TU WIEN Informatics

Automated Project Scheduling in Real-World Test Laboratories

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

Dipl.-Ing. Florian Mischek
Registration Number 01025898

to the Faculty of Informatics

at the TU Wien

Advisor: Priv.-Doz. Dr. Nysret Musliu

The dissertation has been reviewed by:

Andreas Fink

Gabriela Ochoa

Vienna, 14th July, 2022

Florian Mischek

Erklärung zur Verfassung der Arbeit

Dipl.-Ing. Florian Mischek

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

Wien, 14. Juli 2022

Florian Mischek

Acknowledgements

I would like to thank everyone who supported me during my PhD and the writing of this thesis. First and foremost, my gratitude goes to my advisor, Priv.-Doz. Dr. Nysret Musliu, whose encouragement and feedback was invaluable for me. Thanks also to my colleagues, both at the institute and at our industrial partner, who were always available to discuss ideas and provide helpful comments on my work.

Finally, a heartfelt thank you to my partner, Nadja, for her unwavering support and patience with me, and my parents who encouraged me to follow my own path throughout my whole life.

The financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development and the Christian Doppler Research Association is gratefully acknowledged.

Kurzfassung

Komplexe Projektplanungsprobleme entstehen in verschiedensten Umgebungen und die Qualität dieser Pläne hat einen immensen Einfluss auf die Effizienz, Pünktlichkeit, sowie die Kosten eines Projekts. Das manuelle Erstellen eines geeigneten Plans ist allerdings aufwändig, teuer und fehleranfällig, weshalb im Lauf der letzten Jahrzehnte viel Forschungsarbeit in die Automatisierung dieses Prozesses geflossen ist. Aufgrund der Vielzahl an verschiedenen Szenarien, jedes mit seinen eigenen Anforderungen und Eigenschaften, die wiederum spezialisierte Lösungsansätze benötigen, ist manuelle Planung immer noch weit verbreitet. Ein solcher Bereich, in dem automatisierte Planungslösungen bisher noch nicht Fuß fassen konnten, sind industrielle Testlabore, in denen Produkte anhand einer Vielzahl von Leistungsbeschreibungen und internationalen Normen geprüft und zertifiziert werden.

In dieser Dissertation stellen wir ein neues und komplexes Projektplanungsproblem vor, das die speziellen Anforderungen von industriellen Testlaboren abdeckt. Es beinhaltet heterogene Ressourcen mit unterschiedlichen Qualifikationen und Ausstattungen, sowie eine neuartige Anforderung, kurze Aktivitäten zu größeren Einheiten zusammenzufassen, wodurch der Plan flexibler wird und unnötige Aufwände vermieden werden können. Wir beschreiben dieses Problem formal und stellen ein Set von Benchmark-Instanzen unterschiedlicher Größe zur Verfügung, das sowohl reale Instanzen direkt aus dem Labor unseres Industriepartners beinhaltet, als auch zufällig generierte Instanzen, die mit einem neuen und unterschiedlich konfigurierbaren Instanzengenerator erstellt wurden.

Sowohl für dieses Planungsproblem, als auch für ein spezielles Teilproblem, das in der Praxis regelmäßig zu lösen ist, untersuchen wir unterschiedliche innovative Lösungsansätze. Insbesondere entwickeln wir eine Reihe von Nachbarschaften, die unterschiedliche Aspekte des Problems abdecken und gemeinsam die Basis für verschiedene Suchheuristiken bilden. In einer umfassenden empirischen Evaluierung konnten wir zeigen, dass vor allem Simulated Annealing äußerst erfolgreich darin ist, effizient hochqualitative Lösungen für das Planungsproblem zu finden und exakten sowie hybriden Lösungsmethoden auf großen Instanzen teils deutlich voraus ist.

Außerdem entwickeln wir in dieser Arbeit verschiedene problemunabhängige Algorithmen, sogenannte Hyper-Heuristiken, die in der Lage sind, sich automatisch an neue Umstände anzupassen und sogar vorher unbekannte Optimierungsprobleme effizient lösen können. Unsere Hyper-Heuristiken erzielten gute Ergebnisse auf verschiedenen namhaften

NP-schweren Problemen, insbesondere unsere auf Reinforcement Learning basierenden Lösungsansätze. Um diese Algorithmen auch auf das Testlabor-Planungsproblem anwenden zu können, modellieren wir eine Reihe von heuristischen Operatoren für dieses Problem. In unseren Experimenten konnten wir die Effektivität sowie die Vielseitigkeit unserer Hyper-Heuristiken zeigen.

Abschließend beschreiben wir auch die Anwendung unserer automatisierten Lösungsmethoden im Labor unseres Industriepartners, wo sie erfolgreich zur Planung der verschiedenen Laboraktivitäten eingesetzt werden. Für viele andere Planungsprobleme aus der Literatur ist die Anwendung der theoretischen Modelle und Ergebnisse in realen Planungsszenarien eine große Hürde, die oftmals nicht überschritten wird. Daher fassen wir die Herausforderungen zusammen, die uns im Laufe der Einführung automatisierter Planung begegnet sind, sowie die Erkenntnisse, die wir im Rahmen dieses Prozesses gewinnen konnten, als Hilfestellung für zukünftige Anwendungen von automatisierten Planungstechniken in der Praxis.

Abstract

Complex project scheduling problems arise in many different settings and the quality of a schedule typically has a tremendous impact on the efficiency, timeliness, and cost of a project. However, creating good schedules manually is expensive and error-prone, which is why a lot of research has been done on automating this process. Due to the multitude of different problem settings, each with their own unique set of requirements and features which require specialized solution approaches, manual scheduling is still widespread in many areas. One such area is that of industrial test laboratories, where products need to be tested and certified according to a wide range of specifications and international norms.

In this thesis, we introduce a new and complex project scheduling problem that is designed to model the requirements of industrial test laboratories. It features heterogeneous resources with different capabilities as well as a unique grouping aspect, which serves to improve the flexibility of the schedules and reduce unnecessary overheads. We provide a formal definition of this problem, as well as a set of benchmark instances of varying sizes, both real-world instances taken directly from the laboratory of our industrial partner and randomly generated ones using a new and configurable instance generator.

For this problem, as well as a subproblem that is relevant in practice, we develop new and innovative solution approaches. We propose a set of neighborhood structures that cover different aspects of the problems and together serve as the basis for several state-of-the-art search heuristics. Our extensive empirical evaluation shows that simulated annealing is very successful in finding good solutions for this problem and outperforms exact and hybrid methods on larger instances.

In addition, we develop different problem-independent hyper-heuristic algorithms, which are capable of automatically adapting to new instances and even completely unseen problem domains. They achieved a good performance on several well-known NP-hard optimization problems, in particular our hyper-heuristics based on reinforcement learning. We also provide a comprehensive set of low-level heuristics for the test laboratory scheduling problem which allows it to be solved with our hyper-heuristics and we were able to show the effectiveness and generality of this approach.

Finally, this thesis also describes the deployment of our algorithms at the laboratory of our industrial partner, where they are successfully used in practice for scheduling lab operations. For many other problems studied in the literature, adopting the theoretical methods and results into real-world practice has proved difficult and often unsuccessful. We describe the challenges we encountered in this process, as well as the lessons learned and recommendations for future deployments.

Contents

CHAPTER 1

Introduction

An important task in project management is project scheduling, i.e. the assignment of time and resources to different activities, subject to an often large number of complex constraints and objectives. In many settings, performing this task manually requires considerable expertise and effort, while still remaining expensive and error-prone. This has led to a large interest in the automation of the scheduling process over many decades.

The standard problem in this area is the Resource Constrained Project Scheduling Problem (RCPSP), where activities have to be scheduled such that the makespan, the total duration of the schedule, is minimized, while precedence constraints and resource capacities are respected. Unfortunately, this is often not enough to model the requirements of many settings, so a large number of extensions and variants of the RCPSP have been proposed in the past to be able to model these requirements [HB10, HB22]. In general, these variants are NP-hard to solve, as is the RCPSP itself, and many different techniques have been investigated to solve them. Examples of such methods include exact approaches such as Dynamic Programming (DP) [HRD98], Integer Linear Programming (MIP) [CSE18] and most recently Constraint Programming (CP) [SS16], but also heuristic approaches such as dispatching (or priority) rules [VW07], metaheuristics [PPB20], Genetic Algorithms [GMR08], or hyper-heuristics [KKA14].

Still, there are many areas remaining for which no or only limited automated project scheduling approaches exist. One such area is that of industrial test laboratories, where devices are subjected to a large number of different tests, typically to test or certify their performance according to (inter)national norms or industry standards. This setting poses several unique challenges for automated project scheduling approaches, in addition to the usually large instance sizes and time periods: Employee qualifications and requirements on equipment properties are complex and often binding. This complicates the resource assignment aspect of the problem, as not all units of a certain resource may be suitable for any given test. In addition, the tests typically require a considerable amount of setup work, which may be shared between related tests. Efficient batching procedures are necessary to avoid excessive overheads. Finally, the test specifications may change on short notice, which requires both a degree of flexibility in the produced schedules as well as the ability to rapidly update the schedules with new data.

Despite the challenging nature of project scheduling in industrial test laboratories, schedules are still created manually in many laboratories. Partially this is due to the absence of theoretical models and solution approaches that are able to cover their requirements, and partly because the existing theoretical work simply does not make the jump to deployment into actual practice. This gap has been identified for many related settings [Sca20], and there are numerous challenges to be tackled until automated scheduling approaches can actually be employed in a real-world laboratory.

1.1 Research questions

The main goal of this thesis is to develop innovative and efficient scheduling methods that can be used to automatically produce high-quality schedules for real-world industrial test laboratories, while being able to automatically adapt themselves to different environments and problem settings. Towards this goal, we aim to answer the following research questions:

- How can we describe and formally model the requirements of scheduling in a real-world industrial test laboratory? In which ways do they differ from those in existing project scheduling problems?
- What solution techniques are suitable for solving the resulting problems? How does the performance of these techniques change across different instance types?
- How can we enable our methods to adapt themselves to new and unseen problem variants and domains?
- How can these automated scheduling algorithms be deployed in practice and what lessons can we learn from the challenges encountered during this process?

1.2 Contributions

The main contributions of this thesis are the following:

- We introduce the Test Laboratory Scheduling Problem (TLSP), a complex project scheduling problem based on the requirements in an industrial test laboratory. We give a formal definition of the TLSP and identify a practically relevant subproblem, the TLSP-S. For these problems, we also provide a set of diverse benchmark instances, both randomly generated and challenging real-world instances, as well as a configurable instance generator. All benchmark instances are publicly available online.

- We provide complexity proofs to show that even finding feasible solutions for either of the two problem variants is already NP-hard.
- We develop a set of modular and efficient new local search neighborhood structures that cover different aspects of the TLSP. We then employ these neighborhood structures within several state-of-the-art search heuristics and perform an extensive empirical evaluation and comparison, including automated tuning of the search parameters.
- Using the previously introduced neighborhood structures and search heuristics, we develop a comprehensive set of low-level heuristic operators that allow us to model the TLSP as a problem domain for high-level and problem-independent selection hyper-heuristics.
- We introduce our own selection hyper-heuristics, one approach based on Self-Adaptive Large Neighborhood Search and another based on Reinforcement Learning. We empirically analyze and compare different variants for several algorithm components on a standard benchmarking set of common NP-hard problem domains. We show that our algorithms are able to yield good results within the top 10 state-of-the-art approaches on these problem domains and in addition provide competitive results to the best problem-specific algorithms on our newly introduced TLSP problem domain formulation.
- We describe the deployment of our algorithms within an automated scheduling system in the test laboratory of our industrial partner, who perform electromagnetic compatibility tests on car control units and other components, as a case study for the successful use of automated scheduling in industrial laboratories. This includes an analysis of the usage of objective weights both in theory and in daily practice, as well as challenges and lessons learned in this real-world application.

1.3 Publications

The material that is presented in this thesis has been included in several publications.

1.3.1 Journals

- Florian Mischek and Nysret Musliu. A local search framework for industrial test laboratory scheduling. Annals of Operations Research 2021. [MM21b]
- Florian Mischek, Nysret Musliu, and Andrea Schaerf. Local search approaches for the test laboratory scheduling problem with variable task grouping. Journal of Scheduling 2021. [MMS21a]
- Philipp Danzinger, Tobias Geibinger, David Janneau, Florian Mischek, Nysret Musliu, and Christian Poschalko. A System for Automated Industrial Test Laboratory

Scheduling. Accepted for publication at ACM Transactions on Intelligent Systems and Technology 2022. [DGJ^{+}21]

1.3.2 Conferences

- Florian Mischek and Nysret Musliu. A local search framework for industrial test laboratory scheduling (extended abstract). PATAT 2018. [MM18]
- Florian Mischek, Nysret Musliu, and Andrea Schaerf. Local search neighborhoods for industrial test laboratory scheduling with flexible grouping (extended abstract). PATAT 2021. [MMS21b]
- Florian Mischek and Nysret Musliu. Investigating hyper-heuristics for real-world test laboratory scheduling (abstract). EURO 2022. [MM22a]
- Florian Mischek and Nysret Musliu. Reinforcement learning for cross-domain hyper-heuristics. IJCAI 2022. [MM22b]

In addition, the author of this thesis was a co-author of the following articles:

- Tobias Geibinger, Florian Mischek, and Nysret Musliu. Investigating constraint programming for real world industrial test laboratory scheduling. CPAIOR 2019. [GMM19b]
- Philipp Danzinger, Tobias Geibinger, Florian Mischek, and Nysret Musliu. Solving the test laboratory scheduling problem with variable task grouping. ICAPS 2020. [DGMM20]
- Tobias Geibinger, Florian Mischek, and Nysret Musliu. Constraint logic programming for real-world test laboratory scheduling. AAAI 2021. [GMM21]
- Tobias Geibinger, Lucas Kletzander, Matthias Krainz, Florian Mischek, Nysret Musliu, and Felix Winter. Physician scheduling during a pandemic. CPAIOR 2021. [GKK^{+}21]
- Tobias Geibinger, Florian Mischek, and Nysret Musliu. Investigating constraint programming and hybrid methods for real world industrial test laboratory scheduling. Submitted to Journal of Scheduling. [GMM19a]

1.4 Organization

In Chapter 2, we formally define the Test Laboratory Scheduling Problem (TLSP) and describe our instance generator together with the publicly available benchmark instances. This chapter also contains the complexity analysis with an NP-hardness proof. Heuristic solution methods based on local search are introduced in Chapter 3, including search heuristics, neighborhood structures, and an extensive empirical evaluation. Our hyper-heuristic approaches for cross-domain optimization are described and evaluated on a diverse set of problem domains in Chapter 4. Here we also develop our portfolio of low-level operators necessary to formulate TLSP as a new problem domain for selection hyper-heuristics. In Chapter 5, we describe the deployment of our algorithms as part of the automated scheduling system in use in the test laboratory of our industrial partner. Finally, Chapter 6 contains a summary and our conclusions, together with an outlook on possible future work.

CHAPTER 2

The Industrial Test Laboratory Scheduling Problem

Automated solution and optimization approaches to project scheduling problems have been a popular topic of research for a long time. The standard problem in this area is the Resource-Constrained Project Scheduling Problem (RCPSP). In the RCPSP, a number of activities need to be scheduled, such that precedence constraints and capacities of some renewable resources are satisfied and the makespan of the resulting schedule is minimized. Numerous variants and extensions of this problem have been proposed to model the varied requirements of different settings.

One such setting is scheduling activities in industrial test laboratories. Typically, scheduling in these laboratories is still performed manually by human experts due to the high complexity of the requirements on the schedules, both operational and legal, which is expensive, time intensive, and error-prone. Therefore, there is a high need for automated scheduling solutions. In this chapter, we describe the Test Laboratory Scheduling Problem (TLSP), a complex real-world scheduling problem which was developed specifically for the previously described setting in collaboration with our industrial partner. The TLSP is an extension of the RCPSP with several challenging features:

The goal is to find a schedule for a large number of tasks which are distributed into several independent projects. First, the tasks have to be grouped into larger units called jobs, which derive their properties from the tasks they contain. In general, tasks within a job are executed sequentially, but without any predefined order, and therefore jobs have to fulfill all requirements of their tasks over their whole duration. A detailed discussion of the properties of a job and the motivation behind it can be found in Section 2.2.2.

Afterwards, the jobs have to be scheduled (i.e. assigned a mode, timeslots and resources), subject to various constraints. Besides several well-known features of (extensions of) the RCPSP from the literature, such as multiple execution modes and time windows, the

TLSP also features additional constraints imposed by the real-world problem setting. In particular, RCPSP and its variants usually assume that units of a resource are identical and homogeneous and therefore can be used to cover the demand of all tasks. Due to this, an assignment of individual units to tasks is not necessary. Exceptions exist, e.g. in a paper by Dauzère-Pérès et al. [DPRL98] or by Bellenguez and Néron [BN05], and sometimes an equivalent effect is achieved by introducing additional modes for each possible resource assignment (e.g. by Schwindt and Trautmann [ST00] and Bartels and Zimmermann [BZ09]). However, this is practical only for a single resource with very few available values per task. In contrast, TLSP features multiple heterogeneous resources which are shared among all projects, with very general restrictions on which units can be used to perform a job, and potentially large demands.

In addition, jobs in TLSP can be linked to each other, indicating that these jobs must be performed by the same employees[1]. To the best of our knowledge, a similar concept was used only by Salewski et al. [SSD97] and Drexl et al. [DNPS00], where the mode of several jobs is required to be the same.

In real-life practice, it is commonly the case that the grouping of tasks into jobs is already known and only a solution for the scheduling part of the problem is required. This gives rise to a restricted problem variant we denote as TLSP-S, which has a (fixed) list of jobs as additional input, but otherwise follows the same restrictions as TLSP.

The content discussed in this chapter was first published in [MM21b].

In the next section, we discuss other relevant works on project scheduling problems, in particular extensions of the RCPSP. In Section 2.2 we give a formal definition of the TLSP, including input data, constraints, and objectives. We also provide a complexity proof showing the NP-hardness of the TLSP based on a reduction from the RCPSP in Section 2.3. In Section 2.4 we describe the new benchmark instances we provide for the TLSP as well as the instance generator used to generate the random instances. Finally, we sum up our contributions of this chapter in Section 2.5.

2.1 Project scheduling problems in the literature

As the standard problem in the area of project scheduling, the RCPSP has seen vast amounts of work over several decades, including multiple surveys. Here, we focus on literature on problem variants and extensions of the RCPSP that are relevant for the TLSP. Papers and surveys mainly dealing with solution approaches are discussed in the next chapter (Section 3.1).

Brucker et al. [BDM$^+$99] proposed a classification scheme for project scheduling problems using an $\alpha|\beta|\gamma$ notation, similar to existing classification schemes for machine scheduling.

[1]This can be used to model multi-step procedures that require knowledge from previous steps, such as ensuring that documentation of completed tasks is prepared by the employees who actually performed the tasks.

Problem feature	RCPSP extension	Related concept
Task grouping	-	[TS05], [WWH12]
Setup times	Survey: [MWW06]	Batch scheduling, e.g. [ST00], [PK00]
Multiple modes	MRCPSP, Survey: [WJMW11], e.g. [WKS+16]	
Multiple projects	RCMPSP, e.g. [VPLP+19], MMRCMPSP by [WKS+16]	
Heterogeneous resources	[DPRL98], MSPSP by [BN05]	e.g. [ST00], [BZ09]
Linked tasks	-	[SSD97], [DNPS00]
***Objective criteria*:**		
Project completion time	-	Flow time [GMR08]
Due dates	Tardiness, e.g. [VP00]	
Number of employees	-	RIP, e.g. [DK01]

Table 2.1: Main features of the TLSP and their correspondence in other variants of the RCPSP. The last column includes papers that contain similar concepts, which may not be directly applicable for the TLSP.

In this scheme, α describes the resource environment, β activity characteristics, and γ the objective function(s) used.

A comprehensive overview of existing extensions and variants of the RCPSP up until 2010 was performed by Hartmann and Briskorn [HB10]. The authors also published an updated survey focusing on developments in the last decade in 2022 [HB22].

Other surveys of different problem variants were published by Mika et al. [MWW15], as well as Habibi et al. [HBS18].

Many of the extensions to the RCPSP contain problem features that are also found in the TLSP, or at least related concepts. Table 2.1 shows a selection of these features, as well as references from the literature for RCPSP variants that contain these particular features or similar concepts. While some aspects (e.g. multiple modes, setup times) are already well known, others have few (e.g. heterogeneous resources) or no (task grouping, linked tasks) direct correspondence in previously studied problems, to the best of our knowledge.

In the following, we will provide more details about these features and their treatment in the literature, describing both similarities and differences to TLSP where appropriate.

The first of these, and one of the most well-known, is the multi-mode RCPSP (MRCPSP), originally formulated by Elmaghraby [Elm77]. It allows each activity to be performed in one of several modes which can affect duration and resource requirements. Surveys focused solely on MRCPSP formulations are provided by Węglarz et al. [WJMW11], and more recently Noori and Taghizadeh [NT18].

Setup times have been studied extensively in various forms, here we refer to the survey by Mika et al. [MWW06]. Of note is that in particular sequence- or schedule-dependent setup times can be used to model (sequential) batch processing, treated e.g. by Schwindt and Trautmann [ST00] or Potts and Kovalyov [PK00], which is related to the task grouping formalism of the TLSP. Both approaches attempt to remove overheads due to setup times by grouping multiple smaller activities into larger units. The main difference is that in batch processing approaches, the batches arise implicitly from the completed schedule and the scheduled activities correspond to tasks. In contrast, in the TLSP jobs are created explicitly from tasks, which are not directly scheduled at all. In addition, tasks within a job can be freely reordered without inducing conflicts in the TLSP (the formal definition is given in Section 2.2.2). Some exceptions exist, where batches are modeled explicitly as a separate step, in particular in the area of composite manufacturing (e.g. by Tang and Beck [TB20], Winter et al. [WMMP21] or Malapert et al. [MGR12]), but these often use a simple scheduling model with a single (type of) machine. In these kinds of problems, activities within a batch are also typically processed in parallel, which is not the case in the TLSP.

The only paper using a similarly explicit approach to grouping small activities into larger units is by Trautmann and Schwindt [TS05]. They decompose a batch-scheduling problem into two independent subproblems, of which the first assembles the batches and the second produces a schedule for the previously created batches.

Multiple separate projects, with project-specific constraints and objective functions, appear in the Resource Constrained Multi-Project Scheduling Problem (RCMPSP). Works dealing with this problem are e.g. by Gonçalves et al. [GMR08] and Villafáñez et al. [VPLP$^+$19]. Wauters et al. [WKS$^+$16] introduced the Multi-Mode RCMPSP (MMRCMPSP), which combines both multiple modes and multiple projects and was used for the MISTA 2013 challenge.

As mentioned in the previous section, the default assumption in the RCPSP and most of its variants is that units of a resource are interchangeable, i.e. each unit can be used for each activity. A notable exception is by Dauzère-Pérès et al. [DPRL98], who employ very general resource requirements which are even more flexible than those of the TLSP. Unfortunately, the authors make heavy use of (a variant of) permutation schedules, which are unsuitable for the TLSP.

A second example of heterogeneous resources can be found in the Multi-Skill RCPSP (MSPSP), introduced by Bellenguez and Néron [BN05]. In the MSPSP, each resource unit has a number of skills, and resource requirements are given as a multi-set of required skills. Despite this different resource model, a distinct objective function and several other discrepancies, the MSPSP is still quite closely related to TLSP-S. Typical instances for MSPSP contain up to 90 activities, compared to 300+ jobs for instances of practical size for TLSP-S.

Alternatively, heterogeneous resources like those in the TLSP can sometimes be modeled as an MRCPSP by adding additional modes [BN05]. Since each potential subset of assigned resources has to be encoded in a separate mode, this is only feasible for single resources and small numbers of required units and therefore unsuitable for TLSP.

As far as we are aware, the concept of linked jobs, which need to have the same resource units assigned, has not been applied to any variant of the RCPSP before. As mentioned in the introduction, [SSD97] and [DNPS00] deal with a somewhat related concept, where several activities have to be scheduled using the same mode. Of course, this is easier to add on top of the RCPSP, since the assumption of homogeneous resources can be retained.

The RCPSP itself and many of its variants use the total schedule makespan, i.e. the time until the end of the last activity, as the objective criterion to be minimized. However, the makespan alone is not always sufficient to judge the quality of a schedule for real-world applications. This holds in particular for multi-project problem variants, where different problems may not be performed completely simultaneously, but have their own release dates and deadlines. Consequently, different objectives have been proposed, by themselves or in combination, typically either alternative time-based criteria or objectives based on the assigned resources [HB22]. One of the main impacts of alternative (multi-)objective functions is that permutation schedules [Har98], which represent a solution as a permutation of activities and are used in many state-of-the-art works on the RCPSP, are no longer guaranteed to yield optimal solutions.

Multi-project formulations often use a function of the individual project makespans (e.g. the MMRCMPSP uses the sum of the normalized project makespans). This is still different from the project duration objective used for the TLSP: The project makespan depends only on the last activity in the project, while the project duration can also be decreased by delaying the start of a project's first activity. To the best of our knowledge, there is no variant of the RCPSP using this criterion as (part of) its objective function.

Regarding the other objectives of the TLSP, the due dates are equivalent to the minimization of activity tardiness (e.g. [VP00]). Minimizing the number of employees assigned to each project is similar to a per-project version of the resource investment problem (RIP), which has been treated e.g. by Drexl and Kimms [DB08].

Different approaches exist for the combination of multiple objectives, as described in the survey on multi-objective project scheduling problems by Ballestín and Blanco [BB15]. Where the objectives are clearly ordered in importance, a lexicographic objective function can be used (e.g. [WKS$^+$16]). Otherwise, a tradeoff is usually necessary where reducing the value of one objective results in an increase for another. A weighted sum over all objectives is a simple and flexible approach that allows fine-grained control over the relative importance of the objectives. Problem formulations employing such an objective function include papers by Nudtasomboon and Randhawa [NR97], and by Voß and Witt [VW07]. Another popular approach when dealing with multiple objectives is the search for Pareto-efficient solutions (e.g. [YA13]), i.e. solutions which are optimal in the sense that decreasing the value of one objective is only possible by increasing the value of at least one other objective. Solutions to problems with these kinds of objectives are typically given as Pareto-fronts, sets of individually Pareto-optimal solutions. However, finding such solutions sets requires specialized solution approaches that are able to simultaneously consider solution quality and diversity, compared to a weighted sum formulation which is supported by more general optimization techniques. In addition, such an approach would require an additional decision step by a human expert to determine the actually accepted schedule, which is unsuitable for completely automated scheduling solutions.

Finally, we mention two problems dealing with scheduling activities in laboratories, due to the similarities in their overall setting with the TLSP: Bartels and Zimmermann [BZ09] deal with scheduling tests of experimental vehicles. The described problem contains several aspects and constraints similar to the TLSP. However, it uses a different resource model (in particular regarding destructive tests) and uses the number of employed vehicles as the main optimization criterion. Polo Mejia et al. [PMAAL17] developed an integer linear program for scheduling research activities for a nuclear laboratory, using a problem formulation derived from the MSPSP, but with (limited) preemption of activities. Despite a similar setting, the requirements are unfortunately incompatible with those of the TLSP.

Overall, to the best of our knowledge there is no variant of the RCPSP that can fully model the requirements of the TLSP(-S). Regarding TLSP-S, the closest related problem is probably the MSPSP due to similar restrictions on the availability of resources to each activity. However, trying to model the very general restrictions of the TLSP(-S) for the MSPSP would result in a prohibitively large number of skills, not to speak of other features like linked jobs or objective criteria, which are not included in the MSPSP at all. Therefore, a new approach is required to model and solve the TLSP(-S).

2.2 Problem definition

In TLSP, a list of projects is given, which each contain several tasks. For each project, the tasks must be partitioned into a set of jobs, with some restrictions on the feasible partitions. Then, those jobs must each be assigned a mode, time slots and resources. The properties and feasible assignments for each job are calculated from the tasks contained within.

A solution of TLSP is a schedule consisting of the following parts:

- A list of jobs, composed of one or multiple similar tasks within the same project.
- For each job, an assigned mode, start and end time slots, the employees scheduled to work on the job, and an assignment to a workbench and equipment.

The quality of a schedule is judged according to an objective function that is the weighted sum of several individual objectives and should be minimized (Section 2.2.4). Among others, these include the number of jobs and the total completion time (start of the first job until end of the last) of each project.

2.2.1 Input parameters

A TLSP instance can be split into three parts: The laboratory *environment*, including a list of resources, a list of *projects* containing the tasks that should be scheduled together with their properties and the current state of the *existing schedule*, which might be partially or completely empty.

Environment

In the laboratory, resources of different kinds are available that are required to perform tasks:

- *Employees* $e \in E = \{1, \ldots, |E|\}$ who are qualified for different types of tasks.
- A number of *workbenches* $b \in B = \{1, \ldots, |B|\}$ with different facilities. (These are comparable to machines in shop scheduling problems.)
- Various auxiliary lab *equipment* groups $G_g = \{1, \ldots, |G_g|\}$, where g is the group index. These each represent a set of similar devices. The set of all equipment groups is called G^*.

The scheduling period is composed of discrete *time slots* $t \in T = \{0, \ldots, |T| - 1\}$. Each time slot represents half a day of work.

Tasks are performed in one of several *modes* labeled $m \in M = \{1, \ldots, |M|\}$. The chosen mode influences the following properties of tasks performed under it:

- The *speed factor* v_m, which will be applied to the task's original duration.
- The number of *required employees* e_m.

Projects and Tasks

Given is a set P of *projects* labeled $p \in \{1, \dots, |P|\}$. Each project contains *tasks* $pa \in A_p$, with $a \in \{1, \dots, |A_p|\}$. The set of all tasks (over all projects) is $A^* = \bigcup_{p \in P} A_p$.

Each task pa has several properties:

- It has a *release date* α_{pa} and both a *due date* $\bar{\omega}_{pa}$ and a *deadline* ω_{pa}. The difference between the latter is that a due date violation only results in a penalty to the solution quality, while deadlines must be observed.
- $M_{pa} \subseteq M$ is the set of *available modes* for the task.
- The task's *duration* d_{pa} (in time slots, real-valued). Under any given mode $m \in M_{pa}$, this duration becomes $d_{pam} \coloneqq d_{pa} * v_m$.
- Most tasks must be performed on a workbench. This is indicated by the boolean parameter $b_{pa} \in \{0, 1\}$. If required, this workbench must be chosen from the set of *available workbenches* $B_{pa} \subseteq B$.
- Similarly, pa requires *qualified employees* chosen from $E_{pa} \subseteq E$. The required number depends on the mode. A further subset $E^{Pr}_{pa} \subseteq E_{pa}$ is the set of *preferred employees*.
- Of each equipment group $g \in G^*$, the task requires r_{pag} devices, which must be taken from the set of *available devices* $G_{pag} \subseteq G_g$.
- A list of direct *predecessors* $\mathcal{P}_{pa} \subseteq A_p$, which must be completed before the task can start. Note that precedence constraints can only exist between tasks in the same project.

Each project's tasks are partitioned into *families* $F_{pf} \subseteq A_p$, where f is the family's index. For a given task pa, f_{pa} gives the task's family. Only tasks from the same family can be grouped into a single job.

Additionally, each family f is associated with a certain *setup time* s_{pf}, which is added to the duration of each job containing tasks of that family.

Finally, each project p may define *linked tasks*, which must be assigned the same employee(s). Linked tasks are given by the equivalence relation $L_p \subseteq A_p \times A_p$, where two tasks pa and pb are linked if and only if $(pa, pb) \in L_p$.

Initial schedule

All problem instances include an initial (or base) schedule, which may be completely or partially empty. This schedule can act both as an initial solution and as a baseline, placing limits on the schedules of employees and tasks, in particular by defining fixed assignments that must not be changed.

Provided is a set of jobs J^0, where each job $j \in J^0$ contains the following assignments:

- The tasks in the job: $\dot{A}_j$
  - A *fixed* subset of these tasks $\dot{A}_j^F \subseteq \dot{A}_j$. All fixed tasks of a job in the base schedule must also appear together in a single job in the solution.
- The mode assigned to the job: $\dot{m}_j$
- The start and end times of the job: $\dot{t}_j^s$ resp. $\dot{t}_j^c$
- The resources assigned to the job:
  - Workbench: $\dot{b}_j$
  - Employees: $\dot{E}_j$
  - Equipment: $\dot{G}_{gj}$ for equipment group g

Except for the tasks, each individual assignment may or may not be present in any given job. Fixed tasks are assumed to be empty, if not given. In all other cases, missing assignments will be referred to using the value ϵ. Time slots and employees can only be assigned if also a mode assignment is given.

A subset of these jobs are the *started jobs* J^{0S}. A started job $j^s \in J^{0S}$ must fulfill the following conditions:

- It must contain at least one fixed task. It is assumed that the fixed tasks of a started job are currently being worked on.
- Its start time must be 0.
- It must contain resource assignments fulfilling all requirements.

A started job's duration does not include a setup time. In the solution, the job containing the fixed tasks of a started job must also start at time 0. Usually, the resources available to the fixed tasks of a started job are additionally restricted to those assigned to the job, to avoid interruptions of ongoing work in case of a rescheduling.

2.2.2 Jobs and Grouping

For various operational reasons, tasks are not scheduled directly. Instead, they are first grouped into larger units called *jobs*. A single job can only contain tasks from the same project and family.

Jobs have many of the same properties as tasks, which are computed from the tasks that make up a job. The general principle is that within a job, tasks are not explicitly ordered or scheduled; therefore the job must fulfill all requirements of each associated task during its whole duration.

The motivation behind this restriction, which deliberately overconstrains the schedule, is due to a combination of conditions in the laboratory of our industrial partner:

- Tasks of the same family usually have equivalent or very similar requirements in practice.
- Many tasks only cover a small fraction of a timeslot (e.g. half an hour out of a four-hour timeslot). Scheduling tasks directly to timeslots would therefore incur unacceptable overheads due to rounding. An alternative solution to this problem would be shorter timeslots, which would conflict with the flexible working times in the lab.
- Related formalisms, e.g. schedule-dependent setup times [MWW06], would be difficult to apply since the actual setup time between two tasks may depend on multiple resources.
- Tasks frequently need to be reordered or delayed. The chosen formulation guarantees that this is always possible within a job, adding a measure of flexibility to the schedule. Results by Wilson et al. [WWH12] indicate that this flexibility can be useful in the presence of delays during the execution of a schedule.

Let $J = \{1, \ldots, |J|\}$ be the set of all jobs in a solution and $J_p \subseteq J$ be the set of jobs of a given project p. Then for a job $j \in J$, the set of tasks contained in j is $\dot{A}_j$. j has the following properties:

$$\tilde{p}_j \quad \text{and} \quad \tilde{f}_j$$

are the project and family of j.

$$\tilde{\alpha}_j := \max_{pa \in \dot{A}_j} \alpha_{pa}, \quad \tilde{\bar{\omega}}_j := \min_{pa \in \dot{A}_j} \bar{\omega}_{pa}, \quad \tilde{\omega}_j := \min_{pa \in \dot{A}_j} \omega_{pa}$$

are the release date, due date and deadline of j, respectively.

$$\tilde{M}_j := \bigcap_{pa \in \dot{A}_j} M_{pa}$$

is the set of available modes.

$$\tilde{d}_{jm} := \left\lceil (s_{p_j f_j} + \sum_{pa \in \dot{A}_j} d_{pa}) \, * \, v_m \right\rceil$$

is the (integer) duration of the job under mode m. The additional setup time is added to the total duration of the contained tasks.

$$\tilde{b}_j := \max_{pa \in \dot{A}_j} b_{pa}$$

is the required number of workbenches ($\tilde{b}_j \in \{0, 1\}$).

$$\tilde{B}_j := \bigcap_{pa \in \dot{A}_j} B_{pa}$$

are the available workbenches for j.

$$\tilde{E}_j := \bigcap_{pa \in \dot{A}_j} E_{pa}$$

are the employees qualified for j.

$$\tilde{E}_j^{Pr} := \bigcap_{pa \in \dot{A}_j} E_{pa}^{Pr}$$

are the preferred employees of j.

$$\tilde{r}_{jg} := \max_{pa \in \dot{A}_j} r_{pag}$$

are the required units of equipment group g.

$$\tilde{G}_{jg} := \bigcap_{pa \in \dot{A}_j} G_{pag}$$

are the available devices for equipment group g.

$$\tilde{\mathcal{P}}_j := \{k \in J \setminus \{j\} : \exists pa \in \dot{A}_j, pb \in \dot{A}_k \text{ s.t. } pb \in \mathcal{P}_{pa}\}$$

is the set of predecessor jobs of j. Finally,

$$\tilde{L}_p := \{(j, k) \in J \times J : j \neq k \wedge \exists pa \in \dot{A}_j, pb \in \dot{A}_k \text{ s.t. } (pa, pb) \in L_p\}$$

defines the linked jobs in project p.

In addition, a solution contains the following assignments for each job:

- $\dot{t}_j^s \in T$ the scheduled start time slot
- $\dot{t}_j^c \in T$ the scheduled end time slot (exclusively)
- $\dot{m}_j \in M$ the mode in which the job should be performed
- $\dot{b}_j \in B$ the workbench assigned to the job (ϵ if no workbench is required)
- $\dot{E}_j \subseteq E$ the set of employees assigned to the job
- $\dot{G}_{jg} \subseteq G_g$ the set of assigned devices from equipment group g

2.2.3 Constraints

Any feasible schedule must satisfy a number of constraints. For the purpose of modeling these constraints, we introduce additional notation: The set of *active jobs* at time t is defined as $\mathcal{J}_t := \{j \in J : \dot{t}_j^s \leq t \wedge \dot{t}_j^c > t\}$.

H1: Job assignment Each task must be assigned to exactly one job.

$$\begin{aligned} &\forall p \in P, pa \in A_p : \\ &\quad \exists! j \in J \text{ s.t. } pa \in \dot{A}_j \end{aligned}$$

H2: Job grouping All tasks contained in a job must be from the same project and family.

$$\begin{aligned} &\forall j \in J, pa \in \dot{A}_j : \\ &\quad p = \tilde{p}_j \\ &\quad f_{pa} = \tilde{f}_j \end{aligned}$$

H3: Fixed tasks Each group of tasks assigned to a fixed job in the base schedule must also be assigned to a single job in the solution.

$$\begin{aligned} &\forall j^0 \in J^0 : \\ &\quad \exists j \in J \text{ s.t. } \dot{A}^F_{j^0} \subseteq \dot{A}_j \end{aligned}$$

H4: Job duration The interval between start and end of a job must match the job's duration.

$$\begin{aligned} &\forall j \in J : \\ &\quad \dot{t}^c_j - \dot{t}^s_j = \tilde{d}_{j\dot{m}_j} \end{aligned}$$

H5: Time Window Each job must lie completely within the time window from the release date to the deadline.

$$\begin{aligned} &\forall j \in J : \\ &\quad \dot{t}^s_j \geq \tilde{\alpha}_j \\ &\quad \dot{t}^c_j \leq \tilde{\omega}_j \end{aligned}$$

H6: Task precedence A job can start only after all prerequisite jobs have been completed.

$$Constraint \begin{aligned} &\forall j \in J, k \in \tilde{\mathcal{P}}_j : \\ &\quad \dot{t}^c_k \leq \dot{t}^s_j \end{aligned}$$

H7: Started jobs A job containing fixed tasks of a started job in the base schedule must start at time 0.

$$\begin{aligned} &\forall j \in J, j^s \in J^{0S} : \\ &\quad t^F_{j^s} = 1 \wedge \dot{A}^F_{j^s} \subseteq \dot{A}_j \implies \dot{t}^s_j = 0 \end{aligned}$$

H8: Single assignment At any one time, each workbench, employee and device can be assigned to at most one job.

$$\begin{aligned}&\forall b \in B, t \in T :\\ &\quad |\{j \in \mathcal{J}_t : \dot{b}_j = b\}| \leq 1\\ &\forall e \in E, t \in T :\\ &\quad |\{j \in \mathcal{J}_t : e \in \dot{E}_j\}| \leq 1\\ &\forall g \in G^*, d \in G_g, t \in T :\\ &\quad |\{j \in \mathcal{J}_t : d \in \dot{G}_{jg}\}| \leq 1\end{aligned}$$

H9a: Workbench requirements Each job requiring a workbench must have a workbench assigned.

$$\begin{aligned}&\forall j \in J :\\ &\quad \dot{b}_j = \epsilon \iff \tilde{b}_j = 0\end{aligned}$$

H9b: Employee requirements Each job must have enough employees assigned to cover the demand given by the selected mode.

$$\begin{aligned}&\forall j \in J :\\ &\quad |\dot{E}_j| = e_{\dot{m}_j}\end{aligned}$$

H9c: Equipment requirements Each job must have enough devices of each equipment group assigned to cover the demand for that group.

$$\begin{aligned}&\forall j \in J, g \in G^* :\\ &\quad |\dot{G}_{jg}| = \tilde{r}_{jg}\end{aligned}$$

H10a: Workbench suitability The workbench assigned to a job must be suitable for all tasks contained in it.

$$\begin{aligned}&\forall j \in J :\\ &\quad \dot{b}_j = \epsilon \vee \dot{b}_j \in \tilde{B}_j\end{aligned}$$

H10b: Employee qualification All employees assigned to a job must be qualified for all tasks contained in it.

$$\begin{aligned}&\forall j \in J :\\ &\quad \dot{E}_j \subseteq \tilde{E}_j\end{aligned}$$

H10c: Equipment availability The devices assigned to a job must be taken from the set of available devices for each group.

$$\begin{aligned}&\forall j \in J, g \in G^* :\\ &\quad \dot{G}_{jg} \subseteq \tilde{G}_{jg}\end{aligned}$$

H10d: Mode availability [2] The mode assigned to a job must be available for all tasks contained in it.

$$\begin{aligned}&\forall j \in J:\\&\quad \dot{m}_j \in \tilde{M}_j\end{aligned}$$

H11: Linked jobs Linked jobs must be assigned exactly the same employees.

$$\begin{aligned}&\forall p \in P, (j,k) \in \tilde{L}_p:\\&\quad \dot{E}_j = \dot{E}_k\end{aligned}$$

2.2.4 Objectives

The following objectives are used to evaluate the quality of a feasible solution. They arise from the business requirements of our industrial partner and have been formulated in close cooperation with them. In this thesis and prior publications, we also use the term soft constraints for objectives, whose violation should be minimized.

Each soft constraint i contributes a penalty value to the solution quality denoted as C^i. The total objective value of a solution is defined as the weighted sum $\sum_i w^i C^i$ over all soft constraint violations, with individual objective weights w^i.

S1: Number of jobs The number of jobs should be minimized.

$$C^{S1} := |J|$$

S2: Employee project preferences The employees assigned to a job should be taken from the set of preferred employees.

$$\begin{aligned}&\forall j \in J:\\&\quad C_j^{S2} := |\{e \in \dot{E}_j : e \notin \tilde{E}_j^{Pr}\}|\end{aligned}$$

S3: Number of employees The number of employees assigned to each project should be minimized.

$$\begin{aligned}&\forall p \in P:\\&\quad C_p^{S3} := |\bigcup_{j \in J_p} \dot{E}_j|\end{aligned}$$

S4: Due date The internal due date for each job should be observed.

$$\begin{aligned}&\forall j \in J:\\&\quad C_j^{S4} := \max(\dot{t}_j^c - \tilde{\bar{\omega}}_j, 0)\end{aligned}$$

[2] This constraint was not originally listed in [MM21b], but only described in the text. For completeness, we add it here.

S5: Project completion time The total completion time (start of the first test to end of the last) of each project should be minimized.

$$\forall p \in P :$$
$$C_p^{S5} := \max_{j \in J_p} t_j^c \; - \; \min_{j \in J_p} t_j^s$$

Objective S1 favors fewer, longer jobs over more fragmented solutions. This helps reducing overhead (fewer setup periods necessary, rounding of fractional durations) and increases the flexibility of the schedule, since tasks within a job can be freely reordered. Also, it reduces the complexity of the final schedule, both for the employees performing the actual tasks and any human planners in those cases where manual corrections or additions become necessary.

In practice in the lab of our industrial partner, it has proved efficient to have only few employees cover the tests of a single project, due to the presence of project-specific conditions and procedures as well as the need for continual coordination and communication with other parts of the laboratory as well as external clients. This is captured by objective S3.

Objective S4 makes the schedule more robust by encouraging jobs to be completed earlier than absolutely required, so they can still be finished on time in case of delays or other disturbances.

Finally, objective S5 also helps reduce overheads, as longer timespans between the tests of a project would require additional effort to become familiar with project-specific test procedures, as well as storage space for the devices in between tests.

A note on terminology: The standard objective for variants of RCPSP is the makespan, which appears in the literature under many different names, such as (schedule, project, or total) duration, flow time, and also completion time. For the purpose of the TLSP, completion time of a project denotes the timespan from the scheduled start of its first job until the scheduled end of its last job. In contrast, the makespan is counted from a fixed time point, such as the start of the scheduling period or the earliest release time of a project. As a result, the makespan can only be reduced by decreasing the end of the last activity, while the project completion time in the TLSP can also be reduced by postponing the start of a project's first activity.

Of course, the choice of objective weights determines the relative importance of the objectives. In practical applications, we expect them to be chosen interactively according to the current situation. Chapter 5 contains a discussion of the impact of different weight choices as well as their usage in practice in the laboratory of our industrial partner. To provide a common baseline for our evaluations, we have assumed a uniform weight of $w^i = 1$ for all objectives.

2.2.5 The TLSP-S problem

A practically relevant subproblem of the TLSP deals with the case where a grouping of tasks into jobs is already provided for each project, which cannot be changed by the solver. Thus, the goal is to find an assignment of a mode, time slot and resources to each (given) job, such that all constraints are fulfilled and the objective function is minimized. Since this variant focuses on the scheduling part of the problem, we denote it as *TLSP-S*.

In TLSP-S, the number and properties of jobs (see Section 2.2.2) are fixed and can be precomputed from the properties of the tasks they contain.

Without loss of generality, we assume that this fixed initial grouping is provided via the initial schedule. Thus, an instance for TLSP-S can also be given as input to a solver for TLSP and vice-versa, as long as the initial schedule contains a valid job grouping for all tasks.

For TLSP-S, constraints H1-H3 will always be trivially satisfied and can therefore be ignored. Similarly, the penalty induced by objective S1 is constant, since the number of jobs cannot be modified. We still include this penalty in our results for comparability of the solutions with instances of TLSP.

2.3 Complexity analysis of TLSP

In this section, we provide a reduction from RCPSP to TLSP, to show that TLSP is indeed a generalization of RCPSP. Since RCPSP is known to be NP-hard [BLK83], this also shows the NP-hardness of TLSP.

RCPSP can be defined as follows [HB10]: Input is a list of activities $A = \{1, \ldots, |A|\}$. Each activity a has a duration d_a, a list of predecessors $P_a \subset A$, which must be completed before a can start, and resource requirements r_{ak} for each resource $k \in K = \{1, \ldots, |K|\}$. Each resource has a capacity c_k, which denotes the number of available units per timeslot. The goal is to find a start time for each activity, such that precedence constraints are fulfilled, the capacity of each resource is not exceeded in any timeslot and the total makespan (i.e. the end of the last activity in the schedule) is minimal.

For this reduction, we use the decision variant of RCPSP (which is also NP-hard [BLK83]), which takes an integer s as additional input and asks whether a schedule with makespan $\leq s$ exists.

Given an arbitrary instance of RCPSP $\mathcal{I}$, we can then construct a corresponding instance of TLSP as follows:

- It contains a single project p_1.
- For each activity a in $\mathcal{I}$, we create a corresponding task in p_1 with the same duration as a. Each task has its own family (with a setup time of 0).
- Precedences between activities in $\mathcal{I}$ can be directly translated to the tasks of p_1.

- Each task has a release date of 0 and both a target date and a deadline of s.
- There is a single mode with speed factor 1 and no required employees. This mode is available for all tasks.
- For each resource k in $\mathcal{I}$, we create a new equipment group g_k, containing c_k devices.
- Equipment requirements for each task are determined as follows: A task requires r_{ak} devices of group g_k, where a is the activity in $\mathcal{I}$ that corresponds to the task. All devices in each group are available for all tasks.
- There are no workbenches or employees. Tasks do not require workbenches.
- There are no linked tasks.
- The base schedule is empty.

Since each family contains only a single task, the job grouping of each feasible schedule is uniquely determined: There is a separate job for each task, which has exactly the same properties (regarding duration, resource requirements, precedences, linked jobs) as the task it contains. For any activity a in $\mathcal{I}$, there must be exactly one corresponding job j_a, which is the one containing the task corresponding to a in the above construction.

Given a schedule (identified by the set of jobs J) that is a feasible solution for the TLSP instance constructed above, it is easy to see that it directly corresponds to a valid schedule $\mathcal{S}$ for the RCPSP instance $\mathcal{I}$, where each activity a starts at the same time as $j_a \in J$. Precedences are satisfied by construction and the capacity of each resource cannot be exceeded at any time, since otherwise at least one device would have to be assigned to two overlapping jobs in J at the same time (pidgeonhole principle). Further, all jobs in J must end before the deadline s. Since both start and duration of abilities and the corresponding jobs are equal, this also means that all activities in $\mathcal{S}$ must end before s, i.e. the makespan of $\mathcal{S}$ is at most s.

In the other direction, for any instance $\mathcal{I}$ of RCPSP, with a schedule S of makespan at most s, we can find a feasible schedule J for the corresponding instance of TLSP constructed as above. As before, each job $j_a \in J$ should start (and end) at the same time as the corresponding activity a in $\mathcal{S}$. Precedences and time windows are then satisfied by construction. Next, we will show that we can always find an equipment assignment for each job that fulfills both the equipment requirements (constraint H9c) and the single assignment constraint (H8): We look at each job j in order of increasing start time. Let $J_{<j}$ be all jobs considered before j which overlap the start of j, $A_{<j}$ be the activities in $\mathcal{S}$ corresponding to those jobs and a be the activity corresponding to j. Since $\mathcal{S}$ is feasible, the activities in $A_{<j}$ can require a total of at most $c_k - r_{ak}$ units of each resource k (the remaining r_{ak} units are required for a). Therefore, there can be at most $c_k - r_{ak}$ devices assigned to jobs in $J_{<j}$. It follows that there are at least r_{ak} devices left to be assigned to j, which is enough to cover its requirements (H9c). Since jobs are considered in order

of increasing start time, there can also be no other jobs not in $J_{<j}$ that overlap j and have those devices assigned. It follows that constraint H8 must be fulfilled at every step. All other constraints are trivially satisfied.

In conclusion, a schedule for the RCPSP instance with makespan $\leq s$ exists if and only if the corresponding TLSP instance is feasible. Since the reduction can be done in polynomial time, this proves that even the problem of finding any feasible solution for TLSP is already NP-hard.

If the base schedule is not empty, but instead the (unique) feasible job grouping is provided as additional input, we immediately arrive at an instance for TLSP-S. As before, each activity in $\mathcal{I}$ corresponds directly to both exactly one task and exactly one job in the constructed instance. Therefore, the same argument as for TLSP can be used to show the NP-hardness also of TLSP-S.

2.4 New instances for TLSP

In this section, we introduce an instance generator, which can be used to randomly generate instances for both TLSP and TLSP-S based on real-world data. We also propose two sets of new and publicly available instances for TLSP(-S) assembled using our instance generator. Finally, we describe three real-world instances taken from our industrial partner, which are also available online in anonymized form.

2.4.1 Instance generator

In order to be able to generate instances of specific size and complexity on demand, we developed an instance generator for TLSP. It randomly generates instances of various sizes that are based on the real-world data in the laboratory of our industrial partner and can be configured to produce instances of various sizes and properties. The generator was written in Java.

To generate a new instance, one has to specify the number of expected projects and the length of the scheduling period, plus optionally various configuration parameters that refine the desired properties of the instance. From this, an instance is generated as follows: First, the laboratory environment, including the available resources, is defined according to the desired number of projects and the length of the scheduling period. Then, the required number of projects is generated, together with a set of jobs for each project. This is used to populate the reference solution, which is guaranteed to be a feasible solution for the final problem instance. In the third step, task properties (resource requirements and availabilities, precedence constraints, time windows, ...) are defined in accordance with the reference schedule. Finally, the reference solution is modified to become the base schedule, which completes the instance generation.

Mode	v_m	e_m
Single	1	1
Shift	0.6	2
External	1	0

Table 2.2: Task modes used by the instance generator.

Environment generation

From the number of projects given (together with their average total work) and the length of the scheduling period, a measure for the expected workload per time slot can be extracted. The number of employees and workbenches required to achieve a certain mean degree of utilization can be estimated from this measure.

Equipment is generated by a separate component, which can be passed to the instance generator. This component also handles equipment requirements of tasks in Step 2.4.1. Currently supported are two different implementations:

Lab equivalent mode generates devices for exactly those equipment groups that are relevant for planning at our industrial partner. The equipment requirements for tasks also closely corresponds to the distribution of requirements in the real-world laboratory.

General mode is more flexible and creates between 3 and 6 equipment groups, together with corresponding devices. Equipment requirements for each groups are selected to be either unitary (i.e. tasks require at most device of this group) or randomly generated for each project.

In both cases, the number of devices in each group depends on the same measure for workload as for the employees and workbenches, modified by the expected number of required devices per task.

In this step, also the task modes shown in Table 2.2 are generated.

Reference solution

Each project is assigned a certain total workload, which is taken from a distribution that is as close as possible to the real-world data. It is also assigned to a random interval of the scheduling period, where earlier intervals are slightly favored over later ones. Then a number of tasks are created, still without a duration. In general, the expected number of tasks grows with the project's workload. These tasks are then distributed into families and further into jobs. The families can either be taken from the real-world data or generated randomly, depending on the generator configuration.

Each generated job is then randomly assigned a preliminary duration (taken from the total workload of the project), a mode (with only a small probability for *External* mode), and equipment requirements according to the chosen equipment generation mode (see Step 2.4.1). Most jobs will also be set to require a workbench.

Once these parameters are determined, the job is placed into the reference schedule. This is done by first randomly choosing a seed point within the project's assigned interval. Starting at this point, the job is grown outward in both directions as long as any feasible resource assignment for it exists or until the desired duration is reached. If no feasible placement at the whole duration can be found, the position is still accepted as long as the duration is not much smaller than expected. Where this is not the case or no resource assignment exists even for the initial seed, the procedure is repeated for another random seed point, up to a certain maximum number of iterations. While this process cannot guarantee that a feasible schedule can be found, experiments so far have shown that this is sufficient in most cases.

Once a job is scheduled, it is assigned a feasible set of resources. Its final duration is split between its tasks, minus the setup time required for its family.

To model started jobs, the scheduling period is extended to the past by one month (about 40 time slots) in the beginning of this step. After all jobs have been scheduled, the scheduling period is reduced to its original duration. Jobs ending before the new first time slot are removed completely, including their contained tasks – it is assumed that those have been completed before the start of the scheduling period. Those overlapping the first time slot have their duration reduced accordingly (potentially including removing some of the contained tasks) and are defined as started jobs.

Task properties

After the reference schedule has been completed, the remaining properties of the tasks are finalized.

Release dates, deadlines and due dates for tasks of a project are set to the first start, respectively last end, of any job in the project. They are then extended outwards by an additional number of time slots (minimum of 0), which is usually smaller for the due date than for the deadline.

The available resources are set for each job (and all contained tasks) such that they include at least the assigned resources. The number of available resources is taken from the real-world data for employees (including preferred employees) and workbenches, and handled by the chosen equipment generation mode for equipment. A small subset of tasks may also have additional available resources beyond those of the other tasks in the job.

The available task modes are chosen such that tasks of a job with the *External* mode in the reference schedule must be performed in this mode, while all other tasks can be performed in *Single* mode, plus optionally in *Shift* mode. The latter is always the case if the task's job has *Shift* mode in the reference schedule.

Started jobs will always have their time window, available resources and modes set to exactly those values that they are assigned to, to ensure that they cannot be altered by the solver.

The generation of precedence constraints is again delegated to a separate component, which can be set in the configuration. Two implementations are currently supported, both start by building a maximum graph of possible dependencies according to the reference schedule for each project:

Ranked precedence constraints assign ranks to a subset of tasks such that tasks of higher rank have all tasks of lower rank as prerequisites.

General precedence constraints randomly choose arcs in the maximum dependency graph that will result in actual dependencies between tasks.

In both cases, the number of precedence constraints between tasks is rather low and most tasks don't have any prerequisite tasks at all. This circumstance is directly taken from the real-world data, where only few dependencies between tasks appear and contrasts with other project scheduling problems that include tighter constraints on the order of activities.

Finally, possible candidates for linked tasks are identified, both within and between jobs, and a small subset of those is chosen randomly.

Base schedule

The last step in the instance generation process is the derivation of a base schedule from the reference solution.

Again, there are several supported options for this, which the generator can be configured to use. Currently, there are two main implementations supported:

Delete mode Removes some assignments of the jobs in the reference schedule, but leaves the remaining assignments intact.

Random mode Replaces assignments of the jobs in the reference schedule by random values. Resource availability constraints and time windows are respected, but other constraints may be violated by these changes.

Either mode has a parameter that defines the strength of the perturbation (given as the percentage of jobs affected). Further, various flags can modify the behavior, including keeping certain types of assignments intact (e.g. the grouping of tasks into jobs).

Any perturbations do not affect started jobs as well as a number of jobs selected as *fixed*, which have their tasks fixed and either all or some assignments fixed to the current value by restricting the time window, available resources and/or modes to their assigned values.

Parameter	LabStructure	General
Equipment	Lab equivalent	General
Task families	Lab equivalent	General
Precedence	Ranked	General
Base schedule mode	Delete	Delete
Base schedule perturbation	1.0	1.0
Base schedule options	Grouping constant	Grouping constant

Table 2.3: Instance generator configuration parameters for the two provided data sets.

2.4.2 Data sets

Currently, there are two sets of test data available, *General* and *LabStructure*, both with 60 instances each. These were generated for TLSP, but include a grouping of tasks into jobs in the (otherwise empty) base schedules that is guaranteed to have at least one feasible solution. Thus, they are directly usable also for TLSP-S. Feasible solutions for TLSP-S are guaranteed to also be feasible for TLSP under that grouping and have the same objective values.

Both data sets feature instances of various sizes, starting at 5 projects over a period of 88 time slots, up to 90 projects over 782 time slots. A single project contains an average of close to 4 jobs. There are no initial assignments except for the started jobs at the beginning of the scheduling period and a small number ($\approx 5\%$) of jobs whose assignments are fixed.

The difference between the two data sets is that for the LabStructure set, the instance generator was configured to produce instances that are as close as possible to the actual real-world data. In contrast, the General set uses the same distribution of work across projects and tasks, but is more flexible regarding several other structural features, such as equipment groups and job precedence. This distinction was made to obtain a diverse range of instances, including both those very similar to the real-world laboratory and those with different characteristics. The instance generator configurations used to create these two data sets are listed in Table 2.3.

On average, jobs have 5 available workbenches and 6 qualified employees (the different modes each require between 0 and 2 employees). The demand and availability of equipment is more varied and differs a lot between data sets and instances. Three different types of equipment demands appear: Of any given group, jobs require either a single specific device (these are usually project-specific), one out of a small list of options (< 10) or several (12 on average, but with a large variance) out of a large list (depends on the instance size, up to several hundred). The distribution of these types is heavily skewed towards the first two options. Further, the huge number of possible assignments for equipment demands of the third type is offset by the fact that most jobs do not require equipment of these groups and the number of devices per group is large enough that

ID	$\lvert P \rvert$	$\lvert A^* \rvert$	$\lvert J \rvert$	h	$\lvert E \rvert$	$\lvert B \rvert$	$\lvert G^* \rvert$	$\overline{\lvert E_j \rvert}$	$\overline{\lvert B_j \rvert}$	$\overline{\lvert G_{gj} \rvert}$
2019-04	74	856	297	606	22	17	1	5.49	3.06	1*
2019-07	59	678	251	700	24	17	1	5.33	3.17	1*
2019-10	59	660	223	572	19	17	1	5.70	3.48	1*

Table 2.4: The three real-world benchmark instances. For each instance, the table lists the number of projects and tasks, the jobs in the provided grouping (used only for TLSP-S), the length of the scheduling period, followed by the number of employees, workbenches and equipment groups. The last columns contain the mean qualified employees and available workbenches per job, as well as the mean available devices per job and equipment group (only over those jobs that actually require at least one device of the group, about 10% of all jobs). *The generated instances also include tasks with multiple available devices per group. This discrepancy arises from the fact that several equipment groups were not yet considered for planning in practice at the time these instances were created.

feasible equipment assignments can be found quite easily. These resource distributions were adapted from real-world data in our partner laboratory.

In addition to the two randomly generated data sets, we also provide three real-world instances taken directly from the laboratory of our industrial partner in anonymized form. These instances each cover a planning period of approximately one and a half years and contain between 59 and 74 projects. Table 2.4 lists some important parameters of these three instances.

All instances (including the real-world instances) are available for download at `https://www.dbai.tuwien.ac.at/staff/fmischek/TLSP`. Once additional data sets become available, they will also be added there.

2.5 Summary

In this chapter, we have introduced and formally described the Test Laboratory Scheduling Problem (TLSP), a complex and NP-hard real-world scheduling problem arising in industrial test laboratories. It contains several new features that make it challenging to solve and are not typically considered by other work on project scheduling problems. Among these features are heterogeneous resources, linked activities requiring identical resource assignments, non-standard objectives and a unique requirement to group tasks into jobs such that they can be executed sequentially, but in any order within a job. We have also described a variant of this problem (TLSP-S) without the grouping aspect, which is more closely related to the RCPSP.

Solution approaches for both problem variants can be evaluated and compared on a set of benchmark instances we made available, which contain both real-world instances and instances of varying sizes that are randomly generated based on the real-world data. The instance generator used to produce these instances supports a variety of configuration options to allow the generation of instances with different characteristics. This will also aid in future investigations of the instance space characteristics to determine particularly hard configurations.

CHAPTER 3

Local Search for the Test Laboratory Scheduling Problem

In the previous chapter we have introduced the Test Laboratory Scheduling Problem (TLSP). This chapter describes the metaheuristic approaches based on local search we developed to solve the TLSP. Local search is a powerful heuristic technique to solve hard combinatorial optimization problems. It explores the search space of possible solutions by repeatedly moving from one solution to a similar one nearby. The set of solutions that are reachable from any solution via a single such move is called its *neighborhood*. The main challenge for local search methods are local optima, solutions which are the best within their neighborhood, but not globally optimal. Various search heuristics and other techniques have been proposed to deal with these local optima and enable the algorithm to both efficiently reach and later also escape from different areas of the search space.

In order to be able to compare the performance of different search heuristics and neighborhoods, we developed a solver framework for TLSP(-S) which provides a unified program flow, common data structures and utility functionality for TLSP solver implementations (Figure 3.1). The basic building block is that of a *move*, which is a small change to a given solution, such as a replacement of a time slot or a resource assignment for a single job. Each move contains the necessary information to be applied to a schedule and to efficiently evaluate its effects on the solution quality. A basic set of move implementations are provided, which can be combined to form more complex changes. *Neighborhoods* define a set of moves available from a current schedule, and provide functionality to access and iterate over these moves. In addition they also allow for the selection of a random or the best move among those they contain. Neighborhoods can also be combined in various ways to allow for the definition of larger or more complex neighborhood structures. These neighborhoods are employed by *search heuristics*, which implement the strategies to choose a move that should be applied from among one or several neighborhoods in each step of the search.

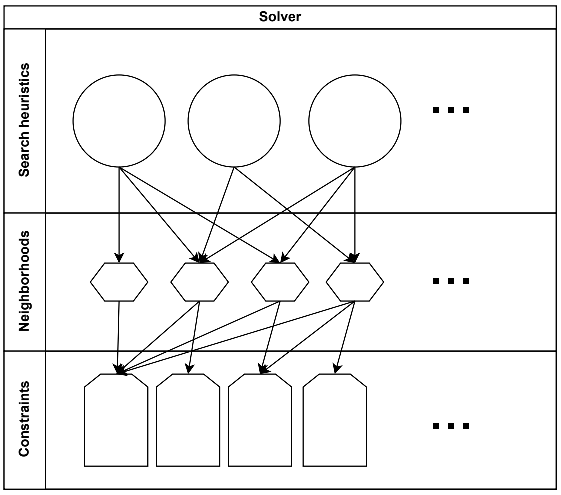


Figure 3.1: Structure of the LS framework for TLSP(-S). Search heuristics call upon neighborhoods which each contain and search over a set of different moves. The neighborhoods are aware of the problem's constraints and how they are affected by each move.

The LS algorithms and neighborhoods described for TLSP-S were first published in [MM21b], while the extension with regrouping moves for the full TLSP together with the Iterated Local Search heuristic is based on the journal paper [MMS21a].

This chapter is structured as follows: We provide an overview over solution methods that have been applied to related problems from the literature in Section 3.1. Sections 3.2 and 3.3 describe the neighborhood structures and search heuristics we developed for the TLSP(-), respectively. An experimental evaluation of our approaches is given in Section 3.4, followed by conclusions in Section 3.5.

3.1 Related literature

As the TLSP(-S) is an extension of the RCPSP, we focus here on literature on solution approaches for this problem and its many variants. We need to distinguish between exact approaches, which produce provably optimal solutions, and heuristics, which usually do not provide any formal guarantees on the solution quality, but often work well even for very large instances where exact approaches become infeasible.

Both exact and heuristic approaches for the RCPSP typically employ a *schedule generation scheme* [KH99] that schedules activities one by one at the earliest possible point in time where doing so would not violate any constraints. The main decision to take is the order in which activities are considered, as this determines the quality of the schedule. It can be shown that for both serial and parallel schedule generation schemes, there is always an order of activities that results in an optimal schedule. This approach is also known as *list scheduling* [Sch96].

Initially, the work on exact solution approaches focused on mathematical programming and implicit enumeration techniques, such as dynamic programming and branch-and-bound [HRD98]. Already the original description of the RCPSP [PWW69] included a mathematical formulation for a zero-one-programming approach. Also other exact techniques such as boolean satisfiability testing (SAT) or Integer Linear Programming (ILP) have been studied [CSE18]. Historically, these approaches were mostly limited to smaller problem sizes or more tightly constrained variants of RCPSP, with some instances with at most 60 activities remaining unsolved for a long time. However, recent years have seen some progress, in particular in the area of Constraint Programming (CP), e.g. by Szeredi and Schutt [SS16], who developed a CP model for MRCPSP that is able to solve instances of realistic sizes.

Given its similarities to TLSP-S, successful solution approaches for MSPSP may be of particular interest. To the best of our knowledge, the best results so far for MSPSP have been achieved by Young et al. [YFS17], who also used a CP approach to solve the problem.

Many early heuristic approaches focused on priority or dispatching rules [KH99] to determine the order of activities for list scheduling. Both single-pass and multi-pass priority rules have been proposed. Next to priority rules, Kolisch and Padman [KP01] identify also other groups of heuristic approaches. These include truncated variants of the previously mentioned branch-and-bound techniques and methods based on disjunctive graphs. The latter augments the existing precedence graphs with disjunctive arcs between activities in minimally unsatisfiable sets such that the resulting schedules is guaranteed to be free of resource conflicts once an orientation is chosen for the disjunctive arcs.

The most success over the last decades has been achieved with metaheuristic techniques, particularly for complex extensions of the RCPSP where the list scheduling approach is no longer appropriate. Multiple surveys have dealt with metaheuristics for the RCPSP and its variants, such as by Kolisch and Hartmann [KH06], Van Peteghem and Vanhoucke [PV14], or Mika et al. [MWW15]. In the following, we give some typical examples for different well-known metaheuristics.

Population-based metaheuristics have been employed frequently, such as genetic algorithms, e.g. by Hartmann [Har98] or by Alcaraz and Morot [AM01]. These approaches usually use activity lists as chromosomes, often augmented with additional genes to represent decoding parameters or deal with problem extensions. Gonęalves et al. [GMR08] describe a genetic algorithm for the RCMPSP where solutions are decoded using a modified parallel schedule generation scheme. Another popular population-based metaheuristic is ant colony optimization, e.g. by Merkle et al. [MMS02], or by Myszkowski et al. [MSOO15] for the MSPSP.

Other metaheuristics are based on local search, such as the tabu search by Nonobe and Ibaraki [NI02] or the simulated annealing metaheuristic by Bouleimen and Lecocq [BL03]. Notably, these approaches typically also make use of list scheduling with a schedule generation scheme. Their neighborhood structures are defined accordingly, which makes them hard to adapt for TLSP(-), which contains several additional constraints and an objective function that is not compatible with list scheduling. The few metaheuristics using a direct schedule representation for RCPSP (e.g. [TS98]) have not been able to comepte with the state-of-the-art approaches.

Most recently, many researchers have also employed hybrid algorithms for RCPSP and its variants, combining different metaheuristics or heuristic and exact techniques. A review of these approaches can be found in a dedicated survey by Pellerin et al. [PPB20]. Such a hybrid approach using a combination of memetic and hyperheuristic methods with Monte-Carlo tree search by Asta et al. [AKK$^+$16] won the 2013 MISTA challenge, which dealt with the MMRCMPSP. The same problem is also treated by Ahmeti and Musliu [AM18], who provided several ideas that were useful for our own methods for TLSP-S.

In general, metaheuristics seem to be the most promising approach to a new variant of RCPSP, such as TLSP(-)S, both due to their ease of implementation and their consistently high performance on existing problem variants in the literature. Where exact solution approaches are desired, CP should certainly be considered, given its recent successes. Such a CP model, written in MiniZinc ([NSB$^+$07]) and using the solver Chuffed ([Chu11], for both the TLSP-S and the TLSP was developed in parallel to the metaheuristic approaches described in this thesis. The models use dedicated decision variables to assign a mode, timeslots, and resources to each job. The task grouping in the TLSP is modeled by having each job represented by one of its tasks, with a mapping from the set of tasks to itself, where each task is mapped to the representative of its job. In addition, we also introduced a hybrid approach based on Very Large Neighborhood Search (VLNS). The VLNS internally uses the CP model(s) to repeatedly solve small subproblems optimally, which consist of only one or a small number of projects, keeping the rest of the schedule fixed. The CP models and a full description of the VLNS approach can be found in the respective publications for the TLSP-S [GMM19a, Gei20] and the TLSP [DGMM20].

3.2 Neighborhood structures

Metaheuristics based on LS explore the solution space by moving from one solution to a similar one that lies nearby in the search space. The solutions that are reachable with a single move form the *neighborhood* of the current solution. The structure of this neighborhood is a crucial component of each approach and must be tailored specific to each problem. Small neighborhoods are easy and efficient to explore but may miss good solutions. On the other hand, large neighborhoods are often slow or even impossible to completely explore, and eventually degenerate into complete enumeration of solutions or random sampling as their size approaches the whole search space.

The suitability of a neighborhood depends a lot on the solution representation used. For RCPSP, and several of its variants, the state-of-the-art metaheuristic approaches represent a schedule using a priority list of activities [PPB20], also called permutation-based solution representation [Har98], employing a schedule generation scheme to construct the actual schedule. Typical neighborhoods then revolve around modifying the order of activities in the priority list, by moving or swapping single activities or (small) subsets of all activities. Suitable extensions to the priority list approach can adapt it to problem variants such as the MRCPSP. However, they all rely on the fact that once the right order of activities is found, the optimal schedule can be determined efficiently by scheduling the activities in order as early as possible. This is not the case for TLSP(-S) both due to the heterogeneous resources which require explicit assignment of individual units and due to the combination of objectives which mean that scheduling activities at the earliest possibility is no longer necessarily optimal. For this reason, we use a direct schedule representation, where mode, start time, and assigned resources are all decision variables for each job, in addition to the tasks it contains to determine the grouping. This also requires developing new neighborhood structures that can be explored and evaluated efficiently on our solution representation.

We first focus on defining neighborhoods for the TLSP-S problem variant with predefined grouping (Section 3.2.1). These can then be extended with neighborhoods containing regrouping moves in Section 3.2.2 which allow us to solve the full TLSP.

3.2.1 Scheduling neighborhoods for the TLSP-S

In TLSP-S, the grouping of tasks into jobs is already provided and fixed. The goal is therefore to find an optimal assignment of mode, timeslots, and resources to each job.

We have developed two different neighborhoods that are suitable for this goal. Both are the combination of several specialized smaller neighborhoods and contain the union of all their moves.

The first set, called *Simple*, contains neighborhoods that each focus on a single aspect of the solution:

- *Mode Change*: Switches the assigned mode of the job to a different value. The start time of the job is kept constant for all moves in this neighborhood, except where the new duration would conflict with time windows or precedence constraints. In these cases, the start time is adjusted to ensure that those constraints are satisfied. Also, the number of assigned employees is adjusted to match the new requirements.
- *Time Slot Change*: Moves the job to a new position in the schedule. As before, time windows and precedence constraints are respected by all possible moves.
- *Resource change*: Switches out a single assigned resource unit (workbench, employee or device) by a different unit of the same type (and group, for equipment).
- *Resource Swap*: Swaps a unit of a resource assigned to this job with a different unit of the same type assigned to an overlapping job. A resource unit is considered for a swap only if it is suitable for its new job.

In some sense, the neighborhoods contained in *Simple* are elementary in that the moves they contain are the smallest possible changes to a single assignment of a job. Still, their combination is sufficient to cover the whole search space of TLSP-S. In other words, an optimal solution can be reached from any starting point using only moves contained in one of these neighborhoods. Strictly speaking, the first three neighborhoods are already sufficient to do so, but the addition of resource swaps adds more options in situations where a single change would result in a prohibitively large number of conflicts.

The second variant is a larger neighborhood, with the main idea that a job is completely removed from the schedule and the neighborhood contains all possible combinations of mode, time slot and resource assignments for that job (as before, time windows and precedence constraints are respected by all moves).

However, the enormous number of potential equipment assignments per job in some instances made some adjustments necessary. For example, instance LabStructure/010 contains 88 devices in equipment group 1, of which 27 are required by job 45, for a total of $\binom{27}{88} \approx 3.3 \times 10^{22}$ different possible equipment assignments for this job alone.

Instead, we employed a reduced version of the neighborhood (*JobOpt*), which keeps the existing equipment assignments intact, and combined it with a Resource Change neighborhood limited to equipment changes (*Equipment change*). We refer to this combination of neighborhoods as *JobOpt+EC*. To further increase the efficiency when the best move in the neighborhood is required, we utilize independencies between assigned resources to precompute and cache the effects of assigning individual units to the job.

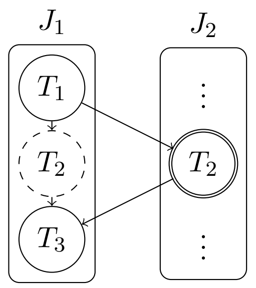
All neighborhoods assume that the jobs are scheduled within their time window, precedence constraints are satisfied, all resource requirements are fulfilled and the assigned mode and resources are available for each job (so constraints H4-H7 as well as H9 and H10 are satisfied). In return they guarantee that this remains the case after any of their contained moves is applied. As long as the same holds true for the initial solution (which is the case for all initialization heuristics we have considered in this thesis), we can be sure that only constraints H8 (Single assignment) and H11 (Linked jobs) can ever be violated by intermediate solutions[1].

3.2.2 Regrouping neighborhoods

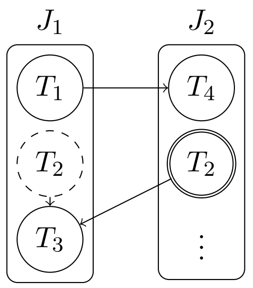
Since the neighborhoods described in the previous section do not affect the task grouping, they alone are not sufficient to solve the more general TLSP. In order to deal with the variable grouping of TLSP, we developed four additional neighborhoods that modify the task grouping.

In addition to the preconditions and guarantees on moves for the scheduling neighborhoods, the regrouping neighborhoods also assume that the current task grouping is valid (H1-H3 are satisfied) and guarantee that this remains the case after a move is applied. Since it can happen that altering the task grouping results in changing resource requirements or

[1]Constraints H1-H3 deal with the task grouping and are trivially satisfied in TLSP-S anyway



(a) Task T_2 is both successor and predecessor of other tasks (T_1 and T_3) in the original job.



(b) Moving task T_2 turns an existing one-directional dependency between the jobs into a cyclic dependency.

Figure 3.2: Two example scenarios where moving a task (T_2) from a job J_1 to another job J_2 creates a cyclic dependency between the two jobs. Arcs between tasks show the task dependencies. In both cases, job J_2 would have to be scheduled both before the start and after the end of job J_1, which is impossible. Further scenarios, potentially involving more than two jobs, exist.

available resources, the solver supports different strategies to restore the validity of the resource assignments.

In the following, a job's time window denotes the interval in which it must be scheduled, including not only its release date and deadline but also precedence relations to other jobs. All neighborhoods involve moving tasks from one job, the *source*, to another job, the *target*.

Single task transfer

This neighborhood contains moves that transfer a single task from the source job to a target job of the same family and project. A number of restrictions apply to which tasks can be moved:

- Fixed tasks or tasks that are the only task of their job cannot be transferred (this case is handled by the Merge neighborhood).
- The mode and all resources assigned to the target job must be available for the transferred task.
- Moving the task must not introduce a cyclic dependency between jobs (see Figure 3.2 for examples).
- Finally, the increased duration of the target job must still fit within its time window (including potentially updated precedence constraints).

The modes and start times of both involved jobs are unchanged, except if it is required to move the target job forward so that its new duration does not conflict with deadlines or successor jobs.

The resource requirements of the source job may decrease due to the transferred task. In this case, two strategies are supported to remove superfluous assigned units: They can be either randomly chosen units or the worst units, i.e. those that currently cause the largest number of conflicts or the largest penalty.

Correspondingly, the resource requirements of the target job may increase. The missing resource units are chosen from the set of unassigned available resources. As with the source job, these can be either random choices or the best units for the job.

Merge

This neighborhood contains moves that merge two jobs of the same family and project, i.e. transfer all tasks of the source job to the target job and remove the source job from the schedule. To be candidates for a merge, the two jobs must fulfill the same requirements as for a task transfer above. The scenarios of Figure 3.2 cannot happen for merges, but cyclic dependencies could still arise if there is a third job that is a successor of the source job and a predecessor of the target job, or vice versa. In this case, the two jobs cannot be merged.

As for a transfer, mode and start time of the target job are not changed, except where necessary for the job to fit into its time window. Analogously, if the resource requirements change, they need to be adjusted using either random or the best available resource units.

Split

The Split neighborhood covers the need of creating new jobs. A subset of the source job's tasks are removed from it and assigned to a newly created job. Also for split moves, care must be taken to ensure that the resulting jobs do not create a cyclic dependency with each other. To ensure this, we require that for each split off task, also all successors in the source job will be split off to the new target job. It follows from this criterion that the newly created target job can be a successor of the source job, but never the other way around.

The start of the source job is adjusted to make room for the split job, if necessary (the combined duration of the reduced source job and the newly created job may be longer than the source job's original duration due to the setup time and rounding). Otherwise, it does not change. The resources of the source job are adjusted if the requirements have changed, using either of the two strategies described for the single task transfer neighborhood.

The target job has the same mode as the source job. Several configuration options are available to determine its start time: It can either start directly following the end of the source job, start at a random position within its time window, or start at the best possible time (with respect to the current schedule).

Regarding the resources assigned to the target job, one option is to duplicate the resource assignment of the source job and adjust it according to either of the two previously described strategies. Two alternative strategies are also supported: The resources can be assigned completely randomly from the available units or the best units from each resource can be chosen to be assigned to the job.

Linear split

The Split neighborhood contains all possible partitions of a job into two parts, except for some restrictions due to fixed tasks or other constraints. The number of these partitions rises exponentially with the number of tasks in a job, which makes it inefficient for algorithms that traverse the whole neighborhood.

To solve this problem, we developed an alternative variant of the Split neighborhood, called Linear split. This neighborhood randomly generates a topological ordering of the tasks in the job for each move. It contains only moves that split this ordering at a certain index, such that all tasks after this index are moved to the newly created job.

This behavior guarantees that the time required to traverse the neighborhood is linear with respect to the number of tasks in the job. The drawback is that it is no longer deterministic, in the sense that it does not always contain the same moves if applied for the same schedule and job.

The topological ordering itself is created by repeatedly choosing a random task that does not have any unchosen predecessors in the job, until all tasks have been chosen. If tasks are chosen randomly in a uniform way, the produced orderings are biased heavily towards orderings that place unconnected tasks early in the ordering. To even out the distribution, we weight tasks by the number of their successors incremented by one.

This does not completely eliminate bias (it leads to double counting of some paths), but drastically reduces it for graphs with many dependencies. A small example of this is shown in Figure 3.3. The given dependency graph has 10 different topological orderings (2 orderings for the left component, 5 positions for task T_5 in each). Of these, 2 (20%) have node T_5 in the first position. With uniform weights, both node T_1 and node T_5 can be selected first with equal probability, leading to 50% of generated orderings starting with T_5. With our adapted weights, node T_1 would get a weight of 4, and thus would correctly be chosen first in 80% of all generated orderings. The remaining bias with adapted weights is seen with nodes T_2 and T_3. Assuming T_1 was selected first, T_5 appears before both T_2 and T_3 in 2 out of 8 possible orderings (25%). However, both nodes get weight 2 due to their common successor T_4. As a result, T_5 is chosen as the second node after T_1 only in $1/(2 + 2 + 1) = 20\%$ of all generated orderings. The discrepancy occurs due to the double-counting of node T_4 in the weights. In general, the remaining bias decreases the more tree-like a dependency graph is.

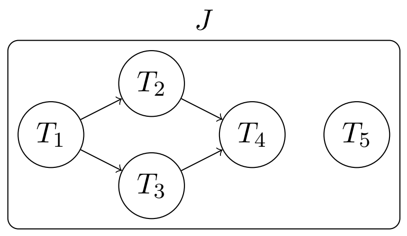


Figure 3.3: Example dependency graph for a job J with 5 tasks.

Unfortunately, this bias cannot be completely eliminated, as truly uniform sampling would require counting the number of all topological orderings, which is already #P-complete by itself [BW91].

The remaining behavior and configuration options for this neighborhood are the same as for the Split neighborhood.

3.3 Search heuristics

Once the neighborhood structures are defined, we next need to decide on a strategy to select a move at each step of the search. The most intuitive way to improve the quality of the solution is to choose and apply the best move in the neighborhood each time. This strategy is called *hill climbing* or *best-improvement* strategy. It has two main drawbacks that make it unsuitable for most hard problems without modification: Most importantly, the solution landscape of these problems almost always features a large amount of local optima, solutions that are at least as good as any neighboring solution but not necessarily globally optimal. Once the hill climbing heuristic reaches such a local optimum, it cannot escape from it again to explore different areas of the search space. Secondly, the exploration of the whole neighborhood at each step is often prohibitively expensive. While caching and clever evaluation strategies may avoid having to independently evaluate each individual move, finding the best move in larger neighborhoods can still cost a lot of time.

In the past, many different metaheuristics have been proposed to deal with these problems, often involving randomness and the acceptance of worsening moves from time to time. For this thesis, we have implemented and evaluated the Min-Conflicts heuristic and Simulated Annealing, two well-known metaheuristics that have been used successfully for solving complex optimization problems before, including project scheduling problems. In addition, we have incorporated both metaheuristics into an Iterated Local Search.

Our search heuristic implementations make use of the available neighborhoods, which encapsulate the problem specific details and can be dynamically changed. Thus, the heuristics are problem-agnostic and can be easily used for TLSP, TLSP-S and other related variants, as long as the employed neighborhoods are adjusted to fit.

3.3.1 Min-Conflict

The min-conflicts (MC) heuristic [MJPL92] was originally developed to solve constraint satisfaction problems (including scheduling applications such as the Hubble Space Telescope scheduling problem). It works by randomly selecting a variable appearing in at least one conflict (violated constraint) and choosing a value for it that minimizes the number of conflicts remaining. The MC technique has been transferred successfully to other problem domains including a project scheduling problem [AM18, AM21] and personnel scheduling [Mus05].

We adapted MC for TLSP(-S) and our solver framework as follows: Choose a job at random that violates at least one constraint, and pick a move from the neighborhood involving the chosen job that minimizes constraint violations (i.e. the best move).

Different selection strategies are possible, due to the distinction between hard and soft constraints. We have experimented with three different variants of Min-Conflict. The first selects jobs from among those violating hard constraints. Once the solution is feasible, also soft constraint violations are considered. The second variant considers both hard and soft constraint violations from the start. Finally, the third variant chooses from all jobs, regardless of their presence in any kind of constraint violation.

For TLSP(-S), the second and third variants are virtually equivalent due to soft constraints S3 (Number of employees) and S5 (Project completion time), whose presence entails that most, if not all, jobs contribute a non-zero penalty to at least one of those objectives[2]. Further, results did not differ in any meaningful way between the first and either of the latter variants in preliminary experiments.

For simplicity, and to avoid having to keep a running account of the jobs involved in constraint violations, the experiments in Section 3.4 have been done using variant three.

A weakness of MC is that it contains no mechanism to avoid repeating already visited solutions and thus might get stuck where several adjacent configurations for most or all jobs are locally optimal. A possible countermeasure is to inject additional randomness into the solution procedure. In our framework, we have also included a random walk (RW) algorithm, that randomly selects a job and performs a random move for the chosen job. RW is combined with MC in a hybrid heuristic that at each step calls RW with a low probability p^{RW}, and MC otherwise (MC+RW).

Also, the search automatically restarts from a new initial solution if no progress has been achieved within a certain number of moves.

3.3.2 Simulated Annealing

Simulated Annealing ([KGV83]) is a well-known metaheuristic that has been employed successfully to solve many NP-complete problems, including RCPSP (e.g. [BL03, LDGN17]).

[2]This is exacerbated in TLSP, where soft constraint S1 (Number of jobs) trivially involves every single job in a soft constraint violation

In theory, SA with an appropriate cooling scheme is guaranteed to find the optimal solution if given infinite computing time [AK89].

In SA, the search starts from a randomly generated solution. In each step, a candidate move is chosen randomly from the available neighborhoods. The difference in objective value Δ due to the chosen move is calculated. If $\Delta < 0$ (for minimization problems), the move is applied. Otherwise, it is still accepted with probability $e^{-\Delta/T}$. Thus, the acceptance probability depends on Δ (smaller values have a larger probability to be accepted) and a parameter T called *temperature* (higher values result in a larger acceptance probability). The temperature starts at an initial value T^0 and is iteratively reduced by a *cooling factor* α, with $0 < \alpha < 1$, after a certain number of steps, until a minimum temperature T^{min} is reached. At this point, the search stops.

The choice of α is crucial because it determines the number of moves until the algorithm halts. In order to ensure that the available time is fully used, i.e. T^{min} is reached right at the timeout, we adjust α according to the number of moves applied per second. At a search speed of m moves per second, i iterations between successive cooling steps, current temperature T, T^{min} will be reached after $u = \frac{i}{m}\log_\alpha(\frac{T^{min}}{T^0})$ seconds. Conversely, for a given timeout of u seconds, we get the following for α:

$$\alpha = \left(\frac{T^{min}}{T^0}\right)^{\frac{i}{um}} \tag{3.1}$$

Since search speed (the value of m) can vary between different instances, processors and also in the course of the search, we initially set α to 1, and periodically update it during the search, using the average measured search speed so far for m, the current temperature instead of T^0 and the remaining time for u.

We have also alternatively experimented with fixed cooling rates, where the search restarts when the minimum temperature is reached, either from a new initial solution or the best solution found so far (reheating). Neither of these options (at different cooling rates - and therefore different numbers of restarts or reheatings) improved upon the solution quality, with short cooling cycles producing infeasible solutions more often and worse penalties in most other cases (see also the results for the integration of SA with ILS in Section 3.4.1).

Finally, a schedule for TLSP(-S) can contain both hard and soft constraint violations. This has to be taken into account when calculating a value for Δ. In our implementation, we have weighted each hard constraint violation by a factor w^H (chosen by the parameter tuner, see below).

3.3.3 Iterated local search

One of the main challenges for metaheuristics is escaping from local optima. Especially as the size of the instances grows, the algorithms are likely to get stuck at or around basins of low objective values. Random restarts are one way to deal with such situations and sample larger areas of the search space, but each restart means throwing away all information about the (hopefully good) solution achieved previously.

ILS also repeatedly executes runs of a metaheuristic internally. In contrast to random restarts however, ILS aims to keep as much information as possible from the previous solution between restarts to provide a good starting point for the next iteration of the inner metaheuristic, while applying a large enough perturbation to reach new areas of the search space. In essence, ILS applies a local search procedure over the search space of locally (near-)optimal solutions.

This often leads to significant improvements compared to single-run metaheuristics or random restarts [LMS03].

Algorithm 3.1 shows the pseudocode of our implementation of ILS in the TLSP solver, which can use any other local search procedure (LS) internally. In this paper, we use either MC+RW or SA as internal heuristics. The best solution found so far is stored and whenever a new solution is not accepted after a run of LS, the current solution is reset back to best known solution.

Algorithm 3.1: Pseudocode for Iterated Local Search

```
   Input: a TLSP instance $I$
1  $S^{best} = S =$ create initial schedule($I$);
2  while $\neg$ timeout reached do
3      $S' =$ LS($S$);
4      if $S' < S^{best}$ then
5          LS $S^{best} = S'$;
6      end
7      if accept($S', S$) then
8          $S = S'$;
9      else
10         $S = S^{best}$;
11     end
12     $S =$ perturb($S$);
13 end
```

We use the same algorithm to create the initial solution as in the single-run version of LS (greedy construction for MC+RW and random for SA).

The remaining components to determine are then the stopping criteria for the internal metaheuristic, the acceptance criterion to determine whether search should continue from the current solution, and the perturbation to apply at each restart.

Regarding the stopping criteria, MC+RW already restarts after a number of unsuccessful moves. This can be immediately reused to instead apply the next iteration of ILS. Since SA does not have such an intuitive stopping criterion (in particular with the dynamic cooling scheme described above), we instead provide each iteration of SA with a separate short timeout, after which it should stop. Naturally, this results in faster cooling cycles as the minimum temperature now has to be reached within the shorter available time.

The most straightforward acceptance criterion is to accept only improving or same cost solutions. However, it can be beneficial to also accept solutions that are slightly worse, in order to reach more distant areas of the search space. We have implemented two different approaches to the ILS acceptance criterion:

Threshold. Accepts any solution that has at most δ conflicts more than the best known solution.

Annealing. Works like the move acceptance criterion in SA, except that the temperature T is fixed. Takes the hard constraint weight w^H as additional parameter.

Finally, the perturbation can be performed either by executing a fixed number m^{RW} of steps of RW (*randomwalk*), or by choosing a subset of all jobs and replacing their mode, time slot and resource assignments by random values (*disrupt*), respecting time windows, mode and resource availabilities. The second option can be configured via the disruption strength s^d, i.e. the fraction of jobs that is chosen, and the strategy to select jobs:

Job. Select randomly among all jobs.

Project. Select randomly among all projects and all jobs of a selected project.

JobConflict. Select jobs involved in conflicts first, then randomly among all jobs.

ProjectConflict. Select projects involved in conflicts first, then randomly among all projects.

3.4 Experimental evaluation

For the experiments, a set of 33 instances of different sizes were chosen (15 each from the General and LabStructure sets described in Section 2.4.2 - one of each size, plus a second instance for the three smallest sizes), plus the three real-world instances available. The instances and some important properties are listed in Table 3.1.

For experiments with TLSP-S, we use the task grouping provided in the reference solution (which is otherwise empty except for the required assignments of started jobs). This task grouping is ignored for all experiments on TLSP, as if the base schedule was completely empty (again with the exception of the started jobs).

The solver framework and all algorithms within it were implemented in Java 8. All our solution approaches are single-threaded and we used a timeout of 10 minutes (600s) per run, unless noted otherwise. We used two different machines for our experiments: The parameter tuning and the evaluations for TLSP-S(published in [MM21b]) were performed

#	**Data Set**	**ID**	$\lvert P\rvert$	$\lvert A^*\rvert$	$\lvert J\rvert$	h	$\lvert E\rvert$	$\lvert B\rvert$	$\lvert G^*\rvert$	$\overline{\lvert E_j\rvert}$	$\overline{\lvert B_j\rvert}$	$\overline{\lvert G_{gj}\rvert}$
1	General	000	5	13	7	88	7	7	3	2.08	3.57	1.5
2	General	001	5	20	8	88	7	7	3	4.88	3.63	15.67
3	LabStructure	000	5	73	24	88	7	7	3	1.84	3.38	11.67
4	LabStructure	001	5	58	14	88	7	7	3	4.36	3.5	0.36
5	General	005	10	86	29	88	13	13	4	4.04	3.48	5.76
6	General	006	10	62	18	88	13	13	6	5.56	4.22	13.28
7	LabStructure	005	10	102	37	88	13	13	3	6.16	4.03	0.65
8	LabStructure	006	10	93	29	88	13	13	3	6.21	3.76	21.01
9	General	010	20	182	60	174	16	16	5	7.42	4.42	11.36
10	General	011	20	273	84	174	16	16	4	7.31	4.3	3.7
11	LabStructure	010	20	224	65	174	16	16	3	6.28	4.43	26.26
12	LabStructure	011	20	213	62	174	16	16	3	7.27	4.24	1.21
13	General	020	15	80	29	174	12	12	5	5.76	3.97	1.12
14	LabStructure	020	15	123	53	174	12	12	3	6.28	4.47	20.63
15	General	025	30	376	113	174	23	23	3	8.26	4.41	5.71
16	LabStructure	025	30	422	105	174	23	23	3	7.52	4.25	39.63
17	General	015	40	405	126	174	31	31	3	9.26	4.48	29.53
18	LabStructure	015	40	429	138	174	31	31	3	7.36	3.57	41.93
19	General	030	60	613	208	174	46	46	6	9.85	4.11	31.45
20	LabStructure	030	60	775	212	174	46	46	3	9.28	4.17	78.16
21	General	035	20	304	76	520	6	6	5	4.24	3.62	8.08
22	LabStructure	035	20	280	71	520	6	6	3	4.3	3.42	11.70
23	General	040	40	714	196	520	12	12	4	6.95	4.47	4.24
24	LabStructure	040	40	661	187	520	12	12	3	6.55	4.51	1.38
25	General	045	60	940	260	520	18	18	6	7.65	4.52	23.95
26	LabStructure	045	60	837	239	520	18	18	3	7.44	4.42	33.65
27	General	050	60	866	270	782	13	13	4	6.89	4.39	3.89
28	LabStructure	050	60	891	247	782	13	13	3	6.97	4.21	23.42
29	General	055	90	1282	384	782	19	19	5	7.27	4.29	26.89
30	LabStructure	055	90	1573	401	782	19	19	3	7.34	4.53	36.76
Lab1	Real-world	-	74	856	297	606	22	17	1	6.02	5.36	1*
Lab2	Real-world	-	59	678	251	700	24	22	1	6.19	5.33	1*
Lab3	Real-world	-	59	660	223	572	19	17	1	5.83	5.39	1*

Table 3.1: The set of test instances used for the experiments. Shown are the data set the instance is taken from and the ID within that set. The following columns list the number of projects and tasks, the jobs in the provided grouping (used only for TLSP-S) and the length of the scheduling period, followed by the number of employees, workbenches and equipment groups. The last columns contain the mean qualified employees and available workbenches per job, as well as the mean available devices per job and equipment group (only over those jobs that actually require at least one device of the group, about 10% of all jobs).
*The discrepancy to the generated instances arises from the fact that several equipment groups were not yet considered for planning at the time this instance was created.

on a Lenovo ThinkPad University T480s with an Intel Core i7-8550U (1,8 GHz). The experiments for TLSP (from [MMS21a]) were done on a benchmark server with 224GB RAM and two AMD Opteron 6272 Processors each with a frequency of 2.1GHz and 16 logical cores. This is the same machine also used for the CP and VLNS approaches [GMM19a, DGMM20], to allow for a fair comparison. The parameter tuning was again performed on the same Lenovo ThinkPad. To account for the hardware performance difference, we decreased the timeout for tuning search parameters for TLSP to 370s, as this resulted in approximately the same number of moves performed. One thing to keep in mind when comparing results for TLSP-S and TLSP is that the experiments for TLSP-S effectively had a longer time available than those for TLSP due to the difference in hardware.

3.4.1 Parameter configuration and tuning

For automated parameter tuning, we used SMAC3 ([HHLB11]), version 0.10.0. Tuning was performed on a set of 30 generated instances chosen in the same way as, but distinct from, the test data set. In each case, we executed four instance of SMAC3 in parallel in shared model mode and allocated a budget of 500 target algorithm runs to each.

Min-Conflict

Initially, we tuned the MC(+RW) heuristic for TLSP-S, using the scheduling neighborhoods of Section 3.2.1 only [MM21b]. In preliminary experiments, both the Simple and the JobOpt+EC neighborhoods achieved comparable results on the training set over several different configurations. We arbitrarily decided to use the JobOpt+EC neighborhood for subsequent tuning and the final evaluations.

There are two construction heuristics available in our solver framework that can be used to generate the initial solution. The first (*Greedy*) iterates over the jobs in order of ascending deadline and greedily assigns to each job the currently best values. The other (*Random*) uses random values for all assignments, though it respects time windows, job precedence and resource availability constraints. The choice of construction heuristic is governed by the parameter *Init*.

While MC itself does not include any parameters, there are two additional parameters due to its combination with RW and the inclusion of restarts: The first is the maximum number of moves without improvement (*MMWI*) before the search restarts from a new initial solution. Finally, p^{RW} denotes the probability at each step that a single move of random walk is performed instead of the min-conflict heuristic. All parameters submitted for tuning to SMAC3, and the best configuration found, are shown in Table 3.2.

To use MC+RW for solving the full TLSP, we added the regrouping neighborhoods TaskTransfer, Merge, and LinearSplit to the existing neighborhoods JobOpt and EquipmentChange. Initially, we used the more general Split neighborhood in place of LinearSplit. However, it quickly became apparent that using the Split neighborhood was computa-

Parameter	**Domain**	**Best**
Init	{Random, Greedy}	Greedy
MMWI	$\{100 \dots 10000\}$	131
p^{RW}	$\{0, 0.05, 0.1, 0.2\}$	0.1

Table 3.2: Tuning parameters for Min-Conflict with Random Walk for TLSP-S. The last column lists the parameter values for the best configuration found.

tionally infeasible, as evaluating all possible moves could take up to eight hours for some of the larger jobs.

Regardless of the type of splitting neighborhood used, we decided to combine it with the Merge neighborhood into a combined neighborhood *(Linear)SplitMerge* where both have equal weights internally. The motivation behind this decision is that they contain basically complementary moves, one creating jobs and the other removing them again. An imbalance in either direction would lead to either many fragmented jobs or few jobs with many wasted attempts to find further merge candidates.

To configure the new neighborhoods for MC+RW, we used the *adjust_best* strategy for resource assignments and *best* for the time slot assignment of the new job created by the Split neighborhood (see Section 3.2.2).

For RW, we assigned equal weights to all neighborhoods.

For the remaining search parameters, we reused the best performing parameter values of TLSP-S.

Simulated Annealing

We also tuned SA first for TLSP-S without regrouping neighborhoods. The parameters to tune for SA are the initial temperature T^0, the minimum temperature T^{min} and the weight factor for hard constraint violations w^H, since the cooling rate is automatically determined from the available time and the number of moves applied per second. The parameters for SMAC3 and tuning results are listed in Table 3.3a.

As with MC+RW heuristic, we saw comparable results for both the Simple and JobOpt+EC neighborhoods and opted to continue the experiments with the JobOpt+EC neighborhood for comparability. Since SA chooses random moves at each step, we can simply use the union of all moves in the two component neighborhoods of JobOpt+EC.

This approach is however no longer suitable when we extend our approach for TLSP by adding the regrouping neighborhoods. Since the number of moves within the regrouping neighborhoods varies a lot and most likely does not correspond to the frequency we would like to use them compared to each other and both JobOpt and EquipmentChange, we need to carefully determine the relative neighborhood weights. This requires a tuning procedure for TLSP.

Parameter	**Domain**	**Best**
T^0	$\{10 \ldots 100\}$	69
T^{min}	$\{0.001, 0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1, 2, 5, 10\}$	0.1
w^H	$\{10, 20, 50, 100, 200\}$	10

(a) Parameters for TLSP-S

Parameter	**Domain**	**Best**
weight-equipmentchange	0 - 0.4	0.353
weight-tasktransfer	0 - 0.2	0.015
weight-splitmerge	0 - 0.2	0.004
adjust-resource	adjust_random, adjust_best	adjust_best
split-resource	adjust_random, adjust_best, random, best	adjust_random
split-timeslot	follow, random, best	random

(b) Additional parameters for TLSP

Table 3.3: Tuning parameters for SA on TLSP-S (a) and the extension to TLSP (b). The last column lists the parameter values for the best configuration found.

The good news is that we can use the full Split neighborhood, as choosing a random move is fast despite its large potential size. As before, we combined it with the Merge neighborhood at equal weights. We therefore arrive at the following set of neighborhoods for which we need to set their relative weights: JobOpt, EquipmentChange, TaskTransfer, and SplitMerge. We defined the weight of the latter three as parameters for SMAC3 to tune, which also determines the weight of JobOpt.

In addition to their relative weights, we also need to determine the strategy that should be used to adjust resource assignments of existing jobs when the requirements change due to a regrouping. Finally, we need to decide how to assign both a time slot and resources to the new job created via the Split neighborhood. While it may seem natural to use random assignments here too, this is not necessarily the best option even for SA.

The full list of tuned parameters for TLSP as well as the best configuration found is shown in Table 3.3b. As for MC+RW, we reused the best configuration found for TLSP-Sfor the remaining search parameters to keep down the number of simultaneously tuned parameters.

Iterated local search

Since the ILS approach was only introduced in our paper on TLSP [MMS21a], we did not perform a separate parameter tuning or evaluation for TLSP-S, instead focusing on TLSP.

Parameter	Domain	Condition	Best configuration MC+RW	SA
perturbation	disrupt, randomwalk	-	randomwalk	disrupt
disrupt-select	job[Conflict], project[Conflict]	disrupt	-	job
disrupt-strength	0.05, 0.1, 0.2, 0.5	disrupt	-	0.5
randomwalk-moves	5, 10, 20, 50, 100	randomwalk	10	-
acceptance	threshold, annealing	-	annealing	threshold
threshold-δ	0, 2, 5, 10	threshold	-	2
annealing-T	10, 20, 50, 100, 200, 500	annealing	100	-
annealing-w^H	10, 20, 50, 100	annealing	50	-
MC-moves	10, 50, 100, 200, 500, 1000	MC+RW	1000	-
SA-timeout	5s, 10s, 100s, 200s, 370s	SA	-	370s

Table 3.4: Parameters passed to SMAC3 to tune the ILS configurations. Not all parameters are used for both inner search heuristics and some depend on other parameter values. The column *Condition* shows under which conditions each parameter is used. The last two columns show the best configuration found by SMAC3 for ILS with SA and with RC+RW, respectively.

Here, we applied independent tuning runs for ILS with MC+RW as the inner search heuristic and for ILS with SA. Parameters common to both heuristics used within ILS are the acceptance criterion of the outer ILS loop and the perturbation to apply before each call to the inner search heuristic. In addition, we need to determine the cutoff for the inner search heuristic (the maximum number of moves without improvement for MC+RW and the per-iteration timeout for SA), as shown in Table 3.4.

In order to keep the number of parameters to a manageable level, we again decided to transfer the non-ILS-specific parameter values for the single-run versions of MC+RW and SA over to ILS.

Interestingly, the best timeout for SA turned out to be 370s, i.e. the whole time available to the solver. As a result, only a single iteration of the ILS loop (see Algorithm 3.1) is ever performed, which makes this configuration equivalent to SA without any restarts at all. The remaining parameters are therefore irrelevant. To rule out a mistake in the tuning process, we repeated the validation with several shorter, manually picked per-iteration timeouts, which all resulted in worse solutions overall. This shows that SA does not benefit from being included in an ILS algorithm, at least for the TLSP.

The situation is different for MC+RW. The best configuration contains a large number of moves without improvement before a restart (e.g., MC-moves = 1000, which is the extreme value of the domain), but the results are significantly better than MC+RW without ILS (see Section 3.4.2). The fact that the selected value is on the extreme end of the value range suggests that it would be worth extending the domain. However, given that with 1000 moves there are already very few restarts and that without restarts the results are significantly worse, we can be confident that no improvements can come from significantly larger values of MC-moves. Concerning the remaining parameters, it appears that smaller disruptions are preferred, since the randomwalk disruption can affect at most 20 jobs (10 moves, at most 2 jobs affected per move). The annealing acceptance criterion with the given parameters means in practice that worsening solutions which contain at most one additional conflict are likely to be accepted (the acceptance probability is 50% at a total difference of about 69).

3.4.2 Evaluation and comparison

We evaluated our local search approaches on our benchmark instance set. Due to the stochastic behavior of the search heuristics, we performed multiple runs on each instance, with different seeds for the (pseudo-) random number generator.

TLSP-S

Table 3.5 shows a comparison of results for the TLSP-S using MC+RW and SA, with the configuration described in the previous section. In addition, we compare with the CP model described in [GMM19a][3], which we also ran on the same machine to ensure comparable results.

From these results, it can be seen that instances can be split into two groups: For small instances with 20 projects or less (instances 1-14 and 21,22), good solutions could be found in most cases. This was much more difficult for larger instances with more than 20 projects.

MC+RW was unable to reliably find feasible solutions for most large instances within the given time. Moreover, even where it does find solutions without conflicts, the resulting penalty is often only slightly better than what was already achieved with the greedy construction heuristic.

SA performed much better and found feasible solutions in more than 92% of all runs. Also the solution quality is consistently better than with MC+RW, except for some small instances.

The CP model could find feasible solutions for all of the instances already within one minute. Within the full time limit, it could prove optimality for 12 of the 16 small instances.

Compared to the results for SA, CP slightly outperforms SA on those instances where it could find optimal solutions. However, SA produced better results for every single other instance, sometimes by more than 30%. In particular, SA decisively outperformed CP on the large instances, i.e. those with more than 20 projects. These results can also be seen in Figure 3.4.

[3]The results in this section are based on [MM21b]. At the time this publication was written, the VLNS approach for the TLSP-S was not yet developed and thus not available for comparison.

#	MC+RW			SA			CP	
	#Feas.	Avg	Best	#Feas.	Avg	Best		
1	10/10	98.0	**98**	10/10	98.0	**98**	**98**	*
2	10/10	73.0	**73**	10/10	73.0	**73**	**73**	*
3	10/10	149.3	**149**	10/10	156.4	152	**149**	*
4	10/10	105.3	**105**	10/10	105.0	**105**	**105**	*
5	10/10	286.9	285	10/10	300.1	287	**283**	*
6	10/10	162.4	**162**	10/10	192.2	177	**162**	*
7	10/10	331.7	327	10/10	307.4	**307**	**307**	*
8	10/10	323.5	314	10/10	312.0	**310**	**310**	*
9	10/10	648.8	625	10/10	502.7	**501**	**501**	*
10	10/10	751.0	725	10/10	565.3	**564**	892	
11	10/10	1049.9	993	10/10	879.0	874	**856**	*
12	10/10	768.9	749	10/10	668.0	**663**	713	
13	10/10	341.0	**340**	10/10	352.1	352	**340**	*
14	10/10	457.9	450	10/10	425.7	422	**420**	*
15	4/10	1841.0	1800	10/10	1090.6	**1087**	1653	
16	8/10	1429.8	1357	10/10	1155.2	**1143**	1561	
17	10/10	1410.9	1381	10/10	1234.0	**1195**	1382	
18	10/10	1760.7	1688	10/10	1375.3	**1364**	1822	
19	6/10	2735.3	2675	10/10	2337.0	**2277**	2650	
20	2/10	2898.5	2853	10/10	2360.6	**2312**	2892	
21	10/10	1007.7	908	10/10	686.6	**683**	930	
22	10/10	1079.7	1033	10/10	771.9	**767**	839	
23	0/10	-	-	6/10	2476.3	**2393**	3531	
24	2/10	2664.0	2484	10/10	1852.5	**1808**	2454	
25	0/10	-	-	8/10	3050.9	**2908**	3281	
26	0/10	-	-	10/10	2805.9	**2724**	3899	
27	3/10	3405.7	3372	10/10	2191.3	**2176**	3146	
28	10/10	2617.2	2585	10/10	2375.8	**2367**	2569	
29	2/10	5406.5	5334	9/10	4428.4	**4208**	4548	
30	10/10	6646.8	6453	9/10	4896.8	**4828**	5905	
Lab1	0/10	-	-	0/10	-	-	**5187**	
Lab2	0/10	-	-	5/10	2689.4	**2658**	3575	
Lab3	0/10	-	-	9/10	2718.2	**2646**	3070	

Table 3.5: Comparison of results for the TLSP-S. MC+RW and SA were both run 10 times each, with different seeds. Columns *#Feas.* contain the number of feasible solutions found, *Best* the best solution found, and *Avg* the average penalty over all feasible solutions. The best solution for each instance found by any of the solvers is marked in bold. Solutions for CP marked with "*" are optimal for the instance.

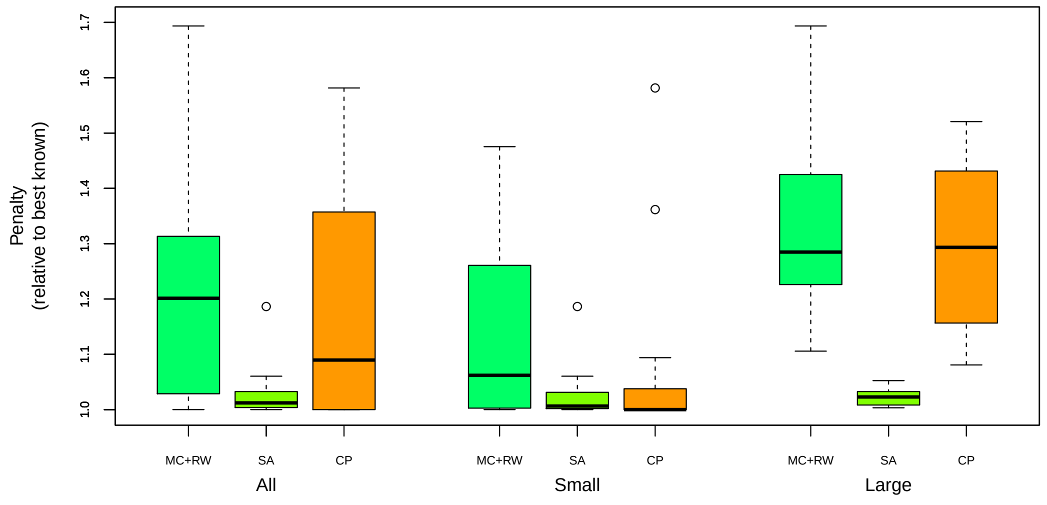


Figure 3.4: Results for the TLSP-S using MC+RW, SA, and CP, with a timeout of 10 minutes per instance. The center and right groups show results only for small (≤ 20 projects) and large instances, respectively. Results were scaled by the best solution known for each instance.

The differences between the results for small and large instances indicates that the number of projects (and thus jobs) is the main factor in determining the time required to solve an instance. In contrast, neither the number of time slots in the scheduling period, nor the number of resources available for each job seem to have a decisive impact on the difficulty of an instance.

We also repeated our experiments with the same configuration for SA with a longer timeout of one hour. Table 3.6 shows the results of these experiments, again compared with the results for the CP model (also with a timeout of one hour).

With the increased time budget, SA could find feasible solutions for all runs on the generated instances and for most runs on the real-world instances. As with the shorter time limit, the solutions produced by SA for all instances where CP could not find optimal solutions are better than those found by CP, in some cases by more than 30%. In particular, this includes all large instances.

For those small instances where optimal solutions have been found, SA achieved solutions with the same or very close penalties.

Compared to the results with a shorter time limit, the penalty for large instances (>20 projects) has improved by nearly 3.8% on average with SA. CP could find optimal solutions for three of the four remaining small instances. However, the results for large instances improved by less than 1% on average.

Figure 3.5 shows a comparison of the performance of the two solvers, both overall and

#	SA			CP	
	#Feas.	**Avg**	**Best**		
1	5/5	98.0	**98**	**98**	*
2	5/5	73.0	**73**	**73**	*
3	5/5	152.4	151	**149**	*
4	5/5	105.4	**105**	**105**	*
5	5/5	300.8	292	**283**	*
6	5/5	191.0	180	**162**	*
7	5/5	307.0	**307**	**307**	*
8	5/5	310.2	**310**	**310**	*
9	5/5	501.0	**501**	**501**	*
10	5/5	564.6	**564**	740	
11	5/5	873.4	872	**856**	*
12	5/5	664.8	663	**656**	*
13	5/5	353.6	352	**340**	*
14	5/5	422.2	422	**420**	*
15	5/5	1086.8	**1086**	1647	
16	5/5	1142.2	**1141**	1561	
17	5/5	1206.0	**1195**	1284	
18	5/5	1361.0	**1360**	1820	
19	5/5	2233.0	**2196**	2650	
20	5/5	2284.6	**2261**	2888	
21	5/5	683.6	683	**679**	*
22	5/5	766.6	765	**765**	*
23	5/5	2276.0	**2200**	3487	
24	5/5	1799.4	**1782**	2452	
25	5/5	2737.8	**2598**	3278	
26	5/5	2633.4	**2605**	3894	
27	5/5	2159.8	**2155**	3130	
28	5/5	2341.2	**2333**	2569	
29	5/5	4204.8	**4038**	4539	
30	5/5	4636.6	**4601**	5904	
Lab1	3/5	3430.7	**3411**	5187	
Lab2	5/5	2618.6	**2580**	3575	
Lab3	3/5	2624.0	**2614**	3070	

Table 3.6: Comparison of results for the TLSP-S with an extended timeout of one hour. Column *Best* contains the best solution found, *#Feas.* the number of feasible solutions found (out of 5) and *Avg* the average penalty over all feasible solutions. The best solution for each instance found by any of the solvers is marked in bold. Solutions for CP marked with "*" are optimal for the instance.

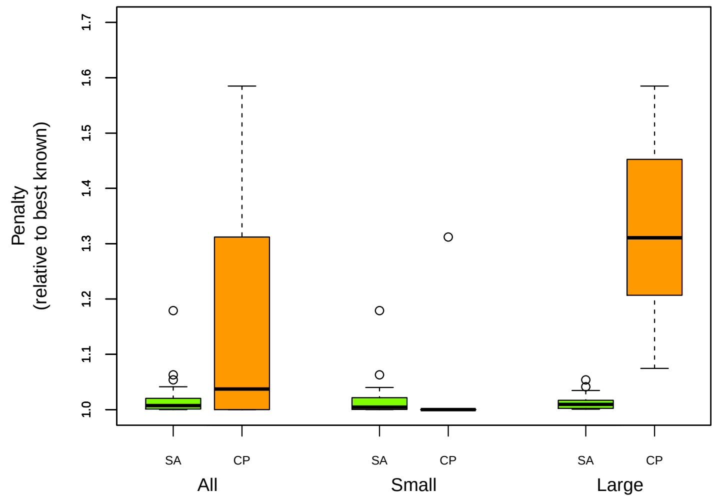


Figure 3.5: Results for the TLSP-S using SA and CP, with a timeout of one hour per instance. The center and right groups show results for small (≤ 20 projects) and large instances, respectively. Results were scaled by the best solution known for each instance.

separated into small and large instances. With the increased running time, CP finds the best known (indeed optimal) solution for all but one of the small instances, but is still outperformed by SA both on the larger instances and in overall quality.

Our results are in line with previous findings for RCPSP (e.g. by [PPB20], for a recent example) that exact approaches are suitable mostly for instances with fewer activities and that heuristics are needed to find good solutions for instances of larger sizes. A possible explanation for this may be that exact approaches in general have to examine a substantial part of the search space, which grows exponentially with the number of activities, while heuristics can often find more efficient paths through that search space, at the cost of losing any guarantees about the final solution reached.

TLSP

Table 3.7 lists the final evaluation results for the TLSP of MC+RW, ILS using MC+RW for the inner loop, and SA. It shows that MC+RW was unable to find feasible solutions for many of the large instances. The addition of ILS improves those results quite a bit, and produces better results for every single instance than MC+RW alone. However, it also had troubles with the large instances, including the real-world instances.

The best results were achieved using SA, which could find at least some feasible solutions for all instances and found the best known solution for many of them, in particular large instances.

Of particular interest are the real-world instances, which proved quite difficult to solve

#	MC+RW			ILS (MC+RW)			SA		
	#Feas	**Avg**	**Best**	**#Feas**	**Avg**	**Best**	**#Feas**	**Avg**	**Best**
1	15/15	58.0	**58**	15/15	58.0	**58**	15/15	58.0	**58**
2	15/15	71.0	**71**	15/15	71.1	**71**	15/15	72.4	72
3	15/15	146.9	143	15/15	145.6	**141**	15/15	156.1	147
4	15/15	108.0	105	15/15	107.2	105	15/15	114.5	**103**
5	15/15	289.2	276	15/15	271.6	**263**	15/15	303.3	296
6	15/15	163.7	160	15/15	162.4	**154**	15/15	165.1	157
7	15/15	325.5	318	15/15	317.8	310	15/15	300.9	**296**
8	15/15	317.1	305	15/15	310.9	302	15/15	302.4	**292**
9	15/15	577.3	548	15/15	531.9	509	15/15	470.6	**456**
10	1/15	878.0	878	15/15	702.3	618	15/15	566.5	**551**
11	12/15	1130.3	1050	14/15	1014.4	984	15/15	938.4	**912**
12	15/15	782.9	766	15/15	745.5	694	15/15	675.8	**666**
13	15/15	336.5	329	15/15	326.7	**321**	15/15	338.2	327
14	15/15	463.0	453	15/15	452.3	443	14/15	420.4	**418**
15	0/15	-	-	9/15	1539.2	1459	15/15	1063.5	**1014**
16	4/15	1475.3	1381	10/15	1347.8	1306	13/15	1236.7	**1216**
17	15/15	1438.6	1358	15/15	1298.1	1233	15/15	1185.3	**1140**
18	2/15	1930.5	1891	12/15	1706.5	1638	15/15	1527.4	**1500**
19	0/15	-	-	0/15	-	-	14/15	2239.9	**2133**
20	0/15	-	-	1/15	3027.0	3027	15/15	2489.7	**2391**
21	15/15	903.1	853	15/15	790.9	733	15/15	673.9	**632**
22	15/15	952.5	911	15/15	968.1	884	15/15	784.6	**755**
23	0/15	-	-	0/15	-	-	12/15	2167.7	**2070**
24	0/15	-	-	11/15	2290.3	2212	15/15	1869.3	**1807**
25	0/15	-	-	0/15	-	-	10/15	2897.6	**2601**
26	0/15	-	-	1/15	3513.0	3513	13/15	2971.3	**2809**
27	0/15	-	-	4/15	3055.5	2883	15/15	2124.3	**2017**
28	15/15	2828.9	2707	15/15	2744.7	2663	15/15	2516.5	**2468**
29	0/15	-	-	0/15	-	-	10/15	4358.5	**3965**
30	0/15	-	-	0/15	-	-	14/15	5104.1	**4989**
Lab1	0/15	-	-	0/15	-	-	4/15	3558.0	**3511**
Lab2	0/15	-	-	0/15	-	-	1/15	2811.0	**2811**
Lab3	0/15	-	-	0/15	-	-	3/15	2616.7	**2606**

Table 3.7: Evaluation results of the different metaheuristic solution approaches for the TLSP. Columns *#Feas* contain the number of feasible solutions found, *Best* the best solution out of all runs and *Avg* the average penalty over all feasible solutions. The best solution for each instance found by any of the solvers is marked in bold.

for SA. We performed a separate analysis for these three instances to determine the cause of this discrepancy. While the generated instances closely follow the real-world data, there is one aspect that was not yet considered at the time the instances were created: Employee absences were modeled only later as blocking tasks over the period of absence, and thus appear only in the real-world instances. Intuitively, absences (partially) split the scheduling period into smaller intervals, which requires more smaller jobs that neatly fit into the gaps between the absences. Thus, the presence of absences requires a heavier focus on (re)grouping moves, which the current configuration SA with its low weight for TaskTransfer, Split and Merge is lacking. To test this hypothesis, we also performed experiments on the real-world instances with increased weights (0.05) for TaskTransfer and SplitMerge, scaling down the weights of the other neighborhoods accordingly. In this configuration, the solver could find feasible solutions for 11 out of 15 total runs (5 runs per instance). Further, SA using the original weights could find feasible solutions for 14 out of 15 runs on modified versions of the real-world instances, where all employee absences have been removed. These results indicate that employee absences require special considerations in order to find feasible schedules quickly in practice, due to the fragmentation of the scheduling period they induce.

We also compare our results for SA to the CP and VLNS based approaches [DGMM20]. Since that paper used timeouts of two hours per instance, we repeated our evaluations with this longer timeout. The results can be seen in Table 3.8.

With this longer timeout, SA could find feasible solutions for nearly all of the runs, including the real-world instances. While SA clearly outperforms CP, VLNS finds slightly better solutions on most instances.

SA still manages to find new best known solutions for the three instances where CP and VLNS could not find solutions at all, as well as for two additional instances (including one of the real-world instances). In general, the relative performance of SA is better on larger instances.

The situation looks different when we compare the performance of SA and VLNS using the original timeout of 10 minutes (Table 3.9). While VLNS still gives better results for the smaller instances, SA finds better solutions for many of the larger ones (see Figure 3.6 for aggregate results).

Comparison between TLSP-S and TLSP

When comparing results between the TLSP-S and the full TLSP, we need to keep a few things in mind: The first is the difference in experimental setup which gives the solvers for TLSP-S an advantage due to being evaluated on a faster machine.

More importantly, we need to take into account that for the TLSP-S, the solver can (and must) use the grouping provided in the base schedule of each instance. On the one hand, the knowledge that the given grouping is already feasible makes it easier to find a conflict-free schedule. The restriction to the fixed grouping also allows the solver

#	SA			VLNS			CP
	#Feas	**Avg**	**Best**	**#Feas**	**Avg**	**Best**	
1	5/5	58.0	58	5/5	57.0	**57**	**57**
2	5/5	72.0	72	5/5	71.0	**71**	**71**
3	5/5	149.4	147	5/5	141.0	**141**	142
4	5/5	106.2	105	5/5	101.0	**101**	119
5	5/5	284.8	263	5/5	240.0	**240**	244
6	5/5	157.6	157	5/5	140.0	**140**	180
7	5/5	296.8	291	5/5	283.0	**283**	355
8	5/5	298.2	293	5/5	283.6	**283**	310
9	5/5	461.2	444	5/5	419.0	**415**	713
10	5/5	557.8	535	5/5	512.2	**499**	1010
11	5/5	915.2	905	5/5	822.8	**816**	1011
12	5/5	667.6	663	5/5	646.8	**643**	764
13	5/5	334.4	331	5/5	308.4	**307**	337
14	5/5	419.8	417	5/5	410.4	**410**	447
15	5/5	992.0	961	5/5	883.0	**867**	1819
16	1/5	1218.0	1218	5/5	1111.4	**1109**	1599
17	5/5	1159.2	1137	5/5	1075.8	**1038**	1416
18	5/5	1475.2	1450	5/5	1341.0	**1328**	1841
19	5/5	1956.8	1869	5/5	1860.0	**1824**	2751
20	3/5	2357.3	2304	5/5	2266.8	**2193**	3146
21	5/5	629.2	602	5/5	547.0	**542**	922
22	5/5	769.0	761	5/5	744.8	**742**	1062
23	5/5	1747.6	**1613**	0/5	-	-	-
24	5/5	1801.4	**1780**	0/5	-	-	-
25	5/5	2280.0	2213	5/5	2217.6	**2135**	4174
26	5/5	2713.0	2667	5/5	2589.6	**2558**	3861
27	5/5	1999.2	1965	5/5	1769.6	**1723**	3874
28	5/5	2470.4	2439	5/5	2258.4	**2235**	3180
29	5/5	3645.2	**3562**	0/5	-	-	-
30	4/5	4605.3	**4532**	5/5	4822.6	4714	6508
Lab1	4/5	3404.3	3389	5/5	3377.0	**3296**	4991
Lab2	5/5	2643.0	**2539**	5/5	2669.2	2595	3339
Lab3	5/5	2609.6	2592	5/5	2599.0	**2590**	2979

Table 3.8: Comparison of results for SA with those of [DGMM20] (VLNS and CP), with a timeout of 2 hours. Columns *#Feas* contain the number of feasible solutions found (out of 5 runs), *Best* the best and *Avg* the average penalty over all feasible solutions. The best solution for each instance found by any of the solvers is marked in bold.

#	SA			VLNS			CP
	#Feas	**Avg**	**Best**	**#Feas**	**Avg**	**Best**	
1	15/15	58.0	58	5/5	57.0	**57**	**57**
2	15/15	72.4	72	5/5	71.0	**71**	**71**
3	15/15	156.1	147	5/5	141.0	**141**	142
4	15/15	114.5	103	5/5	101.0	**101**	119
5	15/15	303.3	296	5/5	242.2	**240**	281
6	15/15	165.1	157	5/5	140.0	**140**	180
7	15/15	300.9	296	5/5	287.0	**283**	397
8	15/15	302.4	292	5/5	284.6	**283**	310
9	15/15	470.6	456	5/5	442.2	**429**	825
10	15/15	566.5	551	5/5	590.2	**547**	1038
11	15/15	938.4	912	5/5	860.8	**840**	1020
12	15/15	675.8	666	5/5	660.6	**653**	781
13	15/15	338.2	327	5/5	311.8	**309**	337
14	14/15	420.4	418	5/5	415.0	**412**	504
15	15/15	1063.5	**1014**	5/5	1106.2	1025	1830
16	13/15	1236.7	1216	5/5	1184.4	**1175**	1669
17	15/15	1185.3	**1140**	5/5	1166.6	1150	1445
18	15/15	1527.4	1500	5/5	1482.2	**1436**	1898
19	14/15	2239.9	**2133**	5/5	2419.2	2350	2772
20	15/15	2489.7	**2391**	5/5	2986.6	2955	3175
21	15/15	673.9	632	5/5	596.8	**570**	992
22	15/15	784.6	**755**	5/5	775.0	763	1113
23	12/15	2167.7	**2070**	0/5	-	-	
24	15/15	1869.3	**1807**	0/5	-	-	
25	10/15	2897.6	**2601**	0/5	-	-	
26	13/15	2971.3	**2809**	5/5	3345.4	3109	3883
27	15/15	2124.3	**2017**	5/5	2750.6	2473	3984
28	15/15	2516.5	2468	5/5	2407.2	**2372**	3180
29	10/15	4358.5	**3965**	0/5	-	-	
30	14/15	5104.1	**4989**	0/5	-	-	
Lab1	4/15	3558.0	**3511**	5/5	4080.8	3923	5004
Lab2	1/15	2811.0	2811	5/5	2896.8	**2779**	3410
Lab3	3/15	2616.7	**2606**	5/5	2687.4	2646	3007

Table 3.9: Comparison of results for SA on the TLSP (see Table 3.7) with those of [DGMM20] (VLNS and CP), under a time limit of 10 minutes. Columns *#Feas* contain the number of feasible solutions found (out of 5 runs), *Best* the best and *Avg* the average penalty over all feasible solutions. The best solution for each instance found by any of the solvers is marked in bold.

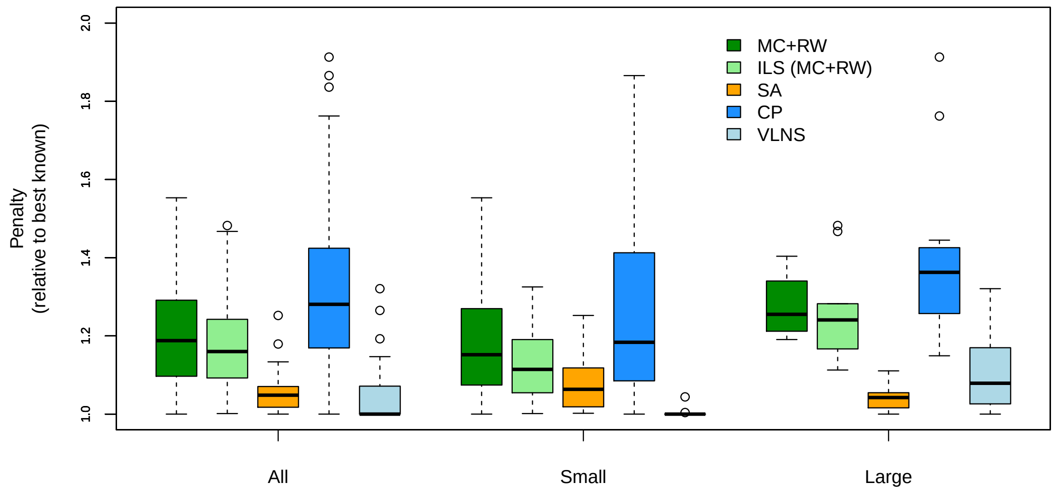


Figure 3.6: Comparison of results for the TLSP using the approaches in Tables 3.7 and 3.9. Results have been scaled by the best known solution for each instance. The center and right groups contain only small ($\leq$ 20 projects) and large instances, respectively. Only feasible results were included, which heavily affects the plots for MC+RW and ILS on large instances.

to simplify the search by precomputing all job properties and ignore any regrouping neighborhoods. On the other hand, while the grouping is guaranteed to be feasible, it is likely not optimal. It is easy to see that any optimal solution for the TLSP is at least as good as the optimal solution of the corresponding TLSP-S instance with any given grouping. Therefore having a flexible grouping has the potential of better solutions, in particular when a good initial grouping for TLSP-S is not known and cannot be easily guessed.

Table 3.10 contains the results for SA for both the TLSP-S and the TLSP, and also a variant of the TLSP where the known grouping of the TLSP-S was used to construct the initial solution (column TLSP*). An aggregate comparison of the three approaches is shown in Figure 3.7. SA for the TLSP manages to produce solutions of the same quality as SA for the TLSP-S, despite the drawback of the unknown grouping (and slower machine environment). The slight overall advantage of the TLSP can mostly be credited to a few outliers, in particular in the first instance, where a different grouping admits solutions with nearly half the penalty of the TLSP-S optimum. Providing the known grouping of the TLSP-S instance to the initial solution construction in the TLSP results in a slight improvement to several solutions, but does not lead to additional feasible solutions. Notably it does not help with finding feasible solutions to the real-world instances.

#	TLSP-S			TLSP			TLSP*		
	#Feas	**Avg**	**Best**	**#Feas**	**Avg**	**Best**	**#Feas**	**Avg**	**Best**
1	10/10	98.0	98	15/15	58.0	58	5/5	58.0	58
2	10/10	73.0	73	15/15	72.4	72	5/5	72.0	72
3	10/10	156.4	152	15/15	156.1	147	5/5	151.4	149
4	10/10	105.0	105	15/15	114.5	103	5/5	107.8	103
5	10/10	300.1	287	15/15	303.3	296	5/5	301.0	296
6	10/10	192.2	177	15/15	165.1	157	5/5	158.0	157
7	10/10	307.4	307	15/15	300.9	296	5/5	300.6	297
8	10/10	312.0	310	15/15	302.4	292	5/5	301.4	297
9	10/10	502.7	501	15/15	470.6	456	5/5	475.8	467
10	10/10	565.3	564	15/15	566.5	551	5/5	568.0	555
11	10/10	879.0	874	15/15	938.4	912	5/5	938.8	930
12	10/10	668.0	663	15/15	675.8	666	5/5	683.4	671
13	10/10	352.1	352	15/15	338.2	327	5/5	337.6	330
14	10/10	425.7	422	14/15	420.4	418	5/5	423.0	417
15	10/10	1090.6	1087	15/15	1063.5	1014	5/5	1038.4	1010
16	10/10	1155.2	1143	13/15	1236.7	1216	4/5	1234.5	1231
17	10/10	1234.0	1195	15/15	1185.3	1140	5/5	1175.6	1153
18	10/10	1375.3	1364	15/15	1527.4	1500	5/5	1522.2	1497
19	10/10	2337.0	2277	14/15	2239.9	2133	5/5	2247.2	2224
20	10/10	2360.6	2312	15/15	2489.7	2391	4/5	2493.8	2426
21	10/10	686.6	683	15/15	673.9	632	5/5	697.0	677
22	10/10	771.9	767	15/15	784.6	755	5/5	795.2	787
23	6/10	2476.3	2393	12/15	2167.7	2070	5/5	2100.8	2044
24	10/10	1852.5	1808	15/15	1869.3	1807	5/5	1856.2	1803
25	8/10	3050.9	2908	10/15	2897.6	2601	4/5	2779.3	2634
26	10/10	2805.0	2724	13/15	2971.3	2809	5/5	2926.4	2897
27	10/10	2191.3	2176	15/15	2124.3	2017	5/5	2082.6	2034
28	10/10	2375.8	2367	15/15	2516.5	2468	5/5	2520.6	2469
29	9/10	4428.4	4208	10/15	4358.5	3965	3/5	4150.0	3697
30	9/10	4896.8	4828	14/15	5104.1	4989	3/5	5233.0	5125
Lab1	0/10	-	-	4/15	3558.0	3511	0/5	-	-
Lab2	5/10	2689.4	2658	1/15	2811.0	2811	1/5	2673.0	2673
Lab3	9/10	2718.2	2646	3/15	2616.7	2606	1/5	2641.0	2641

Table 3.10: Results for the TLSP-S (results from Table 3.5) and the TLSP (Table 3.7) using SA. The third part, TLSP*, lists results for SA where the initial solution was created using the known grouping of TLSP-S. Columns *#Feas* contain the number of feasible solutions found (out of 15 runs for TLSP, 10 runs for TLSP-S, and 5 runs for TLSP*), *Best* the best solution out of all runs and *Avg* the average penalty over all feasible solutions.

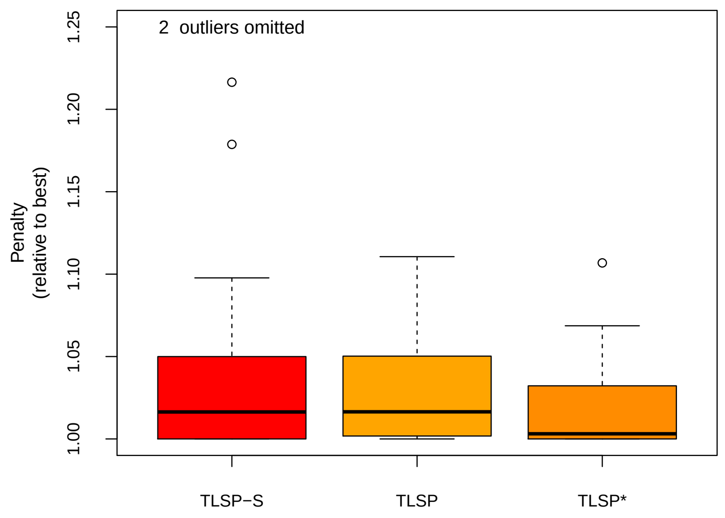


Figure 3.7: Comparison between results for TLSP-S, TLSP, and TLSP* using SA. The penalties for each instance were scaled by the best found solution among those three approaches.

Neighborhood analysis

In order to determine the impact of each neighborhood on the solution quality, we repeated the evaluations for SA on the TLSP, each time with one of the neighborhoods removed. Table 3.11 shows the results of these experiments.

Neighborhoods	% feasible	Average quality
All	90.3%	1
No JobOpt	2.4%	1.86
No EquipmentChange	60.0%	1.03
No TaskTransfer	78.8%	1.03
No SplitMerge	74.5%	1.04

Table 3.11: The performance of SA, using the same configuration as in Table 3.7, but missing one of the neighborhoods. Shown are the percentage of feasible solutions found (out of 5 runs for each of the 33 test instances), and the average quality of the feasible solutions found, scaled to the average performance of SA with all neighborhoods (“All”).

It is immediately obvious that the JobOpt neighborhood is absolutely essential to find feasible solutions at all. This is not surprising, given that it is virtually the only neighborhood that includes mode, time slot and resource changes (the regrouping neighborhoods allow for some very restricted adjustments, but not enough to find non-trivial feasible solutions).

A similar effect appears when removing the EquipmentChange neighborhood, though much less pronounced. Since JobOpt does not modify equipment assignments, the solver basically has to try to find feasible solutions using the equipment assigned in the initial (random) construction. An interesting observation is that for those instances where a feasible solution could be found, the solution quality is not much worse than the baseline performance. This might be related to the fact that equipment assignments don't appear in the objective function.

Removing the task transfer neighborhood has the least effect on the solutions found, probably because its effect can mostly be replicated by a corresponding split, followed by a merge. Still, this neighborhood provides a way of fine-tuning the grouping that is helpful in eliminating the last remaining conflicts.

Without the option to split and merge jobs, the number of jobs for each project (and for each family) cannot be changed from those in the initial, greedy grouping. Since this grouping often leads to infeasible schedules, it is not surprising that the task transfer neighborhood alone cannot repair the conflicts in many cases. Despite the small weight of this neighborhood (only 0.4% of moves), it has a marked impact on the ability of the solver to find feasible solutions.

3.5 Summary

In this chapter, we have introduced and evaluated new metaheuristic methods for both the TLSP-S and the TLSP. In particular, we have developed a set of specialized neighborhoods to deal with the various aspects of both problems. This set can be adjusted and extended to cover different problem variants, such as via inclusion of the regrouping neighborhoods to deal with the flexible task grouping of the TLSP. In our analysis, we have shown that these neighborhoods work well together and each contribute to the performance of the overall solver. We have incorporated these neighborhoods within several well-known metaheuristics, including Simulated Annealing (SA), a combination of the MinConflicts heuristic and random walk (MC+RW), as well as an Iterated Local Search (ILS) procedure using either of the other two metaheuristic for its inner loop.

Our experimental evaluation has shown that in particular SA is able to produce high-quality results for both problem variants. MC+RW had trouble finding feasible solutions for some of the larger instances. The inclusion within ILS helped MC+RW significantly, but still not enough to reliably find conflict-free schedules for realistic sized instances. Conversely, ILS with SA was unable to improve over the performance of SA on its own. Apparently, SA profits more from a slower cooling scheme than from short cooling cycles with regular perturbations and reheatings.

These metaheuristic approaches have also provided the background and implementation framework for exact and hybrid approaches using Constraint Programming (CP) and Very Large Neighborhood Search (VLNS) [GMM19a, DGMM20] respectively. CP turned out to work well on small instances, where it managed to prove optimality for most instances of TLSP-S. In contrast, VLNS provides very good results over all instances sizes and is currently the state-of-the-art solution method for both problem variants. Still, it was unable to outclass SA on large instances, particularly under short time limits.

A separate comparison between the TLSP-S and the TLSP shows that while the former is clearly easier to solve, this is counteracted by the latter's flexibility in finding better solutions. In particular when the information disparity is compensated by allowing solvers for the TLSP the knowledge of a "known-good" initial grouping, as is assumed in TLSP-S instances, the schedules with flexible grouping are clearly better.

CHAPTER 4

Self-Adaptation using Cross-Domain Hyper-Heuristics

The solution approaches for TLSP described in earlier publications [GMM19b, DGMM20] and the previous chapter of this thesis were specifically developed and tuned for the constraints and objectives of the TLSP(-S), including the assumption of uniform weights. However, experience has shown (see also Chapter 5) that conditions in practice will not always match these assumptions, due to priorities on the objectives, change of requirements over time, or deployment in new laboratories with their own weights and requirements. One approach in this case would be to re-evaluate the existing approaches and repeat the parameter tuning each time to ensure that they are still suited for the current situation. However, this would be very expensive in terms of effort and time and is not sustainable in the long term.

Far more promising is the search for solution approaches that are able to automatically adapt their behavior to the current situation, which have the potential to remain useful for a wide range of scenarios and settings. Hyper-heuristics [CKS01] are one example of this kind of general and adaptive methods. They use a set of low-level and problem specific heuristic components, which they combine into a general problem solving algorithm. Powerful and efficient candidates for the low-level components are known for many problem domains, and can be defined in a modular and general way that allows the hyper-heuristic to select the best components for any particular problem configuration, individual instance to be solved, or even current state of the search.

In this chapter, we introduce several new problem independent hyper-heuristics, that are capable of producing good solutions on a wide range of combinatorial optimization problems, including the TLSP. The work described in this chapter includes content of three publications: A technical report [MM21a] introducing hyper-heuristics based on Self-Adaptive Large Neighborhood Search (SA-LNS), an article accepted for publication at IJCAI 2022 [MM22b] containing our approaches using reinforcement learning (RL), and an abstract presented at EURO 2022 [MM22a] covering the modeling and evaluation of the TLSP(-S) as a problem domain for hyper-heuristics. The source code for all hyper-heuristics described in this thesis is available for download as a git repository at `https://gitlab.tuwien.ac.at/florian.mischek/hyper-heuristics-public`.

We first give an introduction to hyper-heuristics in general and the cross-domain optimization problem, including a review of relevant literature, in Section 4.1. In the following Section 4.2, we introduce our hyper-heuristic algorithms based on SA-LNS, together with an empirical evaluation on multiple problem domains. Another hyper-heuristic approach using RL, including an analysis of modeling variants for the algorithm components, is presented and evaluated in Section 4.3. Finally, in Section 4.4 we model the TLSP in the form of a problem domain for selection hyper-heuristic and apply both our hyper-heuristics and state-of-the-art hyper-heuristics from the literature to this new problem domain. The last section, Section 4.5 contains a summary and concluding remarks.

4.1 Hyper-heuristics

Hyper-heuristics are a family of high-level problem solving methods whose defining characteristic is that they operate over a search space of other low-level heuristics (LLHs), instead of directly over a space of solutions [BHK+19]. In this context, the term was first coined in 2000 [CKS01], though the general idea has been around since the 1960s.

The classification scheme for hyper-heuristics proposed by Burke et al.[BHK+10] distinguishes between *selection* hyper-heuristics, which select from among a given list of LLHs and *generation* hyper-heuristics which automatically assemble heuristic operators from smaller components and building blocks. Further distinction is made by whether the LLHs used *construct* a solution from scratch or *perturb* of an existing solution, and by the type of learning, if any, employed by the hyper-heuristic.

Since their inception, hyper-heuristics have seen a large amount of work, for a diverse range of search and optimization problems. A thorough survey of previous works was performed in 2013 [BGH+13], with a revised update to the 2010 classification scheme published in 2019 [BHK+19].

4.1.1 The Cross-Domain Heuristic Search Challenge

A big boost to the popularity of hyper-heuristics was provided by the Cross-Domain Heuristic Search Challenge (CHeSC) 2011 [BGH+11]. For this competition, participants had to develop perturbative selection hyper-heuristics that were then evaluated over a wide range of different problem domains. The problem domain implementations were provided by the organizers, including the solution representation and initialization, the objective function, and a set of perturbative LLHs (or operators) for each domain.

As the goal was to develop general and problem independent approaches, most of the problem specific information was hidden from the hyper-heuristics, a concept called

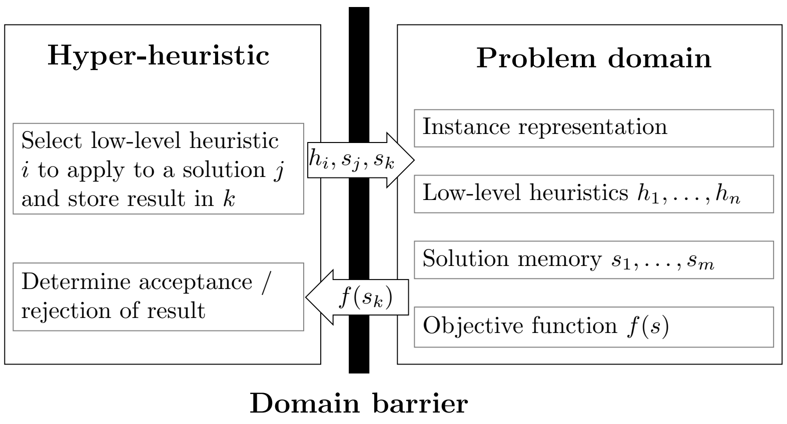

Hyper-heuristic
Select low-level heuristic i to apply to a solution j and store result in k
Determine acceptance / rejection of result
h_i, s_j, s_k
$f(s_k)$
Problem domain
Instance representation
Low-level heuristics $h_1, \dots, h_n$
Solution memory $s_1, \dots, s_m$
Objective function $f(s)$
Domain barrier

Figure 4.1: Problem setting of the CHeSC 2011. Participants had to develop the hyper-heuristics, while the problem domain implementations were provided. The domain barrier limits the exchange of information to ensure problem independence. The problem domain provides indices and general types of LLHs and informs the hyper-heuristic about the current objective value of a stored solution.

domain barrier (see Figure 4.1). Both stored solutions and LLHs could only be accessed via their respective indices. The only additional information made available was the general type of each LLH. The four types of operators available are *mutations*, which apply a small and mostly random change to a solution, *ruin-and-recreate* operators, which destroy a part of the solution and then attempt to repair it heuristically, *local searches*, which apply one or several steps of a local search procedure, and *crossovers* which combine two different solutions.

The six problem domains used for the CHeSC 2011 were the maximum satisfiability problem (MaxSAT), bin packing (BP), flow shop (FS), personnel scheduling (PS), the traveling salesman problem (TSP) and the vehicle routing problem (VRP). Table 4.1 shows the number of low-level heuristics included in the problem domain implementations for these domains.

The first four of these domains were known in advance, while the latter two were hidden until after the competition. For the four initially known problem domains, ten benchmark instances of different sizes were provided. Of these, three were randomly chosen to be used for the final scoring of the competition, plus two additional ones per domain that were previously unknown. The two hidden domains were evaluated on five previously unknown instances each. Points were awarded to a participant according to the ranking of their result on each instance. These points follow a distribution inspired by the Formula 1 scoring, with 10 points awarded to the best result, and 8, 6, 5, 4, 3, 2, 1 points for the subsequent ranks, respectively. The total score of a participant is then the sum of their instance scores, which determines the final ranking.

The competition was won by Mısır et al.[MVDCVB12] whose AdapHH algorithm combines

Domain	**MU**	**RR**	**LS**	**CO**	**Total**
MaxSAT	4	1	2	2	9
BP	3	2	2	1	8
FS	5	2	4	3	14
PS	1	3	4	3	11
TSP	5	1	6	3	15
VRP	4	2	4	2	12

Table 4.1: Low-level heuristics provided for each of the six problem domains, distributed across the four types: Mutation (MU), ruin-and-recreate (RR), local search (LS), and crossovers (CO).

several adaptive mechanisms to manage a set of active LLHs, determine the acceptance of results, and configure search parameters. This algorithm was revised and updated by Adriaensen and Nowé in 2016 [AN16], who streamlined it by removing components that added complexity without improving the solution quality.

The second place was achieved by a self-adaptive Variable Neighborhood Search (VNS-TW) by Hsiao et al. [HCF12], which alternates shaking and local search phases. The third place entry ML by Larose [Lar11], following a single-agent version of an algorithm by Meignan et al.[MKC10], also used cycles of diversification and intensification phases, with adaptive move selection strategies and acceptance criterion.

Overall, there were 20 participating entries to the CHeSC 2011, whose hyper-heuristics cover a wide range of different approaches, including both single-point algorithms that work on a single solution at the time and population-based methods.

The hyper-heuristics developed for the competition were implemented in Java using the software framework HyFlex [OHC+12]. This framework provides interfaces for problem domains and hyper-heuristics, which enables communication between them while enforcing the limits placed by the domain barrier. Since 2011, HyFlex has become the de-facto standard framework for selection hyper-heuristics [BGH+13]. Authors have proposed additional problem domains (e.g. the 0-1 Knap Sack, Quadratic Assignment and Max-Cut problems [AON15]) or extensions to the framework (e.g. by Ochoa et al. [OWHC12]).

In addition, several new hyper-heuristic approaches have been developed using HyFlex. A considerable number of them base their approaches on Iterated Local Search (ILS), that is cycles of a perturbation or diversification phase, followed by a local search or intensification phase. ILS was already successfully used by several CHeSC participants, including VNS-TW [HCF12], ML [Lar11], and PHunter [CXIC12]. Later examples of such an approach are Fair-Share ILS (FS-ILS) [ABN14], which uses a selection and acceptance model that was developed in a semi-automatic offline learning process, and most recently TS-ILS [AOO21], which is partly based on FS-ILS, but uses Adaptive

Thomson Sampling to learn promising perturbation operations.

Of particular interest is QHH [CWL18], which uses Q-learning (a popular RL algorithm) to learn the high-level selection and acceptance operators that are applied within an ILS framework. In contrast, in this thesis we use RL to directly learn which LLHs should be applied, without explicitly distinguishing between perturbative and search operators.

There are also other approaches that are not based on ILS: Sabar and Kendall proposed a Monte-Carlo tree search hyper-heuristic (MCTS-HH) [SK15], which stores promising sequences of LLHs in a search tree. A population-based approach which automatically designs hyper-heuristic operators using Gene Expression Programming (GEP-HH) was developed by Sabar et al. [SAKQ15] in 2015. Individuals each represent a combination of a heuristic selection rule and an acceptance criterion. The algorithm also includes an adaptive memory mechanism, which ensures that the population contains both high-performing and diverse strategies. Asta and Özcan described a tensor-based hyper-heuristic TeBHA-HH [AÖ15] which used tensor-factorization to analyze the relationships between LLHs and hyper-heuristic performance. Finally, Ferreira et al. modeled the problem as a Multi-Armed Bandit (MAB) [FGTRP15], which is a simplified variant of the RL setting with only a single state, and discuss several variants of their hyper-heuristics. They achieved the best results by scaling the achieved rewards by the number of times a low-level heuristic was selected recently, to promote exploration of less frequently used operators (FRAMAB).

A common property of most hyper-heuristics is that they feature adaptive components that change their behavior based on feedback about the progress of the search. This allows them to adapt to the different environments posed by the diverse problem domains and instance properties they have to solve. In many cases, this adaptivity or learning uses some reward or penalty scheme which is similar to RL. However, to be strictly considered RL, two features are required [CWL18, SB18]: Unguided trial-and-error exploration of the state-action space leading to the discovery of good or promising actions, and delayed rewards, which may be realized only after several states (and actions) have been explored. There are a number of selection hyper-heuristics which fulfill these criteria and use RL or RL-inspired components, such as QHH [CWL18] and ML [Lar11], both of which still follow the general ILS scheme. Di Gaspero and Urli [DGU12], who also participated in the CHeSC, proposed a hyper-heuristic that directly matches the RL setting. Their paper includes an analysis of different modeling variants for several algorithm components, however its performance at the competition was not particularly successful. In this thesis, we investigate new approaches for RL-based hyper-heuristics, to show that also a direct application of RL mechanisms can successfully produce good solutions.

Chuang and Smith [CS17, Chu20] investigated an algorithmic model (FI) based on sampling solution chains of varying length. If no solution in a chain leads to an improvement, the current solution is rejected and reset back to the previously best known solution. They showed some theoretical optimality guarantees under the assumption of uniformly random LLH selection for this model (FI-Uniform) and found good solutions in their empirical evaluation even without any learning or adaptation. By extending their model

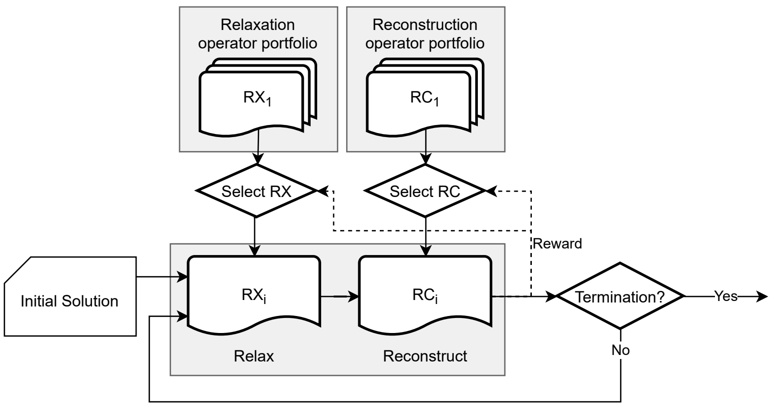


Figure 4.2: Structure of SA-LNS according to [LG07]. Iterations consist of an application of a relaxation operator, followed by a reconstruction operator. Each operator is chosen from a portfolio, taking into account the operators' previous performance.

with basic LLH weighting mechanisms, they were able to further improve their results [Chu20]: In FI-Pruning, low-level heuristics are removed if they did not participate in a successful solution chain after a fixed warmup period. Similarly, FI-Frequency used the performance of low-level heuristics during the warmup period to calculate (fixed) operator weights used for the remainder of the search, while FI-Bigram did the same with pairs of low-level heuristics. The hyper-heuristics we propose in Section 4.3 combine the powerful adaptivity of RL with the solution chains described in [CS17].

4.2 Self-Adaptive Large Neighborhood Search

As mentioned in the previous section, hyper-heuristics which follow the diversification-intensification cycle of ILS have been very successful. An algorithm scheme that works along the same lines is the algorithm based on Self-Adaptive Large Neighborhood Search (SA-LNS) published in 2007 by Laborie and Godard [LG07] to solve scheduling problems. While not described as a hyper-heuristic in the paper itself, the SA-LNS shares many of the characteristics with them.

As in ILS, SA-LNS works by iteratively relaxing parts of a schedule (diversification) and then re-optimizing the relaxed solutions (intensification). Both relaxation and reconstruction operators are dynamically selected from a portfolio, with weights that are updated according to each operator's performance (an overview of the algorithm is shown in Figure 4.2).

More formally, let $w(o)$ be the weight of operator o, which can be either a relaxation or a reconstruction operator. At each iteration, SA-LNS selects and applies one relaxation operator RX_i and one reconstruction operator RC_j from their respective portfolios, using roulette-wheel selecting according to the current weights. The chosen operators are applied to the current solution in order. If this improves the solution, a reward $r = \frac{\Delta c}{\Delta t}$ is obtained, where Δc is the difference in objective function and Δt is the time taken by both operators. If the new solution is worse, it is rejected and the reward is 0 instead. The operator weights are then updated as follows for both operators $o \in \{RX_i, RC_j\}$:

$$w(o) \leftarrow (1 - \alpha)w(o) + \alpha r \tag{4.1}$$

α is a parameter called *learning rate*, which determines how strongly the weights are influenced by the most recent reward(s).

The portfolio of operators in SA-LNS of [LG07] shares several similarities with the pool of low-level heuristics provided by HyFlex problem domains. We therefore adapted and extended the algorithm to make it suitable as a selection hyper-heuristic for cross-domain optimization. The work described in this section was published in a technical report [MM21a].

4.2.1 SA-LNS for cross-domain optimization

To adapt SA-LNS for the HyFlex framework, we defined all low-level heuristics of type *mutation* or *ruin-and-recreate* as relaxation operators and all heuristics of type *local search* as reconstruction operators. Heuristics of type *crossover* were not used, as they do not fit into the single-point scheme of SA-LNS.

HyFlex also supports two different search parameters which can be set to influence the behavior of some LLHs: The *intensity of mutation* (IoM) parameter is mostly used by mutation operators (and some ruin-and-recreate operators) to determine how much of a solution is affected. The *depth of search* (DoS) parameter affects the number of steps taken by local search operators. Both parameters have a domain of $[0 - 1]$. LLHs may use either the IoM or the DoS parameter, though some don't use them at all.

The original SA-LNS deals with search parameters by maintaining a separate list of weights for the possible values of each search parameter per operator. Whenever an operator is selected, values for its search parameters are selected according to their current weights. At the end of each iteration, the selected search parameter values have their weight updated using the same formula as the operator weights themselves. This way, the algorithm learns a separate probability distribution over parameter values for each operator that uses them. This approach requires search parameters to have finite and discrete values. To be able to use it for the IoM and DoS parameters in HyFlex, we split the range of possible parameter values ($[0 - 1]$) into n uniformly spaced discrete values (for this thesis, we used $n = 11$ to obtain increments of size 0.1).

Learning bigram statistics with SA-LNS

The algorithm described in the previous sections chooses relaxation and reconstruction operators for each step independently from each other. However, it is likely that certain operator pairs yield particularly good results or that some relaxation operators work better when followed by some reconstruction operators and worse with others. To be able to detect and learn these kinds of relationships between operators, we extended the operator selection of the previous section as follows:

The selection of the relaxation operator remains unchanged. However, the chosen operator is passed as an additional input to the selection algorithm of the reconstruction operator. The selection algorithm maintains a separate set of reconstruction operator weights for each possible relaxation operator. Selection and updates of weights then work analogously to before, except that only the set of weights matching the given relaxation operator is used. This is similar to the treatment of search parameter values in the original algorithm.

Since this algorithm variant learns bigram probabilities for operator pairs, we call it Bigram SA-LNS (BSA-LNS).

4.2.2 SA-LNS with time windows

While rewards are scaled by the time taken by an operator, the learning rate is not. Thomas and Schaus found in 2018 [TS18] that this may lead to unbalanced weights in cases where the different operators have very unequal running times. In their paper, they investigated the use of LNS as a Constraint Programming (CP) solver. They mainly follow the SA-LNS structure of [LG07] as shown in Figure 4.2, but they use a modified weight function:

The evaluation distinguishes between *local* and *global performance*. Local performance is evaluated over a time window $W = [t^* - s, now]$, where t^* is the time at which the currently best known solution was found and s is a fixed solver parameter. This ensures that the local time window always includes at least one improvement.

Let Δc_i and Δt_i be the improvement achieved and time taken, respectively by operator o_i selected in iteration i of SA-LNS (weights are computed independently for relaxation and reconstruction operators). In addition, t_i denotes the point in time when iteration i was executed. Then Thomas and Schaus define the local ($L(o)$) and total ($T(o)$) performance of an operator o, as well as the overall local (L) and global (T) performance of all operators, as follows:

$$L(o) = \frac{\sum_{i|o_i=o \wedge t_i \in W} \Delta c_i}{\sum_{i|o_i=o \wedge t_i \in W} \Delta t_i} \qquad T(o) = \frac{\sum_{i|o_i=o} \Delta c_i}{\sum_{i|o_i=o} \Delta t_i} \tag{4.2}$$

$$L = \frac{\sum_{i|t_i \in W} \Delta c_i}{\sum_{i|t_i \in W} \Delta t_i} \qquad T = \frac{\sum_i \Delta c_i}{\sum_i \Delta t_i} \tag{4.3}$$

$L(o)$ and $T(o)$ are defined as 0 for operators that have not been selected at all in the relevant time.

The weight of operator o is then calculated as

$$w(o) = (1 - \lambda) * L(o) + \lambda \frac{L}{T} T(o) \tag{4.4}$$

Via the choice of parameter λ ($0 \leq \lambda \leq 1$), the focus can be shifted between local and global performance. Since the efficiency of operators depends on the current solution and is expected to change over the course of the search, local performance is typically more interesting, as it is most likely to match the current "true" performance of an operator. Still, the addition of global performance helps to smooth out the operator weights in case of stochastic fluctuations.

Using reasonable caching of values, the weights can be efficiently updated after each operator application. We also implemented this version of SA-LNS as a HyFlex hyper-heuristic and denote it as Time Window SA-LNS (TWSA-LNS).

Here it should be noted that the algorithm as stated in [TS18] cannot be directly used from the start, as all weights are initially 0. Worse, once the first improvement to the initial solution has been found, there will be exactly one operator[1] with a positive weight, namely the one that led to the improvement. From then on, this operator will be selected at every single iteration and since its total weight can never become 0 again, no other operator will ever be considered.

We solved this problem by starting the algorithm with a warmup phase where initially all operators can be selected with uniform probability. The warmup phase ends once at least γ iterations have found an improvement. This ensures that each operator is applied multiple times and performance estimates computed after the warmup phase are more robust. Similarly, the choice of search parameter values for each heuristic is initially uniformly random, but switches to a weight-based selection once γ successes have been achieved with the heuristic in question.

A warmup phase of fixed length had worse performance in our experiments than the one based on improvements found, so it was not considered for the final evaluations.

Bigram TWSA-LNS

As we did for SA-LNS, we also extended TWSA-LNS to a version learning Bigram probabilities (BTWSA-LNS).

Weights for relaxation operators are calculated according to equation 4.4. However, weights for the reconstruction operators now depend on the relaxation operator selected before. This means that we now need to distinguish between the relaxation operator x_i and the reconstruction operator r_i selected at iteration i. Equations 4.2 and 4.3 for reconstruction operators then look as follows (for each relaxation operator x and reconstruction operator r):

[1]Or rather, one relaxation and one reconstruction operator, as operators are always applied in pairs

$$L^x(r) = \frac{\sum_{i|x_i=x \wedge r_i=r \wedge t_i \in W} \Delta c_i}{\sum_{i|x_i=x \wedge r_i=r \wedge t_i \in W} \Delta t_i} \qquad T^x(r) = \frac{\sum_{i|x_i=x \wedge r_i=r} \Delta c_i}{\sum_{i|x_i=x \wedge r_i=r} \Delta t_i} \tag{4.5}$$

$$L^x = \frac{\sum_{i|x_i=x \wedge t_i \in W} \Delta c_i}{\sum_{i|x_i=x \wedge t_i \in W} \Delta t_i} \qquad T^x = \frac{\sum_{i|x_i=x} \Delta c_i}{\sum_{i|x_i=x} \Delta t_i} \tag{4.6}$$

In other words, a separate set of construction operator weights is maintained for each relaxation operator.

It may now happen that for a selected relaxation operator x the local performance $L^x(r) = 0$ for all reconstruction operators r (if no improvement was found with that relaxation operator in the local time window). In this case, we again select uniformly among all operators.

An initial warmup phase is necessary also for BTWSA-LNS. The warmup phase for the selection of the relaxation operator works as described for TWSA-LNS. Since we have separate sets of weights for reconstruction operators depending on the relaxation operator chosen first, we keep track of the number of improvements found with each relaxation operator. We end the warmup phase for only this operator once that number reaches γ, similar to the treatment of parameter values in the previous section. In other words, the warmup phase for the reconstruction operator selection following relaxation operator x runs until at least γ iterations found an improvement by applying x followed by any reconstruction operator.

This way, successful relaxation operators end their warmup phase early, while unsuccessful or seldom used relaxation operators continue selecting reconstruction operators uniformly until they have gathered enough data to allow robust weight estimations.

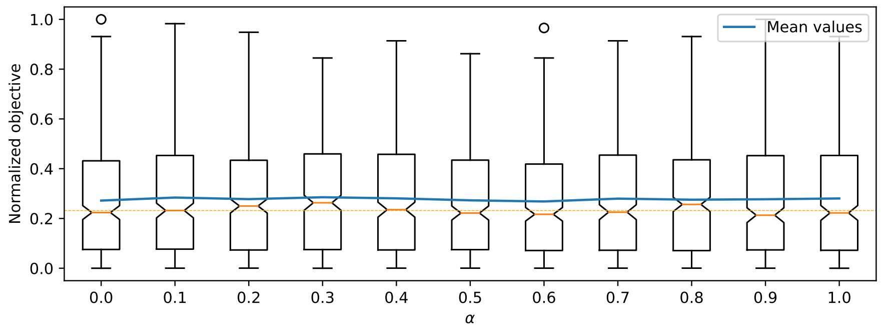
4.2.3 Empirical evaluation

We implemented our approaches as hyper-heuristics in the HyFlex framework and evaluated them on the provided benchmark instances.

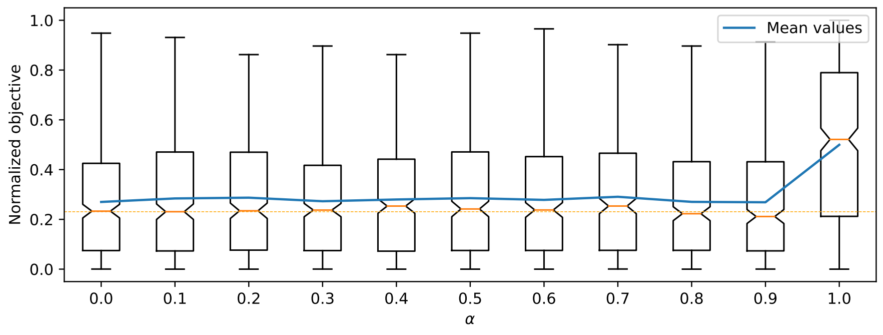
Our experiments were performed each using a single core on a computing cluster with 10 identical nodes of 24 cores, an Intel(R) Xeon(R) CPU E5–2650 v4 @ 2.20GHz and 252 GB RAM. On this setup, the benchmarking tool provided for the CHeSC 2011 [BGH+11] allotted a time of 392s per run (the equivalent of 600s on the competition hardware).

Parameter tuning

The variants of SA-LNS described in the previous section have a few parameters that influence their behavior: SA-LNS and BSA-LNS use the learning rate parameter α, while TWSA-LNS and BTWSA-LNS have λ to shift the focus between local and global performance and s, the length of the local time window beyond the time of the best known solution. In addition, the latter variants also start with the warmup phase, which



(a) SA-LNS



(b) BSA-LNS

Figure 4.3: Impact of different values for the learning rate parameter α on the normalized performance of SA-LNS (a) and BSA-LNS (b). The dotted line shows the median for the configuration with $\alpha = 0.1$ for comparison.

is terminated after at least γ successful iterations (for BTWSA-LNS, the warmup phase is terminated independently for each operator once it reaches the required number of successes).

We analyzed the effects of these parameters on the 10 early instances for the four problem domains known in advance of the CHeSC (MaxSAT, BP, FS, and PS)[2], performing 10 runs per instance and configuration. To make objective values of different problem domains comparable, we normalized them to the range $[0 - 1]$ as follows [DGU12]:

$$o_{norm} = \frac{o - o_{min}}{o_{max} - o_{min}} \tag{4.7}$$

o_{max} and o_{min} are the worst and the best objective value achieved for each instance, respectively.

As it turns out, the performance of the general SA-LNS framework is quite robust towards changes to the parameter values, with most configurations yielding similar results. An

[2]This is the same instance set also available to the competition participants.

example of this can be seen in Figure 4.3, which shows that the solution quality produced with different learning rates is nearly the same for most values. Of particular interest here are the extreme values for α: A value of $\alpha = 0.0$ means that no learning takes place, reducing the operator selection to uniform random choice. Interestingly, this does not affect the quality of results, indicating that learning effects are neither necessary nor beneficial for these variants of SA-LNS. On the other end of the value range, at least BSA-LNS shows significantly worse performance for $\alpha = 1.0$. In this configuration, only the improvement of the most recent application of an operator is used to determine its weight. We conjecture that the main reason for the performance impact is that most iterations are unsuccessful. This sets the used operators' weight to 0 so they can't be selected again until all other weights are 0 too.

For the algorithms using time windows, we defined a baseline configuration and varied the value of each parameter individually. As the baseline, we used a time window size of $s = 10s$, $\lambda = 0.1$, and $\gamma = 10$ successes, following recommendations in [TS18] and results of preliminary experiments.

Results for different values of λ for (B)TWSA-LNS, shifting the focus from local weights at small values to global weights at large values, are shown in Figure 4.4. While the impact is rather small, there is still a visible trend towards a focus on local performance for both variants. Even ignoring global performance completely with $\lambda = 0$ does not result in adverse effects on the solution quality.

No such trend is observable for the window size parameter s (Figure 4.5), with only a slight increase in the normalized objective value for very long windows.

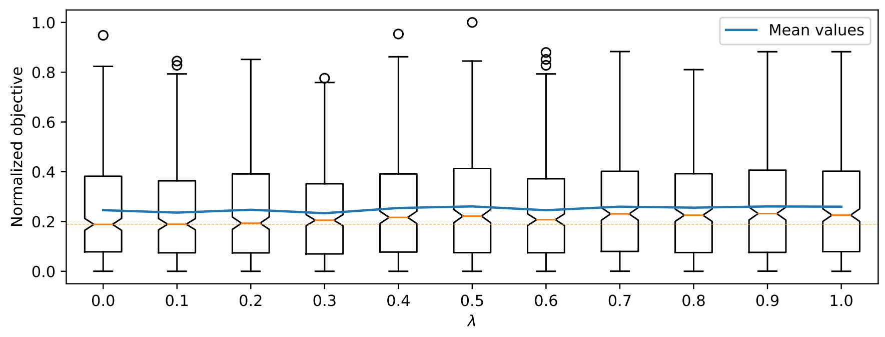
For the length of the warmup period at the beginning of the search (Figure 4.6), the best results where achieved with a value of $\gamma = 10$. Both more and fewer successes than that required for an operator before the warmup period ends for it result in a higher normalized objective score. The intuition behind this result is that too short warmup periods mean that some heuristics may not have a chance to find improvements, while too long warmup periods mean that time is wasted on unpromising heuristics.

Overall, there was no configuration with significantly better performance than the baseline, so we used this configuration for the final evaluation.

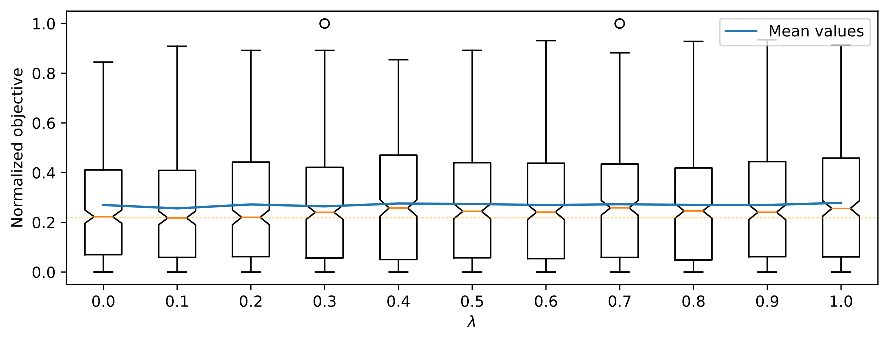
Experimental results

We then evaluated our algorithms on the test set of 30 instances used for the CHeSC 2011, 5 instances for each of the 6 problem domains. As was done for the competition, we used median results over 31 runs per instance to compute our scores. Ranks were computed by pitting each evaluated algorithm individually against the 20 competition participants, using the CHeSC 2011 scoring scheme (points are awarded for each instance to the top 8 results, similar to Formula 1 scores).

Table 4.2 shows the rankings and scores that the discussed variants of SA-LNS would have achieved in the competition.



(a) TWSA-LNS

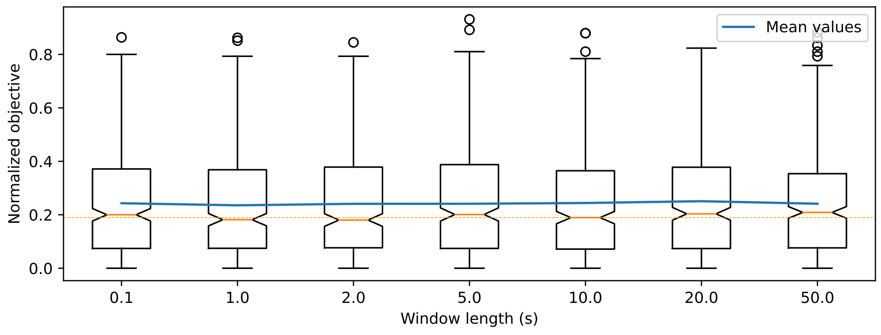


(b) BTWSA-LNS

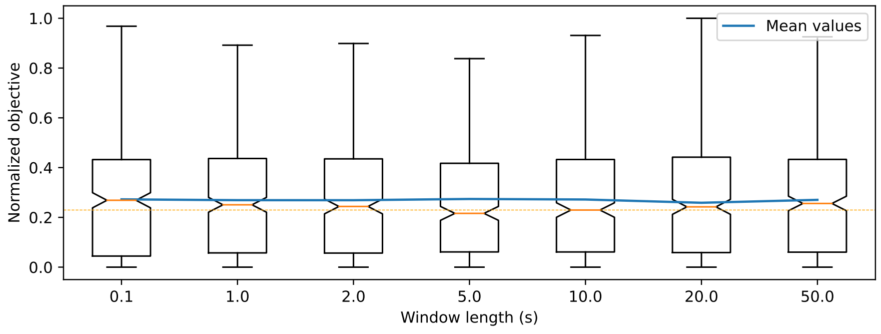
Figure 4.4: Impact of λ on the normalized performance of TWSA-LNS (a) and BTWSA-LNS (b). The dotted line shows the median for the configuration with $\lambda = 0.1$ for comparison.

Method	**Rank**	**Score**	**SAT**	**BP**	**PS**	**FS**	**TSP**	**VRP**
SA-LNS	8	67.00	0.00	22.00	22.50	4.00	15.50	3.00
BSA-LNS	10	60.50	0.00	22.00	16.00	4.00	12.50	6.00
TWSA-LNS	4	97.00	0.00	24.00	16.00	24.00	32.00	1.00
BTWSA-LNS	4	97.50	0.00	25.00	18.00	24.00	28.50	2.00

Table 4.2: Experimental results for the variants of SA-LNS. Shown are the rank that a method would have placed at if it had participated in the CHeSC 2011 together with the 20 original competitors, as well as the score it would have achieved. The remaining columns show the scores achieved for each of the 6 domains.



(a) TWSA-LNS



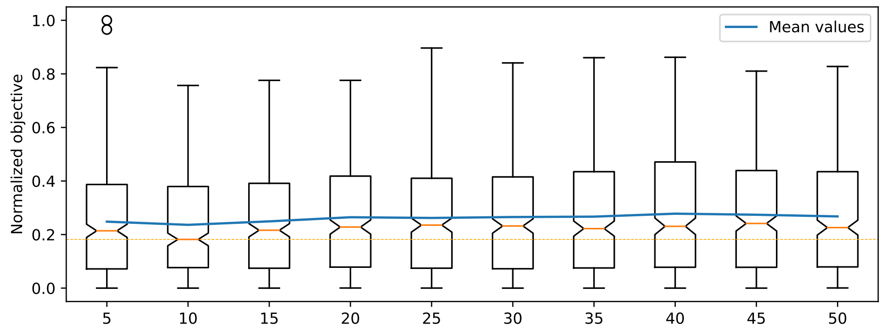
(b) BTWSA-LNS

Figure 4.5: Impact of the parameter s determining the local time window on the normalized performance of TWSA-LNS (a) and BTWSA-LNS (b). The dotted line shows the median for the configuration with $s = 10s$ for comparison.

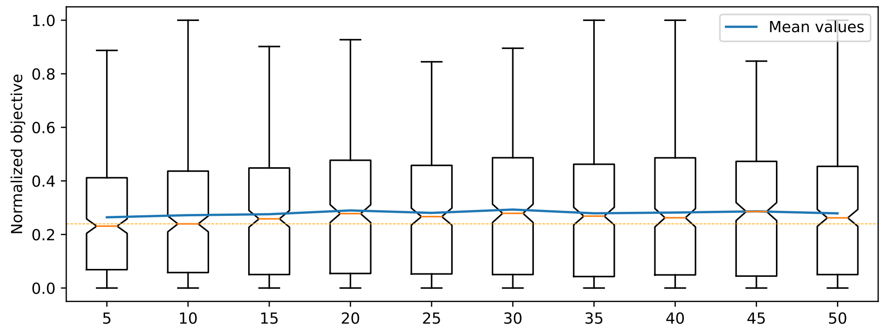
All methods based on SA-LNS perform reasonably well, but the modified weight calculation with time windows leads to a substantial improvement in score and rank. Learning bigram statistics, where weight for reconstruction operators depend on the relaxation operator selected before, did not help in improving the scores further. This may be an indicator that there are only few operator pairs that work particularly well together, but not when paired with other operators. Alternatively, the increased number of weights may be too much to efficiently learn during the search, despite the warmup phase ensuring that there are multiple data points available for each set of weights.

Amplification of low-level heuristics

A technique successfully used in [Chu20] is the usage of *amplified low-level heuristics*. The idea behind it is that many perturbation heuristics only affect small parts of a solution, making it unlikely that a follow-up repair heuristic can profit from a single application of the perturbation. To counteract this, we can repeat the application of a low-level



(a) TWSA-LNS



(b) BTWSA-LNS

Figure 4.6: Impact of the number of successes γ required for completing the warmup period on the normalized performance of TWSA-LNS (a) and BTWSA-LNS (b). The dotted line shows the median for the configuration with $\gamma = 10$ for comparison.

heuristic multiple times, up to a fixed time limit [3]. If any application yields a worse solution it is discarded, otherwise the current solution is replaced with the new solution. By allowing our hyper-heuristics to also select and apply the amplified version of each low-level heuristic, we effectively double the size of our heuristic portfolio, without having to modify the problem domain itself.

On our training set, using amplified low-level heuristics consistently improved the performance of all four variants of SA-LNS. However, this is not the case on the test set, where the performance is evaluated according to the CHeSC 2011 scoring rules (shown in Table 4.3). Here, the use of amplified low-level heuristic is even detrimental to the scores, particularly for BTWSA-LNS.

This discrepancy can be explained by taking a closer look at the results for the different problem domains in the training set (shown in Figure 4.7). Clearly, the MaxSAT problem domain profited significantly from amplified heuristics, while the effect on the other problem domains is minor to nonexistent. However, this improvement for MaxSAT was

[3]We use a limit of 10ms, as was used in [Chu20]

Method	Rank	Score	SAT	BP	PS	FS	TSP	VRP
SA-LNS	8	67.00	0.00	22.00	22.50	4.00	15.50	3.00
(amplified)	10	56.60	0.00	21.00	8.50	7.00	16.00	4.00
BSA-LNS	10	60.50	0.00	22.00	16.00	4.00	12.50	6.00
(amplified)	11	47.50	0.00	21.00	5.50	5.00	14.00	2.00
TWSA-LNS	4	97.00	0.00	24.00	16.00	24.00	32.00	1.00
(amplified)	5	88.50	0.00	23.00	11.00	23.00	29.50	2.00
BTWSA-LNS	4	97.50	0.00	25.00	18.00	24.00	28.50	2.00
(amplified)	11	40.00	0.00	25.00	3.00	1.00	8.00	3.00

Table 4.3: Comparison of SA-LNS variants with and without using amplified low-level heuristics. Shown are the rank and both total and per-domain scores achieved if the hyper-heuristic had participated in the CHeSC 2011 against the 20 original participants.

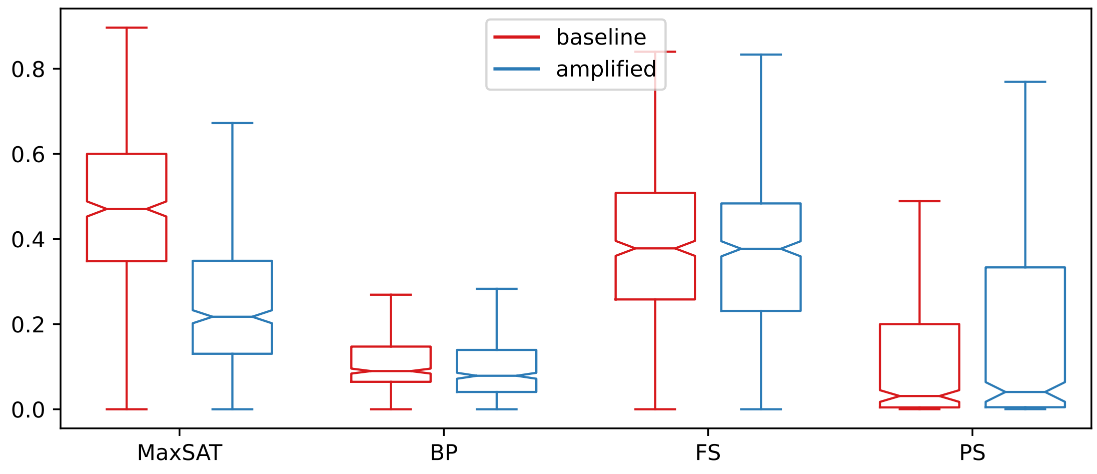


Figure 4.7: Comparison of SA-LNS results for each problem domain with and without amplified low-level heuristics. The plots contain combined results for all four variants of SA-LNS.

apparently still not sufficient for scoring any points on that domain compared to the other participants (compare Table 4.3). The conclusion is still that amplified low-level heuristics are beneficial, however the benefit is simply not reflected by the scores under the relative scoring system used by the CHeSC 2011.

Comparison with newer approaches

We also compared our results with the best scoring configuration, BTWSA-LNS (without amplification), with those of more recent hyper-heuristics. To do so, we simulated another

Rank	Method	Reference	Score	SAT	BP	PS	FS	TSP	VRP
1	TS-ILS	[AOO21]	185.75	33.50	34.00	32.50	31.50	35.25	19.00
2	GEP-HH	[SAKQ15]	130.25	22.00	24.00	0.00	50.00	13.25	21.00
3	FS-ILS	[ABN14]	115.33	33.50	4.00	2.00	27.83	26.00	22.00
4	MCTS-HH	[SK15]	110.08	14.83	22.00	30.00	8.00	17.25	18.00
5	AdapHH	[MVDCVB12]	86.50	12.50	33.00	2.50	15.00	18.50	5.00
6	QHH	[CWL18]	70.75	24.50	1.00	0.00	17.00	22.25	6.00
7	ML	[Lar11]	57.33	1.50	0.00	25.00	14.83	2.00	14.00
8	VNS-TW	[HCF12]	54.50	16.50	0.00	26.50	9.00	2.50	0.00
9	**BTWSA-LNS**		45.50	0.00	14.00	8.50	8.00	15.00	0.00
10	PHunter	[CXIC12]	41.50	1.50	0.00	8.50	0.00	10.50	21.00
11	FRAMAB	[FGTRP15]	39.50	1.50	0.00	16.00	0.00	0.00	22.00
13	FI-Pruning	[Chu20]	34.33	2.33	20.00	0.00	5.00	1.00	6.00

Table 4.4: Selection of results of the extended competition including BTWSA-LNS and other recent hyper-heuristics. Shown are both the total score achieved and the scores on each of the six problem domains.

competition where we added the following hyper-heuristics to the 20 original participants and our own approach: FRAMAB [FGTRP15], GEP-HH [SAKQ15], MCTS-HH [SK15], QHH [CWL18], and TS-ILS [AOO21]. The required median results for these hyper-heuristics on the competition instances were taken from their respective papers. The authors of FS-ILS [ABN14], TeBHA-HH [AÖ15], and the variants of FI [Chu20] only reported relative scores compared to the CHeSC participants, which is not sufficient information to compute their scores with respect to a different set of hyper-heuristics. However, the source code of FS-ILS is available online, which allowed us to rerun the necessary experiments and obtain their results this way. In addition, we were able to re-implement the FI variants following their description in the paper and personal communication with the authors. In our re-implementation, FI-Pruning was the variant that produced the best results (in contrast to the results reported in [Chu20], where FI-Bigram achieved the highest scores), so we used this variant for our comparison. Unfortunately this was not an option for TeBHA-HH, so we could not include this hyper-heuristic in the comparison. Table 4.4 shows the results of the top 10 approaches in this extended comparison.

While BTWSA-LNS was able to outperform several recent hyper-heuristics, including FRAMAB and FI-Pruning, others achieved higher scores, and in particular several hyper-heuristics also using the general ILS structure of perturbation followed by local search, such as TS-ILS, FS-ILS, and QHH.

One aspect that these other hyper-heuristics all have in common is that they allow more than one low-level heuristic application during each local search step, and partially also for perturbation steps. It may be that the limitation to a single low-level heuristic application

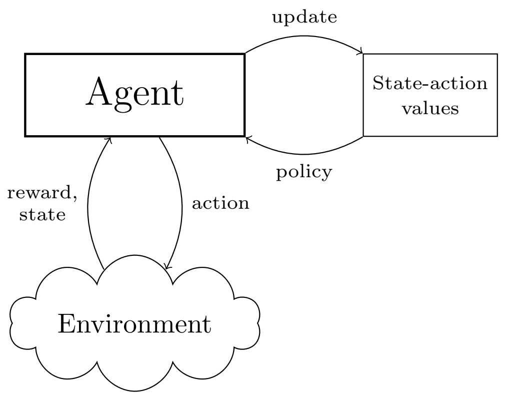


Figure 4.8: Agent-environment architecture often used in reinforcement learning settings. Figure inspired by [DGU12].

in each step of the SA-LNS variants is too strict to reliably find improved solutions. Further research is necessary to determine whether the solutions can be improved by allowing multiple operations per step.

4.3 Reinforcement learning

Reinforcement learning is a popular class of techniques that fall in the category of unsupervised learning. Instead of learning from pre-labeled training data, a reinforcement learning agent learns via interacting with its environment and observing the effects.

For our purposes, we will use the following high-level model (see Figure 4.8): The agent interacts with the environment by iteratively executing one of a discrete set of *actions*. The environment then provides the agent with feedback in the form of the new *state* arising as the result of applying the chosen action and a (real-valued) *reward* achieved due to it. An action-selection *policy* is used by the agent to select the next action, based on its belief about the expected value of each action in the current state (*state-action values*), i.e. its estimate about the expected total reward gained from executing the action (in the long run, also including rewards of future actions). The feedback from the environment is used to update the state-action value estimates to bring them closer to the true values. The overall goal of the agent is to maximize the total reward gained, which is reflected in the policies and value update rules. In many settings, the agent learns the state-action values across multiple *episodes*, which eventually end at some terminal state.

This model can be naturally translated to the setting of cross-domain optimization with HyFlex. The reinforcement learning agent is the hyper-heuristic, which interacts with the environment, the combination of problem domain, instance, and current solution, by selecting a low-level heuristic and applying it to the current solution. This way, we can learn the expected performance characteristics of the different low-level heuristics while improving the solution at the same time.

One option would be to keep doing this until the end of the available time and report the best solution found during the search. We might expect that as the hyper-heuristic learns which low-level heuristics perform well, the solution would keep on improving. This was the approach described in 2012 by Di Gaspero and Urli [DGU12], who participated in the CHeSC 2011. However, as the authors also noted in the discussion of their algorithm, this strategy has a drawback: The stochastic nature of many low-level heuristics, most of all mutations, means that many steps will lead to potentially big increases in the objective function, in particular at the start of the search when the state-action values are still not more than rough guesses. In some cases, these worsening moves will be compensated by later heuristic applications, and may even be necessary to diversify and reach new areas of the search space. In other cases, they will destroy good parts of the solution and get trapped such that it takes a long time to get back to the previous solution quality. If that happens, it would be beneficial if we were able to instead reset the current solution back to a previous state, such as the best known solution found so far.

We can model this behavior in our hyper-heuristic splitting the whole search into several episodes, each consisting of a chain of heuristic applications and their corresponding solutions. Each episode starts at the currently best known solution. If at any point the result of a heuristic application is better than that solution, the episode ends, the best known solution is replaced and a new episode starts from the updated solution. If no such improvement can be found for a certain amount of steps, we also end the episode, but reject the current solution and restart the next episode again from the best known solution so far. An example of this process is shown in Figure 4.9.

Of course, the choice of the length of each episode (which does not have to be constant over the course of the search) then becomes a crucial component of our hyper-heuristic. Too long, and we spend a lot of time on solutions that will be rejected later anyway. Too short, and we will not be able to escape local minima, as solutions will be rejected if they do not immediately lead to improvements.

4.3.1 Components of reinforcement learning hyper-heuristics

A complete reinforcement learning hyper-heuristic for HyFlex has several components that need to be described. We need to determine how the internal state of the environment is represented in the hyper-heuristic, a policy to select actions based on the current state-action values, a function that translates the results of a heuristic application into an appropriate reward, and an update mechanism that adjusts the state-action values based on observations. Finally, we need to determine the number of non-improving steps before we end the episode and reset the current solution to the best known solution so far.

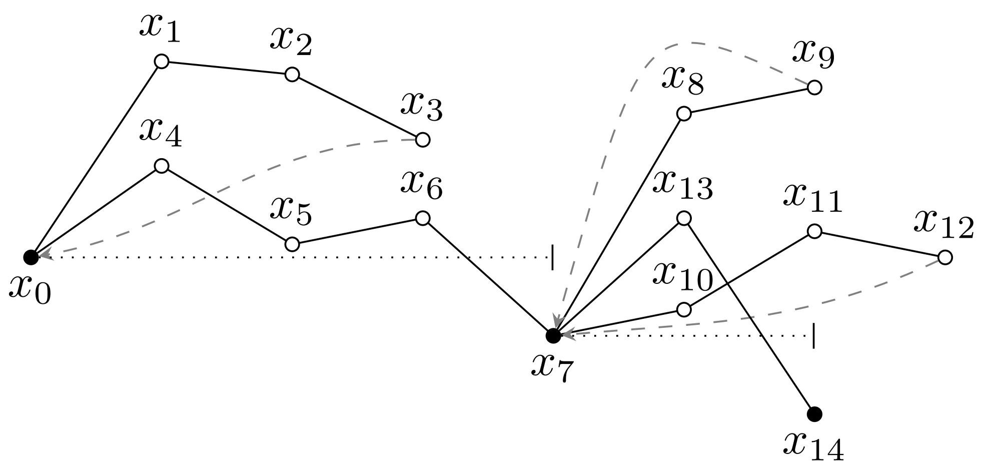


Figure 4.9: Example for episodic solution chains. The vertical position of a solution indicates its objective value (lower is better). Starting from the currently best known solution (x_0), heuristic applications produce chains of new solutions, until either (x_7, x_{14}) a solution is better than the previous best, or (x_3, x_9, x_{12}) the decision is made to abandon the chain and to reset to the best known solution. Each chain is treated as one episode of reinforcement learning.

State representation

Finding a good representation of problem states is an important step for any reinforcement learning approach. On the one hand, the granularity of the states must be fine enough to capture as much relevant information as possible. In fact, they should be Markov, i.e. they should contain enough information that the best action to take can be determined wholly from the current state [SB18]. On the other hand, learning is performed per state[4] and a huge number of states means that many states will be visited only a few times, if at all, which makes learning their action value functions that much harder.

In the context of combinatorial optimization, a natural approach for such a state representation is to use features of the current (and past) solutions. However, the hard domain barrier of HyFlex makes it difficult to extract meaningful information about those. Instead, we can use information about the applied heuristics and their effect on the objective function to build a collection of features that should together form our state representation.

In our analysis, we chose to employ a modular approach that allows us to evaluate different feature combinations. Formally, given a set of *features* $\mathcal{F}$, a *state representation* $S = (F_1, F_2, \ldots, F_n),\ F_i \in \mathcal{F}$ is an ordered subset of those features. A single state $s = (f_1, f_2, \ldots, f_n) \in S$ is then defined as a tuple of all individual feature values.

[4]There are approaches to use reinforcement learning over large, or even continuous state spaces, but for the purpose of this thesis, we will assume that states are discrete and finite.

For this thesis, we investigated the following features:

Last heuristic The index of the last low-level heuristic applied. If the last action was to reset the solution, this takes the empty value ϵ, as the last heuristic cannot possibly have influenced the current solution. The purpose of this feature is to capture and exploit synergies between specific low-level heuristics.

Type of last heuristic The type of the last low-level heuristic applied. Similar to the above, but with the assumption that such synergies are mostly the same across heuristics of the same type.

Tail length If the number of remaining heuristic applications before the solution is reset (the *tail length*) is known in advance, we can use this knowledge to guide our decisions. For example, if the solution will be reset after the next heuristic application (unless it results in an improvement), then it is probably not a good idea to apply a mutation heuristic, as the effect is likely to be negative and thus lost immediately after the reset.

Since the tail length is potentially unbounded, depending on the reset scheme chosen, it is reasonable to limit the domain size of this feature (and thus the size of the state space) by a maximum tail size l^{max}. All tail lengths greater than l^{max} will be treated as l^{max} for the purpose of this feature.

Resets since last improvement Long and medium term performance can be captured by this feature, which considers the number of resets since the last time an improvement was found. Intuitively, a long time without improvements may indicate that the focus has to be shifted towards exploration, via increased applications of mutations and longer chains of heuristic applications between resets. To keep down the domain size of this feature, it is evaluated at discrete indicator values b_i. The result is a boolean array a, where $a[i]$ indicates whether the solution was reset at least b_i times since the last improvement. For the purpose of this thesis, we used a set of three indicator values of 10, 30, and 100 resets.

Time since last improvement As an alternative to the number of resets, we can also consider the time since the last improvement was found. As before, we use indicator values to obtain a boolean array telling us whether an improvement was found within that time. For this feature, we used indicator values of 100ms, 1s, and 10s.

Di Gaspero and Urli [DGU12] used a single-component state representation that captured a measure of reward trend. Unfortunately, the exact formula used to calculate this measure is not provided in their paper, so we could not include it here for comparison.

Action selection policy

The policy determines which action should be selected next at each step. If we somehow knew the true state-action values at each point in the search, the optimal policy would always greedily select the action with the highest value for the current state.

Instead we need to learn those values starting from our initial estimates, which makes it necessary to select suboptimal actions at least sometimes to gain observations about their performance. Indeed, to guarantee convergence towards the true state-action values, each state-action pair has to be selected infinitely many times [SB18].

In fact, if our only goal were to learn the state-action values, we would select uniformly random actions, to ensure that we get a balanced amount of observations for each action. As we are trying to maximize the rewards obtained within a finite amount of time at the same time, we need to select more promising actions (i.e. those with higher state-action value estimates) more often, while still allowing for some variation to improve our estimates.

Different policies can capture this behavior.

ε-greedy policies are a variation on the greedy policy that select the best action (according to the current estimates) most of the time, but with probability ε instead choose uniformly among all possible actions. Such an ε-greedy policy was used e.g. in [DGU12]. They have the advantage that they are independent of the domain and distribution of rewards, which may vary between problem domains and instances depending on the reward function chosen (see below). The choice of ε determines how much of an emphasis is placed on diversification of the selected actions.

Decreasing ε-greedy policies follow the same general principle, except that ε linearly decreases over time to reach 0 at the end of the available time. This captures the fact that the quality of our estimates increases as we gain more observations, allowing us to be more confident that the action with the highest value is a good choice.

Softmax is a policy where the probability of choosing an action $i \in A$ with weight w_i is equal to $\frac{w_i}{\sum_{j \in A} w_j}$, where A is the set of all actions. This means that actions with higher value are more likely to be chosen.participants A necessary condition for this policy is that all state-action values are non-negative, which may make it unsuitable for some reward functions. In addition, values of 0 are also problematic, as the associated action would never be selected again, making it impossible to later update the estimate with new observations. For this reason, the softmax policy requires positive initial state-action values, e.g. $\frac{1}{|A|}$.

We note that the softmax policy is described in [SB18] using exponential weights. However, this would make the behavior of the policy highly dependent on the reward value range, which is why we decided to use a variant with linear weights instead.

ε-softmax To counteract the problematic behavior of the softmax policy with regards to state-action values of 0, we can also add an ε-variant instead. As with ε-greedy, ε-softmax selects uniformly among all actions with probability ε and follows the softmax policy otherwise. This decreases the influence of state-action values on the weights, but allows us to easily deal with values of 0.

Reward function

The final goal of the hyper-heuristic is to find the best possible solution within the available time. To that end, we want to reward heuristics that reliably lead to improvements of the current solution as often and fast as possible.

However, we need to consider that the hyper-heuristic has to operate across different domains, where the achieved objective values vary a lot over several orders of magnitude between domains and even between instances of the same problem domain.

The simplest way to deal with this is to award a constant positive reward (e.g. 1) whenever a heuristic application results in an improved solution, and 0 otherwise. This is the approach used in [Chu20].

The disadvantage of this approach is that it considers neither the magnitude of improvement nor the time taken to achieve it. If one heuristic finds huge gains in a short amount of time, we should prefer it over another heuristic that spends a long time to find a minor improvement, even if the second heuristic is slightly more likely to do so.

Whether the achieved difference in the objective value can be directly used as the reward or whether it needs to be normalized depends on the action selection policy. The observed rewards are used to compute the state-action values, which in turn are the basis for the policy decisions. Some policies (e.g. the exponential version of (ϵ-)greedy described in [SB18]) are sensitive to the magnitude of state-action values and require normalized reward values that have approximately the same magnitude and distribution across different domains to be generally applicable. Others, including all policies investigated in this thesis, do not depend on the reward value range as long as it does not change too quickly during a single run.

Another question is what reward to give to heuristics that do not result in improvements of the objective function. Handing out negative rewards for worsening actions would unfairly penalize mutations and ruin-and-recreate heuristics that may still prepare a solution for later improvements (e.g. by moving out of local minima). We can consider awarding negative rewards for heuristics just before a reset of the solution, to discourage their use for the future, but we would need to somehow balance those penalties with the frequency and magnitude of improvements to avoid action-values that diverge towards negative infinity. A simpler solution is to just give rewards of 0 for non-improvements. Intuitively, this also makes sense, as sequences of heuristic applications that do not result in improvements will eventually lead to a reset, with a final objective value difference of 0.

When it comes to factoring in the time taken by the heuristics, there are also several possibilities. Intuitively, the "best" heuristic is the one that produces the most reduction in objective value per time unit. Defining the obtained reward r as $r = \frac{\Delta o}{\Delta t}$ matches this goal. To obtain the time difference Δt, we can factor in either only the time taken by the heuristic itself, or the total time taken by all heuristics since the last reset. The latter option captures the fact that also preceding heuristics have contributed to the overall result, and that heuristics which find an improvement quickly without needing a long setup via other heuristics are to be preferred.

Overall, we investigated the following reward functions:

Constant (1) Each time an improvement is found, the reward is 1, regardless of objective value.

Delta ($\sum \Delta o$) The reward is the difference in objective value compared to the previous best known solution.

Delta per self time ($\frac{\sum \Delta o}{\Delta t}$) Here, the reward is the objective value difference divided by the time taken by the heuristic.

Delta per total time ($\frac{\sum \Delta o}{\sum \Delta t}$) As above, but the whole time taken by all heuristic applications since the last reset is used for the denominator.

In all cases, if the objective value does not improve due to a heuristic application, the reward is 0.

Learning rate

Many reinforcement algorithms perform incremental updates of the state-action values at certain points (typically after each action or after each episode).

Usually, this update takes the following form [SB18]:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(T - Q(s,a)) \tag{4.8}$$

T is a target number representing the observed value of the state-action pair; its definition varies between different RL methods, but usually includes the observed reward.

The difference between this target number and the old state-action value $(T - Q(s,a))$ is an error in the estimate, which is reduced by updating it towards the observed target. The magnitude of the update depends on the parameter α, called *step size* or *learning rate*.

Theoretical convergence results on stationary problems, i.e. ones where the environment and thus the "true" state-action values do not change, require that α should decrease with time and tend towards 0[5]. This is to ensure that good estimates are not derailed by recent (noisy) observations.

[5]The formal conditions on α can be found in [SB18], p. 33

However, such learning rates often converge very slowly in practice. Theoretical convergence guarantees at infinity are not helpful when the time available for the algorithm to learn is limited. Besides, they are unsuitable for non-stationary problems, where more weight should be put on recent observations as they are more likely to be representative of the current state of the environment. For these reasons, learning rates that do not satisfy these conditions are often used, such as constant values for α.

The problem we are dealing with is certainly non-stationary, as the likelihood of operators to lead to an improvement depends on properties of the current solution which change with time and to which we do not have access so that we might include them as part of the state representation. For example, local search heuristics are guaranteed not to find any improvement for solutions that are local optima for the heuristics, while this is not the case for any other solution.

We have investigated different learning rates for our algorithms. Besides constant values for α (or α^c), we have implemented two time-varying learning rates that start of at α^{max} and decrease over time, called *linear* (α^l_t) and *exponential decline* (α^e_t). In the following, let t be the elapsed time (in seconds) and t^{max} denote the maximum time available to the algorithm.

$$\alpha^l_t = \alpha^{max}\frac{t^{max} - t}{t^{max}} \tag{4.9}$$

$$\alpha^e_t = \alpha^{max}0.99^t \tag{4.10}$$

participants It should be noted that α^l_t becomes 0 at t^{max} while α^e_t also tends towards 0 but stays positive.

State-action value update rule

At any point in time the state-action values represent our current estimates about the expected reward to be gained from executing a particular action in that state. This includes not only the reward immediately awarded to the most recent action, but also rewards due to future actions on the resulting new state and its successors. Obviously, this depends on the action selection policy, as it is the policy that determines the future actions.

Initially, we do not have any knowledge about the low-level heuristics' performance. As time goes on, we can use our observations of actions taken and rewards achieved to update our estimates. Eventually (given enough time), we would like our estimates to converge towards the true values for the expected rewards. There are several methods we can use to achieve that goal, following techniques described in [SB18]. We have implemented and evaluated four popular techniques for this thesis.

Given a set of observed rewards, their mean value is the maximum-likelihood estimate for the expected value of the actual reward distribution. **Monte-Carlo (MC) learning** accordingly calculates the state-action values $q(s, a)$ as the mean of all observed rewards from performing action a in state s. The reward directly following a is added as is, while subsequent rewards within the episode are discounted by a factor $\gamma \in [0, 1]$, to reflect both the fact that earlier improvements are worth more than later improvements and that more recent heuristics are likely to have a stronger influence on the result than those far earlier in the chain.

Let an episode of length T be represented as a list of 3-tuples (s_t, a_t, r_t) denoting the state at step t, $0 \leq t < T$, the action taken, and the reward observed, respectively. In addition, we keep a list of returns $Returns(s, a)$ observed in earlier episodes for each state s and action a. Then we update the state-action values at the end of each episode as follows (for $0 \leq t < T$):

$$G_t = \sum_{i=0}^{T-t-1} \gamma^i r_{t+i} \tag{4.11}$$

$$Returns(s_t, a_t) = \text{Append } G_t \text{ to } Returns(s_t, a_t) \tag{4.12}$$

$$q(s_t, a_t) = average(Returns(s_t, a_t)) \tag{4.13}$$

Instead of learning only at the end of each episode, we can already update our state-action values after each individual action, as the observed reward immediately gives us feedback we can use. Of course, at this point we do not yet know the future rewards that will be observed until the end of the episode. Therefore we need to use our current estimates about the expected rewards, which we already have represented by the state-action value of the successor state and action. Accordingly, this technique is called **SARSA** ("state-action-reward-state-action") and the update of $q(s_t, a_t)$ is performed after choosing $q(s_{t+1}, a_{t+1})$:

$$q(s_t, a_t) \leftarrow q(s_t, a_t) + \alpha(r_t + \gamma q(s_{t+1}, a_{t+1}) - q(s_t, a_t)) \tag{4.14}$$

This makes use of the learning rate parameter α (see Section 4.3.1), which determines the magnitude of the update. α is applied to the difference (or estimation error) between the current estimate $q(s_t, a_t)$ and the sum of the observed reward r_t and the discounted expected future reward $(\gamma q(s_{t+1}, a_{t+1}))$.

A close variant of SARSA is **Q-learning**. While SARSA uses the state-action value of the actually chosen subsequent action, Q-learning assumes that at least eventually the policy will converge towards greedy selection as the state-action value estimates converge towards their true values. The rationale behind this is that we want to maximize the observed rewards, and the greedy policy achieves that goal if the true values are known. Following this assumption, Q-learning uses the state-action value of the (estimated) *best* action for the successor state to update the estimate of the current action. This leads to the following update function, applied after each action once the reward and successor state are observed:

$$q(s_t, a_t) \leftarrow q(s_t, a_t) + \alpha(r_t + \gamma \max_a q(s_{t+1}, a) - q(s_t, a_t)) \tag{4.15}$$

The disadvantage of Q-learning is that the actual policy usually is different from the greedy policy, in particular in the beginning. If we know that the best action is not always chosen, this should somehow also be reflected in the rewards we can expect in the future. A hybrid approach between SARSA and Q-learning is **Expected-SARSA**, which takes into account the probabilities of choosing each possible action. The expectation of the state-action values with respect to the policy π is then used in the update function. Let $\pi(a|s)$ denote the probability of choosing action a when in state s. Then $q(s_t, a_t)$ is updated as follows:

$$q(s_t, a_t) \leftarrow q(s_t, a_t) + \alpha(r_t + \gamma \sum_a \pi(a|s_{t+1}) q(s_{t+1}, a) - q(s_t, a_t)) \tag{4.16}$$

Solution chain length

To avoid getting lost in unfavorable areas of the search space, we periodically reset the current solution to the best known solution found so far. Both a reset and finding a new best solution indicate the end of the current episode of reinforcement learning.

There is of course a tradeoff here regarding the length of each solution chain: Longer chains before a reset have a higher chance of producing a new best solution. On the other hand, they also require a greater investment of time, which is lost if the chain is unsuccessful and the solution has to be reset.

In addition, the optimal solution chain length also depends on the current solution. For example, for local optima we will never find an improvement with chains of length 1 and for many domains, improvements will be easier to find early in the search than late, when the solution is already close to optimal.

Chuang [Chu20] investigated a similar model, under the assumption that low-level heuristics are chosen uniformly. They showed that if the probabilities $q(\ell)$ of finding an improvement within at most ℓ heuristic applications are known for all ℓ, there is a fixed ℓ^* such that it is optimal to reset the solution chain after ℓ^* unsuccessful applications.

For the more realistic case that $q(\ell)$ is unknown, Chuang proved that another strategy is optimal: Starting from an initial solution (after initialization or after a new best known solution was found), episode i is reset after $\mathcal{L}[i]$ unsuccessful applications. $\mathcal{L}$ is an infinite sequence of numbers defined as follows:

$$\mathcal{L}[i] = \begin{cases} 2^{k-1} & \text{if } i = 2^k - 1 \\ \mathcal{L}[i - 2^{k-1} + 1] & \text{if } 2^{k-1} \leq i < 2^k - 1 \end{cases} \tag{4.17}$$

$$\mathcal{L} = (1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, 1, 1, 2, \dots) \tag{4.18}$$

The strategy is called *Luby's strategy*, after Luby et al. [LSZ93], who first described the sequence in 1993.

The optimality proof was done in terms of the expected number of operations until an improvement is found. In practice, we are also interested in the magnitude of improvement (including its potential for further improvement down the line) and the assumption of uniform operator choice does not hold for most policies.

For our evaluations, we used a truncated variant of Luby's strategy which repeats from the beginning after reaching the value $2^{10} = 1024$. This was also done in [Chu20] to ensure that single solution chains cannot become so huge that they take up significant portions of the available time. Luby's sequence is also restarted from the beginning after each successful chain.

For comparison with Luby's strategy, we also implemented and evaluated two different strategies of choosing the chain length: The *2-constant* strategy always uses chains of length 2, i.e. the chain is aborted if two consecutive heuristic applications do not yield an improvement. In contrast, the *inf* strategy does not ever reject a chain. Episodes always end with an improvement to the currently best known solution, except at the absolute end of the search. This is the strategy used in [DGU12] and represents trusting the policy completely.

4.3.2 Empirical evaluation

As for the SA-LNS hyper-heuristics in the previous section, we implemented our RL approaches as hyper-heuristics in the HyFlex framework and evaluated them on the same benchmarking machine described in Section 4.2.3.

Component analysis

To analyze the effects of each algorithm component on the performance, we compared the different variants to a baseline configuration that performed well in preliminary experiments. This baseline configuration uses a state representation consisting of the last heuristic and the tail length up to a maximum length of 5 (LH+TL5), a 0.2-greedy action selection policy, constant rewards for improved solutions, a learning rate of 0.1 where required, and Luby's sequence to determine chain lengths. We repeated all experiments with the four update rules (MC, Q-learning, SARSA, E-SARSA) and performed 10 runs per instance, using the 10 early instances for the four initially known problem domains of the CHeSC. We also normalized the results to the range $[0 - 1]$ as described in Section 4.2.3.

First we compared the performance of the baseline configuration under the four update rules (Figure 4.10). Both MC learning and E-SARSA found results of comparable quality, slightly better than SARSA and Q-learning. Although the differences look small, due to this being normalized objective values they can actually translate to significant improvements for some problem domains - particularly under the CHeSC scoring system, which awards points for relative rank compared to the other competitors instead of absolute values.

For the remaining components, the effect of different configurations was mostly independent of the chosen update rule. For this reason, the figures and analysis presented below

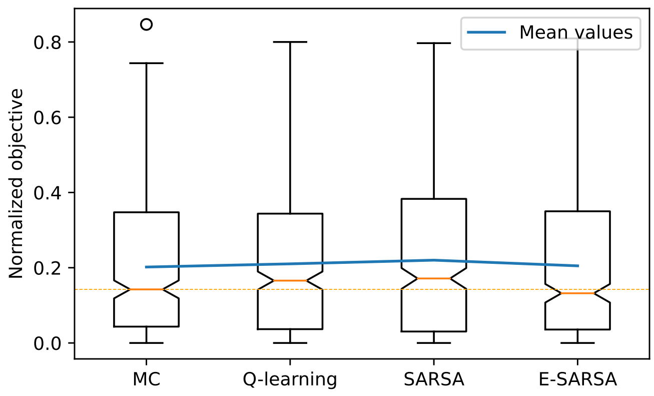


Figure 4.10: Normalized objective values for different state-action value update rules. The dotted line shows the median value of MC learning.

show combined results for all four update rules.

Figure 4.11 shows a comparison of different state representations. It can be seen that the choice of features does not result in drastic changes to the overall results. Surprisingly, even a configuration that does not distinguish between different states at all and learns a single value estimate for each low-level heuristic globally does not perform worse than the baseline configuration. However, the configurations using only the last heuristic applied (LH) or only the type of the last heuristic (LHT) perform slightly better than this.

Differentiating whether the hyper-heuristic found recent improvements with a window size of 10 (RI10) further improves the results by a small margin, both with and without tail length. This window size seems particularly well suited to model medium-term memory, as we did not find any similar improvement and sometimes even worse results with both longer and shorter window sizes. We also did an experiment comparing different cutoff-values for the maximum differentiated tail length, but found no significant effect on the results.

The effect of different action selection policies is shown in Figure 4.12. Both ε-greedy and ε-softmax produced similar results to each other, but a difference can be seen regarding the choice of ε. While a minimum value of ε is required to achieve good results, lower values tend to perform slightly better if ε remains constant over the course of the runtime. This is not the case for variants with decreasing ε, which show consistent performance over the whole value range.

Compared to other components, the choice of the solution chain length turned out to be crucial to achieve good results (see Figure 4.13). Both chains of constant length 1 and chains of infinite length (until an improvement is found or time runs out, no resets) result in particularly bad performance. While chains of longer constant lengths provide better results, following Luby's sequence turned out to be clearly better than any of the evaluated alternatives.

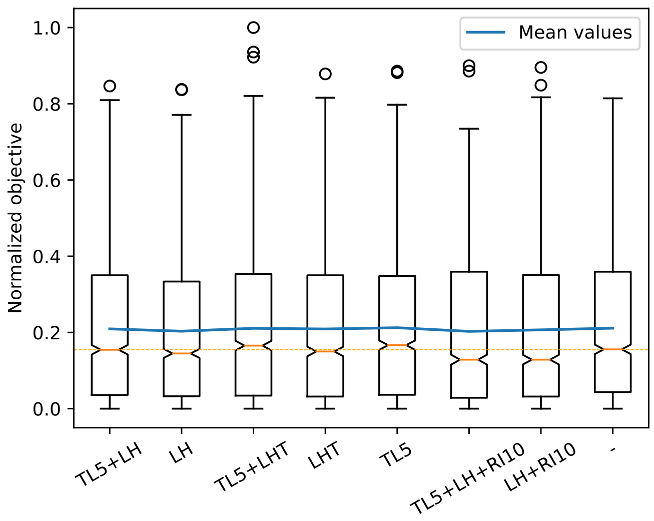


Figure 4.11: Normalized objective scores for state representations with different feature combinations: Tail length (TL5), last heuristic (LH), type of last heuristic (LHT), recent improvements (RI10). The dotted line shows the median of the baseline configuration (TL5+LH). The rightmost entry is a configuration that does not distinguish separate states at all.

A small improvement can also be achieved by assigning rewards equal to the total objective delta divided by the time taken by the last heuristic in the chain (Figure 4.14). This follows the intuition that heuristics which find good solutions fast should be prioritized.

Finally, we also compared different learning rates for Q-learning, SARSA, and E-SARSA but found no significant differences between different values for α and no general trend towards higher or lower values.

Comparison with other approaches

The component analysis in Section 4.3.2 indicates that the following configuration is likely to perform well as a hyper-heuristic: A state representation of LH+H10, with a 0.1-softmax policy with decreasing ε and rewards of $\frac{\sum \Delta o}{\sum \Delta t}$. The solution chain length follows Luby's sequence and state-action values are updated via MC learning. In the following, we denote this configuration as RL.

We evaluated our approach on all six domains contained in HyFlex, using the same set of benchmark instances that were used for the CHeSC 2011. To compare our results with those of the competition participants, we used the same scoring scheme as for the competition, awarding points per instance according to the ranking following the Formula 1 system [BGH$^+$11]. If RL had participated in the competition, it would have achieved second place (see Table 4.5), ahead of the approaches by [HCF12] and by [Lar11].

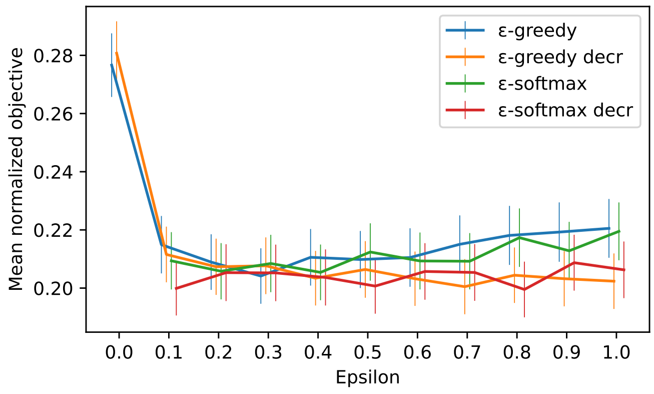


Figure 4.12: Mean normalized objective value for different policies at different values of ε. Softmax with $\varepsilon = 0$ led to numerical stability issues on some instances and is thus not included. The thin vertical lines denote the 95% confidence interval of the mean.

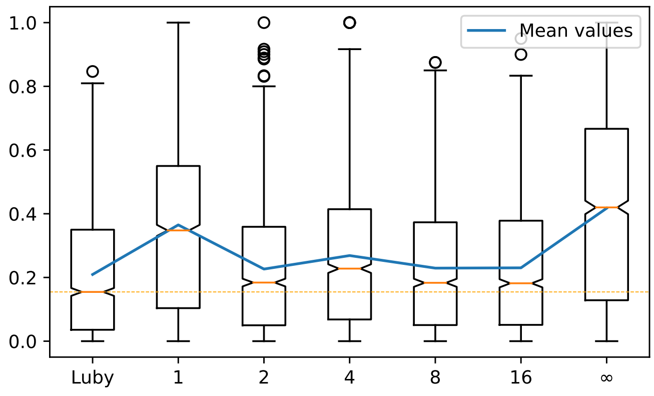


Figure 4.13: Normalized objective scores for different solution chain lengths. The dotted line shows the median of the baseline configuration (Luby).

In addition, we also performed a comparison with more recent hyper-heuristics, as we did in Section 4.2.3. We compared our results with the 20 original participants and the hyper-heuristics FRAMAB [FGTRP15], GEP-HH [SAKQ15], MCTS-HH [SK15], QHH [CWL18], TS-ILS [AOO21], FS-ILS [ABN14], and FI-Pruning [Chu20], for a total of 28 participants in this extended competition. Table 4.6 shows the results of the best approaches.

As expected from the comparison with the original competition participants, RL was able achieve the 7th place, closely behind QHH. The other approaches based on ILS, TS-ILS and FS-ILS, as well as the hyper-heuristics GEP-HH and MCTS-HH are more clearly ahead of RL.

Still, we were able to show that a direct application of reinforcement learning was able

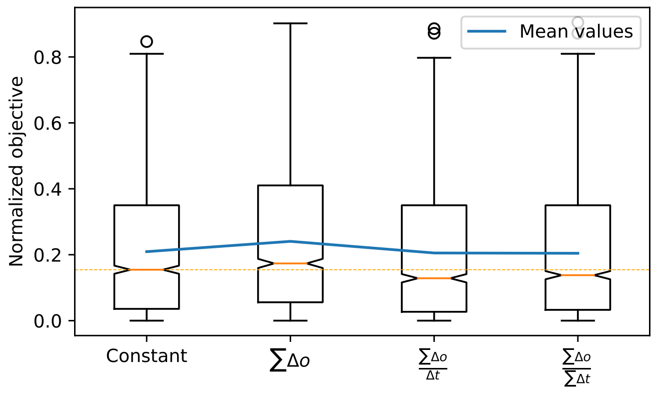


Figure 4.14: Normalized objective scores for different reward schemes. The dotted line shows the median of the baseline configuration (Constant).

Rank	Method	Reference	Score	SAT	BP	PS	FS	TSP	VRP
1	AdapHH	[MVDCVB12]	170.10	31.85	42.00	9.00	35.00	38.25	14.00
2	**RL**		126.60	30.35	37.00	2.00	27.25	18.00	12.00
3	VNS-TW	[HCF12]	125.10	32.85	2.00	39.50	29.50	15.25	6.00
4	ML	[Lar11]	118.50	10.50	8.00	31.00	36.00	12.00	21.00

Table 4.5: Top 4 approaches if RL had been a CHeSC participant. Shown are both the total score achieved and the scores on each of the six problem domains.

to produce good results that rival some of the state-of-the-art methods. In particular, we were able to significantly improve upon the reinforcement learning competition entry by Di Gaspero and Urli [DGU12], who achieved the 18th place in this extended competition. Given the results of parameter tuning described above, the main contributing factor for this success is likely the use of solution chains, which allow the algorithm to recover from bad decisions which would otherwise take a lot of effort to undo. Given the clear success of ILS-like structures for this cross-domain optimization task, it would be interesting to investigate how we can better make use of such structures within our RL framework, potentially improving already our initial estimates of low-level heuristic performance and promising operator sequences.

4.4 Application to Test Laboratory Scheduling

Previously, we have evaluated our hyper-heuristics on the problem domains provided for the CHeSC 2011. However, as general problem solving methods, they should also work well for project scheduling, and in particular for the TLSP. If this indeed turns out to be the case, we can also expect that they will be suitable for future problem variants and instances with different characteristics. For this evaluation we can again use the HyFlex

Rank	**Method**	**Reference**	**Score**	**SAT**	**BP**	**PS**	**FS**	**TSP**	**VRP**
1	TS-ILS	[AOO21]	188.61	33.36	34.00	33.50	31.50	37.25	19.00
2	GEP-HH	[SAKQ15]	126.88	20.62	23.00	0.00	50.00	13.25	20.00
3	FS-ILS	[ABN14]	113.20	33.36	3.00	3.00	26.83	26.00	21.00
4	MCTS-HH	[SK15]	107.24	13.99	19.00	31.00	7.00	18.25	18.00
5	AdapHH	[MVDCVB12]	87.49	10.99	32.00	4.00	15.00	20.50	5.00
6	QHH	[CWL18]	69.24	12.99	1.00	0.00	16.00	22.25	6.00
7	**RL**		67.49	9.49	29.00	0.00	12.00	9.00	8.00
8	ML	[Lar11]	56.20	1.36	0.00	25.00	14.83	2.00	13.00
9	VNS-TW	[HCF12]	55.63	15.62	0.00	27.50	9.00	3.50	0.00
10	PHunter	[CXIC12]	40.36	1.36	0.00	8.50	0.00	10.50	20.00
11	FRAMAB	[FGTRP15]	39.36	1.36	0.00	16.00	0.00	0.00	22.00
15	FI-Pruning	[Chu20]	26.36	1.36	16.00	0.00	4.00	0.00	5.00

Table 4.6: Selection of results of the extended competition including RL and other recent hyper-heuristics. Shown are both the total score achieved and the scores on each of the six problem domains.

framework, by developing a new problem domain for the TLSP.

4.4.1 TLSP as a HyFlex problem domain

To implement TLSP as a problem domain in HyFlex, we need to provide several components: Solutions are stored and manipulated using the existing data structures developed for the metaheuristic solution framework (Chapter 3). These also come with included evaluation methods, including caching for efficient access. Solutions are initialized by grouping tasks into as few jobs as possible and assigning random modes, time slots, and resources to jobs (respecting precedence, time window, and availability constraints). As benchmark instances, we use the set of 30 instances used in previous evaluations (Table 3.1).

HyFlex assumes that the initial solutions and all intermediate solutions produced by low-level heuristics are already feasible. For TLSP, this cannot be guaranteed, as finding feasible solutions is already NP-hard. Therefore we added hard constraint violations to the objective function with a weight of 10000. As the best feasible solutions found for even the largest instances in our benchmark set have objective values well below this number, this allows relatively reliable identification of feasible solutions from the objective value alone.

Low-level heuristics

The low-level heuristics used for the TLSP problem domain are a combination of existing operators, using the TLSP neighborhoods and search heuristics, and new operators inspired by literature on other project scheduling problems. In total, the problem domain includes 6 mutation operators, 13 local search heuristics, 2 ruin-and-recreate operators, and 3 crossovers. All operators assume that constraints H1-H7 as well as H9 and H10 are fulfilled, and guarantee in return that they do not introduce further conflicts for these constraints. In other words, an intermediate schedule may contain conflicts only for constraints H8 (Single assignment) and H11 (Linked jobs).

The mutation operators apply random changes to a given schedule:

Random mode move Randomly changes the assigned mode of a job to different one. Replacement modes can be selected only if this would not introduce conflicts regarding precedences or time windows. If this changes the number of required employees, the randomly chosen employees are added or removed.

Random timeslot move Randomly moves of a job to a different time slot.

Random resource move Randomly replaces a single workbench, employee, or device assigned to a job by a different one.

Random regrouping move Performs a single random move from either of the Split, Merge, or TaskTransfer neighborhoods (see Section 3.2.2).

Randomize A larger mutation that selects a subset of all projects and randomizes the assignments of all jobs, leaving only the grouping intact. The IoM parameter determines the fraction of projects randomized.

Random walk Performs multiple moves of a random walk procedure, using all scheduling and regrouping neighborhoods. The number of moves is proportional to the DoS parameter, with a maximum of 100 moves at DoS = 1.0.

The ruin-and-recreate operators all follow the same principle: They delete assignments from a subset of all jobs and then greedily restore them one job at a time. The order in which the assignments will be restored is always the same: First, fixed assignments are added, then projects are selected in order of increasing release date. Within a project, jobs are selected in an arbitrary topological order.

Greedy reconstruct (multiple projects) Affects all jobs in a subset of all project. Leaves the task grouping unaffected.

Greedy regrouping (single project) Affects all jobs of a single project, including their task grouping. A new grouping of tasks into jobs is also built greedily, assigning tasks to existing matching jobs wherever this is possible without conflict.

The local search heuristics mostly perform one or several moves of a local search procedure with different neighborhoods (see Sections 3.2, 3.3). Those search heuristics allowing worsening moves return the best solution found during the search.

Hill climbing (mode, timeslot) Selects the best change in either mode or time slot for any job at each move. The number of moves is determined by the DoS parameter, with a maximum of 100 moves at DoS = 1.0.

Hill climbing (resources) Selects the best change of a single assigned resource unit for any job at each move. The number of moves is determined by the DoS parameter, with a maximum of 100 moves at DoS = 1.0.

Hill climbing (JobOpt) Selects the best move from the JobOpt+EC neighborhood (see Section 3.2.1) for any job at each move. The number of moves is determined by the DoS parameter, with a maximum of 100 moves at DoS = 1.0.

Hill climbing (regrouping) Selects the best move from the TaskTransfer, Split, or Merge neighborhoods (see Section 3.2.2) for any job at each move. The number of moves is determined by the DoS parameter, with a maximum of 100 moves at DoS = 1.0.

MinConflict (mode, timeslot) Selects the best change in either mode or time slot for a randomly chosen job at each move. The number of moves is determined by the DoS parameter, with a maximum of 100 moves at DoS = 1.0.

MinConflict (resources) Selects the best change of a single assigned resource unit for a randomly chosen job at each move. The number of moves is determined by the DoS parameter, with a maximum of 100 moves at DoS = 1.0.

MinConflict (JobOpt) Selects the best move from the JobOpt+EC neighborhood for a randomly chosen job at each move. The number of moves is determined by the DoS parameter, with a maximum of 100 moves at DoS = 1.0.

MinConflict (regrouping) Selects the best move from the TaskTransfer, Split, or Merge neighborhoods for a randomly chosen job at each move. The number of moves is determined by the DoS parameter, with a maximum of 100 moves at DoS = 1.0.

Simulated annealing (high temperature) Performs a large number of random moves over all available neighborhoods, using the Metropolis acceptance criterion of SA at a fixed temperature of 100. The weight of hard constraint violations is set to 50. At this high temperature, moves that add no more than one or two hard constraint violations will likely be accepted. The number of moves is determined by the DoS parameter, with a maximum of 50000 moves at DoS = 1.0.

Simulated annealing (low temperature) Performs random moves with the Metropolis acceptance criterion as the previous operator, but at a fixed temperature of 5. At this low temperature, only moves that do not result in additional constraint violations have a realistic chance to be accepted. The number of moves is again determined by the DoS parameter, with a maximum of 50000 moves at DoS = 1.0.

Simulated annealing (minimal temperature) Performs random moves with the Metropolis acceptance criterion as the previous two operators, but at a fixed temperature of 1. At this minimal temperature, moves are unlikely to be accepted if they increase the penalty at all, although there is a chance for very small increases. The number of moves is again determined by the DoS parameter, with a maximum of 50000 moves at DoS = 1.0.

Single project CP Uses a CP solver to find the optimal solution for a single, randomly chosen project, while the rest of the schedule is kept fixed. If the optimum cannot be found within a given time, the best solution found is returned instead. The time available is determined by the DoS parameter, with a maximum of 30s at DoS = 1.0.

Job-wise greedy While not a local search in the strict sense, this special operator still fulfills the corresponding criteria of HyFlex, since it never returns a worse solution. It iterates over all jobs and replaces each job's assignments with the locally best ones, relative to the current assignment of all other jobs. Jobs are ordered according to their project's earliest release date, and within a project in an arbitrary topological order.

Finally, the crossover operators were newly developed for inclusion in the TLSP problem domain. They take two parent schedules as input and produce as offspring a new schedule that contains assignments of both parents.

Random project Selects for each project independently whether the offspring should take the assignments of the first or the second parent schedule.

Single point Randomly selects a time slot. Assignments for all projects starting before that point are taken from the first parent, the remaining assignments are taken from the second parent.

Two point Randomly selects two time slots. Assignments for all projects starting before the first or after the second slot are taken from the first parent, the remaining assignments are taken from the second parent.

4.4.2 Evaluation

We evaluated the performance of several state-of-the-art hyper-heuristics on the new TLSP problem domain. Besides our own approaches with SA-LNS and RL, we also acquired the source code of the following hyper-heuristics: AdapHH [MVDCVB12], the winner of the CHeSC 2011, and FS-ILS [ABN14]. For each hyper-heuristic, we executed 15 runs on all 30 benchmark instances included in the problem domain, at a timeout of 10 minutes, using the same machine as in the previous section.

The results are shown on Table 4.7, with a summary on Table 4.8. Both our hyper-heuristic approaches were able to find feasible solutions for all instances, with consistently

#	BTWSA-LNS			RL			AdapHH			FS-ILS		
	#Feas	**Avg**	**Best**	**#Feas**	**Avg**	**Best**	**#Feas**	**Avg**	**Best**	**#Feas**	**Avg**	**Best**
1	15/15	57.0	**57**	15/15	57.0	**57**	15/15	57.0	**57**	15/15	57.0	**57**
2	15/15	71.0	**71**	15/15	71.0	**71**	15/15	71.0	**71**	15/15	71.0	**71**
3	15/15	141.5	**141**	15/15	141.1	**141**	15/15	141.0	**141**	15/15	141.1	**141**
4	15/15	104.8	**101**	15/15	101.1	**101**	15/15	101.1	**101**	15/15	101.7	**101**
5	15/15	286.5	282	15/15	281.7	**276**	15/15	282.0	280	15/15	283.2	279
6	15/15	151.6	142	15/15	144.9	**140**	15/15	142.3	**140**	15/15	145.7	140
7	15/15	310.4	**292**	15/15	302.7	293	15/15	300.3	**292**	15/15	300.3	296
8	15/15	299.9	295	15/15	295.7	291	15/15	296.4	**290**	15/15	297.0	292
9	15/15	460.5	442	15/15	471.1	**432**	15/15	464.9	443	15/15	454.3	441
10	15/15	624.6	566	15/15	640.3	570	15/15	587.7	**540**	15/15	585.7	549
11	15/15	896.0	868	15/15	882.5	852	15/15	868.4	**851**	15/15	901.0	871
12	15/15	684.3	655	15/15	674.2	**652**	15/15	660.5	**652**	15/15	682.2	670
13	15/15	333.0	**317**	15/15	330.9	319	15/15	331.9	321	15/15	326.6	319
14	15/15	422.5	415	15/15	421.4	411	15/15	416.1	**410**	15/15	422.9	418
15	15/15	1235.8	1131	13/15	1261.3	1097	15/15	1182.0	**1056**	14/15	1260.4	1206
16	15/15	1269.1	1201	14/15	1239.4	1194	15/15	1227.3	**1190**	15/15	1258.2	1205
17	15/15	1233.3	1144	15/15	1260.0	1201	15/15	1192.1	1137	15/15	1199.6	**1136**
18	15/15	1532.6	1456	13/15	1490.2	1409	15/15	1433.1	**1372**	15/15	1498.1	1454
19	14/15	2239.2	2073	8/15	2231.9	2152	13/15	2115.7	**1993**	13/15	2200.3	2130
20	11/15	2562.2	2320	9/15	2410.1	2304	11/15	2344.0	**2231**	2/15	2467.5	2463
21	15/15	690.9	650	14/15	690.6	635	15/15	662.6	**615**	13/15	679.4	649
22	15/15	781.0	744	15/15	791.4	752	15/15	754.0	**733**	15/15	785.0	750
23	12/15	2546.8	2139	9/15	2416.2	2000	13/15	2190.2	**1987**	10/15	2263.4	2064
24	15/15	2131.2	2025	13/15	2040.5	1953	15/15	1975.3	**1923**	10/15	2080.9	1988
25	11/15	3366.1	2778	5/15	3014.8	2789	11/15	2693.6	**2497**	10/15	2968.4	2647
26	9/15	3272.3	3043	5/15	3082.0	2827	7/15	2883.6	**2808**	6/15	3133.5	2953
27	15/15	2511.5	2319	15/15	2613.7	2289	15/15	2359.0	**2186**	15/15	2474.5	2284
28	15/15	2550.6	2432	12/15	2523.6	2384	15/15	2386.2	**2350**	13/15	2515.7	2462
29	1/15	3910.0	**3910**	4/15	4573.8	4422	9/15	4228.6	4037	3/15	4601.0	4272
30	3/15	5778.0	5310	1/15	5421.0	5421	4/15	5249.3	**5052**	6/15	5650.5	5293

Table 4.7: Experimental results for 4 hyper-heuristics on the TLSP problem domain, using a timeout of 10 minutes. Shown are for each hyper-heuristic the number of feasible solutions, the average objective value among feasible solutions and the best objective value found. The best solution found for each instance by any of the hyper-heuristics is marked in bold.

good quality that rivals the current state-of-the-art results. They even were overall competitive to FS-ILS, with BTWSA-LNS finding more feasible solutions and both approaches producing more best solutions than FS-ILS. AdapHH achieved the best results overall, finding the best solution on 25 out of the 30 instances and also the most feasible solutions. Interestingly, AdapHH clearly outperformed FS-ILS on the TLSP problem domain, despite their placement on the CHeSC 2011 problem domains.

In addition, we compared our approaches to two state-of-the-art algorithms developed for the TLSP, namely SA described in the previous chapter (Section 3.3 and [MMS21a]) and VLNS [GMM19a]. A direct comparison with the results of Table 4.7 is not possible

Hyper-heuristic	**% Feas**	**% Gap**	**#Best**
AdapHH	0.92	1.04	25
FS-ILS	0.86	1.07	5
BTWSA-LNS	0.90	1.09	7
RL	0.83	1.08	8

Table 4.8: Summary of results from Table 4.7. Shown are the percentage of runs that produced feasible solutions, the average solution quality relative to the best known solution, and the number of instances for which a hyper-heuristic found the best solution.

due to the difference in the used hardware. For this reason, we repeated the experiments with RL and BTWSA-LNS on the same machine used for the experiments in Chapter 3.

The results of this comparison are shown in Table 4.9. It can be seen that while the hyper-heuristics cannot reach the specialized solution approaches in general, their performance is more balanced across instances: For small instances, the hyper-heuristics produce better results than SA and come very close to those of VLNS. For larger instances, both hyper-heuristics outperform VLNS, although they cannot reach the solution quality of SA. Combined with the fact that the hyper-heuristics are additionally also quite successful on completely unrelated problem domains, this underscores the generality of these approaches, particularly where the problem characteristics or the instance sizes are unknown in advance.

4.5 Summary

This chapter dealt with problem-independent selection hyper-heuristics. We have developed new hyper-heuristics, which fall into two categories:

The first approach uses an algorithm scheme following the ILS structure of perturbation-repair cycles, which was very successful also for previous hyper-heuristics. The most successful variant of this approach used a weight function for the low-level operators that takes both global performance and local performance in an adaptive recent time window into account [TS18]. This variant was able to place among the top 4 competitors of the CHeSC 2011 competition, although it was unable to beat more recent hyper-heuristics, including several others using an ILS structure. Further improvements may be possible by allowing multiple operator applications within a single cycle of ILS.

The second approach employs a natural translation of the selection hyper-heuristic setting to RL, with low-level heuristic applications used as actions. For this approach we performed a deep analysis of different options for the remaining algorithm components. It turned out that the most critical modeling choice was the use of solution chains, which reset after a certain number of applications without improvement. This allows the hyper-heuristic to avoid getting stuck in umpromising areas of the solution space. A similar model using pure RL, though without solution chains, was used already in earlier

#	BTWSA-LNS			RL			SA			VLNS		
	#Feas	Avg	Best	#Feas	Avg	Best	#Feas	Avg	Best	#Feas	Avg	Best
1	15/15	57.0	**57**	15/15	57.0	**57**	15/15	58.0	58	5/5	57.0	**57**
2	15/15	71.0	**71**	15/15	71.0	**71**	15/15	72.4	72	5/5	71.0	**71**
3	15/15	142.1	**141**	15/15	141.7	**141**	15/15	156.1	147	5/5	141.0	**141**
4	15/15	105.6	**101**	15/15	102.1	**101**	15/15	114.5	103	5/5	101.0	**101**
5	15/15	288.7	282	15/15	285.1	281	15/15	303.3	296	5/5	242.2	**240**
6	15/15	153.6	147	15/15	148.3	141	15/15	165.1	157	5/5	140.0	**140**
7	15/15	311.2	295	15/15	304.1	292	15/15	300.9	296	5/5	287.0	**283**
8	15/15	303.1	298	15/15	298.9	290	15/15	302.4	292	5/5	284.6	**283**
9	15/15	465.6	435	15/15	478.5	454	15/15	470.6	456	5/5	442.2	**429**
10	15/15	624.5	568	15/15	634.5	556	15/15	566.5	551	5/5	590.2	**547**
11	15/15	926.4	873	15/15	914.5	868	15/15	938.4	912	5/5	860.8	**840**
12	15/15	696.7	656	15/15	689.3	656	15/15	675.8	666	5/5	660.6	**653**
13	15/15	334.3	320	15/15	330.5	324	15/15	338.2	327	5/5	311.8	**309**
14	15/15	428.5	420	15/15	425.4	415	14/15	420.4	418	5/5	415.0	**412**
15	15/15	1333.7	1216	15/15	1366.2	1115	15/15	1063.5	1014	5/5	1106.2	**1025**
16	15/15	1281.9	1206	10/15	1305.4	1251	13/15	1236.7	1216	5/5	1184.4	**1175**
17	15/15	1232.5	1184	15/15	1289.9	1177	15/15	1185.3	**1140**	5/5	1166.6	1150
18	15/15	1588.2	1494	15/15	1560.3	**1412**	15/15	1527.4	1500	5/5	1482.2	1436
19	10/15	2471.9	2232	9/15	2487.1	**2131**	14/15	2239.9	2133	5/5	2419.2	2350
20	1/15	2649.0	2649	6/15	2741.8	2512	15/15	2489.7	**2391**	5/5	2986.6	2955
21	15/15	700.0	627	7/15	727.7	627	15/15	673.9	632	5/5	596.8	**570**
22	15/15	800.3	**749**	15/15	832.1	766	15/15	784.6	755	5/5	775.0	763
23	9/15	2724.6	2374	9/15	2599.9	2178	12/15	2167.7	**2070**	0/5	-	-
24	14/15	2258.2	2043	11/15	2116.4	2016	15/15	1869.3	**1807**	0/5	-	-
25	2/15	2976.5	2968	8/15	3716.4	2765	10/15	2897.6	**2601**	0/5	-	-
26	1/15	3170.0	3170	6/15	3278.8	2945	13/15	2971.3	**2809**	5/5	3345.4	3109
27	15/15	2766.0	2423	12/15	2662.1	2195	15/15	2124.3	**2017**	5/5	2750.6	2473
28	15/15	2673.5	2484	12/15	2667.4	2477	15/15	2516.5	2468	5/5	2407.2	**2372**
29	1/15	4744.0	4744	0/15	-	-	10/15	4358.5	**3965**	0/5	-	-
30	0/15	-	-	1/15	6746.0	6746	14/15	5104.1	**4989**	0/5	-	-

Table 4.9: Comparison of results using hyper-heuristics with approaches specifically tuned for the TLSP (SA and VLNS from Table 3.9), using a timeout of 10 minutes on the same machine used for the experiments in Section 3.4.2. Shown are for each approach the number of feasible solutions found, the average objective value among feasible solutions and the best objective value found. The best solution found by any of these approaches for each instance is marked in bold.

work [DGU12], but without much success. With our improved model, we were able to show that pure RL algorithms are indeed a useful approach for selection hyper-heuristics, as we were able to outscore all but the winner of the CHeSC 2011 and achieved similar results to a state-of-the-art hyper-heuristic employing Q-learning within an ILS framework [CWL18].

In addition, we have modeled the TLSP as a problem domain for selection hyper-heuristics, including a diverse set of low-level operators. With this model, we were able to use both the hyper-heuristics developed in this thesis and state-of-the-art hyper-heuristics from the literature to solve the TLSP. While the previously described problem-specific solution methods for the TLSP (namely SA and VLNS) still have the advantage overall, our results show that the hyper-heuristic approaches provide high-quality solutions and in addition are less sensitive to instance parameters such as the number of projects. This indicates that hyper-heuristics are a promising approach that may also generalize well to other problem variants.

CHAPTER 5

Deployment in a Real-World Industrial Laboratory

Right from the start of our research, the solution methods developed for the TLSP were intended to be deployed in real-world test laboratories. In the literature on combinatorial optimization, the focus is usually put on the theoretical modeling and evaluation of the problem. Often this is still the case if the underlying problem or even the benchmark instances are taken from or inspired by real-world applications. However, an automated scheduling solution that is useful in practice requires more than an algorithm with good performance on some benchmark sets. It must be integrated within a comprehensive software system, which includes data acquisition and management, a configuration interface, and output that is comprehensible and useful for humans. In addition, the deployment process can provide valuable feedback that may improve the models and algorithms in use.

In this chapter, we describe how the algorithms developed and introduced in the previous chapters were included in an automated scheduling system that was deployed in the test laboratory of our industrial partner, which is a large laboratory performing electromagnetic compatibility (EMC) and other tests of car control units as well as similar components. We show how automated scheduling has increased the quality of their schedules and decreased the time and effort spent on creating and maintaining them. The work described in this chapter was performed in close cooperation with the laboratory of our industrial partner, and was accepted for publication in [DGJ$^+$21].

This chapter is structured as follows: In the next section, we discuss related literature on the deployment of automated scheduling methods in real-world settings, as well as give an overview of the current scheduling practice of industrial test laboratories. In Section 5.2, we give an overview of the current software environment in our partner laboratory, with a focus on the scheduling tools and processes. The deployment history and adoption in practice of our automated scheduling methods is described in Section 5.3. In order to adapt to changing requirements, it was necessary to extend the TLSP with additional constraints and objectives, which are given in Section 5.4. Section 5.5 contains an analysis of objective weights, both with respect to their impact on each other and the solution time, and to the current configuration of weights in practice. Finally, we give a summary and some important lessons learned as part of the deployment process in Section 5.6.

5.1 Related literature and state-of-the-art in practice

Next to a considerable amount of theoretical work in the area of project scheduling, many if not most variants of the RCPSP are inspired by real-world requirements. We give here two examples of automated scheduling approaches intended for use in real-world laboratories: Bartels and Zimmermann [BZ09] deal with (destructive) tests of experimental vehicles, while a scheduling method based on MSPSP for a nuclear laboratory is described Polo Mejia et al. [PMAAL17]. The focus of these papers is the theoretical formulation and evaluation of the treated (formal) problems, with no or only minimal description of the actually deployed system.

Despite the extent of the available literature, the adoption of advanced automated scheduling solution in practice is still not commonplace. Several authors have remarked upon this gap between research and practice, and proposed suggestions to improve the situation. Williams [Wil03] already remarked in 2003 that many mathematical models were „motivated by mathematical impressiveness rather than the need to solve real-world problems“, and that many of the more complex techniques were never taken up by project management professionals. Similarly, also Herroelen [Her05] found that „many project scheduling procedures have not yet found their way to practical use“. Herroelen gives an overview of the state of the art in project scheduling practice and describes how both the availability and the adoption of automated scheduling techniques has increased over time. Still, the gap remains even today, as discussed e.g. by Scales [Sca20] in 2020, who gave several proposals to bridge this gap within a Design Science Research framework.

We also performed our own investigation of scheduling practices among industrial test laboratories similar to those of our industrial partner. In most of them, scheduling was still performed manually, with the aid of tools like Microsoft Excel or Microsoft Project, sometimes supported by custom macros for special cases. More commonly used were general laboratory management systems, such as those offered by dacore[1] or systems developed in-house. The focus of these systems is usually on administration and analysis, and automated scheduling capabilities are limited, if supported at all. Quite often, automated scheduling is limited to local changes, only affecting single activities or projects. Only in recent years have commercial laboratory management systems emerged that also offer extended scheduling capabilities, such as the one by Binocs[2].

[1]`https://dacore-dbs.com`
[2]`https://discover-binocs.com/`

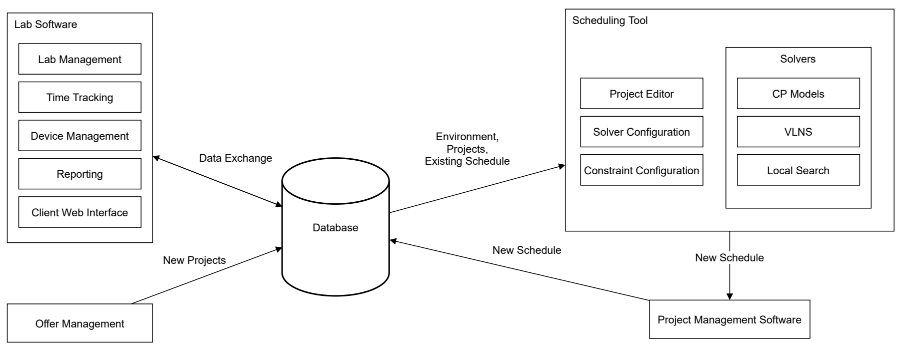


Figure 5.1: Overview of the laboratory management system of our industrial partner.

Unfortunately, the proprietary algorithms typically used in those systems are not open for scientific review. Their adoption into and success in industrial test laboratories remains to be seen over the coming years.

5.2 System overview

The scheduling tool is a component of the overall laboratory management system. An overview of this system is shown in Figure 5.1. At the center of the software infrastructure is a relational database, which connects to several software components. These components provide different services, such as time tracking, device management, and reporting. New projects are first specified using an offer management component, from which they are imported to the database for scheduling. The same component is also used to input updates or revisions to the project data.

Schedules can be created or updated with the automated scheduling tool. When launched, it queries data about the current laboratory environment, projects, and the existing schedule from the central database. The user can then make some limited adjustments to the project data and select one or multiple solver configurations to be executed. When the solving process terminates, the new schedule(s) can be imported into the project management software used at the laboratory. In this case, this is Microsoft Project extended by custom macros. From there, the schedule can be uploaded into the database. This design was motivated by the iterative development process of the scheduling tool. Microsoft Project was originally used as part of the manual scheduling process, and keeping it in the loop enables human expert planners to switch between automatic and manual scheduling as needed as well as quickly check the automatically generated schedules. As the scheduling tool is now close to reaching maturity, the current plan is to integrate some remaining features from Microsoft Project into the custom lab software and let the scheduling tool upload schedules directly to the database.

5.2.1 The automated scheduling tool

This scheduling tool itself is written in Java 8, features a graphical user interface, and includes different solvers. Currently, there are solvers based on CP [GMM19b, DGMM20], VLNS [GMM19a, DGMM20], as well as simulated annealing (Section 3.3).

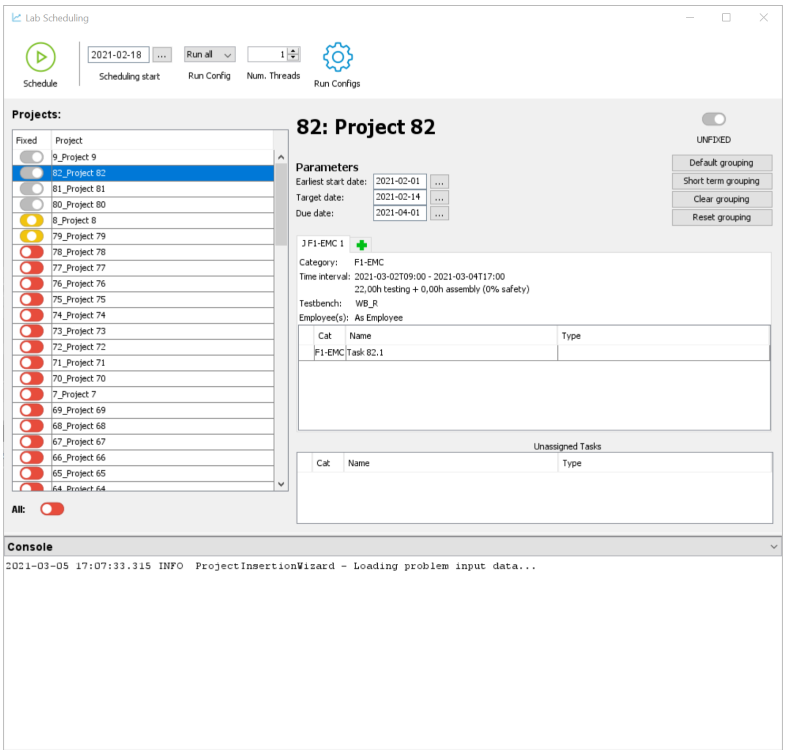
Figure 5.2 shows the main screen. The header menu is used to configure and run solvers, the details of which are described later. The main contents of the screen allow the user to browse projects and make various adjustments. Via the project list on the left, users can select and display individual projects. The sliders in the list can be used to fix projects, which is possible at two levels: When a project is completely fixed (red), its task groupings, time slot assignments, and resource assignments may not be modified by the solver. The purpose of this is two-fold: On the one hand, fixing certain projects may be desired for administrative reasons (e.g. in the case of manual adjustments). When changes are necessary only for limited projects or in a limited time window, it can also speed up the scheduling process to fix everything else. An intermediate option (yellow) is used to fix only the task grouping. Solvers automatically detect if all projects have at least their grouping fixed, and run in TLSP-S mode in that case, which may be more efficient.

When a project is selected, the interface on the right allows the user to modify the task groupings. This can either be done manually, or by applying different grouping heuristics using the buttons on the top right. The user can also modify the start date, target date, and due date (deadline) of projects. Changes to these dates are not automatically applied to the database due to the indirect application of the schedule. Instead, this feature is intended to allow the user to quickly experiment with different project time windows for e.g. placing a new project or rescheduling an existing one.

Aside from working with projects, the scheduling tool also offers various configurations for the solvers themselves. Figure 5.3 shows two dialogs. The run configuration editor in Figure 5.3a allows the user to specify and configure a list of solvers to use, including *Heuristic* (SA), *Exact* (CP) and *Hybrid* (VLNS). These configurations are saved between restarts, and thus several configurations can be created for quick access. Each run configuration also has a *constraint configuration* (Figure 5.3b). While some basic constraints (such as single assignment and start or due dates) are hard-coded and always active, many others can be tweaked, changed from hard to soft constraints, and even turned off. For soft constraints, the user can modify their weight towards the objective function.

5.3 Deployment history

The definition of the TLSP, the development of algorithms to solve it, the implementation of those algorithms within the laboratory, and their deployment in practice were not strictly separated events, but rather synchronous and interwoven processes that started at the begin of our collaboration with the laboratory in 2017 and continue to the current day.



Scheduling start: 2021-02-18
Run Config: Run all
Num. Threads: 1

Projects:

Fixed	Project
	9_Project 9
	82_Project 82
	81_Project 81
	80_Project 80
	8_Project 8
	79_Project 79
	78_Project 78
	77_Project 77
	76_Project 76
	75_Project 75
	74_Project 74
	73_Project 73
	72_Project 72
	71_Project 71
	70_Project 70
	7_Project 7
	69_Project 69
	68_Project 68
	67_Project 67
	66_Project 66
	65_Project 65
	64_Project 64

82: Project 82

Parameters
Earliest start date: 2021-02-01
Target date: 2021-02-14
Due date: 2021-04-01

J F1-EMC 1

Category: F1-EMC
Time interval: 2021-03-02T09:00 - 2021-03-04T17:00
22,00h testing + 0,00h assembly (0% safety)
Testbench: WB_R
Employee(s): As Employee

Cat	Name	Type
F1-EMC	Task 82.1	

Unassigned Tasks

Cat	Name	Type

Console

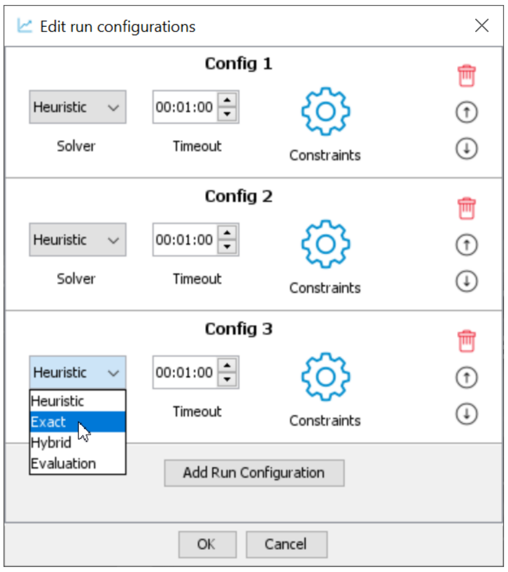
```
2021-03-05 17:07:33.315 INFO  ProjectInsertionWizard - Loading problem input data...
```

Figure 5.2: The main user interface of the scheduling tool.

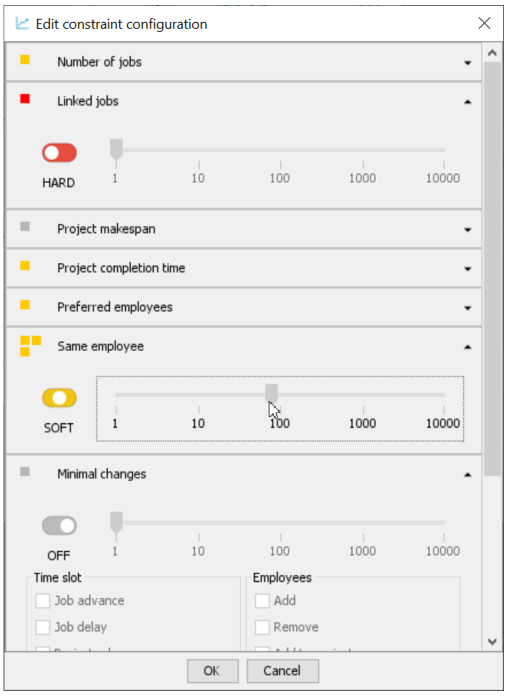
5.3.1 Initial situation

Before the start of the research project, scheduling was performed completely manually by a human expert planner. At that time, the laboratory was considerably smaller than today, with about 10 employees, 4 anechoic testing chambers suitable for ESD tests and 6 workbenches for other tests. The laboratory schedule was manually managed with Microsoft Project, while project data was kept in spreadsheets. Despite considerable domain knowledge and expertise of the human planner, working full-time on maintaining the schedule, several constraints, such as equipment assignment, could not be considered at all or only partially and the duration of the scheduled jobs did not always match the duration of the contained tasks. The resulting conflicts meant that frequent short-term rescheduling was necessary and projects occasionally had to be delayed due to scheduling conflicts.

Detailed statistics and conflict data from this period is no longer available, as the laboratory environment has changed considerable since then. However, a snapshot of the current schedule (which was still manually generated) from the end of 2018 shows a total of 1656 constraint violations. Of these, 526 concern the single assignment constraint (H8), 729 were precedence conflicts (H6), and there were 362 missing or unsuitable resource assignments (H9, H10). The remaining conflicts mostly cover invalid modes or unassigned



(a) Solver configuration



(b) Constraint configuration

Figure 5.3: Dialogs for managing different solver and constraint configurations.

tasks (H1). Furthermore, task durations were automatically matched to the scheduled durations of their jobs for this snapshot and equipment assignment was not considered at all at that time, so potentially a large number of additional conflicts is not even visible in this snapshot.

5.3.2 Requirements and data management

The first step towards automating the scheduling was the definition of the underlying optimization problem. We developed the TLSP by analyzing the processes and requirements of the laboratory, as well as relevant project scheduling literature. The goal was to provide a formal definition of a problem that could both cover the specific requirements of our partner laboratory and be useful for industrial test laboratories in general.

Having this definition then allowed us to determine what kind of data would be needed by an automated scheduling solution, such as information about the available resources and their capabilities, project data, and employee's schedules. Initially, this data was distributed across a variety of locations, including two different relational databases, several spreadsheets, and paper documentation. Additionally, a lot of domain knowledge, in particular with respect to scheduling constraints and objectives, was never formally written down, but based on the expertise of the human planner.

In order to consolidate and clean up all this information, we designed a comprehensive relational database as a replacement for most of the previously distributed and often unstructured data sources. Of course, this database has to kept up to date, which required some changes to the processes in the lab. To keep the impact on the employees and the general efficiency of the lab as low as possible, we designed multiple tools to assist with the data entry and the synchronization between the existing data formats and the database. Over time, some of these tools were integrated into the lab management software, or became obsolete as the old data formats and processes were abandoned in favor of working exclusively with the lab database data. Eventually, the goal is to move towards a completely integrated software environment which offers comprehensive data management options and has no need to rely on external data sources and tools.

While the database was mainly designed for scheduling purposes, it has been continuously extended and it is now accessed by many different services within the lab, such as report generation, employees' time management, live status tracking of projects, and device management (compare Figure 5.1). This way, it has become the core of the lab's software infrastructure and provides benefits beyond its initial purpose. Also the additional reporting processes introduced to keep the database up to date have enabled new and improved services, such as live monitoring of project and test statuses for both laboratory management and external clients.

5.3.3 Testing phase

As soon as the first scheduling algorithms were developed, they were deployed and evaluated on real-world data within the laboratory. At that point, the actual production schedules were still created manually, while at the same time being compared with automatically generated schedules. The first prototypes were also quite limited in functionality: They supported only the TLSP-S and relied on manually or heuristically generated task groupings. In addition, they were limited to adding a single project to the schedule (a scenario that is currently equivalent to fixing all projects save one).

During this phase, a big challenge was dealing with the numerous conflicts in the manually generated schedules, many of which could not be resolved by intelligent rescheduling. While some algorithms, such as SA, were not impacted by the presence of unresolvable conflicts, others, such as CP, could not be used on the real-world data initially. Since a typical schedule covers a time period of one and a half up to two and a half years, these conflicts could be expected to remain present for quite a long time. Here, the evaluation functions of our solvers, as well as the data synchronization tools, were helpful to the human planners in identifying and fixing these conflicts gradually.

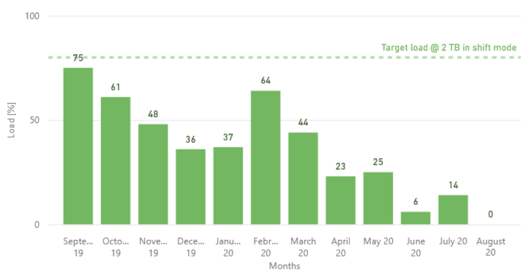
The feedback we got from our industry partner in this phase was enormously valuable for improving our solution algorithms as well as determining areas which could not yet be covered by the TLSP problem definition (see problem extensions in the next section). Eventually, both the solution approaches developed by us and the scheduling tool prototype deployed in the lab become more powerful and could cover a wider range of automated scheduling scenarios. As the confidence in the automatically generated solutions increased, they were directly adopted into the production schedule more and more often. The scheduling workflow described in the previous section, where automatically generated solutions are first transferred and visualized in an external project management software before being uploaded to the database keeps the human planner in the loop and allows them to confirm that the schedules are reasonable and don't contain any unexpected or infeasible results.

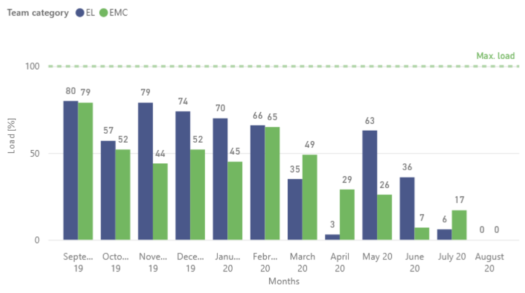
One feature that was very important in that phase was the ability to configure both the solvers and, to a limited extent, the input data directly within the scheduling tool. This allowed the human experts to perform exploratory analysis of different alternatives without having to go through the data synchronization workflow for each variant they wanted to explore.

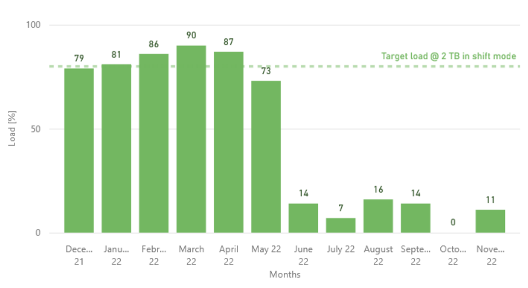
5.3.4 Current situation

At the current stage, scheduling is nearly completely managed by the automated scheduling tool. Whenever changes to the schedule are required, the scheduling system is started with the current data and provides an updated solution. This enables the laboratory to spend significantly less manpower on schedule management compared to other laboratories of similar size (currently 21 employees working on 18 workbenches), while allowing them to provide strict performance guarantees to their clients. In addition, automated scheduling was able to improve the average resource utilization, which can be seen in Figure 5.4, comparing workbench and employee workloads of 2019 (mostly manual scheduling, automated scheduling used for single projects at a time) with that of late 2021. The automated solver was able to find scheduling solutions that make efficient use of the available resources, avoiding costly idle periods.

The exception to automated scheduling are three scenarios which are still handled mostly manually: The first are very small changes where the solution is obvious or easy to find, such as the replacement of a single employee assignment. For these cases, making a manual change to the schedule is quicker and more efficient than going through the whole automated scheduling workflow. The second exception concerns cases where changes are required that simply cannot be done automatically, such as the reduction of a task's duration if it is expected to be finished earlier than planned, the assignment of overtime, or the addition of external staff or equipment for a specific project which could not be performed otherwise. In these cases, the solver does not know about these additionally available options, and they should not be possible in most cases anyway, so they must be scheduled manually. Finally, the third scenario where manual scheduling is still necessary deals with requirements that currently cannot be modeled within the framework of the (extended) TLSP. By now, these cases are rare and mostly concern only specific projects. If any of them turn out to become more important or appear more frequently, it may become necessary to add a further extension to the TLSP that allows the solvers to deal with them automatically. However, we can expect that due to the variety of possible scenarios or requirements, there will always remain rare edge cases that require manual


(a) Workbench utilization in September 2019.


(b) Employee utilization in September 2019.


(c) Workbench utilization in December 2021.

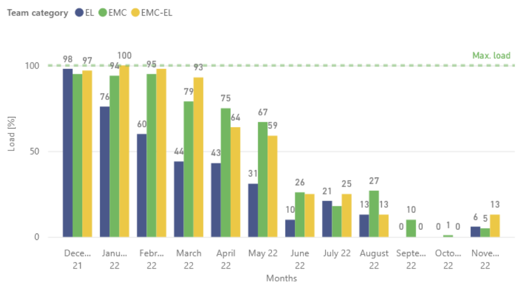

(d) Employee utilization in December 2021.

Figure 5.4: Comparison of resource workloads in the schedules at two different dates, before and after general use of automated scheduling. The plots show average scheduled workloads per month over a period of one year, starting at the given date. The workload of employees is split between employees mostly working on EMC tests and those performing electrical tests (EL). In 2021, there is a third group trained on both types of work.

intervention. Here, it is very important to remain in close contact with the laboratory management to be able to continuously evaluate whether the automated scheduling approaches in use still cover the real-world requirements or whether an adjustment is necessary to deal with any changes.

Currently, the focus of our work with the scheduling system lies on further improving both the efficiency and generality of our solution approaches. In addition, we work together with our industrial partner to integrate the current stand-alone application into the general lab management software. This would allow them to configure and run solvers directly from their standard management interface, as well as improve their capability for exploration of different variants. Finally, we are currently preparing the system for deployment in other test laboratories that have already shown interest in incorporating it into their scheduling workflows.

5.4 Additional objectives and constraints

While the initial problem definition of the TLSP was already quite general and comprehensive, some extensions were still necessary for the successful deployment in practice, for a variety of different reasons: For some aspects, the necessity of being handled by an automated solver only became apparent after some practical experience of working with the scheduling tool could be gathered by the human experts in the lab. Others arose over time due to changing working conditions or customer demands. Finally, some extensions were necessary to prepare the solvers for wider deployment in other laboratories, with their own specific requirements.

The scheduling tool currently supports the following additional objectives over those of the standard TLSP, which can be activated in the constraint configuration:

S6: Makespan Each project should be finished as early as possible, the penalty is equal to the time from the earliest possible start of the project to the end of its last job. The standard TLSP does not use this objective in favor of minimizing the project completion time (counted from the actual start of the first job instead). However, minimizing the makespan is still of interest in many situation, as is evident from its popularity as objective criterion in most other project scheduling problems.

S7: Minimal changes A certain measure of stability in the schedule is often desired, so solvers should not unnecessarily change it each run. This objective compares the assignments of each task's job to its counterpart in the original schedule and penalizes deviations. It can be configured to track different kinds of changes, such as differences in assigned resources, time slots, as well as start and end times of whole projects.

S8: Valid start times Some tasks may have to be started at certain times but can then run on continuously, such as in the case of automated work that should be started by an employee during the week but runs autonomously even over weekends. This objective minimizes the number of jobs that start on a time slot that is not a valid start time for at least one of their tasks. This can also be used as a hard constraint, where each task's job must start on a valid slot.

S9: Preferred modes Similar to preferences over different resource units, we can also define some task modes as preferred over others. If this objective is active, the solver will seek to schedule jobs with more preferred modes. Another use case for this objective is the addition of optional buffer, or reserve, time to deal with unexpected delays or complications. This can be modeled by adding new modes that modify the task durations to include a certain percentage of reserve time. Modes with more reserve time are then preferred over modes including none or only a small percentage of extra time.

All currently deployed solvers support these objectives, in any combination, including their hard constraint version where applicable. For the approaches based on local search, we added them to the delta evaluation performed in the neighborhoods and moves, which did not require any implementation changes in the search heuristics themselves. The CP models are written in a modular form already to allow different constraint configurations, so it is easy to add new modules for these additional constraints and objectives.

5.5 Analysis of objective weights

In previous work and theoretical analysis we have assumed uniform weights for the five standard objectives of TLSP (hereafter called *default TLSP*). For scenarios occurring in practice, these objectives are usually not of equal importance and the weights need to be adapted. In addition, the additional objectives described in the previous section also need to be considered.

5.5.1 Impact on solution times

To test how different objectives affect the runtime of a solver, we conducted experiments on six small instances with 5, 10, and 20 projects, using the exact CP solver. We first evaluated each instance with uniform soft constraint weights. Then, we repeated the evaluation with different configurations where only one soft constraint was active.

When considering only the due date (S4) or preferred employees (S2) objective, the solver always proved optimality within 3 minutes or less. Moreover, a solution with objective value 0 could always be found for each of those two objectives. In fact, this is guaranteed to be always possible due to the way the instances were generated, as described in Section 2.4.1. The number of jobs (S1) objective was always solved to optimality as well, with the solver taking up to 4 minutes. The number of employees (S3) could usually be minimized in a similar time frame, but timed out after 1 hour for one of the large instances with 20 projects.

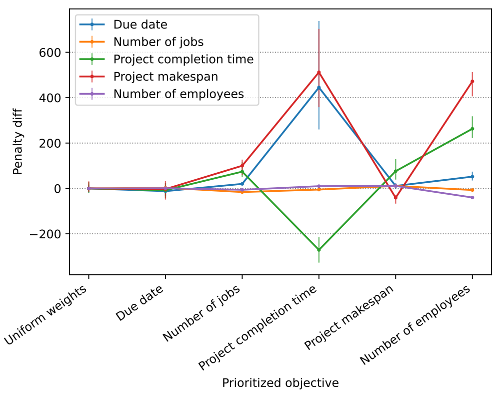
On the other hand, any configuration related to project durations in some manner usually timed out. For 5 of the 6 instances, the solver could not prove optimality within an hour when optimizing the project makespan (S6), project completion time (S5), or under the configuration with uniform weights.

The necessary data required for an analysis of the minimal changes (S7), valid start time (S8), or preferred mode (S9) objectives is not included in the available benchmark instances (they were generated before those extensions were defined).

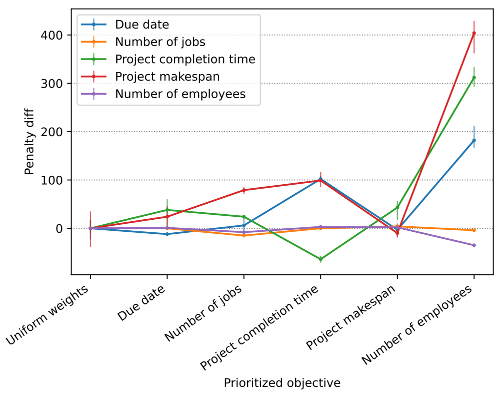
5.5.2 Interaction between objectives

When solving problems with multiple objectives, there is typically a trade off required: Reducing the penalty of one objective results in an increase for the others.

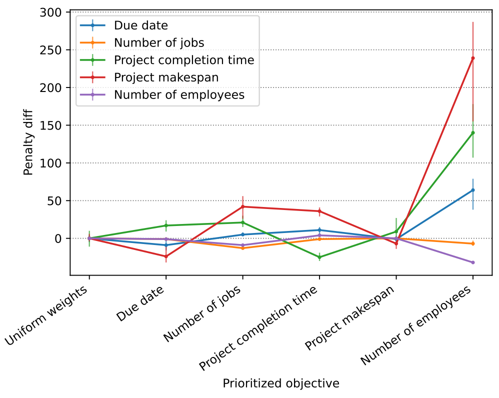
To show the interaction between different soft constraints for large, realistic instances, we performed a different experiment on the 3 real-world instances available. Starting from a baseline where each objective had a uniform weight of 1, we increased the weight of a single objective to 10000. We did so for each soft constraint and compared the results among one another and with the baseline scenario. The preferred employees (S2) objective



(a) Lab1



(b) Lab2



(c) Lab3

Figure 5.5: The effects of increasing the weight of individual objectives to 10000. One graph is provided for each real-world instance we considered. The x-axis represents the objective whose weight was increased. The y-axis tracks the difference between the mean (unweighted) penalties for each objective and those of the baseline configuration ("Uniform weights"). The error bars show the minimum and maximum values over 3 runs. The detailed results are available on Table 5.1.

was not included in this comparison, as it was not supported by the lab environment at the time the instances were created (effectively making all qualified employees preferred for each task). The same is true for S8 and S9. While the real-world instances contain base-schedules that could be used for evaluating S7, the range of configuration options means that the interaction would be completely different depending on the number and type of tracked changes. For this reason, we decided to also leave out S7 in our analysis and focus on the remaining objectives. As a solver, we used VLNS [DGMM20] because of its good performance on large, realistic instances. Each configuration was run 3 times with a timeout of 2 hours.

Inst.	Prioritized objective	S4	S1	S5	S6	S3
Lab1	None (uniform)	199 (14)	274 (1)	3057 (19)	4142 (21)	224 (2)
	Due date (S4)	187 (18)	277 (2)	3049 (26)	4139 (33)	225 (0)
	Number of jobs (S1)	219 (2)	258 (0)	3131 (28)	4242 (30)	218 (2)
	Completion time (S5)	644 (209)	269 (1)	2786 (45)	4654 (143)	234 (3)
	Makespan (S6)	210 (11)	285 (3)	3133 (38)	4101 (18)	235 (3)
	Number of employees (S3)	251 (16)	267 (2)	3320 (40)	4614 (47)	184 (1)
Lab2	None (uniform)	185 (17)	228 (0)	2084 (12)	2532 (30)	188 (0)
	Due date (S4)	173 (2)	228 (3)	2122 (26)	2556 (18)	189 (0)
	Number of jobs (S1)	191 (17)	213 (0)	2108 (2)	2611 (6)	180 (1)
	Completion time (S5)	287 (4)	228 (0)	2020 (5)	2631 (12)	191 (0)
	Makespan (S6)	182 (11)	232 (3)	2127 (18)	2522 (6)	190 (0)
	Number of employees (S3)	367 (20)	224 (1)	2396 (16)	2936 (29)	153 (1)
Lab3	None (uniform)	35 (3)	197 (1)	2220 (8)	2539 (4)	168 (1)
	Due date (S4)	26 (5)	196 (1)	2237 (6)	2515 (11)	167 (1)
	Number of jobs (S1)	40 (2)	184 (0)	2241 (6)	2581 (12)	159 (0)
	Completion time (S5)	46 (4)	196 (1)	2195 (4)	2575 (5)	172 (0)
	Makespan (S6)	34 (3)	197 (1)	2229 (12)	2532 (5)	168 (1)
	Number of employees (S3)	99 (18)	190 (2)	2360 (29)	2778 (59)	136 (1)

Table 5.1: The experimental results for different objective weights. The first column gives the real-world instance that was considered. The second column lists the objective that was given a dominant weight, or "None" when all weights were set to 1. The other columns list the average (unweighted) penalty of each objective over 3 runs, with the standard deviation in parentheses.

The results are listed in Table 5.1 and displayed graphically in Figure 5.5. When comparing the three different instances, it is clear that the same interaction trends become visible in all instances, though the magnitude differs between them. From the table, it also becomes obvious that the objectives can be placed in two groups with different value ranges: Objectives S1 (number of jobs), S3 (number of employees), and S4 (due date) have penalty values below 300 in the configuration with uniform weights, while S5 (completion time) and S6 (makespan) range from 2000 to over 4000. This is expected, as both the latter objectives have to include the total duration of each project. More so than the absolute range of objective values, of interest for optimization and setting weights in practice is rather the change in magnitude due to those weights. Since both objectives have rather high lower bounds individually, they still do not dominate the other objectives, although they do show higher variance between configurations in general.

As expected, the penalty of an objective drops compared to the baseline configuration with uniform weights whenever its weight is increased. The magnitude of this effect depends on the instance and constraint in question. For example, heavily prioritizing the number of project completion time (S5) can decrease the penalty of this objective by up to 200, while optimizing mostly the project makespan (S6) has very little effect overall.

Usually, penalties of most other objectives increase, when a different objective is prioritized. One exception to this is the number of jobs (S1) objective. Focusing on the project completion time (S5) or number of employees (S3) objective can also cause S1 to decrease. This makes sense, because creating fewer jobs results in less setup time added and can make it easier to assign more tasks to the same employees.

Two configurations resulted in the largest overall changes: Prioritizing the project completion time (S5) produced a significant reduction of the penalty for this objective, as remarked earlier, at the cost of a large increase for the makespan (S6) and due date (S4) objectives, particularly for the first two instances. This indicates that the completion time at least for some projects can be reduced by moving it towards the end of its time window. For all three instances, prioritizing the number of employees (S3) resulted in large increases to all time-based objectives. Intuitively, this results from spreading out projects over a longer time period to reduce the amount of parallel work, which requires multiple employees working on the project.

Interestingly, the penalty for the makespan objective (S6) objective is even lower on instance Lab3 when prioritizing due dates (S4) then when it itself is the focus. This may be an artifact due to the limited time available of the solver, since the due date objective is faster to optimize than the makespan, as discussed in the previous section. In general, the penalty for due dates follows that of the makespan objective, which is expected since both profit from projects which finish earlier.

The similarly related objectives for project completion time (S5) and makespan (S6) do not share the same overall direction. Optimizing for completion time significantly reduced it, while increasing project makespans by a comparable and sometimes even larger amount. In contrast, optimizing project makespan seems to have an opposing, but much weaker effect. One potential explanation is that optimizing for project makespans also benefits completion times to a certain degree. On the other hand, many opportunities to improve the completion time of a project may be detrimental to its makespan, if the whole project is shifted to a later point in the schedule. When a project can be scheduled compactly close to its deadline, doing so would essentially maximize its makespan. This could also explain why minimizing the project completion time causes a high number of due date violations.

5.5.3 Selection of weights and configuration in practice

Initially, our industrial partner used uniform weights for all objectives during the first exploratory tests of the system. Over time, the weights have been adjusted several times to better reflect the goals and evolving situation in the lab.

Currently, the focus is put on minimizing the project completion time (S5), at a weight of 100. Experience has shown that schedules often contain gaps between scheduled jobs of a project, which are undesirable for multiple reasons, if the weight of this objective is not high. The number of jobs (S1) and the number of different employees assigned to a project (S3) are minimized with a medium weight of 10. Finally, the preferred employees (S2) and the due date (S4) objectives, together with the new makespan (S6) objective, all have the smallest weight of 1.

The minimization of changes to the base schedule (S7) are limited to avoid only adding additional employees to jobs, also at the lowest weight. Preferred modes (S9) are configured to be used for two different things (via appropriate penalties assigned to the corresponding modes): On the one hand, the solver is instructed to avoid shift work where possible. While shift work is not always avoidable due to the volume of required work and the relative numbers of employees and workbenches, it is both undesirable for employees and expensive. The penalty for shift work is set at the same level as the project completion time (S5), making this a very important goal. On the other hand, flexible levels of reserve time are also handled via preferred modes, where modes with less reserve time have a higher penalty. At its highest (no additional time scheduled at all), this penalty matches the one for shift work, and it decreases quadratically as more reserve time is added, up to a project-dependent maximum. Typically, the solver is configured to only consider modes for a job that do not reduce its current reserve level, though this restriction sometimes has to be lifted for at least some projects to be able to accommodate changes.

A variant of these settings is sometimes used, particularly if a project has to be completely rescheduled after significant delays or changes to the requirements, where the weights of the makespan (S6) and project completion time (S5) objectives are switched, making the solver focus on the former. In these cases, the project's time window is usually very long or open-ended, and the goal is to find the earliest possible time where it could now fit after the delay or changes. Also, most other projects except the existing one are fixed to avoid disruptions due to the changed objective weights.

Yet another configuration is used for the initial scheduling of new projects (either to provide the client with an estimated time window where the project could be performed or to add it to the production schedule after acceptance). Here too many or even all of the existing projects are fixed, in particular projects outside the potential time window of the new project. In this setting, the solver is allowed to freely set reserve levels, in order to fit another project into the schedule. In addition to employee assignments, the minimal changes (S7) objective also tracks delays of the last job in a project and penalizes them, though at a small weight, to discourage the solver from delaying other projects in order to fit the new one.

It may happen that individual projects should be planned with a different set of weights, depending on client wishes and requirements. While the support of such project-specific objective weights is a work in progress, this is currently handled by fixing the remaining projects when (re-)scheduling these projects. Where this is not possible, and more flexibility is required, the minimal changes (S7) can also be used to minimize disruptions to the rest of the schedule due to the modified objective weights.

5.6 Summary

In this chapter, we have provided a description and analysis of the automated scheduling system for the TLSP which is deployed in the test laboratory of our industrial partner and used for the daily management of the laboratory's test schedule.

We have shown how it has successfully improved the quality of their schedules and reduced the effort required to maintain it. Besides, the data management processes necessary to provide consistent and current data for the scheduling tool have resulted in several indirect benefits to the laboratory, such as increasing the confidence in their test plans and providing live monitoring of the tests' status and progress.

The success of the scheduling system in practice hinges on a variety of factors besides the performance of the included solvers: The underlying model must be general enough to represent the varied and dynamic requirements of real-world projects, as well as allow for modification and extensions when those requirements change. The overall scheduling workflow, including data entry and export, must be designed in a way that does not require excessive additional effort. Human planners must have the ability to configure the solver for different usage scenarios, as the desired settings and objective weights change depending on the scenario. Finally, the output of the solver should be in a form that allows the human planners to review and check the solution, which both helps in catching any errors and builds trust in the produced schedules.

Overall, the automated scheduling system developed for the TLSP was successfully deployed in practice and is currently being prepared for use in other test laboratories around the world.

CHAPTER 6

Conclusions

Complex project scheduling problems appear in many real-world settings and automated solution approaches are of tremendous importance for many of them. One such setting is that of industrial test laboratories, which feature a number of unique requirements, which cannot be modeled adequately by existing approaches. In this thesis, we have investigated the use of automated scheduling techniques that are capable of producing high-quality solutions for this challenging environment, as well as other optimization problems.

We have introduced and formally described the NP-hard Test Laboratory Scheduling Problem (TLSP), which extends the MRCPSP with several additional features, including new constraints and objectives. Most importantly, it requires the solvers to group tasks into larger jobs, to streamline the workflow, ensure flexibility during the execution of a schedule, and reduce overheads by avoiding frequent context switches and allowing employees to reuse complex test setups. In addition, resource units are not necessarily identical, due to different qualifications or capabilities, and thus cannot be assigned to all tasks. This requires solvers to also find an explicit assignment of individual units to jobs, instead of only respecting the total capacities as in RCPSP.

For this problem, we have developed an instance generator that creates instances based on the distributions of real-world data and is highly configurable to create random instances with different properties. We have also collected a set of benchmark instances of varying sizes and configurations, including both randomly generated instances and also 3 real-world instances taken directly from the lab of our industrial partner. All benchmark instances are made publicly available for download.

To solve the TLSP as well as practically relevant subproblems, we investigated and compared several different metaheuristics. Towards this end, we developed a set of modular and efficient neighborhood structures to deal with the different aspects of the TLSP, including three new neighborhoods to cover the (re-)grouping of tasks. We then used these neighborhoods within state-of-the-art search heuristics and were able to provide high-quality results, in particular with simulated annealing (SA). Our solver framework provided the basis for later investigations of exact and hybrid solution approaches using CP and a Very Large Neighborhood Search (VLNS), although SA remains the state-of-the-art method for very large instances. Our results show that exact solution techniques (like CP) are very powerful on instances of small and medium size, while heuristic methods become necessary to achieve good results as instance sizes increase.

Since the exact requirements and conditions are likely to vary between different laboratories, we are also interested in general solution techniques that are applicable to a wide range of problem variants and are able to automatically adapt themselves to the current situation. Hyper-heuristics offer just this possibility, as they operate on a set of problem-specific low-level heuristics and can be designed to be otherwise problem-independent. In our investigation, we focused on two different approaches: The first one uses Self-Adaptive Large Neighborhood Search (SA-LNS), which follows the ILS cycle of perturbation-and-repair, based on the success of ILS in previous hyper-heuristics from the literature. Here we investigated different variants to determine the weights of the low-level operators. The second approach models the cross-domain optimization setting as a reinforcement learning (RL) environment, where the low-level operators take the place of the available actions. We performed an extensive analysis of different variants for the components of our RL algorithm and also incorporated a technique to avoid getting stuck in bad areas of the search space by periodically resetting an unsuccessful chain of solutions back to the currently best known solution. We were able to show that this has a significant impact on the success of the search, in particular when using Luby's sequence to determine the length of the solution chains. The empirical evaluation of our hyper-heuristics on six well-known and NP-hard problem domains used in the CHeSC 2011 show that both approaches would have achieved top ranks in this competition, with RL even achieving the second place after the competition winner. While they could not outscore some of the more recent hyper-heuristics on these domains, we were able to significantly improve over the performance of a previous hyper-heuristic approach using RL and show the promise of this paradigm for cross-domain optimization.

We then modeled the TLSP as a problem domain for selection hyper-heuristics by defining a set of low-level heuristics of different types. These low-level heuristics make use of the neighborhood structures and search heuristics we developed for our metaheuristic solution approaches, as well as techniques from the hybrid VLNS. This allowed us to apply our hyper-heuristics as solvers for the TLSP, which were able to successfully provide high-quality solutions despite their problem-independent nature. In fact, a direct comparison with the problem-specific SA and VLNS showed that while those approaches produced slightly better results overall, the hyper-heuristics perform well on a wide range of instances, with less sensitivity to the size and properties of the instance solved. This indicates a high level of generality and is very promising for the performance of the hyper-heuristics on other problem variants.

The solving techniques we developed for the TLSP have also been successfully deployed in the test laboratory of our industrial partner. They are in daily use for the creation and revision of the lab's test schedule as part of a general scheduling and offer management system. Through these automated scheduling solutions we were able to not only drastically reduce the amount of work required to maintain the schedules, but also improve the quality of these schedules and avoid conflicts that lead to delays or missed deadlines. Besides, the need for a consistent and up-to-date database that provides the input data for the solvers has yielded several side benefits, such as faster and more accurate reporting as well as access to additional services and statistics both for the employees within the lab and external clients. In this thesis, we described the scheduling system itself, as well as the history of its deployment, the challenges we encountered during this process, and the lessons we learned from it.

6.1 Future work

While the TLSP is already designed to be general enough to fit the requirements of different industrial test laboratories, it became necessary during the deployment to extend it via additional constraints and objectives, which can be configured and combined dynamically. As now the deployment of our algorithms also in other laboratories is being discussed, it is likely that further extensions will be needed to best cover their unique requirements. A major topic of future research in this area will be the definition of new constraints and objectives, as well as their incorporation within the existing TLSP models.

In addition, it will also be interesting to consider new aspects of scheduling, such as robust and reactive scheduling, to increase the capabilities of schedules to either remain feasible or be easily repaired even in the face of significant delays or disruptions. Another aspect would be the investigation of explainable scheduling, which enables the solver to explain to the human planner why it made certain decisions or why certain conflicts could not be eliminated. In this context, also an analysis of alternate scenarios would be of interest, to provide the laboratory management with insights on administrative changes they could make to the laboratory environment in order to increase the quality of the schedule, e.g. by increasing the number of available resources or providing training that allows employees to perform more different types of tests.

All these extensions and new features will also require changes and modifications to the solvers, or even completely new solution approaches. Our results on hyper-heuristics have already shown some promise to be applicable to many different problem variants, so a focus on these methods seems certainly warranted. In particular with the addition of new constraints and objectives, which will not be present in all laboratories and may be dynamically added, configured, or removed from a certain problem instance, the adaptivity of hyper-heuristics will be very important in ensuring that high-quality solutions can be found in all these situations. The setting of the cross-domain optimization problem poses a strict barrier between the hyper-heuristic and domain-specific knowledge. We would like to investigate how we could further improve our hyper-heuristics by allowing access to some of this information, such as the presence or absence of certain constraints and objectives or characteristics of the instance to be solved. This would provide valuable insights for the scheduling community on general and self-adaptive solution methods.

List of Figures

List of Tables

List of Algorithms

Bibliography

[ABN14] Steven Adriaensen, Tim Brys, and Ann Nowé. Fair-share ils: A simple state-of-the-art iterated local search hyperheuristic. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, GECCO '14, page 1303–1310, New York, NY, USA, 2014. Association for Computing Machinery.

[AK89] E.H.L. Aarts and J.H.M. Korst. *Simulated annealing and Boltzmann machines: a stochastic approach to combinatorial optimization and neural computing.* Wiley-Interscience series in discrete mathematics and optimization. Wiley, United States, 1989.

[AKK$^+$16] Shahriar Asta, Daniel Karapetyan, Ahmed Kheiri, Ender Özcan, and Andrew J. Parkes. Combining monte-carlo and hyper-heuristic methods for the multi-mode resource-constrained multi-project scheduling problem. *Information Sciences*, 373:476 – 498, 2016.

[AM01] Javier Alcaraz and Concepción Maroto. A robust genetic algorithm for resource allocation in project scheduling. *Annals of operations Research*, 102(1):83–109, 2001.

[AM18] Arben Ahmeti and Nysret Musliu. Min-conflicts heuristic for multi-mode resource-constrained projects scheduling. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 237–244. ACM, 2018.

[AM21] Arben Ahmeti and Nysret Musliu. Hybridizing constraint programming and meta-heuristics for multi-mode resource-constrained multiple projects scheduling problem. In *Proceedings of the 13th International Conference on the Practice and Theory of Automated Timetabling-PATAT*, volume 1, 2021.

[AN16] Steven Adriaensen and Ann Nowé. Case study: An analysis of accidental complexity in a state-of-the-art hyper-heuristic for hyflex. In *2016 IEEE Congress on Evolutionary Computation (CEC)*, pages 1485–1492. IEEE, 2016.

[AÖ15] Shahriar Asta and Ender Özcan. A tensor-based selection hyper-heuristic for cross-domain heuristic search. *Information Sciences*, 299:412–432, 2015.

[AON15] Steven Adriaensen, Gabriela Ochoa, and Ann Now'e. A benchmark set extension and comparative study for the hyflex framework. In *Evolutionary Computation (CEC), 2015 IEEE Congress on*. IEEE, 2015.

[AOO21] Stephen A. Adubi, Olufunke O. Oladipupo, and Oludayo O. Olugbara. Configuring the perturbation operations of an iterated local search algorithm for cross-domain search: A probabilistic learning approach. In *2021 IEEE Congress on Evolutionary Computation (CEC)*, pages 1372–1379, 2021.

[BB15] Francisco Ballestín and Rosa Blanco. *Theoretical and Practical Fundamentals*, pages 411–427. Springer International Publishing, Cham, 2015.

[BDM$^+$99] Peter Brucker, Andreas Drexl, Rolf Möhring, Klaus Neumann, and Erwin Pesch. Resource-constrained project scheduling: Notation, classification, models, and methods. *European Journal of Operational Research*, 112(1):3 – 41, 1999.

[BGH$^+$11] Edmund K. Burke, Michel Gendreau, Matthew Hyde, Graham Kendall, Barry McCollum, Gabriela Ochoa, Andrew J. Parkes, and Sanja Petrovic. The cross-domain heuristic search challenge – an international research competition. In Carlos A. Coello Coello, editor, *Learning and Intelligent Optimization*, pages 631–634, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[BGH$^+$13] Edmund K Burke, Michel Gendreau, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and Rong Qu. Hyper-heuristics: A survey of the state of the art. *Journal of the Operational Research Society*, 64(12):1695–1724, 2013.

[BHK$^+$10] Edmund K. Burke, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and John R. Woodward. *A Classification of Hyper-heuristic Approaches*, pages 449–468. Springer US, Boston, MA, 2010.

[BHK$^+$19] Edmund K. Burke, Matthew R. Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and John R. Woodward. *A Classification of Hyper-Heuristic Approaches: Revisited*, pages 453–477. Springer International Publishing, Cham, 2019.

[BL03] K. Bouleimen and H. Lecocq. A new efficient simulated annealing algorithm for the resource-constrained project scheduling problem and its multiple mode version. *European Journal of Operational Research*, 149(2):268 – 281, 2003. Sequencing and Scheduling.

[BLK83] Jacek Blazewicz, Jan Karel Lenstra, and AHG Rinnooy Kan. Scheduling subject to resource constraints: classification and complexity. *Discrete applied mathematics*, 5(1):11–24, 1983.

[BN05] Odile Bellenguez and Emmanuel Néron. Lower bounds for the multi-skill project scheduling problem with hierarchical levels of skills. In Edmund Burke and Michael Trick, editors, *Practice and Theory of Automated Timetabling V*, pages 229–243, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[BW91] Graham Brightwell and Peter Winkler. Counting linear extensions is #p-complete. In *Proceedings of the twenty-third annual ACM symposium on Theory of computing*, pages 175–181, 1991.

[BZ09] J.-H. Bartels and J. Zimmermann. Scheduling tests in automotive r&d projects. *European Journal of Operational Research*, 193(3):805 – 819, 2009.

[Chu11] Geoffrey Chu. *Improving combinatorial optimization.* PhD thesis, University of Melbourne, Australia, 2011.

[Chu20] Chung-Yao Chuang. *Combining Multiple Heuristics: Studies on Neighborhood-base Heuristics and Sampling-based Heuristics.* PhD thesis, Carnegie Mellon University, 2020.

[CKS01] Peter Cowling, Graham Kendall, and Eric Soubeiga. A hyperheuristic approach to scheduling a sales summit. In Edmund Burke and Wilhelm Erben, editors, *Practice and Theory of Automated Timetabling III*, pages 176–190, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[CS17] Chung-Yao Chuang and Stephen F Smith. A study of agnostic hyper-heuristics based on sampling solution chains. In *2017 IEEE Congress on Evolutionary Computation (CEC)*, pages 271–278. IEEE, 2017.

[CSE18] Ripon K. Chakrabortty, Ruhul Sarker, and Daryl L. Essam. A comparative study of different integer linear programming approaches for resource-constrained project scheduling problems. In Ruhul Sarker, Hussein A. Abbass, Simon Dunstall, Philip Kilby, Richard Davis, and Leon Young, editors, *Data and Decision Sciences in Action*, pages 227–242, Cham, 2018. Springer International Publishing.

[CWL18] Shin Siang Choong, Li-Pei Wong, and Chee Peng Lim. Automatic design of hyper-heuristic based on reinforcement learning. *Information Sciences*, 436-437:89–107, 2018.

[CXIC12] C. Y. Chan, Fan Xue, W. H. Ip, and C. F. Cheung. A hyper-heuristic inspired by pearl hunting. In Youssef Hamadi and Marc Schoenauer, editors, *Learning and Intelligent Optimization*, pages 349–353, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[DB08] L.-E. Drezet and J.-C. Billaut. A project scheduling problem with labour constraints and time-dependent activities requirements. *International Journal of Production Economics*, 112(1):217 – 225, 2008. Special Section on Recent Developments in the Design, Control, Planning and Scheduling of Productive Systems.

[DGJ$^+$21] Philipp Danzinger, Tobias Geibinger, David Janneau, Florian Mischek, Nysret Musliu, and Christian Poschalko. A system for automated industrial test laboratory scheduling. *accepted for publication at ACM Transactions on Intelligent Systems and Technology*, 2021.

[DGMM20] Philipp Danzinger, Tobias Geibinger, Florian Mischek, and Nysret Musliu. Solving the test laboratory scheduling problem with variable task grouping. In *International Conference on Planning and Scheduling (ICAPS) 2020*, 2020.

[DGU12] Luca Di Gaspero and Tommaso Urli. Evaluation of a family of reinforcement learning cross-domain optimization heuristics. In Youssef Hamadi and Marc Schoenauer, editors, *Learning and Intelligent Optimization*, pages 384–389, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[DK01] Andreas Drexl and Alf Kimms. Optimization guided lower and upper bounds for the resource investment problem. *Journal of the Operational Research Society*, 52(3):340–351, 2001.

[DNPS00] Andreas Drexl, Rüdiger Nissen, James H. Patterson, and Frank Salewski. Progen/πx – an instance generator for resource-constrained project scheduling problems with partially renewable resources and further extensions. *European Journal of Operational Research*, 125(1):59 – 72, 2000.

[DPRL98] S. Dauzère-Pérès, W. Roux, and J.B. Lasserre. Multi-resource shop scheduling with resource flexibility. *European Journal of Operational Research*, 107(2):289 – 305, 1998.

[Elm77] Salah Eldin Elmaghraby. *Activity networks: Project planning and control by network models.* John Wiley & Sons, 1977.

[FGTRP15] Alexandre Silvestre Ferreira, Richard Aderbal Goncalves, and Aurora Trinidad Ramirez Pozo. A multi-armed bandit hyper-heuristic. In *2015 Brazilian Conference on Intelligent Systems (BRACIS)*, pages 13–18, 2015.

[Gei20] Tobias Geibinger. Investigating constraint programming and hybrid answer-set solving for industrial test laboratory scheduling. Master's thesis, TU Wien, 2020.

[GKK$^+$21] Tobias Geibinger, Lucas Kletzander, Matthias Krainz, Florian Mischek, Nysret Musliu, and Felix Winter. Physician scheduling during a pandemic. In Peter J. Stuckey, editor, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 456–465, Cham, 2021. Springer International Publishing.

[GMM19a] Tobias Geibinger, Florian Mischek, and Nysret Musliu. Investigating constraint programming and hybrid methods for real world industrial test laboratory scheduling. *submitted to Journal of Scheduling*, 2019.

[GMM19b] Tobias Geibinger, Florian Mischek, and Nysret Musliu. Investigating constraint programming for real world industrial test laboratory scheduling. In *Proceedings of the Sixteenth International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR 2019)*, 2019.

[GMM21] Tobias Geibinger, Florian Mischek, and Nysret Musliu. Constraint logic programming for real-world test laboratory scheduling. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(7):6358–6366, May 2021.

[GMR08] J.F. Gonçalves, J.J.M. Mendes, and M.G.C. Resende. A genetic algorithm for the resource constrained multi-project scheduling problem. *European Journal of Operational Research*, 189(3):1171 – 1190, 2008.

[Har98] Sönke Hartmann. A competitive genetic algorithm for resource-constrained project scheduling. *Naval Research Logistics (NRL)*, 45(7):733–750, 1998.

[HB10] Sönke Hartmann and Dirk Briskorn. A survey of variants and extensions of the resource-constrained project scheduling problem. *European Journal of Operational Research*, 207(1):1 – 14, 2010.

[HB22] Sönke Hartmann and Dirk Briskorn. An updated survey of variants and extensions of the resource-constrained project scheduling problem. *European Journal of Operational Research*, 297(1):1–14, 2022.

[HBS18] Farhad Habibi, Farnaz Barzinpour, and S Sadjadi. Resource-constrained project scheduling problem: review of past and recent developments. *Journal of Project Management*, 3(2):55–88, 2018.

[HCF12] Ping-Che Hsiao, Tsung-Che Chiang, and Li-Chen Fu. A vns-based hyper-heuristic with adaptive computational budget of local search. In *2012 IEEE Congress on Evolutionary Computation*, pages 1–8, 2012.

[Her05] Willy Herroelen. Project scheduling—theory and practice. *Production and operations management*, 14(4):413–432, 2005.

[HHLB11] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International Conference on Learning and Intelligent Optimization*, pages 507–523. Springer, 2011.

[HRD98] Willy Herroelen, Bert De Reyck, and Erik Demeulemeester. Resource-constrained project scheduling: A survey of recent developments. *Computers & Operations Research*, 25(4):279 – 302, 1998.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[KH99] Rainer Kolisch and Sönke Hartmann. *Heuristic Algorithms for the Resource-Constrained Project Scheduling Problem: Classification and Computational Analysis*, pages 147–178. Springer US, Boston, MA, 1999.

[KH06] Rainer Kolisch and Sönke Hartmann. Experimental investigation of heuristics for resource-constrained project scheduling: An update. *European Journal of Operational Research*, 174(1):23–37, 2006.

[KKA14] Georgios Koulinas, Lazaros Kotsikas, and Konstantinos Anagnostopoulos. A particle swarm optimization based hyper-heuristic algorithm for the classic resource constrained project scheduling problem. *Information Sciences*, 277:680–693, 2014.

[KP01] R Kolisch and R Padman. An integrated survey of deterministic project scheduling. *Omega*, 29(3):249–272, 2001.

[Lar11] Mathieu Larose. A hyper-heuristic for the chesc 2011. Report submitted for the CHeSC competition, 2011.

[LDGN17] A. Laurent, L. Deroussi, N. Grangeon, and S. Norre. A new extension of the rcpsp in a multi-site context: Mathematical model and metaheuristics. *Computers & Industrial Engineering*, 112:634 – 644, 2017.

[LG07] Philippe Laborie and Daniel Godard. Self-adapting large neighborhood search: Application to single-mode scheduling problems. *Proceedings MISTA-07, Paris*, 8, 2007.

[LMS03] Helena R Lourenço, Olivier C Martin, and Thomas Stützle. Iterated local search. In *Handbook of metaheuristics*, pages 320–353. Springer, 2003.

[LSZ93] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of las vegas algorithms. *Information Processing Letters*, 47(4):173–180, 1993.

[MGR12] Arnaud Malapert, Christelle Guéret, and Louis-Martin Rousseau. A constraint programming approach for a batch processing problem with non-identical job sizes. *European Journal of Operational Research*, 221(3):533–545, 2012.

[MJPL92] Steven Minton, Mark D. Johnston, Andrew B. Philips, and Philip Laird. Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58:161–205, 1992.

[MKC10] David Meignan, Abderrafiaa Koukam, and Jean-Charles Créput. Coalition-based metaheuristic: a self-adaptive metaheuristic using reinforcement learning and mimetism. *Journal of Heuristics*, 16(6):859–879, 2010.

[MM18] Florian Mischek and Nysret Musliu. A local search framework for industrial test laboratory scheduling. In *Proceedings of the 12th International Conference on the Practice and Theory of Automated Timetabling (PATAT-2018), Vienna, Austria, August 28–31, 2018*, pages 465–467, 2018.

[MM21a] Florian Mischek and Nysret Musliu. A collection of hyper-heuristics for the hyflex framework. Technical report, TU Wien, 2021.

[MM21b] Florian Mischek and Nysret Musliu. A local search framework for industrial test laboratory scheduling. *Annals of Operations Research*, 302:533–562, 2021.

[MM22a] Florian Mischek and Nysret Musliu. Investigating hyper-heuristics for real-world test laboratory scheduling. In *EURO 2022: Conference handbook and abstracts: 32nd European Conference on Operational Research (EURO XXXII), Helsinki, Finland, July 3 – 6, 2022*, 2022.

[MM22b] Florian Mischek and Nysret Musliu. Reinforcement learning for cross-domain hyper-heuristics. *accepted for presentation at IJCAI 2022*, 2022.

[MMS02] Daniel Merkle, Martin Middendorf, and Hartmut Schmeck. Ant colony optimization for resource-constrained project scheduling. *IEEE transactions on evolutionary computation*, 6(4):333–346, 2002.

[MMS21a] Florian Mischek, Nysret Musliu, and Andrea Schaerf. Local search approaches for the test laboratory scheduling problem with variable task grouping. *Journal of Scheduling*, 2021.

[MMS21b] Florian Mischek, Nysret Musliu, and Andrea Schaerf. Local search neighborhoods for industrial test laboratory scheduling with flexible grouping. In *accepted for publication in the Proceedings of the 13th International Conference on the Practice and Theory of Automated Timetabling (PATAT-2021)*, 2021.

[MSOO15] Paweł B Myszkowski, Marek E Skowroński, Łukasz P Olech, and Krzysztof Oślizło. Hybrid ant colony optimization in solving multi-skill resource-constrained project scheduling problem. *Soft Computing*, 19(12):3599–3619, 2015.

[Mus05] Nysret Musliu. Combination of local search strategies for rotating workforce scheduling problem. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005*, pages 1529–1530. Professional Book Center, 2005.

[MVDCVB12] Mustafa Mısır, Katja Verbeeck, Patrick De Causmaecker, and Greet Vanden Berghe. An intelligent hyper-heuristic framework for chesc 2011. In Youssef Hamadi and Marc Schoenauer, editors, *Learning and Intelligent Optimization*, pages 461–466, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[MWW06] Marek Mika, Grzegorz Waligóra, and Jan Węglarz. Modelling setup times in project scheduling. *Perspectives in modern project scheduling*, pages 131–163, 2006.

[MWW15] Marek Mika, Grzegorz Waligóra, and Jan Węglarz. Overview and state of the art. In Christoph Schwindt and Jürgen Zimmermann, editors, *Handbook on Project Management and Scheduling Vol.1*, pages 445–490. Springer International Publishing, Cham, 2015.

[NI02] Koji Nonobe and Toshihide Ibaraki. Formulation and tabu search algorithm for the resource constrained project scheduling problem. In *Essays and Surveys in Metaheuristics*, pages 557–588. Springer US, Boston, MA, 2002.

[NR97] Nudtapon Nudtasomboon and Sabah U. Randhawa. Resource-constrained project scheduling with renewable and non-renewable resources and time-resource tradeoffs. *Computers & Industrial Engineering*, 32(1):227–242, 1997.

[NSB$^+$07] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard CP modelling language. In *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, pages 529–543, 2007.

[NT18] Siamak Noori and Kaveh Taghizadeh. Multi-mode resource constrained project scheduling problem: a survey of variants, extensions, and methods. *International Journal of Industiral Engineering & Producion Research*, 29(3), 2018.

[OHC+12] Gabriela Ochoa, Matthew Hyde, Tim Curtois, Jose A. Vazquez-Rodriguez, James Walker, Michel Gendreau, Graham Kendall, Barry McCollum, Andrew J. Parkes, Sanja Petrovic, and Edmund K. Burke. Hyflex: A benchmark framework for cross-domain heuristic search. In Jin-Kao Hao and Martin Middendorf, editors, *Evolutionary Computation in Combinatorial Optimization*, pages 136–147, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[OWHC12] Gabriela Ochoa, James Walker, Matthew Hyde, and Tim Curtois. Adaptive evolutionary algorithms and extensions to the hyflex hyper-heuristic framework. In *International Conference on Parallel Problem Solving from Nature*, pages 418–427. Springer, 2012.

[PK00] Chris N. Potts and Mikhail Y. Kovalyov. Scheduling with batching: A review. *European Journal of Operational Research*, 120(2):228 – 249, 2000.

[PMAAL17] Oliver Polo Mejia, Marie-Christine Anselmet, Christian Artigues, and Pierre Lopez. A new RCPSP variant for scheduling research activities in a nuclear laboratory. In *47th International Conference on Computers & Industrial Engineering (CIE47)*, page 8p., Lisbonne, Portugal, October 2017.

[PPB20] Robert Pellerin, Nathalie Perrier, and Francois Berthaut. A survey of hybrid metaheuristics for the resource-constrained project scheduling problem. *European Journal of Operational Research*, 280(2):395–416, 2020.

[PV14] Vincent Van Peteghem and Mario Vanhoucke. An experimental investigation of metaheuristics for the multi-mode resource-constrained project scheduling problem on new dataset instances. *European Journal of Operational Research*, 235(1):62 – 72, 2014.

[PWW69] A. Alan B. Pritsker, Lawrence J. Waiters, and Philip M. Wolfe. Multiproject scheduling with limited resources: A zero-one programming approach. *Management Science*, 16(1):93–108, 1969.

[SAKQ15] Nasser R. Sabar, Masri Ayob, Graham Kendall, and Rong Qu. Automatic design of a hyper-heuristic framework with gene expression programming for combinatorial optimization problems. *IEEE Transactions on Evolutionary Computation*, 19(3):309–325, 2015.

[SB18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* MIT press, 2018.

[Sca20] Jeff Scales. A design science research approach to closing the gap between the research and practice of project scheduling. *Systems Research and Behavioral Science*, 37(5):804–812, 2020.

[Sch96] J.M.J. Schutten. List scheduling revisited. *Operations Research Letters*, 18(4):167–170, 1996.

[SK15] Nasser R. Sabar and Graham Kendall. Population based monte carlo tree search hyper-heuristic for combinatorial optimization problems. *Information Sciences*, 314:225–239, 2015.

[SS16] Ria Szeredi and Andreas Schutt. Modelling and solving multi-mode resource-constrained project scheduling. In *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, pages 483–492, 2016.

[SSD97] Frank Salewski, Andreas Schirmer, and Andreas Drexl. Project scheduling under resource and mode identity constraints: Model, complexity, methods, and application. *European Journal of Operational Research*, 102(1):88 – 110, 1997.

[ST00] Christoph Schwindt and Norbert Trautmann. Batch scheduling in process industries: an application of resource–constrained project scheduling. *OR-Spektrum*, 22(4):501–524, 2000.

[TB20] Tanya Y. Tang and J. Christopher Beck. Cp and hybrid models for two-stage batching and scheduling. In *Proceedings of the 17th International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR 2020)*, volume 12296 of *LNCS*, pages 431–446. Springer, 2020.

[TS98] Paul R Thomas and Said Salhi. A tabu search approach for the resource constrained project scheduling problem. *Journal of Heuristics*, 4(2):123–139, 1998.

[TS05] Norbert Trautmann and Christoph Schwindt. A minlp/rcpsp decomposition approach for the short-term planning of batch production. In *Computer Aided Chemical Engineering*, volume 20, pages 1309–1314. Elsevier, 2005.

[TS18] Charles Thomas and Pierre Schaus. Revisiting the self-adaptive large neighborhood search. In Willem-Jan van Hoeve, editor, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 557–566, Cham, 2018. Springer International Publishing.

[VP00] Ana Viana and Jorge Pinho de Sousa. Using metaheuristics in multiobjective resource constrained project scheduling. *European Journal of Operational Research*, 120(2):359–374, 2000.

[VPLP+19] Félix Villafáñez, David Poza, Adolfo López-Paredes, Javier Pajares, and Ricardo del Olmo. A generic heuristic for multi-project scheduling problems with global and local resource constraints (rcmpsp). *Soft Computing*, 23(10):3465–3479, 2019.

[VW07] Stefan Voß and Andreas Witt. Hybrid flow shop scheduling as a multi-mode multi-project scheduling problem with batching requirements: A real-world application. *International Journal of Production Economics*, 105(2):445–458, 2007. Scheduling in batch-processing industries and supply chains.

[Wil03] Terry Williams. The contribution of mathematical modelling to the practice of project management. *IMA Journal of Management Mathematics*, 14(1):3–30, 2003.

[WJMW11] Jan Węglarz, Joanna Józefowska, Marek Mika, and Grzegorz Waligóra. Project scheduling with finite or infinite number of activity processing modes – a survey. *European Journal of Operational Research*, 208(3):177 – 205, 2011.

[WKS+16] Tony Wauters, Joris Kinable, Pieter Smet, Wim Vancroonenburg, Greet Vanden Berghe, and Jannes Verstichel. The multi-mode resource-constrained multi-project scheduling problem. *Journal of Scheduling*, 19(3):271–283, Jun 2016.

[WMMP21] Felix Winter, Christoph Mrkvicka, Nysret Musliu, and Jakob Preininger. Automated production scheduling for artificial teeth manufacturing. *Proceedings of the International Conference on Automated Planning and Scheduling*, 31(1):500–508, May 2021.

[WWH12] Michel Wilson, Cees Witteveen, and Bob Huisman. Enhancing predictability of schedules by task grouping. In *BNAIC 2012: 24th Benelux Conference on Artificial Intelligence, Maastricht, The Netherlands, 25-26 October 2012*. Citeseer, 2012.

[YA13] Virginia Yannibelli and Analía Amandi. Hybridizing a multi-objective simulated annealing algorithm with a multi-objective evolutionary algorithm to solve a multi-objective project scheduling problem. *Expert Systems with Applications*, 40(7):2421–2434, 2013.

[YFS17] Kenneth D. Young, Thibaut Feydy, and Andreas Schutt. Constraint programming applied to the multi-skill project scheduling problem. In J. Christopher Beck, editor, *Principles and Practice of Constraint Programming*, pages 308–317, Cham, 2017. Springer International Publishing.