Joint NCC & IIS Bull., Comp. Science, 16 (2001), 67–78
© 2001 NCC Publisher

Application of constraint hierarchy to timetabling problems

T. Yakhno, M. E. Tekin

1. Introduction

At the beginning of each school year, universities everywhere must undertake boring and time-consuming process of slotting students, teachers and lessons into available classrooms. It is a natural scheduling problem and schedule-makers are looking for simple flexible and effective automatic systems.

Scheduling problems are often NP-complete. There are many approaches to deal with them such as algorithmic approach, operation research approach, etc. Algorithmic (or integer programming) approach is hard for scheduling problems, since the domain of the search space is very large. Finding the solution in this search space by traditional search methods is very inefficient. A lot of attention in operation research has been paid to scheduling problems that are based on relatively simple mathematical models. Operation research often aims to achieve a high level of efficiency. This approach has some classical models to use when modeling a practical scheduling problem. The main disadvantages of these models are in that they discard many degrees of freedom and side constraints that exist in the practical scheduling situation. Discarding degrees of freedom and side constraints causes optimal solutions to be eliminated, regardless of the solution method used. Discarding the side constraints may result in a simplified version of the problem and solving this simplified problem might be easier but the solution found can be impractical for the original problem [2].

While operation research methods are efficiency oriented (i.e., they are specialized algorithms for specialized problems), artificial intelligence research tends to investigate more general scheduling models and tries to solve these problems by using the general problem solving techniques. However, in some specific cases, AI algorithms may perform poorly on specific instances compared to operation research algorithms. On the other hand, operation research offers us efficient algorithms to solve the problems that however can be not suitable in practice, while AI algorithms are more applicable [8].

Naturally, we want the best of both worlds, i.e., we want efficient algorithms that can be applied to a wide range of problems [6].

With the emergence of constraint programming, especially the introduction of finite domain constraints into logic programming, the constraint satisfaction approach has started to attract more and more attention due to its effectiveness in solving real-life planning/scheduling problems. Although the constraint programming approach is still a search-based approach, it involves many improvements over the integer and operation research approaches. Generate-and-test nature of problem solving in logic programming is greatly extended with constraint programming, giving it the efficiency of finding optimal solutions in short execution time [7, 9].

Timetable scheduling is a well-known instance of scheduling problems. There are many studies done on this subject and it still attracts many researchers, since it is one of the most challenging problems in the domain. As other scheduling problems, the timetable scheduling problem is also NP-complete [4]. In timetable scheduling, resources are instructors, classrooms, and groups of students and hours of the weeks. They should be allocated for lessons so that the preferences of a student and lecturer should be maximized (optimal solution) without conflicts on scheduling a room, instructor and student in the timetable [1, 10].

Although many similar systems had been developed so far, the main disadvantage of these systems is the lack of flexibility. In nearly all of these systems, users cannot define their own constraints. Hard coded constraints direct the entire search and may leave the user with some unwanted solutions. Giving a user the ability to define his/her own constraints would give more user satisfaction, as well as improve the solution quality.

The present paper considers the application of the hierarchy of constraints to the University timetabling problem. The hierarchy of constraints allows the users to specify their preferences according to which the system is looking for solutions that can satisfy most of the users.

2. Over-constrained systems and constraint hierarchy

Over-constrained systems are constraint satisfaction problems without a solution. In other words, for the variables of the systems, it is impossible to find valuations that exactly satisfy all of the problem constraints. Generally, real world problems fall into this category. In order to handle with this kind of problems, the constraint hierarchies are used. Instead of exact solutions, partial solutions are used that satisfy not all the constraints but different subsets of the given constraints.

In many applications, such as interactive graphics, planning, scheduling, document formatting, and decision support systems, users need to express

their preferences, as well as strict requirements. In such systems, expressing the preferences in the same way as the strict requirements generally results in insolvability of the problem. In order to overcome this situation, an arbitrary number of levels of preference are used, each successive level being more weakly preferred than the previous one.

Definition 2.1. A **labeled constraint** is a constraint labeled with strength. A symbol of a labeled constraint indicates where is the strength of the constraint and where is the constraint itself [3].

When writing a labeled constraint, we usually give symbolic names to different strengths of constraints. These symbolic names can be mapped onto integers $0, \ldots, n$, where n is the number of levels.

Constraints representing strict requirements are called *Hard Constraints*. Constraints representing preference are called *Soft Constraints*. Most of the systems and constraint programming languages allow users to define an arbitrary number of levels for preference, where each successive level being more weakly preferred than the previous one [5].

Definition 2.2. A **constraint hierarchy** H is a multiset of labeled constraints. Given a constraint hierarchy H, H_0 denotes the HARD constraints in H. In the same way, the sets $H_1, H_2, \ldots, H_n$ are defined for preference levels $1, 2, \ldots, n$ [3].

The greater the label is, the weaker the constraint is. A solution to the constraint satisfaction problem embodying constraint hierarchies is not the same as a solution to the constraint satisfaction problem without them. So, we need to revise our definition of a solution to the constraint satisfaction problem.

Now for each value of index n, let us define the proper sets of constraints $H_1, \ldots, H_n$.

H_0 is HARD Constraints,
$H_i = \{c \in H / s(c) = i\}, i \leq n$,
$H_k = \emptyset$ if $k > n$.

Definition 2.3. A solution S to a constraint hierarchy H is a set of valuations for free variables in constraints with the following properties:

- each valuation in S must be such that, after it is applied, all the HARD constraints hold;
- each valuation in S satisfies the non-required constraints as much as possible, respecting their relative strengths.

To formalize this, let us define a set S_0 of valuation such that all the H_0 constraints hold:

$$S_0 = \{\theta / \forall c \in H, c\theta = \textbf{true}\}.$$

Here $c\theta$ denotes the Boolean result of applying the valuation θ to the constraint c.

Then, using S_0, it is possible to define a solution set S by eliminating all potential valuations that are worse than some other valuation, using the special predicate *better* [3]:

$$S = \{\theta / \theta \in H \wedge \forall \sigma \in S_0 \neg \text{better}(\sigma, \theta, H)\}.$$

There are many different ways to specify the predicate *better* [9]. In our application, we will use the so-called *locally-better* predicate, when the system tries to satisfy as much soft constraints as possible, taking into account their strength.

3. Problem description and system input

The Computer Science Engineering Department of Dokuz Eylül University offers undergraduate, Master and PhD degrees to their students. Undergraduate program consists of several courses lectured during 4 years (or 8 semesters). In MS program, each student must take 24 credits for graduation. PhD program is normally a 4-year program.

The Computer Science Engineering Department has to prepare the timetables at the beginning of the semester. Every student normally enrolls to from 5 to 8 courses in one semester.

The Master program courses are lectured only two days of the week. So, these courses should be scheduled taking this constraint into account.

Each course consists of theoretical and practical sections. Each course is normally divided into 2 or 3 blocks of 2 hours (sometimes 3 hours) during the week.

The department building has 6 rooms. Only 3 of them are suitable for a large number of students. Elective courses have a relatively small number of students enrolled to, so, while preparing the timetable, small classrooms are assigned to these courses. Some classrooms can also be assigned to specific courses.

Computer Science Engineering Department currently employs 8 academic staff members and 9 research assistants to help the academic staff. Generally one lecturer conducts 3 lessons per semester. Because of the large number of lecture hours, preferences (preferred hours of the days for giving a lecture, the maximum number of lectures to be conducted per day, etc.) of the lecturers are taken into account while preparing the timetable.

The timetable problem discussed here is to generate a weekly timetable for the Computer Science Engineering Department, i.e. to schedule all the blocks of all the courses to the available classrooms and hours of the week.

The final result should satisfy the following constraints:

- Blocks of the same semester cannot overlap.
- Two blocks with the same lecturer cannot overlap.
- There cannot be overlapping blocks in the same classroom.
- Lecturers and Classroom of a block must be available for its whole duration.
- Blocks of a course should be scheduled on different days.

All the conditions described above make it very difficult to prepare a timetable for the department manually. Generally, the final timetable for the department is prepared after 4 weeks work. Therefore there is a need to develop an efficient system to solve the problem.

The system mainly contains 4 kinds of objects at the input stage of the program. They are:

- Lecturer Objects,
- Classroom Objects,
- Lesson Objects,
- Constraint Objects.

The relations between various classes at the input stage of the system can be seen in Figure 3.1.

As an example, let us consider the specification of the class Lecture (Figure 3.2).

A lecturer object is the entity, where all the information about each academic staff member is stored. This information is

- Name of the lecturer,
- Available hours of the lecturer.

A lecturer may be not available at the department during some periods of the week for some reasons. So, while entering the information to the system, these hours should be identified to the system in order not to have an invalid schedule, i.e., a schedule that employs a lecturer while he/she is not available in the department. A lecturer may also have some other preferences over days. We will consider them as soft constraints. For example, a lecturer can prefer morning classes. So, while these soft constraints are unimportant when we control the validity of a timetable, they can help us to evaluate

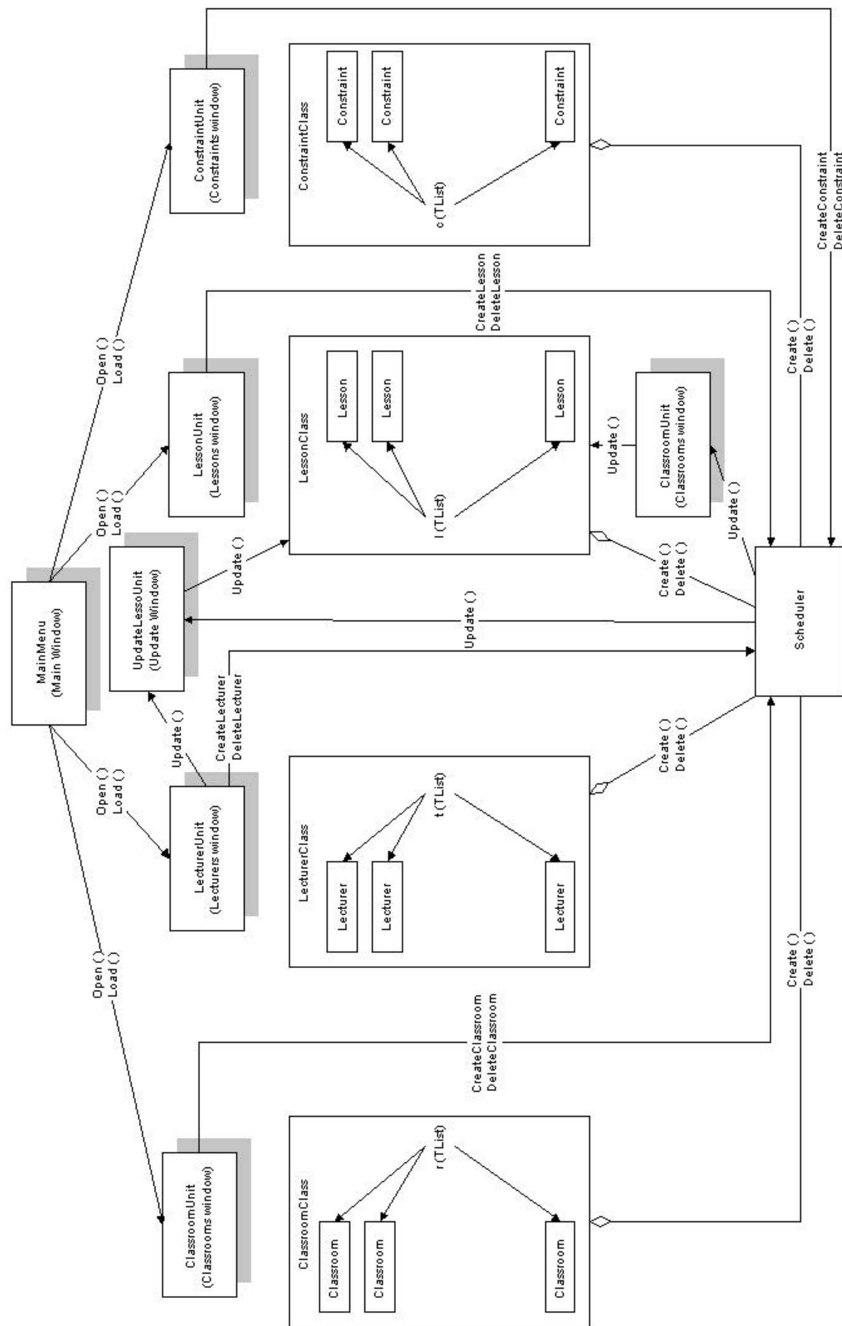

Figure 3.1. The system architecture for the input classes

Figure 3.2. Lecturers window

the final timetables, i.e. a timetable can be more preferable than the other, according to the number of satisfied constraints.

All this availability and preference information is stored in a 5×8 integer array. Each element of the array can take values in the interval $[-3, 2]$.

The value -3 in the array means that the lecturer is not available in the department during the corresponding hour. The values -2 and -1 correspond to unwillingness for lecturing a course during the corresponding hours. The scheduler will try not to assign lessons to hours having a preference of -2 unless there is no other choice.

The value 0 means "neutral" preference of the lecturer. Assignment of a lesson to this hour in the timetable gets no penalty, but it does not also increase the score of the solution found.

The values 1 and 2 in the array correspond to the wish to lecture a course during the marked hours of the timetable. The scheduler will first try to assign a lesson of the lecturer to the hour with the preference equal to 2. By doing this, the solution found will get bonus for making this assignment,

thus having a higher score. Different values of preferences are marked in the **Lectures** window by different colors.

All information related to the lecturers is taken from the *Lecturers window*.

In a similar way, other classes are defined.

The system has got some built-in constraints like "there cannot be overlapping lessons in the same classroom". Besides, a user can define other constraints of different weights. Objects belonging to *Constraint Class* store the user-defined constraints.

All the constraints in the system are binary, so a constraint is a relation between two lesson objects. Since constraint hierarchy is used in the implementation of the system, priorities of the constraints (in other words, their weight) should be defined.

4. Search and constraint solver

Constraint Solver converts all the inputs of the program into variables which should be evaluated later, specifies the domains of each variable, searches for a solution, evaluates the results and selects the best solution for the user.

All of the functions described above are implemented in a single class named *SchedulerClass* which contains multiple methods.

SchedulerClass directs the whole search process. Most of the updates over the input type objects are also carried out by this class. For example, when we delete a classroom from the system, all of the lesson instances should be checked and the domains containing the deleted classroom should be updated. In order to keep this job simple, all the lists containing pointers to different object types are kept in this class. When a change in one list occurs, it is very easy to go over the references of other lists and carry out the updates.

But the main duty of the class is to traverse the search space and find a solution of the problem.

4.1. Building a constraint graph

After each variable and its corresponding domain are created, the system builds the constraint graphs that will direct the search.

Two constraint graphs are used in the system. One of them contains the hard constraints with no weight and the second one contains the soft constraints with various weights.

Hard constraints are not handled in the constraint graph. Satisfaction of these constraints is guarantied by forward checking and arc-consistency methods [9].

After building the hard constraints defined in the system, the constraint objects, i.e. user-defined constraints, are to be checked. A user can also define his/her hard constraints. For example, although lessons A and B do not belong to the same semester and have different lecturers, a user may want them to be scheduled to different hours, because some students might be enrolled to both courses at the same time.

Hard constraint graph contains the constraints that directly conduct the search. If any inconsistencies are detected in the graph, a *backjump* occurs [9].

Hard constraints include only one predicate $EQUAL$ (as well as its negation, $\neg EQUAL$). The predicate $EQUAL(A, B)$ means that the lesson block A must be scheduled exactly on the same hours of the same day as the lesson block B.

Soft constraints are built using the information from the constraint objects and there can be some other soft user-defined constraints of various weights. Some scoring criteria, such as "blocks of the same lesson should not be scheduled to the days following each other", are used after a valid scheduling is found. Soft constraints do not direct the search process. Soft constraints are only used to score the solutions found. Thus, they are used in finding the optimal solution.

4.2. Ordering of the variables

Ordering of the problem variables is important when we need to find any solution to the Constraint Satisfaction Problem [9]. In this case, finding the first solution as soon as possible is the goal of the search process.

But the timetabling problem is the optimization one and we are looking for the best (optimal or near optimal) solution. To achieve this, the entire search tree should be inspected for solutions. Ordering of the variables is of no importance when the entire tree is to be traversed.

In the system, we employed *Fail First Principle* for variable ordering [9]. This is a general heuristic for search. It suggests that the variable which is likely to fail in labeling should be labeled first. By doing this, inconsistencies can be detected earlier.

The labeling complexity can be measured differently. For finite domains, the size of a variable domain is important. A variable with a smaller domain is likely to fail earlier when compared to a variable with a large domain. The number of constraints affecting the variable can also make the labeling difficult.

We made several tests employing both measures. In some problems, ordering according to the domains was performed well, while ordering according to the constraints was very poor in performance. But there were other problems, where ordering according to constraints was very good. The system currently uses the domain size measurement for variable ordering.

4.3. Search

The search algorithm is a recursive algorithm and it embodies *backjumping* and Branch&Bound techniques, which are two of the most efficient search methods in the area of scheduling, together with forward checking algorithm [7].

The algorithm continually tries to go deeper in the search tree if no conflicts arise.

First the algorithm checks, whether the lecturer of the currently scheduled block is overloaded for that day or not. To find it out, the algorithm calls a function *Lecturer_Over_Schedule*, which checks the previous schedules of the lecturer. If the lecturer is not overloaded, then forward checking occurs and domains of the other variables are reduced. If any future variable domain becomes empty, no further labeling will be done (forward checking). The algorithm rolls back and tries to make another schedule for the current block. If the domains of other variables are reduced successfully, then the partial solution score is computed. If it is higher than the bound (the best solution score), then the operations described above work for the next variable. If not, a new labeling is done for the current variable (Branch&Bound).

5. Results and performance

The system is developed under Borland Delphi 5.0 programming environment. The first tests were toy problems, which helped us to test the stability of the system and to inspect the behavior of the system under some certain constraints.

Real data obtained from the previous semesters were used in performance testing. These tests were also used to determine the quality of solutions found.

During the development, many methods had been tried to improve the search performance and the quality of the solution. We're running the system during 2 months to collect the results and, on this basis, to improve the heuristics that guide the search.

Although Branch&Bound and forward checking helps a lot to prune the search tree, the search process still needs 7 days. But the solution was found

after the 5th day. The rest of the search was just pruning. Since we cannot know whether the last found solution is the optimum, we cannot cut off the rest of the search, although it is sometimes useless to continue.

6. Conclusions and future work

Implementation of the final version of the system took 4 months. Most of this time was consumed in a search for good heuristic function parameters. Unlike most of the systems developed so far, this system tries to do a more complete scheduling for the university. It tries to allocate not only the blocks to the hours, but also the classrooms. When this is the case, the search tree grows wider, in other words, the number of candidate solutions increases, which in turn affects the search time in a negative way.

It is certain that the hardware development, especially the CPU technology, will help the search process to be carried out more effectively. But in a problem having a huge search space, relying on the power of hardware is not wise. The developed system works well and can handle many different situations. But this is not enough. More efforts, especially on the scoring mechanism, should be made. The possibility of a parallel search process should also be considered.

References

[1] F. Azevedo, P. Barahona, *Timetabling in Constraint Logic Programming*, Proc. of the 2nd World Congress on Expert Systems, 1994.

[2] P. Baptiste, C. Le Pape, W. Nuijten, *Incorporating Efficient Operations Research Algorithms in Constraint-Based Scheduling*, Proc. of the First International Joint Workshop on Artificial Intelligence and Operations Reseach, 1995.

[3] A. Borning, B. Freeman-Benson, M. Wilson, *Constraint Hierarchies*, LISP and Symbolic Computation: An International Journal, **5**, 1992, 223–270.

[4] T. B. Cooper, J. H. Kingston, *The Complexity of Timetable Construction Problems*, Proc. of the First International Conference on the Practice and Theory of Automated Timetabling (ICPTAT '95), 1995.

[5] B. N. Freeman-Benson, A. Borning, *Integrating Constraints with an Object-Oriented Language*, Proc. of the 1992 European Conference on Object-Oriented Programming, 268–286.

[6] J. I. Lustig, J. Puget, *Constraint Programming and its Relationship to Mathematical Programming*, The Institute for Operations Research and Management Science (INFORMS), Maryland. 2000.

[7] K. Marriott, P. Stuckey, *Programming with Constraints*, The MIT Press, Cambridge, UK, 1998.

[8] S. J. Russell, P. Norvig, *Artificial Intelligence — A Modern Approach*, Prentice Hall, 1995.

[9] E. Tsang, *Foundations of Constraint Satisfaction*, Academic Press, 1993.

[10] C. Yeung, S. Leung, H. Leung, *Applying Constraint Satisfaction Technique in University Timetable Scheduling*, The Practical Application of Prolog: Proc. of the 3rd International Conference on the Practical Application of Prolog, 1995, 683–695.