Interactive knapsacks

University of Tampere
Department of Computer Science
Isto Aho

Foreword

Here is a warning for you, dear reader!

Since publishing this report in 1999, some minor and some serious mistakes has been found from this text. I have commented some of them in my www-pages:

http://www.cs.uta.fi/~tyisah/knapsacks.html

Please, check the page (I will update it occasionally). For example, the decision problems defined on pages 21 and 25–26 have equality, while they should have inequality, thus making some of the given results too pessimistic. This has been corrected in the article based on this report, though. However, there are some other problems, too. If you run into difficulties, please do not hesitate to ask — I will try to deliver an explanation if I can.

26.3.2002, Isto Aho,
Isto.Aho@uta.fi

University of Tampere
Department of Computer Science
Isto Aho: Interactive knapsacks

Abstract

The interactive knapsack problems are generalizations of the classical knapsack problem. Three different new NP-complete problems, IKHD, IKD and MDCS, are presented for the interactive knapsack models, which are also new. IKD and MDCS are shown to be strongly NP-complete.

By using interactive knapsacks we model many 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 containers. Many natural problems related to interactive knapsacks are NP-complete.

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

Contents

1 Introduction

The *interactive knapsack 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 instants, cloning models some control or task, and radiation models the additional activity caused by the control or task.

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 (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 some (real life) problems both of these interaction types are needed, 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.

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

We assume from the reader some basic familiarity with algorithms and complexity analysis. Basic notions and definitions with a few pointers to literature are covered in Section 2, including NP-completeness, NP-completeness in the strong sense, and knapsack problems. Introduction to NP-notation and concepts of complexity analysis is given in Subsection 2.1. We follow generally accepted terminology about knapsacks and related problems. This terminology is recalled in Subsection 2.2.

In Section 4 we give several models for the interactive knapsack problems. Moreover, we present three new NP-complete problems, IKHD, IKD and MDCS for interactive knapsacks, of which IKD and MDCS are shown to be strongly NP-complete. In Section 5 we model some known problems with interactive knapsacks.

The idea behind modeling problems with known specialized algorithms is to study how well algorithms for interactive knapsacks can do over specialized ones. If they do well (or reasonably well), we can cope with one model (the interactive one) and algorithms designed for it.

Knapsack arrays can simulate time intervals of any length partitioned into slots of appropriate length [36, 52]. Many scheduling and planning problems involve decisions in this kind of domain (see Section 5), like load clipping (see Section 7). Moreover, some assignment problems can be placed into a space formed by interactive knapsacks (see Section 6). In each Section we give several examples on how to apply the interactive knapsacks.

The model can and should be presented also in stochastic framework because random variables and processes are used in many applications, like stochastic scheduling [44, 48]. Stochastic interactive knapsacks, however, are not covered in this work.

Application of interactive knapsacks to load clipping is presented in detail. This contains the application of the model, algorithms and empirical tests. Electricity suppliers can use load clipping in load management. There are also other problems in load management that can be modeled with interactive knapsacks. More details, including exact algorithms, test runs and comparisons of the algorithms, can be found from Section 7. The algorithms used include dynamic programming, several heuristics, enumerative method and genetic algorithms. Algorithms are new for the formulation of the load clipping problem presented in this work.

Load clipping is used by electricity suppliers around the world. There is a need to limit the load, when the electricity consumption rise 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 [2]. 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. (Cf. [2, 10, 55].) Each control corresponds to a clone (see 7).

Payback (also called *strikeback*) is a phenomenon caused by the consumption peak after the control of customers devices, like electricity heaters or air conditioners [12, 55]. The energy storage capability [49] is similar to payback, but it occurs before the control. The paybacks and energy storage capabilities are modeled with radiation.

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

The 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 heuristics presented in this work and branch and bound methods, cannot easily take into account the “time window” in which we operate (make controlling plans) and which moves forward as the time passes. 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. In Subsection 7.5 we present a method based on genetic algorithms.

2 Preliminaries

In this section we shortly introduce NP-complete problems (subsection 2.1), basic knapsack problems and their relatives (subsection 2.2). An introduction to design and analysis of algorithms can be found e.g. from [13]. An introduction to complexity analysis can be found from [13, 19]. Martello and Toth [39] 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 [19].

$\mathbb{N}$ 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 $\emptyset$.

2.1 Complexity analysis and NP-completeness

We usually analyze time consumption as a function of the input size. Time consumption can be determined or estimated by the maximum number of steps took by the algorithm. 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.

Definition 1 *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 $n \geq n_0$.*

For example, $2n^3 + 15n^2 + n = O(n^3)$ and $f \cdot g = O(f \cdot g)$. If f is $O(n^k)$ ($k \in \mathbb{N}$), as in the first example, we say that f is a *polynomial* function. If f is not $O(n^k)$, for any $k \in \mathbb{N}$, we say that f is *exponential* (this definition includes e.g. $n^{\log n}$, which is not normally regarded as exponential function).

Definition 2 *If the time complexity function of an algorithm is of order $O(p(n))$, where p is some polynomial function, then the algorithm is a* polynomial time algorithm.

There are some problems, like the knapsack problem, for which we do not know any polynomial time algorithm. This leads to our next definition:

Definition 3 *If the time complexity of an algorithm cannot be bound by any polynomial p, i.e. the time complexity is not of order $O(p)$ for any polynomial p, then the algorithm is an* exponential 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. For these problems we do not know any deterministic polynomial time algorithm which would solve the problem exactly. Some of these problems constitute a subclass consisting of problems for which we have non-deterministic polynomial time algorithms. This subclass of problems is denoted by NP. Assuming that $P \neq NP$, we call a problem *NP-complete*, if it belongs to $NP \setminus P$ and if it is one-to-one transformable to some other 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.

Every problem with a polynomial time algorithm belongs to the class of NP-problems, or $P \subseteq NP$. The question about the inequality $P \neq NP$ has been open since 1971 [19].

As an example of NP-complete problems, consider the partition problem introduced below in the Garey and Johnson style [19].

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

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

We do not show this to be NP-complete. For a proof, see [19] or some book covering the basics of algorithm design. The partition problem is needed later in this work.

NP-hard problems form a superclass of NP-complete problems. Hence, efficient solutions to NP-complete problems do not necessarily help us to solve efficiently the NP-hard problem at hand. Other way works: knowing how to solve an NP-hard problem efficiently would give us a way to solve the NP-complete problems efficiently.

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 is mainly three techniques to show that a polynomial transformation exists between problems [19]. These are *restriction*, *local replacement* and *component design*. Notice that numerous NP-complete problems exhibit proofs not to be categorized into any general proof technique [19]. 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 the known NP-complete problem. (See [19].)

We use restriction and local replacement in this work. 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 also NP-complete ([19]). 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?

To show that SUBSET SUM is NP-complete by restricting it to PARTITION, we have to show first that SUBSET SUM belongs to the set of NP problems. We do that by constructing a non-deterministic polynomial time algorithm, which guesses the answer and verifies it. Our non-deterministic polynomial time algorithm is shown in Figure 1. It guesses the right answer (in polynomial time) non-deterministically (line (1)) and checks the answer in polynomial time (line (2)).

Input: 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 1: A non-deterministic algorithm solving the subset sum problem.

Next, we have to add some restrictions to the subset sum problem so that the resulting problem is identical to the partition problem or that there is an "obvious" one-to-one correspondence between the problems [19]. This correspondence is enough to provide the transformation from PARTITION to SUBSET SUM. In detail, we restrict the subset sum problem to instances where $B = \sum_{n \in N} s(n)/2$. Hence, in the subset sum problem we have to find a subset N' such that $\sum_{n \in N'} s(n) = B$.

However, $B = \sum_{n \in N} s(n)/2 = \sum_{n \in N \setminus N'} s(n)$, and hence if we can solve the subset sum problem with the specified restriction in polynomial time we have a polynomial time solution also for the partition problem.

The partition and the subset sum problems exhibited above are in *decision problem* form. We can state these problems also as *optimization problems*. For example, the subset sum optimization problem has the same instance but now we maximize the sum of chosen elements without exceeding B, i.e. 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 method to find in polynomial time the answer to the decision version of the subset sum problem. Similarly, a polynomial time algorithm for the decision version would solve the optimization version in polynomial time by using binary search. Hence, the optimization version of SUBSET SUM is also NP-complete.

We can solve the subset sum optimization problem exactly with dynamic programming in $O(B|N|)$ time [39]. Thus, SUBSET SUM is solvable in polynomial time of two variables B and $|N|$. The exponential instance occurs when B is much greater than $|N|$.

We also establish results concerning NP-completeness "in the strong sense". Here we define the 3-partition problem, which is NP-complete in the strong sense [19].

First we define two functions (D_Π is the set of the instances of the problem Π); Length : $D_\Pi \to \mathbb{Z}^+$, is a function mapping a problem Π instance I to integer Length$[I]$ and Max : $D_\Pi \to \mathbb{Z}^+$, is a function mapping a problem Π instance I to integer Max$[I]$. The function Length corresponds to the number of symbols used to describe I under some reasonable encoding scheme for problem Π. The function Max, in turn, is intended to map instance I to an integer Max$[I]$ corresponding to the magnitude of the largest number occurring in I. (Cf. [19], pp. 92–95.)

Definition 4 *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. 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 $|N|$ [19].

Definition 5 *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.

In the subset sum problem we may set $\text{Length}[I] = |N| + \log B + \sum \log s(n)$ and $\text{Max}[I] = B$ (as B is polynomially related to $\max\{B, s(n) : n \in N\}$; we may drop items larger than B from each instance). Hence, the subset sum problem is a number problem, because Max can be exponential over Length by increasing B. The dynamic programming gives a pseudo-polynomial time algorithm for the subset sum problem. 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 dynamical programming $O(B|N|)$ gives a polynomial time (in $|N|$) upper bound.

Definition 6 *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 $P = NP$. Next we give the 3-partition problem, which is NP-complete in the strong sense [19].

3-PARTITION
INSTANCE: A finite set A of $3m$ elements, a bound $B \in \mathbb{Z}^+$, and a size $s(a) \in \mathbb{Z}^+$, for each $a \in A$, such that each $s(a)$ satisfies $B/4 < s(a) < B/2$ and such that $\sum_{a \in A} s(a) = mB$.
QUESTION: Can A be partitioned into m disjoint sets $S_1, \ldots, S_m$ such that, for $1 \leq i \leq m$, we have $\sum_{a \in S_i} s(a) = B$?

Notice that the constraint $B/4 < s(a) < B/2$ implies that every S_i must contain exactly three elements from A.

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 the problem Π'. Suppose that Max' and Length' are defined in Π'. A pseudo-polynomial transformation is a function $f : D_\Pi \to D_{\Pi'}$ such that

1. for all instances I of Π the transformation 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 i.e. 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 definition of Π_p imply that f is a polynomial transformation. Thus $\Pi_{p'}$ is strongly NP-complete. (Ch. [19], p. 101.)

2.2 Knapsack problems

The *knapsack problem* 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* one's objective is to

$$\begin{array}{ll} \max & \sum_{j=1}^{n} p_j x_j \\ \text{subject to} & \sum_{j=1}^{n} w_j x_j \leq b, \\ & x_j = 0 \text{ or } 1, \quad j = 1, \ldots, n. \end{array}$$

This problem is also known as the *0–1 knapsack problem*. As an example, consider items with weights 1, 5, 8, 13, 14, 14, 15 and profits 1, 10, 10, 25, 14, 19, 30, respectively. Hence $n = 7$. If $b = 30$, we can choose items 1, 5 and 7 and hence the total weight is 30 and the corresponding value is 45. A better solution is achieved by choosing items 1, 2, 3 and 4 giving total weight 27 and value 46. If we replace item 4 by item 7 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 much along the same way as our example proceeded in the previous section. Also the other knapsack problems introduced in this subsection are NP-complete.

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 (containers) of capacity 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)

$$\begin{array}{lll}
\max & \displaystyle\sum_{i=1}^{m}\sum_{j=1}^{n} p_{ij}x_{ij} & \\
\text{subject to} & \displaystyle\sum_{j=1}^{n} w_{ij}x_{ij} \le b_i, & i = 1, \ldots, m, \\
& \displaystyle\sum_{i=1}^{m} x_{ij} = 1, & j = 1, \ldots, n, \\
& x_{ij} = 0 \text{ or } 1, & i = 1, \ldots, m, \quad j = 1, \ldots, n.
\end{array}$$

Here p_{ij} and w_{ij} are the profit and weight of object j if inserted in knapsack i. 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}. This is not a knapsack problem, but knapsack subproblems are used when solving it. (See [39].)

Table 1 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. We can choose j to belong to any of the knapsacks i only once. This means that each column contains only one non-null variable x_{ij}.

We cannot put all items to the same knapsack, because the total weights 52, 50 and 58 will exceed the given bounds 35, 20 and 25, respectively. If we put items 1, 2, and 3 to the first sack (profit 58 and weight 29) and items 4 and 5 to the second sack (profit 18 and weight 19), our total profit is 76 and we have fulfilled the conditions. This is not optimal. If we change the first item from the first sack to the third sack, our profit will be 88. (We have not calculated the optimum.)

Table 1: Example of GAP.

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

An extension of the knapsack problem is the $0-1$ *collapsing knapsack problem* where we allow the capacity of the knapsack to depend (to be a function) on the number of items it contains [47]. The restriction is now

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

We can extend the knapsack problem similarly and define the *added value knapsack problem* by changing the objective 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 (to be a function) on the number of the items. Still another possibility is to define

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

so that the profit of the knapsack i depends (is a function) on the total weight of the items it contains. We call also (1) as added value knapsack problem: in this work there is no need to distinguish between these two objectives. 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 one application, load clipping.

3 The model of interactive knapsacks

In this section we describe the model of interactive knapsacks. The model is new. In what follows, we talk about knapsacks, containers or plainly about sacks but we mean the same thing. In order to have any interaction between knapsacks we have to have at least two knapsacks. We suppose that there are m containers (knapsacks). Let the containers be ordered so that container i is before container $i+1$ 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 the interval $[i, i]$. Set of all containers is also called a *knapsack array* or a *CS array* (cloned sacks array).

The idea of interaction can be applied in many ways. The containers can have some restrictions which make them interactive. Also the application of decision variables choosing items to the containers can imply interactions. 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: containers can be grouped into container sets and containers in one set may have identical weights and profits for an item while different container sets may differ from each other. We can "implement" for example average rates for each interval with this interaction. An example of this can be found from section 7.

If we select an object j for a container i we consume the capacity b_i of the container i as in the knapsack problems. The interaction of this selection (application of the decision variable) is a function $I_{ij} : \{0, \ldots, m-1\} \to \mathbb{Q}$. Selecting j for i has an influence $I_{ij}(k)$ to other containers $k \neq i$ and the influence depends also on "target" of the interaction. Usually $I_{ij}(k)$ gives an amount, either positive or negative, to be added to the container k. Typically, that amount is some proportion of item j added to container i influencing container k, i.e. we add to both profit and weight of container k values $p_{kj}I_{ij}(k)$ and $w_{kj}I_{ij}(k)$, respectively. Other possibility is to add values $p_{ij}I_{ij}(k)$ and $w_{ij}I_{ij}(k)$, respectively, or some mixed set of the above values, depending on the application.

It is comfortable to use the influence I_{ij} as a function of the distance between

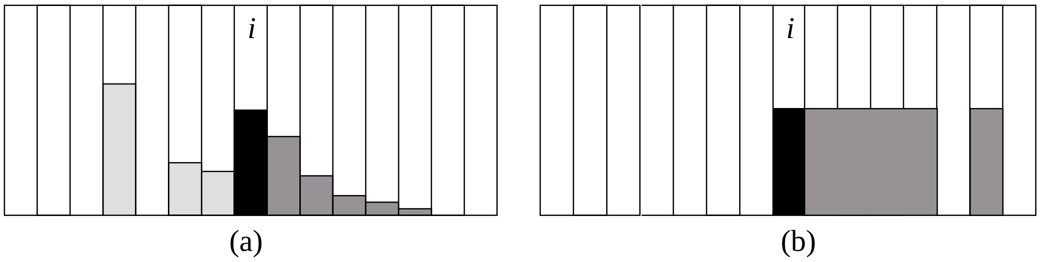

Figure 2: Example of radiation (a) and cloning (b).

the containers i and k and hence, we have

$$I_{ij}(k) = I'_{ij}(k - i), \tag{2}$$

where $I' : \mathbb{Z} \to \mathbb{Q}$. In practice, there is an upper bound u on the interaction distance. This means that in (2) we have $I_{ij}(k) = 0$ when $|k - i|$ is large enough ($> u$).

Figure 2 (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 has its influence to other containers as well. We denote this interaction as a function $r_{ij}(k) : \mathbb{Z} \to \mathbb{Q}$ (similar to I in equation (2)). 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 container 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 mixed amounts $p_{kj}r_{ij}(k)$ and $w_{ij}r_{ij}(k)$, which are used later. The shape of an influence can be arbitrary, like in the left side of i (in light gray). However, in practice the shape usually follows some regular pattern (function) like in the right side of container i (in dark gray). There is no reason why we could not use partly (or totally) negative radiation; indeed, in applications we need negative radiation. For radiation $r_{ij}(k)$ we typically have $-2 < r_{ij}(k) < 2$.

Figure 2 (a) also suggests that we should not apply this interaction scheme recursively. For example, consider the influence $r_{i-1,r_{ij}(i-1)}(k)$ ($k \neq i-1$) of the container $i-1$ 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 (container) i (in light gray) is called the left or the *early radiation* and at the right side (in dark gray) the right or the *back radiation*. The knapsacks involved in radiation caused by object j at container i are denoted by $R_{ij} = [i - u, i-1] \cup [i+1, i+u]$.

Next we describe cloning, the other basic form of interaction. If we put an item j into a container i, the restrictions between containers may imply that we have to put j also into some other containers. This can also be interpreted as interaction based on decision variables. In Figure 2 (b) we have an example of this kind of interaction. By inserting j (in black) into container i, we also have to *copy* j into containers $i+1,\ldots,i+4$ and $i+6$. This kind of interaction is called *cloning*. The cloned parts of item j are *copies* of item j.

Mathematically, this interaction is similar to I in (2). This time I is replaced with a function $c_{ij}(k):\mathbb{Z}\to\{0,1\}$. We could define this in the same manners as we defined radiation, introducing the left and right copies. However, in applications we need mainly right copies. If the weight of item j inserted into sack i is w_{ij}, we have $c_{ij}(k)=1$, when $i+1\le k\le i+u$, otherwise $c_{ij}(k)=0$. This means that we add the cloned object (copy) to container k ($k\ne i$), 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 $w_{ij}c_{ij}(k)=w_{ij}$, when $i+1\le k\le i+u$ and the other containers are kept unchanged.

Usually we are more interested in cloning patterns without holes, unlike in Figure 2 (b). Reasons for this become clear in the following sections. It is also usually sufficient to tell the selection of an object j for the leftmost container i and the number of the cloning containers $c\in\mathbb{N}$. Hence, we only have to clone the object j for containers $i+1,\ldots,i+c$. We say that the whole set of cloned copies with the inserted item contained in the containers $i,\ldots,i+c$ is a *clone*. Clone refers also to the containers $i,\ldots,i+c$. Several clones form a *clone family*. The knapsacks involved in cloning object j starting at i are denoted by $C_{ij}=[i,i+c]_j$. Clone C_{ij} contains also the container 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 cloning are usual. Following the above assumptions, we presume that radiation spreads around the clone C_{ij} instead of container i. This means that cloning is not involved with radiation itself and that we do not separate the first container i, into which the insertion is made, from the other items taking part in cloning by applying the radiation only around i. Hence, the interaction in hybrid models involves containers

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

We define the described hybrid model to be the *0–1 interactive knapsacks model* or the *0–1 cloned sacks model* (or 0–1 CS model shortly).

Figure 3 has item j inserted into container i, for which we have both a clone (in black) and radiation (in gray). There is also item k in container $i-4$ 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 suppressed

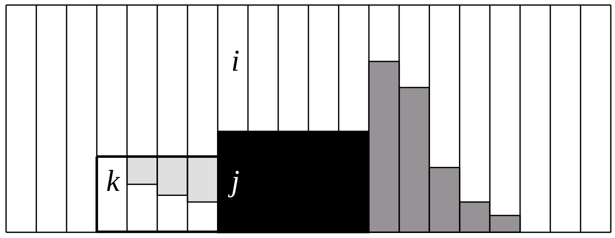

Figure 3: Radiation and cloning together.

the contents of containers $[i-3, i-1]$. The back radiation of length 5 of item j is positive and influences containers $[i+4+1, i+4+5]$. 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 cloning is applied for the resources with rule $w_{ij}c_{ij}(k) = w_{ij}$. From Figure 3 we cannot deduce, how the cloning affects profits: it can be one of the methods presented earlier.

The model of the 0–1 interactive knapsacks can be generalized to the 0–n interactive knapsacks model (or 0–n cloned sacks model) meaning that the same object j may be copied into the same container at most n times. Here we have two options on handling the copies of a clone. We can allow the clones connected to object j to overlap, like in Figure 4 (a). Here the dotted line separates items from their copies. However, in practice the clones are often equal as in Figure 4 (b).

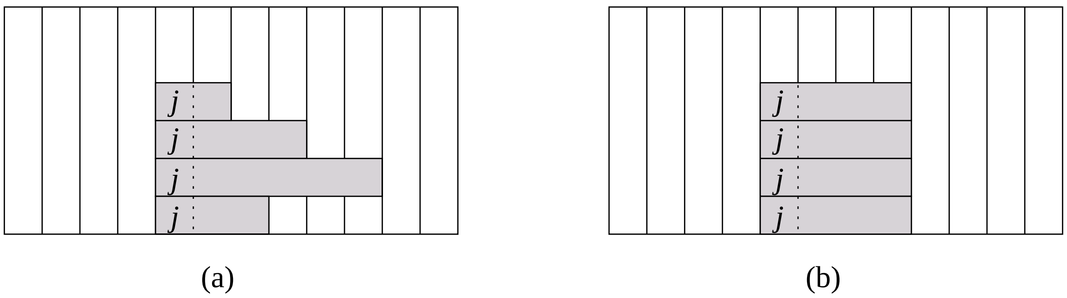

Figure 4: Examples of 0–n cloned sacks.

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* or the *multi-choice clones model* (or multi-choice CS model).

4 A classification of the problems

We start this section with a short description of different kind of cloned sacks problems. We restrict ourselves to the basic problems and some of their variants. Notes concerning the status of NP properties are also provided.

4.1 The 0–1 cloned sacks problems

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

$$I_{ij}(k) = \begin{cases} r_{ij}(k), & \text{when } k \in [i-u, i-1]_j \cup [i+c+1, i+c+u]_j, \\ 1, & \text{when } k \in [i, i+c]_j, \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

to make the following material more readable. Analogically, we define I_{ij} $(i = 1, \ldots, m$ and $j = 1, \ldots, n)$ to be the corresponding intervals.

Our basic problem for m containers and n items is to

$$\max \quad \sum_{i=1}^{m} \sum_{j=1}^{n} x_{ij} \sum_{k=i-u}^{i+c+u} I_{ij}(k) p_{kj} \tag{4}$$

$$\text{subject to} \quad \sum_{j:\exists i: l \in I_{ij}} x_{ij} w_{ij} I_{ij}(l) \leq b_l \tag{5}$$

$$x_{ij} = 0 \text{ or } 1, \tag{6}$$

$$\sum_{i=1}^{m} x_{ij} \leq 1, \tag{7}$$

where $l = 1, \ldots, m$ in (5), $i = 1, \ldots, m$ in (6), and $j = 1, \ldots, n$ in (6)–(7).

Sum (4) counts the profit of the 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 (3). Inequality (5) tells that one cannot overload the containers: it takes into account the effects of the clones and radiations as in (4). Index j goes through objects having some part of either their clone or radiation at the container l (where l is the index of the container for which we are checking this restriction). We need index i, because it settles the object and container together giving amounts x_{ij}, w_{ij} and I_{ij}. Note that we allow the profits to vary (in (4)) but the weights (in (5)) are fixed. (Hence, this is a mixed set type of interaction discussed earlier.)

Condition (6) ensures that each object can be selected only once for a container. Inequality (7), in turn, ensures that each object is selected at most once for all containers $i = 1, \ldots, m$. The problem setting (4)–(7) is called the *0–1 cloned sacks problem*. By changing the condition (6) to

$$x_{ij} \in \mathbb{N} \tag{8}$$

(condition (7) has to be changed accordingly) we have a prototype for the *0–n cloned sacks problem*. By defining the 0–n cloned sacks problems 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 4 (b). To implement 0–n sacks as in Figure 4 (a), we need more decision variables (or given parameters).

We can also extend the problem setting (4)–(7) by modifying (7) to

$$\sum_{i=1}^{m} x_{ij} \leq K, \tag{9}$$

and by adding an extra restriction

$$x_{kj} = 0, \text{ for } i < k \leq i + c, \text{ when } x_{ij} = 1 \tag{10}$$

obtaining the *0–1 multi-cloned sacks problem*. Inequality (9) allows at most K clones in knapsack array. Condition (10), however, is used to prevent the selection of object j for a container 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.

The 0–1 cloned sacks problem has an important special case: $u = 0$, i.e. the case with no spreading (or radiation). This 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 container), it turns out that this is equivalent to GAP.

Theorem 7 *If $c = u = 0$, the 0–1 cloned sacks problem is equivalent to the generalized assignment problem.*

Proof. Notice that conditions (6)–(7) are same for both problems and that they do not contain c or u. Substitution $c = u = 0$ to equation (3) gives

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

for $[i, i+c]_j = [i, i]_j = i$ and for $[i-u, i-1]_j = [i, i-1]_j = [i+c+1, i+c+u]_j = [i+1, i]_j = \emptyset$. Hence, sum (4) is equal to

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

Notice that $I_{ij} = [i, i]_j = i$ and $l \in I_{ij}$ implies $l = i$. Now, in (5) index j runs the objects from 1 to n. We have $l = i \in I_{ij}$, i.e. j indexes through objects for clones containing $l = i$ as a part of the clone, or just only the container $l = i$. This follows from the fact that all the clones are of length one. Thus, (5) turns out to be

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

$(i = 1, \ldots, m)$. Hence, we have turned both (4) and (5) to their counterparts in GAP. □

By assuming that the profits and weights depend only on the objects and not on the objects and containers together, we arrive to the *0–1 multiple knapsack problem* [39]. Algorithms for solving GAP and the 0–1 multiple knapsack problem are provided in [39].

We can extend this problem setting 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 $j = 1, \ldots, n$, and may use them as decision variables among the variables x_{ij}. Or we can introduce $c_{ij} \in \mathbb{N}$ and $u_{ij} \in \mathbb{N}$ for $i = 1, \ldots, m$ and $= 1, \ldots, n$. With these assumptions we may expose other optimization problems that can be modeled with the 0–1 interactive knapsacks (and with other models, too). We say that our problem is the 0–1–n cloned sacks problem if it is otherwise like in (4)–(7), but with variable c_j's and u_j's (or with c_{ij}'s and u_{ij}'s). If we further add conditions (9)–(10) we obtain the 0–1–n multi-cloned sacks problem. (In what follows, we often talk only about 0–1 cloned sacks.)

If we interpret all the m knapsacks as a *time line* $[1, m]$, we can think that a clone of an object j from a container i to $i + c_j$ such that $[i, i + c_j] \subseteq [1, m]$ is a *decision* (or act, task) j *executed* (actualized, realized) on the time interval $[i, i + c_j]$ of *length* $c_j + 1$. Now we can optimize the length, position and number of decisions with different criterions. Restrictions concerning these variables are also possible. Radiation with time has application specific interpretations: the decision j may have some consequences to profits and resources (weights) before or after (or both) the specific decision j is executed at period $[i, i + c_j]$. These questions are considered in greater detail in section 5 and in load clipping application in section 7.

Next we show a result concerning the number of clone families in a knapsack array. We count here families consisting of one type of clones. We suppose conditions (9)–(10).

Theorem 8 *Assume conditions* (9) *and* (10). *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 non-negative integers. Integer v_i (except the last one) gives the number of containers having no clones (or part of any clone) before $(i+1)th$ 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 containers before the first clone and the last integer v_k equals the number of containers 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 and is the number of clone families having k clones of length d in the knapsack array of length m.

By [51] (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

$$\begin{aligned} & \sum_{k=0}^{\lfloor m/d \rfloor} \binom{m - kd + k}{k} \qquad (12)\\ = & \sum_{k=0}^{\lfloor m/d \rfloor} \frac{(m - kd + k)(m - kd + k - 1)\cdots(m - kd + 1)}{k!} \\ \geq & \sum_{k=0}^{\lfloor m/d \rfloor} \frac{(m - kd + 1)(m - kd + 1)\cdots(m - kd + 1)}{k!} \\ = & \sum_{k=0}^{\lfloor m/d \rfloor} \frac{(m - kd + 1)^k}{k!}. \end{aligned}$$

Consider k as a constant equal to $\lfloor m/d \rfloor$. Now we have

$$\frac{(m - kd + 1)^k}{k!} = \Omega(m^k) \qquad (13)$$

as m approaches infinity.

Suppose that (11) is of order $O(m^l)$ for some $l \in \mathbb{N}$, e.g. that (11) is polynomial. Consider the number of clones with $0, 1, \ldots, l + 1$ clones in the knapsack array of length m' ($m' \gg m$) and sum them up. By (12), the number of clone families is of order $\Omega({m'}^{l+1})$, a contradiction. Hence, there cannot exist any $l \in \mathbb{N}$ polynomially bounding the number of clone families as m grows. □

Corollary 9 *When the length of a clone is one ($c = 0$), the number of different clone families in knapsack array is 2^m, and when the length is two ($c = 2$), the number of clone families is F_{m+1}, the $(m+1)$th Fibonacci number.*

Proof. When $c = 0$, sum (11) is of the form

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

When $c = 1$, we have

$$\sum_{k=0}^{m} \binom{m-2k+k}{k} = \sum_{k=0}^{m} \binom{m-k}{k} = F_{m+1}.$$

The latter equality can be found, for example, from [23], p. 289. □

Theorem 8 and Corollary 9 do not tell much about the growth rate of the number of clone families as a function of the length of the knapsack array. For a deeper analysis we need generating functions. The strength of Theorem 8 is in its simplicity. As Corollary 9 suggests, it is intuitively clear that the number of clone families decreases as the length of one 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 8 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 subsection 7.2 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 [1, 2] (see subsections 7.2 and 7.3). It turns out that this approach is NP-complete, as will be shown in the next subsection.

4.2 A NP-complete approach for the 0–1–n cloned sacks

By the previous subsection, it is evident that the 0–1–n cloned sacks problem is NP-complete. (By Theorem 7, GAP is a restriction of the 0–1–n cloned sacks problem.) In this subsection we give the corresponding decision problem and show that it is indeed NP-complete. We also show that one important heuristic approach needs exponential time. (The heuristic version is considered because it is much simpler to use than the original problem.) Even thought this approach is NP-complete, all versions of it are solvable in pseudo-polynomial time, i.e. there exists polynomial time algorithms when we fix its parameters.

Now we consider a heuristic approach, where we optimize one object j ($j = 1, \ldots, n$) at a time. At the same time we assume the conditions (9) and (10). The decision problem (interactive knapsacks heuristic decision, IKHD) is

IKHD
INSTANCE: $x_i \in \{0, 1\}$, profits p_i, weights w_i, lengths of clones c_i, lengths of radiations u_i and functions $I_i : \mathbb{Z} \to \mathbb{Q}$, for containers $i = 1, \ldots, m$, with capacities b_i, and two positive integers P and K.
QUESTION: Is there a distribution of values x_i such that

$$\sum_{i=1}^{m} x_i \sum_{k=i-u_i}^{i+c_i+u_i} I_i(k) p_k = P, \tag{14}$$

and fulfilling the requirements below? We require that the sacks are not overfilled (14), that the clones are separate (15), that each object is selected at most once to a container (16), and that there are at most K clones in the knapsack array (17):

$$x_i w_i I_i(l) \leq b_l, \quad \text{when there is } i : l \in I_i, \tag{15}$$

$$x_k = 0, \text{ for } i < k \leq i + c_i, \text{ when } x_i = 1, \tag{16}$$

$$x_i = 0 \text{ or } 1, \text{ and} \tag{17}$$

$$\sum_{i=1}^{m} x_i \leq K, \tag{18}$$

where $l = 1, \ldots, m$ in (14) and $i = 1, \ldots, m$ in (15) and (16).

IKHD can be solved in polynomial time with the nondeterministic algorithm given in Figure 5. In the first line we guess the correct distribution of values x_i. In the second line we test the instance and then output yes or no, which is the answer to the decision problem. Line (2) can be performed in polynomial time.

(1) **guess** a distribution of values $x_i \in \{0, 1\}$ for $i = 1, \ldots, m$ and $j = 1, \ldots, n$
(2) **if** equation (13) holds with constraints (14)–(17)
(3) **then** output **true**
(4) **else** output **false end**

Figure 5: A nondeterministic algorithm for IKHD problem.

Hence, we have proved

Lemma 10 IKHD $\in$ NP.

In order to prove the NP-completeness of IKHD, we still have to consider its NP-hardness.

Lemma 11 IKHD *is* NP-*hard.*

Proof. We show this by restricting IKHD to PARTITION, a well known NP-complete problem. Suppose that $c_i = u_i = 0$, $K = m$ and b_i is large enough. IKHD is reduced to the problem of finding a distribution of values x_i, for which we have

$$\sum_{i=1}^{m} x_i p_i = P$$

such that $x_i = 0$ or 1 and

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

which is always true. Let $A = \{1, \ldots, m\}$. Now we can ask, if there is a subset $A' \subseteq A$ such that

$$\sum_{i \in A'} p_i = \sum_{i=1}^{m} x_i p_i = P = \sum_{i=1}^{m} (1 - x_i) p_i = \sum_{i \in A \setminus A'} p_i,$$

where $x_i = 1$, when $i \in A'$ and $x_i = 0$, when $i \in A \setminus A'$. This is the partition problem. $\square$

By lemmas 10 and 11 we have shown Theorem 12.

Theorem 12 IKHD *is* NP-*complete.*

The proof of Theorem 12 (or Lemma 11) says that the approach of placing only one type of clones into the knapsack array at a time is NP-complete in general, because it contains an NP-complete subproblem. However, if we consider only problem instances where $c > 0$ and $u > 0$, not allowing the partition problem to occur, we have to give another proof. This is also of practical importance. In the proof of Theorem 13 some indices like $i - u$ and $i + c + u$ go out of the interval $[1, m]$ where the variables and instance parameters are not defined; the corresponding variables will be considered to be zero so that conditions are fulfilled and the objective is not affected.

Theorem 13 IKHD *is* NP-*complete for fixed values of $c > 0$ and $u > 0$ (and $c < m$).*

Proof. Also this version of IKHD clearly belongs to NP.

Our objective is to find a set of x_i's such that

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

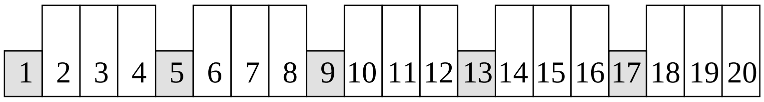

Figure 6: A sample of knapsack array for Theorem 13 ($K = 5$ and $c = 4$).

We now restrict ourselves to the cases where $I_i(k) = 1$, for all i and k where $k \in [i - u, i + c + u]$, and 0 elsewhere. Let $p'_i = p_{i-u} + p_{i-u+1} + \cdots + p_i + \cdots + p_{i+c+u}$, for all i. Hence, the objective is to find x_i's such that

$$\sum_{i=1}^{m} x_i \sum_{k=i-u}^{i+c+u} p_k = \sum_{i=1}^{m} x_i p'_i = P. \tag{19}$$

We transform the subset sum problem to IKHD with fixed $c > 0$. Recall that an instance of SUBSET SUM consists of a finite set N of elements with a given size $s(n)$ and of a positive integer B.

Let $P = B$ and $K = |N|$. From N we obtain the knapsack array of length $m = cK = c|N|$, a partition of the array having K "areas" $A_1, \ldots, A_K$ separated by containers, where each "separator" i has $w_i = b_i = 1$, for $i = 1 + kc$ and $k = 0, \ldots, K - 1$. Suppose that $w_i = b_i = 2$ when $i \neq 1 + kc$. Each A_j consists of c containers. Let $p'_j = s(a_j)$ be the profit obtained from the first container of interval A_j (i.e., $p'_{1+kc} = 1$, for $k = 0, \ldots, K - 1$). Other $c - 1$ containers of the area A_j can have $p'_i = 0$. Each $a_j \in N' \subseteq N$ means that $x_i = 1$, where $i = 1 + (j - 1)c$, other x_i's are 0. This transformation can clearly be done in polynomial time. As an example, consider Figure 6 where there is a separator at index 17 ($17 = 1 + (5 - 1)c$) being also the first container of the area A_5.

We still have to show that the conditions of IKHD are not violated. From each subset sum answer candidate we obtain an answer candidate to IKHD, where the conditions are fulfilled, for each area is of length c and each chosen element $a_j \in N'$ corresponds to the clone put into area A_j and clones cannot overlap because they fill the containers exactly (conditions (14) and (15)). Each x_i is either 1 or 0 (condition (16)). We can choose at most one item $a_j \in N$ to belong to N', meaning that at most one clone is put into each area and the number of areas is K (condition (17)). If the subset sum answer is 'yes', i.e. sum is $B = P$, then the answer to corresponding IKHD problem is 'yes'.

Conversely, suppose we have an algorithm for IKHD and an arbitrary instance of IKHD such that there are $m = cK$ containers, that $w_i = b_i = 1$ for $i = 1 + kc$, that $w_i = b_i = 2$ when $i \neq 1 + kc$, and that $p'_i = 0$ when $i \in [1 + kc + 1, (k + 1)c]$ ($k = 0, \ldots, K - 1$). Other parameters are not fixed. Now we can choose at most one clone for area A_j without violating the conditions (14), (15) and (16), since if the object is chosen to some other container k than the first in area A_j, the clone will not fit into some separator, i.e. container $i = (j-1)c$ having only half of the capacity, and we can select only one item to the first container. This clone corresponds to item a_j to be selected to N'. In our subset sum problem $B = P$ and $|N| = K$ and $s(a_k) = p'_{1+kc}$, for $k = 0, \dots, K-1$. Now IKHD $\in P$ would imply that the subset sum also belongs to P. $\square$

The generalizations of IKHD considered in this work are also NP-complete and, hence, we have the following result.

Corollary 14 IKHD *with container dependent clone lengths and radiation spreads, i.e. with c_i's and u_i's, for $i = 1, \dots, m$, is* NP*-complete.*

The subset sum optimization problem considered in subsection 2.1 has its "hard cases" when B is much greater than $|N|$ meaning at the same time that IKHD is hard to solve when P is much greater than K. Here we have not considered an IKHD situation where we can position clones freely over container array and not only to some predefined slots. It increases the solution space giving more clone families or possible IKHD solutions. IKHD is not, however, a strongly NP-complete problem because it can be solved with a pseudo-polynomial time algorithm.

If in Theorem 13 and in Corollary 14 we have a fixed K, IKHD can be solved in polynomial time by enumerative methods given in subsection 7.2. To see this, consider the proof of Theorem 8. The running time, with 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 (see Theorem 19).

Consider that c_i's are variables. Now we have to change the problem setting so that x_i's are no longer variables, and we establish a condition

$$x_i = \begin{cases} 1, & \text{if } c_i \ge 0, \\ 0, & \text{if } c_i = -1, \end{cases} \tag{20}$$

for $i = 1, \dots, m$. We use constraint (19) instead of constraint (16). Here we let $c_i \in \mathbb{N} \cup \{-1\}$ or give some bounds meaning $c_i \in \{\text{cmin}, \dots, \text{cmax}\} \cup \{-1\}$, where cmin $<$ cmax are some positive integers.

Suppose that a clone length is varying and that there are l different lengths for a clone. We first consider the case where each c_i is a constant and has equal length and where we need a clone family consisting of at most K clones. The number of clone families is now of order $\Omega(m^K)$. Each clone in one clone family can have l different lengths and hence there are at most l^K different possibilities (combinations) for a clone family with K clones is at least $\Omega(l^K m^K)$. This is not tight because it does not take in account the bounds of the knapsack array. However, l^K is a constant according to m and thus, the enumerative method for IKHD runs in polynomial time with variable clone lengths. If K is not fixed and hence, neither m, IKHD with variable clone lengths is also NP-complete. So, we have proved

Theorem 15 IKHD *with variable clone lengths (and radiation spreads), i.e. with variable c_i's (and u_i's) for $i = 1, \ldots, m$, is* NP-*complete.*

In practice we meet variable c_i's and sometimes even variable u_i's. Variable clone lengths give some new directions for scheduling problems and in some control problems they are essential as discussed in sections 5 and 7.

4.3 The 0–1–n cloned sacks problem is NP-complete

IKHD discussed in the previous subsection is enough for showing that the 0–1–n cloned sack problem is also NP-complete. There we have many objects or items to be handled at the same time; the 0–1–n cloned sacks problem is a direct generalization of IKHD. In Theorem 7 we established a link between the 0–1–n cloned sacks problem and GAP, which is NP-complete in the strong sense [39], and hence, reduction to GAP is also possible. First we give the corresponding decision problem for the 0–1–n cloned sacks problem and then give the results.

IKD

INSTANCE: $x_{ij} \in \{0, 1\}$, profits p_{ij}, weights w_{ij}, lengths of clones c_{ij}, lengths of radiations u_{ij} and functions $I_{ij} : \mathbb{Z} \to \mathbb{Q}$, for items $j = 1, \ldots, n$, and containers $i = 1, \ldots, m$ with capacities b_i and two positive integers P and K.

QUESTION: Is there a distribution of values x_{ij} such that $P \in \mathbb{N}$:

$$\sum_{i=1}^{m} \sum_{j=1}^{n} x_{ij} \sum_{k=i-u_{ij}}^{i+c_{ij}+u_{ij}} I_{ij}(k) p_{kj} = P \tag{21}$$

and fulfilling the requirements below? We require that the sacks are not overfilled (21), that the clones are separate (22), that the object is selected at most once to a container (23) and that there are at most K clones in the knapsack array (24):

$$\sum_{j:\exists i:l\in I_{ij}} x_{ij}w_{ij}I_{ij}(l) \le b_l \tag{22}$$

$$x_{kj} = 0, \text{ for } i < k \le i + c_{ij}, \text{ when } x_{ij} = 1 \tag{23}$$

$$x_{ij} = 0 \text{ or } 1, \tag{24}$$

$$\sum_{i=1}^{m} x_{ij} \le K, \tag{25}$$

where $l = 1, \ldots, m$ in (21), $i = 1, \ldots, m$ in (22)–(23), and $j = 1, \ldots, n$ in (22)–(24).

Before IKD definition we noted that IKHD, a special case having only one clone type ($n = 1$) is NP-complete. Similarly, if there is only one container ($m = 1$) implying that $c_{ij} = u_{ij} = 0$ for the items considered, then IKD reduces to the 0–1 knapsack problem. Thus, IKD is NP-complete also in this special case. Algorithms for the 0–1 knapsack problem are given in [39], pp. 13–80.

Theorem 16 IKD*,* IKD *with fixed* $c_{ij} > 0$*, and* IKD *with variable* c_{ij}*'s (and variable* u_{ij}*'s) are* NP*-complete problems in the strong sense.*

Proof. To show that IKD belongs to the class of NP problems, we can proceed as in the case of Lemma 10. Suppose that $c_{ij} = k > 0$, for all i and j. We transform the 3-partition problem containing $3k$ integers to IKD as follows: each $s(a_j) = w_{ij}$, for all i, meaning that for all containers, item j has equal weight. Let $b_i = B$, for each i. Divide the knapsack array of length m to k areas of length m/k having separators between areas, i.e. an extra containers between areas having their capacities equal to zero as well as profits for items in those containers. (We can suppose without loss of the generality that m/k is integer.)

Set each $p_{ij} = 1$ for the first containers in each area, and set other p_{ij}'s to 0. Further, set $P = 3k$, which implies that we have to include each item and that the item (i.e. clone) have to be put in the appropriate place. An item belonging to one of the subsets $S_1 \ldots, S_k$ in the 3-partition, say $a_j \in S_1$, corresponds to a clone j in the area corresponding, in turn, to subset S_1, involving profit p_{1j} and weight w_{1j}. The restrictions are not violated.

This transformation preserves the 'yes'-answer (property 1 in the definition of pseudo-polynomial transformation) and can be computed in polynomial time (property 2). An instance of IKD corresponding to a 3-partition problem has more parameters than the 3-partition problem thus giving as a simple polynomial q_1 (property 3). Also a polynomial q_2 can be formed as $\mathrm{Max}[f(I)]$ and $\mathrm{Max}'[I]$ can be same for both problems, where f is the transformation described above (property 4). Hence, this transformation can be performed in pseudo-polynomial time.

If we can solve IKD in polynomial time, it would give a polynomial time algorithm for the 3-partition problem, which is NP-complete in the strong sense. □

GAP is NP-complete in the strong sense and IKD is a generalization of GAP. In practice we are given, however, some minimum for integers $c_{ij} > 0$ ($i = 1, \dots, m$ and $j = 1 \dots, n$) and GAP does not cover that case, neither the case of variable c_{ij}'s.

5 Applications of IKD

In this section we introduce some applications that can be modeled and solved with IKD. In subsection 5.1 we give applications involving time: scheduling problems. In subsection 5.2 we give other applications, some of which also interpret the knapsack array as a time line. An introduction to the scheduling theory and solution methods is given e.g. by Pinedo [48].

5.1 Planning in time

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 a schedule maximizing the profits so that successive tasks are disjoint, that the resources consumed by all of the tasks of different types are not over exhausted (i.e. items fit into the containers).

This interpretation of the model contains different single machine scheduling problems: we have $w_{ij} = b_i$ for each task j, all containers have equal capacity ($b_i = b_1$, for $i > 1$), and hence, only one task at a time can be selected. We can also restrict the number of times the task is selected; one is a common bound for the number of possible selections. This is obtained by setting $K = 1$ in IKD. Depending on the type of the objective function (if not using (20)) and other additional constraints, this kind of problem is NP-complete or not [19].

Suppose that for all items j, we have $0 < p_{1j} < 1$ and $p_{ij} = p_{1j}^i$, for $1 < i \leq m$, or $p_{ij} = p_{1j}a^{i-1}$, where $p_{1j} > 0$ and $0 < a < 1$ is the discounting factor (i.e. profits are discounted over time). Now the optimal solution will place the clones (items) $j = 1, \ldots, n$ to as left in the knapsack array as possible, because the profits are larger in the left end of the array. Each clone j will be selected at most once (by setting $K = 1$ at (24)). Because each selection increases the sum in (20), all clones will be selected. Notice that different clones are separate because one clone consumes all the capacity of the containers involved by (21) 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 clones j, i.e. by adding a constraints of type

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

we have "sequencing with release times and deadlines on one processor" of [19], a well known NP-complete problem (see Appendix D). 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 not the weights. Other single machine scheduling (sequencing on one processor) problems are discussed in [19]. Some of these problems can be exposed directly with IKD but some need changes to IKD.

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 the task (clone) j and the position i, we have a single machine scheduling in the sense of [27]. In [27] 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.)

Notice that we can expose equivalent cost-functions as in [27] without altering the objective (20). The weighted sum of earliness and tardiness for tasks (or jobs) can be calculated by choosing the profits p_{ij} appropriately. Let the profit coefficients of earliness and tardiness be e_j and t_j, respectively. Let the due date of task j be $d_j \in [1, m]$. We turn the minimization of the disadvantages of the misplacement of the task (earliness or tardiness) used in [27] to the maximization of advantages of appropriate placement of the task. Figure 7 contains a container d_j and two lines corresponding to profits given by the placement of task j. If the last container used by task j is d_j, our profits (20) are maximized.

This behavior is achieved by setting $p_{d_j,j} = \max\{(d_j - 1)e_j, (m - d_j)t_j\}$,

$$p_{d_j-k,j} = p_{d_j,j} - e_j k,$$

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

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

$(1 \le k \le m - d_j)$. All profits are positive, even if e_j and t_j are. We still have to modify the interaction function I_{ij} — either the clone or the radiation part — so that it only takes the last copy of a clone (last moment of a task) into account when counting the profits (20). This at the same time makes condition (21) almost useless which is not dangerous if our containers are so little that we have space only

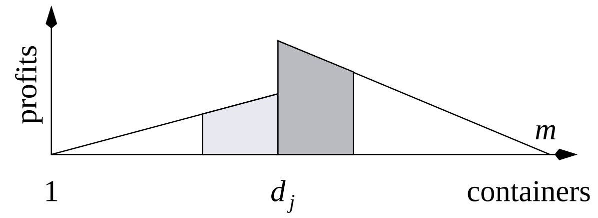

Figure 7: Due date and profits.

for one copy ($w_{ij} = b_i$). The condition (22) ensures that the clones do not overlap and, hence, we will not violate condition (21).

If we do not modify the interaction, 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 7) and a task starting at d_j (dark grey area). The tardiness coefficient is so large that it gives better profit to clone starting at due date than to a clone ending exactly at due date.

By setting $e_j = 0$ our objective counts the weighted sum of tardiness, which is a non-increasing function and, hence, we can use some sophisticated methods given in [27], where in the minimization we have a non-decreasing function. The problem, however, remains NP-hard [19, 27].

The CS's can also be exploited for some multiprocessor (or multiple machine job) scheduling problems. We first set $w_{i1} = w_{ik}$ for $1 < k \leq n$ and for all i (all items have equal weight on each container). Then we set $b_i = Mw_{i1}$ so that at most M processors can be used at a time (i.e. at most M tasks can be selected at a time, in Figure 8 we have $M = 6$).

We use the same approach as earlier by discounting profits $0 < p_{1j} < 1$ over time (i.e. $p_{ij} = p_{1j}^i$, $i = 1, \ldots, m$). 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. Now, in the optimal solution

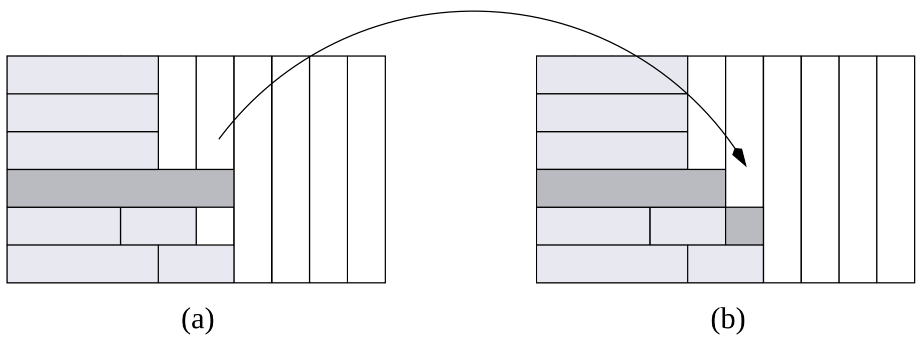

Figure 8: Identical processors.

the completion time of the last task is minimized [36].

This is almost the multiprocessor scheduling problem of [19], 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 8 (a) stating at the same time the processor; the third processor is used even thought the capacities are handled like Figure 8 (b). This implies at the same time that the tasks have to consume resources equally, i.e. $w_{i1} = w_{k1}$ for all i, k.

However, if processors or tasks are not identical, there may be some holes left like in Figure 8 (a) (or the container should have items positioned into certain places for which this model does not suit) and it is harder to determine which processor is used for each task. Multidimensional knapsack arrays to be presented in Section 6 give a more natural setting to multiprocessor problems.

If we have setup or clean-up times [48], we can employ early or back radiation, respectively. Radiation can depend on a task to be made (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 the processor which takes time. Clean-up can be thought more naturally in the machinery scheduling context.

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 (i.e. $0 \leq r_{ij}(k) \leq 1$).

5.2 Other applications

Some applications given in this subsection require modifications to IKD problem setting and, hence, are not direct applications of IKD. However, the modifications are small and the basic concepts of interactive knapsacks, cloning and radiation, are still applied.

By adding extra restrictions of the form

$$m_j \leq c_{ij} \leq M_j \tag{26}$$

(minimum and maximum lengths for a clone) and some minor application specific modifications to IKD with variable length clones, we can model energy management problems, like unit commitment [11] and load clipping [1, 2] (see Section 7).

Constraints (25) can be added to IKD as extra constraints or we can modify the sum (4) 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 $k \in [i+m_j, i+M_j]$, and 0 otherwise. The coefficient $C_{ij}(c_{ij}) = 0$, if c_{ij} is shorter than m_j or longer than M_j, thus reducing the profits to zero. The sum in constraint (5) will be modified similarly.

The load clipping application and its solutions are handled in greater detail in section 7. Notice that in energy management we often *have to* use back radiation and sometimes we can also use early radiation.

The load clipping can be applied also to "load shedding in the internetwork" to reshape the traffic loads at communication lines [53]. The cloning has a natural interpretation: it is the time we prevent some hosts from sending packets. The radiation is more interesting: while people try to make connections and do not succeed, they probably try again later. The time after which they try is random but after we release the line, there will be more consumption than it would be without the packets discarding. One can, of course, model the shape of the increased consumption, the shape of back radiation, like in load clipping [5, 12, 34, 49].

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.

IKD can be also applied to the capacity expansion problem. Saniee [52] and Laguna [36] 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

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

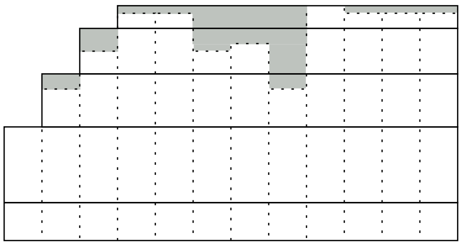

Figure 9: Capacity expansion.

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

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

This is our basic interactive knapsack problem with only one constraint, if we allow the weights, profits and containers to have negative capacity in (4)–(5).

Condition (5) is equal to the condition 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 9 shows an example with five clones, added at times (containers) 1, 2, 3 and 4. At first container we have added two clones.

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 (dark areas). If we allow variable clone lengths and use objective (4) and condition (5), we can optimize capacity expansion and also take into account the time intervals where we have extra capacity (containers 6–8 for topmost item and container 8 for the other item).

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 [20]. This introduces random variables c_{ij} to the IKD model.

Moreover, IKD 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 containers following the task in load clipping used in energy management (see section 7). Intuitively, load clipping is similar to capacity expansion: "we do better, if we can fulfill the restriction with smaller set of 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.

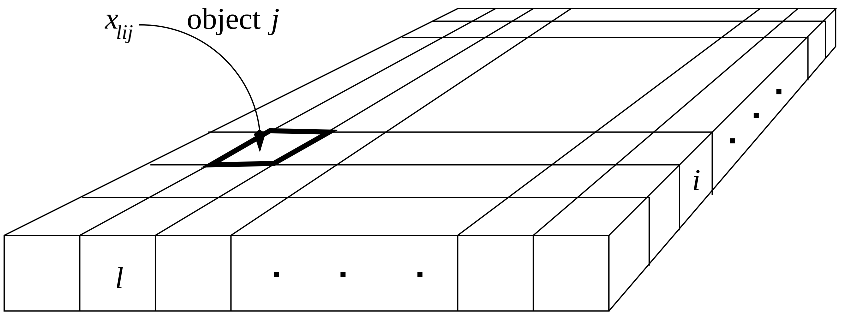

Figure 10: A sample CS-array of order 2.

6 The 0–1 multidimensional cloned sacks

6.1 The 0–1 multidimensional cloned sacks are NP-complete

Suppose that we have M ($\in \mathbb{N}$) 0–1 cloned sacks arrays that interact together just like the knapsacks interact together in 0–1 cloned sacks model. Again we suppose that these CS's are ordered so that we can talk about left, right, earlier and later CS's when compared to some other CS. As it seems, we copy the ideas of interactive knapsacks to this CS-array. Figure 10 clarifies this idea. This recursive idea can be applied several times. We stick first to the easiest case, when the depth is only two, i.e. when we have interactive CS-array of the second order.

Now the object j can be inserted into a CS l ($1 \leq l \leq m_2$) where we can choose a knapsack i ($1 \leq i \leq m_1$). Hence, the decision variable is x_{lij}. Here we have two kind of natural interactions: those internal to l and interactions between different CS's, i.e. between the neighbors of l.

To clarify differences between these interactions, we define the *order* of the 0–1 multidimensional cloned sacks to be k, if there are k CS-arrays. In other words, if the order is k, we have k dimensions of different size $m_1, \ldots, m_k$, where $m_i \in \mathbb{N}$, for $i = 1, \ldots, k$.

A CS-array of order k is a k-dimensional *knapsack space*. The first dimension, lines in space, is discussed in subsections 4.1–4.3. The sample CS in Figure 10 is "in the plane" and is of order 2. A CS-array of order 0 is reserved for the traditional

0–1 knapsack problem and correspondes to a "point in the space".

Next we define the model precisely. Suppose that we are to define CS-array of order q. Hence, we need q indices $i_1 \ldots, i_q$, where each $i_j \in \{1, \ldots, m_j\}$. We denote $J_j = \{1, \ldots, m_j\}$ for $j = 1, \ldots, q$ and $J = J_1 \times \cdots \times J_q$. If A and B are sets, then $A + B = \{a + b \mid a \in A, b \in B\}$.

We change the definitions of I_{ij} (an interval) and $I_{ij}(k)$ (a function from interval I_{ij} to $\mathbb{Q}$) a bit. Intuitively they will remain the same: I_{ij} will hold the containers involved with clones and radiation and function operates similarly in those containers. We suppose that $I_{ij} \subseteq J$ and that I_{ij} maps a coordinate and an item to a set of coordinates, i.e. is a function $I_{ij} : J \times \{1, \ldots, n\} \to J'$, where $J' \subseteq 2^J$. 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 an item j put into a container i to a container $k \in J$. 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 some container, the number of containers to which copies are inserted in each dimension can be fixed for all items and containers, it can depend on a container, or it can be a variable. Further, the number can be equal for all or some dimensions. This kind of a scheme would allow only convex clones (as can be seen in preceding sections). In applications it is comfortable to allow copies to be positioned in non-convex way, e.g. 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 an item j to some container i in the finite container space J causes some interactions. We suppose that the copies C_{ij} will be positioned so that i is the first container of C_{ij} in the ordering of the knapsacks. The radiation spreads around the copies, i.e. around the clone.

The radiation is bounded in an obvious way: when the container $k \in J$ 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 $u \in \mathbb{N}$. Here we may use any metrics we like. (We do not separately consider the clone and its radiation in I_{ij} as we did in the 1-dimensional case.) Hence, the interactions of item j occur inside a q-dimensional ball of radius u centered at i in the container space. Notice that the clone formed by interactions is not necessarily a ball, and i is on the border of clone. Figure 11 contains a clone (grey squares, each describing a container) having the "first" container in dark grey. The radiation is not separated here from copies (other interpretation is that the clone consists of copies only).

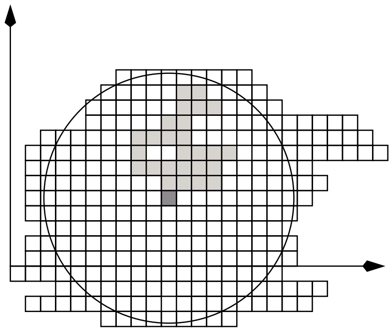

Figure 11: An example of a non-convex clone and its surrounding ball.

In the 0–1 multidimensional cloned sacks problem our objective is to

$$\max \quad \sum_{i\in J}\sum_{j=1}^{n} x_{ij} \sum_{k\in I_{ij}} I_{ij}(k)p_{kj} \tag{27}$$

$$\text{subject to} \quad \sum_{j:\exists i\in J: l\in I_{ij}} x_{ij}w_{ij}I_{ij}(l) \le b_l \tag{28}$$

$$x_{ij} = 0 \text{ or } 1, \tag{29}$$

$$\sum_{i\in J} x_{ij} \le 1, \tag{30}$$

where $l \in J$ in (27), $i \in J$ in (28), and $j = 1, \dots, n$ in (28)–(29). Now we can pose the decision problem for 0–1 multidimensional cloned sacks as follows. Compare formulas (26)–(29) to the one-dimensional case defined in (4)–(7).

MDCS

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 containers b_i, and a positive integer P $(i \in J)$.

QUESTION: Is there a distribution of $n|J|$ values x_{ij} giving the predefined profit P, i.e. such that the sum in (26) equals P and the constraints (27)–(29) are satisfied?

We may modify constraint (29) to

$$\sum_{i\in J} x_{ij} \le K \qquad \text{for each } j. \tag{31}$$

At the same time we add an extra restriction

$$x_{kj} = 0, \text{ for } k \in C_{ij}, \text{ when } x_{ij} = 1, \tag{32}$$

where the C_{ij} is a part of the clone consisting of copies and not radiation. Now an item can be put several times into the container space but the representations may not overlap.

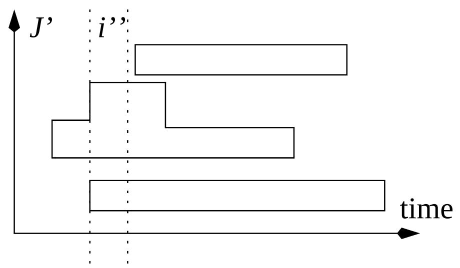

Figure 12: Restricting number of clones in subspaces.

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

$$\sum_{i:\exists i' \in J' : i=(i',i''):x_{ij}=1} x_{ij} \leq K' \qquad \text{for each } j \text{ and } i'' \in J/J', \tag{33}$$

where i is a combination of i' and i'' (denoted shortly and bit misleadingly by $i = (i', i'')$ in (32)) 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, t. Equation (32) counts the clones of type j that occur in subspace J' and restricts the sum with K'. Notice that the existence part can be true for several copies in the same clone but we count each clone at most once. By adding index j into K' we can handle each item separately.

Figure 12 has three clones and they are projected into space spanned by J' and one dimension of J/J', say "time". Suppose that $K' = 2$. The existence quantifier in the sum enables us to count the clone in the middle only once in i''. In $i'' + 1$ the sum will be three. Notice 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 (32) we obtain individual constraints to different container subspaces. Now

$$\sum_{k=1}^{k'} K'_k \leq K \tag{34}$$

can restrict the total number of clones in the whole container space to K, where the k' is the number of constraints of type (32) added. The indexing can also be on items. Also (31) can be generalized to allow for example overlapping in some container subspaces but not in all.

MDCS is NP-complete in the strong sense. We show it by using the next result concerning practical implementations of algorithms.

Theorem 17 *Knapsack arrays of order q, $q \in \mathbb{Z}^+$, can be simulated by knapsack arrays of order one.*

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

$$f(i) = i_1 + (1 + m_1)(i_2 - 1) + (1 + m_1 + m_2)(i_3 - 1) + \cdots + (1 + \sum_{j=1}^{q-1} m_j)(i_q - 1),$$

which we can construct in polynomial time. Every $(1 + m_1)$th container works as a separator having zero capacity. This prevents us to put clones into the space so that their shape will be interpreted falsely. (We can add separators to other dimensions as well and this is still polynomial construction.) □

However, interactions, cloning and radiation have to be applied in the simulations in a more versatile way than described in the earlier sections, allowing non-convex clones (clone may consist of different intervals).

Theorem 18 MDCS *is* NP-*complete in the strong sense.*

Proof. We show that function f defined in the proof of Theorem 17 is a pseudo-polynomial transformation from MDCS to IKD. 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 MDCS. Max$[I]$ is polynomially related to Max$'[f(I)]$ and Length$[I]$ is polynomially related to Length$'[f(I)]$ by Theorem 17. Hence, one-to-one transformation between MDCS and IKD given in Theorem 17 is 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 8), 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 polynomial relations of Max's and Length's. □

Notice, that Theorem 18 holds for knapsack arrays of any order q, $q \in \mathbb{Z}^+$.

6.2 Applications of 0–1 multidimensional cloned sacks

Next we give applications of 0–1 multidimensional cloned knapsacks. The first one is related to assigning tasks and the next three to packing objects.

Miller and Franz [41] study the *multi-period assignment problem*, in which n employees are to be assigned to one of m tasks during T periods (problem statement of Miller and Franz is given in Appendix B). This corresponds to the 0–1 multidimensional cloned sacks problem 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 container and there is no radiation. A clone corresponds to an employee and it is of variable size and shape. Miller and Franz [41] consider a binary-rounding heuristic for their problem.

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 (32). 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 (27) where all weights equal 1. The same holds for I in the copy area. To have all the constraints of Miller and Franz we have to add some extra constraints similar to (27) differing only in the relation: $=$ or $\geq$.

By using clones we can define some constraint more naturally than in the multi-period assignment problem given in [41]. For example, the constraints that relate an employee to work continuously can be obtained by requiring that clones do not contain holes in the dimension J_2 (time). The radiation has no obvious interpretations for assignment problem. We may conclude that MDCS of order two is related to the multi-period assignment problem.

Milenkovic [40] examines the *translational containment* problem (see Appendix C). The objective is to determine a set of translations for k polygons that place them in a non-overlapping configuration inside a polygonal container. This problem, again, can be simulated to a given accuracy by a two dimensional knapsack array. In MDCS presentation we need numerous containers in order discretize the vertices of a polygon to a useable (or required) accuracy. This implies that the clones are also rather large.

We let the weight of each item and container to be 1, including copies used in the clones. Radiation is not used. The profit is 1 for each item and container but for copies the profit is zero. Each knapsack has a capacity of 1 inside the "container area" and 0 otherwise. In [40] each polygon has a set of translations. Each translation corresponds to a possible value of a varying clone. The "displacement" constraints have to be added separately for MDCS. We do not have to touch the MDCS objective in the case of kNN and kCN (see Appendix C) because if we can insert all k items into the container 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* [14]. In strip packing a list of rectangles (h_j, v_j), $1 \leq j \leq n$, with height h_j and width v_j, $0 \leq h_j, v_j \leq 1$, is given. The objective is to position 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 this time translations are not allowed. Therefore, we have fixed clones when using MDCS of order two. The number of the knapsacks in both dimensions depends on the heights and widths: the greatest common divisor determines the number of sacks 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). This also resembles the Tetris game: if we can use explosives, radiation has natural interpretation.

A third kind of packing problem is to pack items in three dimensions. This packing problem has been considered for example in [43]. In the three dimensional packing problem we have a rectangular box (which can be generalized to a container of any shape) into which we put boxes (which also can be generalized to any shape). The boxes (or shapes) are clones having three dimensions.

We arrange the containers in MDCS to have unit capacity; each item and hence each copy of the item in the clone consumes the whole capacity of the container. Thus, two boxes cannot overlap.

If each container 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). Varying clones can be used, however, to put items in different positions into the box.

Multidimensional cloned sacks can be applied also to some multiprocessor scheduling problems, as we noted in subsection 5.1. 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 containers modeling the processors have equal amounts and just enough capacity to take one task at a time. This is the

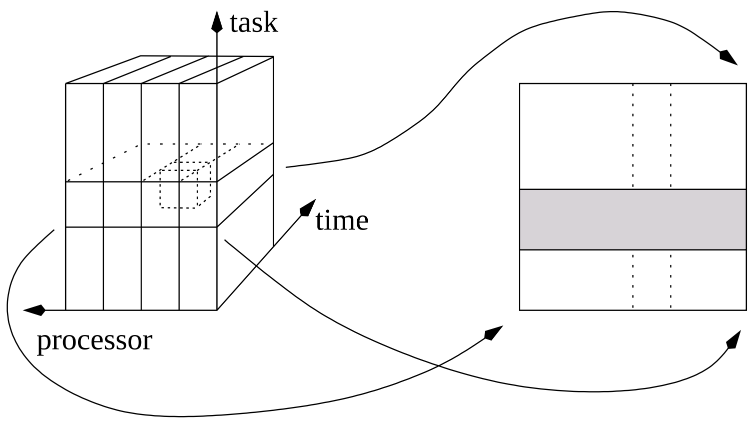

Figure 13: Example of constraint handling in shop scheduling.

multiprocessor scheduling case of [19] or [48] (see Appendix D).

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 can be achieved also 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 dimension consists of containers with capacities equal to the task, 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 containers.

Figure 13 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 containers 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 or clone connected to the task, preventing this task to be done on some other processor at the same time.

Notice 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 13. 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 13.

7 Application in load clipping

The following sections describe a real life application and its possible solutions. We present the load clipping problem introduced in [1], a simplification of the model given in [2]. In subsection 7.1 we set the model and its parameters. In subsections 7.2 and 7.3 we use a heuristic approaches, where we make decisions for one item ("group") at a time. This corresponds to the use of IKHD. In subsections 7.4 and 7.5 we present a method that tries to position several items to a knapsack array at a time. This problem corresponds to IKD. Solutions presented in this work are from [1] and [2] except the enumerative (subsection 7.2) and evolutionary solutions (subsection 7.5) which are new.

Several methods for load clipping are proposed in the literature. Some of these methods might be applicable to variants of IKHD and IKD presented in this work. To be able to perform load clipping tasks, we have to forecast the load consumption [35, 57]. The optimization methods work on these expected values and most of them consider the forecasts as exact values.

Two well-known groups of algorithms for load clipping are enumeration methods and methods based on dynamic programming (DP) [10, 55]. These methods are optimal or nearly optimal but their major drawback is long execution time [1, 2, 12, 55] (see also subsection 7.2). DP has also many other applications in electricity framework [11, 56], some of which are potential application areas of interactive knapsacks.

Fuzzy versions of DP have been developed for integrated load clipping and unit commitment [5, 26]. Unit commitment, however, contains some restrictions not included in our model and fuzziness increases the state space of DP. Caves and Herriges [7] use stochastic DP, where the future prices of electricity are unknown. We assume here that prices (buying and selling rates) are given. The buying and selling rates are used to obtain the profits p_{ij} in IKD.

There are also methods based on linear programming (LP) [34] and on hybrid models using LP and DP [38]. The plain LP is hard to use with our model of load clipping, because it is essentially non-linear. Laurent *et al.*'s [38] approach differs from our approach in the tou-rates (buying and selling rates) and in the constraints.

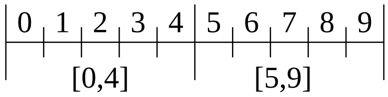

Figure 14: The relationship between an interval and a time point.

However, our DP, unlike our heuristics, is not applicable to as large systems as Laurent *et al.*'s method.

Chen *et al.* [8] 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.* [12]. By using an extra variable, we can achieve better results [2].

Genetic and evolutionary algorithms are widely considered in many applications [42], including the multiple knapsack problem [28], project scheduling [9] and many problems related to electrical framework, e.g. unit commitment [15, 17] and unit maintenance scheduling [29, 37]. 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 [9, 28]. Kim *et al.* [29] develop simulated annealing and tabu search methods for unit maintenance scheduling. We present genetic algorithm for load clipping in subsection 7.5. An introduction to genetic algorithms can be found e.g. from [42].

7.1 The load clipping model

In this subsection 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 subsection.)

The set of clones of type j is called a *plan* or *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 containers. The length of an interval $[a, b]$ is $b - a + 1$. A *clipping situation* Y is a function $Y : [0, m] \to \mathbb{R}$ $(m > 0$ and $m \in \mathbb{N})$. The domain (knapsacks, CS array) $[0, m]$ is called the *optimization interval* and the elements of the goal set are called either *overload* (positive) or *underload* (negative).

The container i in an optimization interval $[0, m]$ is called a *time point*. The phrase "time point i" is used also for the interval (also container in this case) $[i, i]$; this should not cause any confusion (cf. Figure 14).

Figure 14 shows the relation between an interval (knapsack array) and a time

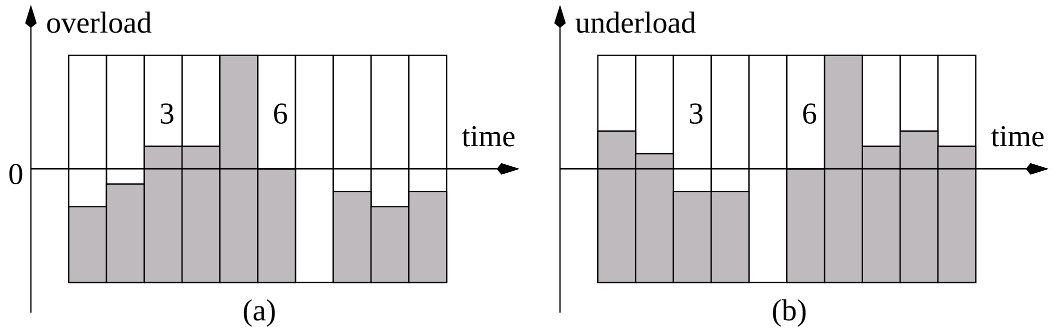

Figure 15: Clipping situation in electricity management (a) and its container representation (b).

point (container). There is an interval $[0, 9]$, which is divided into two subintervals $[0, 4]$ and $[5, 9]$. Numbers $0, \ldots, 9$ refer to the time points (containers).

Function Y maps a container and its contents to a real number representing the filling state of the container. Overload means that the container is not full enough and underload that the container 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.

Figure 15 contains an example of a clipping situation used in electricity management (a) and its representation with interactive knapsacks (b). Notice that the opposite direction of the vertical axis is the only difference between the clipping situation and its representation.

One could avoid this confusing juxtaposition by using some other terms in case of the overload and the underload or by defining “anti-material” items. However, the use of new terms would lead to non-standard terminology (in the point of view of electricity management).

Function Y determines the goal level which is drawn as a horizontal zero line in Figures 15 (a) and (b). Both figures are showing the same situation where we have one overload interval in time slots 3–5. The size of containers is the sum of control capacities (weights of clones, to be explained) plus the summed maximum radiation of controls (clones). In the beginning the containers contain some substance (or undetermined items) corresponding to some electricity consumption levels discretized into equally sized time slots (e.g. five minutes).

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

The goal level giving the best result is achieved in the sixth container in (b) or in the sixth time slot in (a). We say that the sixth container is full or filled enough.

The functions H and T map interval $[0, m]$ into the positive real numbers. Every time point i with overload has a positive real $H(i)$ called *price factor* (buying rate of electricity). If at time point i there is *underload*, the positive real $T(i)$ is the *income factor* (selling rate of electricity). The *overload interval* is an interval $[a, b] \in [0, m]$ such that at every time point $i \in [a, b]$ there is overload (container is too empty), i.e. if $Y(i) > 0$.

The clipping situation (knapsack array that is partially filled) is partitioned into container sets or *hours* $0 = a_1, a_2, \ldots, a_{n+1} = m$ of equal length, i.e. $a_{i+1} - a_i = a_i - a_{i-1}$ $(2 \leq i \leq n)$. For example in Figure 14 the intervals $[0, 4]$ and $[5, 9]$ can be considered as hours. We denote $t = a_{i+1} - a_i + 1$, and say that t is the length of an hour. 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, i.e. $Y(j) = Y(j+1)$, $H(j) = H(j+1)$ and $T(j) = T(j+1)$, 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 containers equally to all containers in that particular hour. This is an example of the third interaction type mentioned earlier.

In what follows, we continue the example started at Figure 14. Table 2 shows sample quantities for overloads (the uppermost line, referring to the filling situation of the containers), price factors (the middle line) and income factors (the lowest line). In the first hour there is no overload (containers are filled enough), but in the second hour there is an overload interval $[5, 9]$ (containers are too empty).

Table 2: An example about overloads, prices and incomes.

Y	-1.05	-1.05	-1.05	-1.05	-1.05	0.5	0.5	0.5	0.5	0.5
H	550	550	550	550	550	230	230	230	230	230
T	40	40	40	40	40	10	10	10	10	10

The total loss is

$$h(Y) = \sum_{i \in [0,m]} p_i(Y(i)), \tag{35}$$

where

$$p_i(x) = \begin{cases} -H(i)x, & \text{if } x \geq 0, \\ T(i)x, & \text{otherwise.} \end{cases} \tag{36}$$

Hence, we always have $p_i(Y(i)) \leq 0$. If there is underload (containers are too full), we lose income and if there is overload (containers are too empty), we have to pay some extra. The sum (34) counts the money lost, so its best possible value is 0. The case in Figure 14 and Table 2 gives by the sum (34) the total cost of -785 (the unit is not fixed).

The sum (34) corresponds to the profit given by the partially filled CS array (clipping situation). The profits given in (35) make (34) to correspond to an added value objective function (1) which depends on the total capacity (weight) of groups (items).

Consider now a situation, where we have a *group* (an item that can be put to a container, or a clone to containers), by which we can lower the overload (fill a container) by making a *control* (by actually inserting the item into a container). A control of a group is 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 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 when making a control $[a, b]$ (placing a clone into a CS array). The total influence of a control (clone) is called the *control amount* (space reserved for the clone) and it is the product of the control capacity w and the control length $b - a + 1$ (hence, the weight is same for all copies of an item in a clone). The control amount of an hour is denoted by W. We have $W = wt$; 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

$$\begin{aligned} g([a_i, a, b, a_j], Y) \quad = \quad & \sum_{k=a_i}^{a_{i+1}-1} p_k(Y(k) - W(a_{i+1} - 1 - a + 1)/t) + \\ & \sum_{k'=i+1}^{j-2} \sum_{k=a_{k'}}^{a_{k'+1}-1} p_k(Y(k) - W) + \\ & \sum_{k=a_{j-1}}^{a_j-1} p_k(Y(k) - W(a_j - 1 - b + 1)/t) \end{aligned}$$

shows how a control $[a, b]$ impacts on total costs.

Function $u : \mathbb{N} \to \mathbb{N}$ maps the control (clone) 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 (clone) of length $b - a + 1$ at time (container) i. Moreover, we always have $r(b - a + 1, i) \geq 0$, where $i \in [b + 1, b + u(b - a + 1)] = R$, and otherwise

$r(b-a+1,i)=0$. Notice that r is the function giving radiation and R is the set of containers involved with radiation. Further, "in practice" we have

$$\sum_{k\in[b+1,b+u(b-a+1)]} r(b-a+1,k) \le w(b-a+1). \tag{37}$$

Constraint (36) simply says that a payback (radiation) does not exceed the control amount (capacities the clone consumes from containers). We use the hours much in the same way with paybacks than with controls.

Next we show the impact of a control $[a,b]$ and its payback to clipping situation Y as a function I. Further, we show how one can calculate all controls of a group into the clipping situation, function Y'. The hours to be affected this time are $[a_i,a,b,a_j,b+u(b-a+1),a_l]$. By function

$$\begin{aligned} I([a,b],Y)(k) &= \\ &\begin{cases} Y(k), & \text{when } 0\le k<a_i,\\ Y(k)-W(a_{i+1}-1-a+1)/t, & \text{when } a_i\le k<a_{i+1},\\ Y(k)-W, & \text{when } a_{i+1}\le k<a_{j-1},\\ Y(k)-\dfrac{W(a_j-b)}{t}+\displaystyle\sum_{k'=b+1}^{a_j-1}\dfrac{r(b-a+1,k')}{t}, & \text{when } a_{j-1}\le k<a_j,\\ Y(k)+\displaystyle\sum_{k''=a_{k'}}^{a_{k'+1}-1}\dfrac{r(b-a+1,k'')}{t}, & \text{when } j\le k'<l \\ & \text{and } a_{k'}\le k<a_{k'+1},\\ Y(k)+\displaystyle\sum_{k'=b+1}^{a_l-1}\dfrac{r(b-a+1,k')}{t}, & \text{when } a_{l-1}\le k<a_l,\\ Y(k), & \text{when } a_l\le k\le m, \end{cases} \end{aligned} \tag{38}$$

$(0\le k\le m)$ we obtain the total influence (all interactions) of a control (clone) $[a,b]$ into the clipping situation (CS array and its filling situation). If the control (clone) starts and stops in the same hour, we cannot directly apply (37). In this situation we have to replace the second, third and fourth lines of (37) with one line

$$Y(k)-W(b-a+1)/t+\sum_{k'=b+1}^{a_j-1} r(b-a+1,k')/t, \text{ when } a_{j-1}\le k<a_j. \tag{39}$$

If the payback (back radiation) starts and stops in the same hour, we have to make a correction similar to (38).

In function I of (37) the first line leaves the unaffected hours as they are. The second line calculates the hour where the control (clone) starts (either the overload decreases or the underload increases). The third line handles the hours that are between the starting and stopping hours, if any. The fourth line handles the hour where the control (clone) stops. The payback (radiation) starts here. The payback

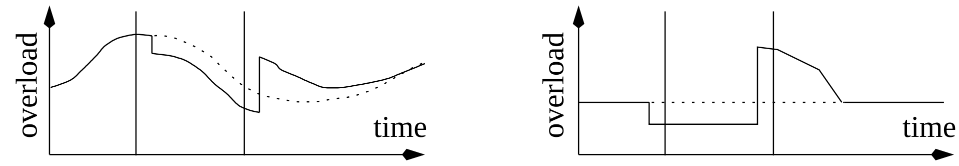

Figure 16: Realistic and theoretical clipping situation with a control.

(radiation) increases the overload (makes room into the containers) and the rest of the control (clone) decreases it (fills the containers). Hence, we use negative back radiation. The fifth line calculates the payback (radiation) for hours where the payback (radiation) continues and the sixth line deals with the last payback (radiation) hour. The last line leaves the other hours as they are.

Figure 16 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 (CS array and its filling situation) without control (clone) and the plain line is a clipping situation (CS array) with control (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 containers giving better profits).

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

$$Y'(C,Y) = \begin{cases} I([a,b],Y) + Y'(C-[a,b], I([a,b],Y)), & \text{where } [a,b] \in C \ (\neq \emptyset), \\ Y, & \text{if } C = \emptyset. \end{cases} \tag{40}$$

When all controls (clones) have been calculated, we can use (34) to find out the value of the new clipping situation (CS array and its filling situation).

Next we consider the constraints. First, the controls (clones) must be separate:

$$\forall [a,b],[a',b'] \in C : ([a,b] \neq [a',b'] \Rightarrow [a,b] \cap [a',b'] = \emptyset). \tag{41}$$

Further, there is a *resting time* related to each control (clone). During the resting time it is not allowed to start a new control (insert a new clone to the CS array). Function $L : \mathbb{N} \to \mathbb{N}$ is increasing and it maps the length of a control (clone) to the length of the corresponding resting time. So, we have

$$\forall [a,b],[a',b'] \in C : ([a,b] \cap [a',b'] = \emptyset \Rightarrow [b, b+L(b-a+1)] \cap [a',b'] = \emptyset). \tag{42}$$

This with (40) is equivalent to (45). Notice that here $[b, b+L(b-a+1)]$ denotes only the interval used for resting time.

We call resting time as *safety area* in interactive knapsacks (term is used in the algorithms). This is taken into account in (45). We also need the *minimum* and *maximum* control times (clone lengths) minc and maxc (we are using variable length clones), respectively, and hence

$$\mathrm{minc} \le c \le \mathrm{maxc}, \tag{43}$$

where c 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 (47).

We can suppose that at every time point (container) i the price factor $H(i)$ is larger than the income factor $T(i)$. By making controls (inserting clones) we can affect the clipping situation, so the optimization problem can be given in the form

$$\begin{aligned}
\max_{C_{ij}\in C} \quad & \sum_{i=0}^{m}\sum_{j=1}^{n} x_{ij} p_{ij}\big(Y'(C,Y)(i)\big) && (44)\\
\text{subject to} \quad & \sum_{j:\exists i: l\in I_{ij}} x_{ij} w_{ij} I_{ij}([i, i+c_{ij}], Y)(l) \le b_l && (45)\\
& x_{kj} = 0, \text{ for } i < k \le i + c_{ij} + L_j(c_{ij}), \text{ when } x_{ij} = 1, && (46)\\
& x_{ij} = 1 \text{ when } c_{ij} > 0, \text{ otherwise } 0, && (47)\\
& \sum_{i=1}^{m} x_{ij} \le K_j, && (48)\\
& \mathrm{minc}_j \le c_{ij} \le \mathrm{maxc}_j && (49)
\end{aligned}$$

where for free j's and i's we have $j = 1, \ldots, n$ and i's $i = 0, \ldots, m$. The sum $\sum_{i=0}^{m}\sum_{j=1}^{n} x_{ij} p_{ij}\big(Y'(C,Y)(i)\big)$ indicates the income lost and its theoretic maximum is 0. Notice 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 $c_{ij} > 0$.

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

7.2 Integer composition

In this subsection we present a heuristic solution to the load clipping problem by positioning and deciding the length of each clone separately for an item j at a time. Thus, we do not use the item index in this subsection. Figure 17 contains the main procedure. 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 $S = \{c_0, \ldots, c_q\}$ of clones ($q \in \mathbb{N}$). This algorithm is based on a composition algorithm given in [51]. Few minor modifications, however, are needed.

The idea is to compose integer m into $2k+1$ integers stored in array s and hence, to interpret each composition as a set of clones containing exactly k separate clones. We achieve this by restricting and interpreting the $2k+1$ integers in s in the following way: each even position has its integer from the set of the allowed lengths S and each odd position has free choice (≤ 0) for its value as long as the sum of all integers is m.

We interpret the set of clones (plan) in the following way. Our first clone (or group) according to the plan starts at s_1 and is of length s_2, the second clone starts at $s_1 + s_2 + s_3$ and is of length s_4, and so on. Here we have included the safety area in the length which have to be excluded from s_2 and s_4 to get the right clone. By inserting the safety area also to array s we avoid the need to check explicitly the safety area. On the other hand, set S is usually almost entire integer interval: some integers are missing as impossible combinations for length and safety area. The conversion is done in line (12) which also calculates the result of the plan contained in s (with equation (43)) and stores the best plan and its value. Details of line (12) are shown in Figure 18. By L^{-1} we mean the only choice for the safety area when inserting a clone of a certain length.

We will next prove that the algorithm in Figure 17 works correctly. Its time consumption will be also analyzed. We will show that the composition algorithm in Figure 17 produces all compositions in lexicographical order. The initialization occurring in lines (1)–(10) takes time $O(k)$. The array s has $2k+2$ different elements of which $2k+1$ are used as a storage for a composition. We set the index k' to $2k+1$ in the first line. The value at s_0 is used as a halting criterion (in line (11)). Array v contains $2k+1$ elements and each element corresponds to an index for the set S; we need only half of the elements. In the initialization phase only even indices of s take values from the set $S = \{c_0, \ldots, c_q\}$. Because we are constructing these compositions in lexicographic order, we initialize the even indices with c_0 except the s_0. The last value s'_k is initialized with $m - 2k'c_0$ so that the sum over s is m, i.e.

$$0, 0, c_0, 0, c_0, \ldots, 0, c_0, 0, c_0, m - 2k'c_0.$$

Each v_i is 0 because in each even position we have value c_0. Initialization takes time $O(k)$.

In line (12) we start with the initialized situation. The last composition in lexicographic order is

$$0, m - 2k'c_0, c_0, 0, c_0, \ldots, 0, c_0, 0, c_0, 0.$$

These conditions hold true because we cannot decrease the elements s_i ($2 \leq i \leq k'$)

Inputs: m, k and $S = \{c_0, \ldots, c_q\}$

$a = 0$, $k' = 2k + 1$, $s_0 = 0$
for $i = 1$ **to** $k' - 1$ **do**
    if i is even **then**
        $s_i = c_0$, $a = a + c_0$
    else
        $s_i = 0$
    end
    $v_i = 0$
od
$s_{k'} = m - a$, $v_{k'} = 0$
while $s_0 = 0$ **do**
    Count the objective from s
    Get the index i of s to move something from s_i to s_{i-1}
    if $i - 1$ is odd **then**
        $s_{i-1} = s_{i-1} + 1$ and $s_i = s_i - 1$
    else
        $a = c_{v_{i-1}+1} - c_{v_{i-1}}$
        $s_{i-1} = s_{i-1} + a$ and $s_i = s_i - a$
        $v_{i-1} = v_{i-1} + 1$
    end
    Form the lexicographical minimum from the elements at indices $[i, k']$ in s
od

Figure 17: The integer composition application giving plans.

$r = 0$, $C = \emptyset$
for $i = 1$ **to** $(k' - 1)/2$ **do**
    Form a new clone $C_i = [r + s_{2i-1},\ r + s_{2i-1} + s_{2i} - L^{-1}(s_{2i})]$
    $r = r + s_{2i-1} + s_{2i}$
od
Count the objective for $C = \{C_1, \ldots, C_{(k-1)/2}\}$
Store C and the result if better than any earlier result

Figure 18: Converting a composition of an integer into a clone set (plan).

in order to increase the element s_1. This composition is processed in line (12). We can assume that line (12) (see Figure 18) takes $O(k)$ time (the clones can be processed in order and their total length is at most $O(k)$; also L^{-1} can be counted beforehand and it takes linear time to traverse the array).

After finding the correct index (line (13)), we decide how much to increase s_{i-1} in lines (14)–(20). If $i-1$ is even we increase s_{i-1} by one, otherwise we increase it by $c_{v_{i-1}+1} - c_{v_{i-1}}$. We have to remove the same amount from s_i, and possibly update v_{i-1} to point some other item of S. After line (13), we must have $i = 1$, when only s_1 can be decreased (this is the last composition in lexicographic order). The position 0 is even, and hence, we increment s_0 and decrement s_1. Then after line (21), we exit the loop by condition at line (11), and the algorithm halts.

The change s_i to s_{i-1} is the smallest possible. There cannot be any number between the previous composition and the composition obtained in line (21). Hence, the algorithm starts from the first composition and produces all compositions in lexicographic order and eventually halts.

Now we have to consider lines (13) and (21) in detail. The details are given in Figures 19 and 20.

---

(1) $i = k$
(2) **while** (i is odd **and** $s_i = 0$) **or** (i is even **and** $s_i = c_0$) **do**
(3) $\quad i = i - 1$
(4) **od**
(5) **if** $i - 1$ is odd **then return** i **end**
(6) **if** $v_{i-1} < |S|$ **and** $s_i \geq c_{v_{i-1}+1} - c_{v_{i-1}}$ **then return** i **end**
(7) **return** $i - 1$

Figure 19: Return an index (line (13) of the algorithm in Figure 17).

The algorithm in Figure 19 starts from the right end of s and moves the index i to left as long as the values s_i are 0 at odd indices and c_0 at even indices, i.e. when those values are the lowest possible. After that the index contains the number of the first position from right end which can be decreased. We are done in line (5), if the $i-1$ is odd, because we can always add to positions not restricted by S.

If the process is still going on, we know that $i-1$ is even. The first condition in line (6) ensures that we can add to s_{i-1} by checking that it is not the maximal item c_q of S. The second condition ensures that the value of s_i is big enough, because we have to add to s_{i-1} the difference between successive items in S. 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.

However, if we cannot add to position $i-1$ and the condition in line (6) does not hold, we know that any amount can be added to position $i-2$ ($i-2$ is odd). In this case the index to be decreased is $i-1$. When handling the last composition, algorithm will return $i = 1$ in line (6), because the only element not containing the minimal value including s_0 is s_1 and we can always add to even index 0 and decrease the maximal element at s_1. In all other cases the index to be returned is larger than one.

Lines (2)–(3) take time $O(k)$ and the rest takes time $O(1)$. Hence, line (13) of the Algorithm in Figure 17 can be performed in $O(k)$ time.

Figure 20 shows the algorithm implementing line (21). It initializes the minimum composition in lexicographic order by calculating the values at indices $[i, k']$ and initializing the same indices ($[i, k']$). The algorithm consists of a loop gathering the values to a temporary variable and initializing everything at the way to the lowest values allowed. The last position is not restricted by S, and hence, we set $s_{k'}$ to be a. This routine takes time $O(k)$.

```
Inputs: i is the starting element and k' is the last index of arrays s and v.
a = 0,
while i ≤ k' do
        if i is odd then
                a = a + s_i, s_i = 0
        else
                a = a + s_i − c_0, v_i = 0, s_i = c_0
        end
        i = i + 1
od
s_k' = a
```

Figure 20: Forming the minimum (line (21) of the algorithm in Figure 17).

We have shown that the algorithm given in Figure 17 halts and forms each composition in lexicographic order. Hence, we have shown that our integer composition works correctly.

Theorem 19 *The algorithm in Figure 17 forms each integer composition in linear time.*

Proof. Initializations take $O(k)$ time, as well as lines (12), (13), and (21). Lines (14)–(20) take $O(1)$ time. Hence, time consumed between each composition is linear in k. □

Theorem 19 shows that enumerative solution is a pseudo-polynomial time algorithm for IKHD, if parameters of IKHD are fixed. (See the discussion after Corollary 14.)

7.3 Dynamic programming

The problem posed by equations (43)–(48) can be solved with dynamic programming (DP) [1, 2, 12, 56]. In [2] we show that an extra state variable is needed for the solution of [12], and in [1] we deepen the results thus obtaining faster DP (or more accurate, if wished). We follow here mainly [1] and give its main results. We consider similar approach as in the previous subsection: 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

$$h^*(Y) = \max_{[a,b]} h^*(Y'(C \cup [a,b], Y)). \tag{50}$$

In (49) the different clones form the state space and the number of clones form the stages. This solution enumerates different clones, and the state space is far too large. The dynamic programming formulation (49) at stage n is equal to the optimization problem

$$h^*(Y) = \max_{[a,b]_1} \dots \max_{[a,b]_n} h^*(Y'([a,b]_1 \cup \dots \cup [a,b]_n)). \tag{51}$$

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 (maxc$-$minc)m states, where minc is the minimum and maxc is the maximum length of a clone and m is the length of the CS 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 (maxc$-$minc)m states. The connections can point to approximately (maxc$-$minc)m states, so we should check about (maxc$-$minc)$^2m^2$ states, which is too much.

We could try to use only the best states from the previous stage. If we check only 20 states, we need approximately 20(maxc–minc)m checkings for each clone after first clone. In practice 20 best states is not enough, if we want to get quickly good economical results with equation (50). For example, the fifth clone would need approximately $4 \cdot 20$(maxc–minc)m state checkings plus the initial calculations for (maxc–minc)m states. Our DP uses less than $5 \cdot 11$(maxc–minc)m states and with Theorem 20 we can half this state space.

Cohen and Wang [12] use two state variables, the number of clones and the clone length. Our tests [1, 2] indicate that two state variable systems are so fast that we can add at least one state variable to improve results. To add a new state variable is a usual way to find better results in DP systems [4] (p. 30). In this work 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 [1]. There are many realistic situations where the extra variable gives more accurate results with economical significance. However, our three variable system (k, d, c) is not optimal (see cf. [1] and the NP-result given in this work).

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 (42) (minimum and maximum clone lengths). The variable clone length c actually contains the control's length obeying conditions (40)–(42) 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

$$(k, d, c)(i). \tag{52}$$

The phrase "stage i" refers to a container (or time point). In practice we have to determine an upper bound for the number of wait states d. Theorem 20 gives an upper bound for d when the item does not have radiation. If the item does have radiation, we assume that d can have $t-1$ (t is the length of an hour) different values. This is not enough, but in practice it is not possible to increase the number of wait states.

In the following theorem, C_i refers to a set of clones whose construction has started at stage i and not the set of clones of type i. (Ch. subsection 3.2 in [1].)

Theorem 20 ([1]) *Suppose that there is no radiation. Suppose further, that from stage a we start to construct a new clone which will stop locally maximizing according to set of clones C_a at stage b. Let i be the first moment of the hour containing a. Now it is enough to choose between the clone sets $C_i, C_{i+1}, \ldots, C_a$ when constructing a new control $[a, b]$.*

We denote $TI = (k, d, c)$ and $TI' = (k', d', c')$. 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. The function

$$h''(Y, TI', TI, i+1) = \begin{cases} 0, & \text{when (55)–(58)}, \\ h(Y''([i-c, i], Y)) - h(Y), & \text{when (60)}, \\ -H', & \text{when (59)}, \\ -\infty, & \text{otherwise}, \end{cases} \tag{53}$$

$(0 \leq i \leq m)$ gives the change in value, when moving from state (k, d, c) of stage i into the state (k', d', c') of the stage $i+1$. In other words, h'' tells the value of the connection between states (k, d, c) and (k', d, c') at stages i and $i+1$, respectively. 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 the cost of making a clone. The last line is used with every other values of the variables TI and TI'. They are impossible since they do not have any reasonable real world interpretation.

The dynamic forward recursion equation is

$$h'(Y, TI', i+1) = \max_{TI}(h'(Y, TI, i) + h''(Y, TI', TI, i+1)) \tag{54}$$

and

$$h'(Y, (k, d, c), 0) = \begin{cases} h(Y), & \text{when } k = d = c = 0, \\ -\infty, & \text{otherwise.} \end{cases} \tag{55}$$

When

$$\begin{aligned} &k' = k+1, \qquad d' = c' = 0, \qquad 0 \le d \le t-2, \qquad \text{and} \\ &\text{maxc} + L'(k, d, c, \text{maxc}+1) - 1 \le c \le \text{maxc} + L(\text{maxc}), \end{aligned} \tag{56}$$

we interpret that the clone at the stage i and in the 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 $L'(k, d, \text{maxc}+1)$ gives the minimum safety area $c - (\text{maxc}+1)$ stages ago for state (k, d, maxc) (see also the condition (60)). We do not use the last line of (55) or the condition (60) in the cases of clone lengths with safety area one, but we rather directly use the last line of condition (60). In the first line 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 the stage i does not change (be better) when we move to the stage $i+1$, and so

$$k' = k, \qquad d' = c' = 0 \qquad \text{and} \qquad d = c = 0. \tag{57}$$

This is the only case with conditions (55) and (60), when DP can make choices about the optimal path. If two paths give the same result, we choose in (53) 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. When

$$k' = k, \qquad d' = d+1 \qquad \text{and} \qquad c' = c = 0, \tag{58}$$

we "move some information from the past" to the present $(i+1)$. With this information we can check what result can be achieved, if we choose the best path d stages ago instead of some other set of clones of the interval $[i-d, i]$. When

$$k' = k, \qquad d' = d, \qquad c' = c+1 \qquad \text{and} \qquad (c' \neq 1, c' \neq \text{maxc}+1), \tag{59}$$

we increment the clone length (the clone started c containers ago). When

$$k' = k, \qquad d' = d \qquad \text{and} \qquad 1 = c' = c+1, \tag{60}$$

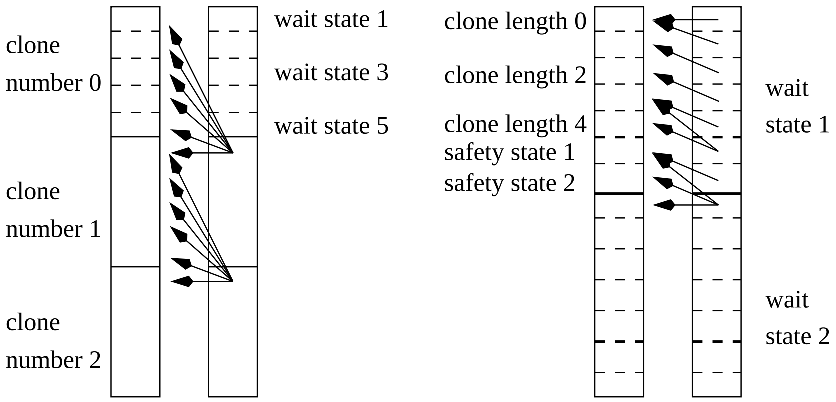

Figure 21: The structure of state space.

we have started a new clone. In this situation we add to the result the positioning cost of clone H'. When

$$k' = k, \qquad d' = d, \qquad c' = \text{maxc} + 1, \qquad \text{and} \qquad \text{minc} \le c \le \text{maxc}, \tag{61}$$

we can calculate the impact of a correct clone on the knapsack array. The clone length (the value of c) gives also the safety area $L'(k, d, c', \text{maxc} + 1) = L(c)$ $(0 < L' \le L(\text{maxc}))$, which is stored as long as it has its effect to the control. Figure 21 shows the state space.

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 from a *path*. When we have the values

$$h'(Y, (k, d, c), m)$$

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). The functions h' and h'' in equations (52)–(54) obey conditions (40)–(42).

7.4 Heuristics

This subsection is based on the work reported in [2]. We develop four heuristic methods for optimization and two different ways to handle the priorities. Priorities are used to choose items in some order, especially when optimizing one item at a time. In tests we show that these algorithms are much quicker than methods based on dynamical 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 tell how many iterative steps the algorithm in question should take. The “Direct action” and “Expensive down" algorithms use parameters by which one can control how these algorithms choose the items and containers 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 containers, i.e. overload intervals.

The algorithms have some common initial actions to be performed before optimization. These are shown in Figure 22. First we extend clones that are already made. The extension is made only when it increases the sum (43). If there are already some clones, then we might be able to fill the containers with this extension and so, we might manage without new clones. Recall that small overfill is much cheaper than small under fill in the containers. In the next lines we check if in the current hour there are under filled containers. The calling times of heuristics may restrict the current hour going on, and so we optimize it separately with a variant of "Direct action".

---

(1) Extend the clones C_j of items $j \in N = \{1, \dots, n\}$, if it improves the result
(2) Find the smallest i, for which $Y(i) > 0$
(3) **if** $i = 0$ **then**
(4)   Optimize the current hour
(5) **end**

Figure 22: "Initial actions for each heuristics."

In Figure 23 we show how "Direct action" works. In the first line we seek the first hour t with under filled containers and in the second line we divide items $\{1, \dots, n\}$ into two sets: to those that can be positioned to the container t and to those that cannot. If we do not find any item to be positioned, then in lines (3)–(8) we try to find an hour to which we can position some items.

If we can put a clone into the hour t in the CS array and if we have iteration steps left, then we perform lines (10)–(20). In line (11) we sort the items by the clone weights (control amounts) and the amount of the space $W_j(t)$ clone j fills in the hour t. Then we choose the clones from the list in ascending order until the sum of the total weight is more than the under fillment of the containers (overload in the hour).

When 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 container at hour t (or very near before it) and the halting times are at first set to the end of hour t (lines (13)–(14)).

When we choose starting times, we obey the constraints concerning the minimum

(1) Find the first hour t, for which $Y(t) > 0$

(2) Choose items $N' \subseteq N = \{1, \dots, n\}$ that can be positioned into hour t

(3) **if** $N' = \emptyset$ **then**

(4)   **while** $Y(t) > 0$ **and** $N' = \emptyset$ **do**

(5)   Find the next hour $t \leq m_t$ such that $Y(t) > 0$

(6)   Choose items $N' \subseteq N$ that can be positioned into hour t

(7)   **od**

(8) **end**

(9) **while** $Y(t) > 0$ **and** $N' \neq \emptyset$ **and** Count > 0**do**

(10)   Count = Count – 1

(11)   Sort the items $j \in N'$ by the clone weight $W_j(t)$

(12)   Choose items $N'' \subseteq N$ such that $\sum_{j \in N''} W_j(t) \geq Y(t)$

(13)   Set the first container (start) a for each clone $j \in N''$

(14)   Set the last containers (stop) b to the end of the hour t

(15)   Postpone each clone $[a, b]$ if it fills too empty containers

(16)   Update the containers at hour $Y(t)$ with clone weights $W_j(t)$

(17)   Choose the last container b for each clone $j \in N''$

(18)   Update the conditions, $N = N \cup N' \cup N''$

(19)   Extend the clones C_j of items $j \in N$

(20)   Perform the lines (1)–(8)

(21) **od**

Figure 23: "Direct action."

clone lengths. The container might be too filled (there is underload) with the used items. This fillment is moved by postponing clones $[a, b]$, i.e. by increasing both a and b (line (15)).

After these operations we update the containers with the clone weights (line (16)) and then try to postpone the halting times. This depends on each item's maximum length maxc_j, on the length of the under filled container interval, and on the parameter indicating appropriate overfilling level.

In line (18) we update the conditions, and in the next line (19) 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 24) we first look for the longest interval of under filled containers and check if some clones can be positioned into this interval. If not, lines (3)–(10) are executed. We start the optimization in chronological order. When we find an interval where we can position a clone, we jump to line (12).

If the length of the interval of under filled containers is only one hour, then we

(1) Find the longest interval with empty space $[s, t] \subseteq [0, m]$, out = false
(2) Choose items $N' \subseteq N = \{1, \ldots, n\}$ that can be positioned into hour s
(3) **if** $N' = \emptyset$ **then**
(4) Find the first empty space interval $[s, t] \subseteq [0, m]$ and set $s' = s$
(5) Choose items $N' \subseteq N$ that can be positioned into hour s
(6) **while** $Y(s) > 0$ **and** $N' = \emptyset$ **do**
(7) Find next empty space interval $[s, t] \subseteq [0, m] : s > s'$ and set $s' = s$
(8) Choose items $N' \subseteq N$ that can be positioned into hour s
(9) **od**
(10) **end**
(11) **while** $Y(s) > 0$ **and** $N' \neq \emptyset$ **and** Count > 0 **and not** out **do**
(12) Count = Count – 1
(13) **if** $t - s = 1$ **then**
(14) One iteration step with “Direct action”
(15) **else**
(16) Sort the items $j \in N'$ by the maximum length maxc_j
(17) Find the fullest containers, giving hour u from the interval $[s, t]$
(18) Choose the first $j \in N'$, for which Lit $Y(u) \leq W_j \leq$ Big $Y(u)$
(19) Choose the location of clone $[a, b]$ of j according to u
(20) Update the knapsack array Y
(21) Update the conditions, $N = N \cup N'$
(22) **if** Y did not change **then**
(23) One iteration step with “Direct action”
(24) **if** Y is still the same **then** out = true **end**
(25) **end**
(26) **end**
(27) Extend the clones C_j of items $j \in N$
(28) Perform the lines (1)–(10)
(29) **od**

Figure 24: “Long is nice.”

optimize it with “Direct action” (line (14)). Otherwise, we execute lines (16)–(25). We first sort the items by the maximum lengths maxc_k. In the next two lines ((17)–(18)) we look for an item that can be positioned to the fullest container in the interval inside the region determined by user’s parameters “Lit” and “Big”. By looking the fullest container 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

(1) Find the hour t, for which $Y' = Y(t) + Y(t+1)$ is biggest, out = false
(2) Choose items $N' \subseteq N = \{1, \ldots, n\}$ that can be positioned into hour t
(3) **while** $Y(t) > 0$ **and** Count **and not** out **do**
(4) Sort the items $j \in N$ by weights w_k
(5) Choose items $N'' \subseteq N'$ for which $j \in N''$: Lit $Y' \leq W_j \leq$ Big Y'
(6) **if** $N'' \neq \emptyset$ **then**
(7) Use each item in N'' once
(8) **if** Y' did not change **then**
(9) One iteration step with "Direct action" with all items $N \cup N' \cup N''$
(10) **if** Y' did not change **then** out = true **end**
(11) **end**
(12) **else**
(13) One iteration step with "Direct action" with all items $N \cup N' \cup N''$
(14) **if** Y' did not change **then** out = true **end**
(15) **end**
(16) $N = N \cup N' \cup N''$
(17) Extend the clones C_j of items $j \in N$
(18) Perform the lines (1) and (2)
(19) Count = Count – 1
(20) **od**

Figure 25: "Big first."

(19). If it is possible to position 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 the knapsack array Y with equation (39) and the conditions (lines (20)–(21)). In lines (22)–(25) we check if Y changed. If not, the iteration halts.

In the end of the algorithm (lines (27)–(28)) we just extend some clones, if 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 have to correct its actions rather often.

Figure 25 shows "Big first" heuristic. In the first two lines we seek the pair of consecutive hours with the emptiest containers and check if they can be filled. If in the clipping situation there is some single hour with empty containers which are emptier than in any other pair of consecutive containers, then we start with it.

Next we sort the items by their weights (line (4)). Then we choose items that fit to the parameters given by the user (line (5)). If there is no items chosen (line (6), $N'' = \emptyset$), then 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 ($N'' \neq \emptyset$), we form in line (7) a clone for each item just once. For each item $j \in N''$, we find out the largest overload pair $Y(t) + Y(t+1)$ the item j can cut. Some items $j \in N''$ that are otherwise suitable, cannot be positioned into the hour pair with the emptiest containers of the whole optimization interval Y, 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 containers 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 containers again so often as in "Long is nice".

"Expensive down" heuristic is shown in Figure 26. 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 position a clone (lines (1)–(2)). If we do not find any items that can be positioned into the most expensive hour (line (3)), then we start seeking as expensive hour as possible in lines (4)–(7).

In line (9) we seek the borders of the whole empty space interval which includes the hour t found earlier. If no t is found, then this line does nothing. If there is an interval where we can position a clone, then we perform lines (10)–(27).

First in line (15) we sort the items by the clone weights each clone can have for the hour t (restrictions might prevent some). Then we choose items like in "Direct action" heuristic, i.e. so that the sum of the clone weights is more than the empty space in the containers of this most expensive hour (line (16)). The place for each clone is formed like in "Long is nice" heuristic in lines (17)–(22). Line (21) is needed for checking if the borders of this empty space interval have changed. At the end (lines (24)–(26)) we perform the same tasks as in the other heuristics.

(1) Find the hour t, for which $H(t)Y(t)$ is the biggest

(2) Choose items $N' \subseteq N = \{1, \ldots, n\}$ that can be positioned into hour t

(3) **if** $N' = \emptyset$ **then**

(4)   **while** $Y(t) > 0$ **and** $N' = \emptyset$ **do**

(5)   Find an unhandled hour t with the biggest $H(t)Y(t)$ value

(6)   Choose items $N' \subseteq N$ that can be positioned into hour t

(7)   **od**

(8) **end**

(9) Find the borders s, u of empty space intervals, i.e. $t \in [s, u]$, usable = true

(10) **while** $Y(t) > 0$ **and** usable **and** Count **do**

(11)   **if not** usable **then**

(12)   One iteration step with "Direct action" with all items $N \cup N' \cup N''$

(13)   **if** Y' did not change **then** usable = false **end**

(14)   **else**

(15)   Sort the items $j \in N$ by the clone weights $W_j(t)$

(16)   Choose items $N'' \subseteq N'$ such that $\sum_{j \in N''} W_j(t) \geq Y(t)$

(17)   **for** $j = 1$ **to** $|N''|$ **do**

(18)   Choose the location of clone $[a, b]_j$ according to t

(19)   Update the knapsack array $Y(t)$ with $W_j(t)$

(20)   Update the conditions, $N = N \cup \{j\}$

(21)   Find the borders s, u of empty space intervals, i.e. $t \in [s, u]$,

(22)   **od**

(23)   **end**

(24)   Extend the clones C_j of items $j \in N$

(25)   Perform the lines (1)–(9)

(26)   Count = Count – 1

(27) **od**

Figure 26: "Expensive down."

The heuristics may be used directly as they are shown above or with additions that take in 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 containers, the items from the next level are used. This continues until the empty containers are filled or all usable items are handled. This priority method is called "start".

Some items with low priorities may not optimize some hours as well as items with high priorities. In these cases the items used earlier 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 the beginning. This means that we do not use the already made set of clones and at the same time, we do not need the checking described so often. We call this method "extending". This method needs time proportional to at least the square of the number of items while linear time is sufficient for the first heuristic.

We also implemented a method combining the advantages of all these heuristics. This method just reads the instructions: the number of repetitions, in which order and with which parameters each heuristic is used.

7.5 Genetic algorithms

Genetic algorithms (GA's) are used efficiently with many different and noteworthy difficult combinatorial problems [42, 50, 28]. As we have already pointed out, GA's are used with unit commitment and other problems related to electricity management [15, 17, 37, 29].

The basic idea behind GA is taken from the evolutionary biology [42]. Figure 27 contains the fundamental structure of a *simple GA*.

(1) Initialize population
(2) **while not** termination **do**
(3)   Produce new individuals
(4)   Insert new individuals into the population
(5) **od**
(6) Report on results

Figure 27: Structure of simple GA.

Usually one codes an *individual* as a bit vector, initializes it in line (1) and applies different "generation" 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 consists of tightly connected, larger parts, i.e. genes form "building blocks" 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.

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. (Cf. [42].)

We use similar data structure for GA as described for the enumerative solution in subsection 7.2 instead of bit vector coding. Each gene has, for a clone, an integer pair (x, y), where x is the number of containers before the clone containing no other clones, and y is the length of the clone. This integer pair coding has the advantage that we can automatically keep some of the restrictions in order, like the safety area. Table 3 contains twelve genes as an example.

Table 3: Example of an individual.

(5,10)	(15, 5)	(5,20)	(15,5)	(5,10)	(15,5)
(10,5)	(5, 5)	(30,5)	(15,5)	(5,10)	(15,5)

An individual is either a set of clones of one type (IKHD) or a class of clone sets (IKD or MDCS). Table 3 has six individuals for IKHD. We may interpret each gene pair as an individual, the upper genes being of type one and the genes on the bottom of type two. For IKD there could be three individuals, left, center and right gene pairs. In load clipping problem we test only the IKHD approach (see section 8).

First we describe the methods implemented for the IKHD problem where each individual is a set of clones.

We implemented two different crossovers. In the first one we randomly select the number of genes two parents will have and swap their tails. 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 3 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 three last clones were on the optimal place before crossover, they both are in non-optimal place after crossover.

In the second version we also select a random swapping point but this time from the whole optimization interval. In the example from interval [1,115]. Then we interpret the clones so that when one container 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. 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 and will be studied later (tabu search is out of the scope in this work).

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 is performed a random choice about adding a new gene (i.e. a new clone). After that follows a random choice about removing the gene (clone) in question. Thus, our individuals are of different length. If a gene is not removed, we randomly choose between adding or subtracting either the first or the second number in the gene thus corresponding 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.

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 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.

The behavior of different combinations of the genetic operations is discussed in greater detail in section 8.

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.

Notice, that we have considered load clipping as static and stationary optimization problem in other methods presented so far. However, load clipping is essentially non-stationary because the clipping situation changes as time passes by. Genetic algorithms for non-stationary function optimization are considered in [16, 21, 45].

Because the non-stationary 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. We justify the droppings 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 [35]. 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 GA's [22] are very helpful when taking into account the stochastic and non-stationary 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 GA's can help in our problem also 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. The parallel GA's are not in the scope of this work.

8 Tests

For every problem instance there theoretically exists a set of items by which the containers can be filled so that there is no loss of income (i.e. the result is zero). Here we count only the losses of income; they are negative reals obtained by equation (43). All tests are driven in Pentium90 with Linux.

In the tests we have three versions of DP: DP1, DP2 and DP3. The DP3 uses Theorem 20 while the DP1 and DP2 do not. The DP1 sorts the items by priorities (prior key) and clone weights (secondary key). The DP2 sorts items by clone weights along.

All methods implement an extra restriction not described in the text (sum of clone lengths is restricted). The DP's cannot take this restriction directly into account [2] and hence we used three different approach to handle the restriction. This causes the differences in the tests with one item at a time. The different sorting methods also cause the differences in the tests having several items. With similar sortings and approaches to this extra restriction all DP's would give same economical results. The DP1 and DP2 would also give approximately same execution times.

First we present five problem instances (the ongoing hour and the next 24 hours) in order to study the capabilities of the algorithms when using several items. Then we do four additional tests, of which three are simulating "normal" load clipping instances and the fourth is an artificial but easy problem. Then we study the behavior of the algorithms with only one item at a time. This is of interest because DP's can produce clones only for one item at a time. Also our GA's operate for one item at a time, although they can be easily extended to handle several items.

In all tests we give for each instance a starting result and the *optimal* result. The starting result is determined directly by equation (43). In the optimal result we also use equation (43) with the difference that we count just the hours with underload (i.e. overfilled containers). This means that the optimum is obtained with respect to empty space intervals that can be filled (overloads that can be cut off) precisely without any extra container filling obtained by radiation or by inserting items. This means also that the optimal result is not the best possible result (but good enough in practice).

In the problem instance tables (e.g. Tables 4, 6 and 8) positive numbers indicate under fillment (overload) and the negative ones over fillment (underload). Recall that each hour consist of twelve containers and these numbers are averaged in each hour. The eigth items used here have their weights between 0.2 to 7.0. In some tests we have used only a subset of the eigth items. In the result tables there are two lines for each heuristic. The priorities column shows which priority method was used and the maximum priority used in the result (the lower the better). In tests the price of overload is 7000 and the income is 50. With different prices and incomes in different hours, DP's and "Expensive down" would give different results.

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	2.010

Table 4: Problem instance (starting result –74015, optimum –235.5).

In the first test (Tables 4 and 5) we have five overload (empty space) intervals with different lengths and with relatively small overloads. The results show that heuristics work very fast and DP's run fast enough. The results are not close to the optimal except with "Direct action". This is caused by the overcutting (or in clone terms, overfilling) property of DP's.

The second test (Tables 6 and 7) is good for "Long is nice" heuristic. The other heuristics tend to use their lenghtly clone weights to the first overload interval while it should be saved for the longest overload interval. These situations are always dependent on the items used.

The third test (Tables 8 and 9) is similar to the second one. The only heuristic performing well is "Big first". Other heuristics waste their resources in wrong places. This is also very dependent on the set of items and their properties and of course on the whole clipping situation.

In the fourth test (Tables 10 and 11) we have a very difficult clipping situation. In every hour there is overload (containers are under filled). The economical savings are rather good (from –331100 to –2284 in 6 seconds and to -96 in 11 minutes). The execution times for DP's are just a little bit too long in every situation, but one may buy some extra time with quick heuristics and then use DP's.

The next three problems given in Tables 12, 15 and 18 simulate "normal" problem instances occurring in load clipping. The first one contains morning and afternoon peaks, the second has an afternoon peak and the last one has a morning peak, where a peak refers to a situation with containers under filled. After these three tests we have an artificial test having under filled containers (i.e. overload) in every hour in

Algorithm	**Priorities**		**Result**	**Execution Time**
Direct action	start	3	**–236**	1 sec.
	extending	3	–281	1 sec.
Long is nice	start	3	–251	1 sec.
	extending	3	–279	1 sec.
Expensive down	start	3	–277	1 sec.
	extending	3	–279	1 sec.
Big first	start	3	–268	1 sec.
	extending	3	–280	1 sec.
DP 1		3	–288	4 min. 55 sec.
DP 2		4	–301	2 min. 15 sec.

Table 5: Results for Table 4.

–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	–1.000

Table 6: Problem instance (starting result –95950, optimum –750).

Algorithm	**Priorities**		**Result**	**Execution Time**
Direct action	start	4	–26503	1 sec.
	extending	4	–43745	1 sec.
Long is nice	start	4	**–3042**	1 sec.
	extending	4	**–3823**	1 sec.
Expensive down	start	4	–26173	1 sec.
	extending	4	**–10822**	1 sec.
Big first	start	4	–44751	1 sec.
	extending	4	–44737	1 sec.
DP 1		3	–5068	1 min. 50 sec.
DP 2		3	**–1289**	1 min. 50 sec.

Table 7: Results for Table 6.

–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	–1.000

Table 8: Problem instance (starting result –95950, optimum –900).

Algorithm	**Priorities**		**Result**	**Execution Time**
Direct action	start	3	–68470	1 sec.
	extending	3	–68470	1 sec.
Long is nice	start	3	–31079	1 sec.
	extending	3	–36003	1 sec.
Expensive down	start	3	–68479	1 sec.
	extending	3	–41423	1 sec.
Big first	start	3	**–21402**	1 sec.
	extending	3	–68470	1 sec.
DP 1		3	**–15070**	1 min. 15 sec.
DP 2		3	**–23947**	1 min. 15 sec.

Table 9: Results for Table 8.

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	1.000

Table 10: Problem instance (starting result –331100, optimum 0).

Algorithm	**Priorities**		**Result**	**Execution Time**
Direct action	start	5	**–2284**	**6** sec.
	extending	4	–15117	**2** sec.
Long is nice	start	5	–16025	25 sec.
	extending	4	–20068	13 sec.
Expensive down	start	5	–14095	10 sec.
	extending	4	**–4917**	14 sec.
Big first	start	5	**–4645**	12 sec.
	extending	4	–12905	8 sec.
DP 1		5	**–3081**	8 min. 10 sec.
DP 2		5	**–96**	10 min. 40 sec.

Table 11: Results for Table 10.

–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	–1.000

Table 12: Problem instance (starting result –84600, optimum –600).

Table 21.

Tables 13, 16, 19 and 22 give results and time consumptions for each heuristic, DP1 and DP2. We left out DP3 and GA's because they are mainly designed to work with one item at a time — we test them separately. The best results are shown in detail in Tables 14, 17, 20 and 23.

In the test given in Table 12 "Long is nice" was a bit surprisingly the best method giving nearly optimal result. The results are near to each other and the economical difference is not meaningful at least for a small electricity supplier. The DP's suffer here from their approach to choose only one item at a time thus leading to overfilling of containers (see Table 14). As earlier, the heuristics are much faster than DP's.

In the test given in Table 15 we have an "afternoon peak". This time DP's and "Direct action" gave almost same result which is again almost optimal.

In the test given in Table 18 we have "morning peak" and this time "Expensive down" was the best method. It used a group from priority level 3 which is not always considered as a good thing if one can manage the situation with lower priority items, even if it gives better economical results.

The test given in Table 21 is the artificial one. Here the DP's found better results than any of the heuristics. Further, DP1 was able to cope with priority 1 group. DP2 gave the best result. However, DP2 needed 17 clones to produce the solution and if each clone costs something, the results may not be so good as some heuristics give.

In the next tests we use only one item at a time. The heuristics, dynamic programming and genetic algorithm are run with three different items for the four previous (in Tables 12, 15, 18 and 21) and three other tests (in Tables 32, 35 and 38). We test also the DP3 which uses the state saving property approach provided by the Theorem 20. The items differ mainly on the length of maximum clone length: the first item has ten times longer maximum length than the second, which in turn, has length of an hour. The third item is somewhere between these two items. The used items also have a bit different clone weights (controlling capacities). Radiation is not used in these tests.

We included two versions of GA, one having 400 generations and the other having 800 generations. Number of generations is used as a termination condition and hence

Algorithm	**Priorities**		**Result**	**Execution Time**
Direct action	start	3	–663	< 1 sec.
	extending	3	–624	< 1 sec.
Long is nice	start	3	**–608**	< 2 sec.
	extending	3	–635	< 1 sec.
Expensive down	start	3	–665	< 1 sec.
	extending	3	–631	< 1 sec.
Big first	start	3	–666	< 1 sec.
	extending	3	–634	< 1 sec.
DP 1		3	–666	4 min. 40 sec.
DP 2		4	–618	2 min. 20 sec.

Table 13: Results for Table 12.

–1.000	–1.000	–1.000	–0.009	–0.036	–0.011	–0.117	–0.138	–0.045
	–0.500	–1.000	–1.000	–1.000	–1.000	–1.000	–0.500	0.000
	–0.011	–0.020	–0.056	–0.043	–0.014	–0.670	–1.000	–1.000

Table 14: Best solution for problem 12.

–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	–1.000

Table 15: Problem instance (starting result –57550, optimum –850).

Algorithm	**Priorities**		**Result**	**Execution Time**
Direct action	start	3	**–860**	< 1 sec.
	extending	3	–875	< 1 sec.
Long is nice	start	3	–867	< 1 sec.
	extending	3	–902	< 1 sec.
Expensive down	start	3	–884	< 1 sec.
	extending	3	–877	< 1 sec.
Big first	start	3	–901	< 1 sec.
	extending	3	–885	< 1 sec.
DP 1		3	**–859**	3 min. 50 sec.
DP 2		4	–862	45 sec.

Table 16: Results for Table 15.

–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	–0.033	–0.033
	–0.006	–0.080	–0.075	–0.039	–0.023	–0.030	–0.870	–1.000

Table 17: Best solution for problem 15.

–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	–1.000

Table 18: Problem instance (starting result –35850, optimum –850).

Algorithm	**Priorities**		**Result**	**Execution Time**
Direct action	start	2	–890	< 1 sec.
	extending	2	–890	< 1 sec.
Long is nice	start	2	–890	< 1 sec.
	extending	2	–890	< 1 sec.
Expensive down	start	3	**–883**	< 1 sec.
	extending	2	–890	< 1 sec.
Big first	start	3	–899	< 1 sec.
	extending	2	–890	< 1 sec.
DP 1		2	–908	4 min. 50 sec.
DP 2		4	**–884**	2 min.

Table 19: Results for Table 18.

–1.000	–1.000	0.000	–0.062	–0.039	–0.132	–0.012	–0.391	–0.045
	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	–1.000

Table 20: Best solution for problem 18.

0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500
	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500
	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500

Table 21: Problem instance (starting result –87500, optimum –0).

Algorithm	**Priorities**		**Result**	**Execution Time**
Direct action	start	3	–279	< 1 sec.
	extending	2	–281	< 1 sec.
Long is nice	start	3	**–212**	< 1 sec.
	extending	2	–281	< 1 sec.
Expensive down	start	2	–281	< 1 sec.
	extending	2	–281	< 1 sec.
Big first	start	3	–266	< 1 sec.
	extending	2	–281	< 1 sec.
DP 1		**1**	**–188**	4 min. 40 sec.
DP 2		4	**–161**	10 min. 10 sec.

Table 22: Results for Table 21.

–0.040	–0.015	–0.012	–0.220	–0.072	–0.244	–0.072	–0.244	–0.072
	–0.244	–0.072	–0.244	–0.072	–0.244	–0.072	–0.244	–0.018
	–0.027	–0.030	–0.059	–0.184	–0.005	0.004	–0.100	–0.020

Table 23: Best solution for problem 21.

gives a rough approximation of the time usage: we chose the number of generations so that the running times are about 30 seconds and one minute. Each test case was run five times and results, both time and economical results, are averages over these runs.

Other GA parameters besides number of generations are: tournament parameter 0.85, elitism 65, size of population 100, crossover probability 0.05, and mutation parameters 0.05 for adding or deleting a clone and 0.2 for 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 the GA change its parameters and both mutation and crossover operator use the mapping to binary vector and back in the operation.

The GA's were tested also separately to find these parameters. We tried all six combinations of the mutation and crossover operators implemented. In the best combination found we used mapping to bit vector both in mutation and in crossover.

The combination where both mutation and crossover operated directly with compact individual representation was clearly the worst. The main problem is that if a small change happens in the head, the tail will change accordingly. If the tail is in optimal position, the change moves it away. This is against the "building block hypothesis" [42]: the optimal tail should not move on local change somewhere else. The crossover can have a large impact on tails, too.

Other combinations usually did find a locally optimal solution. There was no big difference in the average scores for each generation, either. 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 parameters also lead only to situation, where the parameters were converging to zero and the average scores were poor. The local optimum, however, was often found.

Each test is reported in three tables, the first one contains the problem instance, the second execution times and economical results for each item and the third table has the best results for each item. In the second table two first columns (Execution time and Results) are for first solution in third table, the second Execution-Result column pair for second solution in third table and two last columns for third solution in the third table.

The problem instances are given in Tables 12, 15, 18, 21, 32, 35 and 38. Result Tables are 24, 26, 28, 30, 33, 36 and 39 and best results are in Tables 25, 27, 29, 31, 34, 37 and 40.

In the test given in Table 12 almost all methods gave same result. Also the GA's found almost the same result. The major difficulty with GA's at the moment is to fine tune the clones to local maxima; this phenomenon shows up also in the other tests. It is also noteworthy that DP3 used only one third of time needed by other DP's. DP3 used five clones (37,43) (46,86) (90,96) (193,199) (202,260) to achieve its result while heuristics needed only two (40,92) (196,260).

For the second item (with short maximum clone length) the best result was given with the DP1 and DP3. If we allow the DP3 to make as many clones as it wishes, it makes nine clones and achieves the best result (–42326). The GA's were performing well against the heuristics, only "Expensive down" gave better result.

For the third item the best economical results were obtained by DP's (all same). However, the result can be reached by different clone sets: DP2 used (36,42) (54,90) (197,221) (233,259) while DP3 used (36,42) (54,90) (197,222) (234,259). DP3 was also several time faster than DP2. In this test the both GA's were clearly better than the heuristics. In this test the average result for GA400 is better than for GA800. This is explained with the low number of test runs. One should run tests more than five time.

In the test given in Table 15 the first item was best optimized with DP's. The DP3 was allowed to use too few clones to optimize this case well. This time GA's performed as well as heuristics. "Direct action" needed only one clone (184,266) while DP1 used three clones (184,200) (204,253) (257,266).

The second item was also best with DP's. DP3 used again less time than the

Algorithm	**Exec.**	**Result**	**Exec.**	**Result**	**Exec.**	**Result**
Dir. action	< 0.1 s.	–31407	< 0.2 s.	–58020	< 0.1 s.	–40110
Long is nice	< 0.1 s.	–31407	< 0.1 s.	–56892	< 0.1 s.	–40110
Exp. down	< 0.1 s.	–31407	< 0.2 s.	–55200	< 0.2 s.	–38294
Big first	< 0.1 s.	–31407	< 0.1 s.	–56046	< 0.2 s.	–40110
DP 1	3 m. 2 s.	–31407	20 s.	–49320	55 s.	–33381
DP 2	2 m. 55 s.	–31407	17 s.	–55200	2 m. 35 s.	–33381
DP 3	**1 m.**	–31407	7 s.	–49320	15 s.	–33381
GA400	27 s.	–31414	46 s.	–55480	34 s.	–35238
GA800	55 s.	–31413	1m. 35 s.	–55203	1 m. 10 s.	–35788

Table 24: Results for Table 12.

–1.000	–1.000	–1.000	–0.033	0.200	0.700	0.200	–0.033	0.000
	–0.500	–1.000	–1.000	–1.000	–1.000	–1.000	–0.500	–0.030
	0.700	1.200	1.200	0.200	–0.033	–1.000	–1.000	–1.000
–1.000	–1.000	–1.000	–0.060	0.300	1.080	0.440	–0.060	0.000
	–0.500	–1.000	–1.000	–1.000	–1.000	–1.000	–0.500	0.010
	0.730	1.580	1.510	0.230	0.080	–1.000	–1.000	–1.000
–1.000	–1.000	–1.000	0.014	0.514	0.528	0.028	0.014	0.000
	–0.500	–1.000	–1.000	–1.000	–1.000	–1.000	–0.500	–0.067
	0.528	1.595	1.433	0.028	–0.067	–1.000	–1.000	–1.000

Table 25: Best solutions found for problem in Table 12.

Algorithm	Exec.	Result	Exec.	Result	Exec.	Result
Dir. action	< 0.1 s.	–23983	< 0.1 s.	–30325	< 0.1 s.	–29394
Long is nice	< 0.1 s.	–23983	< 0.1 s.	–32708	< 0.1 s.	–28260
Exp. down	< 0.1 s.	–23983	< 0.2 s.	–33198	< 0.1 s.	–31169
Big first	< 0.1 s.	–23983	< 0.2 s.	–30815	< 0.1 s.	–29394
DP 1	2 m. 30 s.	–23957	15 s.	–29902	45 s.	–26161
DP 2	2 m. 25 s.	–23957	13 s.	–29902	1 m. 25 s.	–26161
DP 3	35 s.	–24890	4 s.	–29902	11 s.	–26161
GA400	25 s.	–23983	38 s.	–30464	29 s.	–28117
GA800	55 s.	–23989	1m. 25 s.	–30476	1 m.	–27909

Table 26: Results for Table 15.

–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	–0.033	–0.033
	0.700	1.200	1.200	0.200	–0.033	–0.033	–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	–1.000	–1.000	0.010	0.080
	0.730	1.510	1.580	0.230	0.010	–0.040	–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	–1.000	–1.000	0.014	0.014
	0.528	1.433	1.595	0.028	–0.472	–0.062	–1.000	–1.000

Table 27: Best solutions found for problem in Table 15.

other DP's. Now also GA's performed better than the heuristics. The third item behaves similarly: DP3 is relatively even quicker this time and the GA's are better than the heuristics.

In the test given by Table 18 the results are similar to the previous test. For the first item the GA's are the worst, but on the other hand, the differences are marginal: the same optimization problem for the third item shows how far from the optimal result the heuristics can be — here the GA's worked very well. The local maxima was not achieved because of lack of tuning capability.

In the test given in Table 21 the DP's performed clearly better than the other methods (see results in Table 30). DP3' was allowed to use as many controls as wished and it used 25 for the first item, 21 for the second and nine for the third.

This way it was able to fill the containers so that there was no any under fillment (overload) left for the first item. The other items had little under fillment but not

Algorithm	**Exec.**	**Result**	**Exec.**	**Result**	**Exec.**	**Result**
Dir. action	< 0.1 s.	–8568	< 0.1 s.	–14646	< 0.1 s.	–15174
Long is nice	< 0.1 s.	–8568	< 0.1 s.	–15626	< 0.1 s.	–15174
Exp. down	< 0.1 s.	–8568	< 0.1 s.	–14713	< 0.1 s.	–15249
Big first	< 0.1 s.	–8568	< 0.1 s.	–15136	< 0.1 s.	–15174
DP 1	3 m. 30 s.	–8555	20 s.	–13599	55 s.	–8539
DP 2	3 m. 20 s.	–8555	35 s.	–13599	40 s.	–8539
DP 3	1 m. 8 s.	–8555	7 s.	–13599	16 s.	–8539
GA400	25 s.	–8575	33 s.	–14421	29 s.	–8831
GA800	55 s.	–8572	1 m. 10 s.	–14296	1 m.	–8567

Table 28: Results for Table 18.

–1.000	–1.000	–0.033	–0.033	0.200	0.700	0.200	–0.033	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	–1.000
–1.000	–1.000	–0.060	–0.060	0.440	1.080	0.300	–0.060	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	–1.000
–1.000	–1.000	–0.067	0.014	0.514	0.528	0.028	0.014	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	–1.000

Table 29: Best solutions found for problem in Table 18.

Algorithm	Exec.	Result	Exec.	Result	Exec.	Result
Dir. action	< 0.1 s.	–3558	< 0.2 s.	–59369	< 0.1 s.	–41315
Long is nice	< 0.1 s.	–3558	< 0.1 s.	–60215	< 0.1 s.	–41315
Exp. down	< 0.1 s.	–3558	< 0.1 s.	–60215	< 0.2 s.	–38932
Big first	< 0.1 s.	–3558	< 0.1 s.	–60215	< 0.2 s.	–41315
DP 1	4 m. 40 s.	–188	20 s.	–52220	55 s.	–32473
DP 2	4 m. 30 s.	–188	17 s.	–58100	1 m. 35 s.	–32473
DP 3	1 m. 24 s.	–3753	8 s.	–52220	16 s.	–32473
DP 3'	2 m. 10 s.	**–42**	23 s.	**–3999**	30 s.	**–5216**
GA400	40 s.	–983	50 s.	–55322	54 s.	**–30899**
GA800	1 m. 15 s.	–1408	1 m. 45 s.	–55302	2 m. 15 s.	**–26201**

Table 30: Results for Table 21.

–0.033	–0.033	–0.033	–0.033	–0.033	–0.033	–0.033	–0.033	–0.033
	–0.033	–0.033	–0.033	–0.033	–0.033	–0.033	–0.033	–0.033
	–0.033	–0.033	–0.033	–0.033	–0.033	–0.033	–0.033	–0.033
–0.060	–0.060	–0.060	0.080	0.010	0.010	0.010	0.010	0.010
	0.010	0.080	0.010	0.010	0.010	0.010	0.010	0.010
	0.080	0.010	0.010	0.010	0.010	0.010	0.010	0.150
0.500	0.014	0.014	0.014	0.014	–0.067	0.500	0.500	0.014
	0.014	0.500	0.500	0.176	0.095	0.014	–0.472	0.014
	0.014	0.014	0.014	0.095	0.338	0.014	0.014	–0.067

Table 31: Best solutions found for problem in Table 21.

particularly much. GA's worked better than any of the heuristics with all three items: they even outperformed DP's with maximum number of clones set. The GA's were able to find that many short clones (ten) is better for this case than fewer maximum clone length clones. In the result table the third result is the best of five GA800 results, giving –23732.

In the test given in Table 32 we have five short and low empty space intervals (overload intervals) thus stating an easy problem. Almost all results give local maximums but the GA's once more could not fine tune them self. For the second item DP3' was able to find better result by using more clones than the others.

The problem given in Table 35 is easier than the previous problem. It contains only three empty space intervals and they are smaller than in the previous problem instance. This time all three items could fill in the empty containers. For the second

–1.000	2.000	–1.000	–1.000	–1.000	1.000	1.000	–1.000	–1.000
	–1.000	1.500	1.500	0.500	0.000	–1.000	–1.000	0.500
	0.500	–1.000	–1.000	–1.000	1.000	–1.000	–1.000	–1.000

Table 32: Problem instance (starting result –67250, optimum –750).

Algorithm	**Exec.**	**Result**	**Exec.**	**Result**	**Exec.**	**Result**
Dir. action	< 0.2 s.	–23155	< 0.1 s.	–37850	< 0.1 s.	–15936
Long is nice	< 0.2 s.	–23155	< 0.1 s.	–38273	< 0.2 s.	–15936
Exp. down	< 0.2 s.	–23155	< 0.1 s.	–37850	< 0.2 s.	–15936
Big first	< 0.2 s.	–23155	< 0.2 s.	–37850	< 0.2 s.	–15936
DP 1	3 m. 55 s.	–23155	20 s.	–31970	55 s.	–22740
DP 2	3 m. 50 s.	–23155	17 s.	–37850	3 m. 45 s.	–15936
DP 3	1 m. 2 s.	–23275	7 s.	–31970	16 s.	–22740
DP 3’	1 m. 15 s.	–23162	23 s.	**–27916**	25 s.	–15936
GA400	33 s.	–23281	41 s.	–37850	35 s.	–15951
GA800	1 m. 5 s.	–23290	1 m. 30 s.	–37844	1 m. 10 s.	–15943

Table 33: Results for Table 32.

–1.000	1.200	–1.000	–1.000	–1.000	0.200	0.200	–1.000	–1.000
	–1.000	0.700	0.700	–0.033	0.000	–1.000	–1.000	–0.033
	–0.167	–1.000	–1.000	–1.000	0.200	–1.000	–1.000	–1.000
–1.000	1.160	–1.000	–1.000	–1.000	0.160	0.580	–1.000	–1.000
	–1.000	0.660	1.150	0.010	0.000	–1.000	–1.000	–0.060
	–0.060	–1.000	–1.000	–1.000	0.160	–1.000	–1.000	–1.000
-1.000	1.028	-1.000	-1.000	-1.000	0.028	0.028	-1.000	-1.000
	-1.000	0.528	0.528	-0.067	0.000	-1.000	-1.000	-0.067
	-0.067	-1.000	-1.000	-1.000	0.028	-1.000	-1.000	-1.000

Table 34: Best solutions found for problem in Table 32.

–1.000	–1.000	0.500	0.500	–1.000	–1.000	–1.000	0.500	–1.000
	–1.000	–1.000	–1.000	0.500	–1.000	–1.000	–1.000	–1.000
	–1.000	–1.000	–1.000	–1.000	–1.000	–1.000	–1.000	–1.000

Table 35: Problem instance (starting result –15050, optimum –1050).

Algorithm	Exec.	Result	Exec.	Result	Exec.	Result
Dir. action	< 0.1 s.	–1057	< 0.1 s.	–1633	< 0.1 s.	–1063
Long is nice	< 0.1 s.	–1057	< 0.1 s.	–1633	< 0.1 s.	–1063
Exp. down	< 0.1 s.	–1057	< 0.1 s.	–1633	< 0.1 s.	–1063
Big first	< 0.1 s.	–1057	< 0.1 s.	–1633	< 0.1 s.	–1063
DP 1	2 m. 40 s.	–1057	20 s.	–1062	55 s.	–1063
DP 2	2 m. 30 s.	–1057	17 s.	–1062	1 m. 35 s.	–1063
DP 3	1 m. 16 s.	–1090	7 s.	–1062	15 s.	–1063
GA400	28 s.	–1148	31 s.	–1288	29 s.	–1066
GA800	1 m.	–1140	1 m.	–1076	1 m.	–1067

Table 36: Results for Table 35.

–1.000	–1.000	–0.033	–0.033	–1.000	–1.000	–1.000	–0.033	–1.000
	–1.000	–1.000	–1.000	–0.033	–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	–0.060	–0.060	–1.000	–1.000	–1.000	–0.060	–1.000
	–1.000	–1.000	–1.000	–0.060	–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	–0.067	–0.067	–1.000	–1.000	–1.000	–0.067	–1.000
	–1.000	–1.000	–1.000	–0.067	–1.000	–1.000	–1.000	–1.000
	–1.000	–1.000	–1.000	–1.000	–1.000	–1.000	–1.000	–1.000

Table 37: Best solutions found for problem in Table 35.

item the GA's were performing clearly better than the heuristics.

The test given in Table 38 is very artificial: there are equal amount of under fillment and over fillment alternating about half of the hours in the problem.

The DP3 was much faster that other DP's. The local maxima was achieved by all the methods for the second item, also by the GA's. The test was difficult for the GA with the third item.

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	–1.000	–1.000
	–1.000	–1.000	–1.000	–1.000	–1.000	–1.000	–1.000	–1.000

Table 38: Problem instance (starting result –49900, optimum –900).

Algorithm	**Exec.**	**Result**	**Exec.**	**Result**	**Exec.**	**Result**
Dir. action	< 0.2 s.	–10700	< 0.1 s.	–20500	< 0.2 s.	–2272
Long is nice	< 0.3 s.	–10700	< 0.1 s.	–20500	< 0.3 s.	–2272
Exp. down	< 0.2 s.	–10700	< 0.1 s.	–20500	< 0.2 s.	–2272
Big first	< 0.2 s.	–10700	< 0.1 s.	–20500	< 0.2 s.	–2272
DP 1	4 m. 40 s.	–10700	20 s.	–14620	55 s.	–6954
DP 2	4 m. 30 s.	–10700	17 s.	–20500	1 m. 35 s.	–2272
DP 3	1 m. 18 s.	–10700	7 s.	–14620	16 s.	–6954
GA400	30 s.	–12020	46 s.	–20500	36 s.	–14304
GA800	1 m.	–12022	1 m. 30 s.	–20500	1 m. 10 s.	–14667

Table 39: Results for Table 38.

0.200	–1.000	0.200	–1.000	0.200	–1.000	0.200	–1.000	0.200
	–1.000	0.200	–1.000	0.200	–1.000	–1.000	–1.000	–1.000
	–1.000	–1.000	–1.000	–1.000	–1.000	–1.000	–1.000	–1.000
0.160	–1.000	0.160	–1.000	0.160	–1.000	0.160	–1.000	0.160
	–1.000	0.160	–1.000	0.160	–1.000	–1.000	–1.000	–1.000
	–1.000	–1.000	–1.000	–1.000	–1.000	–1.000	–1.000	–1.000
0.028	–1.000	0.028	–1.000	0.028	–1.000	0.028	–1.000	0.028
	–1.000	0.028	–1.000	0.028	–1.000	–1.000	–1.000	–1.000
	–1.000	–1.000	–1.000	–1.000	–1.000	–1.000	–1.000	–1.000

Table 40: Best solutions found for problem in Table 38.

9 Conclusion

We have developed the basic theory for interactive knapsacks including the model of interactive knapsacks, results concerning NP-completeness, and relations to some well known NP-complete optimization problems. As a main application we introduced the load clipping problem which has important role in electricity supply and management.

The model, the complexity results, and the results showing subproblem relationships are all new. Also the enumerative solution and the genetic algorithms and parts of the tests connected to these are new. All other methods are given in [1, 2]. In [1] we analyze the DP method and also give the result concerning the number of "wait states" needed.

We considered mainly three problems, IKHD, IKD and MDCS. These have further variations: the clones may have fixed lengths, varying or variable lengths. The model was represented with two interaction types: radiation and cloning. In the larger load clipping application we presented a third interaction type. The model terminology follows the standard knapsack terminology. The NP-results, however, are presented in the Garey & Johnson style.

We give several applications for our model. A real life application, load clipping, is used around the world by electricity suppliers. We have not presented nor used all the restrictions electricity suppliers use [1, 2].

The optimization problem was solved with four heuristics and with three versions of DP's and two of GA's. The results of our experiments show that they are suitable for the task they were designed. Both heuristics, DP's and GA can be further developed.

By comparing the algorithms designed for IKD and MDCS against the algorithms designed for subproblems we can study the practical applicability of IKD and MDCS. Our algorithms are designed for a bit more general problem than standard IKD (profits are given by a function, some extra constraints). By some minor changes these algorithms are suitable for the standard IKD and MDCS problems.

Along the way of comparisons, we can always try to find some new problems IKHD, IKD or MDCS generalize. Also new algorithms can be developed like methods based on simulated annealing, tabu search and neural optimizers. Also branch and bound methods could be developed, even though this approach does not seem to be fruitful. Is is also possible to exploit some new properties for some methods when applied to interactive knapsacks, like for dynamic programming.

Some special cases may impose efficient algorithms for interactive knapsacks, like convex or concave profits p_{ij}. Orlin [46] study knapsack problems with convex and concave profits which might give some directions, although the interactive knapsack problems are quite different from the plain knapsack problems.

Some of the methods proposed are too slow for every situation. We may obtain some processing time by forming one plan fast with a heuristics and then there is time to use methods that spend more time.

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

$$\text{“}0000100002000000001000020000''$$

could describe the first (1) and the last (2) containers having 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 with crossover operations and would at least partly maintain the "building blocks". A swapping of two near positions could work as a mutation. We also need a way to easily move a tail one position forward or backward. That could be also used as a rare mutation type.

Also some way to add or remove clones should be used. Alternatively, we could choose to maintain in the population some sets of individuals containing a fixed number of clones. These subsets in the population would not be mixed but they rather should operate with the corresponding subsets in the other individuals.

GA's can be also implemented to give solution before the optimization problem is even given. 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 the new instance given next hour. Now GA has partial answer and it needs less work in order to find the solution.

We have skipped the stochastic interactive knapsacks. Many problems connected to interactive knapsacks are of stochastic nature, including the knapsack problem [25, 30], the multidimensional knapsack problem [18], scheduling [44, 48], and load clipping [7], among others. The load clipping problem considered in this work (and also in [1, 2]) can be generalized to stochastic optimization problem taking into account the unknown price and income factors.

Dynamic knapsack problems are considered in [30]. Interactive knapsacks, 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. However, genetic algorithms presented in this work are well suited for this kind of situation.

We intend to study the properties of interactive knapsacks with respect to the more sophisticated complexity classes [31, 32, 54]. These include, e.g. the classes #P, Opt P, Max P, Mid P, Gap-P and Med P (for definitions and further directions for listed classes, see [54]).

Another theoretical direction is to study, if it is possible to verify locally optimal solutions [33] or to test optimality [6]. Another possibility is to study the existence of approximate solutions [3], i.e. if IKHD, IKD or MDCS belong to the classes FPTAS, PTAS or APX (defined in [3]). Strongly NP-complete problems IKD and MDCS do not belong to FPTAS ([3] by Corollary 1, p. 24), because optimal solution of instance I can be bounded by a polynomial $p(|I|, \mathrm{Max}[I])$. However, many scheduling problems belong to FPTAS, PTAS or APX [24], giving a hint that IKHD might belong to the class FPTAS, and IKD, and MDCS might belong to PTAS or APX.

A Single machine scheduling

Ibaraki and Nakamura [27] 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 non-decreasing, e.g., weighted sum of tardiness or 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 weights of earliness and tardiness of job i.

B Multi-period variable-task-duration assignment problem

We state the multi-period assignment problem using the notations of Miller and Franz [41] 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 $(i = 1, \ldots, N)$ is assigned to a task j $(j = 1, \ldots, M)$ during period k $(k = 1, \ldots, T)$. 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

$$X_{ijk} = X_{i'j'k'}.$$

C Multiple translational containment

Milenkovic [40] studies the multiple translational containment, a NP-hard layout problem and uses the following symbols:

- C is for a polygonal "container" with n vertices,
- P_i, $1 \leq i \leq k$, is a polygon with m_i vertices to be placed in the container,
- 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 $\langle t_1, t_2, \ldots, t_k \rangle$ 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 non-convex polygons inside a non-convex container and kCN for k convex polygons inside a non-convex container. The (r, k)NN *problem* is to find all subsets of size k out of a set of r non-convex polygons such that the k polygons have a valid configuration inside a given non-convex container and (r, k)CN is defined similarly.

D Known NP-complete problems

In this appendix we list the NP-complete problems used in this work. Listed problems are taken directly from [19].

3-PARTITION

INSTANCE: A finite set A of $3m$ elements, a bound $B \in \mathbb{Z}^+$, and a size $s(a) \in \mathbb{Z}^+$, for each $a \in A$, such that each $s(a)$ satisfies $B/4 < s(a) < B/2$ and such that $\sum_{a \in A} s(a) = mB$.

QUESTION: Can A be partitioned into m disjoint sets $S_1, \ldots, S_m$ such that, for $1 \leq i \leq m$, we have $\sum_{a \in S_i} s(a) = B$?

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

$$\sum_{n \in N'} s(n) \leq B \qquad \text{and} \qquad \sum_{n \in N'} p(n) \geq K?$$

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?

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)?

References

[1] Isto Aho. Properties of dynamic programming used in load clipping. Submitted to *Optimization Methods and Software*, 1997.

[2] 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, 1998.

[3] G. Ausiello, P. Crescenzi, and M. Protasi. Approximate solution of NP optimization problems. *Theoretical Computer Science*, 150(1):1–55, October 1995.

[4] Dimitri P. Bertsekas. *Dynamic Programming and Optimal Control*. Athena Scientific, 1996.

[5] 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.

[6] Paolo Carraresi, Fernanda Farinaccio, and Federico Malucelli. Testing optimality for quadratic 0-1 problems. Technical Report TR-95-11, Università Di Pisa, Departimento Di Informatica, December 1996.

[7] Douglas W. Caves and Joseph A. Herriges. Optimal dispatch of interruptible and curtailable service options. *Operations Research*, 40(1):104–112, January–February 1992.

[8] 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.

[9] 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.

[10] 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.

[11] Arthur I. Cohen and Vahid R. Sherkat. Optimization-based methods for operations scheduling. *Proceedings of the IEEE*, 75(12):1574–1591, December 1987.

[12] 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.

[13] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[14] János Csirik and Gerhard J. Woeginger. Shelf algorithms for on-line strip packing. *Information Processing Letters*, 63(3):171–175, August 1997.

[15] 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.

[16] 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. North-Holland, September 1992.

[17] 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.

[18] M. E. Dyer and A. M. Frieze. Probabilistic analysis of the multidimensional knapsack problem. *Mathematics of Operations Research*, 14(1):162–176, February 1989.

[19] Michael R. Garey and David S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-completeness*. W. H. Freeman, 1979.

[20] Richard Gerber, William Pugh, and Manas Saksena. Parametric dispatching of hard real-time tasks. *IEEE Transactions on Computers*, 44(3):471–479, March 1995.

[21] 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. Hillsdale Lawrence Erlbaum, July 1987.

[22] 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. Morgan Kaufmann, July 1993.

[23] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science.* Addison-Wesley, 1989.

[24] Leslie A. Hall. Approximation algorithms for scheduling. In Dorit S. Hochbaum, editor, *Approximation Algorithms for NP-hard Problems*, pages 1–45. PWS Publishing Company, 1995.

[25] Mordechai I. Henig. Risk criteria in a stochastic knapsack problem. *Operations Research*, 38(5):820–825, September–October 1990.

[26] 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.

[27] Toshihide Ibaraki and Yuichi Nakamura. A dynamic programming method for single machine scheduling. *European Journal of Operational Research*, 76(1):72–82, July 1994.

[28] 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. ACM Press, 1994.

[29] 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.

[30] Anton J. Kleywegt and Jason D. Papastavrou. The dynamic and stochastic knapsack problem. *Operations Research*, 46(1):17–35, January–February 1998.

[31] Mark W. Krentel. The complexity of optimization problems. Technical Report TR87-834, Cornell University, Computer Science Department, May 1987.

[32] Mark W. Krentel. The complexity of optimization problems. *Journal of Computer and System Sciences*, 36(3):490–509, June 1988.

[33] Mark W. Krentel. On finding and verifying locally optimal solutions. *SIAM Journal on Computing*, 19(4):742–749, August 1990.

[34] 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.

[35] Seppo Kuusisto, Mikko Lehtokangas, Jukka Saarinen, and Kimmo Kaski. Shortterm electric load forecasting using neural network with fuzzy hidden neurons. To appear in *Neural Computing & Applications Journal*, 1998.

[36] Manuel Laguna. Robust optimization for multiperiod capacity expansion planning in telecommunications. Working paper.

[37] 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, 1996.

[38] 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.

[39] Silvano Martello and Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations.* John Wiley and Sons, 1990.

[40] V. Milenkovic. Multiple translational containment part II: exact algorithms. *Algorithmica*, 19(1/2):183–218, September–October 1997.

[41] Janis L. Miller and Lori S. Franz. A binary-rounding heuristic for multiperiod variable-task-duration assignment problems. *Computers & Operations Research*, 23(8):819–828, August 1996.

[42] Melanie Mitchell. *An Introduction to Genetic Algorithms.* MIT Press, 1996.

[43] 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.

[44] Rolf H. Möhring and Franz J. Radermacher. Introduction to stochastic scheduling problems. Technical Report MIP – 8402, Universität Passau, December 1984.

[45] 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, 1995.

[46] James B. Orlin. Some very easy knapsack/partition problems. *Operations Research*, 33(5):1154–1160, September–October 1985.

[47] 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.

[48] Michael Pinedo. *Scheduling: Theory, Algorithms, and Systems.* Prentice Hall, 1995.

[49] 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.

[50] Colin R. Reeves, editor. *Modern Heuristic Techniques for Combinatorial Problems.* Advanced Topics in Computer Science Series. McGraw-Hill, 1995.

[51] Edward M. Reingold, Jurg Nievergelt, and Narsingh Deo. *Combinatorial Algorithms: Theory and Practice.* Prentice Hall, 1977.

[52] 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.

[53] Andrew Tanenbaum. *Computer Networks.* Prentice-Hall PTR, New Jersey, 1996.

[54] Heribert Vollmer and Klaus W. Wagner. Complexity classes of optimization functions. *Information and Computation*, 120(2):198–219, August 1995.

[55] Deh-chang Wei and Nanming Cheng. Air conditioner direct load control by multi-pass dynamic programming. *IEEE Transactions on Power Systems*, 10(1):307–313, February 1995.

[56] Allen J. Wood and Bruce F. Wollenberg. *Power Generation, Operation and Control.* John Wiley & Sons, 1984.

[57] 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.