A Heuristic Incremental Modeling Approach to Course Timetabling

Don Banks[1], Peter van Beek[1], and Amnon Meisels[2]

[1] Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada T6G 2H1
{banks,vanbeek}@cs.ualberta.ca
[2] Department of Mathematics and Computer Science
Ben-Gurion University of the Negev
Beer-Sheva, Israel 84105
am@cs.bgu.ac.il

Abstract. The general timetabling problem is an assignment of activities to fixed time intervals, adhering to a predefined set of resource availabilities. Timetabling problems are difficult to solve and can be extremely time-consuming without some computer assistance. In this paper the application of constraint-based reasoning to timetable generation is examined. Specifically, we consider how a timetabling problem can be represented as a Constraint Satisfaction Problem (CSP), and propose an algorithm for its solution which improves upon the basic idea of backtracking. Normally, when a backtracking routine fails to find a solution, there is nothing of value returned to the user; however, our algorithm extends this process by iteratively adding constraints to the CSP representation. A generalized random model of timetabling problems is proposed. This model creates a diverse range of problem instances, which are used to verify our search algorithm and identify the characteristics of difficult timetabling problems.

1 Introduction

Timetabling problems arise in many real world situations. Although many computerized techniques exist for timetable construction, obtaining acceptable results is often difficult. In this paper we address the problem of constructing a master timetable of multiple-section courses, along with scheduling students into sections of their requested courses. Thus the global problem consists of two subproblems—often called in the literature the master timetabling subproblem and the student sectioning (or grouping) subproblem. The two subproblems have been addressed separately in the past (see, for example, [1, 9, 10] and references therein). More recently, methods have been proposed which solve the two subproblems in tandem. Aubin and Ferland [1] propose an iterative method to solving the global problem which alternately solves the two subproblems until no further improvement to the solution can be found. The method is heuristic

and is not guaranteed to find a global optimum. Hertz [6] adopts the solution method of Aubin and Ferland and shows how tabu search techniques can be used to potentially find an improved optimum.

In recent years, constraint-based reasoning has gained much attention in the Artificial Intelligence community. Previous constraint-based approaches to timetabling only address one of the two subproblems. For example, Meisels et al. [8] and Yoshikawa et al. [13] consider a high school timetabling problem where each class of students remains together for the entire term and Lajos [7] and Henz and Wurtz [5] address the master timetabling subproblem in a university setting. Feldman and Golumbic [3] address the use of priority constraints in their CSP solution to the student sectioning or grouping subproblem. Chan, Lau, and Sheung [2] present a constraint-based approach to time-tabling that iteratively relaxes some constraints and then "repairs."

In this paper, we propose a novel constraint-based model and solution technique to the global problem for modeling general timetabling problems found in high schools in Edmonton, Alberta, Canada. The problem involves creating a master timetable of multiple-section courses and creating individual student schedules. The schools being studied have eight periods per week, and the weekly schedule remains constant throughout the term. Courses are divided into sections of students, and each section is scheduled into its own period. The input to the problem consists of a list of students' course selections. The objective is to satisfy as many of the student course selections as possible by scheduling the sections of the courses into the eight weekly periods subject to a limit on the size of the classes, limits on the number of teachers of each subject, and a limit on the total number of rooms in the school.

In our CSP model of the problem, we define binary constraints to occur between pairs of courses chosen by the same students, and non-binary constraints to occur over groups of courses to enforce constraints on available teachers and rooms. A novel feature of our CSP model is that our model is heuristic. Satisfying all of the constraints does not guarantee 100% students satisfaction. While the non-binary teacher and room constraints are exact in the model, the domains of the variables are drastically pruned before solving and the binary constraints are heuristic estimations of complex non-binary constraints. However, we find that adding all of the possible binary estimation constraints between pairs of chosen courses often results in no solution existing. The proposed solution method heuristically and incrementally models and solves a timetabling problem. By prioritizing the various binary estimation constraints, our algorithm will iteratively add the constraints to the problem, building upon the solution until no further improvements can be made.

Our algorithm, which iteratively adds constraints before proceeding with a backtracking search, should be contrasted with (i) Aubin and Ferland [1] who iterate between the two subproblems of assigning students to the generated master schedule and *regenerating* a master schedule (that solves also the conflicts that result from the assignment of students to the last master schedule), and (ii) Chan, Lau, and Sheung [2] who iteratively *relax* constraints rather than strengthen

constraints. Our approach can also be contrasted with a widely used optimization technique for CSPs which incrementally adds a constraint which enforces a new cost bound and resolves the model [12, p.94]. In our method all potential constraints are generated by the students' selection of courses. The iterations are generated by incrementally adding more constraints (from a *fixed set*) to the problem. A constraint weighting function is required to decide which constraints will satisfy the most students. A threshold value is used to determine the minimum weighting to include the binary constraints, and a forward checking routine is used to solve the specified CSP—which implies assigning periods to sections—and a master timetable is specified. Then, the procedure iterates and raises the threshold to include more binary constraints, which should increase the number of students successfully scheduled. The iterations continue until too many binary constraints have been added and no solution can be found. In further contrast to Aubin and Ferland, in their method the problem is *modeled* exactly using an integer linear programming formulation and then is *solved* heuristically. In our method, the problem is modeled heuristically—in that we add constraints that prune away large parts of the search space (and may prune away an optimal solution)—and then is solved exactly. Similar to Aubin and Ferland, we do not both model and solve our model exactly because it is computationally infeasible to do so [1].

We demonstrate the robustness of our algorithm by defining a random timetabling problem generator, based on actual high school data, and then solving a diverse range of timetabling problems. We show that in the majority of test cases, our algorithm can satisfactorily schedule 98% of the students on the randomly generated data and the amount of computational time required by the algorithm is reasonable. Furthermore, the generator helps us to identify difficult to solve timetabling problems. These problems can then be recreated for testing on future timetabling algorithms.

The software we have developed is a batch scheduling system. We envision the current scheduler as being part of an interactive decision support system where it would provide the initial schedule which the user could then analyze and modify (see, for example, [4]).

2 The Timetabling Problem

The high school timetabling problems that are the center of the present investigation arise from schools in Edmonton, Alberta, Canada. They are large problems ranging in size from 250 to 2200 students and having between 45 and 400 courses to timetable. A *course* is defined as a group of students who meet three times per week for instruction. There is one and only one teacher assigned to instruct each of these three lessons. The students have much freedom in the courses they will take. Each student provides the school with a list of desired courses four months in advance of the school year. Students have individualized schedules based on the courses that they select.

	Mon.	Tue.	Wed.	Thr.	Fri.
9:00	1	2	1	2	1
10:00	3	4	3	3	2
11:00	5	5	4	5	4
13:00	6	7	6	6	7
14:00	8	8	7	8	–

Fig. 1. Sample blank timetable

There are 8 periods in each week, with each period divided into 3 non-overlapping time slots of one hour in duration. Each day has 5 time slots. A sample blank timetable is shown in Fig. 1. If a student is scheduled to have Chemistry 30 in period 1, she would attend this class every Monday, Wednesday and Friday at 9:00 am. A student may register in at most 8 different courses in a term, but may register in fewer.

Some courses may be in high demand and hence be divided into multiple sections, where each section refers to a specific group of students who will always meet at the same period for the lesson. Most courses have between 1 and 4 sections, but some, such as English 10, have as many as 13. Each section will then be assigned a period number of 1 through 8. Therefore, part of the goal of this timetabling problem is, given all of the student's requests, assign a period number from 1 to 8 to each section of each course, and assign the students to a section for each course chosen, such that no student is assigned to more than one section in the same period. An important constraint is that no more than 30 students may be scheduled into one particular section. Another constraint is that we may not have more courses scheduled in any one period than there are rooms in the school.

The other piece of the puzzle is scheduling the teachers. However, the actual process of determining the teachers' schedules is not a part of the problem at hand, since the administration of the high school prefers to do this themselves. Nonetheless, one of our goals is to have a final solution which *guarantees* that valid teachers' schedules can be generated from the resulting master timetable. Two constraints are that no teacher may teach two lessons at the same time and there is one and only one teacher for every section. Another constraint is that teachers are only available to teach certain subjects. For example, suppose there are exactly 3 Physics teachers. A constraint would therefore assert that

no more than 3 Physics courses can be scheduled in the same period, otherwise there would not be enough teachers for all of them.

The overall terminology and definition of the problem is as follows:

- A school has n students, r rooms, and offers c courses from s subjects.
- Every course C_i, $1 \leq i \leq c$, has an associated subject. For example, Math 10 through Math 33, all belong to the subject, Math.
- Every course C_i, $1 \leq i \leq c$ is subdivided into sections. The local policy is that a section of a course can have at most 30 students in it. Thus, the number of sections needed for a course is determined by dividing the number of students enrolled in the course by 30 and rounding up to the nearest integer.
- Each subject S_i, $1 \leq i \leq s$, has a number of available teachers T_i.
- Each student submits a list of desired courses L_i, $1 \leq i \leq n$, where each $L_i \subset \{C_1, \ldots, C_c\}$.

The problem is to generate a master schedule which assigns each section of each course to a period from $1 \ldots 8$. Additionally, we desire the n individual student timetables to be generated from the resulting assignment of periods to sections, such that each list of course selections L_i is satisfied. This goal is also subject to the constraint that no more than 30 students can be scheduled into one section. The main constraint, that the students must have the courses they have chosen available to them, is of the not-equals variety (e.g. there must be sections of all these courses that are *not assigned equal* periods). However, there are also two distinct capacity constraints in this problem: (i) the total number of sections of each subject S_i must not exceed T_i during any period, and (ii) the total number of sections must not exceed r during any period. The quality of a master schedule is measured by how many students have conflict free schedules. Any solution is guaranteed to have conflict free schedules for the teachers.

3 A Constraint Satisfaction Model

The constraint satisfaction model is a simple means of representing a wide variety of problems. The CSP has three components: *variables*, *domains*, and *constraints*. Each CSP consists of a set of variables $\{x_1 \ldots x_n\}$, each with an associated domain of values $D_1 \ldots D_n$. A solution to a constraint satisfaction problem is an instantiation of each variable to one particular value from its domain, such that none of the constraints are violated. Constraints, therefore, are relations between variables which describe their legal values. For example, suppose variable x has the domain $\{1, 2, 5\}$ and variable y has the domain $\{2, 3, 4\}$. A *binary* constraint—one that proposes the valid instantiations between *two* variables—may exist which says that $x \neq y$. Not all constraints are binary. A *unary* constraint is one that applies to a single variable. A non-binary constraint which includes all n variables of the CSP is known as a *global* constraint.

In order to formulate the timetabling problem as a CSP, we must define what are to be the variables, domains, and constraints.

3.1 Variables

Each variable represents a course C_i, such as Math 10, and all sections of that course. Associated with each variable or course is a number of attributes including the number of students enrolled in that course and the subject S_i of that course. Assigning a value to a variable represents assigning a time period to each section of that course.

3.2 Domains

The domains of the variables each consist of an m-tuple of periods, where m is the number of sections of each course. Each tuple consists of m periods in the range $\{1 \ldots 8\}$. Therefore, in the general case a 3-sections course would have the domain values $(1,1,1), (1,1,2), (1,1,3), \ldots$ and so on. The result of this choice is that domain sizes becomes an unmanageable 8^m, where m is the number of sections of the course. We can remove equivalent solutions by enforcing that the periods occur in ascending order since master timetables that differ only in that they swap two sections of the same course are equivalent. Further, a natural heuristic for the domains we have discovered is to exclude the possibility of duplicate periods appearing in the corresponding tuple of values. If we enforce that no period's value appears more than once in each permutation, then the domain values for the 3-section course become $(1,2,3), (1,2,4), (1,2,5), \ldots (6,7,8)$. The domain size in this case would be 56, instead of 512. The natural heuristic of disallowing multiple sections in the same period reduces the size of the domains from 8^m to at most 70, and therefore the total search space is reduced exponentially, albeit at the expense of potentially ruling out valid solutions. The maximum domain size of 70 occurs when the number of sections is four. This heuristic directly corresponds to what the actual high school schedulers do; only rarely will a course be "doubled up" in the same period.

A variable representing a course with 8 sections would have a domain size of 1, simply containing $(1,2,3,4,5,6,7,8)$. In the event that the course has more than 8 sections, some overlap is impossible to avoid. So, in this case we assume that the first 8 sections of the course have the implicit $(1,\ldots,8)$ distribution, while the remaining sections obey the non-overlap rule.

3.3 Constraints

Student Course Selection Constraints. The given input to the problem consists of the student course selections. The resulting final timetable must be one that somehow has the necessary available sections open for all of the student requests.

We propose a binary constraint which will *estimate* the section assignment needs, the *subset* constraint. Subset constraints occur between courses in the same term. This constraint between two courses says that one course's permutation of period values cannot be a subset of the other. The idea of this constraint is to avoid a student being left with the same period as the only open time for

two courses that he/she is registered for. For example, suppose a student chooses three courses, A, B and C, the first two having two sections, and the latter with one section. If both A and B are given the permutation (4,7), one might conclude that this was fine, the student could take A in period 4, and B in period 7. However, if C were now given the value (4), this is not acceptable. Thus, between *one* pair of these courses there is a subset constraint necessary, which would deem that the two courses' permutation may not be equal to or a subset of the other. With just one subset constraint, together with the natural heuristic of avoiding duplicate courses, we have now guaranteed that the assignment of periods to sections for these three courses will satisfy the student. The general rule for applying estimator constraints is given below, together with examples.

> **Estimator Rule:** If a student selects a course of n sections, and also selects d courses (in the same term) with n or fewer sections, $d \geq n$, the n-section course is subset-constrained by $d - n + 1$ of the other courses.

Example 1. A student picks 3 courses, A, B, and C. A has 1 section, B has 2, C has 3. No binary constraints are needed. Any combination of values for A,B and C will allow the student to attend all three courses. Applying the rule specifically to course C, n is equal 3 sections, but d is equal to 2 sections with 3 or fewer courses. Since d is less than n, the estimator rule does not apply.

Example 2. A student picks 5 courses, A through E. A, B, and C each have 3 sections, D has 2, E has 1. Subset constraints are needed between A-B, A-C and B-C. These will guarantee that 5 different periods will appear in the permutations of A, B and C. D and E could then be anything, and are not constrained.

Example 3. A student picks 8 courses, A through H. Course A has 7 sections, while the rest have only 1. Course A must be constrained with one of the other courses, while each pair of courses B through H must be constrained, for a total of twenty-two binary constraints.

In the third example, the algorithm was left with a seemingly arbitrary choice of which single section course to constrain with course A. However, this selection need not be done randomly. Our algorithm would first check if there are any existing constraints, from previously examined student selections, between course A and courses B through H. If there are any, no new constraint would be needed involving course A. Otherwise, the selection may be made by choosing the course B through H which has the *most* students also registered in course A.

Non-binary Constraints. The non-binary constraints are not meant to be estimations, instead they are exact. All of the non-binary constraints are included in any solution attempt. There exists a non-binary teacher's constraint, which is designed to ensure the final solution will allow for successful scheduling of the teachers. There is one such constraint for each of the sixteen subjects, and the constraint covers all of the courses in each subject. If there are T_i full-time

teachers for a given subject, this constraint says that there may not be more than T_i courses of the subject scheduled at one particular period. The other non-binary constraint is the global room constraint. This constraint is meant to enforce that the school cannot exceed its' capacity. If there are r rooms, then there may not be more than r courses scheduled during one particular period.

4 Solving the High School Timetabling Problem

A CSP with no solution is highly undesirable, since there is nothing of value returned to the scheduler. We need some additional rules which dictate the number of constraints to be used by the CSP solver routine. The algorithm we have developed operates iteratively. The CSP begins with no binary constraints at all, and some instantiation of the variables is found which satisfies the non-binary and unary constraints. The process continues by adding binary constraints that pertain to courses of one section and later to courses with more sections. In other words, the CSP is "repaired" by adding more binary constraints. This value, one section, may be thought of as a threshold. The threshold will be incremented as necessary until there is either 100 percent student satisfaction or no solution is found—and the algorithm halts. The general pseudocode of the algorithm is summarized below.

TIMETABLING ALGORITHM

0. Determine enrollment matrix
1. Initialize thresholds
2. **Repeat**
3. GenerateCSP(threshold)
4. **For** i=1 to $n_{attempts}$ **Do**
5. Randomly order domains
6. Heuristically order variables
7. Solve CSP
8. Schedule students
9. Add constraints by increasing thresholds
10. **Until** no solution exists, or n_{sat}% of the students are scheduled

Step 0. Determine Enrollment matrix. The enrollment matrix is used to determine which pairs of courses are taken together by students, and the frequency of these combinations.

Step 3. GenerateCSP(threshold). This step determines which binary constraints are included in the problem, based on the current thresholds. The algorithm considers adding a binary constraint between two courses only if one or more students has requested that pair of courses. The decision of whether or not to include a binary constraint is based on how many sections each of the courses has and on how many students have requested that pair of courses. Initially, binary constraints are only added between courses that have a single section and which have high demand. On the next iteration, binary constraints

are added between single section courses that have moderate demand. On the iteration after that, those with low demand. On the iteration after that, binary constraints are added between courses that have one or two sections and high demand, and so on.

Step 4. For i = 1 to $n_{attempts}$. Within this algorithm lies a loop that randomizes the domains and solves the CSP, $n_{attempts}$ number of times. For our experiments, $n_{attempts}$ was set at two.

Step 6. Order Variables. The order of the variables refers to the order in which the backtracking routine will assign values to the variables. A good heuristic ordering of the variables can greatly reduce the cost of finding a solution [11]. In our work, the best ordering strategy that has been found is by domain size, smallest to largest.

Step 7. Solve CSP. This step involves most of the computational time and could be any existing CSP algorithm. For our experiments we use a backtracking algorithm known as *forward-checking*. If no solution is found (either because it is proven that no solution exists or because some predefined time limit has been exceeded), for *all* of the $n_{attempts}$ CSPs at the current threshold settings, the program terminates, since too many constraints have been added. (Proceeding to add more binary constraints to an over-constrained problem could not possibly help). The solution found which satisfies the most students course selections is then returned.

Step 8. Schedule the students. After the CSP has been solved, and we are left with a master timetable, the individual students must be scheduled so that their course requests are met. We have determined that the ordering of the students to be scheduled can make a difference. We choose to order the *most* difficult to schedule students first; a "difficult" student to schedule is one who chooses many courses with few sections in them, thereby causing less flexibility in the student's timetable. The number of successfully scheduled students is recorded. If this value exceeds the previous best, the solution is saved, and will be available once the program terminates.

Step 9. Adding constraints by increasing the thresholds. The thresholds are manipulated in order to increase the number of constraints that are in the CSP, and hopefully yield a better solution.

Step 10. Program halting criteria. The algorithm halts once no further improvements can be made. The user can also specify a pre-desired student satisfaction rate, which also results in program termination, once attained. For our experiments, n_{sat} was 99%. In reality, high school timetablers in large schools are pleased to reach 95% student satisfaction.

5 Experimental Results

We gathered data from three local high schools, but we did not proceed to solve the actual high school timetabling problems as some of the data was incomplete and unexplained. As well, each school had their own "exceptions" to the scheduling process. For example, some schools have half credit and double credit

courses that last a single term rather than the full year; some courses are taken by correspondence and not actually attended at the school, and one school has an International Baccalaureate Program whose students are treated differently in the scheduling process. The local high school timetablers solve the problem by hand on a school by school basis, a project with requires many hours of effort beginning months in advance of the new school year.

The approach we took was to create an abstract random model of the high school timetabling problem based on courses that are all full year and one credit. Our goal was to create a wide range of realistic data in order to examine what effect varying the parameters of our problem has on the time required to solve the problem and the overall quality (number of students scheduled) of the solution. A further motivation in creating a random problem generator is in identifying "hard" timetabling problem instances. Our random problem generator is able to create a broad range of timetabling problems, including some that are particularly difficult to solve.

We first identified three critical parameters of the high school timetabling problem at hand: n, the numbers of students in the school; c, the number of courses offered by the school; and r, the number of rooms in the school. We varied the number of students from 750 to 2000 by increments of 250, and the number of courses from 100 to 200 by 50. The number of rooms is a function of the capacity of the school. Given the n students, who choose from the c courses, the minimum value of r is the smallest number of rooms which can accommodate all of the resulting classes. The actual value of r will include "spare" rooms, varied at 0.0%, 2.5%, 5.0% and 7.5% of the minimum in our experiments.

To generate a random timetabling problem, one needs a random set of student course selections. From these selections, the constraints can be formulated, as described in Section 3. The student course selections are modeled by four discrete random variables. There is one random variable to model the grade that the student is in (grade 10, 11, or 12) and one to model the number of courses selected (between 1 and 16). Students from a particular grade are not limited to choosing courses of their grade level. The frequency of the random course selections, as well as frequency of students choosing particular courses at a higher or lower grade level, were estimated from data supplied by local high schools.

Some of our experimental results on our random timetabling model are shown in Table 1. The values shown in the table are the averages of 100 experiments at each of the different values of n, the number of students, c, the number of courses, and r, the number of rooms. For all experiments, the timetabling algorithm was run with $n_{attempts}$ equal to 10 (see Step 4, Section 4), and the algorithm halted if a predefined limit of 100 backtracks was reached (see Step 7) or the number of satisfied students, n_{sat}, was greater than or equal to 99% (see Step 10).

The results are encouraging. In the case where the percentage of spare rooms was 2.5% or greater, which are the realistic cases, we found a master timetable that satisfied 98% of the students in approximately two-thirds of the experiments. The most difficult to solve problems occurred when the number of courses offered was high in comparison to the number of students, such as when 750 stu-

Table 1. Effect of varying number of students, number of courses, and percentage of spare rooms, on percentage of students satisfied. Each data point is the average of 100 trials on random problems. In all trials, the algorithm was terminated when 99% student satisfaction was attained.

(a) Percentage of spare rooms is 0%

	Students					
Courses	750	1000	1250	1500	1750	2000
100	88.6	84.5	94.3	84.7	53.0	62.0
150	72.7	78.9	70.9	81.2	45.7	38.9
200	67.5	67.8	72.9	75.7	47.4	39.6

(b) Percentage of spare rooms is 2.5%

	Students					
Courses	750	1000	1250	1500	1750	2000
100	91.4	98.9	99.2	99.6	99.9	99.9
150	82.1	95.2	99.1	99.1	99.3	99.7
200	75.8	85.7	94.5	98.9	99.1	97.2

dents could select from 200 different courses. In this case, the number of courses with just one section of students (less than thirty enrolled) is high. These courses are the most difficult to schedule, since there is the least flexibility for the students, as they are only offered at one time. Because of the Estimation Rule, these single section courses participate in the most binary subset constraints. In general, as the number of courses (variables in the CSP) increased, the quality of the solutions decreased.

Further results can be seen in viewing the data as the number of students in the random model is varied. In general, as the number of students increase, the overall quality of the solutions also increased. This trend is also a consequence of having fewer single section courses. Having more students in each course results in more sections being allotted, which gives greater flexibility as the students are scheduled into courses.

Varying the number of courses and students has a direct implication on the number of binary constraints in the problem. In order to manipulate the *non-binary* constraints, we have also varied the number of rooms in the school. The global capacity constraint says that no more than r courses can be scheduled at one period, given that there are r rooms in the school. As the number of rooms decreases, the quality of the solutions decrease, because in some cases no solution exists at all. For example, when the percentage of spare rooms is 0.0%, the average success rate of the algorithm on a school with 2000 students and 200 courses is 39.6%. However, of the 100 experiments on random problems at these settings, 60 of the experiments terminated with no initial solution found, and in that case the recorded result was 0% student satisfaction. No statistically significant difference was found between the results for 2.5% spare rooms and the results for 5.0% and 7.5%. This is somewhat surprising since for the smaller problems where the minimum number of rooms is around 25, 2.5% spare rooms only adds one room slack, and for the larger problems where the minimum number of rooms is around 60, 2.5% spare rooms only adds two rooms slack. Nevertheless, it was found that having this few of a number of "spare" rooms was sufficient and greatly improved the quality of the master schedule found by the algorithm as measured by the number of satisfied students.

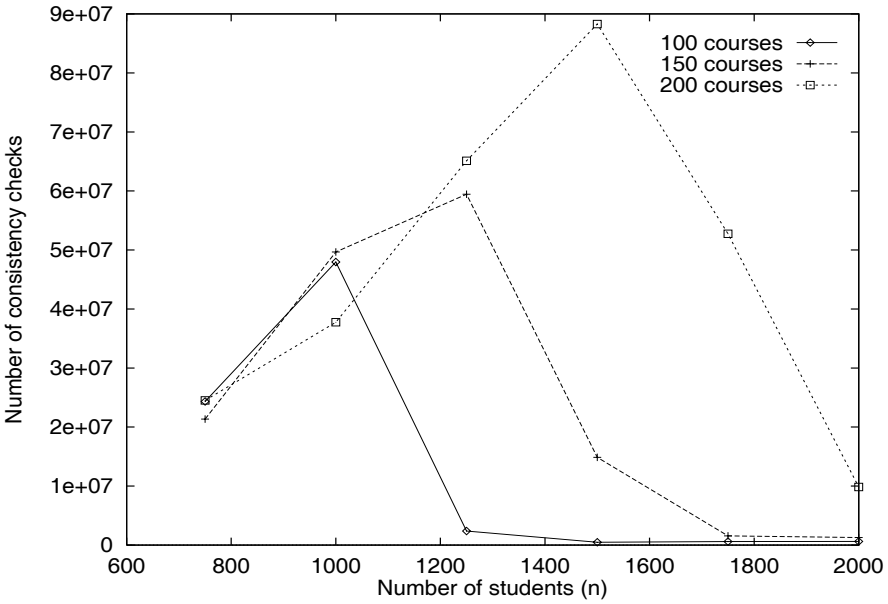

Fig. 2. Effect of number of students on number of consistency checks. For these experiments, the parameters are $c = 100$, 150 and 200 courses, and $r = R_{min} + 2.5\%$ rooms. Each data point represents the average of 100 trials.

The experimental results are also encouraging for their low required run-time. Each solution attempt, whether terminating with no solution, or completing with a master timetable and individual timetables, required less than one hour on a Sun SS4/70-32. The most difficult problems, found when the number of courses was 200 and the number of students was 1500, required approximately 90 million consistency checks on average to reach 99% student satisfaction. On these problems the average run-time was 2107 seconds and the hardest problem took 3404 seconds. The experiments with poor results (less than 80% satisfaction) actually terminated quite quickly, requiring between one and two million consistency checks. The easiest problems occurred when the number of courses was low, and the number of students was high. These problems achieved 99% student satisfaction in roughly two hundred thousand consistency checks—less than five minutes of run-time. Therefore, we conclude that some problems are naturally easy, while in some cases good solutions can be found with more iterations of adding binary constraints. The average number of consistency checks is displayed in Fig. 2. We found that in our experiments the number of consistency checks was an extremely good predictor of run-time and that a plot of average run-time gives the same qualitative shape as in Fig. 2. Thus, for each value c, number of courses, there is an associated number of students n where a peak in problem difficulty is observed.

6 Conclusions

We have proposed a constraint satisfaction model for a local high school timetabling problem. The problem we have studied involves an assignment of eight weekly periods to the sections of school courses. Each of the students provides a list of course selections which are to be satisfied. The sections of the course are scheduled such that there is at least one section for each course available to the student. In the case where there was a small number of spare rooms, which is the realistic case, we found a master timetable that satisfied 98% of the students in two-thirds of the experiments on a random testbed of timetabling problems.

The innovation in our solution method is in the heuristic modeling of the problem—which prunes away large parts of the search space—and in the process of *iteratively* adding constraints to the network. Iterative solutions may be poor at first but will improve to some upper limit, until no solution can be found. The main advantage of our iterative method is there will always be a timetable output to the scheduler, unless the school has too few rooms to accommodate all of the courses. Because of the iterative constraint addition, a "best" solution always exists at any point in the search. The students are individually scheduled using a simple greedy algorithm once the master timetable is completed.

Finally, we have proposed a random model of the school timetabling problem. By identifying the three critical parameters of number students, number of classrooms and number of courses, we have created a diverse testbed of realistic timetabling problems. Furthermore, we have identified some particularly difficult to solve timetabling instances. These particular instances can be *recreated*, by giving the generator the same parameters and same random seed, so that comparisons can be made with other, improved timetabling algorithms.

References

1. J. Aubin and J.A. Ferland. A large scale timetabling problem. *Computers & Operations Research*, 16:67–77, 1989.
2. H. W. Chan, C. K. Lau, and J. Sheung. Practical school timetabling: A hybrid approach using solution synthesis and iterative repair. In *Proceedings of the Second International Conference on the Practice and Theory of Automated Timetabling*, pages 123–131, Toronto, Canada, 1997.
3. R. Feldman and M. C. Golumbic. Constraint satisfiability algorithms for interactive student scheduling. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 1010–1016, Detroit, Mich., 1989.
4. J. A. Ferland and C. Fleurent. SAPHIR: A decision support system for course scheduling. *INTERFACES*, 24:105–115, 1994.
5. M. Henz and J. Wurtz. Using Oz for college time tabling. In *Proceedings of the First International Conference on the Practice and Theory of Automated Timetabling*, pages 162–180, 1995. Available as: Springer Lecture Notes in Computer Science 1153.
6. A. Hertz. Tabu search for large scale timetabling problems. *European Journal of Operational Research*, 54:39–47, 1991.

7. G. Lajos. Complete university modular timetabling using constraint logic programming. In *Proceedings of the First International Conference on the Practice and Theory of Automated Timetabling*, pages 146–161, 1995. Available as: Springer Lecture Notes in Computer Science 1153.
8. A. Meisels, J. El-Saana, and E. Gudes. Comments on CSP algorithms applied to timetabling. Technical report, Department of Mathematics and Computer Science, Ben-Gurion University, 1993.
9. A. Schaerf. A survey of automated timetabling. Technical Report Report CS-R9567, Centrum voor Wiskund en Infirmatica (CWI), Amsterdam, The Netherlands, 1996. To appear in *Artificial Intelligence Review*.
10. G. Schmidt and T. Strohlein. Timetable construction — an annotated bibliography. *The Computer Journal*, 23:307–316, 1979.
11. E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.
12. P. van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, Cambridge, Massachusetts, 1989.
13. M. Yoshikawa, K. Kaneko, Y. Nomura, and M. Watanabe. A constraint-based approach to high-school timetabling problems: A case study. In *Proceedings of the Twelfth National Conference on AI*, pages 1111–1116, Seattle, Wash., 1994.