Sensitivity analysis and efficient algorithms for some economic lot-sizing and scheduling problems

Dissertation
zur Erlangung des akademischen Grades
doctor rerum politicarum (Dr. rer. pol.)

vorgelegt von
Dipl.-Inf. Sergei Chubanov

Siegen 2006

Gutachter: Prof. Dr. Erwin Pesch
Prof. Dr. Peter Letmathe
Prof. Dr. Mikhail Y. Kovalyov

Tag der mündlichen Prüfung: 30.03.2006
Prüfer: Prof. Dr. Erwin Pesch
Prof. Dr. Peter Letmathe
Prof. Dr. Andreas Drexl

Internetpublikation der Universitätsbibliothek Siegen:
urn:nbn:de:hbz:467-2399

Acknowledgements

This thesis presents some parts of my recent research. The thesis would certainly not be written without the support of different people. I am grateful to all of them. I would like to thank my supervisor Prof. Dr. Erwin Pesch for giving me the opportunity to work at his chair at the University of Siegen, for his continuing support and advice, and for the patience when reading and commenting the thesis. I thank also Prof. Dr. Letmathe for careful reading the thesis and for the comments that helped much to improve the thesis text and to correct errors. I am very grateful to Prof. Dr. Mikhail Y. Kovalyov, my another supervisor in Minsk, for suggesting interesting directions of research and for all the support. I gratefully acknowledge Prof. Dr. Mikhail Kovalev, the dean of the faculty of economics of Belarus State University, especially for giving a possibility to work at the chair of Mathematical Economics. I also thank Dr. Alexander Zaporozhets who supervised my first research. My special thanks to Dipl.-Math. Andrei Horbach and Dr. Yury Nikulin. Discussions with these persons were always fruitful.

This research was supported by grant 03PEM1SI of the Bundesministerium für Bildung und Forschung (BMBF), Germany.

Contents

Chapter 1

Introduction

1.1 Preliminary notes

In many fields of human activity one faces the necessity of decision making. In production planning one should decide how many product units to produce in each period, and in manufacturing one should decide, for instance, how to schedule jobs. In other words, one should select one of many possible variants, of many solutions, and all of them are of different quality which can be measured by different criteria like cost or time needed to implement a solution. Obviously, it is reasonable to choose the best one, i.e., an optimal solution. When we deal with a single quality parameter, say, production cost, which we want to minimize or maximize, it is usually easy to determine whether one solution is better than another. If we additionally have a small number of alternatives, we can make an exhaustive search to determine the best one. It would be an acceptable way, but there is a serious obstacle – time required to perform the necessary amount of computations. For instance, provided that we use the exhaustive search, we must explore more than $(n-2)!$ paths in a complete graph with arbitrary positive edge weights to find a shortest path (i.e., one of minimum weight) connecting two given nodes. If $n = 50$, we need billions of years to search all paths even using a fastest computer. But most often we cannot afford even a couple of days just to reach such a modest aim as a solution having minimum cost. Therefore, we need a better way. For the shortest path problem we have mentioned there is a simple algorithm designed by

Dijkstra [20] which needs only αn^2 arithmetic operations where α is a constant. This means that a common personal computer will need less than a second to find a shortest path for $n = 50$. The algorithm of Dijkstra is a good illustration both to the concept of polynomial algorithms and to the fact that sometimes problems that seem to be not solvable in the period of entire human life require seconds to be solved provided that we use an appropriate algorithm. This also reflects the main purpose of *combinatorial optimization*, the branch of applied mathematics, studying algorithms which are "better than finite" [23].

Combinatorial optimization problems have deterministic nature, i.e., all problem parameters are known in advance, whereas in many real situations necessary information is often not completely available. However, combinatorial optimization models and methods can successively be applied even in this case, fore instance, if model parameters such as demand, costs, job release dates, job processing times, etc. are forecasted. Here different methods from statistics and probability theory may be used to guess these values. After making a forecast, a model can be treated as deterministic and then corresponding combinatorial optimization problems are simply formulated using forecast values. Then, having obtained an optimal or approximately optimal solution, one can use them when planning certain activities. Every time when additional and more precise information comes, it is reasonable to correct the problem formulation and solve the problem once more taking into account that a part of the previous solution has been already implemented.

There are many combinatorial optimization problems originating from economics and having various applications there. Some of them, like lot-sizing and scheduling problems, are studied in this thesis.

Let us consider the thesis structure. In Chapter 2, different kinds of a well known single-item capacitated economic lot-sizing problem are studied and different classes of algorithms, like polynomial, subexponential, and approximate, are designed for this problem. In the first sections, we begin with a classical formulation where inventory at the end of every time period is equal to the inventory at the end of the previous period (or, equivalently, at the beginning of the present period) plus a production level in the considered period, and minus the demand in this time period. At the end of Chapter 2, we consider a more complicated

case where the effort needed to produce a product unit may vary not only from period to period, but also within an individual period. While the general case of this problem is shown to be non-approximable, there exist interesting special cases possessing fully polynomial time approximation schemes, a sort of polynomial algorithms finding a solution with any desired precision of approximation.

Chapter 3 is devoted to scheduling problems with parallel machines. A scheduling problem can roughly be formulated as follows. Given processing times for some jobs, schedule this jobs so as a) to meet certain restrictions taking into account a machine environment and additional restrictions like release or due dates; and b) to minimize an objective function, say, the total average completion time.

In the first section of Chapter 3, we will discuss a local search approach based on large-scale neighborhoods. Finding local optima in some of such neighborhoods may be an NP-hard problem. Apart polynomially searchable neighborhoods having exponential size, we consider a neighborhood searchable in pseudopolynomial time by a dynamic programming algorithm which may be converted into a fully polynomial time approximation scheme. We will also discuss some numerical experiments with randomly generated and real-world instances.

The second section of Chapter 3 deals with a two-machine problem where the objective is to minimize a nonlinear function. It is shown, that this scheduling problem generalizes many of those previously considered in the literature. We design an FPTAS for this problem. The algorithm is based on a relaxation of the feasible domain, changing the objective function, and studying properties of the recursive formulation of the problem obtained. Unlike the single-item capacitated lot-sizing problem, we need, apart monotony, some additional assumptions on the cost functions. The reason is that a recursive function upon which our dynamic programming algorithm is based has more than one variable which may take non-polynomially many values on the feasible domain. This leads to a more complicated approach than in the case of the lot-sizing problem.

One can see from the thesis structure that the thesis mostly concerns polynomial approximation algorithms. The methods used in the thesis are mostly based on analyzing how

costs of optimal solutions react to changes in problem data, i.e., on a sort of sensitivity analysis. It will be shown by different examples that additional information hidden in so-called recursive formulations of optimization problems is useful when designing algorithms having some desired efficiency properties.

1.2 Lot-sizing and scheduling models

To introduce a sort of application areas where optimization problems considered in the thesis may arise, in the following two subsections we will discuss informally lot-sizing and scheduling models and optimization problems connected with them [1]. We postpone the more detailed and formal discussion about lot-sizing and scheduling problems to introductory sections of the corresponding chapters.

Production and scheduling models and optimization problems related to them correspond to different levels of decision making. While production models deal with questions like how many of which items should be produced in different time periods, scheduling models are intended to find out in which order and by which equipment these items should be produced. Let us discuss basic notions of lot-sizing and scheduling more extensively.

1.2.1 Lot-sizing models

Production and inventory management are some of those areas where the study of lot-sizing models is originating from. One of simplest lot-sizing models assumes stationarity of the demand rate and production costs. It is called an Economic Order Quantity (EOQ) model. One of those who developed a corresponding EOQ formula to find an optimal production quantity was Harris [32] who introduced it in 1913. (One can also refer to a paper by Zangwill [62] and a technical note by Stadtler [53] for analysis of EOQ formula.) In this thesis, we will

[1] One can consider a model as a collection of fixed and variable parameters, with perhaps interrelations between them, describing different processes or phenomenons, while a problem contains additionally a question asking how to tune model parameters in order to attain certain goals. In optimization problems such goal is minimizing or maximizing some function, called objective function, depending on model parameters.

consider so-called dynamic lot-sizing models where demand, costs, and production capacities may change from period to period. One of the first publications on such dynamic single-item economic lot-sizing problems was a paper of Wagner and Whitin [59] which appeared in Management Science in 1958. In that paper, Wagner and Whitin designed a dynamic programming algorithm for a case with unlimited production capacity at every time period.

It is convenient to begin the discussion of lot-sizing models with the common terminology of inventory and production management which includes such notions as planning horizon, time period, production level, production capacity, inventory level, backlogging, setup costs, production costs, demand, etc.

Assume for a while that only a single product is produced.

A planning horizon is usually defined as the length of time the model projects into the future. A planning horizon is subdivided into time periods. For example, if a planning horizon is one year, then it can be subdivided into quarters or months. A production plan contains information about how many product units should be produced in each period. In other words, for every period a production plan determines a production level or, equivalently, lot-size.

At every time period a production level is upper bounded by a production capacity which is a maximum amount of product units that can be produced in a corresponding period. Production capacity depends on the amount of resources needed for production like machines and working power. If production capacity remains the same from period to period we deal with a so-called constant capacity. In many cases economic lot-sizing problems with constant capacities are easier to solve than more general variants where capacity varies over time. (See for instance an efficient algorithm of van Hoesel and Wagelmans [58] for the case of constant production capacities). It may happen that production capacity is unlimited. Lot-sizing problems with unlimited capacity are called uncapacitated. It is not hard to see that this is a special case of lot-sizing problems with constant capacities. (We just assume production capacity to be sufficiently large).

In the deterministic lot-sizing models that we are considering demand is known in advance for every time period (it can be also some expected value). Due to influence of different

factors, it is often reasonable to produce more than is required at the moment. This means that some amount of the product is not demanded immediately after it has been produced. In this case it forms inventory which should be stored using storage facilities maintaining conditions like temperature and humidity necessary for holding the inventory.

Sometimes there is not enough of a product to satisfy demand or it is even not possible to satisfy demand due to insufficient production capacity, i.e., in other words, an inventory level is negative. In this case we are talking about backlogging, which can be interpreted as an amount of a product produced or acquired at future periods to satisfy demand of the present period.

Any production plan has its own cost consisting in turn of such components as production, inventory, and holding-backlogging costs. The structure of these cost components may differ from one manufacturing system to another. The most basic ingredients of production costs are setup costs and unit production costs. A setup cost is usually incurred when production takes place. This is a fixed cost which can be related to the necessity of setting up or adjusting the equipment before the production starts. Unit production costs are calculated per product unit. They may include costs of resources and labor used to produce it.

Holding costs are those which arise when using storage facilities for inventory. These costs may include both fixed components and those that depend on the inventory level. In many manufacturing models holding costs are linear and therefore can be incorporated in the production cost structure provided that the model does not admit backlogging. Let us explain this more precisely. Let h_t be a cost of holding a product unit during period t. Let $I_t = \sum_{i=1}^{t}(x_i - d_i)$ be the inventory level (x_i are production levels in corresponding periods and d_i are demand values), i.e., the difference between the total production level $\sum_{i=1}^{t} x_i$ by the end of period t and the total demand $\sum_{i=1}^{t} d_i$ by this period. Then the holding cost is equal to $h_t I_t = \sum_{i=1}^{t} h_t x_i - h_t \sum_{i=1}^{t} d_i$. The term $h_t \sum_{i=1}^{t} d_i$ is constant for a particular model and can be omitted when solving the related optimization problem. Terms $h_t x_i$ can be added to the corresponding production costs in periods i. In that way, this model reduces to one without holding costs.

As we have mentioned before, in some lot-sizing models inventory levels can be negative.

In this case backlogging costs should be included into the cost structure. These costs are often interpreted as a kind of penalty taken because demand has been not satisfied on time. Setting backlogging costs to be sufficiently large, we may come to a lot-sizing problem without backlogging since in this case it is more profitable to satisfy demand on time.

The objective function consists so far of two main types of cost functions: production cost functions (they may include setup costs and unit production costs) depending on production levels and holding-backlogging cost functions depending on inventory levels. These cost functions are known in advance at each time period. The shape of these functions may depend on different factors. In many situations concavity of cost functions seems to be a reasonable assumption due to the effect of economies of scale when the per unit production cost decreases as the production level increases. Sometimes however the opposite effect of diseconomies of scale takes place. In this case, the per unit production cost grows beginning with a certain production level. Then a production cost function looks like in Figure 1.1 where Q is a production level after which the per unit production cost grows. The circumstances under which this effect may show up depend on a management structure, the size of the company, its age, the situation at the market, etc. For instance, a firm may need to modernize the equipment to increase productivity or to employ additional personnel which is not yet familiar with specifics of the manufacturing at the firm and therefore requires learning. All this implies additional expenses per product unit. More information about economic properties of production costs and more examples can be found in the work of Canbäck [12].

The described model reflects basic common features of a huge variety of economic lot-sizing problems where the objective is to find an optimal production plan (i.e., one having minimum cost) satisfying all the restrictions imposed by a corresponding production model. Note that a related optimization problem can be described in different ways depending on the supposed methods for solving it.

One distinguishes between single-item and multi-item economic lot-sizing problems (see for instance the book by Tempelmeier [55] for more information). In single-item economic lot-sizing problems only a single product type is considered. This is exactly the problem we

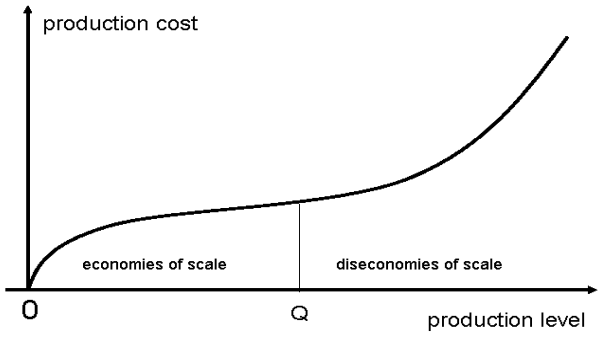

Figure 1.1: Economies and diseconomies of scale.

have already described. Multi-item lot-sizing problems cover more general models with many product types. Usually they have additional restrictions linking product items. The reason is that production of different product types may require the same resources. For instance, a plant produces seamless steel pipes and welded steel pipes. (Seamless steel pipes are usually used to transport gas, while welded pipes can be used in the construction of buildings.) For both of these product types steel is used. Since steel is a limited resource, one should think about how to share it between these two product items at every time period, i.e, there is a restriction linking them.

Both single- and multi-item economic lot-sizing problems are NP-hard in their general form (the complexity results concerning lot-sizing problems can be found in the paper written by Bitran and Yanasse [7]). Branch-and-bound methods are most popular for multi-item lot-sizing problems (see for instance papers of Constantino [18] and [19], Pochet [46], Pochet and Wolsey [47], and a survey of Belvaux and Wolsey [5]), while different variants of dynamic programming algorithms are mostly used for single-item cases (see a paper of Chen et al. [13], and papers of van Hoesel and Wagelmans [57] and [58]). Note that, since single-item economic lot-sizing problems are included in the structure of multi-item ones, one can use methods for solving single-item problems as local search algorithms for multi-item economic lot-sizing problems.

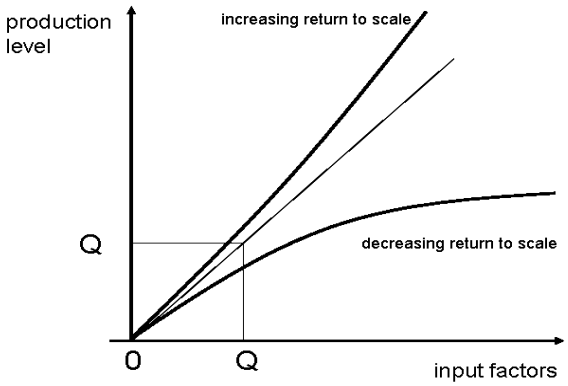

Figure 1.2: Return to scale.

The classical lot-sizing model [59] does not take into account the effect of returns to scale. Remind that returns to scale refer to technical properties of production showing how output changes if we increase the quantity of all input factors by the same amount. Decreasing returns to scale mean that the output grows by less than that amount. If the output increases by more than that amount, we deal with increasing returns to scale. (Examples for both cases are depicted in Figure 1.2.) In other words, production levels may be functions depending on the mentioned amount by which we increase input factors. This leads to a generalization of the classical economic lot-sizing problem which we will study in Section 2.6.

1.2.2 Scheduling models

When planning a manufacturing process, a common question arising among others is which job (or operation), at which time, and by which machine should be processed. In other words, the question is how to schedule jobs under certain conditions like machine or resource availability. Different schedules have different values of, say, the average completion time of all jobs or the completion time of the last job. Optimization of these parameters allows to considerably increase effectiveness of a manufacturing system. Nowadays many books on management science, logistics, and production planning include basic information

about scheduling problems, i.e., optimization problems connected with scheduling models (see for instance the book of Domschke and Drexl [21] on operations research and the book of Thonemann [56] on operations management). There is a huge variety of such problems, and their classification contains a huge amount of optimization criteria. Having not in mind to discuss a classification of scheduling problems in detail, we will rather sketch their basic features. More classification details as well as algorithms to solve scheduling problems can be found, for instance, in the books by Błażewicz et al. [9] and Pinedo [45].

A scheduling model usually includes a certain amount of jobs which are to be processed by several machines or a single machine. Each job possesses its own processing time which may vary depending on a machine which is assigned to process the job. A machine may only process a single job at a time. We may think about a machine as a single unit of a resource which is available at every time. In this case every job requires one unit of resource. (We may extend the model to the case when different jobs require different amounts of resource. Then we come to more general project scheduling models which are not considered in the present thesis.) Depending on the model restrictions, jobs should be processed either preemptively or non-preemptively. If jobs are allowed to be processed preemptively, then processing a job can be interrupted at any time and continued after a while. Otherwise, no interruptions of job processing are allowed which means that once a machine begins to process a job, it should be competed without any interruptions.

There are different reasons which cause such subdivision of scheduling models into those with preemptions and those without them. For instance, in semiconductor industry it is usually not allowed to interrupt the process of annealing a wafer both in the case of a so-called rapid thermal anneal and in the case of a furnace anneal. This is an example of how production technology influences restrictions of a scheduling model.

An example of the case when preemptions are not only allowed but often seem to be a most reasonable issue comes from the area of multitasking computer operating systems. These systems execute software processes in such a way that they seem to run at the same time. This illusion is attained by scheduling processes preemptively: an operating system allows a processor to execute a small portion of one process, then a small portion of another

process, and so on until all processes are completed. Since these portions are not too big, this creates an illusion that the processes are running simultaneously. If these processes were allowed to be executed only one after another, i.e., non-preemptively, it would be annoying for a user since he could not, for instance, quickly switch between different applications and easily transfer the data from one application to another. This is an example of how optimization goals (in this case convenience for a user) may influence scheduling model restrictions.

Apart processing times, release and due dates are important characteristics of jobs. A release date is the time at which a job is available to be processed. Due dates are often included into optimization criteria which impose a sort of penalty if a completion time of a job exceeds a corresponding due date.

Scheduling models may also differ by machine types used to process jobs. Machines are identical if they have the same characteristics and are able to process the same set of jobs. Uniform machines are those which may differ from each other by their speed. In the case of uniform machines processing times of a job at different machines may differ only by a factor depending on a machine. (In Subsection 3.3.3 we will consider a generalization of the notion of uniform machines to the case when machine speed may vary over time.) For unrelated machines this factor may also depend on a job. One can say that unrelated machines are specialized in contrast to uniform machines.

Observe now common optimization criteria for scheduling problems. One of them is the makespan (schedule length). This is the maximum job completion time in a schedule. The makespan has to be minimized provided that we would like to process all the jobs a soon as possible. A flow time of a job in some schedule is a difference between a completion time of this job and its release date. I.e., a flow time is simply the time that a job spends in the system. Average flow time of jobs is an important optimization criterion when scheduling manufacturing processes. Note that minimizing total flow time is equivalent to both minimizing average completion time and total completion time which is a sum of job completion times.

1.3 Notation and terminology

In this section, we give some background needed to understand the subsequent chapters. We use standard terminology and notation, and therefore we will rather sketch definitions than give them formally. More details, especially on complexity theory, can be found, for instance, in the book of Korte and Vygen [39].

1.3.1 Sets

We will use notation $[a, b]$ for a set $\{a, a+1, \ldots, b\}$ of integers. If $a > b$, then $[a, b]$ is empty. We will call such sets *intervals* of integers or simply intervals.

By $\mathbb{R}$, $\mathbb{Z}$, and $\mathbb{Q}$ we denote the set of real, integer, and rational numbers, respectively. If we write $\mathcal{S}_+$ or $\mathcal{S}_{++}$ for a set $\mathcal{S} \subseteq \mathbb{R}$, this means that we take only nonnegative or positive elements of this set, respectively. In the same way we use notations $\mathcal{S}_-$ and $\mathcal{S}_{--}$ for subsets of nonpositive and negative elements.

In algorithms and computer programs implementing these algorithms, different combinatorial structures are applied to represent sets. Lists, priority queues, and heaps are used to store them. These structures allow to access elements of the sets efficiently. With respect to the present thesis, from the algorithmic point of view, it is not so important which data structure is selected. In most of the algorithms that we will consider, the order in which elements of sets are accessed does not play any role and therefore if we apply a certain operation to all elements of a set, then we may assume that each of its elements may be accessed in a single elementary step.

1.3.2 Algorithms and complexity

An algorithm may informally be defined as a sequence of *elementary* (or *basic*) steps (or *operations*) which transform valid input data (or simply input) to output data (or output). An elementary step may be a variable assignment, random access to a variable, and any of simple arithmetic operations like addition, subtraction, multiplication, division, and comparison of numbers.

For instance, an input for the simplex method consists of a set of linear inequalities and a linear objective function. The simplex method generates an optimal solution, if it exists, or recognizes that there is no feasible solution or the linear problem is unbounded.

Let A be an algorithm for a set of valid inputs $\mathcal{P}$ and let $f : \mathcal{P} \to \mathbb{Z}_{++}$ be a positive function defined on all these inputs. If there exists a constant α such that A terminates computation after at most $\alpha f(\mathcal{I})$ elementary steps for any input $\mathcal{I}$, then we say that A runs in $O(f)$ time or, equivalently, the *running time* (or *time complexity*) of A is $O(f)$. (One can say that notation $O(\cdot)$ suppresses a constant.) Function $O(f)$ is called the time complexity estimation function of algorithm A.

Let us consider some examples. Assume that there exists an algorithm for some linear programming problem which performs at most $5n + 1$ elementary steps to find an optimal solution to any instance with n variables. This means that this algorithm runs in $O(n)$ time. Dijkstra's algorithm makes at most αn^2 steps where α is constant to find a path of minimum weight in a graph with n nodes and without cycles of negative weight. This means that Dijkstra's algorithm runs in $O(n^2)$ time.

Further, we rely upon such widely known computational model as a RAM machine. (Abbreviation RAM stands for Random Access Memory.) The RAM machine consists of an infinite array of memory cells, each possessing infinite capacity. Any memory cell may contain an integer number. (Remind that rational numbers can be encoded as pairs of integers.) Thus, any input $\mathcal{I}$ in this computational model is just some sequence of integers $(a_1, \ldots, a_n)$. A length $size(a)$ of a $(0,1)-$string encoding some integer number a is called the size of integer a. To encode a, one needs $\lceil \log(|a| + 1) \rceil$ bits and one additional bit for a sign. Thus, $size(a) = \lceil \log(|a| + 1) \rceil + 1 = O(\log(|a| + 1))$. (Further we often write $O(\log a)$ to shorten notation.) A size $size(\mathcal{I})$ of an input $\mathcal{I}$ is a total length of all integers forming this instance. By this definition, if $\mathcal{I} = (a_1, \ldots, a_n)$, then $size(\mathcal{I}) = O(\sum_{i=1}^{n} \log a_i)$. Input data of many algorithms consist of combinatorial objects like graphs, matrices, and polyhedra that can be described by sequences of numbers. A size of such an object is a binary length of a corresponding sequence of integer numbers. (We will use notation $size(X)$ for the size of some combinatorial object X.) For instance, an integer-valued $m \times n$ matrix $A = (a_{ij})$

can be written as a sequence

$$m, n, a_{11}, \ldots, a_{1n}, a_{21}, \ldots, a_{2n}, \ldots, a_{m1}, \ldots, a_{mn}$$

which can be encoded using

$$size(A) = \sum_{i=1}^{m} \sum_{j=1}^{n} size(a_{ij}) + size(m) + size(n)$$

bits. Analogously, the size of an $m-$vector b is equal to $\sum_{i=1}^{m} size(b_i) + size(m)$, and the size of $n-$vector c equals $\sum_{j=1}^{n} size(c_j) + size(n)$. Therefore, we need at most $size(A) + size(b) + size(c)$ bits to encode an input consisting of a linear program

$$\begin{aligned} \min \quad & cx \\ \text{s.t.} \quad & Ax \leq b. \end{aligned}$$

Consider an example where $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$, $b = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$, and $c = [1, 1]$. Values 1, 2, 3, and 4 can be encoded as $(0,1)-$strings 1, 10, 11, and 100, respectively. The following sequence contains complete information about this linear program:

$$\underbrace{10, 10,}_{m,n} \underbrace{1, 10, 11, 100,}_{A} \underbrace{1, 10,}_{b} \underbrace{1, 1}_{c}$$

Let $poly(\cdot)$ be some polynomial with positive coefficients. If the complexity function $f(\cdot)$ mentioned earlier is defined as $f(\mathcal{I}) = poly(size(\mathcal{I}))$ over valid inputs $\mathcal{I}$, then a corresponding algorithm is said to be *polynomial*. Equivalently, one says that such an algorithm runs in *polynomial time*. While polynomial algorithms are usually considered to be efficient ones, there are algorithms which may perform exponentially many (of the instance size) basic steps for some problem instances.

One also distinguishes pseudopolynomial algorithms. An algorithm is said to run in pseudopolynomial time if for any valid input $\mathcal{I} = (a_1, \ldots, a_n)$ it finds an optimal solution in $O(poly(\sum_{i=1}^{n} |a_i|))$ time where $poly(\cdot)$ is some polynomial. All polynomial algorithms are also pseudopolynomial. There exist pseudopolynomial algorithms for some of NP-hard problems.

1.3.3 Optimization problems

Informally, a problem is a set of instances and a task. In combinatorial optimization, one distinguishes between decision and optimization problems. In a decision problem a task is to verify if a certain predicate holds. For instance, given a system of linear inequalities, verify if there is a vector satisfying this system. In an *optimization problem* a task is to find a solution which optimizes (minimizes or maximizes) a function on a given set. Both the classes, optimization and decision problems, are closely related to each other. It is often that solving optimization problems boils down to solving decision problems and vice versa.

An instance of a minimization problem can be written as

$$\min F(x)$$
$$\text{s.t. } x \in \mathcal{D}$$

where function $F(\cdot)$ belongs to a family $\mathfrak{F}(\mathcal{D})$ of polynomially computable functions and $\mathcal{D}$ belongs to a family $\mathfrak{D}$ of polynomially computable sets. (Under polynomially computable sets we understand those for which we may verify if a given object belongs to it in time polynomial of the instance size.) The abbreviation "s.t." in the above description stands for "subject to". Each element x of the set $\mathcal{D}$ is called a *feasible solution.* Thus, $\mathcal{D}$ is a set of feasible solutions, and the objective is to minimize $F(x)$ subject to $x \in \mathcal{D}$. Sometimes a set of feasible solutions is called a *feasible domain* or a *feasible set.* $\mathfrak{F}(\cdot)$ is a mapping from the family $\mathfrak{D}$ of feasible domains to a set of functions. Such dependence of an objective function set on a feasible domain is necessary to provide compatibility of objective functions and feasible sets. For instance, if $\mathcal{D}$ is a subset of integer $n-$vectors, then $\mathfrak{F}(\mathcal{D})$ may be the set of all rational linear functions of n arguments.

In the described problem the task is to find a feasible solution x with minimum objective value $F(x)$. A feasible solution minimizing the function $F(\cdot)$ is called optimal.

An optimization problem $\mathcal{P}$ so far is a set

$$\mathcal{P} = \{(F, D, goal) | F \in \mathfrak{F}(\mathcal{D}),\ \mathcal{D} \in \mathfrak{D}\}$$

of instances $(F, D, goal)$ where $goal$ may be equal to min or max depending on the kind

of optimization problems we consider: minimization or maximization ones. We will use notation $OPT(\mathcal{I})$ to denote an optimal value (the minimum or maximum objective value of a feasible solution) of a problem instance $\mathcal{I}$. In the present thesis, we will mostly consider minimization problems.

Assume that $\mathfrak{D}$ is a family of all polyhedra $\mathcal{D} = \{x \in \mathbb{Q}^n | Ax \leq b\}$ where $n, m \in \mathbb{Z}_{++}$, $b \in \mathbb{Q}^m$, and $A \in \mathbb{Q}^{n \times m}$. For every such polyhedron $\mathcal{D}$ define $\mathfrak{F}(\mathcal{D})$ as a family of all functions of the form $\min \sum_{j=1}^{n} c_j x_j$, where $c_j \in \mathbb{Q}$ for all $j \in [1, n]$, defined on the set of integer $n-$vectors x. Then we might deal with an integer linear problem each instance of which may be presented in the form

$$\begin{array}{lll} \min & \sum_{j=1}^{n} c_j x_j & \\ \text{s.t.} & \sum_{j=1}^{n} a_{ij} x_j \leq b_i, & \forall i \in [1, m], \\ & x_j \in \mathbb{Z}, & j \in [1, n]. \end{array}$$

where a_{ij}, $i \in [1, m], j \in [1, n]$, are coefficients of matrix A.

We will also use sometimes the notation $\min\{F(x) | x \in \mathcal{D}\}$ for optimization problems. In this case the above mentioned integer linear problem is written as $\min\{cx | Ax \leq b\}$.

Polynomial algorithms are usually considered to be the best we can hope for when dealing with optimization problems. For not many of them polynomial algorithms are known. Basically, the problems arising in practical situations are *NP-hard*[2]. An NP-hard problem is that to which any optimization problem reduces in polynomial time. (We say that optimization problem $\mathcal{P}'$ polynomially reduces to optimization problem $\mathcal{P}$, if there exists some algorithm A for problem $\mathcal{P}'$ performing polynomially many basic operations and calls of an algorithm solving problem $\mathcal{P}$. In other words, algorithm A uses an *oracle* (i.e., the algorithm for $\mathcal{P}$) to access optimal solutions of instances of problem $\mathcal{P}$.) There is a famous mathematical problem which can roughly be formulated as follows: does there exist a polynomial algorithm for any optimization problem?[3] When a problem is known to be NP-hard, which means

[2]Abbreviation "NP" stands for "Nondeterministic polynomial". More detailed information may be found in the book of Garey and Johnson [28].

[3]Further NP will denote the whole class of optimization problems. Notation P will stand for polynomially solvable ones. The above mentioned open problem consists in verifying if P=NP.

that any optimization problem can be polynomially reduced to it, then any proof showing that there is a polynomial algorithm to this NP-hard problem is in the same time a proof of that famous mathematical problem which is still open and seems to be very difficult to deal with. In that way, NP-hardness is an indicator by which we can judge about computational difficulty of solving an optimization problem.

1.3.4 Dynamic programming

Most algorithms in the thesis are based on a technique called dynamic programming. To explain what dynamic programming is, consider a well known Knapsack Problem. Any instance of this problem contains nonnegative integers $c_1, \ldots, c_n$, $w_1, \ldots, w_n$ and W. The objective is to find a subset $\mathcal{S} \subseteq [1, n]$ maximizing value $\sum_{i \in \mathcal{S}} c_i$ subject to constraint $\sum_{i \in \mathcal{S}} w_i \leq W$. A set $\mathcal{S} \subseteq [1, n]$ is uniquely defined by its characteristic vector $(x_1, \ldots, x_n)$ where $x_i = 1$ if $i \in \mathcal{S}$ and $x_i = 0$, otherwise. An instance of this Knapsack Problem can be written as an equivalent mathematical program

$$\begin{aligned} \max \quad & \textstyle\sum_{i=1}^{n} c_i x_i \\ \text{s.t.} \quad & \textstyle\sum_{i=1}^{n} w_i x_i \leq W, \\ & x_i \in \{0, 1\}, \qquad \forall i \in [1, n], \end{aligned}$$

where x_i, $i \in [1, n]$ are considered as variables. For any $t \in [1, n]$ and $s \in [0, W]$ define a reduced instance $\mathcal{I}_{t,s}$ getting rid of variables $x_{t+1}, \ldots, x_n$:

$$\begin{aligned} \max \quad & \textstyle\sum_{i=1}^{t} c_i x_i \\ \text{s.t.} \quad & \textstyle\sum_{i=1}^{t} w_i x_i \leq s, \\ & x_i \in \{0, 1\}, \qquad \forall i \in [1, t]. \end{aligned}$$

Let $\phi_t(s) = OPT(\mathcal{I}_{t,s})$. This is a function of arguments t and s. It is easy to verify that

$$\max_{s \in [0,W]} \phi_n(s) = OPT(\mathcal{I})$$

where $\mathcal{I}$ is the original problem instance. Let $\phi_0(s) = 0$ if $s \geq 0$ and $\phi_0(s) = -\infty$, otherwise. Value $\phi_t(s)$ can be expressed by the formula

$$\phi_t(s) = \max_{x_t \in \{0,1\}} \{\phi_{t-1}(s - w_t x_t) + c_t x_t\}. \tag{1.1}$$

In this formula, values of function $\phi_{t-1}(\cdot)$ are used, i.e., the formula uses values of function ϕ to calculate function ϕ. In other words, function ϕ is defined through itself. I.e., function ϕ belongs to the class of so-called *recursive functions*. Formulas like (1.1), where some terms require computation of "previous" terms, are called recursive formulas. Informally, a dynamic programming algorithm is that which is based on computations of a recursive function and which follows the structure of a corresponding recursive formula.

Normally, a dynamic programming algorithm consists of two stages. At the first stage values of a recursive function are calculated and corresponding values of arguments, at which these function values are attained, are identified. At the second stage a dynamic programming algorithm derives an optimal solution of the corresponding problem instance. The first stage is often called recursion. The second stage is called backtracking. At this stage, the information obtained at the first stage is used. The first stage of a dynamic programming algorithm for KNAPSACK PROBLEM may look as follows.

for $t = 1$ **to** n **do**
    for $s = 0$ **to** W **do**
        compute $\phi_t(s)$ by formula (1.1);
        store a value of x_t at which the maximum in formula (1.1) is attained;
    end do
end do

To store the values of variables x_t delivering a maximum in formula (1.1), one can use an $n \times (W + 1)-$array. Then, at the backtracking stage, whenever we need a value of x_t at which function value $\phi_t(s)$ is attained, we may get it in constant time reading a corresponding location in the array. The backtracking stage constructs a characteristic vector $x^0 = (x_1^0, \ldots, x_n^0)$ of an optimal set $\mathcal{S}$:

$s := \arg\max_{s\in[0,W]} \phi_n(s)$;
for $t = n$ **down to** 1 **do**
    let x_t^0 be the value of x_t at which the maximum in formula (1.1) is attained;
    $s := s - w_t x_t^0$;
end do

The above algorithm runs in $O(nW)$ time which is pseudopolynomial. Many of the algorithms we consider in subsequent sections follow this simple algorithmic scheme.

1.3.5 Approximation algorithms

NP-hard optimization problems are subdivided in turn into many classes distinguishing from each other by different complexity aspects. For instance, NP-hard optimization problems differ with respect to their computational difficulty of obtaining a solution the objective value of which is within a certain percentage of an optimum value. One of the most important classes of approximation algorithms are fully polynomial time approximation schemes. This name, which is usually abbreviated as FPTAS, originates from Garey and Johnson [27] and stands for a polynomial algorithm of certain kind delivering a solution in polynomial time with any desired precision of approximation. The systematic research on approximation algorithms began, in many respects, with the work by Ibarra and Kim [35], who developed the first FPTAS for the knapsack problem, and Garey and Johnson [27] who presented mostly negative results saying that there are NP-hard problems to which a fully polynomial time approximation scheme does not exist unless all NP-hard problems are polynomially solvable.

Let us give a formal definition. A fully polynomial time approximation scheme for a minimization problem $\mathcal{P}$ is a family $\{A_\varepsilon\}_{\varepsilon>0}$ of algorithms, each solving any instance $\mathcal{I} = (F, \mathcal{D}, min) \in \mathcal{P}$ with a corresponding relative error $\varepsilon > 0$ in time polynomial of the instance size and $1/\varepsilon$. In other words, an algorithm A_ε delivers a feasible solution $x \in \mathcal{D}$ with the objective value $F(x) \leq (1+\varepsilon)OPT(\mathcal{I})$ in $poly(size(\mathcal{I}), 1/\varepsilon)$ time where $poly(\cdot,\cdot)$ is some polynomial of two variables. Such solution x is called an $\varepsilon-$approximate solution of instance

$\mathcal{I}$.

If $\mathcal{P}$ is a maximization problem, $\varepsilon-$approximation solutions x to its instances $\mathcal{I}$ are defined as those which satisfy condition $(1+\varepsilon)F(x) \geq OPT(\mathcal{I})$. The remaining part of the definition of an FPTAS for maximization problems is the same as for minimization ones.

The subclass of NP-hard optimization problems for which fully polynomial time approximation schemes have been designed is not very big. For many problems it has been proven that there is no polynomial algorithm solving them with a constant or polynomially bounded relative error unless P=NP. Such problems are called *nonapproximable* with a relative error of corresponding type.

The FPTAS of Ibarra and Kim can be adapted to some special cases of the single-item capacitated economic lot-sizing (CELS) problem. Compared with the knapsack problem, the CELS problem has a more complex structure requiring specific techniques to investigate. For the CELS problem there had no polynomial algorithm been known with a guaranteed performance until the year 2000, when van Hoesel and Wagelmans [58] developed the first FPTAS for the case with concave holding costs and piecewise concave backlogging and production costs with a polynomial number of pieces. The concavity conditions are however not critical for the existence of an FPTAS for the CELS problem, as we will see later. The technique we will discuss relies upon analysis of a recursive function related to a dynamic programming algorithm. As we will show, similar techniques may also be used to analyze some scheduling problems. The analysis of recursive functions is also useful to design polynomial and exponential algorithms which are discussed in the next chapter.

1.3.6 Sensitivity analysis

Let us discuss the first part of the thesis' title. This part contains the words "sensitivity analysis". Speaking very generally, sensitivity analysis is a procedure which is used to find out how output parameters of a model depend on input parameters. In this thesis we mostly use a technique which satisfies this definition. Let us briefly sketch the idea by means of the example of a dynamic programming algorithm from Subsection 1.3.4. That dynamic programming algorithm relies on solving reduced instances of a given problem instance.

Values of the recursive function $\phi_t(\cdot)$ on which the dynamic programming algorithm in Subsection 1.3.4 is based are optimal values of corresponding reduced instances. (Note that the optimal value of the instance can be interpreted as the output of the model.) Formulations of reduced instances differ from each other by values of some input parameters. Optimal values of different reduced instances may however be close or equal. For instance, it may happen that function $\phi_t(\cdot)$ is constant on some interval $[s_1, s_2]$. In this case it is sufficient to store end points of this interval and the corresponding function value to have the complete information about function $\phi_t(\cdot)$ on interval $[s_1, s_2]$. If we know how to efficiently derive a relatively small set to which such end points must belong, this would help us to compute function $\phi_t(\cdot)$ with less effort. As it will be shown in the subsequent chapters, a sensitivity analysis of this kind is a valuable tool either for developing polynomial exact or approximation algorithms.

1.4 Bound improvement procedure

In the subsequent chapters we describe fully polynomial time approximation schemes for different combinatorial optimization problems. In each of them we use a so-called bound improvement procedure devised by Kovalyov [43].

1.4.1 Bound improvement procedure

Let $\mathcal{P}$ be some minimization problem. Assume that an algorithm $A_\varepsilon(L, U)$ possesses the following properties.

(i) If L and U satisfy $0 < L \leq OPT(\mathcal{I}) \leq U$ where $\mathcal{I} \in \mathcal{P}$ and $\varepsilon > 0$, then algorithm $A_\varepsilon(L, U)$ generates a feasible solution with objective value less than or equal to $U + \varepsilon L$;

(ii) The time complexity of $A_\varepsilon(L, U)$ is bounded by a polynomial $poly(size(\mathcal{I}), U/(\varepsilon L))$ for any instance $\mathcal{I} \in \mathcal{P}$.

Bound improvement procedure [43]

Step 1. Set $k = 1$.

Step 2. Calculate $F^{(k)} = 2^{k-1}L$ and apply the algorithm $A_1(F^{(k)}, 2F^{(k)})$. If it finds a solution with a value $F^A \leq 3F^{(k)}$, then set $F^0 = F^{(k)}$ and stop. Otherwise, set $k = k + 1$ and repeat Step 2.

Theorem 1.4.1 ([43]) *If* $0 < L \leq OPT(\mathcal{I}) \leq U$ *and algorithm* $A_\varepsilon(L, U)$ *satisfies properties (i) and (ii), then the bound improvement procedure generates a value* F^0 *satisfying*

$$F^0 \leq OPT(\mathcal{I}) \leq 3F^0$$

in $O(poly(size(\mathcal{I}), 2)\log(U/L))$ *time.*

1.4.2 Modified bound improvement procedure

An algorithm described in this subsection has been reported by Tanaev, Kovalyov, and Shafransky in the monograph [54]. This is an improved variant of Kovalyov's bound improvement procedure described in the previous subsection. In the subsequent chapters, in most cases we use this modified variant, i.e., the name "bound improvement procedure" will stand most often for the modified variant.

We describe the procedure for an arbitrary minimization problem. The procedure allows to find such a value F^0 that $F^0 \leq F^* \leq 3F^0$, where F^* is the minimum objective function value. The procedure is based on applying a specific approximation algorithm. Below we describe properties of this algorithm required for the procedure to be correct and efficient.

Let $F^* > 0$ and assume there exists an approximation algorithm $Y(A, B)$ that satisfies the following properties:

(i) For any positive numbers A and B and any problem instance, algorithm $Y(A, B)$ finds a feasible solution with value $F^Y \leq B + A$, if $F^* \leq B$;

(ii) The running time of algorithm $Y(A, B)$ is bounded by a polynomial $poly(size, B/A)$ of two variables, where $size$ is the problem instance size in binary encoding.

Assume that lower and upper bounds are known such that $0 < L \leq F^* \leq U$. If $U > 3L$, then the proposed procedure finds a value F^0 such that $F^0 \leq F^* \leq 3F^0$.

Hereby, a modification of a bisection search is applied to the interval $[L, U]$.

Bound improvement procedure (Modified)

Step 1 Set $F^0 = L$, $j = 1$, and $a_j = 0$. Find an integer number b_j such that $2^{b_j-1}L < U \leq 2^{b_j}L$.

Step 2 Compute $k_j = \lceil (a_j + b_j)/2 \rceil$ and $F^{(k_j)} = 2^{k_j-1}L$.

Step 3 Apply algorithm $Y(F^{(k_j)}, 2F^{(k_j)})$. There are two cases to consider.

(a) It finds a solution with value $F^Y \leq 3F^{(k_j)}$. In this case, reset $F^0 := F^{(k_j)}$, and if $k_j = b_j$, then the procedure stops. If $k_j < b_j$, then set $a_{j+1} = a_j$ and $b_{j+1} = k_j$.

(b) It finds no solution with value $F^Y \leq 3F^{(k_j)}$. In this case, the procedure stops if $k_j = b_j$. If $k_j < b_j$, then set $a_{j+1} = k_j$ and $b_{j+1} = b_j$.

In both cases (a) and (b), if the procedure does not stop, then reset $j := j + 1$ and go to Step 2.

Theorem 1.4.2 *If values L and U satisfy $0 < L \leq F^* \leq U$ and algorithm $Y(A, B)$ satisfies properties (i) and (ii), then the bound improvement procedure finds a value F^0 such that $F^0 \leq F^* \leq 3F^0$ in time $O\left(poly(size, 2) \log \log(U/L)\right)$.*

Proof. We first show by induction that either $2^{a_j}L \leq F^* \leq 2^{b_j}L$ for each j used in the procedure or there exists an index j such that $F^{(k_i)} \leq F^* \leq 3F^{(k_i)}$ for each $i \geq j$ such that algorithm $Y(F^{(k_i)}, 2F^{(k_i)})$ finds a solution with value $F^Y \leq 3F^{(k_i)}$. It is clear that $2^{a_1}L \leq F^* \leq 2^{b_1}L$. Assume $2^{a_j}L \leq F^* \leq 2^{b_j}L$ and consider an application of algorithm $Y(F^{(k_j)}, 2F^{(k_j)})$ for $k_j = \lceil (a_j + b_j)/2 \rceil$. If the algorithm finds no solution with value $F^Y \leq 3F^{(k_j)}$, then property (i) implies

$$2^{a_{j+1}}L = 2^{k_j}L < F^* \leq 2^{b_j}L = 2^{b_{j+1}}L.$$

If it finds a solution with value $F^Y \leq 3F^{(k_j)}$, then there are two cases to analyze: $F^* \leq 2F^{(k_j)}$ and $2F^{(k_j)} < F^* \leq F^Y$. In the former case,

$$2^{a_{j+1}}L = 2^{a_j}L \leq F^* \leq 2F^{(k_j)} = 2^{b_{j+1}}L.$$

In the latter case,

$$F^{(k_j)} \leq 2F^{(k_j)} < F^* \leq 3F^{(k_j)}$$

and $b_{j+1} = k_j$. Then, for any $i \geq j+1$, inequality $k_i \leq k_j$ is satisfied. Thus, if algorithm $Y(F^{(k_i)}, 2F^{(k_i)})$ finds a solution, then $F^* \leq F^Y \leq 3F^{k_i}$ and

$$F^{(k_i)} \leq 2F^{(k_i)} \leq 2F^{(k_j)} < F^* \leq 3F^{(k_i)}.$$

Consider the last iteration determined by the equation $k_j = b_j$. In this case, $b_j = a_j + 1$. By the induction assumption, we have $F^{(k_j)} = 2^{k_j - 1}L = 2^{a_j}L \leq F^* \leq 2^{b_j}L = 2F^{(k_j)}$ or $F^0 = F^{(k_i)}$, where i is the last iteration in which algorithm $Y(F^{(k_i)}, 2F^{(k_i)})$ has found a solution with value $F^Y \leq 3F^{(k_i)}$. In the latter case, the correctness of the procedure is proved. In the former case, by property (i), algorithm $Y(F^{(k_j)}, 2F^{(k_j)})$ will find a solution with value $F^Y \leq 3F^{(k_j)}$, and therefore, $F^0 = F^{(k_j)}$. Thus, value F^0 found by the procedure satisfies $F^0 \leq F^* \leq 3F^0$.

Let us determine the time complexity of the bound improvement procedure. Since bisection search is performed in the interval $[a_1, b_1]$, the number of iterations of the procedure does not exceed $O(\log(b_1 - a_1))$. From $U > 2^{b_1 - 1}L$ we conclude that $b_1 \leq \log(U/L) + 1$. In each iteration, algorithm $Y(F^{(k_j)}, 2F^{(k_j)})$ is applied once. Property (ii) implies that the running time of this algorithm is $O(poly(size, 2))$ for any k_j. Thus, the overall time complexity of the bound improvement procedure is $O(poly(size, 2) \log\log(U/L))$. ∎

Chapter 2

Single-item capacitated economic lot-sizing problems

2.1 Preliminary notes

In this chapter, we consider a manufacturing system producing a single product over n time periods. Demand d_t is known in advance for every time period $t \in [1, n]$. We will mostly concentrate on a case where product shortage is allowed. Since we consider a sort of penalty costs on product shortage, this case is more general than one where demand at every period should be satisfied by production in this period or in previous ones.

If I_{t-1} denotes the inventory level at the end of time period $t-1$, x_t is the amount of product produced in period t (production level), and d_t is the demand in this period, then the inventory level at the end of period t is determined through the equation

$$I_t = I_{t-1} + x_t - d_t.$$

One can see that from the point of view of the network flow theory this is simply a flow conservation equation: I_{t-1} plus x_t is ingoing flow, I_t is outgoing flow, and d_t is the demand. This is illustrated in Figure 2.1 for four time periods.

At the end of this chapter we will also consider more complex dependencies of inventory levels on production and inventory coming from previous periods.

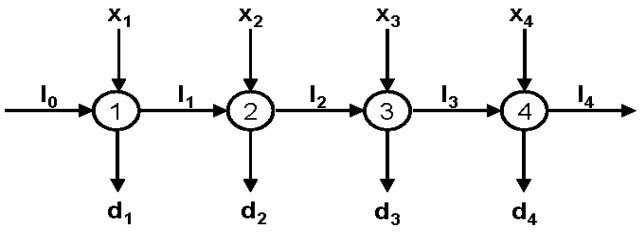

Figure 2.1: Lot-sizing model with four time periods

One considers sometimes the case with additional restriction $I_t \geq 0$. This restriction ensures that there are no product shortages. Equivalent restrictions may include only production levels:

$$\sum_{i=1}^{t} x_i \geq \sum_{i=1}^{t} d_i, \quad \forall t \in [1, n].$$

The word "capacitated" in the problem name reflects restrictions imposed on production levels x_t in each period $t \in [1, n]$:

$$0 \leq x_t \leq u_t.$$

An upper bound u_t on production level x_t is called *production capacity*. A production capacity at some time period is a maximum amount of product units a manufacturing system is able to produce. Production levels x_t may be assumed to be integer which is practically reasonable since in most cases we can choose a unit of measure that fits us best. It can be, for instance, a batch of semiconductor chips or a barrel of oil or a single car in the automobile industry.

For every period, there are cost functions related to production and inventory levels. These costs are called production and holding costs, respectively. If a shortage of a product is allowed, i.e., inventory levels are allowed to be negative, so-called backlogging costs depending on negative inventory levels are considered. (These costs may be interpreted as a kind

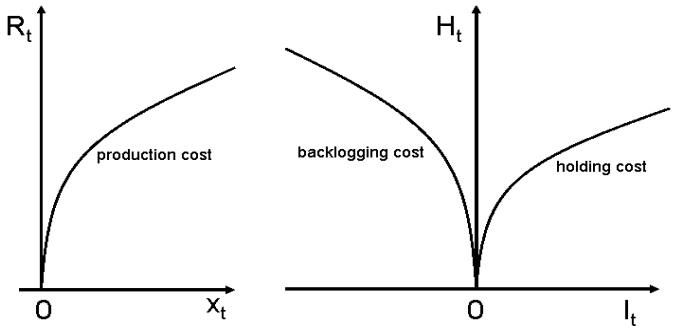

Figure 2.2: Cost functions.

of penalty.) In this case both backlogging and holding costs are combined in a single function for every time period. This function is called *holding-backlogging* cost function. Some production and holding-backlogging cost functions are depicted in Figure 2.2 where function $R_t(\cdot)$ is a production cost function at time period t, and $H_t(\cdot)$ is a holding-backlogging cost function. It is assumed that all cost functions are computable in polynomial time at any point of their domains. Sometimes, only to simplify the exposition, we will assume that cost functions are computable in constant time. In other words, function computation will often be considered as a basic operation. When we consider some special cases of the lot-sizing problem, in order to emphasis that additional restrictions on the cost structure are imposed, we will often use small letters for production and holding-backlogging cost functions, i.e., sometimes we write $r_t(\cdot)$ and $h_t(\cdot)$ instead of $R_t(\cdot)$ and $H_t(\cdot)$.

The task in the single-item capacitated economic lot-sizing (CELS) problem is to find a production plan (how much to produce in each period) with minimum cost.

The above formulation is a classical formulation of the CELS problem. The following results are available. Florian *et al.* [26] proved that this problem is NP-hard for the case when no backlogging is allowed, holding costs are linear and each production cost function is a sum of a fixed setup cost (arising every time when production takes place, i.e., $x_t > 0$) and a linear function. More precisely, the problem they consider has the following form:

$$
\begin{aligned}
\min \quad & \textstyle\sum_{t=1}^{n} r_t(x_t) + h_t I_t, & \\
\text{s.t.} \quad & I_t = I_{t-1} + x_t - d_t, & \forall t \in [1, n], \\
& I_0 = I_n = 0, & \\
& I_t \geq 0, & \forall t \in [1, n] \\
& 0 \leq x_t \leq u_t, & \forall t \in [1, n], \\
& x_t \in \mathbb{Z}, & \forall t \in [1, n].
\end{aligned}
$$

where $r_t(\cdot)$ is a production cost function taking value $s_t + c_t x_t$ if the value of argument x_t is positive and 0, otherwise. In this formulation coefficients h_t are holding costs of a product unit for each period t, values s_t are setup costs arising whenever the production level x_t is positive, and c_t are costs of production of a product unit. Florian *et al.* also proved NP-hardness of the problem under much more restrictive assumptions about cost and capacity structure. Florian and Klein [25] developed an $O(n^4)$ algorithm for the CELS problem where production and holding cost functions are concave. Bitran and Yanasse [7], Rosling [48], Chung and Lin [17] and van Hoesel and Wagelmans [58] developed polynomial time algorithms for special cases of the above formulated version of the CELS problem. The $O(n^3)$ algorithm of van Hoesel and Wagelmans is also able to solve that problem for a more general case when production costs $r_t(\cdot)$ are assumed to be concave without additional structural properties. Dynamic programming algorithms for the CELS problem were derived by Kirka [37], Chen *et al.* [13] and van Hoesel and Wagelmans [58]. Branch-and-bound approaches were suggested by Baker *et al.* [3], Erenguc and Aksoy [24], Chung *et al.* [13] and Lotfi and Yoon [42]. Approximation algorithms were proposed by Bitran and Matsuo [6], Gavish and Johnson [29] and van Hoesel and Wagelmans [57]. The strongest theoretical result on approximation algorithms for the CELS problem was obtained in the latter paper. Van Hoesel and Wagelmans developed a *fully polynomial time approximation scheme (FPTAS)* for the CELS problem with piecewise concave production and backlogging cost functions having a polynomial number of concave pieces.

2.2 An algorithm for the case with linear costs

2.2.1 Problem formulation

We consider the following instance of the CELS problem with a linear objective function:

$$\begin{aligned} \min \quad & \textstyle\sum_{t=1}^{n} c_t x_t \\ \text{s.t.} \quad & I_t = I_{t-1} + x_t - d_t, \quad \forall t \in [1,n], \\ & I_0 = 0, \\ & I_t \geq 0, \qquad \forall t \in [1,n], \\ & 0 \leq x_t \leq u_t, \qquad \forall t \in [1,n], \\ & x_t \in \mathbb{Z}, \qquad \forall t \in [1,n]. \end{aligned}$$

If the instance is feasible, i.e., the set of feasible solutions is not empty, then $d_1 \leq u_1$ due to the requirement that inventory levels should be nonnegative. If $d_t > u_t$ at some period $t \in [2,n]$, then, without changing the set of feasible solutions one can redefine demands at the previous and at the present period as $d_{t-1} := d_{t-1} + (d_t - u_t)$ and $d_t := u_t$. Therefore, without loss of generality we may assume that

$$d_t \leq u_t$$

for every time period $t \in [1,n]$. As a consequence, if $\sum_{i=1}^{t} d_i = \sum_{i=1}^{t} u_i$, then $x_i = u_i$ for all $i \in [1,t]$ in every feasible solution (x, I) and we may consider an equivalent lot-sizing problem for periods $t+1, \ldots, n$. Therefore, without loss of generality, we will assume that

$$\sum_{t=1}^{t} d_i < \sum_{t=1}^{t} u_i$$

for all $t \in [1,n]$.

In the above formulation coefficients c_t are per unit production costs. Note that a problem with linear holding cost functions reduces to a CELS problem instance of the above form. Namely, a cost function $\sum_{t=1}^{n} a_t x_t + \sum_{t=1}^{n} h_t I_t$ (where a_t are production costs) can be rewritten as

$$\sum_{t=1}^{n} a_t x_t + \sum_{t=1}^{n} h_t \sum_{i=1}^{t} (x_i - d_i) = \sum_{t=1}^{n} (a_t + \sum_{i=t}^{n} h_i) x_t - \sum_{t=1}^{n} h_t \sum_{i=1}^{t} d_i.$$

The term $-\sum_{t=1}^{n} h_t \sum_{i=1}^{t} d_i$ can be omitted without changing a set of optimal solutions. If we denote $a_t + \sum_{i=t}^{n} h_i$ by c_t for every $t \in [1, n]$, then we come to a problem with an objective function $\sum_{t=1}^{n} c_t x_t$.

In the paper of Girlich *et al.* [30] it is shown that Edmonds' greedy algorithm to solve linear problems on polymatroids can be adapted for the CELS problem with a linear objective function. That implementation of Edmonds' algorithm runs in $O(n \log n)$ time. (Note that some special cases of the linear CELS problem can be solved by an $O(n \log n)$ algorithm of Janiak and Kovalyov [36] that has been suggested for knapsack type problems.) In the present subsection we will show that there is an alternative way exploiting sensitivity properties of the problem. A simple dynamic programming algorithm that we will consider also runs in $O(n \log n)$ time.

2.2.2 Sensitivity analysis

For every $s \in [0, \sum_{i=1}^{t}(u_i - d_i)]$ (this interval consists of all those values that inventory level I_t may take provided that a solution is feasible) define $\phi_t(s)$ as an optimal value of a partial instance of the form

$$\begin{aligned}
\min \quad & \sum_{i=1}^{t} c_i x_i \\
\text{s.t.} \quad & I_i = I_{i-1} + x_i - d_i, \quad \forall i \in [1, t], \\
& I_0 = 0, \\
& I_i \geq 0, \qquad \forall i \in [1, t], \\
& I_t = s, \\
& 0 \leq x_i \leq u_i, \qquad \forall i \in [1, t], \\
& x_i \in \mathbb{Z}, \qquad \forall i \in [1, t].
\end{aligned}$$

We will assume that $\phi_0(0) = 0$. The following recursive formula is immediately implied by the above formulation:

$$\phi_t(s) = \min\{c_t x_t + \phi_{t-1}(s') | s = s' + x_t - d_t, s' \in [0, \sum_{i=1}^{t-1}(u_i - d_i)], x_t \in [0, u_t]\}. \qquad (2.1)$$

(Note that we suppose $\sum_{i=1}^{0}(u_i - d_i) = 0$ in this formula for $t = 1$.) Let $[b_1, b_2]$ be an integer interval. A function $\psi : [b_1, b_2] \to \mathbb{R}$ is called *convex* on the integer interval $[b_1, b_2]$ if and only if an inequality

$$\psi(a-1) + \psi(a+1) - 2\psi(a) \geq 0, \tag{2.2}$$

holds for all $a \in [b_1 + 1, b_2 - 1]$. Remind that, provided that $\psi(\cdot)$ is convex, $a_{\min}$ is a point at which $\psi(\cdot)$ attains its minimum if and only if

$$\psi(a_{\min}) = \min\{\psi(a) \mid a \in [b_1, b_2] \cap [a_{\min} - 1, a_{\min} + 1]\}.$$

Lemma 2.2.1 *For every $t \in [1, n]$, function $\phi_t(\cdot)$ is convex and piecewise linear with the number of pieces not exceeding t. Moreover, the following statements are true.*

1. $s < Q_{t-1} - d_t \Rightarrow \phi_t(s) = \phi_{t-1}(s + d_t)$.
2. $Q_{t-1} - d_t \leq s \leq u_t + Q_{t-1} - d_t \Rightarrow \phi_t(s) = c_t(s - Q_{t-1} + d_t) + \phi_{t-1}(Q_{t-1})$.
3. $s > u_t + Q_{t-1} - d_t \Rightarrow \phi_t(s) = c_t u_t + \phi_{t-1}(s - u_t + d_t)$.

where

$$Q_{t-1} = \max\left\{Q \in [1, \sum_{i=1}^{t-1}(u_i - d_i)] \mid \phi_{t-1}(Q) - \phi_{t-1}(Q-1) \leq c_t\right\} \cup \{0\}.$$

Proof. We will prove the lemma by induction.

It is easy to see that function $\phi_1(\cdot)$ is linear. A function value $\phi_1(s)$ has the form $\phi_1(s) = c_1 s + c_1 d_1$.

Let us make the following inductive assumptions for the time period $t \geq 2$:

(i) Function $\phi_{t-1}(\cdot)$ is convex.

(ii) Function $\phi_{t-1}(\cdot)$ is piecewise linear, each piece having a form $c_i\alpha + b$ where $i \in [1, t-1]$.

(iii) Statements 1,2, and 3 hold for the time period $t - 1$.

Enumerate all linear pieces of $\phi_{t-1}(\cdot)$ with integer numbers from 1 to some k_t. Let c_{i_j} be the slope value of the linear piece with number j. The convexity of function $\phi_{t-1}(\cdot)$ implies that

$$c_{i_1} \leq \ldots \leq c_{i_{k_t}}.$$

Choose index l so that

$$c_{i_1} \leq \ldots \leq c_{i_l} \leq c_t \leq c_{i_{l+1}} \leq \ldots \leq c_{i_{k_t}}. \tag{2.3}$$

We assume that $l = 0$ if $c_t \leq c_{i_j}$ for all $j \in [1, k_t]$. Let $\Delta_{t-1,p}$ denote a length of an integer interval corresponding to a linear piece of function $\phi_{t-1}(\cdot)$ with a slope value c_p. (Note that the minimum length of such interval is equal to one.) One can see from the definition of Q_{t-1} that $Q_{t-1} = \sum_{j=1}^{l} \Delta_{t-1,i_j}$. (If $l = 0$, then this value is equal to zero.) Speaking informally, this is the point after which we should pay at least c_t for every additional inventory unit at the end of period $t-1$.

Now we prove the first statement. Let s' and $x_t > 0$ deliver a minimum in formula (2.1). Since $s < Q_{t-1} - d_t$ and $s = s' + x_t - d_t$, we have $s' < Q_{t-1} - x_t \leq Q_{t-1}$. This implies the following inequality chain:

$$0 \leq c_t(x_t - 1) + \phi_{t-1}(s' + 1) - (c_t x_t + \phi_{t-1}(s')) =$$

$$= -c_t + \phi_{t-1}(s' + 1) - \phi_{t-1}(s') \leq [s' < Q_{t-1}] \leq -c_t + c_t = 0.$$

According to these inequalities we may conclude that, increasing s' by one and decreasing x_t by one we obtain new values of inventory level s' at the end of period $t-1$ and production level x_t also delivering a minimum in (2.1). In that way, among all pairs (x_t, s') at which a minimum is attained there is one with $x_t = 0$. This implies the first statement.

We prove the second statement. Let some s' such that $s' \neq Q_{t-1}$ and x_t deliver a minimum in (2.1). Since $Q_{t-1} - d_t \leq s \leq u_t + Q_{t-1} - d_t$, value $s - Q_{t-1} + d_t$ belongs to the set $[0, u_t]$.

Consider two cases.

a) $s' \leq Q_{t-1}$.

Note that in this case $s' < Q_{t-1}$ due to our assumption that $s' \neq Q_{t-1}$. Due to the choice of Q_{t-1} we have

$$\frac{\phi_{t-1}(Q_{t-1}) - \phi_{t-1}(s')}{Q_{t-1} - s'} \leq c_t$$

and consequently

$$\phi_{t-1}(Q_{t-1}) \leq c_t(Q_{t-1} - s') + \phi_{t-1}(s')$$

which implies

$$c_t(s - Q_{t-1} + d_t) + \phi_{t-1}(Q_{t-1}) \leq c_t(s - Q_{t-1} + d_t) + c_t(Q_{t-1} - s') + \phi_{t-1}(s') =$$

$$= c_t(s - s' + d_t) + \phi_{t-1}(s') = c_t x_t + \phi_{t-1}(s').$$

b) $s' > Q_{t-1}$.

In this case we have

$$\frac{\phi_{t-1}(s') - \phi_{t-1}(Q_{t-1})}{s' - Q_{t-1}} \geq c_t$$

from which we conclude that

$$c_t(s' - Q_{t-1}) \leq \phi_{t-1}(s') - \phi_{t-1}(Q_{t-1}).$$

Taking into account this inequality, we may write

$$c_t(s - Q_{t-1} + d_t) + \phi_{t-1}(Q_{t-1}) = c_t(s' + x_t - Q_{t-1}) + \phi_{t-1}(Q_{t-1}) \leq$$

$$\leq c_t x_t + \phi_{t-1}(s') - \phi_{t-1}(Q_{t-1}) + \phi_{t-1}(Q_{t-1}) = c_t x_t + \phi_{t-1}(s').$$

It follows from both cases that $x_t = s - Q_{t-1} + d_t$ and $s' = Q_{t-1}$, as well as the previous values of x_t and s', are among those at which a minimum in formula (2.1) is attained.

Statement 3 says that if we set the production level x_t at capacity u_t, then we have a minimum in formula (2.1) provided that inventory level s exceeds value $u_t + Q_{t-1} - d_t$. To prove this statement, we show that if the minimum in (2.1) is attained at $x_t < u_t$, then decreasing s' by one and increasing x_t by one does not increase the overall cost. Notice that, since condition $s > u_t + Q_{t-1} - d_t$ holds, we have $s' + x_t - d_t > u_t + Q_{t-1} - d_t$ from which

we deduce $s' > u_t + Q_{t-1} - x_t > Q_{t-1}$ using our assumption that $x_t < u_t$. Therefore, due to the fact that $s' > Q_{t-1}$, we have

$$\phi_{t-1}(s') - \phi_{t-1}(s' - 1) \geq c_t.$$

This inequality implies

$$\phi_{t-1}(s') + c_t x_t - (\phi_{t-1}(s' - 1) + c_t(x_t + 1)) = \phi_{t-1}(s') - \phi_{t-1}(s' - 1) - c_t \geq 0.$$

Thus, decreasing s' by one and increasing x_t by one we still get a minimum in formula (2.1). This implies the statement 3. ∎

Note that further we will widely use the notation introduced in the proof of the above lemma as well as statements 1-3 proved there.

2.2.3 Dynamic programming algorithm

We may represent function $\phi_t(\cdot)$ as a list $\mathcal{L}_t$ each item of which corresponds to a linear piece. A slope value of each linear piece is one of the coefficients c_i, $i \in [1, t]$, from Lemma 2.2.1. At each list item it is therefore sufficient to store only a corresponding period number i to have the information about a slope value. Moreover, we will not further need an explicit computation of function $\phi_t(\cdot)$ but only its slope. Therefore we store only the following parameters at every item of list $\mathcal{L}_t$:

- number of period i, and
- value $\Delta_{t,i}$.

The dynamic programming algorithm described below follows Lemma 2.2.1. At each iteration of the for-loop of Step 1 it constructs list $\mathcal{L}_t$ representing function $\phi_t(\cdot)$ using the previously constructed list $\mathcal{L}_{t-1}$. Step 2 of the algorithm is a backtracking step at which an optimal solution is constructed.

Algorithm DP

Step 1 $Q_0 := 0$;

$\Delta_{1,1} := u_1 - d_1$;
$\mathcal{L}_1 := \{(1, \Delta_{1,1})\}$; //$\phi_1(\cdot)$ consists of a single linear piece
for $t = 2$ **to** n **do**
let $(i, \Delta_{t-1,i})$ be a head of list $\mathcal{L}_{t-1}$;
insert a new item (t, u_t) into this list
so that list items remain sorted in the nondecreasing order of coefficients c_i;
$s := \Delta_{t-1,i}$;
while $s < d_t$ **do**
delete head $(i, \Delta_{t-1,i})$ of list $\mathcal{L}_{t-1}$;
$s := s + \Delta_{t-1,i}$;
end do
if $s > d_t$ **then**
$(i, \Delta_{t-1,i}) := (i, s - d_t)$;
end if
denote the obtained list by $\mathcal{L}_t$;
if $t < n$ **then** compute Q_t; **end if**
end do
Step 2 $s := 0$;
for $t = n$ **down to** 1 **do**
if $s < Q_{t-1} - d_t$ **then** $x_t := 0$; $s := s + d_t$;
else if $Q_{t-1} - d_t \le s \le u_t + Q_{t-1} - d_t$ **then** $x_t := s - Q_{t-1} + d_t$; $s := Q_{t-1}$;
else if $s > u_t + Q_{t-1} - d_t$ **then** $x_t := u_t$; $s := s - u_t + d_t$; **end if**
end if
end do

The second step of the algorithm uses the proof of statements 1-3 in the proof of Lemma 2.2.1. Just before the while loop of Step 1 starts, the current list $\mathcal{L}_{t-1}$ represents function $\phi_t(s)$ as this function would look like provided that there would be no restriction $s \ge 0$ for its

argument s. The while loop takes into account this restriction by removing an appropriate number of first list items. Recall that without loss of generality we have assumed $d_t \leq u_t$. The value $u_t + Q_{t-1} - d_t$ is therefore nonnegative. This means, by the second statement of Lemma 2.2.1, that an item of the form (t, Δ) is contained in the list $\mathcal{L}_t$ constructed at the tth iteration of the while loop. If $Q_{t-1} - d_t > 0$, then, by Lemma 2.2.1, $\Delta = u_t$.

Note that it is easy to calculate Q_t in $O(n)$ time by exploring list $\mathcal{L}_t$ at every iteration of the for loop at Step 1. There is however a better way allowing to spend only $O(\log n)$ time for calculation of Q_t at each iteration. To show this, we will later assume that the set of items of list $\mathcal{L}_{t-1}$ is maintained at the same time as a 2-3 tree. The 2-3 trees form a subclass of so-called B-trees which were developed by Bayer and McCreight [4]. Let us discuss in more detail what a 2-3 tree is.

Any 2-3 tree may contain three different kinds of nodes:

a) leaves (data nodes),

b) 2-nodes, and

c) 3-nodes.

Leaves of a 2-3 tree contain the data we want to store and access. Others, both 2- and 3-nodes, are so-called information nodes that are only intended to guide us if we want to access leaves, delete them, or insert a new one.

We suppose that every leaf is uniquely identified with a key value k (or simply a key). We will assume that every leaf also contains an integer number that we call a data parameter. Later, when using a 2-3 tree to store the items (i, Δ_i) of lists $\mathcal{L}_t$, we will suppose that key values k of leaves are slope values c_i of corresponding linear pieces. Data parameter values will be equal to corresponding values Δ_i.

A 2-node has two children. For each of them there are links that we denote as l and r. (We will also refer to corresponding subtrees using the same symbols l and r.) For every 2-node a value v is given. (It is called a value of this node.) The following condition should hold:

(c_1): Every value appearing in subtree l must be less than v; and every value appearing in subtree r must be not less than v.

Subtree l is called left subtree, and subtree r is called right subtree of a corresponding node.

A 3-node has three children which means that it contains three links, denote them as l, m, and r, each pointing to a root of a so-called left, middle, and right subtree, respectively. Values v_l and v_r should be identified for each 3-node so that the following condition holds:

(c_2): Every value appearing in subtree l must be less than v_l; every value appearing in subtree m must be not less than v_l and less than v_r; and every value in subtree r must be not less than v_r.

In both the conditions (c_1) and (c_2) under values appearing in a tree we mean values v, v_l, and v_r of nodes belonging to this subtree, and key values of leaves contained in it.

The following condition is also necessary for any 2-3 tree.

(c_3): All paths from a root of any subtree to a leaf must be of equal length.

A 2-3 tree is so far a tree satisfying conditions (c_1-c_3).

Further, if we say "a subtree corresponding to a node N" or "a subtree induced by a node N", we mean a part of the tree consisting of N (this is a root of a subtree) and all its successors linked in the same way as in the original tree. (Remind that N' is a successor of N if there is a sequence of nodes $N, N_1, \ldots, N_k, N'$ such that every node of this sequence, except N, is a child of a previous one.)

To refer to a component of a node we will use a symbol denoting a node and a name of this component. For instance, a record like $N.r$ will mean a link to the root of the right subtree of information node N. A record like $L.k$ will mean a key value of a leaf L. Notation $L.d$ will be used for a data parameter value of a leaf L.

If N_1 and N_2 are children of a node N, then N_1 is called a sibling of N_2, and, vice versa, N_2 is called a sibling of N_1. If N is for instance a 3-node, then its children $N.l$, $N.m$, and $N.r$ are siblings of each other.

Due to the construction of a 2-3 tree, a number of nodes at every level is at most a half of a number of nodes at the lower level. This means that if p is the number of tree leaves,

then the height of the tree is upper bounded by $O(\log p)$. This also implies that the number of information nodes is not larger than the number of leaves.

We will use a 2-3 tree for the purpose of efficiently computing a sum Q of all data parameter values at leaves whose key values k do not exceed some given value. Therefore, we assume that a value $sum(N)$ is stored at every node N, which is a sum of all data parameter values at leaves of the subtree with root N. If N is a leaf, then $sum(N)$ is a value of the data parameter. More precisely,

$$sum(N) = \sum_{L \text{ is a leaf of the subtree induced by } N} L.d$$

where the sum is taken over all leaves of the subtree induced by node N. The following algorithm computes the mentioned value Q in $O(\log p)$ time.

Algorithm 2.2.1

let N be a leaf with maximum key value which is not greater than a given value;
$Q := sum(N)$;
while N is not the root node **do**
    $\hat{N} := N$; // store current node
    let N be a parent node of $\hat{N}$; // go to the next node
    if $N.r = \hat{N}$ **then** // if $\hat{N}$ is a root of the right subtree
      $Q := Q + sum(N.l) + sum(N.m)$;// assume $sum(N.m) = 0$ if N is a 2-node
    end if
    if $N.m = \hat{N}$ **then** // if $\hat{N}$ is a root of the middle subtree
      $Q := Q + sum(N.l)$;
    end if
end do

In this algorithm, we move along a path from a leaf to the root and therefore the while-loop performs at most $O(\log p)$ iterations which is an upper estimate of the tree height.

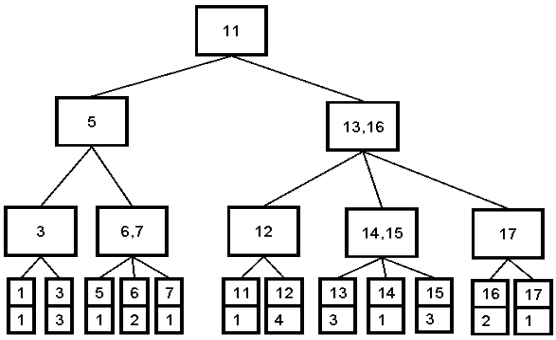

Figure 2.3: A 2-3 tree.

In an example of a 2-3 tree in Figure 2.3, big rectangles correspond to information nodes. For 2-nodes, numbers inside rectangles indicate values v. In the case of 3-nodes both values v_l and v_r are given. Data nodes, i.e., tree leaves, are depicted in the form of pairs of small rectangles. An upper number of each pair is a key value and a lower number is a value of the data parameter. It can be seen from the picture that $sum(N) = 1 + 3 + 1 + 2 + 1 = 8$ for the node N with $N.v = 5$.

Deletion and insertion operations for 2-3 trees make sense only with respect to data nodes, i.e., leaves. Both insertion of a new item into a 2-3 tree and deletion of a node takes $O(\log p)$ time. Although these operations are well described in the literature concerning algorithms and data structures, we need to discuss them here in order to show the way how values $sum(N)$ are maintained at tree nodes N.

We will not discuss how values v, v_l, and v_r are recalculated during insertion and deletion operations because the detailed algorithms for handling 2-3 trees can be found in the literature. We will just observe tree nodes in the order in which they are processed by standard insertion/deletion procedures and show how to compute values $sum(\cdot)$.

An insertion of a new leaf L into a 2-3 tree begins with processing an appropriate *terminal node* which is a node whose children are leaves. Denote this node by N. If it is a 2-node, then L becomes one of the children of this terminal node. (Node N becomes a 3-node in this case.)

If it is a 3-node, then we need to apply more efforts since a 2-3 tree must remain a 2-3 tree after an insertion operation. Node N is split then into two ones, denote them as N_1 and N_2, each receiving two children. Sums $sum(N_1)$ and $sum(N_2)$ are easily computable using the values $sum(\cdot)$ for their children. A parent node of N, denoted by P, becomes a parent node for nodes N_1 and N_2. If P has been a 2-node, then we correct values $sum(\cdot)$ of nodes along the path from P to the root by adding value $sum(L)$. Otherwise, we split it into two 2-nodes P_1 and P_2 in the same way as node N. Then $sum(P_1)$ takes value $sum(P_1.l) + sum(P_1.r)$ and $sum(P_2)$ takes value $sum(P_2.l) + sum(P_2.r)$. We proceed until the root is processed. If the root must be split, then we form a new root having two children. Since every time we move to a higher level of the tree, an insertion takes $O(\log p)$ time. Note that one has to explore all nodes on the path to the root node since information about the sums must be corrected for each of these nodes. An insertion operation with the recalculation of values $sum(\cdot)$ takes so far $O(\log p)$ time.

Let us now discuss how deletion works. Suppose that we want to delete some leaf L whose parent is node P. Set $\delta := sum(L)$. Two situations may occur:

1. Node P is a 3-node. In this case, recalculate $sum(N)$ as

$$sum(N) := sum(N) - \delta$$

   for every node on the path from P to the root node. Delete L.

2. Node P is a 2-node. Recalculate $sum(P)$ as $sum(P) := sum(P) - \delta$. Delete L.

   a. A sibling $\hat{P}$ of P has three children. In this case we may take an appropriate child $\hat{L}$ from node $\hat{P}$ and make $\hat{L}$ a child of node P. Recalculate $sum(\hat{P})$ as $sum(\hat{P}) := sum(\hat{P}) - sum(\hat{L})$. Recalculate $sum(P)$ as $sum(P) := sum(P) + sum(\hat{L})$.

   b. The sibling of P has only two children. In this case we make $\hat{L}$, a sibling of L, a child of $\hat{P}$, recalculate $sum(P)$ as $sum(P) := 0$ and $sum(\hat{P})$ as $sum(\hat{P}) := sum(\hat{P}) + sum(\hat{L})$, and make the following variable reassignment: $L := P$, and $P :=$ parent node of P.

If case 2b took place and P is not a root node, proceed the actions suggested by items 1 and 2 for the new nodes L and P. Otherwise, if P is a root node, delete both L and P. (In this case a sibling of L becomes a root node.) If not, recalculate $sum(N)$ for all nodes on the path from P to the root node in the same way as in the case 1.

Using the fact that the tree height is upper bounded by $O(\log p)$, we may conclude that the deletion operation takes at most $O(\log p)$ time. Both described procedures for insertion and deletion provide correct values $sum(N)$ for all information nodes N.

As we have already mentioned, we will use 2-3 trees to store items of lists $\mathcal{L}_t$. Each list item of the form (t, Δ) will be stored at some leaf L whose key value and data parameter value are defined as $L.k = c_t$ and $L.d = \Delta$, respectively. It might happen that there are two equal coefficients among costs c_t. Therefore, we should adapt a 2-3 tree for this case. This can easily be done if we allow to store a sorted list of items with equal values c_t at each leaf. (Assume that we sort items in the order in which they stand in list $\mathcal{L}_t$.) Then, if we need to delete an item from the tree, we first look whether this list contains a single item. If it does, we perform the deletion procedure for a 2-3 tree which has been already described. Otherwise, we just delete that item from the mentioned sorted list of items corresponding to the same coefficients c_t. The insertion operation works in the same way as before, but at first determines if an item with a key value c_t exists in the tree. If it is the case, then the item we want to insert has to be added to a sorted list of items corresponding to the equal coefficients c_t. Note that, after such adaptation, deletion and insertion operations have the same complexity estimations as before since we may access an item of a sorted list in the time upper bounded by a logarithm of the number of items in this list.

Theorem 2.2.1 *Algorithm DP can be implemented to run in $O(n \log n)$ time.*

Proof. Let us assume that at every iteration of the for loop of Step 1 of algorithm DP items of list $\mathcal{L}_{t-1}$ are organized as a 2-3 tree, i.e., each of the items of this list is a leaf of a 2-3 tree. Denote this tree by $\mathcal{T}$. Assume that for every list item $(i, \Delta_{t-1,i})$ coefficient c_i is taken as a key value of a corresponding leaf. Let us also assume that $\Delta_{t-1,i}$ is a value of data parameter of this leaf. Whenever we delete an item from or insert an item into list

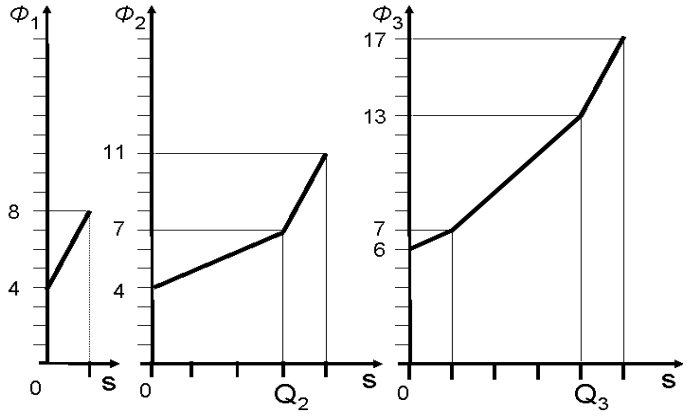

Figure 2.4: Recursive functions.

$\mathcal{L}_{t-1}$, we will perform an appropriate insertion and deletion in tree $\mathcal{T}$. Note that, in the for loop of Step 1, an item corresponding to a certain time period is deleted only once and never appears at further iterations. This means that the algorithm performs at most $O(n)$ deletions at Step 1. The same can be said with respect to insertion operations. The insertion of item (t, u_t) can be performed in $O(\log n)$ time using the binary search in tree $\mathcal{T}$. (Remind that we want list items to be sorted in nondecreasing order of coefficients c_i.) Taking into account the previous discussion about 2-3 trees, we may conclude that the total time needed to maintain tree $\mathcal{T}$ after deletion operations in the for loop is upper bounded by $O(n \log n)$. The values Q_t by definition in Lemma 2.2.1 are sums of those components $\Delta_{t,i}$ of items $(i, \Delta_{t,i})$ of list $\mathcal{L}_t$ which correspond to coefficients c_i which do not exceed c_{t+1}. Following the discussed properties of 2-3 trees and Algorithm 2.2.1, we can calculate such Q_t in $O(\log n)$ time at every iteration. Hence, provided that we use a 2-3 tree in the described way, Step 1 runs in $O(n \log n)$ time. Therefore, since Step 2 runs in $O(n)$ time, we may conclude that algorithm DP can be implemented to run in $O(n \log n)$ time. ∎

Consider an example of the linear lot-sizing problem with $c = (4, 1, 2, 3)$, $u = (2, 3, 3, 2)$, and $d = (1, 0, 2, 2)$. The functions ϕ_1, ϕ_2, and ϕ_3 are depicted in Figure 2.4. From the graph of function ϕ_3 it can be seen that it is obtained by placing a linear piece with the slope value $c_3 = 2$ between two linear pieces of function ϕ_2 and reducing the length of the

t	s	Q_{t-1}	d_t	x_t
4	0	4	2	0
3	2	3	2	1
2	3	0	0	3
1	0	0	1	1

Table 2.1: An example of how the backtracking step works.

interval corresponding to the first linear piece by the demand $d_3 = 2$. This is exactly what algorithm DP suggests to do when constructing function ϕ_3. The values arising in the course the backtracking phase of the algorithm are enclosed in Table 2.1.

2.3 Exponential algorithm

2.3.1 Motivation

Many NP-hard problems can be solved by exhaustive search. Complete enumeration of all tours for an instance of the travelling salesman problem yields an $O(n!)$ algorithm where n is the number of cities. All solutions of an instance of the Knapsack Problem can be enumerated in $O(2^n)$ time where n is the number of items to put into the knapsack. If input coefficients are sufficiently large, these enumeration methods seem to be more preferable exact methods than those running in pseudopolynomial time. From the first point of view there is nothing to do in the area of enumerative algorithms; just enumerate all solutions and output a result. Nevertheless, for many problems there exist algorithms which are exponentially better than other ones. This difference is indeed similar to the difference between polynomial and exponential algorithms. Consider just a few results in the area of exact algorithms.

A simple dynamic programming algorithm of Held and Karp (see [33]) finds an optimal tour for the travelling salesman problem in $O(2^n)$ time. This complexity estimation is considerably better than $O(n!)$ time needed for the exhaustive search of all solutions. The

algorithm of Held and Karp yields the best theoretical complexity estimation among all algorithms for the travelling salesman problem known till now.

The algorithm of Horowitz and Sahni [34] for any instance of the KNAPSACK PROBLEM with n items runs in $O(n2^{n/2})$ time. For those instances where some coefficients are greater than or comparable with $2^{n/2}$, this algorithm can be more attractive then those running in pseudopolynomial time. An $O(2^n)$ enumeration algorithm exploring all feasible solutions differs by an exponential factor from the running time of the algorithm of Horowitz and Sahni. Therefore algorithms like the one of Horowitz and Sahni are often called *subexponential*.

In the next subsection we design a subexponential dynamic programming algorithm for the CELS problem. At the same time this dynamic programming algorithm is pseudopolynomial.

2.3.2 The CELS problem with piecewise linear cost functions

We consider the capacitated economic lot-sizing problem of the following form.

$$\begin{aligned}
\min \quad & F(x, I) = \textstyle\sum_{t=1}^{n} (r_t(x_t) + h_t(I_t)), & & \\
\text{subject to} \quad & I_t = I_{t-1} + x_t - d_t, & \forall t \in [1, n], & \quad (2.4)\\
& I_0 = I_n = 0 & & \quad (2.5)\\
& 0 \le x_t \le u_t, & \forall t \in [1, n], & \quad (2.6)\\
& x_t \in \mathbb{Z}, & \forall t \in [1, n]. & \quad (2.7)
\end{aligned}$$

Note that in this formulation we consider the additional restriction $I_n = 0$ which is common for lot-sizing models. Due to this restriction, there exists a feasible solution if and only if

$$\sum_{t=1}^{n} u_t \ge \sum_{t=1}^{n} d_t. \tag{2.8}$$

Thus without loss of generality we may assume that this condition holds.

In this section, we consider an instance $\mathcal{I}$ of the above problem where cost functions are piecewise linear. Suppose that for all $t \in [1, n]$ functions $r_t(\cdot)$ and $h_t(\cdot)$ may only be not linear at points from some sets $\mathcal{R}_t$ and $\mathcal{H}_t$, respectively. To simplify the exposition, we

assume that $|\mathcal{R}_t|$ is equal to some value R and $|\mathcal{H}_t|$ is equal to some value H for any time period t. We assume that function $r_t(\cdot)$ is defined on the interval $[0, u_t]$ and function $h_t(\cdot)$ is defined on some interval $[b_1^t, b_2^t]$ (later we specify bounds b_1^t and b_2^t as the minimum and the maximum value at which a function $\phi_t(\cdot)$ is defined, respectively). This means that $\mathcal{R}_t$ contains end points 0 and u_t of the interval $[0, u_t]$ and $\mathcal{H}_t$ contains end points b_1^t and b_2^t. Therefore R and H are greater than or equal to 2. The number of linear pieces of functions $r_t(\cdot)$ and $h_t(\cdot)$ is equal to $R-1$ and $H-1$, respectively.

We define the function value $\phi_t(s)$ as the optimal value of the reduced instance

$$\begin{aligned}
\min \quad & F^t(x, I) = \textstyle\sum_{i=1}^{t}(r_i(x_i) + h_i(I_i)), & & \\
\text{subject to} \quad & I_i = I_{i-1} + x_i - d_i, & \forall i \in [1, t], & (2.9)\\
& I_t = s, & & (2.10)\\
& I_0 = 0, & & (2.11)\\
& 0 \le x_i \le u_i, & \forall i \in [1, t], & (2.12)\\
& x_i \in \mathbb{Z} & \forall i \in [1, t]. & (2.13)
\end{aligned}$$

for every $s \in [b_1^t, b_2^t]$ where bounds b_1^t and b_2^t are the minimum and maximum values such that there exists a feasible solution (x, I) of the original instance $\mathcal{I}$ satisfying $I_t = s$.

Let us show that

$$b_1^t = -\sum_{i=1}^{t} d_i + \max\{0, \sum_{i=1}^{n} d_i - \sum_{i=t+1}^{n} u_i\} \tag{2.14}$$

and

$$b_2^t = \min\{\sum_{i=1}^{t}(u_i - d_i), \sum_{i=t+1}^{n} d_i\}. \tag{2.15}$$

For every feasible solution (x, I),

$$0 = I_n = I_{n-1} + x_n - d_n = \ldots = I_t + \sum_{i=t+1}^{n}(x_i - d_i),$$

from which we conclude that

$$I_t = \sum_{i=t+1}^{n}(d_i - x_i).$$

Constraints $0 \le x_i \le u_i$, $\forall i \in [1, n]$, imply that inequalities

$$\sum_{i=t+1}^{n} (d_i - u_i) \le I_t \le \sum_{i=t+1}^{n} d_i$$

hold. Moreover,

$$-\sum_{i=1}^{t} d_i \le I_t \le \sum_{i=1}^{t} (u_i - d_i)$$

hold. This inequality system is equivalent to

$$\max\{-\sum_{i=1}^{t} d_i, \sum_{i=t+1}^{n} (d_i - u_i)\} \le I_t \le \min\{\sum_{i=1}^{t} (u_i - d_i), \sum_{i=t+1}^{n} d_i\}.$$

It remains to show that there are feasible solutions satisfying $I_t = b_1^t$ and $I_t = b_2^t$. It is sufficient to consider solutions (x^1, I^1) and (x^2, I^2) where

$$x^1 = (0, \ldots, 0, \sum_{i=1}^{n} d_i - \sum_{i=\tau}^{n} u_i, u_\tau, \ldots, u_n),\ \tau - 1 = \max\{j \in [1, n] | \sum_{i=1}^{n} d_i - \sum_{i=j+1}^{n} u_i \in [0, u_j]\},$$

and

$$x^2 = (u_1, \ldots, u_\tau, \sum_{i=1}^{n} d_i - \sum_{i=1}^{\tau} u_i, 0, \ldots, 0),\ \tau + 1 = \min\{j \in [1, n] | \sum_{i=1}^{n} d_i - \sum_{i=1}^{j-1} u_i \in [0, u_j]\}.$$

The existence of such solutions is provided by feasibility condition (2.8). One may make sure that

$$I_t^1 = \max\{-\sum_{i=1}^{t} d_i, \sum_{i=t+1}^{n} (d_i - u_i)\}$$

and

$$I_t^2 = \min\{\sum_{i=1}^{t} (u_i - d_i), \sum_{i=t+1}^{n} d_i\}.$$

This implies (2.14) and (2.15).

We need an algorithm efficiently computing function $\phi_t(\cdot)$ not only at a single point in the feasible interval, but at all its points at a time. Note that if for some function $\psi(\cdot)$ we know that it is linear on an interval $[s_1, s_2]$, then, to completely describe the function $\psi(\cdot)$ on the interval $[s_1, s_2]$, it is sufficient to have information about its values at the end points of the interval. This simple observation shows that to have complete information about a piecewise linear function it is sufficient to compute it at ends of linear pieces.

Let $\mathcal{S}_i$ denote the set of all points where $\phi_i(\cdot)$ may be nonlinear. Set $\mathcal{S}_0 := \{0\}$. The following theorem yields an algorithm to find values of function $\phi_t(\cdot)$ at points of the set $\mathcal{S}_t$.

Theorem 2.3.1 *Let all values of $\phi_{t-1}(\cdot)$ at all points from $\mathcal{S}_{t-1}$ be given. Then the set $\mathcal{S}_t$ with associated function values $\phi_t(s)$, $s \in \mathcal{S}_t$, can be found in $O(R|\mathcal{S}_{t-1}| + H)$ time.*

Proof. Suppose that $\mathcal{R}_t = \{\alpha_1, \ldots, \alpha_R\}$, $\mathcal{H}_t = \{\beta_1, \ldots, \beta_H\}$, and $\mathcal{S}_{t-1} = \{s_1, \ldots, s_S\}$, where $s_1 \le \ldots \le s_S$. For all $k \in [2, R]$ and $l \in [2, S]$ define auxiliary functions $\psi_{kl}(\cdot)$ as

$$\psi_{kl}(s) = \min_{s' \in [s_{l-1}, s_l]} \{\phi_{t-1}(s') + r_t(s - s' + d_t) + h_t(s) | s - s' + d_t \in [\alpha_{k-1}, \alpha_k]\}.$$

Function value $\psi_{kl}(s)$ is defined for all

$$s \in [\alpha_{k-1} + s_{l-1} - d_t, \alpha_k + s_l - d_t]. \tag{2.16}$$

The functions $\psi_{kl}(\cdot)$ can be used to calculate $\phi_t(s)$ at any feasible point s by the formula

$$\phi_t(s) = \min_{k,l} \psi_{kl}(s).$$

The set of all values $\psi_{kl}(s)$, where s belongs to $\mathcal{S}_t$ and is feasible for $\psi_{kl}(\cdot)$, can be found in $O(|\mathcal{S}_{t-1}|)$ time using the fact that functions $r_t(\cdot)$ and $\phi_{t-1}(\cdot)$ are linear on the intervals $[\alpha_{k-1}, \alpha_k]$ and $[s_{l-1}, s_l]$, respectively. Note that a function value $\phi_t(s)$ is computable in $O(R|\mathcal{S}_{t-1}|)$ time provided that necessary values $\psi_{kl}(s)$ have been already found.

Let the slope value of function $\phi_{t-1}(\cdot)$ on interval $[s_{l-1}, s_l]$ be equal to a and the slope value of function $r_t(\cdot)$ on interval $[\alpha_{k-1}, \alpha_k]$ be equal to b. Then we may write

$$\psi_{kl}(s) = \min_{s' \in [s_{l-1}, s_l]} \{(a - b)s' | s - s' + d_t \in [\alpha_{k-1}, \alpha_k]\} + b(s + d_t) + h_t(s). \tag{2.17}$$

One can see that

$$s' \in [\max\{s_{l-1}, -\alpha_k + s + d_t\}, \min\{s_l, -\alpha_{k-1} + s + d_t\}].$$

The minimum of value $(a - b)s'$ in formula (2.17) is attained at one of the end points of this interval. Taking additionally into account (2.16), we may conclude that the function $\psi_{kl}(\cdot)$ may be nonlinear only at points of the form $s = \alpha_p + s_q - d_t$, where $p \in [1, R]$ and

$q \in [1, |\mathcal{S}_{t-1}|]$, or $s \in \mathcal{H}_t$. Therefore there are at most $O(R|\mathcal{S}_{t-1}| + H)$ points where function $\phi_t(\cdot)$ may violate linearity. If values of functions $\phi_{t-1}(\cdot)$ at points of set $\mathcal{S}_{t-1}$ are given, then the set $\mathcal{S}_t$ with associated function values $\phi_t(s)$, $s \in \mathcal{S}_t$, can be found in $O(R|\mathcal{S}_{t-1}| + H)$ time. ∎

Corollary 2.3.1 *The cardinality of set $\mathcal{S}_t$ is upper bounded by value $O(R^t H)$.*

Proof. It follows from the proof of the above theorem that $|\mathcal{S}_t|$ is upper bounded by $R|\mathcal{S}_{t-1}| + H$. Therefore, since $|\mathcal{S}_0| = |\{0\}| = 1$, we have

$$|\mathcal{S}_t| \leq R^t + R^{t-1}H + \ldots + RH + H.$$

The value $R^{t-1} + \ldots + R + 1$ does not exceed R^t as $R \geq 2$. Thus the cardinality of set $\mathcal{S}_t$ is upper bounded by $2R^t H$. ∎

Let $\tau = \lceil n/2 \rceil$. We will use a dynamic programming algorithm based on the recursive function $\phi_t(\cdot)$ only to find a part of an optimal solution corresponding to periods $1, \ldots, \tau$. The remaining part will be found by a dynamic programming algorithm based on functions $\widetilde{\phi}_t(\cdot), t \in [1, n,]$ such that a function value $\widetilde{\phi}_{t-1}(s)$, $t \in [2, n+1]$ is equal to an optimal value of a reduced instance

$$\begin{aligned}
\min \quad & \widetilde{F}^t(x, I) = \textstyle\sum_{i=t}^{n}(r_i(x_i) + h_i(I_i)), & & \\
\text{subject to} \quad & I_i = I_{i-1} + x_i - d_i, & \forall i \in [t, n], & \quad (2.18)\\
& I_{t-1} = s, & & \quad (2.19)\\
& I_n = 0, & & \quad (2.20)\\
& 0 \leq x_i \leq u_i, & \forall i \in [t, n], & \quad (2.21)\\
& x_i \in \mathbb{Z} & \forall i \in [t, n]. & \quad (2.22)
\end{aligned}$$

Function $\widetilde{\phi}_t(\cdot)$ is defined on the interval $[b_1^{t-1}, b_2^{t-1}]$, i.e., on the set of all feasible values of the variable I_{t-1}.

Function $\widetilde{\phi}_t(\cdot)$ can be calculated recursively by the formula

$$\widetilde{\phi}_{t-1}(s) = \min_{s' \in [b_1^t, b_2^t]} \{\widetilde{\phi}_t(s') + r_t(s' - s + d_t) + h_t(s') | s' - s + d_t \in [0, u_t]\}$$

at any feasible s.

Let $\widetilde{\mathcal{S}}_i$ denote the set of all points where $\widetilde{\phi}_i(\cdot)$ may be nonlinear. Set $\widetilde{\mathcal{S}}_{n+1} := \{0\}$. Properties of the function $\widetilde{\phi}_t(\cdot)$ are much the same as those of $\phi_t(\cdot)$. In particular, if values of $\widetilde{\phi}_{t+1}(\cdot)$ at all points from $\widetilde{\mathcal{S}}_{t+1}$ are given, then one can find all function values $\widetilde{\phi}_t(s)$, $s \in \widetilde{\mathcal{S}}_t$, in $O(R|\widetilde{\mathcal{S}}_{t+1}| + H)$ time. The cardinality of the set $\widetilde{\mathcal{S}}_t$ is upper bounded by $O(R^{n-t+1}H)$. The proof is completely analogous to that for function $\phi_t(\cdot)$.

The optimal value of the instance $\mathcal{I}$ we consider is calculated as

$$OPT(\mathcal{I}) = \min_{s \in [b_1^\tau, b_2^\tau]} \{\phi_\tau(s) + \widetilde{\phi}_\tau(s)\}.$$

This value is a minimum of function $\phi_\tau(\cdot) + \widetilde{\phi}_\tau(\cdot)$ on its feasible set $[b_1^\tau, b_2^\tau]$. This function may be nonlinear only at points of the set $\mathcal{S}_\tau \cup \widetilde{\mathcal{S}}_\tau$. Piecewise linear functions attain their optima at points where linearity conditions are violated (note that end points of feasible intervals also are among these points). Therefore, provided that functions $\phi_\tau(\cdot)$ and $\widetilde{\phi}_\tau(\cdot)$ have been evaluated on the sets $\mathcal{S}_\tau$ and $\widetilde{\mathcal{S}}_\tau$, respectively, the value $OPT(\mathcal{I})$ is computable in $O(|\mathcal{S}_\tau| + |\widetilde{\mathcal{S}}_\tau|)$ time. Hence the above mentioned computability properties of functions $\phi_t(\cdot)$ and $\widetilde{\phi}_t(\cdot)$ and sets $\mathcal{S}_t$ and $\widetilde{\mathcal{S}}_t$ imply that $OPT(\mathcal{I})$ can be computed in $O(nR^{n/2+1}H)$ time.

A backtracking procedure finding an optimal solution is easily implementable to run in $O(nR^{n/2+1}H)$ time. Thus we have the following theorem.

Theorem 2.3.2 *The instance $\mathcal{I}$ is solvable in $O(nR^{n/2+1}H)$ time.*

If we additionally consider restrictions $I_t \geq 0$, $t \in [1, n]$, then for any $t \in [1, n]$ function $\phi_t(\cdot)$ is nondecreasing.

Consider a special case of this problem where production cost functions are defined as

$$r_t(\alpha) = \begin{cases} g_t, & \alpha > 0, \\ 0, & \text{otherwise}, \end{cases} \tag{2.23}$$

and holding costs are zero.

Using the fact that $\phi_t(\cdot)$ is monotone and the number of different values this function may take is upper bounded by 2^t, we may deduce $|\mathcal{S}_t| \leq O(2^t)$ for any $t \in [1, n]$. Function $\widetilde{\phi}_t(\cdot)$ is also monotone and therefore $|\widetilde{\mathcal{S}}_t| \leq O(2^{n-t})$.

Using these estimates for sets $\mathcal{S}_t$ and $\widetilde{\mathcal{S}}_t$ and the previous discussion, we have the following theorem.

Theorem 2.3.3 *If additional restrictions $I_t \geq 0$ are imposed in the formulation of the CELS problem being considered, holding costs are zero, and production cost functions have the form (2.23), then any instance of such problem can be solved in $O(n2^{n/2})$ time.*

Note that the KNAPSACK PROBLEM can be polynomially reduced to the special case of the CELS problem mentioned in the above theorem.

The time complexity of our algorithm can be estimated more precisely. To do it, we just note that $|\mathcal{S}_t| \leq \sum_{i=1}^{n} d_i$. Therefore our algorithm runs in $O(\min\{nR^{n/2+1}H, n\sum_{i=1}^{n} d_i\})$ time.

2.4 A polynomial algorithm for a capacitated economic lot-sizing problem with piecewise concave cost functions

2.4.1 Introduction

In this section, we study a CELS problem with piecewise concave cost functions. All capacities as well as the points at which holding-backlogging cost functions may violate concavity are assumed to be nonnegative integer linear combinations of positive integers from some given sets. We prove that if the number of elements in these sets is upper bounded by a constant and holding-backlogging cost functions have polynomially many concave pieces, then there is a polynomial-time algorithm for the problem. (Besides these assumptions, coefficients in the mentioned integer linear combinations are considered to be polynomially bounded.)

Some piecewise concave functions are depicted in Figure 2.5. In that picture, the productions cost function $r_t(\cdot)$ consists of two concave pieces, and the holding-backlogging cost

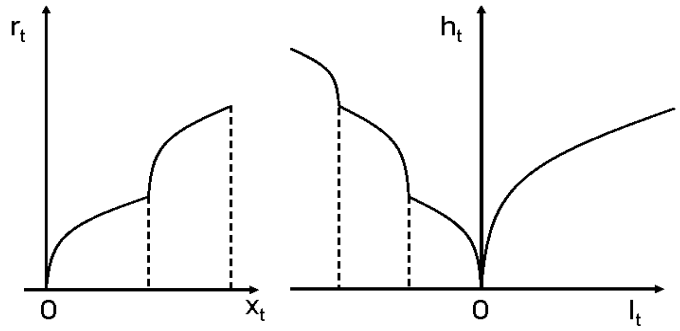

Figure 2.5: Piecewise concave cost functions.

function $h_t(\cdot)$ consists of four pieces, the holding cost function (the part of $h_t(\cdot)$ corresponding to nonnegative values of argument) consisting of only one, i.e., being completely concave.

Florian and Klein [25] developed an $O(n^4)$ algorithm to solve CELS problem with concave production and holding cost functions and with constant capacities (here the word "constant" means that in every period capacities are the same and equal to a given value being a part of the input data). In the case of linear holding cost functions, there exists an $O(n^3)$ algorithm. This algorithm has been designed by van Hoesel and Wagelmans [58].

Van Hoesel and Wagelmans as well as Florian and Klein use concavity of cost functions to obtain certain properties of optimal solutions allowing to decompose the whole problem into simpler subproblems. A similar issue is used by Atamtürk and Hochbaum [1] for some polynomially solvable extensions of the CELS problem with constant capacities. In our approach, we use the fact that a recursive function on which a simple dynamic programming procedure is based has properties repeating in some way those of cost functions.

In the subsequent sections, we will show that there is a polynomial algorithm for a variant of the CELS problem generalizing special cases with constant capacities and concave cost functions previously discussed in the literature.

The present section relies on paper [14] by Chubanov.

2.4.2 Problem formulation

Let $[b_1, b_2] := \{b_1, \ldots, b_2\}$ be an integer interval. A function $\psi : [b_1, b_2] \to \mathbb{R}$ is called *concave* on the interval $[b_1, b_2]$ of integers if and only if an inequality

$$\psi(a-1) + \psi(a+1) - 2\psi(a) \leq 0, \tag{2.24}$$

holds for all $a \in [b_1 + 1, b_2 - 1]$. We will say that $\psi(\cdot)$ is concave at a point $a \in [b_1 + 1, b_2 - 1]$ if (2.24) holds for function $\psi(\cdot)$ at this point. Function $\psi(\cdot)$ violates concavity (or is not concave) at end points of its feasible interval and at all points a of interval $[b_1 + 1, b_2 - 1]$ where condition (2.24) does not hold.

We will use the following easily verifiable properties:

- Let $\psi_1(\cdot)$ and $\psi_2(\cdot)$ both be concave functions defined on an interval $[b_1, b_2]$. A function $\psi(\cdot)$ defined on $[b_1, b_2]$ as a sum of functions $\psi_1(\cdot)$ and $\psi_2(\cdot)$ is concave on $[b_1, b_2]$.
- If a function $\psi(\cdot)$ defined on an interval $[b_1, b_2]$ is concave on this interval, then it attains its minimum at one of the points b_1 or b_2.

Consider the following special case of the single-item capacitated economic lot-sizing problem:

$$\begin{aligned}
\min \quad & F(x, I) = \textstyle\sum_{t=1}^{n} (r_t(x_t) + h_t(I_t)), & & \\
\text{subject to} \quad & I_t = I_{t-1} + x_t - d_t, & \forall t \in [1, n], & \quad (2.25) \\
& I_0 = 0, & & \quad (2.26) \\
& 0 \leq x_t \leq u_t, & \forall t \in [1, n], & \quad (2.27) \\
& x_t \in \mathbb{Z}, & \forall t \in [1, n]. & \quad (2.28)
\end{aligned}$$

Here, given integer coefficients d_t, $t \in [1, n]$, are interpreted as demands in corresponding periods. Notice that if $d_t < 0$, then $|d_t|$ may be interpreted as a supply in the corresponding period t. A production capacity u_t at each period t is assumed to be a nonnegative integer linear combination of points from a given set $\mathcal{C} = \{c_1, \ldots, c_{|\mathcal{C}|}\}$ of nonnegative integers with

coefficients upper bounded by a positive integer value W. I.e., for every period $t \in [1, n]$ we have

$$u_t = \sum_{i=1}^{|\mathcal{C}|} \alpha_i c_i$$

where α_i, $i \in [1, |\mathcal{C}|]$, are coefficients from $[0, W]$.

We make the following assumptions on the cost functions:

- For any $t \in [1, n]$ the production cost function $r_t(\cdot)$ may violate the concavity condition (2.24) only at points having the form $\sum_{i=1}^{|\mathcal{C}|} \alpha_i c_i$ where $\alpha_i \in [0, W]$ for all $i \in [1, |\mathcal{C}|]$.

- For any $t \in [1, n]$ the holding-backlogging cost function $h_t(\cdot)$ may violate the concavity condition (2.24) only at points having the form $\sum_{i=1}^{|\mathcal{E}|} \beta_i e_i$ where $\beta_i \in [-V, V], i \in [1, |\mathcal{E}|]$, for some given positive integer V, and $\mathcal{E} = \{e_1, \ldots, e_{|\mathcal{E}|}\}$ is a given set of nonnegative integers.

- Any cost function is computable in constant time at any point in its feasible domain.

The third assumption means that we consider a function computation as a basic operation. Of course, our discussion remains true for the case of polynomially computable cost functions.

We will call the described problem Capacitated Lot-Sizing.

A solution for an instance of Capacitated Lot-Sizing is a pair (x, I), where $x = (x_1, \ldots, x_n)$ and $I = (I_1, \ldots, I_n)$. Notice that $I_t = \sum_{i=1}^{t}(x_i - d_i)$, $t = 1, \ldots, n$. We call a solution (x, I) feasible if it satisfies (2.25)-(2.28). The objective is to find a feasible solution (x, I) with the minimum value of the objective function $F(\cdot, \cdot)$ which is the sum of the *production cost* functions $r_t(\cdot)$ and the *holding-backlogging* cost functions $h_t(\cdot)$. On positive values of the argument, function $h_t(\cdot)$ is referred to as a *holding cost function*; and on negative values of its argument this function represents a *backlogging cost function*.

Notice that we do not impose the restriction $I_n = 0$ upon feasible solutions. However, our algorithm can easily be modified to deal with this additional restriction. Another way of including the equation $I_n = 0$ into the model is imposing sufficiently high costs on nonzero inventory levels at the last time period. For instance, we may define function $h_n(\cdot)$ so that it is equal to 0 at a zero value of the argument, and equals a sufficiently large value, otherwise.

Then, after deleting equation $I_n = 0$ from the problem formulation, the set of optimal solutions remains the same. Note that $h_n(\cdot)$ defined in this way is not concave only at a zero value of the argument. The case without backlogging can be analogously modelled with sufficiently high backlogging costs.

A simple dynamic programming algorithm is able to solve the problem in pseudopolynomial time. As it will be seen from the discussion in the next section, if $|\mathcal{C}|$ and $|\mathcal{E}|$ are upper bounded by a constant and values V and W are polynomially bounded, then a straightforward pseudopolynomial dynamic programming algorithm can be converted into a polynomial one since a polynomial number of states is sufficient to identify an optimal solution in this case.

Prior to state the basic results regarding the computation of optima, let us discuss the cost structure and other problem features in more details.

Piecewise concave cost functions frequently arise in production and inventory models. Such functions have been studied in details by Zangwill [64]. In his earlier paper Zangwill [63] considers a lot-sizing model where cost functions are piecewise concave.

Assume that production in each period stands for product ordering (this is one of common interpretations). Then values taken by variables x_t, $t \in [1, n]$, may mean order volumes. Suppose that there are $|\mathcal{C}|$ suppliers and an element c_i of the set $\mathcal{C}$ indicates for every supplier i a batch volume within which discounts are possible. (The sizes of batches may differ at different suppliers.) More precisely, the more we acquire within a batch, the less we pay for a product unit. Therefore the acquisition cost function, denote it by $f_t^i(\cdot)$ for the supplier i and period t, will be concave in the interval $[0, c_i]$. This property of the acquisition cost function $f_t^i(\cdot)$ applies to every interval $[\alpha c_i, (\alpha + 1)c_i]$ where α is a nonnegative integer. So, the function $f_t^i(\cdot)$ may violate concavity at multiples of the batch volume c_i. The described situation is similar to one considered by Li, Hsu, and Xiao [41] for a lot-sizing model without production capacities.

Further we assume that we may order at most W batches at every supplier. Of course, whenever an order volume is determined at some period $t \in [1, n]$, we can decide to distribute the order among suppliers so as to minimize the order cost. In other words, we would like

to calculate

$$g_t(x_t) := \min \left\{ \sum_{i=1}^{|\mathcal{C}|} f_t^i(a_i) \;\; | \;\; \sum_{i=1}^{|\mathcal{C}|} a_i = x_t,\; a_i \in [0, Wc_i] \forall i \in [1, |\mathcal{C}|] \right\} \tag{2.29}$$

for the order volume x_t. The following observation shows that function $g_t(\cdot)$ has exactly the structure suggested by our assumptions on the production cost function $r_t(\cdot)$.

Observation 2.4.1 *The function $g_t(\cdot)$ may violate the concavity condition (2.24) only at points x_t having the form*

$$x_t = \sum_{i=1}^{|\mathcal{C}|} \alpha_i c_i$$

where $\alpha_i \in [0, W]$ for all $i \in [1, |\mathcal{C}|]$.

Proof. Let $a^0 = (a_1^0, \ldots, a_{|\mathcal{C}|}^0)$ be a point where the minimum in the formula (2.29) is attained. Then

$$g_t(x_t) = \sum_{i=1}^{|\mathcal{C}|} f_t^i(a_i^0).$$

Suppose that $x_t = \sum_{i=1}^{|\mathcal{C}|} a_i^0$ does not have the form mentioned in the formulation of the observation. Then for some $k \in [1, |\mathcal{C}|]$ the component a_k^0 is not in the set $\{0, c_k, \ldots, Wc_k\}$. Thus $f_t^k(\cdot)$ is concave at a_k^0. Notice that

$$g_t(x_t + 1) \leq \sum_{i=1}^{|\mathcal{C}|} f_t^i(a_i^0) + f_t^k(a_k^0 + 1) - f_t^k(a_k^0)$$

and

$$g_t(x_t - 1) \leq \sum_{i=1}^{|\mathcal{C}|} f_t^i(a_i^0) + f_t^k(a_k^0 - 1) - f_t^k(a_k^0)$$

due to the fact that the points

$$(a_1^0, \ldots, a_{k-1}^0, a_k^0 + 1, a_{k+1}^0, \ldots, a_{|\mathcal{C}|}^0)$$

and

$$(a_1^0, \ldots, a_{k-1}^0, a_k^0 - 1, a_{k+1}^0, \ldots, a_{|\mathcal{C}|}^0)$$

are among the points a over which the minimum in formula (2.29) is taken when evaluating $g_t(x_t + 1)$ and $g_t(x_t - 1)$, respectively. Hence, the following inequality chain holds:

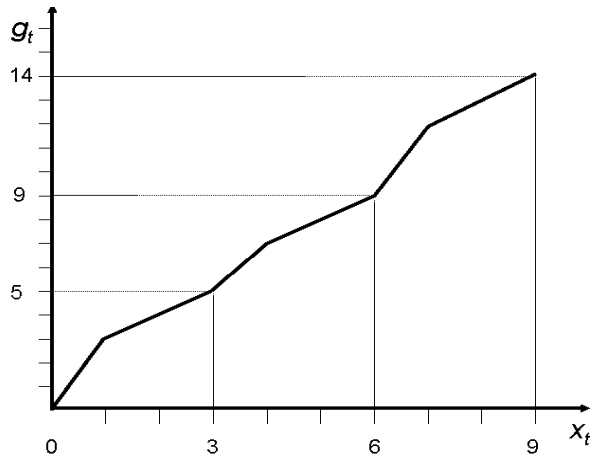

Figure 2.6: Example of a piecewise function g_t.

$$g_t(x_t-1)+g_t(x_t+1)-2g_t(x_t) \le$$
$$\le 2\sum_{i=1}^{|\mathcal{C}|} f_t^i(a_i^0) + f_t^k(a_k^0-1) - f_t^k(a_k^0) + f_t^k(a_k^0+1) - f_t^k(a_k^0) - 2\sum_{i=1}^{|\mathcal{C}|} f_t^i(a_i^0) =$$
$$= f_t^k(a_k^0-1) + f_t^k(a_k^0+1) - 2f_t^k(a_k^0) \le 0.$$

Thus $g_t(\cdot)$ is concave at x_t. The proof is complete. ∎

Consider an example of such function g_t. Assume that there are two suppliers and $c_1 = 3$ and $c_2 = 6$ are corresponding batch sizes. Suppose that we may order at most one batch of each type, i.e., $W = 1$. This means that altogether we may acquire $c_1 + c_2 = 9$ product units. Assume that a cost of a product unit is equal to one, and transportation cost is equal to 2 for the first supplier and to 3 for the second one. Therefore $f_t^1(a) = 2 + a$ if $a > 0$ and $f_t^1(a) = 0$, otherwise. For the second supplier, $f_t^2(a) = 3 + a$ if $a > 0$, and $f_t^2(a) = 0$, otherwise. Then a corresponding function g_t has a form shown in Figure 2.6. It can be seen from the picture that function g_t is not concave at points $0c_1 + 0c_2 = 0$, $c_1 + 0c_2 = 3$, $0c_1 + c_2 = 6$, and $c_1 + c_2 = 9$. (Points 0 and 9 are end points of the interval on which the functions is defined.) In the picture, neighboring points in the function graph are connected with line segments.

So, we have an interpretation of the structure of function $r_t(\cdot)$. Functions like $r_t(\cdot)$ can also be useful to approximate real costs. The piecewise concave structure of holding-backlogging

costs can be interpreted using similar arguments.

Note that by a simple enumeration procedure solving a concave problem for all possible combinations of concave pieces of functions $f_t^i(\cdot)$, $i \in [1, |\mathcal{C}|]$, a value $g_t(x_t)$ may be computed in polynomial time if these functions are in turn polynomially computable, the value W is polynomially bounded, and $|\mathcal{C}|$ is upper bounded by a constant.

In the classical CELS problem, the inventory level I_t in period t is determined by the inventory level in the previous period, the production level x_t, and the demand d_t through the equation

$$I_t = I_{t-1} + x_t - d_t$$

for all t ranging from 1 to n. (I_0 is equal to zero.) If I_t is allowed to be negative, then we deal with so-called backlogging. Atamtürk and Hochbaum [1] define inventory levels more generally through the equation

$$I_t = I_{t-1} + x_t + y_t - d_t$$

with an additional nonnegative variable y_t. This variable expresses an additional capacity that is "acquired". Atamtürk and Hochbaum consider a concave cost function related to the variable y_t. In one of the problems in [1], called constant capacity lot-sizing and subcontracting problem, only the presence of an additional variable y_t makes the problem different from one considered by van Hoesel and Wagelmans. Atamtürk and Hochbaum develop an $O(n^5)$ algorithm for the constant capacity lot-sizing and subcontracting problem.

If we insert a new time period before each period t, set the demand and holding-backlogging costs to zero in this new period, and assume y_l to express production levels in the new periods l, then we come to an equivalent standard lot-sizing problem where production levels are upper bounded by some capacity c at even periods and unbounded at odd periods (corresponding to the new periods inserted). Actually, since $I_n = 0$ in the problem of Atamtürk and Hochbaum, we can restrict y_l by the sum of all demands $\sum_{i=1}^n d_i$. This means that capacities are taken from the set $\{c, \sum_{i=1}^n d_i\}$. Thus we have a special case of Capacitated Lot-Sizing. The algorithm introduced in this paper is able to solve this problem, and thus the problem of Atamtürk and Hochbaum, in polynomial time. One can extend the

problem with two kinds of capacities to the capacities that take their values from any given set of nonnegative integers. This is again a special case of Capacitated Lot-Sizing. If we assume that the capacities take values from a set with cardinality bounded by a constant, then, due to the problem properties which we formulate later, there is a polynomial time algorithm for the corresponding CELS problem with concave production, backlogging, and holding cost functions (concave backlogging and holding cost functions mean that it may only happen that functions $h_t(\cdot)$ violate concavity at the point 0).

2.4.3 Algorithm

In this section, we first discuss properties of a recursive function on which a straightforward algorithm is based. Our analysis includes estimating the number of intervals where the recursive function is concave and determining where to find points at which concavity properties are violated.

Consider an instance $\mathcal{I}$ of problem Capacitated Lot-Sizing. Let $OPT(\mathcal{I})$ denote the optimal value of the instance $\mathcal{I}$. For any time period t and a nonnegative integer value s we construct a reduced instance $\mathcal{I}_{t,s}$ as follows.

$$\begin{aligned}
\min \quad & F^t(x, I) = \textstyle\sum_{i=1}^{t}(r_i(x_i) + h_i(I_i)), & & \\
\text{subject to} \quad & I_i = I_{i-1} + x_i - d_i, & \forall i \in [1, t], & (2.30)\\
& I_0 = 0, & & (2.31)\\
& I_t = s, & & (2.32)\\
& 0 \le x_i \le u_i, & \forall i \in [1, t], & (2.33)\\
& x \in \mathbb{Z}^t. & & (2.34)
\end{aligned}$$

In this mathematical program, only the first t out of n time periods are considered and an additional restriction $I_t = s$ is imposed on the inventory level at the end of period t. Feasible solutions of $\mathcal{I}_{t,s}$ are partial solutions for the original instance $\mathcal{I}$.

For any t and s at which the set of feasible solutions of $\mathcal{I}_{t,s}$ is nonempty, define $\phi_t(\cdot)$ as

$$\phi_t(s) := OPT(\mathcal{I}_{t,s}).$$

Obviously, the function $\phi_t(\cdot)$ is defined on the set $[a_1^t, a_2^t]$ where

$$a_1^t = -\sum_{i=1}^{t} d_i$$

and

$$a_2^t = \sum_{i=1}^{t} (u_i - d_i).$$

We set $\phi_0(0) := 0$ and $a_1^0 = a_2^0 = 0$. For any $t \in [1, n]$ function $\phi_t(\cdot)$ is evaluated recursively by the formula

$$\phi_t(s) = \min_{s' \in [a_1^{t-1}, a_2^{t-1}]} \{r_t(s - s' + d_t) + h_t(s) + \phi_{t-1}(s') | s - s' + d_t \in [0, u_t]\} \tag{2.35}$$

at any point s from $[a_1^t, a_2^t]$. This formula is similar to the one presented by Florian et al. [26] for a variant of problem CELS with no backlogging. The formula follows immediately from the problem formulation. Furthermore, it immediately yields a simple dynamic programming algorithm with running time $O(\sum_{t=1}^{n} u_t \sum_{i=1}^{t} u_i)$ for problem Capacitated Lot-Sizing. In the sequel, we refer to this algorithm as a *straightforward algorithm*.

The following theorem implies that if the cost functions have a polynomial number of concave pieces and both values $|\mathcal{C}|$ and $|\mathcal{E}|$ are upper bounded by a constant, then functions $\phi_t(\cdot)$, $t \in [1, n]$, also have a polynomial number of concave pieces.

Theorem 2.4.1 *For any $t \in [1, n]$, every point at which the function $\phi_t(\cdot)$ is not concave has the form*

$$\sum_{i=1}^{|\mathcal{C}|} \alpha_i c_i + \sum_{i=1}^{|\mathcal{E}|} \beta_i e_i - \sum_{i=t'}^{t} d_i$$

where $t' \in [1, t+1]$, $\sum_{i=t+1}^{t} d_i = 0$, $\alpha_i \in [0, tW]$ for all $i \in [1, |\mathcal{C}|]$, and $\beta_i \in [-V, V]$ for all $i \in [1, |\mathcal{E}|]$. There are at most

$$(tW + 1)^{|\mathcal{C}|} \cdot (2V + 1)^{|\mathcal{E}|} \cdot (t + 1)$$

such points.

Proof. Denote by $\mathcal{S}$ the set of points mentioned in the formulation of the theorem. Let $\mathcal{S} = \{s_1, \ldots, s_p\}$ where $s_1 < \ldots < s_p$.

Let s belong to $[s_{q-1}+1, s_q-1]$ for some $q \in [2,p]$. We further prove that $\phi_t(\cdot)$ is concave at s.

Consider an optimal solution (x, I) of instance $\mathcal{I}_{t,s}$. Select the maximum index $k \in [1,t]$ such that x_k is not a nonnegative integer linear combination of points from set $\mathcal{C}$ with coefficients upper bounded by W, i.e., $x_k \neq \sum_{i=1}^{|\mathcal{C}|} \alpha_i c_i$ for any $\alpha_i \in [0,W]$, $i \in [1,|\mathcal{C}|]$. Such an index exists since otherwise the point

$$s = I_t = \sum_{i=1}^{t} (x_i - d_i)$$

belongs to $\mathcal{S}$. For all $j \in [k,t]$ the equation

$$I_j = I_k + \sum_{i=k+1}^{j} (x_i - d_i)$$

must hold. Since $I_t = s$, we have

$$I_k = s - \sum_{i=k+1}^{t} (x_i - d_i)$$

and, consequently,

$$I_j = s - \sum_{i=j+1}^{t} (x_i - d_i)$$

for all $j \in [k,t]$. (We consider that $\sum_{i=t+1}^{t}(x_i - d_i) = 0$.)

If I_j is an integer linear combination of points from the set $\mathcal{E}$ with coefficients from $[-V,V]$, then, since $\sum_{i=k+1}^{j} x_i$ is an integer linear combination of points $c_1, \ldots, c_{|\mathcal{C}|}$ due to the choice of k, s has the form

$$s = \sum_{i=1}^{|\mathcal{C}|} \alpha_i c_i + \sum_{i=1}^{|\mathcal{E}|} \beta_i e_i - \sum_{i=j+1}^{t} d_i$$

where $\alpha_i \in [0, tW]$ for all $i \in [1,|\mathcal{C}|]$, and $\beta_i \in [-V,V]$ for all $i \in [1,|\mathcal{E}|]$. This contradicts our assumption that s does not belong to $\mathcal{S}$. Thus, by the assumptions on the cost structure, $h_j(\cdot)$ is concave at the point I_j for any $j \in [k,t]$.

Generate a feasible solution (x^1, I^1) for the instance $\mathcal{I}_{t,s-1}$ from solution (x, I) by means of decreasing x_k by 1 (to preserve feasibility, we also make appropriate changes of values

I_j, $j \in [k,t]$). This transformation is legitimate because $x_k > 0$ due to the fact that x_k is not a nonnegative integer linear combination of points from the set $\mathcal{C}$ with coefficients upper bounded by W. The objective value of (x^1, I^1) is

$$F^t(x^1, I^1) = F^t(x, I) + r_k(x_k - 1) - r_k(x_k) + \sum_{i=k}^{t}(h_i(I_i - 1) - h_i(I_i)). \tag{2.36}$$

Now increase x_k and inventory levels I_i, $i \in [k,t]$ by 1. As x_k is not a nonnegative integer linear combination of points from the set $\mathcal{C}$ with coefficients upper bounded by W, we have $x_k < u_k$ and thus the obtained solution, denoted by (x^2, I^2), is feasible for $\mathcal{I}_{t,s+1}$. Its objective value is evaluated as

$$F^t(x^2, I^2) = F^t(x, I) + r_k(x_k + 1) - r_k(x_k) + \sum_{i=k}^{t}(h_i(I_i + 1) - h_i(I_i)). \tag{2.37}$$

Let us check if the concavity property (2.24) holds for $\phi_t(\cdot)$ at point s. As it has been already seen, $h_j(\cdot)$ is concave at the point I_j for all $j \in [k,t]$. Function $r_k(\cdot)$ is concave at x_k by construction. Therefore equations (2.36) and (2.37) and property (2.24) yield

$$\phi_t(s-1)+\phi_t(s+1) - 2\phi_t(s) \le F^t(x^1, I^1) + F^t(x^2, I^2) - 2F^t(x, I) = \\ r_k(x_k - 1) + r_k(x_k + 1) - 2r_k(x_k) + \textstyle\sum_{i=k}^{t}(h_i(I_i - 1) + h_i(I_i + 1) - 2h_i(I_i)) \le 0.$$

Thus $\phi_t(\cdot)$ is concave at the point s. This means that concavity of $\phi_t(\cdot)$ may be violated only at a point from $\mathcal{S}$. To complete the proof, we notice that the cardinality of the set $\mathcal{S}$ is bounded by $(tW + 1)^{|\mathcal{C}|} \cdot (2V + 1)^{|\mathcal{E}|} \cdot (t + 1)$. ∎

The complete information about every function $\phi_t(\cdot)$ is not necessary to find an optimal solution. To see this, we prove the next corollary which shows where the minimum of (2.35) is to find.

Corollary 2.4.1 *A minimum in formula (2.35) is attained at a point $s' = \hat{s}$ either having the form*

$$\hat{s} = s - \sum_{i=1}^{|\mathcal{C}|} \alpha_i c_i + d_t$$

where $\alpha_i \in [0, W]$ *for all* $i \in [1, |\mathcal{C}|]$ *or the form*

$$\hat{s} = \sum_{i=1}^{|\mathcal{C}|} \alpha_i c_i + \sum_{i=1}^{|\mathcal{E}|} \beta_i e_i - \sum_{i=t'}^{t-1} d_i$$

where $t' \in [1, t]$, $\sum_{i=t}^{t-1} d_i = 0$, $\alpha_i \in [0, tW]$ *for all* $i \in [1, |\mathcal{C}|]$, *and* $\beta_i \in [-V, V]$ *for all* $i \in [1, |\mathcal{E}|]$.

Proof. Assume that the minimum in formula (2.35) is attained at a point $s' = \hat{s}$ which is not among the points mentioned in the formulation of the corollary. (Note that the point s' is not then an end point of the feasible interval $[a_1^{t-1}, a_2^{t-1}]$ of function $\phi_{t-1}(\cdot)$.) Therefore function $r_t(\cdot)$ is concave at $s - \hat{s} + d_t$ according to the assumptions on the cost structure and $\phi_{t-1}(\cdot)$ is concave at $\hat{s}$ by Theorem 2.4.1. Thus, either increasing or decreasing $\hat{s}$ by one, we may obtain a point which is not worse than $\hat{s}$. In other words, either for $s'' = \hat{s} + 1$ or $s'' = \hat{s} - 1$, we have

$$r_t(s - s'' + d_t) + h_t(s) + \phi_{t-1}(s'') \leq r_t(s - \hat{s} + d_t) + h_t(s) + \phi_{t-1}(\hat{s})$$

due to concavity properties. The proof is complete.

■

Notice that for every $t \in [1, n]$ there are $O((tW + 1)^{|\mathcal{C}|} \cdot (2V + 1)^{|\mathcal{E}|} \cdot t)$ points of the form mentioned in the corollary.

Let for any $j \in [1, n]$ a set $\mathcal{S}_j$ be defined as the set of all points

$$\sum_{i=1}^{|\mathcal{C}|} \alpha_i c_i + \sum_{i=1}^{|\mathcal{E}|} \beta_i e_i - \sum_{i=j'}^{j} d_i$$

where $j' \in [1, j + 1]$, $\sum_{i=j+1}^{j} d_i = 0$, $\alpha_i \in [(j - n)W, nW]$ for all $i \in [1, |\mathcal{C}|]$, and $\beta_i \in [-V, V]$ for all $i \in [1, |\mathcal{E}|]$.

We assume $\mathcal{S}_0 = \{0\}$.

The following theorem implies that to find

$$OPT(\mathcal{I}) = \min\{\phi_n(s) | s \in [a_1^n, a_2^n]\}$$

it is sufficient to know values of each function $\phi_t(\cdot)$ at feasible points from $\mathcal{S}_t$.

Theorem 2.4.2 *If s is feasible for $\phi_t(\cdot)$ and $s \in \mathcal{S}_t$, then a minimum in (2.35) is attained at a point $s' \in \mathcal{S}_{t-1}$.*

Proof. The theorem is an implication of Corollary 2.4.1. Let $s \in \mathcal{S}_t$. Then

$$s = \sum_{i=1}^{|\mathcal{C}|} \alpha'_i c_i + \sum_{i=1}^{|\mathcal{E}|} \beta'_i e_i - \sum_{i=t'}^{t} d_i$$

where $t' \in [1, t+1]$, $\sum_{i=t+1}^{t} d_i = 0$, $\alpha'_i \in [(t-n)W, nW]$ for all $i \in [1, |\mathcal{C}|]$, and $\beta'_i \in [-V, V]$ for all $i \in [1, |\mathcal{E}|]$. If the minimum in formula (2.35) is attained at a point $s' = \hat{s}$ having the form

$$\hat{s} = s - \sum_{i=1}^{|\mathcal{C}|} \alpha_i c_i + d_t$$

where $\alpha_i \in [0, W]$ for all $i \in [1, |\mathcal{C}|]$, then

$$\hat{s} = \sum_{i=1}^{|\mathcal{C}|} (\alpha'_i - \alpha_i) c_i + \sum_{i=1}^{|\mathcal{E}|} \beta'_i e_i - \sum_{i=t'}^{t-1} d_i.$$

Let us note that $(t-1-n)W \leq \alpha'_i - \alpha_i \leq nW$ for all $i \in [1, |\mathcal{C}|]$ and thus $\hat{s} \in \mathcal{S}_{t-1}$. The second alternative in Corollary 2.4.1 suggests to take $\hat{s}$ directly from $\mathcal{S}_{t-1}$. The proof is complete. ∎

Using this theorem, we may rewrite formula (2.35) as

$$\phi_t(s) = \min_{s' \in [a_1^{t-1}, a_2^{t-1}] \cap \mathcal{S}_{t-1}} \{r_t(s - s' + d_t) + h_t(s) + \phi_{t-1}(s') | s - s' + d_t \in [0, u_t]\} \qquad (2.38)$$

at any point $s \in \mathcal{S}_t$.

To find $\min\{\phi_n(s) | s \in [a_1^n, a_2^n]\}$ (equal to $OPT(\mathcal{I})$), it is enough to consider only those points at which $\phi_n(\cdot)$ is not concave. By Theorem 2.4.1, these points are found in $\mathcal{S}_n$. Together with formula (2.38), this observation leads to a simple dynamic programming (DP) algorithm to solve the problem.

Algorithm DP

Input: An instance $\mathcal{I}$ of Capacitated Lot-Sizing;

Output: An optimal solution (x^0, I^0) of $\mathcal{I}$.

Step 0 Set $\mathcal{S}_0 := \{0\}$.

Step 1 for $t = 1$ **to** n **do**

Using (2.38), evaluate $\phi_t(\cdot)$ at each point from $\mathcal{S}_t$;

For every $s \in \mathcal{S}_t$ store s' at which a minimum is attained;

end do

Step 2 Find $s = \arg\min\{\phi_n(s) | s \in [a_1^n, a_2^n] \cap \mathcal{S}_n\}$;

$I_n^0 := s$;

for $t = n-1$ **down to** 1 **do**

Take s' where a minimum in formula (2.38) is attained;

$I_t^0 := s'$;

$s := s'$;

end do

Set $I_0^0 := 0$;

for $t = 1$ **to** n **do**

$x_t^0 := I_t^0 - I_{t-1}^0 + d_t$;

end do

In Step 1, the algorithm evaluates functions $\phi_t(\cdot)$, $t \in [1, n]$ at points which may arise during the computation of an optimal solution. Step 2 is a backtracking step in which an optimal solution (x^0, I^0) is constructed. Due to the fact that $|\mathcal{S}_t|$ is bounded by $O((2nW)^{|\mathcal{C}|}(2V+1)^{|\mathcal{E}|}n)$ for all $t \in [1, n]$, the algorithm runs in $O((2nW)^{2|\mathcal{C}|}(2V+1)^{2|\mathcal{E}|}n^3)$ time. Hence, we have proven the following theorem.

Theorem 2.4.3 *An optimal solution to any instance of* Capacitated Lot-Sizing *can be found in* $O((2nW)^{2|\mathcal{C}|}(2V+1)^{2|\mathcal{E}|}n^3)$ *time.*

This theorem immediately implies the main result of the section:

Corollary 2.4.2 *If in some instance* $\mathcal{I}$ *of* Capacitated Lot-Sizing *values* V *and* W *are bounded by polynomials of the instance size and* $|\mathcal{C}|$ *and* $|\mathcal{E}|$ *are upper bounded by a constant, then* $\mathcal{I}$ *can be solved in polynomial time.*

2.4.4 Conclusions

We have considered a special case of the well known capacitated economic lot-sizing problem. The polynomial solvability of this special case has been proven under the assumption that the cost functions consist of a polynomial number of concave pieces and some additional restrictions generalizing the cases with constant capacities and concave costs previously considered in the literature. The further research may be done in the direction of complexity improvement and deriving additional properties which would be useful for better understanding of the simple recursive formulation studied in this section.

2.5 An FPTAS for a single-item capacitated economic lot-sizing problem with a monotone cost structure

2.5.1 Introduction

We consider again the capacitated economic lot-sizing problem (which we denoted as Capacitated Lot-Sizing) as in the previous section (but without piecewise concavity assumptions and without restrictions on the capacity structure):

$$\begin{aligned}
\min \quad & F(x,I) = \textstyle\sum_{t=1}^{n}(R_t(x_t) + H_t(I_t)), & & \\
\text{subject to} \quad & I_t = I_{t-1} + x_t - d_t, & \forall t \in [1,n], & \quad (2.39)\\
& I_0 = 0, & & \quad (2.40)\\
& 0 \le x_t \le u_t, & \forall t \in [1,n], & \quad (2.41)\\
& x \in \mathbb{Z}^n. & & \quad (2.42)
\end{aligned}$$

Remind that this problem describes a production of a single product over n time periods. Variables I_t and x_t represent an inventory level and a production level in time period t, respectively. The production level in each period t is bounded from above by the given production capacity u_t. Parameters d_t describe given demands for the product in each time

period t. All values u_t and d_t are assumed to be non-negative integers.

In the above formulation, $R_t(\cdot)$ and $H_t(\cdot)$ denote production cost functions and holding-backlogging cost functions, respectively. Remind that on positive values of the argument, function $H_t(\cdot)$ is referred to as a *holding cost function*; and on negative values of its argument this function represents a *backlogging cost function*.

We make the following assumptions about the cost structure.

(i) The holding-backlogging cost functions $H_t(\cdot)$ are nonincreasing on the set $\mathbb{Z}_-$ of non-positive integers, nondecreasing on the set $\mathbb{Z}_+$ of nonnegative integers, and satisfy the condition $H_t(0) = 0$ for all $t \in [1, n]$.

(ii) The production cost functions $R_t(\cdot)$ are nondecreasing on $\mathbb{Z}_+$ and satisfy the condition $R_t(0) = 0$ for all $t \in [1, n]$.

(iii) Each cost function is computable in constant time at every point in its feasible domain.

To the best of our knowledge, assumptions (i)-(ii) cover all cost structures for problem Capacitated Lot-Sizing encountered in the literature. There is no practical reason to consider other cost structures since production costs should grow as production levels grow, higher stock levels imply higher expenses, and in the case of shortage of the product (negative inventory levels) penalty costs at least do not decline as the product shortage grows.

Problem Capacitated Lot-Sizing is NP-hard for many special cases, see Bitran and Yanasse [7]. We present a fully polynomial time approximation scheme (FPTAS) for this problem under assumptions (i)-(iii).

The following approximation results are available for variants of problem Capacitated Lot-Sizing. Fast heuristic algorithms were developed by Bitran et al. [8] and Axäter [2] for some polynomially solvable cases. The authors assert that these heuristics are useful for large-scale instances. For NP-hard special cases, Bitran and Matsuo [6] suggested approximation formulations solvable in pseudopolynomial time. They provided worst-case performance estimations for some special cases.

It is worth mentioning that Woeginger [61] proved that if a dynamic programming algorithm for some optimization problem has a certain structure, then it can be converted into an

FPTAS. The method of Woeginger is applicable to a large variety of optimization problems, especially to those which come from scheduling, but he explicitly mentions the single-item capacitated economic lot-sizing problem as an example of an optimization problem to which this method does not apply.

Van Hoesel and Wagelmans [57] developed the first FPTAS for the CELS problem. They considered the case of concave (convex) production and backlogging cost functions satisfying the assumptions (i) and (ii). Their method can be extended to the case where either production cost functions or holding cost functions are piece-wise concave (convex) with a polynomially bounded number of pieces. In this case, either backlogging cost functions or production cost functions remain concave (convex). They also considered a restriction $I_n \geq 0$. In our case, this restriction can be modelled by means of sufficiently large backlogging cost $H_n(I_n)$ (for negative values of inventory level I_n).

Van Hoesel and Wagelmans assumed that the cost functions are polynomially computable. Our FPTAS is also applicable in this case. We assume that the cost functions are computable in a constant time just in order to shorten complexity estimation formulas.

Our primary aim in this section is to develop an FPTAS for problem Capacitated Lot-Sizing under the assumptions (i)-(iii) (without imposing any additional conditions on the cost structure). Our secondary aim is to show that a standard dynamic programming approach yields an FPTAS for this problem. Van Hoesel and Wagelmans stated that such an approach is unlikely to be useful in deriving an FPTAS for problem Capacitated Lot-Sizing. Our results disprove this statement.

The rest of this section is organized as follows. In the next section, we present a rounded formulation of problem Capacitated Lot-Sizing and a straightforward dynamic programming algorithm for this formulation. We further establish properties of the corresponding recursive function, which allow us to shrink the state space of the dynamic programming algorithm. The detailed algorithm is presented in Subsection 2.5.3. We convert it into an FPTAS for problem Capacitated Lot-Sizing. A bound improvement procedure needed for the conversion has been described in the introductory chapter. Concluding remarks are given in Subsection 2.5.4.

The present section is based on the paper of Chubanov, Kovalyov, and Pesch [15].

2.5.2 A rounded problem

Let $\mathcal{I}^{\#}$ be an instance of Capacitated Lot-Sizing. Note that assumptions (i) and (ii) imply $OPT(\mathcal{I}^{\#}) \geq 0$. In Subsection 2.5.3, we will show that the case $OPT(\mathcal{I}^{\#}) = 0$ can be identified in polynomial time. Furthermore, if $OPT(\mathcal{I}^{\#}) > 0$, a positive lower bound L on the optimal value $OPT(\mathcal{I}^{\#})$ can be found in polynomial time.

In this subsection, we assume that lower and upper bounds are known such that

$$0 < L \leq OPT(\mathcal{I}^{\#}) \leq U.$$

The value of any feasible solution can be taken as an upper bound U. It can be calculated in $O(n)$ time.

Let us scale the objective function of instance $\mathcal{I}^{\#}$ to obtain a *rounded instance* denoted by $\mathcal{I}$. Given $\varepsilon > 0$, define scaling parameter δ such that

$$\delta = \frac{\varepsilon L}{2n}.$$

The rounded instance $\mathcal{I}$ is as follows.

$$\min \quad f(x, I) = \sum_{t=1}^{n} (\lfloor R_t(x_t)/\delta \rfloor + \lfloor H_t(I_t)/\delta \rfloor), \quad \text{subject to (2.39)-(2.42).}$$

All the assumptions (i)-(iii) are satisfied for the new production costs $r_t(x_t) := \lfloor R_t(x_t)/\delta \rfloor$ and holding-backlogging costs $h_t(I_t) := \lfloor H_t(I_t)/\delta \rfloor$.

Theorem 2.5.1 *An optimal solution for the rounded problem instance $\mathcal{I}$ is an $\varepsilon-$approximate solution for the original instance $\mathcal{I}^{\#}$.*

Proof. Let (x^0, I^0) be an optimal solution of instance $\mathcal{I}$ and (x^*, I^*) be an optimal solution of the original instance $\mathcal{I}^{\#}$. Since both the instances have the same feasible domain, the following chain of inequalities holds.

$$F(x^0, I^0) = \sum_{t=1}^{n} (R_t(x_t^0) + H_t(I_t^0)) \leq \delta \sum_{t=1}^{n} (r_t(x_t^0) + h_t(I_t^0)) + 2n\delta \leq$$

$$((x^*, I^*) \text{ is feasible for the rounded instance})$$

$$\leq \delta \sum_{t=1}^{n} (r_t(x_t^*) + h_t(I_t^*)) + 2n\delta \leq F(x^*, I^*) + \varepsilon L \leq (1+\varepsilon) OPT(\mathcal{I}^{\#}).$$

■

Remark 2.5.1 *An upper bound V on the optimal value $OPT(\mathcal{I})$ of the rounded instance can be computed as $V = \lfloor U/\delta \rfloor$. If $V = 0$, then we can apply a technique of Section 2.5.3 to find an optimal solution of the rounded problem. Therefore, we assume $V > 0$. Furthermore, we assume that for any feasible solution (x, I) of problem* Capacitated Lot-Sizing *inequalities $r_t(x_t) \leq V + 1$ and $h_t(I_t) \leq V + 1$, $t = 1, \ldots, n$, hold. This assumption does not restrict the class of problems we consider: if $r_t(a) > V$ for some $a \in [0, u_t]$, then, without changing the set of optimal solutions, function $r_t(\cdot)$ can be re-defined as $r_t(a) := \min\{r_t(a), V + 1\}$ for all $a \in [0, u_t]$. Function $h_t(\cdot)$ can be re-defined in a similar way.*

We solve the rounded instance $\mathcal{I}$ by a modification of a straightforward dynamic programming algorithm. In the course of this algorithm, functions $\phi_t(\cdot)$, $t \in [1, n]$, are calculated. Value $\phi_t(s)$ is defined as the optimal objective function value of the following mathematical programming problem.

$$\begin{aligned}
\min \quad & f^t(x, I) = \textstyle\sum_{i=1}^{t} (r_i(x_i) + h_i(I_i)), & & \\
\text{subject to} \quad & I_i = I_{i-1} + x_i - d_i, & \forall i \in [1, t], & \quad (2.43) \\
& I_0 = 0, & & \quad (2.44) \\
& I_t = s, & & \quad (2.45) \\
& 0 \leq x_i \leq u_i, & \forall i \in [1, t], & \quad (2.46) \\
& x \in \mathbb{Z}^t. & & \quad (2.47)
\end{aligned}$$

In this program, only the first t out of n time periods are considered and an additional restriction $I_t = s$ is imposed on the inventory level in the time period t. Feasible solutions of this instance are partial solutions of the original rounded instance $\mathcal{I}$. We denote this instance by $\mathcal{I}_{t,s}$.

Function $\phi_t(\cdot)$ is defined on the set $[a_1^t, a_2^t]$ where $a_1^t = -\sum_{i=1}^t d_i$ and $a_2^t = \sum_{i=1}^t (u_i - d_i)$. We set $\phi_0(0) := 0$ and $a_1^0 = a_2^0 = 0$. For any $t \in [1, n]$, function $\phi_t(\cdot)$ is evaluated recursively as

$$\phi_t(s) = \min_{s' \in [a_1^{t-1}, a_2^{t-1}]} \{r_t(s - s' + d_t) + h_t(s) + \phi_{t-1}(s') | s - s' + d_t \in [0, u_t]\} \tag{2.48}$$

at any point s from $[a_1^t, a_2^t]$. This formula is the same as presented in the previous section. It follows immediately from the problem formulation. Remind that it yields a simple dynamic programming algorithm with running time $O(\sum_{t=1}^n u_t \sum_{i=1}^t u_i)$ for the rounded instance $\mathcal{I}$. As well as in the previous section, we refer to this algorithm as a *straightforward algorithm*.

Let $\psi(\cdot)$ be some function over an interval $[a, b]$ of integers. We say that s is a *nonstable point* of this function if $s \in \{a, b\}$ or $\psi(s) \neq \psi(s-1)$. If $\psi(s) = \psi(s-1)$ for some $s \in [a+1, b-1]$, then we call s a *stable point* of the function $\psi(\cdot)$. Denote a set of the nonstable points of the function $\psi(\cdot)$ as $\mathcal{B}_\psi$. We have

$$\mathcal{B}_\psi := \{s \in [a+1, b-1] \mid \psi(s) \neq \psi(s-1)\} \cup \{a, b\}.$$

Note that if values $\psi(s)$, $s \in \mathcal{B}_\psi$, are stored in a heap, each value $\psi(s)$, $s \in [a, b]$, can be calculated in $O(\log |\mathcal{B}_\psi|)$ time. The heap can be constructed in $O(|\mathcal{B}_\psi| \log |\mathcal{B}_\psi|)$ time.

Further we assume that for any $t \in [1, n]$ function $r_t(\cdot)$ is defined on interval $[0, u_t]$, and therefore $\mathcal{B}_{r_t} \subseteq [0, u_t]$ and the end points 0 and u_t are contained in $\mathcal{B}_{r_t}$. We also assume that $h_t(\cdot)$ is defined on interval $[a_1^t, a_2^t]$ and therefore end points a_1^t and a_2^t belong to $\mathcal{B}_{h_t}$.

The straightforward algorithm makes some superfluous computations to solve instance $\mathcal{I}_{t,s}$. The following theorem is a step towards its complexity improvement.

Theorem 2.5.2 *The function $\phi_t(\cdot)$ has at most $O(t^2V^2)$ nonstable points in its feasible domain $[a_1^t, a_2^t]$.*

Proof. Define set $\mathcal{B}_{h_0} := \{0\}$. Recall that for any time period k, $\mathcal{B}_{h_k}$ is the set of nonstable points of the holding-backlogging cost function $h_k(\cdot)$. Construct a set

$$S = \{b - \sum_{l=k+1}^t d_l \mid b \in \mathcal{B}_{h_k}, k \in [0, t]\}, \text{ where } \sum_{l=t+1}^t d_l = 0.$$

Let $S = \{s_1, \ldots, s_p\}$ where $s_1 < \ldots < s_p$. We start with showing that function $\phi_t(\cdot)$ is nondecreasing over each interval $[s_{j-1}, s_j - 1]$, $j \in [2, p]$.

Suppose that $s \notin S$. Consider an optimal solution (x, I) for instance $\mathcal{I}_{t,s}$. By definition, $I_t = s$ and $f^t(x, I) = \phi_t(s)$. If $x = (0, \ldots, 0)$, then $I_t = -\sum_{l=1}^{t} d_l \in S$. Hence, $x \neq (0, \ldots, 0)$. Take the maximum index $k \in [1, t]$ such that $x_k > 0$. For all $j \in [k, t]$, the inventory level I_j can be calculated using the value I_k :

$$I_j = I_{j-1} - d_j = I_{j-2} - d_{j-1} - d_j = \ldots = I_k - \sum_{l=k+1}^{j} d_l.$$

From $s = I_t = I_k - \sum_{l=k+1}^{t} d_l$, we obtain $I_k = s + \sum_{l=k+1}^{t} d_l$. For all $j \in [k, t]$, we get $I_j = I_k - \sum_{l=k+1}^{j} d_l = s + \sum_{l=j+1}^{t} d_l$. Thus, since $s \notin S$, I_j does not belong to $\mathcal{B}_{h_j}$. Then, by construction,

$$h_j(I_j - 1) = h_j(I_j) \tag{2.49}$$

for all $j \in [k, t]$. Decrease x_k by 1 to obtain a feasible solution $(\hat{x}, \hat{I})$ of instance $\mathcal{I}_{t,s-1}$ from the optimal solution (x, I) of instance $\mathcal{I}_{t,s}$. It holds $\hat{I}_l = I_l$ for all $l \in [1, k-1]$ and $\hat{I}_l = I_l - 1$ for all $l \in [k, t]$. Taking into account (2.49), we deduce

$$h_l(\hat{I}_l) = h_l(I_l), \ \forall l \in [1, t].$$

Furthermore, $r_l(x_l) = r_l(\hat{x}_l)$ for all $l \in [1, t] \setminus \{k\}$ and $r_k(\hat{x}_k) \leq r_k(x_k)$. Hence,

$$f^t(\hat{x}, \hat{I}) \leq f^t(x, I).$$

Denote by $(\bar{x}, \bar{I})$ an optimal solution of instance $\mathcal{I}_{t,s-1}$. Then

$$\phi_t(s-1) = f^t(\bar{x}, \bar{I}) \leq f^t(\hat{x}, \hat{I}) \leq f^t(x, I) = \phi_t(s).$$

Therefore, function $\phi_t(\cdot)$ is nondecreasing over each interval $[s_{j-1}, s_j - 1]$, $j \in [2, p]$. Since this function is integer-valued and bounded by $O(tV)$ according to Remark 2.5.1, it has at most $O(tV)$ nonstable points in any interval where it is nondecreasing. The number of such intervals does not exceed $|S| - 1$. By construction and by Remark 2.5.1, we have

$$|S| \leq \sum_{j=1}^{t} |\mathcal{B}_{h_j}| \leq O(tV).$$

Hence, function $\phi_t(\cdot)$ has at most $O(t^2V^2)$ nonstable points over its feasible domain $[a_1^t, a_2^t]$.

■

We now prove a lemma that shows where to find nonstable points of functions $\phi_t(\cdot)$, $t \in [1, n]$. We use the following notation. If D is a set and a is a number, then $D + a$ denotes the set $\{d + a | d \in D\}$. Analogously, if D_1 and D_2 are sets, then $D_1 + D_2$ denotes the set $\{d_1 + d_2 | d_1 \in D_1, d_2 \in D_2\}$.

Lemma 2.5.1 *The set $\mathcal{B}_{\phi_t}$ is a subset of the set $(\mathcal{B}_{\phi_{t-1}} + \mathcal{B}_{r_t} + \{-1, 0\} - d_t) \cup \mathcal{B}_{h_t}$.*

Proof. Assume that s is in the feasible domain of the function $\phi_t(\cdot)$ and

$$s \notin (\mathcal{B}_{\phi_{t-1}} + \mathcal{B}_{r_t} + \{-1, 0\} - d_t) \cup \mathcal{B}_{h_t}. \tag{2.50}$$

We show that s is a stable point of the function $\phi_t(\cdot)$, namely, $\phi_t(s-1) = \phi_t(s)$. Let (x^1, I^1) be an optimal solution of instance $\mathcal{I}_{t,s-1}$ and (x^2, I^2) be an optimal solution of instance $\mathcal{I}_{t,s}$. Consider the following cases.

1. $I_{t-1}^1 + 1 \notin \mathcal{B}_{\phi_{t-1}}$.

   Let (x', I') be an optimal solution of instance $\mathcal{I}_{t-1, I_{t-1}^1 + 1}$. Construct solution (x^0, I^0) such that $(x_t^0, I_t^0) = (x_t^1, I_t^1 + 1)$ and $(x_i^0, I_i^0) = (x_i', I_i')$ for all $i \in [1, t-1]$. Taking into account $I_t^0 = I_t^1 + 1 = s$ and $s \notin \mathcal{B}_{h_t}$, we obtain

   $$f^t(x^0, I^0) = \phi_{t-1}(I_{t-1}^1 + 1) + r_t(x_t^0) + h_t(I_t^0) = \phi_{t-1}(I_{t-1}^1) + r_t(x_t^1) + h_t(I_t^1) = f^t(x^1, I^1).$$

   Since solution (x^0, I^0) is feasible for instance $\mathcal{I}_{t,s}$,

   $$\phi_t(s) \leq f^t(x^0, I^0) = f^t(x^1, I^1) = \phi_t(s-1).$$

2. $I_{t-1}^1 + 1 \in \mathcal{B}_{\phi_{t-1}}$.

   We have $s - 1 = I_t^1 = I_{t-1}^1 + x_t^1 - d_t$. Therefore, s belongs to $\mathcal{B}_{\phi_{t-1}} + x_t^1 - d_t$. If $x_t^1 + 1 \in \mathcal{B}_{r_t}$, then $s \in \mathcal{B}_{\phi_{t-1}} + \mathcal{B}_{r_t} - 1 - d_t$, which contradicts assumption (2.50). Hence, $x_t^1 + 1 \notin \mathcal{B}_{r_t}$. This implies $r_t(x_t^1 + 1) = r_t(x_t^1)$. Since set $\mathcal{B}_{r_t}$ contains 0 and u_t, relations $0 < x_t^1 + 1 < u_t$ must hold. Construct (x^0, I^0) such that $(x_i^0, I_i^0) = (x_i^1, I_i^1)$

for all $i \in [1, t-1]$ and $(x_t^0, I_t^0) = (x_t^1 + 1, I_t^1 + 1)$. From $I_t^0 = I_t^1 + 1 = s \notin \mathcal{B}_{h_t}$, we deduce $h_t(I_t^1) = h_t(I_t^0)$. Thus, $f^t(x^0, I^0) = f^t(x^1, I^1)$. Since solution (x^0, I^0) is feasible for instance $\mathcal{I}_{t,s}$, we obtain $\phi_t(s) \leq \phi_t(s-1)$.

3. $I_{t-1}^2 \notin \mathcal{B}_{\phi_{t-1}}$.

   Let (x', I') be an optimal solution of instance $\mathcal{I}_{t-1,I_{t-1}^2-1}$. By definition, $I'_{t-1} = I_{t-1}^2 - 1$. Construct (x^0, I^0) such that $(x_i^0, I_i^0) = (x'_i, I'_i)$ for all $i \in [1, t-1]$ and $(x_t^0, I_t^0) = (x_t^2, I_t^2 - 1)$. Solution (x^0, I^0) is feasible for instance $\mathcal{I}_{t,s-1}$. Taking into account that $I_t^2 = s$ and therefore $h_t(I_t^2 - 1) = h_t(I_t^2)$ since s is a stable point of function $h_t(\cdot)$ by (2.50), we obtain

   $$f^t(x^0, I^0) = \phi_{t-1}(I_{t-1}^2 - 1) + r_t(x_t^0) + h_t(I_t^0) = \phi_{t-1}(I_{t-1}^2) + r_t(x_t^2) + h_t(I_t^2) = f^t(x^2, I^2).$$

   Similarly to the previous cases, we get $\phi_t(s-1) \leq f^t(x^0, I^0) = f^t(x^2, I^2) = \phi_t(s)$.

4. $I_{t-1}^2 \in \mathcal{B}_{\phi_{t-1}}$.

   Notice that $s = I_t^2 = I_{t-1}^2 + x_t^2 - d_t$. It implies $x_t^2 > 0$ because otherwise, since 0 belongs to $\mathcal{B}_{r_t}$, s belongs to the set $\mathcal{B}_{\phi_{t-1}} + \mathcal{B}_{r_t} - d_t$, which contradicts assumption (2.50). We deduce that (x^0, I^0) with $(x_i^0, I_i^0) = (x_i^2, I_i^2)$ for all $i \in [1, t-1]$ and $(x_t^0, I_t^0) = (x_t^2 - 1, I_t^2 - 1)$ is feasible for instance $\mathcal{I}_{t,s-1}$. Taking into account $r_t(x_t^0) \leq r_t(x_t^2)$, $I_t^0 = I_t^2 - 1 = s - 1$, $I_t^2 = s$, and $h_t(s-1) = h_t(s)$, we obtain $\phi_t(s-1) \leq f^t(x^0, I^0) \leq f^t(x^2, I^2) = \phi_t(s)$.

Cases 1 and 2 imply $\phi_t(s) \leq \phi_t(s-1)$ and cases 3 and 4 imply $\phi_t(s) \geq \phi_t(s-1)$. Thus, $\phi_t(s) = \phi_t(s-1)$, which means that s is a stable point of the function $\phi_t(\cdot)$. Since this statement is valid for all $s \notin (\mathcal{B}_{\phi_{t-1}} + \mathcal{B}_{r_t} + \{-1, 0\} - d_t) \cup \mathcal{B}_{h_t}$, set $\mathcal{B}_{\phi_t}$ is a subset of the set $(\mathcal{B}_{\phi_{t-1}} + \mathcal{B}_{r_t} + \{-1, 0\} - d_t) \cup \mathcal{B}_{h_t}$. ∎

2.5.3 The algorithm

In this section, we describe an algorithm that constitutes an FPTAS for problem Capacitated Lot-Sizing. Our algorithm assumes that bounds L and U are given such that $0 < L \leq OPT(\mathcal{I}^{\#}) \leq U$.

Finding a lower bound

As we already mentioned, $OPT(\mathcal{I}^{\#}) \geq 0$. The case $OPT(\mathcal{I}^{\#}) = 0$ can be identified as follows. Let us bound each variable x_t by $\gamma_t = \max\{a | a \in [0, u_t], R_t(a) = 0\}$ from above, and bound each variable I_t by $\alpha_t = \min\{b \mid b \in [a_1^t, 0], H_t(b) = 0\}$ from below and by $\beta_t = \max\{b \mid b \in [0, a_2^t], H_t(b) = 0\}$ from above. Here a_1^t and a_2^t are the minimum and maximum values of the variable I_t, see Subsection 2.5.2. The feasible domain of problem Capacitated Lot-Sizing with the additional bounds, denoted as D, can be written as the following system.

$$\begin{aligned} I_t &= I_{t-1} + x_t - d_t, && \forall t \in [1, n], \\ I_0 &= 0, && \\ \alpha_t \leq I_t &\leq \beta_t, && \forall t \in [1, n], \\ 0 \leq x_t &\leq \gamma_t, && \forall t \in [1, n], \\ x_t &\in \mathbb{Z}, && \forall t \in [1, n]. \end{aligned}$$

It is easy to see that $F(x, I) = 0$ for $(x, I) \in D$, and equation $OPT(\mathcal{I}^{\#}) = 0$ is equivalent to $D \neq \emptyset$. The above problem reduces to finding a feasible flow in some network which is a polynomially solvable problem. Nevertheless, for the sake of completeness, we show in more detail how to solve the above system.

Since $I_t = \sum_{i=1}^{t} x_i - \sum_{i=1}^{t} d_i$, $t \in [1, n]$, the above system is equivalent to the following system.

$$p_t \leq \textstyle\sum_{i=1}^{t} x_i \leq q_t, \quad \forall t \in [1, n], \tag{2.51}$$

$$0 \leq x_t \leq \gamma_t, \quad \forall t \in [1, n], \tag{2.52}$$

$$x_t \in \mathbb{Z}, \quad \forall t \in [1, n]. \tag{2.53}$$

Here $p_t = \alpha_t + \sum_{i=1}^{t} d_i$ and $q_t = \beta_t + \sum_{i=1}^{t} d_i$. Notice that $q_t \geq 0$, $t \in [1, n]$.

By the definition, $\alpha_t \leq \beta_t$ and, hence, $p_t \leq q_t$, $t \in [1, n]$.

The following transformation leads to an equivalent system. Since $\sum_{i=1}^{t} x_i \leq \sum_{i=1}^{t+1} x_i$, $t \in [1, n-1]$, for any feasible x, we can reset $q_t := \min\{q_i \mid i = n, n-1, \ldots, t\}$, $t \in [1, n]$. Such a resetting incurs the condition $q_1 \leq \cdots \leq q_n$. Assume that this condition is satisfied.

Lemma 2.5.2 *If system (2.51)-(2.53) has a solution, then there exists a solution x^0 of this system, in which $x_1^0 = \min\{\gamma_1, q_1\}$.*

Proof. Let $x = (x_1, \ldots, x_n)$ be a solution of the system (2.51)-(2.53) and $x_1 < \min\{\gamma_1, q_1\}$. Let $i^0 \geq 2$ be the smallest index such that $x_{i^0} > 0$. If such an index does not exist, then, since $q_1 \leq \cdots \leq q_n$, increasing x_1 up to $\min\{\gamma_1, q_1\}$ does not change the feasibility of x.

Calculate $\xi = \min\{x_{i^0}, \min\{\gamma_1, q_1\} - x_1\}$. Increase x_1 by ξ and decrease x_{i^0} by ξ. The new solution remains feasible because $\sum_{i=1}^{t} x_i = x_1 \leq q_1 \leq q_t$ for $t = 1, \ldots, i^0 - 1$ and sums $\sum_{i=1}^{t} x_i$, $t = i^0, \ldots, n$, do not change. If the inequality $x_1 < \min\{\gamma_1, q_1\}$ is still satisfied, repeat the described transformation for the new solution. After a finite number of iterations we will obtain a feasible solution x^0, in which $x_1^0 = \min\{\gamma_1, q_1\}$. ∎

Let us formulate a *reduced system* (2.51)-(2.53) by removing variable x_1 and resetting $p_i := p_i - x_1^0$, $q_i := q_i - x_1^0$, $i = 2, \ldots, n$. By Lemma 2.5.2, if the reduced system (2.51)-(2.53) has no solution, then the original system (2.51)-(2.53) has no solution as well. Furthermore, if $(x_2, \ldots, x_n)$ is a solution for the reduced system (2.51)-(2.53), then $(x_1^0, x_2, \ldots, x_n)$ is a solution to the original system (2.51)-(2.53).

From the above discussion, it follows that system (2.51)-(2.53) has a solution if and only if vector x^0 is feasible such that

$$x_t^0 = \min\{\gamma_t, q_t - \sum_{i=1}^{t-1} x_i^0\},\ t = 1, \ldots, n,\ \sum_{i=1}^{0} x_i^0 := 0.$$

Thus, when coefficients α_t, β_t, and γ_t are known for all $t \in [1, n]$, the case $OPT(\mathcal{I}^{\#}) = 0$ can be identified in $O(n)$ time. If $OPT(\mathcal{I}^{\#}) = 0$, then the corresponding solution can be found in $O(n)$ time as well. Therefore, taking into account assumptions (i) and (ii) which can be used to evaluate coefficients α_t, β_t, and γ_t, we have $O(\sum_{t=1}^{n}(\log u_t + \log(a_2^t - a_1^t)))$ time complexity for verifying the condition $OPT(\mathcal{I}^{\#}) = 0$.

Now assume that $OPT(\mathcal{I}^{\#}) > 0$. In this case, for any feasible solution (x, I), there exists $t \in [1, n]$ such that at least one of the following three inequalities is satisfied: $x_t > \gamma_t$, $I_t < \alpha_t$, or $I_t > \beta_t$. Therefore, any feasible solution (x, I) satisfies the inequality

$$F(x, I) \geq \min\{\psi(a) \mid \psi(a) \neq 0, \psi \in \{R_t, H_t \mid t \in [1, n]\}\}.$$

The right-hand side of this inequality can be taken as a positive lower bound L for the value $OPT(\mathcal{I}^{\#})$. Under assumptions (i) and (ii), this lower bound can be calculated in $O(\sum_{t=1}^{n}(\log u_t + \log(a_2^t - a_1^t)))$ time.

Adjusting recursive formula and dynamic programming algorithm for the rounded instance

For each function $\psi(\cdot)$ over an interval $[a, b]$ and every point $c_0 \in [a, b]$, define a value $\triangle(\psi, c_0)$ as the length of the maximal interval $[c_0, c_1] \subseteq [a, b]$ where $\psi(\cdot)$ is constant, i.e.,

$$\triangle(\psi, c_0) = \max_{[c_0,c_1]\subseteq[a,b]} \{c_1 - c_0 \mid \psi(c) = \psi(c_0)\ \forall c \in [c_0, c_1]\}.$$

Let us go back to the recursive formula (2.48). Rewrite it as

$$\phi_t(s) = h_t(s) + \min_{s' \in [a_1^{t-1}, a_2^{t-1}]} \{r_t(s - s' + d_t) + \phi_{t-1}(s') \mid s - s' + d_t \in [0, u_t]\}. \tag{2.54}$$

Note that the minimum in the formula (2.54) is attained at a point $s' = \hat{s}$ such that

$$b \le \hat{s} \le b + \triangle(\phi_{t-1}, b),$$

$$a \le s - \hat{s} + d_t \le a + \triangle(r_t, a)$$

for some $b \in \mathcal{B}_{\phi_{t-1}}$ and $a \in \mathcal{B}_{r_t}$. This system of inequalities is equivalent to

$$\max\{b, s + d_t - a - \triangle(r_t, a)\} \le \hat{s} \le \min\{b + \triangle(\phi_{t-1}, b), s + d_t - a\}.$$

Since $r_t(\cdot)$ is constant over $[a, a + \triangle(r_t, a)]$ and $\phi_{t-1}(\cdot)$ is constant over $[b, b + \triangle(\phi_{t-1}, b)]$, each value from

$$[\max\{b, s + d_t - a - \triangle(r_t, a)\}, \min\{b + \triangle(\phi_{t-1}, b), s + d_t - a\}]$$

delivers the minimum in the formula (2.54) and, consequently, in the formula (2.48). This fact leads to the equation

$$\phi_t(s) = h_t(s) + \min_{a \in \mathcal{B}_{r_t},\ b \in \mathcal{B}_{\phi_{t-1}}} \{r_t(s - s_{ab} + d_t) + \phi_{t-1}(s_{ab})\}, \tag{2.55}$$

where s_{ab} is an arbitrary point from the interval $[\max\{b, s + d_t - a - \triangle(r_t, a)\}, \min\{b + \triangle(\phi_{t-1}, b), s + d_t - a\}]$, if it is not empty. If this interval is empty, we set the expression under the above minimum to be equal to ∞ for corresponding a and b.

Thus, $\phi_t(\cdot)$ can be calculated by a simple enumeration of all pairs (a, b) such that $b \in \mathcal{B}_{\phi_{t-1}}$ and $a \in \mathcal{B}_{r_t}$. If sets $\mathcal{B}_{\phi_{t-1}}$ and $\mathcal{B}_{r_t}$ are maintained as sorted lists, then the value $\phi_t(s)$ can be calculated in $O(|\mathcal{B}_{\phi_{t-1}}||\mathcal{B}_{r_t}|)$ time for any $s \in [a_1^t, a_2^t]$.

The above discussion justifies the following dynamic programming algorithm, denoted as DP. It outputs an optimal solution (x^0, I^0) for rounded instance $\mathcal{I}$.

Algorithm DP

Step 0 For all $t \in [1, n]$ generate sets $\mathcal{B}_{r_t}$ and $\mathcal{B}_{h_t}$ as sorted lists;

Set $\mathcal{B}_{\phi_0} := \{0\}$;

Step 1 **for** $t = 1$ **to** n **do**

for $s \in (\mathcal{B}_{r_t} + \mathcal{B}_{\phi_{t-1}} + \{-1, 0\} - d_t) \cup \mathcal{B}_{h_t}$ **do**

initialize $(\hat{a}, \hat{b})$ with $(a, b) \in \mathcal{B}_{r_t} \times \mathcal{B}_{\phi_{t-1}}$

such that $\max\{b, s + d_t - a - \triangle(r_t, a)\} \le \min\{b + \triangle(\phi_{t-1}, b), s + d_t - a\}$;

for $(a, b) \in \mathcal{B}_{r_t} \times \mathcal{B}_{\phi_{t-1}}$ **do**

if $\max\{b, s + d_t - a - \triangle(r_t, a)\} \le \min\{b + \triangle(\phi_{t-1}, b), s + d_t - a\}$ **and**

$r_t(\hat{a}) + \phi_{t-1}(\hat{b}) + h_t(s) > r_t(a) + \phi_{t-1}(b) + h_t(s)$

then $(\hat{a}, \hat{b}) := (a, b)$; **end if**

end do

$\phi_t(s) := r_t(\hat{a}) + \phi_{t-1}(\hat{b}) + h_t(s)$;

end do

Let $B := (\mathcal{B}_{r_t} + \mathcal{B}_{\phi_{t-1}} + \{-1, 0\} - d_t) \cup \mathcal{B}_{h_t}$;

Transform B into $\mathcal{B}_{\phi_t}$ and maintain $\mathcal{B}_{\phi_t}$ as a sorted list;

end do

Step 2 $s_1 := \arg\min\{\phi_n(s) | s \in \mathcal{B}_{\phi_n}\}$;

$s_2 := s_1 + \triangle(\phi_n, s_1)$;

$I_0^0 := 0$;

for $t = n$ **down to** 1 **do**

$\hat{s} :=$ any point from $[s_1, s_2]$;
$I_t^0 := \hat{s}$;
initialize $(\hat{a}, \hat{b})$ with $(a, b) \in \mathcal{B}_{r_t} \times \mathcal{B}_{\phi_{t-1}}$
such that $\max\{b, \hat{s} + d_t - a - \triangle(r_t, a)\} \leq \min\{b + \triangle(\phi_{t-1}, b), \hat{s} + d_t - a\}$;
for $(a, b) \in \mathcal{B}_{r_t} \times \mathcal{B}_{\phi_{t-1}}$ **do**
if $\max\{b, \hat{s} + d_t - a - \triangle(r_t, a)\} \leq \min\{b + \triangle(\phi_{t-1}, b), \hat{s} + d_t - a\}$ **and**
$r_t(\hat{a}) + \phi_{t-1}(\hat{b}) + h_t(\hat{s}) > r_t(a) + \phi_{t-1}(b) + h_t(\hat{s})$
then
$(\hat{a}, \hat{b}) := (a, b)$;
end if
end do
$s_1 := \max\{\hat{b}, \hat{s} + d_t - \hat{a} - \triangle(r_t, \hat{a})\}$;
$s_2 := \min\{\hat{b} + \triangle(\phi_{t-1}, \hat{b}), \hat{s} + d_t - \hat{a}\}$;
end do
for $t = 1$ **to** n **do**
$x_t^0 := I_t^0 - I_{t-1}^0 + d_t$;
end do

Remark 2.5.2 *In Step 2, we assume that function value $\psi(s)$ is stored with each point $s \in \mathcal{B}_\psi$ for $\psi \in \{r_t, h_t, \phi_t \mid t \in [1, n]\}$. We also assume that sets $\mathcal{B}_\psi$ are scanned in increasing order of their elements.*

In Step 1, the algorithm calculates values of the functions $\phi_t(\cdot)$, $t \in [1, n]$, at their nonstable points and builds sets $\mathcal{B}_{\phi_t}$. To find set $\mathcal{B}_{\phi_t}$, set $B = (\mathcal{B}_{r_t} + \mathcal{B}_{\phi_{t-1}} + \{-1, 0\} - d_t) \cup \mathcal{B}_{h_t}$ is maintained as a sorted list in each iteration of the main loop of Step 1. To transform B into $\mathcal{B}_{\phi_t}$, we may remove all stable points of function $\phi_t(\cdot)$ from the set B as follows. Starting with the first element s, scan set B in increasing order of its elements until element s' is found such that $\phi_t(s') \neq \phi_t(s)$. Remove all elements found between elements s and s' because they are stable points of function $\phi_t(\cdot)$. Proceed in the same way, starting with s', until all stable

points of this function are removed from the set B. By Lemma 2.5.1, the resulting set B coincides with the set $\mathcal{B}_{\phi_t}$.

An optimal solution is found in Step 2. This step uses nonstable points of the functions $\phi_t(\cdot)$, $t \in [1, n]$, found in the first step. In the last loop of Step 2, production levels x_t^0 are calculated using optimal inventory levels I_t^0 and I_{t-1}^0 for all $t \in [1, n]$.

Determine the time complexity of the algorithm. Using assumptions (i) and (ii), one can generate sets $\mathcal{B}_{r_t}$ and $\mathcal{B}_{h_t}$ and maintain them as sorted lists in $O(|\mathcal{B}_{r_t}| \log u_t)$ and $O(|\mathcal{B}_{h_t}| \log \sum_{i=1}^{t} u_i)$ time, respectively. Therefore, the running time of Step 0 is $O\left(\sum_{t=1}^{n} \left[|\mathcal{B}_{r_t}| \log u_t + |\mathcal{B}_{h_t}| \log \sum_{i=1}^{t} u_i\right]\right)$.

The cardinality of the set $(\mathcal{B}_{r_t} + \mathcal{B}_{\phi_{t-1}} + \{-1, 0\} - d_t) \cup \mathcal{B}_{h_t}$ is bounded by $O(|\mathcal{B}_{r_t}||\mathcal{B}_{\phi_{t-1}}| + |\mathcal{B}_{h_t}|)$ from above. Therefore for every $t \in [1, n]$, the set B can be generated and maintained as a sorted list in $O((|\mathcal{B}_{r_t}||\mathcal{B}_{\phi_{t-1}}| + |\mathcal{B}_{h_t}|) \log(|\mathcal{B}_{r_t}||\mathcal{B}_{\phi_{t-1}}| + |\mathcal{B}_{h_t}|))$ time. Then, by Remark 2.5.1 and Theorem 2.5.2, Step 1 runs in $O(V^6 n^5)$ time.

Since sets $\mathcal{B}_{r_t}$ and $\mathcal{B}_{\phi_{t-1}}$ are maintained as sorted lists and scanned in increasing order of their elements (see Remark 2.5.2), values $\triangle(r_t, a)$ and $\triangle(\phi_{t-1}, b)$ are computable in constant time in the course of the algorithm DP. Thus, Step 2 runs in $O\left(\sum_{t=1}^{n} |\mathcal{B}_{r_t}||\mathcal{B}_{\phi_{t-1}}|\right)$ time, which does not exceed the complexity estimation of Step 1.

Finally, taking into account $|\mathcal{B}_{r_t}| \leq O(V)$ and $|\mathcal{B}_{h_t}| \leq O(V)$, the overall time complexity of algorithm DP can be estimated as

$$O\left(V^6 n^5 + V \sum_{t=1}^{n} \log \sum_{i=1}^{t} u_i\right).$$

Converting algorithm DP into an FPTAS

Recall that we can set $V = \lfloor U/\delta \rfloor$. Therefore, rounded instance $\mathcal{I}$ can be solved in

$$O\left(\left(\frac{U}{\varepsilon L}\right)^6 n^{11} + \left(\frac{U}{\varepsilon L}\right) n \sum_{t=1}^{n} \log \sum_{i=1}^{t} u_i\right)$$

time, which is pseudopolynomial.

The bound improvement procedure of Tanaev, Kovalyov and Shafransky [54] can be used to find lower and upper bounds such that $U/L = 3$. This procedure is an improved

modification of the procedure given by Kovalyov [43]. While the procedure in [43] performs a bisection search over the range $L, L+1, \ldots, U$, the improved procedure in [54] performs a bisection search over the range $0, 1, \ldots, k$, where $L2^{k-1} < U \leq L2^k$. In each iteration of any of the two procedures, algorithm DP can be applied. It uses an artificial upper bound from the interval $2L, 2L+1, \ldots, U$, a lower bound being a half of the upper bound, and $\varepsilon = 1$. The running time of the bound improvement procedure [54] for problem Capacitated Lot-Sizing is

$$O\left(\left(n^{11} + n\sum_{t=1}^{n}\log\sum_{i=1}^{t}u_i\right)\log\log\frac{U}{L}\right).$$

A detailed description of the bound improvement procedure and the proof of its validity is given in Section 1.4.

Theorem 2.5.1, properties of algorithm DP, and the bound improvement procedure imply the following theorem, which is the main result of this section.

Theorem 2.5.3 *Algorithm DP in combination with the bound improvement procedure is an FPTAS for problem* Capacitated Lot-Sizing *under assumptions (i)-(iii). This FPTAS runs in*

$$O\left(n^{11}\left(\frac{1}{\varepsilon^6} + \log\log\frac{U}{L}\right) + n\left(\frac{1}{\varepsilon} + \log\log\frac{U}{L}\right)\sum_{t=1}^{n}\log\sum_{i=1}^{t}u_i\right)$$

time for any $\varepsilon > 0$.

2.5.4 Conclusions

We presented an FPTAS for a capacitated economic lot-sizing problem with a monotone cost structure. Unlike earlier approaches to constructing FPTASes for such problems, we do not change the feasible domain of the problem. Instead, we exploit combinatorial properties of the recursive function in a straightforward dynamic programming algorithm applied to a rounded problem.

Future research can be undertaken to apply ideas of this section to other NP-hard combinatorial optimization problems with similar feasible domains. An improvement of the time complexity of the presented FPTAS is of interest as well.

2.6 Generalizations of the CELS problem

This section is based on the paper [16] by Chubanov, Kovalyov, and Pesch.

2.6.1 Problem formulation

As well as in the previous sections, where we dealt with the classical single-item Capacitated Economic Lot-Sizing (CELS) problem, we consider a single product that has to be manufactured by a single facility within the planning horizon consisting of n time periods.

Previously we considered variables x_t to be production levels at every period t. In the model we consider in this section, it is much more convenient to interpret x_t as amount of a resource used for production. Actually, this value may involve several resource types. For instance, if a product unit requires a units of resource A and b units of resource B, then under x_t units of the resource we imply $x_t a$ units of resource A and $x_t b$ units of resource B. In this case the resource is a sort of a compound resource.

In many real situations it may happen that the same production efforts yield different results at different time moments. There are many reasons influencing this; productivity may vary over time, or some part of the product is defective and thus cannot be used to satisfy demand. In other words, there are external factors not allowing an ideal case when we know exactly that consuming A units of resource results in B units of product. Namely, a total production level at each period t (the amount of product units we have produced by the end of period t) may be expressed by some, in general nonlinear, function $g_t(\cdot)$ depending on resource consumption levels $x_1, \ldots, x_t$. Later we will describe some more natural assumptions about the structure of this function. Now let us formulate the problem which we call capacitated economic lot-sizing problem with a non-uniform resource (which we further call Non-uniform Lot-sizing). The word "non-uniform" reflects not only the fact that $g_t(\cdot)$ may be in general nonlinear, but also a restriction that a resource consumption level x_t takes values from some given set of integers. This restriction is reasonable, for instance, if the resource goes into the manufacturing system in batches (which may be of different size).

So problem NON-UNIFORM LOT-SIZING is formulated as follows.

$$\begin{aligned} \min \quad & F(x) = F(x_1, \ldots, x_n) = \textstyle\sum_{t=1}^{n} (R_t(x_t) + H_t(J_t)), \\ \text{s.t.} \quad & J_t = g_t(x) - D_t, && t \in [1, n], \\ & x_t \in \mathcal{A}_t, && t \in [1, n]. \end{aligned} \tag{2.56}$$

Parameters of problem NON-UNIFORM LOT-SIZING can be defined and interpreted as follows.

- x_t – resource value in time period t, a decision variable.
- $\mathcal{A}_t = \{0, \alpha_1^t, \ldots, \alpha_{k_t}^t\}$ – set of feasible resource values in time period t such that $0 < \alpha_1^t < \alpha_2^t < \cdots < \alpha_{k_t}^t$.
- $R_t(\cdot)$ – *resource (usage) cost function* in time period t.
- $g_t(\cdot)$ – function representing the total number of product units manufactured up to the end of time period t.
- $H_t(\cdot)$ – holding-backlogging cost function.
- D_t – total demand by the end of period t.
- J_t – inventory level at the end of time period t, $J_t = g_t(x) - D_t$.

Note that if we use notation d_i for demand at each period $i \in [1, t]$ as it was in the previous sections, then $D_t = \sum_{i=1}^{t} d_i$.

It is not hard to see that the classical CELS problem is a special case of problem NON-UNIFORM LOT-SIZING, in which one unit of the resource gives rise to one unit of the product, that is $g_t(x) = \sum_{i=1}^{t} x_i$, and $\mathcal{A}_t = [0, u_t]$, $t \in [1, n]$.

We call vector $x = (x_1, \ldots, x_n)$ feasible if conditions (2.56) are satisfied. Let x^* be an optimal solution to problem NON-UNIFORM LOT-SIZING and let F^* be its optimal value, i.e., $F^* = F(x^*)$.

Similar to the literature on the classical CELS problem, we make the following realistic assumptions.

Assumption 2.6.1 *Resource cost functions $R_t(\cdot)$ are positive and non-decreasing on $\mathbb{Z}_{++}$, and $R_t(0) = 0$, $t \in [1, n]$.*

Assumption 2.6.2 *Holding-backlogging cost functions $H_t(\cdot)$, $t \in [1, n]$, are positive and non-increasing on $\mathbb{Z}_{--}$ and positive and non-decreasing on $\mathbb{Z}_{++}$. Furthermore, $H_t(0) = 0$, $t \in [1, n]$.*

Let $g_0(x_0)$ denote an initial inventory level, i.e., the number of product units available at the beginning of the first time period $t = 1$. To simplify the exposition, we assume that $g_0(x_0) = 0$.

The following assumption, along with the constraints $x_t \in \mathcal{A}_t$, $t \in [1, n]$, makes problem Non-uniform Lot-sizing different from the classical CELS problem.

Assumption 2.6.3 *Functions $g_t(x)$ are defined for $x \geq 0$ and recursively computed by the formula*

$$g_t(x) = G_t(g_{t-1}(x), x_t), \ t \in [1, n],$$

where G_t is a non-negative integer function increasing in each argument such that $G_t(\beta, \gamma) = 0$ if and only if $(\beta, \gamma) = (0, 0)$.

Assumption 2.6.3 implies that function $g_t(x)$ solely depends on the first t components of vector x. Furthermore, $g_t(x) = 0$ if and only if $x_1 = \cdots = x_t = 0$.

Assumption 2.6.4 *Each function R_t, H_t and G_t, $t \in [1, n]$, is computable in a constant time at every single point of its feasible domain. Furthermore, given value v and one of the parameters β and γ, the equation $G_t(\beta, \gamma) = v$ can be solved in a constant time with respect to the unknown parameter, $t \in [1, n]$.*

Notice that problem Non-uniform Lot-sizing admits an equivalent formulation, in which each restriction $x_t \in \mathcal{A}_t = \{0, \alpha_1^t, \ldots, \alpha_{k_t}^t\}$ is replaced by a simpler restriction $x_t \in [0, k_t]$ (the same as in the classical CELS problem) and each function $g_t(x)$ is re-defined as $g_t(x) := G_t(g_{t-1}(x), \alpha_{x_t}^t)$, where $\alpha_0^t = 0$. As far as we know, problem Non-uniform Lot-sizing (at least in the general form) was never addressed in the literature.

The remainder of the section is organized as follows. In the next subsection, we demonstrate that under Assumptions 2.6.1-2.6.4 problem NON-UNIFORM LOT-SIZING and several of its special cases cannot be approximated in polynomial time within any polynomially computable relative error unless P=NP. This implies that an FPTAS does not exist for the general case of problem NON-UNIFORM LOT-SIZING and some special cases. In Subsection 2.6.3, we describe an exact dynamic programming algorithm for problem NON-UNIFORM LOT-SIZING. Approximation algorithms for special cases are derived in Subsection 2.6.4 where the dynamic programming algorithm is modified to be an approximation algorithm with a guaranteed absolute error under specific conditions. It is further modified to provide an FPTAS for a special case of problem NON-UNIFORM LOT-SIZING with separable functions $g_t(x)$ and linear holding and backlogging cost functions with polynomially related slopes and to obtain an FPTAS for problem NON-UNIFORM LOT-SIZING with zero holding costs. Subsection 2.6.5 addresses the related problem of minimizing maximum cost. Subsection 2.6.6 deals with a case when product losses are possible due to defects or other factors.

2.6.2 Non-approximability

One of the main results of this section is the following theorem.

Theorem 2.6.1 *Let $\Pi(\cdot)$ be an arbitrary positive polynomially computable nondecreasing function. If $P \neq NP$, then there exists no polynomial algorithm delivering a $\Pi(size(\mathcal{I}))$-approximate solution for any instance $\mathcal{I}$ of problem* NON-UNIFORM LOT-SIZING *under Assumptions 2.6.1-2.6.4 even if*

1) $x_t \in \{0, a_t\}$, $R_t(x_t) = x_t$, $G_t(\beta, \gamma) = \beta + \gamma$, $t \in [1, n]$; and

2) $H_t(J_t) = 0$, $D_t = 0$, $t \in [1, n-1]$, $H_n(J_n) = K|J_n|$.

Proof. Consider the following NP-hard problem PARTITION, see Garey and Johnson [28].

PARTITION: Given $m+1$ positive integers $s_1, \ldots, s_m$ and S such that $\sum_{i=1}^{m} s_i = 2S$, is there a subset $X \subseteq M := \{1, \ldots, m\}$ such that $\sum_{i \in X} s_i = S$?

Given an instance $\mathcal{I}_P$ of PARTITION and the value $\Pi(size(\mathcal{I})) > 0$, we will now construct an instance $\mathcal{I}_{NU}$ of problem NON-UNIFORM LOT-SIZING. Set $n := m$. Define problem

parameters according to 1) and 2) so that $a_t := s_t$, for all $t \in [1, n]$, $D_n := S$, and $K := (1+\Pi(c \cdot size(\mathcal{I}_P)))S$ where constant c is defined as follows. Note that one needs a string of a length which is not greater than $size(\mathcal{I}_P)$ to encode all cost functions for instances satisfying 1) and 2). Therefore there is a positive integer constant c such that $size(\mathcal{I}_{NU})$ does not exceed $c \cdot size(\mathcal{I}_P)$. (This constant can be equal to at most 4 which corresponds to encoding three types of functions in the problem formulation and vector $(a_1, \ldots, a_n)$.) Instance $\mathcal{I}_{NU}$ can be constructed in time polynomial of $size(\mathcal{I}_P)$.

Let x^* be an optimal solution of instance $\mathcal{I}_{NU}$ and let F^* be the corresponding objective value. Denote $X^* = \{t | x_t^* > 0, i \in M\}$. Using 1) and 2), we obtain

$$F^* = \sum_{t=1}^{n} (R_t(x_t^*) + H_t(g_t(x^*) - D_t)) = \sum_{t \in X^*} s_t + K | \sum_{t \in X^*} s_t - S|.$$

It is easy to see that $F^* = S \leq K$ if problem PARTITION has a solution and $F^* > K$ if problem PARTITION has no solution.

Assume that there exists a polynomial algorithm that for any instance $\mathcal{I}$ of problem NON-UNIFORM LOT-SIZING finds a $\Pi(size(\mathcal{I}))$-approximate solution. If it finds a solution for instance $\mathcal{I}_{NU}$ with value $F^0 > K$, then from

$$F^0 \leq (1 + \Pi(size(\mathcal{I}_{NU})))F^* \leq (1 + \Pi(c \cdot size(\mathcal{I}_P)))F^*$$

we obtain

$$F^* \geq F^0/(1 + \Pi(c \cdot size(\mathcal{I}_P))) > K/(1 + \Pi(c \cdot size(\mathcal{I}_P))) = S.$$

In this case, instance $\mathcal{I}_P$ of problem PARTITION has no solution. Conversely, if it finds a solution with value $F^0 \leq K$, then $F^* \leq F^0 \leq K$ and instance $\mathcal{I}_P$ has a solution.

Thus, our assumption implies the existence of a polynomial time algorithm for the NP-hard problem PARTITION, which is impossible unless P=NP. ∎

The proof of Theorem 2.6.1 can easily be modified to prove the following corollaries.

Corollary 2.6.1 *There exists no polynomial algorithm that for any instance $\mathcal{I}$ of problem* NON-UNIFORM LOT-SIZING *finds a $\Pi(size(\mathcal{I}))$-approximate solution under Assumptions 2.6.1-2.6.4 and the "no backlogging" assumption such that $H_t(J_t) = \infty$ for $J_t < 0$, $t \in [1, n]$, unless $P = NP$.*

Corollary 2.6.2 *There exists no polynomial algorithm that for any instance $\mathcal{I}$ of problem* Non-uniform Lot-sizing *finds a $\Pi(size(\mathcal{I}))$-approximate solution under Assumptions 2.6.1-2.6.4 and assumptions $x_t \in \{0,1\}$, $G_t(\beta,\gamma) = \beta + c_t\gamma$, $t \in [1,n]$, unless $P = NP$.*

Corollary 2.6.3 *There exists no polynomial algorithm delivering a $\Pi(size(\mathcal{I}))$-approximate solution for any instance $\mathcal{I}$ of the modification of the classical CELS problem where constraints $x_t \in [0,u_t]$ are substituted with $x_t \in \{0,u_t\}$, $t \in [1,n]$, unless $P = NP$.*

Notice that the non-existence of a polynomial time approximation algorithm with a polynomially computable relative error for any problem implies its NP-hardness and nonexistence of an FPTAS for this problem unless P=NP.

Thus, efficient approximation algorithms like FPTASs can only be constructed for some special cases of the problems considered in this section.

2.6.3 Dynamic programming algorithm

In this subsection, we describe a dynamic programming algorithm, denoted as DP, to solve problem Non-uniform Lot-sizing. Denote $F_t(x) = \sum_{i=1}^{t}(R_i(x_i) + H_i(g_i(x) - D_i))$. Algorithm DP recursively computes the value $F_t(g)$, which is the minimum value of $F_t(x)$, subject to the condition that the values for the first t variables $x_1,\ldots,x_t$ are determined, $x_i \in \mathcal{A}_i$, $i = 1,\ldots,t$, and $g_t(x) = g$. A formal description of algorithm DP is given below.

Algorithm DP

Step 1 (Initialization) Set $F_0(0) = 0$ and $t = 1$.

Step 2 (Recursion) For $g = 0,1,\ldots,U_t$, where $U_t = g_t(\alpha^1_{k_1},\alpha^2_{k_2},\ldots,\alpha^t_{k_t})$, compute the following:

$$F_t(g) = \begin{cases} \min\{F_{t-1}(g') + R_t(x_t) + H_t(g - D_t) \mid (g',x_t) \in \mathcal{Q}_t(g)\}, & \text{if } \mathcal{Q}_t(g) \neq \emptyset, \\ \infty, & \text{otherwise,} \end{cases}$$

where

$$\mathcal{Q}_t(g) = \{(g',x_t) \mid G_t(g',x_t) = g,\ g' = 0,1,\ldots,U_{t-1},\ x_t \in \mathcal{A}_t\}.$$

If $t < n$, then set $t = t + 1$ and repeat Step 2. Otherwise, go to Step 3.

Step 3 (Optimal solution) Calculate an optimal solution value

$$F^* = \min\{F_n(g) \mid g = 0, 1, \ldots, U_n\}.$$

A corresponding optimal solution x^* can be found by backtracking. ■

Let us show the optimality of algorithm DP. We say that a partial solution $x = (x_1, \ldots, x_t)$ is in the *state* (t, g) if $x_i \in \mathcal{A}_i$, $i = 1, \ldots, t$, and $g_t(x) = g$. Consider partial solutions in the same state (t, g). A solution with minimum value of $F_t(x)$ *dominates* all other solutions in the same state in the following sense: it can be extended to a complete feasible solution in the same way as any other solution in the same state and will have smaller or equal value of the objective function. Therefore, it is optimal to select only a dominant solution in each state, as algorithm DP does.

The time complexity of algorithm DP can be estimated as $O(nU_{\max}(|\mathcal{Q}_{\max}| + B_{\max}))$, where $U_{\max}$ is the maximum value of U_t, $\mathcal{Q}_{\max}$ is the set $\mathcal{Q}_t(g)$ of maximum cardinality and $B_{\max}$ is the maximum time to calculate an arbitrary set $\mathcal{Q}_t(g)$ in Step 2. Since function $G_t(\beta, \gamma)$ is increasing in each argument, the equation $G_t(g', x_t) = g$ has a unique solution with respect to g' or x_t if one of these parameters and value g are fixed. Therefore, $|\mathcal{Q}_{\max}| \leq \min\{U_{\max}, |\mathcal{A}_{\max}|\}$ where $\mathcal{A}_{\max}$ is the set of maximum cardinality among sets $\mathcal{A}_t$. Let us estimate $B_{\max}$. Given t and g, set $\mathcal{Q}_t(g)$ can be calculated by one of the following two techniques. Initiate $\mathcal{Q}_t(g) = \emptyset$.

1) Enumerate $g' = 0, 1, \ldots, U_{t-1}$. Given g', find x_t such that $G_t(g', x_t) = g$. If there exists such x_t, then it can be found in $O(\log |\mathcal{A}_t|)$ time by a bisection search over the set $\mathcal{A}_t$ using the assumption that $G_t(\cdot, \cdot)$ is monotone in the second variable. If an appropriate x_t has been found, then add (g', x_t) to $\mathcal{Q}_t(g)$. In this case, $B_{\max} \leq O(U_{\max} \log |\mathcal{A}_{\max}|)$.

2) Enumerate $x_t \in \mathcal{A}_t$. Given x_t, find g' such that $G_t(g', x_t) = g$ and check $g' \in \{0, 1, \ldots, U_{t-1}\}$. The latter condition can be checked in a constant time. If it is satisfied, then add (g', x_t) to $\mathcal{Q}_t(g)$. In this case, $B_{\max} \leq O(|\mathcal{A}_{\max}|)$.

Thus, the overall time complexity of algorithm DP can be estimated as $O(nU_{\max} \min\{U_{\max} \log |\mathcal{A}_{\max}|, |\mathcal{A}_{\max}|\})$. It is a pseudopolynomial algorithm for problem Non-uniform Lot-sizing if Assumption 2.6.4 is satisfied.

Thus, problem NON-UNIFORM LOT-SIZING, while NP-hard and non-approximable with any polynomially computable relative error in polynomial time, is pseudopolynomially solvable.

2.6.4 Approximation algorithms

In this section, we present at first a heuristic approach for the general problem NON-UNIFORM LOT-SIZING. After that, we specify some assumptions and show that this approach provides a solution with a guaranteed absolute error. Further this approach is converted into an FPTAS for a special case of problem NON-UNIFORM LOT-SIZING with separable functions $g_t(x) = \sum_{i=1}^{t} \rho_i(x_i)$, where ρ_i are positive integer functions, and linear holding and backlogging cost functions

$$H_t(J_t) = \begin{cases} -\nu_t^B J_t, & \text{if } J_t < 0, \\ 0, & \text{if } J_t = 0, \\ \nu_t^H J_n, & \text{if } J_t > 0, \end{cases} \tag{2.57}$$

where ν_t^H and ν_t^B are given positive numbers such that the ratio

$$\frac{\nu_{\max}}{\nu_{\min}} := \frac{\max\{\nu_t^H, \nu_t^B \mid t \in [1,n]\}}{\min\{\nu_t^H, \nu_t^B \mid t \in [1,n]\}}$$

is bounded by a polynomial in the problem input length in binary encoding.

An FPTAS for problem NON-UNIFORM LOT-SIZING with zero holding costs is also presented in this subsection.

An algorithm with a guaranteed absolute error

Algorithm DP in the previous section can be modified to be an approximation algorithm for problem NON-UNIFORM LOT-SIZING as follows. Given a positive number δ, introduce auxiliary recursively computable functions

$$g_t^\delta(x) = \delta \lfloor \frac{G_t(g_{t-1}^\delta(x), x_t)}{\delta} \rfloor, \; g_0^\delta(x_0) := 0. \tag{2.58}$$

In algorithm DP, replace function $G_t(\cdot,\cdot)$ by $\delta\lfloor G_t(\cdot,\cdot)/\delta \rfloor$. Find $\mathcal{Q}_t(g)$ by the technique 1) enumerating those g and g' which are multiples of δ that do not exceed U_t and U_{t-1}, respectively.

Denote problem NON-UNIFORM LOT-SIZING with functions g_t^δ, $t \in [1,n]$, as problem NON-UNIFORM LOT-SIZING$_\delta$. Denote an optimal solution to this problem as x^δ and algorithm DP modified for this problem as DP$_\delta$. Feasible domains of both problems coincide.

Theorem 2.6.2 *Given $m > 0$, $\delta > 0$ and functions g_t^δ satisfying (2.58), if holding-backlogging cost functions satisfy*

$$|H_t(g_t^\delta(x) - D_t) - H_t(g_t(x) - D_t)| \le m\delta,\ t = 1, \ldots, n, \tag{2.59}$$

for any feasible x, then x^δ is an approximate solution to the original problem NON-UNIFORM LOT-SIZING *with an absolute error $2mn\delta$, i.e.,*

$$F(x^\delta) \le F(x^*) + 2nm\delta. \tag{2.60}$$

Furthermore, the running time of algorithm DP$_\delta$ can be estimated as

$$O\left(n\left(\frac{U_{\max}}{\delta}\right)^2 \log|\mathcal{A}_{\max}|\right).$$

Proof. The following chain of relations proves inequality (2.60).

$$F(x^\delta) = \sum_{t=1}^{n}(R_t(x_t^\delta) + H_t(g_t(x^\delta) - D_t)) \le \sum_{t=1}^{n}(R_t(x_t^\delta) + H_t(g_t^\delta(x^\delta) - D_t)) + nm\delta \le$$

$$\sum_{t=1}^{n}(R_t(x_t^*) + H_t(g_t^\delta(x^*) - D_t)) + nm\delta \le F(x^*) + 2nm\delta.$$

The first and third inequalities "$\le$" follow from (2.59). The second inequality "$\le$" follows from the fact that x^* is a feasible solution for problem NON-UNIFORM LOT-SIZING$_\delta$.

Since values of auxiliary functions $g_t^\delta(x)$ are multiples of δ, the running time of algorithm DP$_\delta$ can be estimated as $O(n(U_{\max}^\delta)^2 \log|\mathcal{A}_{\max}|)$ where $U_{\max}^\delta = \max_{t=1,\ldots,n}\{\lfloor g_t^\delta(\alpha_{k_1}^1, \alpha_{k_2}^2, \ldots, \alpha_{k_t}^t)/\delta \rfloor\}$. Note that $g_0^\delta(x_0) = g_0(x_0) = 0$ and $g_1^\delta(x) = \delta\lfloor g_1(x)/\delta \rfloor \le g_1(x)$ for any feasible x. It is easy to show by induction that $g_t^\delta(x) \le g_t(x)$, $t \in [1,n]$, for any feasible x. Then

$$U_{\max}^\delta \le \max_{t=1,\ldots,n}\{\lfloor g_t(\alpha_{k_1}^1, \alpha_{k_2}^2, \ldots, \alpha_{k_t}^t)/\delta \rfloor\} = \lfloor \max_{t=1,\ldots,n}\{g_t(\alpha_{k_1}^1, \alpha_{k_2}^2, \ldots, \alpha_{k_t}^t)/\delta\} \rfloor = \lfloor U_{\max}/\delta \rfloor$$

and we deduce that the indicated running time of algorithm DP$_\delta$ is correct. ∎

Corollary 2.6.4 *Let Z be the input length of problem* Non-uniform Lot-sizing *in binary encoding. If for any $\varepsilon > 0$ value δ can be chosen such that inequalities (2.59) are satisfied, $\varepsilon F^*/(2Z^l n) \leq \delta \leq \varepsilon F^*/(2mn)$ and $U_{\max}/F^* \leq Z^k$, where powers l and k are some constants, then algorithms DP_δ constitute an FPTAS for problem* Non-uniform Lot-sizing *with running time $O(Z^{2(l+k)} n^3 \log |\mathcal{A}_{\max}|/\varepsilon^2)$ of each algorithm.*

Proof. If $\delta \leq \varepsilon F^*/(2mn)$, then inequality (2.60) can be extended to the inequality $F(x^\delta) \leq F(x^*) + 2nm\delta \leq (1+\varepsilon)F(x^*)$. Therefore, x^δ is an $\varepsilon-$approximate solution to problem Non-uniform Lot-sizing. The running time of algorithm DP_δ can be estimated as follows. If $\delta \geq \varepsilon F^*/(2Z^l n)$ and $U_{\max}/F^* \leq Z^k$, we have $U_{\max}/\delta \leq 2U_{\max} Z^l n/(\varepsilon F^*) \leq 2Z^{l+k} n/\varepsilon$. It follows from Theorem 2.6.2 that algorithm DP_δ runs in $O(Z^{2(l+k)} n^3 \log |\mathcal{A}_{\max}|/\varepsilon^2)$ time. ∎

An FPTAS for the case of separable functions $g_t(x)$ and linear holding and backlogging cost functions with polynomially related slopes

Assume that $g_t(x) = \sum_{i=1}^t \rho_i(x_i)$, where ρ_i are positive integer functions, functions $H_t(\cdot)$, $t \in [1, n]$, are determined according to (2.57), and $\nu_{\max}/\nu_{\min}$ is bounded by a polynomial of the problem input size. Using Theorem 2.6.1, it is easy to show that this special case of problem Non-uniform Lot-sizing is NP-hard even if all resource cost functions $R_t(\cdot)$ are equal to zero and all coefficients ν_t^H and ν_t^B are equal to one.

We will use the same scaling approach as before. Determine

$$g_t^\delta(x) = \sum_{i=1}^t \delta \left\lfloor \frac{\rho_i(x_i)}{\delta} \right\rfloor, \ t \in [1, n].$$

Note that we have $G_t(\beta, \gamma) = \beta + \rho_t(\gamma)$ and thus

$$\delta \left\lfloor \frac{G_t(g_{t-1}^\delta(x), x_t)}{\delta} \right\rfloor = \delta \left\lfloor \frac{\sum_{i=1}^{t-1} \delta \lfloor \rho_i(x_i)/\delta \rfloor + \rho_t(x_t)}{\delta} \right\rfloor = g_t^\delta(x).$$

Thus, condition (2.58) is satisfied. Furthermore, $|g_t^\delta(x) - g_t(x)| \leq t\delta$ for any feasible x, which implies

$$|H_t(g_t^\delta(x) - D_t) - H_t(g_t(x) - D_t)| \leq \nu_{\max} t\delta.$$

The latter inequality is equivalent to (2.59) for $m = \nu_{\max} t$. Therefore, algorithm DP_δ can be applied to find an approximate solution with an absolute error $2n^2\nu_{\max}\delta$. Let us modify it to provide an FPTAS for the considered special case of problem Non-uniform Lot-sizing.

In the development of this FPTAS and an FPTAS in the next section, we shall use lower and upper bounds on the optimal value F^* of problem Non-uniform Lot-sizing. Below we show how to determine such bounds.

Let LB and UB be such numbers that $0 < LB \leq F^* \leq UB$. For an upper bound, the value of a feasible solution can be taken. For example, $UB = F(x^{\max})$, where $x^{\max} = (\alpha^1_{k_1}, \alpha^2_{k_2}, \ldots, \alpha^n_{k_n})$. Let us show how to verify whether $F^* = 0$ or $F^* > 0$ and, if $F^* > 0$, how to find a positive lower bound.

Lemma 2.6.1 *Let Assumptions 2.6.1-2.6.4 be satisfied. Then $F^* > 0$ if and only if $D_t > 0$ for some $t \in [1, n]$.*

Proof. Assume $D_t = 0$, $t \in [1, n]$. Then by Assumptions 2.6.1-2.6.4, $F^* = F(0, \ldots, 0) = 0$. Now let $D_t > 0$ for some t. Consider two cases: 1) there exists $x^*_t > 0$, $t \in [1, n]$, and 2) $x^* = 0$. If there exists $x^*_t > 0$, then

$$F^* \geq R_t(x^*_t) \geq LB_1 := \min\{R_k(\alpha^k_1) | k = 1, \ldots, n\}.$$

Assumption 2.6.1 implies $LB_1 > 0$. If $x^* = \mathbf{0}$, then

$$F^* = LB_2 := \sum_{t=1}^{n} H_t(-D_t).$$

Assumption 2.6.2 implies $LB_2 > 0$. Thus, if there exists $D_t > 0$, we can set $LB = \min\{LB_1, LB_2\}$. This lower bound can be computed in $O(n)$ time. ∎

For the special case of problem Non-uniform Lot-sizing that we consider, set

$$\delta = \varepsilon LB/(2n^2\nu_{\max}).$$

Denote $f(x) = \sum_{t=1}^{n}(R_t(x) + H_t(g^\delta_t(x) - D_t))$. From relation (2.60), it is easy to see that an optimal solution x^δ to the problem of minimizing $f(x)$, subject to $x_t \in \mathcal{A}_t$, $t \in [1, n]$, is an ε-approximate solution to the considered special case of problem Non-uniform Lot-sizing.

Observe that $f(x^\delta) \leq F^* + 2n^2\nu_{\max}\delta \leq UB' := UB + 2n^2\nu_{\max}\delta$. Therefore, $H_t(g_t^\delta(x^\delta) - D_t) \leq UB'$, which implies

$$D_t - \frac{UB'}{\nu_t^B} \leq g_t^\delta(x^\delta) \leq D_t + \frac{UB'}{\nu_t^H}.$$

We deduce that

$$\frac{D_t}{\delta} - \frac{UB'}{\delta\nu_t^B} \leq \sum_{i=1}^{t} \left\lfloor \frac{\rho_i(x_i^\delta)}{\delta} \right\rfloor \leq \frac{D_t}{\delta} + \frac{UB'}{\delta\nu_t^H}$$

and, since the middle part of the above relation is integer, we get

$$g_t^\delta(x^\delta) \in \mathcal{B}_t := \{\delta \left\lceil \frac{D_t}{\delta} - \frac{UB'}{\delta\nu_t^B} \right\rceil, \delta \left(\left\lceil \frac{D_t}{\delta} - \frac{UB'}{\delta\nu_t^B} \right\rceil + 1 \right), \ldots, \delta \left\lfloor \frac{D_t}{\delta} + \frac{UB'}{\delta\nu_t^H} \right\rfloor\}.$$

Denote $N := \nu_{\max}/\nu_{\min}$. We have

$$|\mathcal{B}_t| \leq O\left(\frac{UB'}{\delta\nu_{\min}}\right) = O\left(\frac{UB}{\varepsilon LB} n^2 N\right), \ t \in [1, n].$$

Algorithm DP_δ can be modified to find x^δ. At iteration t of Step 2 of this algorithm, we can enumerate only values $g \in \mathcal{B}_t$ and $g' \in \mathcal{B}_{t-1}$. In this case, the time complexity of this algorithm becomes

$$O\left(n^5 N^2 \log|\mathcal{A}_{\max}| \left(\frac{UB}{\varepsilon LB}\right)^2\right).$$

This time estimation can be decreased to

$$O\left(n^5 N^2 \log|\mathcal{A}_{\max}| \left(\frac{1}{\varepsilon^2} + \log\log\frac{UB}{LB}\right)\right) \tag{2.61}$$

if we use the second version of the bound improvement procedure (see Section 1.4) before running DP_δ.

If N is bounded by a polynomial of the input size, then the described modification of algorithm DP_δ with incorporated bound improvement procedure constitutes an FPTAS for the considered special case of problem Non-uniform Lot-sizing.

An FPTAS for problem Non-uniform Lot-sizing with zero holding costs

Now we study problem Non-uniform Lot-sizing under the following assumption.

Assumption 2.6.5 *$H_t(J_t) = 0$ for $J_t > 0$, $t \in [1, n]$.*

Theorem 2.6.1 can be easily modified to show that problem NON-UNIFORM LOT-SIZING with zero holding costs is NP-hard. This problem can be used for modelling situations when holding costs are negligibly small as compared with the resource usage and backlogging costs. An important special case of this problem arises when one would like to minimize the resource usage costs, subject to the no backlogging requirement when all the demands must be satisfied. The latter problem can be formulated as the following generalization of the well-known knapsack problem.

$$\begin{aligned} \min \quad & \sum_{t=1}^{n} R_t(x_t), \\ \text{s.t.} \quad & g_t(x) \geq D_t, \ t = 1, \ldots, n, \\ & x_t \in \mathcal{A}_t, \ t = 1, \ldots, n. \end{aligned}$$

No FPTAS is reported in the literature for the above generalized knapsack problem. However, for some practical situations this problem can provide more adequate mathematical modelling than studied in the literature on knapsack problems.

Consider an instance of problem NON-UNIFORM LOT-SIZING with zero holding costs. Let F^* and x^* denote the optimal value and an optimal solution of this instance, respectively. Assume that bounds LB and UB such that $0 < LB \leq F^* \leq UB$ are given. Define $\delta = \varepsilon LB/(2n)$ and formulate the following *rounded instance* as follows:

$$\begin{aligned} \min \quad & f(x) = \sum_{t=1}^{n} \left\lfloor \frac{R_t(x_t)}{\delta} \right\rfloor + \left\lfloor \frac{H_t(g_t(x) - D_t)}{\delta} \right\rfloor, \\ \text{s.t.} \quad & x_t \in \mathcal{A}_t \qquad t = 1, \ldots, n. \end{aligned}$$

All rounded instances of the above form constitute the problem which we call the *rounded problem*. We now establish a relation between the original problem NON-UNIFORM LOT-SIZING with zero holding costs and the rounded problem.

Theorem 2.6.3 *An optimal solution to the rounded instance is an ε-approximate solution to the corresponding original instance of problem* NON-UNIFORM LOT-SIZING *with zero holding costs.*

Proof. Denote an optimal solution to the rounded instance as x^0. Since feasible domains of both instances coincide, x^0 is feasible for the corresponding instance of problem NON-

UNIFORM LOT-SIZING and x^* is in turn feasible for the rounded instance. It remains to show that $F(x^0) \leq (1+\varepsilon)F(x^*)$. We have

$$F(x^0) \leq \delta \sum_{t=1}^{n} (\lfloor \frac{R_t(x_t^0)}{\delta} \rfloor + \lfloor \frac{H_t(g_t(x^0) - D_t)}{\delta} \rfloor) + 2n\delta \leq$$

(x^* is feasible for the rounded instance)

$$\delta \sum_{t=1}^{n} (\lfloor \frac{R_t(x_t^*)}{\delta} \rfloor + \lfloor \frac{H_t(g_t(x^*) - D_t)}{\delta} \rfloor) + 2n\delta \leq F(x^*) + 2n\delta \leq (1+\varepsilon)F(x^*).$$

■

Let us describe a dynamic programming algorithm, denoted as DP(ROU), to solve the rounded instance. Denote $f_t(x) = \sum_{i=1}^{t} (\lfloor \frac{R_i(x_i)}{\delta} \rfloor + \lfloor \frac{H_i(g_i(x) - D_i)}{\delta} \rfloor)$. In the algorithm DP(ROU), we recursively compute the value $E_t(v)$, which is the maximum value of $g_t(x)$, subject to the condition that the values for the first t variables $x_1, \ldots, x_t$ are determined such that $x_i \in \mathcal{A}_i$, $i = 1, \ldots, t$, and $f_t(x) = v$. An upper bound V on the optimal solution of the rounded instance is used in the algorithm. It is determined as follows:

$$f(x^0) \leq f(x^*) \leq \lfloor F(x^*)/\delta \rfloor \leq V := \lfloor UB/\delta \rfloor.$$

A formal description of the algorithm DP(ROU) is given below.

Algorithm DP(ROU)

Step 1 (Initialization) Set $E_0(0) = 0$ and $t = 1$.

Step 2 (Recursion) For $v = 0, 1, \ldots, V$, compute the following:

$$E_t(v) = \max\{G_t(E_{t-1}(b), x_t^{(v,b)}) \mid b = 0, 1, \ldots, v\},$$

where $x_t^{(v,b)}$ is a number that satisfies $x_t^{(v,b)} \in \mathcal{A}_t$ and

$$\lfloor R_t(x_t^{(v,b)})/\delta \rfloor + \lfloor H_t(G_t(E_{t-1}(b), x_t^{(v,b)}) - D_t)/\delta \rfloor = v - b,$$

if any such number exists. If it does not exist, set $G_t(E_{t-1}(b), x_t^{(v,b)}) = -\infty$.

If $t < n$, then set $t = t + 1$ and repeat Step 2. Otherwise, go to Step 3.

Step 3 (Optimal solution) Calculate an optimal solution value

$$f(x^0) = \min\{v \mid E_n(v) > -\infty\}.$$

A corresponding optimal solution x^0 can be found by backtracking. ■

The optimality of this algorithm with respect to the rounded problem can be shown as follows. Consider partial solutions $x = (x_1, \ldots, x_t)$ in the same state (t, v) such that $f_t(x) = v$. Let an arbitrary partial solution in this state be extended to a complete feasible solution. Then a solution with maximum value of $g_t(x)$ can also be extended in the same way to a complete feasible solution with the same or smaller value of the (rounded) objective function $f(x)$. Algorithm DP(ROU) selects such a solution in each state.

A partial solution in the state (t, v) can be obtained from a partial solution in the state $(t-1, b)$, $b \le v$, by selecting an appropriate value $\alpha \in \mathcal{A}_t$ satisfying $\lfloor R_t(\alpha)/\delta \rfloor = a$, $\lfloor H_t(G_t(E_{t-1}(b), \alpha) - D_t)/\delta \rfloor = c$ and $a + c = v - b$. Such a value is denoted by $x_t^{(v,b)}$ in the algorithm. It can be calculated through an enumeration of $v - b + 1$ pairs $(a, c) = (0, v-b), (1, v-b-1), \ldots, (v-b, 0)$ as follows. Consider a fixed pair (a, c). Equations $\lfloor R_t(\alpha)/\delta \rfloor = a$ and $\lfloor H_t(G_t(E_{t-1}(b), \alpha) - D_t)/\delta \rfloor = c$ are equivalent to

$$\delta a \le R_t(\alpha) < \delta(a+1) \tag{2.62}$$

and

$$\delta c \le H_t(G_t(E_{t-1}(b), \alpha) - D_t) < \delta(c+1). \tag{2.63}$$

Since function $R_t(\alpha)$ is non-decreasing and function $H_t(G_t(E_{t-1}(b), \alpha) - D_t)$ is non-increasing in α, the system consisting of (2.62), (2.63), and $\alpha \in \mathcal{A}_t$ can be solved with respect to α in $O(\log |\mathcal{A}_t|)$ time by applying a bisection search over the set $\mathcal{A}_t$. Thus, the existence of the required $x_t^{(v,b)}$ and its value can be determined in $O((v-b)\log |\mathcal{A}_t|) \le O(V \log |\mathcal{A}_{\max}|)$ time.

It is easy to see that the time complexity of algorithm DP(ROU) is bounded by $O(V^2 nD)$, where D is the maximum time to calculate $x_t^{(v,b)}$. Thus, it is bounded by $O((UB/LB)^3 n^4 \log |\mathcal{A}_{\max}|/\varepsilon^3)$. This estimation can be decreased to $O(n^4 \log |\mathcal{A}_{\max}|(1/\varepsilon^3 + \log\log(UB/LB)))$, if we use the bound improvement procedure given by Tanaev, Kovalyov and Shafransky [54] before running DP(ROU).

The time complexity of algorithm DP(ROU) can be improved if the functions have additional properties like the one indicated below.

Assumption 2.6.6 *For each $t \in [1, n]$, functions $G_t(g_{t-1}(x), x_t)$ and $R_t(x_t)$ are concave (convex) in x_t and function $H_t(\cdot)$ is concave (convex) on $\mathbb{Z}_-$.*

If this assumption is satisfied, then one can choose $\delta = \varepsilon LB/n$ in the formulation of the corresponding rounded problem, and round down the summation of R_t and H_t. Under Assumption 2.6.6, value $x_t^{(v,b)}$ can be found in $O(\log |\mathcal{A}_t|)$ time because the function $R_t(\alpha) + H_t(G_t(E_{t-1}(b), \alpha) - D_t)$ is either convex or concave in α and we do not need to consider pairs (a, c) of values of functions R_t and H_t. The time complexity of algorithm DP(ROU) for problem Non-uniform Lot-sizing with zero holding costs under Assumption 2.6.6 reduces to $O(n^3 \log |\mathcal{A}_{\max}|(1/\varepsilon^2 + \log\log(UB/LB)))$.

2.6.5 The problem of minimizing maximum cost

In this section, we briefly discuss the *minmax* version of problem Non-uniform Lot-sizing, in which the objective is to minimize

$$M(x) = \max_{1 \le t \le n} \{R_t(x_t) + H_t(J_t)\}.$$

The minmax problem has a practical interest if a maximum budget in each time period is critical. Let Assumptions 2.6.1-2.6.4 be satisfied for this problem.

Denote by x^M and M^* an optimal solution of the minmax problem and its value, respectively. It is easy to see that equation $M^* = 0$ is equivalent to $F^* = 0$, where F^* is the optimal solution value for the original problem Non-uniform Lot-sizing. Therefore, by Lemma 2.6.1, $M^* = 0$ if and only if $D_t = 0$, $t \in [1, n]$. If it is not the case, then $M^* > 0$ and one can see that

$$0 < M^* \le F^* \le F(x^M) \le nM^*.$$

Let $x^{\varepsilon M}$ and M^ε be an ε-approximate solution to the minmax problem and its value, respectively. We have

$$M^\varepsilon/(1+\varepsilon) \le M^* \le F^* \le F(x^{\varepsilon M}) \le nM^\varepsilon.$$

Thus, exact and ε-approximate solutions to the minmax problem can be used to calculate good lower and upper bounds for the original problem NON-UNIFORM LOT-SIZING.

In some cases, the minmax problem can be easier than the corresponding problem NON-UNIFORM LOT-SIZING. However, it is not much easier in general. The proof of Theorem 2.6.1 can be modified to show that the minmax problem cannot be approximated in polynomial time with any polynomially computable relative error unless P=NP. In the new proof, $\max_t\{s_t\} < S$ can be assumed and the same instance $\mathcal{I}_{NU}$ can be constructed for the minmax problem.

The proof of Theorem 2.6.1 can be used to demonstrate that the considered NP-hard special cases of problem NON-UNIFORM LOT-SIZING are NP-hard for the minmax problem as well. FPTASs can be constructed for these special cases of the minmax problem. Similar ideas can be used in their development. However, better time complexities can be obtained because the value of δ will not be accumulated as it does when we sum up the rounded values in the objective function of problem NON-UNIFORM LOT-SIZING.

2.6.6 An FPTAS for the case with product losses

In this section, we consider the special case where product losses are possible during the manufacturing process. We will assume that no backlogging is allowed.

Let x_t be the amount of product which is produced in period t. The production capacity in each period t is restricted by a nonnegative integer u_t. It may happen that some of the product units that have been produced cannot be used to satisfy the demand due to detected defects. The defective units should be recycled. Units without defects are sent to a warehouse. Recycling costs as well as unit production and unit holding costs contribute to the total cost of expenses in each period. Obviously, the more product units are produced, the more units have defects. This means that recycling costs must grow as the production level x_t grows. Assume that the amount of units without defects is expressed by a given function $p_t(\cdot)$ depending on variable x_t. In our model, this function is nondecreasing and such that $p_t(\beta + 1) \leq p_t(\beta) + 1$ (i.e., production of an additional unit does not increase the number of units without defects by more than one which is most reasonable to assume). The

total cost in period t looks as follows:

$$\text{recycling cost}(x_t - p_t(x_t)) + \text{production cost}(x_t) + \text{holding cost}(J_t - D_t)$$

where J_t is the total amount of product units without defects produced by the end of period t. The inventory level of non-defective product units is equal to $J_t - D_t$ where D_t is the *total demand* by the end of period t (the sum of all demands by the end of this period). The value J_t is identified through the equation $J_t = J_{t-1} + p_t(x_t)$. In the above cost formula, recycling costs grow as the production level grows. Both recycling and production costs may be expressed by a single nondecreasing function $R_t(\cdot)$ that we sometimes call production cost function.

Let us formulate the described model as an optimization problem.

Formally, we consider the following assumptions:

Assumption 2.6.7 $\mathcal{A}_t = [0, u_t]$, $u_t > 0$, *for all* $t \in [1, n]$.

Assumption 2.6.8 $G_t(\alpha, \beta) = \alpha + p_t(\beta)$ *where* $p_t(\cdot)$ *is a nondecreasing function computable in constant time and satisfying the inequality* $p_t(\beta + 1) \leq p_t(\beta) + 1$ *and equation* $p_t(0) = 0$.

Assumption 2.6.9 *Functions* $R_t(\cdot)$ *and* $H_t(\cdot)$ *are increasing.*

Assumption 2.6.10 *All functions participating in problem formulations are integer-valued.*

We can formulate our optimization problem describing the situation with product losses as a subproblem of Non-uniform Lot-sizing without backlogging:

$$\min \quad \sum_{t=1}^{n} (R_t(x_t) + H_t(J_t - D_t)) \tag{2.64}$$

$$\text{s.t.} \qquad J_t = J_{t-1} + p_t(x_t), \qquad t \in [1, n], \tag{2.65}$$

$$J_t \geq D_t, \qquad t \in [1, n], \tag{2.66}$$

$$J_0 = 0, \tag{2.67}$$

$$x_t \in [0, u_t], \qquad t \in [1, n]. \tag{2.68}$$

Further we will refer to this problem as Lot-sizing with Product Waste.

Let $\mathcal{I}$ be an instance of the related rounded problem ROU. Given an integer s, construct a reduced rounded instance $\mathcal{I}_{t,s}$ by the instance $\mathcal{I}$ as

$$\begin{aligned}
\min \quad & \textstyle\sum_{i=1}^{t}(r_i(x_i) + h_i(J_i - D_i)) && && (2.69)\\
\text{s.t.} \quad & J_i = J_{i-1} + p_i(x_i), && i \in [1,t], && (2.70)\\
& J_i \geq D_i, && i \in [1,t], && (2.71)\\
& J_0 = 0, && && (2.72)\\
& J_t = s, && && (2.73)\\
& x_i \in [0,u_i], && i \in [1,t]. && (2.74)
\end{aligned}$$

where $r_t(\cdot)$ and $h_t(\cdot)$ are obtained from corresponding functions $R_t(\cdot)$ and $H_t(\cdot)$ by scaling with parameter δ similar to the previous sections. Let s be such that the set of feasible solutions of the instance $\mathcal{I}_{t,s}$ is not empty. Define $\phi_t(\cdot)$ at any such point s as

$$\phi_t(s) = OPT(\mathcal{I}_{t,s}).$$

The value s may only be from an interval $[D_t, U_t]$ where $U_t = \sum_{i=1}^{t} p_t(u_t)$ is maximum possible total level of product without defects. Let U be an upper bound on the optimal value of the original problem instance. Then, as well as in the previous sections, we may calculate an upper bound on $OPT(\mathcal{I})$ as $V = \lfloor U/\delta \rfloor$. Since all solutions with objective values greater than $V+1$ are not needed, we may, without loss of generality, include this upper bound into the following formula by which the value $\phi_t(s)$ may recursively be computed:

$$\phi_t(s) = \min\{\phi_{t-1}(s') + r_t(x_t) + h_t(s - D_t) | x_t \in [0,u_t], s' \in [D_{t-1}, U_{t-1}]\} \cup \{V+1\}, \quad (2.75)$$

Since the value V is an upper bound on $OPT(\mathcal{I})$, the inequality $\min\{\phi_n(s) | s \in [D_n, U_n]\} \leq V+1$ holds.

Proposition 2.6.1 *The function $\phi_t(\cdot)$ is nondecreasing on its feasible integer interval $[D_t, U_t]$.*

Proof. The proposition follows from two previously made assumptions. The first one is that $g_t(0,\ldots,0) = 0$ and the second one is Assumption 2.6.8. Consider a feasible solution

$x = (u_1, \ldots, u_t)$ of $\mathcal{I}_{t,U_t}$. Consider the component x_t. Decrease it by one. Let $s = g_t(x)$. The new solution x is feasible for $\mathcal{I}_{t,s}$. Repeat until either $x_t = 0$ or $g_t(x) - D_t = 0$. If $g_t(x) - D_t = 0$, then proceed with period $t - 1$, otherwise stop. According to the two mentioned assumptions, the value s should take all values from the interval $[D_t, U_t]$ during the described iterative process. Note that every time reducing components of the vector x we come to a new vector x with objective value not exceeding the previous one. This proves that $\phi_t(\cdot)$ is nondecreasing on $[D_t, U_t]$. ∎

By definition, $\phi_t(\cdot)$ is upper bounded by V on its feasible set $[D_t, U_t]$. In the recursive formula 2.75, we do not need to calculate functions $r_t(\cdot)$ and $h_t(\cdot)$ at points where their values exceed V. Thus we may accept the following assumption provided that u_t and U_t are redefined if necessary.

Assumption 2.6.11 *$r_t(x_t) \leq V$ for all $x_t \in [0, u_t]$ and $h_t(s - D_t) \leq V$ for all $s \in [D_t, U_t]$.*

The bounds u_t and U_t may be redefined in polynomial time using bisection search relying on monotony of functions $r_t(\cdot)$ and $h_t(\cdot)$.

Consider some nondecreasing integer-valued function $\psi : [a, b] \to \mathbb{Z}_+$, where $a, b \in \mathbb{Z}$. Let $\mathfrak{P}_\psi$ be a partition of $[a, b]$ into intervals $[s_1, s_2]$ such that $\psi(\cdot)$ is constant over each of these intervals and inequalities $\psi(s_1) > \psi(s_1 - 1)$ and $\psi(s_2) < \psi(s_2 + 1)$ hold in case $s_1 - 1$ and $s_2 + 1$ belong to $[a, b]$. (In other words, $[s_1, s_2]$ is maximal by inclusion.) Obviously, there is a unique partition with such properties.

Proposition 2.6.2 *Let $\psi(\cdot)$ be computable in constant time. If $\psi(\cdot)$ is nondecreasing and upper bounded by B on $[a, b]$, then $|\mathfrak{P}_\psi| \leq O(B)$ and partition $\mathfrak{P}_\psi$ can be found in $O(B \log(b - a))$ time.*

Proof. Set $s_1 = a$. Let $\psi(s_1) = v$. Using the fact that $\psi(\cdot)$ is nondecreasing, find $s_2 = \max\{s \in [s_1, b] | \psi(s) = v\}$. This can be done in $O(\log(b - a))$ time by means of binary search. Add the pair of end points (s_1, s_2) of the interval $[s_1, s_2]$ to a list and proceed with $s_1 = s_2 + 1$. Note that we can also store a corresponding value of function $\psi(\cdot)$ with every item of the list. When s_1 becomes greater than b, the list represents $\mathfrak{P}_\psi$. Since $\psi(\cdot)$ is nonnegative,

nondecreasing, and integer-valued, we have $|\mathfrak{P}_\psi| \leq B + 1$ which means that we need only $O(B)$ iterations to find $\mathfrak{P}_\psi$. ∎

Further, if we say that a partition $\mathfrak{P}_\psi$ is given, this means that $\mathfrak{P}_\psi$ is presented by a list where each item corresponds to an interval $[s_1, s_2] \in \mathfrak{P}_\psi$ and a related value of function $\psi(\cdot)$. In our consideration, the order of items in such lists does not play any role and therefore we assume that we access their items in consecutive order. This means that every item can be taken from a list in constant time.

Consider partition $\mathfrak{P}_{\phi_t}$. The next proposition follows immediately from the fact that $\phi_t(\cdot)$ is upper bounded by $V + 1$, integer-valued, nonnegative, and nondecreasing.

Proposition 2.6.3 $|\mathfrak{P}_{\phi_t}| \leq O(V)$.

We assume that $\mathfrak{P}_{\phi_0}$ consists of a single set $\{0\}$.

Lemma 2.6.2 *Let $\mathfrak{P}_{\phi_{t-1}}$ be given. Then $\phi_t(\cdot)$ can be computed in $O(V \log u_t)$ time at any point $s \in [D_t, U_t]$.*

Proof. To compute $\phi_t(\cdot)$ we may perform an exhaustive search in the family $\mathfrak{P}_{\phi_{t-1}}$ and take the best feasible x_t and s' in the formula 2.75. More precisely, let $[s_1, s_2] \in \mathfrak{P}_{\phi_{t-1}}$ and v be a value of the function $\phi_{t-1}(\cdot)$ over the interval $[s_1, s_2]$. Solve an instance

$$\begin{aligned} \min \quad & v + r_t(x_t) + h_t(s - D_t) \\ \text{s.t.} \quad & G_t(s', x_t) = s, \\ & s' \in [s_1, s_2], \\ & x_t \in [0, u_t], \end{aligned} \tag{2.76}$$

as follows. Note that due to our assumptions

$$G_t(s', x_t) = s' + p_t(x_t).$$

Then the above instance can be rewritten as

$$\min \quad v + r_t(x_t) + h_t(s - D_t)$$

$$\begin{aligned} \text{s.t.} \qquad & s' + p_t(x_t) = s, \\ & s' \in [s_1, s_2], \\ & x_t \in [0, u_t], \end{aligned}$$

Since $r_t(\cdot)$ is a nondecreasing function, the optimum value of this instance is attained at a minimum feasible value $x_{\min}$ of variable x_t. The function $p_t(\cdot)$ can take all values from the interval $[0, s - s']$ as long as s' and s are feasible for functions $\phi_{t-1}(\cdot)$ and $\phi_t(\cdot)$, respectively. Taking additionally into account that $p_t(\cdot)$ is nondecreasing, we may write

$$x_{\min} = \min\{\alpha \in [0, u_t] | p_t(\alpha) = s - s_2\}.$$

Using monotony of $p_t(\cdot)$, this value can be found in $O(\log u_t)$ time by binary search. To find $\phi_t(s)$, we have to apply the described procedure for every interval $[s_1, s_2]$ from partition $\mathfrak{P}_{\phi_{t-1}}$ and choose the optimum value of the instance (2.76). This gives a time complexity estimation $O(V \log u_t)$ to compute the function value $\phi_t(s)$ at any feasible point s. ∎

Theorem 2.6.4 *Let $\mathfrak{P}_{\phi_{t-1}}$ be given. Then $\mathfrak{P}_{\phi_t}$ can be found in*

$$O(V^2 \log(U_t - D_t) \log u_t)$$

time.

Proof. By Proposition 2.6.3, the cardinality of $\mathfrak{P}_{\phi_t}$ is $O(V)$. Proposition 2.6.2 has been proven under the assumption that function $\psi(\cdot)$ is computable in constant time. Changing the constant time of function computation by the estimation $O(V \log u_t)$ from Lemma 2.6.2 and applying Proposition 2.6.2 to function $\phi_t(\cdot)$, we may conclude that partition $\mathfrak{P}_{\phi_t}$ can be found in $O(V^2 \log(U_t - D_t) \log u_t)$ time. ∎

Summarizing our discussion, we may write an algorithm to find an optimal solution of instance $\mathcal{I}$:

Algorithm 2.6.1

Step 1 *Construct* $\mathfrak{P}_{\phi_1}, \ldots, \mathfrak{P}_{\phi_{n-1}}$;

Let $\mathfrak{P}_{\phi_0} = \{\{0\}\}$;

Step 2 $s := 0$;

for $t = n$ **down to** 1 **do**

Select $[s_1, s_2] \in \mathfrak{P}_{\phi_{t-1}}$ *so that the optimum value of instance (2.76) is minimum;*

$s := s_2$;

$x^0 := \min\{\alpha \in [0, u_t] | p_t(\alpha) = s - s_2\}$;

end do

This algorithm runs in

$$O\left(\sum_{t=1}^{n} V^2 \log(U_t - D_t) \log u_t\right)$$

time. Since $V = \lfloor \frac{U}{\delta} \rfloor$, we have an algorithm finding a $\varepsilon-$approximate solution for every instance of the problem Lot-sizing with Product Waste in time bounded by a polynomial of the instance size, $1/\varepsilon$, and U/L. Applying the Bound improvement procedure implies the following theorem.

Theorem 2.6.5 *There exists a fully polynomial time approximation scheme for* Lot-sizing with Product Waste.

2.6.7 Conclusions

In the present subsection we have introduced and studied a single-item capacitated economic lot-sizing problem with a single discrete non-uniform resource. This problem, denoted as Non-uniform Lot-sizing, represents a generalization of the classical CELS problem. We demonstrated that the general problem Non-uniform Lot-sizing and several of its special cases cannot be approximated with any polynomially computable relative error in polynomial time unless P=NP. We further derived a pseudopolynomial dynamic programming algorithm for the general problem and suggested its modification that in some cases provides an approximate solution with a given absolute error. We described FPTASs for some NP-hard special cases of the problem.

Further research can be undertaken to improve the time complexities of the presented approximation algorithms and to find other important NP-hard special cases of problem

Non-uniform Lot-sizing, to which approximation techniques presented in this paper and in the paper of van Hoesel and Wagelmans [57] can be applied.

Chapter 3

Multi-machine scheduling

3.1 Preliminary notes

Machine scheduling deals with only those scheduling problems where machines are allowed to process at most one job at a time, unlike project scheduling problems where several jobs (projects) may be processed simultaneously depending on resource availability.

To formulate a machine scheduling problem one needs to answer three questions:

- What kind of machine environment has the problem?
- In what kind of job environment should jobs be scheduled?
- Which objective function should be minimized?

A machine environment provides information about machines. It may describe the amount of machines, their types, etc. A job environment is the data describing jobs. By default, any job has to be processed non-preemptively (without interruptions). In this case the difference between the completion time of the job and its start time equals the job processing time.

Graham, Lawler, Lenstra, and Rinnooy Kan [31] suggested a three-field notation $\alpha|\beta|\gamma$ in order to classify machine scheduling problems briefly where field α corresponds to a machine environment, β is related to a job environment, and γ describes an objective function. If $\alpha = P$, then we deal with parallel identical machines, and the number of machines is considered as a part of the input data. If $\alpha = Pm$, then the number of machines is equal

to a constant m. In this case, the number of machines is considered not to be a part of an input.

In many practical situations, a job should pass several stages with parallel machines at each of them, and the order of stages is the same for every job. This means that we may enumerate stages as $1, \ldots, L$ so that a job should visit all stages in increasing order of index before it is complete. (At every stage a job visits a machine only once. A job may have different processing times for every stage.) Manufacturing systems with such properties sometimes are called flexible flow shops and corresponding scheduling problems are called flexible flow shop problems. Obviously, they are generalizations of standard flow shop problems where a single machine is available at every stage. (For detailed review of algorithmic approaches for both flow shop and flexible flow shop problems we refer to the book by Błażewicz et al. [9] and to Kis and Pesch [38].)

For single-stage problems we use notation M_k to denote a machine with index k. For multi-stage problems we also add a stage number to index a machine. Sometimes we will omit letter 'M' referring to machines. For instance, we may say "consider a machine k" which will mean "consider a machine M_k". Analogously, we sometimes refer to jobs using only their indices.

Any schedule can be completely characterized by completion times of the jobs and job assignments to machines. We use only a vector of completion times to denote a schedule. With every schedule $C = (C_1, \ldots, C_n)$ we associate a function $A_C : [1, n] \to [1, m]$ whose value $A_C(j)$ is a number of a machine executing job J_j in schedule C. If machines are identical, then a vector of completion times allows to restore a schedule within a permutation of machines in polynomial time. In other cases, information about machine assignment is more important.

To illustrate our discussion, let us observe scheduling problem $P2||\sum C_j$ where jobs $J_1, \ldots, J_n$ should be scheduled on two parallel machines M_1 and M_2 subject to minimizing the total completion time, i.e., the sum $\sum_{j=1}^{n} C_j$ of completion times C_j. An example of a feasible schedule for an instance of problem $P2||\sum C_j$ with six jobs is given in the form of a Gantt chart in Figure 3.1. There, each machine M_1 and M_2 processes jobs non-preemptively

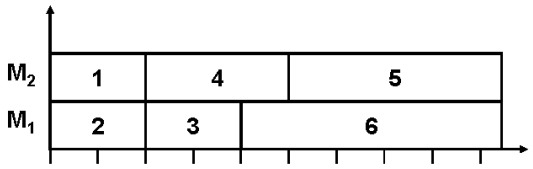

Figure 3.1: An example of a feasible schedule.

in increasing order of their indices. At the same time this is a nondecreasing order of job processing times. The schedule in Figure 3.1 follows therefore a so-called Smith's rule known also under the name of shortest processing time (SPT) rule by which every machine should process jobs in nondecreasing order of processing times. For a single machine, this rule yields an optimal job sequence if the objective is to minimize total completion time $\sum_{j=1}^{n} C_j$, or equivalently average completion time $1/n \sum_{j=1}^{n} C_j$. Hence, for any instance of problem $P2||\sum C_j$ there is an optimal schedule where each machine follows the SPT rule. This means that, for this problem, without loss of an optimal schedule, we may number jobs so that $p_1 \leq \ldots \leq p_n$ and assume that each machine should process jobs in increasing order of their index.

3.2 Local search in large-scale neighborhoods

3.2.1 Introduction

We will consider local search procedures in large scale neighborhoods for the scheduling problem $P|r_j, s_{ij}|\sum C_j$ where s_{ij} are setup times arising if jobs J_i and J_j are executed by the same machine one after another. In a feasible schedule C all job start times should be greater than or equal to corresponding release dates r_j. No preemptions are allowed. The

objective is to find a feasible schedule with minimum total completion time $\sum_{j=1}^{n} C_j$. The described problem is known to be strongly NP-hard.

In our procedures, the basic transformations of schedules are so-called swap, insertion, and deletion moves. A swap move interchanges two jobs in a schedule. For instance, if jobs J_i and J_j are processed by machines M_l and M_k, respectively, then after applying a swap move J_j is replaced by J_i on the machine M_k and J_i is replaced by J_j on the machine M_l. A swap move consists of two insertion moves each of which places a job in an appropriate position on a corresponding machine. A deletion move removes a job from a schedule. It is obvious that any swap move can be accomplished by two deletion and two insertion moves. We will suppose that every move is concluded by shifting jobs as early as possible, without changing their order or machine assignments.

Every transformation of a schedule may change the objective value. Let C' be a schedule obtained from a schedule C by means of applying a swap move to jobs J_i and J_j. We call c_{ij} the swap cost of jobs J_i and J_j if c_{ij} is a difference between objective values of schedules C' and C. If c_{ij} is negative, then this swap move improves schedule C.

Local search procedures for scheduling problems go from schedule to schedule exploring a *neighborhood* of the current schedule at every iteration. A neighborhood of a schedule is a set of schedules that can be obtained from this schedule by a sequence of moves following certain rules. An example of a neighborhood of some schedule C is where all schedules may be obtained from C by a single swap move. Provided that jobs processed by the same machine are also allowed to be swapped, this neighborhood contains $n(n-1)/2+1$ entries (i.e, $n(n-1)/2$ swap variants plus schedule C).

Using only single swap and insertion moves in order to obtain improved schedules is not effective since only small neighborhoods (of polynomial size) are searched. Polynomial search procedures in large-scale neighborhoods of exponential size usually yield much better results. There are different techniques allowing to find reasonably good solutions in large-scale neighborhoods. They can be divided into two big classes. Search procedures of the first class find locally optimal solutions, and those of the second class search neighborhoods approximately. For our problem, we will consider procedures of both classes.

In conclusion, we will show on a real-life instance that the local search procedures for single-stage scheduling problems are also able to essentially improve schedules for flexible flow shop problems.

3.2.2 Exact and approximate search in exponential neighborhoods

Matching neighborhood

Let c_{ij} be the cost of the swap of jobs J_i and J_j in schedule C. Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be disjoint subsets of machine indices. Let for any $l \in \mathcal{S}_1$ and $k \in \mathcal{S}_2$ a value w_{lk} be a minimum among costs c_{ij} such that $A_C(i) = l$ and $A_C(j) = k$. Denote by $(i(l,k), j(l,k))$ a pair of job indices delivering such minimum. In other words, for every pair of machines we choose the most profitable swap of jobs. Assume that $\mathcal{P}$ is a subset of $\mathcal{S}_1 \times \mathcal{S}_2$ such that any two different pairs $(l_1, k_1) \in \mathcal{P}$ and $(l_2, k_2) \in \mathcal{P}$ satisfy $\{l_1, k_1\} \cap \{l_2, k_2\} = \emptyset$. Apply swaps of each job pair $(J_{i(l,k)}, J_{j(l,k)})$ for any pair of machines (M_l, M_k) where $(l, k) \in \mathcal{P}$. The total change in the current cost of the schedule C is calculated as $\sum_{(l,k)\in\mathcal{P}} w_{lk}$. Let $C' = (\mathcal{P}; C)$ denote a schedule obtained from the schedule C by applying corresponding swaps. If $\sum_{j=1} C'_j - \sum_{j=1} C_j < 0$, then we came to a better schedule after our simultaneous swap moves. Denote the set of all such schedules $(\mathcal{P}; C)$ by $\texttt{matching}(C)$. We call the move from C to a schedule $C' \in \texttt{matching}(C)$ a parallel swap move. We use such name due to the fact that one may perform independent swaps to arrive C'. To find the best parallel swap move, we can find a matching of minimum (negative) weight in a weighted graph $G = (\mathcal{N}(G), \mathcal{E}(G))$ where $\mathcal{N}(G) = [1, m]$ is the set of machine numbers and the set of edges $\mathcal{E}(G) = \{\{l, k\} | w_{lk} < 0\}$ corresponds to all improving moves.

The problem of finding a minimum cost matching in graph G can be formulated as the following integer linear program.

$$\min \quad \sum_{e\in\mathcal{E}(G)} w_e x_e \tag{3.1}$$

$$\text{s.t.} \quad \sum_{e\in\mathcal{E}(G), e\cap\{v\}\neq\emptyset} x_e \leq 1 \quad \forall v \in \mathcal{N}(G) \tag{3.2}$$

$$x_e \in \{0, 1\} \qquad \forall e \in \mathcal{E}(G) \tag{3.3}$$

Unlike some formulations having exponential size, in this one we may not omit integrality conditions. Running CPLEX on this formulation, in reasonable time we can find matchings with costs differing from the optimal one by at most 0.1%. (Note that CPLEX might need long time to solve the above integer linear program exactly and this is the reason that we solve the problem approximately.)

The neighborhood $\texttt{matching}(C)$ has an exponential size. One can see that

$$|\texttt{matching}(C)| \geq 2^{\lfloor m/2 \rfloor}$$

for any schedule C.

Proposition 3.2.1 *For any schedule C a local optimum in the neighborhood* $\texttt{matching}(C)$ *can be found in polynomial time.*

Proof. The proposition is implied by the fact that the problem (3.1)-(3.3) can be rewritten as a maximum weight matching problem if we reverse the sign in the objective function and replace "min" by "max". A matching of maximum weight can be found for instance by the algorithm developed by Edmonds [22] in time polynomial of the instance size. ∎

Assignment neighborhood

Consider a schedule C. For every machine M_k, select a job j_k among jobs processed by this machine so that the total completion time of jobs processed by M_k would be as small as possible after deleting this job. We define a neighborhood $\texttt{assignment}(C; j_1, \ldots, j_m)$ as a set of all schedules that can be obtained from C by interchanging jobs belonging to the selected job set $\{j_1, \ldots, j_m\}$. In any of these schedules, a machine processes exactly one of the jobs $j_1, \ldots, j_m$. Formally,

$$\{A_{C'}(j_1), \ldots, A_{C'}(j_m)\} = [1, m]$$

for any $C' \in \texttt{assignment}(C, j_1, \ldots, j_m)$. Every schedule in this neighborhood can be constructed by deleting jobs $j_1, \ldots, j_m$ and reassigning machines processing them one per every job. Thus, to find an optimal schedule in this neighborhood, one has to solve an assignment

problem. This is the reason why we call the described neighborhood assignment neighborhood.

A formulation of the related optimization problem reads as follows:

$$\begin{array}{ll} \min & \sum_j C'_j \\ \text{s.t.} & C' \in \texttt{assignment}(C, j_1, \ldots, j_m) \end{array}$$

We will call this problem Assignment Neighborhood.

Let w_{kl} be the lowest cost of insertion of job j_k into machine M_l. To calculate this value, we may try to insert this job into the machine before or after every job that is to be processed by machine M_l in schedule C, each time evaluating the value $\sum_{j=1}^{n} C'_j - \sum_{j=1}^{n} C_j$ where C' denotes the schedule obtained from C by the insertion of job j_k into the new place. Then we select the insertion variant where $\sum_{j=1}^{n} C'_j - \sum_{j=1}^{n} C_j$ is minimum. We will find so far the most profitable place into which job j_k can be inserted provided that this job is required to be processed by machine M_l. If $\sum_{j=1}^{n} C'_j - \sum_{j=1}^{n} C_j$ is negative, then a corresponding insertion improves the schedule. The problem Assignment Neighborhood boils down to an assignment problem of the form

$$\begin{array}{lll} \min & \sum_{(k,l)} w_{kl} x_{kl} & \\ \text{s.t.} & \sum_{k=1}^{m} x_{kl} = 1, & \forall l \in [1, m], \\ & \sum_{l=1}^{m} x_{kl} = 1, & \forall k \in [1, m], \\ & x_{kl} \in \{0, 1\}, & \forall k, l \in [1, m]. \end{array}$$

Every component x_{kl} taking value 1 in a feasible solution x of this problem corresponds to assignment of job j_k to machine l. An optimal solution yields an optimal reassignment of jobs $j_1, \ldots, j_m$ to machines. Therefore, Assignment Neighborhood reduces to the above assignment problem which is known to be polynomially solvable.

Let us show that it may happen that there is a schedule C that is a local optimum in the matching neighborhood, but, at the same time, it is not optimal in the assignment neighborhood.

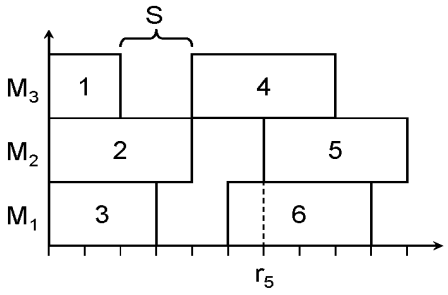

Figure 3.2:

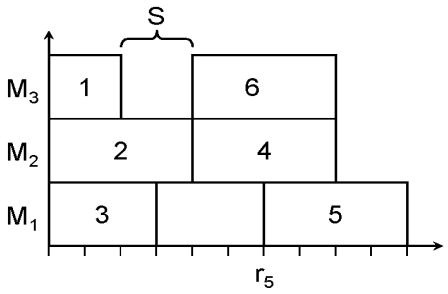

Figure 3.3:

Consider six jobs with processing times $(p_1, \ldots, p_6) = (2, 4, 3, 4, 4, 4)$. Assume that all setup times, except s_{24}, are equal to some positive value S. Let $r_5 = 6$ and all the other job release dates are equal to zero. Suppose that there are three machines. Construct a schedule C^0 as indicated in Figure 3.2. Any pairwise interchange of jobs does not improve the schedule. I.e., schedule C^0 is optimal in the matching neighborhood. At the same time, if we reassigned jobs J_4, J_5, and J_6 to machines M_2, M_1, and M_3, respectively, then we would obtain a better schedule which is shown in Figure 3.3. Since jobs J_4, J_5, and J_6 should be selected to be reassigned due to the definition of the assignment neighborhood, this schedule belongs to $\texttt{assignment}(C^0; 4, 5, 6)$. Note that J_4, J_5, and J_6 are exactly those jobs after deleting of which we obtain a partial schedule with the smallest objective value provided that we are allowed to delete only one job at each machine.

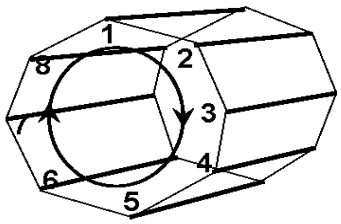

Figure 3.4: Jobs may move only to the next machines in clockwise order.

Rotation neighborhood

To simplify the exposition, consider successor and predecessor functions *succ* and *pred* defined on the set $[1, m]$ such that $succ(t)$ is equal to $t + 1$, if $t < m$, and to 1, otherwise, and $pred(t)$ is equal to $t - 1$, if $t > 1$, and to m, otherwise.

Let C be a feasible schedule. Choose some subset of machines $\mathcal{S} \subseteq [1, m]$. Let C' be a schedule where some job is removed from each machine belonging to $\mathcal{S}$ and reassigned onto the next machine. I.e., job j removed from machine $A_C(j)$ should be reassigned to machine $succ(A_C(j))$ in the new schedule. As one can see, if $A_C(j) = m$, then J_j is reassigned onto machine M_1. If we imagine machines as parallel lines forming a prism in three dimensions as shown in Figure 3.4 and if we enumerate them in clockwise order, then any schedule belonging to the neighborhood we have defined may only be obtained by moves of the chosen jobs to next machines in clockwise order. There are eight machines in the picture. Due to this geometric analogy we call this neighborhood rotation neighborhood and denote it as `rotation`(C) for every feasible schedule C. The related optimization problem is called Rotation Neighborhood.

Provided that every machine processes some job, for each machine we may decide if some job leaves this machine or not. The number of such variants is 2^m, each providing a schedule belonging to the rotation neighborhood. Hence, there are at least 2^m schedules in

this neighborhood.

There is however a polynomial algorithm allowing to find an optimal schedule for any instance $\mathcal{I}$ of ROTATION NEIGHBORHOOD. Let a reduced instance $\mathcal{I}^h_{t,i,j}$, where $h, i, j \in [0, n]$ and $t \in [1, m]$, be formulated by instance $\mathcal{I}$ as

$$\min \quad \sum_{l\in[1,n], A_{C'}(l)\le t} C'_l \tag{3.4}$$

$$\text{s.t.} \quad C' \in \texttt{rotation}(C), \tag{3.5}$$

$$A_{C'}(i) = t = succ(A_C(i)), \qquad i > 0, \tag{3.6}$$

$$A_C(j) = t = pred(A_{C'}(j)), \qquad j > 0, \tag{3.7}$$

$$\left[\begin{array}{ll} A_{C'}(h) = 1, & h > 0, \\ A_C(l) = A_{C'}(l)\ \forall l, A_C(l) = m, & h = 0. \end{array}\right. \tag{3.8}$$

In this instance, only the first t machines are considered in the objective function. In the above formulation, i and j may be zero. If $i = 0$, then only those schedules are considered which do not require moving jobs from machine $pred(t)$ to machine t. If $j = 0$, then jobs being processed by machine t in schedule C must not leave this machine. A positive value of an index i or j, due to restrictions (3.6) and (3.7), indicates that a corresponding job must be moved to the next machine. The constraint (3.8) says that no job may leave machine m if $h = 0$, and job h must move to machine 1, otherwise. This constraint is needed to compute a recursive function that we introduce below. We can solve instance $\mathcal{I}$ by means of solving instances $\mathcal{I}^h_{t,i,j}$ since

$$OPT(\mathcal{I}) = \min_{h,i,j} OPT(\mathcal{I}^h_{m,i,j}).$$

This equation is followed by the fact that domains of instances $\mathcal{I}^h_{m,i,j}$, $h, i, j \in [1, n]$ cover the domain of instance $\mathcal{I}$.

Let $\phi^h_t(i, j) = OPT(\mathcal{I}^h_{t,i,j})$. We assume that $\phi^h_t(i, j) = \infty$ if the domain of the corresponding reduced instance is empty. It is clear that $\phi^h_1(i, j) = \infty$ if and only if $i \neq h$ and $\phi^h_m(i, j) = \infty$ if and only if $j \neq h$.

Recursive computation of function ϕ can be accomplished using the formulas

$$\phi^h_1(h, j) = c_{1hj},$$

$$\phi_1^h(i,j) = \infty \ \ \forall i \neq h,$$

and

$$\phi_t^h(i,j) = c_{tij} + \min_{l\in[0,n]}\{\phi_{pred(t)}^h(l,i)\},$$

where c_{tij} is the minimum total completion time of jobs processed on machine M_t provided that job i processed on machine $pred(t)$ is reassigned onto machine t and job j is reassigned onto machine $succ(t)$. (If $i = 0$, then no job leaves machine $pred(t)$, and if $j = 0$, then no job leaves machine t.) Let us define coefficients c_{tij} precisely. Let $i_1, \ldots, i_s$ be jobs at machine t sequenced as

$$i_1 \to i_2 \to \ldots \to i_s.$$

(Note that j has been moved to the next machine and thus is not among jobs $i_1, \ldots, i_s$.) Insert job i such that job i becomes lth job in this sequence. The new sequence has the form

$$i_1^l \to \ldots \to i_l^l \to \ldots \to i_{s+1}^l.$$

where $i_l^l = i$, $i_k^l = i_k$ for all $k < l$, and $i_{k+1}^l = i_k$ for all $k > l$. There are $s+1$ jobs in the new sequence. Taking the minimum over all possible variants of insertions of job i we have

$$c_{tij} = \min_{l\in[1,s+1]}\left\{\sum_{k=1}^{s+1} C'_{i_k^l}\right\}$$

where

$$C'_{i_1^l} = r_{i_1^l} + p_{i_1^l}$$

and

$$C'_{i_k^l} = \max\{r_{i_k^l}, C'_{i_{k-1}^l} + s_{i_{k-1}^l, i_k^l}\} + p_{i_k^l}, \ \forall k \in [2, s+1].$$

If $i = 0$, then no job is inserted to the above sequence and therefore c_{tij} is the sum of completion times of jobs $i_1, \ldots, i_s$ under the condition that job j (if $j \neq 0$) has been moved to the next machine. If machine $pred(t)$ does not process job i in schedule C or machine t does not process j, then value c_{tij} is supposed to be equal to ∞.

To compute the function ϕ recursively, we may use equations

$$\phi_t^h(i,j) = \phi_t^h(i,0) - c_{ti0} + c_{tij} \tag{3.9}$$

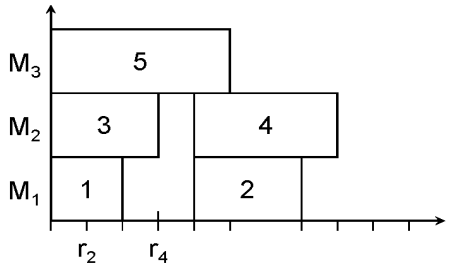

Figure 3.5: The example with five jobs.

which must hold for every job pair $i, j \in [1, n]$ satisfying $A_C(i) = pred(t)$ and $A_C(j) = t$. The equations (3.9) lead to time complexity $O(T)$ where T is the total number of feasible triples (t, i, j) (i.e., those for which $\phi_t^h(i, j) \neq \infty$.) This is best possible if we want to calculate function ϕ_t^h at every feasible point. A corresponding schedule in `rotation`(C) can be found by a standard backtracking procedure.

Consider an example with five jobs. Let job processing times and release dates be presented in the following table:

job	1	2	3	4	5
processing time	2	3	3	4	5
release date	0	1	0	3	0

Let $s_{12} = 2$, $s_{34} = 1$, and the other setup times are zero. Denote by C a schedule depicted in Figure 3.5. The total completion time of jobs in this schedule is equal to 26.

Three tables below provide the information about how coefficients c_{tij} are calculated.

c_{1ij}

	0	1	2
0	9	4	2
5	19	13	9

c_{2ij}

	0	3	4
0	11	7	3
1	17	9	7
2	19	12	9

c_{3ij}

	0	5
0	5	0
3	11	3
4	14	7

We explain how some of these coefficients are computed. Provided that job J_1 moves to machine M_2 and job J_3 leaves this machine, it is optimal to insert job J_1 before job J_4.

Therefore,

$$c_{2,1,3} = r_1 + p_1 + \max\{r_4, r_1 + p_1\} + p_4 = 9$$

Similarly we obtain

$$c_{3,3,0} = r_3 + p_3 + \max\{r_5, r_3 + p_3\} = 11.$$

(Recall that, calculating coefficients c_{tij}, we may only select the place where i should be inserted in, without changing the order of jobs staying on machine M_t.)

The next tables contain values of function ϕ.

$\phi_1^0(i,j)$

	0	1	2
0	9	4	2

$\phi_2^0(i,j)$

	0	3	4
0	20	16	12
1	21	13	11
2	21	14	11

$\phi_3^0(i,j)$

	0
0	25
3	24
4	25

$\phi_1^5(i,j)$

	0	1	2
5	19	13	9

$\phi_2^5(i,j)$

	0	3	4
0	30	26	22
1	30	22	20
2	28	21	18

$\phi_3^5(i,j)$

	5
0	28
3	24
4	25

The minimum value of function ϕ_m^h is equal to $\phi_3^0(3,0) = 24$. Let us describe in more detail the calculation of function values necessary to calculate $\phi_3^0(3,0)$.

$$\phi_1^0(0,0) = c_{1,0,0} = 9$$

$$\phi_1^0(0,1) = c_{1,0,1} = 4$$

$$\phi_1^0(0,2) = c_{1,0,2} = 2$$

$$\phi_2^0(0,3) = c_{2,0,3} + \phi_1^0(0,0) = 16$$

$$\phi_2^0(1,3) = c_{2,1,3} + \phi_1^0(0,1) = 13$$

$$\phi_2^0(2,3) = c_{2,2,3} + \phi_1^0(0,2) = 14$$

$$\phi_3^0(3,0) = c_{3,3,0} + \min\{\phi_2^0(0,3), \phi_2^0(1,3), \phi_2^0(2,3)\} = 24$$

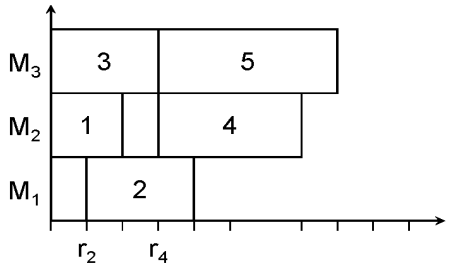

Figure 3.6: A locally optimal schedule.

Following these formulas, to obtain a corresponding schedule, we must move job J_3 to machine M_3 and job J_1 to machine M_2. This schedule is shown in Figure 3.6. It is a local optimum in `rotation`(C).

Dynamic neighborhood

It is often the case that a problem of finding locally optimal solutions in some kind of neighborhoods, like one we will consider, is NP-hard. Usually, we do not really need to know an optimal solution in some neighborhood, but some approximate solution with guaranteed quality. One of the most recent theoretical aspects of approximate local search have been devised by Orlin, Punnen, and Schulz [44].

A neighborhood that we will consider below, and which we call dynamic neighborhood, can be searched exactly in pseudopolynomial time by a dynamic programming algorithm. We will show that such an algorithm can be converted into a fully polynomial time approximation scheme.

Note that if processing times are relatively small, say, less than the number of jobs, it makes sense to find an exact local optimum since the running time of a dynamic programming algorithm will be polynomial in this case. Otherwise, one can search the neighborhood approximately with a guaranteed precision using a fully polynomial time approximation scheme.

Consider a feasible schedule C. Let $\mathcal{J}_k = \{J_{j_1}, \ldots, J_{j_\tau}\}$ be all jobs processed by some

machine M_k, $k \in [1, m]$. We will select M_k such that the total completion time of the job sequence processed by this machine is maximum among the total completion times of those processed by other machines. Let w_{lj} be a minimum cost of swapping some job processed by machine M_l, $l \neq k$, and job J_j processed by machine M_k. In other words,

$$w_{lj} = \min\{c_{ij} | A_C(i) = l\}$$

where c_{ij} is a swap cost of corresponding jobs. Consider the following minimum weight matching problem on a complete bipartite graph:

$$\begin{array}{lll} \min & \sum_{l,j} w_{lj} x_{lj} & \\ \text{s.t.} & \sum_{l \in [1,m] \setminus \{k\}} \sum_{j \in \mathcal{J}_k} x_{lj} = \min\{m, \tau\}, & \\ & \sum_{j \in \mathcal{J}_k} x_{lj} \leq 1, & \forall l \in [1, m] \setminus \{k\}, \\ & x_{lj} \in \{0, 1\}, & \forall l \in [1, m] \setminus \{k\}, j \in \mathcal{J}_k. \end{array}$$

Let x^0 be an optimal solution to this problem instance. Define a set

$$\mathcal{L} = \{J_{l_1}, \ldots, J_{l_\tau}\}$$

where $l_t = l$ if there exists an l such that $x^0_{l,j_t} = 1$, and $l_t = 0$, otherwise. In the latter case $J_{l_t} = J_0$ where notation J_0 will further be used to indicate that there is no alternative for job J_{j_t}. If $l_t > 0$, then the corresponding job J_{l_t} will be considered as an alternative job for job J_{j_t}, i.e., jobs J_{l_t} and J_{j_t} will be allowed to be swapped. Denote by $\texttt{dynamic}(C; \mathcal{J}_k, \mathcal{L})$ the neighborhood consisting of all possible schedules each of which is obtained from C by a swap of jobs J_{j_t} and J_{l_t} where t belongs to a subset of the set $[1, \tau]$.

For the previously considered matching neighborhood, a schedule is a local optimum if and only if there are no single swaps decreasing the objective value. This criterion is not true for the dynamic neighborhood. Let us consider an example to prove this statement. Suppose that there are four jobs J_1, J_2, J_3, and J_4. Let all release dates be zero and job processing times are defined as $p_1 = p_2 = p_4 = 4$ and $p_3 = 3$. Assume that $s_{12} = 0$ and the other setup times are equal to 2. The number of machines in our example equals 3. Schedule jobs as it is shown in Figure 3.7.

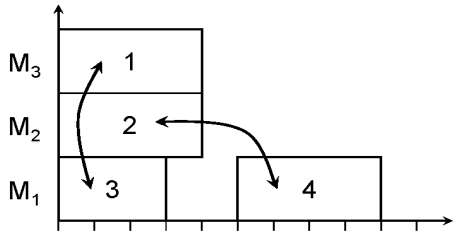

Figure 3.7: Applying both the swaps marked in the picture with arrows leads to a better schedule.

In this figure, jobs J_1 and J_2 are assigned to machines M_3 and M_2, respectively. The remaining jobs J_3 and J_4 are assigned to machine M_1 one after another. The time interval between them is needed to setup machine M_1 to process job J_4 after processing job J_3. It is clear from the picture that there is no single move improving the objective value of the schedule. This means that this schedule is locally optimal in matching and assignment neighborhoods. If we swap jobs J_1 and J_3 and then jobs J_2 and J_4, then we obtain a schedule with a better objective value since the setup time between jobs J_1 and J_2 is equal to zero. This schedule is in a dynamic neighborhood where machine M_1 is selected, and jobs J_1 and J_2 are chosen as alternative jobs to jobs J_3 and J_4, respectively.

Consider an optimization problem related to a dynamic neighborhood:

$$\begin{aligned} \min \quad & F(C') = \textstyle\sum_{j=1}^{n} C'_j \\ \text{s.t.} \quad & C' \in \texttt{dynamic}(C; \mathcal{J}_k, \mathcal{L}) \end{aligned}$$

We will refer to this problem as to Dynamic Neighborhood. An optimal schedule to an instance $\mathcal{I}^{\#}$ of this problem is an optimal schedule in a neighborhood $\texttt{dynamic}(C; \mathcal{J}_k, \mathcal{L})$. The objective value of this locally optimal schedule is equal to $OPT(\mathcal{I}^{\#})$. To solve instance $\mathcal{I}^{\#}$ approximately with a relative error $\varepsilon > 0$, we use scaling with a parameter $\delta = \varepsilon L/n$ where L is a positive lower bound on $OPT(\mathcal{I}^{\#})$. In the instance $\mathcal{I}^{\#}$, replace the objective

function $F(\cdot)$ by a function $f(\cdot)$ defined as

$$f(C') = \sum_{j=1}^{n} \left\lfloor \frac{C'_j}{\delta} \right\rfloor$$

for any schedule C'. Denote the obtained instance by $\mathcal{I}$.

Standard arguments yield the following theorem.

Theorem 3.2.1 *If C' is an optimal solution of $\mathcal{I}$, then C' is an $\varepsilon-$approximate solution of $\mathcal{I}^{\#}$.*

Assume that the jobs on machine M_k are processed in the order

$$J_{j_1} \to \ldots \to J_{j_\tau}.$$

Let $t \in [1, \tau]$ and $j \in \{j_t, l_t\}$, $j \neq 0$. Construct a reduced instance $\mathcal{I}_{t,j,d}$ as

$$\begin{aligned} \min \quad & \textstyle\sum_{i \in \mathcal{S}_t} \lfloor \frac{C'_i}{\delta} \rfloor \\ \text{s.t.} \quad & C' \in \texttt{dynamic}(C; \mathcal{J}_k^t, \mathcal{L}^t), \\ & C'_j \leq d, \\ & A_{C'}(j) = k. \end{aligned} \tag{3.10}$$

where d is a nonnegative integer, set $\mathcal{S}_t$ is defined as

$$\mathcal{S}_t = \{j_1, \ldots, j_t\} \cup \{\text{indices of all jobs on machines } A_C(j_{l_1}), \ldots, A_C(j_{l_t})\},$$

and $\mathcal{J}_k^t = \{J_{j_1}, \ldots, J_{j_t}\}$ and $\mathcal{L}^t = \{J_{l_1}, \ldots, J_{l_t}\}$.

One can see that

$$\min_{j \in \{l_\tau, j_\tau\}} OPT(\mathcal{I}_{t,j,d}) + \sum_{j \in [1,n] \setminus \mathcal{S}_t} \left\lfloor \frac{C_j}{\delta} \right\rfloor = OPT(\mathcal{I})$$

provided that d is sufficiently large.

Note that $V = \lfloor U/\delta \rfloor$ is an upper bound on the value $OPT(\mathcal{I})$. Consider a function ϕ such that

$$\phi_{tj}(d) = \min\{OPT(\mathcal{I}_{t,j,d}), V + 1\}$$

if $j \in [1, n]$, and

$$\phi_{tj}(d) = \infty$$

if $j = 0$.

Proposition 3.2.2 *Function $\phi_{tj}(\cdot)$, $j \in [1, n]$, is nonincreasing.*

Proof. Note that increasing value d in the formulation of the reduced instance $\mathcal{I}_{t,j,d}$, we thereby enlarge the set of schedules feasible for this instance. This implies the proposition. ∎

The above proposition implies that $\phi_{tj}(\cdot)$ is a piecewise-constant function with at most $O(V)$ pieces. I.e., the feasible interval of $\phi_{tj}(\cdot)$ may be partitioned into a set of $O(V)$ intervals on each of which $\phi_{tj}(\cdot)$ is constant.

Let us take some $t \in [1, \tau]$ and $j \in \{j_t, l_t\}$. Let $d^t_{\min}$ be the minimum d such that $\mathcal{I}_{t,j,d}$ has a feasible solution. Let $d^t_{\max}$ be the maximum feasible value of argument d. We base our dynamic programming algorithm on a formula

$$\phi_{tj}(d) = \min_{d' \in [d^{t-1}_{\min}, d^{t-1}_{\max}], i \in \{j_{t-1}, l_{t-1}\}} \{\phi_{t-1,i}(d') + \lfloor (c_{tj} + \max\{d' + s_{ij}, r_j\} + p_j)/\delta \rfloor\} \qquad (3.11)$$

where the minimum is taken over all d' and i satisfying $\max\{d' + s_{ij}, r_j\} + p_j \le d$, and c_{tj} is a sum of completion times of jobs processed on machine $A_C(l_t)$ if $j > 0$ and $c_{tj} = 0$, otherwise. If $j = l_t$, then c_{tj} is calculated provided that jobs j_t and l_t has been swapped. The term $\max\{d' + s_{ij}, r_j\}$ stands for the completion time of job j in the above formula. Value d belongs to the set $[d^t_{\min}, d^t_{\max}]$. We assume that $\phi_{0j_0}(0) = 0$, $\phi_{0l_0}(0) = 0$, and $d^0_{\min} = d^0_{\max} = 0$. We also suppose that $s_{ij} = 0$ if $i \in \{j_0, l_0\}$.

The minimum value of function $\phi_{\tau j}(\cdot)$ on its feasible domain $dom(\phi_{\tau j}) = [d^t_{\min}, d^t_{\max}]$ is equal to $OPT(\mathcal{I})$.

Let $\mathfrak{P}_{tj}$ be a partition of interval $[d^t_{\min}, d^t_{\max}]$ into intervals $[d_1, d_2]$ on each of which $\phi_{tj}(\cdot)$ is constant. They are at most $O(V)$ as we have previously mentioned. We suppose that $\mathfrak{P}_{0j} = \{\{0\}\}$.

Proposition 3.2.3 *Given partitions $\mathfrak{P}_{t-1j}$, $j \in \{j_{t-1}, l_{t-1}\}$, one can calculate $\phi_{tj}(d)$ in $O(|\mathfrak{P}_{t-1j}|)$ time for any feasible d and j.*

Proof. This proposition follows from the recursive formula identifying $\phi_{tj}(d)$. ∎

Proposition 3.2.4 *The set* $\{(d^t_{\min}, d^t_{\max})|t \in [1,\tau]\}$ *can be found in* $O(\tau)$ *time.*

Proof. Simple dynamic programming arguments imply the proposition. Let $d(0,j_0) = d(0,l_0) = 0$. Define

$$d(t,j) = \min\{\max\{d(t-1,i) + s_{ij}, r_j\} + p_j | i \in \{j_{t-1}, l_{t-1}\}\}$$

for every $t \in [1,\tau]$ and $j \in \{j_t, l_t\}$. Value $d^t_{\min}$ can be derived as $d^t_{\min} = \min\{d(\tau,j_\tau), d(\tau,l_\tau)\}$. If we use the described recursion to calculate $d(\cdot,\cdot)$, this needs $O(\tau)$ time. Similar recursive equations can be used to obtain $d^t_{\max}$. ("min" should be replaced by "max" in the recursive formula.) ∎

The following algorithm solves instance $\mathcal{I}$.

Algorithm 3.2.1

Step 1 for $t = 1$ **to** τ **do**

find $d^t_{\min}$ *and* $d^t_{\max}$

end do

construct $\mathfrak{P}_{tj}$, $j \in \{j_t, l_t\}$, $t \in [1,\tau]$;

(For each interval in the partition a corresponding function value is stored.)

Step 2 $d := d^\tau_{\max}$;

$j :=$ *arg* $\min_{j\in\{j_t,l_t\}}\{\phi_{\tau j_\tau}(d), \phi_{\tau l_\tau}(d)\}$;

for $t = \tau - 1$ **down to** 1 **do**

Let minimum in formula (3.11) be attained at $d' = \hat{d}$ *and* $i = \hat{i}$;

$j = \hat{i}$;

if $j = l_t$ **then** *swap jobs* J_{j_t} *and* J_{l_t} *in schedule* C;

end do

Let C' *be the obtained schedule.*

Proposition 2.6.2 of the previous chapter, which must also hold for a nonincreasing function, implies that given $\mathfrak{P}_{t-1j}$, $j \in \{j_{t-1}, l_{t-1}\}$, we can find a partition $\mathfrak{P}_{tj}$ for any $j \in \{j_t, l_t\}$ in

$O(V \log(d^t_{\max} - d^t_{\min}))$ time. Therefore Step 1 runs in $O(nV \log(d^n_{\max} - d^n_{\min}))$ time. In the same time this is a complexity estimation of Algorithm 3.2.1. Combined with the bound improvement procedure this yields $O(\frac{n^2}{\varepsilon} \log(d^t_{\max} - d^t_{\min}))$ algorithm to solve the instance $\mathcal{I}$. Thereby we come to the following theorem.

Theorem 3.2.2 *An $\varepsilon-$approximate schedule in neighborhood* $\texttt{dynamic}(C; \mathcal{J}, \mathcal{L})$ *can be found in $O(\frac{n^2}{\varepsilon} \log(d^t_{\max} - d^t_{\min}))$ time.*

The approximation algorithm can be useful if processing times are sufficiently large.

Modify formula (3.11) so as

$$\phi_{tj}(d) = \min_{\substack{d' \in [d^{t-1}_{\min}, d^{t-1}_{\max}] \\ i \in \{j_{t-1}, l_{t-1}\}}} \{\phi_{t-1,i}(d') + c_{tj} + d | \max\{d' + s_{ij}, r_j\} + p_j = d\} \qquad (3.12)$$

Note that this modification leads to a dynamic programming algorithm whose running time depends linearly on value $d^n_{\max} - d^n_{\min}$. This implies the following theorem.

Theorem 3.2.3 *An optimal solution in neighborhood* $\texttt{dynamic}(C; \mathcal{J}_k, \mathcal{L})$ *can be found in $O(n(d^n_{\max} - d^n_{\min}))$ time.*

3.2.3 Local search algorithm

Combining search in neighborhoods of different types usually yields better solutions than if we apply only moves in a neighborhood of a single type.

We tested our local search procedure for flexible flow shop instances with setup times needed if jobs from different groups are processed one after another by the same machine.

Brah and Hunsucker developed a branch and bound algorithm [10] for the problem version where the objective is to minimize the makespan. Their algorithm can easily be modified to minimize total average flow time. That algorithm, which is designed to find exact solutions, is almost useless for large-scale problem instances.

Another exact solution method was presented by Sawik [50]. He formulates the problem as a mixed integer one and uses a linear integer solver to solve it. In that problem version,

limited buffer capacities are considered as in the paper by Wardono and Fathi [60] who developed a tabu-search heuristic. A survey on exact solution methods can be found in Kis and Pesch [38].

Let us describe the problem precisely. There are n jobs $J_1, \ldots, J_n$. Every job consists of m operations indexed by numbers $1, \ldots, m$. Operations can only be executed in increasing order of index. For every operation with index $l \in [1, m]$, there are m_l parallel machines able to execute it. In other words, there are m stages with parallel machines at each stage, and every job, to be complete, must pass through stages from 1 to m. For every job J_j its processing times $p_{1j}, \ldots, p_{mj}$ at corresponding stages are given. For every pair of jobs J_i and J_j there is a machine-independent setup time s_{ij}. The objective is to find a feasible schedule in order to minimize total average flow time $\frac{1}{n}\sum_{j=1}^{n} C_{mj}$ which is equivalent to minimizing $\sum_{j=1}^{n} C_{mj}$ that is the total completion time at the last stage. Further we will omit factor $1/n$. This multiplier will be taken into account only in the tables presenting computational results.

The problem can be written in the form of the following mixed integer problem.

$$
\begin{array}{lcll}
\min & \sum_{j=1}^{n} C_{mj} & & \\
\text{s.t.} & y_{lj} + \sum_{i\in[1,n]\setminus\{j\}} x_{lij} \le 1 & \forall j \in [1,n], \forall l \in [1,m] & (3.13)\\
& \sum_{j\in[1,n]\setminus\{i\}} x_{lij} \le 1 & \forall i \in [1,n], \forall l \in [1,m] & (3.14)\\
& \sum_{j\in[1,n]} y_{lj} \le m_l & \forall l \in [1,m] & (3.15)\\
& C_{lj} \ge C_{li} + p_{lj} + s_{ij}x_{lij} - Q(1 - x_{lij}) & \forall i,j \in [1,n], i \ne j \forall l \in [1,m] & (3.16)\\
& C_{li} \ge C_{l-1i} + p_{li} & \forall i \in [1,n], \forall l \in [1,m] \setminus \{1\} & (3.17)\\
& C_{1i} \ge p_{1i} & i \in [1,n] & (3.18)\\
& x_{lij} \in \{0,1\} & \forall i,j \in [1,n], i \ne j, \forall l \in [1,m] & (3.19)\\
& y_{lj} \in \{0,1\} & \forall j \in [1,n], l \in [1,m] & (3.20)
\end{array}
$$

Here variables C_{lj} represent job completion times for every stage with index $l \in [1, m]$. Variable y_{li} takes value 1 if and only if job J_i must be the first in any job sequence on a machine processing job J_i at stage l. A zero value of this variable indicates that there are

some other jobs processed before J_i by this machine. Variable x_{lij} takes value 1 if job J_j is processed after job J_i by the same machine, and 0, otherwise. Inequalities (3.13) ensure for every stage l that either job J_i is processed first by a machine at this stage, or there are some other jobs processed before it by the same machine at stage l. According to inequalities (3.14), at most one job may be processed immediately after a particular job. The number of jobs being first on machines at stage l should not exceed m_l for every stage index $l \in [1, m]$. This is guaranteed by inequalities (3.15). Inequalities (3.16) restrict schedules to those where only one job may be processed by a machine at a time. Constant Q in this inequality is chosen to be sufficiently large. Inequalities (3.17) indicate that a job completion time at some stage with index l should be not less than a completion time of this job at the previous stage plus the processing time at the present stage. At the first stage completion times cannot be less than processing times which is guaranteed by inequality (3.18).

Our algorithm is based on a local search procedure applied separately to every stage. Having a partial schedule in hand, we arrange the jobs at the current stage by list scheduling (taking into account release dates), and improve the current average completion time by our local search procedures. When a given criterion is met, say a given iteration limit is reached, we proceed with another stage. As we will see later from the experimental results, the more iterations of the local search procedure we make at every stage, the better solutions we may obtain.

Algorithm 3.2.2

$C_j^0 := 0$ for all $j \in [1, n]$;
Generate schedule C^1 by list scheduling in nondecreasing order of processing times;
for $l = 1$ **to** m **do**
    for $k = 1$ **to** K_l **do**
        $C^l := \arg\min_{C \in \texttt{matching}(C^l)} \sum_{j=1}^{n} C_j$;
        $C^l := \arg\min_{C \in \texttt{dynamic}(C^l; \mathcal{J}_k, \mathcal{L})} \sum_{j=1}^{n} C_j$;
        $C^l := \arg\min_{C \in \texttt{assignment}(C^l, j_1, \ldots, j_{m_l})} \sum_{j=1}^{n} C_j$;
        $C^l := \arg\min_{C \in \texttt{dynamic}(C^l; \mathcal{J}_k, \mathcal{L})} \sum_{j=1}^{n} C_j$;

$C^l := \arg\min_{C \in \texttt{rotation}(C^l; j_1, \ldots, j_{m_l})} \sum_{j=1}^{n} C_j$;

$C^l := \arg\min_{C \in \texttt{dynamic}(C^l; \mathcal{J}_k, \mathcal{L})} \sum_{j=1}^{n} C_j$;

end do

Generate release dates: $r_j := C_j^l$;

if $l < m$ **then**

Generate schedule C^{l+1} by list scheduling in nondecreasing order of release dates;

end if

end do

Recall that a list scheduling procedure is a simple greedy algorithm which takes jobs in a given order and assigns them to machines so that a start time of a job is as early as possible. In the above algorithms, since jobs are sorted in nondecreasing order of release dates, a job is always assigned to the end of the job sequence previously assigned to a corresponding machine.

In the algorithm, provided that there are no setup times, the list scheduling procedure constructs an optimal schedule for the first stage since it follows the shortest processing time (SPT) rule known to yield an optimum for the problem of minimizing total completion time in absence of release dates (see Błażewicz et al. [9]).

Values $K_1, \ldots, K_m$ in the description of the algorithm are input parameters. For every stage l, K_l is the number of iterations on stage l.

We considered instances with three stages. For instances having up to 100 jobs one can derive a satisfactory lower bound rather quickly (in a few minutes) using the linear relaxation of the above mixed integer formulation. To solve the linear relaxation, we have used CPLEX. To tighten our mixed integer formulation, we used valid inequalities of Schulz [51]:

$$\sum_{j \in \mathcal{S}} m_l p_{lj} C_{lj} \geq \frac{1}{2} \left(\sum_{j \in \mathcal{S}} m_l p_{lj}^2 + \left(\sum_{j \in \mathcal{S}} p_{lj} \right)^2 \right)$$

for all stages l and subsets of job indices of the form $\mathcal{S} = [j_1, j_2]$. For a real-life instance with 1437 jobs that we consider below, the objective value of this schedule is within about 61%

#	obj. val.	LB	deviation,%
1	22810,2	20807,131	9,626838991
2	22075,2	20743,1644	6,421564108
3	25940,4	23635,2841	9,752858862
4	22079,8	20030,0157	10,23356312
5	24112,6	22046,9735	9,369206617
6	23307,6	21068,4354	10,62805357
7	22247,5	20002,7552	11,22217803
8	23345,5	20687,5607	12,84800726
9	23310,3	21122,97	10,35521993
10	23403	21006,3733	11,40904556
average	23263,21	21115,06633	10,1866536

Table 3.1: Experimental results for 10 instances with 3 stages and 60 jobs.

of a trivial lower bound on the optimal value. This lower bound is obtained as follows. For every stage l we solve the scheduling problem with m_l machines and jobs having processing times $p_{l1}, \ldots, p_{ln}$. No release dates are taken into account. Total job completion time is the objective function in this problem. As it has been already mentioned, the SPT rule delivers an optimal solution with respect to such objective function. Having solved the problem for every stage, we choose the maximum among the optimal values. This maximum value is a lower bound on the optimal value of the original multistage problem.

Table 3.1 presents experimental results for ten instances with 60 jobs and integer processing times generated randomly within the interval $[1, 10000]$ which most often exceeds practical values of processing times. The number of machines at each of the three stages is 10, 15, and 12, respectively. There is in that way at least 4-5 jobs per machine. In this instance, the first stage is a bottleneck stage in this machine environment (i.e., this is the stage l with the largest ratio $\sum_{j=1}^{n} p_{lj}/m_l$ per machine where m_l is the number of machines on stage l).

For the mentioned instances, we have set parameters K_l in our heuristic procedure as $K_1 = 8$, $K_2 = 16$, and $K_3 = 32$.

Table 3.2 describes experiments with 100 jobs and the same machine environment.

Information for an instance with 100 jobs, $m_1 = 10$, $m_2 = 6$, $m_3 = 20$, (the second stage is a bottleneck) and processing times randomly generated within an integer interval $[1, 10000]$ is presented in Table 3.3. For this instance, we have used the same value K for all parameters $K_{1,2,3}$. Objective values of resulting schedules are in the second column. The

#	Obj. val.	LB	deviation,%
1	28922,9	25321,6314	14,2221034
2	29452,4	25219,386	16,78476233
3	29157,1	24365,1375	19,66729102
4	28397	24272,0032	16,99487581
5	32076,2	27725,3218	15,69279603
6	28203,2	23814,1963	18,43019871
7	30276,6	26543,8183	14,06271569
8	29975,1	25758,5978	16,3693002
9	31496,2	26054,2585	20,88695597
10	35201,8	31072,811	13,28810902
average	30315,85	26014,71618	16,53346433

Table 3.2: Experimental results for 10 instances with 3 stages and 100 jobs.

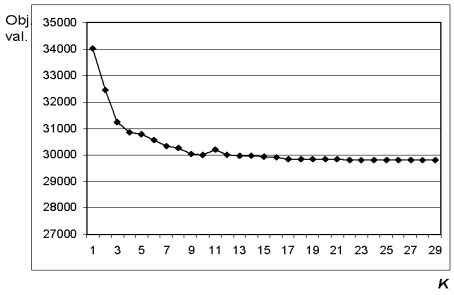

Figure 3.8: Diagram for an instance with 100 jobs and three stages

third column contains estimations of percentage deviation from an optimal objective value. The estimation is made using lower bounds obtained by CPLEX. Similar information is presented in Table 3.4 for a real-world instance, coming from the semiconductor industry, with 1437 jobs. In this instance, there are three stages. At the last stage there are five machine types. Any job may only be processed by a given machine type. There are 15 job groups in that instance. A setup time between two consequentially processed jobs belonging to different groups is equal to 60.

Figures 3.8 and 3.9 visualize Tables 3.3 and 3.4, respectively. Vertical axes show objective function values at corresponding values of the parameter K. In both examples increasing K leads in most cases to improvements of the entire schedule. Since our local search procedures are applied separately to different stages, we have however no guarantee that the objective value of the entire schedule decreases as this parameter increases. This is seen from both examples.

K	value	deviation,%
0	34016,5	40,4827786
1	32445,8	33,9960354
2	31259,9	26,1811349
3	30847,5	29,0984554
4	30800,6	27,3953085
5	30553,5	27,2016189
6	30339	25,2952837
7	30255,6	24,9508549
8	30038,8	24,0555051
9	29990,1	23,8543818
10	30204,9	24,7414719
11	30015,6	23,9596927
12	29980,5	23,8147353
13	29972,8	23,7829355
14	29940,1	23,6478897
15	29890,3	23,4422235
16	29848,8	23,2708351
17	29833,6	23,2080615
18	29844,1	23,2514248
19	29836,4	23,219625
20	29824,5	23,1704799
21	29819,4	23,1494177
22	29817,3	23,140745
23	29816,2	23,1362022
24	29811,5	23,1167919
25	29809,9	23,1101842
26	29807,9	23,1019245
27	29804,5	23,087883
28	29797,9	23,0606261

Table 3.3: Experimental results for a particular instance with 100 jobs and three stages

K	value	deviation,%
0	2660.89	61,9679216
1	2610.8	58,9189518
2	2521.05	53,4558846
3	2510.37	52,8057948
4	2480.82	51,0070913
5	2479.33	50,9163953
6	2461	49,8006513
7	2479.07	50,9005691
8	2462.7	49,90413
9	2454.62	49,4123018
10	2452.22	49,2662142
11	2452.08	49,2576924

Table 3.4: Experimental results for the real-world instance

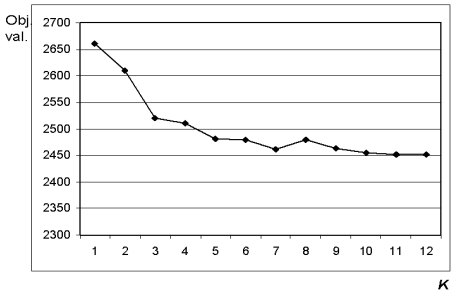

Figure 3.9: Diagram for the real-world instance.

3.2.4 Conclusions

The numerical experiments introduced in this subsection show that the separate local optimization of each stage of the considered three-stage manufacturing system yields an essential improvement in the overall cost. The methods that has been used to find locally optimal solutions for every stage are common for combinatorial optimization. The search in one of the considered neighborhoods is based on a pseudopolynomial dynamic programming algorithm which can be transformed into an FPTAS if it is necessary.

3.3 An FPTAS for a nonlinear scheduling problem

Consider a scheduling problem $Pm||\sum F_{ij}(C_j)$. An input of this problem consists of non-negative nondecreasing functions $F_{ij}(\cdot)$, $j = 1, \ldots, n$, $i \in [1, m]$, and n jobs J_j $(j = 1, \ldots, n)$ with positive integral processing times p_j. Functions $F_{ij}(\cdot)$, $j = 1, \ldots, n$ are assumed to be computable in constant time. Each of m parallel machines $M_1, \ldots, M_m$ can process only one job at a time. No preemptions are allowed, i.e., processing of a job by a machine cannot be interrupted until the job is complete. The goal is to schedule the jobs on m parallel machines (note that m is fixed, it is not a part of the input) so as to minimize $\sum_{j=1}^{n} F_{A_C(j),j}(C_j)$ where C_j, $j = 1, \ldots, n$, are job completion times.

An obvious special case is $Pm||\sum w_j C_j$ where the objective is to minimize the sum of weighted job completion times. There exists a pseudo-polynomial dynamic programming

algorithm for this NP-hard problem (Bruno, Coffman, and Sethi [11]). A fully polynomial time approximation scheme for $Pm||\sum w_jC_j$ was developed by Sahni [49].

For any instance of $Pm||\sum w_jC_j$ it may be assumed without loss of generality that the jobs are indexed so that $p_1/w_1 \leq p_2/w_2 \leq \ldots \leq p_n/w_n$. Simple interchange arguments show that there always exists an optimum schedule which does not contain any idle time, and where each machine processes jobs in nondecreasing order of their indices. Such property can be formulated as a rule: on every machine schedule jobs in nondecreasing order of ratios p_j/w_j. This rule is known as Smith's rule [52].

Many pseudopolynomial dynamic programming algorithms essentially rely upon job sequencing rules similar to Smith's rule. Later we will discuss other scheduling problems where the following restriction may also be admitted without loss of an optimal solution provided that for every instance jobs are sorted and enumerated in appropriate order.

Restriction 3.3.1 *If jobs J_i and J_j are executed by the same machine and $i < j$, then J_i is executed before J_j.*

Note that machines may not be identical and can differ by costs of executing jobs. In the case of identical machines the identity $F_{ij}(\cdot) \equiv F_{kj}(\cdot)$ holds for any $i, k \in [1, m]$, $j \in [1, n]$, and then we omit the first index and refer to $F_{ij}(\cdot)$ as to $F_j(\cdot)$ for every $i \in [1, m]$ and $j \in [1, n]$. If machines are identical, then the information about completion times of jobs is enough to describe a schedule within a permutation of machines.

We will consider cost functions of an important class introduced in the following assumption.

Assumption 3.3.1 *There is a positive constant μ such that each function $F_{ij}(\cdot)$ satisfies the inequality*

$$F_{ij}(\alpha\beta) \leq F_{ij}(\alpha)\beta^{\mu}$$

for real values $\alpha \geq 0$ and $\beta \geq 1$.

The above assumption says that if we increase the argument by the factor of β, then the function value will grow by at most the factor of β^{μ}. That is, value μ is a constant restricting

the degree of the growth factor of function $F_{ij}(\cdot)$. One can see that the objective function $\sum_{j=1}^{n} w_j C_j$ (total weighted completion time) satisfies this assumption.

A similar assumption was considered by Kubiak and Kovalyov for some nonlinear scheduling and partition type problems [40].

The above assumption covers a wide class of functions. These functions can be used to approximate or model real functions. For instance, functions like $y(x) = ax^{\mu}$, where $a > 0$ and $\mu < 1$, can be used to model concave functions. Functions $F_{ij}(\cdot)$ satisfying the above assumption may also belong to a function class formed by nonnegative nondecreasing functions $f : \mathbb{R}_+ \to \mathbb{R}_+$ such that $f(0) = 0$ and function $a(x) = f(x)/x^{\mu}$ defined on $\mathbb{R}_{++}$ is nonincreasing. Indeed, we have

$$f(\alpha\beta) = a(\alpha\beta)(\alpha\beta)^{\mu} \leq a(\alpha)\alpha^{\mu}\beta^{\mu} = f(\alpha)\beta^{\mu}$$

for $\alpha > 0$ and $\beta \geq 1$. If $\alpha = 0$ and $\beta \geq 1$, then

$$0 = f(\alpha\beta) = f(0) = f(0)\beta^{\mu} = f(\alpha)\beta^{\mu}.$$

Let $f(\cdot)$ be differentiable on $\mathbb{R}_{++}$ and such that $f(0) = 0$. Assume that derivative $f'(x)$ does not exceed μ at any point $x > 0$, and $f(x) \geq x$. Then

$$a'(x) = \frac{f'(x)x^{\mu} - \mu x^{\mu-1} f(x)}{x^{2\mu}} \leq \frac{\mu x^{\mu} - \mu x^{\mu-1} x}{x^{2\mu}} = 0$$

for any $x > 0$ which means that $a(x)$ is nonincreasing. Therefore, function $f(\cdot)$ belongs to the mentioned class.

Unlike many approaches, we do not round processing times to obtain a required approximation. Instead, we round the objective function and then analyze properties of a recursive function upon which a straightforward dynamic programming algorithm is based. Using these properties, one may convert the dynamic programming algorithm into a fully polynomial time approximation scheme.

Let $\mathcal{I}^{\#}$ be an instance of the scheduling problem $Pm||\sum F_{ij}(C_j)$. Taking into account Restriction 3.3.1, we have a simple recursive function upon which a pseudopolynomial dynamic programming algorithm may be based:

$$\psi_t(q) = \min_{k \in [1,m]} \{\psi_{t-1}(q') + F_{kt}(C_t) | C_t = q'_k + p_t, q_k = C_t, q'_i = q_i \forall i \in [1,m] \setminus \{k\}\},$$

where $q \in \mathbb{Z}_+^m$. The function value $\psi_0(q)$ is assumed to be equal to zero if q is zero and to ∞, otherwise. The value $\psi_t(q)$ is an objective value of an optimal partial schedule of the first t jobs with workloads q_i on corresponding machines M_i, $i \in [1, m]$.

Further we will say that a function is defined at a point if its value does not equal ∞. All points at which a function is not equal to ∞ constitute a feasible domain of this function. Such points are also called feasible.

Obviously, the function $\psi_t(\cdot)$ is defined for all integer $m-$vectors q such that there exists a feasible schedule of jobs $J_1, \ldots, J_t$ with a workload q_i of each machine $i \in [1, m]$. The basic difficulty which comes from the function structure is that it is an NP-hard problem to verify if an arbitrary nonnegative $m-$vector belongs to the feasible domain of this function. Consider a decision problem where the question is whether one can find pairwise disjoint sets $\mathcal{S}_l \subset [1, t]$, $l \in [1, m]$, such that $\cup_{l \in [1,m]} \mathcal{S}_l = [1, t]$, satisfying equations $\sum_{j \in S_l} p_j = P$ for every $l \in [1, m]$ and a given positive integer P. With respect to our scheduling problem, it is required to check if there is a feasible schedule of jobs $J_1, \ldots, J_t$ on m machines without idle times and with machine workloads, each of which is equal to P. The answer is positive if and only if $m-$vector $(P, \ldots, P)$ belongs to the feasible domain of the function $\psi_t(\cdot)$. As the mentioned decision problem is known to be NP-hard, it is an NP-hard problem to verify if a certain point belongs to this feasible domain.

To overcome the mentioned difficulty, we transform the instance $\mathcal{I}^{\#}$ into an instance of another optimization problem which dynamic programming formulation is based on a recursive function with a rather simple feasible domain.

We relax feasibility conditions as follows. First of all, we allow each job to be executed in parallel by several machines. Denote by x_{kt} the part of the processing time p_t which is spent by the job J_t on the machine M_k, $i \in [1, m]$. Once a job J_t begins to be executed on a machine M_k, $k \in [1, m]$, the execution cannot be interrupted during x_{kt} time units. An idle time of a machine M_k, $k \in [1, m]$ after the execution of job J_t will be denoted by I_{kt}.

Let U be an upper bound on $OPT(\mathcal{I}^{\#})$ and $\widetilde{U} = U((1 + \rho)^{\mu} + \rho)$ where $\rho > 0$ is a parameter which will further control a relative error. Remind that constant μ indicates how fast cost functions may grow. Consider the following optimization problem where the

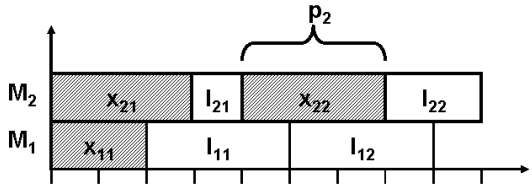

Figure 3.10: An example of a schedule feasible for Parallel Machine Scheduling.

objective is to minimize a function

$$F(x,I)=\sum_{k=1}^{m}\sum_{t=1}^{n}\left[\operatorname{sign}(x_{kt})\left(\widetilde{U}+1+F_{kt}\left(\sum_{j=1}^{t-1}(x_{kj}+I_{kj})+x_{kt}\right)\right)\right]:$$

$$\begin{aligned} \min \quad & F(x,I) \\ \text{s.t.} \quad & \textstyle\sum_{k=1}^{m} x_{kt}=p_t, \ \ \forall t\in[1,n], && (3.21)\\ & x_{kt},I_{kt}\in\mathbb{Z}_+, \ \ \forall k\in[1,m], t\in[1,n]. && (3.22) \end{aligned}$$

We call this problem Parallel Machine Scheduling. There is a one-to-one correspondence between instances of Parallel Machine Scheduling and instances of the scheduling problem $Pm||\sum_{ij}F_{ij}(C_j)$. In this subsection, we denote by $\mathcal{I}$ the instance of Parallel Machine Scheduling corresponding to the instance $\mathcal{I}^{\#}$ of $Pm||\sum_{ij}F_{ij}(C_j)$.

Figure 3.10 illustrates a feasible schedule for Parallel Machine Scheduling with two jobs and two machines. In that schedule, job J_1 is split into two operations, each of nonzero length x_{11} and x_{21}. Job J_2 is processed completely by machine M_2 and therefore x_{22} is equal to processing time p_2.

Lemma 3.3.1 *Feasible solution (x^*,I^*) is optimal for the instance $\mathcal{I}$, if and only if exactly one of the components $x^*_{1t},\ldots,x^*_{mt}$ is nonzero for any $t\in[1,n]$ and a schedule*

$C^* = (C_1^*, \ldots, C_n^*)$ *with job completion times*

$$C_t^* = \sum_{j=1}^{t-1}(x_{kj}^* + I_{kj}^*) + x_{kt}^*, \quad \forall t \in [1, n],$$

where $A_{C^*}(t) = k$ *and* $x_{kt}^* > 0$, *is optimal for instance* $\mathcal{I}^{\#}$.

Proof. Prove the if-part. Let $C^* = (C_1^*, \ldots, C_n^*)$ be an optimal schedule for $\mathcal{I}^{\#}$. Construct a feasible solution for instance $\mathcal{I}$ as follows. Set $x_{kt}^* := p_t$, and $x_{it}^* := 0$ for all $i \in [1, m] \setminus \{k\}$ where k is a number of a machine executing job J_t. Identify idle times I_{kt}^* through equations

$$I_{kt}^* = C_{t'}^* - p_{t'} - C_t^*$$

if job J_t is executed by machine M_k and t' is executed next by M_k in the optimal schedule. In other cases set

$$I_{kt}^* := 0.$$

Show that the obtained feasible solution is optimal for $\mathcal{I}$. Its objective value evaluates as

$$n(\widetilde{U} + 1) + OPT(I^{\#}) \leq n(\widetilde{U} + 1) + \widetilde{U}.$$

In any optimal solution (x^*, I^*) of the instance $\mathcal{I}$, exactly one of the components $x_{1t}^*, \ldots, x_{mt}^*$ is nonzero for any $t \in [1, n]$. (Otherwise, an objective value would be not less than $(n+1)(\widetilde{U}+1)$ and therefore be greater than $n(\widetilde{U} + 1) + \widetilde{U}$ which is an upper bound on $OPT(\mathcal{I})$.) This means that for any optimal solution the schedule C^* is defined correctly in the formulation of the lemma since there is no confusion with the machine assignment $A_{C^*}(\cdot)$. Every solution (x, I) to the instance $\mathcal{I}$ where only one of the components $x_{1t}, \ldots, x_{mt}$ is nonzero for every $t \in [1, n]$ can be easily transformed to a feasible schedule via equations

$$C_t = \sum_{k=1}^{m}\left(\sum_{j=1}^{t-1}\operatorname{sign}(x_{kj})(x_{kj} + I_{kj}) + \operatorname{sign}(x_{kt})x_{kt}\right)$$

where $t \in [1, n]$ and $x_{kt} > 0$ (this nonzero component equals p_t). An objective value of (x, I) is $n(\widetilde{U}+1)$ plus the objective value of the constructed schedule $C = (C_1, \ldots, C_n)$. Hence, the objective value of (x, I) is not less than $n(\widetilde{U} + 1) + OPT(\mathcal{I}^{\#})$. Therefore (x^*, I^*) is optimal for $\mathcal{I}$.

The only-if-part is symmetric to the if-part. The proof is complete. ∎

The lemma immediately implies the following corollary.

Corollary 3.3.1 $OPT(\mathcal{I}) = n(\widetilde{U}+1) + OPT(\mathcal{I}^{\#})$.

The next lemma shows that a slight transformation of the set of feasible solutions of instance $\mathcal{I}$ leads to slight changes of the optimal value.

Lemma 3.3.2 *Let (x^0, I^0) be a feasible solution for instance $\mathcal{I}$ having the lowest objective value among feasible solutions (x, I) satisfying*

$$\sum_{i=1}^{m}\sum_{j=1}^{t}(x_{ij}+I_{ij}) \geq \sum_{j=1}^{t}(p_j + \lfloor \rho p_j \rfloor) - t$$

for all t belonging to some set $\mathcal{S} \subseteq [1,n]$. Then

$$F(x^0, I^0) \leq n(\widetilde{U}+1) + (1+\rho)^{\mu} OPT(\mathcal{I}^{\#}).$$

Proof. Let (x, I) be an optimal solution for instance $\mathcal{I}$ of Parallel Machine Scheduling. Set $\hat{I}_{ij} = I_{ij} + \lfloor \rho x_{ij} \rfloor$ and $\hat{x} = x$. Solution $(\hat{x}, \hat{I})$ is feasible for instance $\mathcal{I}$ and, since in the optimal solution exactly one of components $x_{1j}, \ldots, x_{mj}$ is nonzero for every $j \in [1,n]$, satisfies

$$\sum_{i=1}^{m}\sum_{j=1}^{t}(\hat{x}_{ij}+\hat{I}_{ij}) \geq \sum_{i=1}^{m}\sum_{j=1}^{t}(x_{ij}+I_{ij}+\rho x_{ij}) - t \geq \sum_{j=1}^{t}(p_j+\rho p_j) - t \geq \sum_{j=1}^{t}(p_j+\lfloor \rho p_j \rfloor) - t$$

for all $t \in \mathcal{S}$. We have

$\sum_{k=1}^{m}\sum_{t=1}^{n} F_{kt}(\sum_{j=1}^{t-1}(\hat{I}_{kj}+\hat{x}_{kj}) + \hat{x}_{kt})\ \mathrm{sign}(\hat{x}_{kt}) \leq$
$\sum_{k=1}^{m}\sum_{t=1}^{n} F_{kt}((\sum_{j=1}^{t-1}(I_{kj}+x_{kj}) + x_{kt})(1+\rho))\ \mathrm{sign}(x_{kt}) \leq (1+\rho)^{\mu} OPT(\mathcal{I}^{\#})$

which implies the lemma. The last inequality of the above chain uses Assumption 3.3.1. ∎

Let L be a positive lower bound on $OPT(\mathcal{I}^{\#})$ and $\delta = \left\lfloor \frac{\rho L}{n} \right\rfloor$. Set $V := \lfloor \widetilde{U}/\delta \rfloor$. Consider a function

$$f(x,I) = \sum_{k=1}^{m}\sum_{t=1}^{n}\left[\mathrm{sign}(x_{kt})\left(V+1+f_{kt}\left(\sum_{j=1}^{t-1}(x_{kj}+I_{kj})+x_{kt}\right)\right)\right]$$

where

$$f_{kt}\left(\sum_{j=1}^{t-1}(x_{kj}+I_{kj})+x_{kt}\right)=\left\lfloor\frac{F_{kt}(\sum_{j=1}^{t-1}(x_{kj}+I_{kj})+x_{kt})}{\delta}\right\rfloor.$$

The following rounded instance denoted as $\mathcal{I}_R(\mathcal{S})$ is obtained from instance $\mathcal{I}$ by replacing the objective function $F(\cdot,\cdot)$ by $f(\cdot,\cdot)$ and imposing additional restrictions for all t from some set $\mathcal{S}\subseteq[1,n]$:

$$\begin{array}{ll}\min & f(x,I)\\ \text{s.t.} & (3.21)-(3.22)\\ & \sum_{i=1}^{m}\sum_{j=1}^{t}(x_{ij}+I_{ij})\geq\sum_{j=1}^{t}(p_j+\lfloor\rho p_j\rfloor)-t,\quad\forall t\in\mathcal{S}.\end{array}\tag{3.23}$$

Lemma 3.3.3 *Let (x',I') be an optimal solution of $\mathcal{I}_R(\mathcal{S})$. Then*

$$F(x',I')\leq n(\widetilde{U}+1)+(1+\rho)^{\mu}OPT(\mathcal{I}^{\#})+\rho OPT(\mathcal{I}^{\#}).$$

Proof. Let (x^0,I^0) have a lowest objective value among feasible solutions (x,I) of $\mathcal{I}$ satisfying

$$\sum_{i=1}^{m}\sum_{j=1}^{t}(x_{ij}+I_{ij})\geq\sum_{j=1}^{t}(p_j+\lfloor\rho p_j\rfloor)-t$$

for all $t\in\mathcal{S}$. Taking into account Lemma 3.3.2, we have the following inequality chain:

$$F(x',I')\leq\delta f(x',I')+n\delta\leq\delta f(x^0,I^0)+n\delta\leq F(x^0,I^0)+n\delta\leq$$

$$\leq n(\widetilde{U}+1)+(1+\rho)^{\mu}OPT(\mathcal{I}^{\#})+\rho OPT(\mathcal{I}^{\#}).$$

■

Theorem 3.3.1 *If (x^0,I^0) is an optimal solution for the instance $\mathcal{I}_R(\mathcal{S})$, then*

1. *For any $t\in[1,n]$ there exists exactly one machine index $k\in[1,m]$ such that $x^0_{kt}>0$.*
2. *A schedule with job completion times*

$$C_t=\sum_{j=1}^{t-1}(x^0_{kj}+I^0_{kj})+x^0_{kt},\quad\forall t\in[1,n],$$

and machine assignment $A_C(\cdot)$ *such that* $A_C(t) = k$ *whenever* $x_{kt} > 0$ *is feasible for* $\mathcal{I}^{\#}$ *and satisfies*

$$\sum_{t=1}^{n} F_{A_C(t)t}(C_t) \leq ((1+\rho)^{\mu} + \rho)OPT(\mathcal{I}^{\#})$$

Proof. To prove the theorem one can use Lemma 3.3.3 and arguments similar to those in the proof of Lemma 3.3.1. ∎

The above theorem implies that if (x^0, I^0) is an optimal solution of $\mathcal{I}_R(\mathcal{S})$, then

$$\sum_{k=1}^{m}\sum_{t=1}^{n} F_{kt}\left(\sum_{j=1}^{t-1}(x_{kj}^0 + I_{kj}^0) + x_{kt}^0\right) \leq \widetilde{U}.$$

Therefore

$$F_{kt}\left(\sum_{j=1}^{t-1}(x_{kj}^0 + I_{kj}^0) + x_{kt}^0\right) \leq \widetilde{U}$$

and thus the following assumption may be accepted without changing the set of optimal solutions.

Assumption 3.3.2 *Every cost function* $F_{kt}(\cdot)$ *satisfies* $F_{kt}(a) \leq \widetilde{U} + 1$ *for any nonnegative integer* a.

Assumption 3.3.2 implies that each function $f_{kt}(\cdot)$ is upper bounded by $O(V)$ on all nonnegative integers.

The properties discussed in this subsection allow us to concentrate only on solving the instance $\mathcal{I}_R(\mathcal{S})$. An appropriate choice of ρ gives us a possibility to find an $\varepsilon-$approximate schedule for $\mathcal{I}^{\#}$ by approximately solving the instance $\mathcal{I}$.

3.3.1 Recursive function and its properties

In this section, we study a dynamic programming formulation of an instance $\mathcal{I}_R(\mathcal{S})$ where $\mathcal{S} = \{t \in [1,n] | \sum_{j=1}^{t} p_j \geq 3n/\rho\}$. Note that $\mathcal{S}$ is either empty or $\mathcal{S} = [\tau, n]$ for some $\tau \in [1,n]$. A recursive function that we consider is defined as an optimal value of a reduced instance $\mathcal{I}_{t,q}$, where $t \in [1,n]$ and $q \in \mathbb{Z}_+^m$, obtained from instance $\mathcal{I}_R(\mathcal{S})$ by means of considering

only the first t job indices, relaxing constraints (3.23), and imposing an additional constraint $\sum_{j=1}^{t}(x_{kj}+I_{kj})=q_k$ for every $k\in[1,m]$. Let

$$B_\tau=\sum_{j=1}^{\tau}(p_j+\frac{1}{n}\lfloor\rho p_j\rfloor)-\tau.$$

Define B_l for all $l\in[\tau+1,n]$ as

$$B_l=B_{l-1}+\frac{1}{n}\sum_{j=1}^{l}\lfloor\rho p_j\rfloor+p_l-1$$

Instance $\mathcal{I}_{t,q}$ is formulated as

$$\begin{aligned}
\min\quad & f^t(x,I) && \\
\text{s.t.}\quad & \textstyle\sum_{j=1}^{t}(x_{kj}+I_{kj})=q_k, && \forall k\in[1,m], && (3.24)\\
& \textstyle\sum_{k=1}^{m}x_{kt}=p_t, && \forall t\in[1,n], && (3.25)\\
& \textstyle\sum_{k=1}^{m}\sum_{j=1}^{l}(x_{kj}+I_{kj})\geq B_l, && \forall l\in[\tau,t], && (3.26)\\
& x_{kj},I_{kj}\in\mathbb{Z}_+, && \forall k\in[1,m], j\in[1,t]. && (3.27)
\end{aligned}$$

where

$$f^t(x,I)=\sum_{k=1}^{m}\sum_{j=1}^{t}\left[\operatorname{sign}(x_{kj})\left(V+1+f_{kj}\left(\sum_{i=1}^{j-1}(x_{ki}+I_{ki})+x_{kj}\right)\right)\right].$$

One can see that (3.26) is a relaxation of constraints (3.23). Therefore,

$$OPT(\mathcal{I}_R(\mathcal{S}))\geq\min_q\{OPT(\mathcal{I}_{n,q})\}$$

where the minimum is taken over all feasible q.

Define a function $\phi_t(\cdot)$ at every feasible point q through the equation

$$\phi_t(q)=OPT(\mathcal{I}_{t,q}).$$

According to the formulation of instance $\mathcal{I}_{t,q}$, the set of feasible solutions of this instance is nonempty if and only if $\sum_{i=1}^{m}q_i\geq B_t$. Let (x,I) be an optimal solution of instance $\mathcal{I}_{t,q}$. If $\sum_{k=1}^{m}\sum_{j=1}^{t}(x_{kj}+I_{kj})>B_t$, then there are components I_{ij} that can be decreased so that the equation $\sum_{k=1}^{m}\sum_{j=1}^{t}(x_{kj}+I_{kj})=B_t$ holds and solution (x,I) remains optimal for instance

$\mathcal{I}_{t,q}$. This means that it is sufficient to consider only those vectors q which satisfy $q_k \leq B_n$ for every $k \in [1, m]$. Therefore further we suppose the feasible domain $dom(\phi_t)$ of the function $\phi_t(\cdot)$ to be defined as

$$dom(\phi_t) = \{y \in \mathbb{Z}^m | \sum_{i=1}^{m} y_i \geq B_t, y_i \leq B_n\}$$

in case $t \in [\tau, n]$ and as

$$dom(\phi_t) = \{y \in \mathbb{Z}^m | \sum_{i=1}^{m} y_i \geq \sum_{j=1}^{t} p_j, y_i \leq B_n\},$$

otherwise. Let $\phi_0(0) = 0$ and $dom(\phi_0) = \{0\}$. At any point $q \in dom(\phi_t)$ the value $\phi_t(q)$ is evaluated recursively by the formula

$$\phi_t(q) = \min_{q', x_{kt}, I_{kt}} \left\{ \phi_{t-1}(q') + \sum_{k=1}^{m} \text{sign}(x_{kt})(V + 1 + f_{kt}(q' + x_{kt})) \right\} \tag{3.28}$$

where the minimum is taken over all $q' \in dom(\phi_{t-1})$ and all $x_{kt} \in \mathbb{Z}_+$ and $I_{kt} \in \mathbb{Z}_+$, $k \in [1, m]$, satisfying

$$q'_k + x_{kt} + I_{kt} = q_k, \quad \forall k \in [1, m],$$

and

$$\sum_{k=1}^{m} x_{kt} = p_t.$$

For every $t \in [\tau, n]$ define an auxiliary function $\Phi_t(\cdot)$ on the set

$$dom(\Phi_t) = \left\{ y \in \mathbb{Z}^m | \sum_{k=1}^{m} y_k \geq B_{t-1} + p_t, \quad y_k \leq B_n \ \forall k \in [1, m] \right\}$$

by the same formula as $\phi_t(\cdot)$. (Note that ϕ_{t-1} is not replaced by Φ_{t-1}, but only ϕ_t is changed by Φ_t in that formula.) The set $dom(\Phi_t)$ contains the set $dom(\phi_t)$. By definition, $\Phi_t(q) = \phi_t(q)$ for all $q \in dom(\phi_t)$.

Replace the condition

$$\sum_{k=1}^{m} q_k \geq B_t$$

in the formulation of instance $\mathcal{I}_{t,q}$ by a weaker condition

$$\sum_{k=1}^{m} q_k \geq B_{t-1} + p_t.$$

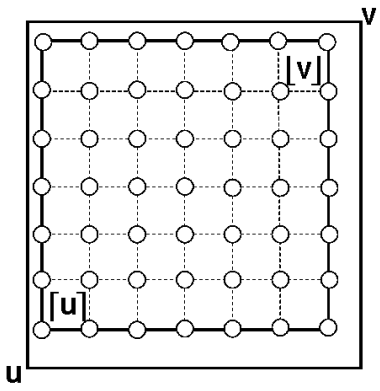

Figure 3.11: Integer Box.

Denote the obtained instance by $\widetilde{\mathcal{I}}_{t,q}$. Any function value $\Phi_t(q)$ is the optimal value of the instance $\widetilde{\mathcal{I}}_{t,q}$. The feasible set of instance $\widetilde{I}_{t,q}$ is not empty for every $q \in dom(\Phi_t)$.

For every $t \in [1, \tau - 1]$ we consider $\Phi_t(\cdot)$ to be identical to $\phi_t(\cdot)$.

Observation 3.3.1 *The function $\Phi_t(\cdot)$ is nonincreasing in each variable.*

Proof. Let (x, I) be an optimal solution for $\widetilde{\mathcal{I}}_{t,q}$. For any $k \in [1, m]$ the idle time I_{kt} can be increased by 1 without increasing the objective function. Therefore, function $\Phi_t(\cdot)$ is nonincreasing in each variable. ∎

We call a set $\mathcal{B} = \{y \in \mathbb{Z}^m | u \leq y \leq v\}$ an *integer box* defined by real vectors u and v. Vector

$$\lceil u \rceil = (\lceil u_1 \rceil, \ldots, \lceil u_m \rceil)$$

will be called a lower corner of the integer box $\mathcal{B}$, and vector

$$\lfloor v \rfloor = (\lfloor v_1 \rfloor, \ldots, \lfloor v_m \rfloor)$$

will be called an upper corner of $\mathcal{B}$. An integer box of the form $\mathcal{B} = \{y \in \mathbb{Z}^2 | u \leq y \leq v\}$ is illustrated in Figure 3.11. The circles correspond to integer points.

Lemma 3.3.4 *For any q contained in an integer box*

$$\mathcal{B} = \{y \in \mathbb{Z}^m | u \le y \le v\} \subseteq dom(\Phi_t),$$

there is $q' \in \mathcal{B}$ satisfying $q' \le q$, $\Phi_t(q') = \Phi_t(q)$, and $q_k = \lceil u_k \rceil$ for some $k \in [1, m]$.

Proof. Let (x, I) be an optimal solution for instance $\widetilde{\mathcal{I}}_{t,q}$. The objective value of (x, I) equals to $\Phi_t(q)$. If $q_i = \lceil u_i \rceil$ for some $i \in [1, m]$, then we are done. Therefore we assume $q_i > \lceil u_i \rceil$ for all $i \in [1, m]$. Assume $I_{ij} = 0$ for all $i \in [1, m]$ and $j \in [1, t]$. Then $\sum_{i=1}^{m} q_i = \sum_{i=1}^{t} p_i$ and thus, since $\mathcal{B} \subset dom(\Phi_t)$, equation $q_i = \lceil u_i \rceil$ holds for all $i \in [1, m]$. We have a contradiction since previously we have assumed that $q_i > \lceil u_i \rceil$ for all $i \in [1, m]$. Hence, there exists $I_{ij} > 0$ for some $i \in [1, m]$ and $j \in [1, t]$. Let $\Delta = \min\{I_{ij}, q_i - \lceil u_i \rceil\}$. Set $I_{ij} := I_{ij} - \Delta$. The objective value of the obtained solution does not exceed $OPT(\widetilde{\mathcal{I}}_{t,q})$. This solution is feasible for $\mathcal{I}_{t,\hat{q}}$, where $\hat{q}$ is obtained from q by decreasing component q_i by Δ, and therefore

$$OPT(\widetilde{\mathcal{I}}_{t,\hat{q}}) \le OPT(\widetilde{\mathcal{I}}_{t,q})$$

or equivalently

$$\Phi_t(\hat{q}) \le \Phi_t(q).$$

Set $q := \hat{q}$. We proceed transforming (x, I) until $q_i = \lceil u_i \rceil$ for some $i \in [1, m]$. After a finite number of iterations we have $q' \in \mathcal{B}$ such that $\Phi_t(q') \le \Phi_t(q)$ and $q' \le q$. Taking into account Observation 3.3.1, we conclude that $\Phi_t(q') = \Phi_t(q)$. The proof is complete. ∎

The above lemma is illustrated for $m = 2$ in Figure 3.12. Since $\Phi_t(q) = \Phi_t(q')$ and function $\Phi_t(\cdot)$ is nonincreasing in each of its arguments, this function is constant over the integer box with corners q' and q. The points belonging to this box are depicted by black filled circles in the picture.

The theorem below indicates that function $\Phi_t(\cdot)$ possesses a simple structure on any integer box lying completely in $dom(\Phi_t)$. Later we will use such boxes to cover the set $dom(\phi_t)$. Then, since $\Phi_t(\cdot)$ and $\phi_t(\cdot)$ are identical on $dom(\phi_t)$, we will have a complete description of $\phi_t(\cdot)$.

Theorem 3.3.2 *If $m = 2$, then any integer box $\mathcal{B} \subset dom(\Phi_t)$ is a union of at most $O(nmV)$ pairwise disjoint integer boxes $\widehat{\mathcal{B}} \subset dom(\Phi_t)$ over each of which function $\Phi_t(\cdot)$ is constant.*

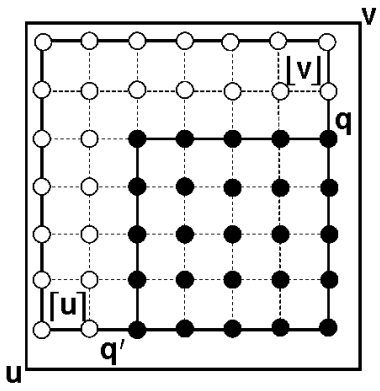

Figure 3.12: Function Φ_t takes the same value at each point belonging to the integer box with corners q' and q.

Proof. Let

$$\mathcal{B} = \{y \in \mathbb{Z}^2 | u \leq y \leq v\} \subset dom(\Phi_t)$$

where $u = (u_1, u_2)$ and $v = (v_1, v_2)$ are integer vectors.

The following iterative process shows how to find integer boxes $\widehat{\mathcal{B}}$ mentioned in the formulation of the theorem. Each of such boxes obtained at iteration k of the iterative process will be denoted by $\widehat{\mathcal{B}}^{(k)}$.

Set $k := 0$. Assume $\mathcal{B}^{(k)} = \mathcal{B}$. Let $z = (z_1, z_2)$ be the upper corner of this integer box. (Initially point (v_1, v_2) is the upper corner.) Lemma 3.3.4 and the fact that Φ_t is nonincreasing in each variable imply that either

$$\Phi_t(z_1, u_2) = \Phi_t(z_1, z_2) \tag{3.29}$$

or

$$\Phi_t(u_1, z_2) = \Phi_t(z_1, z_2). \tag{3.30}$$

In the case (3.29), let

$$\Delta = \max\{a \in \mathbb{Z}_+ | (z_1 - a, u_2) \in \mathcal{B}, \Phi_t(z_1 - a, u_2) = \Phi_t(z_1, z_2)\}.$$

In the case (3.30), calculate Δ as

$$\Delta = \max\{a \in \mathbb{Z}_+ | (u_1, z_2 - a) \in \mathcal{B}, \Phi_t(u_1, z_2 - a) = \Phi_t(z_1, z_2)\}.$$

If (3.29) holds, then define

$$\widehat{\mathcal{B}}^{(k)} = \{y \in \mathbb{Z}^2 | (z_1 - \Delta, u_2) \leq y \leq (z_1, z_2)\}.$$

Otherwise, let the integer box $\widehat{\mathcal{B}}^{(k)}$ be defined as

$$\widehat{\mathcal{B}}^{(k)} = \{y \in \mathbb{Z}^2 | (u_1, z_2 - \Delta) \leq y \leq (z_1, z_2)\}.$$

Since $\Phi_t(\cdot)$ is nonincreasing in each variable, $\Phi_t(\cdot)$ is constant over the constructed box $\widehat{\mathcal{B}}^{(k)}$. By construction, the set $\mathcal{B}^{(k)} \setminus \widehat{\mathcal{B}}^{(k)}$ is an integer box. If this set is empty, then we stop. Otherwise, we set $\mathcal{B}^{(k+1)} := \mathcal{B}^{(k)} \setminus \widehat{\mathcal{B}}^{(k)}$ and $k := k + 1$ and perform kth iteration.

Let

$$b_1^{\min} = \min_{a \in \mathbb{Z}_+} \{\Phi_t(z_1 - a, z_2) | (z_1 - a, z_2) \in \mathcal{B}^{(k)}\}$$

and

$$b_2^{\min} = \min_{a \in \mathbb{Z}_+} \{\Phi_t(z_1, z_2 - a) | (z_1, z_2 - a) \in \mathcal{B}^{(k)}\}.$$

Since $\Phi_t(q) \leq 2nm(V + 1)$ for any $q \in dom(\Phi_t)$ owing to the fact that functions $f_{ij}(\cdot)$ are upper bounded by $V + 1$, the inequality

$$b_1^{\min} + b_2^{\min} \leq 2nm(V + 1)$$

must hold at each iteration. Function $\Phi_t(\cdot)$ is nonincreasing in each variable. Therefore, due to the choice of Δ and to the fact that $\Phi_t(\cdot)$ is integer-valued, every iteration, except the last one, reduces the value $b_1^{\min} + b_2^{\min}$ by at least 1. This entails that we need at most $O(nmV)$ iterations. ∎

Figure 3.13 illustrates the iterative process described in the proof of the above theorem. The numbers inside boxes reflect the order in which these boxes were found. It is seen from the picture that at every iteration the iterative process puts a box so that either a horizontal or vertical bound of the currently uncovered area is shifted.

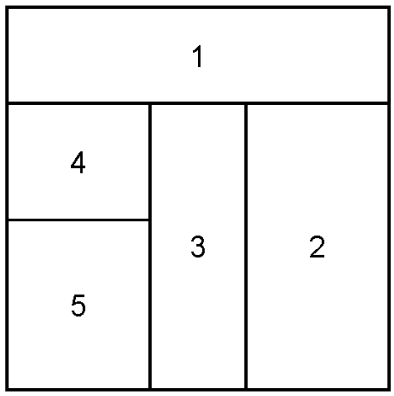

Figure 3.13: Subdividing into boxes on each of which the function is constant.

Theorem 3.3.3 *For any $t \in [\tau, n]$ there exists a family $\mathfrak{F}$ of integer boxes $\mathcal{B} \subset dom(\Phi_t)$ covering the set $dom(\phi_t)$ and having cardinality $|\mathfrak{F}|$ upper bounded by*

$$O\left(m^2\left(\frac{2n}{\rho}\right)^{\frac{1}{\log m - \log(m-1)}}\right).$$

At the same time, this is a running time estimation of an algorithm finding family $\mathfrak{F}$.

Proof. Using bisection arguments, we will show how to construct a family $\mathfrak{F}$.

Notice that the set

$$dom(\phi_t) \setminus \{y \in \mathbb{Z}_+^m | \sum_{k=1}^{m} y_k \geq B_t, \;\; y_k \leq B_t \; \forall k \in [1,m]\}$$

may be covered with m integer boxes

$$\left\{y \in \mathbb{Z}_+^m \;\mid\; 0 \leq y_k \leq B_{t-1} + p_t, \quad B_{t-1} + p_t + 1 \leq y_i \leq B_n, \;\; \forall i \in [1,m] \setminus \{k\}\right\}$$

where $k \in [1,m]$ and an integer box

$$\left\{y \in \mathbb{Z}_+^m \mid B_{t-1} + p_t + 1 \leq y_i \leq B_n, \; \forall i \in [1,m]\right\}.$$

Initialize $\mathfrak{F}$ with these $m+1$ sets. Consider a set

$$\mathcal{T} = \left\{y \in \mathbb{Z}_+^m \mid \sum_{k=1}^{m} y_k \geq B_t, \;\; y_k \leq B_{t-1} + p_t \; \forall k \in [1,m]\right\}.$$

We will iteratively expand the family $\mathfrak{F}$ by adding integer boxes until we are sure that the set of integer points of $\mathcal{T}$ is completely covered by these boxes.

In $\mathcal{T}$ find all vectors v being non-dominated by vectors from $\mathcal{T}$ in respect to the relation $\leq$. (At the first iteration there exists a single such vector.) For every vector v construct the integer box $\mathcal{B}_v = \{y \in \mathbb{Z}^m | u \leq y \leq v\} \subset dom(\Phi_t)$ such that values $v_i - u_i$, $i \in [1, m]$, are all equal and u belongs to the hyperplane

$$\mathcal{H} = \{y \in \mathbb{R}^m | \sum_{i=1}^{m} y_i = B_{t-1} + p_t\}$$

and add this box to the family $\mathfrak{F}$. This means that the lower corner $u \in \mathbb{R}^m$ is identified as

$$u_k = v_k - \frac{1}{m}\left(\sum_{i=1}^{m} v_i - B_{t-1} - p_t\right)$$

for all $k \in [1, m]$. Add all integer boxes $\mathcal{B}_v$ to the family $\mathfrak{F}$. Subtract boxes $\mathcal{B}_v$ from the set $\mathcal{T}$.

Proceed the described iterative process until $\mathcal{T}$ is empty. Notice that at an iteration r there are m^r vectors in the set $\mathcal{T}$ being non-dominated by other vectors from this set.

Initially, the maximum euclidian distance between a vector of set $\mathcal{T}$ and the hyperplane $\mathcal{H}$ is equal to $\frac{\sqrt{m}}{m}(B_{t-1} + p_t)$. By construction, every iteration, except maybe the last one, reduces the maximum euclidian distance between this hyperplane and a vector from of the set $\mathcal{T}$ by at lest the factor of $\frac{m}{m-1}$. Notice that $t \in [\tau, n]$ by the conditions of the theorem and thus all $z \in dom(\phi_t)$ satisfy $\sum_{i=1}^{m} z_i \geq B_t$. I.e., a euclidian distance between every such z and the hyperplane $\mathcal{H}$ is greater than or equal to $\frac{\sqrt{m}}{m}\left(B_t - B_{t-1} - p_t\right)$. Therefore at most

$$O\left(\log_{\frac{m}{m-1}} \frac{\frac{\sqrt{m}}{m}(B_{t-1} + p_t)}{\frac{\sqrt{m}}{m}(B_t - B_{t-1} - p_t)}\right)$$

iterations are needed to meet the stoping criterion of the iterative process. Taking into account that

$$\frac{B_{t-1} + p_t}{B_t - B_{t-1} - p_t} \leq \frac{2n}{\rho}$$

we may conclude that the iteration number is upper bounded by $O(\log_{\frac{m}{m-1}} \frac{2n}{\rho})$. Then the number of integer boxes in the family $\mathfrak{F}$ after the last iteration is upper bounded by

$$O\left(m^{\log_{\frac{m}{m-1}} \frac{2n}{\rho}}\right) \leq O\left(m^2 \left(\frac{2n}{\rho}\right)^{\frac{1}{\log m - \log(m-1)}}\right).$$

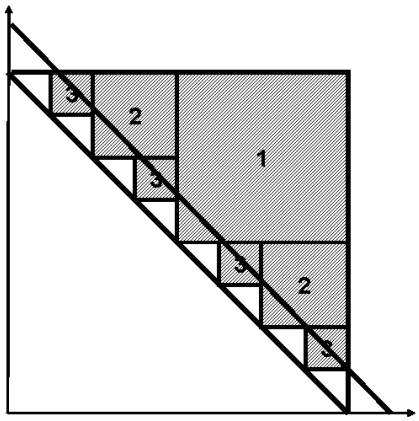

Figure 3.14: Iterative process.

The proof is complete. ∎

Figure 3.14 illustrates the iterative process in the proof of Theorem 3.3.3 for $m = 2$. In this picture, inside each box there is a number of iteration at which this box was put in family $\mathfrak{F}$ in the iterative process. The feasible domain lies above the upper inclined line.

3.3.2 Algorithm

In this section, we discuss a simple dynamic programming algorithm for $m = 2$ which is based on the recursive function properties.

For every $t \in [1, n]$, let us consider a family $\mathfrak{P}_t$ of integer boxes $\widehat{\mathcal{B}}$ such that

$$dom(\phi_t) \supseteq \bigcup_{\widehat{\mathcal{B}} \in \mathfrak{P}_t} \widehat{\mathcal{B}} \tag{3.31}$$

and $\Phi_t(\cdot)$ is constant on every box of family $\mathfrak{P}_t$. If for each box a corresponding function value is given, then, using the fact that $\phi_t(q) = \Phi_t(q)$ for all $q \in dom(\phi_t)$, we may calculate function $\phi_t(\cdot)$ at any point of its feasible domain.

Lemma 3.3.5 *If $\mathfrak{P}_{t-1}$ with related values of function $\Phi_{t-1}(\cdot)$ for every integer box in $\mathfrak{P}_{t-1}$ are given, then for any $q \in dom(\Phi_t)$ the function value $\Phi_t(q)$ can be found in $O(|\mathfrak{P}_{t-1}|V^m)$ time.*

Proof. Calculate $u_k = \max\{a \in [0, B_n] | f_{kt}(a) \leq V\}$. This can be done in $O(\log B_n)$ time since $f_{kt}(\cdot)$ is nondecreasing. The monotony of f_{kt} also implies that there exists a partition

$$\mathfrak{U}_k = \bigcup_{[s_1,s_2]\in\mathfrak{U}_k} [s_1, s_2]$$

of the interval $[0, B_n]$ into at most V intervals $[s_1, s_2]$ on which the multiple of functions $\text{sign}(\cdot)$ and $V + 1 + f_{kt}(\cdot)$ is constant, i.e.,

$$\text{sign}(a)(V + 1 + f_{kt}(a)) = \text{sign}(b)(V + 1 + f_{kt}(b))$$

for all $a, b \in [s_1, s_2]$ and for any $[s_1, s_2] \in \mathfrak{U}_k$. Let $[s_1^1, s_2^1], \ldots, [s_1^m, s_2^m]$ be intervals from the families $\mathfrak{U}_1, \ldots, \mathfrak{U}_m$, respectively. Then the value

$$\phi_{t-1}(q') + \sum_{k=1}^{m}(V + 1 + f_{kt}(x_{kt})) \tag{3.32}$$

is the same for any nonnegative vector

$$(x_{1t}, \ldots, x_{kt}) \in [s_1^1, s_2^1] \times \ldots \times [s_1^m, s_2^m],$$

nonnegative vector $(I_{1t}, \ldots, I_{mt})$, and vector $q' \in \widehat{\mathcal{B}} \cup dom(\phi_{t-1})$, $\widehat{\mathcal{B}} \in \mathfrak{P}_{t-1}$, satisfying

$$\sum_{k=1}^{m} x_{kt} = p_t$$

and

$$q'_k + x_{kt} + I_{kt} = q_k$$

for any $k \in [1, m]$. For each combination of intervals $[s_1^1, s_2^1], \ldots, [s_1^m, s_2^m]$ and any box $\widehat{\mathcal{B}}$ such three vectors can be easily found in constant time if they exist. All possible combinations of the mentioned intervals and boxes yield an enumeration algorithm to find the vectors $(x_{1t}, \ldots, x_{kt})$, $(I_{1t}, \ldots, I_{mt})$, and q' minimizing (3.32). This enumeration algorithm runs in $O(|\mathfrak{P}_{t-1}|V^m)$ time. ∎

Further we assume that $m = 2$ and discuss how to find a partition $\mathfrak{P}_t$ satisfying property (3.31). The following algorithm performs the partitioning.

Algorithm 3.3.1

for $t = 1$ **to** $\tau - 1$ **do**

$\mathfrak{P}_t := \cup_{y \in dom(\phi_t)} \{y\}$.

(I.e., every point y in $dom(\phi_t)$ form a box $\{y\}$ of the partition $\mathfrak{P}_t$.)

end do

for $t = \tau$ **to** n **do**

Run the iterative process in the proof of Theorem 3.3.3

to construct the family $\mathfrak{F}$ covering $dom(\phi_t)$.

Run the iterative process in the proof of Theorem 3.3.2,

to partition each box $\mathcal{B}$ in $\mathfrak{F}$ into boxes $\widehat{\mathcal{B}}$ on each of which $\phi_t(\cdot)$ is constant.

Thereby partition $\mathfrak{P}_t$ is constructed.

end do

At every iteration of the iterative process in the proof of Theorem 3.3.3 one can maintain a list of currently non-dominated points of the set $\mathcal{T}$. Using a point taken from the head of this list, we may construct a box in the way suggested by the iterative process, delete this point, and add at most $m = 2$ new points into the list. We need constant time to generate each of these $m = 2$ points. Each point is added to the list only once. The whole number of points added to the list and considered by the iterative process is upper bounded by $O\left(\frac{1}{\rho} + 1\right)$ according to Theorem 3.3.3. Thus at the same time this upper bound is a time complexity estimation of the the iterative process in the proof of Theorem 3.3.3.

Using the fact that $\phi_t(\cdot)$ is nonincreasing in each argument, the iterative process in the proof of Theorem 3.3.2 can calculate Δ in $O(\log B_n)$ basic operations and computations of $\phi_t(\cdot)$ at each iteration. This iterative process performs at most $O(mnV) = O(nV)$ iterations for a box $\mathcal{B} \subset dom(\phi_t)$. Therefore at every iteration t where $\sum_{j=1}^{t} p_j > 3n/\rho$ the algorithm performs at most

$$O(nV|\mathfrak{F}| \log B_n) \leq O\left(Vn\left(\frac{1}{\rho} + 1\right) \log B_n\right)$$

basic operations and calculations of function $\phi_t(\cdot)$. This number dominates $O(n/\rho)$ basic operations needed to construct $\mathfrak{P}_t$ such that $|\mathfrak{P}_t| \leq O(n/\rho)$ in case $\sum_{j=1}^{t} p_j \leq 3n/\rho$. Every

calculation of the function $\phi_t(\cdot)$ requires at most $O(|\mathfrak{P}_{t-1}|V^2)$ basic operations. Since the iterative process in the proof of Theorem 3.3.2 applied to every box $\mathcal{B}$ in the family $\mathfrak{F}$ constructs partition $\mathfrak{P}_{t-1}$ of the set $dom(\phi_t)$ so as $|\mathfrak{P}_{t-1}| \leq O(nV|\mathfrak{F}|)$, the overall time complexity of the algorithm is upper bounded by

$$O\left(V^4 n^2 \left(\frac{1}{\rho} + 1\right) \log B_n\right). \tag{3.33}$$

Further we assume that every partition $\mathfrak{P}_t$ is represented as a list where each item corresponds to an interval with the related value of the function $\phi_t(\cdot)$. The following backtracking algorithm finds an optimal solution to the instance $\mathcal{I}_R(\mathcal{S})$. We assume that $\mathfrak{P}_0$ consists of a single box $\{0\}$.

Algorithm 3.3.2

Select $\widehat{\mathcal{B}} \in \mathfrak{P}_n$ on which $\phi_n(\cdot)$ is minimum.
Let v be the value taken by $\phi_n(\cdot)$ on $\widehat{\mathcal{B}}$.
Take arbitrary $q \in \widehat{\mathcal{B}}$.
for $t = n$ **down to** 1 **do**
    Select $k \in [1, m]$ and $\widehat{\mathcal{B}}' \in \mathfrak{P}_{t-1}$ so as
    there exists $I_{kt} \geq 0$ such that $q' := (q_1, \ldots, q_{k-1}, q_k - p_t, q_{k+1}, \ldots, q_m) \in \widehat{\mathcal{B}}'$
    and $v - (V+1) - F_{tk}(q_k - I_{kt}) = v'$
    where v' is the value taken by $\phi_{t-1}(\cdot)$ on $\widehat{\mathcal{B}}'$.
    Set $v := v'$ and $q := q'$.
    Set $I^0_{kt} := I_{kt}$ and $I^0_{it} := 0$ for all $i \in [1, m] \setminus \{k\}$.
    Set $x^0_{kt} := p_t$ and $x^0_{it} := 0$ for all $i \in [1, m] \setminus \{k\}$.
end do

The algorithm relies on Theorem 3.3.1, item 1, which says that for any optimal solution (x^0, I^0) of the instance $\mathcal{I}_R(\mathcal{S})$ and for any job index $t \in [1, n]$ there exists exactly one machine index $k \in [1, m]$ such that component x^0_{kt} is positive. This means that this positive x^0_{kt} must be equal to p_t. The algorithm constructs (x^0, I^0) in such a way that for any job

index $t \in [1, n]$ the component I^0_{kt} may be positive only if $x^0_{kt} = p_t$. Here one uses the fact that $\phi_t(\cdot)$ is nonincreasing in each variable and thus other variants will not give a better choice of q'.

The complexity of an iteration of the for-loop in Algorithm 3.3.2 is upper bounded by $O(|\mathfrak{P}_{t-1}|)$. (Only the quantity of boxes in the partition counts due to the fact that we consider a constant number of machines.) Thus the overall time complexity of the backtracking algorithm is dominated by the time complexity estimation of Algorithm 3.3.1. Thus we have the following theorem:

Theorem 3.3.4 *An optimal solution of the instance $\mathcal{I}_R(\mathcal{S})$ can be found in time upper bounded by (3.33).*

If $\mu < 1$ and $\rho = \varepsilon/2$, our algorithm finds an $\varepsilon-$approximate solution. Since it runs in time (3.33), we have a fully polynomial time approximation scheme provided that the bound improvement procedure is additionally used.

Consider now the case when $\mu \geq 1$. Given $\varepsilon > 0$, a rational value between $\frac{(1+\varepsilon/2)^{\frac{1}{\mu}}-1}{2}$ and $(1+\varepsilon/2)^{\frac{1}{\mu}} - 1$ can be found in time polynomial of $1/\varepsilon$. Therefore, due to our previous discussion, we have an algorithm finding an $\varepsilon-$approximate solution in time (3.33) where ρ is replaced by $(1+\varepsilon/2)^{\frac{1}{\mu}} - 1$. To obtain bounds U and L satisfying $U/L = 3$ in polynomial time, we may use the bound improvement procedure. Thus, we have the following theorem:

Theorem 3.3.5 *There exists an FPTAS for $P2||\sum F_{ij}(C_j)$.*

.

3.3.3 Time-varying machine speeds

We now address a case when the speed of each machine may vary over time. Let p_j be job processing times provided that the speed of a machine executing a job equals one. For simplicity we assume that p_j are integer values. Let a polynomially computable function

$$g_i : \mathbb{R}_+ \to \mathbb{R}_+$$

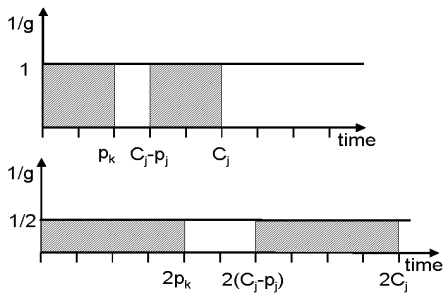

Figure 3.15: Different constant machine speeds

for any time moment express a factor by which a speed of machine M_i differs from the unit speed. (We call this function a *speed function*.) Let us explain it more precisely. Let some job have a processing time p on a machine having unit speed. If machine M_i, during processing this job, had the same speed as at some moment T, then this job would be completed in $g_i(T)p$ time units. Therefore, if C_j is a completion time of job J_j at a machine having unit speed, then

$$G_i(C_j) = \int_0^{C_j} g_i(x)dx$$

is its completion time on machine M_i whose speed is expressed by speed function $g_i(\cdot)$. (Here $G_i(\cdot)$ is an antiderivative of $g_i(\cdot)$.) The value $G_i(C_j - p_j)$ in this case is a start time of job J_j.

Note that a similar definition of the speed function was introduced in the thesis by Schulz [51].

In Figure 3.15, it is shown how job processing and completion time depend on machine speed. Two cases are illustrated. In the first one, a machine having the unit speed processes jobs J_k and J_j with processing times $p_k = p_j = 2$. Idle time after processing job J_k is equal to one. Completion times of jobs J_k and J_j are equal to C_k and C_j, respectively. Start time of job J_j is equal to $C_j - p_j$. In the second case, machine speed is also constant and expressed by speed function $g \equiv 2$. In this case, the job processing times and idle time become two times longer. Completion times of the jobs are equal to $2C_k$ and $2C_j$ now. The start time of job J_j is two times larger now than it was at machine with unit speed.

Further, just for the sake of simplicity, we interpret $G_i(C_j)$ only as a cost we should pay for execution of job J_j if its completion time is C_j. This implies that we consider unit machine speeds.

Simple interchange arguments show that there exists an optimal schedule where every machine processes jobs in nondecreasing order of their processing times. Indeed, let jobs J_l and J_k are executed by the same machine in some schedule C. Assume that $p_l \geq p_k$ and the job J_l is executed strictly before J_k (I.e, there are no intermediate jobs between J_l and J_k.) Let S_l be a start time of job J_l in the schedule C. Exchange the order of job processing so as J_k is executed strictly before J_l and the new start time of J_k is equal to S_l and the new completion time of J_l equals to the completion time C_k of job J_k in the schedule C. Denote the obtained schedule by C'. In the new schedule completion times of all jobs excepting J_l and J_k remain unchanged. The idle time between J_l and J_k also remains the same. The difference Δ between the objective value of new schedule and the objective value of the old schedule is evaluated as

$$\sum_{j=1}^{n} G_{A_C(j)}(C'_j) - \sum_{j=1}^{n} G_{A_C(j)}(C_j) = G_{A_C(l)}(C'_l) + G_{A_C(k)}(C'_k) - G_{A_C(l)}(C_l) - G_{A_C(k)}(C_k).$$

By construction, $C'_l = C_k$ and therefore, since J_l and J_k are executed by the same machine, $G_{A_C(l)}(C'_l) = G_{A_C(k)}(C_k)$. Note that

$$C'_k = S_l + p_k = C_l + p_k - p_l.$$

As a result we have

$$\Delta = G_{A_C(k)}(C'_k) - G_{A_C(l)}(C_l) = G_{A_C(k)}(C_l + p_k - p_l) - G_{A_C(k)}(C_l).$$

Since $p_k \leq p_l$ and functions $G_i(\cdot)$ are nondecreasing, inequality $\Delta \leq 0$ holds which means that by the job interchange we have obtained a schedule not worse than the original one. In that way, if jobs are enumerated so that $p_1, \ldots, p_n$, we may impose Restriction 3.3.1 on feasible solutions without loss of an optimal one. If we assume that for any instance an additional restriction

$$G_{i=1}(\alpha\beta) \leq G_{i=1}(\alpha)\beta^{\mu} \tag{3.34}$$

is satisfied for some constant μ, then we have all necessary components to state, using the previously obtained results, that there exists a fully polynomial time approximation scheme for the scheduling problem $P2||\sum G_j(C_j)$ provided that functions G_j are polynomially computable.

Note that we may also consider rational processing times p_j. Then we can multiply processing times by a common multiple to obtain integer values. To come to a problem which is equivalent to the original one, it remains to modify functions $G_j(\cdot)$ by means of dividing their arguments by the mentioned common multiple. This transformation preserves property (3.34).

3.3.4 Conclusions

In this subsection, a multi-machine scheduling problem with a nonlinear objective has been considered. A fully polynomial time approximation scheme has been derived for the two-machine case. It remains unclear whether a similar approach works in the case of an arbitrary fixed number of machines. To try to find an answer to this question, according the properties obtained in this section, it is sufficient to find out if it is possible to describe the function on a box in the same way as it has been done for two machines. (Remind that for two machines the fact that domains of the recursive functions lay in the (two-dimensional) plane essentially helped us.)

Summary

The present thesis summarizes a part of the recent research conducted by the author. Single-item economic lot-sizing and some scheduling problems are the main focus of the thesis. This of course does not mean that the obtained results are useless for other optimization problems. For instance, the fully polynomial time approximation scheme for the CELS problem developed in Section 2.5 can be used as a local search procedure in large-scale neighborhoods for problems that admit a formulation in the form of a nonlinear network problem. An FPTAS in Section 3.3 is relevant for any problem possessing a recursive formulation with properties similar to those considered in that section.

The idea of dynamic programming algorithms is to use optimal solutions of subproblems to obtain an optimal solution of an original problem. Subproblems are constructed by means of omitting certain restrictions and imposing new ones containing, as a rule, new parameters. The main method used in this thesis is dynamic programming in combination with a sensitivity analysis directed at investigating how optimal values of such subproblems behave when values of these parameters are changed. In one form or another similar approaches are already present in the literature (see for instance the paper by Woeginger [61]). In our particular case, the aim of the sensitivity analysis is to reduce the amount of redundant information in the data structures needed for representation of recursive functions on which dynamic programming algorithms are based. Unlike many approaches suggesting to forbid some points in feasible domains of the recursive functions, most of our approaches do not assume throwing out feasible points. This allows us to avoid many difficulties that may arise when a feasible domain of a problem is changed.

In most of the algorithms considered in the thesis, recursive functions are stored as lists of

primitive items like integer intervals or integer boxes associated with corresponding function values. In many cases the amount of these items depends on the structure of the objective function which often allows to increase the efficiency of dynamic programming algorithms. Another observation which also helps to improve dynamic programming algorithms is that in many situations a corresponding recursive function repeats the structure of an objective function. Namely, if an objective function is concave or linear, it makes sense to try to investigate on which intervals the recursive function is concave or linear.

One of the simplest problems to which our methods are applicable is the CELS problem with a linear objective function. This problem can easily be formulated in the terms of polymatroids and solved by an efficient implementation of Edmond's greedy algorithm (see Girlich et al. [30]). An alternative algorithm of the same efficiency can be based on a straightforward dynamic programming algorithm using the fact that the corresponding recursive function is convex for any time period and, moreover, consists of at most n linear pieces where n is the number of time periods. These properties are proven in Section 2.2 and lead us to an $O(n \log n)$ algorithm provided that we additionally use a certain data structure for the sake of efficient implementation of our algorithm. A simple pseudopolynomial dynamic programming algorithm is convertible to a polynomial one for the linear variant of the CELS problem.

Enumerative algorithms is another area where our sensitivity analysis may be useful. Such algorithms are often most straightforward and easily implementable. In many situations they may perform however better than polynomial or pseudopolynomial algorithms. For example, an instance of the knapsack problem with 10 items having integer weights ranging from 1 to 10000 can be solved by some pseudopolynomial algorithm in at least 10000 basic operations. In the same time a proper implementation of the complete enumeration of all solutions allows to perform at most $O(2^{10})$ basic operations to find an optimal solution which is much better than the pseudopolynomial estimate. For many problems there are techniques intended to expand the range of applicability of the exhaustive search. For example, such techniques are used in the algorithm of Horowitz and Sahni [34] running in $O(n2^{n/2})$ time for any instance of the knapsack problem with n items. This algorithm performs $O(20 \cdot 2^{10})$

basic operations to find an optimal solution if $n = 20$. The exhaustive search algorithm needs $O(2^{20})$ basic operations in this case. One can note that complexity estimates $O(n2^{n/2})$ for the algorithm of Horowitz and Sahni and $O(2^n)$ for the straightforward exhaustive search algorithm differ by an exponential factor, i.e., the difference between these two algorithms is similar to the difference between a polynomial algorithm and an exponential one. To emphasize such difference, algorithms like one designed by Horowitz and Sahni are often called subexponential algorithms. In this thesis we suggest an algorithm for the CELS problem being subexponential and pseudopolynomial in the same time. This algorithm, that is suggested in Section 2.3, relies upon a combination of two variants of recursion. One of them uses enumeration of time periods in the increasing order, and another considers time periods in the decreasing order. To avoid redundant computations, the algorithm, using sensitivity properties of the problem, tries to distinguish intervals where the recursive function is linear. As a result of this approach, we decrease the power in the complexity estimation of the trivial enumerative procedure by the factor of two and obtain thereby a subexponential algorithm from an exponential one. Such algorithm can be useful if the number of periods is relatively small and coefficients in the input data are too large to successfully apply a pseudopolynomial algorithm.

The CELS problem with piecewise concave cost functions is an example of a situation when the complete information about a recursive function is not necessary to obtain an optimal solution by a dynamic programming algorithm. Using concavity properties, in Section 2.4 it is shown that one may compute a recursive function only at certain points without loss of an optimal solution. The choice of these points depends on those where cost functions are not concave. The algorithm presented in Section 2.4 runs in polynomial time for much more general variants of the CELS problem with concavity properties of cost functions than those for which polynomial algorithms were already devised and described in the literature. Note that the variant of the CELS problem in Section 2.4 can be used to approximate less structured instances of the CELS problem.

The algorithm developed in Section 2.5 is a fully polynomial time approximation scheme (FPTAS) which is, by the definition, a polynomial algorithm for any desired precision of

approximation. The FPTAS in Section 2.5 is applicable to a general case of cost functions when only most reasonable monotone properties are assumed. The FPTAS is based on solving a problem with the rounded objective function and uses an efficient description of a recursion.

If we allow a more general dependence of the inventory on production than one in the classical formulation of the CELS problem, then we come to a problem for which there is no polynomial algorithm with a constant performance estimate unless P=NP. (This was shown in Section 2.6.) There are however important special cases of that problem for which FPTASs exist. For instance, there exists an FPTAS for the CELS problem where additionally product losses are possible. This FPTAS follows from sensitivity properties of the problem.

Chapter 3 deals with multi-machine scheduling problems. In Section 3.2 local search procedures in large-scale neighborhoods are considered. One of these procedures is a pseudopolynomial dynamic programming algorithm which is convertible, if it is necessary, into an FPTAS by means of using simple sensitivity analysis based on monotone properties of a recursive function. Although the local search procedures have been developed for single-stage scheduling problems, they are also applicable to multi-stage problems as it was shown by an example of a three-stage multi-machine flow-shop problem of minimizing the average flow time of jobs.

The FPTAS in Section 3.3 can be applied to two-machine scheduling problems with nonlinear objective functions of a sufficiently wide class. This FPTAS is based on the calculation of a recursive function whose two arguments may take pseudopolynomially many different values on the feasible domain. As it has been shown, the FPTAS in Section 3.3 is applicable to the case with time-varying speeds of machines.

A combination of a sensitivity analysis of the sort we consider may in that way be successful method to improve running time of dynamic programming algorithms and even change their complexity status.

Bibliography

[1] A. Atamtürk and D.S. Hochbaum. Capacity acquisition, subcontracting and lot sizing. *Management Science*, **47**:1081–1100, 2001.

[2] S. Axäter. Performance bounds for lot sizing heuristics. *Management Science*, **5**:634–640, 1985.

[3] K.R. Baker, P. Dixon, M.J. Magazine, and E.A. Silver. An algorithm for the dynamic lot-size problem with time-varying production capacity constraints. *Management Science*, **24**:1710–1720, 1978.

[4] R. Bayer and E.M. McCreight. Organisation and maintenance of large ordered indexes. *Acta Informatica*, **1**:173–189, 1972.

[5] G. Belvaux and L.A Wolsey. Modelling practical lot-sizing problems as mixed-integer programs. *Management Science*, **47**:993–1007, 2001.

[6] G.B. Bitran and H. Matsuo. Approxomation formulations for the single-product capacitated lot size problem. *Operations Research*, **34**:63–74, 1986.

[7] G.B. Bitran and H.H. Yanasse. Computational complexity of the capacitated lot size problem. *Management Science*, **28**:1174–1186, 1982.

[8] G.R. Bitran, T.L. Magnanti, and H.H. Yanasse. Approximation methods for the uncapacitated dynamic lot size problem. *Management Science*, **9**:1121–1140, 1984.

[9] J. Błażewicz, K.H. Ecker, E. Pesch, G. Schmidt, and J. Węglarz. *Scheduling computer and manufacturing processes*. Springer, 2 edition, 2001.

[10] S.A. Brah and J.L. Hunsucker. Branch and bound algorithm for the flow shop with multiple processors. *European Journal of Operational Research*, **51**:88–99, 1991.

[11] J.L. Bruno, E.G. Coffman, and R. Sethi. Scheduling independent tasks to reduce mean finishing time. *Communications of the ACM*, **17**:382–387, 1974.

[12] S. Canbäck. *Managerial diseconomies of scale: literature survey and hypotheses anchored in transaction cost economics*. Henley Management College, 2004. Working paper.

[13] H.-D. Chen, D. Hearn, and C.-Y. Lee. A dynamic programming algorithm for dynamic lot-size models with piecewise linear costs. *Journal of Global Optimization*, **4**:397–417, 1994.

[14] S. Chubanov. A polynomial algorithm for a capacitated economic lot-sizing problem with piecewise concave cost functions. Working paper.

[15] S. Chubanov, M.Y. Kovalyov, and E. Pesch. An FPTAS for the single-item capacitated economic lot-sizing problem with monotone cost structure. To appear in Mathematical Programming.

[16] S. Chubanov, M.Y. Kovalyov, and E. Pesch. A single-item economic lot-sizing problem with a non-uniform resource: approximation. Working paper.

[17] C.-S. Chung and C.-H. M. Lin. An $O(T^2)$ algorithm for the NI/G/NI/ND capacitated lot size problem. *Management Science*, **34**:420–426, 1988.

[18] M. Constantino. A cutting plane approach to capacitated lot-sizing with start-up costs. *Mathematical Programming*, **75**:353–376, 1996.

[19] M. Constantino. Lower bounds in lot-sizing models: polyhedral study. *Mathematics of Operations Research*, **1**:101–118, 1998.

[20] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, **1**:269–271, 1959.

[21] W. Domschke and A. Drexl. *Einführung in Operations Research*. Springer, 6 edition, 2005.

[22] J. Edmonds. Maximum matching and a polyhedron with 0,1-vertices. *Journal of Research National Bureau of Standards Section B*, **69**:125–130, 1965.

[23] J. Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, **14**:449–467, 1965.

[24] S.S. Erenguc and Y. Aksoy. A branch and bound algorithm for a single item nonconvex dynamic lot sizing problem with capacity constraints. *Computers and Operations Research*, **17**:199–210, 1990.

[25] M. Florian and M. Klein. Deterministic production planning with concave costs and capacity constraints. *Management Science*, **18**:12–20, 1971.

[26] M. Florian, J.K. Lenstra, and A.H.G. Rinnooy Kan. Deterministic production planning: Algorithms and complexity. *Management Science*, **26**:669–679, 1980.

[27] M.R. Garey and D.S. Johnson. 'Strong' NP-completeness results: motivation, examples, and implications. *Journal of the ACM*, **25**:499–508, 1978.

[28] M.R. Garey and D.S. Johnson. *Computers and intractability - A guide to the theory of NP-completeness*. W.H. Freeman, San Francisco, 1979.

[29] B. Gavish and R.E. Johnson. A fully polynomial approximation scheme for single-product scheduling in a finite capacity facility. *Operations Research*, 38:70–83, 1990.

[30] E. Girlich, M. Höding, A. Zaporozhets, and S. Chubanov. A greedy algorithm for capacitated economic lot-sizing problems. *Optimization*, **2**:241–249, 2003.

[31] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling theory: a survey. *Annals of Discrete Mathematics*, **5**:287–326, 1979.

[32] F.W. Harris. How many parts to make at once. *Factory – The Magazin of Management*, **10**:135–136, 1913.

[33] M. Held and R.M. Karp. The construction of discrete dynamic programming algorithms. *IBM Systems Journal*, **4**:136–147, 1965.

[34] E. Horowitz and S. Sahni. Computing partitions with applications to the knapsack problem. *Journal of the ACM*, **21**:277–292, 1974.

[35] O.H. Ibarra and C.E. Kim. Fast approximation algorithms for the knapsack and sum of subset problem. *Journal of the ACM*, **22**:463–468, 1975.

[36] A. Janiak and M.Y. Kovalyov. Single machine scheduling subject to deadlines and resource dependent processing times. *European Journal of Operational Research*, **94**:284–291, 1996.

[37] O Kirca. An efficient algorithm for the capacitated single item dynamic lot size problem. *European Journal of Operational Research*, **45**:15–24, 1988.

[38] T. Kis and E. Pesch. A review of exact solution methods for the non-preemptive multiprocessor flowshop problem. *European Journal of Operational Research*, **164**:592–608, 2005.

[39] B. Korte and J. Vygen. *Combinatorial optimization: theory and algorithms*. Springer, New York, 2002.

[40] M.Y. Kovalyov and W. Kubiak. Fully polynomial time approximation schemes for decomposable partition problems. In K. Inderfurth, G. Schwodiauer, W. Domschke, F. Juhnke, P. Kleinschmidt, and G. Wäscher, editors, *Operations Research Proceedings 1999*, pages 397–401. Springer, 2000.

[41] Ch.-L. Li, V.N. Hsu, and W.-Q. Xiao. Dynamic lot sizing with batch ordering and truckload discounts. *Operations Research*, **52**:639–654, 2004.

[42] V. Lotfi and Y.-S. Yoon. An algorithm for the single item capacitated lot sizing problem with concave production and holding costs. *Journal of the Operational Research Society*, **45**:934–941, 1994.

[43] Kovalyov M.Y. Improving the complexities of approximation algorithms for optimization problems. *Operations Research Letters*, **17**:85–87, 1995.

[44] Punnen A.P. Orlin, J.B. and A.S. Schulz. Approximate local search in combinatorial optimization. *SIAM Journal of Computing*, **33**:1201–1214, 2004.

[45] M. Pinedo. *Scheduling: theory, algorithms, and systems*. Prentice Hall, 1995.

[46] Y. Pochet. Valid inequalities and separation for capacitated economic lot-sizing. *Operations Research Letters*, **7**:109–116, 1988.

[47] Y. Pochet and L.A. Wolsey. Lot-sizing with constant batches: formulation and valid inequalities. *Mathematics of Operations Research*, **18**:767–785, 1993.

[48] K. Rosling. A capacitated single-item lot-size model. *International Journal of Production Economics*, **30-31**:213–219, 1993.

[49] S. Sahni. Algorithms for scheduling independent tasks. *Journal of the ACM*, **23**:114–127, 1976.

[50] T. Sawik. Mixed integer programming for scheduling surface mount technology lines. *International Journal of Production Research*, **39**:3219–3235, 2001.

[51] A.S. Schulz. Polytopes and scheduling. 1996. Ph.D. thesis.

[52] W.E. Smith. Various optimizers for single-stage production. *Naval Research Logistics Quarterly*, 3:59–66, 1956.

[53] H. Stadtler. The impact of deviating from the EOQ on total variable costs. 2006. Technical Note.

[54] V.S. Tanaev, M.Y. Kovalyov, and Y.M. Shafransky. *Scheduling theory. Group technologies*, pages 41–44. Minsk, IEC NANB, 1998. In Russian.

[55] H. Tempelmeier. *Material-Logistik*. Springer, 6 edition, 2006.

[56] U. Thonemann. *Operations Management: Konzepte, Methoden und Anwendungen*. Pearson Studium, 2005.

[57] C. P. M. Van Hoesel and A. P. M. Wagelmans. Fully polynomial approximation schemes for single-item capacitated economic lot-sizing problems. *Mathematics of Operations Research*, **26**:339–357, 2001.

[58] C.P.M. Van Hoesel and A.P.M. Wagelmans. An $O(T^3)$ algorithm for the economic lot-sizing problem with constant capacities. *Management Science*, **42**:142–150, 1996.

[59] H. M. Wagner and T. M. Whitin. Dynamic version of the economic lot size model. *Management Science*, **5**:89–96, 1958.

[60] B. Wardono and Y. Fathi. A tabu search algorithm for the multi-stage parallel machine problem with limited buffer capacities. *European Journal of Operational Research*, **155**:380–401, 2004.

[61] G.J. Woeginger. When does a dynamic programming formulation guarantee the existence of an FPTAS? *Journal on Computing*, **12**:57–74, 2000.

[62] W. Zangwill. From EOQ towards ZI. *Management Science*, **33**, 1987.

[63] W.I. Zangwill. A deterministic multi-period production scheduling model with backlogging. *Management Science*, **13**:105–119, 1966.

[64] W.I. Zangwill. The piecewise concave function. *Management Science*, **13**:900–912, 1967.