Universität Kassel

Department of Electrical Engineering and Computer Science

Bachelor Thesis

A Local Search Approach for Scheduling Volunteers in Missions of the German Red Cross Society

presented to
Research Group Programming Languages/Methodologies

Tobias Klipp
35010081
May 20, 2021

Examiners:
Prof. Dr. Claudia Fohry
Prof. Dr. Gerd Stumme

Contents

Statutory Declaration

. . . .
Tobias Klipp
Hessisch Lichtenau, May 20, 2021

List of Figures

List of Tables

Code Snippets

1. Introduction

Non-governmental relief organizations, like the German Red Cross Society, represent a crucial part of the civil protection of Germany[1]. The German Red Cross Society is among the largest relief organizations in Germany and operates multiple civil protection units, which are mainly staffed by volunteers [2].

Voluntary management is a complex task[3]. This also applies to the management of voluntary disaster relief units within emergency scenarios. Emergencies can seldomly be planned in advance and offer dynamic conditions, regarding the damage situation and availability of material and volunteers. Due to possible shortages, an efficient and suitable rostering is critical to both, the welfare of the voluntary workforce and the outcoming of the disaster scenario.

Despite this fact, few researchers have addressed the problem of volunteer scheduling, and no research for the rostering of voluntaries in civil protection missions is known to the author. Nonetheless, the rostering of employees is a wide and active field of research and many different solution approaches to rostering problems for professional staff were presented in the past. Notably, the scheduling of medical personal is predominantly examined within the context of the 'Nurse Rostering Problem'[4].

However, contrary to regular staff members, volunteers of the German Red Cross are not bound by contracts. Instead, each volunteer enrolls independently for missions and may not be available at specific times at all. Volunteers are similarly qualified, but only a few possess advanced skills, which are critical to specific tasks. Volunteer shortage is not uncommon and jobs must be prioritized. For the reasons mentioned, rostering introduces great challenges to the planners in charge and is often manually applied due to a lack of automated solutions for this domain.

To address these shortcomings, this thesis presents a solution approach to automatically create schedulings for volunteers in disaster relief and civil protection missions of the German Red Cross Society.

The solution approach consists of two steps. Initially, a construction algorithm is used to create a valid roster. Then, the result is improved by a local search algorithm, which tries to optimize the roster in regard to a configurable rating function. While the construction algorithm creates rosters deterministically, the local search applies random changes to the roster and traverses the available solution space by the application of multiple neighborhood functions. The local search is executed multiple subsequent times and discovered improvements of one iteration are being passed on to subsequent iterations. Within the local search, worse results are accepted but an increasing number of random changes are applied depending on the number of recursive calls without improvement. By this, the available search space gets increased and exploration is induced if the local search is unable to obtain improvements within the neighborhood of the currently best result.

The local search terminates after a specific amount of failures, but the algorithm itself is executed until a specified number of local search iterations is processed.

Volunteers and Jobs are represented in terms of time slots. Time slots are defined by a start and end time and can be split into multiple shorter periods. Rosters are created by assignments of available volunteer and vacant job time slots. Each assignment represents the scheduling of a volunteer to a job within a defined time duration. Assignments of volunteers to jobs must not cover an entire period, which is offered by either the job or the volunteer. Instead, assignments can allocate parts of time durations and new time slots are created for the remaining time. This way, volunteers can be dynamically scheduled between one or more job, while a job can be staffed by multiple volunteers.

However, not every volunteer can satisfy the requirements of a job. Instead, a job demands a specific skillset and only volunteers who satisfy these requirements can be assigned. Qualifications and requirements are represented by collections of medical, leadership, and technical skills. Medical and leadership skills are hierarchical - higher skill levels satisfy the needs of lower skill levels - and represent a specific skill level within the domain. On the contrary, technical skills are not hierarchical but instead are represented by sets of arbitrary size.

Due to the possible shortage of volunteers, jobs are classified as either mandatory, important, or optional. Mandatory jobs are mission-critical and represent key tasks, like the head of operations or leading physician. Important jobs represent specific tasks, which contribute to the success of a mission, while optional jobs are beneficial, but their vacancies introduce no shortcomings.

Rosters and assignments are subject to multiple hard constraints, which are mandatory and must always be fulfilled. Hard constraints are applied before any assignment is created. Therefore, only valid rosters are created by the algorithm. The quality of rosters is rated by the satisfaction of soft constraints. Each soft constraint is defined by a function and represents a specific optimization target. All soft constraints are contained in the rating function and can be prioritized by weights. Due to this, the results of the algorithm can be affected by different combinations of the soft constraint weights to meet specific needs.

The introduced solution approach is implemented within the functional programming language Haskell[5]. The dependencies of the source code were managed with the Haskell Tool Stack[6]. Volunteer and job data are read from an SQLite database, which is accessed by the persistent library of the Yesod framework[7]. The algorithm can be configured by additional parameters, which are parsed from command-line arguments.

The implementation of the solution approach was tested within different experiments based on the data of a real-world mission. It was observed, that the algorithm is able to generate improvements to rosters that are generated by the construction algorithm. Furthermore a correlation between the number of recursive local search calls and the extent of improvements could be examined.

This thesis is organized as follows. Section 2 presents an overview of existing research and describes different concepts which are used within the solution approach. All type definitions, which are used throughout the paper are introduced in section 3. Section 4 defines the hard and soft constraints. The methodology of both, the construction and local search algorithm are explained in section 5. Section 6 defines the rating function, by which rosters are assessed and introduces the neighborhood functions, which are applied by the local search. The implementation of the solution approach is presented within section 7. The experiments are documented in section 8. Section 9 concludes the thesis. A reference to all variables used in this thesis and a user manual for the application source code can be found within the appendix.

2. Related Work

Staff-Scheduling Personal rostering is the subject of extensive and ongoing research. Multiple problems from different application areas, e.g. hospitals and transportation companies, have been studied over time. Ernst et al. provide an overview of staff scheduling problems, their models and solution methods[8].

Hierarchical Skill Levels Some rostering problems introduce hierarchical skill categories, which consist of multiple qualification levels. E.g. a job may require a certain qualification level, which can be satisfied by a staff member owning the same or a higher level. Additionally, multiple skill categories from different domains may be introduced, e.g. technical and medical qualifications. Cordeau et al. examine a multi-skill project scheduling problem with hierarchical skill levels, within the context of a large telecommunication company, where teams of technicians are assigned to tasks[9]. Their solution uses a large neighborhood search algorithm, which chooses between multiple destroy and repair methods.

The Nurse Rostering Problem The rostering of medical staff members is often examined as the "Nurse Rostering Problem". Where a set of nurses with varying qualifications is scheduled to different kinds of shifts, e.g. day or night shifts, over specific time horizons. However, the problem often offers immense complexity because of many and diverse constraints, including work regulations, personal preferences, and task coverage.

Many different instances of the Nurse Rostering Problem were specified in the past. An extensive overview of the existing research is provided by Burke, Causmacker, and Vanden Berghe[4]. That paper provides a collection of literature references and discusses known models and their respective solution approaches.

Classification of Rostering Problems Causmacker and Vanden Berghe introduce a common notation for a staff rostering in general and the nurse rostering problem in particular[10]. The proposed notation is meant to serve a classification framework, which tries to help researchers by specifying their scheduling problems and therefore making them comparable to others.

The well-known $\alpha \mid \beta \mid \gamma$ notation of scheduling problems is used. Category α contains personal and group constraints, regarding the availability and skills of single nurses and the fairness between the schedules of different staff members. Workload constraints (β) define the staff or shift requirements over fixed periods and how many shifts exist and if they are allowed to overlap. Finally, the optimization objectives are defined by category γ. A problem may be optimized regarding one or multiple competing objectives. Common objectives include the minimization of the number of required staff members, the task coverage within specific periods and the robustness of solutions, this is how widely rosters may differ from each other.

Local Search Some rostering problems, including the nurse rostering problem, introduce very large solution spaces, where decent results may not be achievable within sane time durations with systematic approaches. Instead, heuristic approaches like Local Search, are used to generate as good as possible results.

Local Search approaches often consist of two phases. First, an initial solution is created by a construction algorithm. Construction algorithms may try to generate good solutions, or just generate valid results in a random manner.

This initial solution serves as an entry point to a specific section of the solution space. The local search alters the initial state by small modifications and the results are valued by a rating function, which may cover multiple objectives. The set of results which is obtained by these modifications is called a neighborhood. If a better solution is found within the neighborhood, it is chosen as a new seed and another iteration of the local search is started.

The number of changes that are applied to a state determine the result of the algorithm. Small adjustments may lead to an exploitation of the current section of the solution space, which may lead to a local optimum. On the contrary, large adjustments lead to more distant sections of the search space but may not be able to exploit the direct neighborhood.

In practice, local search algorithms switch between exploration and exploitation of the solution space. An overview of different local search strategies is provided by Luke [11].

Local Search with Variable Neighborhoods Bilgin et al. define a generic nurse rostering model and provide a solution based on local search with variable neighborhoods[12]. The model offers much flexibility, e.g. shifts can be defined with variable start and end times, skill categories are not predefined and nurses can own one or more skills. Provided soft constraints include the assignment of the nurses to tasks regarding their primary skill, minimum rest times between two subsequent assignments, and preferences of the staff members. The proposed algorithm consists of two phases and can work with partial complete schedules, previously defined by the user. The preprocessing phase consists of two steps. First, the construction algorithm tries to create an initial feasible solution that fulfills the covering constraints. Second, additional assignments between nurses and shifts are added to the schedule, to meet the weekly job time of each nurse. The second optimization phase tries to enhance the provided solution through local search in combination with a tabu-list. While the termination criterion is not met, the algorithm searches for local optima within one specific neighborhood and changes the strategy as soon as no improvement can be found in the present environment.

Generalized Local Search A generalized local search for employee timetabling problems is introduced by Scharf and Meisels[13]. Within the model, employees are assigned to shifts, which may contain one or more tasks, and are defined with specific start and end times. The search space is constructed by the application of three transition operators. Proposed constraints regard the number of required employees with specific skills within a given shift, the availability of staff members at a specific time, conflicts between shifts, and the workload, this is the number of tasks or shifts to whom an employee may be assigned within a given period. Within the cost function, all constraints are multiplicated with adjustable weights, which are limited by a maximum and minimum value and are either relaxed or tightened, regarding the number of sequential occurrences of constraint violations. In addition to the previously mentioned constraints, the cost function contains a look-ahead factor, which considers the number of conflicting assignments currently preventing the transition to better states. Hill Climbing is used as a local search method but the algorithm stops after a given number of iterations without any improvement to prevent a stuck at local optima.

Progress Control for Iterated Local Search Local search can be used to retrieve and enhance feasible solutions for hard optimization problems. Nonetheless, information about how good a solution is compared to the global optimum is generally not available. The termination time of the optimization process - when to stop the search - is therefore not trivial to decide. Burke et al. try to address the problem by introducing a general progress control for iterated local search, especially for the nurse rostering problem[14]. By creating a set of varying modifications to a common state, the approach tries to prevent the search of small neighborhoods to find globally good solutions. Such seed states are used as input for different executions of a common local search algorithm. The proposed progress control in return tries to estimate the remaining time, one run of the local search algorithm has to spend, until a better local optimum generated from another seed state may be found. Throughout the execution, each optimization step is measured by a cost function, whose results, in turn, are used to calculate the progress estimation. By this, the search process starting from bad seed states may be stopped early, while good starting points are used to find good solutions.

Volunteer Rostering In contrast to regular staff scheduling, few research regarding the rostering of volunteers exists. Falasca and Zobel provide a model for volunteer scheduling within humanitarian organizations[3]. They introduce insights into how humanitarian operations differ from regular work within corporations and which management practices should be considered, e.g. groups of volunteers wish to work together and the preferences of volunteers should be considered to great extend in order to retain the available workforce as long as possible. Furthermore, they describe multiple constraints for task and time-block assignments, which represent the needs of voluntary workers, e.g. unwanted assignments to tasks or shifts, but also the constraints of the overall organization, e.g. the minimum required workforce. Solutions to the described problem are delivered by a fuzzy logic approach to support management decisions, regarding the tradeoff between a extensive job coverage and the prevention of undesired assignments.

Own Previous Work The introduced problem was previously examined within another study project ("Projektarbeit") by the author of this work. This preliminary work examined different construction algorithm approaches to generate feasible rosters, which were meant to serve as input for further optimization methods. As a result, a construction algorithm relying on the classification of jobs by hard-coded rules, based on expert knowledge, was introduced. However, the work included only the programming of the construction algorithm, but was not recorded in a document.

3. Types

This section introduces the type definitions by whom the introduced problem is modeled. Time definitions represent UTC day-times, which are encoded as natural numbers.

3.1. Volunteers

The set of volunteers is represented by V. A single volunteer $v \in V$ is available within a given time duration, defined by a start time $s_v \in \mathbb{N}$ and an end time $e_v \in \mathbb{N}$.

3.2. Jobs

The set of jobs is represented by J. A job $j \in J$ begins at a given time $s_j \in \mathbb{N}$ and ends at a later point in time $e_j \in \mathbb{N}$.

Three job categories exist. First, *mandatory* jobs represent mission critical tasks which must be staffed in any case, e.g. head of operations. Second, *important* jobs embody key tasks, which are significant to a missions success, e.g. team leaders, medics, technicians or drivers. Lastly, *optional* jobs represent redundant or non important tasks and their absence does not compromise the missions success.

Each job is placed into one category by the variable $p_j \in \{mandatory, important, optional\}$.

3.3. Qualifications and Requirements

Volunteers own *qualifications* and jobs demand *requirements*. Qualifications and requirements contain skills from medical, leadership and technical categories.

Medical skills describe the quality of medical training, e.g. "Sanitäter" or "Arzt". Leadership skills express the level of management skills, e.g. "Gruppenführer" or "Zugführer". Medical and leadership skills are hierarchical. That is, higher skill levels satisfy the needs of lower skill levels. Technical skills include special trainings and certifications, e.g. driver-licences and crafting skills.

A qualification q_v or requirement r_j is defined by a tuple $(m_x, l_x, \mathcal{T}_x)$, consisting of a medical skill $m_x \in \mathbb{N}$, a leadership skill $l_x \in \mathbb{N}$ and a set of technical skills $\mathcal{T}_x \in TECH$, where $x \in V \cup J$ and $TECH$ is the set of all technical skills, which is dependent on a specific instance of the problem

3.4. Assignments

A volunteer $v \in V$ is assigned to a job $j \in J$ from a start time s_{vj} until an end time e_{vj}. Assignments are represented by a tuple $a_{vj} = (v, j, s_{vj}, e_{vj})$. Note, that $s_{vj} < e_{vj}$ is always true, since an assignment cannot start after it has ended.

The set of all assignments is defined by $\mathcal{A} \subset V \times J \times \mathbb{N} \times \mathbb{N}$. Therefore, all assignments of a specific volunteer v can be derived by equation 1.

$$A_v = \{a \in \mathcal{A} | a = (v, x, s, e), x \in J \text{ and } s, e \in \mathbb{N}\} \tag{1}$$

The set of all assignments for a particular job can be generated by equation 2.

$$A_j = \{a \in \mathcal{A} | a = (x, j, s, e), x \in V \text{ and } s, e \in \mathbb{N}\} \tag{2}$$

Note, that equation 3 holds, because every assignment consists of both, a job and a volunteer.

$$\mathcal{A} = \bigcup_{v \in V} A_v = \bigcup_{j \in J} A_j \tag{3}$$

3.5. Timeslots

A timeslot $t \in V \cup J \times \mathbb{N} \times \mathbb{N}$ defined by equation 4

$$t = (x, s_t, e_t) \tag{4}$$

represents the availability of a volunteer or vacancy of a job $x \in J \cup V$ during a period defined by a start $s_t \in \mathbb{N}$ and an end $e_t \in \mathbb{N}$.

All available timeslots of a volunteer $v \in V$ are contained in the set T_v(equation 6), while all available timeslots of a job $j \in J$ are included in T_j(equation 5).

$$T_J = \bigcup_{j \in J} T_j \tag{5}$$

$$T_V = \bigcup_{v \in V} T_v \tag{6}$$

3.6. Roster

A roster R is defined by the tuple

$$R = (V, J, \mathcal{A}, T_V, T_J) \tag{7}$$

which includes the set of volunteers V, the set of jobs J, the set of assignments $\mathcal{A}$ and the available time slots of all volunteers T_V and all jobs T_J.

4. Constraints

The following section introduces multiple hard and soft constraints. Hard constraints are applied to ensure the validity of rosters, while soft constraints are used to rate the quality of rosters in order to make them comparable.

4.1. Hard Constraints

Hard constraints are applied to single matches between volunteer and job time-slots as well as assignments and entire rosters.

4.1.1. Match Constraints

A volunteer timeslot $t_v = (v, s_v, e_v) \in T_V$ and a job timeslot $t_j = (j, s_j, e_j) \in T_J$ do match, if they satisfy all match constraints together. If all constraints are fullfilled, t_v is called a candidate for t_j. Only candidates can be assigned to job time slots.

Requirement Satisfaction The volunteer v must fullfill all requirements of the job j.

$$m_v >= m_j \tag{8}$$
$$l_v >= l_j \tag{9}$$
$$\mathcal{T}_j \setminus \mathcal{T}_v = \emptyset \tag{10}$$

That is, the medical and leadership skills of v must be at least as great as the requirements of j (equation 8 and 9) and the technical skills demanded by j must be included in $\mathcal{T}_v$ (equation 10). Note, that volunteers may posess more technical skills then jobs require.

Time Interval Overlap The time slots t_v and t_j must share a common time interval.

$$(s_v \leq s_j \wedge s_j < e_v) \vee (s_j \leq s_v \wedge s_v < e_j) \tag{11}$$

Minimum Overlap Time The time slots t_v and t_j must overlap for a minimal amount of time (equations 12 and 13)

$$overlapDuration(x, y) = min\{e_x, e_y\} - max\{s_x, s_y\} \tag{12}$$
$$overlapDuration(t_j, t_v) >= minDuration \tag{13}$$
$$x \in T_J, y \in T_V$$

4.1.2. Assignment Constraints

A volunteer $v \in V$ and job $j \in J$ must not be assigned more then once within the same time interval (equations 14 and 15).

$$\forall (v, j_a, s_a, e_a), (v, j_b, s_b, e_b) \in A_v : j_a \neq j_b \implies overlapDuration((v, s_a, e_a), (v, s_b, e_b)) = 0 \tag{14}$$

$$\forall (v_c, j, s_c, e_c), (v_d, j, s_d, e_d) \in A_v : v_c \neq v_d \implies overlapDuration((v, s_c, e_c), (v, s_d, e_d)) = 0 \tag{15}$$

$$j_a, j_b \in J, v_c, v_d \in V$$

Note, that *overlapDuration* was defined in equation 12.

4.1.3. Roster Constraints

All mandatory jobs must be fully assigned within a roster $R = (V, J, \mathcal{A}, T_V, T_J)$ (equation 16).

$$\forall (j, s, e) \in T_J \implies p_j \neq mandatory \tag{16}$$

4.2. Soft Constraints

Soft constraints are defined by functions which measure the satisfaction of specific optimization targets. All result values are normalized to values within $[0, 1]$ to provide comparability.

Compactness A volunteer or a job may be distributed over multiple assignments. Each time two subsequent assignees alternate, time or material costs for briefings or travelling can be required. Therefore, few shift changes may be desired to reduce the amount of expenses to the voluntary workforce.

However, overhead costs may vary between different scenarios and can not be directly measured or predicted. It is presumed that a low distribution (*compactness*) of volunteers over multiple job prevents such costs from emerging. Eqution 17 measures the ratio between the number of volunteers with at least one assignment $|V^+|$ and the overall number of assignments $|\mathcal{A}|$.

$$volunteer\_compactness(\mathcal{A}, V) = \frac{|V^+|}{|\mathcal{A}|} \tag{17}$$

$$V^+ = \{v \in V | A_v \neq \emptyset\} \tag{18}$$

The volunteer compactness is high, if few volunteers are distributed among multiple jobs. Note however, that a high *volunteer_compactness* does not guarantee few shift changes, because short assignments of multiple assignees with short time intervals are not sanctioned.

Therefore, the compactness of assigned jobs - how many volunteers are assigned to one job - is measured by equation 19 as well. The *job_compactness* measures the ratio between the number of jobs with at least one assignment $|J^+|$ and the overall number of assignments $|\mathcal{A}|$.

$$job\_compactness(\mathcal{A}, J) = \frac{|J^+|}{|\mathcal{A}|} \tag{19}$$

$$J^+ = \{j \in J | A_j \neq \emptyset\} \tag{20}$$

The *job compactness* is high, if few volunteers are assigned to the same job. Note, that $|\mathcal{A}|$ is allways greater or equal then $|V^+|$ and $|J^+|$, because every assignment includes a job and a volunteer, that is $|J^+| \leq |\mathcal{A}| \leq |J|$ and $|V^+| \leq |\mathcal{A}| \leq |V|$.

Completeness A volunteer $v \in V$ or a job $j \in J$ are partially assigned, if they are included in at least one assignment $A_v, A_j \neq \emptyset$, but offer additional timeslots $T_v, T_j \neq \emptyset$, which are available for further assignments.

Partially assigned jobs represent gaps within the roster, where no volunteer takes care of the tasks offered by a job This may be unintended, if a continuously service is mandatory, e.g. wounded patients must be treated without interruptions.

The *job_completeness* function, defined in equation 21 measures the amount of assigned jobs that are completely staffed.

$$job\_completeness(J) = \frac{|J^+ \setminus J^-|}{|J^+|} \tag{21}$$

$$J^- = \{j \in J | T_j \neq \emptyset\} \tag{22}$$

The result value is low, if few complete assigned jobs exist, while the contrary is true, if jobs are generally completely allocated.

In the same way, the *volunteer_completeness* function, defined in equation 23, measures the amount of volunteers, which are assigned for their entire available time. This may be intended, to reduce the number of deployed volunteers in case of limited transport or supply capacities, or to keep reserves of rested volunteers for shift changeovers.

$$volunteer\_completeness(V) = \frac{|V^+ \setminus V^-|}{|V^+|} \tag{23}$$

$$V^- = \{v \in V | T_v \neq \emptyset\} \tag{24}$$

The result of the *volunteer_completeness* is high, if many volunteers do work for their entire available time instead of short periods.

5. Algorithmic Methodology

The following section describes the algorithms of the local search approach, which generates and optimizes valid rosters according to the constraints from the section 4. The algorithms are additionally presented by descriptive Haskell code.

5.1. Construction Algorithm

The construction algorithm extends a given roster, such that new assignments are generated as long as matches between job and volunteer time slots exist. A descriptive version of the algorithm is shown in code snippet 5.1.

Rosters contain lists of volunteer and job time slots (lines 6,7). While *volunteers* represents the list of all available volunteers with their respective start and end times, *jobs* contains all job vacancies with their beginnings and ends.

Before the first assignment, volunteers and jobs are not split over multiple time slots. Instead, each time slot in *volunteers*, or *jobs* represents initially a complete-time interval. Note, that the initial roster may but does not have to be empty and previous assignments can exist. Therefore, existing rosters can be extended if further jobs or volunteers become available.

```
1  type SearchList = [(JobSlot,[VolunteerSlot])]
2
3  constructRoster :: Roster -> Roster
4  constructRoster roster =
5    let
6    volunteers = volunteerSlots roster
7    jobs = jobSlots roster
8    searchList = createSearchList jobs volunteers
9    assignmentCandidates = findBestMatches searchList
10   bestMatch = maximum assignmentCandidates
11   newRoster = addNewAssignmentToRoster bestMatch roster
12   in
13   if null assignmentCandidates
14   then roster
15   else constructRoster newRoster
```

Code Snippet 5.1: Construction Algorithm

First, the input is converted (line 8) into a search list (line 1). Each entry is a tuple, which consists of a job time slot and a list of its candidates - volunteer time slots, which satisfy the hard constraints together with the job.

After that, the search list is traversed and for each job, the best assignee is evaluated according to an assignment rating function (line 9). The ratings of all generated pairs of jobs and volunteers are compared (line 10) and an assignment from the best match is added to the roster (line 11) .

If the job and candidate time slots do not overlap completely, additional time slots are created. The new time slots represent the remaining time that is not allocated by the new assignment. The newly created time slots are added to the lists of available volunteer or job slots and can be used in further assignments.

When a new assignment is created, the assigned time slots of the job and volunteer are deleted from the list of available volunteer and job time slots. Then a new search list is created from the remaining job and volunteer slots and the algorithm executes a new recursive call. This process is repeated (line 15) until no further assignment candidates - matching job and volunteer pairs - can be found (line 13).

A different *greedy* version of the construction algorithm was previously created within another student project "Projektarbeit" of the author, which was not recorded in a document. It traversed a search list sorted by the length of the candidate lists. The jobs with the least number of candidates were first assigned to their best match. Then, the candidate time slot was deleted from the list of available volunteer time slots, before the best candidate for the next job was evaluated. This way, hard to fill jobs were handled prioritized, while jobs with many candidates were deferred. However, the construction algorithm did not create assignments in regard to a rating function. Because of that, the version introduced in this section was developed for the solution approach presented in this thesis.

5.2. Local Search Approach

The construction algorithm creates a valid roster according to the rating of single assignments. However, it does not consider the global quality of a roster.

The quality of a roster is measured by a rating function described in section 6.1.5, which evaluates the satisfaction of the soft constraints defined in 4.2. The local search approach tries to optimize this rating function according to user defined weights, which model specific preferences, e.g. few partially assigned jobs.

The approach consists of two nested loops. First, the inner loop, shown in code snippet 5.2.1, represented by the local search itself. Second, an outer loop which, shown in code snipped 5.3, executes multiple subsequent local search iterations and returns the best generated roster.

The algorithm can be configured by the *maxIterations* and *maxStagnations* parameters. The *maxIterations* parameter specifies how many subsequent iterations of the outer loop are executed.

5.2.1. Local Search

The local-search algorithm can be configured with the *maxStagnations* parameter. The *maxStagnations* parameter defines, how many worse results are tolerated before the local search stops. and how many random deletions are applied at the beginning of each recursive call of the local search. At the beginning of each execution of the inner loop (*repitition*), a specific amount (*stagnations*) of random assignments gets deleted from the roster(line 4).

```
1  localSearch :: Int -> Int -> Roster -> Roster -> Roster
2  localSearch maxStagnations stagnations bestRoster seed =
3    let
4    randomizedSeed = deleteRandomAssignments stagnations seed
5    neighborhood = applyNeighboorhoodFunctions randomizedSeed
6    bestNeighbor = fillRoster $ getBestNeighbor neighborhood
7    randomNeighbor = getRandomElement neighborhood
8    in
9    if rating bestNeighbor > rating seed
10   then
11    localSearch
12      maxStagnations (max 1 (stagnations-1)) bestNeighbor bestNeighbor
13   else
14    if repWithoutImprovement == maxStagnations
15    then bestRoster
16    else
17    localSearch maxStagnations (stagnations+1) bestRoster randomNeighbor
```

Code Snippet 5.2: Local Search Algorithm

All neighborhood functions, which are defined in section 6.2, are evaluated(line 5). The set of rosters that is obtained thereby is called the *neighborhood* of the *seed*.

The highest rated roster (*bestNeighbor*) from the neighborhood (line 6) is compared to the *seed* (line 9). If no improvement is found, a counter is incremented and a random roster from the neighborhood (*randomNeighbor*) is chosen (line 7) as *seed* for the next repetition(line 17). On the contrary, if *bestNeighbor* introduces an improvement, *stagnations* is decremented and the *bestNeighbor* is remembered as currently best roster(line 12). By decrementing *stagnations*, the discovery of improvements is rewarded, such that further exploration within the promising neighborhood is possible.

Note, that within each *repetition* only one assignment is added, while multiple assignments are deleted at the beginning. Consequently, every neighbor offers a worse rating than the *seed*. To make both rosters comparable, *bestNeighbor* is filled by another execution of the construction algorithm, which adds all further possible assignments.

If improvements can be found within *maxStagnations* (line 14), the *bestRoster* is returned and the local search terminates (line 15).

Search space exploration is enforced by random deletions of assignments at the beginning of each *repetition*. The longer no improvements can be found within the neighborhood of the *seed*, the larger the neighborhood becomes. This way, subsequent cycles around the same local optima shall be avoided, while exploration is induced by increasing the available options.

5.2.2. Iterated Local Search

```
1  iterateLocalSearch :: Int -> Int -> Int -> Roster -> Roster
2  iterateLocalSearch maxIterations iterations maxRepititions seed =
3    let
4    newRoster = localSearch maxRepititions 0 seed
5    newSeed = if (rating newRoster) > (rating seed)
6              then newRoster
7              else seed
8    in
9    if iterations == maxIterations
10   then seed
11   else iterateLocalSearch maxIterations (iterations+1) newSeed
```

Code Snippet 5.3: Iterated Local Search Algorithm

The outer loop executes a specific number (*maxIterations*) of subsequent local search runs. Throughout all executions, the currently best-discovered roster is remembered as *seed*. Initially, this is the result of the construction algorithm.

After each local search iteration, *newRoster* (line 4) is compared to the *seed* (line 5). As soon as a better roster is discovered, it is remembered as *newSeed* and used as input for subsequent local search executions (line 11).

Finally, after *maxIterations* (line 9) the *seed* is returned and the algorithm terminates.

It is assumed that better rosters can be obtained by small adjustments to already good rosters. Therefore, the algorithm recursively improves the currently best-discovered roster, while worse alternatives are ignored.

6. Heuristics

The following section describes the heuristics which are used by the construction and local search algorithms. The heuristics are used to rate assignments and rosters and to traverse the solution space.

6.1. Ratings

The quality of an assignment is determined by the assignment-rating function shown in equation 29. The assignment rating function is composed of multiple rating functions which consider the qualifications and requirements provided by a volunteer and a job. Matches of qualifications and requirements are rated high if they offer equal skill sets. Therefore, assignments of overqualified volunteers are sanctioned, while close matches are rewarded. All results of rating functions are normalized to values in the interval $[0, 1]$ to ensure comparability. The assignment-rating function enables the construction and local search algorithms to compare different assignments. However, an assignment is only created by matching volunteer and job time slots. Whether two time slots fit together is determined by equation 25.

6.1.1. Timeslot Matches

A volunteer time-slot $t_v = (v, s_v, e_v) \in T_V$ and a job time-slot $(t_j, s_j, e_j) \in T_J$ match if equation 25 holds.

$$isMatch(t_v, t_j) = \begin{cases} True, & t_v \text{ and } t_j \text{ satisfy all hard constraints} \\ False, & otherwise \end{cases} \tag{25}$$

6.1.2. Rating of Hierarchically Skills

The rating of hierarchical skills is evaluated by equation 26. The rating function determines how good the qualifications of a volunteer $v \in V$ match the requirements of a job $j \in J$.

$$hierarchical\_skill\_rating(a = (v, j, s, e)) = \frac{1}{2} * (\frac{m_j}{m_v} + \frac{l_j}{l_v}) \tag{26}$$

Note, that the skill values of the assigned job are always lower or equal then those possessed by the assigned volunteer, that is $m_j <= m_v$ and $l_j <= l_v$ are true because of equation 25 and division of zero is therefore prohibited.

6.1.3. Rating of Technical Skills

The matching value of collection skills is calculated by the equation 27.

$$technical\_skill\_rating(a_{vj}) = 1 - \frac{|\mathcal{T}_v \setminus \mathcal{T}_j|}{|\mathcal{T}_v|} \tag{27}$$

Technical skill matches are rated high, if the offered skill sets of a volunteer $v \in V$ and job $j \in J$ contain the same elements. Note, that the technical skill sets of assigned jobs contain always the same skills as the set of the assigned volunteer because of equation 25 and division of zero is therefore prohibited.

6.1.4. Assignment Rating

The *assignment_value* function, defined in equation 29 consists of equations 26 and 27. The value of the rating is determined by the priority of the job, such that assignments of high prioritized jobs are rewarded.

The priority of a job is obtained by equation 28.

$$priority(p_j) = \begin{cases} 5, & p_j = important, mandatory \\ 1, & p_j = optional \end{cases} \tag{28}$$

Note, that mandatory and important jobs provide the same prioritization value. This is because mandatory jobs are always assigned due to hard constraint 16 and higher prioritization values offer no further benefit.

$$\begin{aligned} assignment\_value(a = (v, j, s, e) =& priority(p_j) \\ & *hierarchical\_value(a_{vj}) \\ & *collection\_value(a_{vj}) \end{aligned} \tag{29}$$

Rating of Assignments The *quality* function defined in equation 30 measures the sum of assignment ratings for a set of assignments $A \subset V \cup J \times \mathbb{N} \times \mathbb{N}$.

$$quality(A) = \frac{1}{P} * \sum_{a \in A} assignment\_value(a) \tag{30}$$

$$P = \sum_{(v,j,s,e) \in A} p_j \tag{31}$$

The result of the function is normalized to values from the interval $[0, 1]$ by division with the sum of all job prioritization values P, defined in equation 31.

6.1.5. Roster Rating

A roster $R = (V, J, \mathcal{A}, T_V, T_J)$ is rated by equation 32.

$$\begin{aligned} roster\_rating(\mathcal{S}, \Omega) =& 100 * \frac{1}{W} \\ & * (\omega_1 * quality(\mathcal{A}) \\ & + \omega_2 * volunteer\_compactness(\mathcal{A}, V) \\ & + \omega_3 * job\_compactness(\mathcal{A}, J) \\ & + \omega_4 * volunteer\_completeness(V) \\ & + \omega_5 * job\_completeness(\mathcal{A}, J)) \\ W =& \sum_{\omega \in \Omega} \omega \end{aligned} \tag{32}$$

The *roster_rating* function is composed of the soft constraints functions, described in section 4.2 and the assignment quality function from equation 30. Each part of the function can be prioritized by the weights $\omega \in \Omega$ with $0 \leq \omega \leq 10$. A component can be ignored if its respective weight is set to zero. Different combinations of the weights can be used to model specific optimization targets. E.g. if jobs shall be completely assigned for as long as possible time durations, ω_2 and ω_4 are set to high values, while all remaining weights are either set to lower values or zero.

6.2. Neighborhood

A roster $R = (V, J, \mathcal{A}, T_V, T_J)$ is a specific point within the solution space. The local search uses multiple neighborhood functions to apply specific adjustments to R. The set $\mathcal{N}$, which contains the results of all neighborhood functions, is called the neighborhood of R. The elements of $\mathcal{N}$ are used by the local search to traverse the solution space. That is, each time the local search chooses a specific element of $\mathcal{N}$ it may discover a new neighborhood.

6.2.1. Operators

Operators represent basic operations, which are combined by the neighborhood functions to apply specific changes to R.

Delete Timeslot A volunteer timeslot $t_v \in T_V$ can be deleted from R by the operator defined in equation 33.

$$\begin{aligned} delete\_volunteer\_timeslot(t_v, R) &= (V, J, \mathcal{A}, T_V^-, T_J) \\ T_V^- &= T_V \setminus \{t_v\} \end{aligned} \tag{33}$$

A job time slot can be deleted with the operator defined in equation 34.

$$\begin{aligned} delete\_job\_timeslot(t_j, R) &= (V, J, \mathcal{A}, T_V, T_J^-) \\ T_J^- &= T_J \setminus \{t_j\} \end{aligned} \tag{34}$$

Add Assignment An assignment from a volunteer time slot $t_v = (v, s_v, e_v) \in T_V$ and a job time slot $t_j = (j, s_j, e_j) \in T_J$ is added to R by the *add_assignment* operator defined in 39.

However, each time an assignment is created, the time slots of the assigned volunteers and jobs must be deleted and new time slots for the unallocated periods must be created if the assignment time slots do not overlap completely. Consider the case where the volunteer time slot t_v gets assigned to the job time slot t_j. Both time slots start at the point in time, but the end time of t_v is earlier then the end time of t_j, that is $e_v < e_j$. While the available time of t_v is completely utilized, the job period is only covered partially and can be used for further assignments.

Time slots for unallocated periods are created by the operators defined in equations 35 and 36.

$$frontSplit(t_v, t_j, R = (V, J, \mathcal{A}, T_V, T_J)) = \begin{cases} (V, J, \mathcal{A}, T_V \cup (v, s_v, s_j), T_J) & s_v < s_j, \\ (V, J, \mathcal{A}, T_V, T_J \cup (j, s_j, s_v)) & s_j < s_v, \\ R & otherwise \end{cases} \tag{35}$$

The *front_split* operator defined in equation 35 creates time slots if a time slot starts earlier than its assignment partner. The assignment interval begins from the start of the later time slot. Therefore, the period from the start of the earlier time slot until the start of the later time slot is unallocated by the assignment and must be represented by a new time slot.

$$tailSplit(t_v, t_j, R = (V, J, \mathcal{A}, T_V, T_J)) = \begin{cases} (V, J, \mathcal{A}, T_V \cup (v, e_v, e_j), T_J) & e_v < e_j, \\ (V, J, \mathcal{A}, T_V, T_J \cup (j, e_j, e_v)) & e_j < e_v, \\ R & otherwise \end{cases} \tag{36}$$

If a time slot ends later then it‘s assignment partner. The assignment interval ends at the earliest end time offered by both assignment partners. The remaining time interval from the earlier until the later is created by the *tailSplit* operator defined in 36.

The results of both split operators are combined with the *add_unallocated_timeslots* operator, defined in 37.

$$add\_unallocated\_timeslots(t_v, t_j, R) = frontSplit(t_v, t_j, R) \oplus tailSplit(t_v, t_j, R) \tag{37}$$

where the rosters created by *tailSplit* and *frontSplit* are combined with the $\oplus$ operator.

$$\oplus(R_1 = (V, J, \mathcal{A}_1, T_{V_1}, T_{J_1}), R_2 = (V, J, \mathcal{A}_2, T_{V_2}, T_{J_2})) = (V, J, \mathcal{A}_1 \cup \mathcal{A}_2, T_{V_1} \cup T_{V_2}, T_{J_1} \cup T_{J_2}) \tag{38}$$

An assignment between matching time slots is added with the *add_assignment* operator defined in equation 39.

$$\begin{aligned} add\_assignment(t_v = (v, s_v, e_v), t_j = (j, s_j, e_j), R = (V, J, \mathcal{A}, T_V, T_J)) = R^+ \oplus R^- \\ s = max\{s_v, s_j\} \\ e = min\{e_v, e_j\} \\ R^- = delete\_volunteer\_timeslot(t_v, R) \oplus delete\_job\_timeslot(t_j, R) \\ A^+ = \mathcal{A} \cup \{(v, j, s, e)\} \\ R^+ = add\_unallocated\_TimeSlots(t_v, t_j, (V, J, \mathcal{A}^+, T_V, T_J)) \end{aligned} \tag{39}$$

which adds a new assignment $a = (v, j, s, e)$ from job j and volunteer v to the set of all assignments within the time interval defined from s and e. Note, that every additional assignment always results in an increased rating of the new roster state. Each new assignment steps further into one specific area of the solution space and decreases the number of remaining options.

Recombine Time Slots Subsequent time-slots can be recombined. For a time-slot $t = (x, s, e)$, with $x \in J \cup V$ two surroundings exist. The predecessor $pre = (x, s_{pre}, e_{pre})$, whose end time is equal to s. And the successor $succ = (x, s_{succ}, e_{succ})$, whose start time is equal to the e. Note, that $e_{pre} = s \leq e = s_{succ}$ holds. These subsequent slots can be recombined to a new timeslot t_{new}, with $t_{new} = (v, s_{pre}, e_{succ})$.

$$recombine\_timeslots(R = (V, J, \mathcal{A}, T_V, T_J)) \tag{40}$$

recombines all subsequent time slots in the set of all volunteer and job time slots T_V and T_J respectively. The recombination ensures, that no separate subsequent time slots of the same volunteer or job are contained within the set of volunteer or job timeslots. This is necessary to avoid the fragmentation of continuous time intervals, which lead to short assignment periods although longer time periods were available.

Delete Assignments An assignment is deleted from a Roster R with the *delete_assignment* operator defined in 41.

$$\begin{aligned} &delete\_assignment(a = (v, j, s, e), R = (V, J, \mathcal{A}, T_V, T_J)) = R‘ \\ &\qquad A^- = \mathcal{A} \setminus \{a\} \\ &\qquad T_V^+ = T_V \cup (v, s, e) \\ &\qquad T_J^+ = T_J \cup (j, s, e) \\ &\qquad R‘ = (V, J, \mathcal{A}^-, T_V^+, T_J^+) \end{aligned} \tag{41}$$

which removes an assignment $a = (v, j, s, e)$ from the set of all assignments $\mathcal{A}$. Every deletion potentially increases the amount of remaining assignment options. Because of that, assignment deletions can be used to extend the available search space, if a local optimum is reached by the algorithm.

If an assignment is deleted, new volunteer and job time-slots (v, s, e) and (j, s, e), are created and added to T_V and T_J.

6.2.2. Neighborhood Functions

Neighborhood functions combine one or more of the previous defined operators, to apply a specific change to a roster $R = (V, J, \mathcal{A}, T_V, T_J)$.

Exchange Candidates The volunteers $v, w \in V$ of two assignments $a = (v, j, s_{vj}, e_{vj}, b = (w, k, s_{wk}, e_{wk})$ can be exchanged, if the time durations of the assignments overlap and the qualifications of the volunteers satisfy the requirements of the two assigned jobs $j, k \in J$.

$$\begin{aligned} \mathit{exchange\_candidates}(a, b, R = (V, J, \mathcal{A}, T_V, T_J)) &= \mathit{recombine\_timeslots}(R_a^- \oplus R_b^-) \\ R_a &= \mathit{replace\_candidate}(a, t_w = (w, s_{wk}, e_{wk}), R) \\ R_b &= \mathit{replace\_candidate}(b, t_v = (v, s_{vj}, e_{vj}), R) \\ R_a^- &= \mathit{delete\_volunteer\_timeslot}((v, s_{vj}, e_{vj}), R_a) \\ R_b^- &= \mathit{delete\_volunteer\_timeslot}((w, s_{vw}, e_{vw}), R_b) \end{aligned} \tag{42}$$

The exchange is applied by two replacements. The assignment a is deleted and a new assignment during the same time interval is created by the assigned job time slot of a and the volunteer time slot $t_w = (w, s_{wk}, e_{wk})$ from assignment b. A new time slot t_v of the replaced volunteer v is created and added to T_V in R. Additionally, further time slots are created if t_w does not completely match the period of the job time slot of a. The second assignment is applied in the same way. However, as soon as the second replacement is applied, t_v and t_w are redundant to the new assignments in R_a and R_b and must be deleted.

Volunteer Reassignment A volunteer $v \in V$ can be distributed over multiple assignments A_v. The distribution can be reduced if all assignments of v are deleted, recombined, and reassigned to a longer job time slot. The *reassign_volunteer* function defined in 43 reassigns a complete volunteer time interval.

$$
\begin{aligned}
reassign\_volunteer(v, R = (V, J, \mathcal{A}, T_V, T_J)) =& add\_assignment(t_v^*, t_j^*, R^*) \\
\mathcal{A}^- =& \mathcal{A} \setminus A_v \\
T_V^+ =& T_V \cup \{(v, s, e) | (v, j, s, e) \in A_v\} \\
T_J^+ =& T_J \cup \{(j, s, e) | (v, j, s, e) \in A_v\} \\
R^* = (V, J, \mathcal{A}^-, T_V^*, T_J^*) =& recombine\_timeslots(V, J, \mathcal{A}^-, T_V^+, T_J^+) \\
t_v^* =& t_v^* \in T_V^*,\ t_v^* \text{ is the longest time slot in } T_v^* \\
t_j^* =& t_j^* \in T_J^*,\ t_j^* \text{ is the best match for } t_v^* \text{ in } T_v^*
\end{aligned} \tag{43}
$$

All matches A_v of v are deleted and the new time slots from the periods of the deleted assignments are added to the set of available volunteer time slots T_V and vacant job time slots T_J. The new time slots are recombined and the new time slot t_v^* represents the complete available time interval of the volunteer v. The time slot T_V^* is assigned to the best available job time slot t_j^*.

Job Reassignment Job reassignments can be applied in the same way as volunteer reassignments.

Assignment Splitting The number of assignments can be increased if parts of existing volunteer assignments get deleted and are reassigned to other available volunteers.

The *split_job* function splits an assigned job time slot into two shorter parts. The function searches for the best candidate among all unassigned volunteers for every assigned job. The candidate who improves the sum of rating values for both shorter time slots the most is chosen for the new assignment. The old assignment is deleted and replaced by two new assignments. One assignment contains the initial combination of job and volunteer, the other contains the best-found candidate and the job of the initial assignment.

7. Implementation

The previously introduced local search approach was implemented in the functional programming language Haskell. This section describes the application parameters and the contents of the Haskell modules and presents a few selected functions from the source code. Note, that some parts of the software were created within the previously mentioned "Projektarbeit" of the author, which implemented another version of the construction algorithm. However, only the Database module is imported almost completely, while all other modules were either created from scratch or were adjusted comprehensively within this work. A user-manual for the software can be found within appendix A.

7.1. Parameters

The application can be configured with additional parameters. A list with all parameters is presented in table 7.1.

Parameter	**Name**	Domain	**Explanation**
dbPath	Database Path	String	Path to the volunteer and job database
outOpt	Output Option	Int	0=Debug, 1=NormalRoster, 2=VerboseRoster, 3=VerboseVolunteers
maxIterations	Maximum Iterations	Int	Maximum consecutive iterations of the local search.
maxStagnations	Maximum Stagnations	Int	Maximum recursive calls of the local search without improvements
mD	Minimum Assignment Duration	Int	Shortest allowed assignment duration in seconds
quW	Quality Weight	Int	Weight of the Quality Soft Constraint
vDW	Volunteer Distribution Weight	Int	Weight of the Volunteer Distribution Soft Constraint
jDW	Job Distribution Weight	Int	Weight of the Job Distribution Soft Constraint
vCW	Volunteer Completeness Weight	Int	Weight of the Volunteer Completeness Soft Constraint
jCW	Job Completeness Weight	Int	Weight of the Job Completeness Soft Constraint

Table 7.1: Application Parameters

7.2. Modules

Main This module is used to execute the rostering software. All program parameters, including the values of the hard constraints, the algorithm configuration and the soft constraint weights are parsed, and saved within an instance of the *Configuration* data type. Furthermore, the volunteer and job data are read from a database and stored within lists. The rostering process is started by an execution of the *createRoster* function.

Model The **Model** module contains the implementation of all types defined in section 3. The **Roster** sub-module implements the Roster and the RosteringStats type, which contains different metrics of the rostering process, e.g. the parameters, the number of recursive calls, and the number of fully assigned jobs and volunteers.

The **Configuration** sub-module implements the The Configuration, Constraints, and Weights types. Both local search configurations, the *maxIterations* and *maxStagnations* parameter are contained in the Configuration type. All soft constraint weights, e.g. the *qualityWeight* and the *volunteerDistributionWeight*, are included in the Weights type. The constraints type contains the hard constraint parameters, e.g. the *minDuration*.

The qualifications and requirements are implemented in the **Qualification** sub-module. Note, that the skills types are implemented in the database module.

The **Timeslots** sub-module contains the definition of the Timeslots type class and the definitions of its instances, the VolunteerSlot, JobSlot and Assignment types. Additionally, all functions regarding the manipulation of timeslots are included within this module, e.g. the implementation of the *recombineVolunteers* and *recombineJobs* function mentioned in section 6.2.1. The source code of the *recombineTimeslots* function is presented in code snippet 7.1.

```haskell
recombineTimeslots :: (Timeslot a, Eq a, Show a) => [a] -> [a] -> [a]
recombineTimeslots [] combinations = combinations
recombineTimeslots (slot:slots) combinations =
  recombineTimeslots remainingSlots (combi:newCombinations)
  where
  neighborhood = remainingSlots++combinations
  slotStart = start slot
  slotEnd = end slot
  number = slotNumber slot
  pred = find (\s -> number == slotNumber s &&
                              end s == slotStart) slots
  succ = find (\s -> number == slotNumber s &&
                              start s == slotEnd) slots
  neighbors = catMaybes [pred, succ]
  combi =
    if L.null neighbors
    then slot
    else L.foldr (\l r ->
                   if (end l) == (start r) then setStart r (start l)
                   else if end r == start l then setEnd r (end l)
                   else r) slot neighbors
  remainingSlots  = slots L.\\ neighbors
  newCombinations = combinations L.\\ neighbors
```

Code Snippet 7.1: Recombine Timeslots Function

In each recursive call of the function, a combination of a time slot and its surroundings is created. The surrounding of a timeslot consists of a predecessor and a successor (lines 11,13). If a predecessor exists, its start-time is adopted by the combination(line 19). Likewise, if a successor exists its end time is adopted (line 20). As soon as no timeslots remain within the possibly fragmented set of time slots, the function terminates and the list of combinations is returned (line 2).

Database The Database module includes all functions and types which are used for database operations. The Database module was created within the previously mentioned "Projektarbeit" of the author and was only slightly adjusted within this work.

The database operations were implemented with the persistent library of the Yesod framework[7]. The persistent library is able to convert Haskell type definitions into SQL types and offers type safety within SQL queries. Because of this, instead of raw SQL queries, the *selectList* functions from the persistent library are used, which import data from SQL databases and translate rows of database tables directly into lists of instances of Haskell types. All IO database operations are defined within the **DatabaseIO** sub-module.

However, types that are used within database queries must be defined within a special environment provided by the persistent library. The type definitions are contained within the *DatabaseModel* sub-module. All Haskell types which are included in these database types must be derived by the derivePersistField function. These types are specified within the **DerivedDatabaseModel** sub-module.

CLI The CommandLineIO model includes output functions for different purposes. This includes debugging information of the algorithm and the output of generated rosters. Rosters can be printed either grouped by jobs, or by volunteers. Furthermore, rosters can be printed either in normal, or in verbose mode. In normal mode, simple outputs are generated, which only include the name of the jobs and volunteers and their start end times. The verbose mode adds the qualifications and requirements to each job and volunteer output and prints the lists of unassigned volunteer and job time slots as well

Rostering The Rostering module controls the rostering process and is configurable by the *maxIterations* and *maxStagnations* parameters.

The initial roster is created by the *constructRoster* function, which uses the lists of volunteer and job timeslots, which were queried from the database.

The *iterateLocalSearch* function executes *maxIterations* local search runs consecutively. Throughout the iterations, the best-rated roster is remembered and used as input (seed) for subsequent runs. The rostering process terminates as soon as the last local search iteration finished.

LocalSearch The LocalSearch module contains the implementations of the construction and local-search algorithms described in section 5.

The input for the local search algorithm is recursively created by the *constructRoster* function from the **Construction** sub-module, which implements the construction algorithm described in section 5.1. The *constructRoster* function takes a roster and creates assignments from the lists of remaining job vacancies and available volunteers. The search-list used by the construction algorithm is generated by the *createSearchList* function in the **Preprocessing** sub-module.

The **Rating** module implements the rating-functions defined in section 6. Rosters are rated by the *getRosterValue* function, while assignments are rated by the *getAssignment-Value* function.

The operator functions which are used by the neighborhood functions of the local search, are included in the **Operators** sub-module. E.g. new assignments are added to a roster with the *addNewAssignment* function presented in code snippet 7.2.

```
1  addNewAssignment job vol jobMap conf roster@(Roster jobs vols psas _) =
2    if jobVolDoMatch job vol (constraints conf) roster
3    then addRosterValue conf jobMap newRoster
4    else roster
5    where
6    key = jobNo job
7    slotSplits = splitTimeslots job vol
8    nJob = job { jobSlotStart=slotStart slotSplits,
9                 jobSlotEnd  =slotEnd slotSplits }
10   nVol = vol { volunteerSlotStart=slotStart slotSplits,
11                volunteerSlotEnd =slotEnd slotSplits }
12
13   newAssignments =
14     insertWith (++)(key)
15                [createAssignment nJob nVol (weights conf) jobMap] psas
16   remainingVolunteers = (L.delete vol vols)
17                         ++(volunteerSlots slotSplits)
18   remainingJobs = (L.delete job jobs)
19                   ++(jobSlots slotSplits)
20   newRoster = roster { volunteers=remainingVolunteers,
21                        vacancies=remainingJobs,
22                        assignments=newAssignments}
```

Code Snippet 7.2: addNewAssignment Operator Implementation

The function creates and adds a new assignment from a job time slot and one of its *candidates* to an existing roster. If the time durations of the input slots do not overlap completely, new time slots for the unallocated time-periods are created (line 7). The new assignment is generated by new job and volunteer time slots which represent the overlap-period (lines 9, 11), and is added to the list of all assignments of the job (line

14). The assigned time slots are deleted from the list of vacancies (line 18) and remaining volunteers(line 16), while the time slots of the unallocated time-periods are added (lines 17, 19). After all changes are applied (line 22), the rating of the roster is updated and it is returned (line 3).

The local-search algorithm described in 5.2.1 and all neighborhood functions defined in 6.2 are implemented in the **LocalSearch** sub-module. The *localSearch* function recursively optimizes a roster. Within each recursive call (repetition), all neighborhood functions are evaluated independently. The seed for the next repetition is chosen dependent on the value of the *stagnations* counter. Each time no improvement can be found within the list of results from the neighborhood functions, *stagnations* gets incremented and a random result of the neighborhood functions is chosen. As soon as *stagnations* is equal to *maxStagnations* the local search terminates.

The progress of the localSearch function, which includes the number of repetitions and the value of the best generated roster, is documented within an instance of the *RosteringStats* type. An example for the evaluation of a neighborhood function is presented in code snippet 7.3.

```
1 let jobSplit =
2     (repairRoster  $
3      getBestRoster $
4      L.map (\a -> splitPosition a conf posMap newRoster)
5      remainingAssignments,
6     rosterStats { posSplitCount = (posSplitCount rosterStats)+1})
```

Code Snippet 7.3: Recombine Timeslots Function

The jobSplit neighborhood function is executed for every remaining assignment, which was not included in the random deletions (line 5). Among the list of all results, the best option is chosen (line 3) and further possible assignments are added by an execution of the construction algorithm (line 2). Additionally the number of executions of the *jobSplit* function is incremented in the RosteringStats type instance (line 6).

8. Experiments

The following section examines different aspects of the local search algorithm presented in sections 5 and 7.

8.1. Experimental Setup

The data are obtained from the documentation of a real mission (”Sanitätsdienst”). All volunteer names and the dates of the timeslots were changed to provide anonymity. The data consist of 21 volunteer and 26 job time-slots. The start and end times of the volunteer timeslots differ, while the job time-slots have equal begins and ends. All job time-slots demand equal or more time every volunteer from the data set is able to offer. Because of that, all volunteers can be fully allocated by assignments to jobs, whereas not all jobs can be allocated by assignments due to the shortage of volunteers.

8.2. Local Search Configurations

The following section examines the results of the local search algorithm for different input values of the *maxStagnations* and *maxIterations* parameters.

The test cases of the following experiments are listed in table 8.1.

Test-Cases / Weights	**mD**	**quW**	**jCW**	**vCW**	**jDW**	**vDW**
Quality	1h	1	0	0	0	0
Job Completeness	1h	0	1	0	0	0
Volunteer Completeness	1h	0	0	1	0	0
Job Distribution	1h	0	0	0	1	0
Volunteer Distribution	1h	0	0	0	0	1

Table 8.1: Test-Case Parameters

In each test case, a single soft-constraint is optimized. That is, one weight is set to one, while all others are set to zero. The values of *maxIterations* and *maxStagnations* vary between each experiment.

8.2.1. Greedy Local Search

As soon as *maxStagnations* is set to zero, exclusively improved rosters are accepted by the local search and the algorithm stops, as soon as worse results are generated. Only the current neighborhood of the initial roster is exploited and no further search space exploration is performed by the algorithm. This variant of the algorithm is called *greedy* local search.

To assess, if a *greedy* local search is able to improve the initial roster of the construction algorithm, ten consecutive executions of the algorithm were measured for each test case.

In each execution, the *maxIterations* parameter was increased by one, while the *maxStagnations* parameter was constantly set to zero. Therefore no *stagnations* were tolerated and no random deletions were applied.

The results of this experiment showed, that the local search algorithm was able to improve the initial *Job Completeness* test case rating from 40% to 45% for every value of *maxIterations*. Surprisingly, the results did not show any further improvement to all remaining test cases for every value of *maxIterations*.

It can thus be suggested, that the initial roster of the construction algorithm represented a local optima, to whom no improvements could be found in the available neighborhood by the *greedy* local search.

8.2.2. Exploring Local Search

An increasing number of random deletions enables the local search to *explore* different neighborhoods. It was examined if increasing values for *maxStagnations* can generate improvements compared to the results obtained by the *greedy* local search in experiment 8.2.1.

For every input value of *maxStagnations*, one iteration of the local search was executed. In each test, *maxIterations* was that to one. Therefore, only one subsequent execution of the local search was examined in each execution of the algorithm.

As shown in figure 8.1, the local search was able to improve every test case multiple times. Remarkably, most of the enhancements can be recognized for large values of *maxStagnations*, especially within the interval $[5, 7]$.

The *volunteerCompleteness* was already completely satisfied within the initial roster of the construction algorithm. This is understandable because more jobs than volunteers exist within the data set and every job is at least as long, as the highest time a volunteer is able to offer.

Only the initial job completeness rating could be improved for a *maxStagnations* value of zero, which happens to be the improvement that could also be examined in the previous section.

However, the algorithm was not able to enhance the roster in every execution. Instead, improvements could be found only in a maximum of 50% of all executions for values of *maxStagnations* greater than zero.

The data show that the *exploring* local search algorithm is able to improve the initial roster. For the available data source, the statement is especially true for higher numbers of *maxStagnations*.

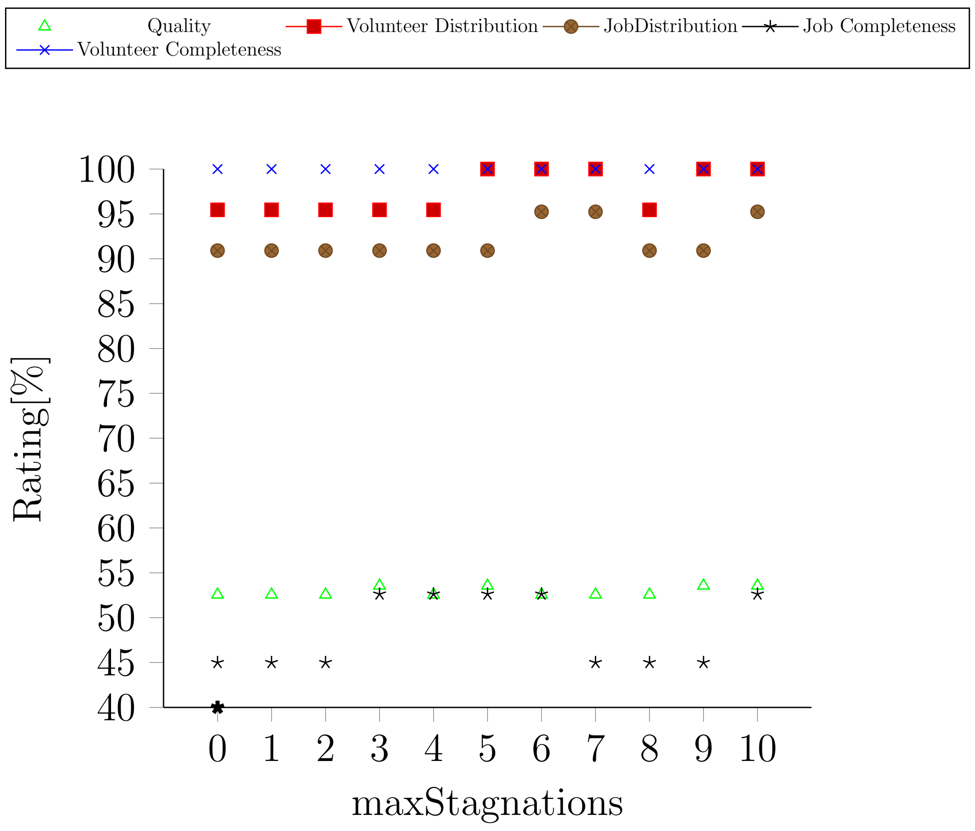

Figure 8.1: Experiment Results for Exploring Local Search

8.2.3. Exploration and Exploitation

The experiment examined if the local search algorithm is able to find improvements, when both exploration and exploitation of the neighborhood are combined.

In the tests, equal values were set for *maxStagnations* and *maxIterations*, while both were increased for every execution of the algorithm.

Each increase of *maxIterations* enlarges the amount of local search *repetitions*. That is, for each additional iteration, the number of *repetitions* increases at a minimum of the value of *maxStagnations*. Therefore, an increase of *maxIterations* is comparable to a linear increase of *maxStagnations*. Note, that *iterations* of the local search use the best-found roster as seed, while independent executions of the local search start from the initial roster. It is assumed, that the *iterated* local search approach offers benefits for large data sets. However, this assumption could not be verified with the available small data set.

Remarkably, the results of the previous experiment from figure 8.1 could be slightly improved for almost all test cases.

The greatest number of improvements were generated for the quality and the volunteer distribution test cases. They enhanced from four to six and five to eight improvements among different executions respectively.

On the contrary, the amount of job completeness enhancements remained the same,

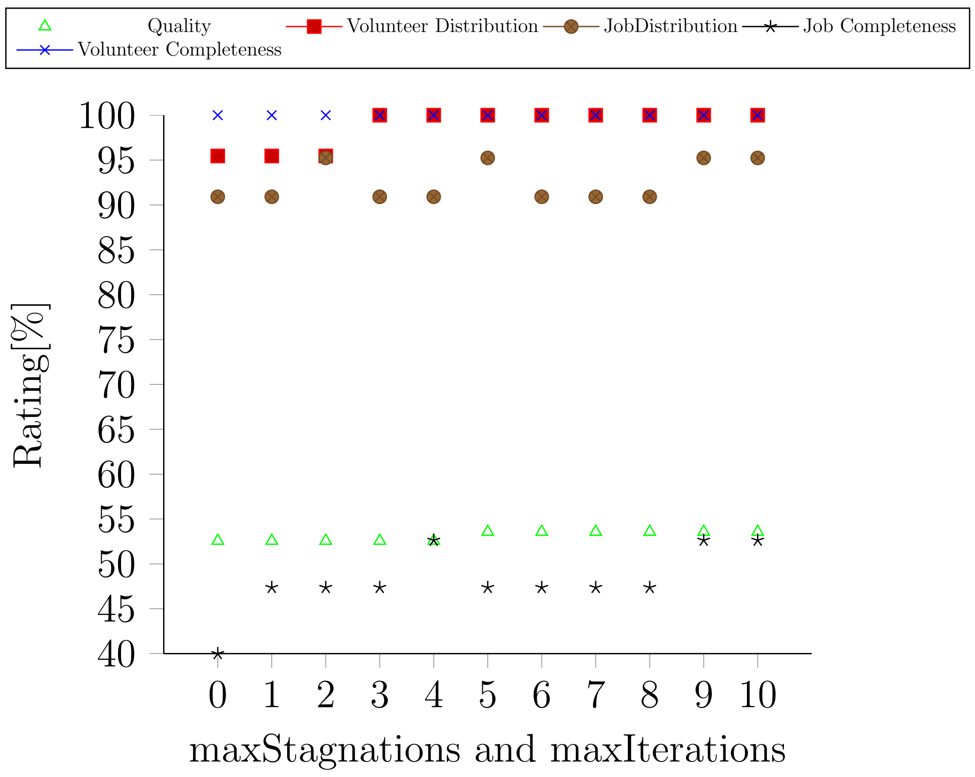

Figure 8.2: Experiment Results for Exploring and Exploiting Local Search

but the quality of the improvements decreased in three cases.

Despite this, the experiment seems to confirm that larger values of *maxStagnations* and *maxIterations* lead to better roster optimizations.

However, caution must be exercised in the interpretation of the results. While improvements are observable generally for higher values of *maxStagnations* and *maxIterations*, it is not clear that the enhancements were generated by synergy effects between both parameters. Given that the obtained results are subject to random deletions, it is not proven, that the improvements were generated by the combination of *iterations* and *stagnations*. Instead, equal results may be receivable by an increase of only the *maxRepetitions* parameter.

8.3. Neighborhood Function Applications

The local search algorithm uses multiple neighborhood functions to improve its input. To evaluate if and how often the result of each function is chosen as a new seed by the local search algorithm, the algorithm was executed ten consecutive times, with *maxStagnations* and *maxIterations* set to five.

The total numbers obtained in all executions are shown in table 8.2.

The sum of recursive calls (repetitions) from all executions is represented in the *Localsearch Repetitions* column. All other columns represent the neighborhood functions of the local search algorithm. The test cases are presented in the rows of the table. Each cell contains the values

The table shows the sum of local search *repititions* and neighborhood-function applications, which were measured in ten independent runs of the local search algorithm. It allows comparisons between the number of times the result of a specific neighborhood function was chosen as a new seed by the local search algorithm.

Test-Case	**LocalSearch Repetitions**	**Exchange Assignee**	**Reassign Volunteer**	**Reassign Job**	**Split Job**
Quality	134	39	20	27	31
Job Completeness	218	54	51	46	41
Volunteer Completeness	70	12	10	19	19
Job Distribution	90	16	26	11	25
Volunteer Distribution	88	22	20	16	18
$\sum$	600	143	127	119	134
Sum of Function Applications		523			

Table 8.2: Neighborhood Function Applications

Remarkably, the results of all neighborhood functions were chosen almost the same number of times overall. However, the neighborhood functions are applied differently often for varying test cases. The rosters generated by the *exchange assignee* function were applied most of the time, while the results of the *reassign job* function were chosen the least.

Most surprisingly, high numbers of *splitJob* results were applied to optimize the *job compactness* soft constraint. This was unexpected because the *split job* function increases the number of short assignments, which in return decreases the value of the *job compactness* soft constraint.

Interestingly, the number of local search *repetitions* varies strongly between different test cases. The data show, that the *job completeness* test case generated by far the most local search repetitions. It also offered the highest rating improvements in graphic 8.2. Notably, the *volunteer completeness* function produced the least amount of repetitions, while no improvements were generated in graphic 8.2.

The correlation between the number of iterations and the number of improvements is noteworthy. It seems that the algorithm produces better results if more repetitions are executed. This is somewhat understandable, because the more the results of subsequent local search repetitions vary, the slower the *stagnations* counter approaches the *maxStagnations* limit and the more recursive calls are performed.

The opposite result is recognizable for the *quality* test case. It produced the second most repetitions but offered the least amount of rating increase within the data of graphic 8.2. Due to this, the observed correlation between the number of recursive calls and the number of improvements may be specific to the completeness functions, but cannot generally be observed for all neighborhood functions.

Note, that the sum of *local search repetitions* does not equal the sum of function applications. The values differ because the number of function executions is not increased within the last *repetition* of the local search, where only the best found roster is returned, but the *repetitions* counter is increased nonetheless.

Overall, the local search algorithm seems to prefer different neighborhood functions in varying scenarios. A correlation between the number of executions and the quality of the improvements may exist, but can not be verified with the present data. However, the discoveries of this experiment are based on a limited number of executions. A major source of uncertainty comes with the random selections of the algorithm, which are able to bias the observations of this experiment. The acquired insights must therefore be treated carefully.

9. Conclusions

This thesis presented a local search approach for the scheduling of volunteers in missions of the German Red Cross Society.

While many known rostering problems are defined by static shifts, the rostering problem introduced in this thesis was defined in terms of independent time-slots with variable start and end times.

The use of time-slots offers great flexibility to the scheduling process but introduces a large search space and challenges to contain consistency in the roster. Therefore, multiple operator functions were defined, by whom safe adjustments to rosters are possible.

The validity of single assignments and the roster are determined by multiple hard constraints, which must always be satisfied. Soft constraints are defined by functions and measure the satisfaction of a single optimization target. The quality of rosters is evaluated by a rating function which is composed by soft constraint functions, which can be prioritized by weights.

The introduced approach consists of a construction algorithm, which fills rosters from lists of volunteer and job timeslots and a local search algorithm, which optimizes the roster in regard to the rating function.

The approach was implemented in the functional programming language Haskell. We found, that Haskell was well suited for this problem and offered many useful features, e.g. type classes and pattern matching, which were extensively used in the implementation.

Experiments showed, that the implemented local search algorithm is able to improve the quality of rosters in regard to different optimization targets. However, the available data set was not sufficient to prove synergy effects between different local search configurations.

Note, that the presented solution method for rostering with variable timeslots is not limited to the treated problem, but may be applied to other domains as well.

Future work should include the implementation of a more sophisticated user-interface and further tests with larger data sets. Also, additional features may be added, e.g. the measurement of distances between subsequent jobs of volunteers, or the scheduling of groups in addition to independent assignments of every volunteer.

Bibliography

[1] B Domres et al. “The German approach to emergency/disaster management”. In: *Medicinski arhiv* 54.4 (2000), 201—203. URL: http://europepmc.org/abstract/MED/11117024.

[2] URL: https://www.drk.de/hilfe-in-deutschland/bevoelkerungsschutz (visited on 05/19/2021).

[3] Mauro Falasca and Christopher Zobel. “An optimization model for volunteer assignments in humanitarian organizations”. In: *Socio-Economic Planning Sciences* 46.4 (2012). Special Issue: Disaster Planning and Logistics: Part 2, pp. 250–260. ISSN: 0038-0121. DOI: https://doi.org/10.1016/j.seps.2012.07.003. URL: https://www.sciencedirect.com/science/article/pii/S0038012112000353.

[4] E.K. Burke, P. De Causmaecker, and G.V. et al. Berghe. “The State of the Art of Nurse Rostering”. In: *Journal of Scheduling* 7 (2004), pp. 441–499. DOI: https://doi.org/10.1023/B:JOSH.0000046076.75950.0b.

[5] URL: https://www.haskell.org/ (visited on 05/19/2021).

[6] URL: https://docs.haskellstack.org/en/stable/README/ (visited on 05/20/2021).

[7] URL: https://www.yesodweb.com/book/persistent (visited on 05/19/2021).

[8] A.T Ernst et al. “Staff scheduling and rostering: A review of applications, methods and models”. In: *European Journal of Operational Research* 153.1 (2004). Timetabling and Rostering, pp. 3–27. ISSN: 0377-2217. DOI: https://doi.org/10.1016/S0377-2217(03)00095-X. URL: https://www.sciencedirect.com/science/article/pii/S037722170300095X.

[9] JF. Cordeau, G. Laporte, and F. et al. Pasin. “Scheduling technicians and tasks in a telecommunications company”. In: *Journal of Scheduling* 13 (2010), pp. 393–409. DOI: https://doi.org/10.1057/jors.2010.86.

[10] Patrick De De Causmaecker and Greet Vanden Berghe. “A categorisation of nurse rostering problems”. In: *Journal of Scheduling* 14 (2010), pp. 3–16. DOI: https://doi.org/10.1007/s10951-010-0211-z.

[11] Sean Luke. *Essentials of Metaheuristics.* second. Lulu, 2013.

[12] Bilgin Bilgin et al. “Local Search Neighbourhoods to Deal with a Novel Nurse Rostering Model”. In: *Annals of Operations Research* 194 (2012), pp. 33–57. DOI: https://doi.org/10.1007/s10479-010-0804-0.

[13] Andrea Schaerf and Amnon Meisels. “Solving Employee Timetabling Problems by Generalized Local Search”. In: *AI*IA 99: Advances in Artificial Intelligence.* Ed. by Evelina Lamma and Paola Mello. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 380–389. ISBN: 978-3-540-46238-5.

[14] E. Burke, T. Curtois, and L. et al. van Draat. “Progress control in iterated local search for nurse rostering”. In: *Journal of the Operational Research Society volume* 62 (2011), pp. 360–367. DOI: `https://doi.org/10.1057/jors.2010.86`.

A. Source Code

The source code of the solution approach‘s implementation, the database and the scripts for the execution of the experiments are included on the attached CD.

B. User-Manual

B.1. Prerequisites

Before the source code can be compiled and build the following software must be installed.

- Glasgow Haskell Compiler, version ≥ 8.6
- Haskell Tool Stack, version ≥ 2.5
- Cabal, version ≥ 2.4

The listed software can be downloaded from

- `https://www.haskell.org/downloads/`.
- `https://www.haskell.org/ghcup/`.

B.2. Project-Structure

The project folder contains multiple directories and files. The contents are described in the tables B.1 and B.2.

Files	**Contents**
backend.cabal	Automatically created with the *stack build* command. Adjustments should be avoided
package.yaml	Dependencies and build configuration
stack.yaml	Automatically generated by the stack init command should not be manually adjusted

Table B.1: Contents of Project Files

Directory	**Contents**
app	The Main module, which is used to execute the application
example_databases	The database with volunteer and job data
experiments	The shell scripts for execution of the experiments from section 8
src	The modules of the project described in section 7

Table B.2: Project Directory Contents

B.3. Build

The project must be initialized previously to the first build process. The initialization may take a while because all dependencies are downloaded. The commands for initialization and compiling are shown in code snippet B.1.

```
# Initilize the project
stack init
# Compile and build the application
stack build
```

Code Snippet B.1: Initialize Project

B.4. Code Execution

The application can be executed with the following command. The list of parameters is presented in table 7.1.

```
stack exec backend-exe dbPath outOpt maxIt maxSt mD quW vDW jDW vCW jCW
```

Code Snippet B.2: Application Execution

B.5. Database

The database is contained in the *example_database* directory and can be accessed via an SQLite browser. The database consists of two tables, *volunteer* and *job*, which are shown in tables B.3 and B.4.

Fields	**Type**	**Value Constraints**	**Description**
id	INTEGER	-	Primary Key, whichmust not be manually adjusted
name	VARCHAR	-	Name of the job
priority	INTEGER	[1-10]	Priority of the job
min_age	INTEGER	-	Minimum required age for this job
start_time	TIMESTAMP	-	Start time of the job
end_time	TIMESTAMP	-	End time of the job
m_q	VARCHAR	EH, San, RS RA, NFS, AZ	Require medical skill of the job
l_q	VARCHAR	NoLq, TF, GF ZF, VBF	Required leadership skill of the job
t_q	VARCHAR	["sCBRNE"], ["sB"] ["sC1"]	Required technical skills of the job

Table B.3: Job Database Table

Fields	**Type**	**Value Constraints**	**Description**
id	INTEGER	-	Primary Key, which must not be be manually adjusted
name	VARCHAR	-	Name of the volunteer
age	INTEGER	-	Age of the volunteer
start_time	TIMESTAMP	-	Start time of the volunteer
end_time	TIMESTAMP	-	End time of the volunteer
m_q m_q	VARCHAR VARCHAR	EH, San, RS RA, NFS, AZ	Medical skill of the volunteer
l_q	VARCHAR	NoLq, TF, GF ZF, VBF	Leadership skill of the volunteer
t_q	VARCHAR	["sCBRNE"], ["sB"] ["sC1"]	Technical skills of the volunteer

Table B.4: Volunteer Database Table

B.6. Experiments

All test cases from section 8 are executed with shell scripts. The scripts are included within the **experiments** directory and can be executed in the following way:

```
# Executes the first experiment
./experiments/greedyLocalSearch.sh

# Executes the second experiment
./exeriments/exploringLocalSearch.sh

# Executes the third experiment
./experiments/explorationAndExploitation.sh

# Executes the fourth experiment
./experiments/neighborhoodFunctionApplicationTest.sh
```

Code Snippet B.3: Experiment Scripts

Variable Reference

Name	Variable	Content	Description
A Volunteer	v	-	A single volunteer
All Volunteers	V	$\{v_1, .., v_n\}, n \in \mathbb{N}$	The Set of all volunteers
A Job	j	-	A single job
All Jobs	J	$\{j_1, .., j_m\}, m \in \mathbb{N}$	The set of all jobs
Start Time	s	$s \in \mathbb{N}$	A start time definition, containing a Unix time value
End Time	e	$e \in \mathbb{N}$	A end time definition, containing a Unix time value
Assignment	a	(v, j, s, e)	An assignment of volunteer v to job j from start s until e
Volunteer Assignments	A_v	$A_v = \{a \in A \mid \text{v is assigned to a}\}$	The set of all assignments of a volunteer v.
Job Assignments	A_j	$A_j = \{a \in A \mid \text{j is assigned to a}\}$	The set of all assignments of a job j.
Assignments	$\mathcal{A}$	$\mathcal{A} \subseteq (V \times J \times \mathbb{N} \times \mathbb{N})$	The set of all assignments.
Volunteer Timeslot	t_v	(v, s, e)	A timeslot of volunteer v from s until e
Volunteer Timeslots	T_v	$T_v = \{t \in T_V \mid \text{t is timeslot of v}\}$	The set of all timeslots of a volunteer v
All Volunteer Timeslots	T_V	$T_V \subseteq (V \times \mathbb{N} \times \mathbb{N})$	The set of all volunteer timeslots
Job Timeslot	t_j	(j, s, e)	A timeslot of job j from s until e
Job Timeslots	T_j	$T_j = \{t \in T_J \mid \text{t is a timeslot of j}\}$	The set of all timeslots of a job j
All Job Timeslots	T_J	$T_J \subseteq (J \times \mathbb{N} \times \mathbb{N})$	The set of all job timeslots
Roster	R	$(V, J, T_V, T_J, \mathcal{A})$	A roster
A weight	ω	$\omega \in \{1, .., 10\}$	The set of weights for the roster rating function
Weights	Ω	$\{\omega_1, .., \omega_4\}$	The set of weights for the roster rating function