FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Solving Multimodal Resource Constrained Project Scheduling Problems

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Martin Sturm
Matrikelnummer 0526296

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Univ.-Prof. Dipl.-Ing. Dr.techn. Günther Raidl

Wien, 06.09.2012

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Solving Multimodal Resource Constrained Project Scheduling Problems

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computational Intelligence

by

Martin Sturm
Registration Number 0526296

to the Faculty of Informatics
at the Vienna University of Technology

Advisor:
Univ.-Prof. Dipl.-Ing. Dr.techn. Günther Raidl

Vienna, 06.09.2012

(Signature of Author)

(Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Martin Sturm
Machstrasse 3, 1020 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

I want to thank my advisor Günther Raidl as well as Hannes Obweger and Martin Suntinger from UC4 for making this thesis possible as well as for providing many interesting ideas and being a great help overall. I also want to thank the company UC4 in general for the provided hardware, the caffeine and especially for not turning off the air condition on the weekends. I also want to greet my fellow graduates at UC4, Philip Limbeck and Albert Kavelar. Their company during the long days in the office was nothing but a pleasure.
Thanks also to my parents - to my father Werner for always questioning me about the progress of the work and to my mother Melitta for not doing so. I also owe serious debt to my girlfriend Claudia for the weekends in the office, her patience and inspiration.

Abstract

This work is concerned with the task of mathematical modeling and the heuristical resolution of complex scheduling problems in the context of automated IT environments. Today's corporate IT systems often contain several hundred thousand automated jobs that have to be executed in the heterogeneous network on a daily basis. Because delays of the execution or finalization of these jobs is often coupled with high costs by using legal service level agreements (SLAs), an optimization of the processes in the system is of crucial importance.
In order to tackle a practical scheduling problem with the means of scientific optimization theory it is necessary to find a formalization in mathematical terms. This subproblem is considered in the first part of this work, where a number of models is examined for their adequacy. Because of their lack of flexibility, the classical machine scheduling models are discarded in favor of the broader project scheduling models. As final formalization the very general multi-mode resource-constrained project scheduling problem with generalized precedence relations (MRCPSP/max or MRCPSP-GPR) is chosen. This choice is justified by the fact that this model allows the modeling of resources in a more adequate way to capture the broader concept of a resource in IT environments. Furthermore it enables the formalization of complex timing constraints in a natural way.
This increased flexibility comes with a tradeoff in the form of increased complexity. It is shown that the resolution of MRCPSP/max instances with mixed integer programming methods is not practical, even for small instances. Therefore in the course of this work a software library has been implemented with the goal to provide solutions with high quality in a practical time frame by using heuristic optimization methods. Following the proposals in the scientific literature genetic algorithms serve as the underlying metaheuristic. This work provides an overview on the details of available implementations from the literature and presents a combination of several approaches in the form of a flexible software library. Furthermore we present new evolutionary operators which are proven to perform well in extensive benchmark tests.

Kurzfassung

Diese Arbeit befasst sich mit der mathematischen Modellierung und heuristischen Lösung komplexer Scheduling Probleme im Kontext automatisierter IT Umgebungen. In solchen Umgebungen werden oft hunderttausende automatisierte Jobs in einem verteilten Netzwerk von heterogenen Maschinen ausgeführt. Da Verzögerungen in der Ausführung beziehungsweise der Beendigung dieser Jobs, aufgrund von Service Level Agreements, oft mit hohen Kosten verbunden sind, ist eine Optimierung der Abläufe eine Aufgabe mit von grundlegender Bedeutung.
Um das Problem mit optimierungstheoretischen Mitteln behandeln zu können muss eine geeignete Formalisierung gefunden werden. In einem ersten Schritt werden einige Modelle auf ihre Adäquatheit hin evaluiert. Aufgrund ihrer mangelnden Flexibilität wird anstelle von klassischen Maschinen-Scheduling Modellen den extensiveren Projekt-Scheduling Modellen der Vorzug gegeben. Die endgültige Formulierung schließlich erfolgt als multimodales ressourcenbeschränktes Projekt-Scheduling Problem mit verallgemeinerten Vorgängerbeziehungen (MRCPSP/max oder MRCPSP-GPR). Dieses trägt nicht nur der Tatsache Rechnung, dass im Gegensatz zu Maschinenumgebungen in IT-Umgebungen das Konzept der Ressource ein erheblich flexiblerer ist, sondern erlaubt auch eine Erweiterung der Abhängigkeiten zwischen Arbeitsschritten. Während diese in Maschinenumgebungen meist von kausalen Abhängigkeiten dominiert werden treten in IT-Umgebungen flexiblere Zeitbedingungen auf.
All diese Flexibilität in der Modellierung führt typischerweise zu einer höheren Komplexität der Modelle. Es wird gezeigt, dass sich das MRCPSP/max schon bei relativ geringer Instanzgröße nicht mehr praktikabel mit exakten Methoden lösen lässt. Daher wurde im Zuge dieser Arbeit eine Software-Bibliothek erstellt, die hochqualitative Lösungen in annehmbaren Zeitrahmen mit heuristischen Optimierungsverfahren liefern kann. Der Literatur zu dem Thema folgend dient hierbei ein genetischer Algorithmus als Metaheuristik. Die Arbeit gibt einen Überblick zu den Details bereits vorhandenen Implementierungen und kombiniert die Ansätze zu einer flexiblen Software-Lösung. Daneben stellen wir auch neue evolutionäre Operatoren vor, die sich bei umfassenden Tests mit Benchmark-Instanzen bewährt haben.

Contents

CHAPTER 1

Introduction

1.1 Motivation

This work concurs with the area of IT process automation. In the complex IT environments in today's big corporations, where hundreds of thousands tasks are worked off on a daily basis, IT process automation is a key technology. Not only the volume, but also the diversity and complexity of IT services calls for intelligent automation techniques that release the responsible technicians from often highly repetitive tasks. This enables an efficient productive administration with reduced costs.
The development of IT process scheduling is rooted in the job scheduling problems that emerged with industrialization and assembly line work. It is not a coincidence that many of the classical scheduling problems have names like Job Shop Problem or Flow Shop Problem. Back then and now the targets are the same: execute complex jobs that might need treatment on different machines in a defined sequential orders with certain degrees of efficiency. Efficiency in this context can be defined in a number of ways. It might be the total time needed to execute a set of jobs, the compliance of given deadlines, the minimization of overall costs or a combination of all the above.
In this work an implementation of such an automation system is examined. The UC4[1] Operation Manager (OM) provides the means to realize all the tasks that have to be dealt with in a modern automated IT environment. This includes the control in diverse and distributed systems, effective runbook automation as well as monitoring the tasks, reporting their outcomes and maintaining a huge amount of data and statistics that record the behavior and performance of the system.
Because of the increasing workload the IT infrastructure has to deal with, an efficient use of the available resources is increasingly important. Especially in highly utilized systems it is a key concern of the executives to fulfill the service level agreements (SLAs) they committed to. These SLAs are usually recorded in a legal contract and formulated in a way that can be implemented in the systems. In general these SLAs are time constraints or performance values. A violation

[1] www.uc4.com

of them is usually coupled with a fee that is augmented over time.
This monetary aspect increases the need for efficient methods even further. Since scheduling problems form a well-known subclass of optimization problems, which is easily explainable because of the obvious possibilities of application in the real world, it seems like a logical step to formalize the problem and process it with the powerful tools provided by optimization theory. This statement summarizes the scope of the work at hand.
In a first step the domain should be explored and formalized with a strict mathematical model. Major requirements for this formalization are a certain extendibility to implement new concepts as well as a focus on flexibility for both modeling approaches and the conception of objective functions. After this, a library of competitive state of the art techniques should be provided to solve problem instances of practical size.

1.2 Problem Statement

In this section the domain of the UC4 Operations Manager is discussed. In the following sections all the objects of interest in the UC4 system will be presented. Afterwards the desired improvements resulting from this work are summarized.

The UC4 Operations Manager

The UC4 Operations Manager (OM) is a program used for task automation and event processing in IT environments. A task in this context is usually a job that contains a number of commands for the operating system or the invocation and control of other software. This may also include file transfers, I/O on databases and the file system or the computation of more complex tasks.
These jobs may be related in either logical or temporal terms. An example for a logical dependency is that some job may only be started if all its predecessors were finished successfully. A temporal condition is that a given job must be started on workdays only.
The second large use case is the appropriate reaction to events. There are a large number of events including examples like the unsuccessful execution of a job, reaching a given limit of free disk space or the arrival of a certain file. In those cases the system may start some alternative jobs or notify some users my automatically send e-mails.
The architecture of the complete UC4 system has a client-server structure (see figure 1.1). The server maintains a central database that holds all the necessary information of the job automation. The clients in this system are a number of agents that are installed on remote machines. Basically there are two types of agents for either direct communication with the operating system or the automated control of an application which runs on the remote machine. If a task should be started, the server extracts the necessary information from the database, uses it to generate an executable job object and delegates it to the appropriate agent. After the execution the agent notifies the server and provides information about the execution.

UC4 Objects

The most basic object in the Automation Engine is the *job*. There are two types of jobs:

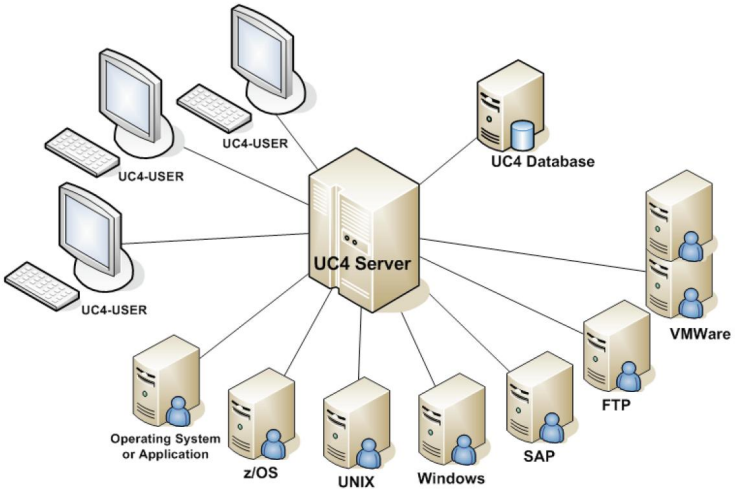

Figure 1.1: This figure shows an example of a UC4 system.

1. Operating system jobs

2. Application jobs

The first interacts directly with the operating system of the machine where the assigned agent is situated, which is typically Windows or UNIX. The latter only communicates with a certain application. The most commonly used application job type is the one for the automation of SAP processes.
There are three parameters that are mandatory to enable the execution of a job:

1. Host: This parameter specifies the machine or the agent where the job is executed. Note that it is also possible to specify a group of agents with an eligible scheduling strategy at this point. This possibility is described later in this section.

2. Login: Without login information it is impossible to access the machine or the application. Login information is managed in so called *login objects* that must be assigned to the job. If given wrong login information or login information for a not sufficiently allowed account the job also cannot be started.

3. Commands or application calls: These define the task itself.

After these parameters are provided and saved the job can be executed. During the execution the user interface provides a panel for monitoring. Here the status of all running tasks is shown. After the execution the system also provides a job report which can be searched for interesting information with different user definable filters.
It is also possible to access the job reports and execution statistics of a number of earlier executions. This number is also user definable.

For the combination of a number of jobs the automation engine provides the *job plan* object. The contained jobs of a job plan can basically be connected by two types of relations:

1. Result-based: The most commonly used is the precedence relation, where a job can start if another job or a number of jobs are finished. This is the simplest possible relation, but the system provides the possibility to define user-specific relations as well.

2. Time-based: Time-based relations can either be date or time dependent. Date dependencies can be managed very efficiently in the system by using so called *calendars*. With them it is possible to define release and completion dates for every job in the job plan.

To define the start time of a single job or a job plan it is either possible to define a *schedule* or to define the *execute periodically* option. Schedules provide the possibility to define the daily to-do list of a system. To express more complex date dependencies it is also possible to use calendar objects like in the job plan dependencies. Since the schedule object was designed to automate the daily operations of a system the minimum period is one day.
Nevertheless the system provides the possibility to realize any other periodic behavior with periodical execution options that can be defined for every job or job plan. There are three possibilities to define the period:

1. Definition of a frequency. The lower limit is an execution every minute.

2. Definition of a gap, which is a time difference to the last execution.

3. Definition of a start time, as it is possible for the schedule object.

Another possibility to start the execution of a job is the usage of *UC4 script*. UC4 provides a scripting language whose scripts can be attached to objects like jobs and job plans and alter their behavior significantly. A typical example for the usage of scripts is reading and writing variable objects. These objects can hold arbitrary variable values that can be used to make the automation more flexible. Furthermore it is possible to set any object attribute dynamically. But it is also possible to start other UC4 objects like jobs, job plans and schedules.
This leads to the last possibility to start the execution of an object: the *event* object. Events are defined as a set of conditions. If all of these conditions hold, the attached script is executed. This script in turn may initialize the execution of other objects.
Basically there exist three different event types:

1. Time Event: A time event is very similar to the schedule object. The user defines a period after whose completion the event is triggered. In contrast to the schedule object the shortest possible period is one minute instead of one day.

2. File System Event: This event class enables the user to define a number of conditions for the underlying file system. A typical use case is the Filewatcher. It may check for the existence of a certain file which has to be transferred from another location and then invokes a number of jobs that need this file for their execution. Another typical example is the monitoring of free disk space.

3. Database Event: Similar to the file system event this event class monitors a database. The conditions can be formulated with the results of SQL queries, variables and static values.

Note that all three classes may also be used within a job plan. Usually this option is realized for the latter two.
After this examination of jobs and temporal relations between them, the existing features for scheduling and workload balancing are presented. The central concept of workload balancing is the *UC4 resource*. This is an integer value that the user may define for every agent and for every job. This value can be chosen by the user and does not necessarily reflect the real performance of an agent or the real workload of a job. Note that jobs can only get executed on agents that provide at least the necessary resources.
The other important concept is the *agent group*. The user has the possibility to arbitrarily group agents with the same operating system. This enables two use cases. If a job has to be executed on a given set of agents, these agents can be grouped and the execution mode *all* can be assigned to this group. This makes the time consuming task of generating a job for every agent superfluous. The more interesting case from the scheduling perspective is if a job can be executed from an arbitrary agent of the group. In that case the system provides four possible execution modes:

1. Any: Execute the job on an agent chosen randomly.
2. First: The job is executed on the first active agent that is found in the table.
3. Next: This option basically realizes a round robin strategy.
4. Load Dependent: This strategy makes use of the UC4 resources and executes the job on the agent that possesses the highest amount of free resources.

There are a number of other objects that are available to the user but have no influence for the problem examined in this work. The most important are listed here:

- *Call*: Provides different means to notify users from within the system.
- *Login*: Storage of login data for the different systems.
- *Calendar*: Enables the user to define complex date constraints.
- *Cockpit*: Provides basic information visualization tools.

Desired Results

The desired result of this work is to explore new ways of improvement for the scheduling methods of the UC4 OM. In the actual form these have a number of drawbacks like having an insufficient model of resources and a too simple prioritization. The only control parameters that can be influenced by the user are the assignment of resource consumption and a priority value and the choice between simple scheduling policies like random or round robin.
There is no possibility for automated planning or to simulate processes for a given period of time. Furthermore the system lacks the possibility to optimize such processes. There is no way

to define any performance measures for solutions or policies.
These problems should be tackled with this project. In the end the we desire a software library with the following capabilities:

- Creation of schedules for a given period of time. A schedule in this context is a list of start times for all the tasks that have to be worked off in the considered time period and an assignment to an agent for every task.
- Evaluation of schedules. When a schedule is created it should automatically be evaluated. This evaluation should be flexible and configurable by the user. Different desired features might be the overall execution time, the accuracy of timing of some jobs or the minimization of additional costs that are caused by SLA violations.
- Expandability of the method. The resulting solutions should be as general as possible to allow expansion. This is important because the UC4 OM is developed as a universal solution to automate IT environments of corporations in many different domains. Furthermore the area is constantly changing and new concepts might have to be integrated in the system.

1.3 Methodological Approach

To achieve the goals set in the previous section it is necessary to translate the real world problem into an optimization problem. This formalization makes it possible to examine the problem from an algorithmic point of view. This in turn has a number of advantages that allow an efficient resolution of the task. First, it is possible to determine the true complexity of the problem, which in turn enables us to choose an appropriate solution approach. Furthermore due to the obvious practical applicability scheduling problems have received an enormous amount of research work. Therefore the chances are high to find promising solution approaches that might be enhanced in the best case. Another advantage of having a formalized problem at hand is, that the solution methods may be compared and evaluated with benchmarks that are used by the scientific community. This is also advantageous for the validation and acceptance of the software.
This section summarizes the findings that were gathered during a preliminary study with experts from UC4. The goal of this study was to find a well-defined model for the OM scheduling that allows capturing every aspect of the system. On the other hand the model should of course be kept as simple as possible to remain applicable with respect to runtime.
With that in mind several models were examined and evaluated. In the following subsection the framework for describing scheduling problems and the classic machine scheduling problems are presented. Then we discuss the shortcomings found for these formalizations that lead to the choice of project scheduling formalizations that is first outlined afterwards, but examined in greater detail in the next chapter.

Machine Scheduling Problems

A basic scheduling problem contains machines and jobs and is concerned with the task of allocating the machines in order to process jobs. Additionally there are a number of constraints

that restrict the processing, for example limitations of the machines or forced orderings of the jobs. The desired output is a timetable or schedule that defines at what point in time a machine is allocated for a certain job.
This section is based on the first chapter of Brucker [2] and follows the same conventions of notation and definition. The author defines for a basic scheduling problem m machines M_j, $j = 1, \ldots, m$ and a set of n jobs J_i, $i = 1, \ldots, n$ is given. A job J_i may in general consist of n_i operations $O_{i1}, \ldots, O_{in_i}$. With these operations two crucial pieces of information are associated. For every operation there exists a subset of machines $\mu_{ij} \subseteq \{M_1, \ldots, M_j\}$ where it can be processed. Furthermore the execution is described with a so called process requirement p_{ij}. This quantity describes how many time units the operation takes on machine j.
Note that even on this level there are differences in the definitions between different authors. Pinedo [21, pp.13] for example relies on a notation without the use of operations and uses the concept of routes for jobs instead.
A concept that is shared between many authors of the field to classify different scheduling problems is the so called three-field notation. It is usually denoted as $\alpha|\beta|\gamma$, where α characterizes the machine environment, β specifies job characteristics and γ holds a definition of the objective function.
Now some commonly known machine environments are presented. Basically they can be divided into two groups. First we consider the models that typically work with jobs containing only one single operation. Therefore in the following list of common values for α the notion of jobs is used.

- $\alpha = \emptyset$ means that every job has to be processed on a specified machine. More formally speaking this means that for every set μ_{ij} it holds that $|\mu_{ij}| = 1$. This special machine environment is called *dedicated machines*.

- $\alpha = P$: This environment represents the opposite to the previous, meaning that there are m *identical parallel machines*. That implies that every job may be processed on every machine and that the process requirements do not differ between them. This means that $p_{ij} = p_i$ holds.

- $\alpha = Q$: A further generalization is the use of *uniform parallel machines*. This model enriches the previously mentioned one by the introduction of a speed factor s_j for every machine. With this concept it is possible to calculate a machine-dependent process requirement with $p_{ij} = \frac{p_i}{s_j}$.

- $\alpha = R$: In the next step the notion of machine speed is extended by also depending on the job at hand. This results in an $m \times n$ matrix holding the factors s_{ij}. The actual process requirement is still calculated as $p_{ij} = \frac{p_i}{s_{ij}}$.

Now we examine the group of the so called multi-operation models. As mentioned before, the jobs in these models consist of a number of operations $O_{i1}, \ldots, {}_{in_i}$. Typically there are precedence relations between the operations, which are mostly based on simple causality. Frequently the processing of one operation sets some preconditions for the processing of its successors. In general causal precedence relations are directed acyclic graphs. Some special cases

are mentioned in the definition of the β field.
Another difference to the previously examined models is concerned with the set of machines for the processing of an operation. Whereas first parallel machines were mentioned the classical specifications of the following models work with dedicated machines. So every operation can only be executed on one machine, implying that $|\mu_{ij}| = 1$.

- G: This model is called the *general job shop*. The following models are restricted versions of this general model.

- J: For the *job shop* the precedence relations of the operations of a job are restricted, such that only chains are allowed:
  $O_{i1} \rightarrow O_{i2} \rightarrow \cdots \rightarrow O_{in_i}$.
  The formulation of the classical job shop problem also does not allow machine repetition. Every job might visit every machine only once.

- FJ: A *flexible job shop* generalizes the classical job shop such that the machines are not dedicated anymore. Instead there exist a number of workcenters that group identical machines in parallel. If a job arrives in such a workcenter it may be processed by an arbitrary machine. The route of the jobs through the machine environment is still defined by their sequence of operations [21, p. 15].

- F: Another well-known special case of the general job shop is the *flow shop*. All the jobs have to be processed on every machine and have to follow the same route through the machine environment. This means it holds that the number of operations $n_i = m$ for every job i and also that $\mu_{ij} = \{M_j\}$ for every job i and every machine j.

- FF: As for the job shop, there also exists a generalization called the *flexible flow shop* where the machines are not dedicated anymore. Again instead of single machines every stage can contain a number of identical machines [21, p. 15].

- O: An *open shop* is another generalization of the flow shop. Again every job has to be processed on every machine, but there is no predetermined sequence of machine. The operations of a job have no precedence relations at all.

Additionally to the first letter(s) the α field has a number attached. Normally this number is a positive integer and defines the number of machines for this very problem instance. For the flexible job shop and the flexible flow shop it denotes the number of workcenters instead.
Now we turn to the β field which holds the characteristics of the jobs. Brucker [2, pp. 3] identifies six elements that define the properties of the jobs and their execution:

- β_1: This field defines whether or not it is possible to pause and resume an activity without losing any time. This property is called *preemption* and is signaled with strings like *prmp* [21] or *pmtn* [2].

- β_2: The second field is used to describe the relations between the jobs. These relations usually indicate the sequences for the execution of jobs. Typically these sequences can be expressed by an acyclic directed graph $G = (V, A)$, where V represents the activities

and every arc $(i, j) \in A$ imposes a constraint stating that activity j may only be started if activity i is finished. This rather general case is expressed by the indicator *prec*. More specialized problem instances work on trees or chains of activities.

- β_3: In many practical problems it is necessary to define release dates r_i for activity i. Release dates restrict the start time of an activity such that it may not smaller than the corresponding release date.

- β_4: This field may be used to specify restrictions concerning the processing times. For example the entry $p_i = 1$ indicates that every activity is processable in one time unit. Other examples are $p_i \in \{1, 2\}$ or $d_i = d$.

- β_5: Where release dates restrict the start times of activities, due dates set deadlines for them. A due date d_i states that the corresponding activity must be finished at time d_i.

- β_6: This field indicates the possibility of batch processing. This means that some machines offer the possibility of processing a number of activities simultaneously.

The last component of the three-field representation of scheduling problems is the γ field, which holds the objective function for the instance. Typical objective functions measure the time needed for the processing of all jobs or minimize the lateness of certain jobs. But it is also possible to evaluate the regularity of tasks or absolute punctuality. Since this question does not directly influence the choice of the model it is postponed to a later section, where it is investigated in the context of the chosen model.

Formulation as Machine Scheduling Problem

During the course of the preliminary study including the experts of UC4 we evaluated all of these classical machine scheduling problems on their appropriateness for the problem at hand. Special attention received the parallel machine approach Rm and the flexible job shop environment FFc. So we picked a model from both, single-operation and multi-operation machine environments.
Now we outline the discussed model translations and take a look on their weaknesses, which in the end lead to their rejection.

Parallel Machines

Together with the experts of UC4 it was concluded that the most appropriate single-operation model was the most general parallel machine model. As stated before this model contains m machines in parallel, n jobs with only one operation and an $m \times n$ matrix s_{ij} for the calculation of individual processing times from the given job processing requirements p_i by $p_{ij} = \frac{p_i}{s_{ij}}$.
In this formalization the machines correspond to the UC4 agents installed on the different machines. If a job i cannot be started on a specific agent m this can be expressed by setting a value value for the corresponding s_{ij} that delivers an enormous processing time.
The translation of tasks in the UC4 system is straightforward. Tasks without subtasks are directly

transformed into jobs and jobplans are transformed to jobs with precedence relations (*prec*). Because of the schedule objects and the possibility of defining periodical executions for the tasks it is also necessary to include the concept of release dates (r_i).
At this point we still do not consider the objective functions, which are left unspecified in the γ field. So the first model is defined by

$$Pm|prec, r_i|\gamma \tag{1.1}$$

Flexible Job Shop

As an alternative attempt the model was encoded as a flexible job shop problem. As stated in the previous section this machine environment is a combination of the parallel machine environment and the job shop environment. The difference to the ordinary job shop is that machines are grouped in so called *workcenters*. These workcenters can process the same operations, which is suitable for modeling the agent groups. Again the machines in this model can be mapped to the UC4 agents.
Since job shop problems are formulated for multi-operation jobs, but a direct transformation of UC4 jobplans to jobs is not possible, because the precedence relations between operations are limited to chains. More complex relations, as they are possible in UC4 jobplans, are only expressible on job level. Nevertheless transformations are possible by grouping chains of tasks into jobs and preserving the precedence relations between these groups.
So the β field must contain the entry for precedence relations and because of the periodic executions there might also be some release dates to consider. The third entry *rcrc* (or recirculation) is specific for job shop problems and indicates that a job may visit a machine or work center more than once [21, p.17].
This leads to the alternative formulation

$$FJc|prec, r_i, rcrc|\gamma \tag{1.2}$$

Open Shop

To overcome the deficiencies of the flexible job shop formulation also the open shop formulation was evaluated. Basically all the concepts and also the problem encoding stay unchanged, but relations between operations are arbitrary. This relaxation supports the encoding by making the superficial approach of grouping chains of operations superfluous. Also the β field entry for recirculation is not necessary in an open shop:

$$Om|prec, r_i|\gamma \tag{1.3}$$

Shortcomings

Even the most general machine scheduling formulation, namely the open shop, fails to capture some of the concepts and restrictions appearing in the system. The main weaknesses identified by the experts of UC4 are outlined here:

- *Timing Constraints:* Precedence relations between jobs and operations in machine scheduling problems are causal precedence relations. This notion is not expressive enough for the problem at hand. This fact also cannot be compensated by the use of release and due dates. In detail the following concepts are needed:

  - *General Precedence Relation:* In machine scheduling problems the minimal difference between the start times of two jobs or operations is always equal to the preceding one. So there is no natural way of defining a "release date" for an operation or job that depends on the start or completion time of some preceding activities, but is independent of their processing times.

  - *Maximum Time Constraints:* The second weakness concerning timing constraints is the lack of a possibility to define a maximum time span between the start or completion times of two jobs or operations. The only workaround for this problem is the use of due dates, but as release dates they are not flexible enough, since they only constrain the start times of certain jobs relative to the start time of the problem. There is no natural way to express due dates relative to other jobs.

- *Machines:* Another point is the concept of machines. Since the machine environments scheduling problems are designed to capture scheduling problems that arise in the manufacturing industry, they are not necessarily suitable for the formalization of problems in IT automation. The following points are problematic:

  - *Agents and Machines:* The agent-machine mapping is not a one-to-one relation in general. It is not unusual that several agents are installed on a single machine. This problem cannot be avoided by using a direct computer-machine relation, because the models do not allow multitasking.

  - *Multitasking:* In classical machine environment scheduling problems every machine can only be occupied by one job at a time. This of course is absolutely inappropriate when modeling a computer environment where every machine can handle a multitude of tasks simultaneously.

  - *Resources:* The only resource considered in the models so far is "work". A job or an operation needs some amount of it to be finished and the machines provide some. The provided amount can only be expressed by a single factor for every machine operation pair. This is not practical for IT environments. First of all it might be of interest to model the computational "work" of the computers in the system on a finer-grained level (e.g.: number of CPUs or amount of RAM). Another aspect of interest that cannot naturally be expressed in machine environment scheduling is the possibility that the processing of an operation might allocate resources on two different machines. A typical and of course frequent example is the task of file transfer or data streaming.

Project Scheduling Problems

The results of the preliminary study clearly suggest that a more general model is needed to capture all the aspects of the problem at hand. The two major weaknesses of the too narrow machine model and the lack of flexible time constraints can both be tackled by the use of a project scheduling problem.
The most influential difference between the two families of scheduling problems is the extension of the concepts of machines to the concept of resources. The classical machine scheduling problems assumed that a machine can only do one job at a time. This is certainly not true for computer systems in an IT environment. Also the introduction of batch processes as sometimes used is not flexible enough and rather suited to model working stations on an assembly line.
Resources in some sense loosen the tight coupling of machine and work piece and transform it to the more abstract level of provider and consumer. The consumers are again rather abstract entities that are not jobs and operations any more, but instead are called activities. Every activity might need various resources for its execution and in the *resource-constrained project scheduling problem (RCPSP)* the task is not to push this resource demand over given limits in every point in time, while still optimizing some objective function. The RCPSP is the basic model that is dealt with in this work and is formally introduced in section 2.1.
In [14, p. 321] the authors define the following types of resource environments, which correspond to the α field:

- *PS*: This denotes the basic RCPSP as described above.
- *PS*∞: The infinity symbol signalizes unlimited resource availability. This means that only timing constraints imposed by the relations between the activities have to be respected. It will be shown that this is much more complex when generalizing these relations.
- *PSc*: This environment specifies a project scheduling problem with cumulative resources and is not in the scope of this work.
- *MPS*: The *multi-mode resource-constrained project scheduling problem* introduces the concept of execution modes of activities, which influence their resource demands. The solution of a scheduling problem is not only to choose start times for the activities any more, but also to assign the optimum execution modes. The work at hand focuses on the resolution of these models.

As for the machine scheduling problems there also exists a β field to describe the relations of the activities in the project scheduling problem. Neumann et al. [14, pp. 321] list three possible entries:

- β_1: It holds that $\beta_1 \in \{prec, temp\}$. The first case describes causal precedence relations that are also used in machine scheduling. The latter describe the much more expressive concept of *temporal constraints*. These can be used to model more complex relations like the definition of time windows or enabling another action after 25% of the predecessor is processed.
- β_2: If $\beta_2 = \overline{d}$ the project's overall time is restricted by a deadline.

- β_3: It is possible to describe sequence-dependent changeover times between activities. This is signaled by setting $\beta_3 = s_{ij}$.

It holds the $\beta \subseteq \{\beta_1, \beta_2, \beta_3\}$. Objective functions which are specified in the γ field do not change.

Benchmarking

Another advantage that arises from the strict formalization of the problem is the ability to compare obtained results with outcomes of state of the art algorithms. Since project scheduling problems arise in a number of real world applications, a lot of effort has been invested into the development of benchmarks sets [18]. These can basically be divided into real world problems and artificial instances.
Since this project is concerned with the resolution of scheduling problems for many different and dynamically changing environments, we focused on benchmark sets that are designed to cover a variety of cases. In an early work Kolisch and Sprecher [17] not only presented a benchmark set that they argued is well-balanced, but also introduced an instance generator (ProGen) to produce an arbitrary number of instances with certain characteristics.
These characteristics were identified by the authors and may roughly be divided into three groups:

- Scale parameters: These parameters can be used to control the size of the generated projects. They contain the number of activities, execution modes and different resources.
- Network parameters: For the characterization of the relationship between the activities a number of parameters is given to emulate the complexity. These include parameters to control the number of preceding and succeeding activities and the number of dependencies of the activities.
- Resource availability: A third group of parameters allows manipulating the availability of resources. This includes the availability itself, the average consumption, or the probability that there is a tradeoff between activity duration and resource consumption.

Note that this list is only a short summarization to provide an idea of the ProGen. The program itself is available for download on the website[2] and might be used to generate problem instances tailored for a specific environment. Further information on the use and parameterization can be found in [17] and [18].
The website not only provides the instance generation tool, but also the benchmark set used in this work and many others. Each benchmark set comes with a file that contains the parameter settings used for the generation. Furthermore there are benchmark results available that show the best known solutions to the problems at hand. New results can be submitted by sending an e-mail in a standardized format.
Furthermore there is a number of papers on the generation tool, the instances itself and commenting on the development of resolution methods, the most recent being from Kolisch and Hartmann [16].

[2] http://129.187.106.231/psplib/

CHAPTER 2

Methodology

2.1 Problem Formalization

In this section we present the formal problems which were identified in section 1.3 to fit the given problem statement. This ection starts with the most basic model that already incorporates resources and causal precedence relations between activities. This model is extended with the possibility to process activities in certain predefined execution modes that influence resource consumption and processing times. Additionally these execution modes can be linked with another type of resource that is not available with a certain amount in every point in time, but limited for the whole project. In a last step the model is extended for generalized time lags. These allow the formalization of complex temporal time constraints that are useful for the problem at hand.
This section closes with an examination of alternative objective functions and the problem translation. The latter shows how to encode the concepts of the automation engine in formalized problem instances.
For the remainder of this chapter the notational guidelines and conceptual definitions of [14] will be followed.

The Resource-Constrained Project Scheduling Problem

The resource-constrained project scheduling problem (RCPSP or PS|prec|f) provides a basic formalization for typical aspects of practical project scheduling. Let us now take a thorough look at the components that form an instance of the problem.
First a number of real activities $1, ..., n$ is given. All of these activities have to be executed and once they are started, they cannot be stopped or paused. In other words, preemption is not allowed. Furthermore two dummy activities 0 and $n + 1$ are added to represent the project's start and completion. Put together they form the set of activities $V = \{0, \ldots, n + 1\}$.
Every activity i has a processing time p_i assigned to it. For real activities it holds that $p_i \in \mathbb{N}$. The dummy activities both have a processing time that equals zero, $p_0 = p_{n+1} = 0$.

A solution of a RCPSP assigns a start time $S_i \in \mathbb{Z}_{\geq 0}$ to every activity in V. Such an assignment $S = (S_i)_{i \in V}$ is called *schedule*. It is obvious that the start time of the dummy node that represents the project's start is zero, i.e. $S_0 = 0$. The time assigned to the project's end is the project's duration which is also called the *makespan*. In project scheduling problems there usually exists a number of constraints concerning the execution of activities. Typically some activities have to be finished before a given activity i is allowed to start. This precedence relation is modeled by defining sets of successors $\mathrm{succ}(i)$ and predecessors $\mathrm{pred}(i)$ for every activity i. Naturally it holds that $\mathrm{pred}(0) = \emptyset$ and $\mathrm{succ}(n+1) = \emptyset$. We can also formulate the constraints imposed by these precedence relations using the assigned start times of the schedules. Consider activity i with a set of successors $\mathrm{succ}(i)$. Then the following condition has to hold:

$$S_j - S_i \geq p_i,\ j \in \mathrm{succ}(i) \tag{2.1}$$

In addition a set of renewable resources $\mathcal{R}$ is given. These resources are required for executing the activities of the project. They are called renewable because a fixed amount of every resource is given in every point in time. Typical examples of such resources are staff members with a specific skill or the computational power provided by host stations. The capacity of every resource $k \in \mathcal{R}$ is given by $R_k \in \mathbb{N}$. Furthermore the resource consumption of an activity i is given by $r_{ik} \in \mathbb{Z}_{\geq 0}$. To integrate these resource constraints into the problem definition, we consider a given solution or schedule $S = (S_i)_{i \in V}$, respectively. The set of active activities for every point in time t can be defined as follows:

$$\mathcal{A}(S,t) = \{i \in V | S_i \leq t < S_i + p_i\},\ t \geq 0 \tag{2.2}$$

With this definition it is possible to formulate the resource consumption of a given schedule over time:

$$r_k(S,t) = \sum_{i \in \mathcal{A}(S,t)} r_{ik},\ k \in \mathcal{R}, t \geq 0 \tag{2.3}$$

This function is also referred to as *resource profile* of *resource utilization function*.
For the complete formalization of the problem also an upper bound for the project's duration is needed. An intuitive bound for this planning horizon is the sum of all the processing times p_i:

$$\overline{vd} = \sum_{i=1}^{n} p_i \tag{2.4}$$

With this upper bound the planning horizon can be limited and the resource constraints can be formulated:

$$r_k(S,t) \leq R_k,\ k \in \mathcal{R}, 0 \leq t \leq \overline{d} \tag{2.5}$$

Finally the RCPSP can be stated as the following mathematical model:

$$\begin{aligned}
\text{Min } S_{n+1}, \quad & \text{s. t.} \\
r_k(S,t) \leq R_k \quad & k \in \mathcal{R}, 0 \leq t \leq \overline{d} \\
S_j - S_i \geq p_i \quad & i \in V, j \in \mathrm{succ}(i) \\
S_i \geq 0 \quad & i \in V \\
S_0 = 0 \quad &
\end{aligned}$$

Instances of the RCPSP can easily be described graphically. A natural way to do so is with an activity-on-node network or AoN network $N = (V, A)$.
In an AoN network every activity is assigned to a node. So given n real activities we obtain the node set $V = \{0, 1, \ldots, n+1\}$ after inserting the two dummy nodes that represent the project's start and end, respectively. There also exists a number of arcs A between the nodes that are used to express the causal precedence relations between activities. So $A = \{(i, j)|i \in V, j \in \mathrm{succ}(i)\}$. Note that AoN networks representing a feasible problem instance do not contain any cycles. Since the precedence relation is causal, which means that the completion of a predecessor is a precondition for the start of the successor, activitys contained in a cyclic structure can never be scheduled.
Additionally to a label that shows the number of the activity every node has at least two other values assigned to it: the activity's execution time and at least one value that denotes the activity's resource requirement. Generally a number of these resources is given and every node holds a vector of resource requirements. Furthermore a complete RCPSP contains a vector that represents the available resource units for every point in time.
Figure 2.1 depicts an exemplary AoN network. It contains six real activities, the two dummy

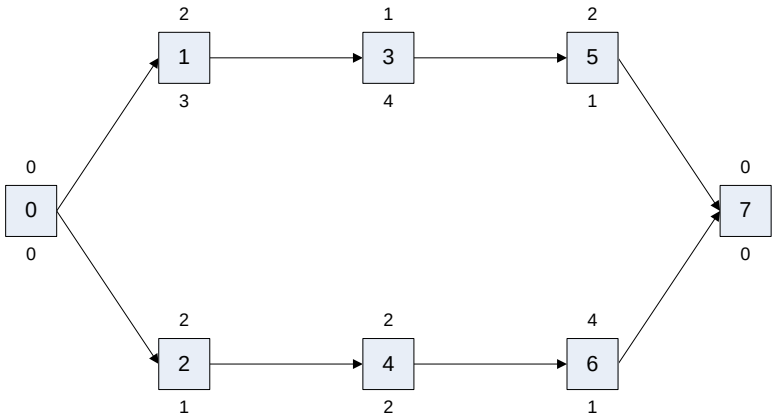

Figure 2.1: This figure depicts a basic RCPSP instance.

nodes and eight arcs to model the precedence relations. Note that activities without a real predecessor get 0 as their only predecessor, whereas activities without a successor are connected to $n + 1$. Of course an insertion of other arcs is possible but would be superfluous.
In addition to this so called temporal network two more values are assigned to each node. On top of it the required processing time is given. This processing time also determines the minimal time difference between the starting points of two adjacent activities in a schedule for the problem instance.
The value under every node specifies the resource consumption of each activity in every time unit between its start time S_i and $S_i + p_i$. In this case only one renewable resource with a capacity of 4 is given.
With the temporal network at hand one can easily compute the possible time windows for the

activities. This includes the earliest start and earliest completion time ES_i and EC_i where

$$\mathrm{EC}_i = \mathrm{ES}_i + p_i,\ i \in V \tag{2.6}$$

as well as the same concepts for the latest start and completion time LS_i and LC_i with

$$\mathrm{LC}_i = \mathrm{LS}_i + p_i,\ i \in V \tag{2.7}$$

The algorithm for the calculation of this values is straightforward (see algorithm 2.1). Since the temporal network of an RCPSP instance does not contain any cyclic structures it is easy to find a topological sorting. This is a sorting of activities that satisfies the condition that for every activity every predecessor in the temporal network is also a predecessor in the sorting. In order to find the earliest starting and completion times it is only necessary to perform a simple forward recursion on such a topologically ordered list of activities.

input : temporal network N, topologically sorted list of activities l

1 $\mathrm{EC}_0 = 0, \mathrm{EC}_0 = 0$;
2 **for** $k \leftarrow 1$ **to** $n + 1$ **do**
3 $\quad j \leftarrow l[k]$;
4 $\quad \mathrm{ES}_j = \max\{\mathrm{EC}_i \,|i \in \mathrm{pred}(j)$;
5 $\quad \mathrm{EC}_j = \mathrm{ES}_j + p_j$;
6 **end**

Algorithm 2.1: Calculation of earliest starting and completion times for a RCPCP instance.

Analogously a backward recursion is performed to compute the latest start and completion time for every activity (see algorithm 2.2). An additional parameter for this algorithm is a planning horizon limit. In practice a project often has a deadline that determines when it has to be finished. If such a deadline is given it can be used to compute the time windows. Otherwise the upper bound of the project's makespan given in equation 2.4 can be utilized.
This upper bound of course is not tight, but it has the advantage that there must exist a time and resource feasible solution within it. For a tighter upper bound the earliest start time of the project's end activity, calculated with the previous algorithm can be used. Note that it is not guaranteed that there exists a solution within this bound, because due to resource constraints an exceeding might be inevitable. It can be seen that the use of the tight upper bound will clearly lead to at least one critical path of precedence relations within the network. A critical path can be recognized by the fact that for every node i on the path it holds that $\mathrm{ES}_i = \mathrm{LS}_i$. A critical activity cannot be postponed without violating the project's deadline or in general increase the project's duration.
So the strict upper bound does provide time windows that can be used to provide precedence feasible solutions. Note again that this does not necessarily mean that a resource feasible schedule can be generated on this basis.

In table 2.1 the time windows for the example in figure 2.1 are shown. Note that for the calculations of the latest schedule limits the strict bound was used. It can be seen that with this

input : temporal network N, topological sorted list of activities l, time bound B

1 $\mathrm{LS}_{n+1} = B, \mathrm{LS}_{n+1} = B$;
2 **for** $k \leftarrow n$ **to** 0 **do**
3 $\quad j \leftarrow l[k]$;
4 $\quad \mathrm{LC}_j = \max\{\mathrm{LS}_i \,|i \in \mathrm{succ}(j)\}$;
5 $\quad \mathrm{LS}_j = \mathrm{LC}_j - p_j$;
6 **end**

Algorithm 2.2: Calculation of latest starting and completion times for a RCPCP instance.

bound the critical path $0 \rightarrow 2 \rightarrow 4 \rightarrow 6 \rightarrow 7$ can be identified.

Aside to the earliest and latest start time of every activity, table 2.1 includes two other values. First there is the total float TF for every activity $i \in V$. This value simply expresses how many time units an activity $i \in V$ can be postponed from its earliest start time with still meeting the deadline.

$$\mathrm{TF}_i = \mathrm{LS}_i - \mathrm{ES}_i = \mathrm{LC}_i - \mathrm{EC}_i, \; i \in V \tag{2.8}$$

Furthermore the interval $[\mathrm{LS}_i, \mathrm{EC}_i[$ is calculated for every activity $i \in V$. This interval is also called the *base time interval* or *unavoidable time interval* for activity i [14, p. 13]. It is clear that if the project is executed within the time limit used to calculate the time windows, activity i will be in progress in this time interval. Note also that for activities i that are non-critical, it holds that $[\mathrm{LS}_i, \mathrm{EC}_i[= \emptyset$. If that is not the case the activity is critical or near-critical. For a near-critical activity $i \in V$ it holds that $0 < \mathrm{TF}_i < p_i$.
Note that for a schedule of the example instance that meets a deadline of 8, which is the earliest start time of the project's end activity, it is necessary that the activities on the critical path are executed consecutively without any interruptions. A look on figure 2.2 reveals that such a schedule cannot be found. The figure shows a schedule and its resulting resource consumption profile over the time.

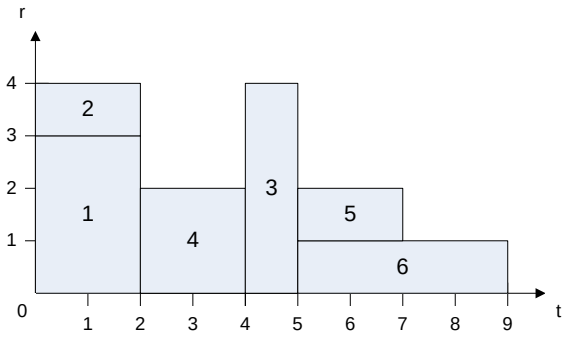

Figure 2.2: In this figure a possible solution schedule for the RCPSP instance in figure 2.1 is shown.

From that figure we can easily deduce that no schedule with duration of 8 can be found. The reason for this is activity 3, which utilizes the entire capacity of the resource for one time unit and has to interrupt the execution of the activities on the critical path for at least one time unit. Therefore a solution of the instance must have at least a makespan of 9, which is the makespan of the given solution. Therefore this solution is optimal.

Although the RCPSP is the simplest model that is examined in this work, it has been reported

i	0	1	2	3	4	5	6	7
p_i	0	2	2	1	2	2	4	0
ES_i	0	0	0	2	2	3	4	8
LS_i	0	3	0	5	2	6	4	8
TF_i	0	3	0	3	0	3	0	0
$[\mathrm{LS}_i, \mathrm{EC}_i[$	$\emptyset$	$\emptyset$	$[0, 2[$	$\emptyset$	$[2, 4[$	$\emptyset$	$[4, 8[$	$\emptyset$

Table 2.1: Time window calculations for the RCPSP depicted in figure 2.1.

to be NP-hard [7, p. 10].

The Multi-Mode Resource-Constrained Project Scheduling Problem

An extensions of the RCPSP is the multi-mode resource-constrained project scheduling problem (MRCPSP). For this variation the model is augmented with two further concepts: *Non-renewable resources* denote resources whose availability is not fully restored for every point in time. There exists only a certain amount for of these resources for the whole project. Like for renewable resources every activity may consume a given amount of them.

For the RCPSP the concept of non-renewable resources does not make sense, because they could only be used to calculate the feasibility of the project in principal. But in the MRCPSP there is not only one way to work off an activity, but a number of *execution modes* for every one of them. Both the process time as well as the resource consumption vector of an activity depend on its chosen execution mode.

Again we will only consider the MRCPSP without preemption. This means, once an activity is started in a certain execution mode it has to be finished without any interruptions. It is also not possible to change the execution mode during the process.

This extension of the model is useful for many real-world tasks. Typically project scheduling is not only concerned with the assignment of start times for the activities that have to be worked off, but also decisions can be made how to execute an activity. With this model different assignments of activities to different types of machines or to workers with different levels of expertise can be considered. Additionally the concept of non-renewable resources provides a natural way to take budget restrictions or consumed matter like a chemical fluid into account.

We will now extend the mathematical formulation of the RCPSP to include the new concepts: First let $\mathcal{M}_i$ be the set of possible execution modes for an activity i. The variable $x_{im_i} \in \{0, 1\},\ i \in V, m_i \in \mathcal{M}_i)$ indicates that activity i is executed in execution mode m_i. Naturally

every activity can only be executed in a single execution mode, thus the following must hold:

$$\sum_{m_i \in \mathcal{M}_i} x_{im_i} = 1,\ i \in V \tag{2.9}$$

For the MRCPSP there must also be a differentiation between the two types of resources. Let $\mathcal{R}^\upsilon$ be the set of non-renewable and $\mathcal{R}^\rho$ the set of renewable resources. Both types of resources have a given capacity $R_k \in \mathbb{N},\ k \in \mathcal{R}^\upsilon \cup \mathcal{R}^\rho$.
We can now consider a subproblem of the MRCPSP which is to find a feasible mode assignment for all activities. A complete mode assignment is a vector $x = (x_{im_i})_{i \in V, m_i \in \mathcal{M}_i}$ that satisfies the condition of providing a unique execution mode for every activity (equation 2.9) and is furthermore resource feasible with respect to the non-renewable resource limits. Let us define the resource consumption of activity i for resource k when executed in mode m_i as $r_{ikm_i} \in \mathbb{Z}_{\geq 0}$. Then the consumption of a non-renewable resource k by an acitivty i can be calculated as

$$r^\upsilon_{ik}(x) = \sum_{m_i \in \mathcal{M}_i} r_{ikm_i} x_{im_i},\ i \in V, k \in R^\upsilon$$

The overall consumption of a non-renewable resource k by the complete execution mode assignment x is given by

$$r^\upsilon_k(x) = \sum_{i \in V} r^\upsilon_{ik}(x),\ k \in R^\upsilon$$

A mode assignment is called resource-feasible, if it satisfies the constraints of the non-renewable resources:

$$r^\upsilon_k(x) \leq R_k,\ k \in R^\upsilon \tag{2.10}$$

Aside from the resource consumptions also the processing times can depend on the chosen execution mode. Hence the processing time of an activity $i \in V$, if executed in mode $m_i \in \mathcal{M}_i$ is denoted by p_{im_i}. Given a mode assignment, the processing time of an activity is

$$p_i(x) = \sum_{m_i \in \mathcal{M}_i} p_{im_i} x_{im_i},\ i \in V$$

With that in mind we can slightly rephrase the time constraints of the RCPSP to

$$S_j - S_i \geq p_i(x),\ j \in \mathrm{succ}(i) \tag{2.11}$$

The same can be done for the resource requirement function of an activity $i \in V$ given a mode assignment x

$$r^\rho_{ik}(x) = \sum_{m_i \in \mathcal{M}_i} r_{ikm_i} x_{im_i}$$

and the notation of the active set of activities at time t given a schedule S and a mode assignment x

$$\mathcal{A}(S, t, x) = \{i \in V | S_i \leq t < p_i(x)\}$$

Again with this function it is possible to describe the resource consumption profile of schedule S with respect to mode assignment x:

$$r_i^\rho(S,t,x) = \sum_{i \in \mathcal{A}(S,t,x)} r_{ik}^\rho(x),\ k \in R^\rho, t \geq 0$$

As before for the RCPSP an upper bound or planning horizon is needed for the formulation of the renewable resource constraints. Again the accumulation of the processing times provides a rather loose bound. The only difference is that in the case of the MRCPSP the maximum processing time of all execution modes is used.

$$\overline{d} = \sum_{i \in V} \max_{m_i \in \mathcal{M}_i} p_{im_i}$$

So finally all necessary concepts for stating the renewable resource constraints are at hand.

$$r_k^\rho(S,t,x) \leq R_i,\ k \in \mathcal{R}^\rho, 0 \leq t \leq \overline{d} \tag{2.12}$$

The complete MRCPSP can be formalized with the following model:

$$\begin{aligned}
\text{Min } S_{n+1}, & \qquad \text{s. t.} \\
\sum_{m_i \in \mathcal{M}_i} x_{im_i} = 1 & \qquad i \in V \\
r_k^\upsilon(x) \leq R_k & \qquad k \in \mathcal{R}^\upsilon \\
r_k^\rho(S,t,x) \leq R_k & \quad k \in \mathcal{R}^\rho, 0 \leq t \leq \overline{d} \\
S_j - S_i \geq p_i(x) & \quad i \in V, j \in \text{succ}(i) \\
S_i \geq 0 & \qquad i \in V \\
S_0 = 0 & \\
x_{im_i} \in \{0,1\} & \quad i \in V, m_i \in \mathcal{M}_i
\end{aligned}$$

Obviously the MRCPSP is a real extension of the RCPSP. In fact a MRCPSP without any non-renewable resources and with only one mode for every activity is a RCPSP. Therefore the MRCPSP is NP-hard.

General Time Constraints

The use of generalized time constraints is an important extension of both, the RCPSP and the MRCPSP. So far time constraints were solely based on causal precedence relations: an activity could only be started, if all its predecessors are worked off. In this section we consider a generalized concept of time constraints which is also referred to as time lags.
Consider a RCPSP as defined earlier. At this point another set of parameters is introduced to the model. Every arc between two activities i and j is labeled with a *minimum time lag* $d_{ij}^{\min} \in \mathbb{Z}_{\geq 0}$. This imposes the following restriction on the two start times S_i and S_j:

$$S_j - S_i \geq d_{ij}^{\min} \tag{2.13}$$

It is easy to see that this generalizes the time constraints used in the previous sections. If for every precedence relation it holds that $d_{ij}^{\min} = p_i$ we obtain a classical RCPSP instance with simple precedence constraints.
With this additional parameters it is now possible to let activities overlap $d_{ij}^{\min} < p_i$ and formalize statements like "acitvity j may start after 50% of activity i is worked off" with $d_{ij}^{\min} = \frac{1}{2}p_i$. It is also possible to enforce that activity j may not be started before activity i by setting $d_{ij}^{\min} = 0$.
Another important concept of project scheduling that can be modeled in a natural way are release dates. Release dates specify the time a given activity is available for processing. To incorporate a release date for activity i of 10 time units in the project, one only has to introduce a minimum time lag between the dummy node that represents the project's start and the respective activity, e.g. $d_{0i}^{\min} = 10$.
Initial activities i with no real activities as predecessors and no given release dates receive minimum time lags of $d_{0i}^{\min} = 0$. So they can be started right at the beginning of the project. Furthermore we assume that terminal activities - these are activities without real successors - have a minimum time lag of at least their processing time to the project's end activity $d_{i,n+1}^{\min} > p_i$. If that condition is not met the affected activities will not be completed at the project's end.
Another condition that has to be taken care of is concerned with non-terminal activities i with a rather long processing time p_i and a successor j with a shorter processing time $d_{ij}^{\min} + p_j < p_i$. In that case it is necessary to introduce an additional minimum time lag between $d_{i,n+1}^{\min} = p_i$ to assure that the activity is completed at the project's end.

Now a different type of time lag is considered: a *maximum time lag* $d_{ij}^{\max}$ between two activities i and j states that activity j must be begun at most $d_{ij}^{\max}$ time units after the start of activity i.

$$S_j - S_i \leq d_{ij}^{\max} \tag{2.14}$$

Together with minimum time lags maximum time lags allow the definition of time windows. To state that an activity i must be started earliest at time $t_1 \in \mathbb{Z}_{\geq 0}$ and latest at time $t_2 \in \mathbb{Z}_{\geq 0}$ two time lags $d_{0i}^{\min} = t_1$ and $d_{0i}^{\max} = t_2$ must be introduced. The application of this idea to the completion of an activity is trivial. It is only necessary to subtract the processing time p_i from the points in time. Time windows can also be specified between two real activities.
It is also possible to enforce the synchronous start of activities by defining time windows of length 0 between them. In general if m activities $i_1, \ldots, i_m$ have to be started on the exact same point in time m time lags are needed to synchronize them. First $m-1$ minimum time lags of length 0 are introduced as a chain between the first $m-1$ activities, then a single maximum time lag $d_{i_m,i_1}^{\max}$ to force $S_1 = S_2 = \cdots = S_m$.
Another important concept used in project scheduling are deadlines. Similar to release dates, which can easily be modeled with minimum time lags between the concerned activity and the project's start activity, deadlines can be expressed with maximum time lags between the concerned activity and the project's start. Assume that the real activity i must be finished at a given point in time $\overline{d} \in \mathbb{N}$. This can be assured by introducing the time lag $d_{0,i}^{\max} = \overline{d} - p_i$. Like with all the other concepts presented, the transformation between start-start or completion-completion constraints can easily be done to every desired configuration.

Now the modifications of the model will be implemented in the AoN networks presented in section previously. Naturally the set of activities V stays the same, but since the relation between these nodes is not a simple precedence relation anymore, the arcs $\langle i, j\rangle \in A$ have to be enriched with additional information. Therefore the weight $\delta_{ij} \in \mathbb{Z}_0$ is introduced. For a minimum time lag $d_{ij}^{\min}$, which refers to the arc $\langle i, j\rangle \in A$ it is defined as $\delta_{ij} = d_{ij}^{\min}$.
A maximum time lag $d_{ij}^{\max}$ is expressed with a backward arc $\langle j, i\rangle$ in the network. Also the weight of such an arc is defined as $\delta_{ji} = -d_{ij}^{\max}$ and is therefore from the set $\mathbb{Z}_0$.
Using these newly introduced weight factors the temporal constraints for minimum and maximum time lags in equations 2.13 and 2.14 can be unified in the new constraint set

$$S_j - S_i \geq \delta_{ij},\ \langle i, j\rangle \in A \tag{2.15}$$

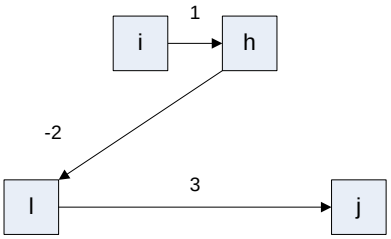

Figure 2.3: This figure shows a path of two minimum and one maximum time lags. The placement of the nodes should support the idea of induced time lags.

Now let us consider paths in AoN networks. Figure 2.3 shows a path including four nodes. The length of this path sums up to two and therefore induces a minimum time lag $d_{ij}^{\min}$ from start node i to source node j, which is also indicated with the dashed arc. In general the length of a path can also be negative. In that case a maximum time lag $d_{ij}^{\max}$ is induced. If the length of the path sums up to 0 either a maximum or a minimum time lag is induced.
With the use of maximum time lags also cycle structures may be introduced to the AoN networks. These are in fact paths having the same start and source node. Note that cycle structures with a path length $l > 0$ are unfeasible. Given equation 2.13 considering the induction of time lags the constraint $S_i \geq S_i + l$ is generated for every activity i in the cycle. Since these constraints state that activities have to be started some time after they have been started, time feasibility is unachievable and therefore problem instance containing a cycle structure with positive path length are not solvable.
There are solution approaches that treat cycle structures as separate subprojects. After a cycle structure is identified it is extended by an artificial start and source node and planned separately. After a valid schedule for the subproblem is found the cycle structure is treated as a single node in the original problem. Of course the time lags for the adjacent nodes have to be adapted and also the resource consumption profile of the subproject has to be added to the overall project. A more thorough treatise on these *decomposition methods* is given in section 2.2. In general the advantage of these approaches is the lowered complexity. On the other hand the separate

treatment of the subprojects also lowers the flexibility.

Another important concept that comes into play with the introduction of maximum time lags is the temporal scheduling network N^+. This network is just an extension of the AoN network used before by an additional arc $\langle n+1, 0\rangle$ which represents a maximum time lag between the start and source nodes of the project. In general the weight of this arc is either given by a predefined deadline or the earliest start time of the project's end node ES_{n+1}. The distance d_{ij} in such a temporal scheduling network is the longest path between two nodes i and j. Since N^+ is a cyclic structure that contains every node, there also exists a path between every node. The calculation of d_{ij} can be done in polynomial time with the well-known Flyod-Warshall algorithm [5] and delivers a number of useful information.

1. Implicit time lags between the activities are made explicit. If $d_{ij} > 0$ there exists a minimum time lag between i and j. If $d_{ij} < 0$ there exists a maximum time lag with value $-d_{ij}$ between i and j.

2. The existence of cycle structures with positive cycle length is easily detected by checking if $d_{ii} = 0$ holds for every activity $i \in V$.

3. All the earliest start times ES_i and therefore the earliest schedule

$$ES = (ES_0, ES_1, ..., ES_{n+1}) \tag{2.16}$$

   is given with $ES_i = d_{0i}$.

4. The same holds for all the latest start times LS_i and the latest schedule

$$LS = (LS_0, LS_1, ..., LS_{n+1}) \tag{2.17}$$

   with $LS_i = -d_{i0}$.

Again these values allow the calculation of the useful concepts of completion times, floats and base time intervals that were already introduced earlier.
Figure 2.4 depicts a temporal scheduling network with multiple cycle structures and the added backward arc $\langle n+1, 0\rangle$ with $\delta_{n+1,0} = -ES_{n+1}$. Table 2.2 shows the corresponding distance matrix, where the earliest schedule $ES = (0, 0, 1, 0, 3, 8, 10)$ and the latest schedule $LS = (0, 4, 6, 1, 5, 8, 10)$ can easily be read off. With this information we can calculate the float and identify the critical path $\langle 0, 5, 6\rangle$

Now the concept of generalized time constraints is finally also applied to the MRCPSP. In the literature the resulting problem is referred to as either MRCPSP/max or MRCPSP-GPR, where GPR stands for general precedence relations. In the three field notation it is MPS|temp|$C_{\max}$. Basically all the needed concepts are already defined. Execution modes, non-renewable resources and the assignment problem were introduced previously and in this chapter the generalized time lag was presented. But an important new aspect is the dependency of the time lags

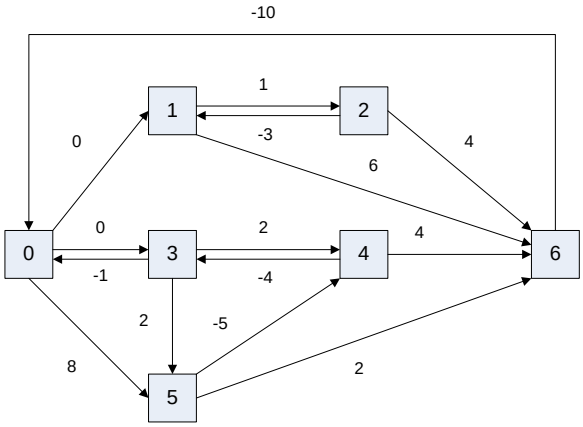

Figure 2.4: This figure depicts the temporal scheduling network N^+ of a RCPSP/max instance.

i /j	0	1	2	3	4	5	6
0	0	0	1	0	3	8	10
1	-4	0	1	-4	-1	4	6
2	-6	-3	0	-6	-3	2	4
3	-1	-1	0	0	2	2	6
4	-5	-5	-4	-4	0	3	4
5	-8	-8	-7	-8	-5	0	2
6	-10	-10	-9	-10	-7	-2	0

Table 2.2: Distance matrix calculation for the example depicted in figure 2.1.

from the execution modes of the corresponding activities. Formally defined the weight function is not a simple arc to value mapping like when dealing with the PS|temp|$C_{\max}$, but also a function of the mode assignment vector x:

$$\delta_{ij}(x) = \sum_{m_i \in \mathcal{M}_i} \sum_{m_j \in \mathcal{M}_|} x_{im_i} x_{jm_j} \delta_{im_i jm_j} \tag{2.18}$$

Consider the example instance depicted in figure 2.5 which is taken from Barrios et al. [1] and consists of six real activities and the two dummy nodes. Each of the real activities has three possible execution modes to choose from. On top of each real activity node there is a vector with three elements that show the processing time for each execution mode. The matrix beneath each real activity node determines the resource consumption. The first column of the matrix shows the consumption of the single non-renewable resource. This non-renewable resource is limited with 10 units in this example. The second column shows the consumption of the renewable resource. Both consumption vectors also contain as many elements as there are execution modes. The weight function maps a $\mathcal{M}_i \times \mathcal{M}_j$ function to every arc of the network. Between real activities these are all 3×3 matrices in this example, whereas the arcs from the start node have only one row and the arcs to the source node have only one column since there is no choice of

execution modes for the dummy nodes.

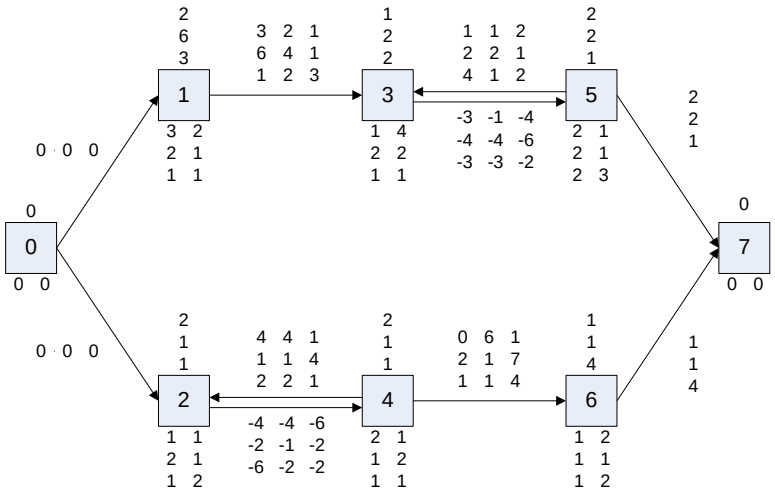

Figure 2.5: This figure shows a basic MRCPSP/max instance.

The determination of the execution modes has an additional aspect in contrast to the simple MRCPSP. Not only the consumptions of the non-renewable resources have to be taken into account, but there is also the topic of time feasibility. Consider for example the mode assignment vector $x = \langle 1, 2, 2, 3, 3, 1, 3, 1 \rangle$. Once a mode vector is chosen the MRCPSP/max is reduced to a RCPSP/max. The corresponding AoN network is depicted in figure 2.6.

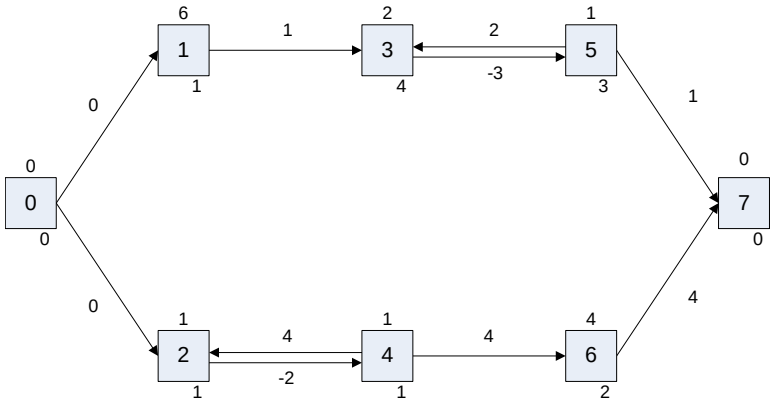

Figure 2.6: This figure depicts the RCPSP/max instance that is the result of assigning the mode vector $\langle 1, 2, 2, 3, 3, 1, 3, 1 \rangle$ to the MRCPSP/max instance of figure 2.5.

The non-renewable resource consumption of the mode assignment is $0+1+1+1+1+1+2 = 7$ which is greater than 10 and therefore resource-feasible. But because of the cycle structure $(2, 4)$ that has a path positive sum of 2 the mode assignment is not-time feasible.
The problem of testing if there exists a time-feasible mode assignment for an MPS|temp|f (where f is an arbitrary objective function) has been shown to be NP-complete [14, p. 151]. Since verification of time feasibility can be done in polynomial time with a longest path calculation of the AoN network the membership in NP is given. The proof of NP-hardness is done by giving a polynomial transformation of the problem to the partially ordered knapsack problem [14, pp. 151].

General Objective Functions

In this section the most commonly used objective functions for project scheduling problems in general are presented. In the previous sections the makespan $C_{\max}$ is used as the default objective function. This is also the standard case in the literature and therefore for all the benchmark instances for the problem at hand. But since one of the requirements given in the problem statement is to provide a certain flexibility regarding the objective function used, a closer look on this topic is provided here.
The problem that is examined here can be characterized as PS|temp, $\overline{d}|f$, or in other words a resource-constrained project scheduling problem with general time constraints, a deadline $\overline{d}$ and a general objective function f. But note that the presented concepts also apply to the multi-mode case.
If a project deadline is given the model for the RCPSP/max presented earlier is extended with the constraint

$$S_{n+1} \leq \overline{d} \tag{2.19}$$

As presented in the same section such a deadline constraint can easily be incorporated in the AoN network with a maximum time lag from the project's end to the project's start dummy node with an arc weight of $\delta_{n+1,0} = -\overline{d}$.
This additional constraint is added because in general objective functions are not necessarily *regular*. An objective function f is regular if it holds that

$$S \leq S' \implies f(S) \leq f(S') \tag{2.20}$$

In other words, the objective function that has to be minimized, has to be nondecreasing in the start times of the activities. This condition obviously holds for $f(S) = C_{\max} = S_{n+1}$. Late objective function that do not exhibit this behavior are presented. These are the reason why the deadline constraint is added in this section. Otherwise there might exist problem instances where the start times of activities increase unlimited while still improving the objective function.
Another common example for regular objective function is the *maximum lateness*

$$f(S) = \max_{i \in V} L_i = \max_{i \in V} (S_i + p_i - d_i) \tag{2.21}$$

Here it is assumed that there exists a due date $d_i \in \mathbb{Z}_{\geq 0}$ for every activity $i \in V$. The *lateness* is simply the difference between the completion time and the due date of a given activity.

The concept of *flow time* on the other hand considers release dates $r_i \in \mathbb{Z}_{\geq 0}$ for every activity $i \in V$. The *flow time* is now defined as the difference between the completion time and the release date of a given activity, $F_i = S_i + p_i - r_i$. Additionally a weight vector w_i^F is given to assign a degree of importance to the particular deviations. So finally the function can be stated as

$$f(S) = \sum_{i \in V} w_i^F F_i = \sum_{i \in V} w_i^F (S_i + p_i - r_i) \tag{2.22}$$

The *tardiness* is a concept similar to the lateness but is limited to positive values, $T_i = \max(L_i, 0) = \max(S_i + p_i - d_i, 0)$. It is also usually used in the form of a weighted sum and called the *weighted tardiness*

$$f(S) = \sum_{i \in V} w_i^T T_i = \sum_{i \in V} w_i^T (S_i + p_i - r_i) \tag{2.23}$$

In [21, pp. 19] the author also mentions the *weighted completion time* $f(S) = \sum_{i \in V} w_i^C C_i$, where again $C_i = S_i + p_i$ and a generalization called the *discounted total weighted completion time* $f(S) = \sum_{i \in V} {w_i}^C (1 - e^{rC_i})$. With this more complex function it is possible to control the discount of the costs with a rate parameter $0 < r \leq 1$.
Another function is *weighted number of tardy jobs*. To define this objective function the following auxiliary function is defined to provide an indicator for every activity if it is tardy.

$$U_i = \begin{cases} 1 & \text{if } C_i > d_i \\ 0 & \text{otherwise} \end{cases}$$

Together with another weight vector the objective function is defined as $\sum_{i \in V}^{T} w_i U_i$.

For the treatment of nonregularity the central concept of *earliness* is defined as $E_i = \max d_i - C_i, 0$. Given another weight vector $w_i \geq 0$ it is possible to define the *weighted earliness function* as

$$f(S) = \sum_{i \in V} w_i^E E_i = \sum_{i \in V} w_i^E (S_i + p_i) \tag{2.24}$$

In contrast to regular objective function use cases for nonregular objective functions are not that obvious. A reason for applying this function is the existence of hight storing costs of a product [14, p. 179]. This function is also not just nonregular, but *antiregular*. Antiregular objective functions have the property that $S \geq S' \implies f(S) \leq f(S')$.
Nevertheless the function does not seem practicable since it only postpones certain activities as much as possible. Another function that combines two of the presented functions is the *weighted earliness-tardiness*

$$f(S) = \sum_{i \in V} w_i^E E_i + w_i^T T_i \tag{2.25}$$

Again it holds that $w_i^E \geq 0$ and $s_i^T \geq 0$. The obvious application is to schedule just-in-time production, where it is one of the central tasks to streamline the storage infrastructure and reduce costs this way. In [14, p. 180] rescheduling is mentioned as another interesting use case. If a calculated schedule gets corrupted by unforeseen events like machine breakdowns or inaccurate

processing time estimates, it is often desirable to generate a new schedule that is as similar as possible to another one. This is not a big priority for scheduling tasks in fully automated environments, but it is for staff scheduling. For the implementation of the function the original start value of an activity $i \in V$ in the corrupted schedule is taken as d_i. This ensures that the shifts of starting times are minimized.
The last objective function mentioned here can be used to distribute activities evenly in the scheduled period of time and is called the *weighted start time deviations*. In contrast to the previously mentioned weighted functions this function requires not only an n-dimensional weight vector but an $n \times n$ matrix $w_{ij} \geq 0$. The function itself is defines as

$$f(S) = -\sum_{i \in V} \sum_{j \in V: j > i} w_{ij} |S_j - S_i| \tag{2.26}$$

In [14, p. 180] the author proposes a calculation of the weight matrix such that the variation of the resource utilization get minimized throughout the planning period.
There is also a number of other objective functions that do not seem appropriate for the problem at hand. These can be found in [14, pp. 175] where the authors also propose a classification in seven classes and provide a profound theoretical treatment as well as practical thoughts on the resolution approaches for the different classes.

Problem Translation

Having formulated the MRCPSP and the MRCPSP/max in the previous sections the problem of generating an instance from an OM has to be dealt with. For a first evaluation of the approach the possible use of UC4 script was not taken into account, because the implementation of a parser would exceed the scope of this work.
So for the remainder of this chapter it is assumed that a planning interval is given and the OM does not contain any script. Note that this means neither script objects nor scripts attached to jobs, job plans or schedules are considered.
In order to determine the set of jobs that must be executed in a given time interval we have to examine the two concepts of schedules and periodic executions. The third UC4 object that can be used to trigger the execution of a job is the time event. Due to the fact that events only work in combination with UC4 scripts, they are not taken into account.
A schedule can contain a list of jobs and job plans and has a minimum period of one day. Since it is unlikely that planning intervals are going to exceed 24 hours, it is sufficient to examine the schedule, extract the desired start time of the included jobs and job plans and add them as jobs to the problem instance.
For periodic executions more work has to be done. There are three different possibilities to define the execution frequency:

1. Every (hhh:mm): This option invokes a job or job plan invocation for the given frequency.
2. Time Gap (hhh:mm): Here it is possible to define a time gap between the completion of one execution and the start of another.
3. Time (hh:mm): In this field the user is allowed to define a specific time.

Furthermore there are two additional parameters that can influence the executions. First the user is allowed to define a time window for the executions. Of course this option is not available if a specific execution time was defined. Additionally it is possible to force the alignment of execution within this time window. Note that it is also possible to assign more complex calendars to the periodic execution which limits the number of days the execution is performed. In this work we omit those implementation details, since they do not influence the formalization itself.

The determination of the execution modes of a job is rather simple. It corresponds with the assignment of a job to a machine. Note that jobs that are assigned directly to a single host by the user only have one execution mode. If the user assigns the host to a group of agents the number of execution modes equals the number of agents within this group.
In the current version of the OM there exists only one estimated runtime which lacks context sensitivity. This deficiency is going to be eliminated in future versions, so we assume that an estimated processing time is available for every assignable agent.

For the determination of the time lags in an instance the schedules, periodic executions and job plans of an OM have to be taken into account. The integration of jobs or job plans included in a schedule's list is straightforward. Since the schedule only triggers the included objects at a predefined point in time, this can be modeled as a release date or minimum time lags respectively. These time lags are inserted between the project start and the considered activity.
As it was mentioned before schedules have a minimum time period of one day and the system allows executing jobs and job plans periodically. The possible modes for these periodic executions are presented earlier in this section. From these modes *Time (hh:mm)* requires the fewest modeling effort, since its behavior is equal to a schedule object and can be expressed with a minimum time lag. The mode *Every (hhh:mm)* is similar with the only difference of allowing a smaller period. Also for the formalization of the *Gap (hhh:mm)* the only difference is that the minimum time lag is not inserted between the start activity and every activity generated from the periodic execution but between the successive activities theirselves.
At this point it must be added that there are alternative ways to express this concepts. This is possible because usually there are no causal relationships between jobs or joplans that are executed in this way. If strict causal relations are at hand, they should be modeled as precedence relations within a job plan. A crucial question for alternative methods is how flexible the start times of these tasks are for the user, since in this formalization the activities can easily be postponed. Other approaches might be the following:

- Rigid formulation: To force the system to execute the activities on strictly defined start points it is possible not only to insert a minimum time lag, but also a maximum time lag having the same but negative arc cost in the opposite direction. The disadvantage of this formulation is that much flexibility of the formulation is lost.

- Flexibility factors: Another approach is also to insert maximum time lags, but not with the strict time lag values as in the rigid formulation. Instead these values can be relaxed with

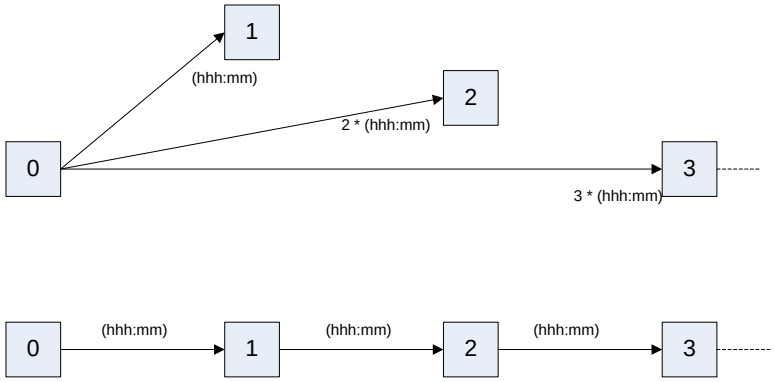

Figure 2.7: Encoding of the different recurrence variations by using only minimum time lags.

some flexibility factor, e.g. 50% of the processing time for the corresponding execution modes. These adjustments could be performed by the user or the system itself could relax the time constraints in case that more flexibility is needed to find a solution.

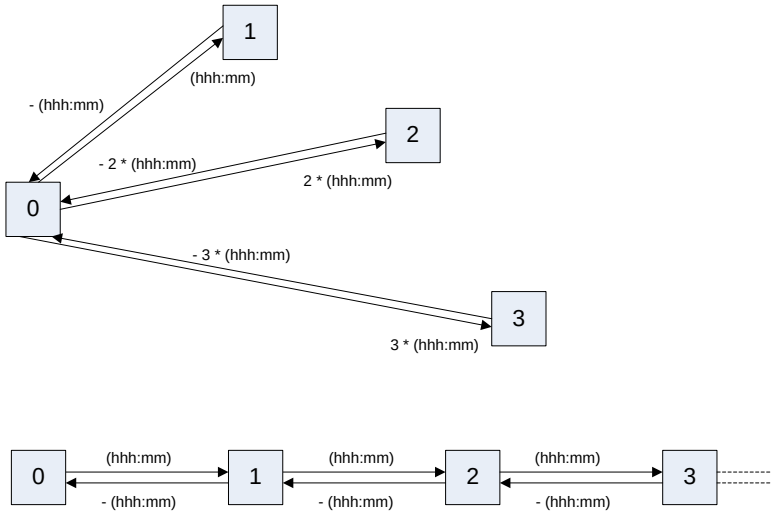

Figure 2.8: Rigid version of the encoding of the recurrence variations by using minimum and maximum time lags

- MRCPSP formulation: It is also attainable to translate the problem in an MRCPSP instance. This seems desirable since the MRCPSP/max is a real extension of the MRCPSP and therefore harder to solve. The missing capability of defining time lags with values different from the processing time can be compensated by the use of appropriate objective functions.
  Following this approach the most suitable choice to achieve a desired start time or a distribution of activities over time that is as even as possible is a combination of *weighted*

earliness-tardiness (equation 2.25) and *weighted start time deviation* (equation 2.26). All the options that can be expressed with the insertion of time lags between the project start node and the corresponding activity can be expressed with the *weighted earliness-tardiness* function, since both the release and due date are given. Additionally the weight vector can be used to prioritize the recurring objects with respect to their accurate execution.
The periodic execution in mode *Gap (hhh:mm)* cannot be evaluated with this objective function since the desired start times of an activity depends on the start time of the directly preceding activity. This can directly be modeled with the *weighted start time deviation*, where only the start time of the direct predecessor is taken into account. This is done by using the appropriate weight matrix.
But the advantage of translating the problem into an MRCPSP has the drawback that the objective functions that have to be used are nonregular. This means that commonly used local search methods that usually try to improve schedules by shifting activities to an earlier point in time are not applicable.

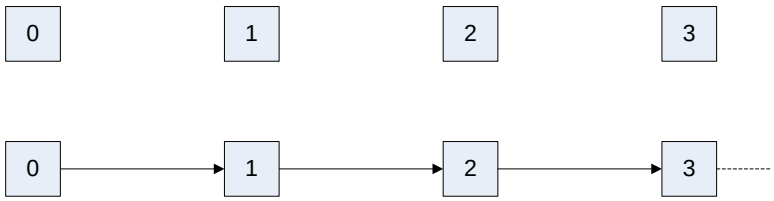

Figure 2.9: Encoding of recurrence variations as MRCPSP instance. If using these encodings, appropriate objective functions must be chosen.

Another important aspect that reduces the complexity when the MRCPSP/max encoding is used is the fact that the UC4 system supports only causal relations between jobs in a job plan. In general the user is allowed to create more general relations, but the default relation is the causal precedence relation. This means that there is no need of using a weight function where each arc $(i, j) \in A$ is described by an $|M_i| \times |M_j|$ matrix. The weights can be expressed with an $|M_i|$-dimensional vector where $\delta_i(m_i) = p_i$. So the time lag corresponds to the processing time. The same holds for release dates and due dates where corresponding time lags can be described by a single element, since they are not influenced by the execution mode of the activity.

Now the non-renewable resources are considered. The system already provides a renewable resource. It is possible to assign a resource limit to each of the agents. On the other hand the execution of a job lowers the available amount of the resource on this agent. After the execution the full amount is available again. This rudimentary conception of renewable resources has several drawbacks:

- It is totally artificial and might fail completely as a description of the actual performance of an agent.

- The same argument holds for the resource consumption.
- There is also no possibility to invent another resource, e.g. to model the amount of Random Access Memory, the number of CPUs and the bandwidth available to the agent.
- There is also another problem with the concept of resources: Resource production is assigned on agent level. But it is not guaranteed that there is only one agent installed on a machine. There might be an agent that is communicating with the OS, one that executes SAP commands and one that controls a virtual machine.
- Maybe the most important aspect to prevent practical usage is the fact that every value has to be defined by the user manually.

These points may be enhanced in the future and are expressible with the concept of renewable resources in both the MRCPSP and the MRCPSP/max.
The second possibility to mock a renewable resource is through the option of limiting the number of simultaneously running instances of certain objects. This possibility allows the user also to synchronize the access to shared resources. This is also expressible by introducing a renewable resource with limit 1 for every shared resource that may only be accessed exclusively. Jobs that need to access this shared resource indicate this in the consumption vector.

In the actual system there does not exist any analogy for non-renewable resource. But for future extensions the concept might be useful. The introduction of non-renewable resources for execution modes can indicate the usage of external resources. This might be an additional server that is purchased. It can also be used to model the ability to outsource certain jobs to an external provider of cloud computing.

2.2 Solution Approaches and Related Works

In this section a summary of the related literature is given. The primary source of references is the current survey of Weglarz et al. [30] which provides an overview for a large number of algorithms designed for various special cases and extensions of project scheduling problems.
In this section we first take a look at exact solution methods and proceed with an overview of notable heuristic methods. We conclude with a justification for the course of action chosen in this work.

Exact Methods

In a preliminary study we formulated the MRCPSP and the MRCPSP/max as a mixed integer linear program in order to evaluate the possibility to solve at least small instances in an exact manner by using the IBM ILOG CPLEX Optimizer. The study revealed that only instances with a rather small number of activities may be tackled this way. The approach worked out well for the benchmark instances with ten activities, but already for 30 activities the solver did not produce any solutions within a reasonable timespan. The detailed results obtained by the

executed test runs are presented in chapter 4.
The mixed integer linear program itself is defined as follows:

$$\text{Min} \quad s_{n+1} \tag{2.27}$$

$$\text{s. t.} \quad s_i - s_j \geq \delta_{j,m_j,i,m_i} * y_{j,m_j,i,m_i} \quad \forall (j,i) \in E, \forall m_j \in M_j, \forall m_i \in M_i \tag{2.28}$$

$$\sum_{m_j \in M_j} \sum_{m_i \in M_i} y_{j,m_j,i,m_i} = 1 \quad \forall (j,i) \in E \tag{2.29}$$

$$\sum_{m_j \in M_j} x_{j,m_j} = 1 \quad \forall j \in V \tag{2.30}$$

$$\sum_{m_i \in M_i} y_{j,m_j,i,m_i} = x_{j,m_j} \quad \forall (j,i) \in E, \forall m_j \in M_j \tag{2.31}$$

$$\sum_{m_j \in M_j} y_{j,m_j,i,m_i} = x_{i,m_i} \quad \forall (j,i) \in E, \forall m_i \in M_i \tag{2.32}$$

$$\sum_{j \in V} \sum_{m_j \in M_j} r^{\vartheta}_{j,m_j,k} * x_{j,m_j} \leq R^{\vartheta}_k \quad \forall k \in R^{\vartheta} \tag{2.33}$$

$$\sum_{j \in V} \sum_{m_j \in M_j} r^{\rho}_{j,m_j,k} * a_{j,m_j,t} \leq R^{\rho}_k \quad \forall k \in R^{\rho}, \forall t = 0 \ldots t_{max} \tag{2.34}$$

$$a_{j,m_j,t} \geq x_{j,m_j} + \sigma_{j,t} + \tau_{j,t} - 2 \quad \forall j \in V, \forall m_j \in m_j, \forall t = 0 \ldots t_{max} \tag{2.35}$$

$$C * \sigma_{j,t} \geq t - s_j + 1 \quad \forall j \in V, \forall t = 0 \ldots t_{max} \tag{2.36}$$

$$C * \tau_{j,t} \geq -t + s_j + \sum_{m_j \in M_j} (p_{j,m_j} * x_{j,m_j}) + 1 \quad j \in V, \forall t = 0 \ldots t_{max} \tag{2.37}$$

$$x_{j,m_j} \in \{0,1\} \quad \forall j \in V, \forall m_j \in M_j \tag{2.38}$$

$$y_{j,m_j,i,m_i} \in \{0,1\} \quad \forall (j,i) \in E, \forall m_j \in M_j, \forall m_i \in M_i \tag{2.39}$$

$$aj, m_j, t \in \{0,1\} \quad \forall j \in V, m_j \in M_j, \forall t = 0 \ldots t_{max} \tag{2.40}$$

$$\sigma_{j,t} \in \{0,1\} \quad \forall j \in V, \forall t = 0 \ldots t_{max} \tag{2.41}$$

$$\tau_{j,t} \in \{0,1\} \quad \forall j \in V, \forall t = 0 \ldots t_{max} \tag{2.42}$$

$$s_j \geq 0 \quad \forall j \in V \tag{2.43}$$

$$s_0 = 0 \tag{2.44}$$

The solution of an MRCPSP/max instance is a mode assignment and a schedule which defines a start time for every activity. These solutions are encoded by the following variables:

1. s_j: Denotes the start time of an activity j which may be any positive integer value or zero 2.43. The start time of the dummy activity representing the project's start time is set to zero 2.44.

2. m_j: Encodes the mode assignment of an activity j. There is no direct representation of the mode assignment as a decision variable in the formulation. Instead each mode assignment is encoded as a set of binary variables.

The formulation uses a number of binary variables to encode the different constraints:

1. x_{j,m_j}: Indicates that activity j is executed in mode m_j 2.38. The constraint that every activity must be executed in exactly one mode is realized by inequality 2.30. Furthermore the non-renewable resource constraints can be expressed with these decision variables and the corresponding resource consumption values $r^{\vartheta}_{j,m_j,k}$. The constraints are formulated with inequality 2.33.

2. y_{j,m_j,i,m_i}: To encode the time lag constraints a variable is needed to indicate the execution modes m_j and m_i of two related activities j and i. Inequality 2.29 makes sure that this joint mode assignment is unique, whereas the inequalities 2.31 and 2.32 assure the consistency of the individual mode assignments with the joint mode assignment.
   Having these indicators available it is easy to formulate the inequality that models the time lag constraints 2.28.

3. $a_{j,m_j,t}, \sigma_{j,t}, \tau_{j,t}$: For the realization of the renewable resource constraints three additional binary variables are used. To indicate that activity j is active at time t and is executed in mode m_j the decision variable $a_{j,m_j,t}$ is used. The variable $\tau_{j,t}$ is used to mark the timespan $[0, s_j + p_{j,m_j})$ whereas $\sigma_{j,t}$ is defined to indicate time values in within $[s, t_{max}]$. These characteristics are defined with the inequalities 2.37 and 2.36 respectively.
   Together with the variable $a_{j,m_j,t}$ they are used to mark the timespan where the activity j is executed with the execution mode m_j as depicted in figure 2.10 and defined with the inequality 2.35. With this indicator value it is possible to implement the constraints concerning the renewable resources 2.34.

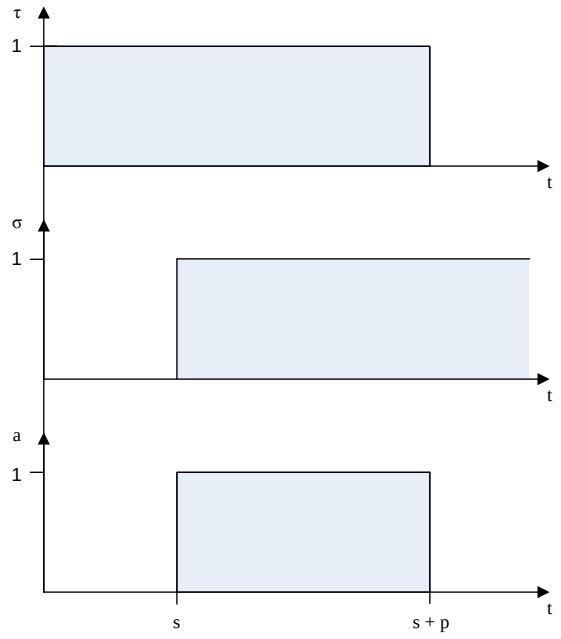

Figure 2.10: Visualization of time dependent MIP variables.

Furthermore we used the constant C which has to be large enough and was chosen to be the same as t_{max} which denotes an upper bound for the problem. Note that performance was not a number one priority in the preliminary study. Instead the goal was to explore the limits of the approach only approximately. For this reason we used the simple upper bounds already presented previously in this section.

Even though small problem instances may be solved with an MIP solver like CPLEX it has been decided that heuristic methods must be used, because the size of practical instances is typically much larger. In their survey Weglarz et al. point out that even the most sophisticated exact algorithms are not able to solve all the benchmark instances of size 30 [30]. The mentioned state of the art approaches are the branch-and-bound implementation of Hartmann and Drexl [8] and the branch-and-cut procedure of Zhu et al. [31]. The latter also relies on the usage of the CPLEX solver and solves 506 of the 552 benchmark instances optimally.

Heuristic Methods

In order to solve NP-hard and NP-complete problems as the MRCPSP and the MRCPSP/max, it is necessary to use heuristic methods if the instance size exceeds a certain limit. These methods are characterized by the fact that they do not necessarily deliver the optimal solution to an optimization problem, but an approximation within an acceptable timeframe.
Heuristic methods may be divided into three kinds: construction heuristics implement procedures to build good solutions from scratch, whereas improvement heuristics work with complete solutions and attempt to improve them incrementally. Methaheuristics on the other hand work with a potentially high number of these heuristics and guide the optimization process in some sense. Many well-known metaheuristics are inspired by processes in nature. The most prominent examples are probably simulated annealing which simulates the behavior of atoms during a controlled cooling process, evolutionary computation which is based on Darwin's theory of the evolution of species and ant colony optimization.
Early heuristic approaches for the MRCPSP and the MRCPSP/max were pure construction heuristics and scheduled activities based on their relative priority, which is determined by so called priority rules. Together with different schedule generation schemes. An example for a typical priority rule is the so called smallest latest start time first (LST) rule, where the activities are ordered based on their latest possible start time LS. A more detailed examination of priority rules and schedule generation schemes is given lateron.
In [30] Weglarz et al. give an extensive overview of different heuristic approaches for the MRCPSP, claiming that the best results are achieved with the genetic algorithms of Lova et al. [19] and van Peteghem and Vanhoucke [20]. The first one enhances earlier algorithms by using additional genes to ensure a more flexible decodification process and a new fitness function for the evaluation of invalid candidate solutions. Furthermore a new mutation operator is introduced. The results of the second paper are the best reported so far. The authors modified existing implementations by using the less common random key representation and using a population management approach where two populations with different characteristics are maintained. Furthermore the decodification process in enriched with a local search procedure.
The MRCPSP/max (also known as the MRCPSP-GPR) did not receive as much attention from the academic community. Nevertheless Barrios et al. presented also a genetic algorithm which

outperforms all previously published methods [1]. The main feature of this implementation is the decomposition of the MRCPSP/max instance into a subproblem concerned with finding mode assignments and a schedule optimization phase. Both subproblems are tackled with a genetic algorithm. Furthermore the authors present a powerful local search procedure.
These genetic algorithms form the starting point for the design of a library that provides the possibility to solve all the scheduling problems that arise in the context of the UC4 OM.
Genetic algorithms are a commonly used method in computational intelligence and are based on the work of Holland who presented the approach in 1975 [12]. Since then a tremendous amount of research has been put into the method and a large variety of applications were found suited to be solved with them.
The paradigm is motivated by evolutionary theory and simplified processes it relies on. In contrast to traditional local search procedures which work with a single candidate solution and improve it by deterministically examine its neighborhood in the solution space, genetic algorithms maintain a whole population of such candidate solutions. These candidates are encoded in some suitable form like strings or vectors of bits in the classical form. In analogy to nature these are often called chromosomes which are composed of alleles - the concrete value of one element of the string or vector. The chromosomes of a candidate solution is also called the genotype, whereas the solution of the optimization problem itself is the phenotype. It is the task of the designer to choose an appropriate encoding/decoding scheme for the stated problem. For an in-depth exploration of the topic of representation refer for example to the book of Rothlauf [24]. This work provides an investigation of different representation schemes as well as their impact on the behavior of the algorithms.
The computation itself is summarized in the following steps:

1. Initialize the population.
2. Calculate the fitness of the candidate solutions.
3. Select candidate solutions that form the next generation.
4. Perform the evolutionary operators.
5. If no termination condition is met, proceed with step 2.

Iterations of a genetic algorithm are analogously to nature called generations. The first generation must be constructed by an initialization method, which can either implement a randomized construction instance, a complete randomized construction or a mixture of both. In general it is desirable to implement a strategy that delivers a population with a high diversity.
Another important design decision is the implementation of the fitness function and - depending on it - the selection strategy. The basic idea of genetic algorithms is that high quality candidate solutions must be selected to influence the production of the next population. So advantageous sequences of their genotype, which encode desirable features of the phenotype, accumulate in the population and guide the search into more promising regions of the solution space. This idea is called the building block hypothesis. After the candidate solutions that should build the next generation are selected the evolutionary operators crossover (or recombination) and mutation are applied. A crossover operation typically takes two input candidates and creates two output

candidates which represents a random combination of the two inputs. If this operator is carefully chosen the resulting candidates will be similar to the input solutions and maintain features that are also present in either one of the input solutions.
The second operator also found in nature is the mutation. It mimics the phenomenon that offspring in nature sometimes shows features that are not present in neither one of the parents. In the context of evolutionary computation this operator works on only one individual and randomly modifies a rather small portion of its genotype. This realizes a diversification of the population and obtains the ability to introduce features that were never present within the population before or reintroduce them, if they got lost in the course of the evolutionary process.
These steps are repeated until some predefined termination condition is satisfied. Typical examples are time limits, the achievement of a satisfying objective function value or the convergence of the algorithm.
This is only a very short overview of the topics that must be covered when designing a genetic algorithm. For each of this steps there are a number of strategies that may guide implementation each with a number of parameters to tune. In the next chapter we will analyze these questions in the context of both the MRCPSP and the MRCPSP/max.

Chosen Method

After a preliminary study it turned out that the problem at hand is not tractable for instance sizes as large as they are expected in a typical UC4 OM environment. The review of the available literature showed that the most powerful approach to tackle both the MRCPSP and the MRCPSP/max is the usage of a genetic algorithm together with local search strategies (memetic algorithms).
In general genetic algorithms also provide the advantage of changeable objective function. This property adds more flexibility when deciding which value(s) must be optimized. This is important, because academic research is focused on the minimization of the execution times of the whole project. Economic or management decisions on the other hand are most often tradeoff problems, where the selected option is a compromise between time, cost and often quality.
For these reasons it was decided that the most suitable solution approach for the scheduling problems in the UC4 environment is the implementation of a genetic algorithm framework that is capable of solving both MRCPSP and MRCPSP/max instances.

CHAPTER 3

Concepts for Genetic Algorithms

3.1 Representation of Schedules

One of the critical design decisions, when realizing a genetic algorithms is which representation should be used. A comprehensive investigation of the topic is provided by Rothlauf [24].
The apparent idea of operating with arrays of start times for the activities proves indecisive after some analysis. The obvious advantage of this direct representation is that there is no need for an additional encoding scheme, which again has to be implemented and consumes time during the search. But the overwhelming drawback of this idea is the enormous extensiveness of the search space of $O(\overline{d}^n)$. For this reason a number of alternative representations for schedules have been developed. Most of them are based on two concepts:

- Ordering: A procedure or a data structure that encodes a ranking on the set of activities.
- Schedule Generation Scheme: A method that iteratively constructs a schedule using the ranking system.

With this approach the search space can be limited to the number of permutations of the set of activities, which is $O(n!)$.
The remainder of this section is organized as follows: Because of the importance of the concept of orderings of activities we start with a theoretical examination of the topic. After that the two commonly used encoding/decoding schemes are presented, followed by five sections that outline the established approaches for the encoding of the activity orderings. The investigation is focused on the activity list and the random key representations, since these were implemented and benchmarked in the course of this work.
Note that this section is in general concerned with the RCPSP and the RCPSP/max. The presence of multiple execution modes is mostly excluded and investigated in the next section.

Orderings of Activities

For the scheduling methods presented in this chapter it is useful to define strict orderings on the set of activities. These orderings take into account the causal precedence relations of the RCPSP or the temporal constraints of the RCPSP/max respectively. So they define when a certain activity is scheduled and ensure that all the causal and temporal conditions to do so are satisfied.

To state the concepts in a formal manner *binary relations* need to be introduced. Consider the binary relation ρ in set V, which is a set of pairs $\langle i, j \rangle$ with $i, j \in V$. Ordering relations are by convention often denoted with the symbol $\prec$ which can be interpreted as "before". So $\langle i, j \rangle \in \rho$ is equivalent to $i \prec j$ which can be read as activity i before activity j.

A *strict ordering* in V is a binary relation $\prec$ in V which exhibits the properties of asymmetry and transitivity. The first states that there are no elements $i, j \in V$ where it holds that $i \prec j$ and $j \prec i$, whereas the second ensures that for any $h, i, j \in V$ if $h \prec i$ and $i \prec j$ then also $h \prec j$.

In [14, pp. 15] the authors discuss the problem of defining such strict orders for the RCPSP/max. In contrast to the RCPSP where the underlying AoN network is acyclic the AoN network of an RCPSP/max instance may contain cycles. A *cycle structure* C of an AoN network N is defined as a *strong component* of the underlying graph structure. A strong component of a directed graph is a maximum subgraph where any two nodes are reachable from each other. So a cycle structure is the maximum set of nested cycles. Consider for example the temporal scheduling network N^+ in figure 2.4 from the previous chapter. Because of the artificial backward arc $\langle n+1, 0 \rangle$ all temporal scheduling networks are generally cycle structures. If the backward arc is removed the underlying AoN network is obtained. This contains two cycle structures $C_1 = \{0, 3, 4, 5\}$ and $C_2 = \{1, 2\}$.

When extracting an ordering from a cycle structure the minimum time lag d_{ij}^{min} between two activities $i, j \in C$ has to be taken into account. If this time lag is positive i has to be scheduled before j. This implies $i \prec j$. A special case is the existence of null cycles. These are cycles composed of arcs with weight 0. Such structures force the concurrent start of all activities included in this cycle. In that case neither $i \prec j$ nor $j \prec i$ holds [14, p. 16].

The use of time lags for the definition of orderings of V leads to the *distance order* $\prec_D$. This special ordering is defined for a project network N with activity set V such that for every $i, j \in V$ with $i \neq j$ $i \prec_D$ holds, if either

- $d_{ij} > 0$ or
- $d_{ij} = 0$ and $d_{ji} < 0$.

The first condition states that if there is a positive minimum time lag from activity i to activity j, i must be started before j which implies $i \prec j$. The second condition ensures that if there exists a maximum time lag between j and i it also has to be considered in the ordering by adding $i \prec j$. Note that this definition rules out the insertion of null cycles.

Furthermore the properties of a strict ordering are given. asymmetry is provided since per definition the existence of positive cycles is prohibited. Therefore it holds that $d_{ij} + d_{ji} \geq 0$ for all $i, j \in V$. Transitivity on the other hand is ensured because of the triangle inequality.

Clearly the longest paths d_{ij} in the temporal scheduling network N^+ have to be computed to establish a distance orders. This can be done for example with the Floyd-Warshall triple algorithm. The two most important aspects of the algorithm shall nevertheless be mentioned here: First the algorithm is quite efficient and exhibits polynomial time complexity and second it allows the detection of positive cycles. This in turn can be used to determine whether or not the instance at hand can be solved at all.
Note that strict orders may not only be represented as sets of pairs but also as graphs. A *precedence graph* $G^{\prec}$ is an acyclic directed graph with the node set V and containing a path from i to j if and only if $i \prec j$. It is possible to construct such a precedence graph by introducing an arc $\langle i, j \rangle$ for ever $i \prec j$. This may lead to graphs $G^{\prec}$ with a number of *redundant arcs*. An arc $\langle i, j \rangle$ is redundant if there already exists another path from i to j. Algorithms for the identification and elimination of such redundant arcs are available and can be found in [6].
At this point the set of *immediate predecessors* of an activity i in $G^{\prec}$ may be defined as $\mathrm{pred}^{\prec}(i)$. Note that this set does not necessarily coincide with the set of predecessors of the AoN network $\mathrm{pred}(i)$ as defined in the previous chapter. In general it holds that $\mathrm{pred}^{\prec}(i) \subseteq \mathrm{pred}(i)$, since the latter may also contain nodes introduced by maximum time lags or nodes from arcs eliminated because of redundancy.
Another well-known strict order from elementary graph theory that plays a role in the generation of schedules is the *topological order* $\prec_T$. This order can be defined for acyclic project networks and states that for any $i, j \in V$ with $i \neq j$, it holds that $i \prec_T j$ if and only if there exists a path from i to j. Note that the condition of $i \neq j$ has to be introduced to ensure asymmetry. The condition of transitivity is obviously fulfilled, since it reduces to simple path concatenation.
Note also that a topological order has to be handled differently when dealing with a RCPSP/max instance than when working with a RCPSP instance. Because topological orders may only work with acyclic graphs the project network N of a RCPSP/max instance is not suitable. Instead the precedence graph $G^{\prec}$ has to be used. When dealing with an RCPSP instance the project network N suffices.

Schedule Generation Schemes

In this section the common methods for schedule generation schemes are presented. These procedures are used to iteratively generate a schedule, usually under some ordering constraints imposed by a ranking of the activities to schedule. Basically there are two principles available. Either an activity-based or a time-based policy may be used for the generation. We present both approaches in detail for both the RCPSP and the RCPSP/max.

Serial Schedule Generation Scheme

The serial schedule generation scheme iteratively extends a partial schedule by one activity in each step. The order in which the activities are considered defines the resulting schedule. Such orders can be defined with priority rules or suitable data structures like an array of integer values, representing a permutation of the set of activities. Priority rule heuristics are applied for the RCPSP for a long time and a lot of research has been done in this area (see Kolisch and

Hartmann [15]). In a computational study by Neumann et al. [14, p. 84] the following priority rules turned out to be the most practicable for the RCPSP and the RCPSP/max:

- LST (smallest latest start time first): $\mathrm{ext}_{h \in E}\, \pi(h) = \min_{h \in E} \mathrm{LS}_h$
- MST (minimum slack time first): $\mathrm{ext}_{h \in E}\, \pi(h) = \min_{h \in E} \mathrm{TF}_h$
- MTS (most total successors first): $\mathrm{ext}_{h \in E}\, \pi(h) = \max_{h \in E} |\,\mathrm{Reach}(h)|$, where Reach is defined as the set of nodes i that can be reached from h in the instance's underlying network graph.
- LPF (longest path following first): $\mathrm{ext}_{h \in E}\, \pi(h) = \max_{h \in E} l(h)$, where $l(h)$ denotes the length of the longest path from h to the project's end node.
- GRD (greatest resource demand first): $\mathrm{ext}_{h \in E}\, \pi(h) = \max_{h \in E} p_h \sum_{k \in \mathcal{R}} r_{hk}$
- RSM (resource scheduling method): This complex rule selects the activity that causes the smallest delay of every other eligible activity under the assumption that there exists a causal relation between every considered activity.
  $\mathrm{ext}_{h \in E}\, \pi(h) = \min_{h \in E} \max 0, \max_{j \in E \setminus \{h\}} (\mathrm{ES}_h + p_h - \mathrm{LS}_j)$

Priority rules can be classified into static and dynamic priority rules. The values of static priority rules can be computed at the very beginning of a generation algorithm and do not need to be updated anymore. Dynamic priority rules exhibit a volatile behavior and depend on dynamic data like the eligible set E_i or the partial schedule.
The presented selection holds three static priority rules namely MTS, LPF and GRD. The first two depend on the graph structure that characterizes the problem instance, whereas the values produced by the latter are defined by the resource demand and the processing time. RSM is clearly dynamic and the last two, LST and MST, can either be implemented in a dynamic or static form by either updating the corresponding values in every iteration or using values from the preprocessing phase.

So in every iteration k an activity j is selected from the set of eligible activities E_k with respect to a given priority function p. The eligible set E_k contains all the activities that may be scheduled in this iteration. When decoding a RCPSP instance this holds all the activities that do not have an unscheduled predecessor.
When an activity is chosen it is scheduled as early as possible. The temporal constraints that originate from the causal precedence relation are respected by updating the earliest possible start time of an activity whenever one of its predecessors gets scheduled. This earliest possible start time serves as a lower bound for a function that delivers the earliest possible start time that satisfies the resource limits. This value finally serves as the start point of the activity.

Algorithm 3.1 outlines the method on a high abstraction level. The used subroutines are the following:

- *ChooseActivity:* In this procedure it is decided which activity j is considered in the actual iteration. This decision is based on the set of eligible activities and their corresponding

input : RCPSP problem instance, priority function p
output: Valid schedule S

```
1 E_0 = {0};
2 for k ← 0 to n + 1 do
3 |   j ← ChooseActivity;
4 |   t ← FindEarliestResourceFeasibleStart;
5 |   S_j ← t;
6 |   UpdateResourceProfile;
7 |   ES ← UpdateEarliestStart;
8 |   E_{k+1} ← UpdateEligibleSet;
9 end
```

Algorithm 3.1: Serial schedule generation scheme for a RCPCP instance.

priority values given by the priority function p. Typically it can be described with the formula

$$j = \min_{i \in E} p(i) \tag{3.1}$$

- *FindEarliestResourceFeasibleStart:* This function delivers the earliest time to schedule a given activity with respect to the available renewable resources. Furthermore the earliest possible start time ES_j is considered to satisfy the time constraints. Formally speaking it searches a minimal time point t^* such that

$$t^* \geq \mathrm{ES}_j \text{ and } r_k(S,t) - r_{jk} \geq 0, \quad \forall k \in \mathcal{R}, t \in [t^*, t^* + p_j] \tag{3.2}$$

- *UpdateResourceProfile:* In this function the resource profile of the partial schedule is updated. This is done by decreasing the available amount of every resource in the time interval the new activity is scheduled.

$$r_k(S,t) \leftarrow r_k(S,t) - r_{jk}, \quad \forall k \in \mathcal{R}, t \in [t^*, t^* + p_j] \tag{3.3}$$

- *UpdateEarliestStart:* To update the earliest start values it is sufficient to consider the successors of the freshly scheduled activity and increase their corresponding values in case that the completion time of the scheduled successor is greater than the earliest start time.

- *UpdateEligibleSet:* For recalculating the set of eligible activities for the next iteration it is checked if the successors of the scheduled activity have unscheduled predecessors left. If no unscheduled predecessor is found, the activity is eligible in the next iteration.

Parallel Schedule Generation Scheme

In contrast to the serial generation scheme which works in an activity-oriented manner by scheduling an activity in every iteration, the parallel generation scheme is time-oriented. Every iteration in this algorithm resembles a time step instead of an activity. The central idea of

input : RCPSP/max instance, priority function p
output: Valid schedule S

$E_0 = \{0\}$;
for $k \leftarrow 0$ **to** $n+1$ **do**
  $j \leftarrow$ `ChooseActivity`;
  $t \leftarrow$ `FindEarliestResourceFeasibleStart`;
  if $t > LS_j$ **then**
    $u \leftarrow u + 1$;
    `Unschedule`;
  end
  else
    `Schedule`;
  end
end

Algorithm 3.2: Serial schedule generation scheme for a RCPCP/max instance.

input : activity triggering the unscheduling operation j^*, violating timespan Δ
output: Updated completed set C and activity bounds ES and LS

$U \leftarrow \{i \in C | \, \mathrm{LS}_{j^*} = S_i - d_{j^*i}\}$;
if $0 \in U$ *and* $u > \overline{u}$ **then**
  `Terminate`;
end
foreach $i \in U$ **do**
  $\mathrm{ES}_i \leftarrow S_i + \Delta$;
  $C_k \leftarrow C_k \setminus \{i\}$;
end
foreach $i \in U$ *with* $S_i > \min_{h \in U} S_h$ **do**
  $C_k \leftarrow C_k \setminus \{i\}$;
end
foreach $j \in V \setminus C$ **do**
  $\mathrm{ES}_j \leftarrow \max d_{0j}, \max_{h \in U} \mathrm{ES}_h + d_{hj}$;
  $\mathrm{LS}_j \leftarrow -d_{j0}$;
  foreach $i \in C$ **do**
    $\mathrm{ES}_j \leftarrow \max \mathrm{ES}_j, S_i + d_{ij}$;
    $\mathrm{LS}_j \leftarrow \min \mathrm{LS}_j, S_i - d_{ji}$;
  end
end
`UpdateResourceProfile`;

Algorithm 3.3: Unschedule procedure used for a RCPCP/max instance.

this approach is to schedule as many activities as possible in such a time step.
Algorithm 3.4 outlines the main characteristics of the approach. Basically every iteration is featured by the following values:

- A time value t_k, since it is obviously ineffective to investigate every point in time. Only the ones where an activity changes their status are of significance for the task.
- The set of active activities A_k and the set of completed activities C_k.
- Another difference between the serial and the parallel generation scheme involves the concept of the set of an iteration's eligible activities E_k. In the serial version it is sufficient that an activity did not have any unscheduled predecessors to be classified as eligible. The effort of finding a valid start point is postponed to the moment, when the activity is in fact scheduled. In the parallel case both the temporal and the resource constraints have to be satisfied for the activities included in E_k.

input : RCPSP problem instance, priority structure P
output: Valid schedule S

1 $k \leftarrow 0$; $t_0 \leftarrow 0$; $A_0 \leftarrow \emptyset$; $C_0 \leftarrow \emptyset$;
2 **while** $C_k \neq V$ **do**
3 $\quad E_k \leftarrow$ `InitializeEligibleSet`;
4 $\quad$ **while** $E_k \neq \emptyset$ **do**
5 $\quad\quad j \leftarrow$ `ChooseActivity`;
6 $\quad\quad S_j \leftarrow t_k$;
7 $\quad\quad$ `UpdateResourceProfile`;
8 $\quad\quad A_k \leftarrow A_k \cup j$;
9 $\quad\quad E_k \leftarrow$ `UpdateEligibleSet`;
10 $\quad$ **end**
11 $\quad t_{k+1} \leftarrow \min_{i \in A_k} S_i + p_i$;
12 $\quad A_{k+1} \leftarrow A_k \setminus \{i | S_i + p_i = t_{k+1}\}$;
13 $\quad C_{k+1} \leftarrow C_k \cup \{i | S_i + p_i = t_{k+1}\}$;
14 $\quad k \leftarrow k + 1$;
15 **end**

Algorithm 3.4: Parallel schedule generation scheme for a RCPCP instance.

A closer examination of algorithm 3.4 shows that it consists of two nested loops. The outer providing the next iteration's characteristic values and the inner realizing the actual scheduling. The procedures *ChooseActivity* and *UpdateResourceProfile* in the inner loop are identical to the ones used for the serial schedule generation scheme. But the two subroutines manipulating the set of eligible activities have to be different, since the underlying concept also diverges.

- *InitializeEligibleSet:* This operation calculates the eligible set for each time step. As in the serial generation scheme an activity must not have an open causal predecessor or be

contained in the completed set itself for being eligible. In contrast to the serial generation scheme the activity must be startable in the given time step without violating any resource constraints. Furthermore it must not be included in the active set.

$$\text{pred}(j) \setminus C = \emptyset, j \notin (A_k \cup C_k) \text{ and } R_k \leq r_{jk}, \quad \forall k \in \mathcal{R}, t \in [t_k, t_k + p_j] \tag{3.4}$$

must hold, then $j \in E_0$.

- *UpdateEligibleSet:* Updating the eligible set in the inner loop of the algorithm basically consists of two steps. First the activity that has been scheduled has to be removed. Then every activity of the set has to be examined to ensure that it is still startable when taking the modified resource profile into account.

Another crucial aspect is that in contrast to the serial generation scheme which produces active schedules the parallel version delivers non-delay schedules (Kolisch [15]). Non-delay schedules have the property that no activity can start earlier without delaying some other. This also holds if preemption of activities is allowed.
This fact implies that the solution space of the parallel schedule generation scheme is smaller, which leads to a major drawback in the method shown by Kolisch [15]: given a regular objective function for the problem, the solution space of the parallel schedule generation scheme might not contain the optimal solution.
Still the method is considered in this work, because in the same paper the author claims that it performs well for highly resource-constrained instances. Furthermore the use of a population-based metaheuristic allows us to use both approaches and integrate the choice of the decoding algorithm into the genotype of the individuals.
Algorithm 3.5 shows how the method can be modified to handle RCPSP/max instances. Note that there are several modifications compared to the version for the RCPSP. Similarly to the serial generation scheme the lower and upper bounds ES_j and LS_j have to be maintained for every unscheduled activity j as the algorithm proceeds. The procedure also calls the same unscheduling method like the serial implementation and uses a counter u to limit the number of unscheduling steps.
The second adaption that has to be made is concerned with the initialization of the eligible set for every time step. The activities provided by the initialization subroutine is not necessarily schedulable at the current time step without violating the resource constraints. Instead the procedure delivers the activities that might be scheduled with respect to the precedence constraints while having the minimum earliest start time.
So the specific point in time in this iteration t_k can directly be extracted from any activity in the set. This possibility makes the maintenance of a set containing the active activities superfluous. On the other hand the method for finding the earliest possible start time with given resource constraints has to be used.
Note that this method provides another point in time t^* and only if this value equals the earliest possible start time of the examined activity it actually gets scheduled. If it is contained in the interval $[\text{ES}_j + 1, \text{LS}_j[$ the earliest start times of all the unscheduled activities are updated. And if it is greater than the latest start time of the given activity an unscheduling step has to be performed.

A detailed example using the GRD priority rule and this generation scheme with multiple unscheduling steps can be found in [14, pp. 90].

input : RCPSP problem instance, priority structure P
output: Valid schedule S

```
1  k ← 0; t_0 ← 0; C_0 ← ∅; u ← 0;
2  while C_k ≠ V do
3      E_k ← InitializeEligibleSet;
4      t_k ← min_{i∈E_k} ES_i;
5      while E_k ≠ ∅ do
6          j ← ChooseActivity;
7          t* ← FindEarliestResourceFeasibleStart;
8          if t* > LS_j then
9              u ← u + 1;
10             Unschedule E_k ← ∅;
11         end
12         else
13             if t* > t_k then
14                 foreach i ∈ V \ C do
15                     ES_j ← max(ES_j, t* + d_ji);
16                 end
17             end
18             else
19                 S_j ← t_k;
20                 UpdateResourceProfile;
21                 C_k ← C_k ∪ j;
22                 foreach i ∈ V \ C do
23                     ES_i ← max(ES_i, S_j + d_ji);
24                     LS_i ← max(LS_i, S_j − d_ji);
25                 end
26                 E_k ← UpdateEligibleSet;
27             end
28         end
29     end
30     k ← k + 1;
31 end
```

Algorithm 3.5: Parallel schedule generation scheme for a RCPCP/max instance.

Activity List Representation

One of the most frequently used representation schemes is the activity list. An activity list

$$\lambda = (i_1, i_2, \ldots, i_n) \tag{3.5}$$

is a permutation of the set of nodes V. Such a list can be interpreted as a priority rule since it imposes an ordering on the elements of V. Formally this priority rule can be defined as $\mathrm{ext}_{h \in E}\, \pi(h) = \min(\omega | h_\omega \in E)$, where E denotes the set of eligible activities as used in the previously presented decoding algorithms. So basically the rule chooses eligible activity i_ω with the smallest index ω.
Furthermore an activity list represents a strict order $\prec_\lambda$ of the set of activities, where $i_\omega \prec_\lambda i_\upsilon$ if $\omega > \upsilon$. This definition naturally satisfies both, assymetry and transitivity. Furthermore the strict order exhibits the property of linearity, which means that for any two activities $i, j \in V$ with $i \neq j$ it holds that $i \prec_\lambda j$ or $j \prec_\lambda i$ [14, p. 94].
Note that in general the solution space of the activity list representation contains $n!$ individuals. But in fact two activity lists L and L' represent the same solution if the orderings $\prec_\lambda$ and $\prec_{\lambda'}$ select the same activity for every eligible set calculated during the schedule generation. So it is save to limit the solution space to permutations that do not assign a smaller index to an activity's predecessor than the activity has itself. Activity lists that satisfy this condition are also called *precedence-feasible*.
Naturally this definition only makes sense for acyclic project networks like they appear for RCPSP/max instances. For RCPSP/max instances on the other hand the appropriate precedence graph $G^{\prec_D}$ can be taken into account. The size of the solution space can be reduced to the number of topological sortings of that graph.

Random Key Representation

Another frequently used representation method is the random key representation [22]. In general an individual is represented by an array

$$\rho = (r_1, r_2, \ldots, r_n) \tag{3.6}$$

where $r_i \in \mathbb{R}$. The interpretation as a priority rule is straightforward and simply reads every random key as the priority value of the corresponding activity. Formally this can be expressed as $\mathrm{ext}_{h \in E}\, \pi(h) = \min(r_h | h \in E)$, given that the common convention is followed that lower random keys mean higher priority.
An advantage of the random key representations is the fact that every solution can be interpreted as a point in an n-dimensional solution space. This is a precondition for the application of certain metaheuristics like scatter search and electromagnetism [4].
Nevertheless activity lists are the preferred representation scheme. A probable reason for that is the study published in [9]. In this paper the authors claim that in general activity lists outperform other representation schemes, including random keys. In [4] on the other hand the authors strongly disagree and stress the fact that the study was solely based on computational resources and no reasons are given to explain the inferior performance. Furthermore they claim that they found these reasons themselves and propose approaches to overcome the weaknesses.
Since we also consider the random key representation as an alternative representation scheme in this study a thorough examination of the arguments in [4] follows. The example used throughout

this section is depicted in figure 3.1 and is taken from the same paper.

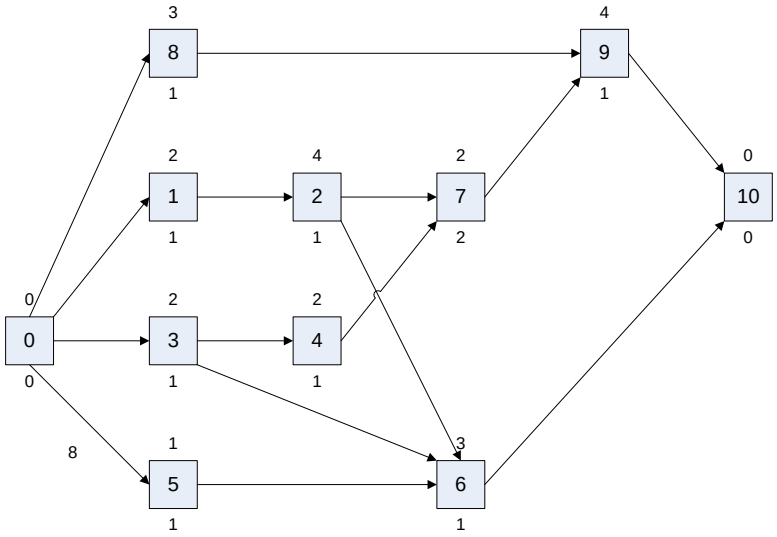

Figure 3.1: RCPSP instance.

The main point of criticism of the claims in [9] is that the solution space of the random key representation is unnecessarily large. In general both approaches, the activity list and the random key approach, suffer from the fact that different genotypes may represent the same phenotype. But it is claimed that the random key approach is much more prone to this. The authors identified four reasons for the unnecessary enlargement of the solution space, where two of them also apply to the activity list representation.

We begin with the random key individual $\rho_1 = (0.9, 1.1, 2.6, 0.7, 2.1, 0.8, 1.0, 1.9, 3.2)$ that

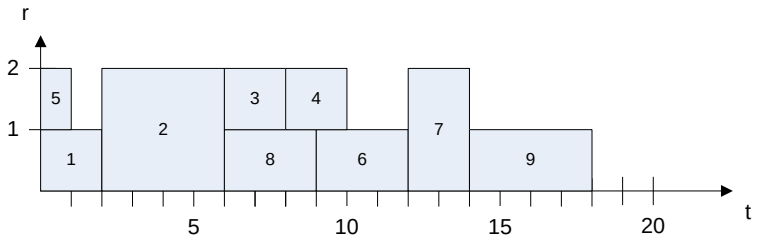

Figure 3.2: Possible schedule for the RCPSP instance depicted in figure 3.1.

results in the schedule shown in figure 3.2, when generated with a serial schedule generation scheme. The following possibilities have to be considered to limit the solution space of the problem:

1. Scaling in Euclidian space: In general random keys may be arbitrary real numbers. Using

them that way is of course far too inefficient since it bloats the solution space infinitely. In this case all the advantages from using a representation in the first place are lost. The straightforward way to get rid of this problem is to replace the priority values with the corresponding rank values. Doing this results in the individual $\rho_2 = (3, 5, 8, 1, 7, 2, 4, 6, 9)$ which is isomorphic to ρ_1. Note that the new representation still does not uniquely represent a single schedule. To obtain this property the issues below also have to be considere.

2. Precedence constraints: Like when accepting every permutation of nodes as an activity list, random keys are not precedence-feasible in general. This means that it is not prohibited that an activity has a higher priority value assigned than one of its predecessors. As with activity lists this does not result in any problems when using either the serial or the parallel schedule generation scheme, since both do not rely on the given representation when calculating the eligible set. But it does result in a unnecessary large solution space. The solution to this problem is to use the serial schedule generation scheme to obtain the rank values for the activities. In the working example the sequence of activities is $1, 2, 8, 5, 3, 4, 6, 7, 9$. This results in the random keys individual $\rho_3 = (1, 2, 5, 6, 4, 7, 8, 3, 9)$.

3. Timing anomalies: Whereas the two previous problems do not appear when working with precedence-feasible activity lists timing anomalies concern them as well. A time anomaly occurs if an activity might be scheduled earlier than another activity with higher priority. This is possible because of lower resource requirements and certain precedence constraints. In that case there also exist two or more representations for the same schedule.
   Consider the example schedule in figure 3.2 and assume that it was generated with a serial schedule generation algorithm. The latest random key representation ρ_3 assigns the values $\rho_3(5) = 4$ and $\rho_3(8) = 3$. So activity 8 has higher priority than activity 5, but it is started later in the solution schedule. The reason for that is that the activities 1 and 2 are scheduled first. This results in a resource profile that does not allow the start of activity 8 earlier than in the sixth time slot. Because of the smaller resource consumption of activity 5 is possible to start it right at the beginning. So obviously this means that the random keys of the two activities may be switched without changing the resulting schedule.
   To overcome this problem Debels et al. propose a stronger topological order of the activities. Valls et al. [29] defined the topological order representation for activity lists as a permutation $(i_1, \ldots i_n)$ of the numbers $1, \ldots, n$ satisfying the condition that $\langle i_\omega, i_\upsilon \rangle \in A$ implies $\omega < \upsilon$. This coincides with the notion of precedence-feasibility as discussed in the previous section. Debels et al. strengthen the condition such that it does not consider the precedence relations explicitly any more, but instead focuses on the start times. For the activity list representation this results in the condition $S(i_\omega) < S(i_\upsilon)$ implies $\omega < \upsilon$ and for the random keys representation culminates to $S_i < S_j$ implies $r_i < r_j$.
   The application of this principle to the example at hand provides the new solution $\rho_4 = (1, 3, 5, 6, 2, 7, 8, 4, 9)$.

4. Activities with the same starting times: The last problem that facilitates the appearance of multiple representations for a schedule is the handling of simultaneous starting times.

Whenever two activities start at the same time priority values may be interchanged without any consequences for the resulting schedule. This problem can be taken care of by assigning the same random key to the affected activities. Debels et al. assign the lowest possible rank, which results in the final and unique representation for the schedule in figure 3.2, namely $\rho_5 = (1, 3, 4, 6, 1, 7, 8, 4, 9)$.
It is also worth noting that this last point affects the activity list representation as well, but cannot be fixed for it.

The implementation of these principles leads to unique standardized random key representation, where each solution is represented by exactly one representation [4].

Priority Rule Representation

Another possibility is to encode a schedule as a list of priority rules where the number of elements equals the number of real activities in the problem instance.

$$\pi = \langle \pi_1, \pi_2, \ldots, \pi_n \rangle \tag{3.7}$$

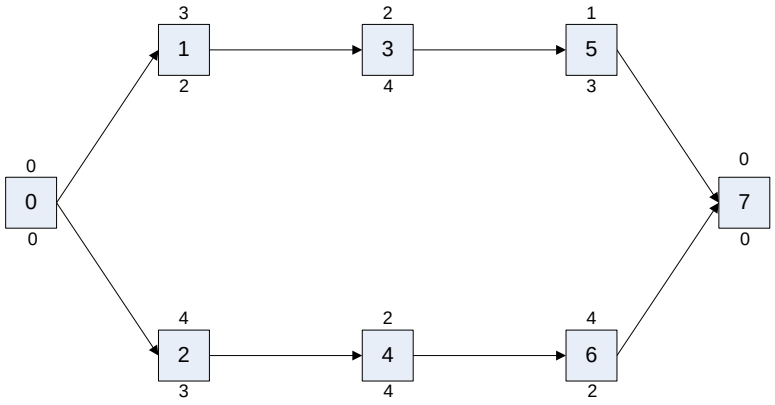

Figure 3.3: RCPSP instance.

To obtain a schedule from this representation either a serial or a parallel SGS can be used. Again the genotype can optionally be extended to encode this choice. The used SGS iteratively schedules the activities and selects the activity with the highest priority calculated with the rule assigned for the actual iteration. Consider the example solution for the project in figure 3.4

$$\pi^E = \langle \mathrm{LFT}, \mathrm{MST}, \mathrm{LST}, \mathrm{MTS}, \mathrm{GRPW}, \mathrm{LST} \rangle \tag{3.8}$$

First the selected SGS assigns start time 0 to the project start dummy activity. In the first real iteration the priority values of the eligible activities is calculated using the LFT priority rule. The activity with the highest priority is selected and scheduled. After updating the set of eligible activities the new priority values are calculated using the MST priority rule. After all real

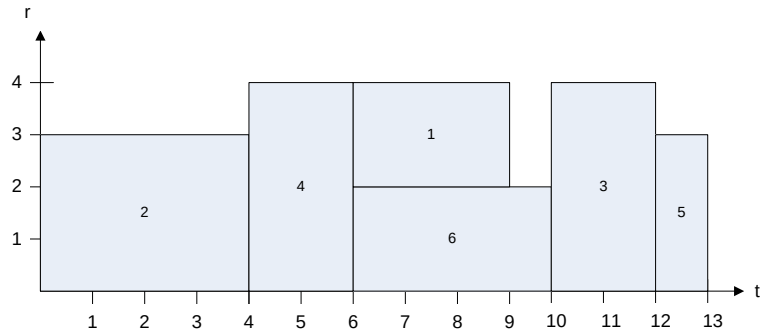

Figure 3.4: Example solution for the instance in figure 3.3.

activities are scheduled the project end activity is assigned to the earliest possible point in time. According to Kolisch and Hartmann [22] the commonly used unary operator for this kind of representation is the random change of a randomly selected index. A neighboring solution of π^E is for example

$$\pi^E = \langle \mathrm{MTS}, \mathrm{MST}, \mathrm{LST}, \mathrm{MTS}, \mathrm{GRPW}, \mathrm{LST} \rangle \tag{3.9}$$

Also binary operators like one-point or two-point crossovers can be applied without any modifications.

Shift Vector Representation

The shift vector representation is another representation scheme designed for local search methods for the RCPSP (see Sampson and Weiss [25]). A shift vector

$$\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_n) \tag{3.10}$$

contains n nonnegative integer values with σ_i representing the shift of activity i relative to its earliest possible starting time ES_i. This approach differs conceptually from the previously mentioned methods since it does not use any orders of the set of activities. Instead it encodes the structure of the schedule in a more explicit way. This implies that no SGS is used in the generation process.
The shift vector encoding the schedule of figure 3.4 is given by

$$\sigma^E = (6, 0, 1, 0, 0, 0) \tag{3.11}$$

Since the generation of schedules is not done by an SGS it is not certain that a given shift vector represents a valid schedule. The point is that for the computation of the earliest start times only time constraints are taken into account (see algorithm 2.1). For example the shift vector

$$\sigma^I = (4, 0, 1, 0, 0, 0) \tag{3.12}$$

represents a schedule where activities 1 and 4 are active at the same time, which leads to a resource constraint violation. This is a severe drawback for the representation especially for using

a genetic algorithm, because it lowers the chance to obtain a feasible individual from a crossover operation. This either forces the use of an additional repair algorithm or lowers the ratio of valid individuals in the population. Since this work also deals with the MRCPSP/max which might include complex time constraints leading to infeasible results of crossover operations, it does not seem advisable to use a representation that adds the burden of resource infeasibility already on the RCPSP level.
Also in the original work the authors did not use the representation for a genetic algorithm but for a local search. They proposed a neighborhood for a vector σ as the set of shift vectors with one different position which also do not violate a given upper bound for the makespan. An example for a neighbor of σ^E is

$$\sigma^N = (6, 0, 1, 0, 1, 0) \tag{3.13}$$

Schedule Scheme Representation

The schedule scheme representation was proposed by Brucker et al. [3] and was initially used for a branch-and-bound algorithm for the RCPSP. Basically a schedule scheme contains the four binary relations between the set of activities (C, D, N, F) which can be described as follows [22]:

- C - conjunction relation: If $(i, j) \in C$ then activity j can only be started if activity i is finished.
- D - disjunction relation: $(i, j) \in D$ implies that the activities may not be active at the same time.
- N - parallelism relation: Provided that $(i, j) \in N$ the corresponding activities have to be active together at least for one time unit.
- F - flexibility relation: This relation does not impose any restrictions.

With an appropriate decoding algorithm like the heuristic method of Baar et al. [28] it is possible to construct feasible schedules that satisfy all the restrictions imposed by the conjunction and the disjunction relation as well as a large number of parallelism restrictions. In general the schedules that are represented by a given scheme are not necessarily feasible. In the same work the authors present a neighborhood function where a move consists either of a swap of a relation element from F to N or the other way around. The considered part of the neighborhood is pruned by using critical path calculation, which is also used to guide the search heuristically.
Since the representation method was only used for a branch-and-cut and a tabu search implementation there is no literature on its applicability for a genetic algorithm. Therefore the method is not considered in detail and the reader is referred to the given material in this section for further details.

3.2 Representation of Execution Mode Selections

Since this work is concerned with the multimodal versions of the RCPSP, the genotype of a solution does not only consist of a priority function encoding an ordering on the set of activities, but also of a data structure representing the chosen execution modes. The widely accepted form of doing so is with a vector of modes

$$\mu = (\mu_1, \mu_2, \ldots, \mu_n), \quad \mu_i \in \mathcal{M}_i \tag{3.14}$$

This notation expresses the fact that the chosen execution mode of activity i is μ_i.
Note that this method differs from the notation used in section 2.1, where the mode selection was defined by a binary vector $x = (x_{im_i})_i \in V, m_i \in \mathcal{M}_i$ satisfying constraint 2.9 that ensures semantically valid mode selection. This binary representation is well-suited for the formalization of the problem as a mathematical model, but it is cumbersome for the problem at hand. Because we intend to use the formulas and definition stated in chapter 2, we define a mapping function m

$$m : \mathcal{M}_1 \times \mathcal{M}_2 \times \cdots \times \mathcal{M}_n \mapsto \{0,1\}^{\sum_{i \in V} |\mathcal{M}_i|}$$

The function can formally be defined as

$$m(\mu) = (x_{11}, \ldots, x_{1|\mathcal{M}_1|}, x_{21}, \ldots, x_{2|\mathcal{M}_2|} \ldots x_{n\mathcal{M}_n|}) : x_{im_i} = 1 \Leftrightarrow \mu_i = m_i \tag{3.15}$$

and clearly esatblishes a bijective relation between the two representations. With this mapping at hand, we use the concepts of chapter 2 without redefinition for the new input parameters. So for example the processing time of activity i using mode selection μ will be denoted as $p_i(\mu)$, which is equivalent to $p_i(m(\mu))$.
With that in mind we may define two important features of given mode selections namely the *leftover capacity* as used by Hartmann [7, p.132] and the *excess of requested resources*. The terminology of the latter used varies in the literature and is adopted in this work from Peteghem et al. [20].
First we define the leftover capacity of a non-renewable resource as

$$\mathrm{LFT}_k^{\upsilon}(\mu) = R_k - r_k^{\upsilon}(\mu), \quad k \in \mathcal{R}^{\upsilon} \tag{3.16}$$

If the leftover capacity is negative for at least one k, the mode selection is infeasible.
The sum of the absolute values of these resource constraint violations is commonly used for fitness evaluations or repair algorithms (see for example [7, pp.132], [19] and [20]). Therefore we define the excess of requested resources as

$$\mathrm{ERR}(\mu) = \sum_{k \in \mathcal{R}^{\upsilon}} \max(0, |\,\mathrm{LFT}_k^{\upsilon}\,|(\mu)) \tag{3.17}$$

Naturally a mode selection μ is resource-feasible if and only if $\mathrm{ERR}(\mu) = 0$. Because this value also describes to which extend a mode selections violates the resource constraints it is applied especially in fitness functions for the MRCPSP. These are examined in greater detail in section 3.4.

3.3 Considerations for Population-based Approaches

This section treats issues that arise when dealing with population-based optimization methods. In the first subsection the basic components of a genetic algorithm are outlines and the concept of hybridization is presented.
In the following subsections the problem-agnostic issues of how to measure the quality of individuals and how to organize the population and the sequence of generations are tackled.
The problem-specific implementations of the evolutionary operators are covered in section 3.4 and 3.5.

Hybrid Genetic Algorithms

Genetic algorithms are a widespread method for heuristic optimization. The approach was proposed in the 1970s by Holland [12] and is an area of research since then. The basic idea of the method is taken from the process of natural evolution, where a population of individuals adapts to its environment.
The basic operators mimic the reproduction in nature by modeling recombination of individuals and mutation. The adaption to the objective is implemented by evaluating the individuals of a population and selecting them for reproduction with a probability that correlates with their fitness. The generic procedure is outlined in algorithm 3.6.
Hybrid genetic algorithms also known as *memetic algorithms* extend genetic algorithms by typ-

```
1  t ← 0;
2  P(t) ← Initialize;
3  Evaluate (P(t));
4  while Terminate ≠ true do
5  |  t ← t + 1;
6  |  Q_s(t) ← Select (P(t − 1));
7  |  Q_r(t) ← Recombine (Q_s(t));
8  |  P(t) ← Mutate (Q_r(t);
9  |  Evaluate (P(t));
10 end
```

Algorithm 3.6: Algorithm for data reduction proposed by Hartmann [7].

ically applying an additional local search method such as hill climbing. The local search is usually done before the individual is evaluated, in order to reflect the full potential of an individual in its fitness evaluation.
An interesting observation of Hartmann in the context of this work is that encoding of the results of a local search in the considered individual is not always a good idea [7, pp.138]. In a computational study he shows that in the long run encoding the improvements found by local search methods decrease the genetic diversity of the populations. This is done by defining similarity metrics between individuals and executing a cluster analysis of the populations. It can be seen that the number of clusters decrease significantly faster when the local search method modifies the genotype instead of only modifying the fitness value.

Fitness Function and Selection Strategy

The goal of a fitness function is to assign a value to each individual that reflects its quality. In many cases this can not be done directly by computing the objective function value for a given individual, because populations often handle individuals that do not satisfy every constraint of the problem instance or the objective function has an inappropriate value range. Therefore two functions can be distinguished:

- An evaluation function $g(I)$, which offers a direct evaluation of an individual in a value range that corresponds to the problem domain. An example in the context of the problem domain at hand is the makespan.

- A fitness function $f(I) = s(g(I))$, where $s(x)$ denotes a scaling function, transforms the raw fitness values in a way that they are more appropriate for the selection function at hand. A fitness function must deliver a non-negative value for all individuals and must also be computable for every individual. Also a larger fitness value must always imply that the solution represented by the individual is better with respect to the objective function of the examined problem.

An important concept in the context of scaling raw fitness values is the selective pressure. This quantity can be computed for every generation as

$$S = \frac{f_{max}}{f_{avg}} \tag{3.18}$$

where f_{max} is the fitness value of the best individual and f_{avg} denotes the average fitness value of the actual population.
A too high selective pressure favors the emergence of superindividuals. These are individuals that dominate the population because of their much higher fitness values, which leads to a large replication rate of them. As a consequence, the genetic diversity of the population decreases rapidly which may result in an early convergence of the algorithm in a local optimum. If the selective pressure is too low, good individuals are not privileged as they should be and the algorithm behaves like a random search.
The second important topic when choosing a fitness function is how to handle an individual's constraint violations. For the problems at hand a number of such constraints have to be considered. A mode selection might consume to many non-renewable resources or the completion time of a preceding activity interferes with the start time of its successors. The simplest way to deal with this, is to assign a fitness value of zero, if the individual is infeasible. This line of action is often not appropriate. If for example the initialization step cannot produce a large number of feasible individuals, or the reproduction cannot be designed to generate only feasible individuals, the algorithm limits itself by excluding too many individuals from the evolutionary process.
To avoid this problem the fitness value can be adapted via *penalization*. Depending on the severity of the violations a weaker fitness value is assigned. Since the actual implementation of this is rather problem specific, the details will be given in the corresponding sections below.

When the fitness values of a population are assigned, the individuals for the recombination operator are chosen based on them. There are a number of strategies that can be selected at this point:

- *Fitness proportional selection*: The most commonly known implementation of a fitness proportional selection strategy is the *roulette wheel selection*. It has got its name because it can be pictured as an imaginary roulette wheel, where each individual owns a segment proportional to its fitness value. If the imaginary ball falls into this segment, the corresponding individual is chosen for recombination. Another implementation of this approach is the *stochastic universal sampling*. In contrast to the roulette wheel selection this algorithm guarantees that the frequency of selections of an individual corresponds with its wheel segment.

- *Rank selection*: The rank selection strategy does not rely directly on the fitness values, but assigns the selection probabilities based on the position of the ordering based on them. Relative distances do not play a role, which can be advantageous to avoid the emergence of superindividuals.

- *Tournament selection*: This selection strategy selects k random individuals from the population and selects the one with the highest fitness value. The size of k controls the selective pressure within the population. This approach is especially useful, if an individual cannot be evaluated without another one. This may be the case if the solution represents a player strategy in a game.

Population Management

Genetic algorithms have also parameters to control the population. An important one is the population size. For practical applications the termination condition is likely to be some time limit which must not be exceeded. So the choice of the population size is always a tradeoff between the size of the genetic pool and the number of generations which can be generated. Also the replacement policy which determines which individuals perish and which may be taken into the next generation might influence the performance of the algorithm.
Here are some commonly known examples for such strategies:

- *Elitism*: This is a commonly known approach to achieve a better efficiency. The idea is for each generation a predefined number of the best individuals are copied from the parent generation without any changes. Obviously the value of the overall best solution is monotonically increasing if this strategy is implemented. The disadvantage here is that the algorithm tends to generate superindividuals with the side-effects of reduced diversity, faster convergence and increased chances to get stuck in a local optimum.

- *Overlapping population*: When implementing the overlapping population strategy, only a predefined fraction of the population is newly generated. This fraction is the called

the *generation gap*. The remainder of the next generation is filled with randomly chosen individuals from the parent generation. An extreme implementation of this strategy is to only generate one individual and replace one individual in the population. This is called a *steady-state genetic algorithm*.

- *Island model*: Another strategy is to organize a number of populations independently from another. Every island may in principle use a differently parameterized genetic algorithm. After a number of generations or a defined amount of time a migration step is executed where some individuals change the island and join another population.
- *Crowding model*: In this model the overall population is again separated in a number of groups or crowds. If a new individual is generated, one crowd is chosen randomly and the most approximate individual in this crows is replaced.

Note that the last strategy calls for some similarity or distance measure for individuals. Hartmann [7, pp. 142] to use the following measure for the mode assignment:

$$\beta(I, I') = \frac{\beta'(\mu_I, \mu_{I'})}{n} \tag{3.19}$$

where I and I' are two individuals and β' denotes the function that counts the same modes for the same activities. For the metric for the activity lists he first defines the set Q, holding all the pairs of activities that are not related with respect to the transitive precedence relation. Then he defines the function α' for two activity list as the sum of these pairs with equal orderings. Again he normalizes the quantity with

$$\alpha(I, I') = \frac{(\alpha'(\lambda_I, \lambda_{I'})}{|Q|} \tag{3.20}$$

The overall similarity measure is then computed as

$$\sigma(I, I') = \frac{\alpha(I, I') + \beta(I, I')}{2} \tag{3.21}$$

This function delivers a quantity in the interval $[0, 1]$, where $\sigma(I, I') = 1$ expresses equality of two individuals and $\sigma(I, I') = 0$ holds for individual with no common feature.
Another approach is used by Debels et al. [4] for a hybrid scatter search algorithm where the unique standardized random key representation is used (see section 3.1). Here the distance of two individuals is computed as the Euclidean distance of the random key vectors in n-dimensional space.

3.4 MRCPSP

This section is concerned with the concepts for genetic algorithms for the resolution of the MR-CPSP. After the presentation of an algorithm which is capable of pruning problem instances in a way that the optimal solution is not changed, a more detailed examination of evolutionary operators is given. These include the initialization algorithms for the population the algorithm starts

with, methods for recombination and mutation as well as an overview of local search procedures and different kinds of fitness evaluations.
For the remainder of this section, an individual is a tuple $I = (\mu, \lambda)$, where μ is a mode assignment vector and λ is an abstract priority structure that can be used for a schedule generation scheme as presented in section 3.1, namely either activity lists or random keys.

Preprocessing

For some problem instances it can be possible to prune the solution space by executing a preprocessing step. Sprecher et al. [27] identified the following properties that indicate that a problem reduction can be done:

- *Non-executable modes:* A mode is called non-executable if either the associated demand of renewable or non-renewable resource is exceeded for at least one resource.
- *Inefficient modes:* Inefficiency of a mode is given, if there exists another mode for the same activity having an equal or smaller processing time and a lower or equal resource demand for every renewable and non-renewable resource.
- *Redundant resources:* Non-renewable resources may also be excluded from an instance, if the sum of maximum demands for all the activities with respect to their corresponding modes is lower or equal to the resource limit.

Since the removal of modes or redundant resources may in turn make other modes inefficient or other resources redundant, Hartmann proposes the following algorithm for data reduction [7, p. 42]:

input : MRCPSP instance
output: reduced MRCPSP instance

```
1 modified ← true;
2 RemoveNonExecutableModes;
3 while modified do
4 |   RemoveRedundantResources;
5 |   modified ← RemoveInefficientModes;
6 end
```

Algorithm 3.7: Algorithm for data reduction proposed by Hartmann [7].

Initialization

The first step of a genetic algorithm is the preparation of an initial pool of individuals. In general this initial population should offer a high degree of diversity, to allow the procedure to explore a large portion of the search space. Typically, initialization methods rely heavily on randomized subprocedures, sometimes paired with a local search or some kind of repair method.
Hartmann for example proposes a procedure to create an individual in three steps [7, p.133].

First a mode assignment μ is generated completely at random. To increase the chance of obtaining a resource-feasible mode assignment a repair method is applied to μ. This repair method makes use of the excess of requested resources $\mathrm{ERR}(\mu)$ as defined in equation 3.17.
If it holds that $\mathrm{ERR}(\mu) = 0$ the mode assignment at hand is already feasible and the repair step can be skipped. If this is not the case the mode m_j of a randomly chosen activity j is altered, which leads to the new mode assignment μ'. If $\mathrm{ERR}(\mu) \geq \mathrm{ERR}(\mu')$ the search continues with the new mode assignment. So basically the proposed method implements a first-fit strategy on the one-flip neighborhood structure. The procedure halts if either the mode assignment is feasible, or a predefined number of attempts is reached. Hartmann suggests limiting the number of attempts with the number of activities n whereas Van Peteghem and Vanhoucke use $4n$ [20].
In the last step the priority function λ has to be created. Hartmann suggests a method to obtain precedence-feasible activity lists by choosing iteratively activities from the set of eligible activities, as defined in section 3.1. The probability of an activity to be chosen next is calculated by using *regret based biased random sampling* based on the values of the latest completion times (as discussed in section 2.1).
To outline the difference between the used sampling method and the commonly used *biased random sampling* method consider some arbitrary priority value function λ. Then the probability of every activity j contained in the eligible set E can be calculated with the formula

$$p(j) = \frac{\lambda(j)}{\sum_{i \in E} v_i} \tag{3.22}$$

Regret based biased random sampling does not rely directly on the priority values [7, p.69]. Instead the so called regret values are calculated with

$$r_j = v(j) - \min_{i \in E} v(i) \tag{3.23}$$

This results in the fact that for at least one activity $r_j = 0$. To ensure that still every schedule in the search space can be generated modified regret values can be computed by

$$r'(j) = (r(j) + \epsilon)^\alpha \tag{3.24}$$

The additional parameter $\epsilon > 0$ prohibits zero values for the new regret values. Additionally another parameter α is introduced to control the influence of the bias. The higher the value of α the more deterministic the selection. An α of zero on the other hand would result in random selection. Hartmann states that $\epsilon = \alpha = 1$ provides good results [7, p.69].
In contrast Van Peteghem and Vanhoucke discard this approach and rely on a random generation of their random key priority function which respects the topological sorting [20].

An alternative approach for the generation of mode assignments can be found in [19] by Lova et al. Instead of starting with a randomly generated mode assignment, the so called *minimum normalized resources (MNR)* procedure computes the mode assignment that minimizes

$$N_{jm} = \sum_{k \in R^v} \frac{r_{jkm}}{k} \tag{3.25}$$

for every activity j and corresponding mode dependent resource demand r_{jkm}.
To obtain a diverse population this MNR mode assignment is randomly changed and then optionally repaired as outlined in algorithm 3.8. It uses the following subprocedures:

- RandomizeModes: Change the mode of up to $n/2$ activities randomly.
- RepairModewise: Examine every activity in a random order and change to a mode which reduces the ERR, if such a mode exists.

This procedure is repeated until either a resource feasible mode assignment is found, or a predefined number of attempts is reached (the authors set this limit to 200).

input : attempt limit l, minimal resource mode vector μ_{NMR}
output: Valid schedule S

```
for k ← 0 to l do
    μ ← μ_NMR;
    μ ← RandomizeModes;
    if μ feasible then
        break;
    end
    μ ← RepairModewise;
    if μ feasible then
        break;
    end
end
```

Algorithm 3.8: Initialization method introduced by Lova et al. [19].

Recombination

Commonly used recombination methods are slight variations of the standard operations like one-point or two-point crossover or uniform crossover. The modifications that have to be added to these operators are described in the following.

X-Point Crossover

X-point crossovers are one of the standard recombination methods for genetic algorithms and can be adapted for individuals based on the activity list and the random key representation. First consider the one-point crossover for two activity list individuals selected as mother $I^M = (\mu^M, \lambda^M)$ and father $I^F = (\mu^F, \lambda^F)$. With the usage of a randomly chosen crossover point $q \in \{1, \ldots, n\}$ two child individuals, namely daughter $I^D = (\mu^D, \lambda^D)$ and son $I^S = (\mu^S, \lambda^S)$ can be constructed.
First consider the activity list $\lambda^D = (i_1^D, \ldots i_n^D)$ of the first child I^D. The first q positions are

directly inherited from the mother's activity list $\lambda^M = (i_1^M, \dots i_n^M)$.

$$i_j^D := i_j^M, \;\; 1 \leq j \leq q \tag{3.26}$$

The remaining positions are inherited from the second activity list $\lambda^F = (i_1^F, \dots i_n^F)$. To ensure validity of the resulting activity list, the positions may not be copied directly. Activities already inherited from the first parent I^M are not allowed to be inserted a second time. Instead λ^F is examined from beginning to end and only activities not already in λ^D are inserted next [7, p.87].

$$i_j^D := i_k^F, \;\; \text{where } k \text{ is the lowest index s.t. } i_k^F \notin \lambda^D \tag{3.27}$$

Given for example the parent activity lists

$$\lambda^M = (1, 3, 6, 4, 2, 5) \text{ and } \lambda^F = (6, 1, 2, 5, 4, 3)$$

a one-point crossover with $q = 3$ results in the offspring

$$\lambda^D = (1, 3, 6, 2, 5, 4)$$

For the generation of the second offspring, it suffices to switch the roles of the parent individuals. The method can be generalized for more than one crossover point. This is done by segmenting the activity list into components defined by the randomly chosen crossover points and copying every other segment directly into the child solution. The other segments are then filled under the same restrictions as used for the one-point crossover.
An important property of this crossover method is the fact that by recombining two precedence feasible individuals the resulting offspring is again precedence feasible. The formal proof of that statement is given by Hartmann [7, pp. 88].

The random key encoding on the other hand allows the standard X-point crossover without any modifications. For the one-point crossover case there are again the parent solutions $I^M = (\mu^M, \rho^M = (r_1^M, \dots, r_n^M))$ and $I^F = (\mu^F, \rho^F = (r_1^F, \dots, r_n^F))$. Again let the randomly chosen crossover point be denoted by q, then the random keys of the daughter solution are defined as

$$r_j^D = \begin{cases} r_j^M & \text{if } 1 \leq j \leq q \\ r_j^F & \text{if } q+1 \leq j \leq n \end{cases} \tag{3.28}$$

The generalization for the X-point crossover is done analogously to the activity list operator.

In both cases the mode assignments are recombined in such a way that the mode selection of an activity does not change. That ensures that the offspring is more congruent to the parent solutions, which in turn results in a higher heritability. The use of operators with high heritability is in general advantageous, because they improve the chances that good features of the parents will also be found in the children.

Uniform Crossover

Another commonly known recombination method is the uniform crossover. Assume that again two activity list encoded solutions I^M and I^F are given, defined analogously to the previous section. Instead of a number of crossover points, an n-elementary sequence $g_i \in \{0, 1\}$ is generated randomly.
Then the i-th elements of the daughter's activity list λ^D is given by choosing the next, not already chosen activity from the mother if $g_i = 1$. Otherwise the next not already chosen activity from the father's activity list is added. Formally stated:

$$i_j^D = \begin{cases} i_k^M, & \text{where } k \text{ is the lowest index s.t. } i_k^M \notin \lambda^D \quad \text{if} g_i = 1 \\ i_k^F, & \text{where } k \text{ is the lowest index s.t. } i_k^F \notin \lambda^D \quad \text{otherwise} \end{cases} \tag{3.29}$$

For an illustration consider the example of the previous section with the given parent solutions. For the binary sequence $(1, 1, 0, 0, 1, 0)$ the following daughter solution is generated.

$$\lambda^D = (1, 3, 6, 2, 4, 5)$$

Like for the X-point crossover a second child is generated by swapping the roles of the two parent solutions. Furthermore it is stated in Hartmann [7, p.89] that again precedence feasible parent solutions generate precedence feasible offspring solutions.

Again for the random keys encoding the standard uniform crossover operator may be used. So the i-th random key is defined by

$$r_j^D = \begin{cases} r_j^M & \text{if } q_i = 1 \\ r_j^F & \text{otherwise} \end{cases} \tag{3.30}$$

Also the treatment of the corresponding mode assignments is unchanged.

Mutation

The mutation operator is utilized to diversify the genetic pool inherent in a population. This is typically done by changing single genes of an individual with a rather low probability. In this section methods for the mutation of priority structures and mode assignments as well as a mutation operator for the whole population are presented.

Mutation of Priority Structures

In [20] Peteghem et al. assign a randomly chosen value to a random key with a probability of 4%. In general this operation may result in precedence-infeasibility of the individual. To avoid this, we implemented a mutation operator that takes the greatest random key of all preceding and the smallest random key of all succeeding activities into account. By drawing the new value

from this interval, precedence-feasibility will be preserved. Note also that when using the SRK representation from Debels et al. [4], the random keys have to be recalculated.
The idea of preserving the precedence-feasibility is also taken into account for the topologically sorted activity lists, for example used by Lova et al. in [19]. Here an activity is randomly moved between the greatest index of the predecessors and the smallest index of the successors.
Another mutation operator for activity lists can be found in [7, p. 90]. Here Hartmann swaps two consecutive activities in the list with a given probability, if the resulting list stays precedence feasible. In both works the probability for such a mutation is fixed with 5%.

Mutation of Mode Assignments

A commonly used mutation scheme for the mode assignment vector is to assign a random mode to an activity with a given probability [7, p. 135]. This operator is executed after the mutation of the priority structure and may result in violations of the non-renewable resource constraints. Note that this independence allows using a different probability value for this mutation. For example Peteghem et al. implements a lower rate of 2% for the mode assignment mutation (compared to the 4% for the random keys) [20].
Lova et al. extend the mutation operator by using different mutation for the resource-feasible and resource-infeasible mode assignments [19]. If a mode assignment is feasible the mentioned random mode is used with a probability of 5%. But if the mode assignment is infeasible the so called *massive mutation* operator is used.
This operator assigns a random mode which may also be the actual one until either the mode assignment is feasible or every activity has been considered. Also the order in which the activities are considered is randomized. In their computational study the authors claim that the usage of this operator improves the results significantly.

Population Diversification

For further diversification of the search Lova et al. [19] implement an additional operator which replaces a part of the population with new individuals. This operator is parameterized by two probability values. The first one controls the probability with which the operator is activated and was set to 70%. The second defines for each individual with which probability it is replaced by a newly generated individual. For this parameter the authors suggest 10%. If an individual is replaced, the new solution is generated with the initialization method setting the limit of attempts to 1.

Local Search

As described in section 1.3 it is a common practice to extend a genetic algorithm with a local search that explores the search space of a solution depending on a neighborhood function.
The typical approach for the MRCPSP is to provide a procedure that enhances the phenotype that is the schedule. In Hartmann [7, pp.135] a description of two commonly used procedures can be found. The basic operation of both is the multi-mode left shift. This operation examines the valid modes of a given activity. Validity in this context means that switching from the actual

mode to the new one does not violate any non-renewable resource constraints. For all of these modes it is tested, if the change may result in a decrease of the activity's completion time. For example the new mode has a smaller duration or its renewable resource profile enables it to be scheduled earlier.
It is important to note that all changes done in this operation do not interfere with the feasibility of the input schedule. Neither may the new mode violate any non-renewable resource constraints nor does the new renewable resource profile and the possibly changed start time exceed the resource limits. Furthermore this procedure never increases the makespan of a solution.
The first procedure applied by Hartmann is the *single pass improvement*. Here for each activity of the input schedule the modes are examined ordered by their duration in a non-decreasing fashion. The activities thmselves are processed in their natural ordering. If a multi-mode left shift can be performed, the schedule is changed and the next activity is handled. This is an implementation of the first-fit strategy. After every activity has been covered the fitness value of the given individual is set to the new makespan. This procedure is applied for every individual of every generation.
An important observation is that the single pass improvement does not exploit the full potential of the given neighborhood, because every activity is examined exactly once. But it is possible that the mode change or the start time shift of some activity enables another improvement for an activity that is already processed. In other words the single pass improvement does not deliver tight schedules.
The natural way to overcome this problem is the repeated application of the single pass improvement, which is called the *multi pass improvement*. In contrast to the single pass version this approach always leads to an local optimum with respect to the multi-mode left shift neighborhood. That means the resulting schedule cannot be improved any more by applying this operation.

An extension of these two operators is presented by Lova et al. in [19]. In this work the authors enrich the procedures of Hartmann with the concept of *double justification* introduced by Valls et al. [29]. This work is concerned with the RCPSP and a local search method which basically executes a left shift and a right shift for the schedule at hand. The computational study in the paper shows that this extension pays off for a number of heuristic optimization algorithms. Lova et al. modified this local search method for the MRCPSP. For that reason they define the two following operators:

1. Multi-Mode Backward Pass (MM-B): The activities are processed in a non-increasing order with respect to their completion times. Then for all feasible modes of an activity j it is tested whether or not a right shift can be executed without violating any constraints. In other words the latest possible starting time in the time window defined by the actual starting time and the starting times of j's successors are assigned. Ties between modes are decided in favor of the mode with the smaller processing time.
   This backward pass might result in a new schedule where $S_0 = \Delta > 0$. In that case all the start times can be shifted to the left by calculating the new schedule S' with $S'_i = S_i - \Delta$ for all activities i.
   Obviously this results in an improvement of the project's makespan by Δ.

2. Multi-Mode Forward Pass (MM-F): The forward pass simply inverts the ideas of the backward pass. Namely the activities are processes in non-decreasing order of their start times and the goal is to find a new start time results in the earliest feasible completion time. Again all the modes a best-fit strategy is used, so every mode is examined and ties are resolved by processing times. The time window where a precedence feasible new start time might be found is bounded by the latest completion time of all the predecessors of an activity and its own completion time.

These two procedures apply the concepts of *right justification* and *left justification* of Valls et al. [29] to the MRCPSP. Double justification is achieved by applying both methods to a given schedule. Since applying the forward pass after the backward pass may lead to a different result than applying the forward pass first Lova et al. extended the genotype of their individuals by one gene holding either a B or an F defining which procedure is executed first. The property is also inherited via the crossover operator. Note also that like the multi pass improvement of Hartmann, this search procedure is applied until no further improvement can be achieved.
A different approach is proposed by Van Peteghem and Vanhoucke. In [20] they present an *extended serial generation scheme* which basically combines the schedule generation scheme with the local search procedure.
Instead of dealing with a complete schedule the extended generation uses a mode improvement procedure during scheduling time. With a predefined probability the mode improvement procedure is executed while scheduling an activity j. Then for every mode of j the corresponding ERR is calculated. If this value does not increase for the new mode, it is checked if the new mode enables a start time that leads to a smaller completion time. Remember that $\mathrm{ERR}(\mu) = 0$ for feasible mode assignments μ. So therefore feasibility is preserved in this procedure. After every feasible mode is checked the one that results in the smallest completion time of j is chosen and the corresponding start time is used in the schedule. The computational study of the authors showed that a probability value of 30% delivers good results.
Algorithm 3.9 outlines the principle in pseudocode.

Evaluation

The fitness function is a crucial component of a genetic algorithm, since it determines which individuals are selected for building the next generation of the population. Typically this function is strongly related to the objective function of the problem at hand, which is the project's makespan in the classical MRCPSP.
But for the MRCPSP considering only the makespan is not sufficient, since the resource constraints might be violated by the mode assignment. The common approach to enforce the satisfaction is these constraints is the penalization of individuals violating them, by assigning a lower fitness value to them.
Hartmann for example proposed the following fitness function in [7, p.132]

$$f_H(\mu, \lambda) = \begin{cases} C_{\max}(\mu, \lambda) & \text{if } \mathrm{ERR}(\mu) = 0 \\ \overline{d} + \mathrm{ERR}(\mu) & \text{otherwise} \end{cases} \tag{3.31}$$

input : activity j, mode vector μ
output: Valid schedule S

```
r ← CalculateERR;
f ← ComputeFinishTime;
for k ← 0 to |M_j| do
    μ_i ← k; r_k ← CalculateERR;
    if r_k ≤ r then
        f' ← ComputeFinishTime;
        if f' < f then
            μ ← μ';
            r ← r_k;
            f ← f';
        end
    end
end
```

Algorithm 3.9: Extended schedule generation scheme introduced by Van Peteghem and Vanhoucke [20].

So if the individual is feasible the makespan is used, whereas an infeasible individual is punished with the project's upper bound calculated with the maximum durations and the amount of missing resources. As Alcaraz et al. stated in [13], this simple fitness function suffers from two major drawbacks:

1. The punishment for infeasibility seems too hard, because the upper bound $\overline{d}$ is poor. Infeasible individuals are practically excluded from the process, which is not desirable.

2. The fitness value of an infeasible individual is independent of the resulting project's makespan.

So Alcaraz et al. formulated an alternative function:

$$f_A(\mu,\lambda) = \begin{cases} C_{\max}(\mu,\lambda) & \text{if } \mathrm{ERR}(\mu) = 0 \\ C_{\max}(\mu,\lambda) + \hat{C}_{\max} - \mathrm{LB} + \mathrm{ERR}(\mu) & \text{otherwise} \end{cases} \tag{3.32}$$

Here two new expressions are used for the calculation of the fitness for infeasible individuals. First $\hat{C}_{\max}$ denotes the maximum makespan of all feasible individuals in the current generation. This is obviously a stronger upper bound than $\overline{d}$ and it is further strengthened by subtracting LB which denotes the length of the project's critical path calculated with minimum processing times.
In [19] Lova et al. pointed out that the fitness value of infeasible individuals is composed of a mixture time units and resource units without any normalization. To overcome this weakness,

they suggested the following formula:

$$f_L(\mu,\lambda) = \begin{cases} 1 - \frac{\hat{C}_{\max} - C_{\max}(\mu,\lambda)}{\hat{C}_{\max}} & \text{if } \mathrm{ERR}(\mu) = 0 \\ \\ 1 + \frac{C_{\max} - \mathrm{LB}}{C_{\max}} + \overline{\mathrm{ERR}}(\mu) & \text{otherwise} \end{cases} \tag{3.33}$$

where $\overline{\mathrm{ERR}}(\mu)$ denotes the normalized value of missing resources, calculated as

$$\overline{\mathrm{ERR}}(\mu) = \sum_{k \in \mathcal{R}^v} \max\left(0, \frac{|\,\mathrm{LFT}_k^v\,|(\mu)}{R^v}\right) \tag{3.34}$$

The fitness value of feasible individuals with the generation's maximum makespan of all feasible individuals will be 1. The better the makespan of another feasible individual is, the closer will the corresponding fitness value to 0. Infeasible individuals on the other hand will always have a fitness value greater than 1, which ensures that infeasibility always implies a lower fitness value. Lova et al. also presented a computational study that showed a significant improvement by using f_L instead of f_H and f_A [19].

3.5 MRCPSP/max

In comparison to the MRCPSP the MRCPSP/max has received a rather small amount of attention lately. The recent survey of Weglarz et al. [30] state that the double genetic algorithm of Barrios et al. [1] outperforms all the previous approaches for medium and large instances. Therefore this section is predominantly concerned with this solution method.
The basic course of action of Barrios et al. is to deal with the increased complexity of the MRCPSP/max in comparison to the MRCPSP is to decompose the problem into two parts. At a first stage a reformulated version of the MRCPSP/max is considered which is concerned with the search for a number promising mode assignments for the solution of the problem.
Based on these mode assignments a second phase is executed where the complete schedule is constructed. Both problems are tackled with genetic algorithms as the name of the approach suggests. The basic idea of the first phase is to supply the initialization method of the second algorithm with a number of mode assignments to increase the chance of finding high quality solutions.
Therefore this section is divided into two subsections. First we examine the design and the individual operators of the algorithm concerned with the search for promising mode assignments. In the second subsection the solution methods for the overall problem are elaborated.

The Best Mode Assignment Problem

The idea of decomposing the MRCPSP/max into subproblems was also presented other authors. First the problem of finding feasible mode assignments is solved. Then these mode assignments are used to solve the scheduling problem itself. In principal there is a distinction between two decomposition strategies:

1. Decomposition: Algorithms implementing this approach realize a strict division of the problem. After the feasible mode assignment is found, the modes are fixed and the MR-CPSP/max instance is transformed into an RCPSP/max instance. An implementation that proceeds this way is the tabu-search of de Reyck and Herroelen [23].

2. Integration: The integration approach does not divide the two phases as strictly. The mode assignment calculated in the first phase might be changed in the second phase. This increase of flexibility results in general in better results. Aside from the double genetic algorithm the branch-and-bound and the priority rule algorithms of Heilmann uses this approach (see [10] and [11]).

In his priority rule algorithm Heilmann states the mode assignment problem (MAP) which is concerned with the search for resource feasible mode assignments. The problem of time feasibility is tackled when solving the RCPSP/max by using a feasibility check and performing backplanning if necessary.
For the double genetic algorithm Barrios et al. extended the MAP stated by Heilmann in order to transform it into an optimization problem, called the best mode assignment problem (BMAP). In this problem the set of decision variables is reduced to the mode assignments for the different activities. The start times are not directly taken into account, but the problem of choosing them is transferred into the objective function which is denoted as the makespan of the optimal schedule associated to a mode assignment. Of course this mode assignment has to be resource and time feasible.
This new problem can also be seen as a transformation of the whole MRCPSP/max which implies that it is of the same complexity and therefore cannot be solved by integer programming. In fact the calculation of the objective function for a given mode assignment is the resolution of an RCPSP/max problem which implies NP-hardness.
The key idea of this reformulation is to approximate the incomputable objective function of a mode assignment by a function that can be calculated by simpler means. The natural way of replacing the strict objective function is to use an upper or a lower bound for the problem.
As an upper bound Barrios et al. suggest to solve the RCPSP/max instance with a simple heuristic like priority rule scheduling. For the double genetic algorithm the authors implemented an approximation by using the critical path of the associated RCPSP/max instance's AoN network. The critical path of the network is the longest path between the two dummy nodes.
Since a time feasible mode assignment results in an AoN network without any cycles of positive length, it is possible to use a label-correcting algorithm which can be implemented to run with complexity $O(|V||E|)$. The authors justified their choice of approximation with this advantageous runtime behavior. This results from the fact that the calculation of the critical path does not need to take the renewable resource constraints into account.
As described above Barrios et al. use a genetic algorithm for the resolution of the approximated BMAP. They argue that a transformation of the MAP into an optimization problem results in the generation of mode assignments with high quality. Furthermore it is stated that the use of a genetic algorithm ensures a high diversity of the resulting population. A detailed examination of the evolutionary operators is given in the following subsections.

Population Management

An individual in this first phase of the algorithm is represented by an mode vector μ and the corresponding vector of remaining non-renewable resources $\mathrm{ERR}(\mu)$.
The algorithm uses a steady-state strategy, which means that in every generation only two individuals of the population are selected to build two candidates for the next generation. Note that the new candidates are only added to the population if the following conditions hold:

- There exists an individual with a worse fitness value in the population.
- There is no individual in the population that has the same fitness value and remaining resources vector.

If both conditions hold a new individual replaces the worst individual in the population.
The individuals for crossover are selected by using regret based biased sampling. This sampling strategy is outlined in section 3.4.
We also implemented an alternative approach with different population management and selection strategies. Instead of a steady-state algorithm a standard genetic algorithm with elitism is used. To provide a population with high diversity two additional selection strategies can be used:

- *SimilarityTournament*: This selection strategy randomly picks three candidates from the population. After that the edit distances for the three possible pairs are calculated. This is the number of activities which have a different execution mode assigned. The score of a candidate is calculated by the sum of the edit distances to the two other candidates. Then the candidate with the highest score is picked for the next generation.

- *SimilarityRoulette*: The similarity roulette combines a fitness-based roulette-wheel selection with the similarity tournament. Instead of picking the three candidates for the tournament randomly, they are selected by roulette-wheel selection with a probability inversely proportional to the length of their corresponding critical path. Again, in the end the candidate with the highest edit distance is selected.

Obviously the pure tournament approach is not really focused on the BMAP anymore. Instead the focal point is shifted to the task of providing a more diverse initial population for the second phase of the genetic algorithm. The second policy is designed to achieve a compromise between the concerns of finding solutions with short critical paths and maintaining a diverse pool of chromosomes in order to realize a flexible search procedure.
A comparison between all the different approaches is given in section 4.2.

Initialization

Now the initialization method of Barrios et al. for the BMAP is considered. For generating an initial population of mode assignments the greedy method of De Reyk et al. was randomized [23]. The original method was used for the generation of a starting solution for a tabu search and assigned modes based on the relative resource consumption. The basic idea behind this method

is to maintain a vector of remaining non-renewable resources for a partial mode assignment. So let $\overline{x} = (x_{im_i})_{i \in V, m_i \in \mathcal{M}_i}$ be a partial mode assignment that not necessarily satisfies the condition in equation 2.9 but instead

$$\sum m_i \in \mathcal{M}_i x_{im_i} \leq 1,\ i \in V \tag{3.35}$$

Then we can use the concepts of non-renewable resource consumption of modes of a single activity $r^{\upsilon}_{ik}(\overline{x})$ and when considering a (partial) mode assignment $r^{\upsilon}_{k}(\overline{x})$ as defined in earlier.
With that in mind the definition of remaining or residual nonrenwable resources can be defined as

$$r^{\text{res}}_k(\overline{x}) = rs^{\upsilon}_k - r^{\upsilon}_k(\overline{x}),\ k \in R^{\upsilon} \tag{3.36}$$

and the average relative non-renewable resource consumption of each mode and each activity is given by

$$r^{\text{avg}}_{im_i} = \sum_{k=1}^{|R^{\upsilon}|} \frac{r_{ikm_i}}{r^{\text{res}}_k(\overline{x})}\ i \in V, m_i \in \mathcal{M}_i \tag{3.37}$$

The mode with the minimum average relative non-renewable resource consumption for an activity i is denoted as m^*_i. If this procedure does not lead to a unique result the mode with the smallest processing time is chosen.
Since the remaining vector changes with every assignment the order in which the activities are considered influences the outcome of the procedure. In [23] the order of the activities is also defined by $r^{\text{avg}}_{im_i}$. In every iteration the activity with the highest value is chosen. This procedure should ensure that the problematic activities are assigned as early as possible to increase the probability of generating a feasible solution in the end.
For the double genetic algorithm this method had to be randomized to produce a broad initial population. The authors state that three different randomization strategies were evaluated:

- Randomization of the next activity selection.
- Randomization of the mode selection.
- Randomization of both selections.

It was stated that the first option lead the best results.

Evaluation

As stated in the introduction of this section, Barrios et al. use the critical path of the corresponding RCPSP/max instance to evaluate a mode assignment. The evaluation procedure also makes use of stochastic subprocedures to repair time or resource invalid individuals.
In a first step mode assignments that violate the resource constraints are repaired. If the repair algorithm fails, the fitness value of the individual is set to a large constant C. Then the individual is checked for time feasibility. This is done by checking the AoN network of the associated RCPSP/max instance for positive cycles. If such cycles occur another stochastic subprocedure is called to change the affected modes.
Only if the individual is feasible with respect to time and resource constraints, the fitness value

input : desired population size n, large constant for worst fitness C
output: population of BMAP solutions

```
1  P = ∅;
2  for k ← 0 to 2 * n do
3  |  μ ← GenerateSolution;
4  |  if μ is not resource feasible then
5  |  |  μ ← RepairResources;
6  |  end
7  |  if μ is resource feasible then
8  |  |  f(μ) ← EvaluateBMAPSolution;
9  |  end
10 |  else
11 |  |  f(μ) ← C;
12 |  end
13 |  P ← P ∪ {μ};
14 end
15 P ← {μ : μ one of fittest n};
```

Algorithm 3.10: Initialization algorithm for the BMAP used by Barrios et al. [1].

is set to the length of the longest path between start and end node. If time feasibility can not be achieved, the value is set to $C/2$.

The repair procedure for mode assignments violating the resource constraints works as follows: To fix an infeasible mode assignment a random number of modes are deleted from the solution. With this partial solution the initialization procedure is repeated. Such a restart can be executed a predefined number of times. The authors suggest the value $n_{\text{times}} = \max(10, \frac{n}{5})$. In general this method can also be used as a local search procedure, but since in this case it is used as a repair method, it is stopped when a feasible solution is generated.

The algorithm for the repair procedure for the resolution of timing constraint violations is outlined in listing 3.12. Every cycle structure of the AoN network is tested for positivity. If the cycle structure is indeed positive a subprocedure is called to fix the modes for the activities in the cycle. If this subprocedure succeeds, these modes are marked as fixed which makes them unchangeable for subsequent subprocedure calls. Otherwise the method returns false to indicate that the mode assignment could not be fixed.

The subprocedure for the repair of a single cycle structure proceeds as follows:

1. The modes of the activities of the cycle are randomly changed if they are not marked as fixed.

2. If the cycle structure is not positive anymore, calculate the new excess of requested resources value (ERR). If $\text{ERR} = 0$ the procedure is stopped and the new mode assignment is returned.

input : resource feasible BMAP solution μ, large constant for worst fitness C
output: population of BMAP solutions

if μ *is not resource feasible* **then**
  $\mu \leftarrow$ `RepairResources`;
end
if μ *is not resource feasible* **then**
  $f(\mu) \leftarrow C$;
end
if μ *is not time feasible* **then**
  $\mu \leftarrow$ `RepairPositiveCycles`;
end
if μ *is time feasible* **then**
  $f(\mu) \leftarrow C_{max}$;
end
else
  $f(\mu) \leftarrow C/2$;
end

Algorithm 3.11: Evaluation algorithm for BMAP solutions introduced by Barrios et al. [1].

input : BMAP solution μ
output: repaired BMAP solution, success indicator success

$\{C_1, C_2, \ldots, C_k\} \leftarrow$ `DetermineCycleStructures`;
$\text{fixed}(i) = 0, \forall i \in V$;
for $i \leftarrow 0$ **to** k **do**
  if $\exists C \subseteq C_k, C$ *positive cycle* **then**
    success $\leftarrow$ `RepairPositiveCycle`;
  end
  if success $= 0$ **then**
    return false;
  end
  else
    $\text{fixed}(i) = 1, \forall i \in C$;
  end
end
if success $= 1$ **then**
  return true;
end
else
  return false;
end

Algorithm 3.12: Repair routine for time infeasible BMAP solutions used by Barrios et al. [1].

3. Otherwise examine all activities which are not contained in the cycle structure and which are not marked as fixed, if a mode change decreases the ERR.

4. If the resulting mode assignment is resource valid it is returned. Otherwise the procedure is repeated a predefined number of times.

Recombination

Barrios et al. also proposed a new crossover operator to regard the cycle structures within the MRCPSP/max AoN network. The *cycle structure-based crossover operator* considers the fact that time feasibility is a crucial feature of a mode assignment individual.
To obtain a new individual from two given individuals, the procedure works as follows:

1. For every cycle structure in the AoN network all the contained mode assignments are inherited from one randomly chosen parent individual.

2. The modes of the remaining activities are also taken from only one of the parents.

For an example consider the MRCPSP/max instance from figure 2.5. This instance contains the cycle structures $\{2, 4\}$ and $\{3, 5\}$. Assume that the following two individuals were selected for crossover:

$$\mu_1 = (1, 1, 1, 1, 2, 2, 3, 1) \tag{3.38}$$
$$\mu_2 = (1, 3, 3, 2, 3, 2, 1, 1) \tag{3.39}$$

If the first cycle is inherited from μ_1, the second one from μ_2 and the remaining modes also from μ_2, the resulting individual is

$$\mu = (1, 3, 1, 2, 2, 2, 1, 1)$$

Mutation

After the creation every individual is mutated. The used mutation operator changes the mode of every activity with a given probability. If this mutation results in a resource constraint violation, the resource repair procedure from the previous section is called.

Integration Approach

In its second stage the double genetic algorithm relies on the integration approach. This means that the mode assignment and the priority structure which controls the scheduling algorithm are modified in parallel. The alternative of finding a mode assignment first, transforming the MRCPSP/max into a RCPSP/max instance and solving the non-modal problem only proved to be the inferior approach [1].
In this subsection we examine the evolutionary operators of the double genetic algorithm.

Representation and Encoding

For representation Barrios et al. choose the activity list scheme as already outlined in section 3.1. An individual is a tuple (μ, λ), where μ denotes the mode assignment vector and λ represents the activity list.
For the decoding of the genotype the authors rely on the serial schedule generation scheme for RCPSP/max instances which is also presented in the same section. As in the approaches for the MRCPSP the multimodal MRCPSP/max instance is transformed into its uni-modal counterpart using the mode assignment vector μ.
Note however that in contrast to the RCPSP, where it can be guaranteed that some schedule can be found, a call to the RCPSP/max scheduler might fail to create a schedule. This occurs, if the time constraints inhibit the generation because of positive cycle structures or because the time windows for some activities are too narrow for scheduling them without violating the limits of some renewable resource.

Population Management

In contrast to the first phase, where a steady-state genetic algorithm is used, Barrios et al. use a generational implementation for the second stage. After the initial population is generated as described in the next subsection, the following procedure is repeated until a predefined time limit is reached:

- All the individuals in the current population are randomly paired up.
- For every pair two offspring solutions are generated. Every new individual is mutated, enhanced by the local search procedure and finally evaluated.
- The population for the next iteration is built from the union of the old generation and the generated offspring by selecting the best n individuals, where n denotes the size of the initial population.

This procedure results in a monotonic decrease of the best fitness value and the average fitness value of the generations. The best solution in the final generation is therefore also the best solution found so far.

Initialization

The input for the initialization method consists of the best BMAP solutions delivered from the first phase of the double genetic algorithm. From these individuals only the feasible ones are considered and their corresponding RCPSP/max instance is calculated.
This instance is fed into the serial RCPSP/max scheduler which computes a generates an activity list based on the following constraints:

- At most 50 generations are executed for a given BMAP solution μ. If a valid schedule can be generated, it is converted into an activity list λ and the individual is ready for the second phase.

- The first try for the individual is deterministically calculated with the LST (smallest latest start time first) priority rule. If this approach fails, subsequent tries use regret based biased random sampling with the same priority rule values.

If the initialization method succeeds, both local search methods which are presented later in this section, are executed for further enhancement of the individual. The authors stress the fact that this operator is the only one in the second phase that implements a decomposition strategy, because it is the only one that does not change the mode assignment and only operates on the priority structure of the individual.

Evaluation

For the evaluation of the individuals Barrios et al. use the same fitness function as for the BMAP. If resource feasibility is not given and can not be established by the resource repair procedure the value is set to a maximum value (lowering their chances to get to the next generation, since the fitness value is not natural). Also if the individual contains positive cycle structures and the repair procedure fails, the evaluation function delivers half of the maximum value.
The only modification with respect to the first phase is that for a feasible individual the makespan of the schedule can be used instead of the approximation with the critical path.

Recombination

Barrios et al. [1] claim that a two-point crossover delivers the best results for the second phase. A detailed examination of x-point crossover operators for the MRCPSP has already been given in the corresponding section. Barrios et al. do not use any modifications for the MRCPSP/max. In this work we also consider an operator tailored for the MRCPSP/max which is based on the observation that the assignment and scheduling of cycle structures is an important feature of a MRCPSP/max solution and the cycle structure -based crossover operator used by Barrios et al. for the BMAP [1] which is outlined above. There we also provide an example for a crossover operation of two mode assignment vectors. Consider again the example in figure 2.5 which contains the cycle structures $\{2, 4\}$ and $\{3, 5\}$ and the two activity lists

$$\lambda^M = (0, 1, 3, 5, 2, 4, 6, 7) \text{ and } \lambda^F = (0, 2, 4, 6, 1, 3, 5, 7).$$

To construct two new lists proceed as follows:

1. Initialize the new lists by copying the original ones.
2. Choose a random subset from the set of cycle structures.
3. For every cycle in this subset move the contained activities to the positions they have assigned in the parent solution from which they are not copied.

For the example at hand and assuming that only the cycle structure $\{2, 4\}$ is taken into account the following solution is constructed:

$$\lambda^D = (0, 2, 4, 1, 3, 5, 6, 7)$$

When using random key encoding the priority value of the activities in the cycle structures are replaced by the one's from the second parent. Note however that this procedure does not result in unique random key representations as described in section 3.1.

Mutation

The mutation operator of the second phase is a combination of two components:

1. Mutation of mode assignment: First the mutation operator of the first phase is called. Note that this operator includes an invocation of the repair procedure presented in the previous section, if the modification of the mode assignment vector results in a violation of the resource constraint or the occurrence of a positive cycle structure.

2. Mutation of the activity list: After that, the activity list is mutated as presented by Hartmann [7, p. 90]. Two consecutive activities are swapped with a certain probability, if the resulting activity list does not violate the precedence relation.

Local Search

As for the MRCPSP algorithms the double genetic algorithm uses local search methods to improve individuals. Based on the double justification procedure by Valls et al. [29] which is presented in section 3.4, Barrios et al. propose two search methods that take the additional constraints of the MRCPSP/max into account.
The first one is called MMDJmax and when given a mode assignment μ and a schedule S as input parameters the procedure executes the following steps for every activity i (the activities are examined in non-increasing order with respect to their completion times):

1. Delete i from the schedule and update the resource profile.

2. For every execution mode $m \in \mathcal{M}_i$ calculate

   - if the mode is resource feasible,
   - the resulting time window $[\mathrm{ES}_i^m, \mathrm{LS}_i^m]$ and
   - the latest possible start time t^* that does not result in a violation of the renewable resource constraints.

3. Modify the schedule such that the new start time of i is the latest t^* found in the previous step for a feasible mode and set μ_i to the mode corresponding to the latest t^*

This procedure roughly corresponds the multi-mode backward pass (MM-B) which was discussed in the section of the local search methods for the MRCPSP. As the MM-B the MMDJmax in this version may deliver a schedule where $S_0 = \Delta > 0$. This indicates that a global left shift may be performed which results in a schedule with a smaller makespan.
The second version of the MMDJmax corresponds to the multi-mode forward pass (MM-F) and searches for the earliest start time of activity i. Also the activities are examined in a non-decreasing order of their start times instead of the non-increasing order of their completion times.

The subsequent execution of these two versions results in double justification as presented by Valls et al. [29].
Note that the MMDJmax has the property that a feasible input always results in a feasible output. The mode of an activity is only modified if the change does not result in a violation of the non-renewable resource constraint. Also the search for a new start time of an activity takes the renewable resource constraints and the time windows into account. And if the input is feasible there is always a feasible solution for every activity, namely the actual one.
Furthermore the implementation also guarantees that the resulting schedule has either the same or a better makespan than the input schedule.

The second local search procedure is called MMDJU and is based on the observation that MMDJmax limits the time windows for the search of new possible start times rather restrictively. This prohibits movements in the presence of restrictive maximum time lags which may be feasible when other activities are adjusted later on.
The proposed resolution of this problem is to only consider time restrictions that are based on activities which are already processed by the method for the calculation of the time window. This course of action may lead into a dead end as it does in the serial schedule generation scheme for the RCPSP/max. By using the unscheduling subprocedure of the schedule generator it is possible to perform a number of trials until the unscheduling limit is met. The only difference compared to the unscheduling procedure implemented in the schedule generator is that the mode of unscheduled activities is set to their original values.
If the unscheduling limit is reached during a local search, the MMDJU switches to an MMDJmax which guarantees that a feasible input also results in a feasible output.
Note that the procedures presented in this section modify both the mode assignment and the schedule (and therefore the activity list) in parallel, which makes the integration methods.

CHAPTER 4

Results

This chapter is divided into three parts: First the results obtained by solving the MRCPSP and the MRCPSP/max with a state of the art MIP solver are presented. This first step was executed to test the limits of such an approach and to verify that a heuristic optimization technique is needed.
The remaining two sections of this chapter is focused on the MRCPSP/max because of three reasons. First the extension with maximum time lags is considered to be a useful enrichment to model the problems within the domain. In section 2.1 we presented a workaround to avoid maximum time lags by modifying the fitness function. But this method seems not only unintuitive, but also complicates the usage of the objective function to optimize not only the makespan of a project but for example costs that arise because of missed deadlines. The resulting objective function is also nonregular which prohibits the usage of our common local search methods. Additionally the MRCPSP received much more attention in the scientific community and recent works already deliver results which are very close to the lower bounds (see for example Lova et al. [19] and van Peteghem and Vanhoucke [20]). Also since every MRCPSP instance is in fact a special case of the of a MRCPSP/max instance it is always possible to solve both with a library which is capable of solving the latter.
The benchmark tests of this chapter are taken from the PSPLib [1]. This library of project scheduling problems was introduced by Kolisch and Sprecher [18] with the goal of providing common test sets to enable the comparison of algorithms within the scientific community. The instances are generated with ProGen, a problem generator which can be parameterized to control instance properties like the complexity of the underlying AoN network or the availability of resources.
The library contains instance sets for not only the MRCPSP and the MRPSP/max but also for RCPSP and the RCPSP/max. The sets are designed to cover diverse instances in order to facilitate robust testing (see Kolisch, Schwindt and Sprecher [17]). Together with the data files the website also provides benchmark files which contain lower and upper bounds for every instance. These data are the basis of the following analysis.

[1] http://129.187.106.231/psplib/main.html

4.1 Mixed Integer Linear Programming Results

As already mentioned in section 2.2 this work is started with a preliminary study to evaluate the possibility of solving problem instances optimally. For the experiments we implemented a Java interface for the IBM ILOG CPLEX Optimizer in the version 12.2, which is a state of the art solver for mixed integer linear programs. Furthermore IBM provides its so called CPLEX Concert to enable users to control the solver from external programs. It also provides a Java library which provides the modules to either load CPLEX models from text files or build them up directly within the code.
With this equipment the mixed integer linear program formulated in section 2.2 was implemented. Note that the formulation is valid for both the MRCPSP and the more general MRCPSP/max. For the experiments we considered both problems, also to estimate the complexity introduced by the generalized precedence relations of the MRCPSP/max.
The test runs were executed on a 2x Dual Core AMD Opteron Processor 270 with 8 GB of memory and constrained with a time limit of ten minutes CPU time. Solutions fall in one of the three categories:

- Optimal: An optimal solution was found within the time limit.
- Suboptimal: There is a solution available, but it is not proven to be optimal.
- No Solution: No solution found at all.

The results of the test runs are summarized in figure 4.1. The first three bars visualize the results for the MRCPSP benchmarks with 10, 30 and 50 activities respectively. The latter two were generated for the MRCPSP/max with 10 and 30 activities. Note the mark at the column for the MRCPSP/max-30, indicating that for these instances the time limit was tripled and set to half an hour. This was necessary because no instance was solved within the ten minutes time limit. This also holds for larger instances of the MRCPSP/max.
The graph also shows the rather high impact of the generalization of the precedence relations. The solver was able to solve nearly a third of the MRCPSP-50 instances, more than 40% of the MRCPSP-30 instances and more than 90% of the MRCPSP-10 instances optimally. Also the average time it took to find such an optimal solution if it was found ranged from 30 seconds for the smallest to 171 seconds for the largest instances.
So obviously the use of heuristic methods is necessary when solving larger instances of the MRCPSP and the MRCPSP/max. Note that there exist more sophisticated formulations of the problem in the literature. Nevertheless there still exist instances, even in the MRCPSP-30 set, where the optimal solution is still not found [30].

4.2 Best Mode Assigment Problem Results

In this section an evaluation of the approaches for solving the Best Mode Assignment Problem (BMAP) as stated by Barrios et al. [1] is given. The resulting mode assignment vectors are used to initialize the population of MRCPSP/max solutions for the second phase of the DGA. As outlined in section 3.5 we implemented alternative approaches to improve the results. Next to

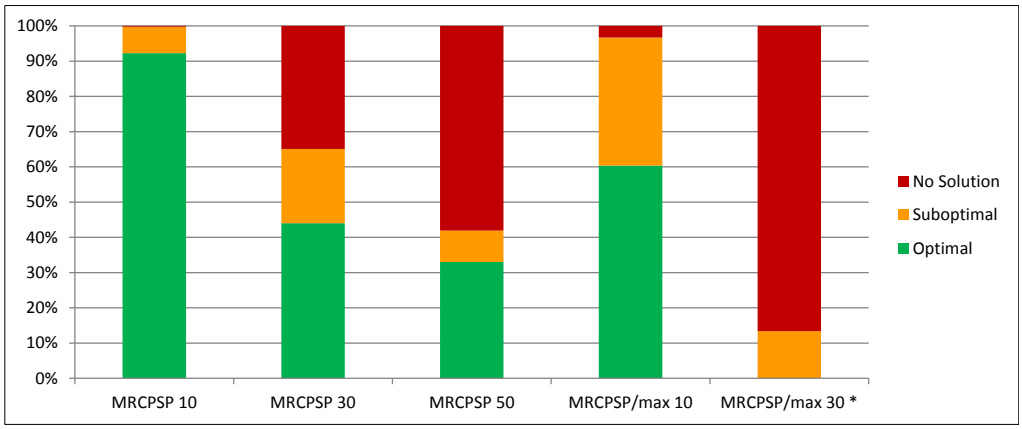

Figure 4.1: Test results for the IBM ILOG CPLEX Optimizer with a time limitation of 10 minutes (* 30 minutes).

the straightforward goals of obtaining populations with a high ratio of valid mode assignments having a short critical path value, we also take into account that high diversity is generally a desirable feature of initial populations when using genetic algorithms. This holds because features that are not present in the initial genotype pool can only enter the population through mutation. Because of the necessary limitation of mutation rates, the probability of introducing a good scheme degrades with its complexity.

For the quantification of the diversity in the result population of a BMAP run the so called Simpson index is used [26]. This coefficient is utilized in ecology to characterize the biodiversity of a habitat. Given a sample of a population consisting of N individuals of S species, the Simpson index is defined as

$$D = 1 - \frac{\sum_{i=1}^{S} n_i(n_i - 1)}{N(N-1)} \tag{4.1}$$

The resulting value corresponds to the probability of choosing two individuals of different species randomly. If every individual is from the same species, this value will be zero and if there is one individual of every species, it will be 1. So the higher the diversity of a population, the higher the value of D.

However the following considerations do not consider the diversity on the individual level but on the level of discrete alleles. This is due to the fact that the Simpson index on the individual level - meaning that two individuals are of the same species only if their genotype is exactly the same - is close to 1 for nearly every approach, especially when the genotype length exceeds 30. Instead the index is calculated for every allele that is the mode assignment for every activity. So S corresponds to the number of execution modes of the activity and N equals the number of individuals in the population. The average Simpson index is calculated as the mean over all populations of the 270 problem instances and activities.

The second metric used for the description of the diversity of the population is the average edit distance. This quantity is calculated by summing up the edit distances of every pair of individuals in the final generation and dividing it by the number of pairs. Again this value is also

averaged over the entire problem instances.
Note that for the calculation of these two quantities only the valid individuals of the population are taken into account. This results in the fact that the values cannot be compared directly when examining two approaches. The value of the similarity metrics are naturally influenced by the validity ratio in an inverse manner.
In the tables 4.1, 4.2 and 4.3 the results for the MRCPSP/max-30, MRCPSP/max-50 and MRCPSP/max-100 instances are summarized. Tests were executed for a steady-state genetic algorithm inspired by the solution procedure of Barrios et al. as presented in section 3.5, but with changed selection modes and a classical generational genetic algorithm with elitism.
The initialization, repair and mutation procedures are taken from Barrios et al. and are also outlined in section 3.5. The depicted results are taken from the runs executed with the cycle-based crossover. Nevertheless the algorithms were also tested with one-point, two-point and uniform crossover operators which did not lead to significantly different results.
All experiments were conducted with a population size of 100 individuals. As an implementation-independent stopping criterion the limitation of generated solutions was chosen to be 1000. An execution step of the repair algorithm was not counted as a generation step. The elitist count for the generational genetic algorithm was set to 10. Furthermore the mutation operator of Barrios et al. was used with a mutation probability of 4%.
So the differences between the six examined implementations are limited to aspects of the population management (steady-state vs. generational with elitism) and the selection policies. The original version of Barrios et al. is also a steady-state algorithm which chooses the individuals for reproduction based on their critical path length values by regret-based biased random sampling. For the experiments at hand selection with roulette wheel, similarity roulette wheel and similarity tournament were used. The replacement policy of the steady-state algorithm is the same as in the implementation of Barrios et al..

Evolution	Selection	Valid	CP	Edit Distance	Simpson
Steady-State	Roulette	87.03 (534.73)	62.27	8.72	0.29
	Sim. Roulette	81.11 (801.85)	63.86	11.20	0.37
	Sim. Tournament	84.86 (485.57)	66.76	12.67	0.42
Generational	Roulette	91.17 (239.25)	77.17	12.27	0.41
	Sim. Roulette	79.53 (731.16)	80.14	16.22	0.54
	Sim. Tournament	74.95 (920.06)	87.12	17.34	0.58

Table 4.1: Results for the BMAP for the MRCPSP/max-30 instances after 1000 generated solutions. For six different evolution scheme/selection strategy combinations the table shows the corresponding average percentage of valid solutions (with variance), the average length of the critical path, the average edit distance and the average value of the simpson index.

Looking at the results for the MRCPSP/max-30 instances depicted in table 4.1, we see most of our expectations to be confirmed. In general the steady-state approach is less explorative than the generational implementations. This leads to better values for the resulting critical paths, but generally less diversity in the resulting population. The second parameter - the selection policy

- is also of high importance. So a steady-state algorithm with similarity tournament selection policy delivers similar results to an algorithm implementing the generational population policy with roulette selection. Nevertheless the latter performs the best with respect to validity finding on average 91 with a rather low variance of 239.25.
With respect to diversity the generational algorithm with similarity tournament selection delivers the best results for both metrics. Of course these results should be interpreted with caution, because they have to be paid with the lowest average validity and the highest variance on this value. Nevertheless the edit distance and the Simpson index improved by a factor of 2 compared to the most conservative implementation - steady-state with roulette selection.

Evolution	Selection	Valid	CP	Edit Distance	Simpson
Steady-State	Roulette	88.11 (509.85)	105.82	13.69	0.27
	Sim. Roulette	81.82 (782.60)	105.74	17.50	0.35
	Sim. Tournament	85.08 (543.51)	111.01	20.47	0.41
Generational	Roulette	89.43 (340.97)	120.99	21.06	0.42
	Sim. Roulette	77.72 (968.43)	124.41	26.29	0.53
	Sim. Tournament	74.99 (1091.60)	131.72	28.26	0.57

Table 4.2: Results for the BMAP for the MRCPSP/max-50 instances after 1000 generated solutions. For six different evolution scheme/selection strategy combinations the table shows the corresponding average percentage of valid solutions (with variance), the average length of the critical path, the average edit distance and the average value of the simpson index.

The results for the MRCPSP/max-50 instances (see table 4.2) confirm the observations from the smaller problem. Still the generational algorithms deliver more diverse populations and the steady-state implementation obtains better critical paths.

Evolution	Selection	Valid	CP	Edit Distance	Simpson
Steady-State	Roulette	88.06 (568.73)	242.05	23.12	0.23
	Sim. Roulette	81.30 (986.64)	240.35	29.27	0.29
	Sim. Tournament	79.58 (963.73)	250.06	40.26	0.40
Generational	Roulette	79.75 (695.67)	256.80	38.06	0.38
	Sim. Roulette	72.57 (998.16)	260.55	45.54	0.46
	Sim. Tournament	47.22 (1625.72)	265.16	50.71	0.51

Table 4.3: Results for the BMAP for the MRCPSP/max-100 instances after 1000 generated solutions. For six different evolution scheme/selection strategy combinations the table shows the corresponding average percentage of valid solutions (with variance), the average length of the critical path, the average edit distance and the average value of the simpson index.

For the large MRCPSP/max-100 instances (see table 4.3) we observe a change in the performance of the algorithms. Here the steady-state algorithm shows better validity result for every

selection policy including the roulette wheel. Also the generational implementation performs poorly when combined with the similarity tournament selection leading to an average validity rate of only 47.11% with an enormous variance of 1625.72. Note also that the diversity metrics in this context cannot be compared to the other results directly, because the number of valid individuals is significantly smaller.

In conclusion we may summarize that the different implementations work generally as expected. The generational implementation delivers more diverse but less fit populations, while this effect may be controlled with the choice of the selection policy. The extreme points of the configurations are the steady-state algorithms with roulette wheel selection on the one hand and the generational implementation with similarity tournament selection on the other. While the latter fails to provide valid individuals, at least for large instances, the first results in populations with a rather small genotype pool.
Obviously there exists a tradeoff between validity, fitness and diversity when solving the BMAP. The influence of the different solution approaches on the MRCPSP/max itself is evaluated in the next section.

4.3 Results for the MRCPSP/max

The final algorithm for the resolution of the MRCPSP/max is basically an extended version of the DGA by Barrios et al. [1]. To evaluate the performance of the library, a number of test runs were executed with varying parameters to explore the impact of the different concepts presented throughout this work. Namely the following parameterizations were taken into account:

- BMAP solution algorithm: Based on the results from the previous section three BMAP solution algorithms were taken into account. The conservative steady-state algorithm with roulette selection (SSR) and the two more explorative generational implementations with similarity roulette (GSR) and similarity tournament selections (GST) respectively. For a detailed treatise of the concepts refers to section 3.5. The parameter settings for the runs correspond to the settings used in the previous section.

- Initialization: Next to the initialization method for the prioritization of the activities presented in section 3.5 which is based on the latest start time priority rule (LST) we also implemented a completely randomized version (RND). In this implementation the chance of every eligible activity to be chosen next is uniformly distributed, which was thought to result in more variability in the genotype pool.

- Encoding and SGS: The implementation allows also the configuration of the chosen encoding scheme. As an alternative to the activity list representation (AL) used in the original DGA, we also implemented the random key encoding (RK) to represent the priority structure (see section 3.1). Furthermore there exists the possibility to use different schedule generation schemes. In addition to the serial SGS (SS) we provide a mixed ver-

sion (MXD) where the SGS of an individual is fixed during initialization (either serial or mixed). Then the SGS is inherited during the recombination step.

- Selection: For selecting individuals for recombination we used stochastic universal sampling (SUS), tournament selection (TS) and rank-based selection (RS).
- Recombination: Aside from the one-point (PX1), two-point (PX2) and uniform (UX) crossover we also tested the cycle-based crossover (CYC) for priority structures which is also outlined in section 3.5.
- Population Size: Tests also included running the algorithm with different population sizes (P100, P200 or P500).

For the introduction of mutation we used the procedure from the original DGA with a mutation probability of 4%. We also used the local search procedure MMDJ proposed by Barrios et al. [1]. The number of rescheduling steps of the MMDJU subprocedure and the SGS was limited to the number of activities in the problem instance.
Recently Kolisch et al. suggested that benchmarks should be computed with respect to the count of generated schedules to enable a fair comparison of different approaches unbiased by implementation details and hardware configurations [16]. Therefore we present results after 1000, 5000 and 10000 generated schedules instead of a limitation of CPU time. Note that a schedule counts if it is generated successfully by an SGS. A modification in course of the execution of a local search procedure does not count as a schedule generation.

The following tables show the results for the MRCPSP/max-30, MRCPSP/max-50 and MRCPSP/max-100 instance sets. We tested the library with a large number of different parameter settings but only provide the most important here. Every table starts with parameter setting which is the best-performing (after the generation of 10000 solutions) found in course of the analysis. The following 12 settings change exactly one of the parameters of the best performing setting. This way it is assured that at least a local optimum with respect to this one-switch neighborhood has been found.
The following six columns show the mean $\%$-gap relative to the lower bounds and the corresponding variance which are determined over all the 270 instances for the three benchmark checkpoints. The last column indicates the statistical significance of the results. The test for statistical significance checked the null hypothesis claiming that the given parameter setting performs the same as the best one with respect to the relative makespan deviation by using the Student's t-test. The *P-value* denoted as ρ in the last column shows the probability of the null hypothesis being true. So the smaller the value the stronger is the evidence that the corresponding parameter setting is worse than the best one.
For the MRCPSP/max-30 instances (see table 4.4) the best performing combination was

$$\text{BEST}_{i30} = \text{(SSR, LST, CYC, AL, P200, MXD, SUS)}$$

which uses a steady-state roulette BMAP algorithm using the LST initialization rule, the activity list representation with mixed scheduling and the cycle-based crossover operator. The population size was set to 200 and for selection the stochastic universal sampling method was used.

This parameter setting delivered the best results after 5000 and 10000 generated schedules. After 1000 schedules the configuration using a random key encoding showed the best results. It is also interesting to see that the selection method had a large impact on the long run. Both the rank and the tournament selection outperformed the sampling method after 1000 schedules, but yield no convincing results in the end. By far the most negative impact on the performance has the switch from the cycle-based to the uniform crossover method which resulted in the worst results in the test set. Also the incorporation of the 2-point crossover, which is the chosen crossover operator of the original DGA, leads to slight decrease of performance. The results of the statistical significance test also show that the best setting performs significantly better than the version with the GSR, RND, UC and PX1. These have a P-value of $< 5\%$. For the other settings the result is not as clear, but still no P-value reaches more than 30%.

Instance	Percentage Gap						ρ [%]
	1000 Schedules		5000 Schedules		10000 Schedules		
	Avg	Variance	Avg	Variance	Avg	Variance	
BEST$_{i30}$	14.08	269.85	11.86	209.26	11.29	187.77	-
GSR	14.07	270.02	12.35	226.57	11.88	204.56	0.37
GST	13.66	268.87	11.95	199.50	11.47	187.71	29.91
RND	14.36	264.46	12.43	209.03	11.90	193.19	0.90
UX	14.75	272.02	13.02	242.61	12.37	219.78	0.90
PX1	14.37	275.43	12.12	210.28	11.81	193.13	2.42
PX2	13.74	280.28	11.97	211.41	11.53	190.50	12.93
RK	13.59	248.64	12.21	212.15	11.60	188.26	12.93
P100	14.27	265.25	12.50	215.29	11.82	188.37	14.43
P500	13.88	265.90	12.21	211.07	11.75	198.67	5.28
SS	13.85	274.92	12.12	206.70	11.57	188.23	24.84
TS	14.02	253.72	12.26	200.50	11.94	195.84	5.44
RS	14.02	264.60	12.18	206.19	11.91	195.26	16.22

Table 4.4: Results for the MRCPSP/max-30 instances solved with different parameter settings.

Now consider the results for the MRCPSP/max-50 instances depicted in table 4.5. For this instance set the best found configuration included the usage of the generational similarity-tournament BMAP algorithm and LST initialization combined with the activity list encoding, the classical serial SGS and again the cycle-based crossover operator. This time a population of 500 individuals proved to be advantageous, but again the stochastic universal sampling method was used.

$$\text{BEST}_{i50} = (\text{GST, LST, CYC, AL, P500, SS, SUS})$$

In contrast to the previous test set, the best performing configuration here delivered the best solutions consistently for all three checkpoints. Again the incorporation of the uniform crossover operator deteriorates the results significantly, whereas the 1-point and 2-point crossovers are closely behind. Another configuration change with a negative influence is the change of the initialization method to random. This change results in the worst performance after 1000 schedules.

Instance	Percentage Gap						ρ [%]
	1000 Schedules		5000 Schedules		10000 Schedules		
	Avg	Variance	Avg	Variance	Avg	Variance	
BEST$_{i50}$	16.81	293.66	13.97	211.88	13.18	184.54	-
SSR	17.03	292.77	14.14	217.75	13.46	196.53	29.23
GSR	17.08	294.96	14.45	222.58	13.78	211.08	3.05
RND	18.17	314.37	14.41	211.42	13.95	196.96	< 0.01
UX	17.27	293.71	14.88	239.49	14.02	214.69	0.26
PX1	17.11	280.73	14.07	214.98	13.23	191.52	87.43
PX2	17.36	297.12	14.15	201.89	13.36	180.29	53.21
RK	16.92	281.40	14.04	210.49	13.68	202.88	6.18
P100	17.82	311.79	14.27	218.32	13.60	207.86	16.64
P200	17.83	307.99	14.05	212.12	13.29	196.27	66.48
MXD	17.11	297.17	14.14	216.54	13.59	196.08	21.10
TS	16.86	286.98	14.17	222.11	13.62	204.14	16.62
RS	16.82	275.72	14.15	214.23	13.64	201.04	8.34

Table 4.5: Results for the MRCPSP/max-50 instances solved with different parameter settings.

The caused damage seems to be repaired in the course of the run, but still leading to mediocre results.
With respect to statistical significance these results are much closer. The 1-point and 2-point crossovers show a high probability of performing equally well as the best setting. Also a decrease of the population size to 200 does not result in too worse results.

The last test set is concerned with the MRCPSP/max-100 instances (table 4.6) and is also the most interesting one, because the library must be able to deal with large problems in practice. The best configuration for this problem class was

$$\text{BEST}_{i100} = \text{(SSR, LST, CYC, RK, P200, SS, SUS)}$$

This is a steady-state roulette BMAP procedure again with the LST initialization implementation. This time the random key representation combined with a serial SGS proved to be the best choice for the encoding/decoding scheme. Again the cycle-base crossover and the stochastic universal sampling method outperformed the alternative approaches. The population size was set to 200 candidate solutions. Note however that the victory in this test set was not undisputed. By far the best start was executed by the configuration using the generation similarity-tournament BMAP algorithm we proposed earlier in this work. For the checkpoint after 5000 generated schedules the configuration using a mixed SGS outperformed the others. Both configurations finished as runner-ups, being only 0.05% behind the best configuration. The change of the crossover operator also resulted in significantly deteriorated results. Again the uniform crossover performed worst consistently over all checkpoints.
The results from the statistical test show similar results with P-values of more than 80% for the generation similarity-tournament BMAP and the mixed scheduling approach. The other settings

Instance	Percentage Gap						ρ [%]
	1000 Schedules		5000 Schedules		10000 Schedules		
	Avg	Variance	Avg	Variance	Avg	Variance	
$BEST_{i100}$	27.06	419.43	21.12	280.12	19.76	251.85	-
GSR	27.12	422.54	21.44	285.05	20.25	259.64	11.49
GST	26.58	407.13	21.16	278.89	19.82	253.67	84.05
RND	28.41	414.91	22.13	291.97	20.60	266.30	0.52
UX	29.41	458.77	24.38	310.71	23.03	281.02	< 0.01
PX1	28.25	436.64	22.63	283.81	21.09	251.05	< 0.01
PX2	28.14	436.29	22.59	297.30	21.25	263.14	< 0.01
AL	27.18	379.23	21.62	297.07	20.09	264.81	10.68
P100	27.32	398.89	21.72	295.56	20.21	262.03	7.81
P500	27.36	410.83	21.74	289.97	20.60	265.11	2.16
MXD	27.31	424.95	21.11	275.86	19.82	243.36	80.41
TS	27.30	429.29	22.46	342.85	20.84	311.75	0.16
RS	27.51	490.02	22.24	347.09	20.79	307.92	< 0.01

Table 4.6: Results for the MRCPSP/max-100 instances solved with different parameter settings.

reach values way below 20%.

For a comparison with recent results from the literature refer to table 4.7. It provides a summary of the results for the benchmarks as reported by Barrios et al. [1] for their DGA, which are to our knowledge the best results obtained so far.
It can be seen that the implementation presented in this work is outperformed by the DGA for the small MRCPSP/max-30 instances, but improves the results for the larger MRCPSP/max-50 and MRCPSP/max-100 instances. Note however that the results are not directly comparable because the stop condition of the DGA is a limitation of CPU time. We still may conclude that the procedure implemented in the course of this work is indeed competitive

Instances	CPU time (s)	Deviation
MRCPSP/max-30	1	15.98
MRCPSP/max-30	5	11.61
MRCPSP/max-30	100	10.46
MRCPSP/max-50	4	15.53
MRCPSP/max-100	4	37.31
MRCPSP/max-100	100	22.05

Table 4.7: Results reported for the DGA by Barrios et al. obtained with a 1.4 GHz PC [1].

4.4 Conclusion

The first important conclusion to infer from the tests in this chapter is that a large portion of the scheduling problems that arise in the considered domain is not solvable optimally. Therefore the usage of heuristic optimization algorithms is indeed necessary. This deduction follows directly from the results obtained in the first section of this chapter. Even a sophisticated MIP solver like CPLEX is not able to solve MRCPSP/max instances optimally or often to find a solution at all.
The second section is concerned with the solution of the best mode assignment subproblem that occur in multimodal scheduling problems. We suggested algorithms that do not only take the validity and the duration of a underlying project into account, but also consider the fact that in general a more diverse population leads to better overall results when genetic algorithms are used. It was shown that the methods indeed increase the diversity in the resulting population. This directly results in more variability for the initial population in the second stage of the algorithm.
This second stage is examined in the third section of this chapter. There we present the results of the best configuration of the algorithm that has been found in the course of this work and show that these configurations cannot be improved by changing a single parameter. The following observations may be stated concerning a good parameter and operator selection:

- There are some operators that consistently deliver better results than their alternative. First of all the initialization method using the LST rule to generate the candidate's priority structure is clearly preferable to choosing the priority randomly. The idea to further diversify the initial population did not pay off.
  Also the stochastic universal sampling method seems to be a robust choice for the selection strategy. Neither rank nor tournament selection could increase the quality of the solutions.
  However the strongest impact on performance had the choice of the recombination method. The proposed cycle-based crossover clearly outperformed the X-point methods. The worst choice being the uniform crossover which obtained the weakest results consistently.

- The best choice of the encoding/decoding scheme on the other hand is not an obvious one. The activity list representation performed best in two of the three cases, but the random key representation outperfomed it for the largest instances. Also the choice of the SGS is not trivial. Especially because the mixed scheme achieved good results for the large instances and even the best ones after a medium runtime (5000 generated schedules).

- The same holds for the used BMAP algorithm. Here the steady-state roulette approach inspired by the original DGA performed good, but was outperformed by the generation similarity-tournament algorithm proposed in this work. The similarity-roulette implementation that was meant as a tradeoff between the two could not achieve the same results.

In summary it can be said that the survey showed that the optimization library implemented in the course of this work fulfills the requirements. It is parameterizable in many ways which should allow the user to solve a variety of scheduling problems with different characteristics. It

was also shown, that the results delivered by the library are competitive, compared to the most sophisticated implementations in the literature.

CHAPTER 5

Summary

5.1 Summary

This thesis is concerned with the resolution of scheduling problems that arise in the context of IT automation. The concrete use case at hand is the domain of the UC4 Operations Manager.
The first chapter is started with an introduction of the objects and concepts within the UC4 OM. After that we state the desired result of this work, which is the creation of a software library that is capable of generating high quality solutions for the scheduling problems. Currently this step is executed by hand as the UC4 OM provides only rudimentary mechanisms of automated scheduling. This manual scheduling gets tedious when the controlled procedures reach a certain level of complexity. For these cases it is desirable to move this task to an optimization program. In order to create such an optimization program it is mandatory to formalize the problem domain in a mathematical model. The decision process that resulted in the actual formalization is documented in section 1.3. Together with experts from UC4 the problem domain described in a number of machine scheduling problems from the literature. Since these are not flexible enough to model all the concepts, it was decided to use more complex multi-mode resource-constrained project scheduling problems.
The second chapter provides a thorough mathematical discussion of various versions of this problem, a description of different objective functions to evaluate their performance and a procedure to translate the actual scheduling problems into the mathematical formulation. After that we outline exact and heuristic optimization approaches of the problem from the literature. This research and a preliminary study strongly suggest that only a heuristic optimization method is capable of solving problem instances with the size and complexity that are inherent in a typical UC4 OM system. The decision to use the a genetic algorithm approach is based on the fact that this solution approach is chosen and documented in a number of scientific papers. Also the approach has been proven to be very successful for similar problems. Furthermore the fact that objective functions are relatively flexible is appealing, because in IT environments which often are regulated by legal service level agreements, the measure of quality is not only overall time. It may be a more complex cost function based on missed deadlines.

After the identification of the MRCPSP/max as the most suitable problem formalization, a survey of the concepts for genetic algorithms to tackle this problem is given. Because it is possible to reduce the MRCPSP/max to an MRCPSP instance we also provide an overview of works on this problem. Furthermore do not examine the operators alone, but examine different possibilities for the representation and the population management of such algorithms. Namely we present different approaches to solve the BMAP, introduce a crossover operator based on the cycle-based crossover of Barrios et al. [1] and evaluate the random keys representation.
In the last section we present a number of results obtained in the course of this work. First of all the findings of the preliminary study are shown. These support our decision to discard the idea of using an MIP solver to generate optimal solutions.
After that we present the results for the best mode assignment problem (BMAP) which is a subproblem of the MRCPSP/max. There we evaluate the modifications we proposed in this work with respect to solution quality and diversity. It could be shown that the introduced changes have desirable effects on the obtained solutions.
For the results in the last section a number of runs were executed with benchmarks used in the scientific community. With these standardized test instances we could verify that our implementation is suitable for problem instances showing a variety of different characteristics. This verification is important, because there is no typical UC4 OM scheduling problem since the clients use the software in very different ways. Furthermore these tests allowed us to compare our work with the state of the art implementations, where they proved to be competitive.

5.2 Future Work

To fully exploit the possibilities of the optimization library created in the course of this work, two weaknesses of the UC4 OM have to be challenged. The first one is that runtimes of executed UC4 objects may vary strongly. At this point in time the product predicts these runtimes with rather simple time series approaches using either sliding mean or linear regression. It only provides a simple outlier detection filter which also must be parameterized manually for every object. These runtime predictions are going to be improved by the introduction of more sophisticated regression algorithms. There is data already present in the system which can be used to refine runtime prediction. Run attributes like the host, the user, the presence of certain command line parameters can be used for machine learning algorithms like regression trees or neural networks.
The second topic is concerned with an appropriate representation of hardware resources. Currently it is up to the client to assign a certain amount of resource supply to agents and demand to objects. This task may be executed automatically by running benchmarks or reading out system information from the host machine. Until this information is incorporated in the UC4 system, the library relies on abstract concepts as queues, UC4 resources and restrictions of parallel runs.

Bibliography

[1] A. Barrios, F. Ballestin, and V. Valls. A double genetic algorithm for the mrcpsp/max. *Computers & Operations Research*, 38(1):33 – 43, 2011. Project Management and Scheduling.

[2] P. Brucker. *Scheduling Algorithms*. Springer Publishing Company, Incorporated, 5th edition, 2007.

[3] P. Brucker, S. Knust, A. Schoo, and O. Thiele. A branch and bound algorithm for the resource-constrained project scheduling problem. *European Journal of Operational Research*, 107(2):272 – 288, 1998.

[4] D. Debels, B. De Reyck, R. Leus, and M. Vanhoucke. A hybrid scatter search/electromagnetism meta-heuristic for project scheduling. *European Journal of Operational Research*, 169(2):638 – 653, 2006.

[5] R. W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5:345–, June 1962.

[6] M. Habib, M. Morvan, and J.-X. Rampon. On the calculation of transitive reduction—closure of orders. *Discrete Mathematics*, 111(1-3):289 – 303, 1993.

[7] S. Hartmann. *Project scheduling under limited resources: models, methods, and applications*. Number Nr. 478 in Lecture notes in economics and mathematical systems. Springer, 1999.

[8] S. Hartmann and A. Drexl. Project scheduling with multiple modes: A comparison of exact algorithms. *Networks*, 32(4):283–297, 1998.

[9] S. Hartmann and R. Kolisch. Experimental evaluation of state-of-the-art heuristics for the resource-constrained project scheduling problem. *European Journal of Operational Research*, 127(2):394 – 407, 2000.

[10] R. Heilmann. Resource–constrained project scheduling: a heuristic for the multi–mode case. *OR Spectrum*, 23:335–357, 2001.

[11] R. Heilmann. A branch-and-bound procedure for the multi-mode resource-constrained project scheduling problem with minimum and maximum time lags. *European Journal of Operational Research*, 144(2):348 – 365, 2003.

[12] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI, USA, 1975.

[13] Alcaraz J., Maroto C., and Ruiz R. Solving the multi-mode resource-constrained project scheduling problem with genetic algorithms. *Journal of the Operational Research Society*, 54(6):614–626, 2003.

[14] J. Zimmermann K. Neumann, C. Schwindt. *Project Scheduling with Time Windows and Scarce Resources*. Springer-Verlag Berlin Heidelberg New York, 2002.

[15] R. Kolisch. Serial and parallel resource-constrained project scheduling methods revisited: Theory and computation. *European Journal of Operational Research*, 90(2):320 – 333, 1996.

[16] R. Kolisch and S. Hartmann. Experimental investigation of heuristics for resource-constrained project scheduling: An update. *European Journal of Operational Research*, 174(1):23 – 37, 2006.

[17] R. Kolisch, C. Schwindt, and A. Sprecher. Benchmark instances for project scheduling problems. In *Handbook on Recent Advances in Project Scheduling*, pages 197–212. Kluwer, 1998.

[18] R. Kolisch and A. Sprecher. Psplib - a project scheduling problem library: Or software - orsep operations research software exchange program. *European Journal of Operational Research*, 96(1):205 – 216, 1997.

[19] A. Lova, P. Tormos, M. Cervantes, and F. Barber. An efficient hybrid genetic algorithm for scheduling projects with resource constraints and multiple execution modes. *International Journal of Production Economics*, 117(2):302 – 316, 2009.

[20] V. Van Peteghem and M. Vanhoucke. A genetic algorithm for the preemptive and non-preemptive multi-mode resource-constrained project scheduling problem. *European Journal of Operational Research*, 201(2):409 – 418, 2010.

[21] M. L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer, 2008.

[22] S. Hartmann R. Kolisch. Heuristic algorithms for solving the resource-constrained project scheduling problem: Classification and computational analysis. In J. Weglarz, editor, *Project scheduling: Recent models, algorithms and applications*, page 147–178. Kluwer, 1999.

[23] B. De Reyck and W. Herroelen. The multi-mode resource-constrained project scheduling problem with generalized precedence relations. *European Journal of Operational Research*, 119(2):538 – 556, 1999.

[24] F. Rothlauf. *Representations for genetic and evolutionary algorithms (2. ed.)*. Springer, 2006.

[25] Scott E. Sampson and Elliott N. Weiss. Local search techniques for the generalized resource constrained project scheduling problem. *Naval Research Logistics NRL*, 40(5):665–675, 1993.

[26] E. H. Simpson. Measurement of diversity. *Nature*, 163(4148), 1949.

[27] A. Sprecher, S. Hartmann, and A. Drexl. An exact algorithm for project scheduling with multiple modes. *OR Spectrum*, 19:195–203, 1997.

[28] S. Knust T. Baar, P. Brucker. *Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization*, chapter Tabu-search algorithms and lower bounds for the resource-constrained project scheduling problem, pages 1–8. Kluwer Academic Publishers, Boston, 1998.

[29] V. Valls, S. Quintanilla, and F. Ballestin. Resource-constrained project scheduling: A critical activity reordering heuristic. *European Journal of Operational Research*, 149(2):282 – 301, 2003.

[30] J. Weglarz, J. Józefowska, M. Mika, and G. Waligóra. Project scheduling with finite or infinite number of activity processing modes - a survey. *European Journal of Operational Research*, 208(3):177 – 205, 2011.

[31] G. Zhu, J. F. Bard, and G. Yu. A branch-and-cut procedure for the multimode resource-constrained project-scheduling problem. *INFORMS J. on Computing*, 18(3):377–390, January 2006.