Applying Constraint Satisfaction Technique in University Timetable Scheduling

Chi-mei Yeung   Suet-mui Leung
Ho-fung Leung

Department of Computer Science
The Chinese University of Hong Kong
Shatin, New Territories
Hong Kong

Abstract

Planning and scheduling are often NP-complete problems. There have been different approaches to tackle these problems. With the emergence of constraint logic programming, the constraint satisfaction approach has started to attract attention due to its effectness in solving these real-life problems. In this paper, we report our experience of applying constraint logic programming to fully automate the time table scheduling process in our faculty, which has been a tedious task for the administration. The system consists of a core for scheduling as well as an X-window user interface. Implemented in CHIP, the system consists of 1500 lines of codes, 80% of which are an X-window user interface. The system is able to produce a time table for the whole faculty in just a few seconds. Our experience shows that applying constraint logic programming is a practical viable means for time table scheduling in a university.

1 Introduction

Planning and scheduling are often NP-complete problems. There have been different approaches to tackle these problems, such as the algorithmic approach and operational research approach. With the emergence of constraint logic programming [4], especially the introduction of finite domain constraints into logic programming [8], the constraint satisfaction approach has started to attract more and more attention due to its effectiveness in solving some real-life planning and scheduling problems [1, 2, 6].

In the paradigm of constraint logic programming, a constraint satisfaction problem can be written in the form of Horn clause logic programs in which the clause bodies may contain constraints. Constraints are generated incrementally during run-time and passed to a constraint solving mechanism which applies domain-dependent constraint satisfaction technique such as linear programming, local propagation, Boolean unification and so on to find a feasible solution for the accumulated constraints. Although such an approach is still search-based, it can often effectively reduce the search space in an *a prior* manner and improve on the inefficiency from the *generate-and-test* nature of problem solving in conventional logic programming.

Course timetable scheduling has been a tedious task for university administration. Its aim is to arrange periods and classrooms to all the courses in an academic year. Every year, administration and academic staff have to hold a lot of meetings before a timetable can be finalised. After a semester begins, amendment to time and place is usually encountered owing to the change in the number of students taking a course and the availability of teachers. We believe that the situation is equally bad at every university. Full automation of timetable scheduling is very much desirable as labour and time can be saved or spent on other administration work. Recently, we have attempted to apply the constraint satisfaction technique to solve the problem in the Engineering faculty. The result is satisfactory.

2 Problem Statement

Every year in our university, every individual department has to design a new timetable for courses in the coming academic year. In our university, a student can major in a subject and minor in another. Therefore courses are classified into major core course, major elective course, minor core course, minor elective course and other elective course.

Different kinds of courses are handled differently in scheduling a timetable. For example, major core courses should not overlap one another. It has been observed that there is a tendency in the combination of major and minor subjects. For instance, students majoring in statistics or mathematics like to minor in computer science. It is necessary for the courses of these subjects not to overlap one another as well.

A course usually consists of lectures of two or three hours per week and tutorials of one hour per week. Lecture time which lasts more than two hours is usually split into two sessions, one of one hour and one of two hours. Several constraints have to be considered. For example, different sessions are habitually assigned to different weekdays. Consecutive sessions should be allocated with the same place. Only one course is allowed at one period in one classroom. Tutorial time cannot coincide with lecture time.

Student quotas tell the maximum numbers of students taking the courses. They are used in allocating classrooms since we always want to optimise our resources. Class sizes of tutorials are usually much smaller than that of lectures. Thus, smaller classrooms are assigned to tutorials.

The availability of teachers is also our concern. Sessions taught by the same teacher must not overlap one another. Certain periods might not be available for some courses because some teacher might only be able to teach at some specific time.

Every department then sends its drafted timetable to the central administration for the allocation of classrooms. If any conflict arises, rearrangement of sessions of courses will be conducted until the designated number of periods and appropriate places are allocated to all the courses.

We have implemented our system on a SUN SPARCstation 10/30 in CHIP[1] [3, 7], a commercially available constraint logic programming system that is best known for its capacity in finite domain constraint solving. The scheduling system includes a constraint satisfaction system that searches for a feasible solution based on the given constraints as well as a user-friendly X-window based user interface. Our X-window user interface is implemented with the XGIP X11 graphics system in CHIP.

[1] We are using CHIP System Version 4.0.1.

3 Finite Domain Constraint Solving in CHIP

CHIP [7] is a commercially available constraint logic programming language that supports various domains of constraints. A CHIP program resembles a Prolog program, except that constraints are allowed in the clause bodies and programmers are allowed to specify the type of unification used in goal-head unifications.

A typical CHIP program for constraint satisfaction problems in finite domain [8] is in the following form

$$
\begin{aligned}
& top(\ldots) :- \\
& \quad constraint_1(\ldots), \\
& \quad constraint_2(\ldots), \\
& \quad \ldots, \\
& \quad constraint_n(\ldots), \\
& \quad labeling([X_1, X_2, \ldots]).
\end{aligned}
$$

where $constraint_i(\ldots)$ are CHIP procedures which specify constraints. A domain variable (d-variable) X is declared with the goal X :: D where D is the initial domain of X. CHIP supports several types of constraints in finite domain, such as equations (including d-unifications [8]), inequalities and disequalities.

Constraints are (partially) solved using forward-checking inference rule (FCIR) and partial looking-ahead inference rule (PLAIR) after they are incrementally generated. Finally, the built-in predicate $labeling$/1 assigns valid values to the unbound domain variables in the list $[X_1, X_2, \ldots]$, by searching the solution space.[2] This is called the 'labelling' process. It should be noted that the search space is reduced in an *a prior* manner [8] during searching in the labelling process: after each search step, the search space is reduced by actively using the unsolved constraints in the system before the next search step. Therefore the searching is much more efficient than that in pure Prolog.

[2]CHIP 4.0.1 provides a more sophisticated 4-argument version of $labeling$, which allows users to have more control on the labelling process.

4 Timetable Scheduling as a CSP

We take the following approach. A complete timetable is represented by a list, each element of which is a structure that represents a single course. The structure contains course code, teacher name, session pattern, teaching periods, corresponding classrooms, course quota and a list of invalid periods. A session pattern specifies the order of single and double periods. For instance, a possible pattern for a 3-session course may be a single period on a day followed by a double period on a later day. Teaching periods are denoted by a list of lecture time and tutorial time. Course quota is a list of two elements which are respectively number of students in a lecture and in a tutorial.

We number the slots in the timetable as slot 10, slot 11, ..., up to 64, from the first period on Monday to the last (fourth) period on Saturday. Classrooms are numbered according to their sizes. The smaller the size is, the smaller the number is. For a course of n periods of lecture, two lists of n domain variables $[Period_1, \ldots, Period_n]$ and $[Room_1, \ldots, Room_n]$ are used, which denote periods and the corresponding classrooms allocated respectively. The initial domain of $Room_i$ is determined by the quota of the course: if the class size is small, then virtually any classroom can be used; if the class size is very large, then only a small number of classrooms are useable. Remember that we assign a smaller classroom number to the smaller classrooms. This is in fact a heuristics to more fully utilise the classrooms. As the labelling process starts with small number, the small classrooms will be used first.

Several types of constraints govern the solution of our system. The teacher availability constraint can be expressed as

$$teaching\_time(Teacher, Dept\_timetable, Teaching\_time),$$
$$alldifferent(Teaching\_time)$$

where the predicate $teaching\_time$/3 gathers all teaching periods of $Teacher$ from the department timetable $Dept\_timetable$ and $alldifferent$/1 is a built-in predicate which specifies that all domain variables in the argument list should be different.

We use the same technique in the case of core courses.

$$clash(Group\_of\_courses, Timetable, Teaching\_time),$$
$$alldifferent(Teaching\_time)$$

where $Group\_of\_courses$ is a list of courses which are in advance stated by the user not to overlap one another. They can be courses within a department or from different departments.

Since lecture time cannot coincide with tutorial time, the program contains the following constraints as well.

$$append(Tutorial\_time, Lecture\_time, Teaching\_time),$$
$$alldifferent(Teaching\_time)$$

The technique is used again.

It is the user who determines the session patterns. A list is used to represent the pattern with each element denoting the number of consecutive periods in the session. The consecutive period constraint can be formulated as

$$\begin{aligned} P_i &= P_j + 1, \\ R_i &= R_j \end{aligned}$$

where P_i and P_j are periods and R_i and R_j their corresponding classrooms. Rooms should be the same for consecutive periods.

Since different sessions are habitually assigned to different weekdays, we introduce a new domain variable W_i for each P_i, which ranges from 1 to 6 representing Monday to Saturday respectively. W_i's and P_i's can be constrained by

$$\begin{aligned} 10W_i &\leq P_i \\ P_i &< 10W_i + 10 \end{aligned}$$

where W_i denotes the day of the period P_i. The equality

$$W_i = W_j$$

represents the days of consecutive periods as they should be on the same day. For different sessions, the disequality

$$W_i \neq W_j$$

is used.

There are two kinds of constraints involving rooms. If a session is scheduled in period P_i with classroom C_i and another session in period P_j with classroom C_j, C_i is required to be different from C_j only if P_i is known to be equal to P_j. We make use of a dummy variable M_i for each C_i with domain size equal to the product of number of rooms and number of periods in a timetable. It is obvious that all M_i must be different since a classroom cannot accommodate two lectures at the same time.

Invalid periods for a teacher should be excluded from the domain of d-variables for periods by disequality. Choice of periods of teachers can be achieved by assigning values to the d-variables directly. The assignment of special classrooms to courses can also be done in this way.

5 User Interface

The graphical user interface in Figure 1 consists of four main windows : initialisation window, input window, run window and print window.

'Initialisation' window in Figure 2 is for the input of invalid periods for the whole university.

'Input' window (Figure 3) allows users to input details of courses. Deletion of courses can be made in this stage in another screen (Figure 4) prompted when the button **Delete** is pressed. 'No clash window' in Figure 5 is a window for specifying courses which cannot overlap one another. All the courses input can be chosen from the menu to group together. Pressing the button **Invalid time** allows users to input invalid periods for a particular course.

'Run' window in Figure 6 shows the result of the program. A box represents a slot in the timetable. When a box is clicked, all the courses assigned to the corresponding slot are displayed. Users can also choose to display courses by teacher, by course or by department. 'Print' window in Figure 7 is a menu for users to print results in various ways.

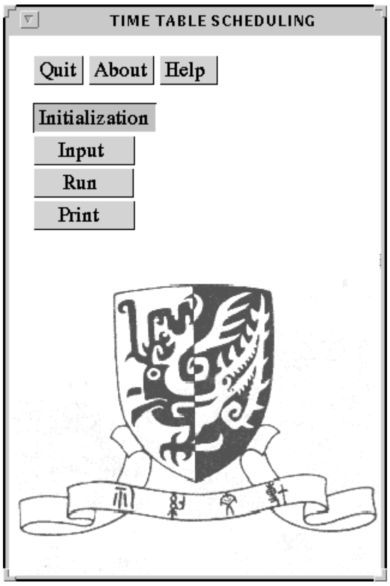

Figure 1: University Timetable Scheduling System

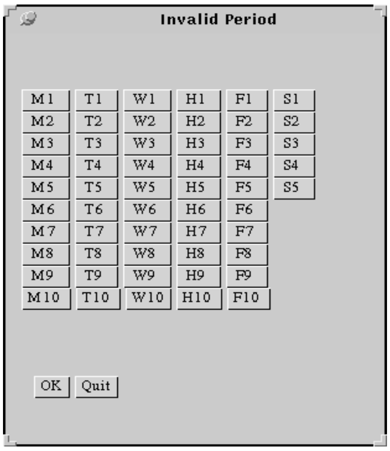

Figure 2: Initialisation Window

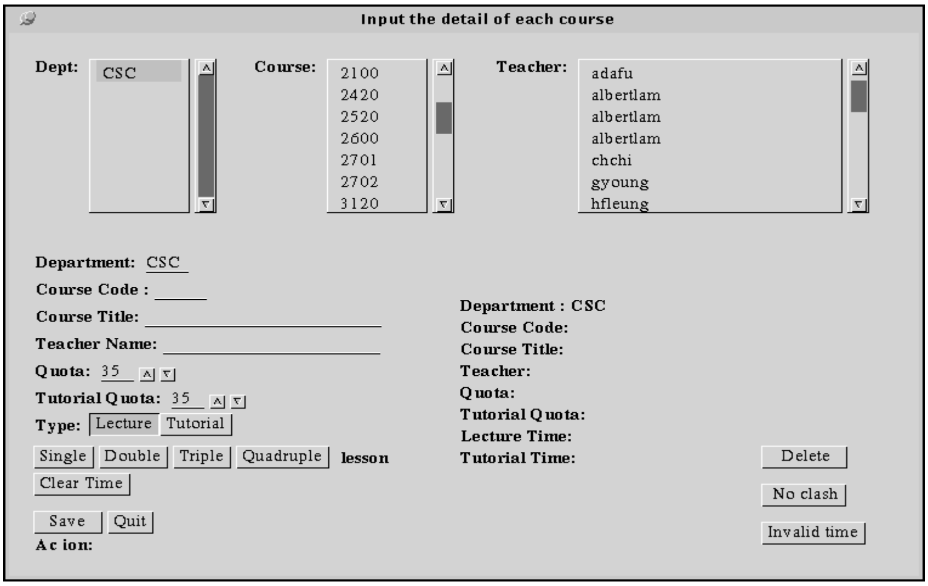

Figure 3: Input Window

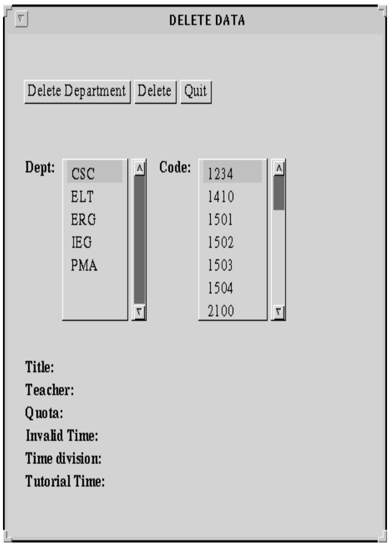

Figure 4: Delete Window

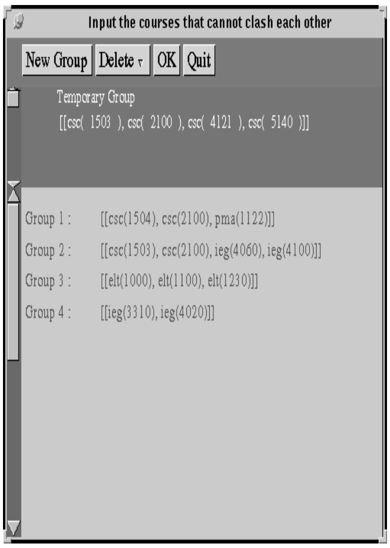

Figure 5: No clash Window

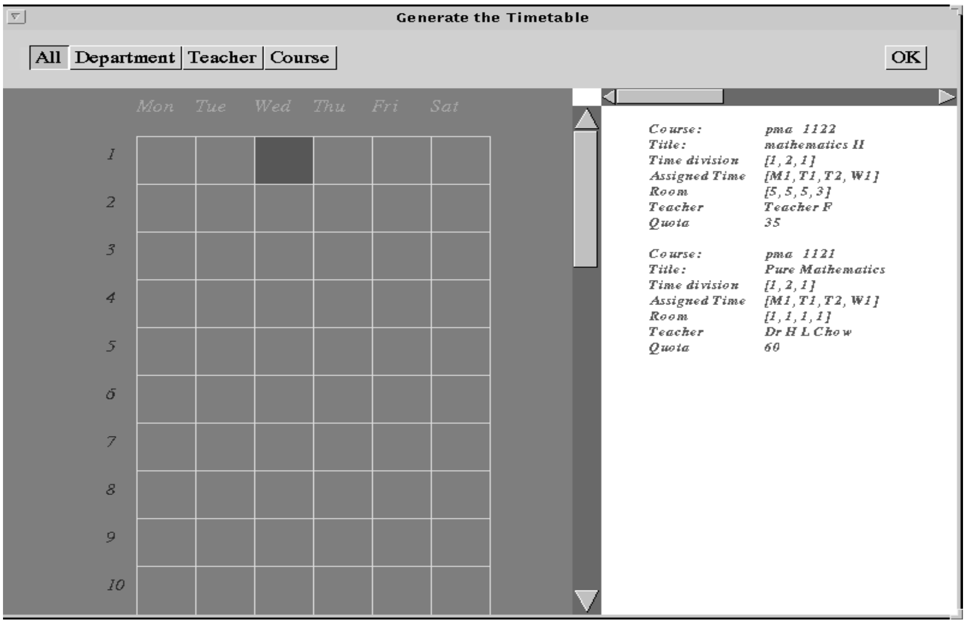

Figure 6: Run Window

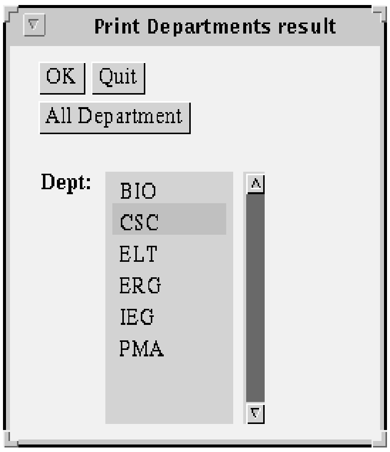

Figure 7: Print Window

6 Empirical Results

The following table shows the CPU time (in seconds) used to schedule the time table in our Faculty.

Number of Depts.	Number of Group of courses	Number of Courses	CPU Time (seconds)
1	5	26	0.840
2	8	35	1.280
3	11	47	2.180
4	14	57	2.830
5	18	72	3.840

The whole system is composed of 1500 lines of CHIP codes. About 80% of them are used for user interface. The system CATTS [5] for a similar purpose, which is implemented in C with algorithmic approach, needs 8000 lines of coding, half of which are for interface. Execution of our system is 5 times faster than CATTS.

7 Conclusion

The timetable scheduling problem is a NP-complete combinatorial problem. In this paper we reported a university timetable scheduling system based on finite domain constraint satisfaction technique. The system solves the problem efficiently. The development time for the program is much shorter than time spending on algorithmic approach. Our experience shows that applying constraint logic programming is a practical viable means for time table scheduling in a university.

References

[1] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving the Car Sequencing Problem in Constraint Logic Programming. In Y. Kodratoff, editor, *ECAI 88 : proceedings of the 8th European Conference on Artificial Intelligence*, pages 290–295, Munich, Germany, August 1988. Pitman Publishing.

[2] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving Large Combinatorial Problems in Logic Programming. *J. Logic Programming*, 8:75–93, 1990.

[3] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. In ICOT, editor, *Proceedings of the International Conference on Fifth Generation C omputer Systems 1988*, pages 693–702, Tokyo, Japan, 1988. ICOT, ICOT.

[4] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Conference record of the 14th Annual ACM Symposium on Principles of Programming Language*, pages 111–119. ACM, 1987.

[5] S. F. Koo and C. L. Lim. Computer-aided time-tabling system. Project Report 94/20, Department of Computer Science, The Chinese University of Hong Kong, Hong Kong, 1994.

[6] M. Perret. Using Constraint Logic Programming Techniques in Container Port Planning. *ICL Technical Journal*, pages 537–545, May 1991.

[7] The COSYTEC team. *CHIP Reference Manual*. COSYTEC SA, Orsay Cedex, France, 1993.

[8] P. Van Hentenryck. *Constraint satisfaction in logic programming*. MIT Press, Cambridge, United States, 1989.