Solving the General Employee Scheduling Problem[☆]

Lucas Kletzander[a,*], Nysret Musliu[a]

[a] *TU Wien, Karlsplatz 13, 1040 Vienna, Austria*

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. 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 a new framework is proposed 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.

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 with good results.

Keywords: Rostering, Task Scheduling, Metaheuristics

[☆]This work was supported by the Austrian Science Fund (FWF): P24814-N23. The financial support by the Austrian Federal Ministry for Digital and Economic Affairs and the National Foundation for Research, Technology and Development is gratefully acknowledged.

[*]Corresponding author

Email addresses: lkletzan@dbai.tuwien.ac.at (Lucas Kletzander), musliu@dbai.tuwien.ac.at (Nysret Musliu)

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 [1, 2]. 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.

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.

The main contribution of this paper is a new framework 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.

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 instances from a new instance generator. The instances from literature cover nurse rostering [3] as well as different problems involving tasks

from [4] and [5].

The remainder of this thesis is organized as follows. In section 2 an overview of related work in employee scheduling is presented. In section 3 the problem definition is presented. Section 4 explains the structure of the framework and its components. In section 5 the evaluation of the framework on the generated instances and the instances from literature is presented. Section 6 provides a summary and an outlook for possible future work.

2. Related Work

Many different versions of employee scheduling problems have been described in the past. Already in [6] an informal description of the General Employee Scheduling (GES) Problem was provided, giving rise to the identification of several common notions in all problems of this kind.

Several reviews of different problem versions are available. In the review on staff scheduling and rostering in [2] several modules in the rostering process are identified.

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 [7].

In the more recent review [1] 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 paper.

The recent review [8] 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 [9]. 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 [10], 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 [3]. A model of the problem is presented by [11]. 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, [12] presents methods that are successfully applied to these instances.

Shift design is described in [13]. 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 [14].

One of the problems using task demands is the Personnel Task Scheduling Problem (PTSP) in [15] and its optimization variant, the Shift Minimisation Personnel Task Scheduling Problem (SMPTSP) in [16] and [17]. 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 [18]. However, better results can be achieved when scheduling shifts and tasks at the same time as stated by [2].

A combination of shift and task assignments called the Task Scheduling and Personnel Rostering (TSPR) problem is described in [4] 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 [19].

A challenging problem is described in [5] and [20]. 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 [5] a constraint-based approach is used to solve the problem, while in [20] 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 [21] in the context of the fast food industry.

Further there are papers providing general modelling and complexity analysis like [22] 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., [23] 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.

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, a new specification format was developed [24] that supports a wide range of problems using different demand specifications and different types of definitions, restrictions or preference specifications. This section 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 XSD[2] file.

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.

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.

[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`

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.

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.

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 including start and end of the scheduling period, the definition of weekends, the time granularity by *timeSlotLength* 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. It contains both a definition of different break types with the constraints regarding their shape and placement and a definition of break configurations determining different break requirements for different shifts.
- `Employees` defines the available employees which can either be specific named employees or variable employees defining a homogeneous pool.

Further their possible skills are specified. Contracts can contain a large number of constraints regarding contractual limitations that restrict the assignments of shifts to employees. Different types of employee preferences can be given as well.

- `Demands` defines the demands that need to be fulfilled. Shift demands give the required number of employees working a shift type for each type of shift and each day and are used in problems like nurse rostering. Time demands specify the number of required employees for each period of time, often requiring shift design and might also incorporate breaks. Task demands specify tasks that need to be scheduled within a certain time window and shifts have to be build around these tasks.

As the formulation is designed to support the majority of constraints used in different employee scheduling problems and therefore includes a vast number of constraints, they can not all be described in detail. More information on individual constraints is given in the technical report [24], the master thesis [25] and the project homepage.[5]

4. Solver Framework

As the domain of employee scheduling can include a large range of constraints and combine various different aspects of scheduling, it would be highly beneficial to have a general framework that can be applied to a wide range of different employee scheduling problems.

Therefore, the main 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 section describes the main components and structure of the newly developed framework for the implementation of heuristic solvers. A download is available.[6]

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.

[5] `http://www.dbai.tuwien.ac.at/proj/arte/ges_format`
[6] `http://www.dbai.tuwien.ac.at/proj/arte/ges_solver`

- 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.
- 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 section, however, will mainly concentrate on the concepts underlying this implementation and only point to implementation details when relevant.

4.1. 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.

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.

4.2. 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.

4.3. 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.

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. 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.

4.3.1. Constraint Hierarchy

There are several different types of constraints that are 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 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.

4.3.2. 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.

4.3.3. 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 (1).

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

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 1.

Algorithm 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.

4.4. 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.

4.4.1. 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.

4.4.2. 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.

4.4.3. 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 paper only problems with non-preemptive tasks that are fixed in time are considered in the evaluation. 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.

4.4.4. 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.

4.4.5. 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 paper 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.

4.4.6. 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.

4.5. 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.

4.5.1. 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.

4.5.2. 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 section.

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.

4.5.3. 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 our instance generator.

The basic algorithm is described as algorithm 2.

Algorithm 2: 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 section.

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 (2).

$$change \leq 0 \quad \text{or} \quad \texttt{getRandom}(0, 1) < e^{-\frac{change}{t}} \tag{2}$$

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.

5. 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.

5.1. General Aspects of Parameter Tuning

In the approach used in this paper 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.

$$p \cdot |E| \cdot outer \tag{3}$$

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.

5.2. 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.

5.2.1. Problem Selection

For the evaluation the Nottingham instances provided by [3] 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.

5.2.2. Parameter Tuning

This evaluation uses all shift moves described in section 4.4.2 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

$$\texttt{max}\left\{\frac{p \cdot |E| \cdot outer}{100}, 100000\right\} \tag{4}$$

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.

5.2.3. Results

Table 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 [3].

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 1: Results on the Nottingham instances.

5.3. Generated Instances

In this section a test set of instances created by the new instance generator described in [25] 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 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.

5.3.1. 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.

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 2: Results on the generated instances.

5.3.2. Results

For each configuration 5 instances were created, leading to 50 instances in total. Table 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.

5.4. Integrated Task Scheduling and Personnel Rostering Problem

This section evaluates the framework on the TSPR as defined in [4]. 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.

5.4.1. 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.

5.4.2. 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.

5.4.3. Results

For the evaluation a set of 360 instances is available. Table 3 shows the results of the algorithm in comparison with the results presented in [4]. 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.

Days	$\|E\|$	Tightness	Skilling	Our results			Their results	
				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 3: Results on the TSPR instances.

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.

5.5. Shift Design Personnel Task Scheduling Problem

The SDPTSP-E is defined in [5]. This problem is based on a company performing drug evaluation and pharmacology research, therefore following the need for strict testing 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 (5), where w_e is the clinical workload assigned to employee e and c_e is the targeted clinical workload for employee e.

$$violation_e = \max_{e \in E}(w_e - c_e) - \min_{e \in E}(w_e - c_e) \tag{5}$$

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

5.5.1. 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[7] indicates a rolling horizon for the minimum weekly rest time as currently implemented in the solver framework, while constraint (6) in [5] 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

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

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.

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

		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 4: Results on the SDPTSP-E instances.

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.

5.5.2. 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 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 (3), however, divided by 10 in order to stay close to the computation time of 5 minutes per instance as in the compared work.

5.5.3. Results

Table 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.

6. Conclusion

This paper proposed a new framework 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.

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 good results.

This offers a range of possibilities for future research in this area. This includes translation of further problems and applying the solver framework. 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.

References

[1] J. Van den Bergh, J. Belin, P. De Bruecker, E. Demeulemeester, L. De Boeck, Personnel scheduling: A literature review, European Journal of Operational Research 226 (3) (2013) 367–385. doi:10.1016/j.ejor.2012.11.029.

[2] A. Ernst, H. Jiang, M. Krishnamoorthy, D. Sier, Staff scheduling and rostering: A review of applications, methods and models, European Journal of Operational Research 153 (1) (2004) 3–27. doi:10.1016/S0377-2217(03)00095-X.

[3] T. Curtois, Employee shift scheduling benchmark data sets, http://www.schedulingbenchmarks.org/, accessed: 2018-01-22 (2017).

[4] P. Smet, A. T. Ernst, G. Vanden Berghe, Heuristic decomposition approaches for an integrated task scheduling and personnel rostering problem, Computers & Operations Research 76 (2016) 60–72. doi:10.1016/j.cor.2016.05.016.

[5] T. Lapgue, O. Bellenguez-Morineau, D. Prot, A constraint-based approach for the shift design personnel task scheduling problem with equity, Computers & Operations Research 40 (10) (2013) 2450–2465. doi:10.1016/j.cor.2013.04.005.

[6] F. Glover, C. McMillan, The general employee scheduling problem. An integration of MS and AI, Computers & Operations Research 13 (5) (1986) 563–573. doi:10.1016/0305-0548(86)90050-X.

[7] H. K. Alfares, Survey, Categorization, and Comparison of Recent Tour Scheduling Literature, Annals of Operations Research 127 (1-4) (2004) 145–175. doi:10.1023/B:ANOR.0000019088.98647.e2.

[8] P. De Bruecker, J. Van den Bergh, J. Belin, E. Demeulemeester, Workforce planning incorporating skills: State of the art, European Journal of Operational Research 243 (1) (2015) 1–16. doi:10.1016/j.ejor.2014.10.038.

[9] E. K. Burke, P. De Causmaecker, G. V. Berghe, H. Van Landeghem, The State of the Art of Nurse Rostering, Journal of Scheduling 7 (6) (2004) 441–499. doi:10.1023/B:JOSH.0000046076.75950.0b.

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

[11] T. Curtois, R. Qu, Computational results on new staff scheduling benchmark instances, Tech. rep., ASAP Research Group, School of Computer Science, University of Nottingham, NG8 1BB, Nottingham, UK (Oct. 2014).

[12] E. K. Burke, T. Curtois, New approaches to nurse rostering benchmark instances, European Journal of Operational Research 237 (1) (2014) 71–81. doi:10.1016/j.ejor.2014.01.039.

[13] N. Musliu, A. Schaerf, W. Slany, Local search for shift design, European Journal of Operational Research 153 (1) (2004) 51–64. doi:10.1016/S0377-2217(03)00098-5.

[14] A. Beer, J. Gartner, N. Musliu, W. Schafhauser, W. Slany, An AI-Based Break-Scheduling System for Supervisory Personnel, IEEE Intelligent Systems 25 (2) (2010) 60–73. doi:10.1109/MIS.2010.40.

[15] M. Krishnamoorthy, A. T. Ernst, The Personnel Task Scheduling Problem, in: P. M. Pardalos, D. Hearn, X. Yang, K. L. Teo, L. Caccetta (Eds.), Optimization Methods and Applications, Vol. 52, Springer US, Boston, MA, 2001, pp. 343–368, dOI: 10.1007/978-1-4757-3333-4_20.

[16] P. Smet, T. Wauters, M. Mihaylov, G. Vanden Berghe, The shift minimisation personnel task scheduling problem: A new hybrid approach and computational insights, Omega 46 (2014) 64–73. doi:10.1016/j.omega.2014.02.003.

[17] M. Krishnamoorthy, A. Ernst, D. Baatar, Algorithms for large scale Shift Minimisation Personnel Task Scheduling Problems, European Journal of Operational Research 219 (1) (2012) 34–48. doi:10.1016/j.ejor.2011.11.034.

[18] A. W. Kolen, J. K. Lenstra, C. H. Papadimitriou, F. C. Spieksma, Interval scheduling: A survey, Naval Research Logistics 54 (5) (2007) 530–543. doi:10.1002/nav.20231.

[19] A. Meisels, A. Schaerf, Modelling and Solving Employee Timetabling Problems, Annals of Mathematics and Artificial Intelligence 39 (1) (2003) 41–59. doi:10.1023/A:1024460714760.

[20] D. Prot, T. Lapgue, 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) (2015) 7286–7298. doi:10.1080/00207543.2015.1037023.

[21] J. S. Loucks, F. R. Jacobs, Tour Scheduling and Task Assignment of a Heterogeneous Work Force: A Heuristic Approach, Decision Sciences 22 (4) (1991) 719–738. doi:10.1111/j.1540-5915.1991.tb00361.x.

[22] P. Brucker, R. Qu, E. Burke, Personnel scheduling: Models and complexity, European Journal of Operational Research 210 (3) (2011) 467–473. doi:10.1016/j.ejor.2010.11.017.

[23] F. Glover, G. A. Kochenberger, Handbook of Metaheuristics, Springer US, Boston, MA, 2003, oCLC: 903188846.

[24] L. Kletzander, F. Mischek, N. Musliu, G. Post, F. Winter, A General Modeling Format for Employee Scheduling, Tech. rep., Database and Artificial Intelligence Group, Institut für Informationssysteme, TU Wien, http://www.dbai.tuwien.ac.at/proj/arte/ (Mar. 2017).

[25] L. Kletzander, A Heuristic Solver Framework for the General Employee Scheduling Problem, Master's thesis, Database and Artificial Intelligence Group, Institute of Logic and Computation, TU Wien, http://www.dbai.tuwien.ac.at/proj/arte/ (2018).