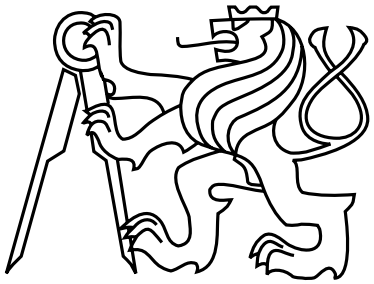
Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science and Engineering



Diploma thesis

Scheduling algorithm for the minimization of setup cost

Bc. Jan Dvořák

Supervisor: Ing. Roman Čapek

Study Programme: Computer engineering

Field of Study: Open Informatics

May 9, 2012

České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra řídicí techniky

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Bc. Jan Dvořák**

Studijní program: Otevřená informatika (magisterský)
Obor: Počítačové inženýrství

Název tématu: **Rozvrhovací algoritmus pro minimalizaci přestaveb strojů**

Pokyny pro vypracování:

1. Seznamte se s problematikou rozvrhování problémů, které zahrnují alternativní výrobní postupy, a dále problémů, v jejichž kritériu se vyskytují časy/ceny přestaveb strojů.
2. Navrhněte datovou reprezentaci problému s alternativními výrobními postupy, zobecněnými relacemi následností a tvrdými omezeními typu "deadline", kde kritériem bude minimalizace celkové doby/ceny přestaveb.
3. Navrhněte a implementujte heuristický algoritmus pro zmíněný problém v jazyce C#. Při návrhu se zaměřte především na časovou náročnost algoritmu při řešení velkých instancí (řádově 1000 úloh).
4. Navržené řešení pečlivě zdokumentujte. Dokumentace musí obsahovat rozbor související literatury, popis problému, popis návrhu a implementace algoritmu a jeho srovnání s existujícími přístupy pro podobné problémy.

Seznam odborné literatury:

[1] Allahverdi, A., Ng, C., Cheng, T., Kovalyov, M. Y., 2008. A survey of scheduling problems with setup times or costs. European Journal of Operational Research 187, pp. 985–1032.
[2] Herrolen, W., Demuelemeester, E., Reyck, B. D., 1997. A classification scheme for project scheduling problems. Katholieke Universiteit Leuven, pp. 1–25.
[3] Brucker, P., Drexl, A., Möhring, R., Neumann, K., Pesch, E., 1999. Resource constrained project scheduling: Notation, classification, models, and methods. European Journal of Operational Research 112, pp. 3–41.

Vedoucí: Ing. Roman Čapek

Platnost zadání: do konce letního semestru 2012/2013

prof. Ing. Michael Šebek, DrSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 14. 11. 2011

Aknowledgements

My thanks go to all those who supported me during the work on this thesis and not just the people who advised me, but also those who supported emotionally and mentally. Especially, I would like to thank my supervisor Ing. Roman Čapek whose help was the most crucial and appreciated and who always adviced me willingly.

Declaration

I hereby declare that I have completed this thesis independently and that I have listed all the literature and publications used.
I have no objection to usage of this work in compliance with the act §60 Zákon č. 121/2000Sb. (copyright law), and with the rights connected with the copyright act including the changes in the act.

In Vozerovice on May 7, 2012 .....

Abstract

This diploma thesis is dedicated to the scheduling problem that is motivated by real manufacture processes. It is a scheduling with alternative process plans subject to the total setup costs minimization, where setup times are sequent-dependent. The considered problem further covers many nowadays and practical constraints such as non-unary resources, release times and deadlines or generalized precedence relations known as time lags. This thesis also includes very detailed review of related works. An exact mathematical model is described, which can be used to solve small instances and a new heuristic algorithm is proposed to solve large instances of the considered problem. The algorithm is then implemented using programming language C# with an emphasis on efficiency and low computational demands. In the end, the proposed algorithm is tested on the wide variety of instances with regard to the time performance and the value of the objective function comparison with existing algorithms and benchmarks.

Abstrakt

Tato diplomová práce je věnována rozvrhovacímu problému, který je motivován reálnou výrobou. Jedná se o rozvrhování s alternativními výrobními postupy s cílem minimalizovat náklady spojené s přestavbou strojů, které jsou navíc závislé na pořadí úloh v rozvrhu. Daný problém dále zahrnuje mnohá praktická omezení jako jsou neunární zdroje, release time a deadline, nebo zobecněné relace následností (time lags). Součástí této práce je i detailní studie podobných problémů. Pro daný problém je uveden exaktní matematický popis, pomocí kterého lze řešit jeho malé instance, a dále pak návrh nového heuristického algoritmu pro řešení velkých instancí. Popsaný heuristický algoritmus je dále implementován pomocí programovacího jazyka C# s cílem efektivně řešit velké instance a to v krátkém čase. Tento algoritmus je otestován na široké paletě instancí, včetně porovnání s jinými algoritmy a již existujícími benchmarky.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

this thesis is dedicated to the problem of minimizing the total setup costs in manufacture and others processes. The goal is to formulate the model of the problem and implement an algorithm to solve the considered problem. An algorithm should be able to solve large instances of the problem such that the schedules have desired property, i.e. with the minimal total setup costs. In this thesis, total setup costs consist of all performed weighted setup times. The considered problem can be classified as the resource constraint project scheduling problem with alternative process plans and sequence-dependent setup times.

Scheduling can be seen as a discipline dealing with assigning activities to resources over the time. The scheduling theory is crucial for each sector in which there is a need to create a schedule by selecting, assigning or concatenating activities that are performed on some kind of resources. Scheduling is an essential part of many real manufacturing processes and services that we daily use and do not even apprehend it (from mobile phones, modern cars, desktops, medicine to production management). Nowadays, scheduling plays more and more important role. Ensuring the optimality or balance of manufacturing processes, services or just order of activities can bring a significant cost reduction. This way, the companies can reduce production costs, produce more goods or have higher profit. The goal of the scheduling is to find a feasible solution of the given problem and moreover, the solution is often required to have a desired property (e.g. the minimum possible duration).

The theory of scheduling is a part of combinatorial optimization which focuses primary on discrete solution of problems. Combinatorial optimization is often applied in algorithm theory, operational research, machine learning, artificial intelligence, etc. Combinatorial optimization is therefore applied when we need discrete values, for example we need to produce a chair, not only 90% of it. These demands makes finding the solution much more difficult. Nevertheless, many real problems are in essence discrete or further indivisible.

Scheduling, or combinatorial optimization in general, is a continuously evolving discipline. Over the years together with rapid technology growth, these disciplines experienced a dramatic progress and became necessary in many very different sectors. Scheduling is not used only for academic purposes, but it is being widely used in practice as well.

There are many approaches to solve scheduling problems, but not each approach is suitable for all problems or does not cover other problems at all. Moreover, some problem can be handled by more different approaches. Choosing the right approach is often not easy and we must consider many variables. Further, we often have to make compromise between quality of the solution and quantity of computational time necessary to create a desired schedule. All these aspects strongly depend on the given problem and must be considered before choosing the approach or algorithm.

1.1 Motivation

Using the scheduling theory, we can optimize manufacturing processes and thus reduce processing or waiting time, storage requirements or increase number of products, etc. There is a large amount of very different problems from many areas dealing with the scheduling and this area is still growing. Without proper schedule, production lines, machines, people, etc. might not be fully utilized, which can cause unnecessary loss. By modeling manufacturing processes, we can even try more scenarios and choose the most suitable one, for example: buy new machine, hire more people, change product priorities, etc. In real process, there is often no place, time or money for such experiments. Moreover, even improvement of only few percentages can change the company's position in the market.

The problem considered in this thesis is motivated by the real application and up to our knowledge there is not existing solution approach for such problem. Therefore, a new approach and algorithm is given to solve the considered problem. Moreover, proposed algorithm is designed for large instances of the problem. Some parts of proposed solution are based on existing approaches, others are newly created.

The goal is to optimize manufacture process from the perspective of sequence-dependent setup times, that are often ignored in other approaches even thought they can play an important role. Setup times can cover time necessary to change tools, reconfigure machine etc. Notice, that during this time period, no work on the resources (machines, etc.) is done, which can cause the entire process flow to be ineffective. Minimizing the setup costs, the machines are more utilized and therefore time and costs are saved. Such minimization can significantly improve whole manufacturing process, especially in cases where these setup times take considerable part of product processing time.

The main motivation of this research is the production in the printing company. In the production, there is one universal paper cutting machine, one collating machine, four semi-automatic perforating machines, two full-automatic perforating machines, two twin wire presses and finally two machines, that combine both perforating and pressing functions. Now, lets describe the production of a simple ring-bound workbook. First, we need to cut papers and folder on the paper cutting machine. For different kind of paper (cardboard) we must use different paper cutting knives. Not to waste time while changing knives, we can cut papers for more workbooks at the same time. After all parts are cut, we must collate folder and paper in the right order. Than, perforating is done. Perforating can be processed on semi-automatic or full-automatic perforating machines. Finally, prepared workbook is put in the press and all the sheets are bound together using twin-wire machine or manually. Moreover, last two steps can be processed using single universal machine.

All the machines are shared among all the manufacturing processes that do not concern only workbooks. Furthermore, automatic machines are often limited by the size of paper and count of sheets, so using them is not always possible or they are used in other processes. With the limited number of machines and need to reconfigure machines for each part of a product, the processes can be ineffective and there might be significant losses. Moreover, the setup

times are sequence-dependent. Sequence-dependent setup times describe the differences in time required to reconfigure or set machines depending on the order of operations. Proper schedule can rapidly improve manufacturing process and reduce the losses. Figure 1.1 shows simplified manufacturing process of the mentioned ring-bound workbook. The goal is to choose only one of the process plans to complete the workbook.

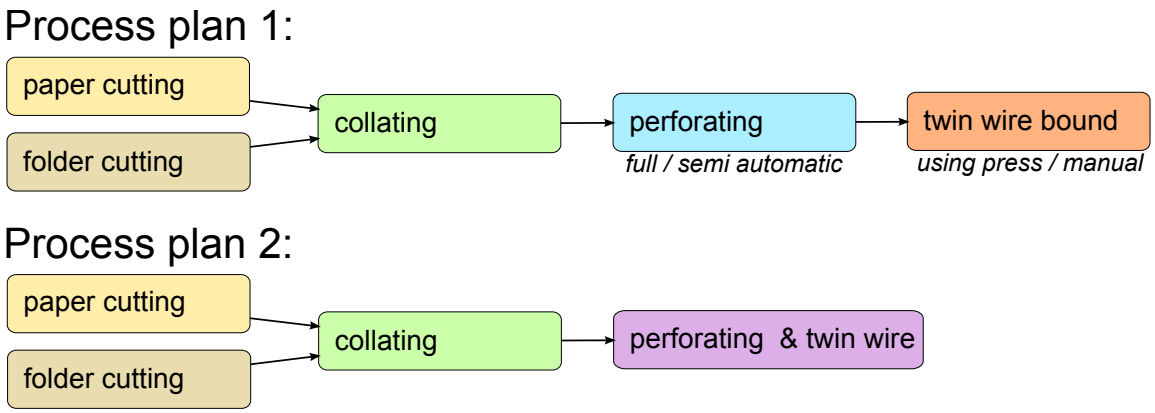


Figure 1.1: Manufacture process of ring-bound workbook

1.2 Contribution

This thesis deals with minimizing the total setup costs in manufacture processes. Setup costs are often neglected among other scheduling problems, that can end up with loss of time and money. Furthermore, the problem considered in this thesis was not dealt before and is based on real application.

The main contributions of this thesis are the mathematical formulation of the problem and proposal of the new heuristic algorithm to solve the considered problem. The proposed algorithm is designed to effectively and fast enough solve very large instances of the problem that consist of 1000 activities.

The considered problem is very extensive, not only from the perspective of all constraints, but also from the perspective of the objective function. The problem includes alternative process plans, which meet nowadays processes and enable us to create more flexible models. Furthermore, sequence-dependent setup times are considered in the objective function. Us-

ing setup times, we can better utilize machines - we "sell" the machinery time. Nowadays, setup times become to play more and more important role in real manufacture processes. Moreover, many real constraints are considered to meet current production demands. Deadlines are used to model latest finish time of a product. Non-negative time lags can be used to define precedence constraints or specify necessary time interval between two phases of the manufacture process. Finally, non-unary resources are used to model situations with more machines of the same kind. With more resources of the same kind, we can process more similar operations in parallel and therefore increase throughput of the whole process.

Using the proposed algorithm we can significantly reduce process costs, particularly in areas, where the production process was scheduled "by hand" or by less sophisticated approaches. The considered problem is very complex and it can therefore cover many for quality of the solution. Considering all the demands, we can produce more effective schedule than by using less sophisticated approaches, where we have to omit some demands either from the perspective of constraints or the solution quality.

1.3 Outline

This diploma thesis consists of eight chapters. The first chapter includes motivation, introduction to the considered problem and contributions. Chapter 2 is dedicated to the detailed review of related works, including solution approaches and techniques. Related works are further categorized due to the view of similarity with the considered problem. Following chapter deals with the problem statement from scheduling basics, over the general description, up to formal definition and classification. Chapter 4 is dedicated to the mathematical model describing the considered problem using mixed integer linear programming notation (MILP). Next chapter describes the proposed heuristic algorithm to solve large instances of the problem in more detail. Chapter 6 then presents implementation details of the proposed heuristic algorithm. Chapter 7 is dedicated to the performance evaluation. The proposed algorithm is compared with existing approaches on both random and standard instances. The last chapter presents a summary of this diploma thesis and future chalenges.

Chapter 2

Related works

This chapter is dedicated to the literature review. The problem considered in this thesis can be classified as the resource constrained project scheduling problem (RCPSP) with none-negative time lags and sequence-dependent setup times where activities are subject to release times and deadlines. Furthermore, we also consider an extension of the RCPSP by alternative process plans. The goal is to minimize the total setup costs. Since the problem considered in this thesis is rather extensive, we will separate the literature review into more parts. First, a review of the RCPSP will be given, then the related works for alternative process plans will be stated. Finally, a review of problems with the total setup costs in the criterion will be given and setup times will be discussed in general.

2.1 Resource constrained project scheduling problem

The RCPSP can be defined, according to Kadrou and M.Najid [24], as a set of activities with a specific skill requirements that has to be processed on a particular work center with limited capacity. Authors also presented an approach to estimate lower bound and a dominance concept based heuristic to solve RCPSP with multiple execution modes. Blazewicz et al. [8] dealt with wide range of scheduling problems in general. Authors also considered RCPSP, categorized it's extensions and showed a mathematical model for each extension.

Brucker et al. [9] proposed notation and classification for many problems from the RCPSP area and presented a review with many different methods to solve RCPSP.

Another slightly different classification scheme for the RCPSP problems was published by Herroelen et al. [23]. Neumann et al. [35] focused on project scheduling with temporal constraints such as time lags, time windows, etc. Authors also proposed RCPSP formulation for the problem with makespan minimization as a criterion.

Multi-mode resource constrained project scheduling problem (MRCPSP) is a generalization of RCPSP. The main difference, compared to RCPSP, lays in the definition of more execution modes for each activity. A mode is defined by an activity's processing time and resource demand. Neumann et al. [35] formulated a mathematical model and a general algorithm scheme for the MRCPSP problem. Reyck and Herroelen [36] proposed a local search based methodology for MRCPSP with generalized precedence constraints with objective to minimize the project duration. Authors discussed many solution strategies such as a tabu search, local search, reducing search using preprocessing, etc. Salewski et al. [38] focused on MRCPSP with mode identity constraints, release times and deadlines. Authors proposed a mathematical model and parallel randomized solution heuristic. Van Peteghem and Vanhoucke [41] dealt with MRCPSP and used resource scarceness characteristics. Authors developed a scatter search algorithm (population-based meta-heuristic) to solve the problem. Wang and Fang [43] proposed an estimation distribution algorithm heuristic for MRCPSP with makespan minimization as a criterion. Deblaere et al. [14] proposed and evaluated an exact scheduling procedure based on Branch & Bound algorithm and also proposed a tabu search heuristic for MRCPSP with criterion to minimize project duration.

In this thesis, we also consider non-negative time lags that represent a generalization of precedence constraints. Main advantage of the generalized precedence constraints lies in the definition of minimal time interval between two activities. In case of classical precedence constraint, this interval is always equal to activity's processing time. Therefore, time lags allow us to build more complex models of relations among activities.

Brucker and Knust [11] dealt with a single-machine scheduling problem considering positive finish-start time lags and release times and they proved an existence of the polynomial time algorithm for the problem. Furthermore, authors presented reduction of many prob-

lems (e.g. preemptive parallel machine problems to unit-time jobs and constant time lags). For more general problem, with positive and negative time lags, release times and deadlines, Brucker et al. [10] presented a Branch & Bound algorithm to minimize the makespan. Lombardi and Milano [32] published an algorithm for problems with both positive and negative time lags, release times and deadlines on non-unary resources where each activity uses a part of resource capacity. The proposed algorithm is used to minimize makespan and it is based on precedence constraint posting approach. Ballestín et al. [3] also dealt with RCPSP problem with minimal and maximal time lags. Authors proposed an evolutionary algorithm built on conglomerate-based crossover operator. The objective is the minimization of makespan. Hamdi and Loukil [21] focused on permutation flowshop problem with minimal and maximal time lags with minimization of makespan as a criterion. Authors stated a mathematical formulation and proposed a genetic algorithm.

2.2 Alternative process plans

The formalism of alternative process plan is a generalization of multi-mode behavior of activities in the MRCPSP problem. It allows us to model more than one way how to complete projects, while not only resource demands, but also number of activities, precedence relations, etc. can differ among alternative process plans.

Barták and Čepek [4], [5] defined a structure called Nested temporal network with alternatives (NTNA) to model alternative process plans. Authors also developed an algorithm to recognize such structure. Generally, NTNA are an acyclic graph where nodes represent activities and edges correspond temporal and branchings constraints. Branching constraint can be of two types - parallel and alternative branching. Authors focused on the modeling the problems with alternative process plans, but there is no scheduling algorithm for such problems. Beck and Fox [6], [7] formulated a constraint-based representation of alternatives, described and compared extensions of different heuristics for solving problem with alternatives. Čapek et al. [42] dealt with RCPSP extended by alternative process plans and sequence-dependent setup times. Authors presented an integer linear programming (ILP) model for exact solution of small instances and a heuristic called iterative resource scheduling with alternatives (IRSA) for larger ones. For the representation of alternative process

plans, authors used a Petri net model. Kis [25] proposed three algorithms for the jobshop problem with processing alternatives. First, a tabu search heuristic, second, a genetic algorithm with crossover operator only and finally, a randomize algorithm for comparison. Leung et al. [30] focused on RCPSP with alternatives that is close to the jobshop problem. Authors proposed an agent based ant colony optimization to minimize the makespan. For the same problem, Li et al. [31] published an agent based evolutionary algorithm. Shao et al. [39] presented an integration model of process planning and scheduling problems which are carried out simultaneously. Authors developed a genetic algorithm to minimize the production time. Choi and Choi [13] dealt with the jobshop problem considering release times and sequence-dependent setup times. Authors' solution is based on a local search with many selection rules (e.g. earliest complete time first). A dispatching rule is used to estimate upper bound.

2.3 Objective function

The objective function considered in this thesis is the minimization of the total setup costs (TSC). The problem of scheduling with setup times, more precisely sequence-dependent setup times, is a part of many real optimization problems and it has been investigated in several studies. The term setup time denotes amount of time between two activities scheduled consequently on the same resource needed to change tools, configuration, etc. Sequence-dependency states different amount of time for each pair of activities. Total setup costs are then equal to the sum of setup costs resulting from all performed setups. Allahverdi et al. [2] dealt with setup times in general and published a survey in which many different problems related to setup times are summarized. Authors also reported solution approaches in the review and furthermore they proposed a notation for all of these problems. Yuan et al. [45] published a study for a metal casting company concerning the minimization of total setup costs in which authors demonstrate the importance of handling setup times by calculating company's savings. Nevertheless the most of publications that consider setup times (or sequence-dependent extension) do not include them into criterion function. Therefore, this section is further divided into two parts - problems with setup times in criterion and problems, where the setup times are considered only as a constraint.

2.3.1 Setup times in criterion

Foccaci et al. [18] dealt with a problem similar to the problem considered in this thesis. Authors proposed a two phase pareto heuristic for the problem with precedence constraints, release times and deadlines. The criterion is to minimize the total setup costs and makespan. In the first phase, makespan is minimized. In the second phase, the total setup costs are minimized, while makespan is not allowed to get worst. Geoffrion and Graves [20] also dealt with a problem similar to the problem considered in this thesis. Geoffrion and Graves focused on the problem with both release times and deadlines while minimizing total setup costs on parallel machines. Authors implemented quadratic assignment algorithm with linear programming adjustment. Wang and Wand [44] focused on single machine earliness tardiness problem with sequence-dependent setup times. Objective function consists of total setup time, earliness and tardiness minimization. Authors proposed a heuristic for hybrid genetic algorithm to find the solution. Mirabi [34] proposed hybrid simulated annealing algorithm for single machine problem with sequence dependent setup times. The objective function is given by the sum of setup costs, delay costs and holding costs.

2.3.2 Other problems with setup times

Finally, we state the problems where setup times play an important role as a constraint, but they are not considered in the objective function. Akkiraju et al. [1] proposed an agent based heuristic called asynchronous team architecture for multiple non-identical machines scheduling problem with sequence-dependent setup times. The nature of their approach lies in generating many initial solutions. These solutions are then further improved. Finally, the best one is chosen. Authors' objective is to minimize the total weighted tardiness and earliness. Mika et al. [33] focused on multi - mode resource constrained scheduling problem with sequence-dependent setup times. For this problem, authors proposed a tabu search heuristic with minimization of the makespan as a criterion. Kopanos et al. [27] dealt with planning in food industry. Authors proposed a mathematical model describing all the links between activities in food processing plans with criterion to minimize the total costs that includes inventory, operating, utilization costs, etc. Krüger and Schol [28] published a heuristic solution framework for multi-project scheduling problem with sequence-dependent

transfer times. These times are applied when a resource is transferred from one project to another. The objective is to minimize the project duration. Drießel and Moench [16] developed a variable neighborhood search heuristic to minimize the total weighted tardiness on parallel identical machines while considering precedence constraints and release times. Gacias et al. [19] focused on similar problem as Drießel and Moench. Authors proposed an exact Branch & Bound algorithm for small instances of the problem and a local search heuristic for larger ones. Authors targeted the minimization of makespan and maximum lateness. Lee and Pinedo [29] proposed a three phase heuristic based on apparent tardiness cost with setups (ATCS) rule for problems on parallel identical machines. Authors' objective was to minimize the total weighted tardiness. Ruiz and Stützle [37] dealt with sequence-dependent setup times flow shop problem with makespan and weighted tardiness objectives. Authors developed an iterated greedy heuristic based on NEH heuristic. Tasgetiren et al. [40] published an article concerning with single machine problem with sequence-dependent setup times. Authors proposed a discrete differential algorithm to minimize total weighted tardiness. This algorithm further uses referenced local search for insertions.

Chapter 3

Problem statement

With increasing importance of scheduling theory and its application in manufacturing processes, controlling, etc., there are also more and more complex problems. Covering all demands and constraints if often very difficult. Using algorithms to solve such problems brings not only simplification of work to production management. Correctly designed algorithms often give us much better results in much shorter time than using manpower.

In the following sections we will closely describe scheduling basics, its complexity and present general description of the problem considered in this thesis. Further, formal definition and classification of the considered problem is given.

3.1 Scheduling basics

The scheduling problem is often specified by a set of activities, resources, constraints and the objective function. Activity (operation, task) represents a real work that needs to be done (assembly, calculation, transfer of information, etc.). Each activity is specified by its processing time and can have other properties such as resource demands, that determine which of resource types can process the activity, or release time and deadline, that specify a time interval in which an activity can be processed, etc.

Resource can be understood as any device such as processor, machine, manpower, money, energy, conduit etc. on which activities are performed (executed). Each resource is of some type and can have non-unary capacity. This means, that more activities can be processed simultaneously (in parallel) on the same resource. Assigning activities to the resources therefore stands for consuming a part of resources capacity. Every scheduling problem can include many different kinds of both activities and resources with a wide variety of constraints. This possible diversity therefore determines the complexity of finding the solutions for these problems.

There are many types of scheduling constraints that have to be fulfilled to obtain a feasible schedule. Such constraints depend on each specific problem, but there are also some general constraints, according to Blazewicz et al. [8]:

- each activity can be processed at most by one resource at a time
- each resource can process at most one activity at a time
- activities cannot be preempted

Furthermore, there are additional constrains used to describe more complex relations in the schedule such as:

- precedence constraints
- setup times

Since nowadays scheduling problems are more and more extensive, we often need to relax some of these constraints, for example:

- activity can be processed in parallel on more resources - *multi-resource activities*
- resources can process more activities at a time - *non-unary capacities*

In this thesis, some of general constraints are relaxed and other additional constraints are used to describe the considered problem (see section 3.3 for more details).

3.2 Problem complexity and solution approaches

The most of scheduling problems belong to the $\mathcal{NP}$–*hard* class, meaning that these problems cannot be solved with polynomial time complexity. Only a small amount of scheduling problems are optimally solvable in polynomial time. Since the computational time to find an optimal solution can grow exponentially, or even worse, we often relax the demand on the optimal solution and we are searching for the solution that is close to the optimal instead. With this relaxation, problems can be solved using much less computational efforts. This is a compromise between the quality of the solution and the time we are willing to spend in finding such solution. Some of the algorithms, r - approximate ones, guarantee the maximal deviation from the optimum, but often we cannot ensure such property.

There are many different approaches to solve scheduling problems. We can divide solution approaches into two main categories - exact algorithms and heuristics. Exact solutions are commonly obtained by using an *integer linear programming* (ILP) approach, constraint programming or by branching schemes. Although these methods ensure optimality of the solution, they are used mainly for small instances of problems because of computational requirements, which usually grow exponentially. Therefore, for larger instances, we often use heuristic algorithms, which do not ensure optimality of the solution (or even finding of the feasible solution), but the computational complexity of such algorithms is much lower. Heuristic can be defined as an algorithm that uses some kind of estimation approach to find a solution. This estimation approach is specific for each problem. For example, for the problem considering the minimal project duration, we can choose activities with the smallest release times first, etc. Heuristics are commonly used for larger instances of problems, where finding an optimal solution would consume more time that we are willing to spare. There are many different basic ideas behind heuristics, many of them are inspired by the nature (genetic algorithms, ant colony algorithms, etc.). But not every approach is suitable for each problem. Choosing the right approach can significantly determine the quality of its outcome. We can say, that usually there is a common approach for a group of very similar problems, for example, problems with setup times are often solved by heuristics based on local search methods. We can use any of approaches, but often it might be for the best, if we base our approach on the related works.

Since the problem considered in this thesis is a generalization of resource constrained project scheduling problem (RCPSP) which belongs into $\mathcal{NP}-hard$ (proof was published by Demeulemeester and Herroelen [15]), this problem belongs to this class as well. Therefore, the ILP model for small instances and the heuristic algorithm for bigger instances of the problem are proposed.

3.3 General description

In this thesis, we consider quite extensive problem that can be classified as a *resource constraint project scheduling problem* (RCPSP) with *alternative process plans*, where resources have non-unary capacities. The objective is the minimization of the total setup costs (TSC). Relations between activities are described using precedence constraints and non-negative time lags. This problem is further extended by sequence-dependent setup times, release times and deadlines.

Nowadays, there are more and more demands to optimization processes. Some of these demands also include parallel and alternative manufacture. Parallel processes are carried out simultaneously, e.g. a factory has more that just one production line. But there is also need to consider alternative manufacture. Typically, there are more than just one way how to complete a product, for example, the product can be processed on one fully automated, universal machine or on more simpler machines operated by a worker. Therefore, alternative process plans, which are used to describe such possibilities, are taken into consideration. More and more often, time that is necessary to change tools, prepare stocks, change configuration of machines, etc. is considered in production processes. This amount of time is called setup time. In this thesis, setup times are sequence-dependent, meaning that for each combination of activities this amount of time can differ. The total setup costs minimization stands for finding such ordering of activities on resources, for which the sum of all considered setup costs is minimized.

In the production (and other) processes, many activities cannot be executed in arbitrary time, since for example material or other additional resource is not available at time zero. Therefore, we consider release times of activities to model the earliest possible time from

which an activity can be processed to model such demands. We also need to specify the latest time in which the product must be finished. Deadline stands for the latest possible completion time of an activity. Another typical restriction are precedence constraints that allow us to model situations, where it is not possible to start an operation of a product, till its previous phase is completed. Precedence constraints can be further extended to so called time lags. Using time lags, more precisely start to start non-negative time lags, we can specify the minimal necessary time interval between start times of two activities. By this constraint, we can model situations where we can start another operation on the same product before an actual operation is fully completed.

3.4 Formal definition and classification

In the following paragraphs, the detailed classification and description of the considered problem is given. The problem can be defined using sets of activities, resources and constraints together with an optimality criterion. Let $\mathcal{A} = \{1 \dots n\}$ be a set of n activities. Further, let $\mathcal{A}_\mathcal{E} = \mathcal{A} \cup \{0, n+1\}$ be an extended set of activities, where activities 0 and $n+1$ restrict the whole project. Activity 0 indicates the project's start and activity $n+1$ its end. To represent a set of m resource types we define $\mathcal{R} = \{\mathrm{R}_1 \dots \mathrm{R}_m\}$. Each resource type $\mathrm{R_q} \in \mathcal{R}$ has a capacity $\theta_q \geq 1$. Each activity i has following parameters: processing time $p_i \geq 0$, release time $r_i \geq 0$, deadline $\tilde{d}_i \geq 0$ and the resource demand $\mathrm{R}_i^q > 0$ for one resource type $\mathrm{R}_q \in \mathcal{R}$. In this thesis, only mono-resource activities are considered, meaning that each activity demands at most one type of resource. In other words, any activity cannot be processed simultaneously on more than one resource type.

We consider sequence-dependent setup times $st_{ij} \geq 0$ for all pairs of activities $(i, j) \in \mathcal{A}^2$. The term setup time, or sequence-dependent setup time, st_{ij} denotes a minimal time between completion time of activity i and start time of activity j while both activities i and j share at least one resource part. Sequence-dependent setup times are therefore taken into consideration if and only if both activities i and j are scheduled subsequently on the same resource type and they share at least one unit of its resource capacity.

Non-negative start to start time lags l_{ij} for all $(i,j) \in \mathcal{A}^2$ are defined for all precedence constrained activities. Such time lags denote a minimal amount of time between start times of activities i and j. Compared to sequence-dependent setup times, time lags have no resource restriction. Time lags allow us to model more complex relations between activities than using classical precedence constraints only.

Alternative process plans are modeled using nested temporal networks with alternatives (NTNA) presented by Barták and Čepek [4]. The goal of the scheduling with alternative process plans lies in the selection exactly one of the process plans for each product and assigning the selected activities to the resources in time. Each process plan consists of activities and their temporal constraints. In this thesis, NTNA are used to describe alternative process plans and time lags (see section 3.5 for more details).

The goal of the scheduling process is to find a feasible solution, if exists, with the best possible value of the objective function. In our case, we select subset $\mathcal{A}^S \subseteq \mathcal{A}$ of all the activities to be scheduled, according to the definition of alternative process plans, while all the constraints are fulfilled and the total setup costs are minimized. To represent the schedule, the following variables are used: $s_i \in \mathbb{R}_0^+$, $v_i \in \{0,1\}$, $f_{ij} \in \{0,1\}$ and $z_{ivk} \in \{0,1\}$. Variable s_i denotes the start time of activity $i \in \mathcal{A}$, v_i determines whether activity i is selected ($v_i = 1$) or rejected ($v_i = 0$). If activities i and j are scheduled subsequently on the same resource type and they share at least one unit of its resource capacity, then $f_{ij} = 1$; $f_{ij} = 0$ otherwise. Finally, if $z_{ivk} = 1$ then activity i is scheduled on the resource part v of resource type k, meaning that activity consumes specific part of resource's capacity, $z_{ivk} = 0$ otherwise.

The total setup costs consist of all performed weighted setup times. This means, that only setup times of subsequently scheduled activities, which share a part of the same resource capacity, are taken into consideration. These setup times are further multiplied by the specific costs. The total setup costs are thus given by the sum of all applied weighted setup times. With variables mentioned above, we can formulate the objective function as $TSC = \sum f_{ij} st_{ij} c_{ij}$ for $\forall (i,j) \in \mathcal{A}^S$, where c_{ij} is the cost of setup time between activities i and j.

The problem considered in this thesis can be classified using an extended notation pro-

posed by Brucker et al. [9] (3.1) or using an extended notation proposed by Herroelen et al. [23] (3.2).

$$PS|nestedAlt, l_{ij}^{min}, ST_{SD}, r_j, \tilde{d}_j|TSC \tag{3.1}$$

$$m1|nestedAlt, min, ST_{SD}, r_j, \tilde{d}_j|TSC \tag{3.2}$$

Both notations are extended by terms $nestedAlt$ to denote alternative process plans, according to Čapek et al. [42] and sequence-dependent setup times ST_{SD} and total setup costs TSC according to Allahverdi et al. [2]. The term PS stands for the project scheduling, $m1$ for m renewable resources, min and l_{ij}^{min} for the minimal start to start time lags, r_j for release times and finally $\tilde{d}_j$ for deadlines.

3.5 Nested temporal networks with alternatives

To represent alternative process plans, the formalism of nested temporal networks with alternatives (NTNA) is used in this thesis. NTNA presented by Barták and Čepek [4], [5] form a special case of temporal networks with alternatives (TNA). Compared to TNA, solving problems described by NTNA can significantly reduce computational demands. For example the assignment problem for instances represented by TNA is NP problem. On the contrary, the assignment problem for instances represented by NTNA are solvable in polynomial time. An example of NTNA are depicted in Figure 3.1 and the following paragraphs provide a description of the NTNA formalism.

3.5.1 Formal definition

NTNA are directed acyclic graph $G = (V, E)$, where each node $i \in V$ corresponds with activity $i \in \mathcal{A}_\mathcal{E}$ and each edge $e = (x, y) \in E$ represents one temporal constraint in the form of non-negative start to start time lag (precedence constraints are specific case of time lags).

To represent parallel and alternative branchings, two labels are further defined for each node $i \in V$: in label $inLabel_i$ and out label $outLabel_i$. Those labels can take values

$\{PAR, ALT, \emptyset\}$. PAR stands for parallel branching, ALT for alternative branching and finally, if there is no branching, θ is applied.

There are two possible kinds of branchings in nodes - input and output. Node i has output branching ($outLabel_i \neq \theta$) if $\delta^+(i) > 1$, where $\delta^+(i)$ is the out degree of node i. This means, that node i has more than one direct successor. For such node i, there is the corresponding node j which has input branching ($inLabel_j \neq \theta$) with $\delta^-(j) > 1$, where $\delta^-(j)$ is the in degree of node j. This means, that node j has more than one direct predecessor. Then a sub-graph beginning with direct successor of node i and finishing with corresponding direct predecessor of node j is called *branch*. Number of branches is thus equal to the output (input) degree of node i (j), since in our approach $\delta^+(i) = \delta^-(j)$. Such part of the graph, that begins with node i and finishes with node j is called *nested sub-graph*.

In the following subsections more detailed description is given. For better clarity, an example from the area of printing products used in introduction is transformed into NTNA instance and is further used to demonstrate the problem.

3.5.2 Alternative branching

Alternative branching is used to describe more ways how to complete a part of a product. Alternative branching begins with node i, whose $outLabel_i = ALT$ (see node 5 in Figure 3.1). For each such node i there exists node j, whose $inLabel_j = ALT$, such that activity j finishes alternative branching (see node 13 in Figure 3.1). Activities i and j form a pair that denotes nested sub-graph (alternative branching) with more alternative branches (see nodes 9 and 12 in Figure 3.1). Only one of those alternative branches has to be selected. Note, that alternative branching can be nested in another alternative branching or in parallel branching.

3.5.3 Parallel branching

Parallel branching is used to describe situations, where more activities can be processed simultaneously. Parallel branching begins with node i, whose $outLabel_i = PAR$ (see node 3 in Figure 3.1) and finishes with node j (see node 14 in Figure 3.1), whose $inLabel_j = PAR$.

In case of nodes with parallel branching, all of branches between activities i and j have to be selected or rejected together. Similarly as in case of alternative branchings, parallel branchings can be nested one in another, etc.

3.5.4 Temporal constraints

Using NTNA, we can also describe temporal constraints. Non-negative start to start time lags are represented by edges between nodes (see edge $e_{8,13}$ in Figure 3.1). The time interval specified by time lag l_{ij} determines the minimal and maximal difference between start times of activities i and j. Since precedence constraints are specific case of time lags, we can model them using NTNA as well. Time lags are not defined for each pair of activities but for precedence constrained only, meaning that time lags are defined only for pairs of activities with logical connections. In the context of this thesis, only lower bound of time lag (see a in Figure 3.1) is specified. The upper bound (see b in Figure 3.1) is equal to ∞.

3.5.5 Nested graphs

Nested temporal networks with alternatives form a special case of TNA, where the graph is composed from either parallel or alternative branchings. These branchings can be further nested on in another. NTNA therefore consist of sub-graphs, where each sub-graph is a parallel or alternative branching. Each sub-graph is delimited by its beginning and finishing node. There is no interleaving among sub-graphs but the case that one sub-graph is nested in another. The basic idea lies in a fact, that in real processes, some parts of a process form a closed sub-parts that can be seen as one bigger processing step from the point of view of the whole process. Barták and Čepek [4] proposed a polynomial-time algorithm to recognize nested networks as well as a simple scheme how to construct such structures. The construction of NTNA are based on replacing of an arbitrary node by either parallel or alternative branching, or by a sequence of activities. Using such procedure, nested form is kept in each step of the construction. An example of NTNA is depicted in Figure 3.1.

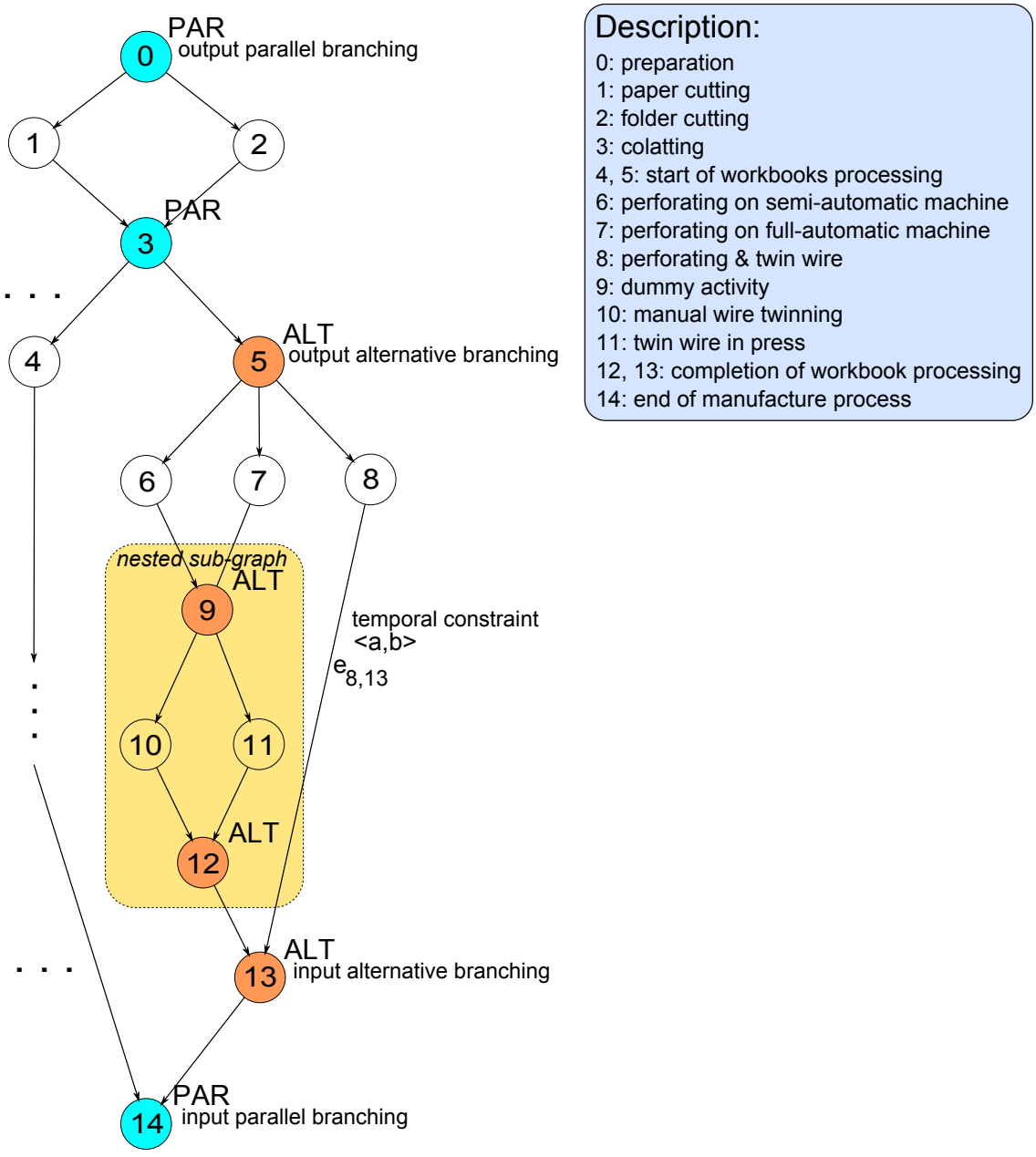


Figure 3.1: Nested temporal networks with alternatives - example

Chapter 4

Mixed integer linear programming model

Integer linear programming (ILP), as an specific case of MILP, is a mathematical technique commonly used, among others, to model and solve optimization problems. Using ILP, we can model all the constraints and objective function as well. ILP is thus a very strong modeling tool.

ILP is given by matrix $\mathcal{A} \in \mathbb{R}^{m \times n}$ and vectors $\mathbf{b} \in \mathbb{R}^m$ and $\mathbf{c} \in \mathbb{R}^n$. The goal is to find vector $\mathbf{x} \in \mathbb{Z}^n$ which satisfies the inequality $\mathcal{A} \cdot \mathbf{x} \leq \mathbf{b}$ such that $\mathbf{c}^T \cdot \mathbf{x}$ is maximal. In general, the goal is to find such values of variables (vector $\mathbf{x}$) that ensure fulfillment of all constraints and the value of the objective function is maximal. Rows of the matrix $\mathcal{A}$ represent all linear constraints and columns represent variables. Values in each row therefore specify linear combination of the variables for each inequality. Vector $\mathbf{b}$ is the vector of right sides of inequalities. This vector contains constants of inequalities. Vector $\mathbf{c}$ specifies multiplicative constant for each variable of vector $\mathbf{x}$. Values of vector $\mathbf{c}$ can be understood as contributes of variables to the objective function. The higher the value, the greater contribution. Each variable in vector $\mathbf{x}$ (solution of the problem) is further limited by its lower bound (LB) and upper bound (UB) and must be of integer value.

Solutions of ILP models can be obtained by ILP solvers. There are both free / open

software solvers, for example *Ip solve*, *GLPK* or *Coin-OR Cbc*, and commercial solvers, for example *CPLEX*, *Mosek*, *Xpress* or *GUROBI*. ILP solvers are based on branching algorithms or cutting plane methods together with bounds estimations. Nowadays, when we can use quality solvers and we possess great computational resources, we can solve ILP models with more than a thousand variables.

The problem of solving the ILP belongs to the $\mathcal{NPC}$ class. This means that computational demands to solve the most of the ILP problems grow exponentially with increasing number of variables. Therefore, ILP solvers are used only for small instances of problems. Without the integrality restriction (linear programming), the problem becomes solvable in polynomial time. However, by rounding the solution obtained by liner programming (LP) we do not necessary gain optimal or even feasible solution. Therefore, LP can be used only as upper and lower bound estimation method. The question is then why to model optimization problem with integer linear programming which is difficult to evaluate and why not use linear programming instead. The answer is quite simple. In manufacturing and other processes dealing with optimization we cannot create 0.8 units of product using 3.2 units of resource.

A hybrid approach that combines both LP and ILP is called mixed integer linear programming (MILP). Using MILP we can model problems that consist not only of discrete (decision) variables but also continuous variables. Relaxing requiriments of some variable to be integers, ILP solver can found desired solution in shorter time. This approach is even more useful when the integer value, as the only one possible, is ensured by context of other variables or equations. In this thesis, MILP model is chosen to use such behavior without lost of accuracy or feasibility.

4.1 MILP model

In this section, a mixed integer linear programming model is given. This model is further used to find exact solutions of small instances using ILP solvers.

For the greater efficiency of the model, capacities of resources are cumulative in the

following model. Capacity of the resource R_q, using cumulative capacities, thus begins on value given by sum of all predecing resources' capacities. The resource's cumulative capacity ends on value given by the sum of the cumulative capacity's beginning and its own capacity. Using mathematical formula, the total cumulative capacity is given as follows: $\theta_q^C = \sum_{j=1}^{q} \theta_i$. Further, variable z_{ivk} is substituted by z_{il}, where $l = \sum_{q=1}^{v-1}(\theta_q) + k$. And vice versa conversion is $v = arg\ min\left(\sum_{q=1}^{v} \theta_q > l\right)$ and $k = l - \sum_{q=1}^{v-1} \theta_q$.

4.1.1 Variables

$$f_{ij} = \begin{cases} 1 & \text{if setup time } st_{ij} \text{ is considered in criterion} \\ 0 & \text{otherwise} \end{cases}$$

$$z_{ik} = \begin{cases} 1 & \text{if activity } i \text{ is assigned to cumulative capacity } k \\ 0 & \text{otherwise} \end{cases}$$

$$x_{ijk} = \begin{cases} 1 & \text{if both activities } i \text{ and } j \text{ are selected and both activities } i \text{ and } j \text{ are assigned} \\ & \text{to capacity } k \text{ and } i \text{ is an arbitrary predecessor of } j \text{ on a capacity } k \\ 0 & \text{otherwise} \end{cases}$$

$$g_{ijk} = \begin{cases} 1 & \text{if both activities } i \text{ and } j \text{ are selected and both activities } i \text{ and } j \text{ are assigned} \\ & \text{to capacity } k \text{ and } i \text{ a direct predecessor of } j \text{ on a capacity } k \\ 0 & \text{otherwise} \end{cases}$$

$$y_{ijk} = \begin{cases} 1 & \text{if both activities } i \text{ and } j \text{ are assigned to capacity } k \\ 0 & \text{otherwise} \end{cases}$$

$$v_i = \begin{cases} 1 & \text{if activity } i \text{ is selected} \\ 0 & \text{otherwise} \end{cases}$$

s_i =start time of activity i

4.1.2 MILP model

$$min \sum_{\forall i \in \mathcal{A}} \sum_{\forall j \in \mathcal{A}} f_{ij} \cdot st_{ij} \cdot c_{ij} \tag{4.1}$$

such that:

$$\sum_{\forall i \in \mathcal{A}_\mathcal{E}} v_i \geq 1 \tag{4.2}$$

$$v_i = v_j \qquad \forall (i,j) \in E \wedge outLabel_i \neq ALT \wedge inLabel_j \neq ALT \tag{4.3}$$

$$v_i = \sum_{\forall j \in \mathcal{A}_\mathcal{E} : (j,i) \in E} v_j \qquad \forall i \in \mathcal{A}_\mathcal{E} : inLabel_i = ALT \tag{4.4}$$

$$v_i = \sum_{\forall j \in \mathcal{A}_\mathcal{E} : (i,j) \in E} v_j \qquad \forall i \in \mathcal{A}_\mathcal{E} : outLabel_i = ALT \tag{4.5}$$

$$s_i + (1 - v_i) \cdot UB \geq r_i \qquad \forall i \in \mathcal{A} \tag{4.6}$$

$$s_i \leq \tilde{d}_i - p_i + (1 - v_i) \cdot UB \qquad \forall i \in \mathcal{A} \tag{4.7}$$

$$s_j - s_i + (2 - v_i - v_j) \cdot UB \geq lij \qquad \forall (i,j) \in E \tag{4.8}$$

$$s_j + p_j + st_{ji} \leq s_i + UB \cdot (x_{ijk} + 1 - y_{ijk}) + UB \cdot (2 - v_i - v_j)$$

$$\forall (i,j) \in \mathcal{A}^2 : i \neq j; \forall k \in \{1 \dots K\} \tag{4.9}$$

$$s_i + p_i + st_{ij} \leq s_j + UB \cdot (2 - x_{ijk} - y_{ijk}) + UB \cdot (2 - v_i - v_j)$$

$$\forall (i,j) \in \mathcal{A}^2 : i \neq j; \forall k \in \{1 \dots K\} \tag{4.10}$$

$$\sum_{k=C+1}^{C+\theta_q} z_{ik} = \mathrm{R}_i^q \cdot v_i \qquad \forall i \in \mathcal{A}; \forall q \in \{1 \dots m\}; C = \sum_{j=1}^{q-1} \theta_j \tag{4.11}$$

$$z_{0k} = 1 \qquad \forall k \in \{1 \dots K\} \tag{4.12}$$

$$z_{n+1\,k} = 1 \qquad \forall k \in \{1 \dots K\} \qquad (4.13)$$

$$y_{ijk} \geq z_{ik} + z_{jk} - 1 \qquad \forall (i,j) \in \mathcal{A}^2_{\mathcal{E}} : i \neq j; \forall k \in \{1 \dots K\} \qquad (4.14)$$

$$y_{ijk} \leq z_{ik} \qquad \forall (i,j) \in \mathcal{A}^2_{\mathcal{E}} : i \neq j; \forall k \in \{1 \dots K\} \qquad (4.15)$$

$$x_{ijk} \leq y_{ijk} \qquad \forall (i,j) \in \mathcal{A}^2_{\mathcal{E}} : i \neq j; \forall k \in \{1 \dots K\} \qquad (4.16)$$

$$\sum_{j=1}^{n+1} g_{ijk} = z_{ik} \qquad \forall i \in \mathcal{A}; \forall k \in \{1 \dots K\} \qquad (4.17)$$

$$\sum_{i=0}^{n} g_{ijk} = z_{jk} \qquad \forall j \in \mathcal{A}; \forall k \in \{1 \dots K\} \qquad (4.18)$$

$$g_{ijk} \leq x_{ijk} \qquad \forall (i,j) \in \mathcal{A}_{\mathcal{E}}{}^2; \forall k \in \{1 \dots K\} \qquad (4.19)$$

$$f_{ij} \cdot UB \geq \sum_{\forall k \in \{1 \dots K\}} g_{ijk} \qquad \forall (i,j) \in \mathcal{A}^2_{\mathcal{E}} \qquad (4.20)$$

where: $K = \sum_{q=1}^{m} \theta_q$

The objective function (4.1) consists of all performed weighted setup times and the goal is to obtain the minimal possible value. Formula (4.2) forces schedule to have at least one selected activity (empty schedule has no relevant significance). Equation (4.3) defines rules for selection of activities in parallel branchings. Constraints (4.4) and (4.5) define rules for selection of activities in alternative branchings. Start time of an activity is constrained by release time and dealine - (4.6) and (4.7), both constraints are applied for selected activities only. Non-negative start to start time lags are considered in (4.8). Constraints (4.9) and (4.10) represent resource constraints, including sequence-dependent setup times. Constraint (4.11) ensures that the number of resource units assigned to each activity is equal to its demand. Constraints (4.12) and (4.13) are used to assign dummy activities 0 and $n+1$ to each resource unit of each resource type. Inequalities (4.14) and (4.15) bound variable y_{ijk} - if both activities are scheduled on the same resource unit, then y_{ijk} must have value 1,

0 otherwise. Formula (4.16) bounds variable x_{ijk} - if activities i and j are assigned to the capacity k, they must be one after another. Equation (4.17) determines that each activity has on each capacity only one direct successor. Similarly, equality (4.19) determines that each activity has on each capacity only one direct predecessor. Constraint (4.19) prevents cycles in the schedule. Finally, using (4.20) we determine whether setup time is to be taken into consideration in the objective function - whether activities are scheduled subsequently on the same resource and share at least one part of its capacity.

Chapter 5

Heuristic algorithm

This chapter is dedicated to the heuristic algorithm description and the ideas behind the algorithm. Since the problem considered in this thesis is complex, an extensive research was done and many solution approaches were taken into consideration. However, up to our knowledge, there is no work dealing with the specific problem considered in this thesis. Therefore, a new heuristic algorithm is proposed to solve the problem.

During the research, we found several interesting ideas in the related works. Some of them are further modified and extended for the purpose of the proposed algorithm. All the modifications are described in the following sections. The proposed algorithm combines some of already known techniques and approaches with our new techniques and approaches developed to achieve good efficiency for the solution of the considered problem.

The heuristic algorithm is designed for large instances of the problem. Moreover, the model considers many real constraints to meet demands of the nowadays production. All the considered constraints together with the objective function make the problem difficult to solve. Therefore, the proposed heuristic algorithm should effectively evaluate more possibilities how to schedule given activities and choose correctly the best possible direction in the process of finding the desired schedule while all the constraints are fulfilled and the value of the objective function is minimized.

The input of the algorithm is an instance of the $PS|nestedAlt, l_{ij}^{min}, ST_{SD}, r_j, \tilde{d}_j|TSC$ (or equivalently $m1|nestedAlt, min, ST_{SD}, r_j, \tilde{d}_j|TSC$) problem (see Section 3.4) defined by the set of activities, set of resources, NTNA instance, matrix of setup times and matrix of setup costs. The output of the algorithm is a schedule S determined by the selection of activities (variable v_i), start times (variable s_i) and their assignment to resources (variable z_{ivk}). The goal is to minimize the total setup costs (TSC).

5.1 Overall algorithm description

The goal of the algorithm in general is to find the feasible solution with desired properties. In our case, we search for the schedule with the minimal total setup costs and we must ensure, that all the constraints are fulfilled. The proposed algorithm can be classified as a heuristic, meaning that the algorithm does not ensure to find the optimal solution, even the feasible one. Since this algorithm is designed for large instances of a very extensive problem, we accept a solution that is close to the optimal one. The main emphasis is placed on the speed of the algorithm. We want to find the solution of large instances in short time.

The proposed heuristic algorithm consists of two basic phases. First, an initial solution is found (if possible) and second, such solution is further improved. Using this separation, we can apply more different techniques to find a better schedule (e.g. a schedule with the lower value of the objective function). Note, that in both phases we must ensure feasibility of the solution by avoiding the violation of the constraints. The description of both phases of the algorithm is given in the following paragraphs. However, if the first phase does not return a feasible solution, the entire algorithm ends up with failure and the second phase is thus not executed.

5.1.1 Initial solution

In the first phase of the algorithm we try to find any feasible solution. The main idea, on which this phase is based, lies in multi-level priority selection of activities. This means that there are many selection rules used in each sub-problem and its partial subproblems,

etc. Basically, the problem of finding the initial solution consists of many layers and each layer has its own rules to select activities. Priority selection is therefore applied both to determine, which activities will be scheduled (selection among alternative process plans, etc.) and to actual placing of the activities into the schedule. Moreover, a backtracking scheme is proposed to recover from partially unfeasible solution. All the details regarding finding the initial solution are further described in Section 5.2.

5.1.2 Schedule improvement

The second phase of the algorithm is dedicated to further improve the initial feasible solution. We will further refer to this part as *Sliding windows*. The inspiration of this approach is based on publication of Foccaci et al. [18]. In the publication, authors proposed an improvement technique based on the local optimization. Authors divided the whole problem into small sub-problems, called *time windows*. Each sub-problem is then solved independently on the others using limited branch and bound algorithm in order to optimize the sum of the setup times. We further extend this technique to meet constraints considered in this thesis and apply it on the considered problem.

5.2 Initial solution

In this section, we will describe the finding of the initial solution, i.e. the first phase of the heuristic algorithm, in more detail. The solution must be feasible, meaning that even the initial solution has to fulfill all the constraints. The first phase can be split into two main parts: preparations and construction of the schedule. Each part and its sub-problems are discussed in the following paragraphs.

At the beginning of the first phase we pre-process given data. Such preprocessing allows us to generate the schedule more effectively later on. First, we propagate release times and deadlines to estimate possible position of activities within the schedule more precisely. Second, we establish selection priorities. This means, that we evaluate each alternative branching to decide, which alternatives are better then the others. Finally, we set back-

tracking rules, that are used to to decide which type of action is applied after no activity can be further scheduled.

After all preparations, the schedule is being constructed. Let *Ready* be a set of activities, that can be scheduled at actual iteration step. An activity is pronounced as *Ready*, if it is not yet scheduled and all its considered predecessor are already scheduled. First, we choose activity i which is the most critical from the perspective of the actual schedule from the *Ready* set. Second, we try to schedule it. If we are unsuccessful, backtracking scheme is applied. Otherwise, we update the *Ready* set by adding ready successor of the just scheduled activity. If node i is the beginning of the alternative branching, we add to the *Ready* set only one of its successors using rules described further in text. Such process is called as an expansion of the activity.

The entire process can be expressed using the following pseudo code:

```
Propagate release times
Propagate deadlines
Establish selection priorities
Set backtracking rules
Establish the Ready set

while Ready is not empty
    activity = Select activity from Ready set
    if activity is successfully scheduled
        Expand activity
    else
        Apply backtracking rules
    end if
end while
```

The following subsections deal with each mentioned sub-problem in more detail.

5.2.1 Propagation of release times and deadlines

Propagation of release times and deadlines enables us to calculate more accurate time window, in which an activity can be scheduled. Propagated release time of activity i ($\widehat{r}_i$) depends on the propagated release times of activity's predecessors and time-lags between such predecessors and activity i. In case of only one predecessor j of activity i, the propagated release time is equal to $\widehat{r}_i = max(r_i, \widehat{r}_j + l_{ji})$. Notice, that time lags are non-negative and define the minimal time interval between start times of two related activities. Setup times cannot be considered, because they depend on specific assignment of activities on resources. Similarly, propagated deadlines of activity i ($\widehat{d}_i$) depends on activity's successors propagated deadlines and time lags between activity i and its successors. Note, that while propagating deadlines, we must also consider processing times of involved activities (see Figure 5.1). In case of only one successor j of activity i, the propagated deadline is calculated as $\widehat{d}_i = min(\tilde{d}_j, \widehat{d}_j - p_j - l_{ij} + p_i)$.

If some activity has more than one successor (predecessor), we must decide which propagated value to use. There are two rules while estimating propagated release times and deadlines. In case of alternative branching, we use an optimistic estimation, in case of parallel branching a pessimistic estimation is used. When we calculate propagated release time of activity i, we use a minimum of all propagated release times, considering also time lags, given by activity's predecessors, in case of alternative branching, i.e. $min(\widehat{r}_j + l_{ji})$ for all $j \in predecessors(i)$. A maximum is considered in case of parallel branching, i.e. $max(\widehat{r}_j + l_{ji})$ for all $j \in predecessors(i)$. Similarly, when we calculate propagated deadlines we use as the value a maximum of all propagated deadlines given by activity's successors in case of alternative branching and a minimum in case of parallel branching. To estimate such values, we must propagate release times and deadlines separately. Release times are propagated from the first activity (0) and deadlines are propagated from the last activity $(n+1)$. For effective propagation, a topological ordering of activities within NTNA graph is used. Doing so, we are sure, that all activity's predecessors (successors) were already propagated and also, that all branchings are considered properly.

If we schedule activity outside this estimated time window, it will finally lead to unfeasible schedule. We cannot schedule activity at time before this time window because it's

already scheduled predecessors will not allow us to do so. Similarly, we cannot finish an activity after the end of the time window, because the successors of the activity will be forced to start later which will at the end cause violation of some deadline constraint. Of course, original release time or deadline of an activity can violate the estimated time window as well.

Propagation therefore states upper and lower bound estimation of activity's start time and allows us hereby to quickly detect unfeasible schedules so we can recover sooner from such situation and thus save valuable computational time.

An example of the entire process is depicted in Figure 5.1

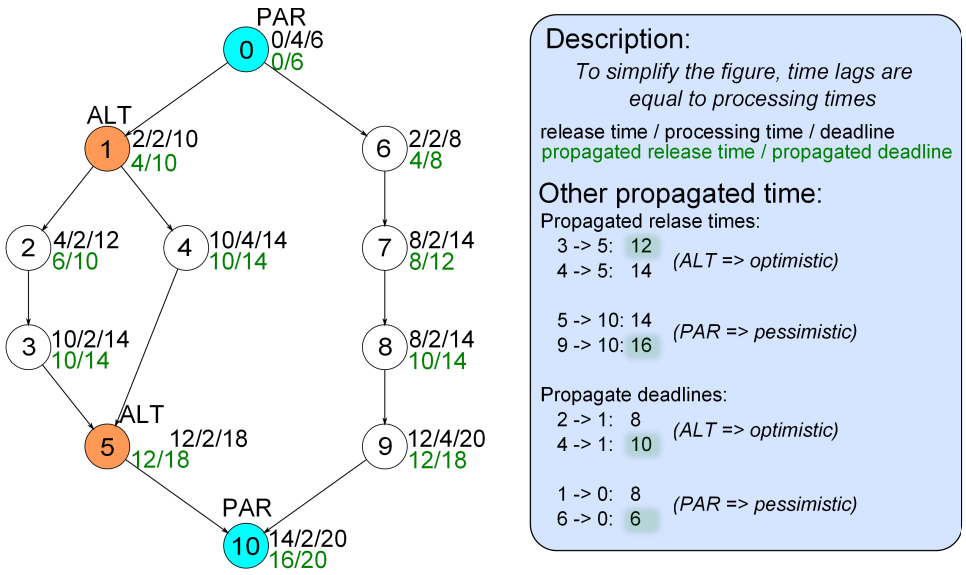


Figure 5.1: Propagation of release times and deadlines

5.2.2 Establishing branching selection priorities

Since the problem considered in this thesis includes also alternative process plans, we must evaluate each alternative branching to decide which branch is potentially better or worse. In the following paragraphs, the method how to rate each branch in alternative branching is described. While scheduling activities, better rated branches are chosen first.

In fact, we determine an ordering of branches for each alternative branching and then choose the branches in such order.

The selection starts from the most nested alternative branching and ends in the top most one. Doing so, we ensure that the nested alternative branchings are resolved before the parent ones. After the selection establishment, the whole alternative branching is further seen as just the best rated branch using rules mentioned bellow. This means, then while establishing branching selection priority for an alternative branching, we have already evaluated all nested alternative branchings and we include into the calculation only the best evaluated branches, not the entire nested alternative branchings.

To facilitate the entire process we can use topological ordering. We start the establishment by the activity (of course, we consider only activities in which the alternative branching begins) with the highest topological number first. This will ensure that the most nested alternative branchings (higher topological number) will be processed before the top most ones (lesser topological number).

The entire priority establishment for an alternative branching consists of three phases. First, we determine number of activities in each branch and the sum of costs of all involved activities in order to calculate average cost. Second, we estimate the cost of each branch (see 5.2.2.1 for more details). Finally, we compare calculated costs and we create ordering of the branches from the best one to the worst one using rules described in section 5.2.2.2. Moreover, alternative branches which lead to the unfeasible solution are directly rejected. Detection of such branch is rather simpler thanks to the propagated release times and deadlines. Any branch with at least one activity with negative cost is unschedulable (see activity 8 the in branch beginning with branch candidate 6 in Figure 5.2).

5.2.2.1 Rating of branches

Let $\mathcal{B}_a = \{B_1 ... B_{\delta^+(a)}\}$ be the set of all branches of the alternative branching that begins in node a and ends in node b. Each branch $B_j \in \mathcal{B}_a$ consists of activities that form a subgraph starting by some successor of activity a and ending by the corresponding predecessor of activity b. See Figure 5.2, where $\mathcal{B}_1 = \{\{2, 3\}, 4\}$.

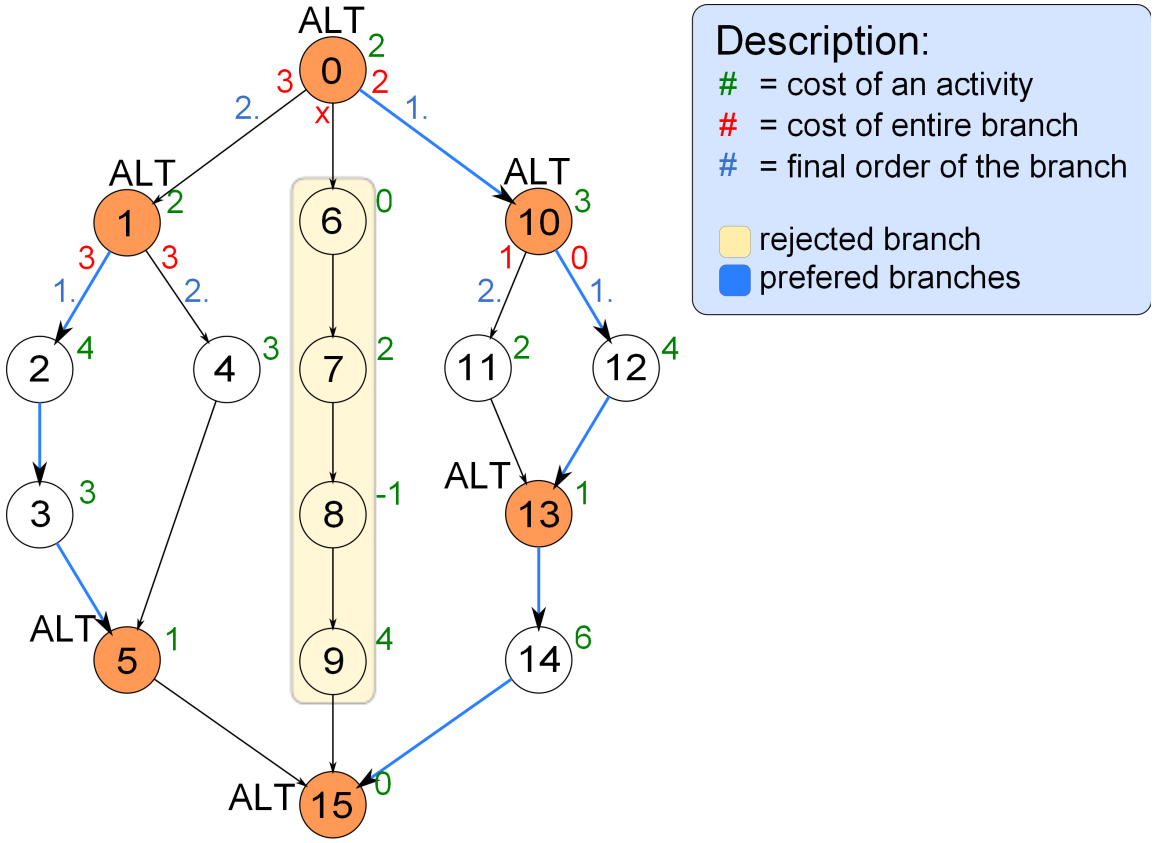


Figure 5.2: Establishment of selection priorities

To evaluate each branch, we first need to calculate so called *cost* of each activity in the alternative branching. Therefore, let $cost_i = propagated\ deadline_i - processing\ time_i - propagated\ release\ time_i = \widehat{d}_i - p_i - \widehat{r}_i$ be a cost of activity i. Cost of an activity determines its flexibility within the schedule. The lower values correspond to activities that have a tight time window and therefore they are more critical in the scheduling process. The higher values correspond to activities that have a wide time window and they possess the larger flexibility where to put them in the schedule.

An arithmetic average of all activities in the alternative branching determined by $\mathcal{B}_a$ is then given as $\overline{cost}_a = average(cost_i)$, for all $i \in \mathcal{B}_a$. The value of $\overline{cost}_a$ determines an average flexibility of the activities in the alternative branching. To estimate the probability of scheduling success for each branch with respect to temporal constraints, we have decided to focus on the more critical activities, i.e. activities with the cost value lower than the

average cost of the alternative branching. Therefore, to calculate the cost the branch $B_j \in \mathcal{B}_a$ the following formula is used $cost_{B_j} = \sum(\overline{cost}_a - cost_i)$ for all $i \in B_j : cost_i \leq \overline{cost}_a$. Only activities with $cost_i \leq \overline{cost}_a$ are assumed for the calculation.

5.2.2.2 Selection rules

After all branches in alternative branchings are rated, we can determine their ordering using the following set of hierarchical rules (if more alternative candidates has the same evaluation, the order is determined by the next rule, etc.). Given rules were derived using large instances of the problem and they represent a compromise for many different possible situations.

1) minimal branch cost

2) minimal count of activities in branch

3) branch candidate with lower topological number

For the first rule we have already eliminated the most flexible activities (activities with cost greater then average cost) in order to focus on the less flexible ones. We prefer a branch, that has the lowest cost $cost_{B_a}$. This way we choose the alternative branch that is the closest from the average cost. The chosen branch has the highest probability of being successfully scheduled since the activities in it are the most flexible ones. Notice, that the establishment is done before building the schedule, so we do not consider assignment on the resources or setup times.

The second rule prefers the alternative branch with lower count of activities. Notice, that rule 1 ended up indecisively, so the branches are of the same cost. The rationale behind this rule is to select lower amount of activities, so the number of scheduling/unscheduling steps in the rest of the algorithm will be most likely lower than for branch with more activities. In other words, we are trying to schedule less activities to save the computational time and use it in another part of the algorithm.

The last rule is applied if previous two rules ended up indecisively. In that case, we prefer branch, whose branch candidate has lesser topological number just to select a branch in a deterministic way.

Notice, that while establishing priorities for alternative branching that begins by activity 0 in Figure 5.2, we do not consider rejected branch beginning with branch candidate 6. Also, we do not consider activities in not selected branches in nested alternative branchings (activities 4 and 11).

5.2.3 Establishment of the set of ready activities

Before the schedule construction is started, we must initialize the *Ready* set. Thanks to the structure of the NTNA only activity 0 has to be added to the *Ready* set (i.e. the set of ready activities). This activity is a root of the entire NTNA graph and is the only activity which is ready in the first iteration of the process of the schedule construction.

5.2.4 Selection of activity from the set of ready activities

Set of ready activities always contains activities, that can be scheduled according to the actual state of the schedule. At the beginning of the algorithm, ready set contains the first activity 0. The set of ready activities is then updated after any change in the schedule (see Sections 5.2.6 and 5.2.7 for more details).

After all preparations are done, we must choose one activity from the set of ready activities in each interaction. Activity is ready, if all its considered predecessors are already scheduled. Basically, we choose the activity, that is the most critical in the context of the actual schedule. This means, that we do not choose activities that are very flexible at the moment, but activity which is the least flexible instead, considering also influence on its successors.

For each activity $i \in Ready$ we calculate effects on its successor. Basically, we determine how would activity i change possible start times of its successor if it is scheduled in this

iteration. The changes of possible start times are reflected in updated successors costs.

First, we estimate the earliest possible start time of each activity i considering all the constraints including sequence-dependent setup times and resource availability. In general, we evaluate the attempt to schedule activity i. Second, we calculate effects on successors for each activity i in the following way. The effect on successor j of activity i is equal to $\widehat{d}_j - p_j - (s_i + l_{ij})$. The evaluation of scheduling attempt for activity i is then equal to the minimal value of effects over all selected successors. Finally, we choose activity i with the minimal evaluation value. Doing so, we find activity, which is the least flexible for the moment. Notice, that we calculate effect only on activity's successors, not on all the remaining activities. This is caused by propagated release times and deadlines. If any successor violates its time window, we cannot further obtain a feasible solution. If successor fits inside the window, we can still find a feasible solution.

5.2.5 Scheduling of an activity

While scheduling an activity, we must consider all the constraints, i.e. temporal, resource and selection constraints. Depending on the assignment on the resource we must also consider the influence of sequence dependent setup times on the position of activities. To schedule an activity properly, we must find the earliest possible start time within its time window. Time window of activity i is determined by its propagated release time $\widehat{r}_i$ and propagated deadline $\widehat{d}_i$. Activity i cannot violate its time window and all constraints, including time lags and sequence-dependent setup times, must be fulfilled. If no time slot is found, scheduling of activity ends with failure and backtracking is applied.

The process of finding time slots on the resource is based on the serial generation scheme proposed by Kolisch [26]. The main idea lies in selection only one activity i in each iteration from the set of *Ready* activities. Selected activity i is then scheduled as soon as possible and removed from the *Ready* set. Further, the *Ready* set is updated with respect to successors of activity i. The selection from the *Ready* set is based on priorities.

To fasten the process of scheduling activity i, we created specific data representations. The time axis of the resource is divided by so called *milestones*. Milestone is a time moment

in which there is any change on a resource. First, we locate the nearest milestone to the desired start time, that is given by time lags and propagated release time. Second, we specify the earliest possible start time by sequence-dependent setup times on each resource part. Finally we estimate free time intervals on each resource part and try to find and intersection of activity's resource demands with free time intervals. This way we obtain assignment of the activity on the resource (for more details see Chapter 6).

After an activity is scheduled, we must update the set of ready activities by exploring successors of the activity. If the activity cannot be scheduled, backtracking rules are applied.

5.2.6 Updating the set of ready activities

After an activity i is scheduled, it is removed from the set of ready activities. Similarly as in serial generation scheme (see Kolisch [26]) we must update the set of ready activities by all activities which are released by activity i. Activity is released, if all its considered predecessors are already scheduled.

Let $\mathcal{S}$ be a set of successors of activity i. Each activity in the set $\mathcal{S}$ is a candidate to be added into set of ready activities. Each candidate is further tested if it fulfills all the rules given bellow. If the candidate succeeds, it is added to the set of ready activities. There are three possible situations regarding just scheduled activity i and each successor $s \in \mathcal{S}$:

1) Activity i is the beginning of an alternative branching:
   While scheduling alternative branchings, we must choose only one branch B_k from all branches $\{B_1 ... B_{\delta^+(i)}\}$ in alternative branching that begins with activity i. In the preparation phase of the proposed initial solution, we have already established selection priorities for each alternative branching. The set of ready activities is thus updated by a single activity (branch candidate). This candidate represents entire alternative branch and is chosen using rules given in Section 5.2.3. When we are selecting from alternative branches, we prefer the best rated one as first and the worst rated one as last.

2) Activity s finishes a parallel branching:

If activity s that finishes parallel branching is in the *Ready* set, all the activities in parallel branchings have to be already scheduled. In other words, activity s (that is not scheduled) is added in to *Ready* set if and only if all of its predecessors $p \in predecessors(p)$ are scheduled. We can imagine activity s as a barrier on which we wait until all activities before this barrier are already scheduled. Then the barrier can be released.

3) Otherwise:
This case covers both the beginning of parallel branching and simple precedence constraint. In both cases, all successors s of activity i are added into *Ready* set.

5.2.7 Backtracking

When scheduling of an activity ends with a failure, backtracking is applied. This technique is used to recover from such situation by discarding a part of the schedule. There are two main backtracking schemes used in the first phase of the heuristic algorithm - change of the selection of activities and change of the ordering on resource. To determine which kind of backtracking will be applied, we use backtracking rules which are determined in the preparation phase of the initial solution.

5.2.7.1 Backtracking rules

Backtracking rules are used to quickly decide which kind of backtracking to apply. The decision is made upon a type of the nearest preceding node with output branching in the NTNA structure. If such branching is alternative, we apply change of selection of activities, otherwise we apply change of the ordering on a resource.

If backtracking from given activity fails, we must try backtracking from another activity again. In case of alternative branching we can choose different branch than is actually selected, otherwise we must choose one of predecessors and start backtracking over again. When the backtracking is not further possible (we try to backtrack from the root activity 0), the entire heuristic algorithm ends with failure.

5.2.7.2 Changing selection of activities

If activity which cannot be scheduled is nested in an alternative branching, we find node i, where the alternative branching begins and we select another alternative branch using previously mentioned rules (see Section 5.2.2.2). Insomuch as there can be at most only one selected branch in each alternative branching, we must unschedule the entire previously chosen alternative branch. We must also update the set of ready activities to reflect actual situation. First, all successors of all unscheduled activities within previously chosen alternative branch are removed from the *Ready* set. Second, the first activity in currently chosen branch is added. The set of ready activities is now updated and we can continue in the main loop of the algorithm.

After all branches for alternative branching are tried out and still the schedule is unfeasible, we perform backtracking from the node i, where the alternative branching begins. If the activity corresponding with the opening node i is unscheduled in another iteration, the actual state of selection of activities is cleared and when such activity i is scheduled again and being expanded, we start choosing branch candidates from the best one over again.

One of possible situations is depicted in Figure 5.3: Activity 6 cannot be scheduled (for example due to overfilled resource). Since activity 6 is nested in alternative branching, we choose next branch candidate according to rules given in 5.2.2.2. In the preparation phase, we established, that the next branch candidate is activity 9. Since there can be at most only one selected alternative branch in each alternative branching, the whole branch beginning with branch candidate 5, e.g. activities 5 and 6, has to be unscheduled. Then, the set of ready activities has to be updated. First, we remove activity 6 from the *Ready* set. Second, we add currently selected branch candidate (activity 9) to the set of ready activities.

5.2.7.3 Change on resource

Change on resource is applied, if the first branching preceding activity i in NTNA is parallel. The main idea lies in reordering the assignment of activities on the resource. The activity, whose scheduling ended with failure is tried to get scheduled earlier in order to

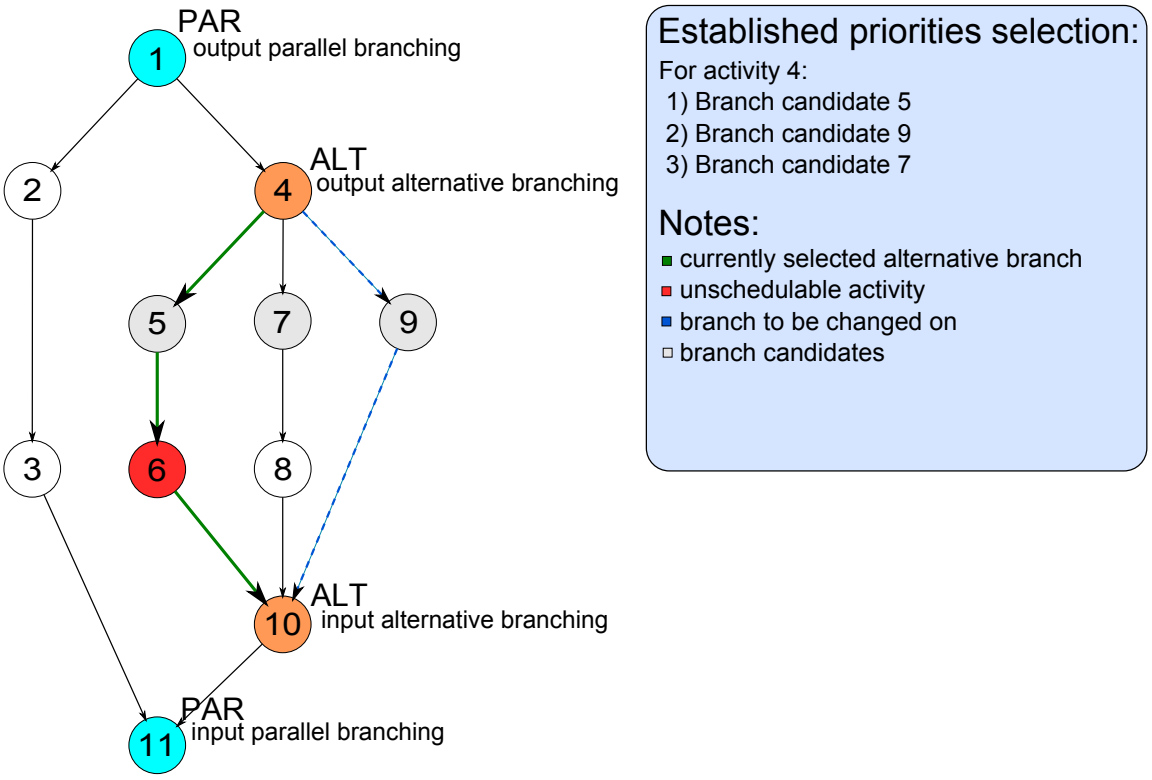


Figure 5.3: Change of alternatives example

fulfill all the temporal constraints.

First, we have to find the nearest activity i_{near} that has lower start time on the resource and is not preceding activity i in the context of NTNA instance. Activity i_{near} and all of its successors are then unscheduled. This way we free a part of the resource. While finding the activity i_{near}, we remember a set $\mathcal{P}$ of all activities between i_{near} and i. The set $\mathcal{P}$ consists of predecessors of activity i on the resource. Second, an activity i and all activities from $\mathcal{P}$ are also unscheduled. Then, we can reschedule all activities from the set $\mathcal{P}$ and activity i again. Notice, that after removing activity i_{near} and all of its successors, activity i and its predecessor in $\mathcal{P}$ can be scheduled earlier. Such shift of activity i is done in order to try to fulfill the violated constraint. Afterwards, the algorithm continues by scheduling the next activity from the *Ready* set.

If the schedule is still unfeasible, we try to backtrack once again from another activity.

A solution of the problem given in Figure 5.3 is depicted in Figure 5.4. While scheduling activity 3, one constraint was violated (deadline restriction). In order to recover from this situation, backtracking rules are applied. In this case it is change on resource. Activity 4 and all its successors (i.e. activities 4 and 9) have to be unscheduled to free the resource. Then activities 2 and 3 are re-scheduled. Then the algorithm continues as usual and activities 9 and 10 are scheduled.

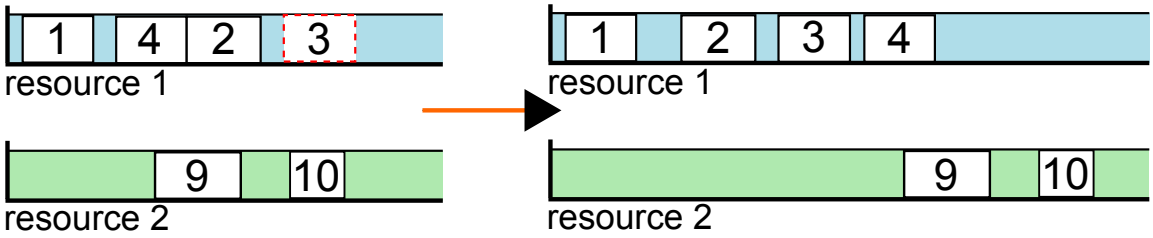


Figure 5.4: Change on resource example

5.3 Sliding windows

Sliding windows is a technique based on the work of Foccaci et al. [18]. It is a heuristic approach that utilizes a local optimization scheme. The main idea lies in the separation current solution into independent *time windows*. Then we optimize each time window separately. This way we consider only a reduced set of all activities.

The goal of this phase is to optimize the given initial solution, in other words to reduce the total setup costs (TSC). Thanks to the separation into smaller time windows, we can try more combinations of assignment of activities to the resource parts. The current solution within the time window (part of the current solution bounded by this window) is dropped and new the solution with minimized TSC is found. If the new solution of the time window is better then the original one, it is integrated into the current solution.

During this phase, we optimize one time window at a time. The alternative branchings are fixed, i.e. we do not try to change alternatives, because it could affect more than just the current time window. The term sliding means, that after time window is optimized, we move to another time window, but these two windows overlap. The optimization of the

independent sliding windows can be run in parallel, but in this thesis, we decided not to extend the algorithm for such functionality. The parallel extension can significantly reduce the computational time, but the implementation of such behavior would be much more complex and therefore it is not used in this thesis.

All the calculations are done over the current time window. In order to make the entire algorithm flexible and mainly clear, we enhanced many methods used in initial solution to consider also given time window. The main loop of this phase is described in the following pseudo-code:

```
Determine the first time window
while not reached the end of the schedule
    Determine activities for the time window
    Determine set of ready activities
    Optimize time window
    if TSC was reduced
        Integrate into solution
    end if

    Determine next time window
end while
```

5.3.1 Determination of the time window

Time window is specified by its borders and activities within. Notice, that time window covers all the resources simultaneously. Time window also contains information about ready activities and resources. Each border is exactly determined by the time in which it is placed. For each border we remember bordering activities, i.e. the first activities before and after the time window in order to calculate setup times. The bordering activities are fixed for the current time window and cannot be rescheduled or shifted in any way. Further, for each border of the time window we determine the milestone that is the closest one to the border, but still outside the time window. These two milestones for each resource determine the

segment which is optimized from the view of the resource. The time window is illustrated on Figure 5.5.

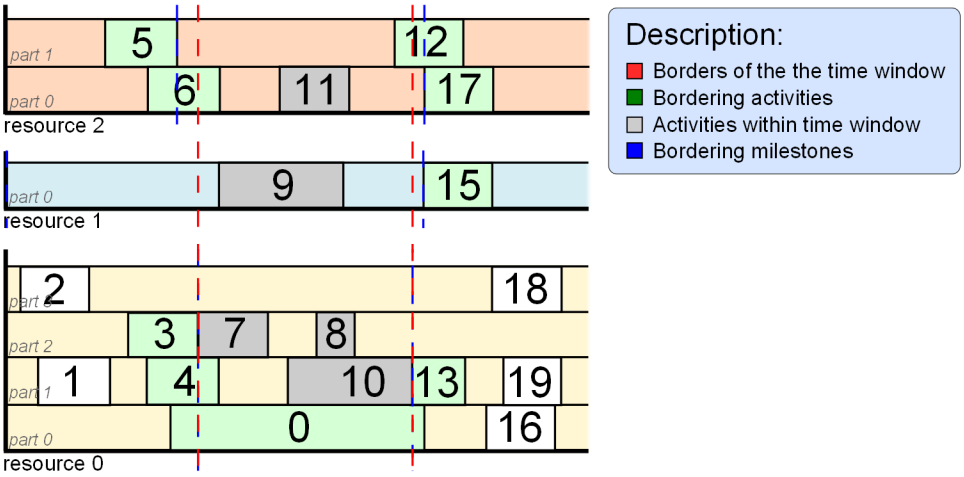


Figure 5.5: Time window and its properties

The size of the time window significantly affects the computation demands for the local optimization. Since the solution within the time window is built from the scratch and many combinations of the assignment of the activities are taken into consideration, we must choose its size cautiously. The maximal size of the time window is given by two parameters:

1) Maximal number of the activities on any resource within the time window.

2) Maximal total number of the activities within the time window.

For the purpose of this thesis, those thresholds were estimated as 15 activities on the resource and 40 activities within the time window. These two constants estimate a balance between computational demands and quantity of assignment combinations.

5.3.1.1 Left border

Left border is a time moment where the time window begins. If there was no time window before (we want to optimize the first one), 0 is chosen as the left border, i.e. the

beginning of the current solution. Otherwise we choose a time in the middle of the previous time window from the perspective of assigned activities as the left border. In other words, the time is in our case determined by the completion time of the 20th activity within the time window.

5.3.1.2 Right border

Similar to left border, right border is the time moment where the time window ends. Closing time of the time window is derived from its left border using rules described in section 5.3.1. After any of mentioned thresholds or the end of the current solution is reached, we use the completion time of the last considered activity as the right border.

5.3.2 Ready activities

Each time window contains information about ready activities. The decision whether the activity is ready or not is slightly modified when using time windows than it was in the initial phase of the algorithm. Activity i within the time window is ready, if all its $predecessors(i)$ are outside the current time window, or are already scheduled in the current time window. Notice, that in this phase activities outside the time window are already scheduled, so we can focus only on predecessors of activity i which are within the time window.

5.3.3 Optimization of the time window

The solution in the current time window is dropped and the new solution is found using different rules than in the initial phase. The optimization of the time window focuses on reduction of the objective function, i.e. TSC. The optimization is based on limited discrepancy search (LDS) presented by Harvey and Ginsberg [22]. This technique is similar to Branch and Bound approach, but the total number of explored solutions is very limited. Generally, the activity from the *Ready* set to be scheduled is chosen by the heuristic rule. In case that the limited discrepancy search is used, there can be some decision points where the heuristic rule is not followed. Such decision point is called *discrepancy*. The process of

finding the solution of the single time window is repeated several times, while the discrepancy is used in different levels of the search tree. The search tree is determined by the sequence in which the activities are scheduled. The best sequence of the activities (the best path in the search tree) is than chosen as the result of the optimization of the time window.

5.3.3.1 Construction of the schedule

Thanks to the limited size of the time window, we can try out more combinations of activities assignment. Using LDS technique we can explore more possibilities. Discrepancy search significantly cuts the search tree so we do not evaluate all combinations of activities assignment. Complete evaluation would cost a lot of computational time. This way we evaluate only the path in the tree with the highest probability of TSC reduction.

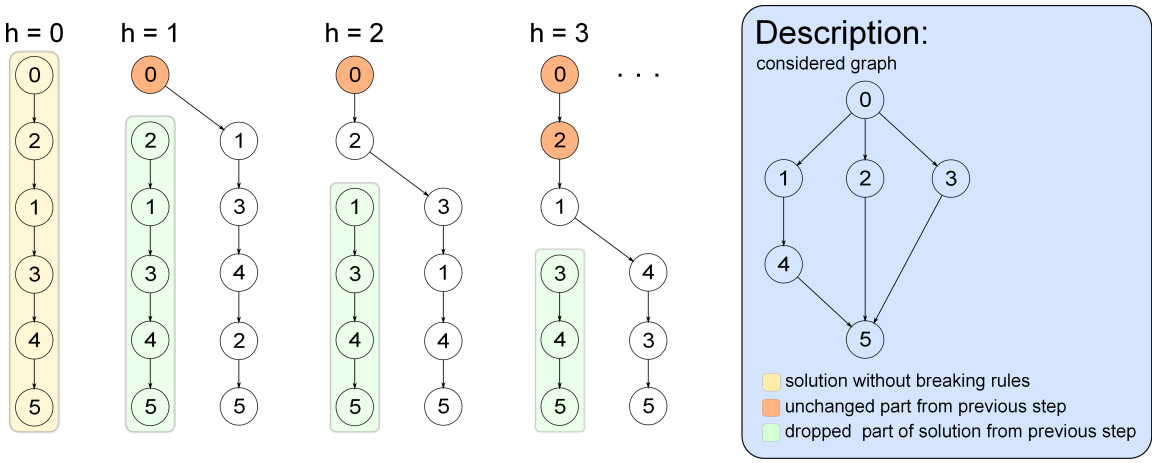


Figure 5.6: Search tree of the time window for different steps of algorithm

In our case we use discrepancy in just one level $h \in \{0 \dots n_{TW}\}$ of the search tree, where n_{TW} is the number of activities within the time window. The level $h = 0$ corresponds to the situation where the heuristic selection rule is followed in each step, i.e. discrepancy is not actually used (see Figure 5.6). While using discrepancy in level h, we can use partial solution up to the level $h - 1$ and reschedule only the rest of the activities. The process of scheduling the activities is similar to the one described in the initial phase of the proposed heuristic algorithm. Unlike the initial phase, no backtracking rules are applied and the rules

for selection of activities from the *Ready* set are different. The process of optimizing a single time window from the view of search tree is depicted on Figure 5.6

The optimization process for the time window with discrepancy is described in the following pseudo-code. Notice, that this method is repeated for each considered level h and the sequence of activities, which led to the solution with the minimal value of the objective function, is returned.

```
Unschedule part of the solution from level h
Determine ready activities

while Ready is not empty
    activity = Select activity from Ready set
    if activity is successfully scheduled
        Update ready activities
    else
        Abort optimization of the time window
    end if
end while
```

5.3.3.2 Selection from ready activities

The selection of activity from the *Ready* set is rather different than in initial phase of the proposed algorithm. In the selection we must consider both sequence-dependent setup times and temporal constraints (release times and deadlines). First, we evaluate a schedule attempt of each activity i and determine the change of setup costs given by the actual assignment. The entire process is similar to the one described in section 5.2.4 in initial phase of the algorithm, but in this case, we focus on the change of the setup costs induced by the activity assignment on the resource parts.

While evaluating schedule attempts of each activity, we also check its cost which is given by formula $\tilde{d}_i - p_i - s_i$. If the cost of activity i is lower then given threshold, then activity

i is selected regardless all setup costs. Since the cost of activity i determines its flexibility, we prefer such activity i over the others to avoid infeasibility caused by violating temporal constraints.

If there is no activity whose cost is lower than given threshold in the *Ready* set, we make a selection according to the induced setup costs. Each scheduling attempt is evaluated in the terms of the total setup costs. We choose the activity whose effect was the lowest one, i.e. activity which would increase the objective function at least. In case that activity has more resource demands (i.e. consumes more parts of the resource) we use as the effect the maximal induced setup costs over all assigned resource parts. This way we do not advantage activities with lower resource demands.

5.3.4 Integrating the solution of the time window

After the time window is optimized, we must decide whether to integrate its solution into the whole schedule. We integrate the time window if its solution improves the value of the objective function. In general, the integration stands for updating activities assignment (start times and assigned resource parts). An example of time window integration into the solution is depicted on Figure 5.7.

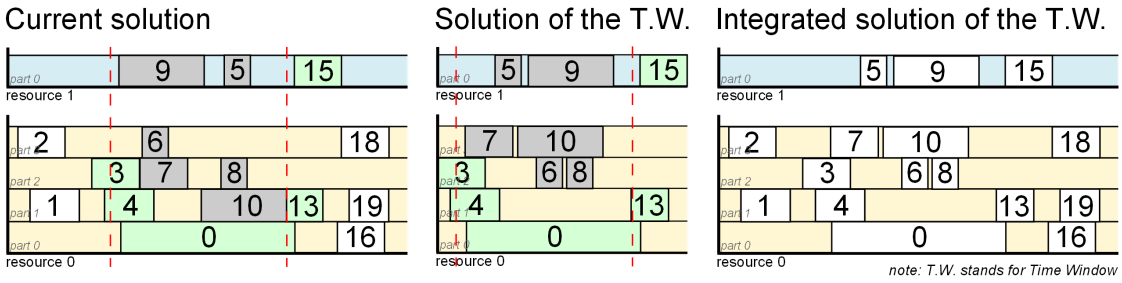


Figure 5.7: Integration of the time window into the solution

Chapter 6

Implementation

In this chapter, we will discuss data representation and present some implementation details to find the desired solution more effectively. Appropriate, even redundant data representation, together with well thought implementation details can significantly reduce the computational time necessary to find the solution. Thus, we can solve larger instances of the considered problem in reasonable time.

For the implementation, programming language *C#* was chosen under Visual Studio 2010 environment. C# is modern and strongly object-oriented programming language that enables us to use high level of code abstraction. Using pre-defined interfaces with desired fields and methods we can use the algorithm even for instances of different problems then the considered one. Also, the advantage of using such programming language is that we can create clear and understandable code. The main disadvantage is the increase in computational complexity then using programming language like C or other low-level languages. Other advantage of the Visual Studio 2010 environment is that it encapsulates many useful tools for application development and without many difficult configuration we can create an application connected to database, create server-client application, etc.

6.1 Data representation

Generally, the algorithm expects data on its input and returns transformed data on its output after the input data are appropriately processed. Data representation must cover all the information necessary to describe the given problem and its solution, which includes in our case information about activities, resources, constraints, alternative process plans, etc. Properly chosen data representation has great influence on both efficiency and asymptotic complexity of the algorithm. Nowadays, when we posses great memory and cache, we can afford to have even redundant data information. Doing so, we can significantly reduce the computational time. But the redundancy must be well thought to be worth it.

The data representation used in the proposed heuristic algorithm can be divided into three groups. First, an input data representation is used to carry out all the information necessary for the algorithm to describe the problem to be solved. Second, an algorithm data representation is given to solve the given problem effectively and is specifically designed for the algorithm itself. Finally, an output data representation is used to describe the solution of the problem given by the algorithm. Notice, that algorithm could also use the input data representation to find the solution without using its own data representation, but it will not be effective. Since the input data representation is designed to be general and to be used fort other algorithms, any operations on such data representation would cause unnecessary loss of time. These are the aspects that led us to use specific partially redundant inner data representation in order to make the heuristic algorithm effective.

6.1.1 Input data representation

Input data representation is used to pass all necessary information about the problem to the algorithm. Moreover, representation should be flexible enough to cover also related problems or generalizations of the problem. This way the input data can be accepted by more different algorithms so we can for example compare theirs solutions, efficiency, etc.

Since *C#* was chosen as the programming language, we use its abstractions to make the input data representation as general as possible and reusable. The input data representation

is divided into two parts: interfaces and classes. Each part is described bellow in more detail.

6.1.1.1 Interfaces

Interface is an abstract data type used to define data entries (fields, methods, etc.). We can imagine interface as a prescription. Interface itself do not posses any information, it is used only to define some desired properties. If a class implements interface, it must define all the data entries of such interface. Class implementing interface can also further add its own fields, methods, etc. Interface can be thus used to define properties, that are common to more classes and can be used for the exchange the data between them. Interface can be used as a parameter of any method or function. In this case, only properties specified by interface are accessible and the rest of the class implementing the interface is hidden. So, if the algorithm expects a data structure implementing some interface, it uses only the data fields specified in interface and does not consider the rest of the fields in the class. Therefore an algorithm can accept more different data representations but all with some common entries.

There are three interfaces used in the implementation of the heuristic algorithm: *IActivity*, *IResource*, *IAlgorithm*. Each interface is described in the following paragraphs.

Interface *IActivity* contains all the necessary information for the single activity, such as: identifier, processing time, release time, deadline, resource, resource demands, etc. All the classes implementing this interface are forced to have all of the specified fields. Doing so, we ensure, that the algorithm has all the necessary information at its disposal. Notice, that interface *IActivity* does not contain start time or information about assignment on the resource. Such values are determined during the process of finding the solution and are a part of the output of the algorithm. At the input of the algorithm such values are irrelevant.

Since the activities are assigned to the resources, we must also specify the resource. Interface *IResource* defines identification, name and capacity of each single resource. Notice, that interface *IResource* does not contain any data structures used while finding the solution. Such properties are specified in classes implementing this interface, but they has no meaning in general. Such structures are used only for the optimization purposes.

The last used interface *IAlgorithm* specifies all the expected outcomes of the algorithm. If more algorithms return a result implementing such interface, we can easily compare outcomes, choose the best one, etc. Interface *IAlgorithm* contains the value of the objective function, generated schedule, the computational time and it specifies functions to set the optimization problem, run the optimization and export the schedule into file. Specific information about activities and their assignment are given in the class implementing this interface. Interface *IAlgorithm* is used to describe common properties of many algorithm outcomes so we can compare them, but we do not say nothing about the solution method itself.

6.1.1.2 Classes

There are altogether six classes to represent the entire input data. Each class carries specific part of the whole problem.

Class *Activity* implements interface *IActivity*. Each instance of this class contains all the information about a single activity for the algorithm.

Class *Resource* implements interface *IResource* and its instances carry out all the necessary information about all the resources for the problem.

Class *ChangeoverTimes* keeps all the sequence-dependent setup times related information. Note, that changeover times can be also called setup times. Since the activities are mono-resource, meaning that each activity can be assigned only to one specific type of resource, changeover times are defined separately for each resource.

To represent non-negative start to start time lags, an instance of the class *TimeLags* is used. Note, that unlike the setup times, time lags constraint activities regardless the resource assignment, so they must be defined as one large matrix.

Class *NestedTemporalNetworkWithAlternatives* contains definition of all the alternative process plans using graph structure: precedences, in labels, out labels etc. This class further adds useful methods to create topological ordering of activities and to find the nearest opening (closing) activity of alternative branchings.

Class *Optimization problem* encapsulates all the mentioned classes and it is therefore used as an input data representation for the algorithm. This class further contains methods to save the given problem to the text file, or to load an instance from specific text file. This class also contains methods to determine and assign precedences (predecessor and successors) to each activity. Also, this class implements function to assign alternative border information (the nearest node with $outLabel = ALT$ and the nearest node with $inLabel = ALT$), etc.

6.1.2 Output data representation

We expect the algorithm to return desired output data representation. In the case of the proposed heuristic algorithm it is a *schedule*. It includes all the activities and their selection and assignment to the resources. Schedule also carries information about its feasibility and the value of the objective function.

Class *Schedule* contains all the information to describe the manufacture process. To do so, we must know entire input data and further add parameters calculated by the algorithm. For the given activities, we add previously mentioned start time and assignment on the resource parts. Further, this class contains information about which activities were selected. Notice, that not all activities have to be selected, because in case of alternative process plans we choose only one branch for each alternative branching, so other activities in the remaining branches in alternative branching are not scheduled.

Another output data representation is used to generate input data for the Interactive Gantt Chart, which was developed at the Center of Applied Cybernetics at Czech Technical University (see bachelor thesis by Dvořák [17] for more details). Using this tool we can clearly see the graphical representation of the entire solution. Interactive Gantt Chart displays all the information about activities, time lags, etc. and is very useful for debugging or visual comparison of more schedules as well.

6.1.3 Algorithm data representation

As it was mentioned above, algorithm data representation is partially redundant, but it significantly reduces computational demands. Unlike input and output data representation,

this representation is adapted for the needs of the proposed algorithm only. This way we can optimize such data structures, so we can obtain the solution in shorter time. Some of algorithm data representation are an extension of input data, others are created newly.

6.1.3.1 Activities

Class *ActivityExtended* implements interface *IActivity* and further adds information about start time and activity assignment to the resource parts. Further, this class contains information about predecessors and successors on resource parts. This is useful to quickly find neighboring activities on the assigned parts of the resource. This information is then used for example to determine setup times or while finding free time slots for activities.

Following Figure 6.1 shows an example of activities' assignment and information about neighboring activities on resource parts.

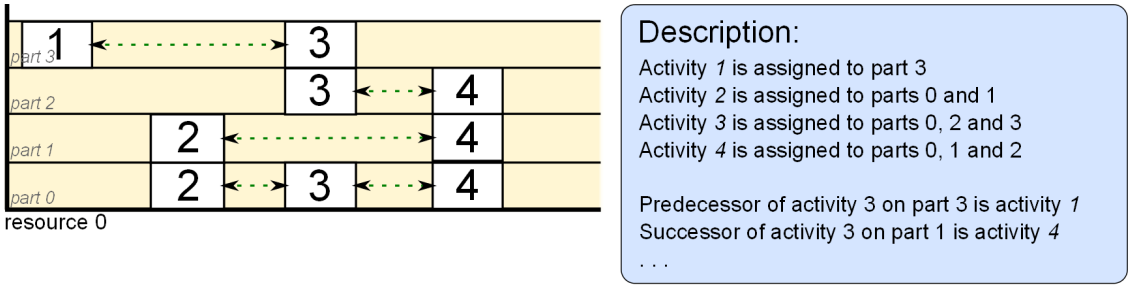


Figure 6.1: Assignment of activities and neighboring activities on resource parts

6.1.3.2 Resources

Class *ResourceExtended* implements interface *IResource* and further adds information about the first and the last activity at each resource part. Using an example depicted in Figure 6.1, the first activity at part 0 is *activity* 2, the last activity at part 2 is *activity* 4, etc.

Each resource also contains sorted list of all its milestones. Further details about milestones and an example are given below. Milestones are used for optimization purposes, primary to estimate free time slots for activities.

6.1.3.3 Milestones

Milestone represents a single moment in time, in which there is any change on the resource from the view of activities assignment. Basically, milestones represent time moments, in which activities start or finish. Milestones are used to quickly find free time slot for the actually scheduled activity.

Each milestone further contains information about assigned activities. Assigned activities are determined specifically for each resource part and represent bordering activities of the milestone. Generally, the assigned activity is such activity on resource part, whose start time is the closest to the time of the milestone, but do not exceed it. The rules used in the process of identification of assigned activities to the milestone can be easily deduced from see Figure 6.2.

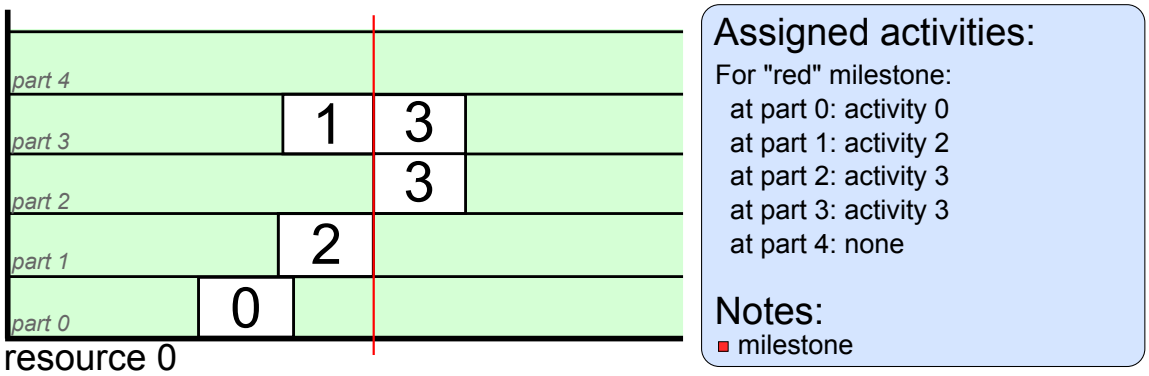


Figure 6.2: Last scheduled activities at milestone

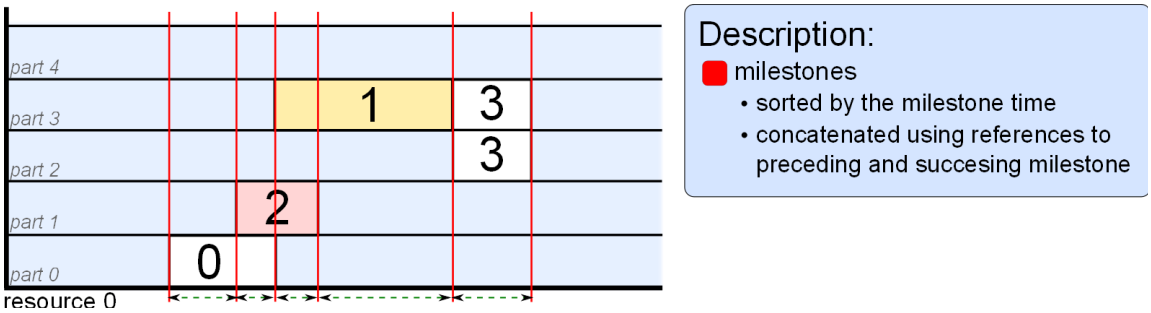


Figure 6.3: Concatenation of milestones

Milestones are kept in sorted list, where the *Key* is the time of the milestone and *Value*

is the milestone itself. Each milestone has also a reference to preceding and succeeding milestone (see Figure 6.3). Using sorted list we can find the nearest milestone for the activity's desired start time. Using concatenation of milestones, we can move on time points while finding free time slots in the process of scheduling the activities.

6.2 Implementation details

In this section, we will describe interesting implementation details of the proposed heuristic algorithm in more detail. The focus is primary paid to improvements used to optimize both the data representation and the proposed algorithm.

6.2.1 Scheduling of activity

While scheduling activity i, we must consider all the constraints - release times, deadlines, non-negative start to start time lags, sequence dependent setup times, resource constraints and activities selection. We try to find the earliest free time slots on the assigned resource using resource milestones, where activity i fits, considering all the constraints. If no time slot is found without violation of constraints, backtracking rules are applied. All necessary steps of the activity scheduling are described in following sections.

Notice, that activities are extended by reference to neighboring activities on each resource part (see Figure 6.1). Therefore, after activity is successfully scheduled, we set neighboring activities of activity i on each assigned capacity and we also redirect concatenation of neighboring activities to reflect actual state.

6.2.2 Unscheduling of activity

While unscheduling activity i, we remove it from the resource. Similarly as while scheduling the activity, we must update neighboring activities. We need to redirect predecessors on resource parts of i to successors on resource parts of i and similarly in the opposite direction.

6.2.3 Scheduling on the resource

Activities can be assigned or removed from the resource parts. We cannot schedule or unschedule only a part of an activity or its partial resource demand. Since we remember the first and the last activity on each part of the resource, we must update them after any change on the assignment of the activities. While scheduling an activity, we can simply compare the first and the last activity on assigned parts with the one being just scheduled. While unscheduling the activity, we can obtain the first and last activity from assigned activities of appropriate milestone.

The whole process of assignment (scheduling and unscheduling of activities) is solved using milestones. Milestone is a data structure used for optimization purposes and is described including its usage in the following paragraph.

6.2.4 Milestones

Since the milestones represent single time moments when there is a change in assignment of the activities on the resource, milestones have to be updated after any change of assignment of activities on the resource.

Note, that each milestone contains information about assigned activities from the view of milestone time on each part of the resource. Therefore, after any change, assigned activities have to be updated. The rules to estimate assigned activity can be clearly seen on Figure 6.2.

Although milestone is partially redundant data structure, it is useful when finding time slots to assign activities. Process of finding time slots for activity i can be expressed as:

```
calculate time window of activity
milestone := locate nearest milestone

while milestone is not null
    time intervals := estimate free time intervals
    sort time intervals by start time
```

```
    determine first start time

    while exists next start time
        try to put activity to the next start time
        if activity fits to the time slot
            return start time and assigned capacities
        end if
        determine next start time
    end while

    milestone := successor of the current milestone
end while
```

Calculation of time window $\langle w_i^s, w_i^e \rangle$ for activity i stands for estimating lower and upper bound of time interval, in which activity can be scheduled fulfilling all temporal constraints. Lower bound $w_i^s = max(\widehat{r}_i, max(s_j + l_{ji})), \forall j \in predecessors(i)$. Similarly upper bound $w_i^e = min(\widehat{d}_i, min(s_j - l_{is} + p_i)), \forall j \in successors(i)$. Only scheduled predecessors and successors are considered for the calculation.

Location of the nearest milestone *milestone* stands for finding the last milestone, whose time is lower or equal then the estimated lower bound of time interval for the activity. Locating the nearest milestone is precipitated by the fact that all milestones are in the sorted list. The constraint can be expressed as: $milestone = last(m.Time \leq w_i^s)$ for all $m \in$ resource milestones.

Since milestones are linked together and contain information about the assigned activities, we can easily determine free time slots $\langle f_s^k, f_e^k \rangle$ on each part k of the resource. We must also include the influence of sequence-dependent setup times. Found intervals are further sorted according to their start time (f_s^k) for easier processing. Let l_m^k be the last scheduled activity on the resource part k of milestone m and m' be the succeeding milestone of m. The value of lower bound is given as $f_s^k = c_{l_m^k} + st_{l_m^k i}$ and the upper bound as $f_e^k = s_{l_{m'}^k} - st_{il_{m'}^k}$. If there is no last scheduled activity on the resource part k, $f_s^k = w_s^i$. If milestone m has no successor, the upper bound is equal to the upper bound of activity's window ($f_e^k = w_e^i$).

Since we use the first and the last activities on the resource parts to prevent large propagation of the assigned activities to the milestones (see 6.2.4.1 for more details), $l^k_{m'}$ can be *null*. In this situation, we must decide, whether the assigned activity is null because of preventing propagation, or there is no scheduled activity yet on the resource part. Decision is made upon the last activity i that is the last on the resource part k ($i = last_k$). If activity i exists and $l^k_{m'}$ is null, we use activity i for the calculation.

When we have all free time intervals, we must decide, whether the activity fits inside them or not. In this step we seek the minimal possible start time of the activity provided that the activity does not violate upper bound of its interval. We must find as many free time intervals as is activity's resource demand. The finding of such start time s_i can be expressed as: $s_i = min(t \in \langle f^k_s, f^k_e - p_i \rangle) : s \geq w^i_s \wedge s + p_i \leq w^i_e$ for at least θ_i resource parts k. Activity i is then assigned to such resource parts.

If there is no enough free space for activity i, we must move to another milestone and repeat the entire process again. If no milestone is found or we move beyond estimated time window, backtracking rules are applied. An example of finding free space on the resource for a single activity is depicted on Figure 6.4.

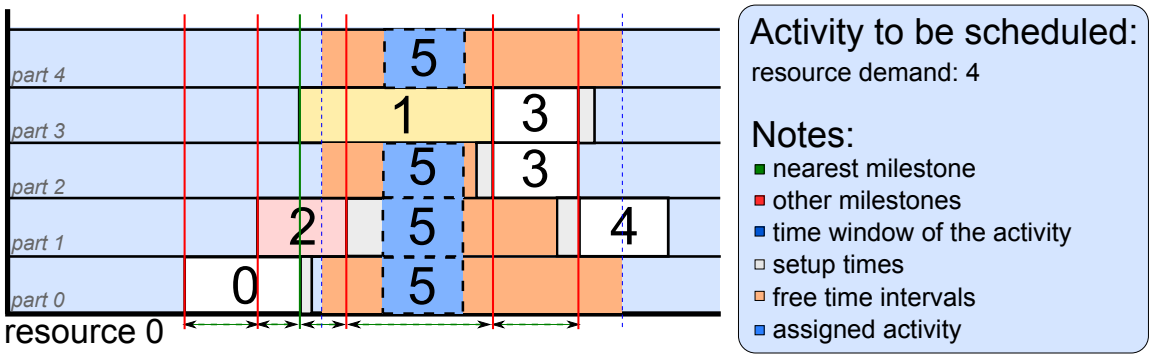


Figure 6.4: Free time intervals estimation and activity assignment

6.2.4.1 Propagation of the last scheduled activities on the resource

After an activity is unscheduled, the milestone is disposed (i.e. deleted) if and only if no activity starts or finishes in such time point. After both insertion and deletion of the

milestone, assigned activities to the milestones must be propagated for each affected resource part so the milestones contain actual values.

The propagation starts at given milestone m and using concatenation of milestones, we move to its successor m'. If there is no assigned activity on milestone m at resource part k, we use the activity from the previous milestone m. The process continues on as long as $j = null$ or $s_j < s_i$. This process is repeated for each affected resource part k which is given by resource demand of just scheduled / unscheduled activity.

Such propagation can lead in the worst case scenario to situation, when we need to update all the milestones. To prevent this behavior, we can use the last activities on the resource parts. If assigned activity i to the milestone on the resource part k is simultaneously the last activity on this part from the perspective of resource assignment, we clear the information about assigned activity for current resource part k on such milestone. This way we can finish propagation much earlier, because if we detect empty assigned activity while propagating, we can stop the process on given resource part k. This can rapidly cut-off the entire propagation time. Since the milestone can contain *null* value instead of assigned activity i if activity i is the last one on the resource part, we must interpret the *null* value correctly. Throughout the algorithm, must decide, whether *null* indicates no assigned activity or the last activity on the resource part. Utilization of this optimization scheme can reduce the computational time, but also brings more possible situations, we must handle within the algorithm.

6.3 Sliding windows

In the following paragraphs we will closely describe data representation and implementation details of the second phase of the proposed heuristic algorithm - the sliding windows. Many methods used in initial phase were further extended to consider also time windows while making the calculations. The remaining methods were implemented as new and they will be discussed in more detail.

6.3.1 Data representation

Each time window is defined by its left and right border. Time window also contains a deep copy of activities which are between its borders and deep copy of all the resources.

This way we can find different solution with no affect on the original data and apply the result only when we decide to do so. The activities are stored in a dictionary in order to quickly determine, whether any activity is inside the time window or not. Each time window further contains a set of ready activities. Both borders of the time window consist of bordering activities and milestones. Time window and its properties are depicted on Figure 5.5.

6.3.1.1 Bordering milestones

After the left border of the time window is calculated, we find the closest milestone to its time on each resource where found milestones are still outside the time window. Similarly, we locate the closest milestones to the right border of the time window. All these milestones determine the time segment on each resource where the optimization will be performed and are also used to locate the bordering activities.

6.3.1.2 Bordering activities

After all bordering milestones for both borders are found, we determine bordering activities for both borders of the time window. Bordering activities are fixed and cannot be rescheduled in any way. We must find bordering activity for each part of each resource in order to determine setup costs at time windows borders, etc. Let i be the assigned activity on milestone m at resource part k. If the start time s_i of activity i is lower then the time of the milestone m, we use activity i as the bordering activity on resource part k. Otherwise, we use the predecessor of activity i on resource part k using concatenation of activities on the resources. This way we ensure that we always select the activity which starts outside the time window. Rules to determine bordering activities are depicted in Figure 6.5

6.3.2 Implementation details

In this section we will discuss interesting and important details regarding the time window optimization in more detail.

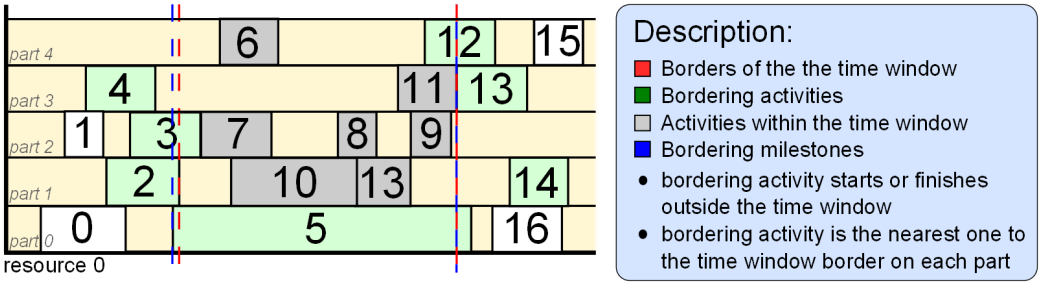


Figure 6.5: Rules to determine bordering activities of the time window

6.3.2.1 Duplication of activities and resources

Each time window contains duplicated version of all activities within the time window and all resources. Using the duplicates we can simply find new solution of the time window and still keep the original one intact. If the found solution has lower value of the objective function, it is integrated into the original solution. In other words, we merge the original solution with the optimized time window.

All auxiliary variables such as start time, assignment, etc. are cleared for the duplicated of activities and resources. Each duplicated resource contains only the default resource milestone at time 0. This way we must consider only bordering activities of the time window without any other restrictions given by the current solution. Since all auxiliary variables were cleared to default values before the solution of the time window is found, they do not possess any information related to the initial solution. All the time window data are self-sufficient. This fact enables us to simply the process of merging the time window into the original solution.

6.3.2.2 Ready activities

Each time window has its own set (dictionary) of ready activities. These activities need to be updated after any change inside the time window. The rules to recognize ready activities within the time window were discussed in section 5.3.3.2. Note, that activities

inside the time window also interact with activities outside because of time lags and sequence dependent setup times. Therefore, we must decide whether to use original activities from current solution or their duplicates inside the time window. Having all the activities inside the time window in dictionary makes the decision quite quick and simple.

6.3.2.3 Time window right border

Estimation of the closing time stands for finding a completion time of the activity which was last added to the time window without overflowing the rules given in section 5.3.1.2. Basically we locate the n-th activity on the single resource or the m-th activity considering all the resources. In each step we find activity i considering all the resources which was not added to the time window yet and has minimal completion time from all not-added activities on the resource (more precisely we find these activities from the opening time of the time window). Than we increment appropriate counters and repeat this procedure until any of thresholds is overflowed.

6.3.2.4 Selection from ready activities

Selection of activities from *Ready* set is based on the serial generation scheme. In each iteration of this phase we choose one activity from the *Ready* set and schedule it. The process of the selection is analogous to the one in initial solution but with different selection rules. First, schedule attempt of each activity from *Ready* set is evaluated. Second, an appropriate activity is chosen using selection rules. The methods are the same as described in the initial phase with a small difference: we only use dedicated delegate to evaluate the schedule attempts with an emphasis on the setup costs. This way we can make the entire procedures behave differently without rewriting its source code.

Unlike the initial phase, discrepancy is used in different levels of the search tree. If discrepancy is not used the entire process of selecting activity from the *Ready* set continues as usual with focus on the setup costs. In case we want to schedule hth activity (activity in level h of the search tree) we use discrepancy. First, each schedule attempt is evaluated and its result is stored. Second, all the results are sorted according to induced setup costs.

Finally, we choose activity at index $\left\lceil \frac{|Ready|}{2} \right\rceil$ in the sorted results, i.e. in the middle.

6.3.3 Optimization of the time window

Optimization of the time window is done in more steps. In each step, schedule for the time window is found in different way. In each step we use discrepancy for selecting an activity from the *Ready* set in single level of the search tree (see 5.6 for illustration). This way we explore more possibilities and finally choose the best one. If any activity cannot be scheduled, entire step ends with failure, i.e. we do not apply any kind of backtracking rules. Se try to schedule he same time window with discrepancy in different levels of the search tree instead.

First, we find a solution of the time window without discrepancy. Moreover, we remember the sequence of activities as they were scheduled. Then we incrementally change the level, in which we use discrepancy within the search tree, which is given partially by the sequence of scheduled activities. Let n be the number of activities in the time window, and h be the level, in which we want to use discrepancy, while h is increasing from 0. Then in each step we do not need to drop entire solution that was found in previous step. Note, that in previous step, we used discrepancy the rule in level $h-1$. So we need to unschedule only activities in the sequence of scheduled activities from index $h-1$ to index n. We can keep partial solution from index 0 to index $h-2$. This way we schedule in each step only $n-h+1$ activities instead of all n activities. This modification reduces the computational demands of this algorithm part. Notice, that after scheduling or unscheduling activities we must change many parameters, propagate last assigned activities on milestones, etc. This way we can avoid any unnecessary operations over the solution or the time window.

6.3.4 Merging of the time window with current solution

After the time window is optimized and its solution further improves the actual one, we perform merge of time window with the current solution. Merging is rather extensive operation. A part of the current solution is to be dropped and the found solution of the time window is taken instead. Notice, that all activities and all milestones are linked together

using references. In this final step, we use all the information about borders of the time window prepared before. Note, that not only assignment of the activities is changed, but also the position and count of milestones can differ (see Figure 5.7 for illustration).

Merging of two parts of the solution should be also fast enough. Therefore, we do not update and re-calculate parameters of all affected activities, milestones, etc. Instead we use the fact, that all activities and milestones has references to its successors and predecessors. This way we only replace references and concatenate objects around the time window borders without modifying all the activities and resources, i.e. original objects are dropped and new ones are used instead. Since we calculated all necessary information about activities and milestones creating the borders of time window, we can now only concatenate left border of the time window with the beginning of the time windows and similarly the end of the time window with its right border. This step is made both for activities and resource milestones. Then we need to propagate the last assigned activities around both the borders and update the first and the last activities on the resource parts. Finally, we replace references of original activities and milestones in current solution with the references from the time window. This way we only work with few bordering activities and milestones instead of re-creating data for the entire time window.

6.3.5 Completion of the algorithm

The entire process of the sliding windows optimization phase is applied to the current solution more times. The specific count of sliding windows repetitions depends on the algorithm configuration. After the first iteration of this phase the assignment of the activities has changed so the objective function is reduced. This does not mean, that the given solution cannot be further improved. Since the actual assignment is different than the one given by initial phase of the algorithm, we can apply sliding windows again and potentially reduce the value of the objective function even more. The entire proposed heuristic algorithm ends after all the iterations of the sliding windows are evaluated. The current solution corresponds to such assignment, which lead to the lowest value of the objective function during entire algorithm evaluation.

Chapter 7

Performance evaluation

This chapter is dedicated to the performance evaluation of the heuristic algorithm proposed in Chapter 5. First, the algorithm is evaluated on the randomly generated instances and compared with *IRSA* algorithm proposed by Čapek et al. [42]. Second, we have used the standard benchmarks of Brucker and Thiele [12] and he have compared our results with the results given in article of Foccaci et al. [18]. Finally, various settings of the proposed algorithm are discussed and tested on larger instances of the problem (500 and 1000 activities).

All the measurements were performed on notebook with the following hardware configuration: Intel Pentium Dual Core 2.00GHz, 4GB DDR2 800MHz RAM under operating system Windows 7 Professional, 64bit edition.

7.1 Description of random instances

Random instances are generated to compare the proposed algorithm with existing *IRSA* algorithm. All random values has uniform probability distribution. This mean, that the probability of getting each number within specified interval is always the same, i.e. $\frac{1}{|interval|}$.

Since *IRSA* does not consider resources with non-unary capacities, all the instances

contain only unary resources and activities with resource demand equal to 1. In this part of testing we do not consider the problem with its full complexity but only a specific sub-problem.

There are three different sets of generated instances. First, the performance is generated on so called *loose* instances, where finding a feasible solution is probably easy to find due to the values of release times and deadlines. Second, *middle* instances are measured, where a solution is more difficult then in case of *loose* instances. Finally, the performance is measured on *tight* instances, where a feasible solution if probably difficult to find due to release times and deadlines restriction. Each set further contains 100 instances for each of 20, 50, 100 and 200 activities. Totally 400 random instances for each set are therefore generated.

Instances are generated with following configuration: The number of activities is given by the size of generated instance. Processing time of each activity $p_i \in \langle 2, 20 \rangle$. Activities has unary resource demands, i.e. $\mathrm{R}_i^q = 1$. The values of release times and deadlines depends on the type of instances we want to generate. For example, to generate *middle* instances with 50 activities, the values lies in intervals $r_i \in \langle 0, 300 \rangle, \tilde{d}_i \in \langle 300, 600 \rangle$. and in intervals $r_i \in \langle 0, 1000 \rangle, \tilde{d}_i \in \langle 500, 1800 \rangle$ to generate *tight* instances with 200 activities. The number of resource types m lies in interval $\langle 1, 5 \rangle$. All the resources have capacity equal to 1, i.e they are unary. Setup times st_{ij} lies in interval $\langle 5, 10 \rangle$, non-negative start to start time lags l_{ij} in interval $\langle 0, 20 \rangle$. The instances are generated with following structure of the $NTNA$. If node i begins parallel branching, the output degree δ_i^+ lies in interval $\langle 5, 10 \rangle$. Similarly, if node i begins alternative branching, the output degree δ_i^+ lies in interval $\langle 2, 4 \rangle$. This way, the *Order strength* of the $NTNA$ does no exceed value 0.6 and the instances are therefore not so simple to solve.

7.2 Comparison with IRSA

For the testing of the proposed algorithm, we have used two different configurations - *Configuration I* and *Configuration II*. The specific values of both configurations are in Table 7.1. Values given in percentage determines the part of activities count in the entire instances. Parameter *Maximal size of the time window* determines the number of activities for each time

parameter	**Configuration I**	**Configuration II**
Maximal size of the time window	33%	24%
Maximal depth of the search tree	2%	1.5%
Sliding windows repetitions	2	4

Table 7.1: Configurations of the proposed algorithm

window. Parameter *Maximal depth of the search tree* states how many times the disrepancy is used within each time window. The number of repetitions of entire *Sliding windows* phase is defined by parameter *Sliding windows repetitions*.

7.2.1 Results

Following Tables 7.2 and 7.3 show results obtained by our algorithm and the IRSA algorithm over the random instances described in previous section. The IRSA algorithm settings were $budgetRatio = 6$ and $maxModifications = 4$. Columns marked as $feas.[\%]$ determine the percentage ratio of feasible solutions found by the algorithms. The values in columns TSC specify an arithmetic average value of the objective function for instances that were successfully solved by both algorithms. Columns $time[ms]$ determine the average computational time to solve a single instance regardless whether solution was found or not. Finally, the last column *impr.*[%] states the improvement of proposed heuristic algorithm over the IRSA algorithm in terms of TSC.

		IRSA			Proposed algorithm			
n	*type*	feas. [%]	TSC	time [ms]	feas. [%]	TSC	time [ms]	impr. [%]
20	loose	100	102	141	100	76	1	25.50
50	loose	100	254	344	100	215	12	15.35
100	loose	100	494	1 002	100	427	77	13.56
200	loose	94	942	3 870	100	824	261	12.53
20	medium	62	77	257	69	79	1	-2.59
50	medium	58	226	842	60	221	8	2.10
100	medium	69	386	1 722	59	371	38	3.92
200	medium	72	707	5 293	62	662	159	6.37
20	tight	44	65	230	31	70	1	-7.69
50	tight	31	183	671	32	183	7	0.00
100	tight	26	302	1 529	33	295	26	2.30
200	tight	37	597	4 666	42	592	119	0.92

Table 7.2: Comparison with IRSA using configuration I

Comparison of the proposed algorithm with IRSA algorithm was measured using two configurations. In both cases, the proposed algorithm gives better values of the objective

		IRSA			Proposed algorithm			
n	*type*	feas. [%]	TSC	time [ms]	feas. [%]	TSC	time [ms]	impr. [%]
20	loose	100	102	141	100	75	1	26.47
50	loose	100	254	344	100	215	5	4.65
100	loose	100	494	1 002	100	427	89	13.56
200	loose	94	942	3 870	100	783	643	16.87
20	medium	62	77	257	69	79	1	-2.59
50	medium	58	226	842	60	220	14	2.65
100	medium	69	386	1 722	59	376	82	2.59
200	medium	72	707	5 293	62	643	280	9.05
20	tight	44	65	230	31	70	1	-7.69
50	tight	31	183	671	32	184	14	-0.54
100	tight	26	302	1 529	33	292	41	3.31
200	tight	37	597	4 666	42	592	205	0.92

Table 7.3: Comparison with IRSA using configuration II

function from a global perspective. Both algorithms solve the random instances with similar percentage of found feasible solutions. The biggest difference between these two algorithm is in computational demands. The proposed algorithm solves the instances in much shorter time, sometimes even in $\frac{1}{100}$ of the time it took IRSA to solve. The proposed heuristic algorithm is very fast and effective, which meets our demands on the algorithm.

7.3 Description of data sets of Brucker and Thiele [12]

The proposed heuristic algorithm is also compared using the general Job shop instances by Brucker and Thiele [12] with the results of Foccaci et al. [18]. General Job shop is a very specific and much easier sub-problem of the problem considered in this thesis. The performance is evaluated on problems with unary resources and no temporal constraints such as release times, deadlines and time lags. General Job shop also does not cover alternative process plans.

7.4 Comparison with algorithm of Focacci

Since we do not have possibility to run the algorithm proposed by Foccaci et al. [18], we compare only values of the objective function, not the computational time. Following Table 7.4 shows the comparison of our algorithm with the one published by Foccaci et al. [18].

	Focacci		Proposed algorithm		
Set	TSC	C_{max}	TSC	C_{max}	$TSC_{impr.}$ [%]
t2-ps12	1 530	1 445	1 010	1 920	27.45
t2-ps13	1 430	1 658	1 330	2 172	7.00
t2-pss12	1 220	1 362	950	1 599	22.13
t2-pss13	1 140	1 522	1 150	1 610	-0.87
avg.	**1 330**	**1 497**	**1 110**	**1 825**	**16,54**

Table 7.4: Comparison with Focacci

7.4.1 Results

Compared with the algorithm described by Foccaci et al. [18], the proposed algorithm improves the value of the total setup costs by 16% in average. The price for the better value of the TSC is the higher value of the makespan (C_{max}).

7.5 Large instances

The proposed heuristic algorithm is designed to handle large instances of the considered problem, which is very complex. So far we have evaluated the performance over relatively small instances and also without all the constraints considered in this thesis. In this section, we focus on randomly generated large instances with 500 and 1000 activities with full problem complexity, i.e. all the constraints such as non-negative start to start time lags, sequence-dependent setup times and non-unary resources and activities with different resource demands. The instances are generated such that the average *Order strength* is 0.6 and also the average *Resource strength* is equal to 0.6. The parameters of the random instance generator are similar as described in section 7.1, but the resources are not unary. Capacity of the resource $\theta_q; \forall q \in \mathcal{R}$ lies in interval $\langle 1, 6 \rangle$ and resource demands R_i^q of activity $i; \forall i \in \mathcal{A}_\mathcal{E}$ lies in interval $\langle 1, 4 \rangle$.

Again, performance is measured for both mentioned configurations of the algorithm (see Table 7.1. Up to our knowledge, there is no similar solved problem to the one considered in this thesis and therefore we cannot compare the results with another approaches or solutions.

Following Table 7.5 shows the measured results for both considered configurations of the algorithm. For comparison, measurement is taken also on smaller random instances.

	Configuration I		Configuration II		
n	TSC	time [ms]	TSC	time [ms]	$TSC_{impr.}$ [%]
20	158	1	158	1	0.00
50	400	16	395	61	1.25
100	859	62	849	142	1.16
200	1 651	352	1 615	811	2.18
500	2 845	1 497	2 841	4 173	0.14
1000	7 794	14 912	7 841	25 800	-0.60

Table 7.5: Large instances of the considered problem

7.6 Configuration of the algorithm

Proper configuration of the algorithm has a great influence on its outcome. We may satisfy with only feasible solution which is found in short time or to optimize it from the view of the total setup costs. We can also affect the quality of the objective function in many ways.

The first parameter which can significantly improve the objective function is the size of the time window. The best results were measured for the time window with dynamic size given by the size of a testing instance. The best ratio $\frac{size\ of\ the\ time\ window}{size\ of\ the\ instance}$ was estimated to the 0.33. The upper bound of the time window size was estimated to the 70 activities ($size\ of\ the\ time\ window = Max(size\ of\ the\ instance \cdot 0.33, 70)$. For example, for the instance consisting of 100 activities we optimize time windows with 33 activities within.

The second parameter determines the depth of the search tree used in the time windows. The deeper the search tree is, the longer it takes to decide. Therefore, for deeper search tree it is reasonable to increase the size of the time window (lesser count of sliding window segments). While using discrepancy in the search tree we temporarily escape from the local optimum in order to obtain possibly better combination of activities assignment to the resources, which could lead to lesser value of the objective function of the entire time window. Even for large instances, the results are not being significantly improved with growing levels where the discrepancy rules were used. We estimate the ratio $\frac{max\ discrepancy\ level}{size\ of\ the\ instance}$ to the value of 0.02.

Finally, the number of repetitions of the entire sliding window procedure was considered. After the first run of the sliding window phase is done, activities might change their

assignment. In the next steps the value of the objective function can be further improved by repeating entire procedure over again. Unlike the previously mentioned configuration parameters, the number of repetitions affects the execution time at most. For smaller instances we can afford to repeat the entire procedure even 10 times, but for larger instances (200 activities and more) the computational demands grow very high.

The dependence of computational time on the size of the solved instance is depicted in Figure 7.1. The graph shows results measured in Section 7.5. Notice, that optimal settings described in preceding paragraphs (used as *Configuration I*) do not give much worse values of the objective function than using settings described as *Configuration II* considering the substantial difference in computational demands of both algorithm configurations.

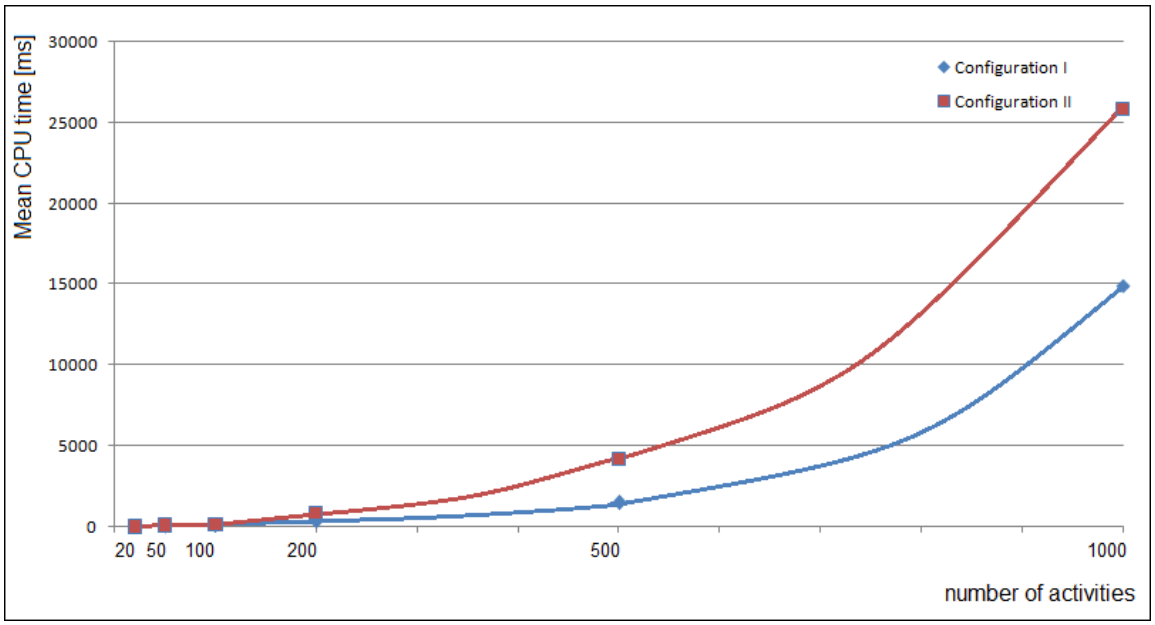


Figure 7.1: Computational time versus size of the instances

Chapter 8

Conclusions

This thesis deals with very complex and extensive problem, that can be expressed as $PS|nestedAlt, l_{ij}^{min}, ST_{SD}, r_j, \tilde{d}_j|TSC$ using Brucker et al. [9] notation or equivalently as $m1|nestedAlt, min, ST_{SD}, r_j, \tilde{d}_j|TSC$ using Herroelen et al. [23] notation. Up to our knowledge, this problem was not dealt before.

The problem considered in this thesis covers many nowadays manufacture constraints such as release times, deadlines, non-negative start to start time lags, sequence-dependent setup times and non-unary resources (resources with capacities greater than one) and is motivated by real manufacture process in the printing company. The goal is to minimize the total setup costs (TSC).

First, an extensive research has been done in order to find related problems to the one considered in this thesis. Second, $MILP$ model was formulated. The proposed mathematical model can be used to find exact solutions of the small instances of the considered problem.

Next, we have developed a new heuristic algorithm to solve large instances of the considered problem. Since the problem is very complex and the instances are very large, the algorithm should be effective and solve the entire problem in short time with satisfying value of the objective function. This algorithm solves large instances consisting of 1000 activities

in very short time. Feasible solution is found within 1 second and optimized one within 8 seconds using appropriate algorithm configuration.

Finally, we have compared the proposed heuristic algorithm with different ones using both random generated and standard instances. The algorithm overperformed both compared algorithms by Čapek et al. [42] and Foccaci et al. [18].

8.1 Summary

The new proposed heuristic algorithm is very fast and solves the instances with low computational demands and still the value of the objective function of the found solution is quality. Moreover, this algorithm exceeded both algorithm to which it was compared.

8.2 Further improvements

The proposed algorithm consists of two main phases - finding an initial solution and the local optimization. The second phase is taken only if the feasible solution is found in the first phase. Since the first phase is fast enough, it can be further strengthened to find more feasible solutions in the set of all instances.

By extending the backtracking scheme we could find more feasible solutions within the set of all instances. The backtracking scheme might be further improved for example by dynamic priority selection rules.

The second phase of the proposed heuristic algorithm called *Sliding windows* is quite scalable. With not so extensive modifications it can be designed to run in parallel. That way we can optimize more disjoint time windows at the same time. Doing so, we could repeat the entire procedure more times in order to obtain better value of the objective function.

In the second phase, the alternatives are already fixed. We could change alternatives

within the found initial solution and then run the sliding windows algorithm part. This way, different alternative plans can be evaluated even in the second phase of the proposed algorithm.

Bibliography

[1] Akkiraju, R., Keskinocak, P., Murthy, S., Wu, F., 2001. An agent-based approach for scheduling multiple machines. Applied Intelligence 14, pp. 135–144.

[2] Allahverdi, A., Ng, C., Cheng, T., Kovalyov, M. Y., 2008. A survey of scheduling problems with setup times or costs. European Journal of Operational Research 187, pp. 985–1032.

[3] Ballestín, F., Barrios, A., Valls, V., 2009. An evolutionary algorithm for the resource-constrained project scheduling problem with minimum and maximum time lags. Journal of Scheduling 14, pp. 391–406.

[4] Barták, R., Čepek, O., 2007. Temporal networks with alternatives: Complexity and model. In: Proceedings of the Twentieth International Florida Artificial Intelligence Research Society Conference (FLAIRS), Florida, USA. AAAI Press, pp. 641–646.

[5] Barták, R., Čepek, O., 2008. Nested temporal networks with alternatives: recognition and tractability. In: Proceedings of the 2008 ACM Symposium on Applied Computing (SAC), Ceara, Brazil. ACM, pp. 156–157.

[6] Beck, J. C., Fox, M. S., 1999. Scheduling alternative activities. In: Proceedings of the sixteenth national conference on Artificial intelligence. American Association for Artificial Intelligence (AAAI), pp. 680–687.

[7] Beck, J. C., Fox, M. S., 2000. Constraint-directed techniques for scheduling alternative activities. Artificial Intelligence 121, pp. 211–250.

[8] Blazewicz, J., Ecker, K. H., Pesch, E., Schmidt, G., Weglarz, J., 1996. Scheduling Computer and Manufacturing Processes. Springer-Verlag New York, Inc.

[9] Brucker, P., Drexl, A., Möhring, R., Neumann, K., Pesch, E., 1999. Resource-constrained project scheduling: Notation, classification, models, and methods. European Journal of Operational Research 112, pp. 3–41.

[10] Brucker, P., Hilbig, T., Hurnik, J., 1999. A branch and bound algorithm for a single-machine scheduling problem with positive and negative time-lags. Discrete Applied Mathematics 94, pp. 77–79.

[11] Brucker, P., Knust, S., 1998. Complexity results for single-machine problems with positive finish-start time-lags. Computing 63, pp. 219–316.

[12] Brucker, P., Thiele, O., 1996. A branch and bound method for the general-shop problem with sequence dependent setup-times. Operations Research Spektrum 18, pp. 145–161.

[13] Choi, I.-C., Choi, D.-S., 2002. A local search algorithm for jobshop scheduling problems with alternative operations and sequence-dependent setups. Computers & Industrial Engineering 42, pp. 43–58.

[14] Deblaere, F., Demeulemeester, E., Herroelen, W., 2011. Reactive scheduling in the multi-mode rcpsp. Computers & Operations Research 38, pp. 63–75.

[15] Demeulemeester, E. L., Herroelen, W., 2002. Project scheduling: A research handbook. Kluwer Academic Publishers.

[16] Drießel, R., Moench, L., 2009. Scheduling jobs on parallel machines with sequence-dependent setup times, precedence constraints, and ready times using variable neighborhood search. In: Proceedings of the International Conference on Computers & Industrial Engineering. IEEE, pp. 273–278.

[17] Dvořák, J., 2010. Interaktivní ganttův diagram. Bachelor's thesis, Czech Technical University in Prague.

[18] Foccaci, F., Laborie, P., Nuijten, W., 2000. Solving scheduling problems with setup times and alternative resources. In: Artificial Intelligence Planning Systems 2000 Proceedings (AIPS), pp. 1–10.

[19] Gacias, B., Artigues, C., Lopez, P., 2010. Parallel machines scheduling with precedence constraints and setup times. Computers & Operations Research 37, pp. 2141–2152.

[20] Geoffrion, A. M., Graves, G. W., 1976. Scheduling parallel production lines with changeover costs: Practical application of a quadratic assignment/lp approach. Operations Research 24, pp. 595–610.

[21] Hamdi, I., Loukil, T., 2011. Minimizing the makespan in the permutation flowshop problem with minimal and maximal time lags. In: Proceedings of the International Conference on Communications, Computing and Control Applications (CCCA). IEEE, pp. 1–6.

[22] Harvey, W. D., Ginsberg, M. L., 1995. Limited discrepancy search. In: Proceedings IJCAI 95.

[23] Herroelen, W., Demuelemeester, E., Reyck, B. D., 1997. A classification scheme for project scheduling problems. Katholieke Universiteit Leuven, pp. 1–25.

[24] Kadrou, Y., M.Najid, N., 2006. A new heuristic to solve rcpsp with multiple execution modes and multi-skilled labor. In: Proceedings of the IMACS Multiconference on Computational Engineering in Systems Applications (CESA). IEEE, pp. 1–8.

[25] Kis, T., 203. Job-shop scheduling with processing alternatives. European Journal of Operational Research 151, pp. 307–322.

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

[27] Kopanos, G. M., Puigjaner, L., Georgiadis, M. C., 2011. Resource-constrained production planning in semicontinuous food industries. Computers and Chemical Engineering 35, pp. 1–16.

[28] Krüger, D., Schol, A., 2009. A heuristic solution framework for the resource constrained (multi-)project scheduling problem with sequence-dependent transfer times. European Journal of Operational Research 197, pp. 492–508.

[29] Lee, Y. H., Pinedo, M., 1997. Scheduling jobs on parallel machines with sequence-dependent setup times. European Journal of Operational Research 100, pp. 464–474.

[30] Leung, C., Wong, T., Maka, K., Fung, R., 2010. Integrated process planning and scheduling by an agent-based ant colony optimization. Computers & Industrial Engineering 59, pp. 166–180.

[31] Li, X., Zhang, C., Gao, L., Li, W., Shao, X., 2010. An agent-based approach for integrated process planning and scheduling. Expert Systems with Applications 37, pp. 1256–1264.

[32] Lombardi, M., Milano, M., 2009. A precedence constraint posting approach for the rcpsp with time lags and variable durations. Computer Science 5732/2009, pp. 569–583.

[33] Mika, M., Waligóra, G., Węglarz, J., 2008. Tabu search for multi-mode resource-constrained project scheduling with schedule-dependent setup times. European Journal of Operational Research 187, pp. 1238–1250.

[34] Mirabi, M., 2010. A hybrid simulated annealing for the single-machine capacitated lot-sizing and scheduling problem with sequence-dependent setup times and costs and dynamic release of jobs. The International Journal of Advanced Manufacturing Technology 54, pp. 795–808.

[35] Neumann, K., Schwindt, C., Zimmermann, J., 2003. Project scheduling with time windows and scarce resources. Springer-Verlag Berlin Heidelberg.

[36] Reyck, B. D., Herroelen, W., 1999. The multi-mode resource-constrained project scheduling problem with generalized precedence relations. European Journal of Operational Research 119, pp. 538–556.

[37] Ruiz, R., Stützle, T., 2008. An iterated greedy heuristic for the sequence dependent setup times flowshop problem with makespan and weighted tardiness objectives. European Journal of Operational Research 187, pp. 1143–1159.

[38] Salewski, F., Schirmer, A., Drexl, A., 1997. Project scheduling under resource and mode identity constraints: Model, complexity, methods, and application. European Journal of Operational Research 102, pp. 88–110.

[39] Shao, X., Li, X., Gao, L., Zhang, C., 2009. Integration of process planning and scheduling - a modified genetic algorithm-based approach. Computers & Operations Research 36, pp. 2082–2096.

[40] Tasgetiren, M. F., Panb, Q.-K., Liang, Y.-C., 2009. A discrete differential evolution algorithm for the single machine total weighted tardiness problem with sequence dependent setup times. Computers & Operations Research 36, pp. 1900–1915.

[41] Van Peteghem, V., Vanhoucke, M., 2009. Using resource scarceness characteristics to solve the multi-mode resource-constrained project scheduling problem. Tech. rep., Faculty of Economics and Business Administration (Ghent University).

[42] Čapek, R., Šůcha, P., Hanzálek, Z., 2012. Production scheduling with alternative process plans. European Journal of Operational Research 217, pp. 300–311.

[43] Wang, L., Fang, C., 2012. An effective estimation of distribution algorithm for the multi-mode resource-constrained project scheduling problem. Computers & Operations Research 39, pp. 449–460.

[44] Wang, L., Wand, M., 1997. A hybrid algorithm for earliness-tardiness scheduling problem with sequence dependent setup time. In: Proceedings of the 36th Conference on Decision & Control. IEEE, pp. 1219–1223.

[45] Yuan, X.-M., Khoo, H. H., Spedding, T. A., Bainbridge, I., Taplin, D. M. R., 2004. Minimizing total setup cost for a metal casting company. Winter Simulation Conference, pp. 1189–1194.

Appendix A

Content of the included CD

```
+ Instances
|-- + Benchmarks
|-- + Random
+ Source code
. dvoraj42.pdf
. README.txt
```

Folder *Instances* contains all instances used in performace evaluation. Instances by Brucker et al. [9] are placed in subfolder *Benchmars*. Subfolder *Random* contains *loose*, *middle* and *tight* set of instances with size 20, 50, 100 and 200 activities. Further, this folder contains large random instances of the considered problem.

Folder *Source code* contains source code of the algorithm in Microsoft Visual Studio 2011 project. Note, that some of the given files are authored by my supervisor Roman Čapek, especially basic classes such as interfaces and solution representation. This way we have common input data structures and we can thus use the same data for more different algorithms in order to compare their efficiency, etc. All files are included with header which states among others authorship.