i, k = 1, \ldots, m$. As a result, condition (4.12) turns to be $\sum_{i=1}^{m} w'_{ik} x_i \leq b_k$ $(k = 1, \ldots, m)$. Again, an item in knapsack i means that we add a weight w'_{ik}, for each knapsack $k = 1, \ldots, m$. Condition (4.15) can be seen as an additional $(m+1)$th knapsack restriction.

We have converted our problem into

$$\max \quad \sum_{i=1}^{m} p'_i x_i \tag{4.24}$$

$$\text{subject to} \quad \sum_{i=1}^{m} w'_{ik} x_i \leq b_k, \qquad k = 1, \ldots, m+1. \tag{4.25}$$

$$x_i = 0 \text{ or } 1, \qquad i = 1, \ldots, m. \tag{4.26}$$

Condition (4.25) is equal to $Wx \leq b$, where W is an $m + 1 \times m$ matrix with positive elements. The last row of W contains ones and $b_{m+1} = 1$. The above problem is a special case of 0–1 MDKP containing $m + 1$ knapsacks and m items.

As pointed out, 0–1 MDKP with the fixed number of knapsacks is in PTAS but otherwise, 0–1 MDKP is hard to approximate. Even though IKHO seems to be easier than 0–1 MDKP with the unfixed number of knapsacks, Theorem 4.8 rules out the possibility of having a polynomial time approximation scheme for IKHO. Moreover, Chekuri and Khanna [22] show that 0–1 MDKP is hard to approximate even when $m = \text{poly}(n)$ (recall that in our case $m = n + 1$).

The instances of IKHO with some fixed $c > 0$ are at least as hard as with $c = 0$. The number of items is not any more m, but $m/(c+1)$. We can write restriction (4.13) as m separate restrictions (m additional knapsacks in 0–1 MDKP):

$$\sum_{i=k}^{k+c} x_i \leq 1, \quad \text{where } i = 1, \ldots, m. \tag{4.27}$$

This does not change the hardness of approximating IKHO by using 0–1 MDKP. We have effectively transformed restriction (4.13) to a linear form so that 0–1 MDKP algorithms based on linear programming relaxation [98] are still usable. Hence, by the results of [22, 90] we have

Theorem 4.9. *For IKHO, we can obtain $t = \Omega(z^*/m^{1/B})$ solutions, z^* is the optimal solution, and B is the size of the smallest knapsack.*

In our case, $B = 1$ by (4.27), and hence, we cannot use improvements of [98] to get $\Omega(t^{B/(B-1)})$ solutions. Theorem 4.9 also holds for the case, where the clone length depends on the knapsack into which we put an item. (The size of W remains the same.) Thus, in the problem setting (4.16), (4.12)–(4.15) we can change every c to be c_i.

Next we show how 0–1 MDKP can be transformed into IKHO. Let W' be the integer weight matrix and p' integer profit vector of 0–1 MDKP. Set $w_i = 1$ and $I_i(k) = w'_{ik}$.

Now, we should find profits p_i such that $p'_i = \sum_{k=i-u}^{i+c+u} I_i(k)p_k$. If we could set $p = W'^{-1}p'$ we would have profits p. If W were invertible square matrix, we would obtain p as above. In this case we should set $u = m$ in order to calculate profits p' correctly. If W is not a square matrix, the use of a matrix pseudo-inverse (see, for example [49]) is difficult, or maybe even impossible. To avoid the difficulties caused by nonsquare weight matrices we could add some dummy knapsacks or items to the instance, so that the weight matrix is square and a solution for the new instance implies a solution for the original instance of 0–1 MDKP. To avoid the difficulties caused by the noninvertible matrices, we can remove some of the knapsacks and store the dependency information so that the solution to the original case would be obtainable.

The above reasoning (somewhat vaguely) argues that IKHO is as hard as MDKP, also in general, and not only in the case of invertible square weight matrices. Thus IKHO is no easier than MDKP. In the following, ZPP denotes the class of polynomial randomized algorithms with zero probability of error. It is commonly believed that containment $P \subseteq ZPP \subseteq NP$ is proper. (See [86, pp. 256 and 272].)

Theorem 4.10. *Unless NP=ZPP, IKHO is hard to approximate to within a factor of $\Omega(m^{1/(\lfloor B \rfloor+1)-\varepsilon})$, for any fixed $\varepsilon > 0$, where $B = \min_i b_i$ is fixed.*

Proof. See the discussion above and the results of [22]. □

At the end of this section we show that IKHO is not fixed parameter tractable. After that we conclude directly that IKO is not in FPT. The proof

uses kKP (k-item knapsack): if IKHO belonged to FPT, then kKP would also belong, which is not the case [18], unless $FPT = W[1]$.

We note that 0–1 MDKP is not in FPT because kKP can be seen as 2-dimensional knapsack problem (belonging to the class of 0–1 MDKP problems). If 0–1 MDKP was in FPT, then also kKP would be in FPT. Because IKHO equals 0–1 MDKP, we have shown that IKHO is not in FPT.

We also give a transformation from kKP into IKHO. Let profits $p_1, \ldots, p_n$, weights $w_1, \ldots, w_n$ and knapsack size b determine a kKP instance. We construct IKHO instance that equals the kKP instance.

Set $b'_0 = b$, and $b'_i = 1$, where $i = 1, \ldots, n$. Hence, in IKHO $m = n$ (and we have $m+1$ knapsacks). Moreover, set $p'_i = p_i - w_i$, and $w'_i = 1$, for $i = 1, \ldots, n$. To calculate the profits correctly, set $p'_0 = 1$. We do not insert any items directly in knapsack 0, and hence $w'_0 = b + 1$. Interaction is of the form

$$I'_i(k) = \begin{cases} 1, & \text{when } k = i, \\ 0, & \text{when } k \neq 0 \text{ and } k \neq i, \\ w_i, & \text{when } k = 0. \end{cases}$$

We set $c = 0$, $u = n + 1$, and $K = k$ for restriction (4.15). When we select item j in kKP, in IKHO we fill the corresponding knapsacks j, and interaction puts w_j in knapsack 0. The profits are the same in both problems:

$$\sum_{i=1}^{m} x_i \sum_{k=i-u}^{i+c+u} I'_i(k) p'_k = \sum_{i=1}^{n} x_i (p'_i + w_i) = \sum_{i=1}^{n} x_i p_i.$$

Figure 4.4 depicts 8 knapsacks. We have selected two items, the fourth and seventh in kKP. Knapsack 0 has corresponding weights $w_4 = 5$ and $w_7 = 2$ of kKP as $w'_4 I'_4(0) = w_4$ and $w'_7 I'_7(0) = w_7$. Because $w'_4 = w'_7 = 1$, we have $I'_4(0) = 5$ and $I'_7(0) = 2$. Knapsacks 4 and 7 are full, because $w'_4 I'_4(4) = 1 = b'_4$ and $w'_7 I'_7(7) = 1 = b'_7$. Further, $w'_4 I'_4(k) = w'_7 I'_7(k) = 0$ for $k = 1, 2, 3, 5, 6, 7, 8$ for knapsack 4, and for $k = 1, \ldots, 6, 8$ for knapsack 7.

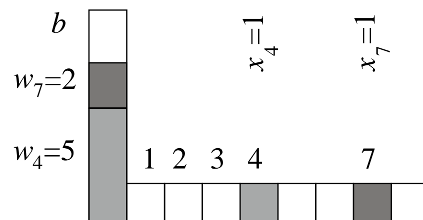

Figure 4.4: IKHO structure solving kKP. Items 4 and 7 are selected in kKP.

4.5 A transformation from GAP into 0–1 MDKP

A direct transformation from RGAP into 0–1 MDKP is also possible by using a similar construction than the one used in the proof of Theorem 4.8. By using

$$x = (x_{1_1} \ \cdots \ x_{1_m} \ \cdots \ x_{n_1} \ \cdots \ x_{n_m})^T$$

(T stands for the matrix transpose) and by adding

$$\underbrace{0 \ \cdots \ 0}_{(j-1)m} \ \underbrace{1 \ \cdots \ 1}_{m \text{ ones}} \ 0 \ \cdots \ 0, \tag{4.28}$$

for each item j in W, where W stands for the weight matrix, we can ensure that only one of the values $x_{j_1}, \ldots, x_{j_m}$ can be one. Ones occur in those positions in b, which correspond to the rows W described by (4.28) above. RGAP gives other rows directly:

$$0 \ \cdots \ 0 \ w_{i1} \ 0 \ \cdots \ 0 \ w_{i2} \ 0 \ \cdots \ 0 \ w_{in} \ 0 \ \cdots \ 0,$$

for each knapsack i. Notice that the first nonzero values take place at positions $i \bmod (m+1)$ and the next (rest) nonzero occur after $m-1$ zeros. The size of W is $(m+n) \times nm$. Vectors p and b are defined in the obvious manner. Again, the profit gap between the instances of RGAP and 0–1 MDKP is the same. To summarize, we have $W =$

$$\begin{bmatrix}
w_{11} & 0 & \cdots & 0 & w_{12} & 0 & \cdots & \cdots & 0 & \cdots & 0 & w_{1n} & 0 & \cdots & 0 \\
0 & w_{21} & 0 & \cdots & 0 & w_{22} & 0 & \cdots & 0 & \cdots & 0 & 0 & w_{2n} & \cdots & 0 \\
 & & \ddots & & & & \ddots & & & \ddots & & & & \ddots & \\
0 & \cdots & 0 & w_{m1} & 0 & \cdots & 0 & w_{m2} & 0 & \cdots & 0 & 0 & \cdots & 0 & w_{mn} \\
1 & \cdots & 1 & 1 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 & 0 & \cdots & \cdots & 0 \\
0 & \cdots & 0 & 0 & 1 & \cdots & 1 & 1 & 0 & \cdots & 0 & 0 & \cdots & \cdots & 0 \\
 & & & & & & & & & \ddots & & & & & \\
0 & \cdots & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 & 1 & 1 & \cdots & 1
\end{bmatrix},$$

and

$$b = (b_1 \ \cdots \ b_i \ \cdots \ b_m \ \underbrace{1 \ \cdots \ 1}_{n})^T, \tag{4.29}$$

where $b_i = 3$ $(i = 1, \ldots, m)$ in the case of RGAP, and

$$p = (p_{1_1} \ \cdots \ p_{1_m} \ \cdots \ p_{n_1} \ \cdots \ p_{n_m})^T, \tag{4.30}$$

where $p_i = 1$ $(i = 1_1, \ldots, n_m)$ for RGAP. (Hence, 0–1 MDKP is APX-hard in the general case, as already known.)

If we had a polynomial time approximation scheme for 0–1 MDKP, we could use it to solve RGAP. Moreover, as the above restriction also works for the general GAP (with positive values only), GAP would also be in PTAS. As noted, 0–1 MDKP is in PTAS [44] only in the cases, where m is fixed. In our transformation an $m \times n$ matrix of an instance of RGAP is transformed into an $(m+n) \times mn$ matrix of an instance of 0–1 MDKP. The transformation of RGAP into 0–1 MDKP introduces new knapsacks for each item, and hence, the number of knapsacks in 0–1 MDKP will not be fixed. Further, we can conclude the following theorem.

Theorem 4.11. *Unless N=NP, there cannot be an approximation preserving transformation of RGAP into 0–1 MDKP with a constant difference between the numbers of knapsacks.*

Proof. Otherwise, we could use the polynomial time approximation scheme of 0–1 MDKP with fixed number of knapsacks directly for RGAP, which in turn, is APX-hard. □

We can transform GAP into IKHO similarly. Recall the proof of Theorem 4.8. Because each item has the same profit in RGAP, we can use only one set of knapsacks j_i: we set $p_{j_1} = 1$ for each j and radiation will handle the rest. We cannot set directly $p_{j_i} = p_{ij}$, since now (4.23) would count too much profits with the second line. However, we can define another set of knapsacks $j'_{i'}$ and interactions

$$I_{j_i}(k) = \begin{cases} w_{ij} & \text{when } k = b'_i, \\ p_{ij} & \text{when } k = j'_{i'} \\ 1 & \text{when } k \in [j_1, j_m], \\ 0 & \text{otherwise.} \end{cases}$$

Knapsacks $j'_{i'}$ can take the radiation, if we set $b'_{j'_{i'}} = p_{ij}$. The number of knapsacks in IKHO representing GAP is $2mn + m$, while for RGAP we need $mn + m$ knapsacks. To prevent $x_{j'_{i'}} = 1$, we set $I_{j'_{i'}}(j'_{i'}) = p_{ij} + 1$.

Chapter 5

Polynomial time instances

After giving the hardness results in the previous chapter, we start looking for the borderline between intractable instances and instances solvable in polynomial time.

First, we consider one item case without radiation in Section 5.1 and with radiation in Section 5.2. The several item case without radiation is studied in Section 5.3 and with radiation in Section 5.4. Some of our constructions can be applied to other problems, which is considered in Section 5.5.

5.1 IKHO instances without radiation

The several item special case similar to the one handled in Theorem 5.1 is GAP, which is NP-complete (see Theorem 4.1).

Theorem 5.1. IKHO $\in$ P, *for* $c = u = 0$.

Proof. By setting $c = u = 0$, (4.16), (4.12)–(4.15) reduces to

$$\max \sum_{i=1}^{m} x_i p_k$$

such that

$$\sum_{i=1}^{m} x_i \leq K$$

and $x_i = 0$ or 1. We sort profits in $O(m \log m)$ time and select the K largest positive numbers. If there are no K positive numbers, then all positive profits are selected. Other x_i values are set to be zero. $\square$

Even though the following theorem also implies that IKHO is solvable in polynomial time for $c = u = 0$, the proof of Theorem 5.1 gives us a more efficient method for solving the problem in that particular special case.

In the following theorem we suppose that c is variable and restricted to be between a given minimum and maximum length. It means that the problem setting (4.16), (4.12)–(4.15) has to be fixed accordingly, but the changes are minor. In load clipping, it is reasonable to assume that some controlling utilities can have variable controlling lengths [3, 6].

Theorem 5.2. IKHO $\in$ P, *for* $u = 0$ *and variable* c *(*$cmin \leq c \leq cmax$*).*

Proof. Let $G = (V, E)$ be a directed graph with edge set E and vertex set $V = \{1, 2, \ldots, m, m+1\}$. We insert two subsets of edges, First, edges $(i, i+1)$ of length 0 and weight 0 correspond to the decision not to select item i, that is, $x_i = 0$.

The second set of edges corresponds to selecting an item and its length. Define $p'_{ij} = \sum_{k=i}^{i+j} p_k$, where $\mathrm{cmin} \leq j \leq \mathrm{cmax}$. The sum stands for an item of length j put in knapsack i (and the associated profit of the item and its clone). An edge $(i, i+j+1)$, where $\mathrm{cmin} \leq j \leq \mathrm{cmax}$, has length $p'_{i,j}$, and corresponds to the item put in knapsack i, where the clone length $(c+1)$ is j. The weight of these edges is 1. If p'_{ij} is negative, an edge is not inserted.

Our choice for j ensures that restriction (4.13) is not violated. An inserted edge $(i, i+\mathrm{cmin}+1)$ is always longer than path $(i, i+1, \ldots, i+\mathrm{cmin}+1)$ of length 0.

Now we should find the longest weight-constrained path between vertices 1 and $m+1$, where the weight of the path is at most K (to comply with (4.15)). This can be solved in polynomial time by Theorem 2.12. □

Corollary 5.3. IKHO $\in$ P, *for* $u = 0$ *and fixed values of* $c > 0$.

Proof. We have $\mathrm{cmin} = \mathrm{cmax} = c$ in the proof of Theorem 5.2. Only two edges $(i, i+1)$ and $(i, i+c+1)$ depart from each i. □

Figure 5.1 illustrates the graph construction for IKHO with $u = 0$. We can go from 1 to $m+1$ only by using the vertices of length 0 meaning that no item is inserted in any knapsack (this occurs, for instance, when each p'_{ij} is negative). We may insert an item into as many knapsacks as it fits (at most $\min(K, m/(\mathrm{cmin}+1))$ times). Figure 5.1 does not show weights as they are all 1. Zeros and $p'_{i,\mathrm{cmax}}$ indicate the lengths (and $p'_{i,\mathrm{cmax}}$ is the length of $(i, i+\mathrm{cmax}+1)$).

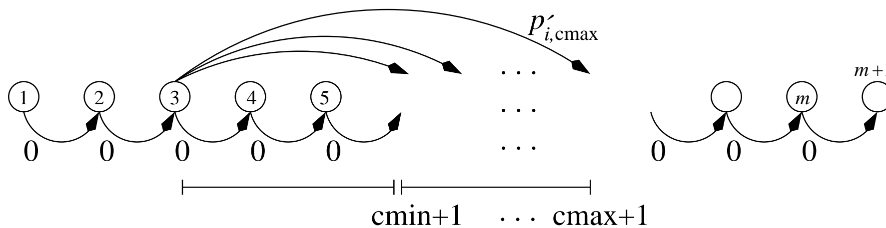

Figure 5.1: The graph constructed in Theorem 5.2. If we insert an item into the third knapsack, we can next go to the vertices which lay at least $\mathrm{cmin}+1$ and at most $\mathrm{cmax}+1$ vertex-pairs to the right.

5.2 IKHO in polynomial time with radiation

Next we show, how to handle the special case of IKHO, where the total length of the clone and radiation ℓ is $O(\log(m))$. In the proof we construct a graph with a source and target vertices s and t, and sets of vertices divided into groups. In each group the vertices are identified by bit strings of length ℓ. The bit string can be interpreted as a selection of items to different knapsacks: for example, v_{10100} corresponds to a situation where $\ell = 5$, $x_{i+0} = x_{i+2} = 1$ (an item is selected for knapsacks i and $i+2$) and $x_{i+1} = x_{i+3} = x_{i+4} = 0$. The value of i depends on the vertex group; two different vertices may share a subscript if they are in different groups. Notation $v_{1,\ldots,\ell}$ means a vertex label with a bit string of length ℓ while v_{10100} refers to a particular bit string of length 5. In what follows, a selection is a bit string of length ℓ and it is interpreted as above.

Lemma 5.4. *Let the maximum total length of the clone and radiation parts be ℓ. An item selection $v_{1,\ldots,\ell}$ (ignoring any other items in the knapsack array) can be checked against restrictions* (4.12) *and* (4.13) *in polynomial time.*

Proof. Suppose we put an item in a knapsack. The corresponding interactions can be calculated in $O(\ell)$ time. A selection $v_{1,\ldots,\ell}$ can contain at most ℓ ones (that is, we can put an item at most ℓ times in the knapsack array). Hence, checking (4.12) takes $O(\ell^2)$ time. Restriction (4.13) ensures that ones in the selection $v_{1,\ldots,\ell}$ are not too close to each other. One pass of the selection is enough (in $O(\ell)$ time) to check this. □

Consider next a group of all selections of length ℓ containing 2^ℓ bit strings. We may conclude directly by the proof of Lemma 5.4 that we need $O(\ell^2 2^\ell)$ time to comply with restrictions (4.12) and (4.13). Yet, by tabulating calculations, we obtain a better bound.

Lemma 5.5. *Let the maximum total length of the clone and radiation parts be ℓ. Every item selection $v_{1,\dots,\ell}$ (ignoring any other items in the knapsack array) can be checked against restrictions* (4.12) *and* (4.13) *in $O(\ell 2^\ell)$ time.*

Proof. We build up a result table having 2^ℓ entries, each of length ℓ. Each entry corresponds to ℓ consecutive knapsacks with fillments (fillings) corresponding to the selections label (bit string).

First, we calculate the effects of including an item for every position in the selection. That is, first we calculate ℓ entries in the result table to the following selections: $v_{0\cdots01}$, $v_{0\cdots010}$, $v_{0\cdots0100}$, and at last, $v_{10\cdots0}$, all in $O(\ell^2)$ time.

Next, we form every bit string of length ℓ with the pre-calculated strings, and construct the rest of the result table. We can combine two bit strings and the corresponding results to a new bit string (and to a new result entry) in $O(2\ell)$ time.

There are $\binom{\ell}{2}$ bit strings having two ones. The bit string and especially the entries in the result table can be constructed in

$$\binom{\ell}{2} O(2\ell)$$

time. We can use the bit strings having 1 ones and $k-1$ ones to combine them to a string having k ones. Thus, the result entries corresponding to k ones need

$$\binom{\ell}{k} O(2\ell) \tag{5.1}$$

time. Summing over (5.1) we have

$$\sum_{k=0}^{\ell} \binom{\ell}{k} O(2\ell) = O(2\ell) \sum_{k=0}^{\ell} \binom{\ell}{k} = O(2\ell)O(2^\ell) = O(\ell 2^\ell).$$

Hence, the total time consumption is $O(\ell^2) + O(\ell 2^\ell) = O(\ell 2^\ell)$. □

The calculations of the profits take the same time as checking (4.12) and (4.13). Thus, to check restrictions and to calculate profits for every selection of length $O(\ell)$, we need $O(\ell 2^\ell)$ time.

Our next step is to construct a graph, which contains every selection so that each legal (according to (4.12) and (4.13)) item combination is achievable. Lengths of the edges correspond to profits given by the inclusions of the items and hence, we want to find out long paths. We construct the graph with the following three lemmas, which form $O(m)$ selection groups each containing every selection of length ℓ. Thus, there are totally $2^\ell O(m)$ selections in the graph. A selection group corresponds to a knapsack.

In each of the following lemmas, let I_i, p_i, w_i, b_j, m and K form an instance of IKHO. Further, we refer with vertex set to the selection group mentioned in the previous paragraph. When we say that the selections can be constructed in polynomial time, we mean that the graph with vertices and edges can be constructed, and the optimal path along with the optimal solution can be found in polynomial time (on m).

We name the vertices in each vertex set $v_{x_i \cdots x_{i+\ell}}$, where $x_i \cdots x_{i+\ell}$ may have 2^ℓ different bit combinations.

Lemma 5.6. *Suppose that the total length of the clone part and radiation is ℓ, where $\ell = \log(m)$. Selections (and optimal solution) for the first ℓ knapsacks of an instance of* IKHO *can be constructed in polynomial time.*

Proof. We start from vertex s and insert $2^\ell = m$ outgoing edges to the first vertex set, which in turn, corresponds to the first ℓ knapsacks.

The vertices in the first vertex set correspond to every item selection combination for the first ℓ knapsacks. If some combination violates (4.12) or (4.13), the corresponding edge is not inserted. The checks against (4.12) and (4.13) can be done in polynomial time (on m) for each candidate edge by Lemma 5.5. The total time for all m edges is $O(\ell 2^\ell) = O(m \log(m))$.

To find out the length of an edge, note that $\ell = u_l + c + 1 + u_r$, where u_l and u_r are the lengths of the left and right radiation, and $c+1$ is the length of the clone part. We take into account only profits of the items that are included in the first u_l knapsacks. Thus, the length of an edge (profit) is

$$\sum_{i=1}^{u_l} x_i \sum_{k=i-u_l}^{i+c+1+u_r} I_i(k) p_k.$$

Moreover, the weight of an edge equals to the number of ones in the first u_l positions in its label. If, on the other hand, we do not include the item in the first u_l knapsacks ($x_1 = \cdots = x_{u_l} = 0$), the profit and the weight are both zero.

We calculate the profits at the same time when we form vertices and edges, and make checks against the restrictions. Hence, the time needed is $O(m \log(m))$. By using the longest weight-constrained path algorithm, we can check (4.15) and find out the longest path (optimal solution) in polynomial time (on m). Our graph construction automatically complies with (4.14). □

As an example, consider an instance, where $c = 0$, the left radiation has length 2, and the right radiation has length 4. Now the length of a selection is $\ell = 2 + 1 + 4 = 7$ and the vertex set contains $2^7 = 128$ vertices.

The lengths (or profits) of edges departing from s use the left radiation parts and the first vertex set corresponds to the inclusions of items into the first u_l knapsacks. Edge $(s, 1110000)$ would have length that corresponds to the inclusion of an item in knapsacks 1 and 2, that is, $x_1 = x_2 = 1$. Restrictions, on the other hand, are checked against the decision made for all seven knapsacks.

The algorithm described in the following lemma is an example of inductive algorithm design principle [70]. Our induction hypothesis is that every legal selection can be reached so that the longest weight-constrained path gives the maximum profits while maintaining the feasibility of the solution.

Lemma 5.7. *Suppose that the total length of the clone part and radiation is ℓ, where $\ell = \log(m)$. Selections for the first $m-(c+u_r)$ knapsacks of an instance of* IKHO *can be constructed and the optimal path can found in polynomial time.*

Proof. We prove the lemma by induction. By Lemma 5.6, we can construct the first vertex set for the first ℓ knapsacks and find out the longest weight-constrained path (the optimal solution) in polynomial time, calculating the profits for the first u_l knapsacks and taking into account restrictions (4.12)–(4.15) for the first ℓ knapsacks.

The induction hypothesis is that the optimal solution can be found for the first q knapsacks, where $q = u_l, \ldots, m - (c + u_r + 1)$. In other words, every legal selection is reached, the profits are calculated for the first q knapsacks correctly, and the restrictions are checked for the first $q+c+1+u_r$ knapsacks. Let $g = q - u_l + 1$. (Now g refers to a vertex set.) In the terms of vertex sets, Lemma 5.6 proves the claim for the first vertex set, $g = 1$, and after it, we have made an induction hypothesis for the first $g = 1, \ldots, m - (c + u_r + 1)$ vertex sets.

We construct a vertex set $g+1$ from the vertices of vertex set g by inserting edges $(v_{x_g \cdots x_{g+\ell}}, v_{x_{g+1} \cdots x_{g+\ell+1}})$, if $v_{x_{g+1} \cdots x_{g+\ell+1}}$ is a legal combination not violating (4.12) and (4.13). There will be at most two edges for each vertex in set g, because $x_{g+\ell+1}$ can have two values. By Lemma 5.5, we can insert edges in $O(m \log(m))$ time.

The edges from set g to $g+1$ correspond to the choice made for knapsack $g + u_l$. If $x_{g+u_l} = 1$ (we include an item in knapsack $g + u_l$), the profit is

$$\sum_{k=g+u_l-u_l}^{g+u_l+c+u_r} I_{g+u_l}(k) p_k. \tag{5.2}$$

Moreover, the weight of the edge is one. On the other hand, if $x_{g+u_l} = 0$, the length and the weight of the edge are zero. Note that the profit for the edge is defined by the state of the cth bit in the vertex label (the next bit after the

left radiation part). The profits can be calculated at the same time when we check the restrictions.

Suppose that a vertex $v_{x_{g+1}\cdots x_{g+\ell+1}}$ in vertex set $g+1$ is legal. The vertex has at most two incoming edges, which can depart from vertices $v_{x_g=0,x_{g+1}\cdots x_{g+\ell}}$ and $v_{x_g=1,x_{g+1}\cdots x_{g+\ell}}$ from vertex set g. Since $v_{x_{g+1}\cdots x_{g+\ell+1}}$ is legal in set $g+1$, vertex $v_{x_g=0,x_{g+1}\cdots x_{q+\ell}}$ in set g has $\ell-1$ common knapsacks $x_{g+1}\cdots x_{g+\ell}$ with the vertex in set $g+1$, and since bit x_g can be zero, we conclude that vertex $v_{x_g=0,x_{g+1}\cdots x_{g+\ell}}$ is legal, and therefore, the graph contains an edge $(v_{x_g=0,x_{g+1}\cdots x_{q+\ell}}, v_{x_{g+1}\cdots x_{g+\ell+1}})$ from set g to $g+1$. By the induction hypothesis, we can reach every legal selection in the first g vertex sets, and thus also the vertex $v_{x_g=0,x_{g+1}\cdots x_{g+\ell}}$, and further, an arbitrary legal vertex $v_{x_{g+1}\cdots x_{g+\ell+1}}$ in vertex set $g+1$.

Now we have a system that contains every selection of length ℓ in the knapsack array. This contains also the last vertex set $m-(c+u_r)$. We have linked selections so that every legal selection is attainable. The number of these selections in every set is $O((m-(c+u_r))m\log(m)) = O(m^2\log(m))$.

By Lemma 5.6, we can find out the solutions for the first vertex set in polynomial time. Then, we inductively form the graph that has a polynomial number of vertices. By Lemma 5.5, we can form each vertex set and the incoming edges (length and weights) in polynomial time. Our graph is acyclic and directed. By Theorem 2.12, the optimal solution can be found in polynomial time on the number of vertices in the graph (recall that $K \leq m/(c+1) \leq m$). Hence the claim. □

The first vertex set (and also the last vertex sets, as we will see) is a special case, because it may have more than one item giving profits and weights, as opposite to the profit calculations (5.2) made in the previous lemma.

Suppose that we have formed vertex set g. To obtain another set of vertices, we drop the first knapsack g and add a new knapsack $g+u_l+c+1+u_r+1$. Consecutive vertex sets contain $\ell-1$ common knapsacks. Let the common subselection be $x_{g+1},\ldots,x_{g+u_l+c+1+u_r}$. The first item x_g can have two values, zero or one, and the new item $x_{g+u_l+c+1+u_r+1}$ can also have two values. Hence, we always have two source vertices as well as two target vertices for each vertex in every vertex set (except for the first and last vertex sets).

Let us continue the example started before Lemma 5.7. The profits for the vertices between vertex sets are obtained by using the leftmost position of the clone part in a selection, that is, the knapsack into which an item is inserted. In our example, an edge departing from vertex 1110000 in the first vertex set would have length that corresponds to the profit of choosing $x_3 = 1$. Hence, it corresponds to the inclusion of the item into the third knapsack, when we calculate the profits. Other values of the selection are not noticed and one

may think that an edge from 1110000 is handled like filter "??1????", when we calculate the profits.

Figure 5.2 gives another example. It demonstrates the case, where $\ell = 4$ ($c = 0$, $u_l = 0$, and $u_r = 3$). Each vertex set has 16 vertices and we have at most 32 edges between the vertex sets.

In the example presented in Figure 5.2, the common subscript part between the first two vertex sets is 234. The first knapsack is moved off and it is replaced by the fifth knapsack, when constructing the second vertex set. The selection of an item to the knapsack i ($x_i = 1$ in the leftmost position of the label of the vertex—the left radiation is not used in this example) is shown by an edge pointing below the horizontal line. The weight of these edges is one. Note that only the coming knapsack can make a choice whether to include an item: the selection is already fixed for the common part. Thus, the profit of including an item is calculated later than the decision to include the item.

Lemma 5.8. *Suppose that the total length of the clone part and radiation of an instance of* IKHO *is ℓ, where $\ell = \log(m)$. Lengths (profits) can be calculated in polynomial time for the edges departing from the last vertex set $m-(c+u_r)$ to the target vertex t.*

Proof. Vertex t is the target of paths. The restrictions are already checked for every selection in the last vertex set $m-(c+u_r)$. Profits (lengths) use the bits occupying the positions to the right (and including) the cth bit.

By Lemma 5.5, we calculate the lengths (profits) in polynomial time for each vertex departing from the last vertex set. The weight of an edge equals

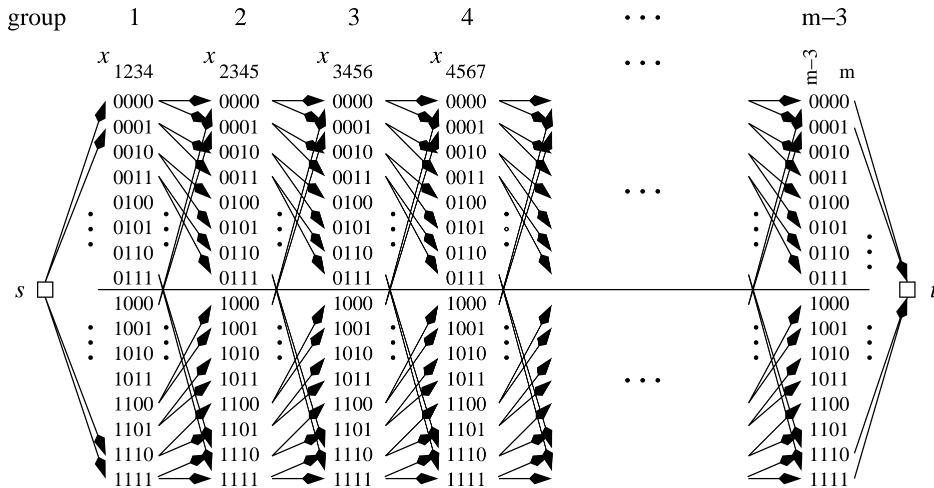

Figure 5.2: The graph structure for Theorem 5.9, when $\ell = 4$.

the number of ones in the bits after the left radiation. □

We continue with the example started before Lemma 5.7. The length of edge $(0001101, t)$ corresponds to the profit of selection $x_{m-3} = x_{m-2} = x_m = 1$ and $x_{m-4} = x_{m-1} = 0$. If we had $x_{m-4} = 1$, we would calculate it into the profits.

In the example presented in Figure 5.2 we use every bit in the selection, when we calculate the profits of edges from the last vertex set to t, because the length of left radiation is zero.

Theorem 5.9. IKHO $\in$ P, *when* $\ell = O(\log(m))$, *where* m *is the number of knapsacks.*

Proof. Let I_i, p_i, w_i, b_j, K, and m form an instance of IKHO. Suppose that the total length of the clone part and radiation is at most ℓ, where $\ell = \log(m)$. We form a graph G with source vertex s and target vertex t in polynomial time as described in Lemmas 5.6–5.8.

By Theorem 2.12, we find the best set of selections by constructing the longest weight-constrained path between vertices s and t in polynomial time. By using weight-constraints, we also check condition (4.15). Because $K \leq m/(c+1) \leq m$, we find the longest weight-constrained path in polynomial time by Theorem 2.12. □

To analyze the running times, note that we have $m - (c + u_r) = O(m)$ vertex sets. Each set has 2^ℓ vertices. Hence, the total number of vertices is $2 + (m - (c + u_r))2^\ell = O(m2^\ell)$. Further, we have 2^ℓ incoming edges to the first vertex set and as many incoming edges to the vertex t. Other $m - (c + u_r) - 1$ vertex sets have $2 \cdot 2^\ell$ edges. We can construct edges (and vertices) for one vertex set in time $O(\ell 2^\ell)$. Hence, we need $O(m\ell 2^\ell)$ time to construct the graph. To find the longest weight-constrained path, we need time $O(K|E|) = O(Km2^\ell)$ (in the graph every weight is positive, see [38] and the references therein for the method). Thus, the total time to construct the graph and to find the path is

$$O(m\ell 2^\ell) + O(Km2^\ell) = O\big((K + \ell)m2^\ell\big). \tag{5.3}$$

If $m = 2^\ell$, we can construct the graph and find out the path in $O\big((K + \log(m))m^2\big)$ time. We can also describe the growth of the running time as the total length of the radiation and the clone increases. Let $\ell = \log(m^k)$, for some $k \in \mathbb{N}$. Now, we need to consider selections of length $2^\ell = m^k$ in (5.3). The overall running time is $O\big((K + k\log(m))m^{k+1}\big)$. Similarly, $\ell = k + \log(m)$ implies selections of length $m2^k$ in (5.3) and $O\big((K + k + \log(m))m^2 2^k\big)$ running time.

Corollary 5.10. *Suppose an instance of* IKHO *with m knapsacks, where the total length of clone and radiation parts is* $\log(m^k)$ *($k \in \mathbb{N}$). This instance can be solved in time* $O\big((K + k\log(m))m^{k+1}\big)$. *If the total length is* $k + \log(m)$, *the running time is* $O\big((K + k + \log(m))m^2 2^k\big)$.

If $\ell = m$, then $k = m/\log(m)$ with the first result of Corollary 5.10 or $k = m - \log(m)$ with the second result. We set k into the second result and obtain the following asymptotic upper bound for the running time: $\big(K + m - \log(m) + \log(m)\big)m^2 2^{m-\log(m)} = (K + m)m^2 2^{m-\log(m)} = (K + m)m2^m$.

To check the running time, we put $k = m/\log(m)$ into the first result and obtain $\big(K + (m/\log(m))\log(m)\big)m^{1+m/\log(m)} = (K + m)m^{1+m/\log(m)}$. Since $m = 2^{\log(m)}$, this turns out to be $(K + m)(2^{\log(m)})^{m/\log(m)}m = (K + m)m2^m$ and hence, both results give the same running times for an instance with $\ell = m$ as expected.

Since $K \leq m$, we obtain $O(2^m m^2)$ running time for an arbitrary instance. (Note that we always have $K \leq m/(c+1)$.) Complete brute force enumeration has the same bound: bit string of length m is enumerated in $O(2^m)$ time and restrictions are checked in $O(m^2)$ time. However, if $K = m/(c + 1)$, we do not need restriction (4.15), because we can insert an item into the knapsack array as many times as we want. In these cases we can use the shortest path algorithms designed for directed acyclic graphs. We can compute the shortest paths from single source in directed acyclic graphs in time $O(|V| + |E|)$ [30].

Corollary 5.11. *If* $K = m/(c + 1)$ *and* $\ell = \log(m^k)$, *an instance of* IKHO *can be solved in time* $O\big(k\log(m)m^{k+1}\big)$. *If* $K = m/(c+1)$ *and* $\ell = k+\log(m)$, *then* $O\big((k + \log(m))2^k m^2\big)$ *time is enough.*

Proof. We need $O(m\ell 2^\ell)$ time to construct the graph, and the number of vertices and edges is $O(m2^\ell)$. Thus, in the first result the number of vertices is $O(m^{k+1})$ and the number of edges has the same bound. Constructing the graph takes $O\big(mk\log(m)m^k\big) = O\big(k\log(m)m^{k+1}\big)$ time. Both constructing the graph and finding the path together take $O\big(k\log(m)m^{k+1}\big) + O\big(m^{k+1} + m^{k+1}\big) = O\big((2 + k\log(m))m^{k+1}\big) = O\big(k\log(m)m^{k+1}\big)$ time.

In the second result we suppose that $\ell = k + \log(m)$. Thus, the number of vertices is $O(m2^k m)$ and the number of edges has again the same bound. Constructing the graph takes $O\big(m(k + \log(m))2^{k+\log(m)}\big) = O\big((k + \log(m)2^k m^2\big)$ time. The construction of the graph and the longest path together take $O\big((k+\log(m)2^k m^2\big) + O\big(2^k m^2 + 2^k m^2\big) = O\big((2+k+\log(m))2^k m^2\big) = O\big((k + \log(m))2^k m^2\big)$ time. □

Similarly, we can obtain running times for situations, where ℓ is some constant ($\ell = O(1)$). The upper bound is now given by $O\big((K + \ell)m2^\ell\big)$. Note

that asymptotically, $O\big((K+\ell)m2^{\ell}\big) = O(Km)$, which is less than the running time of the enumerative solution, see page 42.

From the application point of view, the above results describe the running times well enough. For example, in load clipping heuristics making a control plan for one item (group) at a time are often approximation algorithms [3, 6] (see Chapter 8). Typical lengths can be from 6 to 12 for the clone part and from 9 to 12 units for the radiation. Hence, the total lengths range often from 15 to 24 units. Note that there exist controlling utilities that have much longer total lengths. Moreover, some utilities use small K, while some of the utilities do not need K. For example, street lights or electricity generators have long total lengths, and K is not needed with the generators. In load clipping, the number of time units (the total lengths of items) depends on the granularity of the time interval. A typical value is 5 or 10 minutes, and the optimization interval (m) can be, for instance, 24 hours divided into 5 minutes (or 10 minutes) slots.

5.3 IKO in polynomial time without radiation

The next theorem can be seen as a generalization of Theorem 5.2.

Theorem 5.12. IKO $\in$ P *when* $n = O(1)$ *and* $u = 0$.

Proof. Let c_j, $j = 1, \ldots, n$, be the length of a clone of item j. We construct a graph as follows: let s be the source vertex and t the target vertex. We form m sets of vertices each consisting of at most $(m+1)^n$ vertices. Hence, the total number of vertices is at most $2 + m(m+1)^n$. Each vertex in a set has label $x_{k_1,\ldots,k_n}$, where $0 \le k_j \le c_j$, for each $j = 1, \ldots, n$. Because $c_j \le m$, we have at most $(m+1)^n$ labels.

If $k_1, \ldots, k_n > 0$, we insert an edge $(x_{k_1,\ldots,k_n}, x_{k_1-1,\ldots,k_n-1})$ from set l to set $l+1$, where $1 \le l \le m-1$. If index $k_q = 0$, for some $q = 1, \ldots, n$, we insert two edges from set l to $l+1$, where $1 \le l \le m-1$. The edges are $(x_{k_1,\ldots,k_q=0,\ldots,k_n}, x_{k_1-1,\ldots,k_q=0,\ldots k_n-1})$ and $(x_{k_1,\ldots,k_q=0,\ldots,k_n}, x_{k_1-1,\ldots,k_q=c_q,\ldots,k_n-1})$. The latter edge corresponds to the inclusion of item $l+1$ into the qth knapsack.

If $r > 0$ indices are zero, we insert 2^r edges. A vertex set contains at most $(m+1)^n$ vertices, and hence, the total number of edges between two sets is at most $2^n(m+1)^n$. The length is calculated only for these vertices and it corresponds to the profit obtained by including the r items indicated by the vertex label. Other edges have zero length. The n-weight vector has ones in the indices indicated by the r items, other positions have zeros.

We insert an edge only if it complies with constraint (4.18). If the label of a vertex contains r nonzero indices, we can check (4.18) in $O(r)$ time, because we do not have to worry about radiations of the items before or after the items corresponding to the edge (that is, to the choice made for the knapsack).

We can insert an item into a knapsack only if some index is zero. Combining this with the edges related to nonzero indices ensures that (4.19) cannot be violated.

The source vertex s has at most 2^n outgoing edges to the vertices of the first set. Targets of the edges correspond to the different selections of n items we can make in the first knapsack. Every vertex of the last set has ongoing edge to the target vertex t.

We have constructed an acyclic directed graph with $O(2^n + 2^n(m+1)^n(m-1) + (m+1)^n) = O(2^n(m+1)^{n+1})$ edges. The term 2^n is for edges from s to the first vertex set. Then there are $m-1$ vertex sets, where each set contains $(m+1)^n$ vertices and each vertex has at most 2^n ongoing edges. Last, we have $(m+1)^n$ edges from the last vertex set to t. The graph has at most $2+m(m+1)^n$ vertices. We can construct the graph in $O(2^n n + 2^n(m+1)^n(m-1)n + (m+1)^n n) = O(2^n(m+1)^{n+1}n)$ time. The n's are for the restriction checks.

Next we seek for the longest n-weight constrained path, where the weight constraint is K for each item (obtained from restriction (4.21)). Hence, we have to update $(K+1)^n$ different weight combinations in $2 + m(m+1)^n$ vertices. By Lemma 2.13, this can be done in polynomial time, as $K \leq m/(c_i+1)$ (and n is fixed). □

The edges from $x_{0,\ldots,0}$ in set l to set $l+1$, where $1 \leq l \leq m-1$, have the same target vertices as the edges departing from vertex s. This allows, for example, a solution without any selected item; we just go from s to vertex $x_{0,\ldots,0}$, and go to the vertex with the same label in every vertex set and finally arrive to t.

Figure 5.3 depicts the graph construction of Theorem 5.12. We have drawn the source and target vertices s and t as well as the first three vertex sets, and the last set of vertices. In this example we have three items such that $c_1 = 2$ and $c_2 = c_3 = 1$. The lengths of clones are 3, 2 and 2, respectively. From the source vertex s we can move, say, to the vertex 322 corresponding to the choice $x_{11} = x_{12} = x_{13} = 1$ (all three items are selected into the first knapsack). Other seven vertices to be chosen are 320, 302, 022, 300, 020, 002 and 000. These eight vertices in set l $(2 \leq l \leq m)$ are also the targets of edges departing from vertex v_{000} from the previous vertex set $l-1$.

Consider vertex 010 in vertex set 2. We can insert the first and the third item into knapsack 2 corresponding to set 2. Item two is inserted into knapsack

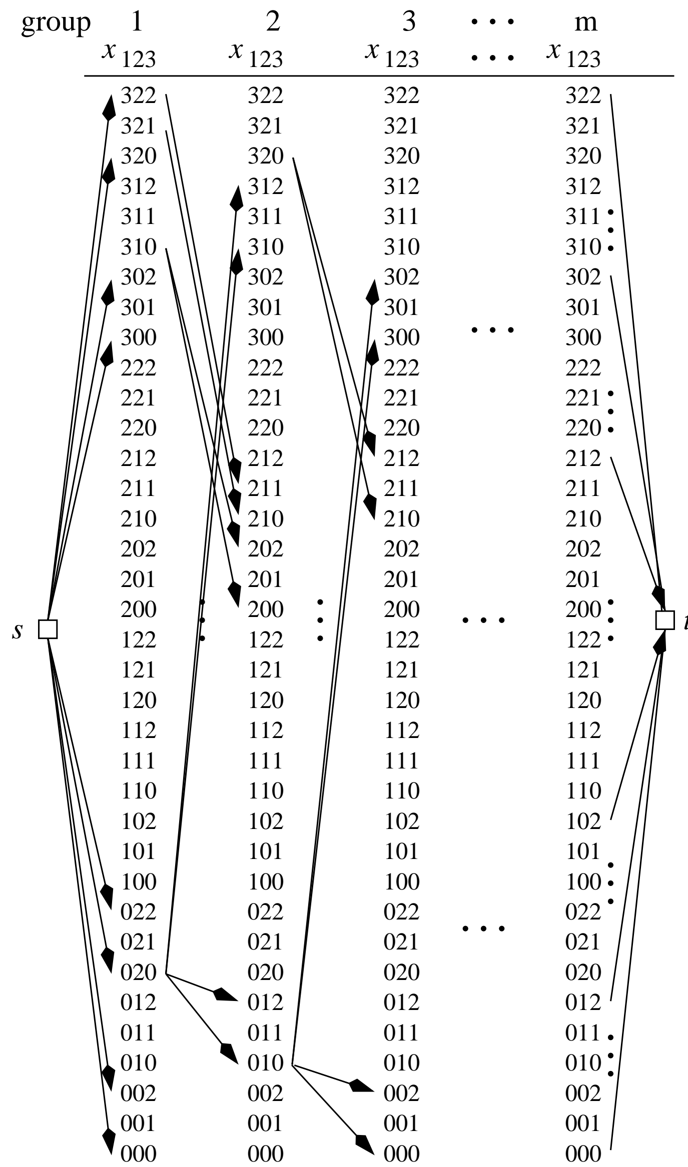

Figure 5.3: The graph structure for Theorem 5.12 with three items such that $c_1 = 2$, $c_2 = 1$ and $c_3 = 1$ (and each $u_i = 0$).

1, and hence, we can insert it again into knapsack 3. This is shown as four edges departing from 010. The target vertices are 302, 300, 002, and 000. Note that the only feasible path to 010 goes through 020 in the first vertex set. By fine tuning the algorithm, we can take this into account so that we skip some of the vertices, when we insert edges. The longer the clone lengths, the larger the savings in running time in the path finding phase.

Similarly, we can go from 320 to the vertices 210 or 212. The first target corresponds to a situation where item 3 is not inserted to the third knapsack. The second target, on the other hand, means that the item is inserted.

The rest of this section concerns the complexity of the presented method. If we neglect the number of edges, the construction of the graph and the search of the longest path by Lemma 2.13 gives $O\big(2^n(m+1)^{n+1}n\big)+O\big(K^n(2+m(m+1)^n)^2\big)$ running time. This is a rough upper bound.

When every index is nonzero, there is only one edge. The number of these vertices in a set is at most m^n, when the number of all vertices is at most $(m+1)^n$. The number of bit strings with l nonzeros is $\binom{n}{l}$ in a bit string of length n, and in other strings, a nonzero can have at most m different values. Thus, the number of strings with $n-l$ zeros is at most $\binom{n}{l}m^l$. When there are l zeros, we insert 2^l vertices in $O(2^l n)$ time (we check restriction (4.18) in $O(n)$ time). Hence, the number of edges is at most $\sum_{l=0}^{n}\binom{n}{l}m^l 2^{n-l}=(m+2)^n$ by binomial theorem 4.2, and we need $O(n(m+2)^n)$ time to insert the edges between two vertex sets.

Thus, the total time to construct the graph is $O\big(n2^n+mn(m+2)^n+n(m+1)^n\big)=O\big(((m+2)^{n+1}+2^n)n\big)$. This is much less than $O\big(2^n(m+1)^{n+1}n\big)$ used in Theorem 5.12.

When constructing the paths, we can store the n-weight combination tables of the vertices for one vertex set at a time. We update the longest paths as we proceed from vertex set to the next one. Each update takes a constant time, and an update is done for the target of each edge. Thus, the number of edges give $O(m+2)^n$ for the updates (recall that n is fixed). We also update a pointer to the longest path. Finally, in the target vertex t we use the pointer to the longest path in constant time.

Hence, we save some space and time leading to running time $O\big(2^n+m(m+2)^n K^n+(m+1)^n\big)=O\big(K^n(m+2)^{n+1}+2^n\big)$ for the longest path. The summands stand for the edges from s, then m times the edges between the vertex sets, where we have at most $(m+2)^n$ edges, and the edges to t, respectively. Hence, the total time to construct the graph and to find the longest path is $O\big(n((m+2)^{n+1}+2^n)\big)+O\big(K^n(m+2)^{n+1}+2^n\big)=O\big((K^n+n)(m+2)^{n+1}+(n+1)2^n\big)$.

In the following corollary we suppose that each item has its own clone

length c_j, $j = 1, \ldots, n$. We do not prove the corollary here, as the proof closely resembles that of Theorem 5.12, in which we used the fact that $c_j \leq m$. However, the interesting instances are those where c_j is much smaller than m: we can insert item at most $m/(c_j+1)$ times into the knapsack array. Note that $2^n(c_1+1)\cdots(c_n+1)$ is a rough upper bound for the number of edges between vertex sets where $(c_1+1)\cdots(c_n+1)$ is the number of vertices in a vertex set. Therefore, the total number of vertices is at most $2^n + 2^n m(c_1+1)\cdots(c_n+1) + (c_1+1)\cdots(c_n+1)$, which states the time needed for graph construction. By adding the time used for path we obtain the following corollary.

Corollary 5.13. *When $u = 0$,* IKO *can be solved in time*

$$O\big(K^n 2^n m(c_1+1)\cdots(c_n+1)\big).$$

5.4 IKO in polynomial time with radiation

The results in the previous section have practical implications, at least for the load clipping problem. Even though some of the previous results are contained in the following results, we can save some time by using the results given in the previous section, since we can construct the edges more directly than in the method to be presented next.

The following theorem is analogical to Theorem 5.9. This time, we have n items taken into account in the labels of the vertices. Compared to Theorem 5.9, we have more edges between vertex sets. Additional edges with the additional restrictions on the weights on edges also affect to the running time analysis.

Suppose first that the items have identical lengths. The graph to be constructed has vertices $v_{x_{ij}\cdots x_{i+\ell,j}}$, where selection $x_{11,\ldots,\ell 1,\ldots,1n,\ldots,\ell n}$ has 2^ℓ different bit combinations. Each vertex corresponds to a selection of n items into ℓ consecutive knapsacks.

Lemma 5.14. *Let the maximum lengths of clone and radiation parts be ℓ for each item. Every item selection $v_{11,\ldots,\ell 1,\ldots,1n,\ldots,\ell n}$ (ignoring any other items in the knapsack array) can be checked against restrictions* (4.18) *and* (4.19) *in $O(n\ell 2^{n\ell})$ time.*

Proof. Consider a selection $v_{11,\ldots,\ell 1,\ldots,1n,\ldots,\ell n}$. We are examining ℓ consecutive knapsacks: there can at most ℓ ones, and the effects of an item can be calculated in $O(\ell^2)$ time (and at the same time we can check the restrictions). Because we have n items, a selection $v_{11,\ldots,\ell 1,\ldots,1n,\ldots,\ell n}$ can be checked against (4.18) and (4.19) in time $O(n\ell^2)$.

Next consider a set of all selections containing $2^{n\ell}$ elements of length $n\ell$. Combining two bit strings (and the corresponding results in the knapsacks) takes $O(2n\ell)$ time. The rest of the proof closely follows that of Lemma 5.5 and we conclude that the total time consumption is $O(n\ell 2^{n\ell})$. □

The proof of the following theorem mimics the proofs of Lemmas 5.6–5.8.

Theorem 5.15. IKO ∈ P *when* $n = O(1)$, $\ell = \log(m)$.

Proof. Let I_{ij}, p_{ij}, w_{ij}, b_j, K, n, and m form an instance of IKO. We also suppose that the total length of radiation and clone parts is $\ell = \log(m)$ for each item.

We start from the source vertex s and insert at most $2^{n\ell} = m^n$ outgoing edges to the first vertex set. (Note that $m = 2^\ell$.) The vertices in the first vertex set correspond to every combination to select n items into the first ℓ knapsacks. If some combination is impossible, to comply with (4.18) and (4.19), the corresponding edge is not inserted. By Lemma 5.14, the first edges take $O(n\ell 2^{n\ell})$ time.

We insert an edge $(v_{x_{ij}\cdots x_{i+\ell,j}}, v_{x_{i+1,j}\cdots x_{i+\ell+1,j}})$ between consecutive vertex sets, if $v_{x_{i+1,j}\cdots x_{i+\ell+1,j}}$ is a legal item selection and does not violate (4.18) and (4.19).

The profit and restriction checks for every vertex between two consecutive vertex sets is done in $O(n\ell 2^{n\ell})$ time. We construct the edges between feasible vertices in $O(2^n 2^{n\ell})$ time (number of edges times number of vertices). Hence, the total time needed to make edges between two vertex sets is $O((2^n + n\ell)2^{nl})$.

We use radiations and clone lengths as in the one item case. That is, for each item, the bit corresponding to the first position of the clone part (right after u_l part) tells, if we should calculate the effects of including the item (length for the edge).

Also weights are set similarly to the one item case. This time, each item has its own coordinate in the n-weight vector. If item j is included ($x_{i+u_l+1,j} = 1$), the corresponding jth bit in the resource bound of the edge is one, otherwise it is zero.

Now, we have a system that contains every selection of length ℓ in the knapsack array with n items. We have linked selections so that every legal selection is attainable. The number of selections is $O(m2^{n\ell}) = O(m^{n+1})$. This bounds the total number of vertices in the vertex sets. The number of edges between two consecutive vertex sets is $2^n 2^{n\ell} = 2^n m^n$ (we have 2^n decisions for selections, and the number of selections is $2^{n\ell}$ in ℓ knapsacks). The graph construction takes totally $O\big(n\ell 2^{n\ell} + m(2^n + n\ell)2^{n\ell} + n\ell 2^{n\ell}\big) = O\big((2^n + n\log(m))m^{n+1}\big)$ time.

Next we have to find the best set of selections. Condition (4.21) restricts, for each item, the number of ones a path can contain. This leads to the n-weight-constrained longest path problem. Recall that our graph is acyclic and directed. As K is polynomially bounded (on m), we can form the longest n-weight-constrained path in polynomial time (on m) by Theorem 2.13 (recall that n is fixed). □

To obtain the second set of vertices from the first set, we drop the first knapsack and add a new knapsack $\ell+1$. As with Theorem 5.9, the consecutive vertex sets contain $\ell - 1$ common knapsacks. Let the common subselection be $x_{2j}, \ldots, x_{\ell j}$, for $j = 1, \ldots, n$. The items in the first knapsack x_{1j} as well as the items in the new knapsack $x_{\ell+1,j}$ can have 2^n value combinations (this is in line with IKHO), which is also the upper bound on the number of incoming and departing edges.

We can extend the method to handle individual total lengths ℓ_j for each item. Figure 5.4 demonstrates the case $n = 2$ and $\ell_1 = 3$ and $\ell_2 = 2$. Each vertex set has 32 vertices and there are 4 edges departing each vertex. Moreover, we have to consider every 2-weight combination when searching for the longest path. The size of the combination table is $(K+1)^2$. Figure 5.4 also demonstrates how to handle the last vertex set, when every item has its own total length ℓ_i.

As an example, consider vertex $x_{111,11}$ at vertex set "123,12", and suppose that $u_l = 0$. This corresponds to a situation, where we include items 1 and 2 into the first two knapsack and item 1 also to the third knapsack. The length of edges $(s, v_{111,11})$ corresponds to the inclusions $x_{1,1} = 1$ and $x_{1,2} = 1$. This is the common length of the four ($= 2^2$) edges departing from vertex set "123,12" to vertex set "234,23". Vertex $x_{110,10}$ in the second vertex set corresponds to a situation where item 1 is not inserted into the fourth knapsack and item 2 is not inserted into the third knapsack. Vertex $x_{110,11}$ corresponds to a situation where item 1 is not inserted into a fourth knapsack while item 2 will be inserted into the third knapsack.

To analyze the running time, note that the graph construction takes time $O(m(2^n + n\ell)2^{n\ell})$. We have $O(m2^{n\ell})$ vertices and the number of edges is $O(m2^n2^{n\ell})$. By Lemma 2.13, we need $O(m(2^n+n\ell)2^{n\ell} + K^n|V|^2)$ time, which equals to $O(m(2^n + n\ell)2^{n\ell} + K^n(m2^{n\ell})^2) = O(K^n m^2 m^{2n}) = O(K^n m^{2(n+1)})$.

Remark, that because our graph has a special structure, the actual time consumption is smaller than the above bound given in Lemma 2.13. We can proceed from one vertex set to the next, and store the n-weight combination tables of dimension $O((K+1)^n)$ in m^n vertices of the vertex set.

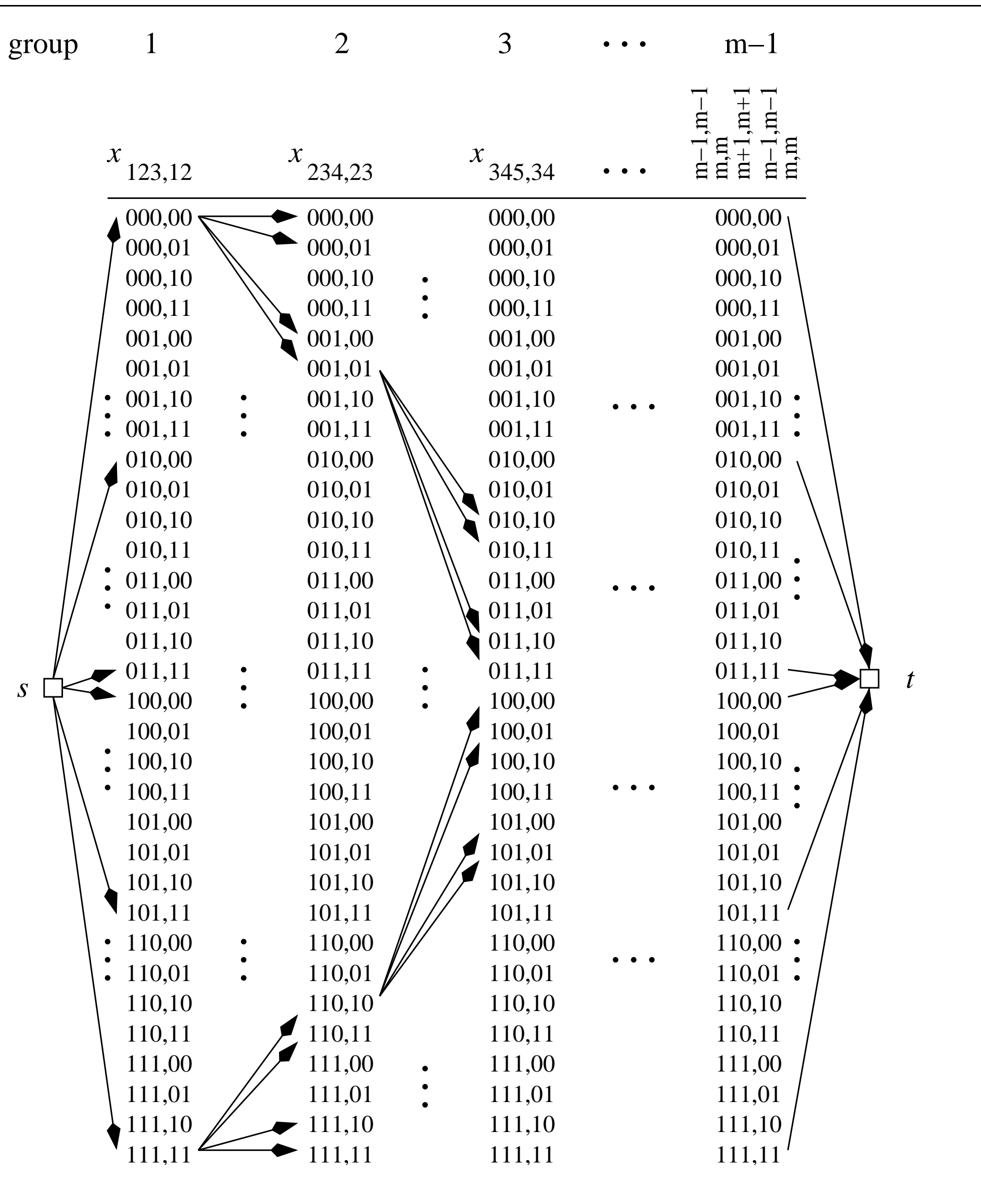

Figure 5.4: The graph structure for Theorem 5.15 with two items ($n = 2$), where $\ell_1 = 3$ and $\ell_2 = 2$. We have $m - 1$ vertex sets, because the minimum total length is two units.

A path contains $O(m)$ vertices, and a vertex set contains m^n vertices. Each vertex contains at most $(K+1)^n$ paths. Hence, the space consumption is $O(K^n m^{n+1})$. The longest path algorithm now runs in time $O(2^{n\ell}) + O(K^n 2^n 2^{n\ell} m) + O(2^{n\ell}) = O(K^n 2^n 2^{n\ell} m) = O(K^n 2^n m^{n+1})$. The first term stands for the edges from the source, the second for the path checks with every weight combination in each vertex, and the last one is for the edges into the target vertex. Note that the paths in the n-weight combination table need to be traversed in $O(m)$ time only at vertex t when we seek the longest path. We can find the longest path in constant time, because we have kept a pointer to it.

Thus, the total time is $O\big(m(2^n + n\ell)2^{n\ell}\big) + O\big(K^n 2^n 2^{n\ell} m\big) = O\big((K^n 2^n + 2^n + n\ell)2^{n\ell} m\big) = O\big((K^n 2^n + 2^n + n\log(m))m^{n+1}\big)$. This is much less than the bound given above and suggested by Lemma 2.13.

We can relax some of the assumptions in the proof. Each item can have its own maximum number K_j of inclusion times, and the total length ℓ_j (that is, the length of left and right radiation u_{jl} and u_{jr}, and the length of clone c_i together). This means that we replace every $n\ell$ with $\ell_1 + \cdots + \ell_n$.

Corollary 5.16. *Assume that each item has parameters K_j and ℓ_j.* IKO *can be solved in $O\big((K_1 \cdots K_n 2^n + 2^n + \ell_1 + \cdots + \ell_n)2^{(\ell_1+\cdots+\ell_n)} m\big)$ time.*

In the time bound of Corollary 5.16, $K_1 \cdots K_n$ is for filling up the n-weight combination table, while $2^{\ell_1+\cdots\ell_n}$ is for the number of vertices in one vertex set. The additional m is the number of vertex sets and 2^n is the upper bound for in and out degree of each vertex in a vertex set. By relaxing $\ell = \log(m)$ to $\ell = \log(m^k)$ or to $\ell = k + \log(m)$, we obtain the following.

Corollary 5.17. *If $\ell = \log(m^k)$, time $O\big((K^n 2^n + 2^n + nk\log(m))m^{nk+1}\big)$ is enough for* IKO. *If $\ell = k+\log(m)$, time $O\big((K^n 2^n + 2^n + n(k+\log(m)))2^{nk} m^{n+1}\big)$ is enough for* IKO.

Proof. If $\ell = \log(m^k)$, we have $2^{nk\log(m)} = m^{nk}$ and $O\big((K^n 2^n + 2^n + n\ell)2^{n\ell} m\big) = O\big((K^n 2^n + 2^n + n + k\log(m))m^{nk+1}\big)$.

If $\ell = k + \log(m)$, then $2^{n(k+\log(m))} = 2^{nk} m^n$. Now $O\big((K^n 2^n + 2^n + n\ell)2^{n\ell} m\big) = O\big((K^n 2^n + 2^n + n(k + \log(m)))2^{nk} m^{n+1}\big)$. □

If $K = m/(c+1)$ for each item, we do not use restriction (4.21). In these cases we can use the shortest path methods (our graph is directed and acyclic), and the result can be formulated as follows:

Corollary 5.18. *Suppose that $K = m/(c+1)$ for each item. If $\ell = \log(m^k)$,* IKO *can be solved in $O\big(((2^n + nk)\log(m))m^{nk+1}\big)$ time. If $\ell = k + \log(m)$,* IKO *can be solved in $O\big((2^n + n(k + \log(m)))2^{nk} m^{n+1}\big)$ time.*

Proof. If $\ell = \log(m^k)$, the number of vertices is $2 + m^{nk}m = 2 + m^{nk+1}$ and the number of edges is $m^{nk} + 2^n m^{nk} O(m) + m^{nk} = O(2m^{nk} + 2^n m^{nk+1})$. The first term is for the edges from s, the next is for the edges between vertex sets, and the last term is for the edges incident with t. Thus, the running time for the shortest path method is $O(|V|+|E|) = O(2+m^{nk+1}+2m^{nk}+2^n m^{nk+1}) = O(2^n m^{nk+1})$. The construction time together with the path search time gives $O\big(m(2^n+nk)\log(m)m^{nk}\big)+O\big(2^n m^{nk+1}\big) = O\big((2^n+(2^n+nk)\log(m))m^{nk+1}\big)$.

If $\ell = k+\log(m)$, the number of vertices is $2+2^{nk}m^n m$. Thus, the shortest path running time is $O(2 + 2^{nk}m^{n+1} + 2^{nk}m^n + 2^n 2^{nk} m^n(m-1) + 2^{nk}m^n) = O((2^n + 1)2^{nk}m^{n+1}) = O(2^{nk+n}m^{n+1})$. The terms are obtained as in the previous paragraph. The total time is $O\big(m(2^n + n(k + \log(m)))2^{nk}m^n\big) + O\big(2^{nk+n}m^{n+1}\big) = O\big((2^n + 2^n + n(k + \log(m)))2^{nk}m^{n+1}\big)$. □

5.5 MDKP and IP in polynomial time

In this section we present new instances of MDKP and IP that are polynomial time solvable. Consider an instances of 0–1 MDKP having m items and m knapsacks b'_i, profits p'_i, and weights given by restriction matrix W', which is a band matrix with bandwidth $O(\log(m))$. (In a band matrix W' we have $w'_{ij} = 0$, for $|i-j| \geq w$, where w is the width.)

We partially transform 0–1 MDKP into IKHO, and form the "selection graph" used in Section 5.2. We use restrictions normally. We set interactions $I_i(k) = w'_{ik}$, weights $w_i = 1$, and knapsack sizes $b_i = b'_i$. The lengths of edges are obtained directly from the profits of the present 0–1 MDKP case (and not through interactions). Hence, restriction (4.13) is fulfilled, because $c = 0$, and (4.15) is not needed as $K = m/(c+1)$ (we may include item into all knapsacks, if other restrictions allow it).

Bandwidth of w is seen as a clone (with clone and radiation part) of length $\ell = 1+2w$. Analogically to Corollary 5.11, we are able to estimate the running times as a function of the bandwidth.

Corollary 5.19. 0–1 MDKP *having band matrix restriction of width* $w = k+\log(m)$ *can be solved in time* $O\big((k+\log(m))2^{2k+1}m^3\big)$. *Width* $w = \log(m^k)$ *implies time* $O\big(k\log(m)m^{2k+1}\big)$.

Proof. As $K = m/(c+1)$, we do not need (4.15). Hence, we use Corollary 5.11, which implies $O\big((v + t\log(m))2^v m^{t+1}\big)$ running time with total length $\ell = v + t\log(m)$. Now, set $\ell = 1 + 2k + 2\log(m)$, (thus $t = 1 + 2k$ and $v = 2$), giving $O\big((1 + 2k + 2\log(m))2^{1+2k}m^{2+1}\big) = O\big((k + \log(m))2^{2k+1}m^3\big)$. Further, $\ell = 1 + 2k\log(m)$ gives $O\big((1 + 2k\log(m))2^1 m^{2k+1}\big) = O\big(k\log(m)m^{2k+1}\big)$. □

Recall Theorem 2.13. It can be used to solve 0–1 MDKP instances. The above case needs $O(K^m m)$ time with a graph having m pairs of edges each corresponding to either selecting an item or not, each edge having m restrictions, and K being the largest number (or the largest size of the m knapsacks). (See also [80].) Further, a brute-force enumeration gives $O(m^2 2^m)$ time [79, p. 125]. Even though we have neglected to consider the lengths of elements of instances, we dare to say that Corollary 5.19 gives much more efficient algorithm for the instances considered.

As an example, consider the following 0–1 MDKP instance. Let $p'_i = i$ $(i = 1, \ldots, 10)$, and let the width of the band matrix be 1. Suppose also that $w'_{i,i-1} = w'_{ii} = w_{i,i+1} = i$. Last, let $b'_i = 30$ (hence, every edge will be shown). Figure 5.5 shows the edges, all directing from left to right. In this example, radiation has length one to the left and length one to the right from the clone part, whose length is one. Thus, a selection is three knapsacks wide and the middle knapsack gives the profit for the edges, except for the edges from s (the leftmost knapsack gives the profit) and to t (the second and third knapsack).

Figure 5.5 shows the length (the profit), if it is not zero. For example, the edge from s to 110 has length 1, because we take into account the profit of the inclusion of the first item (u_l part). Note that we do not use the weights here, as we are allowed to make as many inclusions as we want.

We have marked with a small "a" the edges that have nonzero length. For example, the edge from 011 in the first vertex set to 110 in the second vertex set has length 2, because the edges from the first to the second vertex set take into account the profits of the inclusion of the second item. There are two numbers for each "a", as there are two edges from each vertex in the vertex sets, for example, edges (011,110) and (011,111) between each vertex set.

Edges going to the target vertex t have different weights. There we have to add also the profits of inclusions of the tenth item in addition to the ninth item.

When we construct the graph, we check restriction (4.12) against every selection. For example, vertex 111 in the sixth vertex set corresponds to $x_6 = x_7 = x_8 = 1$. If some of the restrictions (4.12) were violated, the corresponding edges would be missing.

If we want to model the solution where only items 6, 7, and 8 are included, our path starts from s and goes to the upper most vertex of the first vertex set. The path continues to 000 in the third set, 001 in the fourth, and 011 in the fifth. The seventh knapsack has weight 6+7+8=21. The path continues from sixth set to 110 and to 100, from which it goes to t. Figure 5.5 draws this path in thicker line.

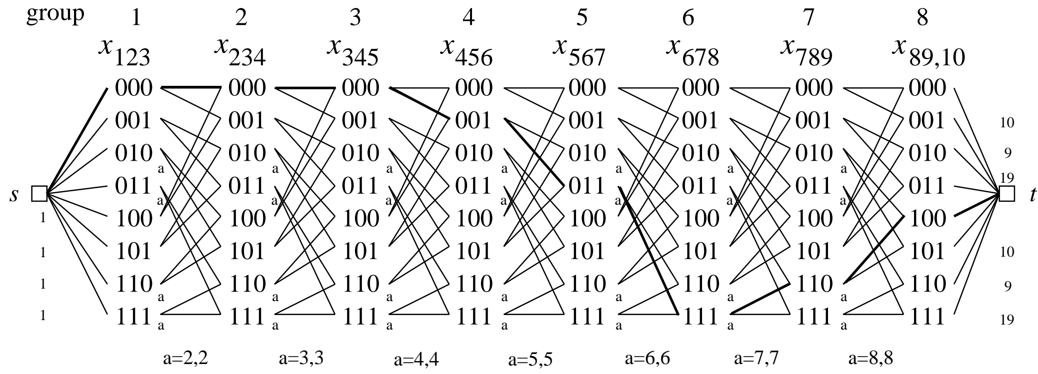

Figure 5.5: The graph obtained for 0–1 MDKP example. We have 10×10 matrix, whose band-width is 1. The thicker line shows a solution path corresponding to $x_i = 0$, $i = 1, \ldots, 5, 9, 10$, and $x_6 = x_7 = x_8 = 1$.

The profits corresponding to the selection $x_6 = x_7 = x_8 = 1$ (and $x_i = 0$, where $i \neq 6, 7, 8$) are given by the edges $(011, 111)$ with profit six (from fifth to sixth set), $(111, 110)$ with profit seven (from sixth to seventh set), and $(110, 100)$ with profit eight (from seventh to eighth set).

After applying the method designed for IKHO to 0–1 MDKP, we would like to apply the proof of Corollary 5.3 accordingly. The counterparts of m restrictions (4.13) of IKHO in 0–1 MDKP are knapsack restrictions $x_i + \cdots + x_{i+c_i} \leq 1$, for $i = 1, \ldots, m$, when inserting an item into the knapsack i in IKHO. If we set $x_i = 1$, then $x_{i+1} = \cdots = x_{i+c_i} = 0$. Knapsack $i + c_i + 1$ can take the next item. When we construct the graph, we draw an edge from (A_i, B_{i+c_i+1}) with length p_i. As there are m rows, we draw m edges corresponding to the restriction rows among other edges. Restriction (4.15) of IKHO can be kept unchanged in 0–1 MDKP. If 0–1 MDKP contains at least one such restriction, then we use the longest weight-constrained path algorithm. Otherwise, we use the longest path algorithm. As the graph is directed and acyclic, we use the shortest path algorithm.

The above application of Corollary 5.3 restricts 0–1 MDKP instances considerable and therefore, it is not as interesting as the result given in Corollary 5.19. Anyhow, the above construction resembles the linear programming problem, where the restrictions are difference constraints [30]. That is, the restrictions are of form $x_j - x_i \leq b_k$. Difference constraints can be used to model, for example, relative timing constraints. Shortest paths can be applied to find a feasible solution for difference constraints [30]. However, if we are to account profits p_i, when $x_i = 1$ (or when $x_i \geq 1$), the application of shortest

paths for difference constraints is not so obvious.

Furthermore, the above results raise another natural question at extending the algorithms of IKO instances. Let an instance of 0–1 MDKP consist of nm items and m knapsacks, where the restriction matrix W' is a concatenation of n band matrices: $W' = [W'_1\ W'_2\ \cdots\ W'_n]$, where W_i $(i = 1, \ldots, n)$ is an $m \times m$ band matrix with a bandwidth of size $O(\log(m))$.

Let w^k_{ij} be the weight of position i, j of W'_k. Now we set parameters for IKO: $w_{ij} = 1$, interactions $I_{ij}(k) = w^j_{ik}$, and $c = 0$, $K = m/(c+1)$ and $b_i = b'_i$. Again, we do not map profits from an 0–1 MDKP instance directly to an IKO instance, but rather use them as lengths in the constructed graph.

Corollary 5.20. 0–1 MDKP *with n "concatenated" band matrix restrictions of width $\ell = \log(m^k)$ can be solved in time $O\big((2^n + n + 2kn\log(m))2^n m^{2kn+1}\big)$. Width $\ell = k + \log(m)$ implies time $O\big((2^n + n + 2nk + 2n\log(m))2^{n+2nk} m^{2n+1}\big)$.*

Proof. Because $K = m/(c+1)$, we do not need restriction (4.21). By combining the results of Corollary 5.18, we obtain that $\ell = v + t\log(m)$ implies $O\big((2^n + n(v + t\log(m)))2^{nv} m^{nt+1}\big)$ time. We map ℓ to $1 + 2\ell$, that is, set $v = 1$ and $t = 2k$ for the first result and obtain $O\big((2^n + n(1 + 2k\log(m)))2^{n\cdot 1} m^{n\cdot 2k+1}\big)$.

Set $v = 1 + 2k$ and $t = 2$ for the second result and now $O\big((2^n + n(1 + 2k + 2\log(m)))2^{n(1+2k)} m^{2n+1}\big) = O\big((2^n + n + 2nk + 2n\log(m))2^{n+2nk} m^{2n+1}\big)$. □

Corollary 5.20 solves in polynomial time also a class of instances of 0–n MDKP, where the weight matrix is an $m \times m$ square matrix and has bandwidth $u = O(\log(m))$, and where each variable can range between 0 and $2^n - 1$ and n is fixed. (In 0–n MDKP we are to maximize pz such that $Wx \leq b$ and $x_i \in \{0, \ldots, n\}$.)

We form a "concatenated" matrix from n matrices. Each submatrix has a factor 2^i, $i = 0, \ldots, n-1$, which is used to multiply the profits and weights. Now we have a situation in which Corollary 5.20 can be applied. The concatenated band matrix restrictions are $W'' = [2^0 W'\ 2^1 W'\ \cdots\ 2^{n-1} W']$. Similarly, the profit vector is $p'' = [2^0 p'\ 2^1 p'\ \cdots\ 2^{n-1} p']$. Knapsack capacities are the same. The value of the variable j is obtained by summing appropriately over the values at each index j of n parts. Because our graphs are directed and acyclic, the above discussion and results also hold for the 0–1 and 0–n IP problems.

For example, suppose an instance of 0–n IP with one band matrix of width $u = 1$ and the parameters given as in the 0–1 MDKP example: $p'_i = i$ $(i = 1, \ldots, 10)$, $w'_{i,i-1} = w'_{ii} = w_{i,i+1} = i$, and $b'_i = 30$. Suppose that each variable can range between 0 and $7 = 2^3 - 1$.

Hence, the concatenated band matrix is $W'' = [1W'\ 2W''\ 4W'']$ and the profit vector is $p'' = [1p'\ 2p'\ 4p']$. Here W'' is a 10×30 matrix and p'' is a 30×1 vector.

The graph has 10 vertex sets, each containing $2^{3 \cdot 3} = 512$ vertices. Moreover, we have $2^3 \cdot 512$ edges between two consecutive vertex sets. Vertex s has 512 outgoing vertices and vertex t has as many incoming edges. The profits for the edges from s are given by the u_l part, that is, by the left most knapsack.

For example, vertex $100, 110, 100$ in the first vertex set corresponds the decision $x_1 = 7$, $x_2 = 2$, and $x_3 = 0$. The length of the edge $(s, v_{100,110,100})$ is $1 \cdot 1 + 2 \cdot 1 + 4 \cdot 1 = 7$. The first knapsack has weight $1 \cdot 1 + 2 \cdot 1 + 4 \cdot 1 = 7$ from the decision $x_1 = 7$ and $2 \cdot 2$ from the decision $x_2 = 2$, totalling 9.

As an another example, consider vertex $011, 001, 001$ in the second vertex set. This vertex corresponds to the decision $x_2 = 0$, $x_3 = 1$, and $x_4 = 7$. Weight of the third knapsack can be calculated: $1 \cdot 3 + 7 \cdot 4 = 31$, which is more than $b_3 = 30$. Hence, there is no incoming edges to this vertex. Note that we can give some estimations about the weights in knapsacks 2 and 4, but the exact amount can only be calculated in the vertex sets 1 and 3, respectively. If the knapsack size of the third knapsack were larger, $011, 001, 001$ would have incoming edges of length 0, since the edges have to depart from vertices such that the middle bits are zeros (we see this from the vertex label, the first positions are zeros).

Chapter 6

Applications of the IK model

In this chapter we introduce some applications that can be modeled and solved with interactive knapsacks using IKHO and IKO. In Section 6.1 we give applications in scheduling, and in Section 6.2 other applications, which also interpret knapsack array as a time line. An introduction to the scheduling theory and solution methods is given, for instance, by Pinedo [88].

Because the applied IK model is general, it can be applied to several problems like, say, 0–1 MDKP or GAP. Usually, the problem specific methods are most suitable and the general optimization techniques must be tailored to different problems.

The given applications work as examples in applying IK model to different problems. We are able to demonstrate the close relationship between different problems, and in some cases IK model can give new insights in well-known and much studied problems. For example, if fixed length clones are usually used, extending the problem to use variable clone lengths or radiation may give new efficient ways to solve or to extend the problem. Yet, we may find new applications for the methods of Chapter 5.

6.1 Applications in scheduling

The knapsack array can be interpreted as a discrete time interval. We can define planning problems as follows: for each task (or act), corresponding to a clone, find the schedule maximizing the profits so that successive tasks are disjoint, and that the resources consumed by all of the tasks are not over exhausted (that is, items fit into the knapsacks).

This interpretation of the model contains different single machine scheduling problems: we have $w_{ij} = b_i$, for each task j, all knapsacks have equal capacity ($b_i = b_1$, for $i > 1$), and hence, only one task at a time can be se-

lected. We can also restrict the number of times a task is selected; one is a common bound for the number of possible selections. This is obtained by setting $K = 1$ in IKO in (4.21). Depending on the type of the objective function (if not using (4.22)) and other additional constraints, this kind of problem may be NP-complete [45].

Suppose that for all items j, we have $0 < p_{1j} < 1$ and $p_{ij} = p_{1j}^i$, for $1 < i \leq m$, or that the profits are of the form $p_{ij} = p_{1j}a^{i-1}$, where $p_{1j} > 0$ and $0 < a < 1$ is the discounting factor (that is, profits are discounted over time). Now the optimal solution will place items $j = 1, \ldots, n$ to leftmost positions available in the knapsack array, because the profits are larger in the left end of the array. Set $K = 1$ in (4.21) so that each item j will be selected at most once. Because each selection increases sum (4.22), all items will be selected. Note that the clones connected to the items are separate because one clone consumes all the capacity of the knapsacks involved by (4.18) and by the fact that $w_{ij} = b_i$. The optimal solution is clearly the one which minimizes the total completion time.

Now we have several options. By adding release times e_j and deadlines d_j for items j, that is, by adding constraints

$$i \in [e_j, d_j], \qquad \text{when } x_{ij} = 1,$$

we end up with SEQUENCING WITH RELEASE TIMES AND DEADLINES, a well known NP-complete problem (see Appendix A.4 and the proof of Theorem 4.7). Actually, we do not have to add these constraints explicitly. We can set $w_{ij} > b_i$, if $i \notin [e_j, d_j]$, not allowing to start the task before release or to stop after deadline. If this is too restrictive, we may handle these constraints by using profits appropriately and neglecting the weights. Other single machine scheduling problems are discussed in [45]. Some of these problems can be exposed directly with IKO but some need changes to IKO.

On the other hand, instead of considering the release times and deadlines we may choose m to be $\sum_{j=1}^{n} c_j$, where c_j is the length of task (clone) j, so that we cannot have idle time between the tasks. Because each p_{ij} depends on both task (clone) j and position i, we have a single machine scheduling in the sense of [59]. In [59] we minimize the objective which is a sum of functions on the completion time for each task. (The problem setting and different objectives are given in Appendix A.1.)

Note that we can expose equivalent cost-functions as in [59] without altering the objective (4.22). The weighted sum of earliness and tardiness for tasks (or jobs) can be calculated by choosing profits p_{ij} appropriately. Let the profit coefficients of earliness and tardiness be e_j and t_j, respectively. Further,

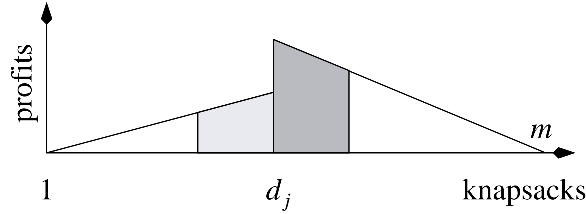

Figure 6.1: Due date and profits.

let the due date of task j be $d_j \in [1, m]$. We turn the minimization of disadvantages of misplacement of the task (earliness or tardiness) used in [59] to the maximization of advantages of appropriate placement of the task. Figure 6.1 contains knapsack d_j and two lines corresponding to profits given by the placement of task j. If the last knapsack used by task j is d_j, our profits (4.22) are maximized.

We achieve this behavior by setting $p_{d_j,j} = \max\{(d_j - 1)e_j, (m - d_j)t_j\}$,

$$p_{d_j-k,j} = p_{d_j,j} - e_j k, \tag{6.1}$$

$(1 \leq k \leq d_j - 1)$ and

$$p_{d_j+k,j} = p_{d_j,j} - t_j k, \tag{6.2}$$

$(1 \leq k \leq m - d_j)$. All profits are positive. Unlike in Figure 6.1, we have to choose constants e_j and t_j such that $(d_j - 1)e_j > (m - d_j)t_j$. Actually, e_j has to be so large that objective (4.22)—if item can be freely positioned—gives the maximum when the end of the clone is at d_j. Remembering this, we can replace (6.1) with any increasing function and (6.2) with any decreasing function. Another possibility is to use interaction I_{ij}, which takes into account the distance from d_j (so that $I_{d_j,j}$ will give the best results).

Again, knapsacks have capacity for one copy ($w_{ij} = b_i$). Restriction (4.19) ensures that the clones do not overlap, and hence, we will not violate condition (4.18).

If e_j and t_j are as in Figure 6.1, tasks may pass the due date and give at the same time better profits. Compare the profits given by two equal length tasks: a task ending at d_j (light grey area in Figure 6.1) and a task starting at d_j (dark grey area). The tardiness coefficient is so large that it gives better profit to item starting at due date than to item, whose clone ends exactly at due date.

By setting $e_j = 0$ our objective counts the weighted sum of tardiness. We can use methods given in [59]. The problem, however, remains NP-hard [45, 59].

IK model can also be exploited for some multiprocessor scheduling problems. We first set $w_{i1} = w_{ij}$, for $1 < j \leq n$ and for all i (all items have equal weight on each knapsack). Then we set $b_i = Mw_{i1}$ so that at most M processors can be used at a time (in Figure 6.2 we have $M = 6$).

We use the same approach as earlier by discounting profits $0 < p_{1j} < 1$ over time (that is, $p_{ij} = p_{1j}^i$, $i = 1, \ldots, m$). Because the profits are positive, we add each task to the schedule because it increases the total profit. Because the profits decrease as i increases, the tasks will be placed to as left as possible. We choose knapsack sizes to fit in items. Now, in the optimal solution the completion time of the last task is minimized [65].

This is almost the multiprocessor scheduling problem of [45], a well-known NP-complete problem. In this setting it is easy to see the number of processors needed; the problem is to find the correct processors. One interpretation is to have capacity consumed like in Figure 6.2 (a) identifying at the same time the processor; the third processor is used even thought the capacities are handled as in Figure 6.2 (b). This also implies that the tasks have to consume resources equally, that is, $w_{i1} = w_{1j}$, for all i, j.

However, if processors or tasks are not identical, there may be holes left like in Figure 6.2 (a) and it is harder to determine which processor is used for each task. Multidimensional knapsack arrays to be presented in Chapter 7 give a more natural interpretation to multiprocessor problems.

If we have setup or clean-up times [88], we can employ early or back radiation, respectively. Radiation can depend on a task (varying length radiation) or it can be processor specific (on identical processors constant length radiation for each task). Setup time occurs, for example, if we have to configure a processor. Clean-up can be thought more naturally in the machine scheduling context.

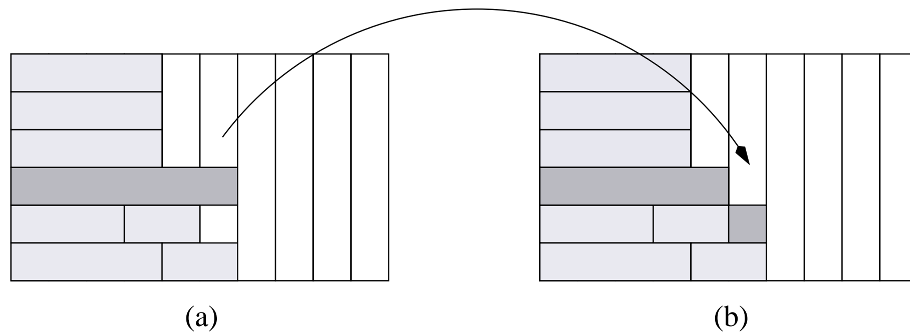

Figure 6.2: Identical processors.

If we use radiation, we have to modify the use of interaction so that radiation does not affect the profits but only the weights (or at least it affects differently the profits than the weights). Moreover, radiation can take all or part of the processors time on setup (that is, $0 \leq r_{ij}(k) \leq 1$).

6.2 Other applications

Some applications given in this section require modifications to IKO problem setting. However, the modifications are small and the basic concepts of IK model, cloning and radiation, are still applied.

By adding extra restrictions of the form

$$\mathrm{cmin}_j \leq c_{ij} \leq \mathrm{cmax}_j \tag{6.3}$$

(minimum and maximum lengths for a clone) and some minor application specific modifications to IKO with variable length clones, we can model energy management problems, like unit commitment [28] and load clipping [3, 6] (see Chapter 8). In energy management we often *have to* use back radiation and sometimes we can also use early radiation.

Constraints (6.3) can be added to IKO as extra constraints or we can modify the sum (4.22) by changing the interaction $I_{ij}(\cdot)$. We replace

$$\sum_{k=i-u_{ij}}^{i+c_{ij}+u_{ij}} I_{ij}(k) p_{kj}$$

with

$$p_{kj} C_{ij}(c_{ij}) \Big(\sum_{k \in C_{ij}} C_{ij}(k) p_{kj} + \sum_{k \in R_{ij}} U_{ij}(k) \Big),$$

where the cloning interaction $C_{ij}(k) = 1$, when $k \in [i + \mathrm{cmin}_j, i + \mathrm{cmax}_j]$, and 0 otherwise. The coefficient $C_{ij}(c_{ij}) = 0$, if c_{ij} is shorter than cmin_j or longer than cmax_j, thus reducing the profits to zero. The sum in constraint (4.18) will be modified similarly.

We can also try to apply load clipping to "load shedding in the internetwork" to reshape the traffic loads at communication lines [99, pp. 390–392]. Traffic packets could have priorities based on prices [99], which means that profits of IK model could be used. Cloning has a natural interpretation: it is the time we prevent some hosts from sending packets. Radiation is more interesting: while people try to make connections and do not succeed, they probably try again later. For instance, if the shaping occurs at the access point of a company [99] for www-traffic, some people will try again. The time

after which they try is random but after we release the line, there will be more consumption than without the load shedding. One can, of course, try to model the shape of the increased consumption, the shape of back radiation, like in load clipping [16, 29, 64, 91]. If the company informs its employees about the control periods, we may also have early radiation.

The same idea can be applied to advertisement and price policies. If people know that some product can be bought later by discount, they will probably wait to make their shoppings. The incomes will be affected and can be modeled with early radiation. The sales will also have its impact to incomes as back radiation.

IKO can be also applied to the capacity expansion problem. Saniee [95] and Laguna [65] apply the *time-dependent knapsack problem* for multi-period capacity expansion. The time-dependent knapsack model is a variant of normal discounted knapsacks where the objective is to

$$\begin{aligned} \min \quad & \sum_{i=1}^{m}\sum_{j=1}^{n} p_j x_{ij} a^{i-1} \\ \text{subject to} \quad & \sum_{k=1}^{i}\sum_{j=1}^{n} w_j x_{kj} \geq b_i, \quad 1 \leq i \leq m, \\ & x_{kj} \in \mathbb{N}, \qquad\qquad j = 1,\ldots,n, \quad k = 1\ldots m, \end{aligned}$$

where a is a discount factor $(0 < a < 1)$. We can rewrite this as

$$\begin{aligned} \max \quad & \sum_{i=1}^{m}\sum_{j=1}^{n} (-p_j) x_{ij} a^{i-1} \\ \text{subject to} \quad & \sum_{k=1}^{i}\sum_{j=1}^{n} (-w_j) x_{kj} \leq -b_i, \quad 1 \leq i \leq m, \\ & x_{kj} \in \mathbb{N}, \qquad\qquad j = 1,\ldots,n, \quad k = 1\ldots m. \end{aligned} \tag{6.4}$$

This is our basic IK problem with only one constraint, provided that we allow the weights, profits and knapsacks to have negative capacity in (4.22) and (4.18).

Restriction (4.18) is equal to (6.4) in capacity expansion problem if we set $c_{ij} = m - i$. This means that after insertion every clone lasts to the end of the knapsack array (varying clone lengths). Figure 6.3 shows an example with five items, added at times (knapsacks) 1, 2, 3 and 4. At first knapsack we have added two items.

Demands in capacity expansion (values b_i) can be of any size, thus leading into a situation where we might have some capacity allocated but not needed

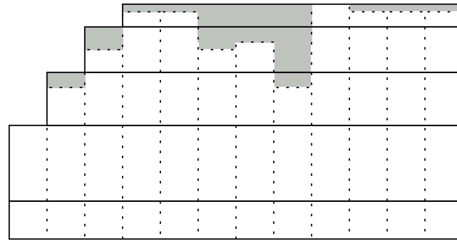

Figure 6.3: Capacity expansion.

(dark areas). If we allow variable clone lengths and use objective (4.22) and condition (4.18), we can optimize capacity expansion and also take into account the time intervals where we have extra capacity (knapsacks 6–8 for the topmost item and knapsack 8 for the other items).

Another possible application is dispatching of hard real-time tasks. In this problem we do not know the actual lengths of tasks until those tasks are run [48]. This introduces random variables c_{ij} to the IKO problem.

Moreover, IKO can be used to find optimal task lengths by using variable clone lengths. Now we can use back radiation naturally: we may know how task j of certain length has influence to the profits and resources following task j. For example, we can approximate the effects of a task (or control) of certain length to knapsacks following the task in load clipping used in energy management (see Chapter 8). Intuitively, load clipping is similar to capacity expansion: we do better, if we can fulfill the restriction with smaller set of items and shorter clones.

The questions about optimal clone lengths are near to discrete optimal control problems but techniques arosing in optimal control are not covered in this work.

Chapter 7

Multi-dimensional interactive knapsacks

The knapsack array can be generalized to knapsack space, which is done in Section 7.1. We present a few basic concepts and discuss, how to add restrictions similar to the ones we have used with IKHO and IKO.

After presenting the model, we give applications in Section 7.2. The motivation given at the start of Chapter 6 also applies here. In some cases our applications are somewhat controversial and speculative: there are models or approaches that work better with the given applications. However, we want to emphasize that features peculiar to IKs may give new insights.

Furthermore, we will show to transform a multi-dimensional IK into a relaxed IKO in Theorem 7.1. This means that the applications can be represented as IKO problems, which are maybe easier to handle than the applications (and multi-dimensional IKs) in more general form. For instance, IKO methods may work as starting point for development of heuristics.

We leave it open to find a problem that can be posed naturally with multi-dimensional knapsacks. If such a problem is found, we may ask, whether we can further extend the methods of Chapter 5.

7.1 0–1 MDIK is strongly NP-complete

Suppose that we have M ($\in \mathbb{N}$) 0–1 knapsack arrays that interact together just like the knapsacks interact together in 0–1 IK model. Again we suppose that knapsack arrays are ordered so that we can talk about left, right, earlier and later arrays when compared to some other knapsack array. Thus, we reuse the ideas of interactive knapsacks to knapsack arrays. Figure 7.1 clarifies this approach.

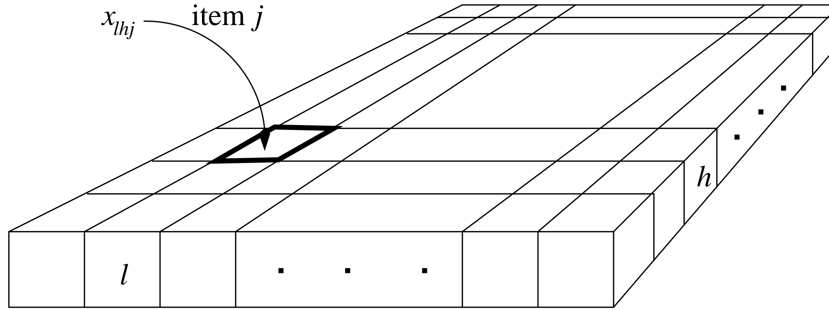

Figure 7.1: A sample IK array of order 2.

Now the object j can be inserted into an array l $(1 \leq l \leq m_1)$ where we can choose a knapsack h $(1 \leq h \leq m_2)$. Hence, the decision variable is x_{lhj}. Here we have two kind of natural interactions: those internal to l and interactions between different arrays, that is, between the neighbors of l.

Next we give examples of k-dimensional *knapsack spaces* and after that, we give a formal definition for k-dimensional knapsacks. The first dimension, lines in space, was discussed in Chapters 4–5. The sample multi-dimensional IK instance in Figure 7.1 is 2-dimensional (it is "in the plane"). A 0-dimensional IK instance is reserved for the traditional 0–1 knapsack problem and corresponds to a "point in the space".

To define the model precisely, we need q indices $i_1 \ldots, i_q$, where each $i_k \in \{1, \ldots, m_k\}$, for $k = 1, \ldots, q$. We denote $J_k = \{1, \ldots, m_k\}$, for $k = 1, \ldots, q$ and $J = J_1 \times \cdots \times J_q$.

We change the definitions of I_{ij} (an interval) and $I_{ij}(k)$ (a function from I_{ij} to $\mathbb{Q}$) a bit. Intuitively they will remain the same: I_{ij} will hold the knapsacks involved with clones and radiation and function operates similarly in those knapsacks. We suppose that $I_{ij} \subseteq J \times \{1, \ldots, n\}$. We also have $I_{ij}(k) : I_{ij} \times J \to \mathbb{Q}$, where $i \in J$ and $j = 1, \ldots, n$. Function $I_{ij}(k)$ gives the effect of item j put in knapsack i to knapsack $k \in J$. The sets involved with cloning and radiation are denoted by C_{ij} and R_{ij}, respectively, and hence, $I_{ij} = C_{ij} \cup R_{ij}$.

When we put item j into a knapsack, the number of knapsacks, where the copies are inserted in each dimension can be fixed for all items and knapsacks. It can depend on a knapsack, or it can be a variable. Further, the number can be equal for all or some dimensions. In applications it is comfortable to allow copies to be positioned in nonconvex way, for instance, to have "holes" in the clones.

We sort the knapsacks with the first dimension as the main key, the second dimension as the second key and so on. This is consistent with the ordering

introduced for the one dimensional arrays.

Adding item j to knapsack i in the finite knapsack space J causes some interactions. We suppose that the copies C_{ij} will be positioned so that i is the first knapsack of C_{ij} in the ordering of the knapsacks. The radiation spreads around the copies.

The radiation is bounded in an obvious way: when knapsack $k \in J$ is far enough from the clone positioned at i ($|i-k| > u$), we have $I_{ij}(k) = 0$, and $I_{ij}(k) \in \mathbb{Q}$, when $|i-k| \leq u$, for some $u \in \mathbb{N}$. Hence, the interactions of item j occur inside a q-dimensional ball of radius u centered at i in the knapsack space. Note that the clone formed by interactions is not necessarily a ball. Figure 7.2 contains a clone (dark grey squares, each describing a knapsack) having the "first" knapsack in black. The radiation is in light gray.

In the 0–1 multi-dimensional interactive knapsack optimization problem (MDIK) our objective is to

$$\max \quad \sum_{i \in J} \sum_{j=1}^{n} x_{ij} \sum_{k \in I_{ij}} I_{ij}(k) p_{kj} \tag{7.1}$$

$$\text{subject to} \quad \sum_{i \in J} \sum_{j=1}^{n} x_{ij} w_{ij} I_{ij}(l) \leq b_l, \quad l \in J, \tag{7.2}$$

$$x_{ij} = 0 \text{ or } 1, \qquad i \in J, \quad j = 1, \dots, n, \tag{7.3}$$

$$\sum_{i \in J} x_{ij} \leq 1, \qquad j = 1, \dots, n. \tag{7.4}$$

Now we can formulate the decision problem for 0–1 multi-dimensional interactive knapsacks as follows.

MDIK
Instance: A finite Cartesian set $J = J_1 \times \cdots \times J_k$, n items giving $n|J|$ integers for profits p_{ij} and weights w_{ij}, $|J|$ integer capacities for knapsacks b_i, for $i \in J$,

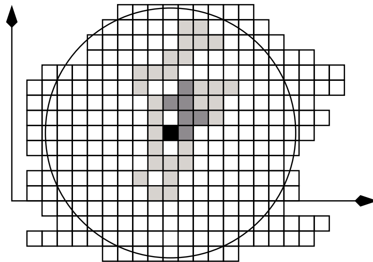

Figure 7.2: An example of a nonconvex clone and its surrounding ball.

and a positive integer P.
Question: Is there a distribution of $n|J|$ values x_{ij} giving profit P or more, such that the constraints (7.2)–(7.4) are satisfied?

We modify constraint (7.4) to

$$\sum_{i \in J} x_{ij} \leq K, \qquad \text{for each } j. \tag{7.5}$$

Further, we add an extra restriction

$$x_{kj} = 0, \text{ for } k \in C_{ij}, \text{ when } x_{ij} = 1. \tag{7.6}$$

Now an item can be put several times into the knapsack space but the representations may not overlap.

If we have more than one dimension, we may use other generalizations as well. For example, consider index sets

$$\sum_{i=(i',i'')} \sum_{j=1}^{n} x_{ij} \leq K' \qquad \text{for each } j \text{ and } i'' \in J/J', \tag{7.7}$$

where i is a combination of i' and i'' (denoted shortly by $i = (i', i'')$ in (7.7)) and index set $J' = J_{a_1} \times J_{a_2} \times \cdots \times J_{a_d}$, $1 \leq d \leq k$, and $a_s \neq a_t$, for each s and t. Equation (7.7) counts the clones of type j occurring in subspace J' and restricts the sum by K'. Note that $i = (i', i'')$ sums over i' and can be true for several copies in the same clone but we count each clone at most once. With K'_j we can handle each item separately.

Figure 7.3 has three clones and they are projected into space spanned by J' and one dimension of J/J', say "time". Suppose that $K' = 2$. We count the clone in the middle only once at moment t. (We could formulate this exactly by using existential quantifier in the sum.) In $t + 1$ the sum is three. Note that the items for clones (where x_{ij}'s equal one) do not have to map into this subspace.

Further, we can specify the number of clones allowed in the subspace consisting of d dimensions letting at the same time more clones to be positioned in other dimensions. By adding constraints of type (7.7) we obtain individual constraints to different knapsack subspaces. Restriction (7.6) can also be generalized to allow overlapping in some knapsack subspaces but not in all.

The rest of this section concerns the relationship between MDIK and IKO and the complexity of MDIK. In the following theorem, we can transform a q-dimensional space to an array.

Theorem 7.1. *A knapsack array can simulate a q-dimensional knapsack space ($q \in \mathbb{Z}^+$).*

Proof. Consider a q-dimensional knapsack space $J = J_1 \times \cdots \times J_q$, where each $J_k = \{1, \ldots, m_k\}$, $k = 1, \ldots, q$. We can establish one-to-one mapping f from a knapsack $i \in J$ to an 1-dimensional array of length $(|J_1| + 1)|J_2| \cdots |J_q|$. If $i = (i_1, \ldots, i_k)$, we have

$$f(i) = i_1 + (1 + m_1)(i_2 - 1) + \cdots + (1 + \sum_{j=1}^{q-1} m_j)(i_q - 1),$$

which can be constructed in polynomial time. Every $(1 + m_1)$th knapsack works as a separator having zero capacity. This prevents us to put clones into the space so that their shape will be interpreted wrongly. (We can add separators to other dimensions as well without affecting the polynomiality of the construction.) □

If we are to use the above transformation, interactions, cloning and radiation have to be applied in the simulations in a more versatile way than described in the earlier chapters. We have to allow nonconvex clones (clone may consist of different intervals). Similar transformation maps multi-dimensional integer programming to 0-1 MDKP, 0-1 IP, or to GAP. However, the restrictions need some extra attention.

Theorem 7.2. MDIK *is* NP-*complete in the strong sense.*

Proof. We show that function f defined in the proof of Theorem 7.1 is a pseudo-polynomial transformation from IKD to MDIK. Suppose that I is an instance of IKD. Functions Length$'[I]$ and Max$'[I]$ are for IKD and Length$[f(I)]$ and Max$[f(I)]$ for MDIK. Max$'[I]$ is polynomially related to Max$[f(I)]$ and Length$[I]$ is polynomially related to Length$[f(I)]$ by Theorem 7.1. Hence, one-to-one transformation between MDIK and IKD given in Theorem 7.1 is

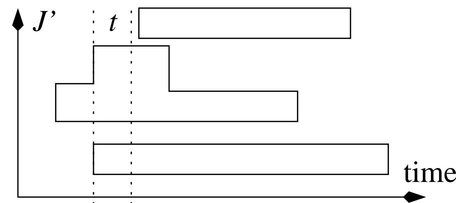

Figure 7.3: Restricting number of clones in subspaces.

pseudo-polynomial: by using separators we can ensure that all constraints are satisfied and hence, the 'yes' answers are preserved (property 1 of the definition of the NP-completeness in the strong sense, page 13), f can be computed in polynomial time for both Length$'[I]$ and Max$'[I]$ (property 2). There also exist polynomials q_1 and q_2 such that properties 3 and 4 are satisfied because of the polynomial relationships of Max's and Length's. □

7.2 Applications of 0–1 MDIK

Next we informally present applications for 0–1 multi-dimensional IK model and MDIK. The first application is related to assigning tasks and the next three to packing objects. At the end we discuss about two scheduling applications.

Miller and Franz [75] study the *multi-period assignment problem*, in which n employees are to be assigned to one of m tasks during T periods (see Appendix A.2). This corresponds to MDIK of order two with extra constraints and small modifications. We have $J = J_1 \times J_2$, where one dimension corresponds to m tasks (J_1) and the other to T time periods (J_2). The length of each clone is one knapsack and there is no radiation. A clone corresponds to an employee and it is of variable size and shape.

Miller and Franz allow more than one item of a type in the knapsack space, but exactly one in the task dimension. This means that one task is needed in each period for an employee. This is achieved with small modification (equality relation) of constraint (7.7). Thus, we have

$$\sum_{i' \in J_1} x_{ij} = 1, \qquad \text{for each } i'' \in J_2,$$

where $i = (i', i'')$, thus allowing many clones in the time axis. An employee will serve the required number of periods for all tasks by adding new constraints

$$|C_{ij}| = K_i, \qquad \text{for each } j.$$

The necessary manpower for each task during all time periods is ensured with (7.2) where all weights equal 1. The same holds for I in the copy area. To obey all the constraints of Miller and Franz we have to add some extra constraints similar to (7.2) differing only in the relation required: $=$ or $\geq$.

By using clones we can define some constraint more naturally than in the multi-period assignment problem given in [75]. For example, the constraints relating an employee to work continuously can be obtained by requiring that clones do not contain holes in dimension J_2 (time). The radiation has no

obvious interpretations for assignment problem. We may conclude that 2-dimensional MDIK is related to the multi-period assignment problem.

Milenkovic [74] examines the *translational containment* problem (see Appendix A.3). The objective is to determine a set of translations for k polygons that place them in a nonoverlapping configuration inside a polygonal knapsack. This problem, again, can be simulated to a given accuracy by a two-dimensional knapsack array. In MDIK presentation we need numerous knapsacks in order to discretize the vertices of a polygon to a required accuracy. This implies that the clones are also rather large.

We let the weight of each item and knapsack to be 1, including copies used in the clones. Radiation is not used. The profit is 1 for each item and knapsack but for copies the profit is zero. Each knapsack has a capacity of 1 inside the "knapsack area" and 0 elsewhere. In [74] each polygon has a set of translations. Each translation corresponds to a possible value of a variable clone. The "displacement" constraints have to be added separately for MDIK. We do not have to touch the MDIK objective in the case of kNN and kCN (see Appendix A.3) because if we can insert all k items into the knapsack we have maximized profits. To handle (r, k)NN and (r, k)CN problems, we need to modify our objective so that we search for all clone sets giving predefined profit k.

Another version of containment and packing problems is the *strip packing problem* [33]. In strip packing a list of rectangles (h_j, v_j), $1 \leq j \leq n$, with given height h_j and width v_j, where $0 \leq h_j, v_j \leq 1$. The objective is to place this list into a vertical strip of unit width and to minimize the total height needed. This is almost identical to the translational containment problem but translations are not allowed here. Therefore, we have fixed clones when using 2-dimensional MDIK. The number of the knapsacks in both dimensions depends on the heights and widths: the greatest common divisor determines the number of knapsacks needed. In order to do that, we have to fix the precision and scale of the heights and widths so that they all are integers. To minimize the total height we use similar approach for profits as in the scheduling problems (discounted profits).

A third kind of packing problem is to pack items in three dimensions considered for example in [77]. We have a rectangular box (which can be generalized to a knapsack of any shape) into which we put boxes (which can also be generalized to any shape).

We arrange the knapsacks in MDIK to have unit capacity; each item and each copy of the item in the clone consumes the whole capacity of the knapsack. Thus, two boxes cannot overlap.

If each knapsack has the amount of profit equal to one for each item, the profits are maximized if we can put maximum amount of items into the box (knapsack subspace) in question. We do not need radiation or variable sized clones (unless the items are made of rubber or are somehow flexible or changeable). Variable clones can be used, however, to put items in different positions into the box.

MDIK can also be applied to some multiprocessor scheduling problems, as we noted in Section 6.1. Here we use two dimensions, one for time and the other for processors: the profits are discounted over time. Suppose that there are K processors. Then cardinality of $K+1$ is used for processors, since we use one "dummy processor" to obtain the constraint that at most M processors can be selected at a time. Each task sent to a processor will radiate 1 to the dummy processor, which has capacity M. Other knapsacks modeling the processors have equal amounts and just enough capacity to take one task at a time. This is the multiprocessor scheduling case of [45] or [88] (see Appendix A.4). Figure 7.4 schedules a task for the second processor in the second moment. Its radiation is shown at the $(K+1)$th row. Two tasks on the first and Kth processor radiate twice as much in the fourth moment.

By adding one more dimension to the above setting, we obtain shop scheduling problems. In shop scheduling, each job j is divided into tasks. Each task has to go through a different processor and each job has to be done. Again, profits are positive, discounted over time, and hence we can ensure that all jobs will be done. Each task has positive profit only on a specific processor, but on the other processors the profit is zero or even negative, so that they would not be selected. This behavior can also be achieved with capacities.

The extra dimension is used as "constraint dimension": each task has a slot there. When we schedule one task of a job to a processor, we radiate a task to the constraint dimension into a slot reserved for this task. The constraint

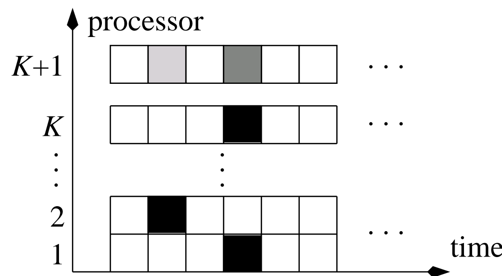

Figure 7.4: Example of multiprocessor scheduling. Dummy processors are on the top row. There is a task in the second moment and two in the fourth.

dimension consists of knapsacks with capacities equal to the task weight, one for each processor and for each time point. By filling appropriate time points for each processor in the constraint dimension, we can ensure that this task cannot be processed on any other processor at the same time, because it would entail overflows in the knapsacks.

Figure 7.5 clarifies this idea. On the left there is the space for multiprocessor scheduling without the “dummy processor”. We have put one item somewhere in the middle of it and the item consumes all the capacity the corresponding knapsacks have. This models a task to be performed on some specified processor taking certain amount of time. On the right there is only one slice of the corresponding “constraint dimension”. The gray area describes the radiation connected to the task, preventing this task to be done on some other processor at the same time.

Note that in this way we would allow many tasks to be processed at the same processor at the same time. This can be prevented by using similar constraint as on the right, not allowing the processing of all the other tasks at this particular processor at the same time. Graphically, it would be a vertical slice having gray upright rectangle on the left in Figure 7.5. If we want to deny a task to be processed twice or more times on the same processor, we just add gray vertical slice between dotted lines on the right in Figure 7.5.

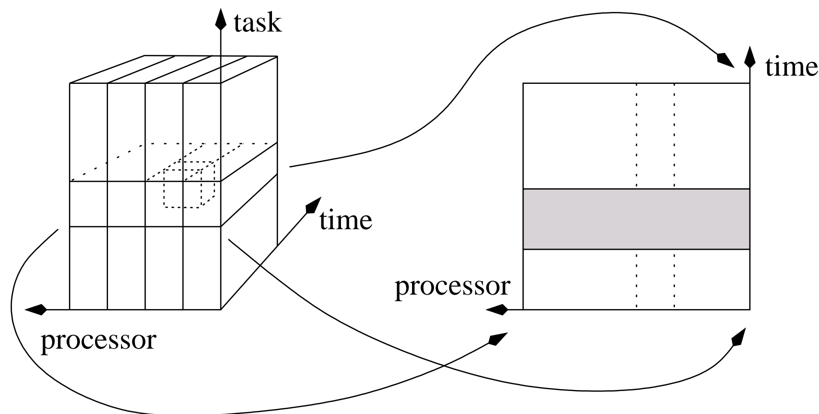

Figure 7.5: Example of constraint handling in shop scheduling.

Chapter 8

Application in load clipping

The chapter describes a real life application of the IK model. We present load clipping introduced in [3, 27, 29], which is a simplification of the model given in [6]. Section 8.1 introduces load clipping and in Sections 8.2 and 8.3 we set the model and its parameters. The next chapter contains methods for load clipping and the comparisons of methods can be found in Chapter 10.

8.1 Introduction to load clipping

Shortage of electricity may cause a supplier to use load clipping; the supplier may turn off the electricity from some of its customers or may start generators to meet the demand. The goal is to minimize the losses caused by buying (expensive) electricity from other suppliers to cover the demand after load clipping. Some customers have a consumption profile containing payback. A typical example of a control group with payback is a residential appliance like electricity heaters or air conditioners. A consumption peak appears after the control period when the devices go back to their normal state [6].

Load clipping is used in connection with energy management systems. Other related terms are demand side management (DSM), load management, direct load control (DLC), (peak) load control, load leveling, (peak) load shaping, valley filling, and interruptible load (control), to name few. See www-links given in Appendix C.1 for general descriptions about energy and demand side management. See also the references given in this section.

Demand side management, energy management, and load management refer to general areas containing several different methods and policies for somewhat similar objectives, which are connected to reducing the energy consumption. Some of the strategies are connected to the direct control of customers appliances also containing load clipping directly or indirectly.

Load clipping has been solved by using various algorithms. Typically, objectives, models and methods differ from paper to paper. Some of these methods might be applicable to IKHO and IKO problems presented in this work. To be able to perform load clipping tasks, we have to forecast the load consumption [107]. The optimization methods work on these expected values and most of them consider the forecasts as exact values.

Objectives include load reduction minimization [27, 29, 102], peak load minimization [64, 67], minimizing production costs [29, 102], and maximization of profits [81].

Two much used groups of algorithms for load clipping are enumeration methods and methods based on dynamic programming (DP) [27, 102]. These methods are optimal or nearly optimal but their major drawback is long execution time [3, 6, 29, 102] (see also Section 9.1). DP has also many other applications in electricity framework [28, 105], some of which are potential application areas of interactive knapsacks.

Fuzzy versions of DP have been developed for integrated load clipping and unit commitment [16, 58]. Unit commitment, however, contains some restrictions not included in our model and fuzziness increases the state space of DP. Caves and Herriges [20] use stochastic DP, where the future prices of electricity are unknown. We assume that prices (buying and selling rates) are given. The buying and selling rates are used to obtain the profits p_{ij} in IKO.

There are also methods based on linear programming (LP) [64, 81] and on hybrid models using LP and DP [67]. The plain LP is hard to use with our load clipping model, because we optimize the clone lengths, which makes the problem essentially nonlinear. Laurent *et al.*'s [67] approach differs from ours in the time-of-use rates (buying and selling rates) and in the constraints. However, our DP, unlike our heuristics, is not applicable to as large systems as Laurent *et al.*'s method.

Chen *et al.* [24] present a two-phase optimization technique, where the first phase identifies candidate schedules for items ("control patterns") and the second phase determines the optimal schedule for an item ("control plan for a group") from the candidates. Chen *et al.* report that their results are approximately as good as in Cohen *et al.* [29]. By using an extra variable, we can achieve better results [6]. Heuristics are considered in [6, 14].

Genetic and evolutionary algorithms are widely considered in many applications [76], including the multiple knapsack problem [60], project scheduling [25] and many problems related to electrical framework, for instance, unit commitment [34, 36] and unit maintenance scheduling [61, 66]. We present genetic algorithm for load clipping in Section 9.5. An introduction to genetic

algorithms can be found, for instance, in [76].

Unit commitment and maintenance scheduling differ a bit from load clipping but the ideas can be easily adopted to load clipping, as well as the ideas from [25, 60]. Kim *et al.* [61] develop simulated annealing and tabu search methods for unit maintenance scheduling.

Furthermore, load clipping is often combined with unit commitment and economic dispatch, and the applied methods include DP [15] (fuzzy DP [16, 58], stochastic DP [20]), binary network flow formulation [24], and evolutionary strategies [47]. Yan and Luh [106] consider unit commitment with "purchasing emergency power with very high prices", a similar motivation as we have. See also [105].

Our DP is somewhat similar to that of [27, 29], and our model has some key ideas common with [81]. We have earlier [6] developed the models and methods of [27, 29] by adding new properties to the models and new states to DP. DP given in [29] is not optimal, if applied to a group at a time and if the loads are evened out on hourly bases [6]. New state variables improve the results. Moreover, DP of [27] optimizes several groups at a time and therefore needs too large state space to be practical in our case.

Our solutions determine the number of controls needed, and a starting time with a duration and resting time for each control (we use 5 minutes precision). Number of controls can also be used as a restriction. Our solutions can be used as a successive optimization method.

Our objective is "in between" the minimizing of load reduction and the minimizing of peak load and is different from the objectives presented in other studies. Purchase transactions of electricity and own production give optimum level to be resold at each hour, while load over the optimum level has (very) high price. If demand is higher than our predefined level, we want to cut (expensive) "over loads" and at the same time minimize the losses caused by decreased sales. We also use purchasing and reselling prices (time-of-use rates) in the formulation of the objective.

Furthermore, our solution can use different objectives without major modifications of the method. The same holds for the prices, if one needs more complex price structures, and for the energy storages of [91, 94].

8.2 Basic model

In this section we settle the basic load clipping model. We provide terms from both interactive knapsacks and load clipping theory domains. (We do not specify item j as a subscript except in the last formulation of this optimization

problem at the end of this section.)

The set of items of type j and clones related to them is called a *control plan* and one clone is a *control* for *group* j corresponding to item. An *interval* $[a, b]$ is the set $a, a+1, \ldots, b$ $(a < b)$ of knapsacks. The length of an interval $[a, b]$ is $b - a + 1$. A *clipping situation* $\mathbf{s}$ is a vector $s_0, s_1, \ldots, s_m$ $(m > 0)$ of reals representing the difference between electricity demand and production in time interval $[0, m]$. We may think that values s_i tell how the knapsacks are already filled.

Domain $[0, m]$ is called the *optimization interval* and values s_i are called either *overload* or *underload*. Overload represents a situation, where demand is higher than combined production and electricity purchases $(s_i \geq 0)$, while underload represents a situation, where combined production and purchases of electricity is above the level of consumption $(s_i \leq 0)$. The knapsack i in $[0, m]$ is called a *time point*. Interval $[i, i]$ also corresponds to knapsack i and to time point i.

Vector $\mathbf{s}$ represents knapsack array, and an element of $\mathbf{s}$ represents the filling state of the knapsack. Overload means that the knapsack is not full enough and underload that the knapsack is too full. In electricity management the terms are used in an opposite way: overload means that an hour has too much consumption and underload means that the consumption level is not high enough. Vector $\mathbf{s}$ also determines the absolute goal level.

The size of knapsacks is determined by the electricity suppliers financial state. A wealthy supplier can buy extra electricity, or enlarge the knapsacks, in the case of overload. In the beginning the knapsacks contain some substance (or undetermined items) corresponding to some electricity consumption levels discretized into equally sized time slots (for example, five minutes).

Figure 8.1 contains an example of a clipping situation used in electricity management (a) and its representation with interactive knapsacks (b). The goal level is drawn as a horizontal zero line. Both figures show the same situation where we have one overload interval in time slots 3–5.

By saying that a knapsack is not full (or is too empty) we mean that filling state has not reached the goal level. Knapsacks 3–5 are not filled enough in (b), that is, the goal level is not reached. Further, the fifth knapsack is totally empty (in (a) we see that it has maximum amount of overload). On the other hand, the first two knapsacks in (b) are too full, or they have underload as can be seen from (a). The seventh knapsack in (b) is totally full, that is, there is no electricity consumption in (a) at the seventh time slot. The goal level giving the best result is achieved in the sixth knapsack in (b) or in the sixth time slot in (a). We say that the sixth knapsack is full or filled enough.

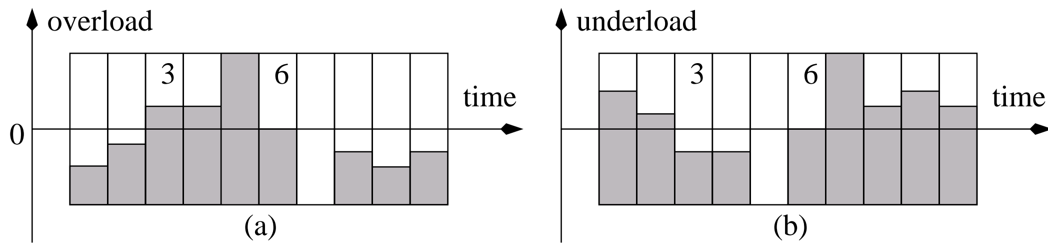

Figure 8.1: Clipping situation in electricity management (a) and its knapsack representation (b).

Note that a clipping situation and its representation are mirror images of each other determined by the goal level in Figure 8.1. One could avoid this confusing juxtaposition by using some other terms in the case of overload and underload, or by allowing items to have negative weights. However, the use of new terms would lead to nonstandard terminology (in the point of view of electricity management).

Another way to approach this problem is not to use the "undetermined" items (pre-filled knapsacks). First, nobody gets electricity, and then, we start to allocate electricity to customers until we obtain the goal level. This leads to larger instances, though.

Every time point i with overload has a positive real P_i called the *price factor* (buying price of electricity). If at time point i there is underload, the positive real R_i is the *revenue factor* (selling price of electricity). The *overload interval* is an interval $[a, b] \subseteq [0, N]$ such that at every time point $i \in [a, b]$ there is overload.

The clipping situation (knapsack array that is partially filled) is partitioned into knapsack sets, or *hours* $0 = a_1, a_2, \ldots, a_{n+1} = m$ of equal length, that is, $a_{i+1} - a_i = a_i - a_{i-1}$ $(2 \leq i \leq n)$. The length $a_{i+1} - a_i + 1$ of an hour is denoted by h. Hour i refers to the interval $[a_i, a_{i+1} - 1]$. Moreover, overloads (or underloads, referring to the filling situation), income factors and price factors do not change during an hour, that is, $s_j = s_{j+1}$, $P_j = P_{j+1}$ and $R_j = R_{j+1}$, where $j \in [a_i, a_{i+1} - 1]$. When a clone only partly overlaps an hour, we divide the effects caused to the overlapped knapsacks equally to all knapsacks in that particular hour. This is an example of the third interaction type mentioned earlier.

Table 8.1 shows sample quantities for overloads (the second line, referring to the filling situation of the knapsacks), price factors (the third line) and income factors (the fourth line). Intervals $[0, 4]$ and $[5, 9]$ present the first and

Table 8.1: An example about overloads, prices and incomes.

i	0	1	2	3	4	5	6	7	8	9
$\mathbf{s}$	-1.05	-1.05	-1.05	-1.05	-1.05	0.5	0.5	0.5	0.5	0.5
P	550	550	550	550	550	230	230	230	230	230
R	40	40	40	40	40	10	10	10	10	10

the second hours, respectively. In the first hour (knapsacks $[0,4]$) there is no overload (knapsacks are filled enough), but in the second hour there is an overload interval $[5,9]$ (knapsacks are too empty).

The total loss is

$$T(\mathbf{s}) = \sum_{i\in[0,m]} p_i(s_i), \tag{8.1}$$

where

$$p_i(s_i) = \begin{cases} -P_i s_i, & \text{if } s_i \geq 0, \\ R_i s_i, & \text{otherwise.} \end{cases} \tag{8.2}$$

Hence, we always have $p_i(s_i) \leq 0$. If there is underload (knapsacks are too full), we lose income and if there is overload (knapsacks are too empty), we have to pay some extra. Sum (8.1) counts the money lost, so its best possible value is 0. The case in Table 8.1 gives by (8.1) the total cost of -785 (the unit is not fixed).

Sum (8.1) corresponds to the profit given by the partially filled knapsack array (clipping situation). Note that (8.1) corresponds to an added value objective (2.12) which depends on the total capacity of groups (weight of items).

Consider now a situation, where we have a *group* (an item that can be put to a knapsack), by which we can lower the overload by making a *control* (fill knapsacks by inserting the item into a knapsack). A control of a group is made in an interval $C_{ij} = [i, i+c] = [a,b]$ (corresponds to a clone, $[a,b]$ is used for clarity in the following). The *controlling capacity* of a group, denoted by w, is the amount by which the group can lower the load in a time point (corresponds to the weight). The controlling capacity is the same for all time points for a group (item), but different groups have different capacities (weights).

The hours $[a_i, a, a_{i+1}, \ldots, a_{j-1}, b, a_j]$, where $a_i \leq a < a_{i+1}$ and $a_{j-1} < b \leq a_j$ (and $a < b$), have to be taken into account for control on $[a,b]$. The total influence of a control is called the *control amount* (space reserved for the item and its clone) and it is the product of the control capacity w and the control length $b - a + 1$ (hence, the weight is the same for all copies of an item in a clone). The control amount of an hour is denoted by W. We have $W = wh$;

this product is used to implement the third type of interaction mentioned. We call W as the *average clone weight* (as opposite to the weight of a clone, w). Function

$$
\begin{aligned}
g([a_i, a, b, a_j], \mathbf{s}) \quad = \quad & \sum_{k=a_i}^{a_{i+1}-1} p_k(s_k - W(a_{i+1} - 1 - a + 1)/h) + \\
& \sum_{k'=i+1}^{j-2} \sum_{k=a_{k'}}^{a_{k'+1}-1} p_k(s_k - W) + \\
& \sum_{k=a_{j-1}}^{a_j-1} p_k(s_k - W(a_j - 1 - b + 1)/h)
\end{aligned}
$$

shows the calculation of the profit change of a control $[a, b]$.

8.3 Payback, restrictions, and goal function

Function $u : \mathbb{N} \to \mathbb{N}$ maps the control length $b-a+1$ to the length of a payback (back radiation). This differs a bit from our former, static use of u. Function $r : \mathbb{N} \times \mathbb{N} \to \mathbb{R}$ describes the amount of the payback (radiation) of a control of length $b-a+1$ at time (knapsack) i. Moreover, we always have $r(b-a+1, i) \geq 0$, where $i \in [b+1, b+u(b-a+1)]$, and otherwise $r(b-a+1, i) = 0$. The paybacks are obtained from control capacities (radiation depends on weight): amounts $r(b - a + 1, i)$ are calculated beforehand for each item, or dynamically if there are many different control lengths $b - a + 1$. In practice, we have

$$\sum_{k \in [b+1, b+u(b-a+1)]} r(b - a + 1, k) \leq w(b - a + 1). \tag{8.3}$$

Constraint (8.3) simply says that a payback (radiation) does not exceed the control amount (capacities the clone consumes from knapsacks). We use the hours much in the same way with paybacks as with controls.

Next we show the impact of a control $[a, b]$ and its payback to clipping situation $\mathbf{s}$ as a function I (functions I' and I'' used in I are defined below). The hours to be affected are $[a_i, a, b, a_j, b + u(b - a + 1), a_l]$. By function

$$
I\big([a, b], \mathbf{s}\big)(k) = \begin{cases}
s_k, & \text{if } 0 \leq k < a_i, \quad \text{or} \quad a_l \leq k \leq m, \\
s_k + I'\big([a, b]\big)(k), & \text{if } a_i \leq k < a_{j-1}, \\
s_k + \big(I' + I''\big)\big([a, b]\big)(k), & \text{if } a_{j-1} \leq k < a_j, \\
s_k + I''\big([a, b]\big)(k), & \text{if } a_j \leq k < a_l
\end{cases} \tag{8.4}
$$

$(0 \leq k \leq m)$ we obtain the total influence (all interactions) of a control (clone) $[a, b]$ into the clipping situation (knapsack array and its filling situation). The first line leaves the unaffected hours as they are. The second line calculates the effects for hours where the control (clone) is. The third line is for both the control and payback calculation, and the fourth line calculates the effects of payback. By $I([a, b], \mathbf{s})$ we mean the new clipping situation obtained after control $[a, b]$.

Effects of a control (the clone part) are calculated with I':

$$I'\big([a,b]\big)(k) = \begin{cases} -W(a_{i+1} - 1 - a + 1)/h, & \text{if } a_i \leq k < a_{i+1}, \\ -W, & \text{if } a_{i+1} \leq k < a_{j-1}, \\ -W(a_j - b)/h, & \text{if } a_{j-1} \leq k < a_j. \end{cases} \tag{8.5}$$

The second line is for the hours between the starting and stopping hours, if any. The first and the third lines handle the hours where the control (clone) starts and stops. These hours may have partial control (in contrast to a full control lasting the whole hour). Controls decrease the overload (items and their clones fill the knapsacks). Paybacks (the radiation part) are calculated with I'':

$$I''\big([a,b]\big)(k) = \begin{cases} \displaystyle\sum_{k'=b+1}^{a_j-1} \frac{r(b-a+1,k')}{h}, & \text{if } a_{j-1} \leq k < a_j, \\ \displaystyle\sum_{k''=a_{k'}}^{a_{k'+1}-1} \frac{r(b-a+1,k'')}{h}, & \text{if } j \leq k' < l \text{ and } a_{k'} \leq k < a_{k'+1}, \\ \displaystyle\sum_{k'=b+1}^{a_l-1} \frac{r(b-a+1,k')}{h}, & \text{if } a_{l-1} \leq k < b + u(b-a+1). \end{cases} \tag{8.6}$$

The payback (radiation) starts in the first line, and in the third line we calculate the last moments having payback. (Both may involve partial hours.) The second line calculates the payback for hours, where each time point will get some payback (knapsacks get radiation). The payback increases the overload (radiation makes room into the knapsacks). Hence, we use negative back radiation.

It would simplify formulas (8.4)–(8.6) a bit if we were not to hourly even out the affects. Another alternative is to let the overloads and underloads vary within the hours and even out the loads when calculating the results. If the control starts and stops in the same hour, we cannot directly apply (8.4). In this situation we calculate the effects for the first hour with

$$s_k - W(b-a+1)/h + \sum_{k'=b+1}^{a_j-1} r(b-a+1,k')/h, \text{ when } a_{j-1} \leq k < a_j, \tag{8.7}$$

and the rest of the payback is calculated with the second line of I''. If the payback (back radiation) starts and stops in the same hour, we have to make a correction similar to (8.7).

Figure 1.2 on page 7 shows two examples of a control (clone). The left one describes a control (clone) in a realistic situation and the right one in a theoretical situation. The vertical lines denote hours. The dotted line is a clipping situation (knapsack array and its filling situation) without control (clone) and the plain line is a clipping situation with control (knapsack array with item and its clone). The left picture shows the advantage of a control (clone): if a group (item) has payback (radiation), we can "move" the overload to the next hour where the overload is cheaper (to fill knapsacks giving better profits).

The effects of all controls C (clones) of a group can be calculated recursively by the function

$$E(C, \mathbf{s}) = \begin{cases} E(C - [a,b], I([a,b], \mathbf{s})), & \text{where } [a,b] \in C \ (\neq \emptyset), \\ \mathbf{s}, & \text{if } C = \emptyset. \end{cases} \tag{8.8}$$

When all controls (clones) have been calculated, we can use (8.1) to find out the value of the new clipping situation (knapsack array and its filling situation).

Next we consider the constraints. First, the controls (clones) must be separate such that for all $[a,b], [a',b'] \in C$ we have

$$[a,b] \neq [a',b'] \Rightarrow [a,b] \cap [a',b'] = \emptyset. \tag{8.9}$$

Further, during the *resting time* it is not allowed to start a new control (insert a new clone to the knapsack array). Function $L : \mathbb{N} \to \mathbb{N}$ is increasing and it maps the length of a control (length of a clone) to the length of the corresponding resting time. So, for all $[a,b], [c,d] \in C$, we have

$$[a,b] \cap [a',b'] = \emptyset \Rightarrow [b, b + L(b-a+1)] \cap [a',b'] = \emptyset. \tag{8.10}$$

This with (8.9) is equivalent to (8.15). Here $[b, b + L(b-a+1)]$ denotes only the interval used for the resting time.

Note that a new control can be started even if the payback still occurs if the resting time does not overlap with the new control. Usually, the resting time is used to prevent a new control to start in the beginning of payback, when the need for extra electricity is the largest. If we start a new control at the end of payback, the change in the payback of new control is so small that it is not usually taken into account. We could also modify equation (8.4) to take into account the previous control (or controls) and its (their) possible

potency to the payback of the present control, when using too short resting time.

We call resting time as a *safety area* in interactive knapsacks. This is taken into account in (8.15). We also need the *minimum* and *maximum* control lengths (clone lengths) cmin and cmax (we are using variable length clones), respectively, and hence

$$\text{cmin} \leq c \leq \text{cmax}. \tag{8.11}$$

Note that $c+1$ is the length of a control $[i, i+c]$. Sometimes we also restrict the number of control times (number of clones) $\sum_{[a,b]\in C} 1$ by some positive integer K. This is achieved by constraint (8.17).

We can suppose that at every time point (knapsack) i prices P_i are larger than revenues R_i. By making controls (inserting clones) C_{ij} we can affect the clipping situation, and make a control plan C. Now, the optimization problem for several items can be given in the form

$$\max_{C_{ij}\in C} \quad \sum_{i=0}^{m}\sum_{j=1}^{n} x_{ij} p_{ij}\big(E(C,\mathbf{s})(i)\big) \tag{8.12}$$

$$\text{subject to} \quad \sum_{i=1}^{m}\sum_{j=1}^{n} x_{ij} w_{ij} I_{ij}([i, i+c_{ij}], \mathbf{s})(l) \leq b_l, \tag{8.13}$$

$$\text{cmin}_j \leq c_{ij} \leq \text{cmax}_j, \tag{8.14}$$

$$x_{kj} = 0, \text{ for } i < k \leq i + c_{ij} + L_j(c_{ij}), \text{ when } x_{ij} = 1, \tag{8.15}$$

$$x_{ij} = 1 \text{ when } c_{ij} > 0, \text{ otherwise } 0, \tag{8.16}$$

$$\sum_{i=1}^{m} x_{ij} \leq K_j, \tag{8.17}$$

where, for free j's and i's, we have $j = 1, \ldots, n$ and $i = 0, \ldots, m$. The sum $\sum_{i=0}^{m}\sum_{j=1}^{n} x_{ij} p_{ij}\big(E(C,\mathbf{s})(i)\big)$ indicates the income lost and its theoretic maximum is 0. Note that our decision variables are c_{ij}'s from which we obtain clones (controls) C_{ij} and integers x_{ij} by the identities $C_{ij} = [i, i+c_{ij}]$ and $x_{ij} = 1$, if $c_{ij} > 0$ (we may suppose that cmin > 1), and the set of clones C.

In the following chapters we mostly talk about clones and back radiation without the application specific terms. In some places, the application specific terms, however, are more convenient to use.

Chapter 9

Methods for load clipping

We have implemented several methods for load clipping [1, 3, 6]. Sections 9.1, 9.2, and 9.3 present methods for the heuristic approach, where we make decisions for one item (group) at a time. This corresponds to IKHO. Section 9.1 implements a method based on integer composition and Section 9.2 derives DP solution.

The DP approach is further analyzed in Section 9.3, where we build up a hierarchy of solutions so that it is possible to trade between speed and accuracy. Moreover, we show how the number of wait states needed can be diminished to be about half of the number used in Section 9.2. This speeds up the whole optimization process approximately by the same factor. State space can also be decreased with multi-pass DP of [102], but then one should be able to relax some constraints. We also show how to add a fourth state variable into DP.

In Sections 9.4 and 9.5 we present methods that try to position several items to a knapsack array at a time. Section 9.4 considers greedy heuristics and Section 9.5 genetic algorithms. Load clipping with several control groups corresponds to IKO.

Extensive testing is reported in Chapter 10, supporting the hypothesis that we need wait and the mentioned fourth state in order to get good results with DP.

9.1 Integer composition

In this section we present a heuristic solution to the load clipping problem by positioning and deciding the length of each clone separately for an item at a time. This approach corresponds to IKHO. Thus, we do not use the item index in this section. First we give a general description of the method, and then show some results.

The composition algorithm has three input parameters: the length of the whole time interval m, number K of clones we want to compose into a plan, and the set of allowed lengths (feasible control lengths) $F = \{c_0, \ldots, c_q\}$ of clones ($q \in \mathbb{N}$). This algorithm is based on a composition algorithm given in [93]. Few minor modifications, however, are needed. For convenience, we summarize the symbols to be used in this section in Table 9.1. Some of the symbols are used in other meaning than before.

The idea is to compose m into $2K+1$ integers stored in vector I and hence, to interpret each composition as a set of clones containing exactly K separate clones. We achieve this by interpreting $2K+1$ integers in I in the following way: each even position has its integer from the set of the allowed lengths F and each odd position has free nonnegative choice for its value as long as the sum of all integers is m. Vector I can be interpreted as a k-digit number with some restrictions on digits. The ordering of compositions is based on this interpretation.

We obtain the set of clones (set of control plans) in the following way from I. The first clone starts at I_1 and is of length I_2, the second clone starts at $I_1 + I_2 + I_3$ and is of length I_4, and so on. Here we have included the safety area in the length which have to be excluded from I_2 and I_4 to get the right clone length. By inserting the safety area also to vector F we avoid the need to check explicitly the safety area.

The conversion of vector I into a set of clones is shown in Figure 9.1. This is line (12) in Figure 9.5. The algorithm in Figure 9.1 calculates the result of the plan contained in I (with equation (8.12)) and stores the best plan and its value. By L^{-1} we mean the only choice for the safety area when inserting a clone of a certain length. Note that L^{-1} can be calculated beforehand into an array, which we traverse in linear time on the number of feasible control lengths q. Note that $K = (k-1)/2$.

Lemma 9.1. *The objective is calculated in $O(K)$ time for a set of clones stored in I.*

Table 9.1: Symbols used in this section.

Element vector	$e = [e_0, e_1, \ldots, e_k]$	Index of the last element in I	k
Integer composition	$I = [I_0, I_1, \ldots, I_k]$	Set of feasible control lengths	F
Time interval	m	Number of feasible control lengths	q
		Number of controls	K

```
    Inputs: I and k
(1) r = 0, C = ∅
(2) for i = 1 to (k − 1)/2 do
(3)         Form a new clone C_i = [r + I_{2i−1}, r + I_{2i−1} + I_{2i} − L^{−1}(I_{2i})]
(4)         r = r + I_{2i−1} + I_{2i}
(5) od
(6) Count the objective for C = {C_1, ..., C_{(k−1)/2}}
(7) Store C and the result, if it is better than any earlier result
```

Figure 9.1: Converting a composition of an integer into a clone set (plan) (line (3) of the algorithm in Figure 9.5).

As an example, let $F = \{4, 6, 8\}$, $k = 2$, and $m = 12$. A composition [0 0 6 0 4 2] sums up to 12 and corresponds to a clone, whose first moment starts at 6, and whose length with safety area is 4. There are two moments after the clones safety area till the end of the optimization interval. (We also show the value of the position with index zero.)

Next we consider the initialization of the integer composition procedure given in Figure 9.2. Vector I has $2K + 2$ different elements of which $2K + 1$ are used as a storage for a composition. We set index $k = 2K + 1$ in the first line. The value of I_0 is used as a halting criterion (in line (2) in Figure 9.5). Array e contains $2K + 1$ elements and each element corresponds to an index for the set F; we need only half of the elements of e. In the initialization phase only even indices of I take values from the set $F = \{c_0, \ldots, c_q\}$.

Because we are constructing the compositions in lexicographic order, we

```
     Inputs: m, K and F = {c_0, ..., c_q}
 (1) a = 0, k = 2K + 1, I_0 = 0
 (2) for i = 1 to k − 1 do
 (3)         if i is even then
 (4)                 I_i = c_0, a = a + c_0
 (5)         else
 (6)                 I_i = 0
 (7)         end
 (8)         e_i = 0
 (9) od
(10) I_k = m − a, e_k = 0
```

Figure 9.2: Initialization of the integer composition (line (1) in Figure 9.5).

initialize the even indices with c_0 and I_0 with 0. The last value I_k is initialized with $m - 2Kc_0$ so that the sum over I is m, that is,

$$I = [0, \underbrace{0, c_0, 0, c_0, \ldots, 0, c_0, 0, c_0,}_{k-1 \text{ positions}} m - 2Kc_0],$$

since $a = \sum_{i=1}^{k-1} c_0 = (k-1)c_0 = kc_0 - c_0$ and $2K = k - 1$. We cannot decrease any of the values at positions $I_1, \ldots, I_{k-1}$, because they are already minimal. Value $m - 2Kc_0$ can be decreased, but because the sum of the values has to be m, we should increase some value at $I_1, \ldots, I_{k-1}$.

Each e_i is 0 because in each even position we have value c_0. Value a represents the time without controls (that is, the number of knapsacks without a clone). Hence, we have shown the following lemma.

Lemma 9.2. *Initialization stops in $O(K)$ time and I contains the lexicographically minimal number representing the integer composition of m.*

We continue the example started after Lemma 9.1. The first composition (lowest 6-digit number) is [0 0 4 0 4 4]. It corresponds to two clones of length 4 (with safety area). The safety area contains the only moments between the clones. The first two 4's cannot be decreased: they are minimal in F. At the end, there are four "un-used" moments.

We need to form the lexicographical minimum for the elements at $I_i, \ldots, I_k$ in Figure 9.5. Figure 9.3 shows how to initialize the minimum composition in lexicographic order. We use values $I_i, \ldots, I_k$.

The algorithm consists of a loop gathering the values to a temporary variable t and initializing everything by the way to the lowest feasible value similarly as in the initialization procedure. Vector e is also initialized to point to

---

Inputs: i is the starting element and k is the last index of I and e.
(1) $t = 0$
(2) **while** $i \leq k$ **do**
(3)   **if** i is odd **then**
(4) $t = t + I_i$, $I_i = 0$
(5)   **else**
(6) $t = t + I_i - c_0$, $e_i = 0$, $I_i = c_0$
(7)   **end**
(8) $i = i + 1$
(9) **od**
(10) $I_k = t$

Figure 9.3: Forming the minimum (line (12) in Figure 9.5).

the corresponding indices. The last position is not restricted by F, and hence, we set I_k to be t. This routine takes time $O(K)$, since $k = O(K)$.

Lemma 9.3. *The lexicographical minimum is formed in $O(K)$ time to $I_i, \ldots, I_k$ of I.*

Let us assume that we have a nonfeasible composition [0 2 5 1 4 0] and that we are to construct the minimum starting at index 2. Thus, 5 is restricted to F, and hence, it becomes 4, in the third index 1 is changed to 0, and hence we put 2 to the last index 5. The composition is now [0 2 4 0 4 2].

The algorithm in Figure 9.4 starts from the right end of I and moves index i to the left as long as I_i is 0 at odd indices and c_0 at even indices, that is, when these values are the lowest possible. After that the index points to the first position from right end which is neither zero nor c_0. If $i-1$ is odd, we are done in line (5), because we can always add to positions not restricted by F. If we did not return in line (5), we continue to line (6), and we know that $i-1$ is even and i is odd.

The first condition in line (6) ensures that we can add to I_{i-1} by checking that it is not the maximal item c_q of F. The second condition ensures that the value of I_i is big enough, because we have to add to I_{i-1} the difference between successive items in F. Hence, line (6) ensures that we can add to position $i-1$ by checking that there is room and that we can decrease position i by the required amount.

However, if we cannot add to position $i-1$ and the one of the conditions in line (6) does not hold, we know that any amount can be added to position $i-2$, because that position is odd. In this case the index to be returned is $i-1$. It is possible that decreasing I_{i-1} leads to infeasible solution. But in the main algorithm in Figure 9.5, $i-2$ is odd, and we go to line (6), after which to line (12). Line (12) in Figure 9.5 organizes $I_{i-1}, \ldots, I_k$ to a feasible partial

(1) $i = k$
(2) **while** (i is odd **and** $I_i = 0$) **or** (i is even **and** $I_i = c_0$) **do**
(3) $\quad\quad i = i - 1$
(4) **od**
(5) **if** $i-1$ is odd **then return** i **end**
(6) **if** $e_{i-1} < q$ **and** $I_i \geq c_{e_{i-1}+1} - c_{e_{i-1}}$ **then return** i **end**
(7) **return** $i-1$

Figure 9.4: Returning an index to the rightmost position, where the next position to the left can be increased (line (4) of the algorithm in Figure 9.5).

solution. Recall that it was possible to decrease I_i at least by one, which is the amount we increase I_{i-2}.

In Figure 9.4, lines (2)–(3) take time $O(K)$ and the rest takes time $O(1)$. Hence, line (13) of the Algorithm in Figure 9.5 can be performed in $O(K)$ time. Hence, we have the following lemma.

Lemma 9.4. *Position that can be increased is found in $O(K)$ time.*

We used a nonfeasible composition [0 2 5 1 4 0]. The composition leading to it is [0 1 6 1 4 0]. In the algorithm of Figure 9.4, after the while-loop, we are in line (5) and $i = 3$ is odd. The condition in the line does not hold. Thus, we continue to line (6). But 1 is smaller than $8 - 6$, and the second condition does not hold in line (6). Hence, we return $i - 1 = 2$. This means that we increase the first 1 and decrease 6 by one in line (6) in Figure 9.5. The next thing after line (6) is to construct the minimum in line (12). Therefore, after [0 1 6 1 4 0], we have [0 2 4 0 4 2]. Note that there is no composition between these two compositions in lexicographic order.

Note that when handling the last composition, the algorithm in Figure 9.4 will return $i = 1$ in the line (6), because the only element not containing the minimal value including I_0 is I_1, and we can always add to even index 0 and decrease the maximal element at I_1. In all other cases the index to be returned is larger than one. The last composition in our example [0 4 4 0 4 0].

We will prove next that the algorithm in Figure 9.5 works correctly and produces all compositions in lexicographical order. In line (3) we start with the initialized situation. The last composition in lexicographic order is

$$I = [0, m - 2Kc_0, c_0, 0, c_0, \ldots, 0, c_0, 0, c_0, 0].$$

This can be shown by a contradiction: we cannot decrease any of the positions $I_2, \ldots, I_k$ in order to increase the value at I_1, and when the first position I_0 is changed, we stop. In the example, we cannot decrease neither of the last two fours in [0 4 4 0 4 0], because they belong to F.

The value of the current composition is calculated in line (3) in $O(K)$ time by Lemma 9.1. After finding the correct index (line (4)), we decide how much to increase I_{i-1} in lines (5)–(11). If $i - 1$ is odd, we increase I_{i-1} by one in line (5). This can be done, because at odd indices there is no restrictions on values, as long as they are nonnegative and everything sums up to m. Note that updated I_i may be nonfeasible. After line (5), we form the lexicographical minimum to $I_i, \ldots, I_k$ in line (12). After that, I_i will have some feasible value obtained from F so that $I_i, \ldots, I_k$ is minimal.

On the other hand, if $i-1$ is even, we set a temporary variable a to equal $c_{e_{i-1}+1} - c_{e_{i-1}}$ in line (8). This amount is added to I_{i-1} and removed from I_i. The value at I_{i-1} is feasible, because index i is chosen in line (4) so that I_{i-1} can be increased by decreasing I_i. Thus, we update e_{i-1} to point to the correct value in the set of feasible lengths F. Note that I_i can be decreased by a, because i is odd, and line (4) has already checked that $I_i \geq a$.

If the current composition is the last one, we must have $i = 1$ after line (4), since only I_1 can be decreased (and I_0 increased). The position 0 is even, and hence, we increase I_0 and decrease I_1. Then after line (12), we exit the loop by condition at line (2), and the algorithm halts.

The change I_i to I_{i-1} is the smallest possible: it is either 1, or $c_{e_{i-1}+1} - c_{e_{i-1}}$, which is the smallest feasible increment of I_{i-1}, when $i-1$ is even. Note that after line (4), none of $I_{i+1}, \ldots, I_k$ cannot be decreased so that the next index to the left could be increased, if i is odd. If i is even, then none of $I_{i+2}, \ldots, I_k$ cannot be decreased, while for I_{i+1} it may be possible. If it were possible to increase I_i by the same amount I_{i+1} can be decreased, we would have returned index pointing to I_{i+1} and not I_i. Hence, there cannot be any number between the previous composition and the composition obtained in line (12).

What if there are several values $I_i, \ldots, I_k$ such that we can decrease many of them a little and then increase I_{j-1} by a (in the case $i-1$ is even)? Let the rightmost index be j such that I_j can be decreased. By the discussion preceding Lemma 9.4, we know that either $j-2$ or $j-1$ can be increased and

```
     Inputs: m, K and F = {c_0, ..., c_q}
 (1) Initialize variables with the method given in Figure 9.2
 (2) while I_0 = 0 do
 (3)        Count the objective from I
 (4)        Get the index i − 1 of I to be increased
 (5)        if i − 1 is odd then
 (6)               I_{i−1} = I_{i−1} + 1 and I_i = I_i − 1
 (7)        else
 (8)               a = c_{e_{i−1}+1} − c_{e_{i−1}}
 (9)               I_{i−1} = I_{i−1} + a and I_i = I_i − a
(10)               e_{i−1} = e_{i−1} + 1
(11)        end
(12)        Construct the lexicographical minimum to I_i, ..., I_k.
(13) od
```

Figure 9.5: The integer composition.

line (12) will put the minimum feasible value to the right end of I.

Therefore, at each round in lines (3)–(12), we increase "the k-digit number" and eventually arrive to the last composition after we have performed enough rounds. Thus, the algorithm starts from the first composition, produces all compositions in lexicographic order and eventually halts. We have shown that the algorithm given in Figure 9.5 constructs each composition in lexicographic order meaning that our integer composition works correctly.

Theorem 9.5. *The algorithm in Figure 9.5 constructs each integer composition in linear time.*

Proof. Initialization takes $O(K)$ time by Lemma 9.2, as well as line (3) by Lemma 9.1, line (4) by Lemma 9.4, and line (12) by Lemma 9.3. Lines (5)–(11) take $O(1)$ time. Hence, the time consumed between each composition is linear on K. □

Theorem 9.5 shows that enumerative solution is a pseudo-polynomial time algorithm for IKHO, if K is fixed. (See the discussion after Corollary 4.6.)

Table 9.2 lists 30 compositions produced, when $F = \{4, 6, 8\}$, $k = 2$, and $m = 12$. Note that between any two consecutive compositions, there cannot be any other composition.

Table 9.2: Integer compositions, when $F = \{4, 6, 8\}$, $k = 2$, and $m = 12$.

1 : [0 0 4 0 4 4]	7 : [0 0 4 2 6 0]	13 : [0 0 6 2 4 0]	19 : [0 1 4 2 4 1]	25 : [0 2 4 1 4 1]
2 : [0 0 4 0 6 2]	8 : [0 0 4 3 4 1]	14 : [0 0 8 0 4 0]	20 : [0 1 4 3 4 0]	26 : [0 2 4 2 4 0]
3 : [0 0 4 0 8 0]	9 : [0 0 4 4 4 0]	15 : [0 1 4 0 4 3]	21 : [0 1 6 0 4 1]	27 : [0 2 6 0 4 0]
4 : [0 0 4 1 4 3]	10 : [0 0 6 0 4 2]	16 : [0 1 4 0 6 1]	22 : [0 1 6 1 4 0]	28 : [0 3 4 0 4 1]
5 : [0 0 4 1 6 1]	11 : [0 0 6 0 6 0]	17 : [0 1 4 1 4 2]	23 : [0 2 4 0 4 2]	29 : [0 3 4 1 4 0]
6 : [0 0 4 2 4 2]	12 : [0 0 6 1 4 1]	18 : [0 1 4 1 6 0]	24 : [0 2 4 0 6 0]	30 : [0 4 4 0 4 0]

9.2 Dynamic programming

The problem posed by equations (8.12)–(8.17) can be solved with dynamic programming (DP) [3, 6, 29, 105]. In [6] we show that an extra state variable is needed for the solution of [29], and in [3] we deepen the results thus obtaining faster DP (or more accurate, if wished). We follow here mainly [3] and give its main results. We consider similar approach as in the previous section: we optimize only one item (or group) at a time.

One way to apply DP is to use very large state space to find optimal set of clones (a control plan) C, for example

$$D_1(C, \mathbf{s}) = \max_{[a,b]} D_1(C \cup [a, b], E(C \cup [a, b], \mathbf{s})), \tag{9.1}$$

where the program is started with $D_1(\emptyset, \mathbf{s})$. Note that $E(\emptyset, \mathbf{s}) = \mathbf{s}$ by (8.8). In (9.1) the different clones form the state space and the number of control times (inclusions of an item) form the stages. This solution enumerates different clones, and the state space is far too large. Moreover, this solution is sub-optimal (as well as the method to be presented). The dynamic programming formulation (9.1) at stage n is equal to the optimization problem

$$D_2(C, \mathbf{s}) = \max_{[a,b]_1} \ldots \max_{[a,b]_n} D_2(C', E([a, b]_1 \cup \cdots \cup [a, b]_n, \mathbf{s})), \tag{9.2}$$

where $C' = C \cup [a, b]_1 \cup \cdots \cup [a, b]_n$. This solution can always prune a part of the clone sets with n different clones (index n at $[a, b]_n$ means the nth clone of the same type as opposite to a clone of item n). At the first stage we have to check approximately (cmax$-$cmin)m states, where cmax is the maximum and cmin is the minimum length of a clone and m is the length of the knapsack array (the number of time points in the optimization interval). At the second stage, when forming the second clone, we have to find a connection to all (cmax$-$cmin)m states. The connections can point to approximately (cmax$-$cmin)m states, so we should check about (cmax$-$cmin)$^2 m^2$ states, which is too much to be repeated $O(n)$ times.

We could try to use only the best states from the previous stage. If we check only 20 states, we need approximately 20(cmax–cmin)m checkings for each clone after the first clone. In practice 20 best states is not enough, if we want to get quickly good economic results with equation (9.2). For example, the fifth clone would need approximately $4 \cdot 20$(cmax–cmin)m state checkings plus the initial calculations for (cmax–cmin)m states. Our DP uses less than $5 \cdot 11$(cmax–cmin)m states and we can half this state space by Theorem 9.8 in the next section.

Note that if we have first found optimal five controls giving clipping situation $\mathbf{s}'$ and then find other five controls being optimal against $\mathbf{s}'$, the global optimum is not necessarely found. We may achieve better results by using only nine controls having no common control with the previous two sub plans. Principle of optimality (see [13]) is not fulfilled. The reason is the averaging used in I in (8.4): a control may affect the prices and revenues of another control in the same hour. We demonstrate this at the end of this section.

Cohen and Wang [29] use two state variables, the number of clones and the clone length. Our tests [3, 6] indicate that two state variable systems are so

fast that we can add at least one state variable (see [13, pp. 30–34]) to improve the results. (See the end of this section.)

We use the state variables *number of clones* k, *wait states* d and *clone length* c. As a result, we have a slower but more accurate system than those with two state variables [3]. The wait states are used to delay the start of a control while k and c have obvious meaning. There are many realistic situations where the extra variable d gives more accurate results with economic significance, although, our three variable system (k, d, c) is not optimal (recall the NP-completeness results in Chapter 4 and see the example at the end of this section).

In the state space we need the clone length, so that the DP can form the optimal clone length and at the same time consider the restriction (8.11) (minimum and maximum clone lengths). State variable c contains the control's length obeying conditions (8.9)–(8.11), and the safety area of the control. Without the number of clones k, DP will find only one clone. With these state variables we define one state of stage $i \in [0, m]$ as a triple

$$(k, d, c)(i). \tag{9.3}$$

A system in state (k, d, c) is defined to be the kth clone (control) of length c, of which start is delayed d time points. Our tests also demonstrate the fact that the three variable (k, d, c) solution does not give the optimal solution in every situation, especially if there is payback (see Chapter 10).

The phrase "stage i" refers to a knapsack (or time point). In practice we have to determine an upper bound for the number of wait states d by Theorem 9.8, when the item does not have radiation. If the item does have radiation, we assume that d can have $h-1$ (h is the length of an hour) different values (we also test other amounts, see Chapter 10).

We denote $S = (k, d, c)$ and $S' = (k', d', c')$. The variables with primes are "new" ones and the variables without primes "old" ones, when forming the connections from "new" stage $i+1$ to "old" stage i. Function

$$D'(\mathbf{s}, S', S, i+1) = \begin{cases} 0, & \text{when (9.7)–(9.10)}, \\ -P, & \text{when (9.11)}, \\ T(E([i-c, i], \mathbf{s})) - T(\mathbf{s}), & \text{when (9.12)}, \\ -\infty, & \text{otherwise}, \end{cases} \tag{9.4}$$

$0 \leq i \leq m$, gives the change in the value, when moving from state (k, d, c) of stage i into state (k', d', c') of stage $i+1$. The first line is used, when the value does not change. The second line is needed, when we make a decision about

the best clone. The third line is used, when we start to place a new clone. In these situations we add to the value cost P of making a clone. The last line is used with every other values of S and S'. They are impossible since they do not have any reasonable real world interpretation.

The dynamic forward recursion equation is

$$D(\mathbf{s}, S', i+1) = \max_{S} D(\mathbf{s}, S, i) + D'(\mathbf{s}, S', S, i+1) \tag{9.5}$$

and

$$D(\mathbf{s}, (k,d,c), 0) = \begin{cases} T(\mathbf{s}), & \text{when } k = d = c = 0, \\ -\infty, & \text{otherwise.} \end{cases} \tag{9.6}$$

When

$$\begin{aligned} &k' = k+1, \qquad d' = c' = 0, \qquad 0 \le d \le t-2, \qquad \text{and} \\ &\text{cmax} + L'(k, d, \text{cmax}+1) - 1 \le c \le \text{cmax} + L(\text{cmax}), \end{aligned} \tag{9.7}$$

we interpret that the clone at stage i and in state (k, d, c) is constructed, which in turn increments the number of clones ($k' = k + 1$). The next clone starts in the state $(k', 0, 0)$ at the stage $i + 1$. The function $L'(k, d, \text{cmax} + 1)$ gives the minimum safety area $c - (\text{cmax} + 1)$ stages ago for state (k, d, cmax) (see also condition (9.12)). We do not use the bottom line of (9.7) nor (9.12) in the cases of clone lengths with safety area one, but we rather directly use the last condition of (9.12). In (9.7) we restrict the number of wait states by $t-2$. Recall that a wait state can get $t - 1$ different values.

Moreover, it is possible that an "old" optimal set of clones at stage i does not change (be better) when we move to stage $i + 1$, and so

$$k' = k, \qquad d' = c' = 0, \qquad \text{and} \qquad d = c = 0. \tag{9.8}$$

This is the only case with conditions (9.7) and (9.12), when DP can make choices about the optimal path. If two paths give the same result, we choose in (9.5) the one with a later clone. This does not have any impact on the result, but in practice we usually want to position the clones as late as possible. Figure 9.6 shows the state structure. Conditions (9.7) and (9.8) are shown on the left. There we have several states, from which we choose the maximum.

When

$$k' = k, \qquad d' = d+1, \qquad \text{and} \qquad c' = c = 0, \tag{9.9}$$

we "move some information from the past" to new stage $i + 1$. With this information we can check what result can be achieved, if we choose the best path d stages ago instead of some other path with the last clone started in

the interval $[i-d,i]$. Figure 9.6 shows this in the middle, where is one control time. The first positions of clones are connected. When

$$k'=k,\quad d'=d,\quad c'=c+1,\quad \text{and}\quad (c'\neq 1, c'\neq \text{cmax}+1), \tag{9.10}$$

we increase the clone length (the clone started c knapsacks ago). When

$$k'=k,\qquad d'=d,\qquad \text{and}\qquad 1=c'=c+1, \tag{9.11}$$

we have started a new clone. In this situation we add to the result the positioning cost of clone P. When

$$k'=k,\quad d'=d,\quad c'=\text{cmax}+1,\quad \text{and}\quad \text{cmin}\leq c\leq \text{cmax}, \tag{9.12}$$

we can calculate the impact of a correct clone on the knapsack array. Clone length c also determines the safety area $L'(k,d,\text{cmax}+1)=L(c)$ $(0<L'\leq L(\text{cmax}))$, which is stored as long as it has its effect to the control. Figure 9.6 shows conditions (9.10)–(9.12) in the right. Condition (9.8) gives the first and the last arrow (the horizontal ones). The next arrow means a new control and this is connected to (9.11). The next three arrows are given by (9.10). In the first resting time position we can select the maximum with (9.12). Condition (9.10) gives the next arrow. The two arrows pointing across control times are connected to (9.7): the left side of Figure 9.6 does not show that there can be several arrows to a control time.

For each state (k,d,c) and for each stage $k>0$, we save the connection referring to some state at the previous stage. The connections form a *path*. When we have the values

$$D(\mathbf{s},(k,d,c),m)$$

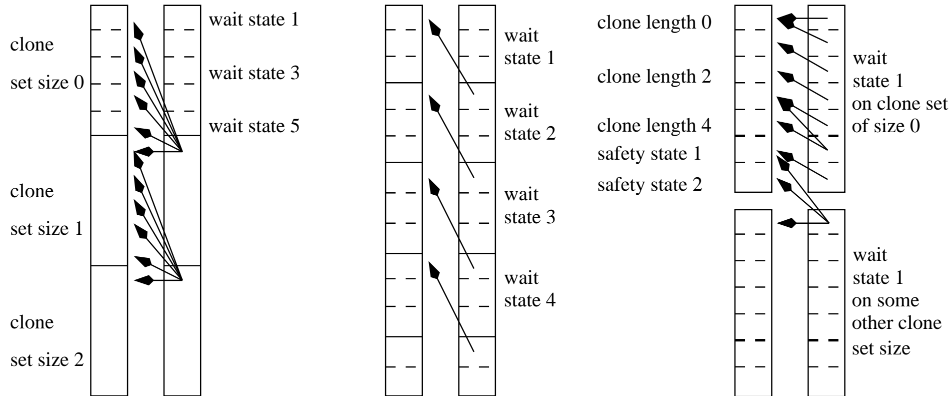

Figure 9.6: The structure of state space.

with appropriate values in k, d and c, we can construct the set of clones C by traversing the path formed by the connections. The path is optimal with respect to the state space used (but not with respect to the problem). Note that functions D and D' in equations (9.4)–(9.6) obey conditions (8.9)–(8.11).

Till the end this section, we consider the necessity of wait states d. At the same time, this works as an example of the DP solution. First, we omit all references to wait states from DP (9.5)–(9.6). This means that condition (9.9) is not used at all.

Let $[1, m = 99]$ be the optimization interval and let a_i $(1 \leq i \leq 10 : a_i = 1 + 10(i-1))$ be the first moments of hours in the interval. Let $s_{a_1} = 1$, $s_{a_3} = 4$ and $s_{a_i} = -1$ $(i \neq 1, 3)$. The time points define also the loads of the corresponding hours (knapsack fillings). In the same way we define the prices and revenues: $P_{a_3} = 3000$, $P_{a_i} = 1000$ $(i \neq 3)$ and $R_{a_i} = 10$ $(i \leq i \leq 10)$. Before any clones the result $T(\mathbf{s})$ is $-(1 \cdot 10 \cdot 1000 + 4 \cdot 10 \cdot 3000 + 1 \cdot 10 \cdot 8 \cdot 10) = -130800$.

Moreover, we suppose that the length of the payback is zero, weight $w = 5$, the cost of a clone $P = 5$, minimum length $\text{cmin} = 4$, maximum length $\text{cmax} = 10$, and the safety area $L(c) = 2$ $(m \leq c \leq M)$. At stage 0 the value of state $(0, 0)$ is $T(\mathbf{s}) = -130800$.

We trace the behavior of the states $(k, \text{cmax}+1)(i)$, for $1 \leq i \leq m$ and $0 \leq k \leq K$, and the states $(k, 0)$, for $1 \leq k \leq K$, as long as it is necessary. Because the condition (9.9) is not in use, the recursion formula (9.5) can choose only between the path giving maximum by condition (9.12) and the two complete paths in conditions (9.7)–(9.8). In all other states the algorithm is unable to choose between different paths.

At stages $1 \leq k < m$ the value of state $(0, 1)(k)$ is always $-130800 - P$. The states $(0, \text{cmax} + 1)(i)$ $(1 \leq i \leq 4)$ have $-\infty$ as their value, because these states are impossible by condition (9.10). At stage 5 we have our first value in state $(0, \text{cmax} + 1)$ with equations (9.5)–(9.6), and (9.12). This value is $-130800 - P + 9900 = -120905$. Note that the overload of 1 turns to underload of -1, so that the "penalty" of -10000 turns to be lost of income of -100.

At stage 7 in state $(0, \text{cmax}+1)$ we can choose between three clone lengths of 4, 5 and 6, which have started at time points 2, 1 and 0, respectively. The corresponding values are -120905, -120955 and -121005, respectively. From these $[3, 6]$ is chosen, because it is the latest of the three alternatives of length four. At stage 8 for state $(1, 0)$ clone $[4, 7]$ of length four is chosen instead of clones $[1, 4]$, $[2, 5]$ and $[3, 6]$. By the same way at stage 13 clone $[9, 12]$ is chosen as optimal for state $(1, 0)(14)$, since we need only 2 time points for cutting the overload (filling the knapsacks). The result is of course same as with $[1, 4]$.

Before stage 20 the best result is in state $(1, 0)$. The second clone (the value of state $(2, 0)$ and the clone set corresponding to it) is not profitable, because an additional P and some amount of underload would be added to the result. However, at stage 22 where state $(0, \mathrm{cmax}+1)$ chooses between six different clone lengths, we gain result -115955 with the shortest clone $[18, 21]$. (The underload increases in every time point in hour 2 by 1.5 and the overload decreases in every time point in hour 3 by 0.5.)

For state $(1, 0)(23)$ we choose a path with clone $[18, 21]$. Also the value of state $(2, 0)(23)$ will be -106060 and its path contains $[9, 12]$ and $[18, 21]$. Table 9.3 has the values of state $(1, 0)$ at different stages. At stages $31, 32, \ldots$ the result in the state $(1, 0)$ would not be better. The clone will just move to place $[23, 30]$ without changing its length.

Table 9.4 has values of state $(2, 0)$ at different stages. Because at stage 23 clone $[18, 21]$ is the best among the first clones, it has to be used when forming the second clone after stage 23. This is shown in states $(2, 0)(29)$ and $(2, 0)(30)$. The results will be worse than with one clone; Table 9.3 shows that with one clone at stages 29 and 30 we cut hour 3 less than at the same stages with two clones as shown in the Table 9.4. At stages 31 and 32 the both will cut as much the hour 3, and the only difference is the unnecessary clone.

The optimal clone set is $[9, 12], [23, 30]$, and its result is -995. Now we cannot obtain as good result with two clones without wait states. If we use three clones, we can obtain near optimal result with states (k, c). The difference is more than a cost P of a clone. This means that the optimal solution is not achieved without the wait states.

The reason for the above phenomenon is the combined use of hours and averaging (8.4)–(8.6). This feature also shows up with the method given in the proof of Theorem 5.2. We conjecture that the two state variable DP equals the method of Theorem 5.2.

Table 9.3: The values of state $(1, 0)$ in stages 23–30 with clones.

23	24	25	26	27	28	29	30
-115955	-100905	-85855	-70805	-55805	-40805	-25805	-10805
$[18, 21]$	$[19, 22]$	$[20, 23]$	$[21, 24]$	$[21, 25]$	$[21, 26]$	$[21, 27]$	$[21, 28]$

Table 9.4: The values of state $(2, 0)$ in stages 23–30 with clones.

23	24	25	26	27	28	29	30
−106060	−91010	−75960	−60910	−45910	−30910	−55960	−40910
$[9, 12]$	$[9, 12]$	$[9, 12]$	$[9, 12]$	$[9, 12]$	$[9, 12]$	$[18, 21]$	$[19, 22]$
$[18, 21]$	$[19, 22]$	$[20, 23]$	$[21, 24]$	$[21, 25]$	$[21, 26]$	$[24, 27]$	$[25, 28]$

9.3 Properties of dynamic programming

Next we consider the properties of the dynamic recursion formula (9.5)–(9.6). First we study how many wait states is needed when there is no payback and after it, we embed additional state variables into our state space.

Consider stage i. A *local* clone (control) for state $(k + 1, 0, 0)$ is a clone formed after kth clone, stopped at stage $j > i + L(\text{cmax})$, and using the clone set formed at stage i for state $(k, 0, 0)$. Stages $s > i$ do not belong to the local clone, provided that we do not use the clone set of state $(k, 0, 0)(i)$ at stage s. This means that the wait states are not used when forming a local clone.

In the next theorem we suppose that all references to wait states have been omitted from the conditions (9.7), (9.8), (9.11), and (9.12).

Theorem 9.6. *With the state space (k, c) we will find, for each stage i, the best local clone following stage i.*

Proof. Consider the clones starting after stage i from state $(k, 0)$ and using clone set C determined by i and $(k, 0)$. Conditions (9.7) and (9.8) choose the best clone for the state $(k+1, 0)(j)$, according to the equation (9.5). Condition (9.7) gives the maximum because of the conditions (9.11) and (9.12). □

Corollary 9.7. *State $(1, 0, 0)(m)$ gives the best possible clone set having one clone.*

Note that the length or the amount of radiation (payback) do not have any consequences in the case of Theorem 9.6. As demonstrated at the end of the previous section, state space (k, c) gives sub-optimal results, which can be improved with wait states (still being sub-optimal).

Let $C_i, C_{i+1}, \ldots, C_a$ be the clone sets having k clones, and stages (time points) $i, i + 1, \ldots, a$, respectively, i being the first time point of an hour. In the next theorem we show that it is enough to choose between clone sets $C_i, C_{i+1}, \ldots, C_a$, when forming $[a, b]$. This refers to the situation in condition (9.7) of DP recursion (9.5), where we check how well the clone set of states

$(k, 0, 0)(i), (k, 0, 0)(i + 1), \ldots, (k, 0, 0)(a)$ work with the clone starting at time point a.

Intuitively, the next theorem is based on the property that if the last clone of some clone set stops with safety area (resting time) in the "previous hour", it will not have any impact on the clones in the "present hour".

Theorem 9.8. *Suppose there is no radiation (payback). Suppose further that from stage a we start a new kth clone, which will stop at stage b locally maximizing clone set C_a. Let i be the first moment of the hour containing a. Now it is enough to choose (with the wait states) from the set of clone sets $C_i, C_{i+1}, \ldots, C_a$, when forming a new clone $[a, b]$.*

Proof. We show that it is not necessary to reach time points earlier than the start of the present hour. To do so we consider situations where it is possible to choose between clone set C_{i-1} of time point $i - 1$, some earlier clone set C_{i-n} $(n > 1)$ and clone sets $C_i, C_{i+1}, \ldots, C_a$, when forming $[a, b]$.

Consider kth clone $[a, b]$ started at time point a. To derive contradiction we suppose some clone set C_{i-n} $(n > 0)$, when deciding the proper clone set for $[a, b]$. It follows that at least one of the clone sets $C_{i-n+1}, \ldots, C_i$ gives at least as good result at stage i than C_{i-n}, because DP (9.5) chooses always the maximum. We can suppose that the clone set in question is C_i, since the result of C_j improves when j increases (not necessarily monotonically). If we choose some of clone sets $C_{i-n}, \ldots, C_{i-1}$ to be used with a clone that starts from a, we obtain better result with clone set C_i. □

Note that the absence of radiation is crucial in the above proof, and the fact that the safety area is coded into the state space. The above theorem lets us to conclude that we need one wait state at the first time point of an hour, two at the second time point and finally $h - 1$ at the last time point of an hour (h is the length of an hour). In other words, we need on the average $(h - 1)/2$ wait states at each time point. (In the previous section we used $h - 1$ wait states at each time point.)

Even though we showed in Theorem 9.8 that the results do not improve by increasing the number of wait states, the state space (k, d, c) does not achieve optimal result when the length of radiation is nonzero. We need at least one more state variable A into the state construction (see [13, pp. 30–34]) to be able to form a better path. With variable A we check the paths, which are not maximums according to (9.5) for the three state variable system.

A *local alternative* of stage i is a clone, which stops at stage i including the safety area and which is not chosen into the clone set by the previous equations and conditions. A three variable system chooses the best alternative among

several, as shown in the left side of Figure 9.6. We set this to be alternative state one. In the alternative state two, we choose the second best path from clone sets of size k for the first state $(k+1)$th clone. The third alternative state uses the third best path found so far and so on. Now our state is $S = (k, A, d, c)$ and $S' = (k', A', d', c')$. Instead of (9.4) we use

$$D'(\mathbf{s}, S', S, i+1) = \begin{cases} 0, & \text{when } (9.7)\text{–}(9.10), \\ -P, & \text{when } (9.11), \\ T(E([i-c, i], \mathbf{s})(i)) - T(\mathbf{s}), & \text{when } (9.12) \text{ or } (9.14), \\ -\infty, & \text{otherwise.} \end{cases} \tag{9.13}$$

When

$$\begin{aligned} &k' = k+1, \quad d' = c = 0 \ \text{ and} \\ &d \ \text{ and } \ c \ \text{ such that } \ D\big(\mathbf{s}, (k, A, d, c), i\big) \ \text{ is the } \ A'\text{th best state,} \end{aligned} \tag{9.14}$$

we choose the A'th best path the clone set of size k and set it to the first state of the A'th alternative state corresponding to the clone set of size $k+1$. We allow A to vary in its range, when we are looking for the A'th best path. Condition (9.14) can be taken into account in the implementation of dynamic forward recursion formula (9.5). When we are looking for the best path, we can easily cater the required amount of paths to find the A'th path. Moreover, the solution given by condition (9.8) has also to be checked when we are looking for the number of best paths.

Starting configuration (9.6) and transition conditions (9.7)–(9.12) work with alternative states without major modifications. Starting solution is calculated only for the first alternative state and conditions (9.7)–(9.12) work inside an alternative state as in the case of three variable system.

Now we have applied alternative states to a situation where we choose between different control plans. We could also apply the alternative states in the clone length decision (condition (9.12)).

9.4 Heuristics

This section is based on the work reported in [6]. We develop four heuristic methods for optimization. At the same time we give two different ways to prioritize the items, which are used to choose items in some order, especially when optimizing one item at a time. In the tests we show that these algorithms are

much quicker than methods based on dynamic programming and enumerative methods. The heuristics are called "Direct action", "Long is nice", "Big first" and "Expensive down".

All heuristics use "Count" parameter by which the user can give an upper bound for the number of iterative steps the algorithm in question can take. The "Direct action" and "Expensive down" algorithms use parameters by which one can control how these algorithms choose the items and knapsacks having too much empty space (overload moments). "Long is nice" and "Big first" algorithms have also two other parameters affecting to the selection of the items to be used. These are later explained in greater detail. We use the phrase *empty space interval* when referring to an interval having too empty knapsacks, that is, overload intervals.

The algorithms have some common initial actions to be performed before optimization. These are shown in Figure 9.7. First we extend controls (clones) that have been started but not yet finished. This is connected to a production system, which takes into account the controls made. The extension is made only when it increases the sum (8.12). If there are already some clones, then we might be able to fill the knapsacks with this extension and so, we might manage without new clones. Recall that small overfill is much cheaper than small under fill in the knapsacks. In the next lines we check if in the current hour there are under filled knapsacks. The calling time of a heuristic may restrict the current hour going on, and so we optimize it separately with a variant of "Direct action". Note that in the heuristics, by s_i we mean the overload of hour t and not the overload of a knapsack i.

In Figure 9.8 we show how "Direct action" works. The number of hours is denoted by m_t and it depends on the length of optimization interval m. In the first line we seek the first hour t with under filled knapsacks and in the second line we divide items $\{1, \ldots, n\}$ into two sets: to those that can be positioned to the knapsack t and to those that cannot. If we do not find any item to be placed, then in lines (3)–(6) we try to find an hour to which we can place some items.

(1) Extend clones C_j of items $j \in N = \{1, \ldots, n\}$, if it improves the result
(2) Find the smallest i (hour), for which $s_i > 0$
(3) **if** $i = 0$ **then**
(4)   Optimize the current hour
(5) **end**

Figure 9.7: "Initial actions for each heuristics."

If we can put a clone into hour t in the knapsack array and if we have iteration steps left, then we perform lines (8)–(18). In line (8) we sort the items by the clone weights (control amounts) and the amount of the space $W_j(i)$ clone j fills in hour i. Then in line (9), we choose the clones from the list in ascending order until the sum of the total weight is more than the under filling of the knapsacks (overload in the hour). If the sum is less than s_i, we select the items that can be selected for hour i. We also check the profits here: if we are able to select items such that the sums is over s_i.

After we have chosen enough items, we have to choose the right starting and halting times for each new clone $[a, b] \in C_j$, where $1 \leq j \leq n$. These clones start in some knapsack at hour i (or very near before it), since the halting times are, at first, set to the end of hour t (lines (10)–(11)). When we choose starting times, we obey the constraints concerning the minimum clone lengths. The knapsack might be over filled (there is underload) with the used items. This choice is moved by postponing clones $[a, b]$, that is, by increasing both a and b (line (12)).

After these operations we update the knapsacks with the clone weights (line

(1) Find the first hour i, for which $s_i > 0$
(2) Choose items $N' \subseteq N = \{1, \ldots, n\}$ that can be placed into hour i
(3) **while** $s_i > 0$ **and** $N' = \emptyset$ **do**
(4) Find the next hour $i \leq m_t$ such that $s_i > 0$
(5) Choose items $N' \subseteq N$ that can be placed into hour i
(6) **od**
(7) **while** $s_i > 0$ **and** $N' \neq \emptyset$ **and** Count > 0 **do**
(8) Sort items $j \in N'$ by $W_j(i)$
(9) Choose items $N'' \subseteq N$ such that $\sum_{j \in N''} W_j(i) \geq s_t$
(10) Set the first knapsack (start) a for each clone $j \in N''$
(11) Set the last knapsacks (stop) b to the end of hour i
(12) Postpone each clone $[a, b]$ if it fills empty knapsacks
(13) Update the knapsacks at hour s_i with $W_j(i)$
(14) Choose the last knapsack b for each clone $j \in N''$
(15) Update the conditions, $N = N \cup N' \cup N''$
(16) Extend clones C_j of items $j \in N$
(17) Perform the lines (1)–(6)
(18) Count = Count – 1
(19) **od**

Figure 9.8: "Direct action."

(13)) and then try to postpone the halting times. This depends on each item's maximum length cmax_j, on the length of the under filled knapsack interval, and on the parameter indicating appropriate overfilling level.

In line (15) we update the conditions, and in the next line (16) we extend clones, if it improves the result. This is needed, because some new clones with radiation may make the extending of some old clones profitable.

In “Long is nice” (see Figure 9.9) we first look for the longest interval of under filled knapsacks and check if some clones can be placed into this interval. If not, lines (3)–(6) are executed. We start the optimization in chronological order. When we find an interval where we can place a clone, we jump to line (8).

(1) Find the longest interval with empty space $[r, t] \subseteq [0, m]$, out = false
(2) Choose items $N' \subseteq N = \{1, \ldots, n\}$ that can be placed into hour r
(3) **while** $s_r > 0$ **and** $N' = \emptyset$ **do**
(4) Find next empty space interval $[r, t] \subseteq [0, m] : r > r'$, set $r' = r$
(5) Choose items $N' \subseteq N$ that can be placed into hour r
(6) **od**
(7) **while** $s_r > 0$ **and** $N' \neq \emptyset$ **and** Count > 0 **and not** out **do**
(8) **if** $t - r = 1$ **then**
(9) One iteration step with “Direct action”
(10) **else**
(11) Sort the items $j \in N'$ by the maximum length cmax_j
(12) Find the fullest knapsacks, giving hour u from interval $[r, t]$
(13) Choose the first $j \in N'$, for which $Ls_u \leq W_j \leq Bs_u$
(14) Choose the location of clone $[a, b]$ of j according to u
(15) Update knapsack array **s**
(16) Update the conditions, $N = N \cup N'$
(17) **if s** did not change **then**
(18) One iteration step with “Direct action”
(19) **if s** is still the same **then** out = true **end**
(20) **end**
(21) **end**
(22) Extend clones C_j of items $j \in N$
(23) Perform the lines (1)–(6)
(24) Count = Count – 1
(25) **od**

Figure 9.9: “Long is nice.”

If the length of the interval of under filled knapsacks is only one hour, then we optimize it with "Direct action" (line (9)). Otherwise, we execute lines (11)–(20). We first sort the items by the maximum lengths cmax_k. In the next two lines (12)–(13) we look for an item that can be placed to the fullest knapsack in the interval inside the region determined by user's parameters L and B (little and big). By looking the fullest knapsack we try to avoid unnecessary overfilling. Overfilling could occur if we use some other hour and choose items which cut that hour efficiently.

When the appropriate item is found, we choose the exact location for it in line (14). If it is possible to place clone j over an hour then we check here whether it is better to extend clone $[a, b]$ before or after hour u. Moreover, we check if there is a need to cover the whole hour u by a clone. After this we update knapsack array $\mathbf{s}$ and conditions with equation (8.8) (lines (15)–(16)). In lines (17)–(20) we check if $\mathbf{s}$ changed. If not, the iteration halts.

In the end of the algorithm (lines (22)–(23)) we just extend some clones, provided that it is profitable. After that we prepare for the next iteration step. This algorithm extends clones very often, because the clones are not (usually) handled in chronological order. This increases time consumption, since the algorithm has to correct its actions rather often.

Figure 9.10 shows "Big first" heuristic. In the first two lines we seek for the pair of consecutive hours with the emptiest knapsacks and check if they can be filled. If in the clipping situation there is some single hour with empty knapsacks which are emptier than in any other pair of consecutive knapsacks, then we start with it.

Next we sort the items by their weights (line (4)). Then we choose items that fit to the parameters L and B given by the user (line (5)). If there is no items chosen (line (6), $N'' = \emptyset$), we try "Direct action" in line (13) with all items N. If no more changes are made, we halt the algorithm.

On the other hand, if we have items to be used ($N'' \neq \emptyset$), in line (7) we form a clone for each item just once. For each item $j \in N''$, we find out the largest overload pair $s_i + s_{i+1}$ the item j can cut. Some items $j \in N''$ cannot be placed into the hour pair, because of the constraints. If the knapsack array is kept unchanged, we try "Direct action" and if the clipping situation is still the same, we halt the algorithm.

"Big first" algorithm uses the extending activity quite often. Anyway, it is quicker than "Long is nice", because it tends to allocate clones so that they overfill knapsacks more than "Long is nice" algorithm. This means that if radiation is going to affect some already filled hour, radiation does not cause that hour to have empty knapsacks again as often as in "Long is nice".

"Expensive down" heuristic is shown in Figure 9.11. This is the only heuristic in which we directly take into account the prices of the empty space intervals. This algorithm is based on "Direct action" and "Long is nice" heuristics.

"Expensive down" first looks for the most expensive hour where we can place a clone (lines (1)–(2)). If we do not find any items that can be placed into the most expensive hour, then we start seeking as expensive hour as possible in lines (3)–(6).

In line (7) we seek the borders of the whole empty space interval which includes hour t found earlier. If no t is found, then this line does nothing. If there is an interval where we can place a clone, then we perform lines (8)–(24).

First in line (9) we sort the items by the clone weights each clone can have for hour t (restrictions might prevent some). Then we choose items like in "Direct action" heuristic, that is, if possible, so that the sum of the clone weights is more than the empty space in the knapsacks of this most expensive hour (line (10)). Otherwise we choose every applicable item. The place for

(1) Find hour i, for which $Y' = Y_i + Y_{i+1}$ is the largest, out = false
(2) Choose items $N' \subseteq N = \{1, \dots, n\}$ that can be placed into hour i
(3) **while** $s_i > 0$ **and** Count > 0 **and not** out **do**
(4) Sort the items $j \in N$ by weights w_j
(5) Choose items $N'' \subseteq N'$ for which $j \in N'' : Ls_i \leq W_j \leq Bs_i$
(6) **if** $N'' \neq \emptyset$ **then**
(7) Use each item in N'' once
(8) **if** s did not change **then**
(9) One iteration step with "Direct action" with $N \cup N' \cup N''$
(10) **if** s did not change **then** out = true **end**
(11) **end**
(12) **else**
(13) One iteration step with "Direct action" with $N \cup N' \cup N''$
(14) **if** s did not change **then** out = true **end**
(15) **end**
(16) $N = N \cup N' \cup N''$
(17) Extend clones C_j of items $j \in N$
(18) Perform the lines (1) and (2)
(19) Count = Count – 1
(20) **od**

Figure 9.10: "Big first."

each clone is formed like in "Long is nice" heuristic in lines (11)–(15). Line (15) is needed for checking if the borders of the empty space interval have changed. At the end (lines (17)–(23)) we perform the same tasks as in the other heuristics.

The heuristics may be used directly as they are shown above or with additions that take into account the priorities one might want to have with the items. We give two simple heuristics for dealing with priorities.

The first one sorts the items by priorities. All items in the first priority level are optimized. Then, if there are still usable items and hours with empty knapsacks, the items from the next level are used. This continues until the empty knapsacks are filled or all usable items are handled. This priority method is called "start" in the tests.

---

(1) Find hour i, for which $P_i s_i$ is the largest
(2) Choose items $N' \subseteq N = \{1, \ldots, n\}$ that can be placed into hour i
(3) **while** $s_i > 0$ **and** $N' = \emptyset$ **do**
(4)   Find an unhandled hour i with the largest $P_i s_i$ value
(5)   Choose items $N' \subseteq N$ that can be placed into hour i
(6) **end**
(7) Find the borders r, u of empty space intervals, i.e. $i \in [r, u]$, out = false
(8) **while** $s_i > 0$ **and** Count > 0 **and not** out **do**
(9)   Sort the items $j \in N$ by $W_j(i)$
(10)   Choose items $N'' \subseteq N'$ such that $\sum_{j \in N''} W_j(i) \geq s_i$
(11)   **for** $j = 1$ **to** $|N''|$ **do**
(12)   Choose the location of clone $[a, b]_j$ according to t
(13)   Update knapsack array s_i with $W_j(i)$
(14)   Update the conditions, $N = N \cup \{j\}$
(15)   Find borders r, u of empty space intervals, i.e. $i \in [r, u]$,
(16)   **od**
(17)   **if s** did not change **then**
(18)   One iteration step with "Direct action" with $N \cup N' \cup N''$
(19)   **if s** did not change **then** out = true **end**
(20)   **end**
(21)   Extend the clones C_j of items $j \in N$
(22)   Perform the lines (1)–(7)
(23)   Count = Count – 1
(24) **od**

Figure 9.11: "Expensive down."

Some items with low priorities may not optimize some hours as well as items with high priorities. In these cases the earlier used items become unnecessary in certain hours and their space filling capacity can be better used somewhere else. This phenomenon is similar to the cases where radiation changes the knapsack array.

The second method with priorities is similar to the first one, except that when moving from one priority level to the next, we start the whole optimization process from scratch. This means that we do not use the already made set of clones, and at the same time, we do not need to check for the unnecessary items so often. We call this method "extending" in the tests. This method runs in time proportional to the square of the number of items while linear time is sufficient for the first heuristic.

9.5 Genetic algorithms

Genetic algorithms (GAs) are efficiently used with many different and noteworthy difficult combinatorial problems [60, 76, 92]. As we have already pointed out, GAs are also used with unit commitment and other problems related to electricity management [34, 36, 61, 66].

The basic idea behind GA is taken from the evolutionary biology [76]. Figure 9.12 contains the fundamental structure of a *simple GA*. Usually one codes an *individual* as a bit vector, initializes it in line (1) and applies different "evolutionary" operators for it in line (3) in order to produce an *offspring*. Typical operators are *mutation* and *crossover*. Individuals form a *population*. In line (4) we select from population some individuals that survive to the next generation with *selection* operator.

An individual consist of *genes*: in the case of bit vector, the genes are bits. A simple crossover operation swaps the tails of two individuals at random places after which the mutation flips each gene at a time from both offsprings with a given probability. The usual assumption in GA is that an individual

```
(1) Initialize population
(2) while not termination do
(3)         Produce new individuals by means of evolutionary operations
(4)         Select new individuals into the population
(5) od
(6) Report the results
```

Figure 9.12: Structure of simple GA.

consists of tightly connected, larger parts, that is, genes form "building blocks" [76] having desirable properties. After some good blocks have been formed to different individuals the crossover should be able to find the individuals combining all the good aspects the parents have.

We produce offsprings from the old population until we have reached the new population. If we are using *tournament* selection children do not necessarily replace their parents; individuals are compared and better ones will survive with some probability.

We may also select some of the best individuals directly to the new population; this selection is called *elitist*. The best individual is found by examining the *fitness* values of individuals. The fitness is calculated with the objective function in question. Each iteration of lines (2)–(5) forms a new *generation*.

The termination is determined with fitness values calculated for each individual. We may terminate, for example, if average fitness does not change for several generations, if we have passed some predefined number of generations, or if some given amount of time has been used for optimization. There are several different operators for mutation, crossover and selection for different problems. (See [76].)

Instead of bit vector coding, we use similar data structure for GA as described for the enumerative solution in Section 9.1. Each gene has, for a clone, an integer pair (x, y), where x is the number of knapsacks before the clone containing no other clones, and y is the length of clone. As opposite to integer composition, the length of clone is the second number. This integer pair coding has the advantage that we can automatically keep some of the restrictions in order, like the safety area. Table 9.5 contains twelve genes as an example.

An individual is either a set of clones of one type (IKHO) or a class of clone sets (IKO or MDIK). Table 9.5 has six individuals for IKHO. We may interpret each gene pair as an individual, the upper genes 1, 3, and 5 being of type one and the genes 2, 4, and 6 on the bottom of type two in the case of IKO. Or, for IKO there could be three individuals, left 1 and 2, center 3 and 4, and right 5 and 6 genes. In load clipping problem we test only IKHO approach (see Chapter 10).

First we describe the methods implemented for the IKHD problem where

Table 9.5: Example of an individual.

1 : (5,10)	(15, 5)	3 : (5,20)	(15,5)	5 : (5,10)	(15,5)
2 : (10,5)	(5, 5)	4 : (30,5)	(15,5)	6 : (5,10)	(15,5)

each individual is a set of clones. We implemented two different crossovers. In the first one we randomly select the swap point (number of genes from start), and swap the tails of two parents. As soon as the first tests started we noticed that this approach leads to difficulties because it affects too strongly to the clones in the tails of individuals. If the tail on the second parent is ideally placed, the different head of the first parent causes the tail to move away from its ideal place, hence reducing the fitness value.

Consider the rows of Table 9.5 describing two individuals. Let the crossover operation occur between the second and the third genes. We see that the crossover point changes the tails and produce individuals totally different from their parents. Before crossover they had equal total lengths of 115 but after the operation the top line is 20 longer than the bottom line. If the last three clones were on the optimal place before crossover, they are in nonoptimal place after the crossover.

In the second version of crossover operator we also select a random swapping point but this time from the whole optimization interval, which is $[1, 115]$ in the example in Table 9.5. Then we interpret the clones so that when one knapsack contains a clone, in the time line we have a bit on, otherwise off. We swap the tails of bit vectors on the time-line and map the situation back to the clone set representation. This way we often split a clone and offend some conditions, usually the minimum or maximum clone length, like point 34, forming a clone of length one into the second offspring in Table 9.5. We decided to remove all illegal clones from offsprings through the bit mapping and hence keeping the tails on their places. However, the removed copies of clones are sometimes quite valuable and this also tends to hinder the progress, but not so much as the first version in general.

We have not tested a version, where we extend (or suppress) the illegal clones in order to make them legal again in the second version of crossover. This resembles the tabu search methods.

Mutation operator has similar problems in load clipping as the crossover has and we ended up with two different mutations. In both versions we traverse through the genes and before each gene a random choice is performed about adding a new gene (that is, a new clone). Without adding new genes, the number of genes in population tends to decrease, because we remove illegal genes in the crossover. After that a random choice about removing the gene (clone) in question is done. Thus, our individuals are of different length.

Tests made it evident that the crossover does not easily move tail of an individual nor change the lengths of clones. Without moving the tail, GA may produce individuals, where every clone (genes in an individual) are sys-

tematically too early or late, and the crossover has difficulties overcoming this problem. Thus, if a gene is not removed, we randomly choose between increasing or decreasing either the first or the second number in the gene. This corresponds to the move of tail or changing the length of a clone (and hence moving the tail), respectively.

The first version of mutation has the same problems as the first crossover: one change in the beginning of the individual has too large impact on the last genes. Hence, we implemented the second version where we tried to keep the tails in their places as well as possible through the mapping to bit vectors and back. The behavior of different combinations of genetic operations is discussed in greater detail in Chapter 10.

As selection we have tournament selection and the elitist selection with tournament. In the tournament selection both of the two parents compare to the offsprings and better ones will survive with some probability, typically 0.75. In the elitist selection we order the population by fitness and then choose a given amount of the individuals directly to the next generation. The amount is the *elitism* parameter in the tests. The rest of the offspring will be generated just like in the tournament selection. The termination is after a given number of generations: we do not use any limiting values to quit earlier.

It is also possible to mutate and crossover the parameters affecting the probabilities by inserting the parameters controlling the probabilities of individuals into the individuals. This increases the number of different combinations to test. We noticed rather soon, however, that static parameters work best.

Note that we have considered load clipping as static and stationary optimization problem in other methods presented so far. However, load clipping is essentially nonstationary because the clipping situation changes as time passes by. In practice we even have some additional restrictions related to the time of day not presented in this work. Genetic algorithms for nonstationary function optimization are considered in [35, 50, 82].

Because the nonstationarity is a consequence of the time flow, we can drop unnecessary parts from the beginning of the individuals. After that we continue the normal operation by making new generations. Dropping the items can be justified by the fact that electricity consumption forecasts are fairly good for the first hours in the optimization interval while the larger errors tend to occur at the last hours [107]. Hence, the head of the individual is good for the optimization problem with high probability. This dropping scheme does not assume anything about the clipping situation at the last hours leaving it all to GA and evolution. And after the time arrives to a moment when we have to give a new control plan (set of clones of each type), the first moments in the

old plan are already in the past.

The parallel GAs [51] are very helpful when taking into account the stochastic and nonstationary nature of load clipping problem. We can at the same time evolve several populations with a bit different clipping situations. When we get some new data about the actual clipping situation we have much of the calculations done.

Parallel GAs can also help in our problem in other ways. If a population gets stuck into a local minima, we can make a few crossovers between different populations that are meant for different situations in hope that we move away from local minima. However, we have not tested parallel GAs.

Chapter 10

Experiments

We made extensive experiments with the methods given in the previous chapter. Each method was implemented but we made comparisons only between heuristics, dynamic programming, and genetic algorithms. The integer composition algorithm was soon left out as an unpractical method.

First, Section 10.1 presents the comparisons made to find out if there are some easy instances for the heuristics. After that, Section 10.2 presents our experiments on harder instances. We also made tests on real life instances (see Section 10.3). Section 10.4 contains experiments with the state structure of DP to confirm the results of Section 9.3.

In the experiments we tested the following seven hypotheses. We did not apply statistical tests. The hypotheses are a bit vague but some assumptions and explanations can be found below. The hypotheses refer to the economic results except the second one.

- H_1: Each heuristic has instances that are natural for them.
- H_2: Heuristics are quick.
- H_3: GAs are better than the heuristics.
- H_4: Longer running time improves the performance of GAs.
- H_5: DPs are the best among the presented methods.
- H_6: DPs with alternative states are better than three variable DPs.
- H_7: Results improve as the number of alternative states is increased.

Heuristics were designed with certain kinds of instances in mind. The first hypothesis is based on the assumption that each heuristic has instances (or sets of instances) where they work the best. Because we have several heuristics, at least one should obtain a good (or tolerable) result. Hypothesis H_2 is for the running times. Another design criterion for heuristics was that they can be applied even if there would be only a couple of seconds time. As

a general purpose methods, the heuristics should not work as well as GAs. The evolution should overcome the local optimums more often than heuristics, thus H_3. Giving more time to the evolutionary operations, the results should improve H_4, because of the stochastic nature of GAs. Even thought DPs do not do full search space exploration, the assumption behind H_5 is that DPs do extensive search. The new state variable, that is, the alternative states should help in the search H_6. Because we can add alternative states for each subsolution found, this widens the number of explored solutions, and therefore H_7. We discuss the validity of hypotheses and give conclusions of the experiments in Section 10.5.

For each problem instance we present the clipping situation, items, and obtained results. Appendix C contains exact data, while some of the data is presented graphically in this chapter. For each instance, we calculated a starting value and a value that only sums up the losses of revenues neglecting the overload hours. The latter value is called *ideal*. If a method gets solution value near the ideal, the method works very well for that particular instance.

Each clipping situation has 25 hours, which totals up to 300 knapsacks with 5 minute slots. Thus, hour consists of 12 knapsacks. Recall that weights are averaged in each hour. All tests were driven in 450MHz Pentium with Linux.

Table 10.1 contains summary of the methods with abbreviations we used. If the sorting method is priority, it is the primary key and item weight is the secondary key. If the sorting method is (item) weight, we do not use priorities at all. The *priorities* column refers to two methods ("start" and "extending") presented at the end of Section 9.4. There are two cleaning methods that try to improve the results: if there are several items optimized one at a time, an item can cause a clone to be too long or unnecessary. In the cleaning we simply test, for each clone, whether it can be removed or made shorter. Last column gives the maximum number of generations for each version of genetic algorithm. (Alike DPs, GAs use similar approach to several items: optimize successively one item at a time. Although GAs could be extended to optimize several items at a time, we have not tested it.) All methods implement an extra restriction, which is the sum of clone lengths (cumulative clone length).

Besides the number of generations, the other GA parameters are: tournament parameter 0.85, elitism 65, size of population 120, crossover probability 0.05, and mutation parameters 0.05 for adding or deleting a clone and 0.2 for the length changing (there is two lengths for a clone: length before start of a clone and length of the clone itself). We did not let GA change its parameters and both the mutation and crossover operator use the mapping to binary vector and back.

Table 10.1: Method abbreviations.

Name	Method	Priorities	Sorting	Cleaning	Max gener.
DA1	Direct action	start			
DA2	Direct action	extending			
LIN1	Long is nice	start			
LIN2	Long is nice	extending			
ED1	Exp. down	start			
ED2	Exp. down	extending			
BF1	Big first	start			
BF2	Big first	extending			
DP1	DP		priority		
DP2	DP		weight		
DP1C	DP		priority	yes	
DP2C	DP		weight	yes	
GA2000P	GA		priority		2000
GA2000W	GA		weight		2000
GA2000PC	GA		priority	yes	2000
GA2000WC	GA		weight	yes	2000
GA1000P	GA		priority		1000
GA1000W	GA		weight		1000
GA1000PC	GA		priority	yes	1000
GA1000WC	GA		weight	yes	1000
GA500P	GA		priority		500
GA500W	GA		weight		500
GA500PC	GA		priority	yes	500
GA500WC	GA		weight	yes	500

The GAs were tested also separately to find these parameters. We tried all the six combinations of the mutation and crossover operators implemented. The best combination found used mapping to bit vector both in the mutation and crossover.

The combination with both the mutation and crossover operated directly with compact individual representation was clearly the worst. The main problem was that if a small change happened in the head, the tail changed accordingly. If the tail was in its optimal position, the change moved it away. This is against the "building block hypothesis" [76]: the optimal tail should not move on a local change to somewhere else. The crossover can also have a large impact on tails.

Other combinations usually did find a locally optimal solution. When the elitism parameter was decreased, the average scores tend to become worse. The local optimum, however, was almost always found. The tournament parameter did not have a crucial impact on finding the local optimum nor on the average scores. The application of mutation and crossover to the parameters of GA led only to situation, where the parameters were converging to zero and the average scores were poor.

We mainly used eight items with weights between 0.2 to 7.0 given in Table C.2 in Appendix C.2. The eight item data set was obtained from a Finnish electricity supplier. Also variants of the given set were used.

In tests the price was 7000 and the revenue was 50 for each hour. With different prices and incomes in different hours, DPs, GAs, and "Expensive down" would give different results, while heuristics would give the same results. In the tables, the results concerning GAs are averages of ten test runs. Further, column K counts the number of clones times ten representing the cost of a clone set (control plan).

10.1 Instances good to heuristics

Next three instances are depicted in Figure 10.1. The grey line indicates the best solution found. Horizontal ticks describe 1 MW load (the lines in the last hour are not in correct positions). Figures 10.2, 10.3, and 10.4 contain the clone sets for each item giving the best solution. Exact values of initial situation, item descriptions, and the best solutions are in the tables in Appendix C.2.

The first test contained five overload (empty space) intervals with different lengths and with relatively small overloads. The radiation (payback) could cause new overloads after the second, third or fourth overload interval. The results show that heuristics are very fast, and DPs and GAs are fast enough. All results are close to the ideal. Note that DPs are able to make better results than the ideal: they do "valley filling". DP sorted by clone weights with cleaning gives the best solution. It used four of the eight items.

The second test was favorable for "Long is nice" heuristic. It contained a short overload interval in the beginning of optimization interval and longer in the middle. With the given item set, the other heuristics tend to use their long clones to the first overload interval while it should have been saved for the longest overload interval. "Long is nice" reached almost the ideal result, while two DPs gave even better results. On average, GAs outperformed heuristics (except "Long is nice"). DP sorted by clone weights with cleaning also gives the best solution for this instance. It used all five items. The overloads have been used to increase the profits with radiation.

The third test was similar to the second one. This time, there were three items and only one clone for each item was allowed. The first overload interval was shorter while the second was as long but a bit larger. The only heuristic performing well was "Big first". The other heuristics waste their resources in wrong places. It is noteworthy that this time BF found the best result. Heuristics and DPs used all the three items. GAs had severe difficulties with this test. Somehow, GAs were able to explore only a small portion of the search space. For example, GA2000WC, GA1000WC and GA500WC returned the same solution in all test runs, thus explaining the same averages.

Cleaning usually improved the results of DPs and seemed to improve the results of GAs as well (although, with ten test runs the randomness plays a role in the results). We expected to obtain the result within 5 minutes (which is the same time as the time represented by a knapsack). Thus, every method was fast enough with the given item sets used in the test.

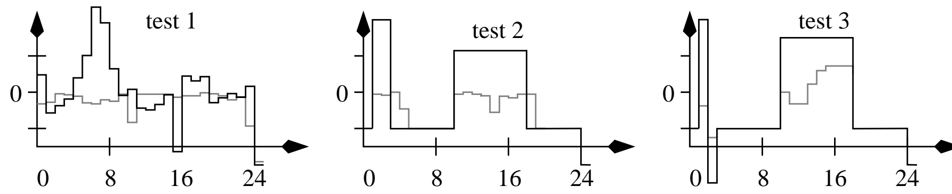

Figure 10.1: The first set of test instances.

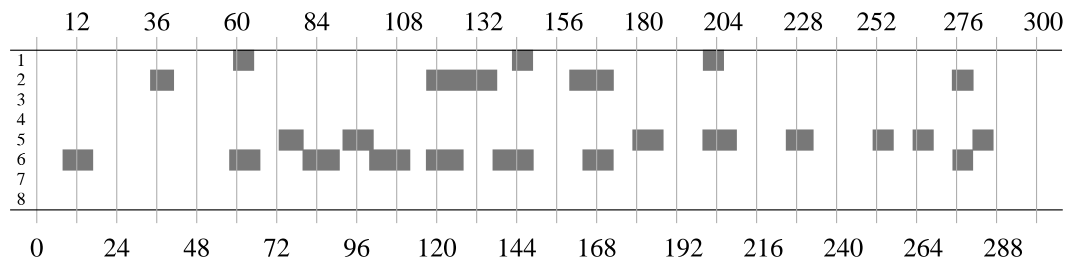

Figure 10.2: Clones for items (control plans) for the best solution found in the first test.

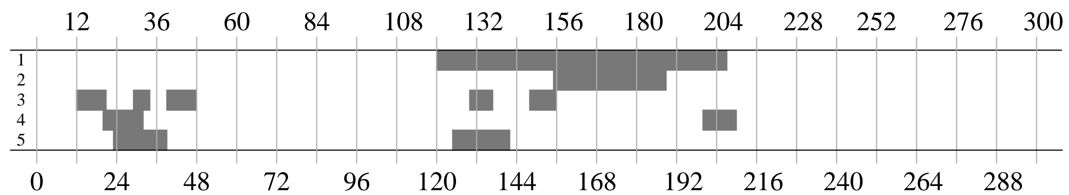

Figure 10.3: Clones for items (control plans) for the best solution found in the second test.

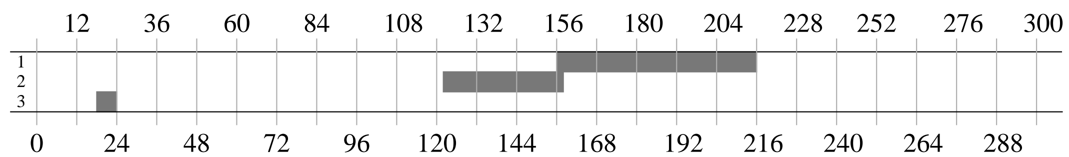

Figure 10.4: Clones for items (control plans) for the best solution found in the third test.

Table 10.2: Results for the first test. Starting result is −60670 and the ideal is −960.

Alg.	Result	K	Time	Alg.	Result	K	Time
DA1	−979	−110	0.1	GA2000P	−1102	−68	129
DA2	−1016	−90	0.1	GA2000W	−1328	−163	164
LIN1	−998	−130	0.2	GA2000PC	−1072	−73	134
LIN2	−1012	−90	0.1	GA2000WC	−1131	−159	162
ED1	−974	−90	0.1	GA1000P	−1113	−69	66
ED2	−1013	−90	0.1	GA1000W	−1313	−160	85
BF1	−1036	−100	0.1	GA1000PC	−1119	−65	65
BF2	−1029	−80	0.1	GA1000WC	−1182	−166	81
DP1	−885	−180	54	GA500P	−1127	−69	32
DP2	−745	−240	45	GA500W	−1305	−156	45
DP1C	−855	−170	53	GA500PC	−1119	−64	32
DP2C	−698	−230	44	GA500WC	−1137	−147	37

Table 10.3: Results for the second test. Starting result is −96400 and the ideal is −1200.

Alg.	Result	K	Time	Alg.	Result	K	Time
DA1	−18459	−110	0.1	GA2000P	−5919	−100	180
DA2	−26694	−110	0.1	GA2000W	−3246	−110	188
LIN1	−1214	−100	0.2	GA2000PC	−3316	−87	183
LIN2	−1414	−100	0.1	GA2000WC	−5777	−91	186
ED1	−15859	−110	0.2	GA1000P	−7310	−100	89
ED2	−8489	−110	0.1	GA1000W	−5048	−106	94
BF1	−31488	−110	0.2	GA1000PC	−5646	−83	90
BF2	−26510	−110	0.1	GA1000WC	−6679	−92	93
DP1	−4443	−110	13	GA500P	−6788	−101	45
DP2	−1186	−110	15	GA500W	−6945	−106	46
DP1C	−3690	−110	13	GA500PC	−3457	−90	46
DP2C	−1128	−110	15	GA500WC	−4921	−95	47

Table 10.4: Results for the third test. Starting result is −99350 and the ideal is −1350.

Alg.	Result	K	Time	Alg.	Result	K	Time
DA1	−68920	−30	0.1	GA2000P	−40592	−30	90
DA2	−68920	−30	0.1	GA2000W	−40500	−20	83
LIN1	−31529	−30	0.1	GA2000PC	−40544	−30	91
LIN2	−36453	−30	0.1	GA2000WC	−40481	−20	84
ED1	−68929	−30	0.1	GA1000P	−40592	−30	45
ED2	−41873	−30	0.1	GA1000W	−40502	−20	41
BF1	−21852	−30	0.1	GA1000PC	−40544	−30	45
BF2	−68920	−30	0.1	GA1000WC	−40481	−20	42
DP1	−22015	−30	6	GA500P	−40596	−30	23
DP2	−30182	−30	6	GA500W	−40506	−20	21
DP1C	−22015	−30	6	GA500PC	−40544	−30	23
DP2C	−30149	−30	6	GA500WC	−40481	−20	21

10.2 Difficult instances

Next two instances are depicted in Figure 10.5. Exact values are in the tables in Appendix C.3. While the second test was artificial, the first could occur in practice when an electricity supplier tries to cut down the loads by buying electricity beforehand (that is, "deciding the sizes of knapsacks"). Because DPs did not use the restriction described in Appendix C.2, they had a small advantage over the other methods.

In the first test we had a difficult clipping situation. There was overload in every hour (knapsacks are under filled). However, the overloads could be cut down quite well. Note that the sum of the weights of the first two items was approximately 1.8, meaning that after these items there were two peaks left, which the rest of the items can handle.

The execution times for GAs with 2000 generations were just a little bit too long, but one may buy some extra time with quick heuristics and then use GAs. However, GAs did not give good results with this instance. Heuristics perform well: either the "start" or the "extending" way to handle priorities has obtained almost the ideal solution. The sorting of items affected the results of DPs. Here we had time to try all DPs in the given 5 minute maximum execution time (the solution can be saved before performing cleaning, hence saving time). If neglecting the cost of clone set, DPs gave better than the ideal solution.

Some of the differences in the results can be explained with the items that were used in the end of the optimization interval. DPs utilized the radiation while heuristics used this only by chance.

The second hard instance could not be cut with the given items. DPs outperformed the other methods and were fast enough. GAs outperformed the heuristics and GAs with 500 generations were fast enough. In both tests, GAs seemed to improve the results with the increase of generations. In this test DPs let some of the items to break the "cumulative length" restriction (see Appendix C.2).

Note that the third item was hardly used at all. The reason for this was that the radiation for item 3 was larger than the clone weight. Hence, item 3 did not comply with (8.3) (we do not require it to comply, even though items used in electricity management usually do). This means that every clone of type 3 increases the total consumption of electricity. Since there were overload in every hour except the in the last one, the use of item 3 was useless.

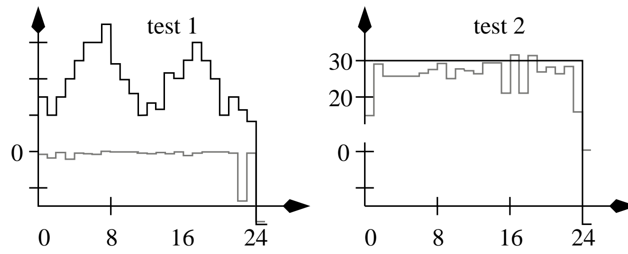

Figure 10.5: Difficult test instances.

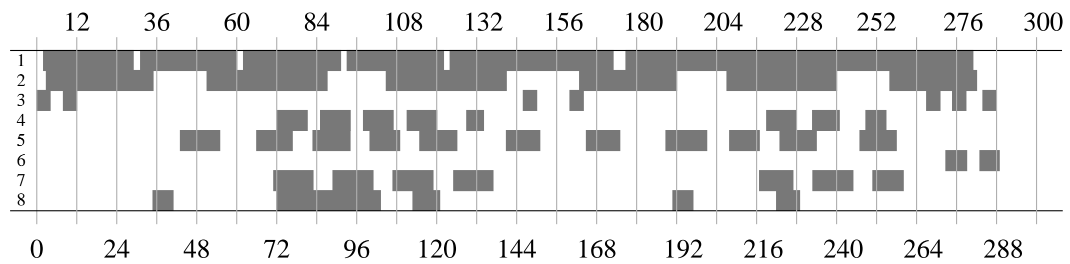

Figure 10.6: Clones for items (control plans) for the best solution found in the first hard instance.

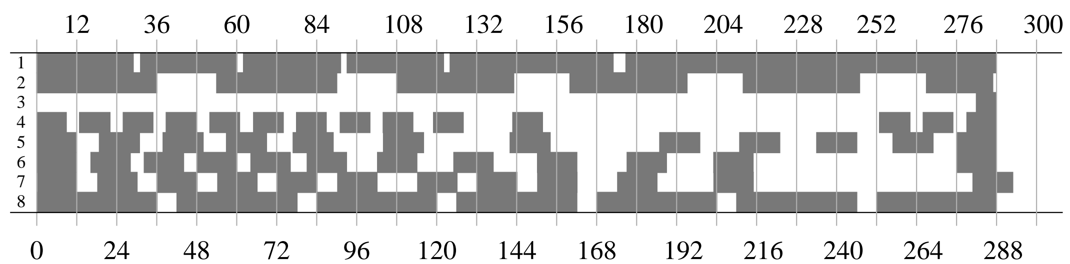

Figure 10.7: Clones for items (control plans) for the best solution found in the second hand instance.

Table 10.5: Results for the first hard instance. Starting result is −324600 and the ideal is −500.

Alg.	Result	K	Time	Alg.	Result	K	Time
DA1	−680	−290	0.6	GA2000P	−17395	−564	523
DA2	−12594	−470	0.8	GA2000W	−10274	−473	466
LIN1	−2605	−390	2	GA2000PC	−10788	−451	523
LIN2	−11035	−370	2	GA2000WC	−6165	−443	452
ED1	−594	−350	1	GA1000P	−22894	−553	259
ED2	−674	−450	2	GA1000W	−8498	−437	216
BF1	−4313	−360	1	GA1000PC	−16384	−470	260
BF2	−639	−420	1	GA1000WC	−6282	−449	227
DP1	−304	−530	103	GA500P	−25561	−551	126
DP2	−12078	−610	106	GA500W	−9230	−445	108
DP1C	−268	−520	103	GA500PC	−14856	−446	121
DP2C	−12082	−600	107	GA500WC	−10332	−436	108

Table 10.6: Results for the second hard instance. Starting result is −5040500 and the ideal is −500.

Alg.	Result	K	Time	Alg.	Result	K	Time
DA1	−4625050	−400	0.8	GA2000P	−4544249	−606	579
DA2	−4625050	−400	0.5	GA2000W	−4547330	−609	577
LIN1	−4578900	−440	3	GA2000PC	−4546253	−600	585
LIN2	−4618620	−450	1	GA2000WC	−4547927	−597	586
ED1	−4621010	−430	2	GA1000P	−4552078	−615	292
ED2	−4623660	−440	1	GA1000W	−4554737	−619	290
BF1	−4625050	−400	1	GA1000PC	−4554199	−598	286
BF2	−4625050	−400	1	GA1000WC	−4550034	−602	288
DP1	−4468947	−700	154	GA500P	−4553593	−618	141
DP2	−4468947	−700	153	GA500W	−4554650	−602	140
DP1C	−4469354	−690	158	GA500PC	−4552940	−602	142
DP2C	−4469354	−690	157	GA500WC	−4552002	−584	138

10.3 Normal instances

In this section we describe three tests: a morning peak, an afternoon peak, and one containing both morning and afternoon peaks. The three instances represent normal problem instances occurring in load clipping.

In the morning peak DP with cleaning and items sorted by weights gave the best solution. The overloads were cut down well and the radiation was utilized to increase the revenues.

Item 3 and its radiation (which is larger than clone weight) was used in the afternoon peak instance. Since the consumption was below the zero-level, we could increase the revenues by using item 3. Heuristics did not see this behavior while DPs and GAs did. The afternoon overloads could be cut well. Heuristics gave results near the ideal. They cut the overloads well but did not increase the revenues, which is not bad at least from the view point of an electricity supplier, since they usually do not want to make controls, unless there is overload.

With both peaks, item 3 was not used as much as with the afternoon instance. Now the second item is used to cut down the first peak and its radiation increased the revenues. Item 2 moved some of the consumption into the valley. Again, overloads were cut down well.

GAs almost achieved the ideal result on average in the morning peak instance. Moreover, they did almost as well as heuristics, also in the morning and afternoon peaks instance. However, in the afternoon peak, and morning and afternoon peaks instances there were much more divergence than in the morning peak instance. DPs gave the best results, as usual. Each method was sufficiently fast.

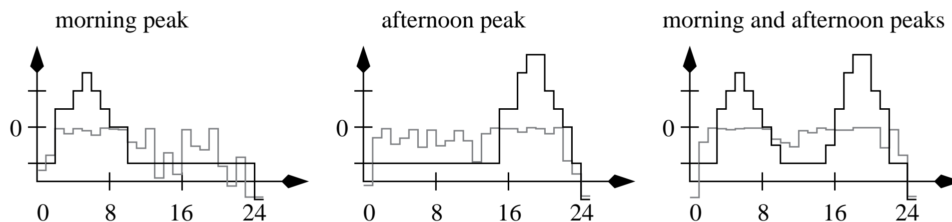

Figure 10.8: Morning and afternoon peaks.

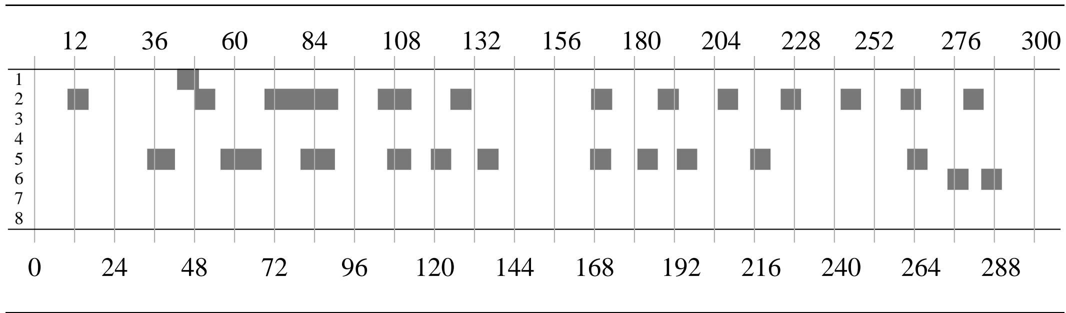

Figure 10.9: Clones for items (control plans) for the morning peak.

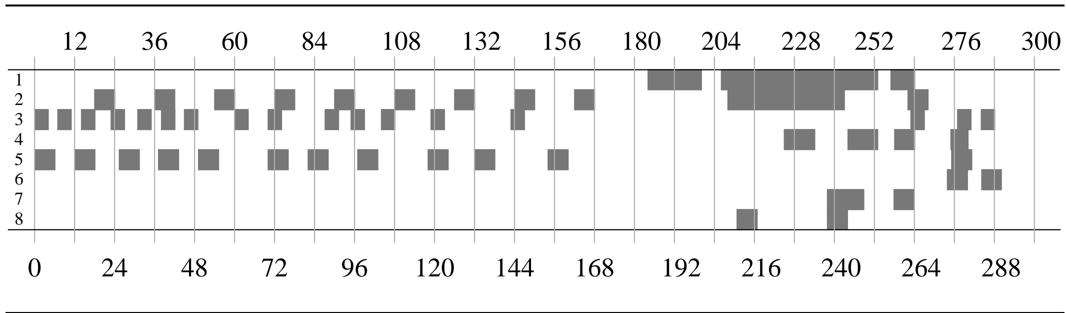

Figure 10.10: Clones for items (control plans) for the afternoon peak.

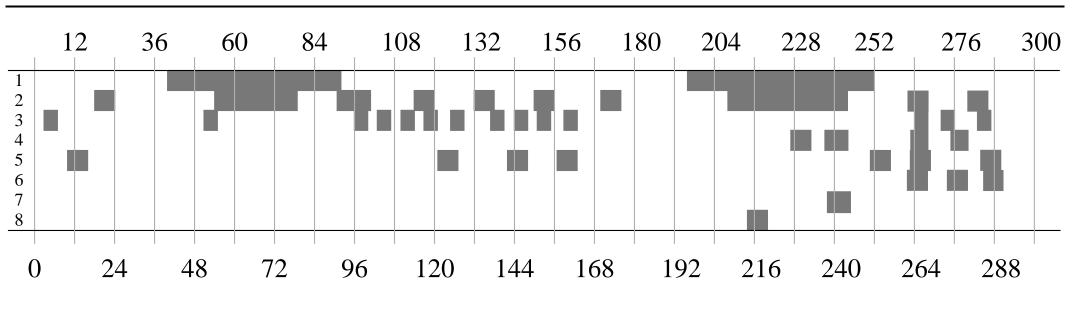

Figure 10.11: Clones for items (control plans) for the morning and afternoon peaks.

Table 10.7: Morning peak results. Starting result is −36300 and the ideal is −1300.

Alg.	Result	K	Time	Alg.	Result	K	Time
DA1	−1340	−20	0.1	GA2000P	−1377	−41	119
DA2	−1340	−20	0.1	GA2000W	−1432	−91	131
LIN1	−1340	−20	0.1	GA2000PC	−1350	−42	122
LIN2	−1340	−20	0.1	GA2000WC	−1357	−80	129
ED1	−1334	−50	0.1	GA1000P	−1382	−41	60
ED2	−1340	−20	0.1	GA1000W	−1411	−82	59
BF1	−1349	−50	0.1	GA1000PC	−1351	−42	62
BF2	−1340	−20	0.1	GA1000WC	−1347	−76	62
DP1	−1150	−150	54	GA500P	−1375	−39	29
DP2	−913	−280	67	GA500W	−1414	−76	28
DP1C	−1164	−140	55	GA500PC	−1351	−42	31
DP2C	−896	−260	69	GA500WC	−1365	−76	28

Table 10.8: Afternoon peak results. Starting result is −58000 and the ideal is −1300.

Alg.	Result	K	Time	Alg.	Result	K	Time
DA1	−1310	−80	0.1	GA2000P	−4114	−142	168
DA2	−1325	−70	0.1	GA2000W	−1688	−119	127
LIN1	−1317	−90	0.1	GA2000PC	−1345	−54	153
LIN2	−1352	−80	0.1	GA2000WC	−1349	−106	145
ED1	−1334	−50	0.1	GA1000P	−8094	−120	78
ED2	−1327	−80	0.1	GA1000W	−1440	−102	66
BF1	−1352	−60	0.1	GA1000PC	−2098	−64	79
BF2	−1335	−50	0.1	GA1000WC	−1335	−87	72
DP1	−595	−540	110	GA500P	−1716	−121	39
DP2	−854	−320	80	GA500W	−1507	−105	34
DP1C	−576	−530	116	GA500PC	−1357	−55	35
DP2C	−731	−310	83	GA500WC	−1325	−93	29

Table 10.9: Morning and afternoon peaks results. Starting result is −85050 and the ideal is −1100.

Alg.	Result	K	Time	Alg.	Result	K	Time
DA1	−1113	−90	0.1	GA2000P	−7931	−124	167
DA2	−1074	−90	0.1	GA2000W	−1161	−137	158
LIN1	−1058	−170	0.2	GA2000PC	−1113	−91	148
LIN2	−1085	−120	0.1	GA2000WC	−1110	−146	169
ED1	−1115	−80	0.1	GA1000P	−1730	−114	78
ED2	−1081	−80	0.1	GA1000W	−1230	−143	78
BF1	−1116	−110	0.1	GA1000PC	−1755	−102	92
BF2	−1084	−70	0.1	GA1000WC	−1110	−129	72
DP1	−394	−430	82	GA500P	−4897	−108	39
DP2	−681	−250	56	GA500W	−1219	−149	43
DP1C	−367	−420	84	GA500PC	−1108	−87	39
DP2C	−624	−240	57	GA500WC	−1099	−138	40

10.4 DP state structure

We tested the running times of DPs and compared the accuracy of results. The running time comparison was made against the maximum clone length cmax. We compared two DP implementations, one with and another without the wait state saving described in Theorem 9.8. We used the first item of Table C.2 in Appendix C.2, and let cmax to vary from 20 to 200 (from 100 minutes to 1 000 minutes). This item did not have radiation.

On the left side of Figure 10.12 is the clipping situation: 23 hours have 0.5 overload and the last two ones do not. Each hour is discretized to twelve points. Hence, with clone of length 8 (40 minutes control) one gets 0.53 cutting capacity (clone weight).

Both DPs gave the same set of clones containing twelve clones each cutting two hours at a time with clones of length 16 (80 minutes control). The clone set is shown in Figure 10.13. After controls, there remained 0.03 MW underload in each hour (shown as the grey area in Figure 10.12), except the last two. In both cases the obtained result was $-1\,590$. (In the beginning the value is $-1\,139\,400$ and the ideal value was -900.)

Only the running times were different between DPs. The right side of Figure 10.12 shows the running times for old DP with full state space and for the new one with half state space. DP with new state space was about twice as fast as the old one, which is consistent with the theory. The number of wait states was 11, because the length of an hour was 12 (see page 114 and Theorem 9.8).

Next we made experiments involving radiation. Wait states were used to overcome the difficulties with the hourly averages, and the radiation could affect hours not directly involved with the clone. We assumed that radiation affects the accuracy and that by increasing the number of wait states we were able to improve the accuracy.

Intuitively, the wait states starting from point b can only "look up" that far

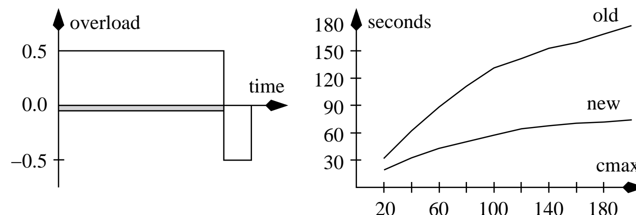

Figure 10.12: Clipping situation and running time on maximum clone length.

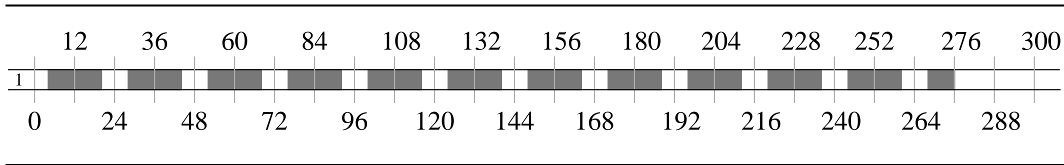

Figure 10.13: Set of clones cutting the overloads of Figure 10.12.

later on at moment a. So if b is the beginning of an hour, we "do not see" into the previous hour at moment a and cannot affect decision made before, that is, make the choice between different clone sets. When there is no radiation, the number of wait states equaling to the number of time points after a start of an hour is enough by Theorem 9.8, because a clone finished in the previous hour does not affect the present hour to be cut. (This does not mean that the system is optimal. All we know is that there is no need to increase the number of wait states.)

However, when we are using radiation, we have to be able to look further into the previous hours in order to increase the accuracy. We have two ways to increase the number of wait states.

By Theorem 9.8 the number of wait states is $a \bmod b$ in current time point a, where b is the length of one hour and a is the current time point. Now, we can increase b, or, on the other hand, we can directly increase the number of wait states to $c + (a \bmod b)$.

Increasing b does not improve the solutions much. The reason is that $a \bmod b$ divides the time line into the disjoint intervals of length b. We cannot "see" into the previous interval and basically, our problem remains. The time line is still divided into disjoint areas and the optimum is easily lost, because radiation can arbitrarily affect the next interval.

By adding c states we improve the ability to see to earlier hours (or into earlier intervals of length b). We were tempted to think that increasing c will improve the solution. Our tests, however, show that while this is mostly true, there are exceptions.

Again, each hour is discretized into twelve time points. The item we used is shown in the upper left corner in Figure 10.14. The black line indicates the radiation for a maximum length clone (cmax = 12, hour) and the gray line for the minimum length clone (cmin = 6, thirty minutes). Clone weight $W = 1.2$ (control capacity is 1.2 MW, or 0.1 for five minutes), safety area is 2, and the number of clones is not restricted.

We tested the item with 14 clipping situations, of which 10 were quite artificial, while 4 (tests 6–9) could be normal clipping situations occurring in reality. Tests 6–9 were similarly shaped and contained the morning and

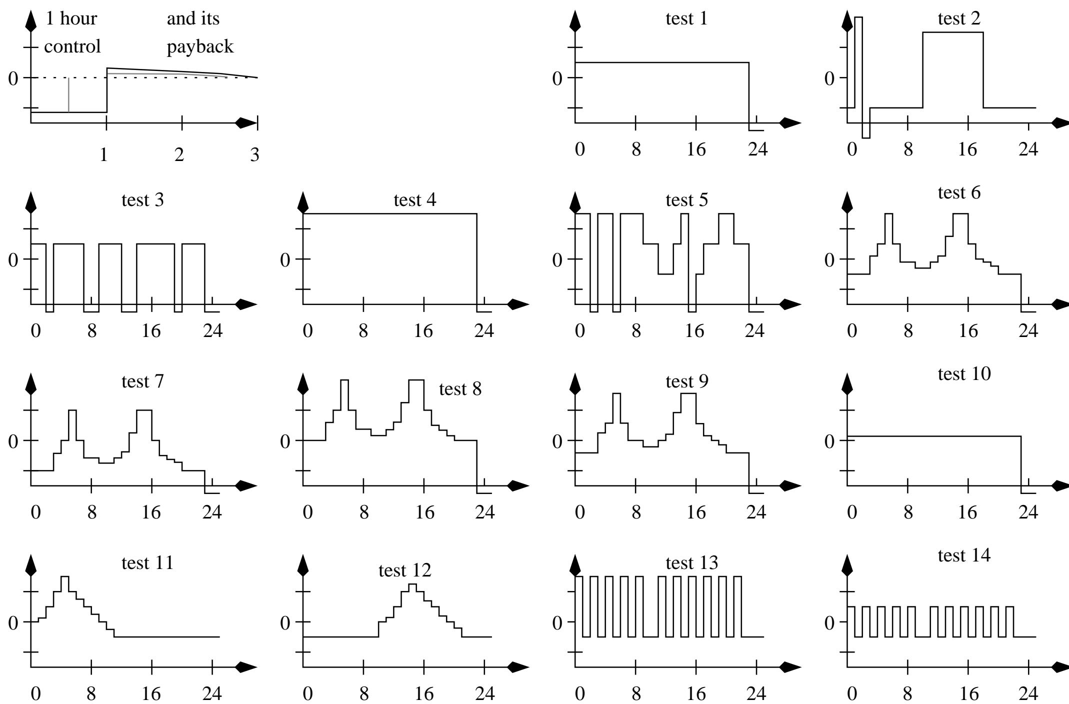

Figure 10.14: Radiation used in the tests and the test cases.

afternoon consumption peaks. The shapes were at different levels giving test cases of different difficulty. Test 11 contained only morning peak while test 12 contained afternoon peak. Other tests were artificial.

In the first test we could have been cut the overload from each overload hour exactly with the presented group with clones of length 5 (25 minutes controls), if there were no radiation. Radiation, however, affected the underload hours (improving the result at the same time) as well as the next hour to be cut (increasing the amount to be cut), because the radiation was two hours long. One MW price for overload is $-99\,000$ and revenue (underload) was -900. Starting values (total losses without clones) are given in Table 10.10.

Figure 10.15 contains running times (in seconds) for the old DP solution and for DPs with $c = 0, 1, 2, 3, 5$ and 11 (horizontal axis). Moreover, $b = 12 = h$. The running times increase almost linearly on the number of wait states.

Table 10.11 contains the results. If a solution is presented only for the old DP, other DPs with different values for c achieved the same result. The results improved when the number of wait states was increased. Note that the old DP has fixed number of wait states, while the new DPs number of wait states depends on time: it is $c + (a \bmod h)$, where a is the time point. The space usage for one wait state is determined by $h \cdot (\mathrm{cmax} + 2)$ for the old DP. The

Table 10.10: Starting values of the tests 1–14.

Test 1	Test 2	Test 3	Test 4	Test 5
−1 141 200	−1 402 200	−852 300	−3 418 200	−1 839 600
Test 6	Test 7	Test 8	Test 9	Test 10
−699 480	−341 100	−1 499 400	−797 310	−228 600
Test 11	Test 12	Test 13	Test 14	
−550 980	−550 980	−1 639 800	−550 800	

new DPs use more space in one hour, when $c \geq 5$.

In the seventh test, however, increasing the number of wait states decreased the result between DPs with $c = 3$ and $c = 5$. This somewhat nonintuitive result follows from the fact that our DP solutions do not fulfill the optimality principle usually stated for dynamic programming solutions, because of radiation and averaged hours (see [13, p. 16]).

The optimality principle is lost because we cannot guarantee that optimal solution at stage i (time point i) entails optimal solution for the rest of the time line. Reason for this is the radiation: it can affect later hours and decisions. This information should be available at the moment when we are deciding the length of a clone. Similarly, if we can first find the best clone, we cannot be sure that the second clone—even if optimal after the first one—gives optimal solution for the whole optimization problem.

In the seventh test, DP with $c = 5$ found in one crucial time point a better solution than DP with $c = 3$. It turned out that the locally better solution was worse for the rest of the optimization in this case. Larger number of additional states handled the situation correctly.

We did not use the alternative states in the test series reported in Table

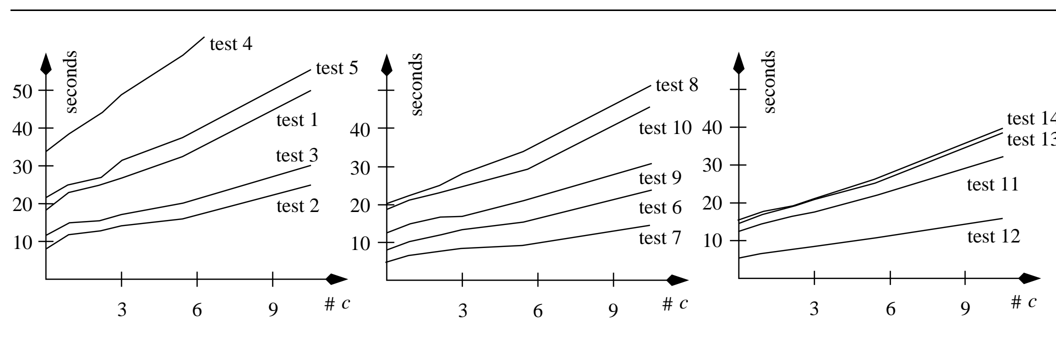

Figure 10.15: Running times.

Table 10.11: Solutions without alternative states (empty means the solution given by old DP).

Test	old DP	$c = 0$	$c = 1$	$c = 3$	$c = 5$	$c = 11$
1.	−3 091					
2.	−532 670					
3.	−11 288					
4.	−1 323 150					
5.	−483 773	−484 248	−484 248	−483 773	−483 773	−483 773
6.	−144 478					
7.	−15 603	−15 662	−15 662	**−15 662**	**−15 778**	−15 603
8.	−368 665	−371 296	−370 238	−368 665	−368 665	−368 665
9.	−174 490	−175 668	−175 668	−175 668	−175 668	−174 490
10.	−8 874	−9 131	−9 131	−8 990	−8 874	−8 874
11.	−68 059	−70 494	−69 657	−68 059	−68 059	−68 059
12.	−68 059	−70 494	−69 657	−68 059	−68 059	−68 059
13.	−366 669					
14.	−9 860					

Table 10.12: The best solutions with alternative states (empty means the solution given by old DP).

Test	old DP	$c = 0$	$c = 1$	$c = 3$	$c = 5$	$c = 11$
1.	−3 091	−3 091	−3 091	**−2 975**	**−2 975**	**−2 975**
2.	**−532 670**					
3.	−10 823	−10 823	−10 823	−10 823	−10 823	**−10 707**
4.	**−1 323 150**					
5.	−483 773	−483 773	−483 773	**−483 705**	**−483 705**	**−483 705**
6.	**−144 478**					
7.	−15 603	−15 662	−15 662	**−15 487**	**−15 487**	**−15 487**
8.	−368 373	−369 069	**−368 046**	−368 162	−368 278	−368 373
9.	−174 490	−175 668	−175 543	**−174 398**	**−174 398**	−174 490
10.	−8 410	**−7 365**	**−7 365**	−7 713	−8 202	−8 177
11.	**−67 827**	−69 242	−68 613	**−67 827**	**−67 827**	**−67 827**
12.	**−67 827**	−68 778	−68 613	**−67 827**	**−67 827**	**−67 827**
13.	**−366 669**					
14.	−8 956	−8 336	−8 351	−8 186	−7 742	**−7 538**

10.11 and Figure 10.15. We run the same tests using 2, 5, 10 and 15 alternative states. Table 10.12 contains the best solutions found among all the test series. Results are remarkable: results were improved in the most cases. Moreover, improvements were relatively high (much over 10% in some tests) and even the old DP solution was improved in some cases.

Test 14 demonstrates informatively the improvement as the number of wait states or alternative states (or both) increases. Test 14 has 0.5 MW overload in every other hour and the rest have the same amount of underload (in the middle, there are two underload hours, 9 to 11, see Figure 10.14). Results and running times of the test 14 are shown in Figures 10.16 and 10.17.

Starting solution (no clones) was $-550\,800$. DPs with one and two alternative states built up similar set of clones till stage 81 (that is, 6 hours 45 minutes after the beginning). The best result found so far gave $-355\,171$ with clones [1,6], [19,29], [43,53] and [67,77]. At the stage 82, DP with two alternative states found a set of clones giving $-355\,089.2$ with clones [1,5], [9,13], [23,31], [47,57], and [67,77]. By choosing the second best clone set ([1,5], [9,13], [23,31], and [47,57] with $-405\,807.7$) at the stage 66 we have found better set of clones than by using the locally best alternative ([1,6], [19,29], [43,53] with $-404\,157.9$).

The second best clone set at stage 66 cut overload more accurately (there were not so much underload than with the best clone set). It also incurred more radiation into the seventh hour so that the result was not the best (overload costs much more than underload). This increase caused by the radiation was less than the amount of clone being one moment longer, which, in turn, also caused the seventh hour to be cut more precisely with the second best path than with the best. By increasing the number of alternative states to 15, the first different stage was 56. Similarly to the previous case, the old clone set at phase 42 ([1,6] and [19,29]) was locally better than [5,9], [21,30] (at least 15th best) but after clone [43,53] at the stage 56 the worser set of clones at the stage 42 gave better result than the best clone set.

We also tried DPs with 30 and 100 alternative states. System with 30 states improved the result first time from 15 state system at stage 80 and system with 100 states improved the result first time from 30 state system at stage 61.

DP with 15 alternative states gave $-8\,336$, with 30 states $-7\,872$ and with 100 states $-7\,214$, which was better than the solution given by 15 alternative and 11 additional wait states (see Table 10.12). Our conclusion is that a clipping situation with many overload intervals most likely benefits from the use of alternative states.

We also studied with the same problem instance, how alternative states and

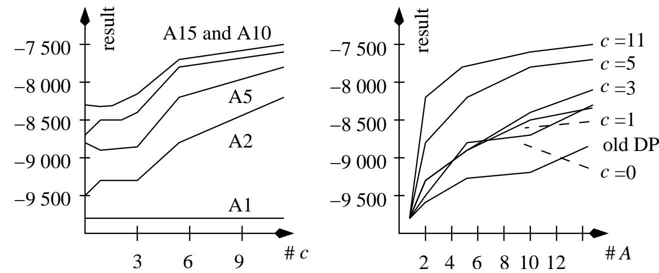

Figure 10.16: Wait and alternative states, results for test 14. $\#c$ is the number of additional wait states and $\#A$ is the number of alternative states.

wait states improve the results together and how they affect the running times. The left hand side of Figure 10.16 contains results for different alternative state amounts (1, 2, 5, 10 and 15). As the number of wait states is increased, the results improve in general. There are few exceptions, however. A few wait states may do worse than DP with $c = 0$ (see lines for A5 and A15). Most of the time 11 additional states to wait states gave better results than less wait states. By using only one alternative state (corresponds to a system, where no alternative state usage is implemented), the number of used additional wait states was irrelevant for this instance.

On the right side the same data is plotted for five different additional wait state amounts as well as for the old DP system. We conclude that the number of alternative states is much more crucial for the results than the number of wait states. Both state variables are needed, though. Alternative states also improve the results of DP system with fixed amount of wait states, which is the case in old DP. We did not try to find the best combination for the number of alternative and wait states as we wanted to keep the running times tolerable

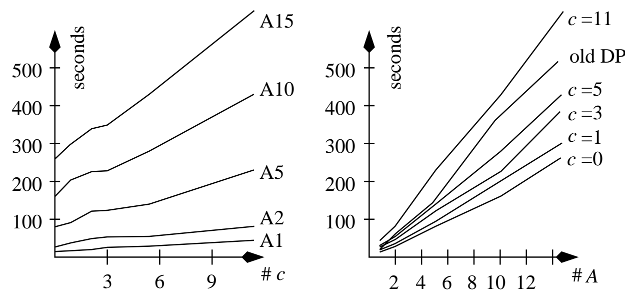

Figure 10.17: Wait and alternative states, running times for test 14. $\#c$ is the number of additional wait states and $\#A$ is the number of alternative states.

for the test runs.

Figure 10.17 contains running times for test 14. Execution decelerates almost linearly as the number of alternative states is increased. The same also holds for the number of wait states.

The number of wait states in one hour is $\sum_{a=0}^{h-1}\big(c+(a \bmod h)\big) = ch+h(h-1)/2$ (where $c = 0,\ldots,h-1$ and a is the moment). Hence, the increase of c by one gives $h-1$ additional wait states for one hour. This is proportionally less than the increase brought by the increase of the number of the alternative states by one, which is the number of wait states in one hour. This explains why the increase of the number of alternative states decelerates more the running times than the increase of the number of wait states.

10.5 The validity of hypotheses

The first hypothesis (H_1) seems to be valid. We are able to characterize the heuristics by suitable problem instances. The only exception is "Direct action" in these experiments. Although it is also possible to find instances were this method is the best, it is harder to characterize "Direct action" with instances. At least one of the heuristics obtained tolerable (near the ideal) result or results near the best. "Big first" gave the best result once (see Table 10.3). On the second difficult test the heuristics performed badly (see Table 10.6).

Hypothesis H_2 is valid. Most of the time, one second was enough for heuristics and in a couple of tests the heuristics needed more. The maximum time was three seconds which is in the bounds given, and which is much less than the time used by GAs and DPs. We have time to try all heuristics in the time consumed by GAs and DPs.

GAs do not seem to be clearly better than the heuristics in the experiments. Hence, we conclude that H_3 is not valid. Usually, at least one of the heuristics gave better solution than GAs. In some test, all heuristics performed better or at least as well. Only on the second difficult test GAs were better than the heuristics (see Table 10.6). The reason for bad behaviour maybe lies in the difficulties to maintain the "building block" hypothesis. Another reasons can be in the problem modeling by GAs (in the solution representation) and in the fine tuning of the solutions. Nevertheless, we think that by working out the above problems, it is possible to build GAs so that they outperform the presented heuristics.

The fourth hypothesis (H_4) seems to be valid. In most cases, 2000 generations gave the best results, although, in some easy instances the additional generations did not guarantee the better performance. Now, the results are

averages among 10 test runs. We could also do the test runs until we obtain an acceptable result and use the average number of test runs. This way, we would obtain some estimates, how many times GAs should be run (to be sure that a tolerable solution will be found).

The fifth hypothesis (H_5) is clearly valid. Usually, DPs gave the best or near the best results. In three tests, only two of the DPs were the best or near the best, see Tables 10.2, 10.3 and 10.5. The next hypothesis (H_6) is also valid which is shown by Table 10.12. By using alternative states, results were improved in ten out of fourteen tests.

Finally, the last hypothesis (H_7) seems to hold by test case 14. Larger number of alternative states also improved the performance in three other tests (tests 5, 9, and 10). In other test cases, the alternative states either did not have any impact or the improvement was given directly with two alternative states. We also noticed that while the additional alternative states improved the results, the number of wait states (c part) needed for the best solution varied as the number of alternative states changed.

Some of the above conclusions need further exploration if we are to fully reject or accept them: more test cases and test runs, and statistical tests should be applied. Especially, the current experiments are not extensive enough for hypotheses H_4 and H_7.

Chapter 11

Conclusion

We have developed the basic theory for interactive knapsacks including the model of interactive knapsacks, complexity results, and relations to some well-known NP-complete optimization problems. As a main application we introduced the load clipping problem used in electricity supply and management. IK model, the problems defined for it and their complexity properties, applications, and the presented methods with the experiments are new, although, most of the results have already been published in [1]–[6]. Additionally, a lot of corrections are presented in this work.

We have considered three decision problems, IKHD, IKD and MDIK and two optimization problems, IKHO and IKO. These have further variations: for example, the clones may have fixed lengths, varying or variable lengths. The model was represented with two interaction types: radiation and copying. In the larger load clipping application we presented a third interaction type.

The above five IK-problems were shown to be NP-complete, and IKD and MDIK (and IKO) to be strongly NP-complete. IKHO is $W[1]$- and APX-hard, and hence, so are the other problems. Moreover, IKHO was shown to equal 0–1 MDKP.

After these negative results, we turned our attention to the cases, which can be recognized in polynomial time. We have constructed polynomial time algorithms for special cases of IKHO, IKO, 0–1 MDKP (and 0–1 IP) and 0–n MDKP (and 0–n IP), n fixed, which have easily recognizable weight structure on the item weights. The structures have a size parameter that characterizes the running times of different instances. Eventually, we can even characterize the classes of instances, where the size parameter is not fixed. Similarly, the growth in n in the problems with several items is described as well.

We have left some problems open. For instance the running time analysis handling the sizes of instance parameters concerning individual weights,

profits, and knapsack sizes. In some applications this is not crucial and we may assume that parameters fit into a single memory word and can be read and written in constant time. Obviously, this assumption does not hold in all applications.

We have given several applications for our model. Some of the presented applications are a bit theoretical in nature, but they emphases the close relationship between different problems. A real life application, load clipping, is used by electricity suppliers. We have not presented nor used all the restrictions electricity suppliers use [3, 6].

Load clipping was solved with four heuristics, several versions of DPs and GAs. The results of our experiments show that the implementations are suitable for the task they were designed. Heuristics, DPs, and GAs can be further developed.

In DP, the properties of the state space have been analyzed, and quicker optimization algorithms are formed without sacrificing the accuracy of the results when payback is not used. Moreover, we have found practical ways to improve the results by increasing the state space when payback is used. We have presented detailed DP solutions with three and four state variables. Our DPs are sub-optimal. If the result accuracy is not crucial, one can drop wait states away, arriving to a faster two state variable solution of [29].

Next we list some fruitful research directions. Starting with the IK model, we have bypassed the stochastic interactive knapsacks. Many problems connected to interactive knapsacks are of stochastic nature, including the knapsack problem [54, 62], the multi-dimensional knapsack problem [40], scheduling [78, 88], and load clipping [20], among others. The load clipping problem considered in this work can be presented as stochastic optimization problem taking into account the unknown price and income factors as well as the uncertainty in the consumption forecast.

We also intend to study the properties of interactive knapsacks with respect to the other complexity classes presented in the literature. These include, for instance, the classes #P, Opt P, Max P, Mid P, Gap-P, Med P and Max FSLIP (for definitions and further directions for listed classes, see [101] and [10]). Another theoretical direction is to study, if it is possible to verify locally optimal solutions [63] or to test optimality [19].

A question related to load clipping application is the variable clone length. Some of the controllable utilities have minimum and maximum control lengths and the optimization method should determine the number of controls (clones), and their starting times and lengths. The problems (4.16), (4.12)–(4.15) and (4.22), (4.18)–(4.21) can be easily modified to allow the decision of the clone

length. The proof of Theorem 5.2 describes a special case, where the utilization of the variable clone length is easy and efficient.

We leave it open to incorporate the variable clone lengths to the other methods presented in Chapter 5. Especially, we would like either to redesign the method given in the proof of Lemma 5.7, or to design a completely new method. If neither of the approaches works, we could establish some hardness results showing the inefficiency of the case handled in Theorem 5.9.

Another line of possible improvements is connected to the clone length: in the method of Theorem 5.2 we are able to jump several knapsack at a time while the other methods do not allow that. The reason is radiation. Even though we do not use radiation in Theorem 5.12, the method does not use long jumps. Further, it may be possible to replace the method behind Theorem 5.9 so that longer jumps based on the clone length are possible and at the same time the radiation is handled correctly.

We can also pose open some other natural questions related to the methods presented in Chapter 5. Now the weight matrices have special structure that contains shape (band matrix, length of radiation and clone). The band matrix restrictions can be easily generalized to use the left and right radiation as lower and upper diagonals, but we would like to establish that the shape is not essential as long as there are not too many knapsacks. We also know that the shape cannot be relaxed too much, since for instance two variables per inequality are enough to make the problem NP-complete [55].

0–n MDKP in Section 5.5 uses the same bounds for each decision variable. To enhance the usability of the presented approach, we would like to have separate ranges for each decision variable.

The presented methods (of Chapter 5) may work well with some other models and methods. For example, difference constraints of the form $x_j - x_i \leq b_k$, can solve problems containing relative timing constraints [30] (in polynomial time by using shortest path methods, see also [43]). In a way, we can interpret difference constrains as giving a minimum and maximum lengths for controls (minimum and maximum c of a clone), which are pairs of consecutive variables: x_1 gives end time and x_2 the start time. We can also state that two consecutive clones (controls) cannot overlap by making restrictions $x_3 - x_2 \geq 0$.

If we want to use some objective function, the application of difference constraints is not straightforward. Moreover, the use of some extra constraints (knapsack capacities, for example) is not easy to model with difference constraints. It would be interesting to see, if one can efficiently add difference constraints to the presented methods, and thus broaden the class of problems

that can be solved in polynomial time.

By choosing appropriate interaction functions I, profits p, weights w and knapsack capacities b, we can imitate some scheduling problems, like "sequencing with release times and deadlines on one processor" or "single machine scheduling" (see Chapter 6 and Section 7.2). Can we utilize the methods presented in this paper to those problems? And of course, we want to find out other problems, in which the presented methods work well.

Contrary to the above line of thought, we may also ask, how general is the applied approach of coding restrictions into the vertex sets and then search the path. It may be directly applicable to a large class of problems without using the IK model or MDKP in between. We may try to obtain restriction matrices with desired properties, or to code restrictions directly to a similar graph we used.

Methods presented in Chapter 5 may be used in the design of heuristics. There are at least two approaches: we may try to reduce either the size of the vertex sets or the number of sets (and somehow ensure that the we comply with the problem restrictions). Another approach is connected to the applications: we may decrease their solution space somehow and then apply the presented methods to give exact solutions for the reduced instance.

Sometimes the application may imply a natural way to reduce the number of vertex sets. For example, in load clipping there may be time intervals, when there is no need to optimize. Even though we can recognize those intervals easily in load clipping, we have to be prepared to optimize the whole time interval. The running time analyses can be utilized here. If the actual optimization interval can be reduced enough, we may use the method presented for IKHO. Otherwise we may use larger set of controlling utilities (that is, use larger n in IKO). The decision is dynamic and cannot be tested beforehand, as opposite to comparing the running times beforehand to help the choice of an appropriate method.

The shortest resource bounded methods we utilized to achieve the given results are also interesting. The use of shortest paths is a much studied subject and we have efficient solutions for the directed acyclic graphs. Since our graph constructions are highly symmetric, it may be possible to find out the shortest paths more efficiently. Furthermore, we can save space by updating the paths at the same time as constructing graphs so that only a vertex set is kept in memory. It may give faster solutions (which is the case, for example, in the discussion preceding Corollary 5.16). Other possibility is to form paths depth first (see, for example, [13, pp. 62–81]) and apply, say, branch and bound.

Some special cases may impose efficient algorithms for IK problems IKHO

and IKO, or for load clipping, like convex or concave profits p_{ij}. Orlin [83] study knapsack problems with convex and concave profits which might give some directions, although the IK problems are quite different from the plain knapsack problems.

Load clipping also raises some possible research directions. When we made comparisons, state variable k (control count) seemed to behave "softly enough", so that we can reduce the number of states used. For example, if we have just began our optimization or if we are near the end (the current stage number is near zero or m), we do not need every state in a stage. Moreover, the control length c may have some properties, by which we can further speed up the algorithms.

The fine tuning problem of GA can hopefully be solved by combining tabu search techniques into the GA. Also different codings should be tested: for example

"00001000020000000100020000"

could describe the first (1) and the last (2) knapsacks corresponding to a control (or to a clone). Now a crossover is allowed between 2's and 1's but not between 1's and 2's. This would prevent us breaking the clones (controls) with crossover operations and would at least partly maintain the "building blocks". A swapping of two near positions may work as a mutation. We also need a way to easily move a tail one position forward or backward. That could also be used as a rare mutation type.

We also need a way to add or remove clones (controls). Alternatively, we could choose to maintain in the population some sets of individuals containing a fixed number of clones (controls). These subsets in the population would not be mixed but they rather should operate with the corresponding subsets in the other individuals.

Dynamic knapsack problems are considered in [62]. IK problems, especially when applied to load clipping, have very dynamic nature: we first optimize for one time setting and then after a while, we have to optimize again. GAs presented in this work are well suited for this kind of situation. In load clipping the optimization is repeated every hour and the instances do not differ much between different hours. By forming a couple of nearby instances from the currently solved instance we have a situation, where at least one solution is near to the new instance. Now GA has a partial answer and it needs less work in order to find the solution. We should study, how well other methods can take into account the partial solutions in dynamic situations.

Appendix A

Referenced problems

A.1 Single machine scheduling

Ibaraki and Nakamura [59] give the single machine scheduling problem in the following way. The problem is to determine the optimal sequence of n jobs in set $N = \{1, \ldots, n\}$ without idle time on a single machine. Each job becomes available at time 0, requires integer processing time p_i and incurs a cost $g_i(C_i)$ if completed at time C_i.

The objective is to minimize

$$\sum_{i \in N} g_i(C_i),$$

the cost of the sequence of n jobs. Cost $g_i(\cdot)$ can be nondecreasing, for example, the weighted sum of tardiness or the weighted sum of earliness and tardiness,

$$g_i(C_i) = h_i \max\{d_i - C_i, 0\} + w_i \max\{C_i - d_i, 0\},$$

where $d_i \in \mathbb{Z}_+$ is the due date of job i and $h_i, w_i \in \mathbb{Z}_+$, are the weights of earliness and tardiness of job i.

A.2 Multi-period variable-task-duration assignment problem

We state the multi-period assignment problem using the notations of Miller and Franz [75] and follow closely their presentation of the problem. They consider a multi-period assignment problem in which N employees are to be assigned to one of M tasks during each of T periods. Variable $X_{ijk} = 1$ means that employee i ($i = 1, \dots, N$) is assigned to a task j ($j = 1, \dots, M$) during period k ($k = 1, \dots, T$). Otherwise $X_{ijk} = 0$.

The objectives can be to

$$\min \sum \sum \sum C_{ijk} X_{ijk}$$

or to

$$\max \sum \sum \sum C_{ijk} X_{ijk},$$

where C_{ijk} is the cost or benefit associated with each assignment. Tasks are covered by necessary manpower by constraints

$$\sum_i X_{ijk} \lesseqqgtr r_{jk} \qquad \text{for all } j \text{ and } k,$$

where r_{jk} is the number of employees required for each task j during period k. Each employee is assigned to on task during each time period

$$\sum_j X_{ijk} = 1 \qquad \text{for all } i \text{ and } k.$$

Employees serve also the required number of periods for all tasks

$$\sum_j X_{ijk} = r_{ij} \qquad \text{for all } i \text{ and } j,$$

where r_{ij} is the number of periods of task j required by employee i. Other constraints might include individuals who must be assigned to a specific task simultaneously, assigning multi-period tasks to contiguous time periods, or specifying prerequisite relationships between tasks

$$X_{ijk} = X_{i'j'k'}.$$

A.3 Multiple translational containment

Milenkovic [74] studies the multiple translational containment, a NP-hard layout problem and uses the following symbols:

- C is for a polygonal "knapsack" with n vertices,
- P_i, $1 \leq i \leq k$, is a polygon with m_i vertices to be placed in the knapsack,
- V_i, $1 \leq i \leq k$, is the set of translations of P_i that places it in the interior of C,
- U_{ij}, $1 \leq i \leq k$, is the set of displacements between P_i and P_j such that they do not overlap each other ($U_{ij} = -U_{ji}$).

A *valid configuration* for $P_1, P_2, \ldots, P_k$ inside C is a list of translations $\langle t_1, \ldots, t_k \rangle$ such that

$$t_i \in V_i, \qquad 1 \leq i \leq k, \qquad \text{and} \qquad t_j - t_i \in U_{ij}, \qquad 1 \leq i < j \leq k.$$

The kNN *problem* is to find a valid configuration for k nonconvex polygons inside a nonconvex knapsack and kCN for k convex polygons inside a nonconvex knapsack. The (r, k)NN *problem* is to find all subsets of size k out of a set of r nonconvex polygons such that the k polygons have a valid configuration inside a given nonconvex knapsack and (r, k)CN is defined similarly.

A.4 Some known NP-complete problems

In this appendix we list the NP-complete problems used in this work. Most of the problems are taken directly from [45].

PARTITION
Instance: A finite set N and a size $s(n) \in \mathbb{Z}^+$ for each $n \in N$.
Question: Is there a subset $N' \subseteq N$ such that

$$\sum_{n \in N'} s(n) = \sum_{n \in N \setminus N'} s(n)?$$

SUBSET SUM
Instance: A finite set N, a size $s(n) \in \mathbb{Z}^+$ for each $n \in N$ and a positive integer B.
Question: Is there a subset $N' \subseteq N$ such that the sum of the sizes of the elements in N' is exactly B?

SIZED SUBSET SUM
Instance: Like in SUBSET SUM, and a positive integer k.
Question: Is there a subset $N' \subseteq N$ of size k such that the sum of the sizes of the elements in N' is exactly B?

KNAPSACK
Instance: A finite set N, a size $s(n) \in \mathbb{Z}^+$ and a profit $p(n) \in \mathbb{Z}^+$ for each $n \in N$, a size constraint $B \in \mathbb{Z}^+$, and a profit goal $K \in \mathbb{Z}^+$.
Question: Is there a subset $N' \subseteq N$ such that

$$\sum_{n \in N'} s(n) \leq B \qquad \text{and} \qquad \sum_{n \in N'} p(n) \geq K?$$

0–1 MDKP
Instance: Finite set X of pairs (x, b), where x is an m-tuple of nonnegative integers and b is a nonnegative integer, an m-tuple c of nonnegative integers, and a nonnegative integer B.
Question: Is there an m-tuple y of nonnegative integers such that $xy \leq b$ for all $(x, b) \in X$ and such that $cy \geq B$?

We define similarly 0–n MDKP, 0–1 IP, 0–n IP, and GAP.

LONGEST PATH

Instance: Graph $G = (V, E)$, length $l(e) \in \mathbb{Z}^+$ for each $e \in E$, positive integer K, specified vertices $s, t \in V$.

Question: Is there a simple path in G from s to t of length K or more, i.e., whose edge lengths sum to at least K?

SHORTEST WEIGHT-CONSTRAINED PATH

Instance: Graph $G = (V, E)$, length $l(e) \in \mathbb{Z}^+$, and weight $w(e) \in \mathbb{Z}^+$ for each $e \in E$, specified vertices $s, t \in V$, positive integers K, W.

Question: Is there a simple path in G from s to t with total weight W or less and total length K or less?

In the case of LONGEST WEIGHT-CONSTRAINED PATH, we are to find a path with total length K or more. In the k-resource bounded problem we have a k-tuple of weights $w(e)$ and a k-tuple W.

HAMILTONIAN PATH

Instance: Graph $G = (V, E)$.

Question: Does G contain a Hamiltonian path?

SEQUENCING WITH RELEASE TIMES AND DEADLINES

Instance: Set T of tasks and, for each task $t \in T$, a length $l(t) \in \mathbb{Z}^+$, a release time $r(t) \in \mathbb{Z}_0^+$, and a deadline $d(t) \in \mathbb{Z}^+$.

Question: Is there a one-processor schedule for T that satisfies the release time constraints and meets all the deadlines, i.e., a one-to-one function $\sigma : T \to \mathbb{Z}_0^+$, with $\sigma(t) > \sigma(t')$ implying $\sigma(t) \geq \sigma(t') + l(t')$, such that, for all $t \in T$, $\sigma(t) \geq r(t)$ and $\sigma(t) + l(t) \leq d(t)$?

MULTIPROCESSOR SCHEDULING

Instance: Set T of tasks, number $m \in \mathbb{Z}^+$ of processors, length $l(t) \in \mathbb{Z}^+$ for each $t \in T$, and a deadline $D \in \mathbb{Z}^+$.

Question: Is there an m-processor schedule for T that meets the overall deadline D, i.e., a function $\sigma : T \to \mathbb{Z}_0^+$ such that, for all $u \geq 0$, the number of tasks $t \in T$ for which $\sigma(t) \leq u < \sigma(t) + l(t)$ is no more than m and such that, for all $t \in T$, $\sigma(t) + l(t) \leq D$?

JOB-SHOP SCHEDULING

Instance: Number $m \in \mathbb{Z}^+$ of processors, set J of jobs, each $j \in J$ consisting of an ordered collection of tasks $t_k[j]$, $1 \leq k \leq n_j$, for each such task t a length $l(t) \in \mathbb{Z}_0^+$ and a processor $p(t) \in \{1, 2, \ldots, m\}$, where $p(t_k[j]) \neq p(t_{k+1}[j])$ for all $j \in J$ and $1 \leq k < n_j$, and a deadline $D \in \mathbb{Z}^+$.

Question: Is there a job-shop schedule for J that meets the overall deadline, i.e., a collection of one-processor schedules σ_i mapping $\{t : p(t) = i\}$ into $\mathbb{Z}_0^+$, $1 \leq i \leq m$, such that $\sigma_i(t) > \sigma_i(t')$ implies $\sigma_i(t) \geq \sigma_i(t') + l(t)$, such that $\sigma(t_{k+1}[j]) \geq \sigma(t_k[j]) + l(t_k[j])$ (where the appropriate subscripts are to be assumed on σ) for all $j \in J$ and $1 \leq k < n_j$, and such that, for all $j \in J$, $\sigma(t_{n_j}[j]) + l(t_{n_j}[j]) \leq D$ (again assuming the appropriate subscript on σ)?

Appendix B

IK problem repository

We list a number of problems for IK model. Most of the present problems can be motivated by load clipping. The presented list is demonstrates, what kinds of problems can be easily presented with IK model. We start from simple problems and add features to them. Some of the problems are artificial, while most of them can be motivated by load clipping.

In each following optimization problem we have $x_i, x_{ij} \in \{0,1\}$, profits p_i, p_{ij}, weights w_i, w_{ij}, lengths of clones c, c_i, c_{ij}, lengths of radiations u, u_i, u_{ij} and functions $I_i, I_{ij} : \{1,\ldots,m\} \to \mathbb{Q}$, for knapsacks $i = 1,\ldots,m$ and items $j = 1,\ldots,n$, with capacities b_i, and a positive integers K, K_j.

B.1 Fixed clone and radiation lengths

IKHO (interactive knapsack heuristic optimization problem). Our aim is to

$$\begin{aligned}
\max \quad & \sum_{i=1}^{m} x_i \sum_{k=i-u}^{i+c+u} I_i(k) p_k, & \\
\text{subject to} \quad & \sum_{i=1}^{m} x_i w_i I_i(l) \le b_l, & l = 1,\ldots,m, \\
& x_k = 0, \text{ for } i < k \le i+c, \text{ when } x_i = 1, & i = 1,\ldots,m, \\
& x_i = 0 \text{ or } 1, & i = 1,\ldots,m, \\
& \sum_{i=1}^{m} x_i \le K. &
\end{aligned}$$

IKO (interactive knapsack optimization problem). Our aim is to

$$\begin{array}{lll}
\max & \sum_{i=1}^{m}\sum_{j=1}^{n} x_{ij} \sum_{k=i-u}^{i+c+u} I_{ij}(k)p_{kj} & \\
\text{subject to} & \sum_{i=1}^{m}\sum_{j=1}^{n} x_{ij}w_{ij}I_{ij}(l) \leq b_l, & l = 1,\dots,m, \\
& x_{kj} = 0, \text{ for } i < k \leq i + c_j, \text{ when } x_{ij} = 1, & i = 1,\dots,m, \quad j = 1,\dots,n, \\
& x_{ij} = 0 \text{ or } 1, & i = 1,\dots,m, \quad j = 1,\dots,n, \\
& \sum_{i=1}^{m} x_{ij} \leq K_j, & j = 1,\dots,n.
\end{array}$$

MDIKO (interactive multi-dimensional knapsack optimization problem). We denote $J_j = \{1, \dots, m_j\}$ for $j = 1, \dots, q$ and $J = J_1 \times \dots \times J_q$. Moreover, $I_{ij} = C \cup R$ for all i and j, and not $C_{ij} \cup R_{ij}$ as for variable case. Our aim is to

$$\begin{array}{lll}
\max & \sum_{i\in J}\sum_{j=1}^{n} x_{ij} \sum_{k\in I_{ij}} I_{ij}(k)p_{kj} & \\
\text{subject to} & \sum_{i=1}^{m}\sum_{j=1}^{n} x_{ij}w_{ij}I_{ij}(l) \leq b_l, & l \in J, \\
& x_{ij} = 0 \text{ or } 1, & i \in J, \quad j = 1,\dots,n, \\
& \sum_{i\in J} x_{ij} \leq K_j, & j = 1,\dots,n.
\end{array}$$

AVKHO (interactive added value knapsack heuristic optimization problem). Let $S_k = \sum_{i=1}^{m} x_i w_i I_i(k)$. Our aim is to

$$\begin{array}{lll}
\max & \sum_{i=1}^{m} x_i \sum_{k=i-u}^{i+c+u} I_i(k)p_k(S_k), & \\
\text{subject to} & S_i \leq b_i, & i = 1,\dots,m, \\
& x_k = 0, \text{ for } i < k \leq i + c, \text{ when } x_i = 1, & i = 1,\dots,m, \\
& x_i = 0 \text{ or } 1, & i = 1,\dots,m, \\
& \sum_{i=1}^{m} x_i \leq K. &
\end{array}$$

AVKO (interactive added value knapsack optimization problem). Let $S_k = \sum_{i=1}^{m}\sum_{j=1}^{n} x_{ij}w_{ij}I_{ij}(k)$. Our aim is to

$$\begin{aligned}
\max \quad & \sum_{i=1}^{m}\sum_{j=1}^{n} x_{ij} \sum_{k=i-u}^{i+c+u} I_{ij}(k)p_{kj}(S_k) && \\
\text{subject to} \quad & S_i \le b_i, && i = 1,\dots,m, \\
& x_{kj} = 0, \text{ for } i < k \le i+c, \text{ when } x_{ij} = 1, && i = 1,\dots,m, \quad j = 1,\dots,n, \\
& x_{ij} = 0 \text{ or } 1, && i = 1,\dots,m, \quad j = 1,\dots,n, \\
& \sum_{i=1}^{m} x_{ij} \le K_j, && j = 1,\dots,n.
\end{aligned}$$

B.2 Clone lengths depending on item and knapsack

The lengths (forms) of clones and radiation can depend on three different entities: item, knapsack or both. These problems are similar to the ones defined in the previous section. We still use x's as our decision variables. In some cases we saw that these problems are as easy as the fixed clone length variants.

IKHO In IKHO with clone lengths depending on item and knapsack our aim is to

$$\begin{aligned}
\max \quad & \sum_{i=1}^{m} x_i \sum_{k=i-u_i}^{i+c_i+u_i} I_i(k)p_k, && \\
\text{subject to} \quad & \sum_{i=1}^{m} x_i w_i I_i(l) \le b_l, && l = 1,\dots,m, \\
& x_k = 0, \text{ for } i < k \le i+c_i, \text{ when } x_i = 1, && i = 1,\dots,m, \\
& x_i = 0 \text{ or } 1, && i = 1,\dots,m, \\
& \sum_{i=1}^{m} x_i \le K. &&
\end{aligned}$$

IKO In IKO with clone lengths depending on item and knapsack our aim is to

$$\begin{aligned}
\max \quad & \sum_{i=1}^{m}\sum_{j=1}^{n} x_{ij} \sum_{k=i-u_{ij}}^{i+c_{ij}+u_{ij}} I_{ij}(k)p_{kj} && \\
\text{subject to} \quad & \sum_{i=1}^{m}\sum_{j=1}^{n} x_{ij}w_{ij}I_{ij}(l) \le b_l, && l = 1,\dots,m, \\
& x_{kj} = 0, \text{ for } i < k \le i+c_{ij}, \text{ when } x_{ij} = 1, && i = 1,\dots,m, \quad j = 1,\dots,n, \\
& x_{ij} = 0 \text{ or } 1, && i = 1,\dots,m, \quad j = 1,\dots,n, \\
& \sum_{i=1}^{m} x_{ij} \le K_j, && j = 1,\dots,n.
\end{aligned}$$

B.3 Variable clone lengths

Using variable c's the problems change more than between the fixed and dependent cases. In applications variable c's are more common but variable u's also occurs as well as variable $I_{ij}(\cdot)$'s. We have several ways to implement variable clone lengths, of which we present one: we focus only on variable c and radiation u depending on clone length c, knapsack and item.

We have added more constraints as they can be naturally introduced to the problems in question; that is, constraints setting minimum and maximum lengths for c. We also need constraints linking c and x, and the decided length for interaction function.

IKHO In IKHO with variable clone lengths our aim is to

$$\begin{aligned}
\max \quad & \sum_{i=1}^{m} x_i \sum_{k=i-u_i(c_i)}^{i+c_i+u_i(c_i)} I_i(k, c_i) p_k, \\
\text{subject to} \quad & \sum_{i=1}^{m} x_i w_i I_i(l, c_i) \leq b_l, && l = 1, \ldots, m, \\
& \text{cmin} \leq c_i \leq \text{cmax}, && i = 1, \ldots, m, \\
& x_k = 0, \text{ for } i < k \leq i + c_i, \text{ when } x_i = 1, && i = 1, \ldots, m, \\
& x_i = \begin{cases} 1, & \text{if } c_i \geq 0, \\ 0, & \text{if } c_i = -1, \end{cases} && i = 1, \ldots, m, \\
& \sum_{i=1}^{m} x_i \leq K.
\end{aligned}$$

IKO In IKO with variable clone lengths our aim is to

$$\begin{aligned}
\max \quad & \sum_{i=1}^{m} \sum_{j=1}^{n} x_{ij} \sum_{k=i-u_{ij}(c_{ij})}^{i+c_{ij}+u_{ij}(c_{ij})} I_{ij}(k, c_{ij}) p_{kj} \\
\text{subject to} \quad & \sum_{i,j} \in I_{ij} x_{ij} w_{ij} I_{ij}(l, c_{ij}) \leq b_l, && l = 1, \ldots, m, \\
& \text{cmin}_j \leq c_{ij} \leq \text{cmax}_j, && i = 1, \ldots, m, \quad j = 1, \ldots, n, \\
& x_{kj} = 0, \text{ for } i < k \leq i + c_{ij}, \text{ when } x_{ij} = 1, && i = 1, \ldots, m, \quad j = 1, \ldots, n, \\
& x_i = \begin{cases} 1, & \text{if } c_{ij} \geq 0, \\ 0, & \text{if } c_{ij} = -1, \end{cases} && i = 1, \ldots, m, \quad j = 1, \ldots, n, \\
& \sum_{i=1}^{m} x_{ij} \leq K_j && j = 1, \ldots, n.
\end{aligned}$$

Appendix C

Material for load clipping

C.1 Web resources on energy management

General descriptions about energy and demand side management can be found, for example, through the following www-pages:

- http://dsm.iea.org
- http://www.eia.doe.gov/cneaf/electricity/dsm/dsm_sum.html
- http://www.eren.doe.gov/EE/power_dsm.html
- http://www.peaklma.com
- http://www.transmission.bpa.gov/orgs/opi/Power_Stability/index.shtm.

The last link in the above list contains many links to direct load control. The glossaries of terms can be found, for example, in the following www-pages:

- http://www.eia.doe.gov/cneaf/electricity/dsm/dsm_gloss.html
- http://www.lanl.gov/projects/cctc/resources/library/glossary/glossary.html.

C.2 Data for the first test set

In the load clipping tables (for instance in Table C.1) positive numbers indicate under fillings (overload) and the negative ones over fillings (underload). Recall that each hour consist of twelve knapsacks and these numbers are averaged in each hour.

In Tables C.2, C.5, and C.8, sa means safety area and sa 2 means the safety area per cent of the clone length. Cum. m means cumulative minutes, that is, how much we are allowed to put clones into the knapsack array.

If K is not specified (∞ in the table), we obtain K by $K = (\text{cum. m})/\text{cmin}$ for DPs. Heuristics and GAs work well without K while DPs cannot easily take it into account. This can lead to a nonfeasible solution, if the instance is hard, because we can insert K clones of maximum length, thus exceeding the cumulative upper bound. Radiations of the items are given in Figure C.1.

In the following, we first list the clipping situation, then the used items, and last, the best result, for each instance.

Table C.1: The first instance (starting result −60670, ideal −960).

0.500	–0.530	–0.330	–0.160	0.460	1.040	2.380	1.890	0.640
	–0.130	0.100	–0.400	–0.430	–0.310	–0.080	–1.670	0.470
	0.390	0.470	–0.310	–0.170	–0.040	–0.150	0.190	–14.500

Table C.2: Items used in the first test.

Item	Priority	w	cmin	cmax	sa	sa2	cum. m	K
1.	1	0.8	6	120	2	10%	∞	∞
2.	2	0.972	6	36	12	50%	120	∞
3.	3	0.75	4	6	3	60%	60	∞
4.	3	0.75	5	9	4	50%	60	∞
5.	3	4.7	6	12	6	60%	60	∞
6.	4	7	6	12	4	40%	40	∞
7.	4	0.35	6	12	6	50%	60	∞
8.	5	0.5	6	36	6	10%	72	∞

Table C.3: The best solution for the first test.

–0.283	–0.274	–0.055	–0.085	–0.101	–0.320	–0.328	–0.168	–0.278
	–0.032	–0.888	–0.051	–0.049	–0.046	–0.058	–0.105	–0.086
	–0.082	–0.006	–0.036	–0.123	–0.194	–0.141	–9.959	–9.219

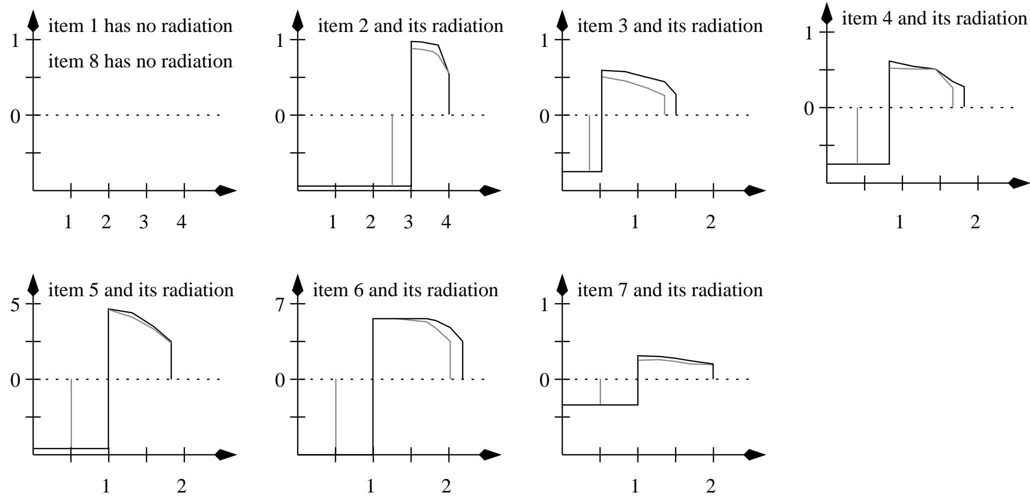

Figure C.1: The radiation patterns of items. Ticks are hours in horizontal axis and MWs in vertical axis. Black line describes a maximum length clone and grey line a minimum length clone. Both clones end in the same moment.

Table C.4: The second instance (starting result −96400, ideal −1200).

–1.000	2.000	2.000	–1.000	–1.000	–1.000	–1.000	–1.000	–1.000
	–1.000	1.200	1.200	1.200	1.200	1.200	1.200	1.200
	1.200	–1.000	–1.000	–1.000	–1.000	–1.000	–1.000	–10.000

Table C.5: Items used in the second test. Radiations are like for the similar items in the first instance.

Item	Priority	w	cmin	cmax	sa	sa2	cum. m	K
1.	1	0.8	6	120	2	10%	∞	1
2.	2	0.972	6	36	12	50%	120	1
3.	3	0.75	5	9	4	50%	60	5
4.	3	4.7	6	12	6	60%	60	2
5.	5	0.5	6	36	6	10%	72	2

Table C.6: The best solution for the second test.

–1.000	–0.019	–0.045	–0.002	–0.496	–1.000	–1.000	–1.000	–1.000
	–1.000	–0.017	–0.005	–0.011	–0.073	–0.572	–0.103	–0.176
	–0.012	–0.019	–1.000	–1.000	–1.000	–1.000	–1.000	–10.000

Table C.7: The third instance (starting result −99350, ideal −1350).

–1.000	2.000	–3.000	–1.000	–1.000	–1.000	–1.000	–1.000	–1.000
	–1.000	1.500	1.500	1.500	1.500	1.500	1.500	1.500
	1.500	–1.000	–1.000	–1.000	–1.000	–1.000	–1.000	–10.000

Table C.8: Items used in the third test. Radiations are like for the similar items in the first instance.

Item	Priority	w	cmin	cmax	sa	sa2	cum. m	K
1.	1	0.8	6	120	2	10%	∞	1
2.	2	1.8	6	36	12	50%	120	1
3.	3	4.7	6	12	6	60%	60	1

Table C.9: The best solution for the third test.

–1.000	–0.350	–1.261	–1.000	–1.000	–1.000	–1.000	–1.000	–1.000
	–1.000	0.000	–0.300	–0.300	0.244	0.591	0.700	0.700
	0.700	–1.000	–1.000	–1.000	–1.000	–1.000	–1.000	–10.000

C.3 Data for the hard instances

Both hard instances use the items given in Table C.2.

Table C.10: The first hard instance (starting result −324600, ideal −500).

1.500	1.000	1.500	2.000	2.500	3.000	3.000	3.500	2.400
	2.000	1.600	1.400	1.000	1.200	2.200	2.000	2.500
	3.000	2.500	2.000	1.000	1.500	1.200	0.800	–10.000

Table C.11: The best solution for the first difficult test.

–0.055	–0.179	–0.024	–0.221	–0.013	–0.026	–0.039	–0.006	–0.020
	–0.012	–0.015	–0.037	–0.015	–0.064	–0.037	–0.008	–0.109
	–0.023	–0.009	–0.003	–0.004	–0.045	–1.420	–0.014	–2.962

Table C.12: The second hard instance (starting result −5040500, ideal −500).

30.000	30.000	30.000	30.000	30.000	30.000	30.000	30.000	30.000
	30.000	30.000	30.000	30.000	30.000	30.000	30.000	30.000
	30.000	30.000	30.000	30.000	30.000	30.000	30.000	–10.000

Table C.13: The best solution for the second difficult test.

15.300	29.300	26.900	26.500	26.800	26.200	27.300	28.200	29.500
	25.400	28.900	27.500	25.900	29.900	29.900	21.900	32.800
	21.600	32.400	27.000	28.500	26.700	28.200	15.900	0.040

C.4 Data for normal instances

All instances given in this section use the items given in Table C.2.

Table C.14: Morning peak (starting result −36300, ideal −1300).

–1.000	–1.000	0.500	0.500	1.000	1.500	1.000	0.500	0.000
	0.000	–1.000	–1.000	–1.000	–1.000	–1.000	–1.000	–1.000
	–1.000	–1.000	–1.000	–1.000	–1.000	–1.000	–1.000	–10.000

Table C.15: The best solution for the morning peak.

–1.162	–0.780	–0.059	–0.205	–0.013	–0.075	–0.236	–0.043	–0.051
	–0.056	–0.449	–0.545	–0.040	–1.473	–0.688	–1.358	–0.078
	–0.525	–0.665	–0.059	–1.107	–1.618	–0.803	–1.981	–3.856

Table C.16: Afternoon peak (starting result −58000, ideal −1300).

–1.000	–1.000	–1.000	–1.000	–1.000	–1.000	–1.000	–1.000	–1.000
	–1.000	–1.000	–1.000	–1.000	–1.000	–1.000	0.500	0.500
	1.500	2.000	2.000	1.000	0.500	0.100	–1.000	–10.000

Table C.17: The best solution for the afternoon peak.

–1.516	–0.704	–0.039	–0.449	–0.277	–0.059	–0.592	–0.088	–0.529
	–0.399	–0.087	–0.399	–0.983	–0.179	–0.025	–0.033	–0.033
	–0.023	–0.001	–0.003	–0.012	–0.014	–0.309	–1.545	–3.223

Table C.18: Morning and afternoon peak (starting result −85050, ideal −1100).

–1.000	–1.000	–1.000	0.500	1.000	1.500	1.000	0.500	0.000
	–0.500	–1.000	–1.000	–1.000	–1.000	–1.000	–0.500	0.500
	1.500	2.000	2.000	1.000	0.500	–1.000	–1.000	–10.000

Table C.19: The best solution for the morning and afternoon peaks.

–1.805	–0.389	–0.025	–0.033	–0.065	–0.033	–0.004	–0.001	–0.030
	–0.320	–0.445	–0.563	–0.144	–0.180	–0.009	–0.011	–0.033
	–0.031	–0.001	–0.008	–0.004	–0.549	–0.066	–0.744	–1.846

Bibliography

[1] Isto Aho. Interactive knapsacks. Licentiate's thesis. Technical Report A-1999-5, University of Tampere, Department of Computer and Information Sciences, March 1999. [Online]. Available: http://www.cs.uta.fi/reports/r1999.html.

[2] Isto Aho. Interactive knapsacks. *Fundamenta Informaticae*, 44(1–2):1–23, September 2000.

[3] Isto Aho. Notes on the properties of dynamic programming used in direct load control scheduling. Technical Report A-2000-12, University of Tampere, Department of Computer and Information Sciences, August 2000. [Online]. Available: http://www.cs.uta.fi/reports/r2000.html.

[4] Isto Aho. On the approximability of interactive knapsack problems. In *SOFSEM 2001: Theory and Practice of Informatics, 28th Conference on Current Trends in Theory and Practice of Informatics*, Lecture Notes in Computer Science 2234, pages 152–159, November 2001.

[5] Isto Aho. New polynomial-time instances to various knapsack-type problems. Technical Report A-2002-4, University of Tampere, Department of Computer and Information Sciences, April 2002. Accepted for publication in *Fundamenta Informaticae*. [Online]. Available: http://www.cs.uta.fi/reports/r2002.html.

[6] Isto Aho, Harri Klapuri, Jukka Saarinen, and Erkki Mäkinen. Optimal load clipping with time of use rates. *International Journal of Electrical Power & Energy Systems*, 20(4):269–280, May 1998.

[7] Sanjeev Arora. The approximability of NP-hard problems. In *Proceedings of the Thirtieth annual ACM Symposium on Theory of Computing*, pages 337–348, May 1998.

[8] Sanjeev Arora and Carsten Lund. Hardness of approximations. In Dorit S. Hochbaum, editor, *Approximation Algorithms for NP-hard Problems*, pages 399–446. PWS Publishing Company, 1995.

[9] G. Ausiello, P. Crescenzi, and M. Protasi. Approximate solution of NP optimization problems. *Theoretical Computer Science*, 150(1):1–55, October 1995.

[10] Ian Barland, Phokion G. Kolaitis, and Madhukar N. Thakur. Integer programming as a framework for optimization and approximability. *Journal of Computer and System Sciences*, 57(2):144–161, October 1998.

[11] J.E. Beasley. Population heuristics. In P.M. Pardalos and M.G.C. Resende, editors, *Handbook of Applied Optimization*, pages 138–157. Oxford University Press, Oxford, 2002.

[12] J.E. Beasley and N. Christofides. An algorithm for the resource constrained shortest path problem. *Networks*, 19(4):379–394, July 1989.

[13] Dimitri P. Bertsekas. *Dynamic Programming and Optimal Control*. Athena Scientific, 1996.

[14] R. Bhatnagar, J. Latimer, L.J. Hamant, A.A. Garcia, G. Gregg, and E. Chan. On-line load control dispatch at florida power & light. *IEEE Transactions on Power Systems*, 3(3):1237–1243, August 1988.

[15] R. Bhatnagar and S. Rahman. Dispatch of direct load control for fuel cost minimization. *IEEE Transactions on Power Systems*, PWRS-1(4):96–102, November 1986.

[16] K. Bhattacharyya and M.L. Crow. A fuzzy logic based approach to direct load control. *IEEE Transactions on Power Systems*, 6(3):708–714, May 1996.

[17] Paola Cappanera. *Discrete Facility Location and Routing of Obnoxious Activities*. PhD thesis, Dip. di Matematica, Universitá di Milano, 1999. [Online.] Available: http://www.di.unipi.it/~cappaner.

[18] Alberto Caprara, Hans Kellerer, Ulrich Pferschy, and David Pisinger. Approximation algorithms for knapsack problems with cardinality constraints. *European Journal of Operational Research*, 123(2):333–345, July 2000.

[19] Paolo Carraresi, Fernanda Farinaccio, and Federico Malucelli. Testing optimality for quadratic 0-1 problems. *Mathematical Programming*, 85(2):407–421, 1999.

[20] Douglas W. Caves and Joseph A. Herriges. Optimal dispatch of interruptible and curtailable service options. *Operations Research*, 40(1):104–112, January–February 1992.

[21] Ashok K. Chandra, D.S. Hirschberg, and C.K. Wong. Approximate algorithms for some generalized knapsack problems. *Theoretical Computer Science*, 3(3):293–304, December 1976.

[22] Chandra Chekuri and Sanjeev Khanna. On multi-dimensional packing problems. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 185–194, January 1999.

[23] Chandra Chekuri and Sanjeev Khanna. A PTAS for the multiple knapsack problem. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 213–222, January 2000.

[24] Jianming Chen, Fred N. Lee, Arthur M. Breipohl, and Rambabu Adapa. Scheduling direct load control to minimize system operational cost. *IEEE Transactions on Power Systems*, 10(4):1994–2001, November 1995.

[25] Runwei Cheng and Mitsuo Gen. Evolution program for resource constrained project scheduling problem. In *Proceedings of the First Conference on Evolutionary Computing*, volume 2, pages 736–741, June 1994.

[26] P.C. Chu and J.E. Beasley. A genetic algorithm for the multidimensional knapsack problem. *Journal of Heuristics*, 4(1):63–86, June 1998.

[27] Wen-Chen Chu, Bin-Kwie Chen, and Chun-Kuei Fu. Scheduling of direct load control to minimize load reduction for a utility suffering from generation shortage. *IEEE Transactions on Power Systems*, 8(4):1525–1530, November 1993.

[28] Arthur I. Cohen and Vahid R. Sherkat. Optimization-based methods for operations scheduling. *Proceedings of the IEEE*, 75(12):1574–1591, December 1987.

[29] Arthur I. Cohen and Connie C. Wang. An optimization method for load management scheduling. *IEEE Transactions on Power Systems*, 3(2):612–618, May 1988.

[30] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, Massachusetts, 1990.

[31] Pierluigi Crescenzi and Viggo Kann. How to find the best approximation results — a follow-up to Garey and Johnson. *ACM SIGACT News*, 29(4):90–97, December 1998. [Online]. Available: http://www.nada.kth.se/~viggo/problemlist/compendium.html.

[32] Pierluigi Crescenzi, Viggo Kann, Riccardo Silvestri, and Luca Trevisan. Structure in approximation classes. *Electronic Colloquium on Computational Complexity (ECCC)*, 3(066), 1996.

[33] János Csirik and Gerhard J. Woeginger. Shelf algorithms for on-line strip packing. *Information Processing Letters*, 63(3):171–175, August 1997.

[34] Dipankar Dasgupta. Unit commitment in thermal power generation using genetic algorithms. In *Proceedings of the Sixth International Conference on Industrial & Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE-93)*, pages 374–383, June 1993.

[35] Dipankar Dasgupta and Douglas R. McGregor. Nonstationary function optimization using the structured genetic algorithm. In *Proceedings of the Second Conference on Parallel Problem Solving from Nature*, Parallel Problem solving from Nature 2, pages 145–154, September 1992.

[36] Dipankar Dasgupta and Douglas R. McGregor. Short term unit-commitment using genetic algorithms. In *Proceedings of the 1993 IEEE International Conference on Tools with AI*, pages 240–247, November 1993.

[37] Rod G. Downey and Michael R. Fellows. Fixed-parameter tractability and completeness II: On completeness for $W[1]$. *Theoretical Computer Science*, 141(1–2):109–131, April 1995.

[38] Irina Dumitrescu and Natashia Boland. Algorithms for the weight constrained shortest path problem. *International Transactions in Operational Research*, 8(1):15–29, January 2001.

[39] Martin Dyer, Alan Frieze, Ravi Kannan, Ajai Kapoor, Ljubomir Perkovic, and Umesh Vazirani. A sub-exponential time algorithm for approximating the number of solutions to a multidimensional knapsack problem. *Combinatorics, Probability and Computing*, 2(3):271–284, 1993.

[40] M.E. Dyer and A.M. Frieze. Probabilistic analysis of the multidimensional knapsack problem. *Mathematics of Operations Research*, 14(1):162–176, February 1989.

[41] Henning Fernau. Parameterized maximization. Technical Report WSI-2001-22, Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, 2001.

[42] Henning Fernau and Rolf Niedermeier. An efficient exact algorithm for constraint bipartite vertex cover. *Journal of Algorithms*, 38(2):374–410, February 2001.

[43] John P. Fishburn. Solving a system of difference constraints with variables restricted to a finite set. *Information Processing Letters*, 82(3):143–144, May 2002.

[44] 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(1):100–109, January 1984.

[45] Michael R. Garey and David S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-completeness.* W. H. Freeman, 1979.

[46] Minos N. Garofalakis, Banu Özden, and Avi Silberschatz. Resource scheduling in enhanced pay-per-view continues media databases. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases*, pages 516–526, August 1997.

[47] A.J. Gaul, E. Handschin, W. Hoffmann, and C. Lehmköster. Establishing a rule base for a hybrid es/xps approach to load management. *IEEE Transactions on Power Systems*, 13(1):86–93, February 1998.

[48] Richard Gerber, William Pugh, and Manas Saksena. Parametric dispatching of hard real-time tasks. *IEEE Transactions on Computers*, 44(3):471–479, March 1995.

[49] Philip E. Gill, Walter Murray, and Margaret H. Wright. *Practical Optimization.* Academic Press, London, 1981.

[50] David E. Goldberg and Robert E. Smith. Nonstationary function optimization using genetic algorithms with dominance and diploidy. In *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*, pages 59–68, July 1987.

[51] V. Scott Gordon and Darrell Whitley. Serial and parallel genetic algorithms as function optimizers. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 177–183, July 1993.

[52] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science.* Addison-Wesley, 1989.

[53] R. Hassin. Approximation schemes for the restricted shortest path problem. *Mathematics of Operations Research*, 17(1):36–42, February 1992.

[54] Mordechai I. Henig. Risk criteria in a stochastic knapsack problem. *Operations Research*, 38(5):820–825, September–October 1990.

[55] Dorit S. Hochbaum, Nimrod Megiddo, Joseph (Seffi) Naor, and Arie Tamir. Tight bounds and 2-approximation algorithms for integer programs with two variables per inequality. *Mathematical Programming*, 62(1):69–83, October 1993.

[56] Arild Hoff, Arne Løkketangen, and Ingvar Mittet. Genetic algorithms for 0/1 multidimensional knapsack problems. In *Proceedings Norsk Informatikk Konferanse, NIK '96*, November 1996.

[57] Robert C. Holte. Combinatorial auctions, knapsack problems, and hill-climbing search. In *Proceedings of AI'2001 (The 14th Canadian Conference on Artificial Intelligence)*, Lecture Notes in Artificial Intelligence 2056, pages 57–66, June 2001.

[58] Yuan-Yih Hsu and Chung-Ching Su. Dispatch of direct load control using dynamic programming. *IEEE Transactions on Power Systems*, 6(3):1056–1061, August 1991.

[59] Toshihide Ibaraki and Yuichi Nakamura. A dynamic programming method for single machine scheduling. *European Journal of Operational Research*, 76(1):72–82, July 1994.

[60] Sami Khuri, Thomas Bäck, and Jörg Heitkötter. The zero/one multiple knapsack problem and genetic algorithms. In *Proceedings of the 1994 ACM Symposium on Applied Computing*, pages 188–193, March 1994.

[61] Hyunchul Kim, Yasuhiro Hayashi, and Koichi Nara. An algorithm for thermal unit maintenance scheduling through combined use of GA SA and TS. *IEEE Transactions on Power Systems*, 12(1):329–335, February 1997.

[62] Anton J. Kleywegt and Jason D. Papastavrou. The dynamic and stochastic knapsack problem. *Operations Research*, 46(1):17–35, January–February 1998.

[63] Mark W. Krentel. On finding and verifying locally optimal solutions. *SIAM Journal on Computing*, 19(4):742–749, August 1990.

[64] C.N. Kurucz, D. Brandt, and S. Sim. A linear programming model for reducing system peak through customer load control programs. *IEEE Transactions on Power Systems*, 11(4):1817–1824, November 1996.

[65] Manuel Laguna. Applying robust optimization to capacity expansion of one location in telecommunications with demand uncertainty. *Management Science*, 44(11):S101–S110, November 1998.

[66] W.B. Langdon. Scheduling maintenance of electrical power transmission networks using genetic programming. In *Late Breaking Papers at the Genetic Programming 1996 Conference*, pages 107–115, July 1996.

[67] Jean-Charles Laurent, Guy Desaulniers, Roland P. Malhamé, and François Soumis. A column generation method for optimal load management via control of electric water heaters. *IEEE Transactions on Power Systems*, 10(3):1389–1400, August 1995.

[68] Edward Yu-Hsien Lin. A bibliographical survey on some well-known non-standard knapsack problems. *Information Systems and Operations Research*, 36(4):274–317, November 1998.

[69] Michael J. Magazine and Maw-Sheng Chern. A note on approximation schemes for multidimensional knapsack problems. *Mathematics of Operations Research*, 9(2):244–247, May 1984.

[70] Udi Manber. *Introduction to Algorithms, A Creative Approach.* Addison-Wesley, 1989.

[71] S. Martello, D. Pisinger, and P. Toth. New trends in exact algorithms for the 0-1 knapsack problem. *European Journal of Operational Research*, 123(2):325–332, July 2000.

[72] Silvano Martello and Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations.* John Wiley & Sons, 1990.

[73] Silvano Martello and Paolo Toth. Generalized assignment problems. In *Algorithms and Computation, Third International Symposium, ISAAC '92*, Lecture Notes in Computer Science 650, pages 351–369, December 1992.

[74] V. Milenkovic. Multiple translational containment part II: exact algorithms. *Algorithmica*, 19(1/2):183–218, September–October 1997.

[75] Janis L. Miller and Lori S. Franz. A binary-rounding heuristic for multi-period variable-task-duration assignment problems. *Computers & Operations Research*, 23(8):819–828, August 1996.

[76] Melanie Mitchell. *An Introduction to Genetic Algorithms.* MIT Press, 1996.

[77] F.K. Miyazawa and Y. Wakabayashi. An algorithm for the three-dimensional packing problem with asymptotic performance analysis. *Algorithmica*, 18(1):122–144, May 1997.

[78] Rolf H. Möhring and Franz J. Radermacher. Introduction to stochastic scheduling problems. Technical Report MIP – 8402, Universität Passau, December 1984.

[79] George L. Nemhauser and Laurence A. Wolsey. *Integer and Combinatorial Optimization.* John Wiley & Sons, 1999.

[80] G.L. Nemhauser and Z. Ullmann. Discrete dynamic programming and capital allocation. *Management Science*, 15(9):494–505, May 1969.

[81] Kah-Hoe Ng and Gerald B. Sheblé. Direct load control — a profit-based load management using linear programming. *IEEE Transactions on Power Systems*, 13(2):688–695, May 1998.

[82] Khim Peow Ng and Kok Cheong Wong. A new diploid scheme and dominance change mechanism for non-stationary function optimization. In *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 159–166, July 1995.

[83] James B. Orlin. Some very easy knapsack/partition problems. *Operations Research*, 33(5):1154–1160, September–October 1985.

[84] Maria A. Osorio, Fred Glover, and Peter Hammer. Cutting and surrogate constraint analysis for improved multidimensional knapsack solutions.

Technical Report HCES-08-00, The University of Mississippi, Hearin Center for Enterprise Science, 2000.

[85] Christos H. Papadimitriou. On the complexity of integer programming. *Journal of the ACM*, 28(4):765–768, October 1981.

[86] Christos H. Papadimitriou. *Computational Complexity.* Addison-Wesley, Reading, MA, 1994.

[87] Ulrich Pferschy, David Pisinger, and Gerhard J. Woeginger. Simple but efficient approaches for the collapsing knapsack problem. *Discrete Applied Mathematics*, 77(3):271–280, August 1997.

[88] Michael Pinedo. *Scheduling: Theory, Algorithms, and Systems.* Prentice Hall, 1995.

[89] David Pisinger. *Algorithms for Knapsack Problems.* PhD thesis, Dept. of Computer Science, University of Copenhagen, 1995. (Online.) Available: http://www.diku.dk/users/pisinger/95-1.pdf.

[90] P. Raghavan and C.D. Thompson. Randomized rounding: a technique for provably good algorithms and algorithmic proofs. *Combinatorica*, 7(4):365–374, 1987.

[91] B. Rautenbach and I.E. Lane. The multi-objective controller: a novel approach to domestic hot water load control. *IEEE Transactions on Power Systems*, 11(4):1832–1837, November 1996.

[92] Colin R. Reeves, editor. *Modern Heuristic Techniques for Combinatorial Problems.* Advanced Topics in Computer Science Series. McGraw-Hill, 1995.

[93] Edward M. Reingold, Jurg Nievergelt, and Narsingh Deo. *Combinatorial Algorithms: Theory and Practice.* Prentice Hall, 1977.

[94] Paresh Rupanagunta, Martin L. Baughman, and Jerold W. Jones. Scheduling of cool storage using non-linear programming techniques. *IEEE Transactions on Power Systems*, 10(3):1279–1285, August 1995.

[95] Iraj Saniee. An efficient algorithm for the multiperiod capacity expansion of one location in the telecommunications. *Operations Research*, 43(1):187–190, January–February 1995.

[96] Alexander Schrijver. *Theory of Linear and Integer Programming.* Wiley-Interscience Publication, Chichester, 1998.

[97] David B. Shmoys and Éva Tardos. An approximation algorithm for the generalized assignment problem. *Mathematical Programming*, 62(3):461–474, December 1993.

[98] Aravind Srinivasan. Improved approximation guarantees for packing and covering integer programs. *SIAM Journal on Computing*, 29(2):648–670, November 1999.

[99] Andrew Tanenbaum. *Computer Networks.* Prentice-Hall PTR, 1996.

[100] Arne Thesen. A recursive branch and bound algorithm for the multidimensional knapsack problem. *Naval Research Logistic Quarterly*, 22:341–353, 1975.

[101] Heribert Vollmer and Klaus W. Wagner. Complexity classes of optimization functions. *Information and Computation*, 120(2):198–219, August 1995.

[102] Deh-chang Wei and Nanming Chen. Air conditioner direct load control by multi-pass dynamic programming. *IEEE Transactions on Power Systems*, 10(1):307–313, February 1995.

[103] H. Martin Weingartner and David N. Ness. Methods for the solutions of the multidimensional 0/1 knapsack problem. *Operations Research*, 15(1):83–103, January–February 1967.

[104] L.A. Wolsey. A view of shortest route methods in integer programming. *Cahiers du Centre d'Études de Recherche Opérationelle*, 16:317–335, 1974.

[105] Allen J. Wood and Bruce F. Wollenberg. *Power Generation, Operation and Control.* John Wiley & Sons, 1984.

[106] Houzhong Yan and Peter B. Luh. A fuzzy optimization-based method for integrated power system scheduling and inter-utility power transaction with uncertainties. *IEEE Transactions on Power Systems*, 12(2):756–763, May 1997.

[107] Zuwei Yu. A temperature match based optimization method for daily load prediction considering dlc effect. *IEEE Transactions on Power Systems*, 11(2):728–733, May 1996.

[108] Mark Ziegelmann. *Constrained Shortest Paths and Related Problems.* PhD thesis, Universität des Saarlandes, 2001. [Online.] Available: http://www.mpi- sb.mpg.de/∼mark/.