Constraint-Based Timetabling - a Case Study

A. M. Abbas*
Department of Computer Science
University of Balamand
Tripoli, Lebanon
Email: abbas@balamand.edu.lb

E.P.K. Tsang †
Department of Computer Science
University of Essex
Colchester, CO4 3SQ, England
Email: edward@essex.ac.uk

Abstract

This paper reports a case study in applying Constraint-Satisfaction techniques to university and school timetabling. It involves the construction of a substantial, carefully specified, fully tested and fully operational system. The software engineering aspect of Constraint-Satisfaction is emphasized in this paper. Constraint-Satisfaction problems are expressed in a language more familiar to the formal software engineering community. This brings Constraint-Satisfaction one step closer to formal specification, program verification and transformation; issues extensively studied in software engineering. In problem formulation, explicit domain constraints and heuristic information are made explicit. Moreover, the user's needs are considered more closely; for instance, when the program fails to find a solution, useful indications are produced to help in relaxation or reformulation of the problem.

Keywords: *Constraint-Satisfaction, timetabling, software engineering.*

1. Introduction

Timetabling is an instance of task scheduling, which is a practical and hard problem that attracted researchers from a variety of different backgrounds and disciplines [3]. The study of the timetable generation process is important because of the sensitive role timetables play specifically in the management of people intensive institutions such as hospitals and universities.

This process is looked at in the literature from many different angles (see [8] for a good overview and more references). Correspondingly, there have been many approaches to the realization of these views using a wide variety of problem-solving paradigms (see [9] and [13] for sample instances). In comparison, constraint-satisfaction techniques figure prominently.

2. The Context

This paper reports a case study in the application of Constraint-Satisfaction techniques to timetabling, in the context of a small and young university, where students come from a variety of different backgrounds.

This context provides a particularly fitting test for Constraint-Satisfaction techniques, because students taking the same kind of courses are seldom a significant majority in any one class. Without taking this variety into consideration, a timetable will very likely prevent a meaningful number of students from taking intended courses due to time conflict.

In this context, students go at the start of every semester through a *pre-registration* period, where they choose the classes they would like to attend during the semester. Administrators would then gather enrollment information and try to piece together a timetable that would suit instructors, students and other university requirements, resources and facilities.

The task of producing by hand, a satisfactory, conflict-free timetable is always a tedious, time-consuming process. The main difficulty is the absence of any dependable idea for doing that, beyond an exhaustive search of all available alternatives. This means plenty of trial-and-error attempts. What makes matters worse is that, in the absence of adequate planning, the essentially random pre-registration process

*Contact Author

†Partially sponsored by EPSRC funds GR/H75275, GR/L20122, GR/M46297 and a Research Promotion Fund from the University of Essex

would very often result in unresolvable conflicts between various student choices.

3. A Software Engineering Approach to Constraint Satisfaction

The Timetable Generator project was initiated in response to this problem. The first period of the project was spent investigating the possibility of benefiting from several techniques such as the best-first search algorithm. However, later implementations had more to gain from Constraint-Satisfaction techniques [10], thus resulting in a much more usable program.

Most research on Constraint-Satisfaction is focused in algorithms. In fact, Constraint Programming is already mature enough for real life applications [12]. Therefore, more attention should be paid to its software engineering aspect, in which very little has been done apart from the CHIC project [1, 11].

In this context, we shall introduce a formal language for specifying Constraint-Satisfaction problems. This will be used to formalize timetabling as a Constraint-Satisfaction problem before deciding on an algorithm for its solution. This approach carries more weight specifically in the context of ongoing research on high-level Constraint-based programming [2, 4, 7].

3.1. Specifying A Constraint-Satisfaction Problem

A Constraint-Satisfaction Problem (CSP) can be defined by the following three components (definition modified from [10]):

- a finite set of variables $A = \{a_1, a_2, \cdots, a_n\}$.
- a collection $B = (B_a)_{a \in A}$, each of its elements is a finite set of values of arbitrary types that the corresponding variable a is associated with.
- a set S of subsets of A over which a collection of constraints $C = (C_s)_{s \in S}$ is specified. Each constraint C_s ties together the variables of the subset s in such a way as to restrict the values they may simultaneously take.

A CSP solution has to find, for each of the variables $a \in A$, a value $b \in B_a$ satisfying all relevant constraints; that is, the set of pairs $\{< a, b >; a \in s \wedge b \in B_a\}$ must satisfy the constraint C_s, for each $s \in S$. An accurate description of the task that reflects this simultaneity would be to require the construction of a function f that associates every element $a \in A$ with an element $b \in B_a$ in such a way that $C(f)$ is satisfied:

$$CSP \equiv (\exists f \in A \rightarrow B) C(f)$$

3.2. Solving A Constraint-Satisfaction Problem

A general algorithm that can solve this class of problems starts from the CSP specification above and, at the end, returns a result that is a *solution* or a $failure$.

$$Result \equiv Solution \vee Failure$$

A *solution* would be the function f mentioned in the CSP specification above. However, since A is assumed to be finite, this function may also be viewed as a list of pairs $< a, b > \in A \times B_a$, chronologically ordered according to the time they are generated (earliest first). On the other hand, a $failure$ arises when no solution can be generated because the constraints are too tight or contradictory.

```
//A: variables, B: values, C: constraints
Result SCHEDULE (A, B, C) {
// the timetable initially empty
timetable = ∅;
// {x ∈ A; ¬scheduled(x)} ≠ ∅ initially
scheduling = true;
// select a variable
a = highestpriority({x ∈ A; ¬ scheduled(x)});
while (scheduling) ∧({x ∈ A; ¬scheduled(x)} ≠ ∅){
// select a value
b = mostsuitable({y ∈ B_a; ¬considered(y)
      ∧ C(timetable+ < a, y >)});
// Successful selection
if found(b) {
   add(< a, b >, timetable);
   a = highestpriority( {x ∈ A; ¬scheduled(x)});}
   // Unsuccessful selection, dead end,
else {
   // select a backtracking point
   t = mostsensible({x ∈ A; scheduled(x)});
   if found(t) { // backtracking point found
      erase(a .. t, timetable); a = t;}
   else scheduling = false;}} // general failure
return(( {x ∈ A; ¬scheduled(x)} = ∅)? timetable :
failure);
}
```

A General CSP Algorithm

The pseudo code above basically implements Gaschnig's BackJumping algorithm [5]. However, a couple of remarks are due to clarify the underlying assumptions:

- For any given $a \in A$ and during the process of locating b, constraints are checked with respect to

already scheduled elements of A only. When b is found, it is marked as *considered* to leave it out of consideration, in case of backtracking for an alternative value in B_a. Moreover, unsuccessful attempts at locating b are recorded on a *failure table* as it is used in Gaschnig's Backmarking algorithm [5]. This table is used, in case backtracking is required, for determining the most sensible point t to backtrack to. This *failure table* is reconstructed from scratch for every new $a \in A$, and the size of this table is of the order of the size of B_a.

- The existence of this *failure table* reduces the process of finding t to a simple linear scan. After finding t, all elements $a' \in A$ between a and t have to be marked as *not scheduled* and all the b's in the corresponding $B_{a'}$'s as *not considered*. These B'_a are in fact going to be searched again. This clearly has negative effects on the efficiency of the algorithm. The extent of that may be reduced by an intelligent choice of the next a to schedule [6].

Any further elaboration on the above outline will have to await domain-specific information with regard to the variables, values and constraints (see below).

4. The Timetabling Problem Specification

On the basis of the discussion of the previous sections, the requirements of the problem can be captured in the following specification:

$$Timetabling \equiv$$
$$(\exists f \in Courses \rightarrow Times) Suitable(f)$$

$$Suitable(f) \equiv$$
$$(\forall c, c' \in Courses)(c \neq c') \Rightarrow \neg Conflict(c, c', f)$$

$$Conflict(c, c', f) \equiv$$
$$(\exists x \in Students) c \in s(x) \wedge c' \in s(x)$$
$$\wedge < c, b > \in f \wedge < c', b' > \in f \wedge b \cap b' \neq \emptyset$$

where s is a function returning the set of courses taken by the student x.

4.1. Variables and Values

The variables are obviously the courses on offer. In case the same course is given to more than a single group of students (i.e. a multiple-section course), each section will need to be considered as a distinct variable. This makes sense because each section will generally have its own distinct timetabling requirements. Moreover, the role of an instructor vis-à-vis the timetable generation process is no different from that of a student. For this reason, we will only mention students without any loss of generality.

The space of values is the space of all available teaching times during the week. The unit value is a time *slot*. A time *slot* is defined as a time interval over one day of the week. This is specified by a triplet: $< d, t, h >$, where d is a number denoting one day of the week, t is a number denoting the starting time of the slot and h is the *length* of a slot; i.e. number of half-hours this slot consists of. For example, the triplet $< 1, 1, 3 >$ denotes the time interval on $Monday$, between $8:00AM$ and $9:30AM$. Hence, the first constraint that a slot value $< d, t, h >$ should respect is:

$$C_0 \equiv WD(d) \wedge ([t..t+h] \subseteq WH(d)\}$$

Where $WD(d)$ means that d is a working day and $WH(d)$ represents the set of working hours of day d.

4.2. Composite Values

Each course is associated with a number of credits. This number is used to indicate the number of teaching hours associated with this course, the number of slots that those hours may be distributed over, and the constraints this distribution has to respect.

Credit No.	Hrs. No.	Slot No.	Slots length
1	3	1	6
2	2	1 ∨ 2	4 ∨ (2,2)
3	3	2 ∨ 3	(3,3) ∨ (2,4) ∨ (2,2,2)
4	4	2 ∨ 3	(4,4) ∨ (2,2,4) ∨ (2,3,3)

Course-Slot Distribution Table

Thus, depending on its number of credits, a course might be associated with a composite value (i.e. a set of slots). For example, a 1-credit course is given over a single 3-hour slot. A 2-credit course is given over one 2-hour slot or two 1-hour slots. A 3-credit course can be given over two 1.5-hour slots or three 1-hour slots, etc. A 4-credit course is given two 2-hour slots, etc (see table above).

This table is used to make sure that only slots of the right size are ever considered for courses of a given type: i.e. number of credits. Hence the second constraint:

$C_1 \equiv size(slot)$

Multiple-slot courses are subject to an additional constraint that any two different slots $< d, t, h >$ and $< d', t', h' >$, associated with the same course, should start at the same time during the day. Moreover, these slots should be separated by a gap of one day at least:

$$C_2 \equiv (< d, t, h > \neq < d', t', h' >) \Rightarrow (t = t')$$

$$C_3 \equiv (< d, t, h > \neq < d', t', h' >) \Rightarrow (\|d - d'\| > 1)$$

4.3. Ordering and Generators

Seeing the complexity of these values and in order to make their management easier, we impose a total ordering on them. Given two slots $< d, t, h >$ and $< d', t', h' >$, we define:

$$(< d, t, h > \leq < d', t', h' >) \equiv (d < d') \vee ((d = d')((t \leq t'))$$

Such an ordering may be used as the basis of a value generator (VG). This approach has a number of advantages, since the values of a domain will generally be too numerous to store explicitly. For instance, this value generator may be used to produce a next value of a slot every time this value is needed. Accordingly, earlier values are *considered* first. Furthermore, when a current value is being *considered*, earlier ones are implicitly marked as *considered* and later ones are still as yet *not considered*.

In addition to that, each slot of each course can have its own generator (SG), which produces values of the appropriate type. Obviously, to be useful, this generator should be able to generate all valid values without neither missing nor repeating any.

Following this line of reasoning, any single course (a slot set) constitutes a miniature-timetabling problem, having its own variables, values and constraints. Thus, a miniature program (SSG) has been constructed. This can generate a composite value for each such a set, respecting all relevant constraints.

4.4. The Constraints

A timetable is considered unsuitable if there any student is enrolled in different courses whose times overlap, or if this course overlaps with the times this student cannot attend the course. Such information comes from a set Σ of records for every student of the institution. Each record is a pair $< cs, ss >$ defined as follows:

- cs is the set of courses taken by the student.
- ss is the set of slots during which the student cannot attend classes.

A course generator (CG), that produces the highest-priority course to schedule, is assumed here. The ordering relation ($<$) over courses is defined with respect to the time at which each such course is so produced. Details of this generator will be explained in the next section.

Now, a slot s being considered for a course c will cause no conflict if the following conditions hold for every record $< cs, ss > \in \Sigma$:

$$C_4 \equiv (c \in cs) \Rightarrow (s \cap ss = \emptyset)$$

$$C_5 \equiv (c \in cs) \Rightarrow (\forall c' \in cs(c' < c) \Rightarrow (\forall s' \in time(c')(s \cap s') = \emptyset)$$

Here, the function $time$, applied to an already scheduled course, will return the set of slots allocated for this course. We also have the following definitions:

$$(s \cap ss) = \emptyset \equiv \forall s' \in ss((s \cap s') = \emptyset)$$

$$< d, t, h > \cap < d', t', h' >= \emptyset \equiv (d = d') \Rightarrow max(t, t') \geq min(t + h, t' + h')$$

Another constraint that should be satisfied by a slot is that no student should have too much load on any single day. That is, the total load per day should not exceed a certain given maximum M. That is, a slot $< d, t, h >$ for a course c causes no conflict if the following condition holds for every record $< cs, ss > \in \Sigma$:

$$C_6 \equiv (c \in cs) \Rightarrow sum(\{length_d(c'); c' \in cs(c' \leq c\}) \leq M$$

where $length_d(c) \equiv (\exists < d, t, h > \in time(c)?\ h\ : 0)$

5. Details of the Timetabling Algorithm

One major task the algorithm is doing at every iteration is choosing the next course to schedule. A good choice here will obviously have a major impact on its global efficiency. The general strategy is based on choosing the course judged hardest to schedule; i.e. the course with the tightest constraints. The following are the factors that are judged to reduce most the ease with which a course c is scheduled:

- the number of slots this course is currently taking: $size(time(c))$
- the length of time this course will occupy during the week: $sum(\{h; < d, t, h > \in time(c)\})$
- the total number of student records this course belongs to: $size(\{cs; < cs, ss > \in \Sigma \wedge c \in cs\})$
- the number of courses belonging to each one of these records: $sum(\{size(cs); < cs, ss > \in \Sigma \wedge c \in cs\})$
- the length of time constraints associated with each one of these records: $sum(\{sum(\{h; < d, t, h > \in ss\}); < cs, ss > \in \Sigma \wedge c \in cs\})$

5.1. The Backtracking Scheme

The current path the algorithm is following to schedule the current course may reach a dead end. Repairing this failure can be attempted if it can be traced back to a decision related to a previously scheduled course. This way, undoing that decision might make further progress possible. Here, we are interested in the most critical pamount of work left to be done, before the final solution is oint to backtrack to. This is for the sake of minimizing the reached, or before hope in finding any solution is totally lost.

When selecting a fresh course c to schedule, a fresh *failure table* is created along with it. The number of entries of this table is the number of all possible slot values associated with this course. Now, while scheduling the course and for each slot value that is considered, the corresponding entry on the table will have records of all courses whose interaction with c caused that entry to fail.

5.2. The Backtracking Point

When c fails to schedule, we go through a minimization process that leaves, at each entry of the table, the earliest of all courses listed at that entry, then, through another maximization process, we go through all entries determining the latest of all courses left on the table. If found, that will be the course to backtrack to. This can be nicely expressed by the following:

$$max(\{ \\ min(\{c'; (c' < c) \wedge (v \cap time(c') \neq \emptyset)\}); \\ v \in \cup\{time_s; s \in time(c)\}\})$$

Where $time_s$ is used to denote the set of all possible values of the slot s (see [10], for more details on this $minmax$ process).

5.3. Constraint Ordering

A closer look at the CSP algorithm reveals that almost all the work that it is doing is spent on Constraint Verification. Therefore, at least an equivalent amount of effort should be invested in optimizing Constraint Verification.

The insight into that resides in the following key idea: since local failure is unavoidable in general, it is best to catch it early. In fact, we have seven key constraints: $C_0, C_1, \cdots, C_6$ are being used, through a chain of processes, exactly to filter out values from the initial rough domain of all possible slot values, till these become valid course times at the end.

In order to optimize the amount of work done to achieve this task, the application of these constraints along this chain is ordered as follows: $C_0, C_1, C_3, C_2, C_6, C_4$ and C_5. The guiding intuition behind this ordering is: a bad value, travelling along this chain, should be detected and eliminated from further consideration as early as possible, and with the least amount of computation.

6. Constraint Optimization

The list Σ of student records, fed to the program, does not guarantee that the underlying constraints are automatically satisfiable. In such a situation, the program will simply return $failure$. For these reasons, we looked into the idea of a timetable with tolerable conflicts. The idea is to attach weights (i.e. costs) to each of the constraints. After that, the objective of the algorithm is altered in the following way: instead of having to satisfy all constraints, a course is added to the timetable if its conflict costs are affordable.

7. Program efficiency

The general CSP algorithm implements BackJumping [5], which is complete for satisfiability problems, in the sense that it will not miss a solution if one exists. In theory, the algorithm has a worst-case situation, in the sense that it can sometimes run for too long before it gives a solution or before it can say that no solution exists. Two escape routes have been devised to avoid this situation:

- Plenty of care is paid to the design of the initial input data, in order to eliminate from considera-

tion many of the intractable situations that may arise in practice.

- An escape option that allows the program to exit without a solution (if that appears to be taking too long) but with useful indications supplied to ease the constraints for a more successful run.

8. Concluding Summary

This paper reports a successful application of constraint technology. The software engineering aspect of Constraint-Satisfaction is emphasized in this project. We have taken a formal approach to specify a timetabling problem. A university and a school have used the timetable generation program presented in this paper, which is well tested and fully operational. The problem that we have specified is general enough, and therefore our experience should be useful to other researchers with similar applications.

9. Acknowledgment

The first author would like to thank staff and students at the faculty of sciences and engineering at the University of Balamand, for all the patience they have shown bearing the consequences of the system's output during its early years.

References

[1] The chic-2 project. http://www-icparc.doc.ic.ac.uk/chic2/.

[2] A. Abbas and E. Tsang. Toward a general language for the specification of constraint-satisfaction problems. *CP-AI-OR'01*, April 2001.

[3] E. Burke and M. E. Carter. *The Practice and Theory of Automated Timetabling*, volume Vol. 1 and 2. Springer Lecture Notes in Computer Science Series, 1996 and 1998.

[4] P. Flener, B. Hnich, and Z. Kiziltan. Compiling high-level type constructors in constraint programming. *PADL'01 (Practical Aspects of Declarative Languages)*, 2001.

[5] J. Gaschnig. Experimental case studies of backtrack vs. waltz-type vs. new algorithms for satisfying-assignment problems. *Proc. 2nd National Conference of the Canadian Society for Computational Studies of Intelligence*, pages 19–21, 1978.

[6] M. L. Ginsberg. Dynamic backtracking. *Artificial Intelligence Research*, 1, 1993.

[7] B. Hnich and P. Flener. High-level reformulation of constraint programs. *JFPLC'01*, 2001.

[8] L. P. Reis and E. Oliveira. A language for specifying complete timetabling problems. *PATAT2000 Proceedings*, August 2000. Burke and Erben (Eds).

[9] M. A. Trick. A schedule-then-break approach to sports timetabling. *PATAT2000 Proceedings*, August 2000. Burke and Erben (Eds).

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

[11] E. Tsang, P. Mills, R. Williams, F. J., and J. Borrett. A computer aided constraint programming system. *PACLP*, pages 81–93, 1999.

[12] M. Wallace. Practical applications of constraint programming. *Journal of Constraints*, 1(1 and 2):139–168, 1996.

[13] G. M. White and B. S. Xie. Examination timetables and tabu search with longer term memory. *PATAT2000 Proceedings*, August 2000. Burke and Erben (Eds).