Charles University in Prague
Faculty of Mathematics and Physics

Constraint-based Timetabling

Ph.D. Thesis

Prague, 2005 Tomáš Müller

Acknowledgements

Many thanks to Doc. RNDr. Roman Barták, PhD. for patient and systematic guidance in creation of this Ph.D. thesis.

Declaration

I declare that this thesis was composed by myself, and all presented results are my own, unless otherwise stated.

Tomáš Müller

1. Introduction

Constraint programming is a natural tool for describing as well as solving a lot of problems from various areas. Its major advantage is its capability of precise declarative description of a problem using relations between variables. It is based on a strong theoretical basis and it has wide practical applications in areas of evaluation, modelling, and optimisation.

Timetabling is one of the typical examples of constraint programming application. The task is to allocate activities in time and space respecting various constraints and to satisfy as nearly as possible a set of desirable objectives. A typical constraint is the request that activities which are using the same resource (e.g., a room, a machine, an operator, …) can not overlap in time or that a resource is of a certain capacity, restricting e.g. how many activities can use it at the same time. In addition, there are usually relations between activities and constraints restricting what resources an activity should or can use.

There are a lot of timetabling problems from various areas, for example, there is course, examination, transport, workforce, sport timetabling etc. In this thesis we will concentrate on course timetabling.

There are two major objectives of this Ph.D. thesis: We would like to find, describe and experimentally verify a constraint-based algorithm which is applicable to course timetabling problems as well as to other constraint satisfaction and optimisation problems. Moreover, with such an algorithm, we would like to tackle a real-life large scale timetabling problem. The whole Ph.D. work was motivated by this possibility to create an algorithm which is able to solve a given real-life problem and which can produce a solution fully acceptable by the users.

The thesis is organized as follows. In the following chapter, we give a brief overview of the constraint satisfaction problem and of various approaches to solving this problem. We also extend the traditional definition of the constraint satisfaction problem into a minimal perturbation problem that is more suited for dynamic problems. In such problems, changes in the problem definition are occurring after a solution to the initial formulation has been reached. The minimal perturbation problem incorporates these changes, along with the initial solution, as a new problem whose solution must be as close as possible to the initial solution.

Chapter three briefly describes various timetabling problems. It also defines the class timetabling problem at Purdue University. It is a real-life large scale problem that includes features of over-constrained as well as optimisation problems. The goal is to timetable more than 800 lectures to a limited number of lecture rooms (about 50) and to satisfy as many as possible individual course requests of almost 30,000 students.

In the fourth chapter, the iterative forward search algorithm is presented to solve both initial and minimal perturbation problems. This algorithm is close to local search methods; however, it maintains partial feasible assignments as opposed to the complete conflicting assignments characteristic of local search. Similar to local search, it processes local changes in the assignment. This allows us to generate a complete solution and to improve the quality of the assignment at the same time.

Chapter five contains several extensions of the iterative forward search algorithm. The most important, conflict-based statistics, is proposed to improve the quality of the final solution. Conflicts during the search are memorized and their potential repetition is minimized. In this chapter, we also present how this conflict-based statistics can be used within a general local search algorithm. Another presented extension allows the iterative forward search algorithm to dynamically maintain arc consistency during the search. Finally, we also present how to transform this algorithm into the dynamic backtracking algorithm.

In chapter six, various experiments with initial as well as minimal perturbation problems are presented. The comparison is made on a random binary constraint satisfaction problem, on a random placement problem and on the timetabling problem of Purdue University from chapter three. It surveys various settings of the algorithm, with and without its extensions. There is also a comparison of iterative forward search algorithm with some other local search algorithms and of the conflict-based statistics used within two basic local search algorithms. Comparison of solutions given by the described algorithm with a hand-made solution of class timetabling problem at Purdue University is also an important part of this chapter.

Finally, chapter seven concludes the thesis and chapter eight contains the bibliography. There are also four appendixes: Appendix A describes the implementation of the iterative forward search algorithm in Java. Appendix B contains an example how to solve random binary constraint satisfaction problems with the presented algorithm. Appendix C presents another timetabling problem we solved with the presented iterative forward search algorithm. Appendix D lists the content of the attached CD-ROM.

2. Overview

Many real-life industrial and engineering problems can be modelled as finite constraint satisfaction problems (CSP) [Tsa93]. A CSP consists of a set of variables associated with finite domains and a set of constraints restricting the values that the variables can simultaneously take. In a complete solution of a CSP, a value is assigned to every variable from the variable's domain, in such a way that every constraint is satisfied.

Most algorithms for solving CSPs search systematically through the possible assignments of values to variables. Such algorithms are guaranteed to find a solution, if one exists, or to prove that the problem has no solution. They start from an empty solution (no variable is assigned) that is extended towards a complete solution satisfying all the constraints in the problem. Backtracking occurs when a dead-end is reached. The biggest problem of such backtrack-based algorithms is that they typically make early mistakes in the search, i.e., a wrong early assignment can cause a whole subtree to be explored with no success. There are several ways of improving standard chronological backtracking. Look-back enhancements exploit information about the search which has already been performed, e.g., backmarking or backjumping [DF02]. Look-ahead enhancements exploit information about the remaining search space via filtering techniques (e.g., via maintaining arc consistency described in [BR97, BR01]) or variable and value ordering heuristics [MF00]. The last group of enhancements is trying to refine the search tree during the search process, e.g., dynamic backtracking [Gin93].

Local search algorithms [MF00] (e.g., min-conflict [MJP92] or tabu search [GH97]) perform an incomplete exploration of the search space by repairing an infeasible complete assignment. Unlike systematic search algorithms, local search algorithms move from one complete (but infeasible) assignment to another, typically in a non-deterministic manner, guided by heuristics. In general, local search algorithms are incomplete, they do not guarantee finding a complete solution satisfying all the constraints. However, these algorithms may be far more efficient (wrt. response time) than systematic ones in finding a solution. For optimisation problems, they can reach a far better quality in a given time frame.

There are several other approaches which try to combine local search methods together with backtracking based algorithms. For example, the decision repair algorithm presented in [JL02] repeatedly extends a set of assignments (called decisions) satisfying all the constraints, like in backtrack-based algorithms. It performs a local search to repair these assignments when a dead-end is reached (i.e., these decisions become inconsistent). After these decisions are repaired, the construction of the solution continues to the next dead-end. A similar approach is used in the algorithm presented in [Sch97] as well.

2.1. Constraint Satisfaction Problem

Definition 2.1 *(CSP).* A constraint satisfaction problem (CSP) is a triple $\Theta = (V,D,C)$, where

- $V = \{v_1,v_2,\dots,v_n\}$ is a finite set of variables,
- $D = \{Dv_1,Dv_2,\dots,Dv_n\}$ is a set of domains (i.e., Dv_i is a set of possible values for the variable v_i),
- $C = \{c_1,c_2,\dots,c_m\}$ is a finite set of constraints restricting the values that the variables can simultaneously take.

Definition 2.2 *(assignment).* Let Θ be a CSP, an assignment of the variables from V is $\eta \subseteq \{v/a | v\in V \& a\in Dv\}$ where $\forall$ v/a, $w/b \in \eta$ $v = w \Rightarrow a = b$. An element v/a of η means that variable v has assigned value *a*. An assignment is complete iff $|\eta| = |V|$ (i.e., all variables are assigned).

Definition 2.3 *(solution to CSP).* A solution to the constraint satisfaction problem Θ is a complete assignment σ of the variables from V that satisfies all the constraints.

For many constraint satisfaction problems it is hard or even impossible to find a solution in the above sense. For example, for over-constrained problems [FW92], there does not exist any complete assignment satisfying all the constraints. Therefore other definitions of problem solution like Partial Constraint Satisfaction were introduced [FW92]. In papers [BMR03, BMR04], we proposed a new view of the problem solution based on a new notion of maximal consistent assignment. This approach is strongly motivated by the university timetabling problem but we believe that it is generally applicable. The basic idea behind is to assign as many variables as possible while still keeping the rest of the problem "consistent". It means that the user may later relax some constraints in the problem (typically some of the constraints among the non-assigned variables that cause conflicts) so that after this change the assignment can be extended to other variables.

Formally, we define consistency of a problem Θ using the given consistency technique ζ as follows:

Definition 2.4 *(consistency technique).* Consistency (or filtering) technique ζ is a function that for the given CSP $\Theta = (V,D,C)$ returns a new set of domains $D' = \{D'v_1,D'v_2,\dots,D'v_n\}$, where $\forall i$ $D'v_i \subseteq Dv_i$, so that the property of the consistency technique ζ holds true for the CSP $\Theta' = (V,D',C)$. If the property cannot be achieved on the problem Θ, consistency technique ζ returns *fail*.[1]

[1] There exist some consistency techniques that do not fall into this scheme, for instance, path consistency [MH86] where inconsistent pairs of values are filtered. But, these techniques are not much used in practical applications.

Definition 2.5 *(consistency check).* Consistency check $\zeta(\Theta)$ is *true* if and only if the consistency technique ζ deduces no failure on the CSP Θ, otherwise, $\zeta(\Theta)$ is *false*.

For instance, arc consistency (AC) technique removes values from variables' domains that are inconsistent with constraints (see Definition 2.6).

Definition 2.6 *(arc consistency).* The pair of variables (v_i,v_j) is arc consistent if and only if for every value $x \in Dv_i$ of variable v_i which satisfies the constraints on v_i there is some value $y \in Dv_j$ of variable v_j such that the assignment $\sigma = \{v_i/x, v_j/y\}$ is permitted by the constraint between v_i and v_j.

Definition 2.7 *(arc consistent problem)* A given CSP Θ is arc consistent if and only if every pair of variables (v_i,v_j) is arc consistent.

We say that the constraint satisfaction problem is consistent if the consistency technique deduces no failure (e.g., for arc consistency, the failure is indicated by emptying some domain).

Definition 2.8 *(consistency).* A given CSP Θ is consistent respecting consistency technique ζ if and only if the consistency check $\zeta(\Theta)$ is *true*.

Let Θ be a CSP and σ be a (partial) assignment of variables, then we denote $\Theta\sigma$ application of the assignment σ to the problem Θ, i.e., the domains of the variables in σ are reduced to a singleton value defined by the assignment.

Definition 2.9 *(application of the assignment σ to the problem Θ).* $\Theta\sigma = (V,D',C)$ is application of the (partial) assignment σ to the problem $\Theta = (V,D,C)$ if and only if $D' = \{D'v_1,D'v_2,\ldots,D'v_n\}$ where

- $\forall i\ (\exists a \in Dv_i\ v_i/a \in \sigma) \Rightarrow D'v_i = \{a\}$
- $\forall i\ (\neg\exists a \in Dv_i\ v_i/a \in \sigma) \Rightarrow D'v_i = Dv_i$

Definition 2.10 *(consistent assignment).* A partial assignment σ is consistent respecting consistency technique ζ if and only if the consistency check $\zeta(\Theta\sigma)$ is *true*, where $\Theta\sigma$ is application of the assignment σ to the problem Θ.

Note that a complete consistent assignment is a solution of the problem. Note also that backtracking-based solving techniques typically extend a partial consistent assignment towards a complete (consistent) assignment.

As we already mentioned, for some problems there does not exist any complete consistent assignment; these problems are called over-constrained. In such a case, we propose to look for the maximal consistent assignment.

Definition 2.11 *(maximal consistent assignment).* A partial assignment σ is maximal consistent assignment for a given CSP Θ if there is no consistent assignment σ' with a larger number of assigned variables, i.e., $|\sigma'|>|\sigma|$.

We can also define a weaker version, so called locally maximal consistent assignment.

Definition 2.12 *(locally maximal consistent assignment).* Locally maximal consistent assignment σ is a consistent assignment that cannot be extended to another variable(s). This means that there is no $\sigma' \supset \sigma$ that is consistent.

Notice the difference between the above two notions. The maximal consistent assignment is defined using the cardinality of the assignment (the number of assigned variables) so it has a global meaning while the locally maximal consistent assignment is defined using a subset relation, i.e., it is not possible to assign an additional variable without getting inconsistency. It is pretty easy (fast) to extend any consistent assignment to a locally maximal consistent assignment. In fact, every branch of the search tree defines such a locally maximal consistent assignment. Apparently, the maximal consistent assignment is the largest (using cardinality) locally maximal consistent assignment.

Example (maximal consistent assignments):

Let V = {a,b,c,d,e} be a set of variables with domains D = {D_a={1,2}, D_b={1,2,3}, D_c={2,3}, D_d={2,3}, D_e={2,3}} and C = {a≠b, b≠c, c≠d, c≠e, d≠e} be a set of constraints. Assume that we use arc consistency as the technique for checking consistency of CSP Θ = (V,D,C). Then:

- σ = {a/1} is a locally maximal consistent assignment for Θ which is not a maximal consistent assignment ($|\sigma|=1$),
- γ = {a/2, b/1} is a maximal consistent assignment for Θ ($|\gamma|=2$).

If a constraint satisfaction problem has a solution then any maximal consistent assignment is the solution. Thus, looking for a maximal consistent assignment is a general way of solving CSPs because it covers both standard CSPs as well as over-constrained problems. Moreover, it is not necessary to know in advance whether the problem is over-constrained or not. Still, it may be hard to find a maximal consistent assignment for some problems. In such a case, we propose to return the largest locally maximal consistent assignment that can be found using given resources (e.g., time). This approach has a strong real-life motivation, for example in timetabling and scheduling problems [MB02, RM03] it means that the system allocates as many activities as possible in given time (and no more activity can be allocated without a change of the current allocation). Typically, the solving algorithms based on the above idea select some sub-space of the solution space. For this sub-space, they find a maximal consistent assignment which is a locally maximal consistent assignment in the original solution space. For example, the LAN search algorithm [VR02] restricts the number of assignments tried per variable.

2.2. Minimal Perturbation Problem

Most existing solvers are designed for static problems. These problems can be expressed, solved by appropriate means, and the solution applied without any change to the problem statement. Many real-life problems [Koc02, VJ03, SW00, Ian04], however, are subject to change. Additional input requirements produce a new problem derived from the original problem. The dynamics of such a problem may require changes during the solution process, or even after a solution is generated. In many real situations, it is necessary to alter the solution process so that the dynamic aspects of the problem definition are taken into account.

Problem changes may result from changes to environmental variables, such as broken machines, delayed flights, or other unexpected events. Users may also specify new properties based on the solution found so far. The goal is to find an improved solution for the user. Naturally, the problem solving process should continue as smoothly as possible after any change in the problem formulation. In particular, the solution of the altered problem should not differ significantly from the solution found for the original problem.

There are several reasons to keep a new solution as close as possible to the existing solution. If the solution has already been published, such as the assignment of gates to flights, frequent changes would confuse passengers. Moreover, changes to a published solution may necessitate other changes if initially satisfied wishes of users are violated. This may create an avalanche reaction.

Dynamic problems appear frequently in real-life planning and scheduling applications where the task is to “minimally reconfigure schedules in response to a changing environment” [SW00]. Dynamic changes in the context of timetabling problems have started to be studied at [EGJ03]. Issues of interactive timetabling which needs to handle dynamic aspects of the problem were discussed in [CDJD04, PMM04, MB02]. A survey of existing approaches to dynamic scheduling can be found in [Koc02]. In an annotated bibliography on dynamic constraint solving [VJ03], it is notable that only four papers were devoted to the problem of minimal changes.

The minimal perturbation problem was described formally in [SW00] and solved by a combination of linear and constraint programming as a 5-tuple $\Pi = (\Theta, \alpha, C_{add}, C_{del}, \delta)$ where:

- Θ is a CSP (i.e., a triple (V,D,C), where V is a set of variables, D are domains for V, and C is a set of constraints);
- α is a solution to Θ (i.e., a complete assignment satisfying the constraints from C)

- C_{add}, C_{del} are constraint removal and addition sets;
- δ is a function that measures the distance between two complete assignments (perturbation).

A complete assignment β is a solution to Π iff it is a solution to CSP (V, D, C^*), where $C^* = (C \setminus C_{del}) \cup C_{add}$), and $\delta(\alpha,\beta)$ is minimal.

Notice that the above formulation of MPP is for hard CSPs where all the constraints must be satisfied by a complete assignment of variables. Moreover, it allows addition and retraction of constraints only so the set of variables is not changing.

Our view of MPP differs from the above definition in several ways. First, we formulate MPP for soft CSPs, i.e., the best incomplete assignments are compared. Second, we allow more general changes in the problem formulation; in particular both the set of constraints and the set of variables (together with domains) can be changed. Last but not least, our definition of the function δ measuring distance between assignments is more concrete in comparing differences in the assignments.

2.2.1. A Formal Model

In papers [BMR03, BMR04], we presented a new formal model of the minimal perturbation problem that is applicable to over-constrained problems as well as to problems where finding a complete solution is hard. Recall that the idea of MPP is to define a solution of the altered problem in such a way that this solution is as close as possible to the (partial) solution of the original problem.

Definition 2.13 *(MPP).* A minimal perturbation problem (MPP) is quadruple $\Pi = (\Theta, \Theta', F, \alpha)$, where:

- Θ, Θ' are two CSPs called an *initial problem* and a *changed problem,*
- F is a mapping of the variables from Θ to Θ', and
- α is a (locally) maximal consistent assignment for Θ called *initial assignment*.

The function F defines how the problem Θ is changed in terms of variables. It is (almost) one-to-one mapping of the variables from Θ to the variables from Θ'. For some variables v from Θ, the function F might not be defined which means that the variable v is removed from the problem. However, if the function F is defined then it is unique (it is a one-to-one mapping).

Definition 2.14 *(mapping of variables between problems).* F is a function of the variables from $\Theta = (V,D,C)$ to $\Theta' = (V',D',C')$ so that

- domain of F: *dom*(F) $\subseteq$ V
- range of F: *rng*(F) $\subseteq$ V'
- $\forall v,u \in dom(F)\ v \neq u \Rightarrow F(v) \neq F(u)$.

Also, for some variables v' from Θ', the origin might not be defined (i.e., there is no variable v such that F(v) = v'), which means that the variable v' is added to the problem. Notice also that the constraints and domains can be changed arbitrarily when going from Θ to Θ'. We do not need to capture such changes using the mapping functions like F because we are concerned primarily about the variable assignments.

Definition 2.15 *(distance set).* Let σ be a (partial) assignment for $\Theta = (V,D,C)$ and γ be a (partial) assignment for $\Theta' = (V',D',C')$. Then we define $W_\Pi(\sigma,\gamma)$ as a set of variables v from Θ such that the assignment of v in σ is different from F(v) in γ, i.e.,

$$W_\Pi(\sigma,\gamma) = \{v \in V \mid v/h \in \sigma \ \&\ F(v)/h' \in \gamma \ \&\ h \neq h'\}.$$

We call $W_\Pi(\sigma,\gamma)$ a *distance set* for σ and γ in Π and the elements of the set are called *perturbations*.

Definition 2.16 *(solution to MPP).* A solution to the minimal perturbation problem $\Pi = (\Theta, \Theta', F, \alpha)$ is a (locally) maximal consistent assignment β for Θ' such that the size of the distance set $W_\Pi(\alpha,\beta)$ is minimal.

The idea behind the solution of MPP is apparent – the task is to find the best possible assignment of the variables for the new problem in such a way that it differs minimally from the existing variable assignment of the initial problem.

Let us summarize now the two criteria used when solving MPP: the first criterion is maximizing the number of assigned variables, the second criterion is minimizing the number of perturbations between the resultant solution and the previous (initial) solution. These criteria are combined lexicographically to get an objective function.

Example:

Let $\alpha=\{a/1,b/3\}$ be the initial solution of a CSP Θ with variables $V=\{a,b,c\}$ and Θ' be a new CSP with variables $V'=\{b,c,d\}$, domains $D' = \{D_b=\{1,3\}, D_c=\{1,2,3\}, D_d=\{2,3\}\}$, and constraints $C'=\{b\neq c, c\neq d, d\neq b\}$. Assume that there is a mapping $F:\{b\rightarrow b, c\rightarrow c\}$ of variables from Θ to Θ'. Then the problem Θ' has the following solutions (maximal consistent assignments):

- $\beta_1 = \{b/1,c/2,d/3\}$ ($W_\Pi(\alpha,\beta_1) = \{b\}$),
- $\beta_2 = \{b/1,c/3,d/2\}$ ($W_\Pi(\alpha,\beta_2) = \{b\}$),
- $\beta_3 = \{b/3,c/1,d/2\}$ ($W_\Pi(\alpha,\beta_3) = \{\}$),

but only the assignment β_3 is a solution of MPP $\Pi = (\Theta, \Theta', F, \alpha)$.

2.3. Optimisation Problems

In many real-life applications, we do not want to find any solution but a good solution. The quality of a solution is usually measured by some application dependent function called *objective function*. The goal is to find such solution that satisfies all the constraints and minimise or maximise the objective function respectively. Such problems are called Constraint Satisfaction Optimisation Problems (CSOP).

Definition 2.17 *(CSOP).* A constraint satisfaction optimisation problem (CSOP) is a quadruple $\Phi = (V,D,C, f)$ where (V,D,C) is a standard CSP and f is an objective function which maps every consistent (partial) assignment to a numeric value.

The task is to find such solution that is optimal regarding the objective function f, i.e., it minimises or maximises the objective function.

Definition 2.18 *(solution to CSOP).* A solution to the constraint satisfaction optimisation problem $\Phi = (V,D,C,f)$ is a (locally) maximal consistent assignment σ for (V,D,C) such that the objective function $f(\sigma)$ is minimal (or maximal).

In order to find the optimal solution, we potentially need to explore all the solutions of CSP and compare their values using the objective function. We usually do not need to find the very best solution, but some good enough is sufficient. So, for instance, we can look for a solution where the objective function is below a given threshold.

The objective function is sometimes expressed using so called *soft constraints*, which are very similar to standard constraints (called *hard constraints* in optimisation problems), but they do not need to be necessarily satisfied. We are looking for a solution where the number of violated soft constraints is minimal. Also, these soft constraints can be of different weights, objective function is then expressed for instance as a sum of weights of the violated soft constraints.

As for minimal perturbation problem, we define a function called *perturbation penalty*, which represents the cost of the changes in the solution, i.e., the cost of the variables assigned to different values than the initial values.

Definition 2.19 *(perturbation penalty).* Let $\Pi = (\Theta, \Theta', F, \alpha)$ be an MPP. Perturbation penalty g is a function that maps a (partial) assignment σ for Θ' to a numeric value.

In our work, we express this perturbation penalty as a sum of individual costs of every variable v_i that contains an initial value a_i in its domain, but it is assigned to a different value in the solution σ (i.e., $F(v_i)$ is assigned to a value different from a_i):

Definition 2.20 *(perturbation penalty via non-initial assignment cost).* Let $w(v,a,b)$ be a cost of the variable $F(v)$ being assigned to value b instead of a which is the initial assignment for variable v, i.e., $v/a \in \alpha$, $F(v)/b \in \sigma$. Perturbation penalty is then

$$g(\sigma) = \sum\{w(v,a,b) \mid v \in W_\Pi(\alpha,\sigma),\ v/a \in \alpha,\ F(v)/b \in \sigma\}$$

Formally, a minimal perturbation optimisation problem can be defined as follows:

Definition 2.21 *(MPOP).* Minimal perturbation optimisation problem is a 6-tuple $\Psi = (\Phi, \Phi', F, \alpha, g, w_g)$, where

- $\Phi=(V,D,C,f)$, $\Phi'=(V', D', C', f')$ are two CSOPs called an *initial optimisation problem* and a *changed optimisation problem,*
- F is a mapping of the variables from Φ to Φ',
- α is a (locally) maximal consistent assignment for Φ called *initial assignment,*
- *g* is the perturbation penalty function that measures differences between the initial assignment α and a consistent assignment of Φ'
- w_g is a number between 0 and 1

We use value w_g to compare importance of the objective function *f'* of the changed CSOP and the above discussed perturbation penalty:

Definition 2.22 *(objective function for MPOP).* Objective function *h* for an MPOP $\Psi = (\Phi, \Phi', F, \alpha, g, w_g)$ where $\Phi'=(V', D', C', f')$, is a function that maps every consistent (partial) assignment σ of a CSP (V',D',C') to a numeric value as follows:

$$h(\sigma) = (1-w_g)\, f'(\sigma) + w_g\, g(\sigma)$$

Solution of an MPOP is then a (locally) maximal consistent assignment of the changed problem that minimises the objective function.

Definition 2.23 *(solution to MPOP).* A solution to the minimal perturbation optimisation problem $\Psi = (\Phi, \Phi', F, \alpha, g, w_g)$ where $\Phi'=(V', D', C', f')$ is a (locally) maximal consistent assignment β for the CSP (V',D',C') such that the objective function $h(\beta)$ from Definition 2.22 is minimal.

Note that the MPOP is an extension of MPP described in the above chapter. We can, for instance, model MPP using MPOP if we ignore objective function *f'* (w_g is set to 1) and if $g(\sigma) = |W_\Pi(\alpha,\sigma)|$.

3. Timetabling

Sometimes, the words schedule, sequence and timetable are loosely used as if were synonymous. But, there can be certain distinctions between these terms observed in the literature [Wer85, Wren96, BKJW97, AM99, Bar00, Koc02].

A *timetable* shows when particular events are to take place. It does not necessarily imply an allocation of resources. Thus a published bus or train timetable shows when journeys are to be made on a particular route or routes. It does not tell us which vehicles or drivers are to be assigned to particular journeys. The allocation of vehicles and drivers is part of the scheduling process. Although timetabling is strictly the design of the pattern of journeys, this pattern may be devised as part of a process which bears in mind whether it is likely that an efficient schedule may be fitted to the resulting journey pattern.

In the rail domain, the term timetabling is often used to refer the construction of a path (with times) for a train through a system. A class timetable shows when particular events are to take place. In an infants' school where a single teacher is responsible for all the activities of a particular class, and where these activities all take place in the same room, a timetable is nothing more than a statement as to the times at which particular activities will take place. By contrast, a university examination timetable will normally include room assignments drawn up in the knowledge of group sizes and of special facilities needed. A university class timetable has also to take into account the availability of individual lecturers. The activities of drawing up examination and university class timetables may be considered as scheduling activities.

A *sequence* is simply an order in which activities are carried out. For example, the order in which jobs are processed through the machines of a factory, if jobs pass through each machine in the same order, is a sequence. Sequencing may take into account costs related to one particular job being followed by another (e.g., machine conversion costs). The problem of sequencing jobs in these circumstances is known as a flow shop problem.

A *schedule* will normally include all the special and temporal information necessary for a process to be carried out. This will include times at which activities are to take place, statements as to which resources will be assigned where, and work plans for individual personnel or machines.

The goal of scheduling in its broadest sense is to solve practical problems relating to the allocation, subject to constraints, of resources to objects being placed in space-time, using or developing whatever tools may be appropriate. The problems will often relate to the satisfaction of certain objectives.

A.Wren defines scheduling, timetabling, sequencing and rostering [Wren96] as follows:

Definition 3.1 *(scheduling).* Scheduling is the allocation, subject to constraints, of resources to objects being placed in space-time, in such a way as to minimise the total cost of some set of the resources used.

Common examples of scheduling are transport scheduling or delivery vehicle routing which seek to minimise the numbers of vehicles or drivers and within that minimum to minimise the total cost. Another example is job shop scheduling which may seek to minimise the number of time periods used, or some physical resource.

Definition 3.2 *(timetabling).* Timetabling is the allocation, subject to constraints, of given resources to objects being placed in space-time, in such a way as to satisfy as nearly as possible a set of desirable objectives.

Examples of timetabling are class and examination timetabling and some forms of personnel allocation (e.g., manning of tools booths subject to a given number of personnel).

Definition 3.3 *(sequencing).* Sequencing is the construction, subject to constraints, of an order in which activities are to be carried out or objects are to be placed in some representation of a solution.

Examples of sequencing are flow-shop scheduling and the travelling salesman problem.

Definition 3.4 *(rostering).* Rostering is the placing, subject to constraints, of resources into slots in a pattern. One may seek to minimise some objective, or simply to obtain a feasible allocation. Often the resources will rotate through a roster.

Some problems may fit to more than one of the above definitions, and the terms tend to be used rather loosely in the workplace and in the scheduling community.

In some of the above satisfying or to minimising was referred. It should be remarked that many of these problems which we are treating do not have a well-defined objective. We may sometimes justify the use of ameliorating or non-optimising methods partly because different players will have different views of the objective, but in reality such methods are often used simply because no optimising (or exact) method is practicable.

Timetabling has long been known to belong to the class of problems called NP-complete [CK96], i.e., no method of solving it in a reasonable (polynomial) amount of time is known.

3.1. Academic Timetabling Problems

There are three main classes of academic timetables [Sch99]:

- **School Timetabling:** The week scheduling for all the classes of an elementary or a high school, avoiding teacher meeting two classes in the same time, and vice versa;
- **Course Timetabling:** The week scheduling for all the lectures of a set of university courses, minimizing the overlaps of lectures of courses having common students;
- **Exam Timetabling:** The scheduling for the exams of a set of university courses, avoiding to overlap exams of courses having common students, and spreading the exams for the students as much as possible.

The school timetable describes when each class has a particular lesson and in which room it is to be held. The actual content of the timetable is largely driven by the curriculum, the number of hours of each subject taught per week is often set nationally. Each class consists of a set of pupils, who must be occupied from the time they arrive until the time they leave school, and a specific teacher being responsible for the class in any one period.

Teachers are usually allocated in advance of the timetabling process, so the problem is to match up meetings of teachers with classes to particular time slots so that each particular teacher meets every class he or she is required to. Obviously each class or teacher may not be involved in more than one meeting at a time. Often, it is required that each teacher has at least one morning or afternoon free per week. Many other similar constraints may exist.

The (university) course timetabling problem consists in scheduling a set of lectures for each course within a given number of rooms and time periods. The main difference with the (high) school problem is that university courses can have common students, whereas school classes are disjoint sets of students. If two classes have common students then they conflict, and they cannot or should not be scheduled at the same period. Moreover, school teachers always teach to more than one class, whereas in universities, a professor may teach only one course. In addition, in the university problem, availability of rooms (and their size and equipment) plays an important role, whereas in the high school problem they are often neglected because, in most cases, we can assume that each class has its own room. We will discuss course timetabling in more detail in the following section 3.2.

The examination timetabling problem requires the teaching of a given number of exams (usually one for each course) within a given amount of time. The examination timetabling is similar to the course timetabling, and it is difficult to make a clear distinction between the two problems. In fact, some specific problems can be formulated both as an examination timetabling problem and a course timetabling one.

Nevertheless, it is possible to state some broadly-accepted differences between the two problems. Examination timetabling has the following characteristics (different from course timetabling problem) [Sch99]:

- There is only one exam for each subject.
- The conflicts condition is generally strict. In fact, we can accept that a student is forced to skip a lecture due to overlapping, but not that a student skips an exam.
- There are different types of constraints, e.g., at most one exam per day for each student, and not too many consecutive exams for each student.
- The number of periods may vary, in contrast to course timetabling where it is fixed.
- There can be more than one exam per room.

3.2. Course Timetabling

In this thesis, we will concentrate on university course timetabling problems. These problems are subject to many constraints that are usually divided into two categories: “hard” and “soft” [BKJW97].

Hard constraints are rigidly enforced. Examples of such constraints are:

- No resource (students or staff) can be demanded to be in more than one place at any one time.
- For each time period there should be sufficient resources (e.g. rooms, invigilators, etc) available for all the events that have been scheduled for that time period.

Soft constraints are those that are desirable but not absolutely essential. In real-world situations it is, of course, usually impossible to satisfy all soft constraints. Examples of soft constraints are:

- Time assignment: a course may need to be scheduled in a particular time period.
- Time constraints between events: one course may need to be scheduled before/after the other.
- Spreading events out in time: students should not have lectures of the same course in consecutive periods or on the same day.
- Coherence: professors may prefer to have all their lectures in a number of days and to have a number of lecture-free days. These constraints conflict with the constraints on spreading events out in time.
- Resource assignment: professors may prefer to teach in a particular room or it may be the case that a particular exam must be scheduled in a certain room.

- Continuity: Any constraints whose main purpose is to ensure that certain features of student timetables are constant or predictable. For example, lectures for the same course should be scheduled in the same room, or at the same time of day.

Moreover, usual course timetabling consists of many different departments where each department offer a multitude of courses from which students are required to take some and then may choose a number of others. In most cases, each department is responsible for its own timetable and must try to take into account the timetables of other departments.

3.2.1. Basic Search Problem

There are various definitions of the course timetabling problems. In [Wer85, Sch99], course timetabling is defined as the following search problem:

Definition 3.5 *(course timetabling).* There are q courses $K_1, K_2, \ldots K_q$, and for each i, course K_i consists of k_i lectures. There are r *curricula* $S_1, S_2, \ldots S_r$, which are groups of courses that have common students. This means that courses in S_l must be scheduled all at different times. The number of periods is p, and l_k is the maximum number of lectures that can be scheduled at period k (i.e., the number of rooms available at period k). The formulation is the following: find y_{ik} ($\forall\ i = 1,\ldots q$; $\forall\ k = 1,\ldots p$) , so that

- $\forall\ i = 1,\ldots q\ \ \sum\{ y_{ik} \mid k = 1,\ldots p\} = k_i$
- $\forall\ k = 1,\ldots p\ \ \sum\{ y_{ik} \mid i = 1,\ldots q\} \leq l_k$
- $\forall\ k = 1,\ldots p\ \forall\ l = 1,\ldots r\ \ \sum\{ y_{ik} \mid i \in S_l\} \leq 1$
- $\forall\ i = 1,\ldots q\ \ \forall\ k = 1,\ldots p\ \ y_{ik} \in \{0,1\}$

where $y_{ik} = 1$ if a lecture of course K_i is scheduled at period k, and $y_{ik} = 0$ otherwise.

The first constraint imposes that each course is composed of the correct number of lectures. The second constraint enforces that at each time there are not more lectures than rooms. The third constraint prevents conflicting lectures to be scheduled at the same period.

Problem from Definition 3.5 can be shown to be NP-complete through a simple reduction from the graph colouring problem (see [Wer85]).

A formulation equivalent to Definition 3.5 is based on the *conflict matrix* instead of on the curricula. The conflict matrix $C_{q \times q}$ is a binary matrix such that $c_{ij} = 1$ if courses K_i and K_j have common students, and $c_{ij} = 0$ otherwise.

In [Wer85, Sch99], the course timetabling problem also includes the following objective function:

Definition 3.6 *(course timetabling objective function).*

- $f(y) = \sum\{d_{ik} y_{ik} \mid i = 1,\ldots q;\ k = 1,\ldots p\}$

where d_{ik} is the desiderability of having a lecture of course K_i at period k.

In [Tri92] the conflict matrix $C_{q \times q}$ is considered with integer values, such that c_{ij} represents the number of students taking both courses K_i and K_j. In this way c_{ij} represents also a measure of dissatisfaction in case a lecture of K_i and a lecture of K_j are scheduled at the same time. The objective is measured by the global dissatisfaction obtained as the sum of all dissatisfactions of the above type.

Preassignments and unavailabilities can be expressed by adding a set of constraints of the following form:

Definition 3.7 *(preassignments and unavailabilities).*

- $\forall\ i = 1,\dots q\ \ \forall\ k = 1,\dots p\ p_{ik} \le\ y_{ik} \le a_{ik}$
  where $p_{ik} = 0$ if there is no preassignment, and $p_{ik} = 1$ if a lecture of course K_i is scheduled at period k;
  $a_{ik} = 0$ if a lecture of course K_i cannot be scheduled at period k,
  $a_{ik} = 1$ if a lecture of course K_i can be scheduled at period k.

In [Wer85], unavailabilities are expressed as preassignments with *dummy* lectures.

Reduction to Graph Colouring

De Werra [Wer85] shows how to reduce a course timetabling problem (see Definition 3.5) to graph colouring: Associate to each lecture l_i of each course K_j a vertex m_{ij}; for each course K_j introduce a clique between vertices m_{ij} (for $i = 1,\dots q$). Introduce all edges between the clique for K_{j1} and the clique K_{j2} whenever K_{j1} and K_{j2} are conflicting.

In case of unavailabilities, introduce a set of p new vertices, each one corresponding to a period. The new vertices are all connected each other. This ensures that each one is assigned to a different colour. If a course cannot have lectures at a given period, then all the vertices corresponding to the lectures of the course are connected to a vertex corresponding to the given period. Conversely, if a lecture must take place at a given time, then the vertex corresponding to that class is connected to all period vertices but the one representing the given period.

3.3. Approaches to Automated Timetabling

Simple, problem-specific heuristic methods can produce good timetables, but the size and complexity of modern university timetabling problems has provoked a trend towards more general problem solving algorithms, or metaheuristics, such as simulated annealing, evolutionary algorithms, and tabu search. Problem-specific heuristics may be employed in the context of such an algorithm to reduce the number of possible solutions processed, or to locally optimise a solution. Constraint Logic Programming is also a popular approach.

3.3.1. Sequential Methods

These methods order events using domain heuristics and then assign the events sequentially into valid time periods so that no events in the period are in conflict with each other [Car86]. In sequential methods, timetabling problems are usually represented as graphs where events are represented as vertices, while conflicts between the events are represented by edges. For example, if some students have to attend two events there is an edge between the nodes which represent this conflict. The construction of a conflict-free timetable can therefore be modelled as a graph colouring problem. Each time period in the timetable corresponds to a colour in the graph colouring problem and the vertices of a graph are coloured in such a way so that no two adjacent vertices are coloured by the same colour.

3.3.2. Cluster Methods

In these methods the set of events is split into groups which satisfy hard constraints and then the groups are assigned to time periods to fulfil the soft constraints. An early paper to describe this approach was written by White and Chan [WC79]. Different optimisation techniques have been employed to solve the problem of assigning the groups of events into time periods. The main drawback of these approaches is that the clusters of events are formed and fixed at the beginning of the algorithm and that may result in a poor quality timetable.

3.3.3. Constraint Based Approaches

In these methods a timetabling problem is modelled as a set of variables (i.e., events) to which values (i.e., resources such as rooms and time periods) have to be assigned to satisfy a number of constraints [BPS99, Whi00, Wal94, Cra96, Sch99, AM99, Bar00, CDJD04]. Usually a number of rules is defined for assigning resources to events. When no rule is applicable to the current partial solution a backtracking is performed until a solution is found that satisfies all constraints.

3.3.4. Meta-heuristic Methods

A variety of meta-heuristic approaches such as simulated annealing, tabu search, genetic algorithms and hybrid approaches have been investigated for timetabling. Meta-heuristic methods begin with one or more initial solutions and employ search strategies that try to avoid local optima. All of these search algorithms can produce high quality solutions but often have a considerable computational cost.

3.4. Purdue Timetabling Problem

Our work is motivated by the class timetabling problem at Purdue University [RM03, MR04]. Here a timetable for large lecture classes is constructed by a central scheduling office in order to balance the requirements of many departments offering large classes that serve students from across the university. Smaller classes, usually focused on students in a single discipline, are timetabled by “schedule deputies” in the individual departments. Such a complex timetabling process, including subsequent student registration, takes a rather long time. Initial timetables are generated about half a year before the semester starts. The importance of creating a solver for a dynamic problem increases with the length of this time period and the need to incorporate various changes that arise.

As for Fall 2004 semester, this problem consists of about 830 classes (forming almost 1800 meetings) having a high density of interaction that must fit within 50 lecture rooms with capacities up to 474 students. Room availability is a major constraint for Purdue. Overall utilization of the time available in rooms exceeds 78%; moreover, it is around 94% for the four largest rooms. About 90,000 course requests by almost 30,000 students must also be considered. 8.4% of class pairs have at least one student enrolment in common.

The timetable maps classes (students, instructors) to meeting locations and times. A major objective in developing an automated system is to minimize the number of potential student course conflicts which occur during this process. This requirement substantially influences the automated timetable generation process since there are many specific course requirements in most programs of study offered by the University.

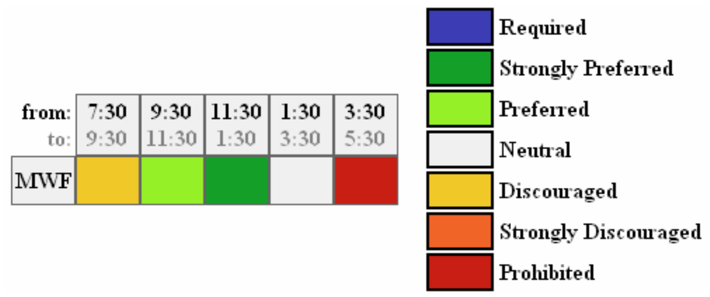
To minimize potential time conflicts, Purdue has historically subscribed to a set of standard meeting patterns. With few exceptions, 1 hour $\times$ 3 day per week classes meet on Monday, Wednesday, and Friday at the half hour (7:30, 8:30, 9:30, ...). 1.5 hour $\times$ 2 day per week classes meet on Tuesday and Thursday during set time blocks. 2 or 3 hours $\times$ 1day per week classes must also fit within specific blocks, etc. Generally, all meetings of a class should be taught in the same location. Such meeting patterns are of interest to the problem solution as they allow easier changes between classes having the same or similar meeting patterns.

Another aspect of the timetabling problem that must be considered is the need to perform student sectioning. Most of the classes in the large lecture problem (about 75%) correspond to single-section courses. Here we have exact information about all students who wish to attend a specific class. The remaining courses are divided into multiple sections. In this case, it is necessary to divide the students enrolled in each course into sections that will constitute the classes.

Currently, the timetable for Purdue University is constructed manually. We have proposed an automated timetabling system to solve the initial as well as the minimal perturbation problem in [MR04, MRB05]. This solution is based on the iterative forward search algorithm described in the following chapters.

Problem Representation

Due to the set of standardized time patterns and administrative rules enforced at the university, it is generally possible to represent all meetings of a class by a single variable. This tying together of meetings considerably simplifies the problem constraints. Most classes have all meetings taught in the same room, by the same instructor, at the same time of day. Only the day of week differs. Moreover, these days and times are mapped together with the help of meeting patterns, e.g., a 2 hours × 3 day per week class can be taught only on Monday, Wednesday, Friday, beginning at 5 possible times (see Figure 3.1).

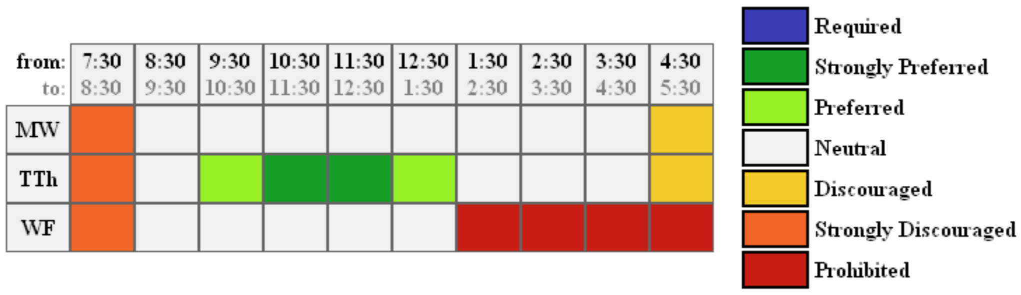


from: / to:	7:30 / 9:30	9:30 / 11:30	11:30 / 1:30	1:30 / 3:30	3:30 / 5:30
MWF	🟨	🟢	🟩	⬜	🟥

🟦	Required
🟩	Strongly Preferred
🟢	Preferred
⬜	Neutral
🟨	Discouraged
🟧	Strongly Discouraged
🟥	Prohibited

Fig. 3.1. An example of time preferences for 2 hours × 3 days per week class

Or, for instance, a 1 hour × 2 day per week class can be taught only on Monday+Wednesday, Wednesday+Friday or Monday+Friday, beginning at 10 possible times (see Figure 3.2).



from: / to:	7:30 / 8:30	8:30 / 9:30	9:30 / 10:30	10:30 / 11:30	11:30 / 12:30	12:30 / 1:30	1:30 / 2:30	2:30 / 3:30	3:30 / 4:30	4:30 / 5:30
MW	🟧	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	🟨
TTh	🟧	⬜	🟢	🟩	🟩	🟢	⬜	⬜	⬜	🟨
WF	🟧	⬜	⬜	⬜	⬜	⬜	🟥	🟥	🟥	🟥

🟦	Required
🟩	Strongly Preferred
🟢	Preferred
⬜	Neutral
🟨	Discouraged
🟧	Strongly Discouraged
🟥	Prohibited

Fig. 3.2. An example of time preferences for 1 hour × 2 days per week class

In addition, all valid placements of a course in the timetable have a one-to-one mapping with values in the variable's domain. This domain can be seen as a subset of the Cartesian product of the possible starting times, rooms, etc. for a class represented by these values. Therefore, each value encodes the selected time pattern (some alternatives may occur, e.g., 1.5 hour × 2 day per week may be an alternative to 1 hour × 3 day per week), selected days (e.g., a two meeting course can be taught in Monday+Wednesday, Tuesday+Thursday, Wednesday+Friday), and possible starting times. A value also encodes the instructor and selected meeting room. Each such placement also encodes its preferences (soft constraints), combined from the preference for time, room, building and the room's available equipment. Only placements with valid times and rooms are present in a class's domain. For example, when a computer (classroom equipment) is required, only placements in a room containing a computer are present. Also, only rooms large enough to accommodate all the enrolled students can be present in valid class placements. Similarly, if a time slice is prohibited, no placement containing this time slice is in the class's domain.

The variable and value encodings described above leave us with only two types of hard constraints to be implemented: resource constraints (expressing that only one course can be taught by an instructor or in a particular room at the same time), and group constraints (expressing relations between several classes, e.g., that two sections of the same lecture can not be taught at the same time, or that some classes have to be taught one immediately after another).

There are three types of soft constraints in this problem. First, there are soft requirements on possible times, buildings, rooms, and classroom equipment (e.g., a computer or a projector). These preferences are expressed as integers:

- -2 … strongly preferred
- -1 … preferred
- 0 … neutral (no preference)
- 1 … discouraged
- 2 … strongly discouraged

As mentioned above, each value, besides encoding a class's placement (time, room, instructor), also contains information about the preference for the given time and room. Room preference is a combination of preferences on the choice of building, room, and classroom equipment. The second group of soft constraints is formed by student requirements. Each student can enrol in several classes, so the aim is to minimize the total number of student conflicts among these classes. Such conflicts occur if the student cannot attend two classes to which he or she has enrolled because these classes have overlapping times. Finally, there are some group constraints (additional relations between two or more classes). These may either be hard (required or prohibited), or soft (preferred), similar to the time and room preferences (from -2 to 2).

Additional Constraints

Except the constraints described above, there are several additional constraints which came up during our work on this lecture timetabling problem. These constraints were defined in order to make the automatically computed timetable solution acceptable for users from Purdue University.

First of all, if there are two classes placed one after another so that there is no time slot in between (also called back-to-back classes), distances between buildings need to be considered. The general feeling is that different rooms in the same building are always reasonable, moving to the building next door is to be discouraged, a couple of buildings away strongly discouraged, and any longer distance prohibited.

Each building has its location defined as a pair of coordinates [x,y]. The distance between two buildings is estimated by Euclides distance in a two dimensional space, i.e.,

$$(\Delta x^2 + \Delta y^2)^{½}$$

where Δx and Δy are differences between x and y coordinates of the buildings. As for instructors, two back-to-back classes are infeasible to teach when such difference is more than 200 meters (hard constraint). The other options (soft constraints) are:

- if the distance is zero (same building), then no penalty,
- if the distance is above zero, but not more than 50 meters, then the placement is discouraged,
- if the distance is between 50 and 200 meters, the placement is strongly discouraged

Our concern for distance between back-to-back classes for students is different. Here it is simply a question of whether it is feasible for students to get from one class to another during the 10-minute passing period. At present, the distance between buildings not more than 670 meters is considered as an acceptable travel distance. For the distance above 670 meters, the classes are considered as too far. If there is a student attending both classes, it means a student conflict (same as when these classes are overlapping in time).

Next, since the automatic solver tries to maximize the overall accomplishment of soft time and room constraints (preferences), the resultant timetable might be unacceptable for some departments. The problem is that some departments define their time and room preferences more strictly than others. The departments which have not defined time and room preferences usually have most of their classes taught in early morning or late evening hours. Therefore, we introduced the departmental time and room preferences balancing mechanism. The solver is trying to fulfil the time and room preferences as well as to balance the used times between individual departments. This means that each department should use each time unit (half-hour, e.g., Monday 7:30 – 8:00) in a similar portion to the other time units used by the department.

At first, for each department and time unit, there is a number stating how many times each time unit can be used (i.e., how many placements of all classes from the department can be placed over the time unit). For instance, if there are two 1 hour × 2 days per week classes, the time unit Wednesday 8:00 – 8:30 can be used four times, i.e., each of these classes can be placed either on Monday-Wednesday or Wednesday-Friday from 8:00 till 9:00. Than, an *average fill factor* is computed for each department and time unit. It is a ratio between the computed number of placements using the time unit and the total number of placements of all classes from the department (it is sixty for the above example with two classes, each class can be placed in thirty different times if all possible times are allowed). So, this factor states the overall usage of a time unit for a department. The reason for computing such number is the fact that some times are used much more than others (e.g., if the department has all the classes in *n* hours hour × 3 days per week time pattern, only Monday, Wednesday and Friday are used, see Figure 3.1). The *initial allowance*, which states how many times each time unit can be used by a department is computed from this *maximal fill factor*: it is the *maximal fill factor* increased by the given percentage (20% is used in our tests) and rounded upwards to the first integer number. The *overall department balancing penalty* of a solution is the sum of overruns of this *initial allowance* over all time units and departments. The intention is to keep this number as low as possible.

Finally, since all of the classes are at least two time slots long (60 minutes), an empty time slot of a room which is surrounded by classes on both sides (i.e., the room is not used for 30 minutes between two consecutive classes) is considered useless – no other class can use it. The number of such useless half-hours should be minimized. Also the situation when a room is occupied by a class which is using less than ⅔ of its seats is discouraged. Both these soft constraints are considered much less important than all the constraints described above.

4. Iterative Forward Search Algorithm

The iterative forward search (IFS) algorithm that we propose here is based on ideas of local search methods [MF00]. However, in contrast to classical local search techniques, it operates over feasible, though not necessarily complete solutions. In such a solution, some variables can be left unassigned. Still all hard constraints on assigned variables must be satisfied. Similarly to backtracking based algorithms, this means that there are no violations of hard constraints.

Working with feasible incomplete assignments has several advantages compared to the complete infeasible assignments that usually occur in local search techniques. For example, when the solver is not able to find a solution (i.e., a complete feasible assignment), a largest feasible partial assignment (using cardinality) can be returned. Especially in interactive timetabling applications, such assignments are much easier to visualize, even during the search, since no hard constraints are violated. For instance, two lectures never use a single resource (e.g., a classroom) at the same time. Moreover, because of the iterative character of the search, the algorithm can easily start, stop, or continue from any feasible assignment, either complete or incomplete.

In this section, we present the iterative forward search algorithm which is the backbone part of this thesis as well as the timetabling software made for the Purdue University. The framework based on this IFS algorithm (written in Java) is described in more detail in appendixes A and B. It is well extendable and it can be used for solving lecture timetabling problems as well as for other constraint-based problems. Some of the general, problem independent extensions of this algorithm are described in the following chapter 5. In order to present the general purpose of this algorithm, it is described here for solving general finite constraint satisfaction and optimisation problems.

4.1. Related Works

Local search algorithms [MF00] (e.g., min-conflict [MJP92] or tabu search [GH97]) perform an incomplete exploration of the search space by repairing an infeasible complete assignment. Local search algorithms move from one complete (but infeasible) assignment to another, typically in a non-deterministic manner, guided by heuristics. In general, local search algorithms are incomplete, they do not guarantee finding a complete solution satisfying all the constraints. But, unlike systematic search algorithms, they do not suffer from the early mistake problem: as soon as a decision is suspected to lead to a dead-end, it can be undone, without having anything to lose. Also, these algorithms may be far more efficient (wrt. response time) than systematic ones in finding a solution. For optimisation problems, they can reach a far better quality in a given time frame.

4.1.1. Local Search Approaches

The term *local search* or *neighbour search* expresses the idea that these algorithms modify an inconsistent assignment locally to move to a better assignment. During each iteration step, only assignments from the *neighbourhood* of the current assignment are considered and one of them is picked. There are many ways how to define neighbourhood of an assignment. Usually, a value of one variable is changed.

There are two basic local search algorithm schemes, *hill-climbing* and *min-conflict*. Both of them usually start from a randomly (or heuristically) selected assignment and they repeatedly perform local steps to their neighbourhood till a solution is found or the time limit exceeded. But, they differ in the way how the neighbour assignments are selected.

Hill-climbing Algorithm

Hill-climbing [MF00] always selects the best assignment out of all the neighbours. This means the assignment which minimizes the number of violated constraints. In case of optimisation problems, it picks up the neighbour assignment which minimizes the objective function (e.g., the number of violated soft constraints) among the assignments with the minimal number of violated hard constraints. When there is no better assignment than the current one, the search is stuck in a local optimum. The *hill-climbing* algorithm usually restarts the search from another initial (e.g., randomly selected) assignment. The name of the algorithm, *hill-climbing*, is derived from its original principle when a maximum was searched by climbing – increasing the evaluation value.

Min-conflict Algorithm

On the other hand, *min-conflict* [MJP92] algorithm chooses the best assignment only from a subset of the neighbour assignments. Usually, it randomly selects any conflicting variable, i.e., a variable that is involved in an unsatisfied constraint, and then picks a value which minimizes the number of violated constraints. If no such value exists, it picks randomly one value that does not increase the number of violated constraints (the current value of the variable is picked only if all the other values increase the number of violated constraints). Note, that the pure *min-conflicts* algorithm is not able to leave a local minimum. In addition, if the algorithm achieves a strict local minimum, it does not perform any move at all and, consequently, it does not terminate.

Min-conflict Random Walk Algorithm

Because the pure min-conflict algorithm cannot go beyond a local minimum, some noise strategies were introduced in it. Among them, the *random-*

walk strategy has become one of the most popular ones. For a given conflicting variable, the random-walk strategy picks randomly a value with probability p_{rw}, and applies the min-conflict heuristic with probability 1- p_{rw}. Note that the same strategy can be used in *hill-climbing* as well, i.e., with the probability p_{rw} a random neighbour assignment is selected.

Tabu Search Algorithm

Tabu search [GH97] is another method to avoid cycling and getting trapped in a local minimum. It is based on the notion of *tabu list*, which is a special short term memory (usually containing pairs <variable, value>) that maintains a selective history, composed of previously encountered configurations or more generally pertinent attributes of such configurations. A simple *tabu-search* strategy consists in preventing configurations of the tabu list from being recognised for the next l_{ts} iterations (l_{ts}, called tabu tenure, is the size of tabu list). Such a strategy prevents the search from being trapped in short term cycling and allows the search process to go beyond local optima. Tabu restrictions may be overridden under certain conditions, called *aspiration criteria*. Aspiration criteria define rules that govern whether next configuration is considered as a possible move even when it is tabu. One widely used aspiration criterion consists of removing a tabu classification from a move when the move leads to a solution better than that obtained so far.

4.1.2. Hybrid Approaches

The idea of mixing traditional systematic search approaches with local search is not new. Those hybrid approaches have led to good results on large scale problems. Three categories of hybrid approaches can be found in the literature [PG96, Sch97, RR98, JL02, Pre04]:

- performing local search before or after a systematic search;
- performing a systematic search improved with local search at some point of the search: at each leaf of the tree (i.e., over a complete assignment) but also at nodes in the search tree (i.e., on partial assignments);
- performing an overall local search, and using systematic search either to select a candidate neighbour or to prune the search space

Decision Repair Algorithm

For instance, the *decision repair* algorithm (see Fig. 4.1.), presented in [JL02] falls into the third category above. It repeatedly extends a set of assignments (called decisions) satisfying all the constraints, like in systematic search algorithms. It performs a local search to repair these assignments when a dead-end is reached (i.e., these decisions become inconsistent). After these decisions are repaired, the construction of the solution continues to the next dead-end.

```
procedure decision-repair(V,D,C)
                      //a CSP problem is the parameter
  $C_D$ = any initial set of decisions;
          //decisions are constraints as well, i.e., Variable=value
  while conditions of failure not satisfied do
    C' = C ∪ $C_D$;
    switch obviousInferences(Φ(V,D,C'))
      case no solution:
        k = conflict explaining the failure;
        $C_D$ = neighbour($C_D$, k);
      case solution:
        return C';
      default:
        $C_D$ = extend($C_D$);
    end switch
  end while
  return failure;
end procedure
```

Fig. 4.1. The decision-repair algorithm

The decision-repair algorithm starts with a partial solution which is a result of a set of decisions. It first applies a filtering technique Φ. When no inconsistency is detected, the algorithm adds a decision that extends the current partial solution, and the search continues. When a dead-end is reached, we know that there is an incompatibility between the decisions made so far. The algorithm tries to *repair* that set of decisions. A *conflict* is identified (the smaller the conflict, the better), and the conflict is used to choose a judicious neighbour of the current set of decisions. For example, a judicious neighbour may be obtained by performing a local change on the current set of decisions: negate one of the decisions that occur in the conflict. The function *obviousInference* is able to examine a set of constraints in order to decide whether to stop the computation or not.

Constrained Local Search Algorithm

Another approach is used in the *constrained local search* algorithm presented in [Pre00, Pre04]. The algorithm is constructed by randomizing the backtracking component of a systematic search algorithm; that is, allowing backtracking to occur on arbitrary chosen variables. It has an integer parameter called the noise level stating on how many variables the algorithm will backtrack (selected by procedure *backtrackVariables*). See Figure 4.2 for the algorithm.

The constrained local search algorithm iteratively extends a partial feasible assignment (via assigning a selected value to a selected variable, only values consistent with the existing assignments are considered) until a complete assignment is found or a dead end is reached. When the dead end is reached, which means that an unassigned variable with no value consistent with the existing assignment is selected, a given number of variables is unassigned (stated by noise level parameter ε, selected either randomly or heuristically) and unpropagated if a filtering algorithm is used. The algorithm then continues extending the partial assignment again.

```
procedure cls(V,D,C,ε) // ε is the noise level
  σ = {};              //current assignment
  while σ is not complete do
    assigned = {A∈V | A assigned in σ}
    unassigned = V – assigned;
    A = selectVariable(unassigned);
    values = {a∈D_A; σ ∪ {A/a} is consistent};
    if (values is empty) then
      for all v in backtrackVariables(assigned, ε) do
        unassign v in σ and unpropagate;
      end for
    else
      a = selectValue(D_A);
      σ = σ ∪ {A/a};
    end if
  end while
  return σ;
end procedure
```

Fig. 4.2. The constrained local-search algorithm

Constructive Backtracking-free Algorithm

A similar approach, combining backtracking-free algorithm and local search is presented in [Sch97]. The algorithm iteratively extends a feasible partial assignment until a dead-end is reached. At this point, it performs a local search phase which makes local changes on the current partial assignment. Thereafter, the construction continues up to the next dead-end. The whole procedure stops either when a complete assignment is reached (positive answer) or when a predetermined number of local search phases have been accomplished (negative answer). See Figure 4.3 for the algorithm.

This approach differs from previous ones in at least two aspects: At first, it revises the partial assignment by making use of a full run of local search, instead of a fixed number of changes. Next, the local changes are selected with the additional objective of improving the possibility of the partial assignment to be completed. That is, local search is driven not only by the feasibility (and optimality) of the current partial assignment, but also by so called *look-ahead factor*. Furthermore, in order to make local search effective, the respective weights given to the three different components of the cost function that guides local search (feasibility, optimality, and look-ahead) are dynamically changed.

```
procedure cbf(V,D,C)
  σ = {};            //current assignment
  while σ is not complete do
    assigned = {A∈V | A assigned in σ}
    unassigned = V – assigned;
    A = selectVariable(unassigned);
    values = {a∈D_A; σ ∪ {A/a} is consistent};
    if (values is empty) then
      if last trial then return failure;
      β = σ;
      repeat
        move = selectMove(σ);
        makeMove(move, σ);
        if improves(σ, β) then β = σ;
      until last iteration or lower bound reached;
      σ = β;
    else
      a = selectValue(values);
      σ = σ ∪ {A/a}
    end if
  end while
  return σ;
end procedure
```

Fig. 4.3. The backtracking-free algorithm combined with LS

The procedure *improves* relies on a score function that assesses the quality of each assignment. Such function counts the number of constraint violations, thus measuring the distance to feasibility. For optimisation problems, it also takes into account the objective function of the problem. The function also includes a look-ahead factor (which estimates the likelihood of the remaining sub-problem to be solvable). Furthermore, since it is computed on partial assignments, only constraints regarding the instantiated variables are taken into account. For the same reason, the objective function is not computed exactly, but it is generally estimated using a lower bound (in a similar way as branch-and-bound procedures).

Unlike the above approaches, our algorithm operates more like the local search method – it does not execute a local search after a dead-end is reached but it applies the exact same local steps during search. In each iteration step a partial feasible assignment can be extended by an arbitrary assignment of a value to a variable and the consistency is enforced by a problem-independent consistency technique which can unassign some of the already assigned variables in order to make the partial assignment consistent with the selected assignment. Moreover, this makes the algorithm easy to implement and also to extend with various selection heuristics and other techniques like for instance filtering algorithms.

4.2. Iterative Forward Search Algorithm

Iterative forward search works in iterations (see Figure 4.4. for algorithm). During each step, a variable A is initially selected. Typically an unassigned variable is chosen like in backtracking-based search. An assigned variable may be selected when all variables are assigned but the solution found so far is not good enough (for example, when there are still many violations of soft constraints). Once a variable A is selected, a value *a* from its domain D_A is chosen for assignment. Even if the best value is selected (whatever "best" means), its assignment to the selected variable may cause some hard conflicts with already assigned variables. Such conflicting assignments are removed from the solution and become unassigned. Finally, the selected value is assigned to the selected variable.

```
procedure ifs(V,D,C,α) // an initial assignment α is the parameter
   σ = α;           // current assignment
   β = α;           // best assignmen
   while canContinue(σ) do          // CSP problem Φ=(V,D,C) is
      A = selectVariable(σ);        // a global parameter
      a = selectValue(σ, A);        // for all used functions
      η = conflicts(σ, A, a); //conflicting assignments
      σ = (σ - η) ∪ {A/a};    //next assignment
      if better(σ, β) then β = σ;
   end while
   return β;
end procedure
```

Fig. 4.4. The iterative forward search algorithm

The algorithm attempts to move from one (partial) feasible solution σ to another via repetitive assignment of a selected value *a* to a selected variable A. During this search, the feasibility of all hard constraints in each iteration step is enforced by unassigning the conflicting assignments η (computed by function *conflicts*). The search is terminated when the requested solution is found or when there is a timeout expressed, for example, as a maximal number of iterations or available time being reached. The best solution found is then returned.

The above algorithm schema is parameterized by several functions, namely

- the termination condition (function *canContinue*),
- the solution comparator (function *better*),
- the variable selection (function *selectVariable*) and
- the value selection (function *selectValue*).

Formalization

Constraint satisfaction can be defined as follows: A constraint *c* of variables $a_1,a_2,..a_n$ is satisfied with an assignment σ, if the assignment σ contains all variables $a_1,a_2,..a_n$ and the constraint *c* is satisfied with this assignment σ or the assignment σ can be extended to an assignment γ which contains all variables $a_1,a_2,..a_n$ such that the constraint *c* is satisfied with this assignment γ.

Definition 4.1 *(restriction of an assignment to a constraint).* Let $\Theta = (V,D,C)$ be a CSP, restriction of an assignment σ to a constraint $c \in C$ is

$$\sigma{\downarrow}c = \{v/a \mid v/a \in \sigma \ \&\ v \in dom(c)\}$$

where *dom*(c) represents a set of variables on which the constraint c is defined.

Definition 4.2 *(constraint satisfaction).* Constraint $c \in C$ is satisfied with an assignment σ of a CSP $\Theta = (V,D,C)$ if and only if

- $|\sigma{\downarrow}c|=|dom(c)|$ and $c(\sigma{\downarrow}c)$ holds true or
- $\exists\, \gamma \supseteq \sigma$ assignment of Θ where $|\gamma{\downarrow}c|=|dom(c)|$ and $c(\gamma{\downarrow}c)$

During the search, after each iteration step, we have an assignment σ of a subset of all variables (as described in Chapter 2.1). This assignment is feasible, which means that every hard constraint is satisfied with this assignment σ:

Definition 4.3 *(feasible assignment).* Assignment σ of a CSP $\Theta = (V,D,C)$ is feasible if and only if all constraints are satisfied with the assignment σ, i.e., $\forall c \in C$

- $|\sigma{\downarrow}c|=|dom(c)|$ and $c(\sigma{\downarrow}c)$ holds true or
- $\exists\, \gamma \supseteq \sigma$ assignment of Θ where $|\gamma{\downarrow}c|=|dom(c)|$ and $c(\gamma{\downarrow}c)$

This means that after each iteration step we have an assignment which is consistent wrt. a consistency technique that only enforces satisfaction of all hard constraints as defined above. Similarly, consistency of the assignment respecting any arbitrary consistency technique ζ (see Definition 2.4) can be enforced during the search. This means that after every iteration step, we have an assignment σ which fulfils the consistency check ζ(Θσ), where ζ is the consistency technique and Θ is the solved constraint satisfaction problem.

The task of the function *conflicts* is to enforce such consistency. It returns a subset of the current assignment $\eta \subseteq \sigma$, such that the new assignment $(\sigma - \eta) \cup \{A/a\}$ is consistent respecting the used consistency technique ζ (A is the variable and *a* is the value selected in the current iteration step).

Definition 4.4 *(property of function conflicts).* Let $\Theta = (V,D,C)$ be a CSP, σ be a consistent (partial) assignment of Θ, $A \in V$ be a selected variable and $a \in D_A$ be a selected value. Function *conflicts* returns $\eta \subseteq \sigma$ such that the assignment $\gamma = (\sigma - \eta) \cup \{A/a\}$ is a consistent (partial) assignment of Θ respecting some given consistency technique ζ.

In the above Definition 4.4, we assume that in the domain D_A of variable A are only values that are themselves consistent with all constraints respecting the consistency technique ζ. This means that for every value $a \in D_A$, consistency check $\zeta(\Theta\{A/a\})$ is true. As for the consistency based on constraint satisfaction (see Definitions 4.2 and 4.3), this means that for every value *a* from D_A, all constraints are satisfied with the assignment $\{A/a\}$. (i.e., $\forall c \in C$ $c(\{A/a\})$ is *true*). Otherwise, if such an inconsistent value *a* is selected, the resultant assignment γ can never be consistent since it contains the assignment $\{A/a\}$ that is not consistent (e.g., $\exists c \in C$, $c(\{A/a\})$ is *false*). These inconsistent values can be permanently filtered from the domains at the beginning of the search with no harm (e.g., by calling of the consistency technique on the problem with an empty assignment), since there cannot be a (partial) consistent assignment containing such values.

Obviously, we are looking for a strict subset of the current assignment σ (concerning cardinality) which satisfies the property from Definition 4.4. A minimal subset is the best, but it could be expensive to compute. Our current implementation does not try to find such a minimal set of “conflicting” variables. Instead, it tries to compute a *good* one quickly.

There is also a correspondence between these sets of “conflicting” variables and *nogood* sets in backtracking based algorithms [JDB00]. A nogood set is a subset of the current (partial) assignment that cannot be satisfied (i.e., no feasible solution contains this set). Potentially, a variable, different from A, from each such nogood set that can be computed from the assignment $\sigma \cup \{A/a\}$, needs to be selected into our set of conflicts.

4.2.1. Termination Condition

The termination condition determines when the algorithm should finish. For example, the solver should terminate when the maximal number of iterations or some other given timeout value is reached. Moreover, it can stop the search process when the current assignment is good enough, e.g., all variables are assigned and/or some other solution parameters are in the required ranges. For example, the solver can stop when all variables are assigned and less than 10% of the soft constraints are violated. Termination of the process by the user can also be a part of the termination condition.

4.2.2. Solution Comparator

The solution comparator compares two assignments: the current assignment and the best assignment found. This comparison can be based on several criteria. For example, it can lexicographically order assignments according to the number of unassigned variables (a smaller number is better) and the number of violated soft constraints.

4.2.3. Variable Selection

As mentioned above, the presented algorithm requires a function that selects a variable to be (re)assigned during the current iteration step. This function is equivalent to a variable selection criterion in constraint programming. There are several guidelines for selecting a variable [Dech03]. In local search, the variable participating in the largest number of violations is usually selected first. In backtracking-based algorithms, the first-fail principle is often used, i.e., a variable whose instantiation is most complicated is selected first. This could be the variable involved in the largest set of constraints or the variable with the smallest domain, etc.

We can split the variable selection criterion into two cases. If some variables remain unassigned, the “worst” variable among them is selected, i.e., first-fail principle is applied. This may be, for example, the variable with the smallest domain or with the highest number of hard and/or soft constraints.

The second case occurs when all variables are assigned. Because the algorithm does not need to stop when a complete feasible assignment is found, the variable selection criterion for such case has to be considered as well. Here all variables are assigned but the assignment is not good enough, e.g., in the sense of violated soft constraints. We choose a variable whose change of a value can introduce the best improvement of the assignment. It may, for example, be a variable whose value violates the highest number of soft constraints.

It is possible for the assignment to become incomplete again after such an iteration because a value which is not consistent with all hard constraints can be selected in the value selection criterion. This can be also taken into account in the variable selection heuristics.

4.2.4. Value Selection

After a variable is selected, we need to find a value to be assigned to the variable. This problem is usually called “value selection” in constraint programming [Dech03]. Typically, the most useful advice is to select the best-fit value. So, we are looking for a value which is the most preferred for the variable and which causes the least trouble as well. This means that we need to find a value with the minimal potential for future conflicts with other variables. For example, a value which violates the smallest number of soft constraints can be selected among those with the smallest number of hard conflicts.

To avoid cycling, it is possible to randomize the value selection procedure. For example, it is possible to select the N best values for the variable and choose one of them randomly. Or, it is possible to select a set of values so that the heuristic evaluation for the worst value in this group is maximally p percent higher than the heuristic evaluation of the best value (where smaller value means better evaluation). Again, the value is selected randomly from this group. This second rule inhibits randomness if there is a single very good value.

4.2.5. Conflicting Assignments

Like in other traditional constraint satisfaction frameworks, the input problem consists of variables, values and constraints. Each constraint is defined over a subset of the problem variables and it prohibits some combinations of values which these variables can simultaneously take. In many CSPs, all constraints are binary (or the problem is transformed into an equivalent problem with only binary constraints before the search is started) since most of the consistency and filtering techniques are designed only for binary constraints. In such a case, the function *conflicts* is rather simple and it returns an unambiguous subset of the given assignment. It enumerates all the constraints which contain the selected variable and which are not consistent with the selected value. It returns all the variables of such constraints, different from the selected variable.

Definition 4.5 *(function conflicts for binary CSP).* Let $\Theta = (V,D,C)$ be a binary CSP, σ be a feasible (partial) assignment of Θ, $A \in V$ and $a \in D_A$. Function *conflicts* returns $\eta \subseteq \sigma$ such that: $\eta = \{B/b \mid B/b \in \sigma \;\&\; ((B=A \;\&\; b \neq a) \vee \exists c \in C\, (\neg c(\sigma \cup \{A/a\}) \;\&\; \{B/b\} \in \sigma{\downarrow}c))\}$

All assignments from σ that are involved in any constraint not satisfied (with the new assignment $\sigma \cup \{A/a\}$) are returned. Also, if the selected variable A is already assigned (so a different value has been selected for A), it has to be unassigned first (previous assignment of the variable A is to be returned).

Lemma 4.1 Let $\Theta = (V,D,C)$ be a binary CSP (i.e., $\forall c \in C\ |dom(c)|=2$) that is arc consistent, σ be a feasible (partial) assignment of Θ, $A \in V$ and $a \in D_A$. Function *conflicts* from Definition 4.5 returns a set of conflicting assignments $\eta \subseteq \sigma$ that is minimal (concerning cardinality of η) and $\gamma = (\sigma - \eta) \cup \{A/a\}$ is a feasible (partial) assignment of Θ.

Proof of feasibility: Let $\gamma = (\sigma - \eta) \cup \{A/a\}$ be infeasible, then there is a constraint $c \in C$ that is not satisfied with γ. This means that c is also not satisfied with $\sigma \cup \{A/a\}$. Since c was satisfied with σ (σ is a feasible assignment), $A \in dom(c)$. Because c is binary, there is $B \in dom(c)$, $A \neq B$. There are the following possibilities:

- $\exists b \in D_b$, $\{B/b\} \in \sigma$: but since $(\neg c(\{A/a, B/b\}) \;\&\; \{B/b\} \in \sigma{\downarrow}c)$, $\{B/b\}$ is in the set η, c is satisfied with γ which is a contradiction.
- $\forall b \in D_b$, $\{B/b\} \notin \sigma$: but since $\exists b' \in D_b\ c(\{A/a, B/b'\})$ (since c is satisfied in σ and the problem is arc consistent), c is satisfied in $\sigma \cup \{A/a\}$ and therefore also in $(\sigma - \eta) \cup \{A/a\}$.

Proof of minimality: $\forall \{B/b\} \in \eta$, either

- $B = A$ and $b \neq a$: In this case $\{B/b\}$ has to be in η since A cannot be assigned twice (γ has to be an assignment),

- or $\exists c \in C\ \neg c(\{A/a, B/b\})$ since all constrains are binary.

In both cases, $\{B/b\}$ has to by contained in any conflicting set otherwise γ is either not assignment (A is assigned twice) or not feasible (there exists a constraint that is not satisfied). So, every set of conflicting variables has to contain the set η computed according to the Definition 4.5.□

On the other hand, most of real problems have plenty of multi-variable constraints, like, for instance, a resource constraint in timetabling. Such resource constraint enforces the rule that none of the events which are using the given resource can be overlapping in time (if the resource has capacity one) or that the amount of the resource used at a time does not exceed its capacity. It is not very useful to replace such a resource constraint by a set of binary constraints (e.g., prohibiting two overlapping placements in time of two particular events using the same resource), since this approach usually ends up with thousands of constraints. Also, there is usually a much more effective consistency and/or filtering technique working with the original constraint (for instance, "cumulative" constraint [RM03] is usually used for modelling resource constraints in CLP).

Using multi-variable constraints, the set of conflicts returned by function *conflicts* can differ according to its implementation (but it must satisfy the property from Definition 4.4). For instance, we can have a constraint A+B=C where A and C are already assigned to A=3 and C=5 (i.e., $\sigma = \{A/3, C/5\}$). Then if the assignment B=3 is selected, either A or C or both A and C can be unassigned to make the problem {A/3, B/3, C/5} consistent with the constraint A+B=C. Intuitively, there should be a minimal number of variables unassigned in each iteration step (we are trying to increase the number of the assigned variables during the search). Also, for many constraints, it is possible to find inconsistencies even when not all variables of the constraint are already assigned. For instance, if there are two lectures using the same room at the same time, we know that one of them needs to be unassigned even when there are unassigned lectures which will also need to be placed in that room.

In our IFS solver (described in more detail in Appendix A), each hard constraint needs to implement the procedure *computeConflicts* which returns all the already assigned variables that are incompatible with the selected assignment. This procedure is called for all constraints which contain the selected variable in an ordered manner (in the order the constraints were inserted into the system). Furthermore, this order can be changed during the search. Moreover, the computed set of conflicts is passed to this *computeConflicts* procedure as a parameter, so the constraint can "see" which conflicts are already selected for unassignment by previously processed constraints and can use this information to unassign a smaller number of variables. For example, if there is a constraint A+B=C not satisfied with the new assignment C=*c*, the *computeConflicts* procedure can pick the variable (either A or B) that is already selected by some of the prior constraints. This way, we are not computing the very minimal set of conflicts, however, we allow for computing this set in an efficient way. It can be also tuned for a particular problem by changing the order of constraints. For our timetabling problems, we do not need to take this order into account, since most of the constraints are resource constraints of singleton capacity where the conflicting assignments are exactly given (i.e., the already assigned events which are using the same resource as the selected one and which overlap in time with it). The function *conflicts* is outlined in Figure 4.5.

```
procedure conflicts(σ, A, a)
  η ={};    //resultant set of conflicting variables
  for each constraint c∈C so that A∈dom(c) in a given order do
    η = η ∪ computeConflicts(c, σ, A, a, η);
  end for
  return η;
end procedure
```

Fig. 4.5. The conflicts procedure

Also note that each constraint can keep its notion about the assigned variables. For instance, the resource constraint of a particular room can memorize a look-up table stating what lecture is assigned in what time slot(s), so for the computation of the conflicting lectures it only looks through the appropriate fields of this table. The implementation is based on listening to “variable assigned” and “variable unassigned” events. Also note that this default consistency technique is defined on a problem level and it can be changed by a more dedicated one, implemented for a particular problem. For more details, see Appendix A.

Let *computeConflicts*(c, σ, A, a) be a function similar to the procedure *computeConflicts* from Figure 4.5 that returns assignments conflicting with the new assignment $\sigma \cup \{A/a\}$ for the given constraint c. The only difference is that the function in Figure 4.5 gets a list of inconsistent assignments computed for $\sigma \cup \{A/a\}$ on the previously visited constraints. The following Lemma 4.2 shows that using such incremental computation of the set of conflicting variables still produces the feasible (partial) assignment at the end of each iteration.

Lemma 4.2 Let Θ = (V,D,C) be a CSP, σ be a feasible (partial) assignment of Θ, $A \in V$ be a selected variable and $a \in D_A$ be a selected value. If $\forall c \in C$ η_c = *computeConflicts*(c, σ, A, a) is a subset of σ so that $c((\sigma - \eta_c) \cup \{A/a\})$, the new assignment $\gamma = (\sigma - \eta) \cup \{A/a\}$ where $\eta = \{ B/b \mid \exists c \in C\ A \in dom(c)\ \&\ B/b \in \eta_c \}$ is a feasible (partial) assignment of Θ.

Proof: The proof of Lemma 4.2 immediately follows from the definition of a feasible assignment (see Definition 4.3) stating that an assignment is feasible if all constraints are satisfied with the assignment (i.e., $\forall c \in C\ c(\gamma)$) and from the fact that $\gamma = (\sigma - \eta) \cup \{A/a\} \subseteq (\sigma - \eta_c) \cup \{A/a\}$ since $\eta \supseteq \eta_c$. Because σ is feasible assignment, all constraints that does not contain variable A are satisfied in the new assignment γ as well.□

4.3. IFS for Minimal Perturbation Problem

Let us first describe the meaning of perturbation in our approach. The changed problem differs from the initial problem by input perturbations. An input perturbation means that a variable must have different values in the initial and changed problem because of some input changes (e.g., a course must be scheduled at a different time in the changed problem).

The solution to the minimal perturbation problem (MPP) [SW00, BMR03, BMR04] can be evaluated by the number of additional perturbations. They are given by subtraction of the final number of perturbations and the number of input perturbations. An alternative approach is to consider variables in the initial and in the new problem which were assigned differently [RRH02, BMR03, BMR04]. As before, we need to minimize the number of such differently assigned variables.

Despite the local search nature of the algorithm, there are some adjustments needed to be able to effectively solve the MPP. The purpose of these adjustments is to minimize the number of additional perturbations. The easiest way to do this is to adopt variable and value selection heuristics which prefer the previous assignments (but not all the time, to avoid cycling).

For example, value selection heuristics can be adopted to select the initial value (if it exists) randomly with a probability P (it can be rather high, e.g., between 50-90%). If the initial value is not selected, the original value selection can be executed. Also, if there is the initial value in the set of best-fit values (e.g., among values with the minimal number of hard and soft conflicts), the initial value can be preferred as well. Otherwise, a value can be selected randomly from the constructed set of best-fit values. A disadvantage of such selection is that the probability P has to be selected carefully: if it is too small, the search can easily move away and the number of additional perturbations will grow during the search. If it is too high, the search will stick too much with the initial solution and, if there is no solution with a small amount of additional perturbations it will be hard to find a feasible solution.

Another approach is to limit the number of additional perturbations during the search. Furthermore, like in branch and bound, such a limit can be decreased when a feasible solution with the given number of perturbations is found. For example, if the number of additional perturbations is equal to or greater than the limit, the initial value has to be selected. Otherwise, if the number of additional perturbations is below the limit, the original value selection strategy is followed. The number of additional perturbations can also include variables that are not assigned yet whose initial values cause a hard conflict with the current assignments.

The above approaches can also be combined together, which can help to divide their influence during the search.

Variable selection heuristics can also be adopted to find a solution with a small number of perturbations. For example, when all variables are assigned, a variable that has an initial value but such value is not assigned to it should be selected, e.g., randomly among all variables that have not the initial value assigned, and that participate in the highest number of violated soft constraints.

4.4. Summary

In this section, we have presented the iterative forward search algorithm which is a mixture of systematic search and the local search approach. In the following section, we will discuss some of its extensions and later on we will present some computational results of this algorithm used on a CSP problem as well as on the large lecture timetabling problem on Purdue University.

The very first version of this algorithm was presented in [MB01] and in the diploma thesis [Mul01] as an ad-hoc solution for the iterative lecture timetabling problem. Its application on lecture timetabling problem on Mathematics and Physics Faculty of Charles University was presented in [MB02]. The applicability of this algorithm on the n-queens problem was presented in [Mul02].

The iterative forward search algorithm in the form as it is presented in this chapter for solving of a general CSP with various extensions (conflict-based heuristics, maintenance of arc consistency, its extension towards dynamic backtracking) was presented in [MBR04]. Its application to the minimal perturbation problem of Purdue University timetabling was presented in [MR04, MRB05]. Also, we used this algorithm in comparison with a branch&bound algorithm designed for solving MPP problem in [BMR04] where it was better than the proposed branch&bound algorithm.

5. IFS Extensions

In this section, we present some of the problem-independent extensions of the iterative forward search algorithm. We present the conflict-based statistics which is a learning technique that helps the solver to escape a local optimum. We also describe how the presented conflict-based statistics can be used inside a traditional local-search algorithm. Next, we will present how to dynamically maintain arc consistency during the search using explanations. We also present how to change an incomplete iterative forward search algorithm into a complete systematic search algorithm called dynamic backtracking.

5.1. Conflict-based Statistics

Value ordering heuristics play an important role in solving various problems. They allow choosing suitable values for particular variables to compute a complete and/or optimal solution. Problem-specific heuristics are usually applied because problem-independent heuristics are computationally expensive. Here we propose an efficient problem-independent approach to value selection whose aim is to recognize good and poor values.

We have applied this so called conflict-based statistics (CBS) in our iterative forward search algorithm [MBR04, MR04, MRB05]. This combination helped us to solve a large-scale timetabling problem at Purdue University. Here we describe a general scheme for the conflict-based statistics and apply it to local search and iterative forward search methods.

5.1.1. Related Works

Methods similar to CBS were successfully applied in earlier works [DF02, JL02]. In the weighting-conflict heuristics presented in [JL02], a weight is associated with each decision (assignment of a value to a variable). It characterizes the number of times that the decision has appeared in any conflict. It also takes the arity of a conflict into account. Each time a conflict is found, the weight of its decision constraints (i.e., assignments which are in the conflict) is increased by $1/r$ where r is the arity of the conflicting constraint. These weights are used for selections of decisions which are negated when a dead-end is reached.

In our approach, the conflict-based statistics works as an advice in the value selection criterion. It helps to avoid repetitive, unsuitable assignments of the same value to a variable. In particular, conflicts caused by this assignment in the past are memorized. In contrast to the weighting-conflict heuristics proposed in

[JL02], conflict assignments are memorized together with the assignment which caused them. Also, we propose our statistics to be unlimited, to prevent short-term as well as long-term cycles.

5.1.2. General Conflict-based Statistics

The main idea behind conflict-based statistics is to memorize conflicts and discourage their future repetition. For instance, when a value is assigned to a variable, conflicts with some other assigned variables may occur. This means that there are one or more constraints which prohibit the applied assignment together with the existing assignments. A counter, tracking how many times such an event occurred in the past, is stored in memory. If a variable is selected for an assignment (or reassignment) again, the stored information about repetition of past conflicts is taken into account.

Conflict-based statistics is a data structure that memorizes hard conflicts which have occurred during the search together with their frequency and assignments which caused them.

Definition 5.1 *(CBS).* Conflict-based statistics is an array

$$\mathrm{CBS}[\mathrm{V}_a = v_a \rightarrow \neg\, \mathrm{V}_b = v_b] = \mathrm{c}_{ab}$$

which means that the assignment $\mathrm{V}_a = v_a$ caused c_{ab} times a hard conflict with the assignment $\mathrm{V}_b = v_b$ in the past.

Note that it does not imply that these assignments $\mathrm{V}_a = v_a$ and $\mathrm{V}_b = v_b$ cannot be used together in case of non-binary constraints. The proposed conflict-based statistics does not actually work with any constraints. It only memorizes the conflict assignments together with the assignment which caused them. This helps us capture similar cases as well, e.g., when the applied assignment violates a constraint different from the past ones, but some of the created conflicts are the same. It also reduces the total space allocated by the statistics.

Also note that we do not exactly express what "conflict" mean in general, since it can vary on the algorithm where the conflict-based statistics is to be used. For instance, in the iterative forward search, conflicting assignments (i.e., assignments that are incompatible with the new assignment, see chapter 4.2.5) can be used.

The conflict-based statistics can be implemented as a hash table. Such structure is empty in the beginning. During computation, the structure contains only non-zero counters. A counter is maintained for a tuple $[\mathrm{A} = a \rightarrow \neg\, \mathrm{B} = b]$ in case that the value a was selected for the variable A and this assignment $\mathrm{A} = a$ caused a conflict with an existing assignment $\mathrm{B} = b$. An example of this structure

$$\mathrm{A} = a \quad \rightarrow \quad 3 \times \neg\, \mathrm{B} = b, \;\; 4 \times \neg\, \mathrm{B} = c, \;\; 2 \times \neg\, \mathrm{C} = a, \;\; 120 \times \neg\, \mathrm{D} = a$$

expresses that variable B conflicts three times with its assignment b and four times with its assignment c, variable C conflicts two times with its assignment a and D conflicts 120 times with its assignment a, all because of later selections of value a for variable A. This structure can be used in the value selection heuristics to evaluate conflicts with the assigned variables. For example, if there is a variable A selected and its value *a* is in conflict with an assignment B = *b*, we know that a similar problem has already occurred 3 times in the past, and the conflict A = *a* can be weighted with the number 3.

5.1.3. Conflict-based Statistics in Iterative Forward Search

The IFS algorithm attempts to move from one (partial) feasible solution to another via repetitive assignment of a selected value to a selected variable. During each step, a variable and a value from its domain are chosen for assignment. This may cause some hard conflicts with already assigned variables. Such conflicting assignments are removed from the solution and they become unassigned. Finally, the selected value is assigned to the selected variable.

Conflict-based statistics memorizes these unassignments together with the assignment which caused them. Let us suppose that a value v_0 is selected for a variable V_0. To enforce feasibility of the new assignment, some previous assignments $V_1 = v_1$, $V_2 = v_2$, ... $V_n = v_n$ need to be unassigned (assignments V_1/v_1, V_2/v_2, ..V_n/v_n are returned by the function *conflicts*). As a consequence we increment the counters

$CBS[V_0 = v_0 \rightarrow \neg\, V_1 = v_1]$,
$CBS[V_0 = v_0 \rightarrow \neg\, V_2 = v_2]$,
… ,
$CBS[V_0 = v_0 \rightarrow \neg\, V_n = v_n]$

Definition 5.2 *(CBS for IFS).* Let $\Theta = (V,D,C)$ be a CSP, σ be a current feasible (partial) assignment of Θ, $A \in V$ and $a \in D_A$ selected variable and value respectively in the current iteration. The counters $CBS[A = a \rightarrow \neg\, B = b]$ where

$$B/b \in \underline{conflicts}(\sigma, A, a)$$

are incremented.

The conflict-based statistics is being used in the value selection criterion. A trivial min-conflict value selection criterion selects a value with the minimal number of conflicts with the existing assignments. This heuristics can be easily adapted to a weighted min-conflict criterion (see Figure 5.1).

Here the value with the smallest sum of the number of conflicts multiplied by their frequencies is selected. Stated in another way, the weighted min-conflict approach helps to select a certain value that might cause more conflicts than another value. The point is that these conflicts are not so frequent, and therefore they have a lower weighted sum. Our hope is that it can considerably help the search to get out of a local minimum.

```
procedure selectValue(σ, A)
   bestValues = {}; bestNrConfs = 0;

   for each a∈D_A do
      nrConfs = 0;
      for each B/b ∈ conflicts(σ, A, a) do
         nrConfs += 1+CBS[A=a -> B≠b];
      if (bestValues is empty) or (bestNrConfs > nrConfs) then
          bestValues = {a}; bestNrConfs = nrConfs;
      else if (bestNrConfs == nrConfs) then
          bestValues = bestValues ∪ {a};
      end if
   end for

   a = randomly selected a value from bestValues;
   for each B/b ∈ conflicts(σ, A, a) do
      CBS[A=a -> B≠b]++;
   end for

   return a;
end procedure
```

Fig. 5.1. Weighted min-conflict value selection criterion.

Space Complexity

Let us study the space complexity of the data structure needed for conflict-based statistics. We will consider two different variables V_a and V_b linked by a constraint prohibiting concurrent assignments $V_a = v_a$ and $V_b = v_b$. In the worst case, a counter exists for each pair of such possible assignments $V_a = v_a$ and $V_b = v_b$. However, each increment of a counter in the statistics means an un-assignment of an assigned variable. Therefore each counter CBS[$V_a = v_a \rightarrow \neg V_b = v_b$] = n in the statistics means that there was an assignment $V_b = v_b$ which was unassigned n times when v_a was assigned to V_a.

Lemma 5.1 *(space complexity of CBS in IFS).* The total sum of all counters in the CBS plus the current number of assigned variables equals to the number of processed iterations plus the number of assigned variables in the initial assignment.

$$\sum\{CBS(V_a=v_a\rightarrow\neg V_b=v_b) \mid V_a,V_b\in V, v_a\in Dv_a, v_b\in Dv_b\} + |\sigma| = iter + |\alpha|$$

where α is the initial assignment (the one IFS started from), *iter* is the number of iterations processed so far and σ is the current assignment (i.e., the assignment after *iter*-th iteration).

Proof: The Lemma 5.2 is a direct consequence of the fact that there is exactly one assignment done in every iteration step. There were $|\alpha|$ variables assigned before IFS started and *iter* assignments made so far. After *iter*-th iteration, there are $|\sigma|$ variables assigned. So, there were $|\alpha|$ + *iter* - $|\sigma|$

unasignments in the past *iter* iterations and this is also the number of increments in CBS made.□

Therefore, if the above described hash table (which is empty at the beginning and does not contain empty counters) is used, the total number of all its counters will never exceed the number of iterations processed so far.

Extensions

We plan to study the following extensions of the conflict-based statistics:

- If a variable is selected for an assignment, the above presented structure can also tell how many potential conflicts a value can cause in the future. In the above example, we already know that four times a later assignment of A = *a* caused that value *c* was unassigned from B. We can try to minimize such future conflicts by selecting a different value of the variable B while A is still unbound.
- The memorized conflicts can be aged according to how far they have occurred in the past. For example, a conflict which occurred 1000 iterations ago can have half the weight of a conflict which occurred during the last iteration or it can be forgotten at all.

Furthermore, the presented conflict-based statistics can be used not only inside the solving mechanism. The constructed „implications“ together with the information about frequency of their occurrences can be easily accessed by users or by some add-on deductive engine to identify inconsistencies and/or hard parts of the input problem. The user can then modify the input requirements in order to eliminate problems found and let the solver continue the search with this modified input problem. Actually, this feature allows discovery of all inconsistent data inputs during solution of the Purdue University timetabling problem.

5.1.4. Conflict-based Statistics in Local Search

Local search algorithms perform an incomplete exploration of the search space by repairing an infeasible complete assignment γ. In each iteration step, a new assignment γ’ is selected from the neighbouring assignments N(γ) of the current assignment γ. A neighbourhood of an assignment can be defined in many different ways, for instance, a neighbour assignment can be an assignment where exactly one variable is assigned differently. This way, a single variable is reassigned in each move.

From the conflict-based statistics’ point of view, we would like to prohibit a move (a selection of a neighbouring assignment) which repetitively causes the same inconsistency. An inconsistency can be identified by a variable whose assignment becomes inconsistent with assignments of some other variables, or a constraint which becomes violated by the move.

Definition 5.3 *(neighbouring assignments).* Let $\Theta = (V,D,C)$ be a CSP, γ be a complete (but potentially infeasible) assignment of Θ. A set of neighbouring assignments is denoted by $N(\gamma) \subseteq \Psi$, where Ψ is the set of all complete assignments $\Psi=\{\sigma \mid \sigma = \{v_1/a_1, \ldots, v_n/a_n\} \;\&\; \forall i \; a_i \in Dv_i \;\&\; |\sigma|=|V| \}$.
Usually, every assignment $\lambda \in N(\gamma)$ has to fulfil some property, e.g., there is a metrics M (e.g., a number of differently assigned variables) and a threshold τ, stating

$$\forall \lambda \in N(\gamma) \;\; M(\lambda,\gamma) < \tau.$$

Simply, in each iteration step, one or more variables are reassigned. These reassignments can cause that one or more unchanged variables become inconsistent with the new assignment. This means that there is a constraint which was satisfied by the previous assignment and it is violated in the new assignment. The reason is that it prohibits concurrent value assignment of some unchanged variable(s) and some reassigned variable(s). The conflict-based statistics can memorize this problem (i.e., unchanged variables become inconsistent) together with its reason (i.e., reassigned variables). Moreover, we can use the same structure of counters $CBS[V_a = v_a \rightarrow \neg\, V_b = v_b] = c_{ab}$ as above.

More precisely, we have reassigned variables V_1, V_2, ... V_n that caused unchanged variables W_1, W_2, ... W_m to become inconsistent. Let us suppose that v_i is a new value assigned to V_i and w_j is a value assigned to W_j. If a constraint between V_i and W_j becomes violated, the counter $CBS[V_i = v_i \rightarrow \neg\, W_j = w_j]$ is incremented. Note also that such constraint might operate over more than two variables and some of its variables might already be inconsistent in the prior iteration because of another constraint.

Definition 5.4 *(reassigned variables).* Let $\Theta = (V,D,C)$ be a CSP, α, β be complete (infeasible) assignments where $\beta \in N(\alpha)$, newly assigned variables are

$$diff_V(\alpha,\beta)=\{v \mid v \in V \;\&\; v/a \in \alpha \;\&\; v/b \in \beta \;\&\; a \neq b\}$$

Definition 5.5 *(newly violated assignments).* Let $\Theta = (V,D,C)$ be a CSP, α, β complete (infeasible) assignments where $\beta \in N(\alpha)$ and $v \in diff_V(\alpha,\beta)$. A set of newly violated assignments is
$viol_V(\alpha,\beta,v)=\{w/b \in \alpha \cap \beta \mid \exists c \in C \; v,w \in dom(c) \;\&\; c(\alpha) \;\&\; \neg c(\beta)\}$

Definition 5.6 *(variable-based CBS for LS).* In each iteration step (let $\beta \in N(\alpha)$ is selected), the counters $CBS[v = a \rightarrow \neg\, w = b]$ where

$$v \in diff_V(\alpha,\beta) \;\&\; v/a \in \beta \;\&\; w/b \in viol_V(\alpha,\beta,v)$$

are incremented.

For example, there might be values v_1 and v_2 assigned to variables V_1 and V_2 respectively. As a consequence two constraints become inconsistent:

- the constraint C_1 prohibits the assignment $V_1 = v_1$ with an existing assignments $W_1 = w_1$ and $W_2 = w_2$,

- the constraint C_2 prohibits both assignments $V_1 = v_1$, $V_2 = v_2$ with $W_3 = w_3$ and $W_4 = w_4$, but $W_4 = w_4$ is already inconsistent because of some other constraint C_3.

Then, the following counters are incremented:

- $CBS[V_1 = v_1 \rightarrow \neg\, W_1 = w_1]$ and $CBS[V_1 = v_1 \rightarrow \neg\, W_2 = w_2]$ wrt. C_1
- $CBS[V_1 = v_1 \rightarrow \neg\, W_3 = w_3]$ and $CBS[V_2 = v_2 \rightarrow \neg\, W_3 = w_3]$ wrt. C_2

The conflict-based statistics is used in the move selection criterion. For example, if there is a reassignment $V_a = v_a$ contained in the move, and it causes an unchanged assignment $V_b = v_b$ to become inconsistent, such move can be weighted by the counter $CBS[V_a = v_a \rightarrow \neg\, V_b = v_b]$.

As for space complexity of the CBS structure, again, there could be a counter for each pair of possible assignments $V_a = v_a$ and $V_b = v_b$, where $V_a \neq V_b$ and there is a constraint between variables V_a and V_b which can prohibit concurrent assignments $V_a = v_a$ and $V_b = v_b$. Unfortunately, we cannot precisely limit the speed how the above structure will grow as we did for IFS, but since the number of conflicts should decrease during the search, the structure should grow slowly as well.

In some cases, it might be easier to identify an inconsistency not as a variable whose assignment becomes inconsistent but a constraint which becomes violated by the move. For instance, if there are only binary constraints in the problem, it is easier to check that both variables are assigned and compatible than to check whether there is a constraint connected to a value which is not consistent. On the other hand, sometimes it is easier (and even more straightforward) to check the consistency of variables. For instance, we might rather check if two lectures which take place in the same room are not overlapping than to check that the room constraint is violated. Note that for Random Placement Problem there is only one constraint, but it is connecting all the variables (see Chapter 6.2).

The above described conflict based statistics can also be used for memorizing the reason, why a constraint becomes inconsistent. Now, it is an array

$CBS[V_a = v_a \rightarrow \neg\, C_b\,]$,

where C_b is the constraint which becomes violated because of the recent assignment $V_a = v_a$.

Definition 5.7 *(newly violated constraints).* Let $\Theta = (V,D,C)$ be a CSP, α, β be complete (infeasible) assignments where $\beta \in N(\alpha)$ and $v \in \mathit{diff}_v(\alpha,\beta)$. A set of newly violated constraints for v is

$$\mathit{viol}_c(\alpha,\beta,v)=\{c \in C \mid v \in \mathit{dom}(c)\ \&\ c(\alpha)\ \&\ \neg c(\beta)\}$$

Definition 5.8 *(constraint-based CBS for LS).* In each iteration step (let $\beta \in N(\alpha)$ is selected), the counters $CBS[v = a \rightarrow \neg\, c]$ where

$$v \in \mathit{diff}_v(\alpha,\beta)\ \&\ v/a \in \beta\ \&\ c \in \mathit{viol}_c(\alpha,\beta,v)$$

are incremented.

In the example above, if the constraints C_1 and C_2 become inconsistent by the move where values v_1 and v_2 are assigned to variables V_1 and V_2 (note that C_3 is already inconsistent because of the assignment $W_4 = w_4$ and some others), the following counters are incremented:

- CBS[$V_1 = v_1 \rightarrow \neg C_1$] since C_1 prohibits $V_1 = v_1$ to be assigned together with the existing assignments $W_1 = w_1$ and $W_2 = w_2$.
- CBS[$V_1 = v_1 \rightarrow \neg C_2$] and CBS[$V_2 = v_2 \rightarrow \neg C_2$] since C_2 prohibits both assignments $V_1 = v_1$, $V_2 = v_2$ with $W_3 = w_3$ and $W_4 = w_4$.

5.2. Maintaining Arc Consistency

Because the presented IFS algorithm works with partial feasible assignments, it can be easily extended to maintain arc consistency during the search. This can be done by using well known dynamic arc consistency algorithms (e.g., by AC|DC algorithm published in [NB94] or DnAC6 published in [Deb96]) which are widely used in Dynamic CSPs [VJ03].

Moreover, since only the constraints describing assignments (constraint Variable = value) can be added and removed during the search, approaches based on explanations [JDB00, Jus03] can be used as well. In this section, we present how these explanations, which are traditionally used in systematic search algorithms, can be used in our iterative forward search approach in order to maintain arc consistency.

5.2.1. Related Works

Arc consistency (AC) technique removes values from variables' domains that are inconsistent with constraints. In particular, the pair of variables (V_i,V_j) is arc consistent if and only if for every value x in the current domain of V_i which satisfies the constraints on V_i there is some value y in the domain of V_j such that $V_i = x$ and $V_j = y$ is permitted by the constraint between V_i and V_j (see Definition 2.6).

There are several arc consistency algorithms starting from AC-1 [Mac77] and concluding somewhere at AC-7 [BF99]. These algorithms are based on repeated revisions of arcs till a consistent state is reached or some domain becomes empty. The most popular among them are AC-3 [Mac77], AC-3.1 [ZY01], AC2001 [BR01], AC-4 [MH86] and AC-6 [Bes94]. The AC-3 algorithm performs re-revisions only for those arcs that are possibly affected by a previous revision. It does not require any special data structures as opposed to AC-4 or AC-6 that work with individual pairs of values to eliminate potential inefficiency of checking pairs of values again and again.

Arc consistency algorithms can be easily adapted to add constraints incrementally. However, they are ineffective to relax a constraint because they are not able to determine the set of values that must be restored in the domain. So, to remove a constraint with these algorithms we have to reset the domains and to add all the remaining constraints on the initial CSP.

In AC|DC algorithm published in [NB94], which is based on AC-3, a constraint retraction is done in three steps: the first one proposes a set of restorable values for the variables connected by the deleted constraint. Then, the consequences of these potential additions are propagated throughout the constraint network. Finally, arc consistency is applied starting from the variables whose domain has been enlarged, working only on the restorable values to filter out the ones that are inconsistent with respect to the relaxed problem.

In the arc-consistency algorithm DnAC-4 published in [Bes91], a justification for each deleted value is stored. The algorithm uses these justifications to determine the set of values that have been removed because of the relaxed constraint and so can process relaxations incrementally. DnAC-4 is an adaptation of AC-4 algorithm. There is also an algorithm DnAC-6 [Deb96] which is based on AC-6 algorithm.

The explanation-based approach [JDB00, Jus03] also memorizes why the value was removed from the variable's domain:

Definition 5.9 *(explanation).* An explanation,

$$V_i \neq v_i \leftarrow (V_1 = v_1 \ \&\ V_2 = v_2 \ ...\ \&\ V_j = v_j)$$

describes that the value v_i cannot be assigned to the variable V_i since it is in a conflict with the existing assignments $V_1 = v_1$, $V_2 = v_2$, ... $V_j = v_j$.

This means that there is no complete feasible assignment containing assignments $V_1 = v_1$, $V_2 = v_2$, ... $V_j = v_j$ together with the assignment $V_i = v_i$ (these equalities form a no-good set [Jus03]).

So, for instance, if the value v_1 is no longer assigned to the variable V_1, the inequality $V_i \neq v_i$ needs to be revised. If there is no other reason why the value v_i cannot be assigned to the variable V_i, v_i is returned to the domain of the variable V_i. Otherwise, there is a new explanation attached to the inequality $V_i \neq v_i$.

5.2.2. IFS with MAC

During the arc consistency maintenance, when a value is deleted from a variable's domain, the reason (forming an explanation) can be computed and attached to the deleted value. Once a variable (say V_x with the assigned value v_x) is unassigned during the search, all deleted values which contain a pair $V_x = \mathrm{v}_x$ in their explanations need to be revised. Either such value can be still inconsistent with the current (partial) assignment (a different explanation is attached to it in this case) or it can be returned back to its variable's domain. Arc consistency is maintained after each iteration step, i.e., the selected assignment is propagated into the not yet assigned variables. When a value v_x is assigned to a variable V_x, an explanation $V_x \neq v_{x'} \leftarrow V_x = v_x$ is attached to all values $v_{x'}$ of the variable V_x, different from v_x.

In the case of forward checking (only constraints going from assigned variables to unassigned variables are revised), computing explanations is rather easy. A value v_x is deleted from the domain of the variable V_x only if there is a constraint which prohibits the assignment $V_x = v_x$ because of the existing assignments (e.g., $V_y = v_y$, … $V_z = v_z$). An explanation for the deletion of this value v_x is then $V_x \neq v_x \leftarrow (V_y = v_y \,\&\, ...\, V_z = v_z)$, where $V_y = v_y \,\&\, ...\, V_z = v_z$ are assignments contained in the prohibiting constraint. In case of arc consistency, a value v_x is deleted from the domain of the variable V_x if there is a constraint which does not permit the assignment $V_x = v_x$ with other possible assignments of the other variables in the constraint. This means that there is no support value (or combination of values) for the value v_x of the variable V_x in the constraint. An explanation is then a union of explanations of all possible support values for the assignment $V_x = v_x$ of this constraint which were deleted. The reason is that if one of these support values is returned to its variable's domain, this value v_x may be returned as well (i.e., the reason for its deletion has vanished, a new reason needs to be computed).

Note that in our implementation, we consider all constraints to be binary. Arc consistency is maintained over a non-binary constraint, only when such a constraint implements an optional method *isConsistent* stating whether the constraint is satisfied with an assignment of two of its variables, i.e.,

$$\underline{isConsistent}(c,A,a,B,b) \Leftrightarrow c(\Theta\{A/a,B/b\}),$$

where $\Theta=(V,D,C)$ is a CSP, $c \in C$, $A,B \in dom(c)$, $A \neq B$, $a \in D_A$, $b \in D_B$. For instance, in case of a resource constraint, method *isConsistent* checks whether the given two activities consuming the same resource are overlapping in time or not. Then, a value *a* is removed from the domain of variable A if there is a constraint c and a variable $B \in dom(c)$, and there is no $b \in D_B$ so that *isConsistent*(c,A,*a*,B,*b*) is true (there are no support values in the domain of variable B). An explanation for $A \neq a$ is a union of explanations of all values *b*' that were deleted from D_B for which *isConsistent*(c,A,*a*,B,*b*') holds true.

```
procedure ifs-mac(V,D,C)
   σ = β = {};    // current and best assignments
   // enforce arc consistency of the input problem
   if (makeAC() is false) return σ;
   while canContinue(σ) do
      A = selectVariable(σ);
      a = selectValue(σ, A);
      for each B/b ∈ conflicts(σ, A, a) do unassign-mac(σ,B,b);
      assign-mac(σ,A,a);
      if better(σ, β) then β = σ;
   end while
   return β;
end procedure
```

Fig. 5.2. The iterative forward search algorithm with MAC

As for the arc consistency maintenance inside IFS algorithm, the IFS algorithm scheme remains as it is described in the previous chapter, we only need to enforce arc consistency of the initial assignment and to maintain the arc consistency during the search. The maintenance of arc consistency is done during unassignment of the conflicting assignments (returned by function *conflicts*) and during assignment of the selected value to the selected variable (see Figure 5.2).

An explanation $V_x \neq v_x \leftarrow (V_y = v_y \; \& \; ... \; V_z = v_z)$ is denoted by the table EXP[$V_x \neq v_x$] that contains a set of assignments explaining the cause of the removal of value v_x from the domain of the variable V_x , i.e., $\{V_y/v_y, ... V_z/v_z\}$. If the value v_x is in the domain of variable V_x, EXP[$V_x \neq v_x$] is *null*. Note that we do not explicitly change the domains of variables during the search, but a current domain of a variable A is defined by the set {$a \in D_A$ | EXP[$A \neq a$] is *null*}, where D_A is the initial domain of the variable A (after the problem is made arc consistent).

An example implementation of the arc consistency maintenance is indicated in the Figure 5.3. It is an extension of the IFS algorithm that only enforces assignment feasibility (see Definition 4.3) during the search. Other possibility is to put all this logic into *conflicts* function which returns the assignments that needs to be unassigned in order to make the problem consistent respecting some given consistency technique ζ (see Definition 4.4).

Procedure *assign-mac*(σ, A, *a*) enforces arc consistency of the new assignment $\sigma \cup \{A/a\}$. If we allow an inconsistent value *a* of variable A to be selected by the value selection heuristics (*a* is not in the current domain of the variable A, i.e., there is an explanation attached to $A \neq a$), the value *a* needs to be returned into the domain of variable A first. This can be done by repeated selection and unassignment of one of the assignments from the explanation of $A \neq a$, until the value *a* is returned to the domain of variable A.

Procedure *unassign-mac*(σ, A, *a*) “undos” the propagations made by the assignment $\sigma \cup \{A/a\}$ that are consistent with the assignment σ. It means that all explanations that contain assignment A/a need to be revised. This is done via recomputation of all of these explanations followed by arc consistency maintenance over those which are not consistent with the assignment σ (the consistent values are returned into their variables’ domains).

Note that the sets of current support values (returned by function *currentSupports*) do not need to be computed over and over, but they can be pre-computed at the beginning of the search and then maintained during the search, like it is usually done in AC-4 algorithm [MH86].

```
procedure assign-mac(σ, A, a)
  while (EXP[A≠a] is not null) do
     select B/b from EXP[A≠a];
     unassign-mac(σ, B, b);
  end if
  //perform the assignment
  σ = σ ∪ {A/a};
  //adapt explanations of the given variable
  queue = {};
  for each a′ ∈ $D_A$ so that a≠a′ and EXP[A≠a′] is null do
     EXPL(A≠a′) = {A/a};
     queue += A/a′;
  end for
  propagate(queue);
end procedure

procedure unassign-mac(σ, B, b)
  //perform the unassignment
  σ = σ - {B/b};
  //remove explanations that contain B/b
  back = {};
  for all A/a where B/b ∈ EXP[A≠a] do
    EXP[A≠a] = null;
    back += A/a;
  end for
  //and revise values that have an explanation removed
  queue = {};
  for each A/a ∈ back do
    //if variable A is assigned to some other value a′:
    //  attach an explanation A≠a <- {A=a′}
    //  this is needed because variables can be unassigned in a
    //  different order they were assigned
    if (∃a′∈$D_A$ a≠a′ & A/a′∈σ) then
      EXP[A≠a] = {A/a′};
      queue += A/a;
    else
      revise(A,a,queue);
    end if
  end for
  //enforce arc consistency of the new solution
  propagate(queue);
end procedure

//all values of B supporting assignment A/a in constraint c
procedure supports(c, A, a, B)
  sup = {};
  for each b ∈ $D_B$ do
    if isConsistent(c, A, a, B, b) then sup += b;
  end for
  return sup;
end procedure
```

(continues on the next page)

```
//all values from the current domain of B supporting assignment A/a
 //in constraint c
procedure currentSupports(c, A, a, B)
  sup = {}; //go only over variables of the current domain of B
  for each b ∈ D_B so that EXP[B≠b] is null do
    if isConsistent(c, A, a, B, b) then sup += b;
  end for
  return sup;
end procedure

 //explanation of A≠a when there is no support value from the
 //current domain of variable B in constraint c
procedure computeExplanation(c, A, a, B)
  expl = {};
  for each b ∈ supports(c, A, a, B) do
    expl = expl ∪ EXP[B≠b];
  end for
  return expl;
end procedure

  //revise consistency of the assignment A/a, add it into the queue
  //when it is not consistent (i.e., there is a constraint c and a
  //variable B so that there are no current support from B that
  //satisfies constraint c with the assignment A/a
procedure revise(A, a, queue)
  for each constraint c so that A∈dom(c) and each B∈dom(c) do
    if (currentSupports(c,A,a,B) is empty) then
      EXP[A≠a] = computeExplanation(c, A, a, B);
      queue += A/a;
    end if
  end for
end procedure

  //propagate the assignments that are no longer consistent –
  //i.e., check whether there are some other values that are
  //no longer supported (which were initially supported by the
  //assignments in the queue)
procedure propagate(queue)
  while (queue not empty) do
    A/a <- remove and return first element from queue;
    for each constraint c so that A∈dom(c) and each B∈dom(c) do
      for each b ∈ currentSupports(c,A,a,B) do
        if (currentSupports(c,B,b,A) is empty) then
           EXP[B≠b] = computeExplanation(c, B, b, A);
           queue += B/b;
        end if
      end for
    end for
  end while
end procedure
```

Fig. 5.3. IFS with MAC – changes in assign and unassign procedures.

In traditional dynamic arc consistency algorithms (e.g., AC|DC, DnAC, ...) the value selection function chooses a value only among the values in the current domain of the variable, i.e., among the values that are not pruned by arc consistency. Using the presented explanations-based approach gives us more flexibility since we know the cause of deletion of a deleted value (each deletion has an explanation attached). For instance, in the value selection function, we can select a value not only from the current domain of the selected variable but also a value which was previously “deleted” via MAC. If a deleted variable is selected, it can become feasible by repeatedly unassigning a selected value from its explanation until the value is returned to the selected variable's domain.

For instance, in the following chapter, we compare two possibilities how to treat a case when there is a variable with an empty domain (i.e., all its values were deleted via MAC) detected. In the first case (denoted IFS MAC), while there is a variable with an empty domain, the algorithm selects and unassigns a variable that is present in explanations of values of the variable with an empty domain (a probability of a selection of a variable corresponds with the frequency of presence of the variable in the explanations of values of the empty domain variable). In the second case (denoted IFS MAC+), the algorithm continues extending the solution even when there is a variable with an empty domain. If the

selected variable does not contain any value in the current domain, one of its removed values is selected (via min-conflict value selection) and returned into the selected variable’s domain by repeatedly unassigning a randomly selected value from its explanation. Note that the second variant is more suited for a problem where we want to compute the largest feasible solution (in the number of assigned variables) in case of an over-constrained problem.

5.3. IFS as Dynamic Backtracking with MAC

In this section, we describe how the presented iterative forward search framework can be used to mimic dynamic backtracking (DB) [Gin93] search with the arc consistency maintenance (MAC) [Bes91]. In a sense, the presented IFS algorithm with MAC can be seen as an extension of DB with MAC, e.g., described in [JDB00], towards the local search based methods.

5.3.1. Related Works

Dynamic backtracking algorithm is described in Figure 5.4 (taken over from [JDB00]). Procedure <u>dbt</u> performs the main loop which tries to assign values to variables until a complete consistent assignment has been found. Procedure <u>assignAndCheck</u> determines whether the new partial solution (including the new assignment A/*a*) is consistent. If not, this procedure returns a nogood (a set of assignments responsible for the dead-end), explaining the failure. In order to restore a coherent state of computation, the procedure <u>handleContradiction</u> jumps to another consistent partial assignment. Domains and explanations are restored by the procedure <u>updateDomains</u>.

```
procedure dbt(V, D, C)
  $\sigma = \{\}$; //current solution
  while ($\sigma$ is not complete) do
    A/a = chooseAssignment($\sigma$);
    E = assignAndCheck($\sigma$, A, a);
    if (E is not success) then handleContradiction(E, $\sigma$);
  end while
  return $\sigma$;
end procedure

procedure assignAndCheck($\sigma$, A, a)
  for each $a' \in D_A$ so that $a \neq a'$ do
    $EXP[A \neq a'] = \{A/a\}$;
  end for
  $\sigma = \sigma \cup \{A/a\}$; $D_A = \{a\}$;
  c = checkConstraints($\sigma$,A);
  if (c is success) then return success;
  return $\{A/a\} \cup \{B/b \mid B \in dom(c)\ \&\ B/b \in \sigma\}$;
end procedure

procedure checkConstraints($\sigma$,A)
  for each constraint c so that $A \in dom(c)$ do
    if (c not satisfied with $\sigma$) then return c;
  end for
  return success;
end procedure

procedure handleContradiction(E, $\sigma$)
  if (E is empty) then return fail;
  A/a <- most recent assignment of E;
  updateDomains($\{B/b \mid A/a \in EXP[B \neq b]\}$);
  $\sigma = \sigma - \{A/a\}$; $D_A = D_A - \{a\}$;
  $EXP[A \neq a] = E - \{A/a\}$;
  if ($D_A$ is empty) then
      E' = union of $EXP[A \neq a]$ for all values a;
      handleContradiction(E', $\sigma$);
  end if
end procedure

procedure updateDomains(back)
  for each $B/b \in$ back do
    $EXP[B \neq b]$ = null;
    $D_B = D_B \cup \{B/b\}$;
  end for
end procedure
```

Fig. 5.4. Dynamic Backtracking

Procedure checkConstraints checks whether the constraints are consistent with the new assignment. If not, this procedure returns such a failing constraint. From that constraint, assignAndCheck computes a nogood. This nogood contains only the assignments involved in the failure.

Procedure handleContradiction is the contradiction handling mechanism. The assignment to be undone is determined and *backtracking* (or more exactly *backjumping*) is achieved by removing irrelevant nogoods which is performed by the updateDomains procedure.

In fact, dynamic backtracking does not perform real backtracks. When a dead-end occurs, it reconsiders only the most recent assignment that caused the contradiction. Especially, all the assignments that did not cause the dead-end remain unchanged. This is why dynamic backtracking has an additive behaviour on independent sub-problems.

In [JDB00], constraint propagation is integrated into the algorithm scheme above. First, when a failure occurs, computation of nogoods (the variable assignments in the failing constraint) is extended. Effects of propagation (value removals) are taken into account: eliminating explanations produced by the filtering algorithm need to be kept. Second, when an assignment is being undone, putting back in the domains values with irrelevant explanations is not sufficient since there may exist another relevant explanation for the deleted value. Value restoration needs to be confirmed by the propagation algorithm. This is similar to what is done for maintaining arc-consistency in dynamic CSPs.

5.3.2. IFS as Dynamic Backtracking with MAC

Dynamic backtracking with MAC can come out of the above presented IFS with MAC via the following modifications and/or restrictions. The purpose of this chapter is to show that the incomplete (local search based) iterative forward search algorithm can be turned into a complete (backtracking based). For more details about dynamic backtracking algorithm with MAC see [JDB00].

- Variable selection function selectVariable always returns an unassigned variable. If there are one or more variables with empty domains, one of them is returned in the variable selection function.
- Value selection function selectValue always returns a value from the selected variable's domain (i.e., not-deleted value); if there is no such value, it returns *null*.
- When all the variables are assigned, the solver terminates and returns the found solution (termination condition function canContinue). In case of branch&bound technique the existence of a complete solution should lower the bound so that a conflict arises, which leads to some unassignments.
- If the selected value is *null* (which means that the selected variable has an empty domain), a union of all assignments which prohibits all the values of the selected variable (a union of assignments of all values' explanations) is computed. The last assignment made is selected (each variable can memorize an iteration number, when it was assigned for the last time). This assignment has to be unassigned, all other assignments from the computed union are taken as an explanation for this unassignment. If the computed explanation is empty (e.g., $V_x \neq v_x \leftarrow \emptyset$), the value can be permanently removed from its variable's domain because it can never be a part of a complete solution.

If the computed union is empty, there is no complete solution and the algorithm returns fail (see Figure 5.5 for details).

```
procedure ifs-dbt(V,D,C)
   σ = β = {};    // current and best assignments
   // enforce arc consistency of the input problem
   if (makeAC() is false) return σ;
   while canContinue-dbt(σ) do
      A = selectVariable-dbt(σ);
      a = selectValue-dbt(σ, A);
      if (a is null)
         if (backtrack(σ, A) is false) return fail;
         end if
      else
        for each B/b ∈ conflicts(σ, A, a) do unassign-mac(σ,B,b);
        assign-mac(σ,A,a);
        if better(σ, β) then β = σ;
      end if
   end while
   return β;
end procedure

procedure backtrack(σ, A)
   E = {};
   for a in all values of variable A do
     E = E ∪ EXP[A≠a];
   end for
   if (E is empty) return false;
   B/b <- most recent assignment of E;
   unassign-mac(σ,B,b);
   EXP[B,b] = E – {B/b};
   return true;
end procedure
```

Fig. 5.5. IFS as dynamic backtracking with MAC.

Like in the above presented IFS MAC algorithm, arc consistency maintenance and its undo are called automatically after each assignment and unassignment, respectively.

5.4. Summary

In this chapter, we presented various extensions of the iterative forward search algorithm which we have described in chapter 4. The conflict-based statistics is the most important one since it helped us to be able to find high quality complete solutions for both the initial as well as the minimal perturbation problem of the large lecture timetabling problem at Purdue University [MR04, MRB05].

6. Experimental Results

The iterative forward search algorithm together with all the presented extensions has been implemented in Java. It contains a general implementation of the iterative search algorithm. The general solver operates over abstract variables and values with a selection of available extensions, basic general heuristics, solution comparators, and termination functions. It may be customized to fit a particular problem (e.g., as it has been extended for Purdue University timetabling) by implementing variable and value definitions, adding hard and soft constraints, and extending the parametric functions of the algorithm. For more details about iterative forward search framework see appendixes A and B or the API (javadoc) documentation on the attached CD-ROM. The results presented here were computed on 3GHz Pentium 4 PC running Windows XP professional, with 1 GB RAM and JDK 1.5.0.

The presented IFS algorithm performs an incomplete exploration of the solution space with no guarantee of finding a complete (and optimal) solution satisfying all the constraints. The purpose of this chapter is to experimentally verify the following properties:

- IFS is applicable on various constraint satisfaction and optimisation problems.
- IFS is competitive with other (mainly local-search) algorithms. Moreover, it performs very well on optimisation problems, especially when it is used together with the presented conflict-based statistics.
- The conflict-based statistics can be successfully used within a local search algorithm as it is described in the previous chapter.
- IFS can be used on over-constrained problems, where there is no complete solution.
- IFS is applicable on both initial (standard CSP) as well as minimal perturbation problem. Moreover, we can use the same algorithm, only with slight modifications in the variable/value selection heuristics in the whole solution production cycle.
- IFS can be used for solving real-life large scale (timetabling) optimisation problems.

Variations of IFS

Since the iterative forward search as it is described in chapter 4 is more like a framework than an exact definition of a single algorithm (it needs to be parameterised by the selection of the variable/value selection, termination and heuristics), we study several variants of this algorithm differentiated by the choice of the value selection heuristic and the extensions described in chapter 5. The compared algorithms are:

- **IFS RW(p_{rw})** ... min-conflict selection of values with p_{rw} random walk. This means, that with the given probability p_{rw}, a value is selected randomly from all values of the selected variable's domain.
- **IFS TS(l_{ts})** ... tabu search, where l_{ts} is the length of tabu list which is used to avoid cycling. Repeated selection of the same pair (variable, value) is prohibited for the given number of subsequent iterations.
- **IFS CBS** ... min-conflict value selection where conflicts are weighted according to the conflict-based statistics (as described in Chapter 5.1)
- **IFS MAC** ... arc-consistency maintenance; if there is a variable with an empty domain, a variable which caused a removal of one or more of values is selected and unassigned. This is done so that a value v_x of such a variable V_x with an empty domain is selected randomly and a randomly selected assignment from the explanation $V_x \neq v_x$ is unassigned.
- **IFS MAC+** ... arc-consistency maintenance; the algorithm continues extending the solution even when there is a variable with an empty domain. If the selected variable does not contain any value which was not removed from its domain via MAC, one of its removed values is selected (via min-conflict value selection).
- **DBT MAC** ... dynamic backtracking algorithm with arc consistency maintenance (as described in Chapter 5.3)
- **DBT FC** ... dynamic backtracking algorithm with forward checking

For all these variants, an unassigned variable is selected randomly (see Figure 6.1) and the value selection is based on min-conflict strategy (see Figure 6.2). This means that a value is randomly selected among the values whose assignment will cause the minimal number of conflicts with the existing assignments. The search is terminated when a complete solution is found or when the given time limit is reached. As for the solution comparator, a solution with the highest number of assigned variables is always selected.

```
procedure selectVariable (σ)
   // current (partial but feasible) assignment is the parameter
   unassigned = all variables that are not assigned in σ;
   return randomly selected variable from unassigned;
end procedure
```

Fig. 6.1. Variable selection criterion (IFS CBS, IFS TABU, IFS MCRW)

Because we attempt to solve large scale problems, maintaining arc consistency (algorithms IFS MAC and DBT MAC) is based on AC3 algorithm (e.g., see [Tsa93]). For instance, in the Purdue University timetabling problem we have about 800 variables (there is a variable for each course) with the total number of more than 200,000 values (there is a value for each location of a course in the timetable, including a selection of time(s), room and instructor). Furthermore, nearly every two variables are related by some constraint, e.g., typically there is at least one room they can both use. Due to the memory reasons, this prohibits any consistency method which is based on memorizing supports for each pair of values or for each pair of value and variable.

```
procedure selectValue(σ, A)
   // current (partial but feasible) assignment and the selected
   // variable are the parameters
   if (random walk) begin
      if (random()<p_rw) return randomly selected value from D_A;
   end
   bestNrConfs = 0;  bestValues = {};
   for each a∈D_A do
      η = conflicts(σ, A, a);
      if (tabu search) then
          if A/a in tabu-list then continue;
            //jump to the next value
      end if
      nrConfs = |η|;
      if (conflict-based statistics) then
          for each B/b∈η do
             nrConfs += CBS[A=a->B≠b];
      end if
      if (bestValues is empty) or (bestNrConfs > nrConfs) then
          bestValues = {a};
          bestNrConfs = nrConfs;
      else if (bestNrConfs == nrConfs) then
          bestValues = bestValues ∪ {a};
      end if
   end for
   a = randomly selected value of bestValues;
   if (tabu search) then
      add A/a at the end of tabu-list;
      if (|tabu-list|>l_ts) then
        remove the first element from tabu-list;
   end
   if (conflict-based statistics) then
      for each B/b∈conflicts(σ, A, a) do
         CBS[A=a->B≠b]++;
      end for
   end if
   return a;
end procedure
```

Fig. 6.2. Value selection criterion (IFS CBS, IFS TABU, IFS MCRW)

6.1. Binary Random CSP

In the following experiments we compare several mutations of the iterative forward search algorithm and its improvements on the Random Binary CSP with uniform distribution [Bes96]. A random CSP is defined by a four-tuple (n, d, p_1, p_2), where n denotes the number of variables and d denotes the domain size of each variable, p_1 and p_2 are two probabilities. They are used to generate randomly the binary constraints among the variables. p_1 represents the probability that a constraint exists between two different variables and p_2 represents the probability that a pair of values in the domains of two variables connected by a constraint is incompatible. We use a so called model B [MPSW98] of Random CSP (n, d, n_1, n_2) where $n_1 = p_1 n(n-1)/2$ pairs of variables are randomly and uniformly selected and binary constraints are posted between them. For each constraint, $n_2 = p_2 d^2$ randomly and uniformly selected pairs of values are picked as incompatible.

The following graphs (see Figures 6.3 and 6.4) present the number of assigned variables in percentage to all variables wrt. the probability p_2 representing tightness of the generated sparse problem CSP(50, 12, 250/1250, p_2) and dense problem CSP(25, 15, 198/300, p_2) respectively. The average values of the best achieved solutions from 25 runs on different problem instances within 60 second time limit are presented.

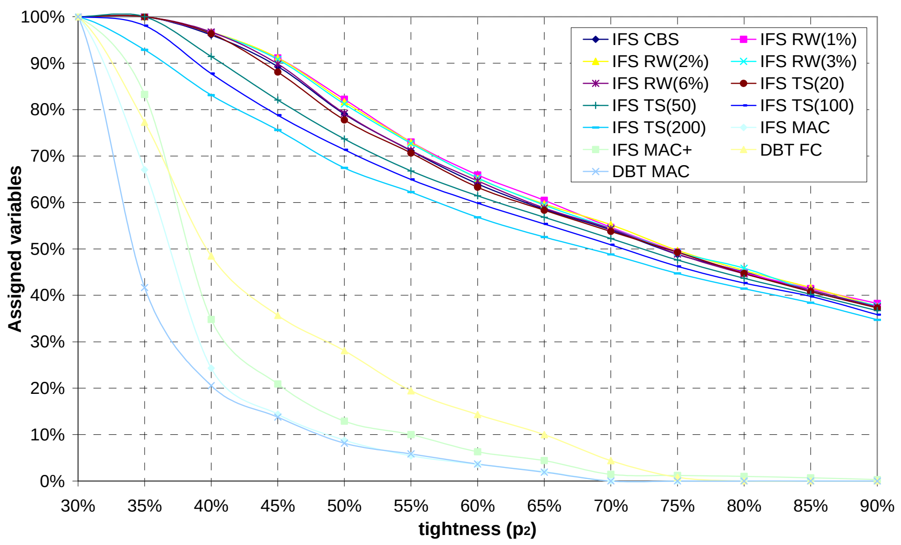


Fig. 6.3. CSP(50, 12, 250/1250, p_2), number of assigned variables.

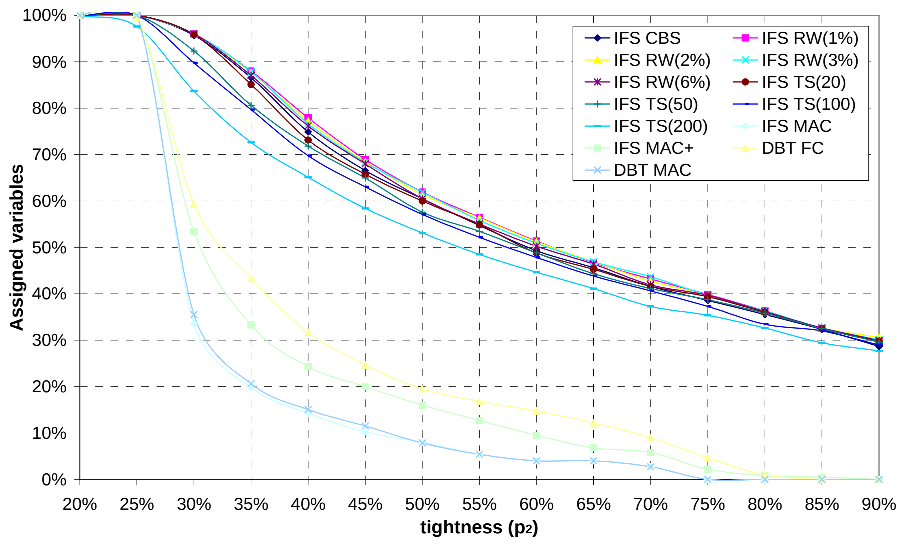


Fig. 6.4. CSP(25, 15, 198/300, p_2), number of assigned variables.

Each of the compared algorithms was able to find a complete solution within the time limit for all the given problems with a tightness under 30% for CSP(50, 12, 250/1250, p_2) and under 20% for CSP(25, 15, 198/300, p_2). Achieved results from min-conflict random walk, tabu search and the conflict-based statistics seem to be very similar for this problem (tabu search seems to be slightly worse, min-conflict slightly better than conflict-based statistics). Also, it is not surprising that a usage of consistency maintenance techniques lowers the maximal number of assigned variables, e.g., both dynamic backtracking with MAC and IFS with MAC extend an incomplete solution only when it is arc consistent with all unassigned variables. As we can see from the above figures, we can get better results when we allow the search to continue even if there is a variable with an empty domain.

The following graph (see Figure 6.5) presents the number of assigned variables in percentage to all variables obtained by IFS CBS algorithm, with respect to the probabilities p_1 and p_2 representing density and tightness of the generated problem CSP(25, 15, p_1, p_2) respectively. The average values of the best achieved solutions from 10 runs on different problem instances within 60 second time limit are presented.

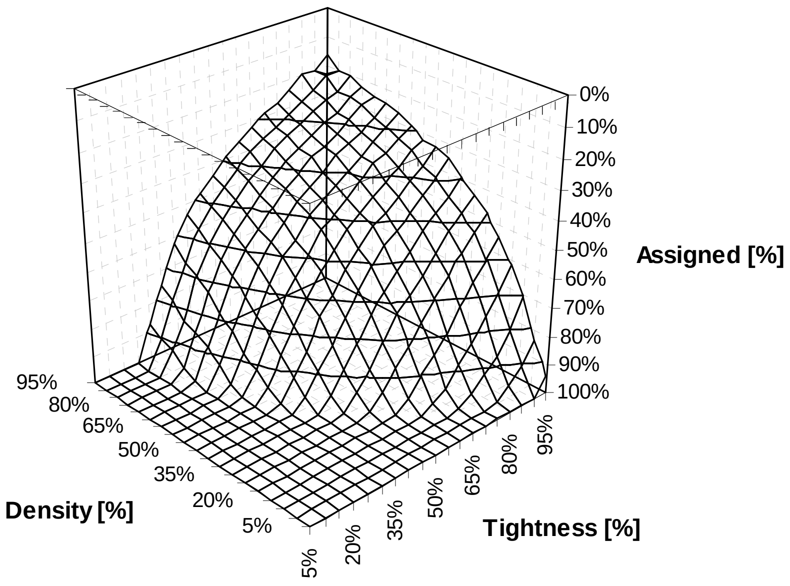


Fig. 6.5. CSP(25,15,p1,p2), number of assigned variables for IFS CBS.

Weighted Random Binary CSP (minCSP)

For the following results (Figures 6.6 and 6.7), we turned the random CSP problem into an optimisation problem (CSOP). The goal is to minimize the total sum of values for all variables. Note that each variable has d generated values from 0, 1, ... d-1. For the comparison, we used CSP(50, 12, 250/1250, p_2) and CSP(25, 15, 198/300, p_2) problems with the tightness p_2 taken so that every measured algorithm was able to find a complete solution for (almost) each of 10 different generated problems within the given 60 seconds time limit.

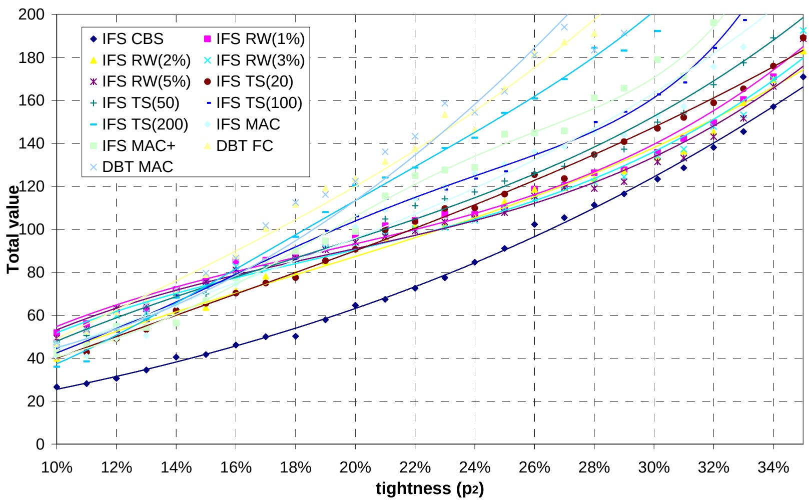


Fig. 6.6. minCSP(50, 12, 250/1250, p_2), the sum of all assigned values.

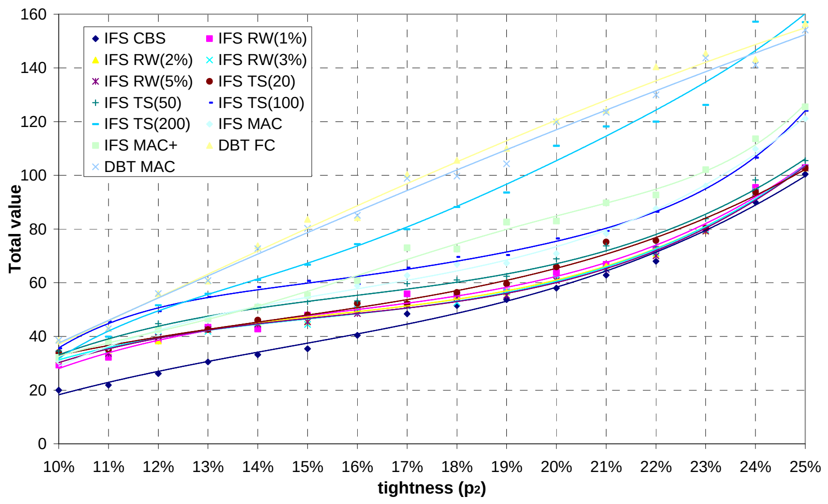


Fig. 6.7. minCSP(25, 15, 198/300, p_2), the sum of all assigned values.

All algorithms can be easily adopted to solve this minCSP problem by selecting an assignment with the smallest value among the values minimizing the number of conflicts. But, the conflict-based statistics can do better. Here, we can add the value of the assignment to the number of conflicts (weighted by CBS). Then, a value with the smallest sum of the value and the conflicts weighted by their previous occurrences is selected. We can afford this approach because the weights of repeated conflicts are being increased during the search, and the algorithm is much more likely to escape from a local minimum than the other compared algorithms.

For this problem, the presented conflict-based statistics was able to give better results than other compared algorithms. The algorithm is obviously trying to stick much more with the smallest values than the others, but it is able to find a complete solution since the conflict counters are rising during the search. Such behaviour can be very handy for many optimisation problems, especially when optimisation criteria (expressed either by some objective function or by soft constraints) go against the hard constraints.

Local Search

In this section we compare the presented conflict-based statistics with various local search algorithms. For all the compared local search algorithms, a neighbour assignment is defined as an assignment where exactly one variable is assigned differently. The compared algorithms are:

- **HC** … hill-climbing algorithm always selects the best assignment among all the neighbour assignments
- **HC RW** … hill-climbing random walk; with the given probability p_{rw} a random neighbour is selected. Otherwise, the best neighbour is selected.

- **HC TS(l_{ts})** … tabu search, where l_{ts} is the length of the tabu list; the best assignment among all the neighbour assignments is always selected. Moreover, if such an assignment is contained in the tabu list (a memory of the last l_{ts} assignments), the second best assignment is used and so on (except of an aspiration criteria, which allows to select a tabu neighbour when the best ever found assignment is found).
- **HC CBS** … hill-climbing algorithm with conflict-based statistics (constraint-based version); the best assignment is always selected as well, but the newly created conflicts are weighted according to the conflict-based statistics.
- **MC** … min-conflict algorithm selects a variable in a violated constraint randomly. Its value which minimizes the number of conflicting constraints is chosen.
- **MC RW(P_{rw})** … same as min-conflict, but with the given probability P_{rw} a random neighbour is selected.
- **MC TS(l_{ts})** … same as min-conflict, but there is a tabu-list of the length l_{ts} used.
- **MC CBS** … same as min-conflict, but the newly created conflicts are weighted according to the conflict-based statistics.

Figures 6.8 and 6.9 present the number of conflicting constraints wrt. the probability p_2 representing tightness of the generated sparse problem CSP(50, 12, 250/1250, p_2). The average values of the best achieved solutions from 10 runs on different problem instances within the 60 second time limit are presented.

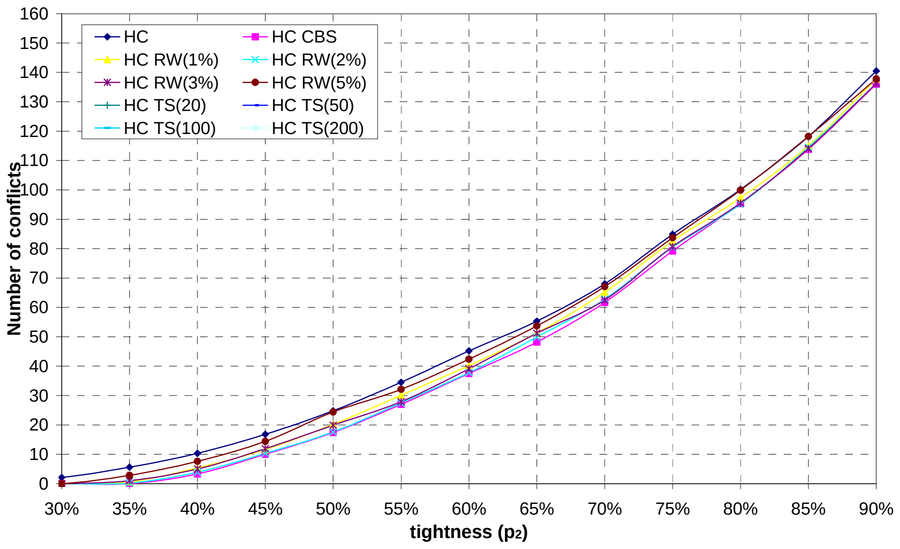


Fig. 6.8. CSP(50, 12, 250/1250, p_2), the number of conflicting constraints for hill-climbing algorithms.

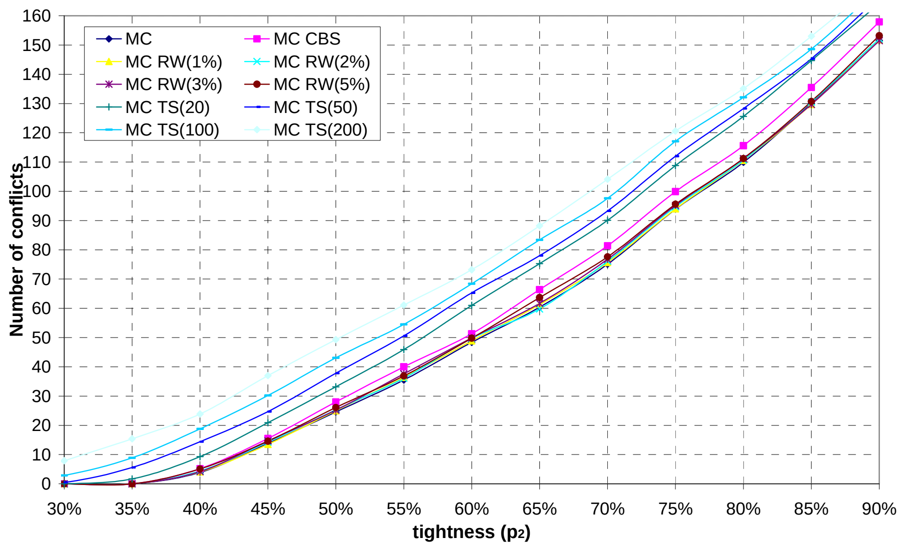


Fig. 6.9. CSP(50, 12, 250/1250, p_2), the number of conflicting constraints for min-conflict algorithms.

Figures 6.10 and 6.11 present the number of conflicting constraints wrt. the probability p_2 representing tightness of the generated dense problem CSP(25, 15, 198/300, p_2).

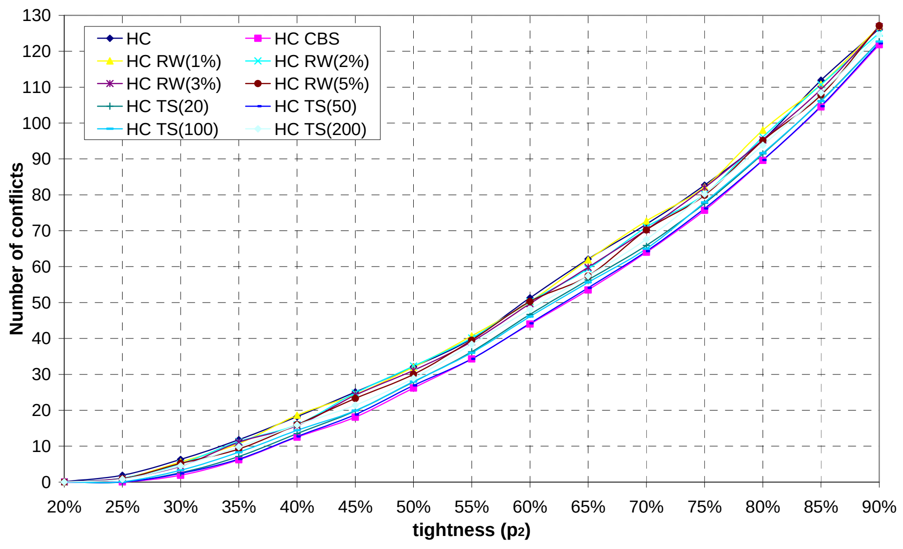


Fig. 6.10. CSP(25, 15, 198/300, p_2), the number of conflicting constraints for hill-climbing algorithms.

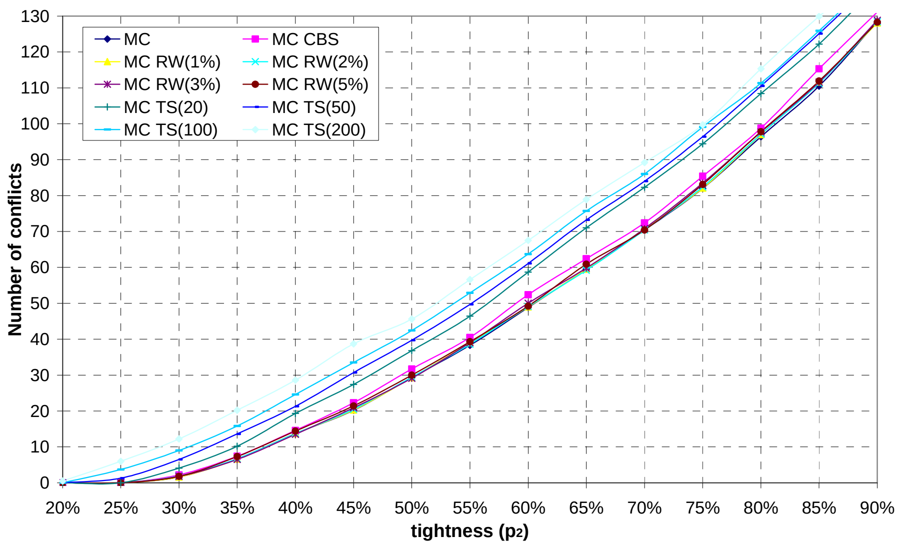


Fig. 6.11. CSP(25, 15, 198/300, p_2), the number of conflicting constraints for min-conflict algorithms.

Overall, hill-climbing algorithms produce better results than min-conflict algorithms on the tested problems. MC CBS is slightly worse than MC RW algorithms, but better than MC TS algorithms. As for hill-climbing, HC CBS seems to be slightly better than all other tested algorithms on the given random binary CSP problems. Moreover, there is no algorithm specific parameter (which usually depends on the solved problem) unlike in the compared methods (e.g., the length of the tabu-list).

Weighted Random Binary CSP (minCSP)

The following results (Figures 6.12 to 6.15) are computed on the weighted random binary CSP. As for IFS, we used the problems CSP(50, 12, 250/1250, p_2) and CSP(25, 15, 198/300, p_2) with the tightness p_2 taken so that every measured algorithm was able to find a complete solution for (almost) each of 10 different generated problems within the given 60 seconds time limit.

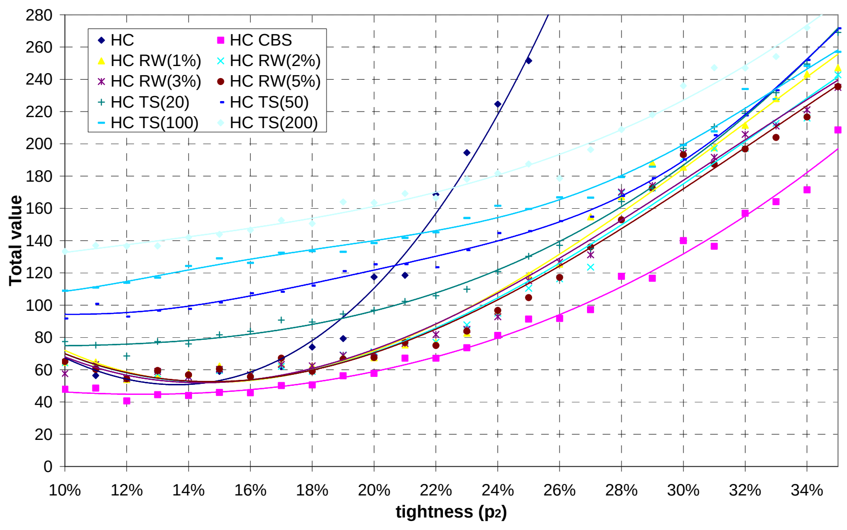


Fig. 6.12. minCSP(50, 12, 250/1250, p_2), the sum of all assigned values for hill-climbing algorithms.

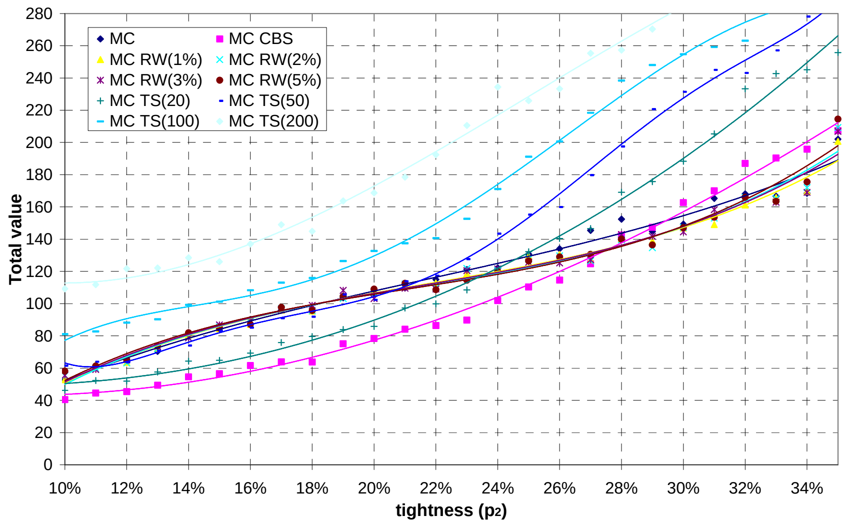


Fig. 6.13. minCSP(50, 12, 250/1250, p_2), the sum of all assigned values for min-conflict algorithms.

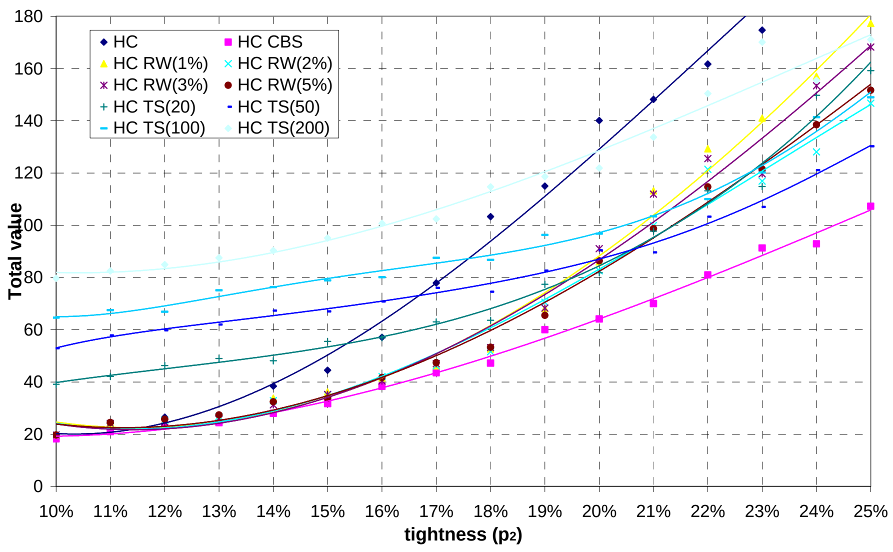


Fig. 6.14. minCSP(25, 15, 198/300, p_2), the sum of all assigned values for hill-climbing algorithms.

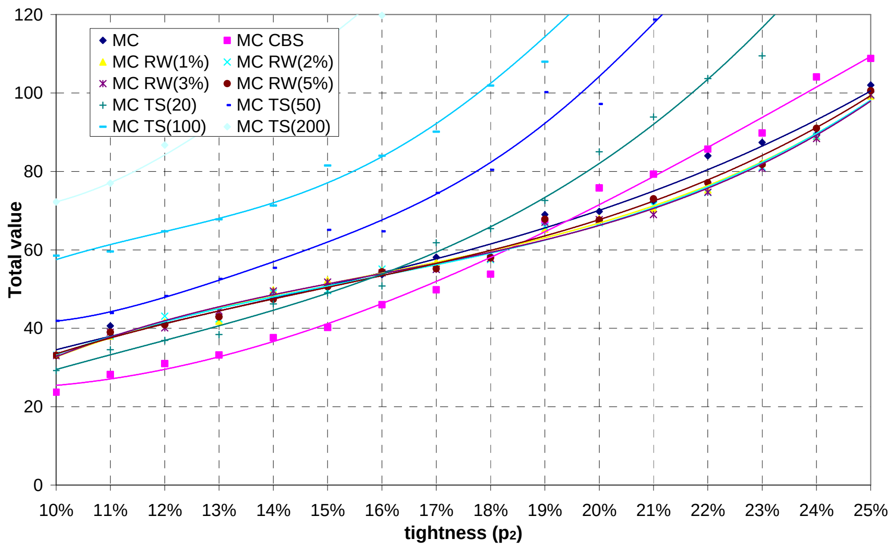


Fig. 6.15. minCSP(25, 15, 198/300, p_2), the sum of all assigned values for min-conflict algorithms.

Overall, hill-climbing algorithms produce better results than min-conflict algorithms on the tested problems. As for hill-climbing, HC CBS is able to give the best results of all tested algorithms on both problems.

6.2. Random Placement Problem

The random placement problem (RPP; for more details, see http://www.fi.muni.cz/~hanka/rpp/) seeks to place a set of randomly generated rectangles (called objects) of different sizes into a larger rectangle (called placement area) in such a way that no objects overlap and all objects' borders are parallel to the border of the placement area. In addition, a set of allowable placements can be randomly generated for each object. The ratio between the total area of all objects and the size of the placement area will be denoted as the filled area ratio.

Definition 3.8 *(RPP).* Random placement problem is a CSP $\Theta = (V,D,C)$ with the following properties:

- $V=\{x_1,y_1,x_2,y_2,\ldots, x_n,y_n\}$
- $D=\{Dx_1,Dy_1,Dx_2,Dy_2,\ldots,Dx_n,Dy_n\}$, $\forall i\ Dx_i = \{minx_i, minx_i +1, \ldots, maxx_i\}$, $Dy_i = \{miny_i, miny_i +1, \ldots, maxy_i\}$
- $C=\{c\}$, where c is the following not-overlap constraint
  $\forall i, \forall j, i\neq j \Rightarrow (x_i+dx_i\leq x_j \vee x_j+dx_j\leq x_i \vee y_i+dy_i\leq y_j \vee y_j+dy_j\leq y_i)$

where:

- n is the number of objects in RPP,
- x_i,y_i are coordinates of i-th object,
- $minx_i$, $miny_i$, max_i, $maxy_i$ are bounds of the i-th object,
- (dx_i,dy_i) is the size of i-th object,
- (R_x,R_y) is the size of placement area,
- $\forall i\ \ 0 \leq minx_i \leq maxx_i+dx_i < R_x\ \&\ 0 \leq miny_i \leq maxy_i+dy_i < R_y$

RPP allows us to generate various instances of the problem similar to a trivial timetabling problem. The correspondence is as follows: the object corresponds to a course to be timetabled – the x-coordinate to its time, the y-coordinate to its classroom. For example, a course taking three hours corresponds to an object with dimensions 3×1 (the course should be taught in one classroom only). Each course can be placed only in a classroom of sufficient capacity – we can expect that the classrooms are ordered increasingly in their size so each object will have a lower bound on its y-coordinate.

In this chapter, we present capabilities of iterative forward search on the random placement problem, in solving both initial (i.e., standard CSP) as well as minimal perturbation problem.

6.2.1. Initial Problem

The following experiments were accomplished on 8 sets of problems, each concerning 200 objects with filled area ratio of the range of 75%, 80%, 85%, … 100%. Each set contains 50 different problems with the given filled area ratio. Clearly all problems in the last two sets are over-constrained. However, this may also be true for other problems with the filled area ratio "close" to 100 %. All these problem instances are taken from http://www.fi.muni.cz/~hanka/rpp/. The sizes of the generated objects are 2x1 (80.4% of all objects), 3x1 (16.6% of all objects), 4x1 (2.6% of all objects) and 6x1 (0.4% of all objects).

Table 6.16 shows the number of problem instances completely solved in each set by the tested algorithms within 5 minute time limit.

Filled Area Ratio:	**75%**	**80%**	**85%**	**90%**	**95%**	**100%**
IFS CBS	50	50	50	45	47	33
IFS RW(1%)	50	50	50	45	47	38
IFS RW(2%)	50	50	50	45	47	38
IFS RW(3%)	50	50	50	45	47	37
IFS RW(5%)	50	50	50	45	47	37
IFS TS(20)	50	50	50	45	46	37
IFS TS(50)	50	50	50	45	46	37
IFS TS(100)	50	50	50	45	47	35
IFS TS(200)	50	50	50	45	46	35
IFS MAC	49	48	43	15	1	0
IFS MAC+	50	50	47	23	8	1

Fig. 6.16. Number of problems solved for each set of problems.

Figure 6.17 presents the average time needed to find a complete solution with respect to the filled area ratio. The average numbers over all problem instances from each set of problems are presented.

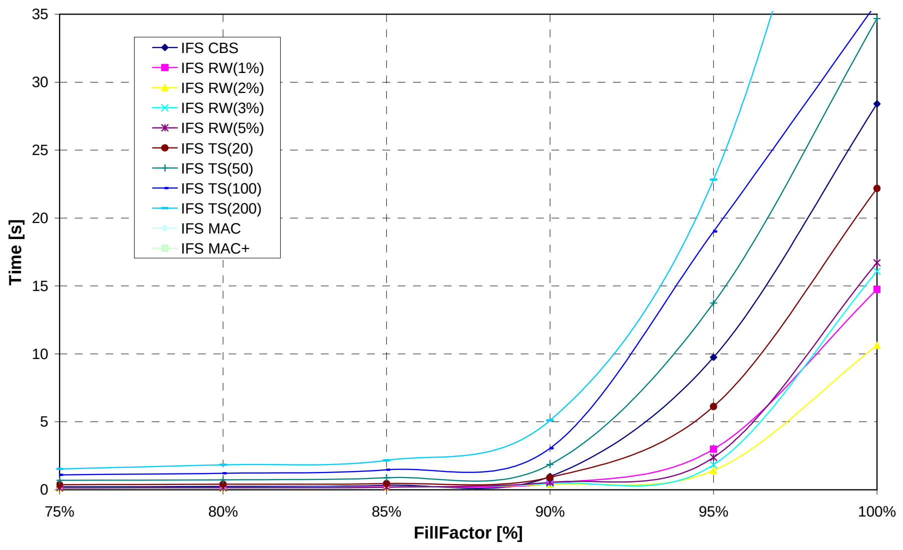


Fig. 6.17. Average time wrt. filled are ratio.

Dynamic backtracking algorithm (with either MAC or FC) was not able to solve any problem instance at all, the average number of assigned variables in percentage to all variables for DBT MAC were 90.6%, 89.9%, 84.5%, 84.5%, 83.3% and 81.68% for filled area ratio of 75%, 80%, 85%, 90%, 95% and 100% respectively.

In [Ver03] a comparison of limited assignment number search (LAN) [VR02], dynamic backtracking (DB) [Gin93], limited discrepancy search (LDS) [HG95] and simulated annealing (SA) [KGV83] on the same problem instances is made. Neither DB nor LDS were able to find any complete solution on the given instances. The comparison of LAN and SA algorithms from this work is presented on Figure 6.18.

Filled Area Ratio:	75%	80%	85%	90%	95%	100%
LAN	49	50	47	26	6	0
SA	47	44	40	22	13	6

Fig. 6.18. Number of problems solved for each set of problems.

6.2.2. Minimal Perturbation Problem

In this chapter we present some experiments on RPPs for solving minimal perturbation problem. MPP instances were generated as follows: First, the initial solution was computed. The changed problem differs from the initial problem by input perturbations. An input perturbation means that both x coordinate and y coordinate of a rectangle must differ from the initial values, i.e. $x \neq x_{initial}$ & $y \neq y_{initial}$. For a single initial problem and for a given number of input perturbations, we can randomly generate various changed problems. In particular, for a given number of input perturbations, we randomly select a set of objects which should have input perturbations. The solution to MPP can be evaluated by the number of additional perturbations. They are given by subtraction of the final number of perturbations and the number of input perturbations.

The following experiments were accomplished with IFS CBS algorithm on 7 problems, each concerning 200 objects with different filled area ratio. Filled area ratio is displayed in the parenthesis next to the name of the problem instance. The tested problem instances are available on the attached CD-ROM.

Figure 6.19 shows the number of additional perturbations as a function of the number of input perturbations. Both numbers are in percentage to the number of objects in the problem (which is 200).

The IFS CBS algorithm was able to find a complete feasible solution in each test run. Moreover, the number of additional perturbations was very low. Similarly as in the input problem, IFS RW and IFS TS were able to return very similar results as IFS CBS.

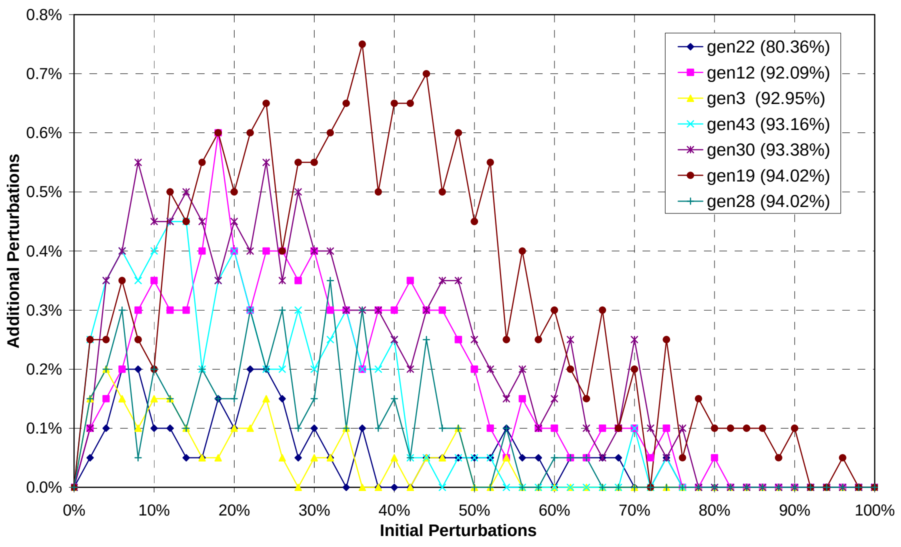
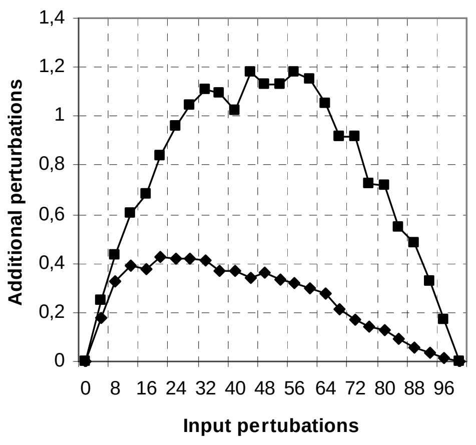


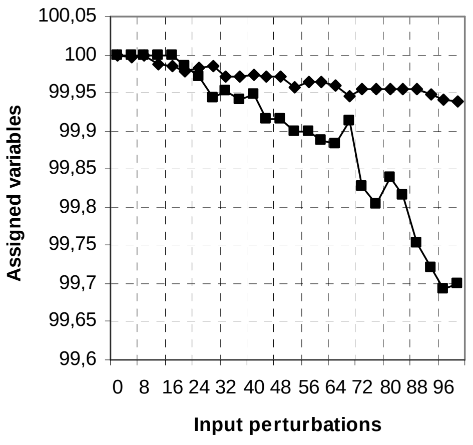
Fig. 6.19. Additional perturbations wrt. input perturbations.

In [RBM04], we proposed a branch-and-bound algorithm based on an incomplete LAN (limited assignment search) algorithm [VR02] to solve MPP problems. The Figure 6.20 presents the results from [RBM04], where the IFS algorithm was compared with the proposed branch-and-bound algorithm on the random placement problem. We used fifty initial problems and five MPPs per an initial problem. We compared the algorithms on the problems consisting of 100 objects with filled area ratio 80% and we did the experiments for input perturbations from 0 to 100 with the step 4. Thus 0 to 100% relative input perturbations are covered.

The compared branch-and-bound algorithm was implemented in SICStus Prolog with the use of disjoint2 global constraint [COC97] over x-coordinates and y-coordinates to ensure that the objects will not overlap. Unfortunately, this represents a difference in the notion of the consistency of a partial assignment between the compared branch-and-bound and IFS algorithms.

Figure 6.20 shows the number of additional perturbations, the number of assigned variables, and the CPU time as a function of the number of input perturbations for both algorithms. We expect the comparison to be the most meaningful for a smaller number of input perturbations (up to about 25%) because this is the area of the highest interest for MPPs. Here the branch-and-bound algorithm seems to be comparable with IFS solver in terms of the solution quality. Still, the IFS algorithm is much faster there.





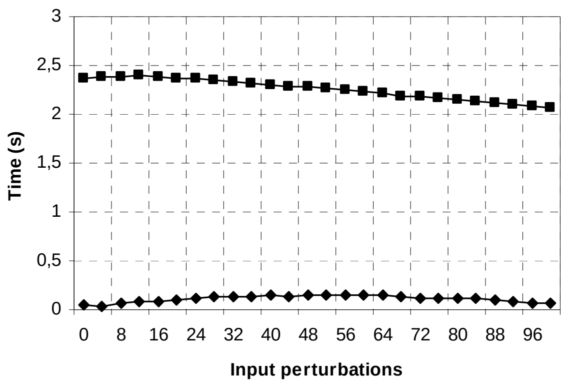


Fig. 6.20. Comparison of the IFS (♦) with branch-and-bound algorithm (■).

6.3. Purdue Timetabling Problem

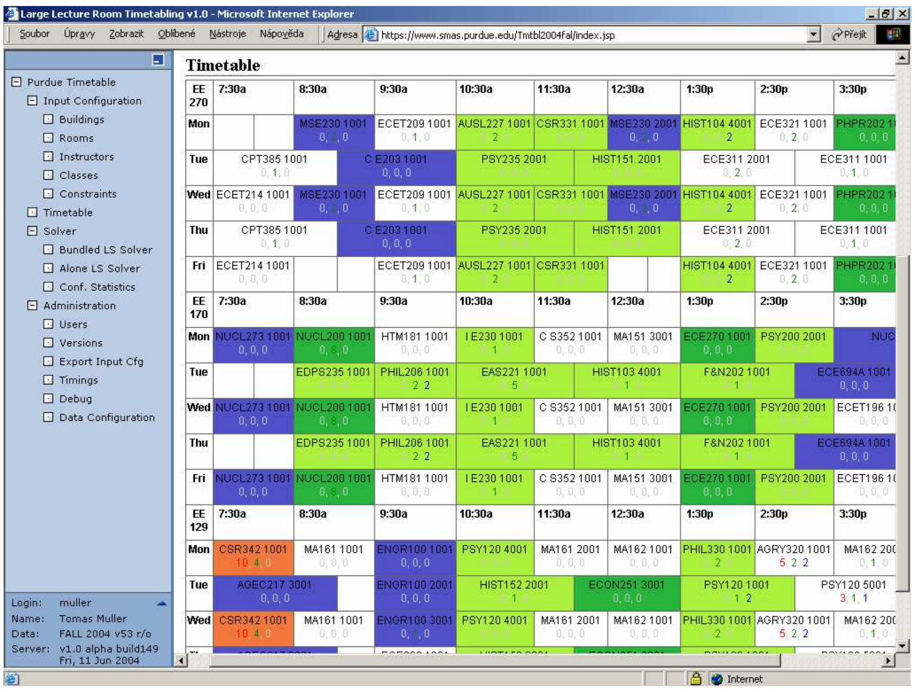
In this chapter, we present capabilities of iterative forward search on the real-life course timetabling problem of Purdue University (see chapter 3.4 for the description of the problem). Results from solving both initial as well as minimal perturbation problems are presented. Comparison of solutions given by the IFS algorithm with a hand-made solution is also an important part of this chapter.

The following experiments were performed on the complete Fall 2004 data set, including 830 classes to be placed in 50 classrooms. The classes included represent 89,677 course requirements for 29,808 students. We have achieved similar results with Fall 2001, Spring 2005 and Fall 2005 data sets as well, even though they are quite different in the number of requirements (Fall 2004 is the most constrained one out of these four data sets). See the following table for more details. The *total number of hours* represents the number of hours allocated by a class summed over all classes. The *total number of all placements* represents the number of all possible placements (i.e., placements which do not break any unary hard constraint) of a class summed over all classes.

Data set (term)	Fall 2001	Fall 2004	Spring 2005	Fall 2005
Total number of classes	747	830	793	856
Total number of meetings	1630	1791	1685	1810
Total number of used half-hours	3648	4060	3862	4096
Total number of rooms	41	50	50	51
Total number of group constraints	153	171	147	178
Total number of students	28994	29810	26087	29318
Total number of course requirements	81328	89677	74923	86874
Total number of all values	210466	168314	186892	221577

Fig. 6.21. Comparison of data sets for Purdue University Timetabling

Besides the discussed IFS solver, the timetabling application for Purdue University also contains a web-based graphical user interface (written using Java Server Pages) which allows management of several versions of the data sets (input requirements, solutions, changes, etc.), browsing the resultant solutions (see Figure 6.22), and tracking and managing changes between them.



Login: muller
Name: Tomas Muller
Data: FALL 2004 v53 r/o
Server: v1.0 alpha build149 Fri, 11 Jun 2004

Timetable

EE 270	7:30a	8:30a	9:30a	10:30a	11:30a	12:30a	1:30p	2:30p	3:30p
Mon		MSE230 1001 0, , 0	ECET209 1001 0, 1, 0	AUSL227 1001 2	CSR331 1001	MSE230 2001 0, , 0	HIST104 4001 2	ECE321 1001 0, 2, 0	PHPR202 1 0, 0, 0
Tue	CPT385 1001 0, 1, 0		C E203 1001 0, 0, 0	PSY235 2001		HIST151 2001	ECE311 2001 0, 2, 0		ECE311 1001 0, 1, 0
Wed	ECET214 1001 0, 0, 0	MSE230 1001 0, , 0	ECET209 1001 0, 1, 0	AUSL227 1001 2	CSR331 1001	MSE230 2001 0, , 0	HIST104 4001 2	ECE321 1001 0, 2, 0	PHPR202 1 0, 0, 0
Thu	CPT385 1001 0, 1, 0		C E203 1001 0, 0, 0	PSY235 2001		HIST151 2001	ECE311 2001 0, 2, 0		ECE311 1001 0, 1, 0
Fri	ECET214 1001 0, 0, 0		ECET209 1001 0, 1, 0	AUSL227 1001 2	CSR331 1001		HIST104 4001 2	ECE321 1001 0, 2, 0	PHPR202 1 0, 0, 0
EE 170	**7:30a**	**8:30a**	**9:30a**	**10:30a**	**11:30a**	**12:30a**	**1:30p**	**2:30p**	**3:30p**
Mon	NUCL273 1001 0, 0, 0	NUCL200 1001 0, 8, 0	HTM181 1001 0, 0, 0	I E230 1001 1	C S352 1001 0, 0, 0	MA151 3001 0, 0, 0	ECE270 1001 0, 0, 0	PSY200 2001	NUC
Tue		EDPS235 1001	PHIL206 1001 2 2	EAS221 1001 5		HIST103 4001 1	F&N202 1001 1		ECE694A 1001 0, 0, 0
Wed	NUCL273 1001 0, 0, 0	NUCL200 1001 0, 8, 0	HTM181 1001 0, 0, 0	I E230 1001 1	C S352 1001 0, 0, 0	MA151 3001 0, 0, 0	ECE270 1001 0, 0, 0	PSY200 2001	ECET196 1 0, 0, 0
Thu		EDPS235 1001	PHIL206 1001 2 2	EAS221 1001 5		HIST103 4001 1	F&N202 1001 1		ECE694A 1001 0, 0, 0
Fri	NUCL273 1001 0, 0, 0	NUCL200 1001 0, 8, 0	HTM181 1001 0, 0, 0	I E230 1001 1	C S352 1001 0, 0, 0	MA151 3001 0, 0, 0	ECE270 1001 0, 0, 0	PSY200 2001	ECET196 1 0, 0, 0
EE 129	**7:30a**	**8:30a**	**9:30a**	**10:30a**	**11:30a**	**12:30a**	**1:30p**	**2:30p**	**3:30p**
Mon	CSR342 1001 10, 4, 0	MA161 1001 0, 0, 0	ENGR100 1001 0, 0, 0	PSY120 4001	MA161 2001 0, 0, 0	MA162 1001 0, 0, 0	PHIL330 1001 2	AGRY320 1001 5, 2, 2	MA162 20 0, 1, 0
Tue	AGEC217 3001 0, 0, 0		ENGR100 2001 0, 0, 0	HIST152 2001 1		ECON251 3001 0, 0, 0	PSY120 1001 1 2		PSY120 5001 3, 1, 1
Wed	CSR342 1001 10, 4, 0	MA161 1001 0, 0, 0	ENGR100 3001 0, 1, 0	PSY120 4001	MA161 2001 0, 0, 0	MA162 1001 0, 0, 0	PHIL330 1001 2	AGRY320 1001 5, 2, 2	MA162 20 0, 1, 0

Fig. 6.22. Generated timetable in web-based graphical user interface.

Student Scheduling

Many courses at Purdue University consist of several sections, with students enrolled in the course divided between them. Sections are often associated together by some constraints. For example, sections of the same course should not overlap. Each such section forms one class which has its own preferences. Therefore each section is treated separately - there is a variable for each section.

An initial sectioning of students into course sections is processed. This student sectioning is based on Carter's [Car00] homogeneous sectioning and it is intended to minimize future student conflicts. However, there is still a possibility of improving the solution with respect to the number of student conflicts. This can be achieved via section changes during the search.

In the current implementation, sectioning is altered only by switching student enrolments between two different sections of the same course. Each student enrolment in a course with more than one section is processed. An attempt is made to switch it with a student enrolment from a different section. If this switch decreases the total number of student conflicts, it is applied.

We have compared two possibilities for switching these student enrolments. The first possibility is during the search, after a course is placed in the timetable. If a class is part of a course with multiple sections, an attempt is made to switch students with other sections of the course. Also, when a course has only one section, the system tries to move some students in multi-section courses who have a conflict with this class.

The second possibility, which appears to be much faster, but with similar results, is to switch students only when the best solution is found. In this case, the students are switched in the current solution, before it is stored as the best solution. All classes are processed and attempted switches are made between students in the same course. Note that a switch of a student enrolment can be followed with subsequent switches, so that classes can be processed more than once.

Search Algorithm

The quality of a solution is expressed as a weighted sum combining soft time and classroom preferences, satisfied soft group constraints and the total number of student conflicts. This allows us to express the importance of different types of soft constraints. The following weights are considered in the sum:

- $W_{student}$ … weight of a student conflict,
- W_{time} … weight of a time preference of a placement,
- W_{room} … weight of a classroom preference of a placement,
- W_{constr} … weight of a preference of a satisfied soft group constraint,
- $W_{instrdist}$ … weight of a distance instructor preference of a placement (as described in chapter 3.3.2, it is discouraged if there are two subsequent classes taught by the same instructor but placed in different buildings not farther than 50 meters, strongly discouraged if the buildings are more than 50 meters but less than 200 meters far)
- $W_{deptbal}$ … weight of the overall department balancing penalty (number of the time units used over initial allowances summed over all times and departments, see chapter 3.3.2 for details)

- $W_{uslhour}$ … weight of a useless half-hour (empty half-hour time segments between classes, such half-hours cannot be used since all events require at least one hour)
- $W_{bigroom}$ … weight of a too large classroom ($W_{bigroom}$ for each classroom that has more than 50% excess seats)

Note that preferences of all time, classroom and group soft constraints go from -2 (strongly preferred) to 2 (strongly discouraged). So, for instance, the value of the weighted sum is increased when there is a discouraged time or room selected or a discouraged group constraint satisfied. Therefore, if there are two solutions, the better solution of them has the lower weighted sum of the above criteria.

The termination condition stops the search when the solution is complete and good enough (expressed by the solution quality described above and, in case of minimal perturbation problem also by the number of allowed perturbations). It also allows for the solver to be stopped by the user. Characteristics of the current and the best achieved solution, describing the number of assigned variables, time and classroom preferences, the total number of student conflicts, etc., are visible to the user during the search.

The solution comparator prefers a more complete solution (with a smaller number of unassigned variables). In case of minimal perturbation problem, a solution with a smaller number of perturbations among solutions with the same number of unassigned variables is preferred. If both solutions have the same number of unassigned variables (and perturbations), the solution of better quality is selected.

If there are one or more variables unassigned, the variable selection criterion picks one of them randomly. We have tried several approaches using domain sizes, number of previous assignments, numbers of constraints in which the variable participates, etc., but there was no significant improvement in this timetabling problem towards the random selection of an unassigned variable. The reason is, that it is easy to go back when a wrong variable is picked - such a variable is unassigned when there is a conflict with it in some of the subsequent iterations.

When all variables are assigned, an evaluation is made for each variable according to the above described weights. The variable with the worst evaluation is selected. This variable promises the best improvement in optimisation.

We have implemented a hierarchical handling of the value selection criteria. There are three levels of comparison. At each level a weighted sum of the criteria described below is computed. Only solutions with the smallest sum are considered in the next level. The weights express how quickly a complete solution should be found. Only hard constraints are satisfied in the first level sum. Distance from the initial solution (MPP), and a weighting of major preferences (including time, classroom requirements and student conflicts), are considered in the next level. In the third level, other minor criteria are considered. In general, a criterion can be used in more than one level, e.g., with different weights.

The above sums order the values lexicographically: the best value having the smallest first level sum, the smallest second level sum among values with the smallest first level sum, and the smallest third level sum among these values. As mentioned above, this allows diversification between the importance of individual criteria.

Furthermore, the value selection heuristics also support some limits (e.g., that all values with a first level sum smaller than a given percentage P_{th} above the best value [typically 10%] will go to the second level comparison and so on). This allows for the continued feasibility of a value near to the best that may yet be much better in the next level of comparison. If there is more than one solution after these three levels of comparison, one is selected randomly. This approach helped us to significantly improve the quality of the resultant solutions.

In general, there can be more than three levels of these weighted sums, however three of them seem to be sufficient for spreading weights of various criteria for our problem.

The value selection heuristics also allow for random selection of a value with a given probability P_{rw} (random walk, e.g., 2%) and, in the case of MPP, to select the initial value (if it exists) with a given probability P_{init} (e.g., 70%).

Criteria used in the value selection heuristics can be divided into two sets. Criteria in the first set are intended to generate a complete assignment:

- Number of hard conflicts, weighted by $V_{conf,1}$ in the first level, $V_{conf,2}$ in the second level and $V_{conf,3}$ in the third level.
- Number of hard conflicts, weighted by their previous occurrences (see section 5.1 about conflict-based statistics) and by $V_{wconf,1..3}$.

Additional criteria allow better results to be achieved during optimisation:

- Number of student conflicts caused by the value if it is assigned to the variable, weighted by $V_{student,1..3}$.
- Soft time conflicts caused by a value if it is assigned to the variable, weighted by $V_{time,1..3}$.
- Soft classroom conflicts caused by a value if it is assigned to the variable (combination of the placement's building, room, and classroom equipment compared with preferences), weighted by $V_{room,1..3}$.
- Preferences of satisfied soft group constraints caused by the value if it is assigned to the variable, weighted by $V_{constr,1..3}$.
- Difference in the number of assigned initial values if the value is assigned to the variable (weighted by $V_{\Delta init,1..3}$): -1 if the value is initial, 0 otherwise, increased by the number of initial values assigned to variables with hard conflicts with the value.
- Distance instructor conflicts caused by a value if it is assigned to the variable (together with the neighbour classes), weighted by $V_{instrdist,1..3}$.
- Difference in department balancing penalty, weighted $V_{deptbal,1..3}$.

- Difference in the number of useless half-hours (number of empty half-hour time segments between classes that arise, minus those which disappear if the value is selected), weighted $V_{uslhour,1..3}$.
- Classroom is too big: 1 if the selected classroom has more than 50% excess seats, weighted by $V_{bigroom,1..3}$.

Let us emphasize that the criteria from the second group are needed for optimisation only, i.e., they are not needed to find a feasible solution. Furthermore, assigning a different weight to a particular criteria influences the value of the corresponding objective function (e.g., see Figure 6.23 with comparison for optimisation criteria $V_{student,1..3}$ and $V_{time,1..3}$). The solver returns good results in reasonable time (e.g., in 30 minutes time limit) when the total sum of the weights used in additional criteria in the first level corresponds to one half of the weight $V_{wconf,1}$. The weights in the second level usually correspond to the weights used for the solution quality comparison ($W_{student}$, W_{time}, W_{room}, W_{constr} , $W_{instrdist}$, $W_{deptbal}$, $W_{uslhour}$ and $W_{bigroom}$).

Below, we present two types of experiments. The first investigates finding an initial solution (e.g., when all requirements are placed in the system). This is followed by experiments on the minimal perturbation problem (e.g., where there is an existing solution plus a set of changes to be applied to it). Solving an initial problem can be seen as a special case of MPP where all variables are new and therefore have no initial values.

If not stated otherwise, the solution quality weights $W_{student}$, W_{time}, W_{room}, W_{constr}, $W_{instrdist}$, $W_{deptbal}$, $W_{uslhour}$ and $W_{bigroom}$ in the solution quality weighted sum are set to zero in the following experiments. First level weight for the weighted hard conflicts $V_{wconf,1}$ is set to 1, all other weights in the value selection criterion are set to zero. Also, there is no random value selection (P_{rw}=0) and there is a 10% threshold limit (P_{th}=0.1) between levels. This way, by default, only the hard constraints are considered during the search. We will show how the other weights influence the search process and the overall solution quality. Also, if not stated otherwise, distances between buildings are not considered and department balancing is not used. The results presented in the following chapters were computed on 1GHz Pentium 3 PC running Windows 2000, with 512 MB RAM and JDK 1.4.2.

6.3.1. Initial Problem

Figures 6.23 and 6.24 show the computational results for 8 independent experiments. *Time* refers to the amount of time required by the solver to find the presented solution. *Satisfied enrolments* gives the percentage of satisfied requirements for courses chosen by students. *Preferred time* and *preferred room* correspond to the satisfaction of time and room preferences respectively. 100% corresponds to a case when all classes are placed in their most preferred times or rooms, 0% means a case when the least preferred locations are used. *Useless half-hours* gives the percentage of empty half-hour time segments between classes to all empty half-hours. *Too big rooms* gives the usage of classrooms that have more than 50% excess seats (given percentage of all assigned classes is placed into such big classrooms). *Instructor preferences* corresponds to the satisfaction of distance instructor preferences (100% means that there is no discourage or strongly discouraged case, 0% means that all two subsequent classes taught by the same instructor are strongly discouraged because of the distance). And finally, *Dept. balancing penalty* presents the overall department balancing penalty. Zero in such case means that there is no class placed over the initial balancing allowances. Preferences of soft group constraints are not presented, since there are no such constraints in the Fall 2004 data set (all group constraints are either required or prohibited).

Test case	No preference	Students	Time	Room
Assigned variables [%]	100.00 ± 0.00	100.00 ± 0.00	100.00 ± 0.00	100.00 ± 0.00
Time [min]	0.16 ± 0.03	8.45 ± 4.40	18.68 ± 6.50	0.17 ± 0.01
Satisfied enrolments [%]	98.26 ± 0.15	**99.74 ± 0.02**	98.20 ± 0.13	98.18 ± 0.24
Preferred time [%]	62.54 ± 1.19	65.33 ± 1.45	**98.75 ± 0.13**	62.14 ± 0.94
Preferred room [%]	63.64 ± 2.29	62.60 ± 1.66	62.82 ± 2.07	**98.58 ± 0.29**
Useless half-hours [%]	1.64 ± 0.23	1.64 ± 0.16	1.42 ± 0.14	1.66 ± 0.19
Too big rooms [%]	27.20 ± 1.06	25.31 ± 0.59	23.77 ± 0.53	26.76 ± 0.96

Fig. 6.23. Solutions of the initial problem (no preferences, students, time or room optimised)

Test case	Useless half-hours	Too big rooms	Distance instructors	Distance students
Assigned variables [%]	100.00 ± 0.00	100.00 ± 0.00	100.00 ± 0.00	100.00 ± 0.00
Time [min]	19.14 ± 8.26	0.05 ± 0.01	0.18 ± 0.04	7.72 ± 7.54
Satisfied enrolments [%]	98.04 ± 0.15	98.19 ± 0.08	97.46 ± 0.26	**99.53 ± 0.02**
Preferred time [%]	60.69 ± 2.75	61.43 ± 0.87	61.80 ± 0.85	66.44 ± 1.89
Preferred room [%]	60.09 ± 1.88	65.69 ± 1.88	64.67 ± 1.96	64.03 ± 0.97
Useless half-hours [%]	**0.84 ± 0.15**	1.74 ± 0.25	1.55 ± 0.17	1.83 ± 0.14
Too big rooms [%]	25.63 ± 0.69	**17.21 ± 0.18**	27.77 ± 0.73	24.64 ± 0.47
Instructor preferences [%]	- -	- -	**100.00 ± 0.00**	90.34 ± 3.92

Fig. 6.24. Solutions of the initial problem (useless half-hours, too big rooms or distances either for instructors or students are optimised)

A complete solution was found on every run of all experiments. Average values together with their RMS (root-mean-square) variances of the best achieved solutions from 10 different runs found within 30 minute time limit are presented.

The experiment marked *No preference* presents average solutions obtained without any preferences on the soft constraints. All solution quality weights W and value selection weights V are set to zero, except of the weight $V_{wconf,1}$=1 (weight of the weighted hard conflicts in the first level of the value selection).

The following five experiments marked *Students, Time*, *Rooms*, *Useless half-hours* and *Too big rooms* are minimizing just one of the criteria: the student conflicts, violated time preferences, violated room preferences, the number of empty half-hour segments between classes and the usage of classrooms that have more than 50% excess seats. *Students* experiment uses the same weights as *No preference* experiment, but student weights are the following: $V_{student,1}$=0.5, $V_{student,2}$=$W_{student}$=1. Similarly, *Time* experiment uses weights $V_{time,1}$=0.5, $V_{time,2}$=W_{time}=1, *Rooms* experiment weights $V_{room,1}$=0.5, $V_{room,2}$=W_{room}=1, *Useless half-hours* experiment weights $V_{uslhour,1}$=0.5, $V_{uslhour,2}$=$W_{uslhour}$=1 and *Too big rooms* experiment weights $V_{bigroom,1}$=0.5, $V_{bigroom,2}$=$W_{bigroom}$=1.

The last two experiments of Figure 6.24 marked *Distance instructors* and *Distance students* are minimizing either overall distance instructor conflicts or student conflicts. In both cases distances between buildings are considered. Recall that as for students, if the distance between two following classes is more than 670 meters, the joint enrolments of such classes are considered as student conflicts (it is not possible for a student to attend both classes, same as when these classes are overlapping in time). Experiment *Distance Students* uses the same weights as the experiment *Students*, there is around 0.21% of additional student conflicts caused by the distances of the buildings (note that distances are not considered in the experiment *Student*). This represents about 183 student conflicts. *Distance instructors* experiment uses the same weights as *No preference* experiment, but distance instructor preference weights are the following: $V_{instrdist,1}$=0.5, $V_{instrdist,2}$=$W_{instrdist}$=1. There is no case of two subsequent classes taught by the same instructor but placed in two different buildings in this case.

Test case	All, but no distance, no dept. bal.	All, but no distance	All, but no dept. bal.	All
Assigned variables [%]	100.00 ± 0.00	100.00 ± 0.00	100.00 ± 0.00	100.00 ± 0.00
Time [min]	14.61 ± 4.61	20.36 ± 5.16	13.77 ± 3.89	21.48 ± 4.82
Satisfied enrolments [%]	99.59 ± 0.02	99.52 ± 0.04	99.31 ± 0.03	99.26 ± 0.03
Preferred time [%]	95.04 ± 0.34	92.21 ± 0.39	94.61 ± 0.29	91.85 ± 0.28
Preferred room [%]	74.95 ± 2.43	75.79 ± 2.60	72.80 ± 2.73	72.31 ± 1.90
Useless half-hours [%]	1.40 ± 0.22	1.52 ± 0.19	1.48 ± 0.24	1.47 ± 0.10
Too big rooms [%]	22.43 ± 0.51	22.55 ± 0.78	22.37 ± 0.55	22.59 ± 0.69
Instructor preferences[%]	- -	- -	95.80 ± 1.32	95.74 ± 2.17
Dept. balancing penalty	- -	3.38 ± 1.92	- -	5.00 ± 1.07

Fig. 6.25. Solutions of the initial problem (all preferences constraints optimised)

Figure 6.25 shows the results from another 4 experiments, now with all soft constraints enabled. Again, a complete solution was found on every run of all experiments and average values together with their RMS variances of the best achieved solutions from 10 different runs found within 30 minute time limit are presented.

The experiment marked *All, but no distance, no dept. bal.* combines all the above weights. Student conflicts and time preferences are weighted equally, room preferences are considered much less important. Useless half-hours and too big rooms are considered as very minor criteria. The weights are as follows:

- Student conflicts: $W_{student}$=1, $V_{student,1}$=0.2, $V_{student,2}$=1
- Time preferences: W_{time}=1, $V_{time,1}$=0.2, $V_{time,2}$=1
- Room preferences: W_{room}=0.2, $V_{room,1}$=0.0, $V_{room,2}$=0.2
- Useless half-hours: $W_{uslhour}$=0.05, $V_{uslhour,1}$=0.0, $V_{uslhour,2}$=0.05
- Too big rooms: $W_{bigroom}$=0.05, $V_{bigroom,1}$=0.0, $V_{bigroom,2}$=0.05

The *All, but no distance* experiment comes out of the previous experiment, but with department balancing enabled. Department balancing weights are $W_{deptbal}$=0.5, $V_{deptbal,1}$=0.1, $V_{deptbal,2}$=0.5. The initial balancing allowance was computed from the maximal fill factor as described in chapter 3.3.2 (increased by 20% and rounded upwards). Accomplishment of time preferences is worse than in the previous experiment, but the departmental balancing is much better and the resultant solutions are more acceptable from the users point of view. Without department balancing, the overall balancing penalty is about 300 which is unacceptable.

The *All, but no dept. bal* experiment comes also out of the experiment *All, but no distance, no dept. bal.*, but now the distances between buildings are considered. Distance instructor preference weights are set as follows: $W_{instrdist}$=1.0, $V_{instrdist,1}$=0.2, $V_{instrdist,2}$=1.0.

The last test from Figure 6.25 (marked *All*) most closely corresponds to reality. Here all the soft preferences are considered. Also, both distances between buildings and departmental balancing features are used. The weights are combined from the previous tests (same as in *All, but no distance, no dept. bal.* experiment, plus department balancing weights as in *All, but no distance* experiment and distance instructor preferences as in *All, but no dept. bal* experiment). Such solutions were very well accepted by the schedule representatives at Purdue University.

Test case	No CBS
Assigned variables [%]	98.42 ± 0.20
Time [min]	24.08 ± 4.42
Satisfied enrolments [%]	99.52 ± 0.06
Preferred time [%]	94.62 ± 0.43
Preferred room [%]	83.77 ± 1.49
Useless half-hours [%]	1.48 ± 0.27
Too big rooms [%]	22.78 ± 0.57

Fig. 6.26. Solutions of the initial problem (without conflict-based statistics).

Finally, the last experiment (Figure 6.26, marked *No CBS*) presents average solutions obtained from the solver without conflict-based statistics. The weights on soft constraints are the same as in the previous experiment (marked *All, but no distance no dept. bal.*). But there is $V_{conf,1}$=1 (weight of a hard conflict) instead of $V_{wconf,1}$=1 (weight of a hard conflict weighted by CBS). $V_{wconf,1}$ is set to zero. The solver was not able to find a complete solution within the given 30 minute time limit, not even when 2% random walk selection was used P_{rw}=0.02 to avoid cycling. Furthermore, there were at least 5 unassigned classes after 3 hours of run. On the Purdue Timetabling Problem, conflict-based statistics proved itself not only as a technique which can improve the solution quality, but as a technique which can help us to find a complete feasible solution.

IFS MAC was able to assign only about 65% of variables. IFS MAC+ assigned about 94% variables. Consistency was maintained over all hard constraints. We plan to use MAC+ only over the group constraints, e.g. a precedence constraint between two or more courses or not-overlap constraint between a lecture and its seminars. However, the used data set contains only 201 of such constraints, so there is no significant difference between a solution with and without MAC+ (IFS CBS versus IFS CBS with MAC+ on group constraints).

Figure 6.27 compares several experiments giving different stress on student conflicts and time preferences. Average values from the best solutions of 10 different runs found within 30 minute time limit are presented.

Only student conflicts or time preferences are considered in the border experiments marked *students* and *time* respectively. In the middle (experiment marked 1:1), student conflicts and time preferences are equally weighted. The experiment marked 3:1 prefers student conflicts three times as much as time preferences (i.e., weights of student conflicts are three times higher than weights of time preferences) and vice versa. For instance, the experiment marked 1:2 has the following weights: $V_{wconf,1}$=1, $V_{student,1}$=0.2, $V_{time,1}$=0.4, $V_{student,2}$=$W_{student}$=1, $V_{time,2}$=W_{time}=2.

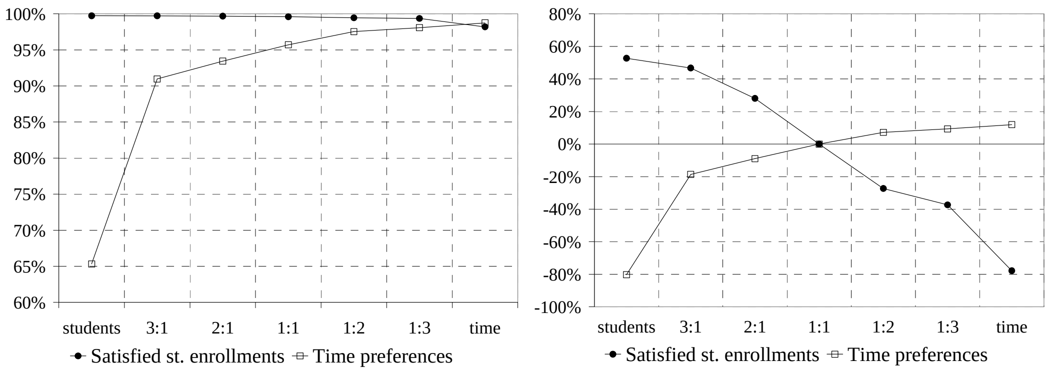


Fig. 6.27. Comparison of satisfied student enrolments and time preferences: average quality of the solution (left), improvement of the solution in terms of percentage of the 1:1 solution (right).

Comparison with manual solution

Figure 6.28 presents a comparison of solutions which were generated by the IFS CBS solver with the manual solution made at Purdue University for the semester Fall 2005. As for IFS CBS, a complete solution was found on every run. Average values together with their RMS (root-mean-square) variances of the best achieved solutions from 10 different runs found within 30 minute time limit are presented. The same settings were applied as in the test marked *All* from Figure 6.28.

Test case	IFS CBS	Manual
Assigned variables [%]	100.00 ± 0.00	100.00
Time [min]	12.01 ± 3.77	a week
Satisfied enrolments [%]	99.39 ± 0.01	98.20
Preferred time [%]	92.69 ± 0.34	89.02
Preferred room [%]	75.27 ± 1.42	83.04
Useless half-hours [%]	3.46 ± 0.63	4.11
Too big rooms [%]	20.78 ± 0.45	20.92
Instructor preferences [%]	97.29 ± 1.15	94.71
Dept. balancing penalty	7.60 ± 5.02	311

Fig. 6.28. Comparison of a solution generated by IFS CBS and a manually created solution for Fall 2005.

The solutions generated by IFS CBS are better than the manual solution in many aspects. Moreover, using different weights for optimisation criteria, it can be tuned quite well.

6.3.2. Minimal Perturbation Problem

The following experiments were conducted on one of the complete initial solutions computed in the previous set of experiments (column marked *All, but no dept. bal* in Figure 6.28). Input perturbations were generated such that a given number of randomly selected variables were not allowed to retain the values they were assigned in the initial solution. Therefore, these classes can not be scheduled to the same placement as in the initial solution (either room or starting time must be different). Only variables with more than one value in their domains were used. For each number of input perturbations, ten different sets of input perturbations (i.e., variables with initial values prohibited) were generated. The following figures show the average parameter values of the best solutions found within 10 minutes.

The aim of the first set of experiments is to find a suitable setting for P_{init} (probability of selection of an initial value) and $V_{\Delta init,1..3}$ (difference in the number of assigned initial values). In each experiment, we have executed 10 tests for each of 10, 20, 30, ... 100 input perturbations respectively (100 runs in total). The average numbers of assigned variables together with the average numbers of additional perturbations are presented in Figure 6.29. One or a combination of the criteria is used in each experiment. The second column refers to the set of criteria described in Figure 6.30.

Test case		Assigned variables [%]	Number of perturbations
P_{init}	Δinit		
0.5	0	100.00	13.83
0.6	0	99.98	13.48
0.7	0	99.96	13.33
0.8	0	99.95	12.94
0	2	100.00	31.40
0.6	2	99.99	13.26
0	1	100.00	13.70
0.6	**1**	**100.00**	**11.90**

Fig. 6.29. Comparison of several approaches to MPP.

Δinit	$V_{\Delta init,i}$	$V_{students,s}$	$V_{time,t}$	$V_{room,r}$
0	-	$0.25_{s=1}$, $1.0_{s=2}$	$0.25_{t=1}$, $1.0_{t=2}$	$0.2_{r=2}$
1	$0.5_{i=1}$	$1.0_{s=2}$	$1.0_{t=2}$	$0.2_{r=2}$
2	$1.0_{i=2}$	$0.25_{s=1}$, $1.0_{s=3}$	$0.25_{t=1}$, $1.0_{t=3}$	$0.2_{r=3}$

Fig. 6.30. Meaning of Δinit

Let us explain the contents of this table. For instance, the expression $0.25_{s=1}$, $1.0_{s=2}$ in the column marked $V_{students,s}$ means that $V_{students,1}$ is set to 0.25 and $V_{students,2}$ is set to 1. The first case (Δinit=0) corresponds to the settings of the *All, but no dept. bal* experiment. In remaining Δinit sets, we tried to decrease the importance of other value selection criteria in comparison with the $V_{\Delta init}$ criterion. For Δinit=1, the first level value selection criterion $V_{\Delta init,1}$ is used and the other optimisation criteria which were placed in the first level are disabled ($V_{student,1}$, $V_{time,1}$ are set to zero). And the third line Δinit=2 corresponds to a case when the second level value selection criterion $V_{\Delta init,2}$ is used and the other optimisation criteria from the second level ($V_{student,2}$, $V_{time,2}$, $V_{room,2}$) are moved to the third level.

Let us discuss particular experiments from Figure 6.29. In the first four experiments (marked P_{init}=0.5, ..., P_{init}=0.8), the minimal perturbation problem was solved only by changing the value selection criteria so that it selected an initial value with a given probability (50%, 60%, 70% and 80% respectively). Otherwise, it worked exactly as *All, but no dept. bal* experiment, since all the other weights were the same. As the P_{init} probability is rising, we can see that the average number of additional perturbations is descending, but the algorithm is loosing the ability to find a complete solution in every run (in the given 10 minute time limit).

Similarly, we can see that using just the second level value selection criterion $V_{\Delta init,2}$ is able to find a complete solution all the time, but the average number of additional perturbations is too high. A combination with the 60% probability of an initial value selection helps to improve the average number of additional perturbations, but again, there were some cases where a complete solution was not found.

Using the first level value selection criteria $V_{\Delta init,1}$ seems to be very promising. With this criterion, we were able to find a complete solution to all the presented experiments. Moreover, the experiment marked P_{init}=0.6, Δinit=1 (combining $V_{\Delta init,1}$ with 60% initial value selection probability) gave us the best results from the above experiments, since the average number of additional perturbations was the lowest. The following results (Figures 6.31 and 6.32) were computed using the weights from this experiment.

Figure 6.31 presents the average number of additional perturbations (variables that were not assigned to their initial value though not prohibited). Additional perturbations are presented wrt. the absolute number of input perturbation (i.e., up to about 13.4% of input perturbations is considered). The best solution found within 10 minutes from each experiment is taken into account. The number of additional perturbations grows with the number of input perturbations.

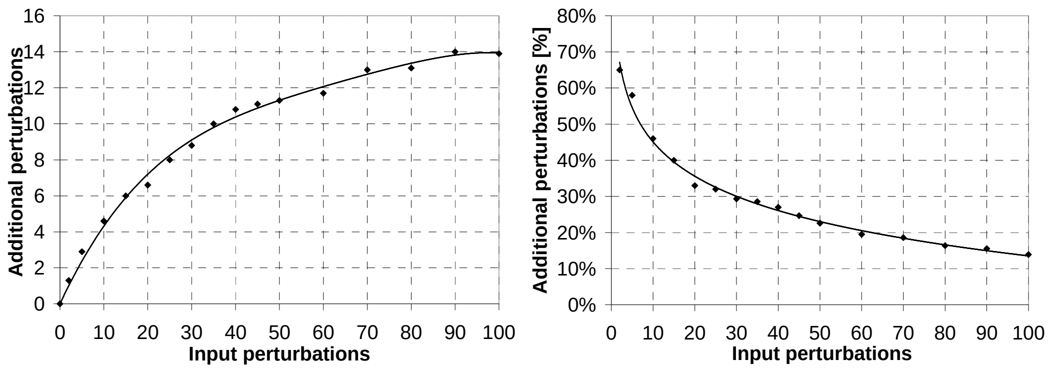


Fig. 6.31. Absolute number of average additional perturbations (left) and average additional perturbations in terms of percentage of the number of input perturbations (right).

The graph on Figure 6.32 (left) shows the average quality of the resulting solutions in the same manner as presented in Figure 6.25. Because the initial solution is (at least locally) optimal, and because the number of additional perturbations is the primary minimization criteria, it is not surprising that the quality of the solution declines with an increasing number of input perturbations. The weighting between time preferences, student conflicts, and other parameters considered in the optimisation can have a similar influence as seen in the initial solutions.

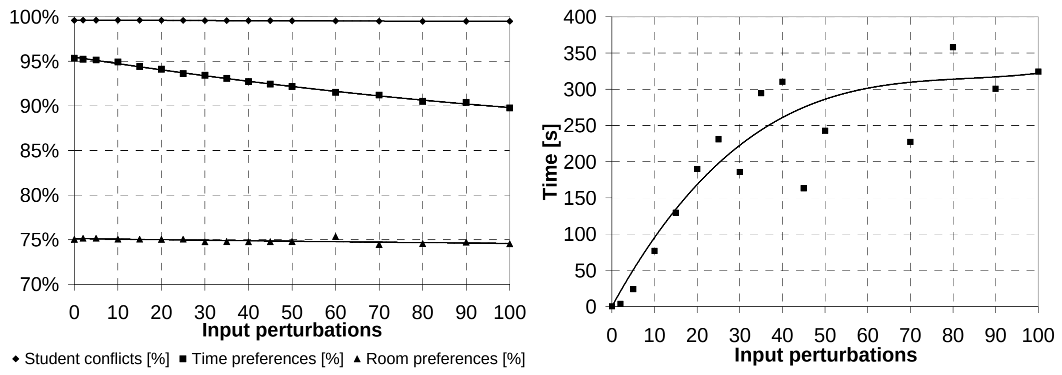


Fig. 6.32. Average solution quality (left), average time (rigt).

Finally, the graph in Figure 6.32 (right) presents the average time needed to find the best solution. Note that a 10 minutes time limit for finding the best solution was set. The influence of this limit is seen mostly on the right portion of the chart, where the number of input perturbations exceeds 50.

Perturbations in practise

In practise, the strategy for computing perturbations needs to be extended. For example, a change in time is usually much worse than a movement to a different classroom. The number of enrolled/involved students should also be taken into account. Another factor is whether the solution has already been published or not.

The priorities for evaluating perturbations are as follows. Before publishing timetable:

- minimize number of classes with time changes,
- minimize number of student conflicts,
- optimise satisfaction of problem soft constraints.

After publishing the timetable:

- minimize number of additional (new) student conflicts,
- minimize number of students with time changes,
- minimize number of classes with time changes,
- optimise satisfaction of problem soft constraints.

In both cases, the number of classes with room change is not significant at all. Before the timetable is published, minimizing the number of classes with time changes is the most important criteria for the MPP as long as it does not create too many additional student conflicts in the process. Therefore, as a compromise, the cost (in equivalent conflicts) of changing the time assigned to a class equals a number like 5% of the students enrolled in that class. Otherwise none of our other criteria would have any importance.

Similar properties apply between other criteria as well. To fulfil all these needs we have created a function (called perturbations penalty, see Definition 2.19) which can be computed over a partial solution. This is a weighted sum of various perturbations criteria like the number of classes with time changes or the number of additional student conflicts. This perturbation penalty is added as an extra optimisation criterion to the solution comparator and to value selection criterion, so we can also setup the weights between this perturbation penalty and other (initial) soft constraints.

6.3.3. Summary

We have proposed and implemented a solution to a large scale university timetabling problem. Our proposal includes a new iterative forward search algorithm that is extended by conflict-based statistics which can be generalized to other search algorithms. Both ideas combined together suffice to solve the problem and the role of additional heuristics can be minimized. Our problem solver is able to construct a demand-driven timetable as well as incorporate dynamic aspects. The initial solution generated by our solver satisfies the course requests of more than 99% of students together with about 95% of time requirements. The automated search was able to find suitable times and classrooms for all classes. The experiments with a MPP give us very promising results as well. Within 10 minutes, the solver was able to find a complete, high quality solution with a small number of additional perturbations.

Moreover, the used heuristics can be tuned to maximally fulfil the user requirements, e.g., when there is a need of a trade-off between several objective functions. We have demonstrated this, for instance, in the experiment giving different stress on the satisfied student enrolments and time preferences for Purdue University timetabling problem (see figure 6.27).

6.4. Summary

In this chapter, we demonstrated several properties of iterative forward search algorithm on a set of various constraint satisfaction and optimisation problems. On random binary CSP and RPP, we presented its capabilities of solving “normal” as well as over-constrained problems in comparison with DBT and LS algorithms. On the weighted random binary CSP we can clearly see that the presented conflict-based statistics together with either IFS or LS can perform very well on optimisation problems. On RPP and Purdue University timetabling problem we presented the capabilities of IFS to solve minimal perturbation (optimisation) problems. Finally, on Purdue University timetabling problem we presented applicability of IFS to a real-life large scale optimisation problem.

We believe that the presented iterative forward search algorithm, together with the presented extensions (mainly conflict-based statistics), can be used for many other real-life constraint satisfaction and optimisation problems. We hope that the presented properties are valid for such problems as well.

7. Conclusion

In this thesis, we have presented an iterative forward search algorithm which is capable of solving various timetabling as well as general constraint satisfaction and optimisation problems. It is based on local search, but it works with partial feasible solutions, so it is capable of returning a (partial) solution any time during the search. This might be a very important feature, especially if the algorithm is used in an interactive manner. It can start from any (partial) solution and it can be used for both initial and minimal perturbation problem. We have also presented various extensions of this algorithm which can improve the quality of the returned solutions as well as applicability of the algorithm on various problems.

Also, the presented algorithm works well on the real-life large scale course timetabling problem at Purdue University. The generated solutions were very well accepted on Purdue University and they are going to use this solver in practice as of semester Spring 2006. Moreover, we are going to extend this solver to be used not only for the generation of the central timetable but also for all the departmental timetabling problems. These problems are of different structure and also there are some other constraints which need to be implemented.

The major contributions of this work are: We have defined a minimal perturbation (optimisation) problem. This definition is applicable on various dynamic problems where the task is to find a solution of a modified problem that is as near as possible to the solution of the original problem. Next, we have developed the iterative forward search algorithm which is capable as we believe of solving various constraint satisfaction and optimisation problems as well as minimal perturbation (optimisation) problems. We have also presented the conflict-based statistics which can be used in the framework of IFS or a local search algorithm and we have shown that it could dramatically improve the results especially when solving optimisation problems. Finally, we were able to solve Purdue university large lecture room timetabling problem and we are going to continue using the presented approaches for the departmental problems as well. Also, we have published four data sets (from four different semesters, also present on the attached CD-ROM) of Purdue timetabling problem in a clear, anonymous form which can be used as an interesting timetabling benchmark.

8. Bibliography

AM99 S. Abdennadher and M. Marte. *University timetabling using constraint handling rules.* Journal of Applied Artificial Intelligence, Special Issue on Constraint Handling Rules, 1999.

Bar00 R. Barták. *Dynamic Constraint Models for Planning and Scheduling Problems.* In New Trends in Constraints, LNAI 1865, pp. 237-255, Springer, 2000.

Bes91 Christian Bessière. *Arc-consistency in dynamic constraint satisfaction problems.* In AAAI-91, pages 221-226, Anaheim CA, 1991

Bes94 C. Bessiére. *Arc-consistency and arc-consistency again.* Artificial Intelligence, 65(1):179–190, 1994.

Bes96 Christian Bessière. *Random uniform csp generators*, 1996. http://www.lirmm.fr/~bessiere/generator.html.

BF99 C. Bessiére and E. C. Freuder. *Using constraint metaknowledge to reduce arc consistency computation.* Artificial Intelligence, 107(1):125–148, 1999.

BKJW97 E. Burke, J. Kingston, K. Jackson, R. Weare. *Automated University Timetabling: The State of the Art*. The Computer Journal 40 (9) 565-571, 1997.

BMR03 Roman Barták, Tomáš Müller, and Hana Rudová. *Minimal Perturbation Problem – A Formal View.* Neural Network World (2003), vol. 13, no. 5, p. 501-511.

BMR04 Roman Barták, Tomáš Müller, and Hana Rudová. *A new approach to modelling and solving minimal perturbation problems.* In Recent Advances in Constraints, pages 233–249. Springer Verlag LNAI 3010, 2004.

BPS99 S. C. Brailsford, C. N. Potts, B. M. Smith. *Constraint Satisfaction Problems: Algorithms and Applications*. European Journal of Operational Research 119 557-581, 1999.

BR97 C. Bessière and J. C. Régin. *Arc consistency for general constraint networks: Preliminary results.* In Proceedings of 15th International Joint Conference on Artificial Intelligence (IJCAI-97), pages 398–404, Nagoya, Japan, 1997.

BR01 C. Bessière and J. C. Régin. *Refining the basic constraint propagation algorithm.* In Proceedings IJCAI'01, pages 309–315, Seattle WA, 2001.

Car86 M. W. Carter. *A Survey of Practical Applications of Examination Timetabling Algorithms.* Operations Research 34 193-202, 1986.

Car00 Michael W. Carter. *A comprihensive course timetabling and student scheduling system at the University of Waterloo.* In Edmund Burke and Wilhelm Erben, editors, PATAT 2000 - Proceedings of the 3rd international conference on the Practice And Theory of Automated Timetabling, pages 64–82, 2000.

CDJD04 Hadrien Cambazard, Fabien Demazeau, Narendra Jussien, and Philippe David. *Interactively solving school timetabling problems using extensions of constraint programming.* In Edmund K. Burke and Michael Trick, editors, PATAT 2004 - Proceedings of the 5th International Conference on the Practice and Theory of Automated Timetabling, pages 107–124, 2004.

CK96 T. B. Cooper and J.H. Kingston. *The Complexity of Timetable Construction Problems.* In the Practice and Theory of Automated Timetabling, ed. E.K. Burke and P. Ross, pp. 283-295, Springer-Verlag, 1996.

COC97 Mats Carlsson, Greger Ottosson, and Bjorn Carlson, *An open-ended finite domain constraint solver,* In Programming Languages: Implementations, Logics, and Programming. Springer-Verlag LNCS 1292, 1997.

Cra96 J. M. Crawford. *An approach to resource constrained project scheduling.* In Artificial Intelligence and Manufacturing Research Workshop, 1996.

Deb96 R. Debruyne. *Arc-consistency in dynamic CSPs is no more prohibitive.* In Proceedings of 8th Conference on Tools with Artificial Intelligence (TAI'96), pages 299–306, 1996.

Dech03 Rina Dechter. *Constraint Processing.* Morgan Kaufmann Publishers, 2003.

DF02 Rina Dechter and Daniel Frost. *Backjump-based backtracking for constraint satisfaction problems.* Artificial Intelligence, 136(2):147–188, 2002.

EGJ03 Abdallah Elkhyari, Christelle Guéret, and Narendra Jussien. *Solving dynamic timetabling problems as dynamic resource constrained project scheduling problems using new constraint programming tools.* In Edmund Burke and Patrick De Causmaecker, editors, Practice And Theory of Automated Timetabling, pages 39–59. Springer-Verlag LNCS 2740, 2003.

FW92 Freuder, E.C., Wallace R.J., *Partial Constraint Satisfaction*, Artificial Intelligence, 58:21-70, 1992.

GH97 Philippe Galinier and Jin-Kao Hao. *Tabu search for maximal constraint satisfaction problems.* In Proceedings 3rd International Conference on Principles and Practice of Constraint Programming, pages 196–208. Springer-Verlag LNCS 1330, 1997.

Gin93 Matthew L. Ginsberg. *Dynamic backtracking.* Journal of Artificial Intelligence Research, pages 23–46, 1993.

HG95 William D. Harvey and Matthew L. Ginsberg. *Limited discrepancy search.* In Chris S. Mellish, editor, Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, pages 607–615. Morgan Kaufmann, 1995.

Ian04 Ian Miguel. *Dynamic Flexible Constraint Satisfaction and its Application to AI Planning.* Springer, 2004.

JDB00 Narendra Jussien, Romuald Debruyne, and Patrice Boizumault. *Maintaining arcconsistency within dynamic backtracking.* In Principles and Practice of Constraint Programming, pages 249-261, 2000.

JL02 Narendra Jussien and Olivier Lhomme. *Local search with constraint propagation and conflict-based heuristics.* Artificial Intelligence, 139(1):21–45, 2002.

Jus03 Narendra Jussien. *The versatility of using explanations within constraint programming.* In Habilitation thesis of Universit de Nantes, France, 2003.

KGV83 S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. *Optimisation by simulated annealing.* Science, Number 4598, 13 May 1983, 220, 4598:671–680, 1983

Koc02 Waldemar Kocjan. *Dynamic scheduling: State of the art report.* Technical Report T2002:28, SICS, 2002.

Mac77 A. K. Mackworth. *Consistency in Networks of Relations.* Artificial Intelligence, 8:99–118, 1977.

MB01 T. Müller and R. Barták. *Interactive Timetabling*. In Proceedings of the ERCIM Workshop on Constraints, Prague, June 2001

MB02 Tomáš Müller and Roman Barták. *Interactive Timetabling: Concepts, Techniques, and Practical Results.* In Burke, Edmund; Causmaecker, Patrick De (eds.): Proceedings of the 4th International Conference on the Practice and Theory of Automated Timetabling (PATAT2002), Gent, 2002, pp. 58-72.

MBR04 T. Müller, R. Barták, H. Rudová. *Iterative Forward Search: Combining Local Search with Maintaining Arc Consistency and a Conflict-based Statistics.* In LSCS'04 - International Workshop on Local Search Techniques in Constraint Satisfaction, 2004.

MF00 Zbigniew Michalewicz and David B. Fogel. *How to Solve It: Modern Heuristics.* Springer, 2000.

MH86 R. Mohr and T. C. Henderson. *Arc and path consistency revisited.* Artificial Intelligence, 28(2):225–233, 1986.

MJP92 Steven Minton, Mark D. Johnston, Andrew B. Philips, and Philip Laird. *Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems.* Artificial Intelligence, 58:161–205, 1992.

MPSW98 Ewan MaxIntyre, Patrick Prosser, Barbara Smith, and Toby Walsh. *Random Constraint Satisfaction: theory meets practice.* In Michael Maher and Jean-Francois Puget (eds.): Principles and Practice of Constraint Programming - CP98. Springer-Verlag LNCS 1520, pp. 325-339, 1998.

MR04 Tomáš Müller and Hana Rudová. *Minimal Perturbation Problem in Course Timetabling.* In PATAT 2004 - Proceedings of the 5th international conference on the Practice And Theory of Automated Timetabling, pages 283-303, 2004.

MRB05 T. Müller, H. Rudová, R. Barták *Minimal Perturbation Problem in Course Timetabling.* Practice And Theory of Automated Timetabling, Selected Revised Papers, 2005. To appear.

Mul01 T. Müller. *Interactive Timetabling***,** Master Thesis, KTIML MFF UK, Prague, September 2001

Mul02 T. Müller. *Interactive Heuristic Search Algorithm.* In Proceedings of the CP'02 Conference - Doctoral Programme, Ithaca, September 2002. Springer-Verlag LNCS 2470, pp. 765, 2002.

NB94 B. Neveu and P. Berlandier. *Maintaining arc consistency through constraint retraction.* In Proceedings of the IEEE International Conference on Tools for Artiocial Intelligence (TAI), pages 426–431, New Orleans, LA, 1994.

PG96 G. Pesant, M. Gendreau, *A view of local search in constraint programming,* In Proceedings of Principles and Practice of Constraint Programming, Springer, Berlin, 1996, pp. 353–366.

PMM04 Sylvain Piechowiak, Jingxua Ma, and René Mandiau. *EDT-2004: An open interactive timetabling tool.* In Edmund K. Burke and Michael Trick, editors, PATAT 2004 - Proceedings of the 5th International Conference on the Practice and Theory of Automated Timetabling, pages 305–321, 2004.

Pre00 S. D. Prestwich. *A Hybrid Search Architecture Applied to Hard Random 3-SAT and Low-Autocorrelation Binary Sequences.* Sixth International Conference on Principles and Practice of Constraint Programming, Lecture Notes in Computer Science vol. 1894, Springer-Verlag, 2000, pp. 337-352.

Pre04 S. D. Prestwich. *Exploiting Relaxation in Local Search*. In First International Workshop on Local Search Techniques in Constraint Satisfaction, pages 49-61, 2004

RM03 Hana Rudová and Keith Murray. *University course timetabling with soft constraints.* In Edmund Burke and Patrick De Causmaecker, editors, Practice And Theory of Automated Timetabling, Selected Revised Papers, pages 310–328. Springer-Verlag LNCS 2740, 2003.

RR98 E.T. Richards, B. Richards, *Non-systematic search and learning: An empirical study,* In Proceedings of Conference on Principles and Practice of Constraint Programming, Pisa, 1998, pp. 370–384.

RRH02 Yongping Ran, Nico Roos, and Jaap van den Herik. *Approaches to find a nearminimal change solution for dynamic CSPs.* In Fourth International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimisation Problems, pages 373–387, 2002.

Sch97 Andrea Schaerf. *Combining local search and look-ahead for scheduling and constraint satisfaction problems.* In Proceedings of 15th International Joint Conference on Artificial Intelligence (IJCAI-97), pages 1254–1259, Nagoya, Japan, 1997.

Sch99 A. Schaerf. *A survey of automated timetabling.* Artificial Intelligence Review 13(2), pages 87-127, 1999

SW00 Hani El Sakkout, Mark Wallace. *Probe backtrack search for minimal perturbation in dynamic scheduling.* CONSTRAINTS, 4(5):359–388, 2000.

Tri92 A. Tripathy. *Computerised decision aid for timetabling – A case analysis.* Discrete applied mathematics, 35 (3), pp. 313-323, 1992

Tsa93 E. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.

Ver03 Kamil Veřmiřovský. *Algorithms for Constraint Satisfaction Problems.* Master Thessis, Masaryk University, Brno 2003

VJ03 Gérard Verfaillie and Narendra Jussien. *Dynamic constraint solving, 2003.* A tutorial including commented bibliography presented at CP 2003.
See http://www.emn.fr/x-info/jussien/CP03tutorial/.

VR02 Kamil Veřmiřovský and Hana Rudová, *Limited Assignment Number Search Algorithm.* In Maria Bielikova (ed.): SOFSEM 2002 Student Research Forum, 2002, pp. 53-58.

Wal94 M. Wallace. *Applying constraints for scheduling.* In Constraint Programming, volume 131 of NATO ASI Series Advanced Science Institute Series, Springer, 1994.

WC97 G. M. White, P. W. Chan. *Towards the Construction of Optimal Examination Timetables*. INFOR 17 219-229, 1979.

Wer85 D. Werra. *An Introduction to Timetabling*. European Journal of Operation Research 19 (1985), 151-162.

Whi00 G. M. White. *Constrained Satisfaction, Not So Constrained Satisfaction and the Timetabling Problem*. A Plenary Talk in the Proceedings of the 3rd Int. Conf. on the Practice and Theory of Automated Timetabling, pp. 32-47, 2000.

Wren96 A. Wren. *Scheduling, Timetabling and Rostering – A Special Relationship?* In the Practice and Theory of Automated Timetabling, ed. E.K. Burke and P. Ross, pp. 46-75, Springer-Verlag, 1996.

ZY01 Y. Zhang and R. Yap. *Making AC-3 an Optimal Algorithm.* In Proceedings of IJCAI-01, 316-321, (2001).

Appendix A IFS Framework

The iterative forward search algorithm was implemented in Java. The implementation is very general and it can be easily reused for modelling and solving various problems. In this chapter, some implementation aspects of the iterative forward search framework are discussed. For more details, consult the implementation or API (JavaDoc) documentation on the attached CD-ROM.

The general implementation works with abstract classes describing variables, values and constraints. There is a class describing model, which is basically a container of available variables and constraints and a class describing solution which has the ability to store the best ever found solution. Next, there are some interfaces and general implementation of necessary heuristics, namely variable, value selections, a solution comparator and a termination condition. Finally, there is a solver which together with the given heuristics implements the iterative forward search algorithm.

Moreover, there is a set of listeners which can be registered on various levels (on a constraint, a variable, a model, a solver or a solution) and a plug-in mechanism for solver extensions. Using these hooks, various extensions (as for instance the conflict-based statistics or MAC) can be implemented.

A.1 Solver

As described in the chapter 4, the solver repeatedly selects a variable, a value, assigns the value to the variable and checks whether the solution is the best ever found until a termination condition succeeds. The implementation is in class Solver which can be found in the package ifs.solver. Following Figure 4.1 presents its core functionality.

```
class Solver {
  //termination condition
  TerminationCondition iTerminationCondition;
  //variable selection
  VariableSelection iVariableSelection;
  //value selection
  ValueSelection iValueSelection;
  //solution comparator
  SolutionComparator iSolutionComparator;
```

(continues on the next page)

```
  Solution solve(Model model) {

    Solution solution = model.createInitialSolution();

    //while not terminated
    while (iTerminationCondition.canContinue(solution)) {

      //select variable
      Variable variable =
        iVariableSelection.selectVariable(solution);

      //select value
      Value value =
        iValueSelection.selectValue(solution, variable);

      //(un)assign the selected value to the selected variable
      if (value!=null)
        variable.assign(value);
      else
        variable.unassign();

      // if the solution is the best ever found then memorize it
      if (iSolutionComparator.isBetterThanBest(solution))
        solution.saveBest();

    } //end while

    //restore the best ever found solution
    solution.restoreBest();

    return solution;
  }
}
```

Fig. A.1. Core of the IFS solver (class ifs.solver.Solver)

Solution class contains some information about the solution and the functions for storing and restoring the best ever found solution.

```
class Solution {
  public long getIteration(); //current iteration
  public double getTime(); //current solution time
  public Model getModel(); //model

  //store and restore the best solution
  public void saveBest() { getModel().saveBest(); }
  public void restoreBest() { getModel().restoreBest(); }
}
```

Fig. A.2. Core of the solution (class ifs.solver.Solution)

A.2 Model

The model, which is implemented by class Model (package ifs.model) contains all the variables and constraints in the problem. Moreover, it contains some useful functions which can be used for instance by the heuristics. The most interesting is the function *conflictValues* which returns all the values which are assigned and which are in a hard conflict with the given value. This means the values which have to be unassigned if the given value is selected for the assignment.

```
class Model {
  //variables
  public Vector variables();
  public void addVariable(Variable variable);
  public void removeVariable(Variable variable);

  //constraints
  public Vector constraints();
  public void addConstraint(Constraint constraint);
  public void removeConstraint(Constraint constraint);

  public Set conflictValues(Value value) {
    HashSet conflictValues = new HashSet();
    for (Enumeration e=value.variable().constraints().elements();
         e.hasMoreElements();) {
            Constraint c = (Constraint)e.nextElement();
            c.computeConflicts(value, conflictValues);
     }
     return conflictValues;
  }
  void saveBest() {
    for (Iterator i = iVariables.iterator(); i.hasNext(); ) {
      Variable variable = (Variable)i.next();
      variable.iBestAssignment = variable.iAssignment;
    }
  }

  void restoreBest() {
    for (Iterator i = iVariables.iterator(); i.hasNext(); ) {
      Variable variable = (Variable)i.next();
      variable.iAssignment = variable.iBestAssignment;
    }
  }

  ...
}
```

Fig. A.3. Core of the model (class ifs.model.Model)

A.2.1 Variables

The Variable class (package ifs.model) is an abstract implementation of a variable. Every variable can contain an assigned value, the initial assignment (in case of minimal perturbation problem) and the best ever found assignment. Also, it keeps track of the constraints in which the variable participates. The most interesting functions are *assign* and *unassign* which assign and unassign a value to the variable.

```
class Variable {
  Value iAssignment = null; //assigned value
  Value iInitialAssignment = null; //initial value (MPP)
  Value iBestAssignment = null; //best assignment value
  Collection iConstraints; //constraints which contain this variable

  Collection getValues() {
    //to be implemented: variable's domain
  }

  void unassign() {
    Value oldValue = iAssignment;
    iAssignment = null;
    for (Iterator i = iConstraints.iterator(); i.hasNext();) {
      Constraint constraint = (Constraint)i.next();
      constraint.unassigned(this, oldValue);
    }
  }

  void assign(Value value) {
    if (iAssignment != null) unassign();
    iAssignment = value;
    for (Iterator i = iConstraints.iterator(); i.hasNext();) {
      Constraint constraint = (Constraint)i.next();
      constraint.assigned(this, oldValue);
    }
  }
}
```

Fig. A.4. Core of a variable (class ifs.model.Variable)

A.2.2 Values

There is also an abstract class which implements a single value, class Value (package ifs.model). The most interesting property is that each value knows about the variable to which it belongs. Also, there is a method which compares two values. For general implementation of optimisation heuristics, every value can be converted to integer (function *toInt*). Of course, there can be made more dedicated, problem dependent heuristics which take into account a possible complexity of the value, if needed.

```
class Value {
  public Variable variable();
  public void setVariable(Variable variable);

  public void assigned(long iteration);
  public void unassigned(long iteration);

  public boolean valueEquals(Value value);
  public int toInt();
}
```

Fig. A.5. Core of a value (class ifs.model.Value)

A.2.3 Constraints

As indicated in chapter 4, the most important function of every constraint is the computation of the conflicts which will the selected value cause if assigned (method *computeConflicts*). Also, it is up to a constraint to unassign the conflicting variables caused by the constraint when there is a value assigned to a variable (method *assigned*). It also keeps track of what variables are involved in the constraint.

```
class Constraint {
  public Vector variables();

  void computeConflicts(Value value, Set conflicts) {
    //this method needs to be implemented by all constraints!
  }

  void unassigned(Value value) {}

  void assigned(Value value) {
    Set conflicts = new HashSet();
    computeConflicts(value, conflicts);
    for (Iterator i = conflicts.iterator(); i.hasNext(); ) {
      Value conflictingValue = (Value)i.next();
      conflictingValue.variable().unassign();
    }
  }
}
```

Fig. A.6. Core of a constraint (class ifs.model.Constraint)

A.3 Heuristics

There are four interfaces (Figures A.7. – A.10.) for implementing the heuristics which guides the IFS solver. Namely, there is the variable selection heuristics, the value selection heuristics, the solution comparator and the termination condition.

```
public interface VariableSelection {
  public Variable selectVariable(Solution solution);
}
```

Fig. A.7. Variable selection (class ifs.heuristics.VariableSelection)

```
public interface ValueSelection {
  public Value selectValue(Solution solution,
                           Variable selectedVariable);
}
```

Fig. A.8. Value selection (class ifs.heuristics.ValueSelection)

```
public interface SolutionComparator {
  public boolean isBetterThanBestSolution(Solution currentSolution);
}
```

Fig. A.9. Solution comparator (class ifs.solution.SolutionComparator)

```
public interface TerminationCondition {
    public boolean canContinue(Solution currentSolution);
}
```

Fig. A.10. Termination condition(ifs.termination.TerminationCondition)

In the following sections the general implementations of the above heuristics are discussed.

A.3.1 Variable Selection

The very basic (but sufficient for all the experiments) implementation of the variable selection criterion picks an unassigned variable randomly. If there is no unassigned variable, one of the assigned ones is selected.

```
public class GeneralVariableSelection implements VariableSelection {

  public Variable selectVariable(Solution solution) {
    if (!solution.getModel().unassignedVariables().isEmpty()) {
      return (Variable)ToolBox.random(
                      solution.getModel().unassignedVariables());
    } else {
      return (Variable)ToolBox.random(
                      solution.getModel().assignedVariables());
    }
}
```

Fig. A.11. General implementation of the variable selection criterion (class ifs.heuristics.GeneralVariableSelection)

A.3.2 Solution Comparator

Current solution is better than the best ever found solution if there is no best solution yet saved or if it has lower number of unassigned variables. If the current solution has the same number of assigned variables as the best ever found solution, the better solution has the lowest sum of the assigned values (method *model.getTotalValue* sums *value.toInt()* over all assigned values). There is also a general solution comparator which takes the number of perturbations into account for the minimal perturbation problem (ifs.solution.MPPSolutionComparator).

```
public class GeneralSolutionComparator
  implements SolutionComparator {

    public boolean isBetterThanBestSolution(Solution solution) {
      if (solution.getBestInfo()==null) {
        //there is no best solution yet saved
        return true;
      }
      int currentUnassigned =
              solution.getModel().unassignedVariables().size();
      int bestUnassigned =
              solution.getModel().getBestUnassignedVariables();
      if (bestUnassigned != currentUnassigned) {
        return bestUnassigned > unassigned;
      }

      int curentValue = solution.getModel().getTotalValue();
      int bestValue = solution.getBestValue();
      return currentValue < bestValue;
    }
}
```

Fig. A.12. General implementation of the solution comparator (class ifs.solution.GeneralSolutionComparator)

A.3.3 Termination Condition

The solver should stop when there is a complete solution found (e.g., if there is no optimisation involved) or when the given number of iterations or timeout is reached. There is also a general implementation which takes into account the minimal number of perturbations in case of solving the minimal perturbation problem (class ifs.termination.MPPTerminationCondition).

```java
public class GeneralTerminationCondition
  implements TerminationCondition {
    private int iMaxIter;
    private double iTimeOut;
    private boolean iStopWhenComplete;

    public boolean canContinue(Solution currentSolution) {
      if (iMaxIter>=0 && currentSolution.getIteration()>=iMaxIter) {
        //Maximum number of iteration reached.
        return false;
      }

      if (iTimeOut>=0 && currentSolution.getTime()>iTimeOut) {
        // Timeout reached.
        return false;
      }

      if (iStopWhenComplete) {
        boolean ret = (!currentSolution.getModel().
                          unassignedVariables().isEmpty());
        //Complete solution found.
        return ret;
      }

      return true;
    }
}
```

Fig. A.13. General implementation of the termination condition (class ifs.termination.GeneralTerminationCondition)

A.3.4 Value Selection

General implementation of value selection criterion is quite complicated. It covers several basic heuristics (random-walk, CBS, tabu-search, MAC). Also it works for the initial as well as minimal perturbation problem.

```
public class GeneralValueSelection implements ValueSelection {
  //random walk selection
  double iRandomWalkProb = 0.0;

  //weight of a conflict
  double iWeightCoflicts = 1.0;
  //weight of a value (value.toInt())
  double iWeightValue = 0.0;

  //TABU-SEARCH: size of tabu-list
  int iTabuSize = 0;
  //TABU-SEARCH: tabu-list
  ArrayList iTabu = null;
  //TABU-SEARCH: pointer to the last value in the tabu-list
  int iTabuPos = 0;

  //Minimal perturbations problem
  boolean iMPP = false;
  //MPP: initial selection probability
  double iInitialSelectionProb = 0.0;
  //MPP: limit on the number of perturbations
  int iMPPLimit = -1;
  //MPP: weight of the difference in initial assignments
  double iWeightDeltaInitialAssignment = 0.0;

  //Conflict based statistics (null if not present)
  ConflictStatistics iStat = null;
  //CBS: CBS weighted conflict weight
  double iWeightWeightedCoflicts = 0.0;

  //MAC: null if there is no arc-consistency
  MacPropagation iProp = null;
  //MAC: allow selection of removed values (MAC+)
  boolean iAllowNoGood = false;

  public Value selectValue(Solution solution,
                   Variable selectedVariable) {

    if (iMPP && selectedVariable.getInitialAssignment() != null) {
      //Minimal perturbations problem
      if (solution.getModel().unassignedVariables().isEmpty()) {
        //complete solution – decrease MPP limit if used
        if (solution.getModel().perturbVariables().size() <=
                                                    iMPPLimit) {
          iMPPLimit =
             solution.getModel().perturbVariables().size() - 1;
        }
      }
      if (iMPPLimit >= 0 &&
        solution.getModel().perturbVariables().size() > iMPPLimit) {
        //MPP limit reached – initial value has to be assigned
        return selectedVariable.getInitialAssignment();
      }
      if (ToolBox.random() <= iInitialSelectionProb) {
        //with the given probability, initial value is selected
        return selectedVariable.getInitialAssignment();
      }
    } //MPP
```

(continues on the next page)

```
    Vector values = selectedVariable.values();

    if (ToolBox.random() <= iRandomWalkProb) {
      //random-walk
      return (Value)ToolBox.random(values);
    }

    if (iProp != null) {
      //MAC: select one of the not-removed values (usually always)
      Collection goodValues = iProp.goodValues(selectedVariable);
      if (!goodValues.isEmpty())
         values = new Vector(goodValues);
      } else if (!iAllowNoGood) {
         //all values are removed and the selection of
         //not-removed values is prohibited
         return null;
      }
    }

    //values with the lowest weighted sum
    Vector bestValues = null;
    double bestWeightedSum = 0;

    //go through all the values
    for (Enumeration i1 = values.elements();i1.hasMoreElements();) {
      Value value = (Value)i1.nextElement();
      if (iTabu != null && iTabu.contains(value)) {
        //value is in the tabu-list
        continue;
      }

      if (value.equals(selectedVariable.getAssignment())) {
        //do not pick the same value as it is currently assigned
        //if there is a value assigned to the selected variable
        continue;
      }

      //conflicting values
      Collection conf = solution.getModel().conflictValues(value);

      double weightedConflicts = 0.0; //CBS weighted conflicts
      if (iStat != null) {
        weightedConflicts = iStat.countRemovals(
                            solution.getIteration(), conf, value));
      }

      //MPP: difference in initial assignments
      long deltaInitialAssignments = 0;
      if (iMPP) {
        //go through all conflicts
        for (Iterator it1 = conf.iterator(); it1.hasNext();) {
          Value aValue = (Value)it1.next();
          if (aValue.variable().getInitialAssignment() != null) {
            //not assigned to an initial value -> good to unassign
            deltaInitialAssignments--;
          }
        }
```

(continues on the next page)

```
        if (value.equals(selectedVariable.getInitialAssignment()) {
          //value is different from initial value -> bad to assign
          deltaInitialAssignments++;
        }
        if (iMPPLimit >= 0 &&
             (solution.getModel().perturbVariables().size()
                + deltaInitialAssignments) > iMPPLimit) {
             //assignment exceeds MPP limit
             continue;
        }
      }

      //weighted sum of several criteria
      double weightedSum =
        (iWeightDeltaInitialAssignment * deltaInitialAssignments)
      + (iWeightWeightedCoflicts * weightedConflicts)
      + (iWeightCoflicts * conf.size())
      + (iWeightValue * value.toInt());

      //store best values
      if (bestValues == null || bestWeightedSum > weightedSum) {
        bestWeightedSum = weightedSum;
        if (bestValues == null)
          bestValues = new Vector();
        else
          bestValues.clear();
        bestValues.add(value);
      } else if (bestWeightedSum == weightedSum) {
        bestValues.add(value);
      }
    } //end of the for cycle over all values

    Value selectedValue = (Value)ToolBox.random(bestValues);
    if (selectedValue == null) {
      //no value in the bestValues -> select randomly
      selectedValue = (Value)ToolBox.random(values);
    }

    //In case of tabu-search, put into tabu-list
    if (iTabu != null) {
      if (iTabu.size() == iTabuPos)
        iTabu.add(selectedValue);
      else
        iTabu.set(iTabuPos, selectedValue);
      iTabuPos = (iTabuPos + 1) % iTabuSize;
    }

    return selectedValue;
  }
}
```

Fig. A.14. General implementation of the value selection criterion (class ifs.heuristics.GeneralValueSelection)

Appendix B Examples

In this chapter, an implementation of Random Binary CSP using IFS framework is presented. For more details or other problems discussed in chapter 6, consult the implementation or API (JavaDoc) documentation on the attached CD-ROM. There are the following packages: ifs.example.csp (Random Binary CSP), ifs.example.rpp (Random Placement Problem) or ttsolver (Purdue University Timetabling Problem).

B.1 Random Binary CSP

First of all, we need to define a variable and a value. The only thing which needs to be implemented is the domain of a variable (see *computeValues* method in the Figure B.1.). Note that the following example is complete and it was not simplified, there is nothing more to be written to be able to execute IFS on random binary CSPs.

```
public class CSPVariable extends ifs.model.Variable {
  public CSPVariable(int domainSize) {
    super(null);  //no intial value
    setValues(computeValues(domainSize));
  }
  public Vector computeValues(int domainSize) {
    Vector values = new Vector();
    for (int i=0; i<domainSize; i++)
      values.add(new CSPValue(this,i));
    return values;
  }
}
```

Fig. B.1. Definition of CSP variable (ifs.example.csp.CSPVariable)

```
public class CSPValue extends ifs.model.Value {
  public CSPValue(Variable variable, int value) {
      super(variable, value);
  }
}
```

Fig. B.2. Definition of CSP value (ifs.example.csp.CSPValue)

Next, we need to define binary constraints between CSP variables. Array *iIsConsistent* is used for memorizing what pairs of values are compatible, the method *init* generates these compatible pairs. The method *isConsistent* checks the consistency of a pair of given values (note that it needs to take the values in the correct order). Method *computeConflicts* checks whether the other assigned variable (diferent from the one which is going to be assigned) has a value compatible with the selected value.

```
public class CSPBinaryConstraint extends ifs.model.BinaryConstraint{
  boolean iIsConsistent[][] = null;
  int iNrCompatiblePairs;

  public CSPBinaryConstraint(int nrCompatiblePairs) {
      iNrCompatiblePairs = nrCompatiblePairs;
  }

  void swap(int[][] allPairs, int first, int second) {
    int[] a = allPairs[first];
    allPairs[first] = allPairs[second];
    allPairs[second] = a;
  }

  public void init(Random rndNumGen) {
    int numberOfAllPairs =
      first().values().size() * second().values().size();
    int[][] allPairs = new int[numberOfAllPairs][];
    int idx = 0;

    iIsConsistent =
     new boolean[first().values().size()][second().values().size()];

    for (Enumeration i1=first().values().elements();
         i1.hasMoreElements();) {
      CSPValue v1 = (CSPValue)i1.nextElement();
      for (Enumeration i2=second().values().elements();
           i2.hasMoreElements();) {
        CSPValue v2 = (CSPValue)i2.nextElement();
        iIsConsistent[v1.toInt()][v2.toInt()] = false;
        allPairs[idx++] = new int[] {v1.toInt(), v2.toInt()};
      }
    }

    for (int i=0; i<iNrCompatiblePairs; i++) {
      swap(allPairs, i,
           i+(int)(rndNumGen.nextDouble()*(numberOfAllPairs-i)));
      iIsConsistent[allPairs[i][0]][allPairs[i][1]] = true;
    }
  }
```

(continues on the next page)

```
  public boolean isConsistent(Value value1, Value value2) {
    if (value1==null || value2==null) return true;
    if (isFirst(value1.variable())) {
      return iIsConsistent[value1.toInt()][value2.toInt()];
    } else {
      return iIsConsistent[value2.toInt()][value1.toInt()];
    }
  }

  public void computeConflicts(Value selectedValue, Set conflicts) {
    if (isFirst(selectedValue.variable())) {
      if (!isConsistent(selectedValue, second().getAssignment()))
        conflicts.add(second().getAssignment());
    } else {
      if (!isConsistent(selectedValue, first().getAssignment()))
        conflicts.add(first().getAssignment());
    }
  }
}
```

Fig. B.3. Definition of CSP constrain (ifs.example.csp.CSPBinaryConstraint)

Next, we need to implement the model (see Figure B.4.). The binary CSP is generated according to the given parameters. First of all, variables and constraints are generated and added into the model. Next, the constraint graph is constructed and constraints are initialized.

```
public class CSPModel extends ifs.model.Model {
  public CSPModel(int nrVariables, int nrValues, int nrConstraints,
                  int nrCompatiblePairs, long seed) {
    generate(nrVariables, nrValues, nrConstraints,
             nrCompatiblePairs, seed);
  }

  void swap(Variable[][] allPairs, int first, int second) {
    Variable[] a = allPairs[first];
    allPairs[first]=allPairs[second];
    allPairs[second]=a;
  }

  public void generate(int nrVariables, int nrValues,
            int nrConstraints, int nrCompatiblePairs, long seed) {
    Random rnd = new Random(seed);

    for (int i=0; i<nrVariables; i++) {
      CSPVariable var = new CSPVariable(nrValues);
      addVariable(var);
    }
```

(continues on the next page)

```
    for (int i=0; i<nrConstraints; i++) {
      CSPBinaryConstraint c =
            new CSPBinaryConstraint(nrCompatiblePairs);
      addConstraint(c);
    }

    int numberOfAllPairs =
              variables().size()*(variables().size()-1)/2;
    Variable[][] allPairs = new Variable[numberOfAllPairs][];

    int idx=0;
    for (Enumeration i1=variables().elements();
         i1.hasMoreElements();) {
      Variable v1 = (Variable)i1.nextElement();
      for (Enumeration i2=variables().elements();
           i2.hasMoreElements();) {
        Variable v2 = (Variable)i2.nextElement();
        if (v1.getId()>=v2.getId()) continue;
        allPairs[idx++]=new Variable[] {v1,v2};
      }
    }

    idx = 0;
    for (Enumeration i1=constraints().elements();
         i1.hasMoreElements();) {
      CSPBinaryConstraint c = (CSPBinaryConstraint)i1.nextElement();
      swap(allPairs, idx,
              idx+(int)(rnd.nextDouble()*(numberOfAllPairs-idx)));
      c.addVariable(allPairs[idx][0]);
      c.addVariable(allPairs[idx][1]);
      c.init(rnd);
      idx++;
    }
  }
}
```

Fig. B.4. Definition of CSP model (ifs.example.csp.CSPModel)

That's all. The following Figure B.5. presents a code which will execute the IFS RW(2%) solver on the CSP(25,12,198/300,36/144) problem. The best solution found within 60 seconds is then printed.

```
public static void main(String[] args) {
  int nrVariables = 25;
  int nrValues = 12;
  int nrConstraints = 198;
  double tigtness = 0.25;

  int nrAllPairs = nrValues*nrValues;
  int nrCompatiblePairs = (int)((1.0-tigtness)*nrAllPairs);
  long seed = System.currentTimeMillis();
```

(continues on the next page)

```java
  //configuration
  ifs.util.DataProperties cfg = new ifs.util.DataProperties();
  cfg.setProperty("Termination.Class",
                   "ifs.termination.GeneralTerminationCondition");
  cfg.setProperty("Termination.StopWhenComplete","true");
  cfg.setProperty("Termination.TimeOut","60");
  cfg.setProperty("Comparator.Class",
                        "ifs.solution.GeneralSolutionComparator");
  cfg.setProperty("Value.Class",
                          "ifs.heuristics.GeneralValueSelection");
  cfg.setProperty("Value.WeightConflicts", "1");
  cfg.setProperty("Value.RandomWalkProb", "0.02");
  cfg.setProperty("Variable.Class",
                       "ifs.heuristics.GeneralVariableSelection");

  //solver and model intialization
  CSPModel model =
    new CSPModel(nrVariables,nrValues,nrConstraints,
               nrCompatiblePairs,seed);
  ifs.solver.Solver solver = new ifs.solver.Solver(cfg);
  solver.setInitalSolution(model);

  //solver execution
  solver.start();
  try {
    solver.getSolverThread().join();
  } catch (InterruptedException e) {}

  //take the best ever found solution
  ifs.solution.Solution solution = solver.lastSolution();
  solution.restoreBest();

  //print some results
  System.out.println("Best solution found after "+
      solution.getBestTime()+" seconds ("+
      solution.getBestIteration()+" iterations).");
  System.out.println("Number of assigned variables is "+
      solution.getModel().assignedVariables().size());
  System.out.println("Total value of the solution is "+
      solution.getModel().getTotalValue());
  int idx=1;
  for (Enumeration e=solution.getModel().variables().elements();
       e.hasMoreElements();) {
    CSPVariable v=(CSPVariable)e.nextElement();
    if (v.getAssignment()!=null)
      System.out.println(
               "Var"+(idx++)+"="+v.getAssignment().toInt());
  }
}
```

Fig. B.5. Example of solver execution (ifs.example.csp.SimpleTest)

Appendix C   Simple Timetabling Problem

In [Mul01, MB01, MB02], we proposed a simplified model for timetabling problems consisting of a set of resources, a set of activities, and a set of dependencies between the activities (see http://kti.mff.cuni.cz/~muller/ttbench/). Time is divided into time slots with the same duration. Every slot may have assigned a constraint, either hard or soft: a hard constraint indicates that the slot is forbidden for any activity, a soft constraint indicates that the slot is discouraged. We call these constraints “time preferences”. Every activity and every resource may have assigned a set of time preferences, which indicate forbidden and discouraged time slots.

Activity (which can be, for instance, directly mapped to a lecture) is identified by its name. Every activity is described by its duration (expressed as a number of time slots), by time preferences, and by a set of resources. This set of resources determines which resources are required by the activity. To model alternative as well as required resources, we divide the set of resources into several subsets – resource groups. Each group is either conjunctive or disjunctive: the conjunctive group of resources means that the activity needs all the resources from the group, the disjunctive group means that the activity needs exactly one of the resources (we can choose from several alternatives). An example can be a lecture, which will take place in one of the possible classrooms and it will be taught for all of the selected classes. Note that we do not need to model conjunctive groups explicitly because we can use a set of disjunctive groups containing exactly one resource instead (the set of required resources can be described in a conjunctive normal form). However, usage of both conjunctive and disjunctive groups simplifies modelling for the users.

Resource is also identified by its name and it is fully described by time preferences. There is a hard condition that only one activity can use the resource at the same time. For instance, such resource can represent a teacher, a class, a classroom, or another special resource at the lecture timetabling problem.

Finally, we need a mechanism for defining and handling direct dependencies between the activities. It seems sufficient to use binary dependencies only that define relationship between two activities. In [MB01], we defined three temporal constraints: the activity finishes before another activity, the activity finishes exactly at the time when the second activity starts, and two activities run concurrently (they have the same start time).

The solution of the problem defined by the above model is a timetable where every scheduled activity has assigned its start time and a set of reserved resources that are needed for its execution (the activity is allocated to respective slots of the reserved resources). This timetable must satisfy all the hard constraints, namely:

- every scheduled activity has all the required resources reserved, i.e., all resources from the conjunctive groups and one resource from each disjunctive group of resources,
- two scheduled activities do not use the same resource at the same time,
- no activity is scheduled into a time slot where the activity or some of its reserved resources has a hard constraint in the time preferences,
- all dependencies between the scheduled activities are satisfied.

Furthermore, we want to minimize the number of violated soft constraints in the time preferences of resources and activities, i.e., the total number of used slots that are discouraged over all resources and activities.

In this section we present some results for the Simple Timetabling Problem, achieved using randomly generated problems with the size of 20 classes, rooms, and teachers and 10 slots per day (5 days). These problems have the following properties:

- Randomly selected 5% of all slots for each resource (class, room, teacher) and activity (lecture) are prohibited.
- 30% of all slots for each resource and activity are marked as discouraged.
- Moreover, there are 50 binary precedence hard constraints between lectures.
- The average length of an activity is about 2.5 time slots.
- Every activity has associated a class and a teacher and a randomly selected set of available rooms (with the size from 1 to 10).
- There exists a complete feasible timetable.

The objective here is to find a complete feasible timetable which meets all the hard constraints and which minimizes the number of discouraged time slots. See [Mul01] for more details about the problem generator.

The problem is modelled in such a way that every lecture is represented by a variable, a resource as a constraint and every possible location of an activity in the time and space is represented by a single value. It means that a value stands for a selection of the time (starting time slot), and one of the available rooms. Binary dependencies are of course represented as constraints as well. As for the solver, exactly the same procedures are used as in weighted CSP, but now the weight of a value represents the number of discouraged time slots it uses.

Figures C.1 and C.2 present the number of assigned lectures (in the percentage of all lectures) and solution quality (number of occupied discouraged time slots) wrt. the fill factor (average usage of classes, rooms and teachers). The average values of the best achieved solutions from 10 runs on different problem instances within the 5 minute time limit are presented.

Both IFS MAC and IFS MAC+ were not able to find a complete solution even for 50% filling of the timetable within the given 5 minute time limit. IFS MAC was able to assign in average about 64% of all variables and IFS MAC+ about 80% of variables.

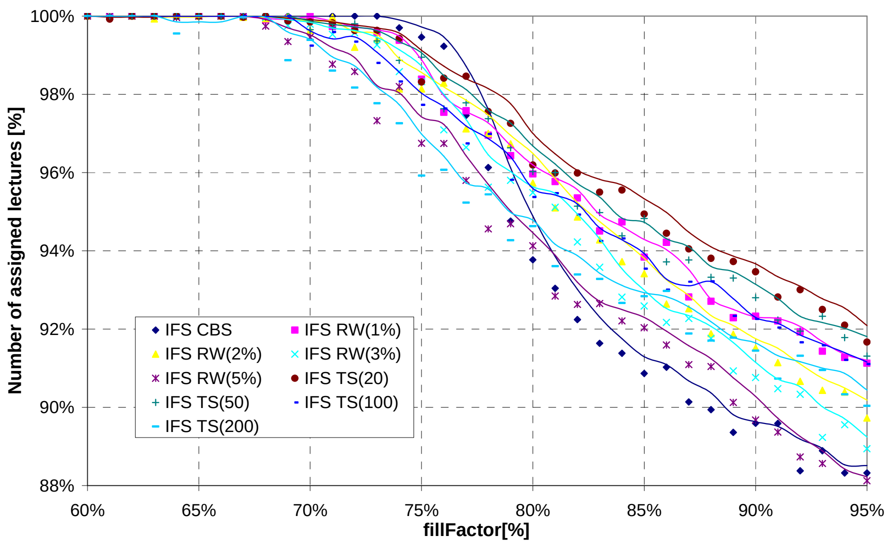


Fig. C.1. Number of assigned lectures.

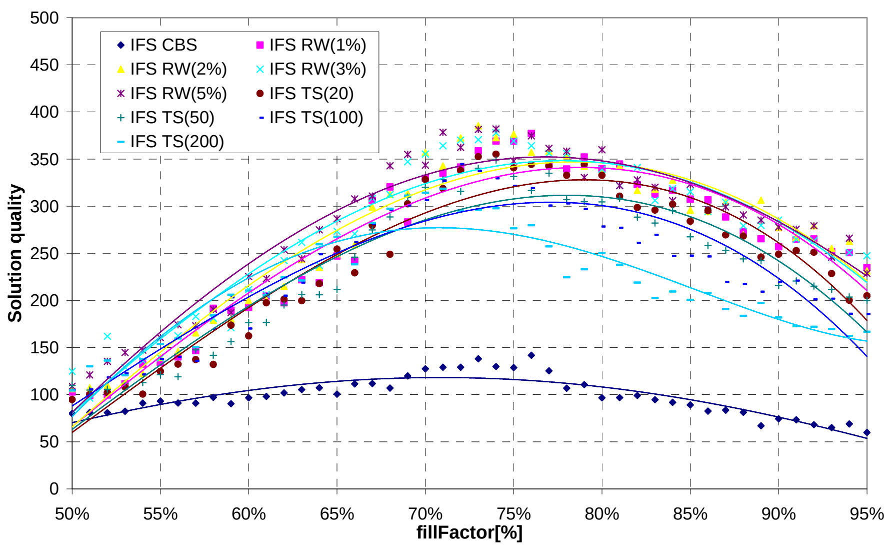


Fig. C.2. Solution quality (the total number of used discouraged time slots, lower number of these slots means better solution).

IFS CBS was able to give us a complete solution up to 73% filling of the timetable in every run. Moreover, it was able to find much better solutions in the number of used discouraged time slots than all other tested algorithms.

C.1 Timetabling Problem at Charles University

For solving a real-life timetable problem at the Faculty of Mathematics and Physics at Charles University, Prague we extended the above discussed simple timetabling problem.

Model

First, let us look at the traditional classroom capacity constraint. We are scheduling lectures and we know in advance how many students will attend the lectures (with the exception of alternative lectures, see below). Thus, the capacity restriction is naturally modelled by assigning only the classrooms with enough capacity to the lecture. Note that there are groups of alternative resources attached to each lecture, so the group describing the classrooms contains only the classrooms with enough capacity.

Second, we should be able to model alternative activities per resources (not only alternative resources per activity). In particular, we have several alternative lectures and several classes (groups of students) that should attend the lecture independently of which particular one. These alternative lectures can be taught simultaneously by different teachers in different classrooms or one teacher has these lectures at different times etc. We want to maximise the number of available alternatives for the students under the hard constraint that at least one alternative must be available for each class.

To describe alternative activities, we introduced a new entity into our model, *a group of alternative activities*. This group is assigned to some resources with the following constraints:

- each resource to which the group of alternative activities is assigned must have at least one free slot to which some activity from the group of alternative activities can be allocated (a hard constraint),
- the number of available alternative activities for all resources that have assigned a group of alternative activities should be maximised (modelled as a soft constraint).

Note that we are working with partial timetables as well, so the above hard constraint is relevant only to schedules where all the activities from the group of alternative activities are allocated (no look ahead there). Also notice that we are encoding the objective function using soft constraints, in particular this objective function is encoded in location selection heuristic that minimises the number of violations of soft constraints (see bellow).

The next group of hard constraints is derived from the organisation structure of the faculty. The lectures are taught in three different buildings in Prague so if there are two lectures taught in different buildings and sharing either a teacher or a class then there must be enough time to move between the buildings. In particular, there must be at least one time slot between the lectures or the lectures are taught in the same building. Moreover, we prefer the lectures to be taught in the same building where the teacher has his or her office. Finally, the number of crossovers between the buildings during the day should be minimised both for classes and for teachers.

The last group of constraints concerns the time preferences. Some subjects have more lectures per week. In such a case, the lectures must be scheduled to different days. Note that such a constraint can be easily described using the direct dependency between the lectures. Each class should not have more than ten teaching hours per day and more than six hours without a break. Similarly, each teacher should have neither more than eight hours per day nor more than six hours without a break. Finally, there is a preference to daytime. For example, the early morning hours or late evening hours are less preferred. Also Friday afternoon is not preferred. These preferences are described as a number (from least preferred -3 to most preferred 3) for each time slot in the week. We call this number a global time preference. The last time constraint, which is very weak, says that the number of breaks (free slots between the lectures per day) should be minimal both for teachers and for students.

Let us now summarise all the additional constraints. We can divide them into hard constraints that must be satisfied and soft constraints expressing the preferences. The additional hard constraints are:

- two (not alternative) lectures of the same subject cannot be taught on the same day,
- the capacity of each classroom cannot be exceeded,
- each student (that has to attend some lectures, which are alternative) must have possibility to attend at least one of the alternative lectures,
- there is at least one hour (slot) break between every two lectures which go one after another, and that share either the same teacher or the class and that are taught in different buildings.

The additional soft constraints are:

- one student should not have more than ten hours per day and should not have more than six hours without a break,
- one teacher should not have more than eight hours per day and should not have more than six hours without a break,
- the number of crossovers during a day for teachers and classes should be minimal,
- a lecture should be taught in the same building where the teacher has his or her workplace
- maximise the number of alternatives which students can attend,
- maximise the sum of used time slots multiplied by the global time preference of each slot,
- minimise the number of free slots between the first and the last lecture of the day for all teachers and classes.

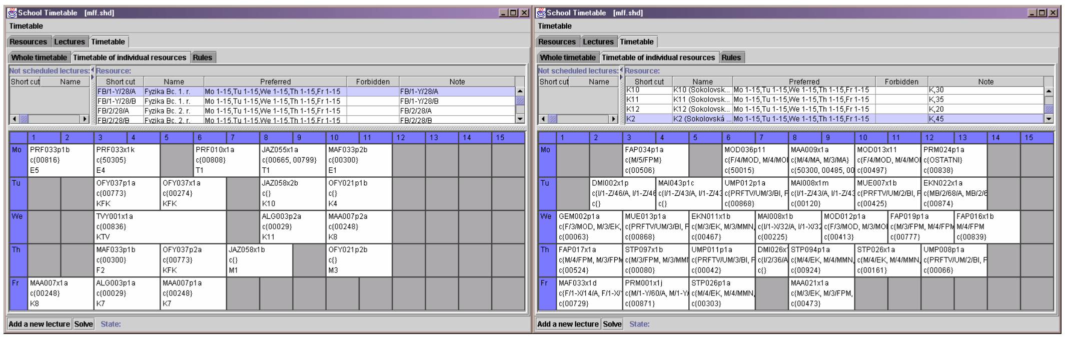
Results

The above described problem was tested using real data from an Fall 2001 semester at the Faculty of Mathematics and Physics, Charles University (see Figure C.3). The problem size and structure was as follows:

- 5 days per week, 15 time slots per day,
- 746 lectures, which have to be centrally scheduled (with average duration 2.03 time slots, teaching hour = 45 minutes), in total 1512 timeslots,
- 349 classes or sub-classes (454 groups of classes),
- 479 teachers,
- 41 classrooms (but only 30 can be used),
- 3 different locations (buildings).

It takes approximately 2 to 4 minutes to solve the problem without any user intervention. Moreover, the system provides interactive capabilities, so the user can easily adjust the timetable, e.g. via the drag and drop technique. This way the user can guide the system or he or she can express some preferences that can be hardly encoded in the soft constraints. The following list shows some features of the timetable found by the system (with no user intervention):

- all activities were scheduled (and all hard constraints were satisfied),
- there were 76 crossovers for the classes and 7 crossovers for the teachers (crossover means change of building during a day),
- there were only 21 cases when a class had more than 10 hours a day and one case when a class had more than 6 hours without a break,
- there was no case when a teacher had more than 8 hours a day or more than 6 hours without a break,
- a class could attend on average 84% of alternative lectures announced for it,
- on average 84% of lectures were scheduled to the same building where the teacher has his office,
- 74% of lectures were scheduled between 3rd and 11th slot (from 9:00 till 16:25), on Fridays between 3rd and 6td slot (from 9:00 till 12:15);
- 87% of lectures were scheduled between 2nd – 12th slot (from 8:10 till 17:15), on Fridays between 2nd – 7th slot (from 8:10 till 13:05);
- only 3% of lectures were scheduled on or after 14th slot (from 18:10) and on or after 10th slot (from 14:50) on Fridays



School Timetable [mff.shd]

Timetable

Resources | Lectures | Timetable

Whole timetable | Timetable of individual resources | Rules

Not scheduled lectures:

Short cut	Name

Resource:

Short cut	Name	Preferred	Forbidden	Note
FB/1-Y/28/A	Fyzika Bc. 1. r.	Mo 1-15,Tu 1-15,We 1-15,Th 1-15,Fr 1-15		FB/1-Y/28/A
FB/1-Y/28/B	Fyzika Bc. 1. r.	Mo 1-15,Tu 1-15,We 1-15,Th 1-15,Fr 1-15		FB/1-Y/28/B
FB/2/28/A	Fyzika Bc. 2. r.	Mo 1-15,Tu 1-15,We 1-15,Th 1-15,Fr 1-15		FB/2/28/A
FB/2/28/B	Fyzika Bc. 2. r.	Mo 1-15,Tu 1-15,We 1-15,Th 1-15,Fr 1-15		FB/2/28/B

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Mo	PRF033p1b c{00816} E5		PRF033x1k c{50305} E4			PRF010x1a c{00808} T1		JAZ055x1a c{00665, 00799} T1		MAF033p2b c{00300} E1					
Tu			OFY037p1a c{00773} KFK		OFY037x1a c{00274} KFK			JAZ058x2b c{} K10		OFY021p1b c{} K4					
We			TVY001x1a c{00836} KTV					ALG003p2a c{00029} K11		MAA007p2a c{00248} K8					
Th			MAF033p1b c{00300} F2		OFY037p2a c{00773} KFK		JAZ058x1b c{} M1			OFY021p2b c{} M3					
Fr	MAA007x1a c{00248} K8		ALG003p1a c{00029} K7		MAA007p1a c{00248} K7										

Add a new lecture | Solve | State:

School Timetable [mff.shd]

Timetable

Resources | Lectures | Timetable

Whole timetable | Timetable of individual resources | Rules

Not scheduled lectures:

Short cut	Name

Resource:

Short cut	Name	Preferred	Forbidden	Note
K10	K10 (Sokolovsk...	Mo 1-15,Tu 1-15,We 1-15,Th 1-15,Fr 1-15		K,30
K11	K11 (Sokolovsk...	Mo 1-15,Tu 1-15,We 1-15,Th 1-15,Fr 1-15		K,35
K12	K12 (Sokolovsk...	Mo 1-15,Tu 1-15,We 1-15,Th 1-15,Fr 1-15		K,20
K2	K2 (Sokolovská ...	Mo 1-15,Tu 1-15,We 1-15,Th 1-15,Fr 1-15		K,45

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Mo			FAP034p1a c{M/5/FPM} c{00506}			MOD036p11 c{F/4/MOD, M/4/MOI c{50015}		MAA009x1a c{M/4/MA, M/3/MA} c{50300, 00485, 00		MOD013x11 c{F/4/MOD, M/4/MOI c{00497}		PRM024p1a c{OSTATNI} c{00838}			
Tu		DMI002x1p c{I/1-Z/46/A, I/1-Z/46 c{}		MAI043p1c c{I/1-Z/43/A, I/1-Z/43 c{}		UMP012p1a c{PRFTV/UM/3/BI, F c{00868}		MAI008x1m c{I/1-Z/43/A, I/1-Z/43 c{00120}		MUE007x1b c{PRFTV/UM/2/BI, F c{00425}		EKN022x1a c{MB/2/68/A, MB/2/6 c{00874}			
We	GEM002p1a c{F/3/MOD, M/3/EK, c{00063}		MUE013p1a c{PRFTV/UM/3/BI, F c{00868}		EKN011x1b c{M/3/EK, M/3/MMN, c{00467}		MAI008x1b c{I/1-X/32/A, I/1-X/32 c{00225}		MOD012p1a c{F/3/MOD, M/3/MOI c{00413}		FAP019p1a c{M/3/FPM, M/4/FPM c{00777}		FAP016x1b M/4/FPM c{00839}		
Th	FAP017x1a c{M/4/FPM, M/3/FPM c{00524}		STP097x1b c{M/3/FPM, M/3/MMI c{00080}		UMP011p1a c{PRFTV/UM/3/BI, F c{00042}		DMI026x1 c{I/2/36/A c{}	STP094p1a c{M/4/EK, M/4/MMN, c{00924}		STP026x1a c{M/4/EK, M/4/MMN, c{00161}		UMP008p1a c{PRFTV/UM/2/BI, F c{00066}			
Fr	MAF033x1d c{F/1-X/14/A, F/1-X/1 c{00729}		PRM001x1j c{M/1-Y/60/A, M/1-Y c{00871}		STP026p1a c{M/4/EK, M/4/MMN, c{00303}			MAA021x1a c{M/3/EK, M/3/FPM, c{00473}							

Add a new lecture | Solve | State:

Fig. C.3. The system generates compact timetables: a timetable for a class (left), a timetable for a classroom (right)

Appendix D CD-ROM Content

This thesis includes a CD-ROM with electronic form of this thesis, implemented program, program documentation and source codes and several examples.

Folder or file	**Content**
\www\index.html	CD-ROM content
\www\publications.html	List of publications
\doc\phd-thesis05.pdf	This PhD thesis in PDF format
\doc\cv.pdf	Curriculum Vitae
\doc*.pdf	Other publications
\src\ifs	Source code of the implemented IFS program
\src\ttsolver	Source code of the Purdue University Timetabling program
\data\purdue	Example input data for Purdue Univ. Timetabling
\data\purdue\solution	Example solutions to Purdue Univ. Timetabling
\data\rpp	Example input data for Random Placement Problem
\data\rpp-mpp	Example input data for RPP (minimal pert. version)
\data\tt	Timetabling problem from [Mul01,MB01,MB02] (called Simple Timetabling Problem)
\doc\api\index.html	JavaDoc (source code) documentation
\lib	Compiled program
\bin	Example scripts
\cfg	Example configurations
\extra	Bonus: interactive timetabling program from my master thessis
\tools	JDK 1.5.0, Apache Ant 1.6.2

Fig. D.1. Included CD-ROM Content

Note that the result of this thesis is not a program with a fancy graphical user interface solving a particular timetabling problem, but a Java library that is capable of solving various CSP, CSOP, MPP and MPOP problems. Some of such problems (the ones that are discussed in the previous chapters) are implemented and available on the CD-ROM.

Due to some technical as well as legal issues it was not possible to put the graphical user interface for Purdue Timetabling Problem on the CD-ROM.

For more details, see the documentation on the attached CD-ROM (starting from \www\index.html).