FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

A Heuristic Solver Framework for the General Employee Scheduling Problem

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Lucas Kletzander, BSc
Matrikelnummer 01225758

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Priv.-Doz. Dr. Nysret Musliu

Wien, 24. Jänner 2018

Lucas Kletzander	Nysret Musliu

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

A Heuristic Solver Framework for the General Employee Scheduling Problem

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computational Intelligence

by

Lucas Kletzander, BSc
Registration Number 01225758

to the Faculty of Informatics

at the TU Wien

Advisor: Priv.-Doz. Dr. Nysret Musliu

Vienna, 24th January, 2018

Lucas Kletzander	Nysret Musliu

Erklärung zur Verfassung der Arbeit

Lucas Kletzander, BSc
Julius Raab Promenade 10/12, 3100 St. Pölten

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, 24. Jänner 2018

Lucas Kletzander

Acknowledgements

I deeply want to thank my advisor, Priv.-Doz. Dr. Nysret Musliu for his support and his valuable feedback.

Further I want to thank my family and friends for their support and guidance.

This work was supported by the Austrian Science Fund (project: P24814-N23)

Kurzfassung

In vielen Berufen ist es erforderlich, in verschiedenen Schichten zu arbeiten, um unterschiedliche Anforderungen abzudecken. Dazu zählen Bereiche im Gesundheitswesen, in Sicherheitsdiensten, im Transportwesen, in der Produktion oder in Callcentern. Dabei muss eine Vielzahl an Bedingungen erfüllt werden um gültige Schichtpläne zu gestalten. Die Anforderungen können auf unterschiedliche Arten festgelegt werden, diverse Gesetze müssen eingehalten werden und die Zufriedenheit der Angestellten muss berücksichtigt werden. Damit ist es nicht nur zunehmend schwerer, derartige Pläne manuell zu erstellen, umso mehr Angestellte und Bedingungen zu berücksichtigen sind, sondern auch sehr zeitraubend. Somit sind automatisierte Lösungen notwendig, um im Wettbewerb zu bestehen. Allerdings ist es auch hier schwer, in annehmbarer Zeit gute Lösungen zu erreichen, da viele dieser Probleme NP-schwer sind.

Während nicht in jedem Problem alle erdenklichen Bedingungen zu berücksichtigen sind, ist es mühsam, jeweils eine neue Formulierung und eine entsprechende Lösungsmethode zu entwickeln. Diese können dann oft nur schwer auf ähnliche Probleme übertragen werden. Auf der anderen Seite ist es eine große Herausforderung, eine allgemeine Formulierung und dazu passende Lösungsmethoden zu entwickeln, da schon zahlreiche Teilprobleme alleine NP-schwer sind.

Somit ist die erste Aufgabe, einen Beitrag zur Formulierung des General Employee Scheduling (GES) Problems zu leisten, die eine große Bandbreite an derartigen Personalplanungsproblemen darstellen kann. Eine umfassende Literatursuche wird ausgeführt, um zahlreiche verschiedene Probleme dieser Art in der Formulierung abzubilden.

Der Hauptbeitrag dieser Arbeit ist die Entwicklung eines neuen Frameworks für das GES-Problem, mit dem verschiedene heuristische Lösungsmethoden implementiert und auf verschiedene Probleme angewandt werden können. Dies wird umgesetzt, indem ein einheitlicher Umgang mit Bedingungen und die Möglichkeit für die Implementierung verschiedener Nachbarschaften geschaffen wird, die dann in unterschiedlichen Algorithmen wiederverwendet werden können. Weiter wird eine neue Suchmethode entwickelt und in diesem Framework implementiert. Ein Generator für neue Instanzen wird bereitgestellt, um Anforderungen und Bedingungen in neuen Arten zu kombinieren, die in der Literatur noch nicht untersucht werden, und damit neue Benchmark-Instanzen zu erhalten.

Um die Anwendbarkeit für eine Vielzahl von Problemen zu zeigen, nehmen wir unterschiedliche Probleme aus der Literatur, die unterschiedliche Anforderungsarten und

Bedingungen verwenden, übersetzen diese in unsere Formulierung und wenden unsere Lösungsmethode auf diese Instanzen und unsere eigenen Instanzen an.

Die Ergebnisse zeigen, dass zahlreiche Probleme aus unterschiedlichen Bereichen der Personalzeitplanung in unserer Formulierung dargestellt werden können und das Framework bei diesen Problemen erfolgreich angewandt werden kann. Der Vergleich mit den Ergebnissen aus der Literatur zeigt, dass der implementierte allgemeine Algorithmus gute Ergebnisse für die meisten Instanzen aus diesen Problemen liefert und eine gute Basis für die Entwicklung von spezialisierten Algorithmen in unserem Framework bildet.

Abstract

In many professions the demand for work requires employees to work in different shifts to cover varying requirements including areas like health care, protection services, transportation, manufacturing or call centers. However, there are many constraints that need to be satisfied in order to create feasible schedules. The demands can be specified in various ways, different legal requirements need to be respected and employee satisfaction has to be taken into account. Not only is it increasingly difficult to generate schedules by hand for more employees and more requirements, it is also very time consuming. Therefore, automated solutions are mandatory to stay competitive. However, even then it is often hard to provide good solutions in reasonable time as many of the problems are NP-hard.

While not each problem will require the whole set of available restrictions, it is cumbersome to develop a new specification format and corresponding solver for each problem. Often these can not be well applied to similar problems differing in some requirements. On the other hand it is a challenging task to provide a general formulation and solution methods that can solve large integrated problems, as even several sub-problems on their own are known to be NP-hard.

Therefore, the first objective is to give a contribution to the formulation of the General Employee Scheduling (GES) problem that can be used to specify a wide range of such scheduling problems. An extensive literature review is conducted to determine a wide range of employee scheduling problems in order to cover them in the GES formulation.

The main contribution of this thesis is the development of a new framework for the general employee scheduling problem that allows the implementation of various heuristic algorithms and their application to a wide range of problems. This is realized by proposing a unified handling of constraints and the possibility to implement various moves that can be reused across different algorithms. Further, a new search method is developed and implemented in the framework. An instance generator is provided that can combine the demands and constraints in ways that are not yet covered by literature to provide new benchmark instances.

In order to show the applicability to a wide range of problems, we take different problems from literature that cover different types of demand and constraints, translate their instances to our formulation and apply our solver to those instances as well as our own instances.

The results show that several problems from different areas of employee scheduling can be modelled in our formulation and the framework can successfully be applied to all of them. The comparison with the results from literature shows that the implemented general purpose algorithm can provide good results for most instances across all problems and provides a good foundation for the development of more specialized algorithms in the framework.

Contents

CHAPTER 1

Introduction

In many professions the demand for work requires employees to work in different shifts to cover varying requirements including areas like health care, protection services, transportation, manufacturing or call centers. However, this problem can come in many shapes [VdBBDB+13, EJKS04]. The demand might be to assign employees to certain shifts that are already fixed like in nurse rostering. It might also be necessary to design shifts in a way that there is always a certain number of employees present. Sometimes tasks are given and the shifts have to be designed to cover these tasks.

On the other hand shifts can not be assigned freely. Legal requirements can be very strict in demanding times between shifts, certain patterns or sequences of shifts or days off that are required or forbidden and much more. Employees might have different contracts that might specify very differing requirements for each employee. On some occasions it might also be necessary to schedule breaks as well in order to guarantee that still enough employees are available for duty.

Further, the employees themselves often specify their own requests like days they would like to have on or off, shifts they want to avoid or other employees they want to work with or avoid. There might also be measurements of fairness between employees that need to be considered. In order to increase employee satisfaction it is important to include such wishes as well.

To reduce cost and maximize effectiveness, companies want to find schedules that cover all the demands in an effective way. Ineffective scheduling might require the hiring of temporary employees that increase the cost, while schedules that do not respect all the legal constraints can lead to penalties and employee dissatisfaction. Not only is it increasingly difficult to generate schedules by hand for more employees and more requirements, it is also very time consuming. Therefore, automated solutions are mandatory to stay competitive. However, even then it is often hard to provide good solutions in reasonable time as many of the problems are NP-hard.

Many different problems that are NP-hard and different approaches to get solutions to those problems are described in literature. These problems can include task scheduling, break scheduling, shift scheduling, rostering any much more. Relevant reviews are [VdBBDB+13, EJKS04] for different problem variants, classifications and solution methods, [DBVdBBD15] for inclusion of skills, [Alf04] for tour scheduling or [BDCBVL04] for nurse rostering.

While not each problem will require the whole set of available restrictions, it is cumbersome to develop a new specification and corresponding solver for each version. Often these can not be well applied to similar problems differing in some requirements. Therefore, it would be highly beneficial to have a framework suitable for application on various problems without the need to design a new formulation from scratch. On the other hand it is a challenging task to provide a general formulation and solution methods that can solve large integrated problems, as even several sub-problems on their own are known to be NP-hard.

1.1 Aim of this Thesis

Therefore the first aim of this thesis is to provide a contribution to the formulation of the General Employee Scheduling (GES) problem that can be used to specify a wide range of such scheduling problems. An extensive literature review is used to make sure that different problem version can be specified in the provided formulation.

Further a new framework shall be designed that allows the implementation of various heuristic solvers to be applied to a wide range of problems specified in our formulation. A unified handling of constraints and reusable moves will be designed to reach this goal.

Next a new algorithm shall be implemented in the framework to show the applicability to several different problems. The evaluation will be performed on different problems from literature using different types of demand and constraints in order to test the robustness of the framework.

An instance generator for new instances combining constraints in new ways not yet explored in literature will be provided as well. This generator shall be configured by a wide range of parameters. Several newly created instances will be evaluated with the framework as well.

1.2 Contribution

This thesis gives a contribution to the GES Format which is an Extensible Markup Language (XML) format that is designed to be able to describe different problem variants used in literature and combine them in new ways that have not yet been investigated. It allows problems to be specified using a wide range of constraints that are either hard constraints or soft constraints inducing a penalty for violations. This allows to cover

many constraints from different problems in literature as well as the specification of demands in different ways that are used in literature.

The main contribution is the framework that is developed allowing the implementation of various heuristic solvers for different kinds of problems specified in our formulation while increasing reusability and easy adaptation to new problem variants. This is done by providing a unified handling of constraints where individual constraints can easily be introduced or changed, while all constraints are handled in a common way. Moves are build in a way to promote reusing them in different algorithms as all moves follow the same structure while new moves can easily be added. Various algorithms can be implemented and either be applied on their own or as part of a larger algorithm.

In order to be able to test solution methods on various instances exhibiting different characteristics and different degrees of difficulty, an instance generator is developed that can combine a wide range of different constraints that are expressible in the GES format. Its focus is to allow easy generation of large numbers of random instances while still being able to specify or bound several characteristics of the instances via a configuration file. Moreover, special effort is invested to create instances that are feasible in a real world scenario. This includes factors like reasonable shift lengths and sequences or break and task sequences.

A new approach based on Simulated Annealing is implemented in this framework and applied to various benchmark instances from literature for comparison as well as to the instances from the instance generator. The instances from literature cover nurse rostering [Cur17] as well as different problems involving tasks from [SEVB16] and [LBMP13].

The results show that our formulation and framework can be applied well to all different problems that are mentioned above. Most constraints can easily be described in our format, while special constraints can be added to the framework with low additional effort. Our general purpose algorithm shows reasonable results compared to the specialized algorithms in the various problems in good runtime.

The remainder of this thesis is organized as follows. In chapter 2 an overview of related work in employee scheduling is presented. In chapter 3 the problem definition and the corresponding XML format are presented. In chapter 4 the instance generator is described. Chapter 5 explains the structure of the framework and its components. In chapter 6 the evaluation of the framework on the generated instances and the instances from literature is presented. Chapter 7 provides a summary and an outlook for possible future work.

CHAPTER 2

Related Work

Many different versions of employee scheduling problems have been described in the past. Already in [GM86] an informal description of the General Employee Scheduling Problem was provided, giving rise to the identification of several common notions in all problems of this kind.

There is always some form of demand that needs to be fulfilled. Different types of demand can be specified depending on the problem. The work is grouped into shifts denoting consecutive periods of work assigned to an employee. The placement of such shifts is then dependent on different constraints. Either fixed shift types are already given and constraints only shape their placement or the shape of the shifts has to be designed as well. Shifts might also include breaks.

Finally there is a pool of employees that can be assigned shifts depending on various constraints. Employees might have different contracts, skills or working preferences.

Several reviews of different problem versions are available. In the review on staff scheduling and rostering in [EJKS04] several modules in the rostering process are identified.

Module 1 is demand modelling, where demand can be specified in different ways. The first one is task based, where a list of individual tasks is defined, often with skill requirements. Flexible demand is often modelled using forecasting techniques and stated as a number of employees needed, e.g., per hourly interval. These kinds of demand are called time demands in the context of this thesis. Further shift based demands are defined where the number of employees required per shift type is specified. The typical application area for such demands is nurse rostering.

The second module is days off scheduling, dealing with the differentiation of working days and days off. Shift scheduling deals with deciding what shifts are to be worked and how many employees per shift are needed. Selecting the shape of shifts is referred to as shift design in this thesis.

Module 4 is the line of work construction that deals with the arrangement of shifts incorporating constraints like sequences or other patterns. Task assignment deals with the assignment of tasks to shifts, finally staff assignment deals with the assignment of individual workers to lines of work.

The survey presents three main factors influencing differences between different problems. The degree to which days off scheduling, line of work construction and task assignments are integrated is one of them, the others are which of the described modules are relevant and the type of demand that is specified.

Transportation systems, call centers, health care systems, protection and emergency services, civic services and utilities, venue management, financial services, hospitality and tourism, retail and manufacturing are presented as application areas for staff scheduling in this paper.

The combined scheduling of days off and assigning shift sequences to employees is known as the Tour Scheduling Problem (TSP). For a review presenting several different approaches to solve the TSP see [Alf04].

In the more recent review [VdBBDB+13] hundreds of papers are classified according to different characteristics that are described. Some important characteristics are summarized in the following.

Frequent contractual constraints can refer to full time, part time or casual employments, they also frequently include skills. Scheduling often involves individual assignment, but can also rely on crew scheduling. Decisions often involve task scheduling, group scheduling, shift sequences or scheduling of time periods. Shifts can be placed differently across the day, either with fixed start and end times or with the requirement for shift design. Coverage constraints are often included as hard constraints, but can also be soft constraints. Overstaffing and understaffing might be allowed and treated in different ways.

Several different ways of including cost, e.g., per employee, per day or per task can be distinguished. A balanced workload as well as employee preferences are frequently used. Lots of different time-releated constraints regarding the number and sequence of assignments, the workload, the time between assignments and much more are identified.

Presented solution methods include several types of mathematical programming, constructive heuristics, improvement heuristics, simulation, constraint programming and others. Some problem variants also incorporate uncertainty, however, this case is not further incorporated in this thesis.

The recent review [DBVdBBD15] focuses on work including skills. This review distinguishes different skill classes, the hierarchical and categorical class and deals with different ways to incorporate skill substitution. It investigates in detail how different papers deal with the definition and assignment of skills.

Methods in nurse rostering are reviewed in [BDCBVL04]. The nurse rostering problem originates in hospital staff scheduling for nurses. It typically involves several different,

predefined shift types with various staffing requirements and several constraints restricting the way nurses can be assigned to these shifts. It might also contain skills that are required for the assignment. The review again categorizes different methods to approach such problems. There are also variations that consider cyclic or rotating schedules like in [Mus06], where heuristic methods for such problems are presented.

In order to evaluate the performance on nurse rostering for our approach we focus on the instances from [Cur17]. A model of the problem is presented by [CQ14]. They provide a range of generated instances from small to large including some very challenging large instances where no optimal solutions are known. Various real life instances are provided as well, [BC14] presents methods that are successfully applied to these instances.

Shift design is described in [MSS04]. In these types of problems the shifts types are not fixed, but shifts have to be defined by the algorithm. The assignment of breaks is included in [BGM+10].

One of the problems using task demands is the Personnel Task Scheduling Problem (PTSP) in [KE01] and its optimization variant, the Shift Minimization Personnel Task Scheduling Problem (SMPTSP) in [SWMVB14] and [KEB12]. In this case tasks need to be assigned to shifts that are already predefined. The SMPTSP further considers minimization of the required number of employees.

This problem also relates to the interval scheduling problem, for a survey see [KLPS07]. However, better results can be achieved when scheduling shifts and tasks at the same time as stated by [EJKS04].

A combination of shift and task assignments called the Integrated Task Scheduling and Personnel Rostering Problem (TSPR) is described in [SEVB16] and taken as one of the problems for evaluation of our framework. It uses constructive heuristics based on column generation and other decompositions as well as very large neighborhood search and integer programming to obtain good results.

This paper also contains an overview of various papers that deal with task assignments and what kinds of tasks they consider. A related class of Employee Timetabling Problems (ETPs) is also defined and modelled in [MS03].

A challenging problem is described in [LBMP13] and [PLBM15]. The Shift Design and Personnel Task Scheduling Problem with Equity Objective (SDPTSP-E) not only considers shift design at minute granularity together with task assignments, but also a special equity objective and the scheduling of breaks.

In [LBMP13] a constraint-based approach is used to solve the problem, while in [PLBM15] a two-phase method is presented where the assignment of shifts and the assignment of tasks are treated in alternating phases.

A heuristic approach to a similar problem also dealing with shift design and the assignment of tasks is already presented in [LJ91] in the context of the fast food industry.

When dealing with new problem variants, many approaches generate their own instances. However, the details on how these instances were generated vary. In [CQ14] some reasons for introducing new generated instances are described. For [SEVB16] the instance generator was published. For the problem in [LBMP13], several parameters like the placement of tasks are described for the generated instances. There is also further work available on how to generate useful instances in [VM09] and [MSS04].

Further there are papers providing general modelling and complexity analysis like [BQB11] including some results that even some special cases in certain problems can already lead to NP-hardness.

There is also much work on different heuristic optimization techniques in general, e.g., [GK03] gives a good overview of several techniques including simulated annealing. The application of various metaheuristics to employee scheduling problems is covered by many of the surveys stated above.

CHAPTER 3

Problem Definition and Specification Format

In General Employee Scheduling a wide range of different constraints needs to be considered to allow the specification of different requirements without the need to introduce a new problem formulation for each variant of the problem.

Based on the analysis of various employee scheduling problems in literature, we decided on the problem variants and constraints to be covered in order to include a wide range of problems using different demand specifications and different types of definitions, restrictions or preference specifications. This chapter presents the main ideas and the structure of our new formulation as well as an overview of the different specification options that are available.

In order to specify this formulation in a way that is both human-readable and machine-readable, XML[1] is a useful format that allows to structure the large amount of specification options. Further XML formats can be extended easily without breaking the structure of already existing instances. Therefore the GES formulation is specified as an XML Schema Definition (XSD)[2] file. For details refer to the technical report at [KMM+17].

Some XML problem formats already exist, e.g., the AutoRoster[3] and ShiftSolver[4] modelling formats. However, our formulation combines the possibilities of these formats, extends them with more options and provides a homogeneous and structured formulation allowing new combinations of constraints and demands not yet investigated in literature.

[1] http://www.w3pdf.com/W3cSpec/XML/2/REC-xml11-20060816.pdf
[2] https://www.w3schools.com/xml/schema_intro.asp
[3] http://www.staffrostersolutions.com/support/autoroster-problem-data.php
[4] http://www.staffrostersolutions.com/support/shiftsolver-problem-data.php

3.1 Problem structure

The problem deals with the scheduling of shifts as well as optionally tasks and breaks for a set of employees over a certain period of days. The period length is denoted as p and is fixed for each instance. The set of employees E considered for a solution might be fixed or variable.

A schedule assigns either a day off or precisely one shift on each day $0 \leq i < p$ to each employee $e \in E$. Each shift s has a type $type_s$, a start time $start_s$ and an end time end_s. Shifts might overlap to the next day, but they must not overlap each other. The available types of shifts as well as their placement can be guided by a large number of constraints.

The whole schedule including possible task and break assignments is called the schedule, when just talking about the shift assignments, we speak of the roster. The schedule for an individual employee is called an employee schedule, the schedule for a specific day a daily schedule.

If the scheduling of tasks or breaks is required, each shift s in the schedule can contain a list of task parts T_s where each part $t \in T_s$ has defined start and end times $start_t$ and end_t and the ID of the corresponding task demand $demand_t$. Note that we speak of task parts as tasks might be preemptive. More details are described in the demand specification.

Further a list of breaks B_s, where each break $b \in B_s$ has start time $start_b$ and end time end_b as well as a break type $type_b$ can be defined. Tasks and breaks in a valid schedule have to lie within their enclosing shift and must not overlap each other. Again a large number of constraints guides the placement of these elements.

Time spans, while allowing different formulations in the format, are always considered to be in minutes in this specification and refer to differences between time points. A time point can be relative to a specific day (e.g. a shift on day 5 starts at 20:00 and ends at 4:00 on the next morning) or absolute, calculated from 0:00 on day 0. All IDs in the specification are considered to include alphanumerical characters, "." and "_".

Constraints can either be hard constraints in which case they do not define a weight or they can be soft constraints inducing a penalty for each violation. In this case two attributes define the penalties. A numerical value *weight* defines the weight of the violation. Further a function can be specified. The penalty is then calculated from the violation *violation* as follows.

- `Constant`: if $violation > 0$ then $weight$ else 0
- `Linear` (default function): $weight \cdot violation$
- `Quadratic`: $weight \cdot violation^2$

If necessary, both the format and the solver framework can easily be extended to include further penalty functions.

Each instance can have an optional ID. For each instance several main parts are considered.

- `General` defines global properties of the instance as well as some flags indicating specific types of problems.
- `Tasks` defines the available task types.
- `Shifts` defines the available shift types and the constraints regarding their shapes and occurrences.
- `Breaks` defines the available break types and the constraints guiding their placements.
- `Employees` defines the available employees, their possible skills as well as their contracts and a large number of constraints regarding contractual limitations as well as employee preferences.
- `Demands` defines the demands that need to be fulfilled.

3.2 General Definitions

General definitions about the instance are specified in the `General` part. The following elements are defined in this part.

- Period length p: This value specifies the number of days in the planning period. It can either be specified directly or by giving the start and end date of the period.
- First week day $wd(0)$: The weekday of the first day in the planning period. This can be inferred from a given start date or specified directly if p is specified directly.
- Start time *start* and end time *end*: On the first day of the planning period, shifts are not allowed to start before *start*, on the last day of the planning period they are not allowed to extend beyond *end*.
- Weekend definition: Per default each shift s starting on a Saturday or Sunday is considered a weekend shift. This might be changed to an arbitrary set of weekdays.

  It might also be specified using a start weekday $startday_{we}$, an end weekday $endday_{we}$ as well as the precise starting time $start_{we}$ and ending time end_{we}. The boolean value *startCompleteOnly* determines if shifts starting before, but ending within the weekend should be counted as weekend, *endCompleteOnly* does the same for the end. If *true*, only shifts lying fully within the weekend are counted as weekend shifts. This allows to model requirements like all shifts starting after Friday 20 pm, but before Monday 4 am are considered to be weekend shifts.

  $isWeekend(s)$ uses the given definition of weekends to determine whether a shift is considered a weekend shift from now on.

- Time slot length *timeSlotLength*: This value specifies the time granularity for the instance. The default of 1 allows minute precision in scheduling, but often shift and task assignments are fixed to, e.g., half or full hours only which is reflected by this setting.
- *ignoreIndividualRosters*: This flag indicates problems only using variable employees, where only the number of employees of each type working each particular shift is required for the solution.
- *shiftDesign*: This flag indicates problems that do not have fixed shift types that are predefined (e.g., one morning shift 6:00 to 15:00 and one afternoon shift 13:00 to 22:00), but require shift design.
- *cyclicSchedule*: This flag indicates problems that require either cyclic schedules, where at the end of the planning period each employee restarts their schedule, or rotating schedules, where each employee continues with the schedule of the previous employee.
- *allowSequenceCutoff*: This flag can be set individually for the start and the end of the planning period to specify whether a required sequence of shifts can be cut off at the border of the planning period or has to be finished within.

3.3 Tasks

Each task (part) t has a task type $type_t$. The set of task types T is defined in the section `Tasks`. An example of possible task types in a scheduling problem could be preparation, work, maintenance and cleaning. Each type $tt \in T$ can have the following attributes.

- ID id_{tt}: The ID identifies the task type.
- A name $name_{tt}$: A human-readable name for the task type.
- A set of prerequisite task types $Prerequisites_{tt}$: These task types have to be performed before the current type can be performed, e.g., work requires preparation to occur beforehand.
- $isWork_{tt}$: This flag indicates whether the task type is considered working time. E.g., preparation and cleaning might have to be considered in the schedule, but are not regarded as actual working time.
- $reAcquaintancePeriod_{tt}$: The time span it takes until the next task can be started after working on this task type. It might be used to include fixed cleaning and closing duties where the employee cannot perform the actual task any more, but is not yet available for the next task.

Further task types can be grouped in task groups for easier reference. A task group $tg \in TG$ is simply a set of task types.

3.4 Shifts

Each shift s has a shift type $type_s$. The set of shift types S is defined in the section `Shifts`. An example of possible shift types could be a morning shift from 6:00 to 15:00, an afternoon shift from 12:00 to 20:00 and a night shift from 20:00 to 6:00. Each shift type $st \in S$ provides the following properties.

- ID id_{st}: The ID identifies the shift type.
- A name $name_{st}$: A human-readable name for the shift type.
- A label $label_{st}$: A label for the display of rosters, e.g., "M" for a morning shift.
- An arbitrary number of constraints can define possible boundaries for the shift times. It is possible to specify the minimal and maximal start times $minStart_{st}$ and $maxStart_{st}$, minimal and maximal end times $minEnd_{st}$ and $maxEnd_{st}$ as well as minimal and maximal length $minLength_{st}$ and $maxLength_{st}$ of the shift.

  Note that it is possible to use both hard and soft constraints and combine them even for the same boundary. For example it is possible to have a hard constraint for a minimal start time of 8:00 together with a soft constraint for a start time of 10:00 to state that a shift should not start earlier than 10:00 if possible, but not earlier than 8:00 under all circumstances.
- A set $Limits_{st}$: Optionally limits can be defined for the number of different employees working a shift. These limits $minLimit_{st}$ and $maxLimit_{st}$ can further be restricted to a certain range of days. E.g., it is possible to specify that from day 0 to day 5 at most three different employees can work night shifts.
- A set $ValidDays_{st}$: The shift type can be restricted to an arbitrary collection of days in the planning period. For example a specific shift type can be defined that is only allowed on weekends and holidays.
- A set $FixedShiftTasks_{st}$: Certain tasks might have to be executed by every employee working this shift, e.g., the morning shift might require machine startup as the first task. For each such task $t \in FixedShiftTasks_{st}$ the length $length_t$ and the relative position of the task start from either the start or the end of the shift $relativePosition_t$ are specified.

Further shift types can be grouped in shift groups for easier reference. A shift group $sg \in SG$ is simply a set of shift types.

Global shift constraints can be defined as well. Currently two such constraints c are supported. For both constraints the set of shift types affected by the constraints can be restricted to the set $S_c \subseteq S$. The constraints are as follows.

- `AverageShiftLength`: The average shift length constraint can set minimal or maximal values for the average length *avgLength* of all shifts in the planning horizon matching the type restriction *MatchingShifts*, where

$$avgLength = \sum_{s \in MatchingShifts} \frac{end_s - start_s}{|MatchingShifts|} \ . \tag{3.1}$$

- `ShiftInstances`: The shift instance constraint can set minimal or maximal values to the number of different instantiations of a shift type. An instance of a shift type is defined as a unique pair of start and end times for a certain shift type. E.g., shifts 6:00 to 16:00 and 7:00 to 16:00 would both be valid instances of a shift type with minimal starting time 5:00, maximal starting time 8:00 and fixed ending time 16:00.

The flag *allowBorderShifts* defines whether shifts on the last day extending beyond midnight are allowed.

Further a time slot length for shifts *shiftSlotLength* is defined. This might be needed if shifts e.g., should only start at full hours, but tasks and breaks within the shifts might be scheduled at 10-minute-precision.

3.5 Breaks

Each break b has a break type $type_b$. The set of break types B is defined in the section `Breaks`. Each break type $bt \in B$ can have the following properties.

- ID id_{bt}: The ID identifies the break type.
- Constraints for the minimal and maximal length of the break $minLength_{bt}$ and $maxLength_{bt}$ can be defined.
- Constraints for the minimal and maximal start time of the break $minStart_{bt}$ and $maxStart_{bt}$ as well as the minimal and maximal end time $minEnd_{bt}$ and $maxEnd_{bt}$ can be defined.
- The constraints for the minimal and maximal start of the break can also be defined relative to the start of the shift by $minStartShift_{bt}$ and $maxStartShift_{bt}$. The minimal and maximal end of the break can be defined relative to the end of the shift with $minEndShift_{bt}$ and $maxEndShift_{bt}$.
- A factor $0 \leq workLength_{bt} \leq 1$: This factor defines how much of the break is considered work time.

- Constraints for the arrangement of breaks can be defined as well. The constraints $minWorkBefore_{bt}$ and $maxWorkBefore_{bt}$ specify the distance to the previous break. E.g., a break can be defined to occur not earlier than three hours after the previous break. Equally $minWorkAfter_{bt}$ and $maxWorkAfter_{bt}$ define the distance to the next break.

In addition to break types there are break configurations that control which breaks can be scheduled for which shifts. Each break configuration bc in the list of break configurations BC contains a shift filter f_{bc}. For each shift the first configuration in the list with a matching filter is applied. A shift filter f_{bc} can contain the following properties.

- A collection of days $Days_f$: The configuration is only applied to shifts on one of the specified days.
- The minimal and maximal length of the shift can be specified by $minShiftLength_f$ and $maxShiftLength_f$.
- The minimal and maximal start time of the shift can be specified by $minShiftStart_f$ and $maxShiftStart_f$. The minimal and maximal end of the shift can be specified by $minShiftEnd_f$ and $maxShiftEnd_f$.
- A set of contracts C_f: The configuration only matches shifts where the employee the shift is assigned to is hired under at least one of the given contracts.
- A set of shift types S_f: Only shifts of the given types are matched.
- The minimal and maximal work length of the shift can be specified by $minWorkLength_f$ and $maxWorkLength_f$.

The break configuration bc itself can contain the following properties.

- A set of break types B_{bc}: This set specifies the break types that can be used in the given break configuration. For each break type a minimum and maximum number of occurrences can optionally be specified. Additionally a flag $ordered_{bc}$ can be set to indicate that B_{bc} is ordered.
- The relative occurrence of the first break related to the start of the shift can be specified by $minWorkBeforeFirst_{bc}$ and $maxWorkBeforeFirst_{bc}$. The time between the last break and the end of the shift can be specified by $minWorkAfterLast_{bc}$ and $maxWorkAfterLast_{bc}$.
- The total amount of break time in the shift can be specified either as an absolute value with $minTotalBreakTime_{bc}$ and $maxTotalBreakTime_{bc}$ or relative to the length of the shift with $minTotalBreakTimeFraction_{bc}$ and $maxTotalBreakTimeFraction_{bc}$ in the interval $[0; 1]$.

Break configurations can be used, e.g., to specify that all shifts starting before 12:00 and ending after 14:00 should have a lunch break or that night shifts get more breaks than day shifts.

3.6 Employees

The section `Employees` consists of several important parts of the problem definition. These are the definition of skills, contracts, available employees and employee preferences.

3.6.1 Skills

The set Sk of skills defines an arbitrary number of skills.

Each skill $sk \in Sk$ might substitute a set of other skills $IncludesSkill_{sk}$. However, each substitution can have a penalty value. For example, the skill "Leading nurse" might substitute the skill "Nursing", but as common nursing duties prevent the leading nurse from organisational duties, such assignments might be penalised.

3.6.2 Contracts

The section `Contracts` defines a set of contracts C. Each contract $c \in C$ can define a wide range of constraints.

- ID id_c: The ID identifies the contract.
- An ID of a parent contract $extends_c$: Optionally a contract might extend another contract. In this case all constraints from the parent contract apply to this contract as well. This might be useful if there are common elements in many contracts together with few individual parts, for example when employees are hired for different weekly hours, but have the same contractual constraints otherwise.
- A flag $allowWorkSwitchesPerShift_c$: This flag controls whether employees with this contract can switch between different task types or skill assignments in a single shift. If *false*, then only tasks of the same type or demands using the same skill are considered for each shift. This can be used if switching between different task types requires, e.g., different gear for the employee and is therefore unwanted.
- The value $workSwitchLimitPerShift_c$ can set upper and lower bounds to switches between different task types or skills. E.g., this can be used to require at least one switch per shift to prevent monotonous assignments.
- An arbitrary number of pattern constraints can be defined. Those constraints restrict the arrangement of shifts for an employee.

- `CountConstraint`: This constraint specifies minimal and maximal numbers of occurrences of either days without a shift or a set of shift types. Optionally the constraint can be restricted to a range of days. This constraint can be used, e.g., to specify that from day 0 to day 6 at most 2 night shifts can be assigned to an employee working under this contract.
- `SequenceConstraint`: This constraint specifies minimal and maximal lengths of sequences of either days without a shift or with a set of shift types. Optionally the constraint can be restricted to a range of days. This can be used to model, e.g., that at least two days off have to be scheduled in a row or that no more than 3 night shifts can be assigned in a row.
- `WeekendCount`: This constraint counts the number of working weekends and applies upper or lower bounds to this number. Again the constraint can be restricted to a certain range of days. A working weekend is defined as a weekend where the employee works any shift s where $isWeekend(s)$ is *true*.
- `WeekendSequence`: Similar to the previous constraint this one restricts sequences of working weekends. Again a range of days might be specified.
- `ForbiddenSequence`: This constraint defines a sequence that is not allowed in the schedule. Optionally any collection of days can be specified that should be considered as starting days for sequences, otherwise the sequence might start at any day. A list of shift matches is specified. Each match might be a day off or a set of shift types. Convenience specifications like a set of unmatched shift types to match all shift types except a few exist as well.

  A forbidden sequence is found if and only if the whole list of matches is found in the given order starting at one of the specified days. A possible use is to prevent a morning shift following a night shift or to prevent having to work a night shift before a free weekend.
- `IdenticalSequence`: This constraint gives minimal and maximal bounds for an identical sequence. Further a collection of possible days to start the sequence might be given, otherwise it applies to all days in the planning period. An identical sequence is defined as a sequence of days with the same shift type assignment. This might be a day off or a specific shift type. E.g., this constraint can be used to specify that Monday to Friday the same shift type should be scheduled to an employee.
- `MatchConstraint`: This constraint is generic and allows to match arbitrary patterns. It can specify minimal and maximal numbers of occurrences of the patterns and a set of patterns. Each occurrence of any of these patterns counts towards the number that is regarded for the constraint.

  Each pattern might be restricted to a collection of starting days. Then a list of shift matches is specified like in the forbidden sequence constraint.

- An arbitrary number of conditionals can be defined. These model if-then conditions where the constraint itself is only evaluated if the condition holds.

The definition of conditionals consists of two parts. First a set of variables V_c is defined. Each variable $v \in V_c$ might be either a `MatchVariable` or a `WorkloadVariable` and is identified by an ID id_v. A `MatchVariable` contains a `MatchConstraint` and identifies it with the variable ID. A `WorkloadVariable` contains a `WorkloadConstraint` as defined later and and identifies it with the variable ID.

Now a set of conditionals $Cond_c$ is defined. Each conditional $cond \in Cond_c$ is assigned the constraint weight and the two parts `If` and `Then`. Both of those parts can contain boolean expressions using the variables from V_c.

When a conditional is evaluated, first the boolean expression from the `If`-part is evaluated. If this part is *false*, the conditional is ignored. If it is *true*, then the `Then`-part is evaluated. Now if this part is *false*, the penalty is applied.

This might be used to model situations like if a shift is longer than 10 hours, there must not be a morning shift on the next day.

- A set of provided skills Sk_c: This set includes all skills that an employee hired under this contract provides. Note that certain skills might substitute further skills as defined in the previous section.
- A set of valid shift types S_c: If this optional element is provided, only shifts in this set can be assigned to employees working under this contract.
- A set of fairness constraints can be defined that applies to all employees working under this contract. Currently one constraint of this type is supported. It places an upper limit to the gap between the lowest and highest workload (as defined later) among employees working under this contract.
- A flag $usageOptional_c$: If this flag is set, using an employee under this contract is optional. It can be combined with a penalty weight for each employee used under this contract. This can be used, e.g., to model individual external employees that might be hired if needed.
- Minimum rest times necessary between two shifts $minRestTime_c$ can be defined.
- Minimum weekly rest times that have to be respected between two shifts at least once a week are defined by $minWeekRestTime_c$.

  Note that this definition is not precise and different problem formulations might use slightly differing versions of this constraint. For example it could be interpreted as at least one matching rest time per calendar week (Monday to Sunday) or at least one matching rest time per rolling horizon of a week.
- Minimal and maximal values for the workload $minWorkload_c$ and $maxWorkload_c$ of the employee can be defined. These can optionally be restricted to certain ranges in the planning period, e.g., to model weekly workload requirements. Further a flag indicates whether a particular workload constraint uses the shift length as measure

or only the actual working time where some breaks and tasks might not be counted as working time.

- Shift start times and shift end times can be restricted as well. For each of these restrictions minimal and maximal values can be specified by $minStart_c$, $maxStart_c$, $minEnd_c$ and $maxEnd_c$. Additionally the restriction might be applied only to a specific collection of days in the planning period or only to specific shift types. This might, e.g., be used to prevent employees of this contract from working later than 23:00.

- In a similar way shift lengths can be restricted by $minLength_c$ and $maxLength_c$, again optionally filtered by a collection of days or shift types. Additionally this constraint allows a flag to specify whether to use shift length or actual working time.

- Further a similar constraint can be used to restrict the length of assignments to the same task type. Once again minimal and maximal values $minTaskLength_c$ and $maxTaskLength_c$, as well as restrictions to a collection of days and certain task types, can be specified.

3.6.3 Employees List

In this section the set of available employees E is defined. Employees can either be specific named employees or variable employees defining a homogeneous pool of employees that can be hired if necessary. All employees $e \in E$ have the following properties.

- ID id_e: The ID identifies the employee or the type of variable employee.

- A set of contracts C_e: Each employee can be assigned an arbitrary number of contracts. The schedule for this employee has to respect all the constraints from all assigned contracts.

- Each employee can have a set of preferences. These include the following specifications.

  - A set of penalized tasks $PenalizedTask_e$: Each penalized task is specified by the task type or a set of task types together with a penalty weight. Each time the specified task types are assigned to this employee, the penalty is added. This can be used to model situations where an employee is less qualified to do a task, while still able to do it in principle or to model employee preferences regarding tasks.
  - A set of shift-off requests $ShiftOffRequest_e$: Each request specifies the collection of days targeted by the request together with the set of shift types that are not wanted on these days. If multiple days are specified, the penalty is added exactly once as soon as an unwanted shift is scheduled on any of the given

days. Note that a day off can be requested by declaring all shifts (in the format with the shortcut `AnyShift`) as unwanted. This constraint can be used to model day off requests as well as requests, e.g., to have a free evening by specifying shift types blocking the evening.

  - A set of shift-on requests $ShiftOnRequest_e$. Each request again can specify a collection of days and a set of shift types. The request is fulfilled as long as at least one shift of any required type is scheduled on at least one of the specified days. Otherwise the specified penalty is added.

- A set of preassigned shifts $PreAssignments_e$: These specifications either define fixed periods where no shifts can be assigned or shifts that are already predefined.

  - `Shift`: Each preassignment of this type defines a shift that is fixed in the schedule. It contains a fixed shift type and day of assignment. Start and end time might be fixed as well. If they are not given, start and end times can be determined according to the shift type by the solver. If they are given, flags indicate whether further extension is possible. Therefore the shift could be fixed from 8:00 to 12:00, but allow arbitrary extension at the end, possibly resulting in a shift 8:00 to 16:00 in the final schedule.

    Further task assignments, mastered skills and break assignments can be defined. These assignments ignore the usual rules guiding their applicability to this shift. This even allows to assign temporary skills or tasks that could normally not be executed. Each task, skill and break assignment has a start time and a length, as well as the corresponding task type, skill ID or break type.

    Note that in this case the day specification explicitly allows dates or days before the beginning of the scheduling horizon. This can be used to specify a history of previous shifts that is relevant for shift arrangement constraints like sequences of shifts.

  - `NoShift`: Each preassignment of this type specifies a start day and time as well as an end day and time. No assigned shifts are allowed to overlap with this interval. This constraint can be used to model fixed absences like a scheduled holiday.

Named employees can further specify a name $name_e$.

Variable employees can additionally define the following properties.

- A cost $cost_e$: This cost is added for each variable employee of this type that is scheduled in the solution.

- Bounds for the minimal and maximal number of employees of this type can be given by min_e and max_e.

- Relative bounds for the fraction that this type of employee takes up in the total number of employees can be given by $minFraction_e$ and $maxFraction_e$ in the interval $[0; 1]$.

Finally an arbitrary number of employee pairing requests can be specified. An employee pairing pr defines the following properties.

- A weight $weight_{pr}$
- A collection of days $Days_{pr}$: The pairing has to take place on at least one of the specified days to be counted. In case of more complicated pairings the first day of the pairing has to be within this collection.
- A set of assignments $Assignment_{pr}$: Each assignment $a \in Assignment_{pr}$ contains the ID of an employee id_a and a sequence of shift matches, each either matching a day off or a set of shift types.

A pairing pr is matched, if and only if a day $i \in Days_{pr}$ is found where for all employees e with an assignment $a \in Assignment_{pr}$, such that $id_e = id_a$, their whole sequence of shift matches is matched starting from day i.

A pairing in the simple case can state two employees working the same shift on a particular day. However, it can also span multiple employees and multiple days, including constructions, e.g., like having one employee work the shift another particular employee worked the day before.

Pairings can either be requested in the form of a `Pair` request, in which case the penalty is added if the pairing is not found in the schedule, or a `NotPair` request can be used to declare unwanted pairings. In this case the penalty is applied if the pairing is found in the schedule.

3.7 Demands

The demands that need to be fulfilled are specified in the section `Demands`. There are three different ways the demands can be specified. Each problem can choose one of these options.

3.7.1 Shift Demands

One way to specify the demands is to give the required number of employees working a shift type for each type of shift and each day. This specification is typically used in rostering problems like nurse rostering. Shifts are typically fixed in time in these kinds of problems.

A typical example would be an instance where there are morning shifts, afternoon shifts and night shifts and for each day three employees should work morning shifts, four employees afternoon shifts and two employees night shifts, while on weekends the requirement for afternoon shifts changes to two.

Such problems are specified by giving a set of shift demands D_s. Each shift demand $d \in D_s$ specifies the following properties.

- ID id_d: The ID identifies the demand.
- The ID of a shift type $type_d$: This ID specifies the type of shift that is required.
- A collection of days $Days_d$: The demand has to be fulfilled on each day specified here.
- The requirements are specified as a set of minimal and maximal numbers of employees that have to work this shift by min_d and max_d. Each requirement might optionally include a skill requirement that employees have to fulfil.

  Therefore it is possible to require, e.g., in total at least five employees where at least two of them possess a certain skill by using two lower bounds. Like in many other constraints is also possible to specify required and preferred levels by mixing hard and soft requirements.

3.7.2 Time Demands

A different way to specify demands is to use a time-based formulation. In this way for each period of time the required number of employees is specified. These problems often require shift design and might also incorporate breaks as employees currently on a break typically do not count towards the required number of employees.

A typical example would be a problem where from 6:00 to 10:00 at least three employees are needed, from 10:00 to 14:00 at least five, from 14:00 to 22:00 at least three and from 22:00 to 6:00 at least two. However, at no time more than six employees should work at the same time.

Such problems are specified by giving a set of time demands D_{time}. Each time demand $d \in D_{time}$ specifies the following properties.

- ID id_d: The ID identifies the demand.
- A start time $start_d$ and an end time end_d: These values specify the time interval the demand refers to.
- A collection of days $Days_d$: The demand has to be fulfilled on each day specified here.

- The requirements are specified as a set of minimal and maximal numbers of employees that have to work in the specified period of time by min_d and max_d just like for shift demands. Again skills might be required as well.

3.7.3 Task Demands

The final way to specify demands is to base the schedule around tasks. A task is a unit of work that has to be scheduled within a certain time window and requires at least one employee. If multiple employees are required to complete a task, it is assumed that they have to work on it at the same time. Further the same employees have to work the whole task, it is not possible that one employee starts the work on the task and hands over to another employee for the rest of the task.

This formulation is required when the work requirements come in the form of tasks that are predefined. E.g., in a factory there might be certain tasks on an assembly line that have to be done. Further there might be cleaning or maintenance duties. The shifts should now be planned in such a way that all these tasks can be covered.

Such problems are specified by giving a set of task demands D_{task}. Each task demand $d \in D_{task}$ specifies the following properties.

- ID id_d: The ID identifies the demand.
- The ID of a task type $type_d$: This ID specifies which task type this task belongs to.
- The length of the task $length_d$: The length of the task is fixed.
- Constraints for the start and end time of the task can be specified by $start_d$ and end_d. These might be hard bounds or soft bounds, also any combination is possible.
- A collection of days $Days_d$: The demand has to be fulfilled on each day specified here.
- A flag $allowSplit_d$: This flag determines whether the task execution might be split into several parts, i.e., if the task is preemptive. Additionally it is possible to specify that the task can only be interrupted by breaks (but not unassigned working time or other tasks) and to specify a minimum length $minLength_d$ for each part.
- A set of prerequisite tasks $Prerequisites_d$: Each of these tasks specified by their demand IDs, has to be executed before the given task can be scheduled.
- The requirements are specified as a set of minimal and maximal numbers of employees that have to work on this task by min_d and max_d just like for shift demands. Again skills might be required as well.

3.8 Solution Format

Corresponding to the GES format for specification of instances we developed an XML format for the specification of solutions, the GES solution format. It can directly hold a schedule in the following way.

The enclosing `Solution` element can hold a reference to the instance file as well as a set of employee schedules. Each of these schedules for an employee e is identified by the ID of the employee id_e. Further a list of shifts is provided.

Each shift s contains the type of the shift $type_s$, the day i on which the shift is scheduled and optionally the start time $start_s$ and the end time end_s. These might be skipped in rostering problems where each shift type has fixed starting and ending time.

Each shift can also contain an arbitrary number of tasks and breaks. Breaks b contain their break type $type_b$ as well as start and end time $start_b$ and end_b. Tasks t contain the ID of the corresponding demand $demand_t$ as well as start and end time $start_t$ and end_t.

CHAPTER 4

Instance Generation

In order to properly test and evaluate solution techniques for optimization problems, it is necessary to have a sufficient set of test instances to compare and measure performance. Ideally there should be instances exhibiting different characteristics and different degrees of difficulty, which could be related to metrics like the size of the instance or the number and shape of constraints. Of course it is beneficial to have real world data to ensure that solution techniques can be applied to real-world scenarios. We will also consider such instances in the evaluation. On the other hand, scenarios from the real world often capture one particular shape of the given problem, exhibiting similar characteristics across instances and it is often not possible to collect enough independent real-world scenarios from different sources.

Therefore, this chapter describes a new generator for random instances for the GES problem. Its focus is to allow easy generation of large numbers of random instances while still being able to specify or bound several characteristics of the instances via a configuration file. Moreover, special effort is invested to create instances that are feasible in a real world scenario. This includes factors like reasonable shift lengths and sequences or break and task sequences.

This generator can be used to generate new instances using various constraints in combinations that have not yet been investigated in literature. A download will be available.[1]

4.1 Generation Workflow

The approach taken for the generator is to first create a feasible schedule with no hard or soft constraint violations and put effort into creating such a schedule in a way that the result is plausible for real world scenarios.

[1] http://www.dbai.tuwien.ac.at/proj/arte/ges_instances

The configuration is read from a simple text file. Several parameters are available and will be described in more detail later. Default values are present, only values diverting from the default need to be specified. All random numbers are picked from a global random number generator that can be set with a seed to recreate a particular instance by just knowing the used seed.

In the second step a valid schedule and the corresponding constraints are created based on the configuration settings and random decisions. This step consists of three parts itself. First, the roster is created, including a variety of constraints specifying shift types by setting, e.g., allowed start times or lengths and the patterns that are allowed to occur including lengths of shift sequences, forbidden shift sequences, workload boundaries and more. Shifts are distributed to model demand fluctuations like on weekends as well as different employees with full or part time contracts and personal preferences.

Then breaks are scheduled first defining the types of breaks and the configurations that break assignments have to match and then assigning breaks to the shifts created in the previous step.

Before adding the tasks, skills are created and distributed among the employees. Then various types of tasks are generated and fit into the shifts taking into account the breaks that are already scheduled.

The requirements and constraints are built from the schedule in a way that none of the constraints is violated, while typically setting narrow bounds around the schedule to restrict the existence of feasible solutions that are very different to the current solution.

The final main step is the transformation where the whole setting is transformed into the problem specification. In many cases constraints built during the creation of the schedule can be transformed immediately. For some cases like preferences elements of the schedule are picked at random and declared to be a preference.

While such an approach includes the possibility to construct schedules in earlier phases that are hard to work with in later stages or might in the worst case not be completable, the separation helps to reduce the complexity of each step. Also, the generator might use just some of the stages and, e.g., output a rostering problem without breaks or tasks, depending on the required demand.

This process allows the generation of a wide range of problems guided by both randomness and user preferences.

Four parameters define the general shape of the instance.

- `days`: Defines the period length p.
- `history`: Defines the number of days h a history of assignments is created for.
- `timeSlotLength`: Defines the corresponding parameter *timeSlotLength*.
- `shiftSlotLength`: Defines the corresponding parameter *shiftSlotLength*.

4.2 Shift Generation

The shift generation consists of several steps, starting with shift types and instances, then building rosters for individual employees and finally combining them to a full roster. This process is guided by the following parameters.

- `shiftTypes`: Defines the number of shift types $|S|$.
- `shiftAvg`: Defines the average number of employees working each shift type per day, indirectly influencing the number of employees in the instance.
- `weekendFactor`: Defines the percentage of `shiftAvg` to use on weekends.
- `shiftVariation`: Defines a factor roughly describing that the most frequent shift type should occur about `shiftVariation` times as often as the rarest one, i.e., 1 means shift types appear evenly.
- `shiftSequenceRegularity`: Defines the probability of an employee to work the same shift type on the following day. This is typically desired in real world scenarios as employees usually do not want to jump wildly between different shift types.
- `dayDifference`: This value states the desired maximum difference in occurrences of the same shift type on different days.

4.2.1 Shift Types and Shift Instances

For the generation of shift types and instances some more parameters are relevant.

- `minShiftStart`: Defines a lower bound for $minStart_{st}$ for all $st \in S$.
- `maxShiftStart`: Defines an upper bound for $maxStart_{st}$ for all $st \in S$.
- `minShiftLength` and `maxShiftLength`: These values provide outer bounds for the selection of $minLength_{st}$ and $maxLength_{st}$ for each $st \in S$.
- `minShiftInstances` and `maxShiftInstances`: These values set bounds on the number of shift instances that are generated per shift type.
- `shiftInstanceDifference`: This value bounds the maximal difference in starting times and lengths for instances of the same shift type.
- `shiftInstanceConstraintType`: Defines whether the problem will contain a `ShiftInstance` constraint and how it will be built.
- `shiftDefinitionTightness`: This factor defines how tight the definitions on shift types will be in the problem instance, e.g., 1 will restrict starting times and lengths to only the value range used in the generator, while lower values will allow starting times and lengths in larger time windows.

For each shift type a valid starting time and length are chosen as a prototype for this type. Then the required number of instances is created by changing both starting time and length within the given bounds for each instance. No pair of shift instances is allowed to have exactly the same starting time and length.

The final time windows for the types to be used in the instance formulation are then calculated from the chosen instances and the given tightness value.

4.2.2 Employee Roster Generation

The next step in the process is to create a large candidate set of employee rosters. Each of those potentially forms a row in the final roster. Instead of building the rows trying to precisely fulfil a set of possibly contradicting constraints for, e.g., total work time, shift sequences and more, the approach used here is different. The following parameters shape this step.

- `contracts`: Defines the number of contracts $|C|$.
- `contractVariation`: Again a factor that describes how much more often the most frequent contract should appear compared to the least frequent one.
- `contractTimes`: A list of target working times in minutes across the whole planning period is given, where each value corresponds to one contract.
- `minOn`, `maxOn`, `minOff` and `maxOff`: Each of these is given as an array of values for the corresponding contracts. The values are transformed to shift sequence constraints for the instance.

Now a large set of candidates is created for each contract. These rows span the whole scheduling horizon and the requested history $p + h$. As the rows are created sequentially, the sequences of consecutive shifts are already incorporated at this step. As weekends can be set to have differing requirements and this would not be reflected by just choosing the lengths uniformly from the intervals given in the configuration, a more sophisticated approach is taken.

For each weekday a probability distribution is created for choosing the length of a sequence starting on this weekday. These distributions make is less likely for a working sequence to span across the weekend, but more likely for a free sequence.

Additionally the shift instance for the first working day of a sequence is chosen according to a probability distribution where the most likely shift type is more likely than the least likely shift type by the factor `shiftVariation`. All other shift types have a likelihood between the two extremes.

According to `shiftSequenceRegularity`, the shift type is kept for the next day in an ongoing sequence, otherwise a new shift is chosen as described before. Clashes between shift instances overlapping each other are prevented.

4.2.3 Employee Roster Restriction

The amount of rows that are generated actually depends on the restrictions that will be performed in the next step. Currently there are options to restrict the total work time across the whole scheduling horizon, the weekly work time and the patterns, which means that some shifts must not occur on consecutive days. More restrictions could be added easily.

In general restrictions specify their application and can either be skipped (`NONE`) or applied individually per contract (`LOCAL`) or for all contracts at once (`GLOBAL`). However, for example the total work time restriction only makes sense per contract as the contract types mainly distinguish different full and part time employments.

The application of a restriction typically works by sorting all shift rows according to the feature to be restricted, e.g., according to their total work time. Then a certain part of the rows at the beginning and the end of the sorted list is discarded, guided by a `restrictionFactor` for the specific restriction. This way outliers are eliminated and, depending on the percentage to remove, only a small range of values remains for this feature. In the most extreme case, a restriction value of 1.0 would restrict the feature to just one common value for all remaining rows, however, at the cost of generating a significant overhead of rows that are discarded.

For the pattern restriction the process is slightly different as first the occurrences of possible forbidden patterns are counted across the rows and then among the patterns that are occurring, the `patternRestrictionCount` least frequent ones are chosen as forbidden. This process makes sure that forbidden patterns are not infeasible anyway, but at the same time not too many rows are discarded in the restriction.

One more optimization step is included if the amount of candidate rows to generate is too large. As soon as the current number of candidates passes a threshold, the next restriction is applied immediately and the results of the restriction are directly applied to all further candidates, discarding them immediately if they do not fit.

4.2.4 Employee Roster Selection

Finally the roster is finished by selecting the desired number of employee rosters. First a contract is chosen again based on a distribution where the factor between the highest and lowest likelihood of a contract is given by `contractVariation`.

Next a selection of rows matching the contract is chosen at random from the shift rows that survived the restrictions. The goal in this step is to generate an even distribution of shift types across the days where for the same shift type the number of employees working on this shift type does not fluctuate more than `dayDifference`. Additionally the lower requirements on weekends should be incorporated. This is done by evaluating all shift rows in the selection and adding the one that best fits these requirements. The evaluation is based on the sum of squared differences between actual daily numbers and desired daily numbers of shifts per shift type.

New employees are added until the threshold of `shiftAvg` average shifts per shift type and day is met.

4.2.5 Preferences and History

The previous step already results in a complete roster for all employees. In a post-processing step this schedule is used to generate further data for the problem description. Each of the following proceedings specifies an employee configuration, which is an operation processing a complete roster for a single employee and creating a part of the problem description from it.

First shift preferences are set for employees based on the schedule. For each employee the number of preferences is chosen randomly within the boundaries `minShiftPreferences` and `maxShiftPreferences` specified in the configuration. The chosen number of days are then picked randomly from the scheduling horizon. For each chosen day, either a no-shift preference is generated if the day is free or a preference for the assigned shift type if the day is a work day. The preference weights are assigned randomly within the bounds `minShiftPreferenceWeight` and `maxShiftPreferenceWeight` specified in the configuration.

The second part is the generation of the history for a given employee. In this step the first h days are converted into fixed assignments that are added to the problem specification in the transformation phase.

If the demand type `demandType` is set to `SHIFTS`, the process stops here and skips to the transformation, where a rostering problem is produced as a result.

4.3 Break Generation

Again the break generation is done in several steps. First break types and break configurations are specified, then break schedules are created for individual shift instances according to the break types and configurations. Finally specific break schedules are assigned for all shifts in the schedule.

4.3.1 Break Types

First the number of break types specified in the configuration by `breakTypes` is created. Each break type can either be a normal break or a “lunch” break which refers to a breaks that needs to be scheduled at most one time during a shift and in a specified time window defined from the beginning of a shift.

Note that despite the name this break can occur at any time of the day. However, the time window for this break type starts somewhere between 10% and 33.3% of the maximum shift length and ends at the latest at 80% of the maximum shift length, therefore setting these break types to occur roughly in the middle of the shift. These boundaries are

provided with a random weight in the range 1 to 10. The first break type is guaranteed to be a normal break, all others are lunch breaks with the probability `lunchBreakFactor`.

For all break types the minimal and maximal length $minLength_{bt}$ and $maxLength_{bt}$ are set to be at least one time slot and at most the maximum length specified by `maxBreakLength`. Additionally, the latest time this break can be scheduled within a shift relative to the shift end is set to at least one time slot, at most 20% of the maximum shift length. Further, the work time needed before this break type can be scheduled is bounded by `minBreakDistance` and `maxBreakDistance`.

4.3.2 Break Configurations

Break configurations can be created in four different versions specified by `breakConfigurationType`.

- `COMMON`: One configuration is applied for all shifts allowing all break types.
- `PER_SHIFT_TYPE`: Each shift type gets its own configuration.
- `PER_CONTRACT`: For each contract there is a different configuration.
- `PER_SHIFT_LENGTH`: Three different configurations are created depending on possible lengths of the shift.

For all types except `COMMON` each configuration contains a random selection of break types except for the guarantee that at least one non-lunch break is included.

4.3.3 Shift Schedule Generation

The next step is similar to the shift row generation. For each generated shift instance a certain number of detailed shift schedules are created. In this process breaks are scheduled for the given shift instance according to the matched configuration. For the type `PER_CONTRACT` candidates for each contract are produced.

The detailed schedules are created by sequentially trying to assign breaks to the given shift instance according to the matching configuration and random choices.

In each iteration first possible break types are collected. Normal break types are possible if the time before the break, the break time and the distance to the shift end still fit into the remaining shift time. Lunch breaks are additionally checked for their time windows and whether they were already used in this schedule.

Next one of the possible breaks is chosen at random and start and end times for this break are calculated. These are chosen at random within the intervals defined for the chosen break type, however, the possible end of the shift or the time window of lunch breaks are taken into account to prevent start and end times that would lead to violations.

A special case occurs if the currently chosen break would result in a now possible lunch break to be impossible to schedule later. This is checked for all lunch breaks, in case a such an occurrence is found the lunch break is chosen immediately. Note that this increases the chance that a lunch break is included in all shifts if allowed by the configuration, but there is no guarantee. However, this is tracked and the break requirements are either set to at most one or precisely one lunch break depending on the results of this scheduling process later.

Finally, just like for the shift rows, depending on `breakTimeFractionConstraintType` there is a restriction process where the shift schedules are sorted and a certain percentage at the ends is dropped. This time the restriction is done on the break time fraction which is defined as $\frac{breakTime}{shiftLength}$. In addition to specifying the percentage to drop by `breakTimeFractionRestriction`, a second parameter `breakTimeFractionFocus` allows to set which percentage of the dropped data should be at the lower value range and which on the higher range. This allows to generate shifts with concentrated bounds on the break time fraction either in the higher or lower ranges of what is possible according to the break configurations. Note that this selection step is done individually for each break configuration.

4.3.4 Shift Schedule Selection

In the last step, each shift of each employee is assigned a matching detailed schedule according to the shift instance and the contract of the employee. For each break configuration the range of break time fractions that are actually used are tracked as well as the work time after the last break to use as a constraint in the transformation. Further, the usage of lunch breaks is tracked at this step as already mentioned earlier.

This phase results in a schedule for all employees with shift assignments and all break assignments according to break type and break configuration constraints. If `demandType` is set to `TIME`, the generation stops at this point and outputs a scheduling problem where the demand is given in workforce requirements per time slot, requiring shifts and possibly breaks, but no tasks to be scheduled.

4.4 Task Generation

The next phase is the task generation. In order to allow the assignment of tasks according to skills provided by employees, the generation of skills is the first step in this process, followed by the generation of tasks embedded in the schedule of shifts and breaks that is received from the earlier phases in the generator.

4.4.1 Skill Generation

The number of skills to generate is specified by `skills`. While there are many ways to deal with skills in the context of employee scheduling, the generator currently implements the following approach. Skills are given per employee, each employee can have any subset

of the defined skills. Each task requires precisely one skill of the employees working on the task.

For each skill a probability value is generated. The bounds for this value are specified by `skillMin` and `skillMax`. The value is the probability of each employee to have this specific skill. Therefore, low values lead to rare skills, high values to common skills.

Next for each employee their subset of skills is generated. To simulate a diverse pool of employees where some have a wide range of skills while others have only a limited set of skills, an additional factor is chosen between `skillDistribution` and 1. Now a random boolean dooms this employee to be either "unskilled", in which case the factor is multiplied with the probability to have a skill, or "skilled", in which case the factor reduces the probability to not have a skill. Bounding the factor to values close to one results in a rather uniform pool of employees, going closer to 0 results in very diverse skill sets. Finally, the result is recomputed if the skill set is empty resulting in every employee having at least one skill.

4.4.2 Task Generation

Tasks are scheduled within the frame of the already scheduled breaks. Therefore, it is possible to set the configuration in a way that there is no feasible task schedule in this step. However, as long as the interval between breaks and the minimum task length do not contradict or if tasks are allowed to have breaks in between, no problems arise. The following parameters guide the task generation.

- `taskTypes`: Defines the number of task types.
- `taskGenerationType`: Specifies the options of this task type regarding preemption.
- `minTaskLength` and `maxTaskLength`: These values set bounds for possible values of $length_d$ for all task demands except long tasks and background tasks.
- `commonTaskFactor`: Defines the likelihood of tasks that require multiple employees.
- `taskWindowRange`: Defines the size of the time windows for task demands, a value of 0 creates tasks with fixed execution times.
- `maxReAcquaintance`: Defines the maximal possible $reAcquaintancePeriod_{tt}$ for each task type.
- `taskDistanceRange`: This value specifies the maximum distance between tasks assigned to the same employee in the generated schedule.

First, the required number of task types is created. As most constraints are specified on a per-task base, rather than for all tasks of a task type, this does not have a lot of influence on the final problem specification. A reacquisition time needed before another

task can be assigned to the same employee is chosen per task type with the maximum possible value as specified.

However, the task type internally distinguishes different ways tasks can be scheduled. There are four different types.

- Indivisible tasks: These tasks have a length within the boundaries set in the configuration and can not be split in any way, not even by going on a break.
- Tasks allowing breaks: Same as indivisible tasks, except that they allow breaks to be scheduled in between.
- Long tasks: Those tasks have a their maximum length set to $3 \cdot$`maxTaskLength` and allow breaks as well, as they are expected to often take longer then the maximum interval between breaks.
- Background tasks: These tasks have their maximum length boundary set to half the normal setting, but they can be interrupted at any time. Other tasks or work time without task assignments might be scheduled in between the parts. This represents low priority work that can be scheduled in a wider time window whenever there is time.

`taskGenerationType` specifies up to which type this list should be supported. Additionally, while tasks require only one employee per default, some indivisible tasks might be set as common, meaning that several employees need to attend this task at the same time. The frequency of such tasks is guided by `commonTaskFactor`.

4.4.3 Task Scheduling

The task types allowed by the configuration are now scheduled for each shift in the scheduling horizon. The tasks are scheduled sequentially along the shift.

In each step at first the common tasks are checked whether one of them can be assigned. Common tasks are declared as such after scheduling a whole shift for the first five percent of the employees. There, with the specified probability, one of the tasks in this shift is declared as common at random. It is rarely the case that this task can be scheduled for another employee as shift times have to match and no breaks have to be scheduled for the assignment period as well as the required skill needs to be available. Therefore, possible common tasks are always chosen to be scheduled for further employees.

Otherwise the start time for the next task is chosen. The beginning of the first task is chosen randomly less than a complete shift time slot length into the shift. This also means that in schedules with the same time slot lengths for shifts and tasks each shift immediately starts with a task. Between tasks the time is chosen randomly within the limit `taskDistanceRange`. However, if it is still possible to schedule a task before the next break or the end of the shift without that idle time, but not any more with the idle

time, than this additional time is not added. If the current start time lies within a break, it is set to the end of the break.

Next the type of the task is chosen from the task types already described above. The length is chosen within the bounds set for the specified type. For indivisible tasks the maximal length is restricted to prevent overlap with the next break. If this is not possible at all, either a task type allowing breaks is scheduled if allowed by the configuration or the start time is increased by one time slot and the next iteration is started.

Finally the selected task is added to the shift. It might be separated by breaks, which are respected when setting the start and end times for individual parts of the task. Special care is given to the end of the shift. If the task would end after the shift, its length is reduced such that the minimal length is still respected, but the end of the task is somewhere within the last shift time slot length of the shift. If this adaptation is not possible, the task scheduling process for this shift is finished.

If possible, the task is added and the start time for the next task is set to the end time of the scheduled task plus the associated reacquaintance time if there is any. For added tasks a time window is generated according to `taskWindowRange`. This range is then split randomly and one part is attached before the start, the other part after the end of the task in order to obtain the window in which the task is allowed to be scheduled.

One post-processing step is done in order to properly include background tasks. As their parts are scheduled like individual tasks during the process, the tasks are processed sequentially. For each such task encountered, all following tasks of this type ending within 5 times the maximum task length are fused together to form one long-running background task.

4.5 Transformation

The final main part of the process is the transformation of the created schedule into both the problem formulation and the example solution. This process is just a straight-forward translation of constraints and definitions used in the generation of the schedule into the GES format as well as the transformation of the final schedule into the GES solution format.

CHAPTER 5

Solver Framework

The next goal is to provide a framework for the implementation of solvers that can be used to solve different problems specified in the GES format. This chapter describes the main components and structure of the newly developed framework for the implementation of heuristic solvers. A download will be available.[1]

As the format can specify various problems that differ in both the used demands and constraints, the focus in the optimization problem will depend on the instance. Therefore, most likely it will be too hard to provide an algorithm that can deal with all problems very well, instead the focus is to provide a possibility to implement different algorithms within the same framework to allow adaptation to various problems as well as increased reusability and reduced additional effort for applying the same algorithm to different problems.

This is possible by providing a unified constraint handling process for easy and independent implementation of new constraints, a common move structure that allows to implement various moves and reuse them in different algorithms and the possibility to design and reuse various algorithms.

The main components of the framework are as follows.

- Instance and solution representation.
- A conversion mechanism to transform instances or solutions from the specification format into the internal representation and solutions from the internal format to the specification format.
- A constraint mechanism that allows to handle constraints independently from each other.

[1] http://www.dbai.tuwien.ac.at/proj/arte/ges_solver

- A specification of moves that allows the implementation of different kinds of moves that are reusable across algorithms.

- A specification of algorithms that do the actual work utilizing the previously defined concepts.

The implementation was done in Java 8. This chapter, however, will mainly concentrate on the concepts underlying this implementation and only point to implementation details when relevant.

5.1 Instance and solution representation

The representation of the problem instance is heavily based on the GES formulation. Mainly these parts of the framework just provide the instance data and store potential solutions without much functionality.

5.1.1 Instance representation

The base class `Instance` encapsulates an instance of the problem. This class itself contains the general settings about the instance from the `General` section in the GES format. Additionally it also provides the function `isWeekend`, allowing the check whether a shift on a particular day is considered a weekend shift independent of the weekend specification as defined in the problem definition.

Further it keeps track of the history specification. If shifts before the specification period are provided, the day index starts with the earliest shift on day 0, the length of the history (in days) is h. It also provides access functions for constraints of different types that are described later.

All other definitions are encapsulated in the respective elements as shown in figure 5.1. These again correspond to the specifications in the format.

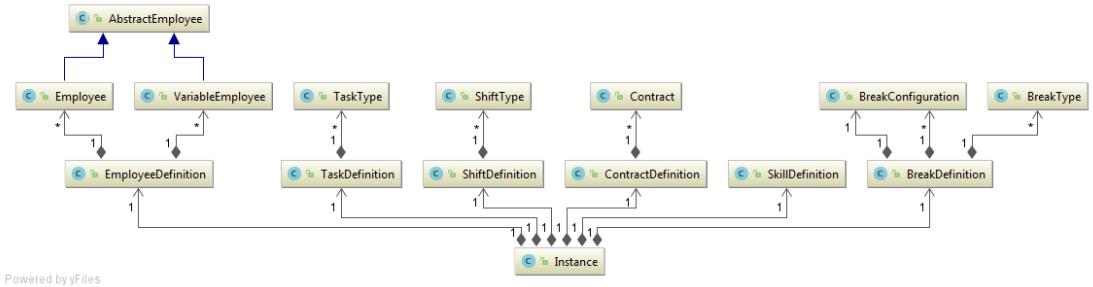

Figure 5.1: Structure of the instance representation.

The task and shift definitions contain maps of the corresponding task and shift types, each of them accessible by their ID. The constraints like start and end times of shift types are kept as lists of corresponding constraints that are described later.

The break definition contains the list of break configurations, additionally a configuration without any constraints as a fallback and the available break types. The shift filter within break configurations immediately provides the method `applies` to check whether a certain shift, assigned to a certain employee at a specific day, matches the specified filter.

The skill definition stores possible skills, but also provides a method `getSkillMap` that evaluates included skills and, given a base set of skills, provides all applicable skills including their penalties.

Employees follow their distinction between named employees and variable employees as in the format, using a common base class for universal properties.

Further one of the demand types is represented by a corresponding element. The demands are individually kept for each day, if a demand is assigned to a collection of days, the specification is replicated accordingly.

5.1.2 Solution representation

A solution or solution candidate is again represented in a similar way to the solution format. Figure 5.2 shows the structure of the solution representation.

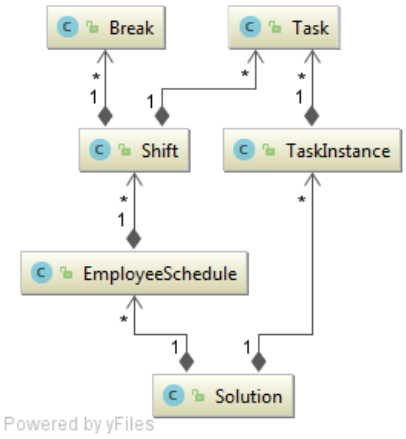

Figure 5.2: Structure of the solution representation.

Each solution is assigned the evaluation value *evaluation* and contains a roster which is a map of schedules for individual employees. These employees correspond to either named employees from the problem description or instantiations of variable employees and are identified by their ID.

Each individual schedule then contains an array of shifts with one entry for each day. An empty element corresponds to a day off, otherwise it contains a shift s including the shift type $type_s$ as well as the start and end time $start_s$ and end_s. Further a list of breaks, each break b containing the break type $type_b$ and start and end time $start_b$ and end_b, can be included.

The handling of tasks, however, is not only done like in the solution format, where a list of tasks is provided for each shift just as for the breaks. This is still an available option, but it is not flexible enough.

The reason for this is that an algorithm might not desire such a close coupling between tasks and shifts where tasks can only be assigned within shifts. Instead, it might be beneficial to separately deal with the assignment of tasks to employees, even if the corresponding employee currently has no shift scheduled at that time. The shift might then be scheduled based on the need given by the assigned tasks.

In order to allow flexible handling of tasks, first the concept of a task instance is defined. A task instance ti is defined for each task demand $d \in D_{task}$ and keeps track of the current assignment of this task. As task demands might allow preemptive scheduling, a list of task parts P_{ti} is defined where each part $p \in P$ has defined start and end times $start_p$ and end_p. Note that tasks not allowing preemption simply consist of one part. Further in many problems task start and end times are fixed leading to precisely one part with predetermined start and end times. The task instance also keeps a set of employee IDs E_{ti} corresponding to the employees the task is assigned to. The start time of the task instance is defined as $first_{ti} = start_{P_{ti}[0]}$.

For access to the set of tasks assigned to an employee, an ordered index of task instances is kept for each employee schedule. This index associates the start times of the task instances $first_{ti}$ with the corresponding demand IDs $demand_{ti}$.

Further, both tasks and shifts can be marked as fixed, meaning that algorithms are not allowed to change these assignments. These flags are used for preassigned shifts and tasks, but could also be utilized in cases like when an algorithm is expected to just work on a subproblem.

A intrinsic property of the problem is that an employee can only work one shift at a time and that in a feasible solution tasks and breaks can only appear within shifts and may not overlap. These properties are also tracked for each employee schedule by a special constraint, the overlap constraint. It is explained in more detail later.

5.2 Conversion Mechanism

In this work all problems that are provided to the solver framework are specified in the GES format. Also the internal formulation of instances, solutions and constraints is closely related to the format. However, the framework has a designated converter layer decoupling the format from the internal representation.

This allows changes in both the format and the solver framework to be carried out independently, with only the converter layer needing to be adapted to those changes. It also allows to build converters for custom specification formats, therefore using the solver framework without being bound to the GES format.

The XSD specifying the format was used to create the corresponding Java classes using Java Architecture for XML Binding (JAXB)[2]. The conversion itself is done using a hierarchy of converters all implementing a common interface method `convert(source, target)`.

The outermost evocation performs the conversion from `SchedulingHorizon`, the XML tag enclosing an instance, to `Instance`. For individual parts, further converters are used to perform the specific conversion. For the purpose of getting the right converter, a converter provider manages the set of converters and returns the corresponding converter for pairs of source and target classes.

This has the benefit that individual converters might be replaced via the converter provider. For example most constraints come with their own converter. If now a specific problem requires one of the constraints to be treated in a different way than usual, the corresponding converter can be replaced and incorporate the changes.

A conversion context is provided to all converters allowing them to access the XML data, the instance converted so far as well as the initial solution. E.g., preassignments are transformed into fixed shifts that are immediately put into the initial solution. The conversion context also provides the converter provider.

Just as for problem instances there are also converters to convert solutions from the internal representation to the XML solution format and to read existing solutions from the XML format.

5.2.1 Date and Time Conversion

Further the context also provides common date conversion and weight conversion utilities. The date and time converter is initialized using the period length p and history length h. It transforms all day and date notions into a day index starting with day 0 at the earliest day specified in the history. Further time spans are transformed from the different specification options into minutes, time points are transformed into minutes of the day. All kinds of collections of days are transformed into boolean arrays indicating which days the collection contains.

5.2.2 Constraint Weighting Strategies

The weight converter transforms all kinds of constraint penalty strategies into the format used internally. To keep this as general as possible and allow extensions going beyond the specifications in the format, this is done in a general fashion as described in (5.1).

[2] `http://www.oracle.com/technetwork/articles/javase/index-140168.html`

$$penalty = factor \cdot \texttt{function.apply}(\texttt{difference.apply}(value)) \tag{5.1}$$

`difference` is a function determining the distance to the expected state. The most common measures used as difference are as follows.

- Minimum boundary *threshold*: $\texttt{max}\{threshold - value, 0\}$
- Maximum boundary *threshold*: $\texttt{max}\{value - threshold, 0\}$
- Identity: Direct application of the constraint to *value* (equal to a maximum of 0 for $value \geq 0$).

`function` now determines how to penalize the difference. The three strategies from the format are as follows.

- `constant`: if $value > 0$ then *weight* else 0
- `linear`: $weight \cdot value$
- `quadratic`: $weight \cdot value \cdot value$

However, arbitrary different strategies like higher polynomials, logarithmic or step functions can easily be implemented.

Finally, the *factor* is an internal factor that defaults to 1 and might be used internally by the algorithm.

Further, a weight and a weighting function are only given in case of soft constraints. However, in many heuristic approaches it is beneficial to allow infeasible solutions, but to penalize violations in the evaluation function. For this purpose a hard constraint weight provider is given to the weight converter. It allows to specify the function `function` for hard constraints. The default is as follows.

if $value > 0$ then `NaN` else 0

Therefore per default hard constraint violations are not allowed. However, the provider can individually per constraint class specify different strategies and switch to penalties with arbitrary penalty functions for some or all of the hard constraints.

5.3 Constraints

The main concept behind the handling of constraints in the framework is to have all constraints obey the same structure of usage by using a common abstract class `Constraint` and a hierarchy of derived classes for specific types of constraints. Then

each constraint is treated independently without direct interaction with other constraints, but in a common process that is the same for all constraints. Therefore for each move the relevant constraints can be collected, processed and evaluated in a common way while individual constraints can easily be added, removed or replaced.

Each constraint c has access to the instance, an optional label for display and its current value $value_c$. In heuristic solvers typically there is the need to evaluate the changes a move would cause in the solution quality and then, depending on the result, either choose to execute or abort the move. Therefore, each constraint stores an additional value $newValue_c$ that represents the value of this constraint including uncommitted changes while $value_c$ represents the committed state.

The process of applying changes to a constraint is as follows.

- Incorporate changes: Depending on the type of the constraint, there are different ways to notify the constraint of changes. The constraint now incorporates these changes and updates $newValue_c$, but is able to revert the changes if necessary.
- `evaluate`: As the evaluation process is to only reevaluate constraints where it is necessary, this function returns the difference $newValue_c - value_c$.
- `execute`: If the move is accepted, constraints are told to execute the changes, meaning that $value_c$ is set to $newValue_c$ and the record of changes can be discarded.
- `abort`: If the move is not accepted, the constraints are told to revert the changes, also setting $newValue_c$ back to $value_c$.

Further constraints typically have one or more weighting strategies that are used to obtain the constraint value from the actual value of the property the constraint restricts. Note that it would be possible to use only one weight strategy per constraint, however, e.g., when there is a minimum and maximum boundary for the same property, or when there are multiple boundaries with different weighting strategies like a hard and a soft boundary for the same property, it is beneficial to only incorporate the changes once and apply all boundaries within the same constraint. Therefore, technically multiple constraints in the problem specification can be mapped to the same constraint within the framework.

5.3.1 Constraint Hierarchy

There are several different types of constraints that are shown in figure 5.3 and explained as follows. The difference in the categories is the type of changes these constraints are interested in.

- `ShiftConstraint`: This type of constraint contains two methods to add or remove a shift from the schedule together with the information which employee the shift is

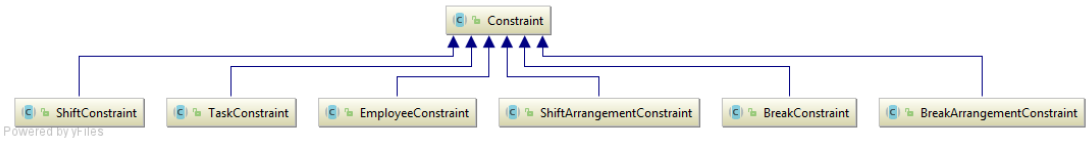

Figure 5.3: Constraint hierarchy.

assigned to. This type represents constraints dealing with individual shifts. This includes, e.g., shift start and end times or shift requirements in case the demand is given as shift demands.

- `ShiftArrangementConstraint`: This type of constraint is used when not only single shifts, but their arrangement matters for the value of the constraint. For this purpose, changes are presented to these constraints by giving all changes in an employee schedule at once passing the employee, the array of previous shifts and a map of shift changes. Presenting all changes in an affected row at once potentially allows these constraints to handle calculations more efficiently than presenting them one by one. Also for this type of constraint the surrounding shifts are important for each calculation making it necessary to pass the whole row of shifts.

  Additionally the abstract class provides methods to find the previous or next shift matching some shift filter, either before or after the changes of shifts. The filter can be any evaluation on shifts, typically filters match certain shift types, e.g., find the next day off in the employee schedule.

  Further constraints of this type have to deal with sequences that get cut off at the beginning or the end of the planning period. Therefore the abstract class provides a method to check for any time sequence whether it should be cut off (and therefore not considered for evaluation) or seen as a sequence that ends with the limit of the planning period. This selection refers to the flags *allowSequenceCutoff* in the instance definition.

  A typical example for this type of constraint is a required sequence of shifts, e.g., to have at least three working days in a row.

- `TaskConstraint`: This type of constraint contains two methods to add or remove a task instance. It is used for constraints that deal with the shape or placement of tasks. An example would be a constraint for the number of employees assigned to a task as used when task demands are specified.

- `BreakConstraint`: This type of constraint contains two methods to add or remove a break. It deals with restrictions for individual breaks like their start or end times.

- `BreakArrangementConstraint`: This type of constraint is used when not only single breaks, but the arrangement of breaks relative to each other or relative to the shift is relevant for the evaluation. For this purpose, all changes within a shift

are presented to this constraint by giving the original shift and a map of break changes. Once again this allows to process all changes at once in a more effective way if necessary for the constraint. Examples include restrictions on the working time between breaks.

- `EmployeeConstraint`: This type of constraint is used in combination with variable employees and contains two methods for adding and removing employees.

Further the abstract base class provides a range of `applies` methods with different arguments to implement restrictions for the application of constraints.

- Type restrictions: Several constraints restrict the set of shift types they are applied to, but are still not specific to just one shift type. E.g., global shift constraints like the average shift length can be restricted to arbitrary sets of shift types.
- Day restriction: Some constraints are only applied on a specified collection of days in the planning period, e.g., each Sunday. The corresponding method checks whether a day lies within the specified collection.
- Range restrictions: For some constraints the day restrictions are specified as a range of days, e.g., for weekly workload constraints having optional start and end days. Therefore the range restriction checks for a range of days whether they overlap with the range set for the constraint.

5.3.2 Constraint Handling

For each move it is important to evaluate the effects that the execution of this move has on the various constraints. On the other hand, there are often large amounts of various constraints of different types present in the current problem. Reevaluating all these constraints on each change might result in high runtimes as for example the change of a task assignment for one employee does not result in any changes in constraints regarding the shift start and end times or the sequence of days off. Therefore, it is highly beneficial to restrict the set of constraints that is reevaluted for each move.

On the other hand, constraints occur across the whole instance definition in different shapes and contexts. Requiring each move to seek and find all constraints that are required to reevaluate on their own would be a large implementation effort and discourage the implementation of new moves.

Therefore the `Instance` is the main anchor point in providing access to the relevant constraints. It provides a method for each type of constraint as described above requiring some search criteria for the affected constraints like the employee and day where the change occurs. In turn these methods return all constraints that could be affected by the move by further delegating the search to the relevant parts of the instance definition. E.g., requests for break constraints are delegated to the break definition, requests for

shift constraints are delegated both to the shift definition and to the shift demands in case demands are specified this way.

This way of handling constraints allows to significantly reduce the number of constraint reevaluations while providing simple access to the constraints for the moves.

5.3.3 Overlap Constraint

A special constraint that is not covered by the types explained previously is the overlap constraint. As shifts and tasks are scheduled by an algorithm, several undesired states might occur. As shifts can reach into the next day, several shifts might overlap (e.g., when an algorithm decides to schedule a morning shift immediately after a night shift). Further tasks assigned to an employee might overlap with each other or with scheduled breaks or tasks might be assigned to an employee not having a shift at that time.

In order to capture all these violations, the overlap constraint is used. For each employee e one such constraint exists. For each time slot i, it counts the availability of the employee $availability_e[i]$ stating the number of shifts that are assigned to this employee at time slot i. Therefore, a value of 0 means that the employee is absent, a value of 1 means that the employee is working and a value > 1 means that the employee is assigned multiple shifts at once.

Further the occupation of the employee $occupation_e[i]$ is defined as the number of tasks and breaks assigned to the employee at time slot i. Therefore, if the occupation is higher than the availability, the assignment is not feasible as this would either mean that a task is assigned at a time without a shift or too many tasks or breaks are assigned at once.

Now a violation is calculated for each time slot i as described in (5.2).

$$violation_e[i] = \texttt{max}\{availability_e[i] - 1, 0\} + \texttt{max}\{occupation_e[i] - availability_e[i], 0\} \quad (5.2)$$

The sum of these violations across the whole time horizon is considered the value of the constraint and can be penalized as any other hard constraint via the hard constraint weight provider.

The methods provided by this constraint to notify it of changes allow to add or remove a shift, therefore changing the availability, and to add or remove an occupied period of time specified by day, start and end time, therefore changing the occupation. The constraint for each employee is directly associated with the corresponding employee schedule for easy access.

Note that this constraint needs evaluation for almost every possible move. Further the execution time of the methods provided by this constraint are in a linear dependency to the number of time slots that are affected. This typically results in the main influence of time granularity on the runtime across the whole framework. While several constraints

store uncommited changes in maps or similar data structures, the overlap constraint was optimized to only use primitive data structures, in particular arrays of fixed size and pointers to elements in these arrays, as the frequent use makes other structures too slow to use.

More precisely, the array $availabilityChange_e$ stores where changes occurred, the array $availabilityOld_e$ stores the previous values and the pointer $availabilityCount_e$ counts the number of changes. Increasing the availability at time slot i now results in the execution of algorithm 5.1.

Algorithm 5.1: Efficient change history.

```
1 availabilityChange[availabilityCount] = i;
2 availabilityOld[availabilityCount++] = availability[i]++;
```

Now if the changes are executed, the index *availabilityCount* is reset to 0, otherwise the changes are restored in reversed order until the index reaches 0. These arrays exist for the occupation values as well.

5.3.4 Constraint Overview

For most of the constraints this thesis will not go into detail regarding their implementation. The available constraints are already described in the problem definition. The goal is to implement constraints as simple as possible to allow fast execution times as constraint evaluation is performed a large number of times through the execution of an algorithm. Basic constraint implementations can be as simple as follows in algorithm 5.2.

Algorithm 5.2: Shift start constraint implementation.

```
1  public class ShiftStartConstraint extends ShiftConstraint {
2      public WeightStrategy<Integer> strategy;
3      public ShiftStartConstraint(Instance instance) {
4          super(instance);
5      }
6      public void addShift(String id, Shift shift) {
7          newValue += strategy.evaluate(shift.start);
8      }
9      public void removeShift(String id, Shift shift) {
10         newValue -= strategy.evaluate(shift.start);
11     }
12 }
```

The minimum weekly rest time constraint is mentioned at this point as it allows different interpretations as already explained in the problem definition. In the current implementation shifts form work blocks whenever they are separated by a break shorter than the

minimum weekly rest time *minWeekRest*. Single shifts might form their own shift blocks. The constraint implementation now checks the length of these blocks, a violation of the constraint for an employee schedule with work blocks *WB* is computed as described in (5.3).

$$violation = \sum_{b \in WB} \texttt{max}\{length_b - (7 \cdot 24 \cdot 60 - minWeekRest), 0\} \tag{5.3}$$

Therefore, the current interpretation demands that for each period of a week over a rolling time horizon there has to be a continuous break of at least *minWeekRest* within this period.

Further in some occasions specifications in the problem definition are not explicitly stated as constraints in the sense of this framework, but can rather be read as definitions that are always respected in the solution generation. However, it might be beneficial to treat them like hard constraints in the currently presented framework of constraints and allow penalized violations in an algorithm.

This is for example implemented for fixed times off shifts specified in the preassignments of an employee. Note that employees can specify preferences to stay off shift on particular days which are obviously formulated as constraints that can either be hard or soft. However, also fixed times off shift are treated as hard constraints that might allow penalized assignments of shifts on these days within an algorithm. This allows the moves to schedule shifts without dealing with possible overlaps with those time periods on their own as violations are treated via the common constraint evaluation.

5.4 Moves

Moves are the most important building blocks of any algorithm implemented in this framework. They allow to prepare arbitrary changes to the current solution candidate, to evaluate the impact of those changes by using the constraint mechanisms described before and finally execute or discard the proposed changes depending on the decision from the algorithm.

The abstract class `Move` is the base class for each move. It gets access to the instance and offers the following methods.

- `prepare`: This method prepares the execution of the move. Moves have to offer the common prepare method and select the parameters like the employee or day that should be changed on their own. All moves currently implemented allow the specification of a selection strategy that can either perform randomized selection or follow more specific selection strategies in this process. Further moves will typically offer a prepare method requesting the required parameters for direct application of the move, e.g., to parameters that are selected by the algorithm.

The preparation includes checking whether the move can be applied at all. E.g., if a shift change shall be applied on a day without a shift, the preparation will return *false* to indicate it cannot be applied. If the preparation is successful, *true* will be returned.

Preparation will fix the parameters for the move if not already given. Further the execution of the move will be prepared, but not yet committed similar to the constraints. All relevant constraints are presented the changes via the functions specified by the corresponding constraint type. All constraints that are affected are cached for further processing.

- `evaluate`: This function triggers the corresponding evaluation function in all cached constraints and collects the results.
- `execute`: This function triggers the corresponding execution of the changes in all cached constraints and clears the cache. Further moves will commit the changes to the current solution candidate in this step.
- `abort`: This function triggers the corresponding abort of changes in all cached constraints and clears the cache. Further moves will discard all changes to the current candidate solution.

In order to simplify handling the constraints, the base class offers a method for each type of constraint that fetches the relevant corresponding constraints from the instance via the instance methods, propagates the changes to these constraints and adds them to the constraint cache.

5.4.1 Move Development

In order to reach good results, moves should be able to cover the whole search space of the problem. In the most basic version, this actually does not need a lot of different moves. It is required to add and remove shifts, to add and remove breaks (if the problem contains breaks at all) and to add and remove tasks (if the problem contains tasks at all).

However, just sticking to the basic moves will not result in good performance. This can easily be seen looking at a roster where employee e_1 is assigned shift s_1 at day i, while employee e_2 is assigned shift s_2 on that day. Now assume due to constraints the opposite assignment of s_2 to e_1 and s_1 to e_2 would be better. Clearly we can achieve this by removing both shifts and adding them back in the opposite assignment. However, it is quite possible that removing any of the shifts results in a large penalty that prevents an algorithm from going this way. Clearly, a move that immediately switches those two shifts would be beneficial.

On the other hand, adding a single shift to an employee obviously takes less time than adding a whole sequence of shifts to an employee. Therefore, when designing more complex moves, the runtime has to be considered, as in the same time more of the

primitive moves can be considered, while fewer of the potentially more useful moves can be investigated.

In the following, we propose several moves that we implemented so far and describe them along with the motivation to include them.

5.4.2 Shift Moves

The first set of moves deals with shift assignments.

- `AddOrRemoveShift`: This move implements the primitive shift move. The parameters are an employee and a day. If there is already a shift on this day, it is removed, otherwise a new shift is generated and assigned.
- `ChangeShift`: This move again takes an employee and a day as parameters. It is only applicable if a shift is assigned on the selected spot. Now the start or end time of the shift is changed within the boundaries of the shift type definition. This move is only useful if shift design is required and can handle the requirement for slight adaptation of shift times much more efficiently than removing an already well, but not ideally placed shift completely and replacing it with a new shift.
- `ChangeShiftType`: This move again takes an employee and a day and is only applicable if the selection contains a shift. This time, however, the type of the shift is changed. Removing and adding a shift would create a day off in the process that might not be desired which is prevented by this move.
- `CreateSequence`: This move takes an employee, a starting day and a length for the sequence. Then it overwrites all shifts within this sequence either with a sequence of days off, with a sequence of identical shifts or with a sequence of shifts of any type. This move is more useful, the more sequence constraints matter for this instance. Instead of hoping that randomly created shifts form a sequence, this move explicitly creates such sequences.

  Note that for random parameter selection the maximum was set to 7. Typically required sequences are not longer than this value and the runtime grows with the length while the acceptance rate gets reduced.
- `SwapShiftsBetweenEmployees`: This move takes a day and two employees and switches the shift assignments of these two employees on the selected day. The reason is to preserve the overall daily roster, i.e., the number of assignments of each shift type on this day, while moving shifts between employees.
- `SwapShiftsWithinEmployee`: This move takes two different days and one employee and switches the employee's assigned shifts on the two selected days. This preserves the overall assignments of this employee, e.g., the total workload while allowing changes for the daily rosters.

- `SwapPeriodBetweenEmployees`: This move takes a start and end day as well as a pair of employees and switches the schedules between these employees within the given interval. This can be beneficial when sequences of shifts are constrained as whole sequences can be moved at once. Again for random selection the maximum interval length is set to 7 days to prevent too runtime-intensive moves.
- `SwapPeriodWithinEmployee`: This move follows the same reason as the previous one, but changes the sequences within the same employee. Parameters are the employee, two start days and the length of the sequence to exchange. Again for random selection the maximum interval length is set to 7.
- `ReduceShiftLength`: This move accepts an employee, a day, whether to reduce start or end of the shift and the amount of reduction. The changes it performs are actually a subset of the `ChangeShift` move specifically used to reduce the length of shifts. This is used in specific occasions as described in the next section.

Note that it depends on the way shifts are created whether the given moves can reach the whole search space. While in principle every move could decide how to create or change shifts on their own, in the current implementation a common shift generator is used. This generator can create shifts in any shape within the outer hard bounds specified by the problem definition, therefore allowing to cover the whole search space regarding shifts.

5.4.3 Task Moves

Next a range of moves to deal with task assignments is presented.

- `AddOrRemoveTaskAssignment`: This move models the primitive adding and removing of task assignments. It takes a task instance and an employee as parameters. If the task instance is already assigned to this employee, it is removed, otherwise it is assigned to this employee.

  Note that several problems require each task to be assigned to exactly one employee. However, as the format and the framework allow tasks that need to be assigned to multiple employees, this possibility is also reflected in these moves.
- `ChangeTaskAssignment`: This move takes a task instance ti and a pair of employees e_1 and e_2 as input. It is applicable if the task is assigned to employee e_1, but not to e_2 and proceeds by moving the task assignment from e_1 to e_2. This skips the need to temporarily unassign the task or assign it to both employees at the same time as it would be necessary using only the primitive moves.

  Note that this move does not care whether assignments are already present for e_2 during the execution time of the task.
- `SwapTaskAssignments`: This move takes the same parameters as the previous one and also performs the same change for the assignment of the specified task. However,

this time all task assignments of e_2, where the begin time lies within the execution time of ti, are moved to e_1. This allows to swap assignments without temporarily causing too many overlapping assignments that might prevent the move.

Once more the coverage of the search space depends on the way new task assignments are generated. In this thesis only problems with non-preemptive tasks that are fixed in time are considered. Therefore, new task assignments are generated according to that. In order to cover the search space possible by the specification format, the generation would need to be extended to split tasks into several parts and choose a time within the given time windows.

5.4.4 Mixed Moves

So far all moves where dedicated to either only shifts or only tasks. However, it might also be beneficial to have combined moves. E.g., it is possible that a shift is already matched well to contain a list of tasks, but it would be better to have another employee work this whole shift including the task assignments.

This is what `SwapShiftAndTasksBetweenEmployees` does. The move takes two employees and one day as arguments just like `SwapShiftsBetweenEmployees` and swaps the assigned shifts. However, this time for each shift all tasks starting within the shift are moved to the other employee as well.

5.4.5 Break Moves

Note that breaks, unlike tasks, are directly associated with shifts and therefore immediately moved with them. This, however, does not mean that breaks and their constraints can be neglected when moving shifts. The corresponding break configuration might change depending on the shift assignment.

The problems that are evaluated in this thesis do not use breaks in the full potential the formulation allows. A shift might only have one break of a specified length. Therefore, the moves currently implemented do not cover the whole range of possibilities regarding break scheduling.

The move `FixedBreakScheduler` takes an employee and a day as input. The move is applicable if there is a shift at the selected spot. It removes all breaks that are currently scheduled and tries to find a spot where the break of fixed length should be scheduled taking into account the tasks that are scheduled for this shift.

5.4.6 Initialization

Further there is one special move which is the `Initialization`. This move is necessary for all constraints to properly initialize themselves. It does not change the given solution candidate, but it propagates the whole solution to the respective constraints.

This move is used at the beginning of an algorithm. It might be applied to an empty solution or to any given solution. In particular it can be used to evaluate a given solution and therefore check whether it is feasible as well as retrieve the solution value.

5.5 Algorithm

The framework allows the implementation of algorithms in a general way. The interface `Algorithm` contains just one method `apply(instance, solution)`. The arguments are the problem instance giving access to all the definitions and constraints and a potential solution. This might be an empty schedule or a partial or feasible solution the algorithm is given as a starting point.

An algorithm therefore does not need to do all the work on its own. It might rely on other algorithms itself that solve parts of the problem or it might just focus on certain aspects of the problem.

An algorithm can use an arbitrary selection of moves. As these moves are independent from the algorithm, they can also be reused in different algorithms. The way algorithms handle their moves and choose which one to evaluate and execute is completely up to the algorithm.

5.5.1 Solution Checker

One simple algorithm of particular importance is the solution checker. This algorithm simply performs the initialization move on the instance and solution it receives and returns the result of the evaluation. As no specific hard constraint weight provider is used, it returns `NaN` for infeasible solutions and the solution value caused by the soft constraint violations for feasible solutions.

5.5.2 Helper Algorithms

The algorithm framework can be used to design algorithms only dealing with particular aspects of the problem that might be called from another algorithm internally. Two such algorithms proved to be useful in the evaluation of the problems described in the next chapter.

The algorithm `MinimizeShifts` systematically goes through all shifts in the schedule and tries to reduce the shift length by either moving the start or end time of the shift. This is repeated as long as the solution does not get any worse. This is useful in task-based problems where periodically unused shift time can be removed in order to reduce problems with the maximum working load.

The algorithm `RemoveUnnecessaryShifts` does a similar task, but actually tries to remove whole shifts as long as this does not result in penalties for the solution. This can be useful in task assignment scenarios where shifts without matching tasks might be created.

5.5.3 Simulated Annealing

As a proof of concept a new algorithm based on simulated annealing is implemented in the framework and applied to several problems from literature as well as the instances from the instance generator.

The basic algorithm is described as algorithm 5.3.

Algorithm 5.3: Simulated annealing implementation.

```
   Data: The instance and a starting solution
   Result: The updated solution solution
 1 initialize(instance, solution);
 2 t ← t_start;
 3 changeCount ← 0;
 4 while changeCount < maxCount do
 5 |   for j ← 0 to innerIterations do
 6 |   |   move ← chooseMove();
 7 |   |   if move.prepare(solution) = false then
 8 |   |   |   continue;
 9 |   |   end
10 |   |   change ← move.evaluate();
11 |   |   if acceptMove(change) = true then
12 |   |   |   move.execute(solution);
13 |   |   |   solution.value ← solution.value + change;
14 |   |   |   if change < 0 then
15 |   |   |   |   changeCount ← max{changeCount + change, 0};
16 |   |   |   end
17 |   |   else
18 |   |   |   move.abort();
19 |   |   end
20 |   |   optionalProcessing(solution);
21 |   end
22 |   changeCount ← changeCount + 1;
23 |   t ← t · coolingRate;
24 end
25 postProcessing(solution);
```

The structure of the algorithm is the same for all the problems that are evaluated in this thesis. This highlights the reusability aspect of the framework as the same algorithm can easily be adapted to different problems. Some of the parameters, however, are changed depending on the problem in order to take care of the specific focus of each problem. These choices are further explained in the next chapter.

The initialization in line 1 creates the moves the algorithm wants to use. Further the

initialization move is executed. The temperature t is set to its starting value.

The overall structure of the algorithm consists of two main loops. The inner loop is executed a set amount of times at each temperature level. The outer loop is set to be executed as long as relevant improvements can be achieved. This is guided by *changeCount*. This parameter is increased each outer iteration, but decreased every time the current solution is improved. When this counter reaches a set value, the algorithm is stopped. The current implementation of $maxCount = 100$ ensures that a solid state has been reached once the algorithm stops.

The temperature decrease is guided by a factor *coolingRate* that is applied to the temperature each outer iteration.

Moves are chosen and evaluated within the inner loop of the algorithm. The function `chooseMove` selects a move to be evaluated for each iteration. This selection is done randomly with different probabilities for each move. The current implementation uses a `NavigableMap` for the moves with the cumulative probabilities as the key. A move can then be selected by choosing a random number in $[0; 1]$ and taking the next move in the map where the key is greater or equal to the selected number.

The chosen move is then prepared by letting the move itself choose where to apply. All currently implemented moves delegate this decision to a given selection strategy. If application is not possible, the next iteration is started.

The effect of the move on the solution value is evaluated and stored in *change*. The acceptance criterion for any move is calculated by (5.4).

$$change \leq 0 \quad \text{or} \quad \texttt{getRandom}(0, 1) < e^{-\frac{change}{t}} \tag{5.4}$$

If the move is accepted, its `execute` method is called and the solution value is updated. Further, for solution improvements *changeCount* is updated. Otherwise, the move is aborted.

At the end of the iteration, further processing of the solution might be included. E.g., periodical executions of helper algorithms like `MinimizeShifts` are possible.

After the execution of the whole algorithm, post-processing procedures might be included. Again, this might be used to reduce shift lengths or get rid of useless shifts. Further, as the algorithm and the implemented moves only use the task instance specification for task assignments, these are at this point transformed into the actual shift-based formulation that is used in the solution format.

5.6 Visualization

In order to visualize solutions, a visualization tool was developed. It parses the solution and the instance XML files and provides three views showing different aspects of the

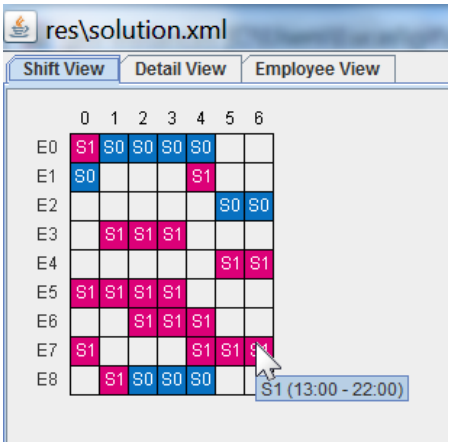

Figure 5.4: Visualization of a shift roster.

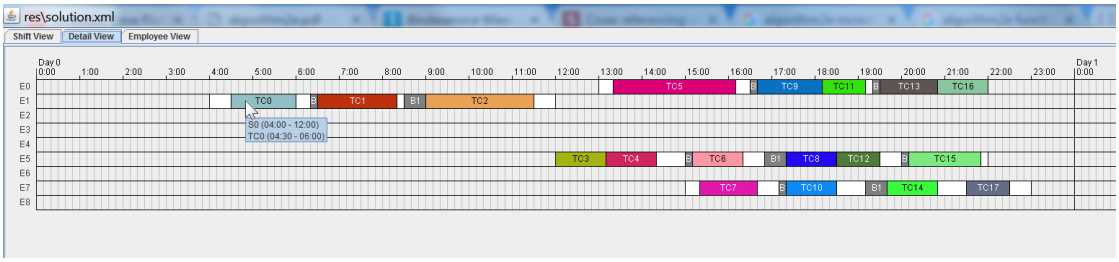

Figure 5.5: Visualization of the detailed schedule.

solution. This tool was implemented using Java Swing[3].

Figure 5.4 shows the shift roster for the scheduling horizon. Hovering over a shift shows the details of this shift as a tooltip.

Figure 5.5 shows the detailed schedule. Again hovering shows both the details of the shift as well as the details of the task or break if there is any. The time granularity of the view is adapted to the *timeSlotLength* of the schedule, however, it can also be manually set.

Figure 5.6 shows the employee schedule for the selected employee. This is useful to examine sequences of shifts or resting times between shifts.

[3] https://docs.oracle.com/javase/tutorial/uiswing/index.html

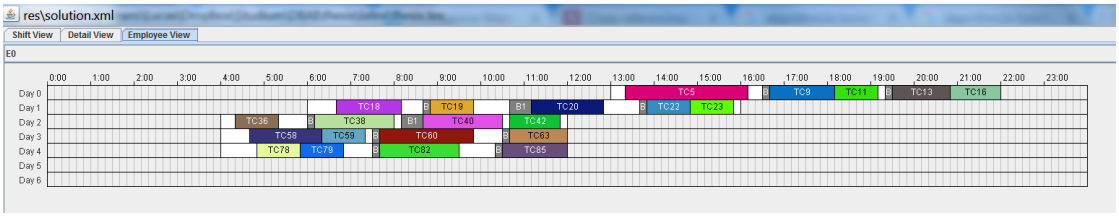

Figure 5.6: Visualization of an employee schedule.

CHAPTER 6

Evaluation

For the evaluation of the framework several different problems from literature as well as some instances from the instance generator are used. While it would be possible to develop specific algorithms for each problem that are specialized to the demands and constraints of the particular problem, the approach in this evaluation is to use the same algorithm as explained in the previous chapter and apply it to different problems. This highlights the adaptability of the framework to different problems.

Specific adaptations that were needed for the individual problems are pointed out in each section. General considerations regarding the parameter design and their evaluation are discussed before the specific problems.

All instances were evaluated on an Intel i7-6700K CPU with 4.0 GHz each using one thread. For the evaluation the algorithm was executed three times on each instance as results vary slightly from run to run, the best results are presented. All reported runtimes are in seconds.

6.1 General Aspects of Parameter Tuning

In the approach used in this thesis all hard constraint violations are penalized by using a specific hard constraint weight provider per problem. As the problems differ in their selection of constraints and the importance of the constraints, the individual weights need to be chosen separately for each problem.

However, there is a common strategy that leads to good results regarding hard constraints. The weight should be high enough that results reliably do not include violations, but not much higher than that. The reason is that otherwise the algorithm gets more restricted in executing moves as moves violating hard constraints induce higher penalties.

Further the starting temperature is directly related to the higher values of penalty weights that are used either by soft constraints or for penalizing hard constraints. The starting temperature should usually be set somewhere in the region above the largest penalty.

Values much higher lead to lots of penalized moves being executed that are undone later, just increasing runtime. Starting at temperatures too low on the other hand makes moves violating those constraints very unlikely and potentially results in a bad coverage of the search space as bounds set by constraints with high penalties cannot be overcome by the algorithm.

The selection of the number of inner iterations and the cooling rate are typically representing the tradeoff between runtime and solution quality. The more time on each temperature level the algorithm spends, either by using slow cooling or many inner iterations, the better the exploration of the search space typically gets, but on the other hand this process takes more time.

In this evaluation the number of inner iterations will depend on the size of the instance to scale the runtime of the algorithm depending on the instance size. The standard value is calculated as follows, where *outer* is the number of outer iterations.

$$p \cdot |E| \cdot outer \tag{6.1}$$

Therefore, the number depends on the size of the roster, further more time is spent on lower temperatures. As the number of possible moves is very large and towards the end of the algorithm only few moves can lead to an improved solution, the algorithm spends more time there trying to still find improvements by more thorough exploration of possible moves.

The same can be true for different moves, where simple moves are very fast to execute and change only small parts of the solution, but more complicated, yet slower moves might allow to overcome barriers in the search space where simple moves struggle. Therefore, simple moves (those that only change a single shift or task) are executed 100 times more often, unless stated otherwise for a problem, to keep runtimes reasonable while still allowing complicated moves in the process. Note that while not all problems use all available moves, the implementation of the moves is the same for all evaluated problems.

The selection strategy for deciding where to apply a move is random selection, as different attempts biased towards areas with more constraint violations either did not result in significant improvements or took too long to decide.

6.2 Nurse Rostering

While the focus of the other problems evaluated in this thesis are task demands where the combination of rostering and task scheduling makes up the main challenge, an evaluation on a set of nurse rostering benchmark instances was performed to evaluate the performance regarding these kinds of problems.

6.2.1 Problem Selection

For the evaluation the Nottingham instances provided by [Cur17] where evaluated. These provide a set of 24 generated instances ranging from 2 to 52 weeks, 8 to 150 employees and up to 32 shift types.

Shifts are fixed in time, therefore, no shift design is necessary. The instances provide the following constraints.

- Forbidden shift sequences (length 2)
- Maximal number of assignments per shift type for each employee
- Minimal and maximal total workload
- Minimal and maximal number of consecutive shifts
- Minimal number of days off
- Maximal number of working weekends
- Fixed days without shifts
- Shift requests for particular shifts with different weights $[1; 3]$
- Shift off requests for particular shifts with different weights $[1; 3]$

The demands are given as shift cover with penalties of 100 for lower levels and penalties of 1 for higher levels.

All the constraints can directly be modelled in the GES format.

6.2.2 Parameter Tuning

This evaluation uses all shift moves described in section 5.4.2 except `ChangeShift` and `ReduceShiftLength` as shift times are predefined. The fast and simple moves `AddOrRemoveShift` and `ChangeShiftType` are used 10 times as often as the others as they are faster, but result in less change in the potential solution.

Following the strategy of repeated increases in hard constraint penalties until feasible solutions are reached, the following weights were chosen as penalties. All weighting strategies are linear.

- `WorkloadConstraint`: 100 (per minute of violation)
- `ShiftSequenceConstraint`: 1000
- `ShiftCountConstraint`: 1000

- `ForbiddenSequenceConstraint`: 1000
- `WeekendCountConstraint`: 1000
- `NoShiftConstraint`: 1000

The number of inner iterations is kept lower and with an additional upper bound leading to

$$\texttt{max}\left\{\frac{p \cdot |E| \cdot outer}{100}, 100000\right\} \tag{6.2}$$

as some of the larger instances lead to exorbitant runtimes otherwise.

The starting temperature was set to 100000 as lower temperatures still lead to early local optima in several cases. In order to still keep the runtime in reasonable bounds, the cooling rate was set to 0.99. Lower values freeze the roster faster, potentially resulting in worse results, higher values increase the runtime further.

6.2.3 Results

Table 6.1 shows the results of the evaluation in comparison with the best known results. Results in bold are proven optimal results.

The results show that for most instances except the very large ones (20 to 24) the algorithm can find good results in comparably fast runtime, while in the average they are only 28% worse than the best known solutions.

This builds a promising base for more specialized developments of rostering algorithms in the framework or an extended evaluation given that several of the best known solutions were computed in hundreds of hours according to the changelog on [Cur17].

6.3 Generated Instances

In this section a test set of instances created by the instance generator described in chapter 4 is evaluated. In contrast to the other instances evaluated in this chapter they all allow a feasible solution with no soft constraint violations, therefore an optimum value of 0.

While the generator allows a wider range of possible configurations, for the evaluation a set of instances with task demands for non-preemptive tasks was created. Break scheduling is not considered. However, the instances include shift design within set boundaries for different shift types. Each shift type allows shift design within certain bounds. Further, sequence constraints are present for both shifts and days off.

A set of three skills is defined with different distributions of skills among employees. Each task requires one of these skills. Most tasks require one employee, in contrast to later

Instance	Result	Time	Feasible	Best known	% difference
Instance1	613	7	yes	**607**	1.0
Instance2	929	12	yes	**828**	12.2
Instance3	1024	18	yes	**1001**	2.3
Instance4	1736	19	yes	**1716**	1.2
Instance5	1450	34	yes	**1143**	26.9
Instance6	2367	39	yes	**1950**	21.4
Instance7	1102	43	yes	**1056**	4.4
Instance8	1716	76	yes	**1300**	32.0
Instance9	538	80	yes	**439**	22.6
Instance10	4992	141	yes	**4631**	7.8
Instance11	3705	183	yes	**3443**	7.6
Instance12	4564	481	yes	**4040**	13.0
Instance13	2828	2999	yes	**1348**	109.8
Instance14	1780	164	yes	**1278**	39.3
Instance15	5445	316	yes	3834	42.0
Instance16	4271	137	yes	**3225**	32.4
Instance17	7858	217	yes	**5746**	36.8
Instance18	7038	294	yes	**4459**	57.8
Instance19	5110	543	yes	3149	62.3
Instance20	*12316*	2204	no	4943	
Instance21	*25565*	5359	no	21159	
Instance22		-		33155	
Instance23		-		17428	
Instance24		-		48777	

Table 6.1: Results on the Nottingham instances.

problems some require multiple employees at once. Tasks are up to 8 hours long. There are full time and part time employees with different constraints as follows.

- Minimal and maximal number of consecutive working days
- Minimal and maximal number of consecutive days off
- Maximal workload over the planning period
- Forbidden sequences of length 2

Instances differ in the period length, the time slot length, the number of shift types, the distribution of skills and the presence of history data.

Days	*timeSlotLength*	$\|S\|$	Skilling	History	Optimal	Penalty	Time
7	60	2	Common	No	5	-	78.4
7	10	2	Common	No	4	100	141.0
7	60	5	Common	No	4	100	258.6
7	60	2	Diverse	No	5	-	61.2
7	60	2	Common	Yes	5	-	75.6
28	60	2	Common	No	4	300	596.4
28	10	2	Common	No	4	180	771.8
28	60	5	Common	No	2	167	2415.8
28	60	2	Diverse	No	5	-	708.6
28	60	2	Common	Yes	5	-	522.2

Table 6.2: Results on the generated instances.

6.3.1 Parameter Tuning

For this problem all moves except for the generation of breaks are used. Hard constraints are penalized as follows by the usual procedure.

- `ShiftStartConstraint`, `ShiftEndConstraint`: 10
- `ShiftSequenceConstraint`: 100
- `ForbiddenSequenceConstraint`: 100
- `TaskRequirementConstraint`: 100
- `OverlapContraint`: 2 (per minute of violation)
- `WorkloadConstraint`: 0.5 (per minute of violation)

The algorithm uses a starting temperature of 1000, a cooling rate of 0.995 and the usual amount of inner iterations.

6.3.2 Results

For each configuration 5 instances were created, leading to 50 instances in total. Table 6.2 presents the results per category. Penalty values are calculated as the average over non-optimal results only.

The algorithm can find optimal results for 43 out of 50 instances. As expected, the larger instances with four weeks both take longer and are harder to solve optimally compared to the smaller instances.

As per category only one property is changed compared to the first category for each period length, the effects of individual settings can be evaluated. Increasing the time granularity

in combination with more, but shorter tasks ($timeSlotLength = 10$) significantly increases runtime and reduces the number of optimal solutions that are found.

Increasing the number of shift types along with the number of employees ($|S| = 5$) shows the largest effect both on runtime and the probability to stop before the optimal result.

Both making skill distribution more diverse and providing a history, on the other hand, did not make the solutions any worse, nor did they result in significant increases of the runtime.

6.4 Integrated Task Scheduling and Personnel Rostering Problem

This section evaluates the framework on the TSPR as defined in [SEVB16]. In this problem the demands are specified as task demands which are fixed in time and not preemptive. Possible shift types are also given and fixed in time. Further, a set of employees is specified and for each employee the set of possible tasks is defined.

The period length is either 7 or 28 days, the number of employees ranges from 10 to 40 and there are 4 different shift types. The following constraints are defined as soft constraints with a weight of 1.

- Minimal and maximal number of worked days per employee
- Minimal and maximal number of assignments to each shift type per employee
- Maximal number of consecutive working days
- Minimal and maximal number of consecutive days off
- Complete weekends, i.e., either shifts on both Saturday and Sunday or both days free
- Forbidden shift sequences (length 2)

The task demands are considered hard constraints. Further the instances are generated with different parameters of skilling, which defines how many tasks each employee can perform, as well as the tightness of the instance.

6.4.1 Modelling the Problem in the GES Format

Most of the demands and constraints can directly be transformed into the GES format. The specification of the set of tasks each employee can perform was transformed into a set of skills. Each task requires a unique skill and for each employee a set of mastered skills corresponding to those specified tasks is given.

6.4.2 Parameter Tuning

For this problem all defined moves regarding shifts and tasks are used except `ChangeShift` and `ReduceShiftLength` which only apply to problems with shift design.

As the weights for soft constraint violations are low, for the hard constraints a weight of 10 for the task requirements and a weight of 2 for the overlap constraint (per minute of violation) was sufficient to get feasible results for most instances.

Corresponding to low weights for constraint violations, a starting temperature of 100 was used together with the standard amount of inner iterations. The cooling rate was set to 0.99 in order to restrict the runtime to the values used in the compared paper. Here, the runtime was restricted to 1 hour per instance. With the current parameter setting this is also respected in this evaluation.

6.4.3 Results

For the evaluation a set of 360 instances is available. Table 6.3 shows the results of the algorithm in comparison with the results presented in [SEVB16]. For each category 10 instances were evaluated, the average results are presented. Results in bold indicate proven optimal solutions. Results in italics indicate that they were only computed over feasible solutions.

The results show that for 327 out of 360 instances a feasible solution can be found. The compared work finds feasible solutions for all instances. The results show that almost all problems occur on high skilling levels, especially for large instances. This indicates where further improvements should focus.

As the execution time is connected to the size of the instance, for all but four categories our approach produces results significantly faster in comparison. However, the results regarding soft constraint violations are not yet competitive in most cases. For several small instances the results get very close to the best known solutions, for others there is still a gap to cross. Note that for the category with the highest result of 1211 in their approach, our average is better, however, only calculated over the feasible instances. This might indicate that their algorithm ran into the runtime boundary too fast potentially allowing our algorithm to provide better results given the feasibility issues can be resolved.

In total the results show that our approach can easily be applied to this problem and provides reasonable results for a general purpose algorithm. Therefore we see potential in applying our framework to this problem with more specialized algorithms to get competitive results.

6.5 Shift Design Personnel Task Scheduling Problem

The SDPTSP-E is defined in [LBMP13]. This problem is based on a company performing drug evaluation and pharmacology research, therefore following the need for strict testing

				Our results			Their results	
Days	$\|E\|$	Tightness	Skilling	Result	Time	% feasible	Result	Time
7	10	0.6	0.3	25.4	94.6	100	**21.3**	0.8
7	10	0.6	0.6	10.2	87.3	100	6.6	915.0
7	10	0.6	1.0	4.9	87.9	100	**3.1**	446.6
7	10	0.9	0.3	33.3	89.5	100	**30.5**	0.1
7	10	0.9	0.6	*36.8*	96.3	80	19.9	1316.2
7	10	0.9	1.0	24.7	98.2	100	**8.3**	1712.3
7	20	0.6	0.3	20.2	250.3	100	9.4	3600
7	20	0.6	0.6	11.9	239.5	100	1.5	1374.3
7	20	0.6	1.0	12.1	200.5	100	1.9	2622.1
7	20	0.9	0.3	*66.3*	197.6	80	45.4	3600
7	20	0.9	0.6	75.4	195.1	100	34.5	3600
7	20	0.9	1.0	63.4	201.1	100	24.0	3600
7	40	0.6	0.3	29.6	450.5	100	11.6	3591.4
7	40	0.6	0.6	23.7	448.2	100	0.7	3600
7	40	0.6	1.0	20.5	443.5	100	**0.0**	3600
7	40	0.9	0.3	*188.3*	424.0	80	135.0	3600
7	40	0.9	0.6	163.8	430.4	100	113.5	3600
7	40	0.9	1.0	157.4	437.1	100	50.0	3600
28	10	0.6	0.3	*100.9*	385.9	90	**76.5**	11.1
28	10	0.6	0.6	52.6	397.4	100	23.4	3600
28	10	0.6	1.0	36.8	419.1	100	12.5	3600
28	10	0.9	0.3	155.4	397.9	100	**129.5**	1.1
28	10	0.9	0.6	*157.0*	442.0	10	111.4	3600
28	10	0.9	1.0	*168.6*	399.8	90	89.0	3600
28	20	0.6	0.3	108.6	872.9	100	68.0	3600
28	20	0.6	0.6	65.8	903.9	100	20.8	3600
28	20	0.6	1.0	62.8	916.4	100	26.7	3600
28	20	0.9	0.3	*398.0*	907.0	10	324.0	3600
28	20	0.9	0.6	*457.0*	947.3	60	321.2	3600
28	20	0.9	1.0	461.5	994.1	100	268.7	3600
28	40	0.6	0.3	145.6	2143.4	100	108.9	3600
28	40	0.6	0.6	127.7	2161.6	100	68.5	3600
28	40	0.6	1.0	113.1	2146.3	100	16.7	3600
28	40	0.9	0.3	*1053.7*	2674.4	70	1211.0	3600
28	40	0.9	0.6	1032.3	2875.0	100	857.3	3600
28	40	0.9	1.0	993.3	2623.8	100	541.6	3600

Table 6.3: Results on the TSPR instances.

protocols that need to be followed to the minute in order to comply to the regulations. It contains task demands, requires shift design and even break scheduling. It also defines a special fairness constraint.

The problem is given with a period length of one week and the number of tasks ranging from 100 to 400. Non-preemptive tasks are given with fixed start and end times. The tightness, referring to the task workload per worker, is varied among instances. A set of skills is defined, either with only common skills or 5% rare skills that are only mastered by 20% of the workers.

Tasks are distributed according to the industrial background with 50% of them occurring in the morning with a peak around 8 am, 40% in the evening and 10% at night. Tasks have a probability of 10 % to occur on the weekend and are distributed across different lengths from 5 minutes to 5 hours with the peak around one hour.

A working day in this definition starts and ends at 6 am. Tasks starting in different days according to this definition belong to different daily schedules.

The following hard constraints are provided.

- Maximal daily duration of 11 h
- Maximal daily working time of 10 h
- Maximal weekly working time of 48 h
- Minimal daily rest time of 11 h
- Minimal weekly rest time of 35 h
- Maximal number of consecutive working days of 6

Further there are constraints regarding breaks depending on the shift. For each employee a history regarding work assignments in the previous week is provided, as well as a list of mastered skills and time intervals where the employee must not be assigned.

Each employee might also have compulsory tasks that do not count as clinical work, but that are predefined and have to be assigned.

The primary goal is to assign all tasks, the secondary goal is defined as a measure of fairness between employees. In times when no clinical tasks are performed, the employees are expected to perform administrative duties that do not follow a strict schedule. As the levels of administrative work for each employee differ, a targeted clinical workload is assigned to each employee. The employee should do clinical tasks in order to get as close as possible to this targeted workload, leaving the rest of their time for administrative work. This is called the equity constraint. Its value is defined by (6.3), where w_e is the clinical workload assigned to employee e and c_e is the targeted clinical workload for employee e.

$$violation_e = \max_{e \in E}(w_e - c_e) - \min_{e \in E}(w_e - c_e) \tag{6.3}$$

The secondary goal is now defined as the minimization of $violation_e$.

6.5.1 Modelling the Problem in the GES Format

Unlike the previous problems, this one needs some more preparations to transform it into the GES format. Most of the given hard constraints, however, are easy to transform. Both the daily duration and daily working time are transferred into `ShiftLengths` constraints within a contract using different settings for the unit. Minimum daily and weekly rest time can directly be transferred as well as the shift sequence constraint. Note that the data description[1] indicates a rolling horizon for the minimum weekly rest time as currently implemented in the solver framework, while constraint (6) in [LBMP13] indicates the minimum rest time can occur anywhere within each calendar week.

First problems arise when trying to model the history. The GES format provides a simple way to specify the history by directly giving the previous schedule as preassignments in the instance. The SDPTSP-E format gives the history as the number of days worked since the last day off, the number of minutes since the last weekly break, and the number of minutes since the start respectively end of the last shift of the previous week. However, using this specification it is possible to give conflicting values and this seems to be the case for several of the instances. The other option would be an error in the interpretation of the given data on our side. Either case promotes the use of the GES format where the XML format allows easier reading of the instances for humans as well as a history specification that reduces the possibility of inconsistent formulations. In case the conversion ran into conflicts, the last shift of the previous week is included as specified, the given number of days since the last day off is then added backwards starting from this last shift.

Next the assignment of tasks needs to be considered. The problem specification contains studies. Each task is assigned to one study and employees might not be allowed to work on all studies. This is simply translated to a further set of skills. Now each task requires an employee having both the correct skill and the correct study-skill.

A bigger problem is the assignment of tasks only to shifts of the same daily schedule. The purpose of this constraint is to prevent shifts starting in the middle of the night and continuing along the following day. The result is that tasks starting at 6 am or later must not be assigned to night shifts reaching out from the previous day. On the other hand night shifts might extend far beyond 6 am when, e.g., a task goes from 5 to 9 am.

Therefore, the `TaskToShiftConstraint` is implemented in the algorithm. It contains methods to add or remove a shift as well as to add or remove a task and counts the number of tasks starting at 6 am or later assigned to night shifts on the previous day.

[1] `https://sites.google.com/site/ptsplib/`

The next step to consider is break scheduling. The original problem formulation contains different breaks for different shifts. However, the authors chose to only focus on lunch breaks as the employees are very flexible regarding their breaks. The considered requirement is that shifts starting before 12:00 and ending after 14:30 with a length of more than 5 hours should have a lunch break of one hour. These requirements can be transformed into a break configuration immediately. The placement of the break, however, is not considered directly in the compared work. Instead, as long as the task assignments of the shifts spare one hour of shift time for the break, the requirement is considered as fulfilled. The break time does not need to be in one block.

In this evaluation we chose to model the break as one block of one hour that can be placed anywhere in a matching shift. Note that the formulation in the compared work would have been possible as well by allowing breaks of arbitrary length with a sum of precisely one hour, however, this would have been more difficult to schedule in the given framework than one hour as a block.

Further note that in the GES format and our framework, even the original more complex break definitions could be modelled without any further adjustments except development of corresponding new moves that can handle more sophisticated break scheduling. However, in the evaluation we wanted to stay close to the original formulation for comparison.

Finally the original instances also contain some further information like shift preferences without information on how to weight them or notions of flexibility of tasks, simple tasks or preaffected workers. As these are not mentioned in the corresponding papers, we did not include them in our evaluation.

The `EquityConstraint` also needed to be implemented in the framework. It contains methods to assign a task to an employee or remove such an assignment. It keeps track of the currently assigned amount of clinical workload and the targeted workload for each employee and therefore can compute $violation_e$.

Note that both new constraints do not immediately fit into the constraint hierarchy where `TaskConstraint` deals with tasks without caring about their specific assignment to shifts or employees. Therefore, the new constraints each form their own type. This results in the moves having to propagate changes to these constraints separately. However, this just amounts to one line of code per constraint and move.

6.5.2 Parameter Tuning

This problem uses all available moves except those that change sequences of shifts at once. The reason is that at minute time granularity these are rather slow while in this problem shift sequences play only a minor roll, as there only needs to be one free day per week in order to fulfil those sequences.

The number of breaks per shift is penalized by 10, overlap violations by 10 per minute of overlap and workload violations by 0.5 per minute of violation. All other hard constraints

		Our results				Their results			
$\|D_{task}\|$	Tightn.	Compl.	Ineq.	% ass.	Time	Compl.	Ineq.	% ass.	Time
100	600	50	33	98.9	54	53 / 54	28	97.6	145
100	800	20	29	98.6	41	42 / 47	35	97.8	155
100	1000	0	-	96.5	29	11 / 21	72	96.7	167
200	600	58	34	99.2	125	59 / 60	34	99.0	166
200	800	32	27	99.2	94	50 / 55	35	98.6	138
200	1000	1	19	98.0	73	22 / 35	42	97.7	156
300	600	52	37	99.2	238	58 / 58	40	99.7	191
300	800	35	38	99.4	164	53 / 58	38	99.2	186
300	1000	4	16	99.0	135	42 / 56	46	98.8	173
400	600	51	55	98.7	393	59 / 59	47	99.8	236
400	800	42	42	99.5	260	55 / 59	44	99.7	202
400	1000	5	54	99.1	203	40 / 56	51	99.0	196

Table 6.4: Results on the SDPTSP-E instances.

have a weight of 100.

The starting temperature is also set to 100 in combination with slow cooling of 0.995. The number of inner instances is as defined in (6.1), however, divided by 10 in order to stay close to the computation time of 5 minutes per instance as in the compared work.

6.5.3 Results

Table 6.4 shows the results of the evaluation. In total there are 720 instances, for each category as listed in the table there are 30 instances with only common skills and 30 instances including rare skills.

The second number in the compared complete results represents the maximum possible number of complete solutions in this category, for the others it is proven that no complete solution exists. Inequity values are only calculated across complete results, the percentage of assigned tasks only over non-complete results.

As a disclaimer, the comparison might not be fully accurate due to some uncertainties mentioned in the conversion process as well as the slightly different handling of breaks. Nevertheless, the results offer a good indication of the performance of the algorithm.

The evaluation shows good results on the given instances. While the number of complete instances is lower, especially in instances with high tightness, the evaluation of both the inequity on complete instances as well as the percentage of assigned tasks on incomplete instances shows competitive results in comparison. Therefore, some further improvements targeted towards resolving those few tasks that cannot be assigned might very well lead to competitive overall results.

The runtime is lower in 8 out of 12 categories, with only one of our categories exceeding the targeted runtime of 5 minutes. However, in many of the smaller instances we can reach a comparable level of results in significantly shorter runtime.

CHAPTER 7

Conclusion

This thesis gave a contribution to the formulation of a wide range of employee scheduling problems covering different types of demands and including shift design, break scheduling and task scheduling as well as a wide range of different constraints.

A new framework was developed that allows independent handling of various constraints in a unified way, promoting easy addition or change of constraints. A common way of implementing and handling moves was provided that allows easy integration of new moves as well as their reusability across different algorithms. A new general purpose simulated annealing algorithm and a set of moves were implemented in the framework.

To evaluate the framework, the algorithm was applied to several different problems. Well-known benchmark instances from literature were transformed into the GES formulation, where the formulation proved to be applicable to a wide range of different specifications. Further an instance generator with a large set of configuration parameters was developed that allows the creation of various new benchmark instances including combinations of constraints that have not yet been investigated in literature.

Finally the algorithm was successfully applied to both the problems from literature and several newly generated instances with low adaptation effort. The algorithm provided solid results for all problems and could even incorporate new constraints like the equity constraint with very good results.

This offers a range of possibilities for future research in this area. The instance generator can be used to generate more complex instances than the ones investigated in this thesis. This includes instances with a mixture of non-preemptive and preemptive tasks as well as different break types in varying configurations in various combinations with or without shift design or strict sequence constraints.

The problem formulation proved to be applicable to a wide range of problems, therefore, translating further problems and applying the solver framework is another area with

potential. Regarding the solver framework itself, new algorithms should be implemented either specialized to particular problems to push for new, better results to benchmark instances, or to improve the widespread applicability of a general purpose solver. New, more sophisticated moves might be implemented in order to improve results across various algorithms.

List of Figures

List of Tables

List of Algorithms

Acronyms

ETP Employee Timetabling Problem. 7

GES General Employee Scheduling. ix, xi, 2, 3, 5, 9, 24, 25, 35, 37, 38, 40, 41, 61, 65, 69, 70, 73

JAXB Java Architecture for XML Binding. 41

PTSP Personnel Task Scheduling Problem. 7

SDPTSP-E Shift Design and Personnel Task Scheduling Problem with Equity Objective. 7, 66, 69, 71, 77

SMPTSP Shift Minimization Personnel Task Scheduling Problem. 7

TSP Tour Scheduling Problem. 6

TSPR Integrated Task Scheduling and Personnel Rostering Problem. 7, 65, 67, 77

XML Extensible Markup Language. 2, 3, 9, 24, 41, 55, 69

XSD XML Schema Definition. 9, 41

Bibliography

[Alf04] Hesham K. Alfares. Survey, Categorization, and Comparison of Recent Tour Scheduling Literature. *Annals of Operations Research*, 127(1-4):145–175, March 2004.

[BC14] Edmund K. Burke and Tim Curtois. New approaches to nurse rostering benchmark instances. *European Journal of Operational Research*, 237(1):71–81, August 2014.

[BDCBVL04] Edmund K. Burke, Patrick De Causmaecker, Greet Vanden Berghe, and Hendrik Van Landeghem. The State of the Art of Nurse Rostering. *Journal of Scheduling*, 7(6):441–499, November 2004.

[BGM+10] Andreas Beer, Johannes Gartner, Nysret Musliu, Werner Schafhauser, and Wolfgang Slany. An AI-Based Break-Scheduling System for Supervisory Personnel. *IEEE Intelligent Systems*, 25(2):60–73, March 2010.

[BQB11] Peter Brucker, Rong Qu, and Edmund Burke. Personnel scheduling: Models and complexity. *European Journal of Operational Research*, 210(3):467–473, May 2011.

[CQ14] Tim Curtois and Rong Qu. Computational results on new staff scheduling benchmark instances. Technical report, ASAP Research Group, School of Computer Science, University of Nottingham, NG8 1BB, Nottingham, UK, October 2014.

[Cur17] Timothy Curtois. Employee shift scheduling benchmark data sets. `http://www.schedulingbenchmarks.org/`, 2017. Accessed: 2018-01-22.

[DBVdBBD15] Philippe De Bruecker, Jorne Van den Bergh, Jeroen Beliën, and Erik Demeulemeester. Workforce planning incorporating skills: State of the art. *European Journal of Operational Research*, 243(1):1–16, May 2015.

[EJKS04] A.T Ernst, H Jiang, M Krishnamoorthy, and D Sier. Staff scheduling and rostering: A review of applications, methods and models. *European Journal of Operational Research*, 153(1):3–27, February 2004.

[GK03] Fred Glover and Gary A Kochenberger. *Handbook of Metaheuristics*. Springer US, Boston, MA, 2003. OCLC: 903188846.

[GM86] Fred Glover and Claude McMillan. The general employee scheduling problem. An integration of MS and AI. *Computers & Operations Research*, 13(5):563–573, January 1986.

[KE01] Mohan Krishnamoorthy and Andreas T. Ernst. The Personnel Task Scheduling Problem. In Panos M. Pardalos, Donald Hearn, Xiaoqi Yang, Kok Lay Teo, and Lou Caccetta, editors, *Optimization Methods and Applications*, volume 52, pages 343–368. Springer US, Boston, MA, 2001. DOI: 10.1007/978-1-4757-3333-4_20.

[KEB12] M. Krishnamoorthy, A.T. Ernst, and D. Baatar. Algorithms for large scale Shift Minimisation Personnel Task Scheduling Problems. *European Journal of Operational Research*, 219(1):34–48, May 2012.

[KLPS07] Antoon W.J. Kolen, Jan Karel Lenstra, Christos H. Papadimitriou, and Frits C.R. Spieksma. Interval scheduling: A survey. *Naval Research Logistics*, 54(5):530–543, August 2007.

[KMM+17] Lucas Kletzander, Florian Mischek, Nysret Musliu, Gerhard Post, and Felix Winter. A general modeling format for employee scheduling. Technical report, Database and Artificial Intelligence Group, Institut für Informationssysteme, TU Wien, March 2017. `http://www.dbai.tuwien.ac.at/proj/arte/`.

[LBMP13] Tanguy Lapègue, Odile Bellenguez-Morineau, and Damien Prot. A constraint-based approach for the shift design personnel task scheduling problem with equity. *Computers & Operations Research*, 40(10):2450–2465, October 2013.

[LJ91] John S. Loucks and F. Robert Jacobs. Tour Scheduling and Task Assignment of a Heterogeneous Work Force: A Heuristic Approach. *Decision Sciences*, 22(4):719–738, September 1991.

[MS03] Amnon Meisels and Andrea Schaerf. Modelling and Solving Employee Timetabling Problems. *Annals of Mathematics and Artificial Intelligence*, 39(1):41–59, September 2003.

[MSS04] Nysret Musliu, Andrea Schaerf, and Wolfgang Slany. Local search for shift design. *European Journal of Operational Research*, 153(1):51–64, February 2004.

[Mus06] Nysret Musliu. Heuristic methods for automatic rotating workforce scheduling. *International Journal of Computational Intelligence Research*, 2(4):309–326, 2006.

[PLBM15] D. Prot, T. Lapègue, and O. Bellenguez-Morineau. A two-phase method for the shift design and personnel task scheduling problem with equity objective. *International Journal of Production Research*, 53(24):7286–7298, December 2015.

[SEVB16] Pieter Smet, Andreas T. Ernst, and Greet Vanden Berghe. Heuristic decomposition approaches for an integrated task scheduling and personnel rostering problem. *Computers & Operations Research*, 76:60–72, December 2016.

[SWMVB14] Pieter Smet, Tony Wauters, Mihail Mihaylov, and Greet Vanden Berghe. The shift minimisation personnel task scheduling problem: A new hybrid approach and computational insights. *Omega*, 46:64–73, July 2014.

[VdBBDB+13] Jorne Van den Bergh, Jeroen Beliën, Philippe De Bruecker, Erik Demeulemeester, and Liesje De Boeck. Personnel scheduling: A literature review. *European Journal of Operational Research*, 226(3):367–385, May 2013.

[VM09] Mario Vanhoucke and Broos Maenhout. On the characterization and generation of nurse scheduling problem instances. *European Journal of Operational Research*, 196(2):457–467, July 2009.