Chapter 4

Introduction to the Shift Scheduling Domain

"I like work . . . I can sit and watch it for hours."

—unknown

In this chapter an explanation of the domain of shift scheduling is given. The introduction shows why shift scheduling can be quite a problem worth being further examined. Next, several definitions concentrate on the various key terms of this specific domain that are used throughout this and the next chapter. Furthermore, an excursus to the field beyond the technical implications of the problem domain shows how fast the domain is growing, turning it into a 'foggy swamp' search space and making optimization more and more difficult. Nevertheless, the section on the requirements gives various hints on what to focus on when we try to find the constraints of our problem domain. The chapter concludes with an examination of the optimization issues in this problem domain.

4.1 Introduction

Right from its beginning the ∗FLIP++ project has always been strongly coupled with problems encountered in the process of steel production. This was partly due to the fact that the entire project has been initiated by a research cooperation with the Austrian steel production industry. A wide range of publications have been published on the steel production domain over the last couple of years out of this fruitful research cooperation. In [20] a more in-depth discussion of this domain is given in connection with fuzzy scheduling. Moreover, Dorn and Shams [4] discuss an expert system approach designed for this domain.

With versatility being a key objective of the ∗FLIP++ project, next choice was to move away from the original steel production domain. The advantages should be twofold: Firstly, we expect to gain further experience about the process of knowledge extraction and transformation into a ∗FLIP++ compatible format, which should lead us eventually to a system that is a quite generic tool as far as the representation of domain knowledge is concerned. Secondly, another problem domain may also give us numerous hints on weaknesses of the system. These weaknesses may be localized in the choice of optimization method, in the performance of the system, and in the representational power provided by the various fuzzy tools of the FLIP++

library.

The choice in favor of the domain of shift scheduling was a promising one out of various reasons. To begin with, there has been quite a lot of research in this area only recently. One major conclusion drawn out of these research efforts is the fact that one runs rather soon into major difficulties in this area when conventional optimization methods are applied, e.g. with simplex, enumeration or backtracking methods (for a more in-depth discussion on this particular aspect turn to [7]). This is always a good starting point for the introduction of a fuzzy constraint satisfaction system, one like ∗FLIP++. And indeed, as we will see later on, shift scheduling is a good ground for such an approach. According to Gärtner and Wahl [8] a high degree of fuzziness can be attributed to many requirements encountered with a shift scheduling problem. They also argue that due to the complexity of the problem and the lack of powerful optimization methods it is more important to move the focus from automation of design towards aiding design. The project team also believes that ∗FLIP++ should be a more supportive tool that allows to find an optimal schedule via the interaction with the knowledge engineer. Nevertheless, optimism is still there and ∗FLIP++ strives to eventually accomplish a system that comes close to an automation of design. Moreover, the flexibility offered by ∗FLIP++ when it comes to the definition of fuzzy variables, fuzzy constraints and aggregation operators should make it superior compared to classical optimization methods which often show a lack of representational power. Moreover, the repair-based optimization process that tries to tackle constraint violations with specifically defined repair steps seems to be more promising than classical approaches.

Furthermore, the problem of developing 'good' shift schedules is a highly practical application. It also has a lot of consequences on the people that are involved and concerned which makes it even more interesting. These sometimes more social than technical implications will be discussed in more detail in Section 4.3. Although my prototype solution does not include the social requirements in full scale, I believe that the excursus is well-justified. It investigates whether a future enhancement of the application towards this direction can be done efficiently; a further probation of the representational power of ∗FLIP++.

4.2 Definitions

In the following several important definitions are given. We concentrate on the key terms that are used throughout this and the next chapter. Some illustrations should try to ease the understanding of the various definitions.

Shift

A *shift* defines duties (e.g., morning shift from 6 a.m. to 2 p.m.). Shifts name a certain time interval at one or more specific days of the week where operation is required. Of course, it is possible to have different shifts for different days of the week. For example, many shift scheduling problems require particular shifts at the weekend. When taking a decision on the structure of a shift, one has to consider various aspects, such as length of operation time, breaks, overlapping.

Figure 4.1 shows a graphical definition of various shift types.

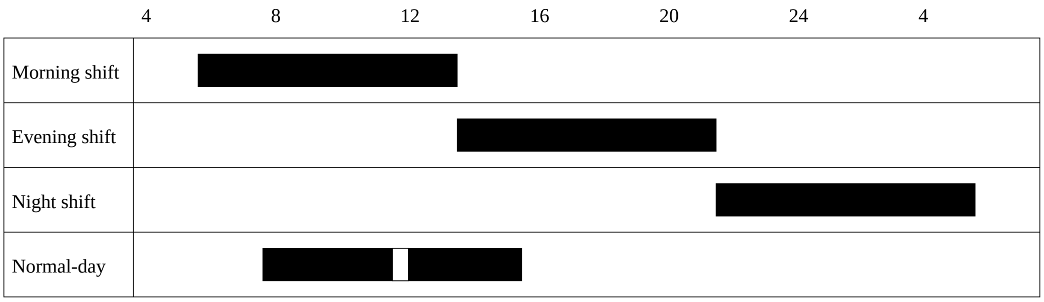


Figure 4.1: Shifts

Groups/Subgroups

It is very common, especially in the shift schedules of production plants to aggregate employees in functional and/or organizational units. These units are simply called *groups*. One can also think of a certain hierarchy of grouping: with groups at the top level and below *subgroups* as further partitions of these groups. At the bottom level comes a single employee. At each level it is known how many employees are needed to form the respective group. As will be shown in the next chapter, our problem uses two levels, namely group and subgroup. Of course, this particular restriction is only due to the specific application problem discussed in this thesis and is not at all a necessity.

Roster

A *roster* defines which employees have to work on which shift on a specific day. The requirements for the operation are the primary guideline for the design of a roster. If it is known on which days of the week and at what times of the day how many employees are required for operation, it is possible to allocate an appropriate shift and group at that time. For example, on Monday two groups in the morning shift have to be available in order to make sure a sufficient level of operation. Figure 4.2 is an example for a roster with a length of 50 days and 5 groups. It allocates the shifts in clusters of 6 shifts and 4 days off using a rotation pattern.

DAY	1	2	3	4	5	6	7	8	9	10
Group 1	M	M	E	E	N	N				
Group 2			M	M	E	E	N	N		
Group 3					M	M	E	E	N	N
Group 4	N	N					M	M	E	E
Group 5	E	E	N	N					M	M

Figure 4.2: Roster with the length 50 days (M. . . Morning shift, E. . . Evening shift, N. . . Night shift)

Generally, we can expect rosters of a more complex nature than the above one. Then, the description of the shift-sequence may not be as easy as with a rotation cycle. Furthermore, an increased number of both groups and group levels introduces a further degree of complexity. It is also quite common and sometimes necessary depending on the nature of the problem at hand to define rosters of a longer duration (e.g., 48 weeks).

Shift schedule

With the previous definitions, we can now take a further step and define the term *shift schedule* as an aggregated view. Shift schedules fix the distribution of working hours of employees. They are primarily defined by shifts and rosters.

Time horizon

The *time horizon* of the shift scheduling process consists normally of at least three layers. At the *day* level the position and internal structure of shifts have to be defined. This corresponds to the definition of a shift given above. The *week* level consists of rosters that fix the nominal work force per day in the period of a week. Over the period of a *year* the nominal and actual work force have to be balanced. Several sources can cause the differences between nominal and actual work force. Among them are higher levels of production, further education, holidays, sickness leave, etc. They are mainly handled by overtime or stand-by staff (included as an extra group/subgroup in the design process).

4.3 Beyond the technical issue

The social implications of shift work arrangements are at least as important as the more technical issue of finding an appropriate shift schedule that fits into the constraints model of operation.

The idea of replacing shift work by individual flexibilization of working time is challenged by various arguments. Especially in the service sector, where consumption and production normally coincide, shift work is an undeniable instrument for work arrangements (e.g., in hospitals). Also, the tendency towards longer opening hours in the commerce sector weakens the correspondence between company working hours and individual working hours, i.e., work can be shifted more into the evening hours or morning hours, or it can be even split into several shifts.

When designing a shift schedule, the organizational environment has to be analyzed first. It is a serious mistake to separate the pure scheduling problem of shift scheduling from its context. Successful shift schedule design can only proceed with a cooperative strategy where the computer system supports a flexible construction of a solution as well as its analysis and evaluation. Furthermore, the system should be based on requirements that represent an aggregate view of all groups and interests concerned.

The following list shows some of the possible effects on both employees and employers caused by the introduction of a new shift schedule:

- income and costs
- the position of leisure time
- physical stress (sleep, relaxation, etc.)
- mental stress (workload, restrictiveness of the shift schedule, etc.)
- social consequences concerning family life, friends, social, cultural, and sporting activities
- working hours of the company

Among the groups that are mainly concerned by shift work and its changes are:

- employees directly affected by the shift schedule
- personnel managers, shop-stewards
- craftsmen and foremen in responsibility of detailed work assignments
- employees entrusted with short-term staff assignments (holidays, illness, ...)
- departments indirectly affected by the shift schedule
  - maintenance and service

  - bus service, porters, medical service
  - controllers

- interest groups outside the company
  - major customers
  - public transport
  - employment offices
  - trade unions and employers' organizations

Of course, there is not only a one way impact of shift schedules on these groups. The influence comes also from the other direction and is a major factor in the design of a shift schedule. For example, in an industry where the various production stages are done in a vertical line by different companies, a just-in-time concept introduced by one powerful company can force the companies in previous stages, its suppliers, to adopt the same shift schedule system. Also, the role of trade unions and employers' organizations is in many industries a strong one based on a long tradition.

The large variety of different interests involved in the design process has a strong impact. Each violation encountered by a shift schedule has to be handled according to the strength of the interest and group involved. Even a shift schedule that may be of high quality in technical terms, may not be so favorable if it violates the social interests of an important group (e.g., a union). Besides, the introduction of a shift schedule cannot repeated again and again since switching from one schedule to another is quite costly and does not happen from one day to another. At the other end of the spectrum is the fact that time resources are limited. For those entrusted with the design and implementation of a new shift schedule it is fairly difficult to discuss extensively various schedules with both employees and employers as the normal day-to-day duties are an obstacle. It is simply not possible to take some time off and devote it exclusively to the planning process. Relying on the services of consultants may be too costly and increases the risk of failure due to communication problems between company staff and the consultants from outside.

The involvement of many groups of often diverging interests makes the design process of new shift schedules a sensitive task. Gärtner and Wahl [8] call the domain therefore a 'swamp'. Moreover, the inherent fuzziness of social requirements and the difficulty to grade and formalize them turns this swamp into a 'foggy' one. Eventually, the time resources allocated to shift schedule design are—as with most design problems—rather scarce. Using the metaphor once again, in order to arrive at a good shift schedule at the end of the design process in time one has to 'run in a foggy swamp'.

4.4 Requirements

The requirements that have to be taken into account during the design process of shift scheduling can be quite enormous. First, there are mainly technical ergonomic requirements which should be satisfied by a good shift schedule. Then, legal restrictions in combination with collective agreements put a shift schedule into a more or less restrictive framework. Finally, the social requirements discussed in the previous section have to be taken into consideration. Bearing in mind the various groups and interests mentioned earlier in connection with the non-technical requirements, it is fairly easy to imagine a rather high overall number of constraints encountered in shift scheduling. In [8] an estimation finds "200–300 or even more requirements" that have to be met under rather optimistic assumptions. Many of these requirements lead to conflicts making the initial scenario even worse. Although the high number of requirements reduces a fairly large solution space (to get a grasp of the amount of possible solutions, just imagine the possible combinations of shift allocations in a period of a year with several different shift types), they can lead to search spaces with few admissible solutions.

Whereas the more technical requirements, such as ergonomic and legal restrictions, can be modeled formally, even if sometimes in a tricky way, the social requirements do not lend themselves for being modeled in a formal language due to their individual and often subjective nature. Consequently, I will concentrate more on the technical requirements in the next chapter and neglect the issue of representing non-technical constraints. The interpretation of a computer evaluated shift schedule is hence up to the knowledge engineer or domain expert and takes place outside the computer system. Although a rather difficult task, the representation of non-technical, subjective constraints within the supportive computer system should be an area of future investigation. Especially with the arrival of more powerful systems and machines, it remains to be seen how soon this task can be completed.

discussion of possible constraints

4.5 Optimization

The aim of optimization is to find good schedules within the pool of possible solutions. When speaking of optimization, we also have to draw attention to the evaluation of a shift schedule. Without some sort of evaluation function a grading of shift schedules is not possible. Generally, evaluations can be:

- Global/Local

  Global evaluations consider the whole schedule and provide an overall score. A local evaluation regards only parts of a schedule and hence provides hints on problem areas and improvement.

- Overall/Features

  Overall evaluations provide an overall score for a shift schedule or part of it. A feature evaluation concentrates on a specific feature of a shift schedule (e.g., average night work per month). Both types can be global or local.

The part of ∗FLIP++ responsible for evaluation is DynaFLIP++. It evaluates constraints that form a hierarchy and provides, in principle, all of the above mentioned types of evaluation. The optimization in ∗FLIP++ is predominantly guided by an overall, global evaluation score. As hints on improvement strategies constraint violations are derived from both local and global feature evaluations.

Two important aspects considering evaluations are:

- how well does the evaluation handle antagonistic behavior of contraints
- are different constraints treated differently (e.g., by the use of a relative importance scheme)

Due to the complex nature of shift scheduling environments with many conflicting and ill-defined constraints, it is rather difficult to rely solely on a computer-based evaluation. The evaluation by the user is still needed and even absolutely necessary if the computer system models only the technical requirements of the shift scheduling problem. A continuous fine-tuning and adaptation of requirements is necessary in the design cycle. Especially, bearing in mind the risk of overlooking requirements or using wrong, ill-defined constraints.

The choice of optimizing algorithm is mainly influenced by the size of the solution space. Classical optimization methods like simplex, enumeration, or backtracking techniques show deficiencies since they work their way through a huge search space without guidance. The application of heuristics and repair-based improvement techniques appears to be far more promising, particularly in connection with a strong evaluation tool like DynaFLIP++. With the application presented in the next chapter, I opted for an iterative deepening strategy using repair based improvement. Using a genetic algorithm would also be well-justified, but was neglected due to performance issues (the number of evaluations is considerably higher leading to unsatisfying computation times with the current implementation of DynaFLIP++).

Chapter 5

The Shift Scheduling Application

5.1 Introduction

This chapter gives an explanation of the shift scheduling application and how it is embedded in the ∗FLIP++ environment. A detailed description of the problem to solve makes the reader acquainted with the subject-matter. The discussion of the problem also includes some precomputations and premises necessary for the solution stage. Then, the approach taken towards a solution is explained step by step. First, the constraints that have been taken into consideration are discussed. Second, the repair steps representing the modification operators on the problem instantiation are described. Third, we take a look at the optimization method applied and how it connects to the problem instantiation. Finally, the whole application would be worthless without the revelation of new insights and results which are presented at the end of the chapter. Before, we proceed with the description of the problem some comments will be given on issues that will not be included or only slightly touched upon in this discussion. A side-effect of the shift scheduling application is that it should ultimately provide connectivity to a visual design tool for shift schedules that is already in use. The communication between these two systems follows a standard specification for shift schedules. The design of domain specific classes is prepared for the application of this standard. However, the discussion of this more technical detail should not be a part of this chapter. Moreover, the generation of an initial solution, which is an absolute necessity for the way ∗FLIP++ works on the problem, is an optimization problem of its own. Therefore I will only show in brief how such an optimization could take place, while the actual implementation relies on a manually produced initial problem instantiation. The initial instantiation does also satisfy a variety of hard constraints which are subsequently not taken into consideration in the formulation of ∗FLIP++ constraints. Furthermore, due to the implementation problems with DynaFLIP++ and OptiFLIP++, I deviated from the original intention of solely using the functionality of these modules, which resulted in some functions that per definitionem actually belong to DynaFLIP++ being moved to DomFLIP++ mainly out of performance reasons. This concentrates mainly on the interpreted constraint script file where some computations have been moved to DomFLIP++ in order to work around some slow interpreted functions of DynaFLIP++.

5.2 Problem description

Admittedly, the shift scheduling problem taken into consideration for this thesis has been cut short to some degree in order to arrive at a solution rather quickly. In particular, the number of represented constraints could be reduced by focusing on a problem instantiation with quite simple shift types. For example, the use of a night shift has been completely neglected. Consequently, all constraints referring to night shifts could be left out. Nevertheless, restricting ourselves to a narrower definition of the problem does not reveal limits of our system. Once a proper representation of a problem is found, enhancing the constraints of the problem does not cause any difficulty at all.

The objective of our problem is to find a shift schedule for twelve employees. These employees are aggregated in three groups with an even distribution, i.e., each group consists of four employees. The groups are further divided into subgroups of two employees each. Each of the subgroups is fully covering the requirements of operation, hence no interdependencies between the various subgroups have to be taken into consideration. Ruling out interdependencies is a further simplification that is rather rare in real world problems. Again, as mentioned above, such a simplification does not hurt the judgement of the basic functionality of $*$FLIP++ in connection with a shift scheduling problem. For ease of use, the groups are named *A*, *B*, and *C* with subgroups *A1*, *A2*, *B1*, *B2*, *C1*, and *C2*. The working hours are 38.5 hours on a weekly basis. Weekly working hours can vary over the length of the shift schedule, but on average the 38.5 hours are the limit. Shifts of 8–9 hours (variable) and 12 hours have breaks of 30 minutes, whereas shifts of 4 hours do not involve any breaks. The 30 minutes break of the variable shift is calculated as 20 minutes unpaid and 10 minutes paid work. The 30 minutes break of the 12 hours shift is considered to be wholly unpaid. There are three different types of shifts defined in Figure 5.1.

The most common shift type is the *day shift*, this can also be seen from the roster presented next. The length of this shift is also considered to be flexible, i.e., it can be anywhere between 8 and 9 hours for the whole shift schedule.

This provides some leeway concerning the working hours requirements.

name	length (in hours)	shortcut
day shift:	8–9	(TD)
day shift at weekends:	4	(TDWE)
shift substitution at weekends:	12	(SWWE)

Figure 5.1: Shift type definitions.

The roster is defined in Figure 5.2 and displays the requirements of operation. It can be easily detected that the shifts required for operation remain the same week by week with only one exception. On every third Saturday of the shift schedule a different requirement is given due to maintenance work. Because of this the cycle of the operation plan is set to three weeks.

	Mon	Tue	Wed	Thu	Fri	Sat	Sun
Week 1	5 SG TD	5 SG TD	5 SG TD	2 G TD	2 G TD	1 SG TDWE + 1 G SWWE	1 SG TDWE
Week 2	5 SG TD	5 SG TD	5 SG TD	2 G TD	2 G TD	1 SG TDWE + 1 G SWWE	1 SG TDWE
Week 3	5 SG TD	5 SG TD	5 SG TD	2 G TD	2 G TD	2 G SWWE	1 SG TDWE

SG...Subgroup, G...Group
TD...day shift, TDWE...day shift weekend, SWWE...shift substitution weekend

Figure 5.2: Operation plan.

So far, the various parameters of the problem have been given. Before we proceed with the description of the solution for this specific problem, some sort of precomputations has to be done. This is necessary due to the following reason: The optimization methods applied in the ∗FLIP++ environment are all dependent on an initial solution. It does not really matter whether the quality of the solution is good or bad, as several studies have already shown (e.g., in [5]). As the automatic generation of valid initial solutions has not been part of the objective of this thesis, the system relies on the provision of a preliminary problem instantiation (= initial solution) from outside. So it is up to the knowledge engineer to provide such a solution. This is where the precomputations come into play. They determine various parameters of the problem instantiation:

- length of the shift schedule cycle
- number of shifts allocated per employee/subgroup/group during the entire shift schedule cycle
- length of the various shift types

The first two parameters are needed to generate a valid initial solution. The length of the shift schedule cycle remains constant throughout the optimization. Including such a parameter as a variable in the scope of the optimization process would lead to an enormous search space. It determines the size of a problem instantiation. The length of the cycle is derived from the cycle of operation and the number of minimal units of operation. In our specific problem the cycle of operation is three weeks which can be verified in Figure 5.2. The minimal unit of operation is a subgroup, and there exist six of them. Hence it follows that the shift schedule cycle must have a length that is equal to 18 weeks, because this is the minimal multiple of 6 and 3 that allows to rotate the operation cycle for each subgroup. Once a shift schedule of this length is found, it can be applied in consecutive sequences and the working hours of all subgroups and the employees respectively are the same over the length of one cycle.

In order to generate an initial problem instantiation, it is necessary to define how many shifts of each shift type have to be allocated for a shift schedule of the above computed cycle length while still satisfying the 'hard' requirements of operating hours and average working hours per week. The computation proceeds as follows:

According to the definitions of the shifts the 12 hours shift corresponds to 11.5 hours of working time, the 4 hours shift corresponds to 4 hours working time, and the *day shift* is variable, 10 minutes of the break are counted as working time. The operation plan tells us what amount of shifts is required for an entire 18 week shift schedule and the subgroups, respectively. Figure 5.3 displays the shift requirements for our problem.

Actually, it is not necessary to break down the figures of required shifts on a per employee basis since the smallest unit of operation is a subgroup.

shift type	entire shift schedule	subgroup
TD	414	69
TDW	30	5
SWWE	48	8

Figure 5.3: Shift requirements.

Now that we have the number of shifts that has to be allocated for each shift type and operation unit over the length of the entire shift schedule cycle, we can determine the value for the variable *day shift* length.

Equation 5.1 simply equates the number of hours worked by adding up the products of shift length and number of shifts applied on the left side with the number working hours needed, 18 weeks times the average working hours per week (38.5). The variable x stands for the length of the variable *day shift*. After the unpaid break time of 20 minutes has been added, the result for the gross shift length is a time of approximately 8 hours 45 minutes.

$$69\times x+5\times 4+8\times 11.5 = 18\times 38.5 \Rightarrow x = 8.42+0.3\dot{3} = 8.75\dot{3} \approx 8\,h\,45\,m \quad (5.1)$$

The computation that has been done for the length of the *day shift* is interesting since the shift length is an important input for constraints on working hours.

In summary, the precomputations done here are in combination with the generation of an initial problem instantiation a nice example of a scheduling problem itself. The problem is to find an initial solution that satisfies the hard constraints of operating hours and average working hours per week. The requirement of operating hours is satisfied by proper modification operators (repair steps) that move each shift type only to well-defined positions of the shift schedule, e.g. a *day shift* is only possible from Monday to Friday with at most five subgroups allocated. The requirement of average working hours has been resolved by the above computations and is subsequently inherently satisfied by providing each shift type in restricted numbers. For each unit of operation there exists a pool that holds the numbers of shifts that are available for the unit. In our example this would require six identical pools each referring to a subgroup and its shift requirements (69 *day shifts*, 5 *day shifts at weekends*, and 8 *shift substitutions at weekends*). An optimizing algorithm could now start to allocate shift after shift from a pool of shifts to the shift schedule. A solution is reached when the pool of shifts empty, i.e. all shifts have been allocated to the shift schedule, and hence none of the requirements is violated.

5.3 Solution

First, I want to clarify that the solution of the above described problem is not solely focused on the actual, optimal problem instantiation. The notion is to be comprehended in a wider context. With solution, I mean as well the process of knowledge transformation that takes places when the problem is formulated in a computer readable format after an intensive consultation of a human domain expert, in this case using the various representation mechanisms provided modules of $*$FLIP++. This task is a crucial one, since most failures are made by misinterpretations or leaving out parts of domain knowledge. Experience shows that this activity is a more gradual one requiring permanently shifting back and forth between communication with the domain expert and the actual design phase; by the way, a practice that should be observed with any serious software development project. Actually, the representation results in constraints that are formulated in a constraint script file and several classes and data structures that hold the dynamic domain data (for a discussion of *dynamic domain data* and *static domain data* refer to Section 3.2.3). Both forms of representations will be shown in the following sections.

Coming back to the actual solution, repair steps and the method of optimization are the key aspects in the discussion of how to arrive at a 'better' solution. The term 'better' in this sense is a subjective one, it relies on the definition of the constraints. The constraints are more or less the guidelines that tell the optimization process through their evaluation scores what kind of problem instantiations is preferred. Hence, the entire notion of a solution is subjective; it depends on how constraints are defined by the human expert. If there are ill-defined constraints they may hinder the optimizing algorithm in making its path towards a 'better' solution. Also, two many requirements may lead to an 'over-constrained' problem that is intractable in terms of optimization.

5.3.1 Constraints

The constraints define certain requirements of the problem domain. Normally, constraints are of dynamic nature, i.e., their concrete instantiations depend on the instantiation of the problem. In our application this means that different shift schedules lead to different constraint instantiations of the same type of constraint. Hence it is necessary to define constraints in a language-like style that is based on the use of variables. Each evaluation run of DynaFLIP++ is based on variables whose values are fed by the problem instantiation of DomFLIP++. Initially, I started to compute all the dynamic variables of the constraints in the constraint script file where the constraints are also defined. Unfortunately, the constraint script interpreter did not prove to be very strong in performance. As a consequence, I moved all the computations that depend on the *dynamic domain data* into the *CShiftSchedulingDomain* class resulting in superior performance mainly due to the use of compiled *C++* methods. However, the constraint script file is still necessary because it uses the computed variables to initialize the corresponding fuzzy variables and fuzzy constraints, aggregates the constraints into hierarchy, and finally evaluates the constraints. In the following I will shortly explain some of the constraints that have been implemented.

Constraint WorkingHours

The average working hours per week are fixed for our problem due to the precomputations that restrict the allocation of shifts per subgroup and hence per employee. Still, it is of course possible that over the shift schedule cycle the weekly working hours deviate from the average working hours. A rule of thumb tells us that an employee should not work more than 50 hours per week. This can be easily formulated as a constraint. For each week and each subgroup of the schedule the amount of working hours in computed with a method of the *CShiftSchedulingDomain* class. Then, CSI (= constraint script interpreter) variables (actually a matrix for the entire shift schedule) are initialized before the evaluation takes place. There, a fuzzy variable is initialized with the value and subsequently used in a simple compare constraint. All the constraints of this type are then aggregated in the constraints hierarchy and finally evaluated. Figure 5.4 shows the

```
// definition of operator sets
...
// definition of parameter sets
...
// definition of fuzzy variables
...
// definition of constraint hierarchy
...

FOR i = 1 TO NumberOfGroups STEP 1
   FOR j = 1 TO LengthOfCycle STEP 1
      // initialize fuzzy variable with computed working hours
      SET workinghours = c_workinghours[i][j]
      // formulate constraint
      FUZZYCOMPARE c_workinghours IS workinghours < 50
      // evaluate constraint
      EVAL c_workinghours
      // put constraint into branch 'workinghours' of
      // constraint hierarchy
      AGGREGATE workinghours AND c_workinghours group[i].week[j]
   NEXT j
NEXT i
```

Figure 5.4: CSI section of constraint 'workinghours'.

corresponding CSI section.

In the CSI section, the rather lengthy definitions for the fuzzy variables, the parameter sets, the fuzzy operators, the compare table for fuzzy comparisons, and the constraint hierarchy are not shown. They are of course an important instrument when it comes to fine-tuning of constraints. A large variety of parameters allows to influence the way a constraint is evaluated.

Constraint of even distribution of working hours

This constraint tries to guide the evaluation towards smooth shift schedules, meaning that deviations between weekly working hours should be relatively small. It computes the difference in working hours between consecutive weeks and detects those pairs of weeks where the difference is rather high. This should lead to a more evenly distributed sequence of shifts and hence working hours.

Constraint to avoid shift sequences

Shift scheduling research has shown that shift sequences with six and more shifts on consecutive days should be avoided. After several consecutive shifts rest periods get more and more important. If night shifts are also part of the operation plan, than the acceptable length of shift sequences is reduced further. During a scan of the shift schedule blocks of sequences and their lengths are detected. The constraint will then lead to bad evaluations for shift sequences of six and more shifts.

Constraint for weekends

Weekends are a crucial area with most shift scheduling problems. Often, a human expert has to pay especially attention to this area of the shift schedule. In our application, two different constraints are concerned with the evaluation of weekends. First, one constraint simply checks the number of free weekends for one subgroup and leads to better evaluations the more free weekends there are. Second, another constraint tries to avoid the use of shift allocations on consecutive weekends for the same subgroup, since this is another rule of thumb a shift scheduling expert would take into consideration.

5.3.2 Class structure

CShiftSchedulingDomain

The class *CShiftSchedulingDomain* (see Figure 5.5 for a sketchy description) encompasses the functionality that is necessary to represent the problem described above. Furthermore, it provides the means for optimization. The class is derived from the class *CDomainInstance* and hence inherits all the methods and attributes from this class. *CDomainInstance* has been already discussed in Chapter 3. *CShiftSchedulingDomain* specializes *CDomainInstance* so that the particulars of the shift scheduling domain can be represented.

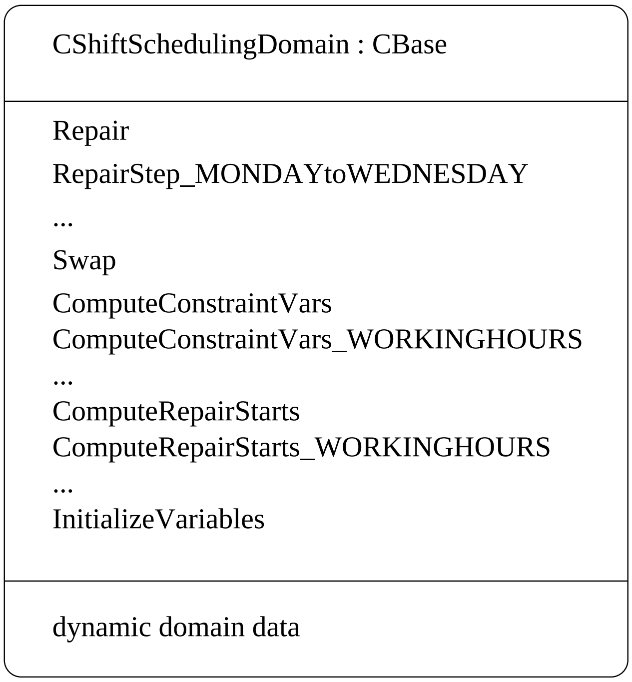


Figure 5.5: Class *CShiftSchedulingDomain*.

The class redefines the method *CShiftSchedulingDomain:Repair* that chooses, depending on the position in the schedule, a repair step. The repair steps are implemented in the methods *CShiftSchedulingDomain:RepairStepX*, where *X* stands for the number of the repair step. In our problem we have actually four different types of repair steps. They will be discussed in more detail in Section 5.3.4. Basically, all repair steps can be reduced to a simple swap operation on the schedule which is implemented in the private method *CShiftSchedulingDomain:Swap*.

The method *CShiftSchedulingDomain:ComputeConstraintVars* is responsible for the preparation of variables that are needed by the constraint script file. It computes for each constraint all the variables for the current instantiation of the problem. Due to the number of constraints, it is recommended to write a separate method for each Constraint. For example, the values of the variables of the constraint WORKINGHOURS are computed in *CShiftSchedulingDomain:ComputeConstraintVars_WORKINGHOURS*. These variables are then used in the method *CShiftSchedulingDomain:InitializeVariables*. This method transforms the variables into CSI (= constraint script interpreter) variables and initializes the interpreter properly before executing the constraint script for the evaluation of the problem instantiation.

The *CShiftSchedulingDomain:ComputeRepairStarts* method derives possible start positions for the repair based optimizing algorithms from the evaluations of the constraints. Since each constraint has its particular definition that leads to quite specific derivations of repair start recommendations, the method relies again on the services of private methods for each constraint. For example, the repair start recommendations for the constraint WORKINGHOURS are computed in the method *CShiftScheduling-Domain:ComputeRepairStarts_WORKINGHOURS*.

Furthermore, as with all classes there exist methods for input/output of the whole set of domain data which comprises *static domain data* and *dynamic domain data*; a concept already explained in Chapter 3.

Among the class attributes is the inherited *schedule*. A shift schedule can be easily represented as a matrix of shift allocations with the dimensions unit of operation (= subgroup in our example application) and day respectively. Therefore, a matrix of integers, which identify the allocated shift type, has been chosen for the implementation of the *CSchedule* class that determines the type of the *schedule*. In addition, all domain relevant attributes reside in the *CShiftSchedulingDomain* class; just to number a few of them, the length of the operation plan, the length of the shift schedule, amount of working hours per week, etc. Moreover, there are references to containers of classes that represent shifts, groups/subgroups, and the operation plan. These classes are briefly explained below.

CShift

The class *CShift* is a straightforward implementation of a shift type. It holds attributes like name and length of the shift (e.g., *day shift* of 8 hours). Besides, it contains information about how many shifts of this type are required for the entire shift schedule, which is determined in the already discussed precomputations of the problem. All shifts are aggregated in a sequence that is an attribute of *CShiftSchedulingDomain*.

CGroup

The units of operation, mainly groups and subgroups, are represented in the class *CGroup*. It stores attributes such as the group name (e.g., A1), the number of employees in the group, the level in the aggregation hierarchy (e.g., a group that contains subgroups). Similar to *CShift*, groups are aggregated in a sequence that is referred to by *CShiftSchedulingDomain*.

COperation

The class *COperation* provides the requirements of the roster, i.e., how many shifts of a certain shift type are required on a specific day (e.g., five subgroups with *day shifts* on Monday). An aggregation of such classes allows to represent the operation plan in a compact form. Such a sequence of *COperation* classes is again used by *CShiftSchedulingDomain*.

5.3.3 Data structure

The concept of most data structures that can be identified both in connection with the shift scheduling problem and DomFLIP++ in general lends itself for the application of sequence-like implementations. Deviating from the previous implementations of ∗FLIP++ modules, also different to first version of DomFLIP++, I waived the use of linear lists or similar structures. Subsequently, all data structures used throughout DomFLIP++ have been implemented by the use of sequence containers provided by STL (see Appendix B for more information about STL).

The class *CSchedule* that represents the shift schedule is represented as a matrix of integers, which is in STL terms a vector of vectors of integers. The integers identify the allocated shift type, the position in the matrix determines the concerned day and subgroup, respectively.

The repair starts, evaluation data, groups and shifts are all aggregated by the use of the vectors. The obvious advantage is that all these structures are treated consistently in the same way facilitating implementation tremendously. Furthermore, a plethora of algorithms is supplied by STL that makes the solution of many detail problems much more convenient allowing the programmer to concentrate on the real issues of the application.

5.3.4 Repair steps

Repair steps represent the modification operators of a problem. They allow to move from one valid instantiation of the problem to an other. Such modifications are the basis for every optimizing algorithm that tries to arrive at—in terms of evaluation—superior problem instantiations by modifying the actual problem instantiation.

Generally, it is always possible to identify certain types of repair steps for a problem. In our example, the definition of repair steps is almost entirely dictated by the plan of operation, i.e. the roster. With the use of the roster (see Figure 5.2) it is possible to define for each day what type of shift and how many of it must be allocated. Due to performance reasons, repair steps should be preferably kept simple since they are used quite heavily during the optimization process. Mostly, the modifications for a problem can be broken down to simple swap or move operations. This has also been shown for the application in the steel production domain in [12].

Repair steps represent modifications of the shift schedule. Without any guidance these modifications would take place randomly. To avoid that, the positions where constraint violations are identified are used as input for repair steps. This can be any position in the shift schedule. As we will see in the next paragraph, in our problem the position in the shift schedule will also determine which repair step is applied. In our application, repair steps take the position as an input and then try to find a possible modification while iterating through the shift schedule beginning from a start position. The indication of a start position allows some leeway in the application of the repair step. One is not restricted concerning the start of the search for a successful modification. The start position could be a random one, or the same as the position of the violation, or whatever seems to be appropriate for the problem at hand.

Taking a closer look at the roster of our shift scheduling problem shown in Figure 5.2, one can identify four main clusters where operations might be the same within that cluster. From Monday to Friday the operation is identical and there is only one shift type involved. Since Thursday and Friday do have a more complex operation plan, as they use different operation units, namely groups, we have to split this cluster into two: One modification operator will be defined for modifications based on Monday to Wednesday, another one will be defined for those based on Thursday to Friday. The third modification operator is focused on changes that involve the *day shift at weekends* on Saturday and Sunday. Finally, a repair step for the *shift substitutions at weekends* on Saturday is defined. In the following the four different types of repair steps will be explained.

Repair step 'Monday–Wednesday'

The operation plan shows that on the days from Monday to Wednesday there is only one shift type required, the *day shift*. If a violation of a constraint and its derived repair start position is within this range, there are two ways of applying a modification operator.

First, the position of the violation, which is characterized by the day and the subgroup (= smallest unit of operation in our problem) affected, can be swapped with another position of the range Monday to Wednesday. Actually, two swaps have to be made in order to preserve the requirements of the operation plan. The reason is straightforward: From Monday to Wednesday five subgroups are allocated with *day shifts* resulting in one subgroup without a shift. If the violation is on a position where a *day shift* is allocated, a swap is sought with a position in the same subgroup where no shift is allocated. Because the number of shifts allocated to subgroup is not affected by such a swap this hard constraint is preserved. Furthermore, by making a second opposite swap the hard constraint for the required shifts at one day is preserved, too. Both swaps involve an exchange of a free subgroup, i.e., one that has no shift allocated, with a *day shift* subgroup. The operation is illustrated in Figure 5.6.

Second, since the *day shift* can also be found on Thursday and Friday, it is also possible to seek for swaps destinations within this range. There is only one thing that has to be considered in addition to the above swap operation, on Thursday and Friday groups are required. Consequently, a swap operation must take care of this and preserve the group structure of

DAY	1	2	3	4	5	6	7	8	9	10
Subgroup A1	TD	TD	TD	TD				TD	TD	TD
Subgroup A2	TD	TD	TD	TD		TDWE			TD	TD
Subgroup B1	TD	TD	TD	TD	TD			TD	TD	
Subgroup B2				TD	TD		TDWE	TD		TD
Subgroup C1	TD	TD	TD		TD	SWWE		TD	TD	TD
Subgroup C2	TD	TD	TD		TD	SWWE		TD	TD	TD

Violation at (Subgroup A2, Day 1)

Swap 1: (Subgroup A2, Day 1) - (Subgroup A2, Day 8)

Swap 2: (Subgroup B2, Day 1) - (Subgroup B2, Day 8)

Figure 5.6: Repair step 1 for Monday to Wednesday.

the operation requirements on Thursday and Friday. This results in a more complex two-step modification described in Figure 5.7.

DAY	1	2	3	4	5	6	7	8	9	10
Subgroup A1	TD	TD	TD	TD				TD	TD	TD
Subgroup A2	TD	TD	TD	TD		TDWE			TD	TD
Subgroup B1	TD	TD	TD	TD	TD			TD	TD	
Subgroup B2				TD	TD		TDWE	TD		TD
Subgroup C1	TD	TD	TD		TD	SWWE		TD	TD	TD
Subgroup C2	TD	TD	TD		TD	SWWE		TD	TD	TD

Violation at (Subgroup A2, Day 1)

Swap 1: (Subgroup A2, Day 1) - (Subgroup A2, Day 5)

Swap 2: (Subgroup B2, Day 1) - (Subgroup B2, Day 5)

Swap 3: (Subgroup A1, Day 5) - (Subgroup A1, Day 10)

Swap 4: (Subgroup B1, Day 5) - (Subgroup B1, Day 10)

Figure 5.7: Repair step 2 for Monday to Wednesday.

Repair step 'Thursday/Friday'

The repair step of the Thursday to Friday range is similar to the one described above. Again, there are two variants of the modification operator.

First, a swap from the range Thursday to Friday to itself is made. Because the operation on these days works solely with groups, the swaps have to be effected on the group level to preserve this structure. Figure 5.8 illustrates the modification.

DAY	1	2	3	4	5	6	7	8	9	10
Subgroup A1	TD	TD	TD	TD				TD	TD	TD
Subgroup A2	TD	TD	TD	TD		TDWE			TD	TD
Subgroup B1	TD	TD	TD	TD	TD			TD	TD	
Subgroup B2				TD	TD		TDWE	TD		TD
Subgroup C1	TD	TD	TD		TD	SWWE		TD	TD	TD
Subgroup C2	TD	TD	TD		TD	SWWE		TD	TD	TD

Violation at (Subgroup A2, Day 4)

Swap 1: (Subgroup A1, Day 4) - (Subgroup A1, Day 5)

Swap 2: (Subgroup A2, Day 4) - (Subgroup A2, Day 5)

Swap 3: (Subgroup C1, Day 4) - (Subgroup C1, Day 5)

Swap 4: (Subgroup C2, Day 4) - (Subgroup C2, Day 5)

Figure 5.8: Repair step 1 for Thursday to Friday.

DAY	1	2	3	4	5	6	7	8	9	10
Subgroup A1	TD	TD	TD	TD				TD	TD	TD
Subgroup A2	TD	TD	TD	TD		TDWE			TD	TD
Subgroup B1	TD	TD	TD	TD	TD			TD	TD	
Subgroup B2				TD	TD		TDWE	TD		TD
Subgroup C1	TD	TD	TD		TD	SWWE		TD	TD	TD
Subgroup C2	TD	TD	TD		TD	SWWE		TD	TD	TD

Violation at (Subgroup B2, Day 4)

Swap 1: (Subgroup B1, Day 4) - (Subgroup B1, Day 10)

Swap 2: (Subgroup B2, Day 4) - (Subgroup B2, Day 9)

Swap 3: (Subgroup C1, Day 4) - (Subgroup C1, Day 10)

Swap 4: (Subgroup C2, Day 4) - (Subgroup C2, Day 9)

Figure 5.9: Repair step 2 for Thursday to Friday.

Second, similar to the second variant of the ‘Monday–Wednesday’ repair step, it is also possible to make a swap between Thursday or Friday and the range from Monday to Wednesday. Preserving the group structure requires a more complex modification that involves four swaps. The example in Figure 5.9 shows such a modification.

Repair step 'Saturday/Sunday'

This repair step is focused on the shift type *day shift weekend*. The operation plan allows this shift to be used on Sundays and with restrictions on Saturdays. The swap operation is defined rather straightforward. The shift is always applied in subgroups, so that no complex swap operation is required. Figure 5.10 shows an example of this type of repair step.

DAY	5	6	7	8	9	10	11	12	13	14
Subgroup A1				TD	TD	TD	TD		TDWE	
Subgroup A2		TDWE			TD	TD	TD			
Subgroup B1	TD			TD	TD		TD	TD	SWWE	
Subgroup B2	TD		TDWE	TD		TD	TD	TD	SWWE	
Subgroup C1	TD	SWWE		TD	TD	TD		TD		TDWE
Subgroup C2	TD	SWWE		TD	TD	TD		TD		

Violation at (Subgroup A2, Day 6)

Swap 1: (Subgroup A1, Day 6) - (Subgroup A1, Day 13)

Swap 2: (Subgroup A2, Day 6) - (Subgroup A2, Day 13)

Figure 5.10: Repair step for Saturday/Sunday *day shift weekend*.

Repair step 'Saturday'

Similar to the previous one, this repair step is also a straightforward implementation. It swaps shifts of the type *shift substitution weekend*. These shifts are always applied in groups. Therefore, a swap must always include groups thus preserving the requirements of the operation plan. Figure 5.11 illustrates this kind of repair step.

DAY	5	6	7	8	9	10	11	12	13	14
Subgroup A1				TD	TD	TD	TD		TDWE	
Subgroup A2		TDWE			TD	TD	TD			
Subgroup B1	TD			TD	TD		TD	TD	SWWE	
Subgroup B2	TD		TDWE	TD		TD	TD	TD	SWWE	
Subgroup C1	TD	SWWE		TD	TD	TD		TD		TDWE
Subgroup C2	TD	SWWE		TD	TD	TD		TD		

Violation at (Subgroup C1, Day 6)

Swap 1: (Subgroup B1, Day 6) - (Subgroup B1, Day 13)

Swap 2: (Subgroup B2, Day 6) - (Subgroup B2, Day 13)

Swap 3: (Subgroup C1, Day 6) - (Subgroup C1, Day 13)

Swap 4: (Subgroup C2, Day 6) - (Subgroup C2, Day 13)

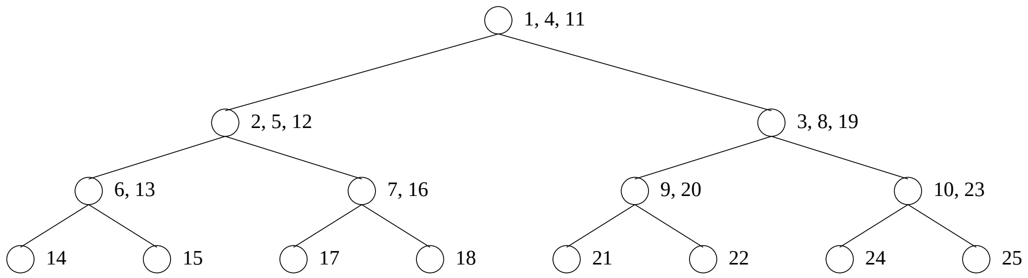
Figure 5.11: Repair step for Saturday *shift substituion weekend*.

5.3.5 Optimization

Although, OptiFLIP++ (see Section 2.4) does already provide two algorithms for optimization, I decided to take a new approach for the application of shift scheduling. First, the interface of OptiFLIP++ has not been defined in a sufficient generic way to be appropriate for the purposes of the shift scheduling application. The two algorithms of OptiFLIP, a tabu search repair based hill climbing algorithm and a genetic algorithm, have been designed tightly coupled with the steel production application of ∗FLIP++. This produced quite satisfying results within this environment, but it did not prove to be an easy task to adapt the structure of OptiFLIP++, in particular the interface, to the shift scheduling application. For example, the steel production domain with its job scheduling has always schedules that can be represented with linear sequences, whereas the shift schedules have a the format of a matrix; which did not correspond to the OptiFLIP++ interface. Hence, a new approach has been taken by the implementation of another optimizing algorithm. The key objective was to come up with a simple and quick solution that does not sacrifice too much of the performance that has already been pushed to the limits by the use of DynaFLIP++ and its interpreter.

The new algorithm that has been implemented is based on *iterative improvement techniques*. Such an algorithm tries to improve an initial preliminary schedule iteratively. The modifications, or repair steps, are the operations to move from one schedule to another. The problem of getting trapped in a local optimum, can be overcome in several ways.

Our approach, which is called *iterative deepening*, works as follows: In a first try, a search to depth 1 is made. This means that starting from the initial schedule one modification is executed. The execution of one modification does imply that all possible modifications of that type are tried on the violated position. If no improvement is found, a depth-first backtracking search to depth 2 is performed in a second attempt. This means that the modification operator is applied twice which also allows to accept a worse intermediate schedule in terms of evaluation as long as the schedule at the end of the sequential application of modification operators is better. If still no improvement is found, the depth is increased iteratively until either a successful repair step is achieved or a limit is reached causing the algorithm to abort. Figure 5.12 shows how the nodes of the search tree are visited by the algorithm.



1, 2, 3 ... depth 1
4, 5, 6, 7, 8, 9, 10 ... depth 2
11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25 ... depth 3

Figure 5.12: How the iterative deepening algorithm traverses the search tree.

A convenient feature of an iterative improvement technique is the anytime feature, i.e., the algorithm can be stopped at any time and always provides the best solution reached so far.

Of course, there is a great portion of leeway in how to define such an algorithm specifically. In this application, I added some additional features.

First, the algorithm starts with an initial schedule that has been evaluated with DynaFLIP++. The evaluation produces a list of evaluated constraints that is sorted by the severity of the violations. From each violated constraint we can derive possible candidates for modification, i.e., start positions, according to the type of the constraint. To increase the number of possible modifications tried in each depth level of the search, the algorithm tries to do all possible modifications by traversing the list of violated constraints and for each constraint applying modification operators on all start positions. For example, the algorithm starts with a depth level equal to 1, takes the most severely violated constraint, and starts with the first recommended possible start candidate. Then it applies the modification operator to this position until it finds an improvement or all possible operations with this specific start candidate have been tried. When the search for one start candidate is exhausted, the next one is tried, and so. If the list of start candidates has been traversed, the next violated constraint is selected. If all these tries fail, the depth level will be increased and the entire process starts all over again. The description makes clear that the possible steps tried by the algorithm are numerous. The success of the search is strongly coupled with some parameters like the number of violated constraints and possible modification start candidates that are taken into consideration. Figure 5.13 shows a sketch implementation of iterative deepening.

```
RepairDeep()
{  int Depth = 1, MaxDepth = 5;
   int Tries = 0;

   Evaluate();
   BestEvaluation = GetEvaluation();
   BestSchedule = GetSchedule();
   while ( Depth < MaxDepth )
   {  // RepairTry is a recursive functions that tries a
      // repair step by applying a modification operator
      // Depth times.
      bool Improved = RepairTry( Depth );
      if ( Improved )
      {  BestEvaluation = GetEvaluation();
         BestSchedule = GetSchedule();
         Depth = 1;
      }
      else
         Depth++;
      if ( BreakConditionFulfilled() ) break;
   }
};
```

Figure 5.13: A sketch of the iterative deepening optimizing algorithm.

5.4 Results

In the following I briefly summarize the results that have been attained with the problem so far. There is no doubt that there is still a lot of ground to cover as far as the fine-tuning of constraints is concerned. Nevertheless, the optimizations that can be done to this time are very promising and show the strength of the concept of the ∗FLIP++ system. Admittedly, the performance of the system has been pushed to its outmost limits by the implementation of the shift scheduling application and hence the focus of future developments should be on improvements of this crucial ingredient of the system.

The problems that have been encountered with the performance of the system during the optimization runs are mainly caused by the somewhat inefficient implementation of intensively used FLIP++ and DynaFLIP++ routines. Consequently, I took appropriate measures to simplify the problem. These measures predominantly concentrated on freeing the system from the burden of sophisticated and intensive computations. Randomizing various features of the optimizing algorithm in the way discussed in the previous section proved to be one of the more successful measures. These modifications resulted in an optimizing algorithm that behaves like a greedy hill-climbing optimization method, very similar to the well-known GSAT and similar iterative improvement optimization algorithms (turn to [16] for more details). Initial results are depicted in Figure 5.14. As one sees, results improve for approximately 1000 iterations and then stabilize around an overall satisfaction degree of 0.69. Several runs are depicted to show the variability of results obtained with the randomized search.

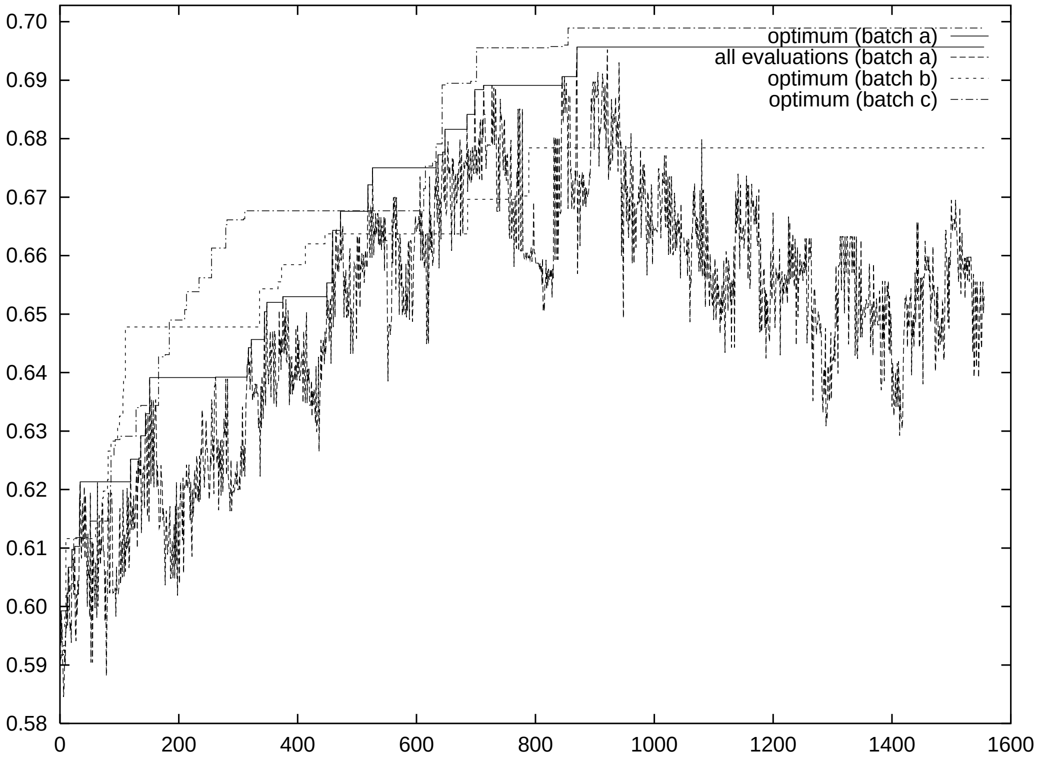


Figure 5.14: Optimization results.