Master's Thesis

On the In-Flight Rest Assignment Problem

Ron Wenzel, B.Sc.
ron.wenzel@fu-berlin.de

March 1, 2016

Supervisors:
Prof. Dr. Ralf Borndörfer*
Prof. Dr. Günter Rote†

Freie Universität Berlin
Department of Computer Science

*Zuse Institute Berlin, Department for Mathematical Optimization
†Freie Universität Berlin, Institut für Informatik

Danksagung

Mein Dank gilt Prof. Dr. Ralf Borndörfer und Dr. Marika Karbstein für die Betreuung, die vielen Besprechungen, und hilfreichen Tipps und Ratschläge, die meine Arbeit positiv gelenkt haben.

Dank gilt auch meiner Familie, deren Unterstützung mir mein Studium und diese Arbeit ermöglicht hat.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben. Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Berlin, March 1, 2016

Ron Wenzel

Abstract

The airplane has changed the world in a tremendous way. Efficient scheduling of airmen and aircrafts is of considerable importance for cost-effectiveness of companies.

Attentiveness of flight crew members is vital as fatigue can lead to severe accidents. Therefore, duty times of flight crews are strictly limited. Long distance flights may be difficult to schedule with only one set of crew members. Furthermore, pertubations of the schedules may entail exchanging the entire crew, which confounds multiday schedules. A new EU regulation introduced in-flight rest: a schedule may extend pilots' duty times if they rest for a certain time in designated crew compartments provided aboard airplanes. Of course they have to be replaced in that period of time.

This thesis examines the in-flight rest assignment problem, which is the decision problem whether a given schedule allows for all crew members to take their compulsory rest. The problem can be seen as multimachine scheduling problem. Efficient algorithms for special cases were developed and an alternative approach for entire hard cases is discussed.

Preface

The following thesis is written in the first-person plural in order to include the reader in the thoughts and ideas.

Table of Contents

1 Introduction **1**

1.1 Motivation . . . 1

1.2 Outline . . . 2

2 In-Flight Rest **3**

2.1 European Regulations . . . 3

2.1.1 Cockpit Crew . . . 5

2.1.2 Cabin Crew . . . 5

2.2 Mathematical Model . . . 7

2.3 Simplified Model . . . 9

2.3.1 Extended Simplified Model . . . 12

2.4 Complexity . . . 13

3 Related Problems **15**

3.1 Assignment Problems . . . 15

3.1.1 Assignment Problem . . . 15

3.1.2 Generalised Assignment Problem . . . 16

3.2 Machine Scheduling Problems . . . 18

3.2.1 General Problem . . . 18

3.2.2 High-Multiplicity . . . 20

4 Algorithms for the In-Flight Rest Problem **23**

4.1 Rest for Cockpit Crew and Free Choice of Accommodation . . . 23

4.1.1 Problem . . . 23

4.1.2 Algorithm . . . 25

4.1.3 Correctness and Running Time . . . 28

4.2 In-Flight Rest for Two Rest Length and Free Choice of Accommodation 38

4.2.1 Problem . . . 38

4.2.2 Algorithm . . . 39

4.2.3 Best-Filling Initialisation . . . 43

4.2.4 Generation of Rules . . . 44

4.2.5 Correctness and Running Time . . . 48

4.3 Algorithm for Simple IFR 50

5 Computational Results **53**
5.1 Number of Rules 53
5.2 Runtime of the Algorithm 4.2 55

Bibliography **59**

A Examples **65**
A.1 Example for Algorithm 4.1 65
A.2 Number of Rules 68

B Source Code **71**
B.1 Python Code for Algorithm 4.1 71
B.2 Python Code for Algorithm 4.2 72
B.3 Python Code for Converting Simple IFR to GAP 75

1 Introduction

1.1 Motivation

In the previous century the invention of the airplane changed the world in a tremendous way. While in the early age of aviation civil flights were reserved for the wealthy, they have now become affordable for most people in the world. Increasing aircrafts activities each year reached an average of more than 100'000 flights per day in 2014 [Sta14].

The airline industry demands for fast ways of scheduling their fleet and their personnel in a cost-efficient manner. Although scheduling issues emerge in different fields of transportation, scheduling in aviation is of major importance. The costs for aircraft maintenance and operation are very high due to demanding safety standards and pilots earning top wages.

At the beginning schedules where made by hand but improving computers enabled algorithmic solutions to the problem. More complex computers allow for more and more complex mathematical models.

Most existing solutions split the big problem of airline scheduling into different subproblems: schedule generation, fleet assignment, maintenance routing, crew scheduling [Bar+03]. First, all appropriate connections according to market demands are generated. Then estimated number of passengers or freight determine the adequate selection of the aircraft types. The exact assignment of aircrafts to the fleet is done in the next step where routine maintenance checks in specific airports are considered as well. Last, the crew members for each aircraft are assigned to specific flights. An

efficient interface between these subproblems is vital for an efficient solution to the complete problem.

A big problem are pertubations of the calculated schedule due to bad weather conditions, accidents, strikes, and other unpredictable events. The maximal duty times of pilots are strictly limited by the government. Long waiting times may cause an exchange of the crew if they are not able to complete their schedule. This is quite likely to happen during the winter time as de-icing procedures for airport and aircraft often delay departure times.

A relatively new problem is fatigue management in airline operations. Unadapted sleep cycles and jet leg reduce the attentiveness of crew members. Especially for crew members in command of operating the aircraft this is crucial. Extended duty times increase the risk of accidents enormously.

Even short naps may recover concentration and attentiveness in prolonged work periods [Din+88]. Current legislation allows for extended duty times for cockpit and cabin crew if they rest within their working schedule. This establishes a new subproblem to the airline scheduling problem: the in-flight rest assignment problem (IFR). The problem considers a rest schedule for crew members in available rest compartments during a flight, and is closely related to the multimachine scheduling problem.

This thesis considers the decision problem whether in-flight rest is possible in given schedule. The entire problem is NP-complete and therefore no polynomial algorithm can exist if P $\neq$ NP. IFR can be represented by an integer program. In the past decades much research was performed in integer programming, thus, good algorithms like the simplex algorithm, which is fast in most cases, have been developed.

In this thesis we take a different approach: we focus on cases which exploit the structure of IFR, and solve them in polynomial time.

1.2 Outline

In the second chapter we present some legal information on the in-flight rest problem, and develop mathematical models on this basis. In the third chapter we survey related problems and give a literature overview. The fourth chapter presents polynomial algorithms that solve easy cases of the in-flight rest assignment problem and shows an approach how to solve the hard cases. The last chapter examines the performance of the developed algorithms.

2 In-Flight Rest

2.1 European Regulations

The European Commission issued an EU regulation in 2014 concerning fatigue management in airplane operation. This regulation applies to cockpit crew and cabin crew in commercial air transportation, as well as, charter operations [Eur14a]. At the same time the European Aviation Safety Agency (EASA) published certification specifications and guidance material [Eur14b] concerning the aforementioned EU regulation.

A proper fatigue management is vital to the safety of crew and passengers. Therefore the current framework of legislation provides for a restriction of flight duty periods (FDP)[1] of cockpit crew and cabin crew. Both crew types are regulated differently. The airline has to prevent fatigue caused by night duties by granting sufficient leisure time. Also, alternating night and day shifts, like in police services, are to be prevented in order to circumvent sleep disorders of crew members [Eur14b].

Therefore, the maximum permitted daily FDPs are dependent on the status of acclimatisation of the crew member to the time zone, the start time of the FDP, and the number of included flight legs[2] in the schedule of the FDP.

Table 2.1 shows the maximum duty times for crew members, who are acclimatised to a time zone. Unacclimatised crew members or crew members with an unknown state may only work, as if they started their FDP at the worst possible time: 17:00 - 4:59. While FDPs with many legs tend to be domestic, and therefore provide

[1] A flight duty period (FDP) is the total time of a work day including all flight sectors.
[2] A flight leg is a single flight from a departure airport to a destination airport.

Table 2.1: Maximal flight duty times for acclimatised crew members [Eur14a]

start of FDP	**1-2 legs**	**3 legs**	⋯	**10 legs**
5:00 - 5:14	12 hrs	11.5 hrs	⋯	9 hrs
⋮	⋮	⋮		⋮
6:00 - 13:29	13 hrs	12.5 hrs	⋯	9 hrs
⋮	⋮	⋮		⋮
15:00 - 15:29	12 hrs	11.5 hrs	⋯	9 hrs
⋮	⋮	⋮		⋮
17:00 - 4:59	11 hrs	10.5 hrs	⋯	9 hrs

a lot of opportunities to exchange the crews, it can be quite challenging to find an appropriate crew for long distance flights as the crew cannot be changed while airborne. Furthermore exchange crews have to be moved to oversea locations in advance.

Disturbances in air traffic can cause huge challenges to proper scheduling. If pertubations occur, e.g., bad weather conditions that entail long waiting times at the airport, the crew sometimes has to be exchanged before the first take-off in their schedule.

However, under certain circumstances prolonged FDPs are permitted in compliance with legislation by introducing in-flight rest. The FDP must be limited to 3 sectors, and appropriate rest facilities must be provided. As high quality rest facilities are not always available on board, rest in low comfort compartments is rewarded with lower duty time extensions than rest in high comfort cabins.

Generally, the rest compartments of cockpit crew and cabin crew are separated. Each of those may provide seats of different classes and in different quantity. For flight crew members the quality of the rest facilities influences the maximal duty time, and for cabin crew members it influences the minimum rest time, and therefore indirectly the maximum duty times. Aircrafts may be equipped with the following categories of rest facilities [Eur14b]:

Class 3 rest facility Maximum inclination of 50°, leg and foot support, separation at least by a curtain to provide darkness, some sound mitigation, and not adjacent to seats occupied by passengers.

Class 2 rest facility (additional to class 3 requirements): Maximum inclination of 45°, minimal size, reasonably free from disturbances.

Class 1 rest facility Flat or nearly flat surface (maximum angle of inclination of 10°), separated compartment, adjustable light, isolation from noise and disturbances.

2.1.1 Cockpit Crew

The rest times of cockpit crew members can only take two possible values. The crew members in control of the aircraft during take-off and landing require a rest period of 120 minutes, all other crew members of the flight crew need 90 minutes of rest. For crew members to take their rest it is an obvious necessity that additional crew members are scheduled on the flight, to replace them in their off-time. The maximal FDPs can be lengthened according to table 2.2.

A picture of a rest compartment for cockpit crew can be seen in fig. 2.1a.

Table 2.2: Maximum extentended FDP of cockpit crew due to in-flight rest [Eur14b]

	rest facility		
	Class 1	**Class 2**	**Class 3**
one additional crew member	16 hrs	15 hrs	14 hrs
two additional crew members	17 hrs	16 hrs	15 hrs

2.1.2 Cabin Crew

For cabin crew members the necessary rest is determined by the length of the flight duty time and the available seat classes. Table 2.3 gives an overview of the minimal rest times.

An attentive observer might see that the increase of the rest time for extended duty times is nearly 1.15-fold from 14:31 hrs onwards.

A rest compartment for cabin crew is pictured in fig. 2.1b.

(a) cockpit crew (b) cabin crew

Figure 2.1: Rest compartments aboard of a Boeing 747-830 of Deutsche Lufthansa, put to service in 2013 (photography: Ralf Borndörfer, 2015)

Table 2.3: Minimum in-flight rest for cabin crew member [Eur14b]

flight duty time	**rest facility**		
	Class 1	**Class 2**	**Class 3**
up to 14:30 hrs	1:30	1:30	1:30
14:31 hrs - 15:00 hrs	1:45	2:00	2:20
15:01 hrs - 15:30 hrs	2:00	2:20	2:40
15:31 hrs - 16:00 hrs	2:15	2:40	3:00
16:01 hrs - 16:30 hrs	2:35	3:00	—
16:31 hrs - 17:00 hrs	3:00	3:25	—
17:01 hrs - 17:30 hrs	3:25	—	—
17:31 hrs - 18:00 hrs	3:50	—	—

2.2 Mathematical Model

The in-flight rest problem in this thesis does not consider how valid schedules for pilots may be reckoned. We focus on the decision problem whether a given set of schedules fulfils the legal requirements for in-flight rest, i.e., if it is possible to assign every crew member a proper time and place for his repose.

Subsequently we will develop a mathematical model to describe the in-flight rest problem.

As flight crew and cabin crew perform their rest in different compartments the problem can be split into two instances. The models that we create apply on both crew types. They do not consider crew types differences as we perform a separate calculation for the two groups.

The general problem is as follows:

2.1 Problem – In-Flight Rest Assignment Problem (IFR)

We are given the sets $C = \{1, \ldots, n\}$ of crew members, $L = \{1, \ldots, m\}$ of flight legs and $T = \{1, 2, 3\}$ of seat classes.

Each crew member $c \in C$ has the following properties:

$L_c \subseteq L$ as set of legs that are part of c's schedule, and

$r_c \in \mathbb{N}$ as required rest.

Each seat class $t \in T$ defines a function

$e_t : \mathbb{N} \mapsto \mathbb{N}$ which defines the extension of the rest time due to the seat class.

Each leg $l \in L$ has the following properties:

$s_{l,t} \in \mathbb{N}$ as number of available seats of class $t \in T$,

$p_l \in \mathbb{N}$ as number of crew members allowed for simultaneous rest, and

$A_l \subseteq \{0, 1, \ldots, d_l - 1\}$ as set of points in time at which rest is allowed (excluding *service times*[a]), where $d_l \in \mathbb{N}$ is the time length of leg l.

We are looking for an assignment $f : C \mapsto L \times T \times \mathbb{N}$ of crew members to leg, seat class, and start time for its rest, obeying

$$\underset{c \in C}{\forall} : f(c) = (l, t, a) \Rightarrow l \in L_c, \tag{2.2.1}$$

$$\underset{c \in C}{\forall} : f(c) = (l, t, a) \Rightarrow [a, \ldots, a + e_t(r_c) - 1] \subseteq A_l, \tag{2.2.2}$$

$$\forall_{l\in L}\ \forall_{t\in T}\ \forall_{a\in A_l} : \left|\left\{c \in C | f(c) = (l,t,a'), a \in [a',\ldots,a' + e_t(r_c) - 1]\right\}\right| \leq s_{l,t}, \tag{2.2.3}$$

$$\forall_{l\in L}\ \forall_{a\in A_l} : \left|\left\{c \in C | f(c) = (l,t,a'), a \in [a',\ldots,a' + e_t(r_c) - 1]\right\}\right| \leq p_l. \tag{2.2.4}$$

[a] take-off or landing for flight crew or serving of scheduled meals for cabin crew

The constraints for a valid solution are in a less formal form:

- the crew member may only rest on a flight which is a part of his schedule (eq. (2.2.1)),
- the complete rest duration must be within the permitted time points available for rest (eq. (2.2.2)),
- we cannot use more rest compartments in parallel than available on the aircraft (eq. (2.2.3)), and
- we need to obey restrictions on how many crew members may rest in parallel as the minimum number of crew members on duty must be strictly adhered to (eq. (2.2.4)).

As observed in the previous chapter the function e_t is only necessary for members of the cabin crew (value about $e_t(x) = 1.15^{t-1}x$). For the cockpit crew the necessary rest does not change in different seat classes.

Not modelled, however, is the possible shortening of admissible duty times of cockpit crew members resting on low-class seats. We cannot influence the underlying airline scheduling problem, and therefore not change the duty time. An higher layer has to make sure that this requirement is adhered to, and appropriate seat classes are available. This seems manageable as the number of pilots on any flight is relatively small (maximum of 3-4 pilots on long distance flights) while a minimum of two pilots in duty is required at all times. Thus, in most situations at the utmost two pilots rest simultaneously.

2.3 Simplified Model

On closer inspection of the problem we can reduce the number of input parameters. We can never use more rest facilities in parallel than legally admissible for simultaneous resting. Thus, we can discard all unusable lower-class seats and supersede p_l:

$$s_{l,t} := \begin{cases} s_{l,t} & \text{for } p_l \geq \sum_{i=1}^{t} s_{l,i} \\ p_l - \sum_{i=1}^{t-1} s_{l,i} & \text{for } p_l < \sum_{i=1}^{t} s_{l,i} \text{ and } p_l - \sum_{i=1}^{t-1} s_{l,i} > 0 \\ 0 & \text{otherwise} \end{cases} \quad ,\forall l \in L, \forall t \in T. \tag{2.3.1}$$

We give the set L a new connotation: now it describes a certain rest accommodation on a certain leg which is continuously available. This works as follows:

- For every rest accommodation on a leg we produce a copy of the leg. In any of those copies only one rest can take place in parallel.
- If no rest is allowed in certain time intervals the rest accommodations are split in two parts at any point of unavailability. Now any rest accommodation $l \in L$ is available continuously.
- If the seat is of a lower seat class than class 1 the total availability d_l is truncated appropriately according to e_t. This is possible as our e_t is approximately linear.

Therefore, the set L_c contains now all the rest accommodations that are available on any flight of its schedule. Thus, we can disregard all leg constraints as now the number of rest facilities is limited to the maximal permitted number of simultaneous rests and no constraints of forbidden time intervals are to be regarded. The complete simple model is as follows:

2.2 Problem – Simple In-Flight Rest Assignment Problem (SIFR)

We are given the sets $C = \{1, \ldots, n\}$ of crew members and $L = \{1, \ldots, m\}$ of rest accommodations,

$P_c \subseteq L$ as set of permitted rest accommodations for each crew member $c \in C$,

$r_c \in \mathbb{N}$ as required rest of crew member $c \in C$, and

$d_l \in \mathbb{N}$ as capacity of rest accommodation $l \in L$.

We are looking for an assignment $f : C \mapsto L$ of crew members to rest accommodations, obeying

$$\underset{c \in C}{\forall} : f(c) \in P_c, \qquad \text{(rest in permitted accommodation)} \qquad (2.3.2)$$

$$\underset{l \in L}{\forall} : \sum_{c \in f^{-1}(l)} r_c \leq d_l. \qquad \text{(honour accommodation's capacity)} \qquad (2.3.3)$$

Members of the cockpit crew have only two possible values for the required rest (see section 2.1.1):

$$r_c \in \{90, 120\}, \qquad \forall c \in C. \qquad (2.3.4)$$

Integer Programming Formulation

For convenience we also present problem 2.2 as an integer programming formulation:

IP-Formulation for SIFR

We introduce a matrix to store the permitted accommodations:

$$p_{c,l} = \begin{cases} 1 & \text{if } l \in P_c \\ 0 & \text{if } l \notin P_c \end{cases}, \qquad \forall c \in C, l \in L$$

We are looking for a decision variable $x_{c,l} \in \{0, 1\}$ for $c \in C$, $l \in L$ subject to:

$$\underset{c \in C}{\forall} : \quad \sum_{l \in L} x_{c,l} = 1 \qquad \text{(one rest per crew member)} \qquad (2.3.5)$$

$$\underset{\substack{c \in C \\ l \in L}}{\forall} : \quad x_{c,l} \leq p_{c,l} \qquad \text{(rest in permitted accommodation)} \qquad (2.3.6)$$

$$\underset{l \in L}{\forall} : \quad \sum_{c \in C} x_{c,l} \cdot r_c \leq d_l \qquad \text{(honour accommodation's capacity)} \qquad (2.3.7)$$

Example

The following example is designed for illustration purposes only as it normally would not require in-flight rest. Bigger instances tend to be less illustrative.

2.1 Example – Instance of a SIFR Problem

We contemplate a problem instance for cockpit crew. Let $C = \{\mathrm{A}, \mathrm{B}, \ldots, \mathrm{F}\}$ be the set of crew members and $C = \{1, 2, \ldots, 6\}$ the set of rest accommoda-

tions. The crew member A, C, D, and F need a rest time of 120 minutes, B, and E need a rest time of 90 minutes.

Let us consider the following flight network.

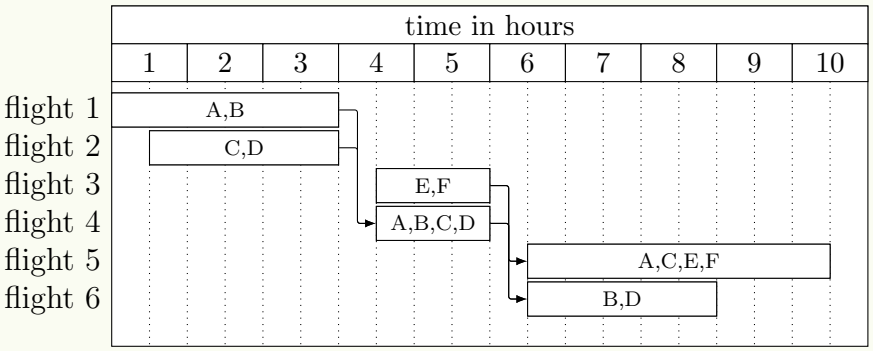


Figure 2.2: Simple flight network

Now we try to find an assignment $f(c)$ from crew members to rest accommodations. The solid arcs denote a valid solution, the dashed arcs are possible assignments.

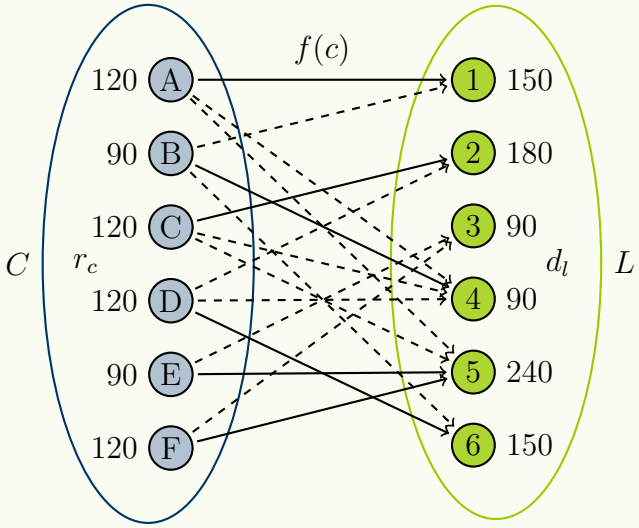


Figure 2.3: Sound solution for the flight network in fig. 2.2.

If we regard the IP formulation we can also create an adjacency matrix:

- in every column only one number can be placed,
- the sum of each column must be at least the number in the heading, and
- the sum of each row must be at most the number on the left.

The grey fields are unfeasible because they are not part of the particular crew members schedule.

lenght	rest	120 **A**	90 **B**	120 **C**	120 **D**	90 **E**	120 **F**
150	**fl. 1**	120					
180	**fl. 2**			120			
90	**fl. 3**						
90	**fl. 4**		90				
240	**fl. 5**					90	120
150	**fl. 6**				120		

Figure 2.4: Adjacency matrix of the problem instance.

2.3.1 Extended Simplified Model

If the extension of the rest time due to the seat class e_t is not linear, as assumed in the previous model, we give an alternative model including this as well. We call this the extended SIFR. It includes seat classes like in IFR and respects prolonged rest times on lower-class seats for non-linear e_t as well.

2.3 Problem – Extended SIFR (eSIFR)

We are given the sets $C = \{1, \ldots, n\}$ of crew members, $L = \{1, \ldots, m\}$ of rest accommodations, and $T = \{1, 2, 3\}$ of seat classes, and

$P_c \subseteq L$ as set of permitted rest accommodations for each crew member $c \in C$,

$r_c \in \mathbb{N}$ as required rest of crew member $c \in C$,

$d_l \in \mathbb{N}$ as capacity of rest accommodation $l \in L$, and

$t_l \in T$ as seat class of rest accommodation $l \in L$.

Each seat class $t \in T$ defines a function

$e_t : \mathbb{N} \mapsto \mathbb{N}$ which defines the extension of the rest time due to the seat class.

We are looking for an assignment $f : C \mapsto L$ of crew members to rest accommodations, obeying

$$\underset{c \in C}{\forall} : f(c) \in P_c, \qquad \text{(rest in permitted accommodation)} \tag{2.3.8}$$

$$\underset{l \in L}{\forall} : \sum_{c \in f^{-1}(l)} e_{t_l}(r_c) \leq d_l. \qquad \text{(honour accommodation's capacity)} \tag{2.3.9}$$

2.4 Complexity

The problems SIFR, eSIFR, and IFR are NP-complete. Garey and Johnson gave in *Computers and intractability* (1979) an overview of different NP-complete problems [GJ79]. One of these problems is multiprocessor scheduling (p. 238) which is basically the same as examined in section 3.2.

Garey and Johnson consider the following problem instance: Let $T = \{1, \ldots, n\}$ be a set of tasks, $m \in \mathbb{N}$ the number of processors, $l_t \in \mathbb{N}$ the length of task t, and $D \in \mathbb{N}$ the deadline for all processors. Is it possible to schedule all tasks on the given processors without violating any processor's deadline?

It is easy to see that multiprocessor scheduling can be solved with SIFR, as it is a special case of it. We only need to set a common duration d_l for all rest accommodations and set $P_c = L$ for any crew member. Thus, it is reducible to SIFR in polynomial time.

$$\text{MPS} \preceq_p \text{SIFR} \tag{2.4.1}$$

We also can solve SIFR with eSIFR and IFR, thus,

$$\text{SIFR} \preceq_p \text{eSIFR}, \tag{2.4.2}$$

$$\text{SIFR} \preceq_p \text{IFR}. \tag{2.4.3}$$

A solution to all these problems can be verified in polynomial time, so they are NP-complete.

Although the entire problems are NP-complete, special cases are solvable in polynomial time. We present efficient polynomial algorithms in chapter 4.

3 Related Problems

3.1 Assignment Problems

Assignment problems are a class of problems where tasks are to be processed by agents. First, we describe the *Assignment Problem* where the set of agents and the set of tasks are of equal size, and each agent has to process exactly one tasks. Then we show a more generalised version of the AP where an agent may process as many tasks as possible without exceeding his capacity.

3.1.1 Assignment Problem

3.1 Problem – Assignment Problem

Let $A = \{1, 2, \ldots, n\}$ be the set of agents and let $T = \{1, 2, \ldots, n\}$ be the set of tasks, both equal in size.

For all $a \in A$ and $t \in T$ we are given

$c_{a,t} \in \mathbb{N}$ as costs if task t is assigned to agent a.

We are looking for a *bijection* $f : T \mapsto A$ minimising the total costs

$$\text{cost}(f) = \sum_{t \in T} c_{f(t),t}.$$

This problem can be solved strongly polynomial. In 1955 Kuhn presented the famous *Hungarian Algorithm* [Kuh55] with a time complexity bounded by $\mathcal{O}(n^4)$ [Fra05].

In 1976 Lawler presented an improved version that runs in $\mathcal{O}(n^3)$ time [JV86].

3.1.2 Generalised Assignment Problem

The generalised assignment problem (GAP) was first introduced by Ross and Soland (1975) as a more generalised version of the assignment problem [RS75]. While each task is assigned to an agent, it is possible to assign multiple tasks to a single agent as long as we comply with the agent's capacity. Each agent has different resource requirements to complete a certain task.

More formally the generalised assignment problem is given as follows:

3.2 Problem – Generalised Assignment Problem

Let $A = \{1, 2, \ldots, m\}$ be the set of agents and let $T = \{1, 2, \ldots, n\}$ be the set of tasks.

For all $a \in A$ and $t \in T$ we are given

$c_{a,t} \in \mathbb{N}$ as costs if task t is assigned to agent a,

$r_{a,t} \in \mathbb{N}$ as resources required for assigning task t to agent a, and

$b_a \in \mathbb{N}$ as resource units available to agent a.

We are looking for an assignment $f : T \mapsto A$ of tasks to agents, obeying

$$\underset{a \in A}{\forall} : \sum_{t \in f^{-1}(a)} r_{a,t} \leq b_a, \qquad \text{(respect agent's capacity)} \tag{3.1.1}$$

minimising the total costs

$$\text{cost}(f) = \sum_{t \in T} c_{f(t),t}.$$

While the assignment problem is solvable in polynomial time, the GAP is NP-complete[1].

Ross and Soland described a branch-and-bound algorithm for solving GAP. They use binary knapsacks as subproblems to determine the bounds.

A similar problem of the GAP is the general transportation problem [RS75; Lou64] where the tasks allow for multiplicities as well, i.e., all tasks j with multiplicity k_j have to be processed by the group of agents k_j times.

[1] The multiprocessor scheduling, which is NP-complete [GJ79], can be reduced to GAP.

Fisher et al. (1986) published a branch-and-bound algorithm [FJV86], which they claim to be much faster than existing algorithms at that time, like Ross and Soland's [RS75] or a branch-and-bound algorithm conceived of by Martello and Toth [MT81]. It is based on lagrangian relaxation and uses stronger bounds which results in two orders of magnitude fewer nodes in the branch-and-bound tree [FJV86].

A new approach was taken by Jörnsten and Näsberg who solved an equivalent problem of the GAP named EGAP (equivalent GAP) [JN86]. This enhances the solution procedure based on lagrangian relaxation.

A great step forward was the algorithm presented by Guignard and Rosenwein [GR89; Ros86]. They claimed that the algorithm is able to process up to 500 decision variables while existing algorithms back then were not capable of processing more than 100 variables. They use a dual lagrangian branch-and-bound algorithm. The improved enumeration procedure for the branch-and-bound tree was already presented in Rosenwein's dissertation [Ros86], where he also reviewed existing algorithms for the GAP.

In 1992 Cattrysse and Van Wassenhove compared different relaxation methods used for bounding as well as branching strategies [CV92] and examined running times of existing algorithms, although no exact comparisons were possible due to different implementation and different computing environments.

Savelsbergh [Sav97] developed a branch-and-price algorithm to generate optimal integer solutions for GAP and discussed different branching strategies.

A further improvement in solving GAP was made by De Farias and Nemhauser (2001). They presented a family of inequalities for the polytope of the general assignment problem [DN01]. Those may be used in a branch-and-cut approach for faster executions and their utility is shown using a branch-and-cut solver.

Pigatti et al. presented a branch-and-cut-and-price algorithm [PDU05] which outperforms previous algorithms and solved three of five unsolved problem instances of the OR-library[2]. They also presented a new method called *ellipsoidal cuts* which helps to create efficient heuristics.

The latest exact algorithm[3] is given by Posta et al. [PFM12]. They solve the optimisation problem *GAP* by solving a series of decision problems. They determine a lower bound $\bar{z}$ for the optimal solution of the problem, and solve the decision problem, whether a solution for $\bar{z}$ exists or not. If no such solution exists they increase the lower bound $\bar{z}$ by one until they obtain a solution.

[2]Library for operations research problems: `http://people.brunel.ac.uk/~mastjjb/jeb/info.html`

[3]according to Sadykov et al. [Sad+15]

The decision problem is solved by a lagrangian branch-and-bound method. They prune the branch-and-bound tree by applying effective variable-fixing rules that exploit the fact that any job is assigned to exactly one agent.

Although their approach of successively solving a decision problem for each bound sounds inefficient, it turns out to be very fast in practice as they obtain a fairly good initial bound. They give some ideas why the algorithm behaves in this manner.

Variants

Krumke and Thielen (2013) examined a variant of the GAP where each agent has to use a minimum of its resource units processing tasks if employed or has to process no tasks at all [KT13]. This is useful if a small workload is undesirable. They consider different versions of this GAP variant and give polynomial time algorithms or approximations.

3.2 Machine Scheduling Problems

3.2.1 General Problem

Machine scheduling is not a single problem but a big set of different problems. Generally it is as follows:

3.3 Problem – Machine Scheduling Problem

A *Machine Scheduling Problem* is the challenge of scheduling n jobs on m machines subject to an optimality criterion. Different constraints may apply on machines and jobs.

A scheme for describing a certain problem was first conceived of by Graham, Lawler et al. [Gra+79] and Lawler et al. [LLK82]: the 3-field problem classification. Nowadays it is the most frequently used classification when discussing machine scheduling.

Notation of machine scheduling problems: 3-field problem classification by Graham et al. [Gra+79; LLK82]

Every machine scheduling problem is expressed by $\alpha|\beta|\gamma$, where α describes the *machine environment*, β the *job characteristics*, and γ denotes the *optimality criteria*.

The *machine environment* α expresses the type of machines in use. The

simplest cases are that we have only one machine ($\alpha = 1$), or that all machines are identical ($\alpha = P$). If the machines work equally, but have a different processing speed, they are called uniform ($\alpha = Q$). If the processing speed is job dependent they are called unrelated ($\alpha = R$). More complex expressions denote multistage machine environments like open shops: a job has to be processed consecutively by different machines.

The *job characteristics* β apply certain constraints on how the jobs are to be scheduled and describe the properties of the jobs themselves. E.g., *job preemption* may allow postponing of jobs for later processing ($\beta = pmtn$). If jobs require resources, these may be restrained. Precedence constraints ($\beta = prec$) can enforce a sequential arrangement in which jobs have to be processed. Job properties may be given by release dates of a job, i.e., the first possible start time of a job, and processing times.

Lastly, the *optimality criteria* γ describes the function that has to be optimised in order to solve the given problem optimally. Prominent examples are minimal makespan and minimal total completion time[a]. The makespan is the time point where the last job exits the system, the total completion time is the sum of all job completion times. Therefore the subject of minimal makespan is to complete all the jobs in a minimal amount of time, while minimal total completion time contemplates the mean completion time of the jobs.

More complex optimisation objectives are due dates dependent, e.g., minimal tardiness, earliness, and so forth.

Examples:

- $1|prec|\sum C_j$: minimal total completion time on one machine with precedence constraint
- $P||C_{\max}$: makespan on identical machines
- $Q|pmtn, prec|\sum C_j$: minimal total completion time on uniform machines with preemption and precedence constraint

[a] The completion time of a job is the time point where the job leaves the system.

Most of the machine scheduling problems are NP-complete [GJ79]. Even if the number of machines is $m = 2$, it is still NP-complete (the subset sum problem can be reduced to the case $m = 2$). Nevertheless efficiently solvable cases exist: for $\mathrm{P}||\sum C_j$ (minimal total completion time on identical machines) an algorithm exists running in $\mathcal{O}(n \log n)$ time [CP01] which was developed by Conway et al. [CMM67].

For a great overview on standard machine scheduling problems the reader is referred to the book of Brucker [Bru07]. Uwazie compiled in his master's thesis (2012) an overview of the state of the art of solutions to minimise the makespan [Uwa12].

3.2.2 High-Multiplicity

For our purpose a specialisation of the machine scheduling problem is far more interesting. We have only a very small set of different jobs and most of the jobs are identical. Especially for cockpit crew only two different rest times are possible, independent of the seat class (see section 2.1.1).

If many jobs are identical they may be partitioned into groups with a single description each. Those problems are referred to as *High-Multiplicity Machine Scheduling Problem*s due to Hochbaum and Shamir [HS91].

Single machines

Among the first who discussed an exploitation of identical jobs in the context of scheduling was Psaraftis in 1980 [Psa80]. He described a dynamic programming approach for scheduling them on a single machine with a basic cost function.

Hochbaum and Shamir first used the term *high multiplicity scheduling problem* for this type of problem in 1991 [HS91]. They applied this concept on a single machine as well and focused on different optimisation objectives, such as minimal weighted number of tardy jobs, and minimal total weighted tardiness. They presented algorithms with polynomial running time.

Shallcross studied a similar problem, where identical jobs are to be grouped into batches and processed together [Sha92]. Each batch has certain setup costs and the jobs in a batch are finished when the last job is finished. The objective is to minimise the average time of completion. He presented an algorithms which runs in polynomial time in the logarithms of the input parameters.

Parallel machines

The concept of high multiplicity encoding was applied on parallel machines by Granot et al. [GST97]. They considered the minimisation of the total weighted completion time. As the entire problem is NP-hard they analysed special cases where the weights of all jobs are equal or all jobs have the same costs, and discovered that those problems can be solved in polynomial time. For the special case of equal costs for all jobs they gave an algorithm running in $\mathcal{O}(mn + n \log n)$ time.

In 2001 Clifford and Posner compiled the complexity state of different high-multiplicity problems and showed the polynomial solvability for some problems [CP01]. They considered, for the optimisation criteria minimisation of total job completion time and minimisation of the makespan, preemptive and non-preemptive versions,

different machines speeds (identical, proportional, and unrelated machines), the decision problems, and the optimisation problems.

Filippi and Romanin-Jacur updated the complexity state of some problems examined by Clifford and Posner (2001) and provided many exact algorithms for polynomial solvable cases and asymptotically exact algorithms for hard cases [FR09]. They stated that all provided algorithm have a polynomial running time and are easy to implement.

In 2005 [Bra+05] introduced a complexity framework for single machine and non pre-emptive high multiplicity scheduling problems [Bra+05]. In 2007 they extended this framework to more general classes of multiplicity scheduling problems [Bra+07].

Fixed number of job execution times

If the number of different job execution times is constant, further optimisation is possible. McCormick et al. refer to this problem as MSPC (*Multiprocessor Scheduling Problem with C job lengths*) [MSS01].

3.4 Problem – Multiprocessor Scheduling Problem with C job lengths

Let C be the number of different job types and let $l_1, \ldots, l_C$ be the distinct job lengths for those types. Furthermore

$n_k \in \mathbb{N}$ is the number of jobs of type $k \in \{1, \ldots, C\}$, and

$$n = \sum_{k=1}^{C} n_k.$$

We have m machines with machine dependent deadlines of $D_k, k \in \{1, \ldots, m\}$, i.e., machine k is continuously available in the time interval $[0; D_k]$. A machine can only process one job at a time.

The question is: does a schedule exist that processes all n jobs on the m machines honouring all machine deadlines?

Leung showed that, if the number of different job types is fixed, the problem can be solved in polynomial time [Leu82]. He presented an algorithm based on dynamic programming with a time complexity of $\mathcal{O}(\log p \cdot \log m \cdot n^{2(C-1)})$, where p is the largest execution time.

McCormick et al. proposed an improved algorithm for the case $C = 2$ (MSP2m) which can be implemented to run in $\mathcal{O}(m \log D_{\max})$ time, where $D_{\max}$ is the latest machine deadline [MSS01]. If $D_1 = D_2 = \ldots D_m = D$ they presented an algorithm that even achieves a running time of $\mathcal{O}(\log^2 D)$. They give counter examples why their idea is not extendible on $C > 2$.

Later we will discuss an algorithm which depends on the job lengths rather than on the machine deadline.

Detti discussed in 2008 a variant of the MSP2 [Det08]: each job may only be processed by a machine belonging to a continuous interval $\{1, 2, \ldots, p\}$ where $p \leq m$. All the machines have the same deadline D. He presented an algorithm with polynomial running time.

4 Algorithms for the In-Flight Rest Problem

4.1 Rest for Cockpit Crew and Free Choice of Accommodation

4.1.1 Problem

We consider a special case of problem 2.2 for flight crew members (see eq. (2.3.4)): we allow any crew member to rest in any rest accommodation, i.e., the crew stays together on the same aircrafts and will not be split during the duty time (eq. (4.1.2)).

We know that flight crew members have only two possible rest durations: 90 minutes and 120 minutes. We notice that the possible rest periods have a greatest common divisor of 30 minutes, so we can regard all the time periods as multiples of 30 minutes. So we can consider the rest durations as the natural numbers 3 and 4.

We restrict the rest durations r_c for the crew members to 3 and 4 time units:

$$\underset{c \in C}{\forall} : r_c \in \{3, 4\}. \tag{4.1.1}$$

As previously announced we allow all crew members to rest in any accommodation, that is

$$\underset{c \in C}{\forall} : P_c = L. \tag{4.1.2}$$

Thus we do not have to calculate the exact assignments. It is sufficient to subdivide the provided capacities d_l of rest compartments $l \in L$ into blocks of size 3 and 4 such that every crew member gets a place to rest. This allows us to group the set crew members and represent each group by a figure:

- Let $q_3 \in \mathbb{N}$ be the quantity of crew members needing a rest of 3 time units.
- Let $q_4 \in \mathbb{N}$ be the quantity of crew members needing a rest of 4 time units.

Furthermore, the vector of durations $(d_l) = (d_1, d_2, \ldots, d_m)$ is now reduced to full 30 minutes time periods, e.g., $(7, 9, 3, 4)$, for possible rest durations of 210, 270, 90, and 120 minutes. So, we want to split all durations d_l into blocks of 3 and 4, i.e., b_l^3 and b_l^4.

The complete problem is as follows:

4.1 Problem

We are given two sets $L = \{1, \ldots, m\}$ of rest accommodations and $R = \{3, 4\}$ of possible rest durations,

$q_r \in \mathbb{N}$ as the quantity of crew members needing a rest of $r \in R$, and

$d_l \in \mathbb{N}$ as capacity of rest accommodation $l \in L$.

We are looking for a subdivision of all $d_l, l \in L$ into r-sized blocks ($r \in R$) denoted as $b_l^r \in \mathbb{N}$, obeying:

$$\underset{l \in L}{\forall} : \sum_{r \in R} r \cdot b_l^r \leq d_l \qquad \text{(valid subdivision)} \tag{4.1.3}$$

$$\underset{r \in R}{\forall} : \sum_{l \in L} b_l^r \geq q_r \qquad \text{(accommodation for every crew member)} \tag{4.1.4}$$

The problem 4.1 is a special case of MSPC (problem 3.4).

Problem 4.1 is special case of MSPC (problem 3.4)

We may solve a problem 4.1 with a problem 3.4 in the following manner:

Problem 4.1 is a instance of MSPC with two possible job lengths: 3, 4. The number of machines and the number of rest accommodations are both m. The machine deadlines D_k are the availabilities of the rest accommodations d_l. The requested number of times slots q_3 and q_4 are the number of jobs n_1 and n_2.

$$C \leftarrow 2, \tag{4.1.5}$$

$$l_1 \leftarrow 3,\ l_2 \leftarrow 4, \tag{4.1.6}$$
$$n_1 \leftarrow q_3,\ n_2 \leftarrow q_4, \tag{4.1.7}$$
$$\forall_{k=l}^{l \in L} : D_k \leftarrow d_l. \tag{4.1.8}$$

4.1.2 Algorithm

Preliminaries

For convenience we define $\tilde{b^r}$ as abbreviation for the total sum of all r-sized blocks in our subdivision.

4.1 Definition

The expression $\tilde{b^r}$ connotes the sum of all b_l^r in our current subdivision:

$$\tilde{b^r} \stackrel{\text{def}}{=} \sum_{l \in L} (b_l^r). \tag{4.1.9}$$

Idea

The idea of the algorithm is that we first split all the capacities of the accommodations into time slots that fill them best. In this way no usable space is wasted. Then we transform this subdivision by applying certain *rules* to transform it into a valid solution.

We refer to these *rules* later and give formal description of the algorithm first.

An exemplary run of the algorithm is provided in section A.1.

4.1 Algorithm – IFR for Flight Crew: Problem 4.1

Input: $d = (d_1, d_2, \ldots, d_m)$ as vector of capacities $d_l \in \mathbb{N} \setminus \{0, 1, 2\}$ of rest accommodations $l \in L$
$q = (q_3, q_4)$ as required number of time slots of length 3 resp. 4

Output: $(b_l^3, b_l^4)_{l \in L}$ as subdivision of $(d_l)_{l \in L}$ satisfying eqs. (4.1.3) and (4.1.4)

▷ Initialise

forall $l \in L$ **do**
  if $d_l \neq 5$ **then**
    $b_l^4 \leftarrow d_l \bmod 3$
    $b_l^3 \leftarrow (d_l - b_l^4 \cdot 4) \div 3$
  else
    $(b_l^3, b_l^4) \leftarrow (0, 1)$

▷ Calculate feasible subdivision

if $\tilde{b}^3 < q_3 \wedge \tilde{b}^4 > q_4$ **then** ▷ blocks of 3 needed
  forall $l \in L$ **do** ▷ rule 5, table 4.2
    while $b_l^4 \geq 1 \wedge \tilde{b}^4 - q_4 \geq 1 \wedge q_3 - \tilde{b}^3 \geq 1$ **do**
      $(b_l^3, b_l^4) \leftarrow (b_l^3 + 1, b_l^4 - 1)$
else if $\tilde{b}^3 > q_3 \wedge \tilde{b}^4 < q_4$ **then** ▷ blocks of 4 needed
  forall $l \in L$ **do** ▷ rule 1, table 4.2
    while $b_l^3 \geq 4 \wedge \tilde{b}^3 - q_3 \geq 4 \wedge q_4 - \tilde{b}^4 \geq 3$ **do**
      $(b_l^3, b_l^4) \leftarrow (b_l^3 - 4, b_l^4 + 3)$
  forall $l \in L$ **do** ▷ rule 2, table 4.2
    while $b_l^3 \geq 3 \wedge \tilde{b}^3 - q_3 \geq 3 \wedge q_4 - \tilde{b}^4 \geq 2$ **do**
      $(b_l^3, b_l^4) \leftarrow (b_l^3 - 3, b_l^4 + 2)$
  forall $l \in L$ **do** ▷ rule 3, table 4.2
    while $b_l^3 \geq 2 \wedge \tilde{b}^3 - q_3 \geq 2 \wedge q_4 - \tilde{b}^4 \geq 1$ **do**
      $(b_l^3, b_l^4) \leftarrow (b_l^3 - 2, b_l^4 + 1)$

▷ Finished

if $\tilde{b}^3 < q_3 \vee \tilde{b}^4 < q_4$ **then**
  ▷ Failed.
if $\tilde{b}^3 \geq q_3 \wedge \tilde{b}^4 \geq q_4$ **then**
  ▷ Success.

Best-filling Subdivision

For this algorithm to work it is crucial that the initial subdivision does not waste any space. Table 4.1 shows such a subdivisions for certain blocks. They are calculated by reducing the number by 4, until 3 divides it. Italicised capacities cannot be used completely as there will always be an unused rest – nevertheless they are optimal.

capacity	*1*	*2*	3	4	*5*	6	7	8	9	10	11	12	(12)	⋯
blocks of 3	*0*	*0*	1	0	*0*	2	1	0	3	2	1	4	(0)	⋯
blocks of 4	*0*	*0*	0	1	*1*	0	1	2	0	1	2	0	(3)	⋯

Table 4.1: Best-filling subdivision into blocks of 3 and blocks of 4

We now give a definition when a subdivision is optimal, i.e., it is best-filling.

4.2 Definition – Best-filling Subdivision of a Duration

Let d denote a duration and b^r a valid subdivision of d, i.e., $\sum_{r \in R} r \cdot b^r \leq d$. b^r is a *best-filling subdivision* iff no valid subdivision $b^{r\prime}$ of d exists with $\sum_{r \in R} r \cdot b^{r\prime} > \sum_{r \in R} r \cdot b^r$.

4.3 Definition – Usable Space of a Duration

Let d denote a duration and b^r a best-filling subdivision of d. Then $\hat{d} = \sum_{r \in R} r \cdot b^r$ is the *maximal usable space* of d.

Transforming the subdivision

Now we can transform these blocks to other blocks in order to satisfy eq. (4.1.4). All transformation rules are shown in table 4.2. Every transformation rule has a penalty which shows how much time is rendered unusable.

There are two types of rules: rules that shift the balance of blocks from blocks of 3 to blocks of 4 (rules 1, 2 and 3) and rules that shift the balance in the opposite way (rules 4 and 5).

The algorithm shifts the balance in the needed direction, therefore we only need one set of rules during a single execution: 1, 2, 3 or 4, 5. We apply step by step the next rule that is applicable on any set of blocks (b_l^3, b_l^4). We prefer rules with lower penalty, as they render less space unusable, i.e., we apply a rule with higher penalty only if no rule with a lower penalty can be applied.

	balance of		penalty	
	blocks of 3	blocks of 4	total	weighted
no. 1	-4	3	0	0
no. 2	-3	2	1	$1/9$
no. 3	-2	1	2	$2/6$
no. 4	4	-3	0	0
no. 5	1	-1	1	$1/4$

Table 4.2: Transformation rules from blocks of 3 to blocks of 4 and vice versa

Our solution is valid if eq. (4.1.4) holds, i.e., we have enough time slots to accommodate all the requested rest periods:

$$q_3 \leq \tilde{b^3} \qquad \wedge \qquad q_4 \leq \tilde{b^4} \tag{4.1.10}$$

If at some point the overall capacity in all blocks is smaller than the overall capacity required, as in

$$3\tilde{b^3} + 4\tilde{b^4} < 3q_3 + 4q_4. \tag{4.1.11}$$

we can never transform it into a valid solution.

Notice that our initialisation prefers blocks of 3 to blocks of 4, for this renders rule 4 of table 4.2 superfluous.

4.1.3 Correctness and Running Time

4.4 Lemma

The subdivision of the initialisation in algorithm 4.1 is best-filling (definition 4.2).

Proof. Let d denote a duration. Our algorithm distinguishes the following cases:

Case 1. $(d \neq 5)$

Our subdivision is (b^3, b^4), where

$$b^4 = d - 3 \cdot \left\lfloor \frac{d}{3} \right\rfloor,$$
$$b^3 = \frac{d - b^4 \cdot 4}{3}.$$

$b^4 \in \mathbb{N}$, because d is a integer. $b^3 \in \mathbb{N}$, because $b^4 \equiv d \bmod 3$ and $4 \equiv 1 \bmod 3$, and therefore $4b^4 \equiv d \bmod 3$.

The complete space used by those blocks is

$$\begin{aligned} 3 \cdot b^3 + 4 \cdot b^4 &= (d - b^4 \cdot 4) + 4 \cdot b^4 \\ &= d, \end{aligned}$$

thus the space is fully used, so there is no other subdivision with a better usage and it is best-filling according to definition 4.2.

Case 2. $(d = 5)$
There is no subdivision which would fully use a duration of 5. So the best-filling subdivision is a subdivision of total size 4, with a single block of 4.

□

4.5 Lemma

At any time in the algorithm 4.1 the following holds true:

$$3 \cdot b_l^3 + 4 \cdot b_l^4 \leq d_l, \qquad \forall l \in L \tag{4.1.12}$$

Proof. As our initialisation procedure is best-filling (lemma 4.4) our condition

$$3 \cdot b_l^3 + 4 \cdot b_l^4 \leq d_l, \qquad \forall l \in L \tag{4.1.12}$$

holds true after initialisation.

There are only 4 lines in algorithm 4.1 which change b_l^3 and b_l^4: lines 10, 14, 17 and 20. Let $b_l^{3'}$ and $b_l^{4'}$ denote the old values prior to the assignment. So, for any $l \in L$ for the new assigned variables the following holds true:

line 10: $3 \cdot b_l^3 + 4 \cdot b_l^4 = 3 \cdot (b_l^{3'} + 1) + 4 \cdot (b_l^{4'} - 1) = (3 \cdot b_l^{3'} + 4 \cdot b_l^{4'}) - 1$

line 14: $3 \cdot b_l^3 + 4 \cdot b_l^4 = 3 \cdot (b_l^{3'} - 4) + 4 \cdot (b_l^{4'} + 3) = 3 \cdot b_l^{3'} + 4 \cdot b_l^{4'}$

line 17: $3 \cdot b_l^3 + 4 \cdot b_l^4 = 3 \cdot (b_l^{3'} - 3) + 4 \cdot (b_l^{4'} + 2) = (3 \cdot b_l^{3'} + 4 \cdot b_l^{4'}) - 1$

line 20: $3 \cdot b_l^3 + 4 \cdot b_l^4 = 3 \cdot (b_l^{3'} - 2) + 4 \cdot (b_l^{4'} + 1) = (3 \cdot b_l^{3'} + 4 \cdot b_l^{4'}) - 2$

Thus, with $3 \cdot b_l^{3'} + 4 \cdot b_l^{4'} \leq d_l$, $\forall l \in L$ for the values prior to the assignment

$$3 \cdot b_l^3 + 4 \cdot b_l^3 \leq d_l, \qquad \forall l \in L \tag{4.1.12}$$

holds true at any time. □

4.6 Lemma

If one of the following conditions holds after initialisation, it holds true on termination as well:

$$\tilde{b}^3 \geq q_3 \tag{4.1.13}$$
$$\tilde{b}^4 \geq q_4 \tag{4.1.14}$$

Proof. The algorithm distinguishes the following cases:

Case 1. (Both $\tilde{b}^3 \geq q_3$ and $\tilde{b}^4 \geq q_4$ hold true after initialisation.)
The algorithm does not do anything in this case as the current subdivision is already a solution.

Case 2. (Only $\tilde{b}^3 \geq q_3$ holds after initialisation.)
Lines 14, 17 and 20 change the b_l^3 and therefore $\tilde{b}^3$. The condition of the while statements in lines 13, 16 and 19 checks priorly if $\tilde{b}^3 - q_3$ is big enough to maintain the condition in eq. (4.1.13) after the application of the rule in the following line.

For instance $(b_l^3, b_l^4) \leftarrow (b_l^3 - 4, b_l^4 + 3)$ is only executed if $\tilde{b}^3 - q_3 \geq 4$. The other lines are analogous.

Thus, the condition eq. (4.1.13) always holds true for $\sum_{l \in L} b_l^3 = \tilde{b}^3$.

Case 3. (Only $\tilde{b}^4 \geq q_4$ holds after initialisation.)
For this case we have only one rule that is applied (see line 10). This is analogous to case 2.

□

4.7 Lemma

In any $l \in L$ the sum of the number of executions of rule 2 and rule 3 is at most one.

Proof.

Case 1. (The while condition of rule 1 in line 13 breaks because $\forall l \in L : b_l^3 \leq 3$.)
Any $l \in L$ has at most 3 blocks of 3. Rule 2 requires 3 blocks of 3 for any transformation, rule 3 requires 2 blocks of 3 for any transformation. Thus, at most one of the two rules may be applied and only one application is possible.

Case 2. (The while condition of rule 1 in line 13 breaks, but $\exists l \in L : b_l^3 > 3$.)
In this case $\tilde{b}^3 - q_3 < 4$ or $q_4 - \tilde{b}^4 < 3$ must hold true (condition of while loop). Rule 2 requires 3 blocks of 3 for any transformation and produces 2 blocks of 4;

rule 3 requires 2 blocks of 3 for any transformation and produces 1 block of 4. The conditions of the while loop in lines 16 and 19 entail that at most one of the rules 2 and 3 is applied altogether.

□

4.8 Theorem

The algorithm 4.1 is correct and can be implemented to run in $\mathcal{O}(m)$ time and to require $\mathcal{O}(m)$ space.

Proof. In order to prove the correctness of the algorithm we will show the following two statements:

1. If no solution exists, the algorithm will not find one.
2. If a solution exists, the algorithm will find a solution.

1. If no solution exists, the algorithm will not find one.
Let d, q denote a proper input to the algorithm 4.1 for which no solution exists, i.e., there is no fragmentation (b_l^3, b_l^4) obeying

$$3 \cdot b_l^3 + 4 \cdot b_l^4 \leq d_l, \qquad \forall l \in L \tag{4.1.15}$$
$$\tilde{b}^3 \geq q_3, \tag{4.1.16}$$
$$\tilde{b}^4 \geq q_4. \tag{4.1.17}$$

If the algorithm would find a solution, the solution would satisfy eqs. (4.1.16) and (4.1.17) due to line 22. Furthermore eq. (4.1.15) would hold due to lemma 4.5. This is contradictory to the assumption.

2. If a solution exists, the algorithm will find a solution.
Let d, q denote a proper input to the algorithm 4.1 and let (β_l^3, β_l^4) be a valid solution:

$$3 \cdot \beta_l^3 + 4 \cdot \beta_l^4 \leq d_l, \qquad \forall l \in L \tag{4.1.18}$$
$$\tilde{\beta}^3 \geq q_3, \tag{4.1.19}$$
$$\tilde{\beta}^4 \geq q_4. \tag{4.1.20}$$

As our initialisation is best-filling (lemma 4.4) the following must hold true after initialisation:

$$\underset{l \in L}{\forall} : 3 \cdot b_l^3 + 4 \cdot b_l^4 \geq 3 \cdot \beta_l^3 + 4 \cdot \beta_l^4, \tag{4.1.21}$$
$$3 \cdot \tilde{b}^3 + 4 \cdot \tilde{b}^4 \geq 3 \cdot \tilde{\beta}^3 + 4 \cdot \tilde{\beta}^4. \tag{4.1.22}$$

We can deduce from eqs. (4.1.19), (4.1.20) and (4.1.22):

$$\tilde{b}^3 \geq q_3 \qquad \vee \qquad \tilde{b}^4 \geq q_4. \tag{4.1.23}$$

If $\tilde{b}^3 \geq q_3 \wedge \tilde{b}^4 \geq q_4$ our best-filling subdivision is a solution.

So, we consider either $\tilde{b}^3 < q_3$ or $\tilde{b}^4 < q_4$ in the following two cases. For each case either $\tilde{b}^3 \geq q_3$ or $\tilde{b}^4 \geq q_4$ must hold true at any point due to lemma 4.6.

Case 1. ($\tilde{b}^3 < q_3$)

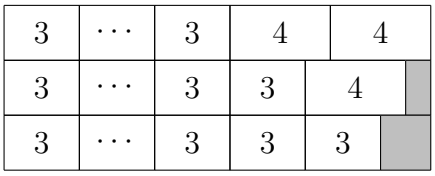
As our initialisation procedure favours blocks of 3 to blocks of 4, our initial subdivision satisfies

$$\underset{l \in L}{\forall} : b_l^4 \leq 2. \tag{4.1.24}$$

We notice that

$$\tilde{\beta}^3 + \tilde{\beta}^4 \leq \tilde{b}^3 + \tilde{b}^4 \tag{4.1.25}$$

must hold true because of eq. (4.1.21) and the fact that any subdivision with more blocks of 4 would use less blocks in total and any subdivision using less blocks of 4 cannot have more blocks in total. The following figure illustrates this fact:



We only have one transformation rule (line 10). This increases b_l^3 by 1 and decreases b_l^4 by 1.

Equations (4.1.19) and (4.1.20) entail

$$\tilde{\beta}^3 + \tilde{\beta}^4 \geq q_3 + q_4. \tag{4.1.26}$$

Since $b_l^3 + b_l^4$ remains constant and eqs. (4.1.25) and (4.1.26) hold we must eventually get $\tilde{b}^3 \geq q_3$. $\tilde{b}^4 \geq q_4$ must hold true as well due to lemma 4.6. Our subdivision is valid as lemma 4.5 shows.

Thus, we have obtained a solution which satisfies our conditions eqs. (4.1.3) and (4.1.3). This is a contradiction to the assumption that we do not find a solution.

Case 2. ($\tilde{b}^4 < q_4$)

In order to prove this case we transform the existing solution β in a solution the algorithm would find. Then we apply the algorithm in reverse and show that this yields our best-filling subdivision as produced by our initialisation procedure.

Preliminaries

For abbreviation let $\boxed{3}$ denote a block of size 3, and $\boxed{4}$ a block of size 4. $\boxed{3}\boxed{3}\boxed{3}\boxed{3}$ are four blocks of $\boxed{3}$. $\boxed{3}\boxed{3}\boxed{3}\boxed{3}$ implies $\boxed{3}\boxed{3}\boxed{3}$, so a subdivision with 5 blocks of $\boxed{3}$ contains $\boxed{3}\boxed{3}\boxed{3}\boxed{3}$, $\boxed{3}\boxed{3}\boxed{3}$, $\boxed{3}\boxed{3}$, and $\boxed{3}$.

Furthermore $\boxed{2}$ is an unused space of size *exactly* 2, $\boxed{1}$ is an unused space of size *exactly* 1. Thus, $\boxed{2}$ does *never* imply $\boxed{1}$. Unused space refers to not used space in matters of the maximal usable space (definition 4.3).

At any point in the following transformations we fill implicitly any usable gap with $\boxed{3}$ if possible. This entails that β_l always has at most a $\boxed{2}$ block of unused space in $\hat{d}_l$.

Restricting number of $\boxed{4}$

We now transform the existing solution β to a solution that the algorithm would generate. First of all we assume that

$$\tilde{\beta}^4 = q_4, \tag{4.1.27}$$

since we can always reduce the number $\boxed{4}$ by simply deleting them due to eq. (4.1.20).

We now exchange $\boxed{3}$ for $\boxed{4}$ and vice versa such that

$$\underset{l \in L}{\forall} : \beta_l^4 \geq (\hat{d}_l \bmod 3). \tag{4.1.28}$$

In doing so the number of $\boxed{4}$ remains constant, the number of $\boxed{3}$ may increase because of our preliminary.

This process of exchanging is possible because:

- We have enough $\boxed{4}$ in total as this is the "basis" of our best-filling subdivision. If there were not be enough $\boxed{4}$, β would have less $\boxed{4}$ as our best-filling subdivision; as $\tilde{\beta}^4 = q_4$ this is the previous case.
- We can always replace $\boxed{4}$ by $\boxed{3}$ as it is smaller.
- If $\hat{d}_l \equiv 2 \mod 3$ and we have only $\boxed{3}$, there is an unused rest $\boxed{2}$. We can swap in two $\boxed{4}$ for two $\boxed{3}$. Initially there have to be at least two $\boxed{3}$ as $\hat{d}_l = 2$ and $\hat{d}_l = 5$ is not possible (see definition 4.3).

- If $\hat{d}_l \equiv 2 \mod 3$ and we have only one $\boxed{4}$, there is an unused rest 1. We can swap in a $\boxed{4}$ for a $\boxed{3}$.
- If $\hat{d}_l \equiv 1 \mod 3$ and we have only $\boxed{3}$, there is an unused rest 1. We can swap in a $\boxed{4}$ for a $\boxed{3}$.

Now simple logic shows that the following two statements hold true for any β_l:

If we have an unused space of exactly 2, we have at least $\boxed{4}$. (4.1.29)

If we have an unused space of exactly 1, we have at least $\boxed{4}\boxed{4}$. (4.1.30)

Otherwise $\beta_l^4 \geq (\hat{d}_l \mod 3)$ would not hold true.

Minimise unused space by transformations that keep number of $\boxed{4}$ constant

Now we transform β by completing the following steps. We only process a transformation if neither of the previous ones is applicable. Note that transformations may become applicable again. Note that neither of the transformations change the validity of the statements (4.1.29), (4.1.30).

Let β_{l_1} and β_{l_2} be two subdivisions with $l_1, l_2 \in L$, and $l_1 \neq l_2$.

(T.1) For any β_{l_1} and β_{l_2} with 2:

- Does β_{l_3}, $l_3 \in L$ with $\boxed{3}\boxed{3}\boxed{3}$ exist? Transform $\boxed{3}\boxed{3}\boxed{3} \rightarrow \boxed{4}\boxed{4}1$, $\boxed{4}2 \rightarrow \boxed{3}\boxed{3}$, $\boxed{4}2 \rightarrow \boxed{3}\boxed{3}$.
- Do β_{l_1} or β_{l_2} contain a $\boxed{3}$? Transform $\boxed{3}\boxed{4}2 \rightarrow \boxed{4}\boxed{4}1$, $\boxed{4}2 \rightarrow \boxed{3}\boxed{3}$.

(T.2) For any β_{l_1} with 1, and β_{l_2} with 2:

- Does β_{l_3}, $l_3 \in L$ with $\boxed{3}\boxed{3}\boxed{3}\boxed{3}$ exist? Transform $\boxed{3}\boxed{3}\boxed{3}\boxed{3} \rightarrow \boxed{4}\boxed{4}\boxed{4}$, $\boxed{4}\boxed{4}1 \rightarrow \boxed{3}\boxed{3}\boxed{3}$, $\boxed{4}2 \rightarrow \boxed{3}\boxed{3}$.
- Does β_{l_1} contain a $\boxed{3}$? Transform $\boxed{3}\boxed{4}\boxed{4}1 \rightarrow \boxed{4}\boxed{4}\boxed{4}$, $\boxed{4}2 \rightarrow \boxed{3}\boxed{3}$.
- Does β_{l_2} contain $\boxed{3}\boxed{3}$? Transform $\boxed{3}\boxed{3}\boxed{4}2 \rightarrow \boxed{4}\boxed{4}\boxed{4}$, $\boxed{4}\boxed{4}1 \rightarrow \boxed{3}\boxed{3}\boxed{3}$.

(T.3) Do β_{l_1} and β_{l_2} with 1 exist?

- Does β_{l_3}, $l_3 \in L$ with $\boxed{3}\boxed{3}\boxed{3}\boxed{3}$ exist? Transform $\boxed{3}\boxed{3}\boxed{3}\boxed{3} \rightarrow \boxed{4}\boxed{4}\boxed{4}$, $\boxed{4}\boxed{4}1 \rightarrow \boxed{3}\boxed{3}\boxed{3}$, $\boxed{4}\boxed{4}1 \rightarrow \boxed{3}\boxed{4}2$.
- Do β_{l_1} or β_{l_2} contain a $\boxed{3}$? Transform $\boxed{3}\boxed{4}\boxed{4}1 \rightarrow \boxed{4}\boxed{4}\boxed{4}$, $\boxed{4}\boxed{4}1 \rightarrow \boxed{3}\boxed{4}2$.

In those transformations the number of $\boxed{4}$ remains constant, and the number of $\boxed{3}$ increases or remains constant. The exchanged blocks are equal in size, thus, the transformations are feasible. No infinite loop is possible if the number of $\boxed{3}$ increases. The only transformation where it remains constant is $\boxed{3}\boxed{4}\boxed{4}\,1 \rightarrow \boxed{4}\boxed{4}\boxed{4}$, $\boxed{4}\boxed{4}\,1 \rightarrow \boxed{3}\boxed{4}\,2$. Here the 1 are eliminated, so no loop is possible as well.

If our transformation eliminates all unused spaces, the new constructed blocks are best-filling, therefore, eq. (4.1.28) must hold true. A created $\boxed{4}\boxed{4}\,1$ holds a minimum of two $\boxed{4}$ and therefore satisfies eq. (4.1.28) as well. A created $\boxed{4}\,2$ contains one $\boxed{4}$ more than the best-filling $\boxed{3}\boxed{3}$. Thus eq. (4.1.28) still holds true after all transformations.

Swap unused space to the β_l afore

Now we swap the blocks of $\boxed{3}$ to the β_l more rear. We do only perform a transformation if no previous is applicable. Note that transformations may become applicable again. Let $l_1, l_2 \in L$ and $l_1 < l_2$:

(S.1) Do β_{l_1} containing $\boxed{3}\boxed{3}$, and β_{l_2} having 2 exist? Swap $\boxed{3}\boxed{3}$ and $\boxed{4}\,2$.

(S.2) Do β_{l_1} containing $\boxed{3}\boxed{3}\boxed{3}$, and β_{l_2} having 1 exist? Swap $\boxed{3}\boxed{3}\boxed{3}$ and $\boxed{4}\boxed{4}\,1$.

(S.3) Do β_{l_1} containing $\boxed{3}\boxed{4}\,2$, and β_{l_2} having 1 exist? Swap $\boxed{3}\boxed{4}\,2$ and $\boxed{4}\boxed{4}\,1$.

(S.4) Do β_{l_1} containing $\boxed{3}\boxed{3}\boxed{3}\boxed{3}$, and β_{l_2} containing $\boxed{4}\boxed{4}\boxed{4}$ exist? Swap $\boxed{3}\boxed{3}\boxed{3}\boxed{3}$ and $\boxed{4}\boxed{4}\boxed{4}$.

(S.5) Do β_{l_1} containing $\boxed{3}\boxed{4}\boxed{4}\,1$, and β_{l_2} containing $\boxed{4}\boxed{4}\boxed{4}$ exist? Swap $\boxed{3}\boxed{4}\boxed{4}\,1$ and $\boxed{4}\boxed{4}\boxed{4}$.

The solution β that we have obtained now has the following properties:

(P.1) Each β_l has at least as many $\boxed{4}$ as our best-filling subdivision (eq. (4.1.28) still holds true).

(P.2) If a β_{l_2} with $\boxed{3}\boxed{3}\boxed{3}\boxed{3}$ exists, we have at most one β_{l_1} with unused space (see (T.2), (T.3)). For this eventuality $l_1 \leq l_2$ must hold true (see (S.1), (S.2)).

(P.3) If a β_{l_2} with $\boxed{3}\boxed{3}\boxed{3}$ exists, we have at most one β_{l_1} with an unused space of 2 (see (T.1)). For this eventuality $l_1 \leq l_2$ must hold true (see (S.1)).

(P.4) $\tilde{\beta}^4 = q_4$ as this was our precondition and we never changed the number of $\boxed{4}$.

Run algorithm in reverse
We now set $b_l = \beta_l$, $\forall l \in L$ and run the algorithm in reverse and show that our b is transformed into the best-filling subdivision that our algorithm yields in its initialisation procedure.

As b is a solution our algorithm yields *success* (line 22).

As we do not have a sufficient amount of $\boxed{4}$ in this case we consider the transformations beginning in line 11. Thus, only applications of rule 1, rule 2, and rule 3 are possible. Table 4.2 shows the unused rest created on each rule application. Only rule 2 and rule 3 create an unused space.

Lemma 4.7 states that on each b_l only rule 2 or rule 3 may be applied, never both. Further, the two rules are applied at most one time. This entails: on each subdivision with 2 rule 3 has to be applied backwards; on each subdivision with 1 rule 2 has to be applied backwards.

Reversing rule 3
Only rule 3 produces an unused space of 2. May $l_{\text{end}} \in L$ be the biggest element of $l \in L$ where b_l contains an unused rest 2. If no such b_l exists, we skip this step and directly proceed with reversing rule 2.

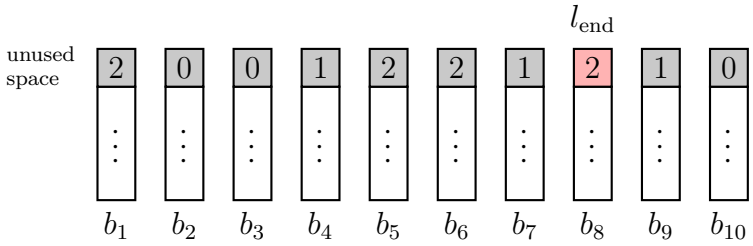


Figure 4.1: Begin of reverse algorithm: $l_{\text{end}} = 8$

The last step of the algorithm was the application of rule 3 in lines 18 to 20. This means the condition of the while loop in line 19 does no longer hold true after processing l_{end}. This is obviously the case, since b is a solution, but our while condition requires $q_4 - \tilde{b}^4 \geq 1$.

We walk through the $l \in L$ in reverse, beginning at l_{end}. At any b_l where a 2 exists we apply the rule 3 in reverse.

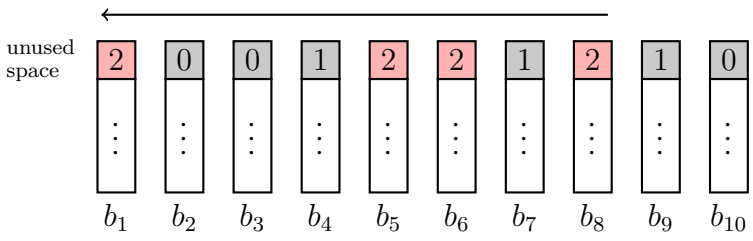


Figure 4.2: Reverse application of rule 3

Why would the algorithm choose exactly the b_l where we reversed rule 3? Rule 3 can only be applied if a subdivision has at least two $\boxed{3}$. If (S.1) is applied all b_l, $l < l_{\text{end}}$ contain at most a single $\boxed{3}$ so no application of any rule is possible. The algorithm can only apply rule 3 on the $l \in L$ that contained 2.

Reversing rule 2

In the next step we reverse the application of rule 2. This is analogue to the reversal of rule 3.

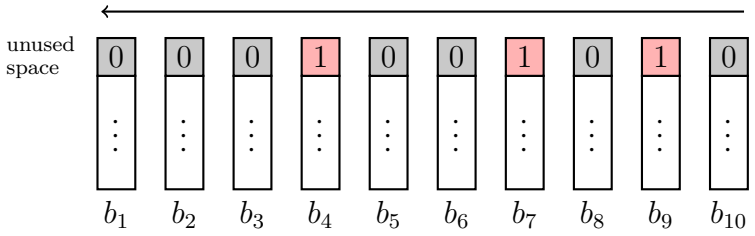


Figure 4.3: Reverse application of rule 2

Why would the algorithm choose exactly the b_l where we reversed rule 2? (S.2) makes sure, that no $\boxed{3}\boxed{3}\boxed{3}$ exists before the last application of rule 2. Furthermore, (T.1) renders a coexistence of two $\boxed{3}\boxed{4}\,2$ impossible, which would create a $\boxed{3}\boxed{3}\boxed{3}$. (S.5) makes sure that an existing $\boxed{3}\boxed{4}\,2$ (which is now a $\boxed{3}\boxed{3}\boxed{3}$) is behind the last rule 2 application of our algorithm. Thus, the algorithm does apply the rule 2 exactly on those subdivisions where we just reversed rule 2.

Now all subdivisions are best-filling, as no unused rest is existing anymore.

Reversing rule 1

Now we reverse rule 1 in any $l \in L$ as often as possible.

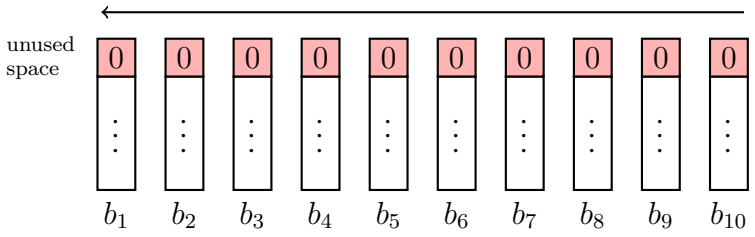


Figure 4.4: Reverse application of rule 1

Thus,

$$\underset{l \in L}{\forall} : b_l^4 < 3. \tag{4.1.31}$$

A best-filling subdivision for blocks of size 3 and 4 requires

$$\underset{l \in L}{\forall} : \beta_l^4 \equiv \hat{d}_l \mod 3. \tag{4.1.32}$$

This entails

$$\underset{l \in L}{\forall} : \beta_l^4 = (\hat{d}_l \bmod 3), \tag{4.1.33}$$

and this is the definition of our best-filling subdivision.

The algorithm is correct.

Running time and space requirement
The initial breakdown of the durations into blocks is done in $\mathcal{O}(m)$ time, where $m = |L|$ is the number of rest accommodations.

The application of each rule may be performed in constant time on each rest accommodation, as the number of possible applications, i.e., the number of while loop circles, can be predetermined easily. It is not done here for convenience, the next algorithm 4.2 shows this principle. Thus each application of a rule on the complete set of rest accommodation may be done in $\mathcal{O}(m)$ time. As the number of rules is constant, the running requirement for the rule application is $O(m)$.

Thus, the complete algorithms runs in $O(m)$ time.

We use for the calculation only a matrix of size $2 \times m$ so the space requirement is $\mathcal{O}(m)$. □

4.2 In-Flight Rest for Two Rest Length and Free Choice of Accommodation

4.2.1 Problem

We will modify problem 4.1 to allow arbitrary values for the length of rests. Note that this problem is the *Multiprocessor Scheduling Problem with C job lengths* with $C = 2$ (see problem 3.4) named by McCormick et al. [MSS01]. They refer to this exact problem as MSP2m and developed an algorithm (see section 3.2.2).

4.2 Problem

We are given two sets $L = \{1, \ldots, m\}$ of rest accommodation and $R = \{r_1, r_2\}$, $r_1, r_2 \in \mathbb{N}$, $r_1 < r_2$ of possible rest durations, and

$q_r \in \mathbb{N}$ as the quantity of crew members needing a rest of $r \in R$,

$d_l \in \mathbb{N}$, $d_l \geq r_1$ as capacity of rest accommodation $l \in L$.

We are looking for a subdivision of all $d_l, l \in L$ into r-sized blocks ($r \in R$) denoted as $b_l^r \in \mathbb{N}$, obeying:

$$\underset{l \in L}{\forall} : \sum_{r \in R} r \cdot b_l^r \leq d_l \qquad \text{(valid subdivision)} \tag{4.2.1}$$

$$\underset{r \in R}{\forall} : \sum_{l \in L} b_l^r \geq q_r \qquad \text{(accommodation for every crew member)} \tag{4.2.2}$$

4.2.2 Algorithm

The idea behind the algorithm is the same as of the algorithm in the previous chapter: if we first split all the accommodations' capacities into time slots that fill them best, we reduce all rest accommodations to the maximal usable space. Then we transform this subdivision by applying certain *rules* to transform it into a valid solution.

The outline of the algorithm is as follows:

1. Initialise $(b_l^{r_1}, b_l^{r_2})$, $l \in L$ with a best-filling subdivision of d_l, $l \in L$.
2. Determine transforming direction according to total quantities $(\tilde{b}^{r_1}, \tilde{b}^{r_2}) \overset{\text{def}}{=} (\sum b_l^{r_1}, \sum b_l^{r_2})$: transform r_1-sized blocks into r_2-sized blocks, or vice versa.
3. Produce rules for the transformation direction.
4. Apply rules successively on all legs if possible, ordered by increasing penalty, until a solution is obtained, or abort.

Main Algorithm

First of all we give a formal description of the main algorithm.

4.2 Algorithm – IFR with Two Rest Lengths: Problem 4.2

Input: $r_1, r_2 \in \mathbb{N}, 0 < r_1 < r_2$ as possible lengths of the rest periods
$g = \gcd(r_1, r_2)$ as greatest common divisor of r_1 and r_2
$q = (q_{r_1}, q_{r_2})$ as req. number of time slots of len. r_1 resp. r_2
$d = (d_1, d_2, \ldots, d_m)$ as vector of capacities $d_l \in \mathbb{N}, d_l \geq r_1$ of rest accommodations $l \in L$

Output: $(b_l^3, b_l^4)_{l \in L}$ as subdivision of $(d_l)_{l \in L}$ satisfying eqs. (4.2.1) and (4.2.2)

▷ Normalise

if $g > 1$ **then**
  $r_1 \leftarrow \frac{r_1}{g}, \quad r_2 \leftarrow \frac{r_2}{g}, \quad$ **forall** $l \in L$ **do** $d_l \leftarrow \left\lfloor \frac{d_l}{g} \right\rfloor$

▷ Initialise

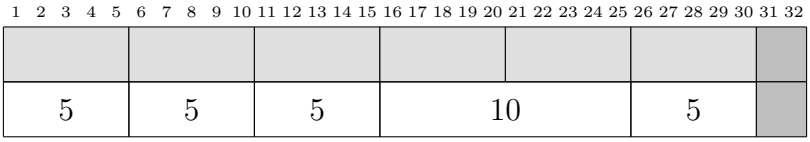
$(b_l^{r_1}, b_l^{r_2}) \leftarrow \text{init}(r_1, r_2, d)$
if $q_{r_1} - \tilde{b}^{r_1} < 0 \wedge q_{r_2} - \tilde{b}^{r_2} > 0$ **then**
  $\mathcal{R} \leftarrow$ `generate_rules`$(r_1, r_2, \boxed{r_1 \rightarrow r_2})$
  for $(o_1, o_2, p) \in \mathcal{R}$, p increasing **do** ▷ it. $\mathcal{R}$ with incr. penalty
    for $l \in L$ **do**
      $c_{\text{total}} \leftarrow \min\left(\left\lfloor \frac{\tilde{b}^{r_1} - q_{r_1}}{-o_1} \right\rfloor, \left\lceil \frac{q_{r_2} - \tilde{b}^{r_2}}{o_2} \right\rceil \right)$ ▷ min(possible,need)
      if $c_{\text{total}} = 0$ **then break** and next rule
      $c_{\text{local}} \leftarrow \left\lfloor \frac{b_l^{r_1}}{-o_1} \right\rfloor$ ▷ applications possible in $(b_l^{r_1}, b_l^{r_2})$
      $c \leftarrow \min(c_{\text{total}}, c_{\text{local}})$
      $(b_l^{r_1}, b_l^{r_2}) \leftarrow (b_l^{r_1} + c \cdot o_1, b_l^{r_2} + c \cdot o_2)$ ▷ Apply rule c times.
else if $q_{r_2} - \tilde{b}^{r_2} < 0 \wedge q_{r_1} - \tilde{b}^{r_1} > 0$ **then**
  $\mathcal{R} \leftarrow$ `generate_rules`$(r_1, r_2, \boxed{r_1 \leftarrow r_2})$
  for $(o_1, o_2, p) \in \mathcal{R}$, p increasing **do**
    for $l \in L$ **do**
      $c_{\text{total}} \leftarrow \min\left(\left\lfloor \frac{\tilde{b}^{r_2} - q_{r_2}}{-o_2} \right\rfloor, \left\lceil \frac{q_{r_1} - \tilde{b}^{r_1}}{o_1} \right\rceil \right)$
      if $c_{\text{total}} = 0$ **then break** and next rule
      $c_{\text{local}} \leftarrow \left\lfloor \frac{b_l^{r_2}}{-o_2} \right\rfloor$
      $c \leftarrow \min(c_{\text{total}}, c_{\text{local}})$
      $(b_l^{r_1}, b_l^{r_2}) \leftarrow (b_l^{r_1} + c \cdot o_1, b_l^{r_2} + c \cdot o_2)$
if $q_{r_1} - \tilde{b}^{r_1} \leq 0 \wedge q_{r_2} - \tilde{b}^{r_2} \leq 0$ **then**
  ▷ Success!
else
  ▷ No solution.

Now we will discuss the steps of the algorithm.

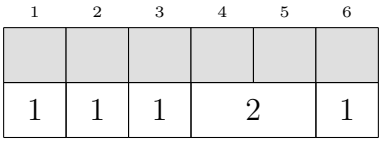
Normalising the Problem

First of all we divide the input parameters representing a time duration by $\gcd(r_1, r_2)$. This is possible because of the same reason the algorithm 4.1 processed the in-flight rest problem for 90 minutes and 120 minutes with the time durations 3 and 4.

For instance if we take $r_1 = 5$, $r_2 = 10$ a $d_l = 32$ could be subdivided as follows.



It is easy to see that at most 30 time units of the space is usable so we can safely reduce the figures $r_1 = \frac{5}{\gcd(5,10)} = 1$, $r_2 = \frac{10}{\gcd(5,10)} = 2$ and d_l for all $l \in L$ accordingly.



Applying Rules

After a best-filling initialisation of the rest accommodations we need to transform our current state, if possible, into a valid solution. In order to do so we generate the rules for the proper transforming direction. We either have too many blocks of r_1 or too many blocks of r_2, otherwise there would not be a solution to our problem instance.

Like in the previous algorithm 4.1 we apply the rules ordered by their penalty. In order to decrease the running time of the algorithm we check how many times we can apply a rule. c_{total} denotes the maximum number of rule applications possible to achieve our goal or the maximum possible rule applications without decreasing the number of source blocks of our transformation too much, whichever is smaller. c_{local} is the number of possible rule applications in our current rest accommodation l.

The core of our algorithm is how to find the best-filling subdivision and how to find proper rules.

4.3 Algorithm – Best-filling Subdivision

Input: $r_1, r_2 \in \mathbb{N}, 0 < r_1 < r_2, \gcd(r_1, r_2) = 1$ as possible lengths of the rest periods
$d = (d_1, d_2, \ldots, d_m)$ as vector of capacities $d_l \in \mathbb{N}, d_l \geq r_1$ of rest accommodations $l \in L$

Output: $(b_l^{r_1}, b_l^{r_2})_{l \in L}$ as best-filling subdivision of (d_l) (definition 4.2)

```
1  function init(r_1, r_2, d) is
       ▷ Build dictionary for optimal values smaller than r_1 · r_2
2      let split be an array of size r_1 · r_2
3      for j ∈ {0, ..., r_1 − 1} do
4          for i ∈ {0, ..., r_2 − 1} do
5              if i · r_1 + j · r_2 ≥ r_1 · r_2 then break
6              else
7                  split[i · r_1 + j · r_2] ← (i, j)
       ▷ Any unset value is set to the last proper value (no better
         subdivision possible)
8      last ← (0, 0)
9      forall i ∈ 0, ..., r_1 · r_2 − 1 do
10         if isset(split[i]) then last ← split[i]
11         else split[i] ← last
       ▷ Build dictionary to resolve modulo operation for values ≥ r_1 · r_2
12     let mods be an array of size r_1
13     forall i ∈ {0, ..., r_1 − 1} do
14         mods[i · r_2 mod r_1] ← i
       ▷ subdivide all rest accommodations using the dictionaries
15     forall l ∈ L do
16         if d_l < r_1 · r_2 then
17             (b_l^{r_1}, b_l^{r_2}) ← split[d_l]
18         else
19             b_l^{r_2} ← mods[d_l mod r_1]
20             b_l^{r_1} ← (d_l − b_l^{r_2} · r_2) / r_1
21     return (b_l^{r_1}, b_l^{r_2})
```

4.2.3 Best-Filling Initialisation

Our first step of the algorithm is to find a best-filling subdivision as initialisation for the availabilities of all rest accommodations. An procedure that solves this is presented in algorithm 4.3.

We will now prove that the presented subdivision procedure is indeed best-filling.

4.9 Lemma
The subdivision in algorithm 4.3 is best-filling.

Proof.

Case 1. $(d_l \geq r_1 \cdot r_2)$
Line 19 assigns to $b_l^{r_2}$ the smallest non-negative value that satisfies

$$b_l^{r_2} \cdot r_2 \equiv d_l \mod r_1, \tag{4.2.3}$$

thus, $0 \leq b_l^{r_2} < r_1$, $b_l^{r_2} \in \mathbb{N}$ must hold true as well which entails

$$b_l^{r_2} \cdot r_2 < r_1 \cdot r_2 \leq d_l. \tag{4.2.4}$$

The value $b_l^{r_2}$ exists because $\gcd(r_1, r_2) = 1$ due to the input requirement of the procedure.

We may conclude from eqs. (4.2.3) and (4.2.4), that

$$b_l^{r_1} = \frac{d_l - b_l^{r_2} \cdot r_2}{r_1} \in \mathbb{N}, \tag{4.2.5}$$

which implicates

$$b_l^{r_1} \cdot r_1 + b_l^{r_2} \cdot r_2 = d_l, \qquad b_l^{r_1}, b_l^{r_2} \in \mathbb{N}. \tag{4.2.6}$$

We used all the space, thus, this subdivision is best-filling.

Case 2. $(d_l < r_1 \cdot r_2)$
By its definition the array `split` saves for all $i \in \mathbb{N}$, $0 \leq i < r_1 \cdot r_2$ an optimal subdivion $(b_l^{r_1}, b_l^{r_2})$ if

$$b_l^{r_1} \cdot r_1 + b_l^{r_2} \cdot r_2 = i, \qquad b_l^{r_1}, b_l^{r_2} \in \mathbb{N} \tag{4.2.7}$$

has a solution. If not it saves the subdivision of maximal value i' which has a solution to

$$b_l^{r_1} \cdot r_1 + b_l^{r_2} \cdot r_2 = i' \leq i, \qquad b_l^{r_1}, b_l^{r_2} \in \mathbb{N}. \tag{4.2.8}$$

This is by the definition of best-filling a best-filling subdivision.

□

4.2.4 | Generation of Rules

In the following listing we will present the way the rules for transforming the blocks are generated. We have two different directions for transforming: from smaller to bigger blocks ($\boxed{r_1 \to r_2}$) and from bigger to smaller blocks ($\boxed{r_1 \leftarrow r_2}$).

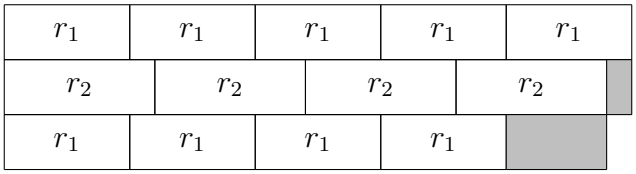
4.4 Algorithm – Generation of Rules

Input: $r_1, r_2 \in \mathbb{N}, 0 < r_1 < r_2$ as possible lengths of the rest periods
`rule_dir` $\in \{\boxed{r_1 \to r_2}, \boxed{r_1 \leftarrow r_2}\}$
Output: $\mathcal{R}$ as ordered set of rules

1 **function** `generate_rules`$(r_1, r_2,$ `rule_dir`$)$ **is**
▷ `Calculate all possible rules`
2 let `rules` be a list
3 **if** `rule_dir` $= \boxed{r_1 \to r_2}$ **then**
4 **for** $i \in \{1, \ldots, r_1\}$ **do**
5 $n \leftarrow \left\lceil \frac{r_2 \cdot i}{r_1} \right\rceil$
6 $p \leftarrow n \cdot r_1 - i \cdot r_2$
7 `rules`.$append((-n, i, p))$
8 **else**
9 **for** $i \in \{1, \ldots, r_1\}$ **do**
10 $t \leftarrow \left\lfloor \frac{r_2 \cdot i}{r_1} \right\rfloor$
11 $p \leftarrow i \cdot r_2 - t \cdot r_1$
12 `rules`.$append((t, -i, p))$
▷ `Discard superfluous rules`
13 $\mathcal{R} \leftarrow \{\}$
14 $p_{\text{last}} \leftarrow \infty$
15 **forall** (o_1, o_2, p) **in** `rules` **do**
16 **if** $p < p_{\text{last}}$ **then**
17 $\mathcal{R} \leftarrow \mathcal{R} \cup \{(o_1, o_2, p)\}$
18 $p_{\text{last}} \leftarrow p$
19 **return** $\mathcal{R}$

Generation of rules

As $r_2 > r_1$ we use the r_2-sized blocks as basis for our transformation rules. We calculate how many blocks of r_1 we need to produce this amout of r_2-sized blocks respectively the amount of r_1-sized blocks we are able to produce.



This approach results in the following rules:

$r_1 \to r_2$		$r_2 \to r_1$
needed r_1	**blocks** r_2	**produced** r_1
$\left\lceil \frac{r_2 \cdot 1}{r_1} \right\rceil$	1	$\left\lfloor \frac{r_2 \cdot 1}{r_1} \right\rfloor$
$\left\lceil \frac{r_2 \cdot 2}{r_1} \right\rceil$	2	$\left\lfloor \frac{r_2 \cdot 2}{r_1} \right\rfloor$
$\left\lceil \frac{r_2 \cdot 3}{r_1} \right\rceil$	3	$\left\lfloor \frac{r_2 \cdot 3}{r_1} \right\rfloor$
$\vdots$	$\vdots$	$\vdots$
r_2	r_1	r_2

Discarding of superfluous rules

We may discard rules of the rule set as they can be derived from existing rules. The algorithm discards rules in line 16 by no copying them into the set of rules $\mathcal{R}$. The deletion procedure arose from an observation of the complete rule set.

The following proof shows that this is approach is feasible.

4.10 Lemma

Algorithm 4.4 does not discard any necessary rule.

Proof. [1]

We have a case for each direction of transformation.

[1] Thanks to Jan Putzig for the idea behind this proof.

Case 1. (Transformation direction $\boxed{r_1 \to r_2}$)

Let $(-n_i, i, p_i) \in \texttt{rules}$ the first rule which is deleted wrongfully from the set of rules and let $(-n_l, l, p_l) \in \texttt{rules}$ be the last rule that was added to $\mathcal{R}$.

Thus, all rules $(-n_j, j, p_j)$, where $0 < j < i$, are representable by different applications of rules in $\mathcal{R}$.

All penalties satisfy

$$r_1 > p_j \geq 0. \tag{4.2.9}$$

As $(-n_i, i, p_i)$ is deleted the following must hold true:

$$p_i \geq p_l. \tag{4.2.10}$$

For p_1 and p_i holds true:

$$p_1 \equiv r_2 \qquad \text{mod } r_1, \tag{4.2.11}$$
$$p_i \equiv i \cdot r_2 \equiv i \cdot p_1 \qquad \text{mod } r_1. \tag{4.2.12}$$

We may express p_i as

$$p_i = \Big(p_l + \underbrace{\big(p_1 \cdot (i-l)\big) \bmod r_1}_{<r_1 - p}\Big) \bmod r_1. \tag{4.2.13}$$

$\big(p_1 \cdot (i-l)\big) \bmod r_1 < r_1 - p$ must hold true, as

$$p_i = (p_l + x) \bmod r_1 \tag{4.2.14}$$
$$\text{(with eq. (4.2.10))} \Rightarrow \quad x \geq r_1 - p \Rightarrow x \geq r_1. \tag{4.2.15}$$

Thus, we may conclude

$$p_i - p_l = \big(p_1 \cdot (i-l)\big) \bmod r_1 \tag{4.2.16}$$
$$\Leftrightarrow \qquad p_i - p_l = p_{i-l} \tag{4.2.17}$$

Now we add the rules no. l and no. $i - l$

$$n_l \cdot r_1 + n_{i-l} \cdot r_1 = l \cdot r_2 + p_l + (i-l) \cdot r_2 + p_{i-l} \tag{4.2.18}$$
$$\text{(with eq. (4.2.17))} \Leftrightarrow \quad (n_l + n_{i-l}) \cdot r_1 = i \cdot r_2 + p_i = n_i \cdot r_1 \tag{4.2.19}$$

Equations (4.2.17) and (4.2.19) show that we may express rule no. i as follows:

$$\begin{aligned}(-n_i, i, p_i) &= (-n_l - n_{i-l}, i - l + l, p_l + p_{i-l}) \\ &= (-n_l, l, p_l) + (-n_{i-l}, i-l, p_{i-l}).\end{aligned} \tag{4.2.20}$$

Thus the rule $(-n_i, i, p_i)$ is superfluous as it can be expressed with the rules no. l and no. $i-l$, which are representable by the set of rules $\mathcal{R}$. This is contrary to our assumption.

Case 2. (Transformation direction $\boxed{r_1 \leftarrow r_2}$)
Let $(t_i, -i, p_i) \in \texttt{rules}$ the first rule which is deleted wrongfully from the set of rules and let $(t_l, -l, p_l) \in \texttt{rules}$ be the last rule that was added to $\mathcal{R}$.

Thus, all rules $(t_j, -j, p_j)$, where $0 < j < i$, are representable by different applications of rules in $\mathcal{R}$.

All penalties satisfy

$$r_1 > p_j \geq 0. \tag{4.2.21}$$

As $(t_i, -i, p_i)$ is deleted the following must hold true:

$$p_i \geq p_l. \tag{4.2.22}$$

For p_1 and p_i holds true:

$$p_1 \equiv -r_2 \qquad \text{mod } r_1, \tag{4.2.23}$$
$$p_i \equiv -i \cdot r_2 \equiv i \cdot p_1 \qquad \text{mod } r_1. \tag{4.2.24}$$

We may express p_i as

$$p_i = \Big(p_l + \underbrace{\big(p_1 \cdot (i-l)\big) \bmod r_1}_{<r_1 - p}\Big) \bmod r_1. \tag{4.2.25}$$

Thus, we may conclude

$$p_i - p_l = \big(p_1 \cdot (i-l)\big) \bmod r_1 \tag{4.2.26}$$
$$\Leftrightarrow \qquad p_i - p_l = p_{i-l} \tag{4.2.27}$$

Now we add the rules no. l and no. $i-l$:

$$t_l \cdot r_1 + p_l + t_{i-l} \cdot r_1 + p_{i-l} = l \cdot r_2 + (i-l) \cdot r_2 \tag{4.2.28}$$
$$\text{(with eq. (4.2.27))} \Leftrightarrow \qquad (t_l + t_{i-l}) \cdot r_2 + p_i = i \cdot r_2 = t_i \cdot r_2 + p_i \tag{4.2.29}$$

Equations (4.2.27) and (4.2.29) show that we may express rule no. i as follows:

$$\begin{aligned}(-i, t_i, p_i) &= (-i + l - l, t_l + t_{i-l}, p_l + p_{i-l}) \\ &= (-l, t_l, p_l) + (-(i-l), t_{i-l}, p_{i-l}).\end{aligned} \tag{4.2.30}$$

Thus the rule $(-i, t_i, p_i)$ is superfluous as it can be expressed with the rules no. l and no. $i - l$, which are representable by the set of rules $\mathcal{R}$. This is contrary to our assumption.

□

The number of rules that are rendered by this algorithm are listed in section A.2 for values $r_1 < r_2 \leq 50$, for both directions of transformation.

4.2.5 Correctness and Running Time

4.11 Theorem

Let $\nabla_1 = \dfrac{r_1}{\gcd(r_1, r_2)}$ and $\nabla_2 = \dfrac{r_2}{\gcd(r_1, r_2)}$.

Then, the algorithm 4.2 can be implemented to have a worst case running time of $\mathcal{O}\big(m\nabla_1 + \nabla_1\nabla_2\big)$. It can be implemented to require $\mathcal{O}\left(m + \nabla_1\nabla_2\right)$ space.

Proof.

Running Time

The first step of the algorithm is to divide all integer values that represent "space" by the greatest common divisor of r_1 and r_2. Thus, the running time is a function of ∇_1 and ∇_2 rather than r_1 and r_2. The greatest common divisor is an input parameter so we do not consider its calculation.

The subdivision procedure of the algorithm requires $\mathcal{O}(\nabla_1\nabla_2 + \nabla_1)$ time for building the look-up tables and $\mathcal{O}(m)$ time for processing all rest accommodations. The rules are generated in $\mathcal{O}(\nabla_1)$ time. Thus, the initialisation takes $\mathcal{O}\big(m + \nabla_1\nabla_2\big)$ time in total.

The cardinality of the set of bound $\mathcal{R}$ is bounded by r_1, as this is the number of the initially produced rules. As the penalties are integral and the penalties of all not deleted rules are smaller than the penalty of the first rule, we may state

$$\begin{aligned} |\mathcal{R}| &\leq \max\left(p_1^{r_1 \to r_2}, p_1^{r_1 \leftarrow r_2}\right) + 1 \\ &= \max\left(\nabla_2 \bmod \nabla_1, (-\nabla_2) \bmod \nabla_1\right) + 1. \end{aligned} \tag{4.2.31}$$

In the worst case this is no better bound than r_1.

The main procedure of the algorithm processes all rules in a loop, applying them successively on each of the m rest accommodations. The application of a rule on a

single rest accommodation takes constant time. In the worst case all rules remain applicable in total to the very end, and no early breaking is possible.

Thus, the running time for the main procedure of the algorithm is $\mathcal{O}(m\nabla_1)$.

In total this results in a running time of $\mathcal{O}\big(m\nabla_1 + \nabla_1\nabla_2\big)$

Space Requirement
The algorithm requires $\mathcal{O}(m+1)$ space for the input parameters. The subdivision procedure initialises look-up tables of size $\mathcal{O}(\nabla_1\nabla_2 + \nabla_1)$ and saves the subdivision in a vector of size $\mathcal{O}(m)$. The set of rules has a size of $\mathcal{O}(\nabla_1)$.

The additional space requirement of the main procedure is constant.

Thus, the total space requirement is $\mathcal{O}\left(m + \nabla_1\nabla_2\right)$. □

Although $|\mathcal{R}| = \nabla_1$ is the worst case, we will see in section 5.1 that average number of rules is far less.

4.12 Conjecture

The algorithm 4.2 is correct.

In this thesis we do not provide a proof for the correctness of algorithm 4.2. However, we proved that the algorithm uses a best-filling initialisation (lemma 4.9) and that the algorithm uses a complete set of rules (lemma 4.10).

The algorithm is a generalisation ofalgorithm 4.1 and follows the same basic principles. For algorithm 4.1 a proof of correctness is provided in section 4.1.3.

4.3 Algorithm for Simple IFR

We now briefly discuss an approach to solve the complete simple IFR problem 2.2. As stated in section 2.4 this problem is NP-complete.

The problem is very similar to the generalised assignment problem (see problem 3.2). We can transform a SIFR in the following manner:

Transforming SIFR to GAP

First, we equate the set of crew members C with the set of tasks T, and the set of rest accommodations L with the set of agents A. The tasks are now *rest jobs* which are to be performed by a *rest accommodation/agent*.

$$T \leftarrow C \tag{4.3.1}$$

$$A \leftarrow L \tag{4.3.2}$$

As the rest accommodations are now agents, the capacity constrains of rest accommodations apply on the resource constraints of the agents:

$$\forall_{a=l}^{l \in L} : b_a \leftarrow d_l \tag{4.3.3}$$

The resource requirements for a *rest job* is the same as in our SIFR model. If no rest is possible the resource requirement is ∞.

$$\forall_{a=l}^{l \in L} \forall_{t=c}^{c \in C} : r_{a,t} \leftarrow \begin{cases} r_c & l \in P_c \\ \infty & \text{otherwise} \end{cases} \tag{4.3.4}$$

The costs for each possible assignment is 1, all other assignments get the costs $n+1$. Any number greater than one would be appropriate as well but $n+1$ enables a branch-and-bound algorithm to discard all solutions including this cost factor immediately.

$$\forall_{a=l}^{l \in L} \forall_{t=c}^{c \in C} : c_{a,t} \leftarrow \begin{cases} 1 & l \in P_c \\ n+1 & \text{otherwise} \end{cases} \tag{4.3.5}$$

We are looking now for a solution with costs of exactly n. If the minimal costs for this problem are bigger than n we know, that no such solution exists.

The GAP has a cost function and separate resource constraint. Thus, it is also easily possible to modulate the extended SIFR:

Variation for eSIFR

For the extended problem we may vary the resource constraint as follows.

$$\forall_{a=l}^{l\in L} \forall_{t=c}^{c\in C} : r_{a,t} \leftarrow \begin{cases} e_{t_l}(r_c) & l \in P_c \\ \infty & \text{otherwise} \end{cases} \tag{4.3.6}$$

Now lower class rest accommodations consume more resources than higher class seats.

The latest exact algorithm for the GAP was given by Posta et al. [PFM12] (see section 3.1.2). Posta supplied an implementation of his algorithm in C in a git repository [Pos12].

Posta et al. solved the optimisation problem with a series of decision problems. Thus, the source code can be altered to just solve one decision problem.

For the inquisitive reader, section B.3 provides Python code for transforming a SIFR problem into an input for the implementation of Posta. The for-loop in *main.c* of Posta's code, containing the variable `z_ub` for the bound $\bar{z}$, must be altered to process only one decision problem for costs of $\bar{z} = n$. Be aware that the `done_flag` does not necessarily state whether a solution is obtained. It may be set by a timeout. The flag `bb->found_feasible` in the function `bb_search` expresses whether a solution was found.

Performance measures were not performed as no comparison and no test data were available. Furthermore, the implementation of Posta does not directly yield a solution to the problem and just solved the decision problem.

The problem discussed in this section should be research further in the future.

5 Computational Results

In this chapter we examine the algorithm 4.2 on its computational characteristics. We have no other algorithm for comparison. McCormick et al. did not provide any source code or computational results.

In section 4.2.5 we analysed the worse case complexity of algorithm 4.2. Now we will analyse how the algorithm behaves in with test data.

5.1 Number of Rules

In section 4.2.5 we showed that (if $\gcd(r_1, r_2) = 1$) the worst case for the cardinality of the set of rules $\mathcal{R}_{r_1}$ is r_1.

However, the average of the number of rules behaves differently. In fig. 5.1 we see the mean cardinality of the set of rules. For any $r_1 = 1, \ldots, 5000$ we calculated how many rules are generated with an arbitrary r_2.

Thus, we note following correlation:

$$|\mathcal{R}_{r_1}| \approx \frac{1}{C} \cdot \log^2\left(r_1\right), \qquad C \in \mathbb{R} \tag{5.1.1}$$

In a machine scheduling environment with alternating job lengths this reduces the mean runtime.

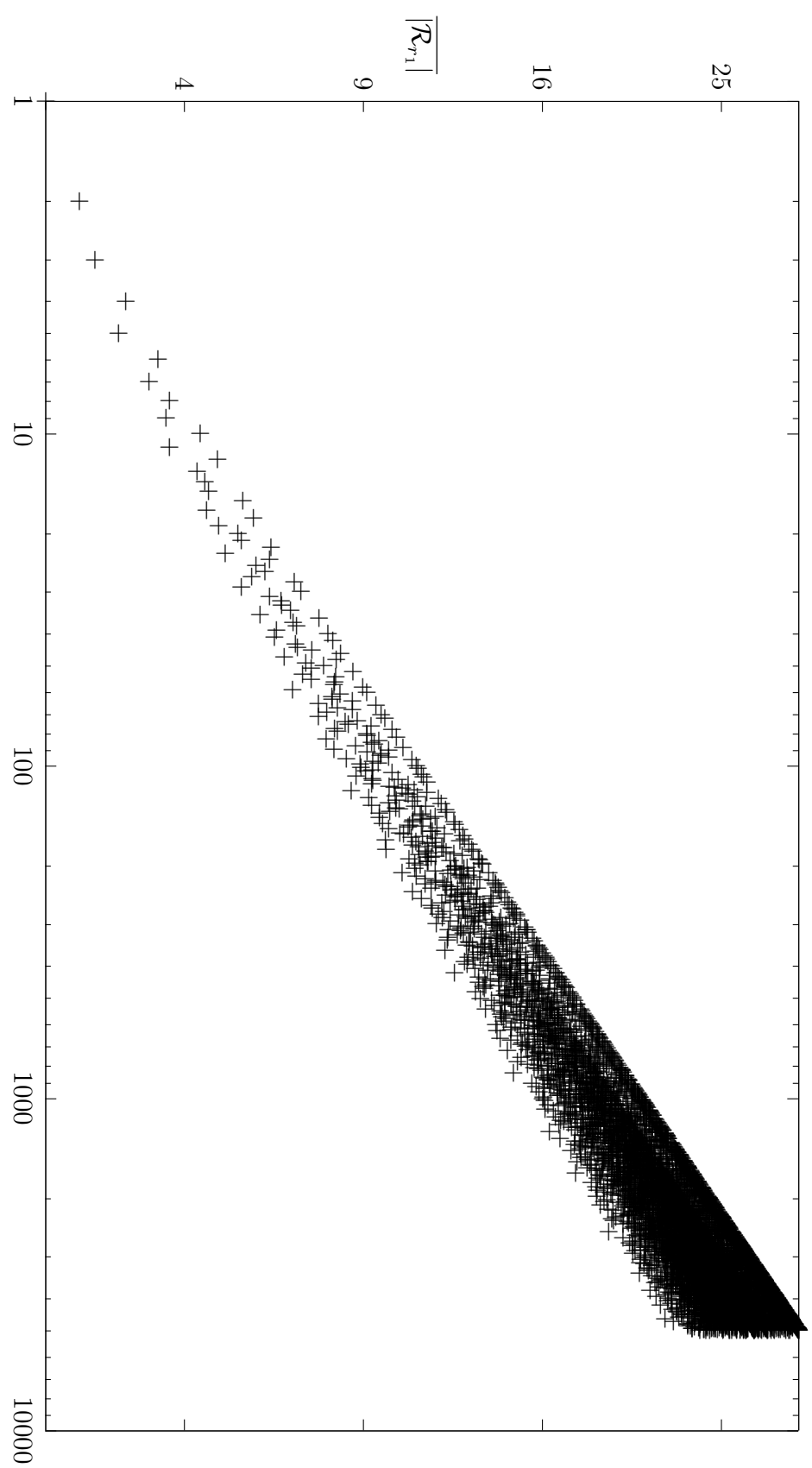


Figure 5.1: Average cardinality of $\mathcal{R}$ depending on r_1 for arbitrary r_2

We formulate this as a hypothesis:

5.1 Hypothesis

The average runtime for arbitrary r_1 and r_2 of algorithm 4.2 is

$$\mathcal{O}\left(m \cdot \log^2\left(r_1\right) + \frac{r_1 \cdot r_2}{\gcd^2(r_1, r_2)}\right) \tag{5.1.2}$$

5.2 Runtime of the Algorithm 4.2

The algorithm 4.2 was implemented in Python 3. The source code is provided in section B.2. The following tests where run on a *Intel® Core™ i7-2620M CPU @ 2.70GHz* with 8 GB of RAM.

Any data point depicted in the figures is a mean value of 100 test runs. We choose to use $r_1 = 7$ and $r_2 = 10$ as values for our block sizes, as the number of rules for each transformation direction is 3[1] Thus, we have no influence of the transformation direction on the result. Other values for r_1 and r_2 were tested as well, but as they showed no distinguishable difference, we only provide the following figures for $r_1 = 7$ and $r_2 = 10$.

For generating a test case a "solution" (β_l^3, β_l^4) is created and transformed into the smallest d_l that contain this solution. Then a random offset $\leq r_1$ is added to any d_l. Now the algorithm is executed to find a solution for $q_3 = \tilde{\beta}^3$ and $q_4 = \tilde{\beta}^4$.

Influence of m

At first we analyse correlation of m and the runtime. Figure 5.2 shows that the running time increases linearly with m.

The little steps at about 700, 1400, and 2100 could be caused by the implementation. There is no mathematical reason why they could appear. Test cases with other r_1 and r_2 and other parameter variants revealed that these steps always occur at the same positions. A possible explanation is that any increase m by 700 requires additional resources that have to be acquired.

Influence of $\max(d_l)$

Now we examine whether the maximal size of d_l, $k = 1, \ldots, m$ has an influence on the run time of the algorithm. Our complexity analysis stated an independence of the runtime and $d_{\max} = \max_{l \in \{1,\ldots,m\}}(d_l)$. Figure 5.3 approves this fact. There is no correlation recognisable, the runtime is nearly constant for small and big possible

[1] We skip the rule with $p = 0$ for the transformation $r_2 \to r_1$ as it is never applied because our best-filling fragmentation favours blocks of r_1.

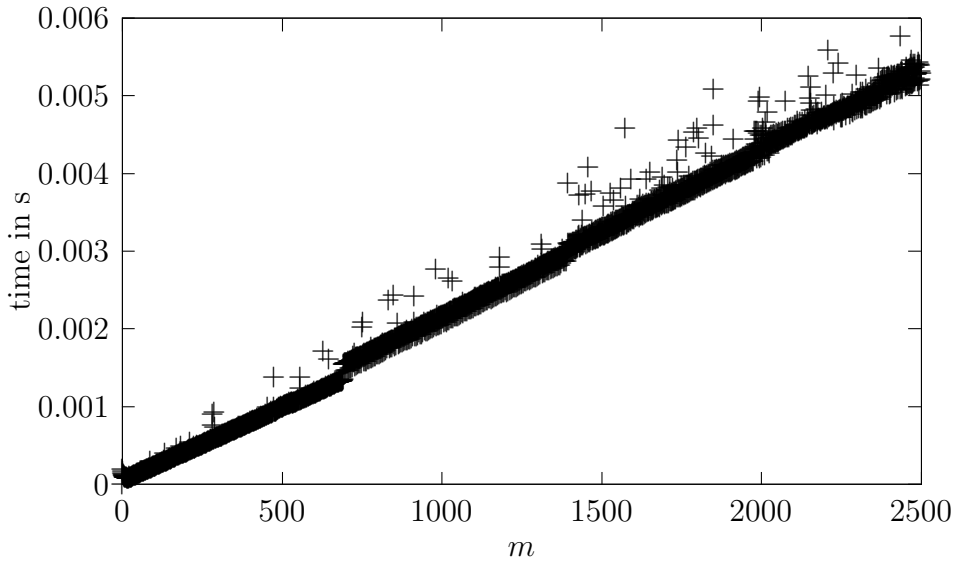


Figure 5.2: Runtime performance of algorithm 4.2 for $r_1 = 7$, $r_2 = 10$

values of d_l. A small decline of runtime is visible for very small $d_{\max}$. The reason is, that in small d_l only a few rules are applicable.

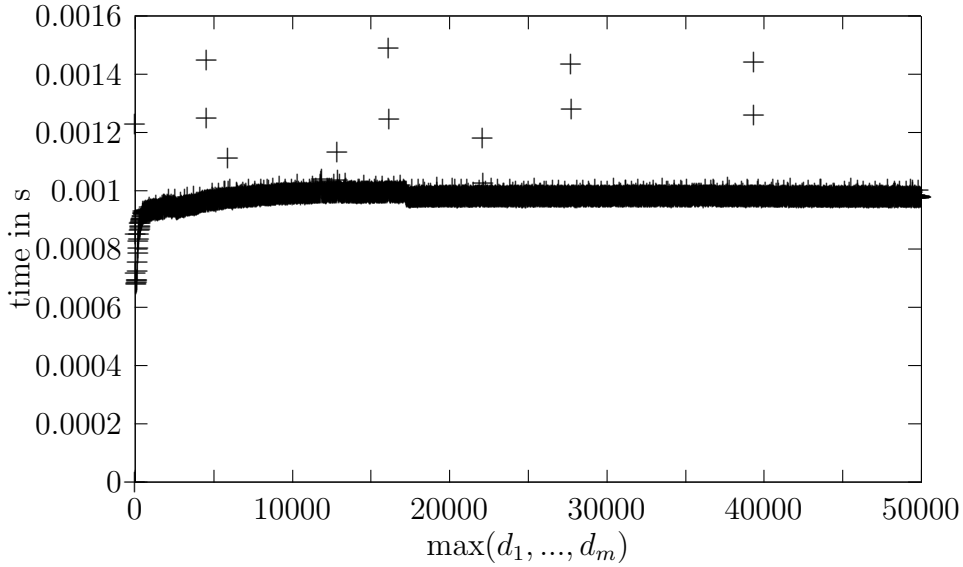


Figure 5.3: Runtime performance of algorithm 4.2 for $r_1 = 7$, $r_2 = 10$

Bibliography

[Bar+03] Cynthia Barnhart, Amy M. Cohn, Ellis L. Johnson, Diego Klabjan, George L. Nemhauser, and Pamela H. Vance. "Handbook of Transportation Science". In: ed. by Randolph W. Hall. Boston, MA: Springer US, 2003. Chap. Airline Crew Scheduling, pp. 517–560. ISBN: 9780306480584.

[Bra+05] Nadia Brauner, Yves Crama, Alexander Grigoriev, and Joris Van De Klundert. "A framework for the complexity of high-multiplicity scheduling problems". In: *Journal of combinatorial optimization* 9.3 (2005), pp. 313–323.

[Bra+07] Nadia Brauner, Yves Crama, Alexander Grigoriev, and Joris Van De Klundert. "Multiplicity and complexity issues in contemporary production scheduling". In: *Statistica Neerlandica* 61.1 (2007), pp. 75–91.

[Bru07] Peter Brucker. *Scheduling algorithms*. Vol. 5. Springer, 2007.

[CMM67] R.W. Conway, W.L. Maxwell, and L.W. Miller. *Theory of scheduling*. Addison-Wesley Educational Publishers Inc, 1967.

[CP01] John J. Clifford and Marc E. Posner. "Parallel machine scheduling with high multiplicity". In: *Mathematical programming* 89.3 (2001), pp. 359–383.

[CV92] Dirk G. Cattrysse and Luk N. Van Wassenhove. "A survey of algorithms for the generalized assignment problem". In: *European journal of operational research* 60.3 (1992), pp. 260–272.

[Det08] Paolo Detti. "Algorithms for multiprocessor scheduling with two job lengths and allocation restrictions". In: *Journal of Scheduling* 11.3 (2008), pp. 205–212.

[Din+88] David F. Dinges, Wayne G. Whitehouse, Emily Carota Orne, and Martin T. Orne. "The benefits of a nap during prolonged work and wakefulness". In: *Work & stress* 2.2 (1988), pp. 139–153.

[DN01] I.R. De Farias and George L. Nemhauser. "A family of inequalities for the generalized assignment polytope". In: *Operations Research Letters* 29.2 (2001), pp. 49–55.

[Eur14a] Europäische Kommission. *Verordnung (EU) Nr. 83/2014 der Kommission*. Amtsblatt der Europäischen Union. Jan. 29, 2014. URL: `http://www.lba.de/SharedDocs/Downloads/DE/B/Rechtsvorschriften/VO_83_2014.pdf`.

[Eur14b] European Aviation Safety Agency. *Commercial Air Transport by Aeroplane — Scheduled and Charter Operations CS-FTL.1*. Certification Specifications and Guidance Material. Jan. 31, 2014. URL: `https://easa.europa.eu/document-library/certification-specifications/cs-ftl1-initial-issue`.

[FJV86] Marshall L. Fisher, Ramchandran Jaikumar, and Luk N. Van Wassenhove. "A multiplier adjustment method for the generalized assignment problem". In: *Management Science* 32.9 (1986), pp. 1095–1103.

[FR09] Carlo Filippi and Giorgio Romanin-Jacur. "Exact and approximate algorithms for high-multiplicity parallel machine scheduling". In: *Journal of Scheduling* 12.5 (2009), pp. 529–541.

[Fra05] András Frank. "On Kuhn's Hungarian method—a tribute from Hungary". In: *Naval Research Logistics (NRL)* 52.1 (2005), pp. 2–5.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and intractability. A guide to the theory of NP-Completeness*. Bell Telephone Labaoratories, Incorporated. W. H. Freeman and Company, 1979. ISBN: 0716710447.

[GR89] Monique Guignard and Moshe B. Rosenwein. "An Improved Dual Based Algorithm for the Generalized Assignment Problem". In: *Operations Research* 37.4 (1989), pp. 658–663.

[Gra+79] Ronald L. Graham, Eugene L. Lawler, Jan Karel Lenstra, and A.H.G. Rinnooy Kan. "Optimization and approximation in deterministic sequencing and scheduling: a survey". In: *Annals of discrete mathematics* 5 (1979), pp. 287–326.

[GST97] Frieda Granot, Jadranka Skorin-Kapov, and Amir Tamir. "Using quadratic programming to solve high multiplicity scheduling problems on parallel machines". In: *Algorithmica* 17.2 (1997), pp. 100–110.

[HS91] Dorit S. Hochbaum and Ron Shamir. "Strongly polynomial algorithms for the high multiplicity scheduling problem". In: *Operations Research* 39.4 (1991), pp. 648–653.

[JN86] Kurt O. Jörnsten and Mikael Näsberg. "A new Lagrangian relaxation approach to the generalized assignment problem". In: *European Journal of Operational Research* 27.3 (1986), pp. 313–323.

[JV86] Roy Jonker and Ton Volgenant. "Improving the Hungarian assignment algorithm". In: *Operations Research Letters* 5.4 (1986), pp. 171–175.

[KT13] Sven O. Krumke and Clemens Thielen. "The generalized assignment problem with minimum quantities". In: *European Journal of Operational Research* 228.1 (2013), pp. 46–55.

[Kuh55] Harold W Kuhn. "The Hungarian method for the assignment problem". In: *Naval research logistics quarterly* 2.1-2 (1955), pp. 83–97.

[Leu82] Joseph Y.-T. Leung. “On scheduling independent tasks with restricted execution times”. In: *Operations Research* 30.1 (1982), pp. 163–171.

[LLK82] Eugene L. Lawler, Jan Karel Lenstra, and A.H.G. Rinnooy Kan. *Recent developments in deterministic sequencing and scheduling: a survey*. Springer, 1982.

[Lou64] Janice R. Lourie. “Topology and computation of the generalized transportation problem”. In: *Management Science* 11.1 (1964), pp. 177–187.

[MSS01] S. Thomas McCormick, Scott R. Smallwood, and Frits C.R. Spieksma. “A polynomial algorithm for multiprocessor scheduling with two job lengths”. In: *Mathematics of Operations Research* 26.1 (2001), pp. 31–49.

[MT81] Silvano Martello and Paolo Toth. “An algorithm for the generalized assignment problem”. In: *Operational research* 81 (1981), pp. 589–603.

[PDU05] Alexandre Pigatti, Marcus Poggi De Aragao, and Eduardo Uchoa. “Stabilized branch-and-cut-and-price for the generalized assignment problem”. In: *Electronic Notes in Discrete Mathematics* 19 (2005), pp. 389–395.

[PFM12] Marius Posta, Jacques A. Ferland, and Philippe Michelon. “An exact method with variable fixing for solving the generalized assignment problem”. In: *Computational Optimization and Applications* 52.3 (2012), pp. 629–644.

[Pos12] Marius Posta. *Generalized Assignment Problem solver*. Git repository. GitHub. 2012. URL: `https://github.com/postamar/gap-solver/`.

[Psa80] Harilaos N. Psaraftis. “A dynamic programming approach for sequencing groups of identical jobs”. In: *Operations Research* 28.6 (1980), pp. 1347–1359.

[Ros86] Moshe B. Rosenwein. “Design and application of solution methodologies to optimize problems in transportation logistics”. PhD thesis. University of Pennsylvania, 1986.

[RS75] G. Terry Ross and Richard M. Soland. “A branch and bound algorithm for the generalized assignment problem”. In: *Mathematical programming* 8.1 (1975), pp. 91–103.

[Sad+15] Ruslan Sadykov, François Vanderbeck, Artur Pessoa, and Eduardo Uchoa. “Column generation based heuristic for the generalized assignment problem”. In: *XLVII Simpósio Brasileiro de Pesquisa Operacional, Porto de Galinhas, Brazil* (2015).

[Sav97] Martin Savelsbergh. “A branch-and-price algorithm for the generalized assignment problem”. In: *Operations research* 45.6 (1997), pp. 831–841.

[Sha92] David F. Shallcross. “A polynomial algorithm for a one machine batching problem”. In: *Operations Research Letters* 11.4 (1992), pp. 213–218.

[Sta14] Statista GmbH. *Anzahl der Flüge in der weltweiten Luftfahrt von 2009 bis 2014 (in Millionen)*. 2014. URL: `http://de.statista.com/statistik/daten/studie/411620/umfrage/anzahl-der-weltweiten-fluege/` (visited on Feb. 22, 2016).

[Uwa12] Daniel Uwazie. “Approaches to Makespan”. Diploma thesis. Technische Universität Berlin, Fachbereich Mathematik, 2012.

A Examples

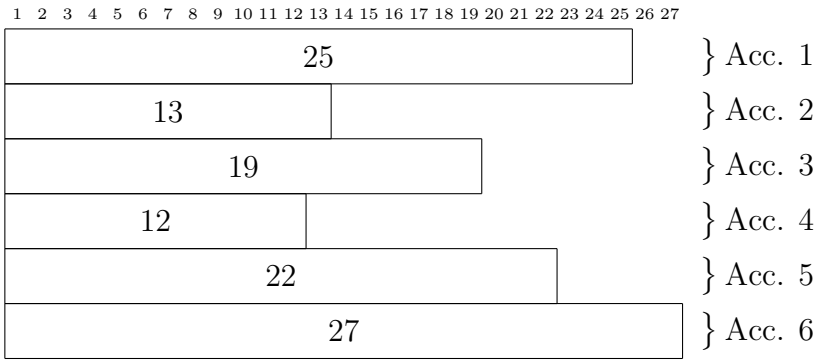
A.1 Example for Algorithm 4.1

Let us consider the following input for algorithm 4.1:

$$d = (25, 13, 19, 12, 22, 27), \tag{A.1.1}$$
$$q = (12, 20). \tag{A.1.2}$$

We now execute the algorithms, the decisions of the algorithm are commented.

▷ Requiring $12 \times \boxed{3}$ and $20 \times \boxed{4}$.



▷ Best-filling subdivision:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	
3	3	3	3	3	3	3	4																				} Acc. 1
3	3	3	4																								} Acc. 2
3	3	3	3	3	4																						} Acc. 3
3	3	3	3																								} Acc. 4
3	3	3	3	3	3	4																					} Acc. 5
3	3	3	3	3	3	3	3	3																			} Acc. 6

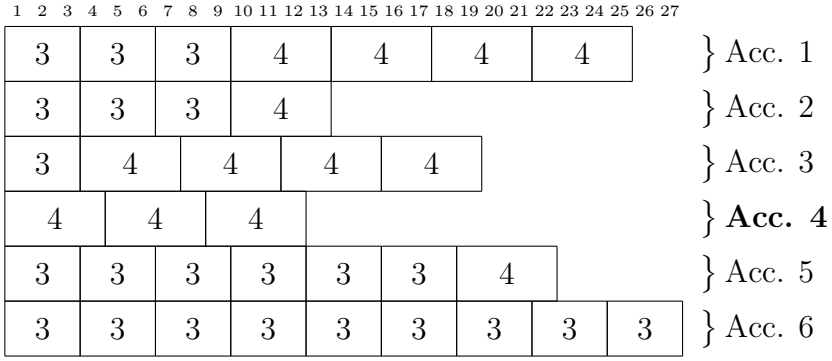
▷ After initalisation: need -22 × $\boxed{3}$ and 16 × $\boxed{4}$.
▷ Starting rule (-4,3)!
▷ Apply on accommodation 1:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	
3	3	3	4	4	4	4																					} **Acc. 1**
3	3	3	4																								} Acc. 2
3	3	3	3	3	4																						} Acc. 3
3	3	3	3																								} Acc. 4
3	3	3	3	3	3	4																					} Acc. 5
3	3	3	3	3	3	3	3	3																			} Acc. 6

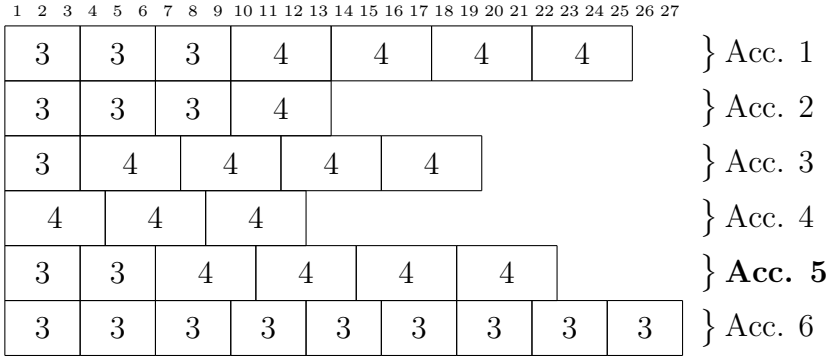
▷ Still need -18 × $\boxed{3}$ and 13 × $\boxed{4}$.
▷ Apply on accommodation 3:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	
3	3	3	4	4	4	4																					} Acc. 1
3	3	3	4																								} Acc. 2
3	4		4		4		4																				} **Acc. 3**
3	3	3	3																								} Acc. 4
3	3	3	3	3	3	4																					} Acc. 5
3	3	3	3	3	3	3	3	3																			} Acc. 6

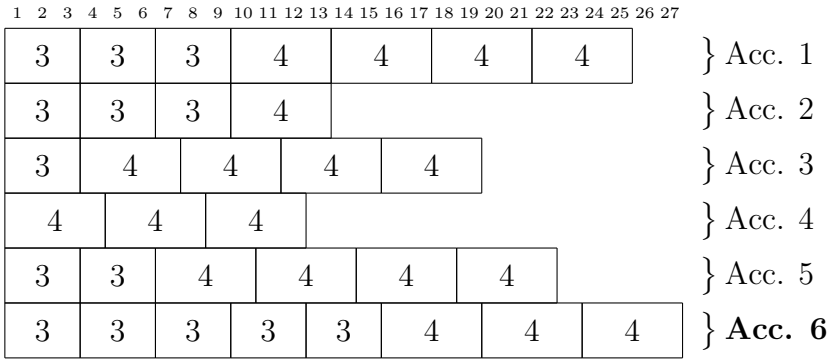
▷ Still need -14 × $\boxed{3}$ and 10 × $\boxed{4}$.
▷ Apply on accommodation 4:



▷ Still need -10 × $\boxed{3}$ and 7 × $\boxed{4}$.
▷ Apply on accommodation 5:

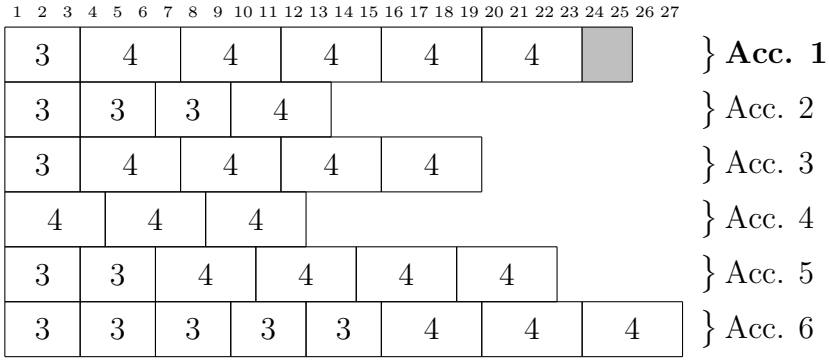


▷ Still need -6 × $\boxed{3}$ and 4 × $\boxed{4}$.
▷ Apply on accommodation 6:

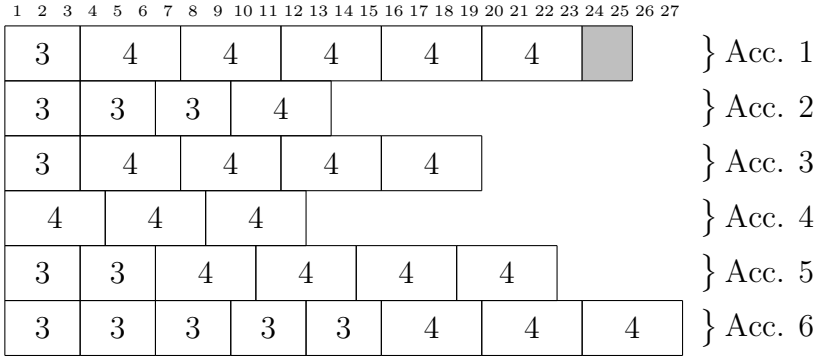


▷ Still need -2 × $\boxed{3}$ and 1 × $\boxed{4}$.
▷ Cannot apply rule anymore (-4,3): to few resources required or available.
▷ Starting rule (-3,2)!
▷ Cannot apply rule (-3,2): to few resources required or available.
▷ Starting rule (-2,1)!

▷ Apply on accommodation 1:



▷ Still need 0 × $\boxed{3}$ and 0 × $\boxed{4}$.
▷ Cannot apply rule (-2,1): to few resources required or available.
▷ Success! The completed subdivion is as follows:



A.2 Number of Rules

The following two figures show the number of rule generated by algorithm 4.4 depending on r_1 and r_2. For many instances the number of rules is fairly small.

Table A.1: Number of rules for the transformation direction $r_1 \to r_2$

$r_2 \downarrow$ / $r_1 \to$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49
2	1																																																	
3	1	2																																																
4	1	(1)	3																																														
5	1	2	2	4																																													
6	1	(1)	(1)	(2)	5																																												
7	1	2	3	2	3	6																																											
8	1	(1)	2	(1)	3	(3)	7																																										
9	1	2	(1)	4	2	(2)	4	8																																									
10	1	(1)	3	(2)	(1)	(2)	3	(4)	9																																								
11	1	2	2	2	5	2	4	4	5	10																																							
12	1	(1)	(1)	(1)	3	(1)	3	(2)	(3)	(5)	11																																						
13	1	2	3	4	3	6	2	3	3	4	6	12																																					
14	1	(1)	2	(2)	2	(3)	(1)	(2)	5	(3)	5	(6)	13																																				
15	1	2	(1)	2	(1)	(2)	7	2	(2)	(2)	5	(4)	7	14																																			
16	1	(1)	3	(1)	5	(2)	4	(1)	3	(3)	3	(3)	5	(7)	15																																		
17	1	2	2	4	3	2	3	8	2	4	6	4	4	6	8	16																																	
18	1	(1)	(1)	(2)	3	(1)	4	(4)	(1)	(2)	3	(2)	4	(4)	(5)	(8)	17																																
19	1	2	3	2	2	6	3	4	9	2	3	4	3	6	6	6	9	18																															
20	1	(1)	2	(1)	(1)	(3)	2	(2)	5	(1)	3	(2)	7	(3)	(3)	(4)	7	(9)	19																														
21	1	2	(1)	4	5	(2)	(1)	3	(3)	10	2	(2)	4	(2)	(3)	4	5	(6)	10	20																													
22	1	(1)	3	(2)	3	(2)	7	(2)	3	(5)	(1)	(2)	5	(4)	3	(4)	5	(5)	7	(10)	21																												
23	1	2	2	2	3	2	4	2	5	4	11	2	4	3	8	4	7	5	7	8	11	22																											
24	1	(1)	(1)	(1)	2	(1)	3	(1)	(2)	(3)	6	(1)	3	(3)	(3)	(2)	5	(3)	7	(5)	(7)	(11)	23																										
25	1	2	3	4	(1)	6	4	8	3	(2)	5	12	2	3	(2)	5	3	4	4	(4)	6	8	12	24																										
26	1	(1)	2	(2)	5	(3)	3	(4)	2	(3)	5	(6)	(1)	(2)	3	(3)	9	(3)	5	(4)	5	(6)	9	(12)	25																								
27	1	2	(1)	2	3	(2)	2	4	(1)	4	3	(4)	13	2	(2)	6	4	(2)	4	8	(4)	6	8	(8)	13	26																						
28	1	(1)	3	(1)	3	(2)	(1)	(2)	9	(2)	6	(3)	7	(1)	3	(2)	3	(5)	3	(3)	(3)	(5)	6	(6)	9	(13)	27																					
29	1	2	2	4	2	2	7	3	5	2	3	4	5	14	2	4	4	5	10	4	5	4	8	8	7	10	14	28																				
30	1	(1)	(1)	(2)	(1)	(1)	4	(2)	(3)	(1)	3	(2)	4	(7)	(1)	(2)	5	(2)	5	(2)	(3)	(5)	5	(4)	(5)	(7)	(9)	(14)	29																		
31	1	2	3	2	5	6	3	2	3	10	3	4	4	6	15	2	3	4	4	6	3	6	9	6	5	6	9	10	15	30																	
32	1	(1)	2	(1)	3	(3)	4	(1)	5	(5)	2	(2)	3	(4)	8	(1)	3	(3)	7	(3)	11	(3)	4	(3)	5	(5)	7	(7)	11	(15)	31																
33	1	2	(1)	4	3	(2)	3	8	(2)	4	(1)	(2)	7	6	(5)	16	2	(2)	3	3	(4)	(2)	5	(4)	4	6	(5)	7	8	(10)	16	32															
34	1	(1)	3	(2)	2	(2)	2	(4)	3	(3)	11	(2)	4	(3)	6	(8)	(1)	(2)	3	(4)	4	(6)	3	(4)	6	(4)	9	(6)	9	(8)	11	(16)	33														
35	1	2	2	2	(1)	2	(1)	4	2	(2)	6	2	5	(2)	(3)	6	17	2	4	(2)	(2)	4	12	4	(3)	10	5	(4)	9	(6)	10	12	17	34													
36	1	(1)	(1)	(1)	5	(1)	7	(2)	(1)	(3)	5	(1)	4	(4)	(3)	(4)	9	(1)	3	(2)	(3)	(3)	5	(2)	4	(4)	(3)	(4)	5	(5)	7	(8)	(11)	(17)	35												
37	1	2	3	4	3	6	4	3	9	4	5	12	3	3	3	4	7	18	2	3	6	8	6	7	3	4	5	4	6	6	6	8	9	12	18	36											
38	1	(1)	2	(2)	3	(3)	3	(2)	5	(2)	3	(6)	2	(3)	8	(4)	5	(9)	(1)	(2)	5	(3)	3	(4)	13	(3)	7	(6)	5	(6)	7	(6)	8	(9)	13	(18)	37										
39	1	2	(1)	2	2	(2)	4	2	(3)	2	6	(4)	(1)	3	(3)	4	5	(6)	19	2	(2)	4	5	(3)	6	(2)	(3)	4	11	(4)	10	6	(6)	10	11	(12)	19	38									
40	1	(1)	3	(1)	(1)	(2)	3	(1)	3	(1)	3	(3)	13	(2)	(2)	(2)	7	(5)	10	(1)	3	(3)	3	(2)	(3)	(7)	3	(3)	6	(3)	7	(4)	7	(7)	(7)	(9)	13	(19)	39								
41	1	2	2	4	5	2	2	8	5	10	3	4	7	2	3	5	5	5	7	20	2	4	4	4	4	6	14	4	5	6	4	5	5	10	10	8	10	14	20	40							
42	1	(1)	(1)	(2)	3	(1)	(1)	(4)	(2)	(5)	3	(2)	5	(1)	(2)	(3)	3	(3)	7	(10)	(1)	(2)	3	(2)	9	(4)	(5)	(2)	5	(3)	7	(4)	(5)	(5)	(5)	(6)	9	(10)	(13)	(20)	41						
43	1	2	3	2	3	6	7	4	3	4	2	4	4	14	3	6	9	4	7	8	21	2	3	3	5	3	4	8	3	6	5	12	6	7	6	6	7	9	12	14	21	42					
44	1	(1)	2	(1)	2	(3)	4	(2)	2	(3)	(1)	(2)	4	(7)	2	(2)	4	(3)	4	(5)	11	(1)	3	(2)	7	(5)	5	(4)	15	(3)	5	(4)	(3)	(5)	11	(5)	7	(7)	11	(10)	15	(21)	43				
45	1	2	(1)	4	(1)	(2)	3	3	(1)	(2)	11	(2)	3	6	(1)	4	3	(2)	5	(4)	(7)	22	2	(2)	(2)	4	(2)	7	6	(2)	4	8	(5)	4	(4)	(4)	7	8	(7)	(8)	11	(14)	22	44			
46	1	(1)	3	(2)	5	(2)	4	(2)	9	(3)	6	(2)	7	(4)	15	(2)	4	(5)	4	(4)	6	(11)	(1)	(2)	5	(4)	5	(3)	5	(8)	3	(4)	4	(7)	5	(5)	5	(7)	7	(8)	9	(11)	15	(22)	45		
47	1	2	2	2	3	2	3	2	5	4	5	2	4	6	8	2	5	5	3	8	5	8	23	2	4	6	3	10	4	5	16	4	4	5	13	5	6	6	11	8	11	10	13	16	23	46	
48	1	(1)	(1)	(1)	3	(1)	2	(1)	(3)	(2)	5	(1)	5	(3)	(5)	(1)	3	(2)	10	(3)	(4)	(6)	12	(1)	3	(3)	(3)	(3)	3	(3)	7	(2)	(3)	(5)	5	(3)	5	(7)	(5)	(5)	11	(7)	10	(11)	(15)	(23)	47	
49	1	2	3	4	2	6	(1)	8	3	2	3	12	4	(2)	6	16	3	4	5	4	(3)	6	9	24	2	3	4	(2)	6	4	5	9	3	4	(3)	6	4	8	12	8	6	(6)	8	12	12	16	24	48	
50	1	(1)	2	(2)	(1)	(3)	7	(4)	5	(1)	6	(6)	3	(4)	(3)	(8)	2	(3)	4	(2)	5	(5)	8	(12)	(1)	(2)	3	(3)	4	(2)	5	(5)	17	(3)	(3)	(4)	8	(4)	5	(4)	6	(6)	7	(8)	(9)	(12)	17	(24)	49

The numbers in brackets do not satisfy $\gcd(r_1, r_2) = 1$.

Table A.2: Number of rules for the transformation direction $r_1 \leftarrow r_2$

$\downarrow r_2$ $r_1 \rightarrow$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49
2	1																																																
3	1	2																																															
4	1	(1)	2																																														
5	1	2	3	2																																													
6	1	(1)	(1)	(2)	2																																												
7	1	2	2	4	3	2																																											
8	1	(1)	3	(1)	3	(2)	2																																										
9	1	2	(1)	2	5	(2)	3	2																																									
10	1	(1)	2	(2)	(1)	(3)	4	(2)	2																																								
11	1	2	3	4	2	6	3	3	3	2																																							
12	1	(1)	(1)	(1)	3	(1)	4	(2)	(2)	(2)	2																																						
13	1	2	2	2	3	2	7	4	5	4	3	2																																					
14	1	(1)	3	(2)	5	(2)	(1)	(4)	3	(3)	3	(2)	2																																				
15	1	2	(1)	4	(1)	(2)	2	8	(3)	(2)	3	(2)	3	2																																			
16	1	(1)	2	(1)	2	(3)	3	(1)	5	(3)	6	(2)	4	(2)	2																																		
17	1	2	3	2	3	6	4	2	9	4	3	4	5	3	3	2																																	
18	1	(1)	(1)	(2)	3	(1)	3	(2)	(1)	(5)	5	(2)	4	(3)	(2)	(2)	2																																
19	1	2	2	4	5	2	4	3	2	10	5	4	7	3	3	4	3	2																															
20	1	(1)	3	(1)	(1)	(2)	7	(2)	3	(1)	6	(3)	3	(4)	(2)	(2)	3	(2)	2																														
21	1	2	(1)	2	2	(2)	(1)	4	(2)	2	11	(4)	4	(2)	(3)	6	5	(2)	3	2																													
22	1	(1)	2	(2)	3	(3)	2	(4)	5	(2)	(1)	(6)	4	(3)	8	(3)	4	(3)	4	(2)	2																												
23	1	2	3	4	3	6	3	8	3	4	2	12	5	6	3	5	3	4	3	3	3	2																											
24	1	(1)	(1)	(1)	5	(1)	4	(1)	(3)	(3)	3	(1)	7	(4)	(3)	(2)	4	(2)	3	(2)	(2)	(2)	2																										
25	1	2	2	2	(1)	2	3	2	5	(2)	3	2	13	6	(3)	4	9	5	7	(2)	5	4	3	2																									
26	1	(1)	3	(2)	2	(2)	4	(2)	9	(3)	3	(2)	(1)	(7)	6	(4)	3	(5)	4	(4)	6	(3)	3	(2)	2																								
27	1	2	(1)	4	3	(2)	7	3	(1)	4	6	(2)	2	14	(5)	4	5	(2)	5	3	(3)	4	3	(2)	3	2																							
28	1	(1)	2	(1)	3	(3)	(1)	(2)	2	(5)	3	(2)	3	(1)	8	(4)	7	(3)	10	(3)	(2)	(3)	4	(2)	4	(2)	2																						
29	1	2	3	2	5	6	2	4	3	10	5	4	4	2	15	6	5	4	3	6	4	8	3	3	5	3	3	2																					
30	1	(1)	(1)	(2)	(1)	(1)	3	(4)	(2)	(1)	5	(2)	5	(2)	(1)	(8)	5	(3)	4	(2)	(4)	(3)	5	(2)	(2)	(3)	(2)	(2)	2																				
31	1	2	2	4	2	2	4	8	5	2	6	4	4	3	2	16	7	5	5	4	11	4	3	4	7	6	3	4	3	2																			
32	1	(1)	3	(1)	3	(2)	3	(1)	3	(2)	11	(3)	7	(3)	3	(1)	9	(5)	4	(3)	3	(6)	6	(2)	5	(4)	4	(2)	3	(2)	2																		
33	1	2	(1)	2	3	(2)	4	2	(3)	4	(1)	(4)	3	3	(2)	2	17	(6)	7	8	(3)	(2)	5	(3)	9	4	(3)	4	5	(2)	3	2																	
34	1	(1)	2	(2)	5	(3)	7	(2)	5	(3)	2	(6)	4	(4)	3	(2)	(1)	(9)	7	(4)	5	(3)	12	(4)	4	(5)	3	(3)	3	(3)	4	(2)	2																
35	1	2	3	4	(1)	6	(1)	3	9	(2)	3	12	4	(2)	(2)	4	2	18	7	(4)	(3)	6	3	7	(3)	3	5	(2)	3	(2)	3	3	3	2															
36	1	(1)	(1)	(1)	2	(1)	2	(2)	(1)	(3)	3	(1)	5	(3)	(3)	(2)	3	(1)	10	(5)	(4)	(5)	5	(2)	6	(4)	(2)	(3)	8	(2)	6	(2)	(2)	(2)	2														
37	1	2	2	2	3	2	3	4	2	4	3	2	7	6	8	6	3	2	19	8	5	4	4	4	13	6	5	10	4	5	7	4	5	4	3	2													
38	1	(1)	3	(2)	3	(2)	4	(4)	3	(5)	6	(2)	13	(4)	3	(3)	5	(2)	(1)	(10)	6	(5)	9	(4)	3	(7)	4	(3)	6	(3)	4	(4)	4	(3)	3	(2)	2												
39	1	2	(1)	4	5	(2)	3	8	(2)	10	3	(2)	(1)	6	(3)	5	4	(2)	2	20	(7)	6	5	(4)	4	(2)	(5)	7	3	(4)	3	5	(3)	3	3	(2)	3	2											
40	1	(1)	2	(1)	(1)	(3)	4	(1)	5	(1)	5	(2)	2	(7)	(3)	(2)	3	(3)	3	(1)	11	(6)	8	(3)	(3)	(3)	14	(4)	4	(2)	4	(2)	4	(3)	(2)	(2)	4	(2)	2										
41	1	2	3	2	2	6	7	2	3	2	5	4	3	14	6	4	4	4	4	2	21	8	6	6	6	4	3	8	5	4	11	6	9	3	3	6	5	3	3	2									
42	1	(1)	(1)	(2)	3	(1)	(1)	(2)	(3)	(2)	6	(2)	4	(1)	(5)	(4)	9	(2)	3	(2)	(1)	(11)	8	(4)	4	(4)	(3)	(2)	6	(3)	4	(6)	(3)	(5)	(2)	(2)	4	(3)	(2)	(2)	2								
43	1	2	2	4	3	2	2	3	5	4	11	4	5	2	8	4	3	5	3	3	2	22	9	8	5	10	7	4	15	5	5	3	5	4	5	8	7	4	3	4	3	2							
44	1	(1)	3	(1)	5	(2)	3	(2)	9	(3)	(1)	(3)	4	(2)	15	(4)	5	(5)	7	(2)	3	(1)	12	(6)	5	(4)	5	(3)	3	(8)	5	(3)	(2)	(4)	3	(3)	5	(4)	3	(2)	3	(2)	2						
45	1	2	(1)	2	(1)	(2)	4	4	(1)	(2)	2	(4)	7	3	(1)	6	7	(2)	4	(2)	(2)	2	23	(8)	(5)	6	(3)	4	5	(2)	7	4	(3)	12	(3)	(2)	4	4	(3)	(2)	5	(2)	3	2					
46	1	(1)	2	(2)	2	(3)	3	(4)	2	(3)	3	(6)	3	(3)	2	(8)	5	(3)	5	(4)	5	(2)	(1)	(12)	7	(5)	5	(6)	5	(3)	16	(5)	8	(3)	7	(4)	10	(3)	5	(3)	6	(3)	4	(2)	2				
47	1	2	3	4	3	6	4	8	3	4	3	12	4	3	3	16	5	4	10	3	6	4	2	24	9	6	9	4	6	6	3	9	7	5	3	6	5	6	3	4	3	4	3	3	3	2			
48	1	(1)	(1)	(1)	3	(1)	7	(1)	(2)	(5)	3	(1)	4	(4)	(2)	(1)	7	(3)	3	(3)	(3)	(3)	3	(1)	13	(7)	(5)	(4)	11	(3)	4	(2)	(6)	(4)	6	(2)	6	(3)	(4)	(2)	3	(2)	4	(2)	(2)	(2)	2		
49	1	2	2	2	5	2	(1)	2	5	10	6	2	5	(2)	3	2	9	5	4	6	(2)	4	3	2	25	10	7	(4)	5	6	5	4	17	7	(3)	5	13	4	3	4	9	(2)	7	3	5	4	3	2	
50	1	(1)	3	(2)	(1)	(2)	2	(2)	3	(1)	3	(2)	7	(3)	(2)	(2)	17	(5)	5	(2)	4	(3)	3	(2)	(1)	(13)	9	(6)	6	(3)	5	(4)	3	(9)	(4)	(5)	4	(7)	7	(2)	6	(5)	8	(4)	(2)	(3)	3	(2)	2

The numbers in brackets do not satisfy $\gcd(r_1, r_2) = 1$.

B Source Code

B.1 Python Code for Algorithm 4.1

```
#!/usr/bin/python3

legs = [7,9,16,12,13,2]

rest = (5,10)

#best-filling subdivision
def subdivide(figure):
    if figure < 3:
        return (0, 0)
    if figure == 5:
        return (0,1)

    b4 = figure % 3
    b3 = (figure - 4 * b4) // 3

    return (b3, b4)

def total(legs):
    return tuple([sum(x) for x in zip(*legs)])

def apply_rule_if_possible(blocks, req, bal3, bal4, durs):
    for i in range(len(blocks)):
        t = total(blocks)
        while blocks[i][0] >= -bal3 and blocks[i][1] >= -bal4:
            if not (    bal3 < 0 and bal4 > 0 and req[0] - t[0] <=
                bal3 and req[1] - t[1] >= bal4
```

```
                    or bal4 < 0 and bal3 > 0 and req[0] - t[0] >= bal3
                        and req[1] - t[1] <= bal4):

                    return blocks
                blocks[i] = (blocks[i][0] + bal3, blocks[i][1] + bal4)
                t = total(blocks)
    return blocks

def algo(legs, req):
    #init
    blocks = [subdivide(l) for l in legs]
    t = total(blocks)

    if t[0] < req[0] and t[1] > req[1]:
        blocks = apply_rule_if_possible(blocks, req, 1, -1, legs)
    elif t[0] > req[0] and t[1] < req[1]:
        blocks = apply_rule_if_possible(blocks, req, -4, 3, legs)
        blocks = apply_rule_if_possible(blocks, req, -3, 2, legs)
        blocks = apply_rule_if_possible(blocks, req, -2, 1, legs)

    t = total(blocks)
    if t[0] >= req[0] and t[1] >= req[1]:
        print('Success!')
    else:
        print('Failed!')

    return blocks

b = algo(legs, rest)
```

B.2 Python Code for Algorithm 4.2

```
#!/usr/bin/python3
import math

d = [30555, 20847, 18803, 20386, 12188, 12313, 25526, 13629, 27349,
     36081, 27634, 26996, 26221, 40293, 27438, 34475, 14256, 17029,
    12709, 38824, 23378, 14332, 38348, 8107, 41965, 38428, 20855,
    4048, 41864, 43943, 9589, 7506, 9725, 29969, 25682, 5905, 2635,
    9303, 17933, 34795, 41709, 8712, 10376, 29019, 20375, 6518,
    33828, 30221, 36662, 36550]

q = (1259, 1281)

r1 = 101
r2 = 809

def init(d, r1, r2):
    split = [None] * (r1 * r2)
```

```
    for j in range(0, r1):
        for i in range(0, r2):
            if i*r1 + j*r2 >= r1*r2:
                break
            else:
                split[i*r1 + j*r2] = (i, j)
    last = (0,0)
    for i in range(0, r1*r2):
        if split[i] == None:
            split[i] = last
        else:
            last = split[i]

    mods = [None] * r1
    for i in range(0, r1):
        mods[(i*r2) % r1] = i

    res = [None] * len(d)

    for i in range(0, len(d)):
        if d[i] < r1*r2:
            res[i] = split[d[i]]
        else:
            b2 = mods[d[i] % r1]
            b1 = int((d[i] - b2 * r2)/r1)
            res[i] = (b1, b2)

    return res



def generate_rules(r1, r2, norm_dir):
    all_rules = []
    if norm_dir: #r1->r2
        for i in range (1, r1 + 1):
            need = math.ceil(r2*i / r1)
            penalty = need*r1 - i*r2
            all_rules.append((-need, i, penalty))
    else:
        for i in range (1, r1 + 1):
            trans = (r2*i) // r1
            penalty = i*r2 - trans*r1
            all_rules.append((trans, -i, penalty))

    rules = []
    penalty = r2 #infinity
    for i in range(0, len(all_rules)):
        (a, b, pen) = all_rules[i]
        if pen < penalty:
            rules.append((a, b, pen))
            penalty = pen
```

```
64
65
66      rules.sort(key=lambda r: r[2])
67      return rules
68
69  def total(blocks):
70       return tuple([sum(x) for x in zip(*blocks)])
71
72  def diff(a, b):
73      (a1, a2) = a
74      (b1, b2) = b
75      return (a1-b1, a2-b2)
76
77  def apply_rule(b, n, rule, times):
78      (b1, b2) = b
79      (n1, n2) = n
80      (bal1, bal2, pen) = rule
81      return ((b1+bal1*times, b2+bal2*times), (n1-bal1*times, n2-bal2
            *times))
82
83  def algo(d, r1, r2, q):
84      gcd = math.gcd(r1,r2)
85      if gcd > 1:
86          r1 = r1 // gcd;
87          r2 = r2 // gcd;
88          d = [e // gcd for e in d]
89
90
91      b = init(d, r1, r2)
92
93      #calculate how much is required (n: needed)
94      (n1, n2) = diff(q, total(b))
95
96      if n1 <= 0 and n2 <= 0:
97          return b
98      elif n1 < 0:
99          rules = generate_rules(r1, r2, True)
100         for (bal1, bal2, pen) in rules:
101             if n2 <= 0:
102                 break
103             for i in range(0, len(b)):
104                 times = (-n1)//(-bal1) #how often possible in total
105                 times = min(math.ceil(n2/bal2), times) #how often
                        needed in total
106                 if times <= 0:
107                     break
108
109                 (b1, b2) = b[i]
110                 if b1 < -bal1: #not sufficient in this subdivision
111                     continue
112                 times = min(b1//(-bal1), times)
```

```
113
114                 (b[i], (n1,n2)) = apply_rule((b1, b2), (n1, n2), (
                        bal1, bal2, pen), times)
115
116     elif n2 < 0:
117         rules = generate_rules(r1, r2, False)
118         for (bal1, bal2, pen) in rules:
119             if n1 <= 0:
120                 break
121             for i in range(0, len(b)):
122                 times = (-n2)//(-bal2) #how often possible in total
123                 times = min(math.ceil(n1/bal1), times) #how often
                        needed in total
124                 if times <= 0:
125                     break
126
127                 (b1, b2) = b[i]
128                 if b2 < -bal2: #not sufficient in this subdivision
129                     continue
130                 times = min(b2//(-bal2), times)
131
132                 (b[i], (n1,n2)) = apply_rule((b1, b2), (n1, n2), (
                        bal1, bal2, pen), times)
133
134     if n1 > 0 or n2 > 0:
135         return None
136
137     return b
138
139
140 b = algo(d, r1, r2, q)
```

B.3 Python Code for Converting Simple IFR to GAP

```
1  #!/usr/bin/python3
2
3  crew_cnt = 6
4  acc_cnt = 6
5
6  crew_rest = [120, 90, 120, 120, 90, 120]
7  acc_avail = [150, 180, 90, 90, 240, 150]
8
9  accs = [[1,2],[3,4],[5,6],[1,2,3,4],[1,3,5,6],[2,4]]
10
11 infty = max(acc_avail) + 1
12
13 costs = [[crew_cnt+1 for x in range(crew_cnt)] for y in range(
       acc_cnt)]
14 res = [[infty for x in range(crew_cnt)] for y in range(acc_cnt)]
```

```
15
16 for a in range(acc_cnt):
17     for crw in accs[a]:
18         res[a][crw - 1] = crew_rest[crw - 1]
19         costs[a][crw - 1] = 1
20
21
22 #costs 2x
23
24 #acc_avail
25
26 def savetofile(file, str):
27     f = open(file,"w")
28     f.write(str)
29     f.close()
30
31
32 #no. accommodation, no. crew members
33 out_str = "{0} {1}\n".format(acc_cnt, crew_cnt)
34
35 #cost matrices for assignments
36 for a in range(acc_cnt):
37     cost = costs[a]
38     for c in cost:
39         out_str += str(c)+" "
40     out_str += "\n"
41
42 #resource matrices for assignments
43 for a in range(acc_cnt):
44     re = res[a]
45     for r in re:
46         out_str += str(r)+" "
47     out_str += "\n"
48
49 #capacities of accommodations
50 for a in range(acc_cnt):
51     out_str += str(acc_avail[a])+" "
52
53
54 savetofile("inflight_rest_gap", out_str)
```