An Investigation of the Course-Section Assignment Problem

Zeki Bayram

Eastern Mediterranean University,
Computer Engineering Department,
Famagusta, T.R.N. Cyprus
`zeki.bayram@emu.edu.tr`
`http://cmpe.emu.edu.tr/bayram`

Abstract. We investigate the problem of enumerating schedules, consisting of course-section assignments, in increasing order of the number of conflicts they contain. We define the problem formally, and then present an algorithm that systematically enumerates solutions for it. The algorithm uses backtracking to perform a depth-first search of the implicit search space defined by the problem, pruning the search space when possible. We derive a mathematical formula for the algorithm's average-case time complexity using a probabilistic approach, and also give a brief overview of its implementation in a WEB application.

1 Introduction

In this paper we deal with the course-section assignment problem that arises in university settings. Although the investigated problem is specific to Eastern Mediterranean University (North Cyprus), the results can easily be generalized to other university contexts.

At Eastern Mediterranean University, classes are taught Mondays through Fridays, between 8:30 a.m. and 4:30 p.m. A given course is opened in one or more sections. Each section of a course meets for 3 or 4 hours each week. The job of the student advisor during the registration period is to decide which courses the student should take, and find a set of sections for those courses such that the number of conflicts is minimized.

The problem we investigate is enumerating, in increasing order of the number of conflicts, course-section assignment schedules, given an initial list of courses that the student should take. The student advisor can then select the schedule s/he sees fit.

Course scheduling problems, when formulated as decision problems, are known to be NP-complete or even NP-hard [1], and finding the optimal solution to the problem is computationally intractable as the input sizes become large. The course-section assignment variation of the course scheduling problem is a function problem, and is at least as hard as the decision version ("Is there a course section assignment with k conflicts or less?"). However, in the real-life case we handle here, the maximum input sizes to the problem have a reasonable upper bound (i.e. a student can take only a limited number of courses which the advisor selects for him/her, and each course has a limited number of sections that are open at any one time) and an approach that systematically but intelligently and incrementally searches the full space of possibilities becomes feasible.

Assuming that the maximum number of courses a student takes is N and the maximum number of sections a course can have is K, then at most K^N combinations of course sections must be considered. But even then, generating all combinations of course groups at one time and then sorting them by the number of conflicts they contain can be prohibitively expensive, both in terms of time and space. Instead, the search space should be pruned when possible, and the solutions must be *incrementally* generated, in increasing order of the number of conflicts they contain. When enough solutions (defined as the best X solutions, where X is specified by the user) have been generated, the algorithm should stop.

The algorithm we describe does precisely that, and it is fast enough to be executed in a WEB browser's JavaScript engine.

The remainder of this paper is organized as follows. Section 2 formally defines the "course-section assignment problem." Section 3 contains an algorithm that systematically generates course-section assignments in increasing order of the number of conflicts. In section 4 we perform a mathematical analysis of the worst-case and average-case time complexity of this algorithm using combinatorial arguments. Section 5 gives a brief discussion of the issues concerning the JavaScript implementation of the algorithm. This is followed in section 6 by a representative survey of other approaches to course scheduling, and finally in section 7 we have the conclusion and future research directions.

2 Formal Definition of the Course-Section Assignment Problem

In this section we formally define what we mean by the "course-section assignment problem."

Definition 1. *A* `meeting-time` *is a day-period pair, such as* $< Monday, 3 >$, *meaning the third period (i.e. 10:30) on Monday.*

Definition 2. *The function* `rep(D,P)` *is defined as* $(val(D) * 8) + P$, *where* $< D, P >$ *is a meeting-time and* `val` *is a function mapping each working day to its position in the week, starting from 0, e.g.* $val(Monday) = 0$, $val(Tuesday) = 1$ *etc. Consequently, rep(D,P) is a unique integer representation of a meeting-time* $< D, P >$. *No two distinct meeting-times have the same integer representation, since there are exactly* 8 *periods every day.*

Definition 3. *A* `course-section assignment` *is a function that maps a course to one of its sections.*

Definition 4. *The function* `meetingTimes(C,S)` *returns the set of meeting times of section S of course C. Formally, $x \in meetingTimes(C, S)$ iff $< D, P >$ is a meeting-time of section S of course C and $x = rep(D, P)$.*

Definition 5. *The function* `nconf(assign)` *takes a course-section assignment as an argument and returns the number of conflicts it contains. Specifically, let $< C_1, \ldots, C_n >$ be a list of courses and* `assign` *be a course-section assignment. $nconf(assign)$ is defined as*

$$|meetingTimes(C_1, assign(C_1))| + \ldots + |meetingTimes(C_n, assign(C_n))| - \\ |meetingTimes(C_1, assign(C_1)) \cup \ldots \cup meetingTimes(C_n, assign(C_n))|$$

Definition 6. *Given a list of courses $< C_1, \ldots, C_n >$, the* `course-section assignment problem` *is to generate a sequence $< ass_1, ass_2, \ldots >$ of course-section assignments in such a way that every assignment appears exactly once in the sequence, and if ass_i comes before ass_j in the sequence, then $nconf(ass_i) \leq nconf(ass_j)$.*

3 The Scheduling Algorithm

3.1 Data Representation

We represent the times at which a course section is taught with a bitmap consisting of 40 bits (5 days, 8 periods, one bit for each $Day - Period$ combination). A "1" in a position means that the course is taught at that time slot, and a "0" that it is not. The first 8 bits in the bitmap are used for the 8 time slots on Monday, the next 8 bits for Tuesday etc. To determine whether two course sections conflict, we just $\bigwedge$ (logical "and") the corresponding bitmaps, and if the result is other than 0, then they conflict. Using this representation, we can also determine the number of conflicts by counting the number of 1's in the result (this can be done in constant time using using a lookup table t where $t[i]$ contains the number of 1's in the binary representation of i [2]). Using bitmaps with logical "or" and "and" operations, determining whether n course sections conflict has $O(n)$ time complexity.

3.2 The Algorithm in Pseudo-code

The algorithm in figure 1 finds course-section assignments in increasing order of the number of conflicts by traversing in a depth-first fashion an implicit tree whose nodes consist of a bitmap representing the day-time slots taken up by the courses considered so far. The root of this tree is always a bitmap of 40 bits that contains all "0"s.

The main identifiers in the algorithm are as follows. $current[i]$ contains the bitmap of slots taken up by $course_1$ through $course_i$. The sections of courses that are selected as we travel down the implicit tree are stored in the $result$ array, i.e. $result[i]$ contains the section selected for $course_i$. $next[i]$ contains the

Input:

1. C, the maximum number of conflicts that can be tolerated.
2. List of courses $course_1, course_2, \ldots, course_L$ for which we need to find schedules with at most C conflicts.
3. K, the number of sections per course.
4. The function $meeting\_times(i, j)$ that gives the bitmap for course i, section j.
5. $max\_solutions$, the maximum number of solutions that should be generated

Output: All combinations of course sections such that the number of conflicts does not exceed C and solutions are generated in increasing order of the number of conflicts they contain, up to a maximum of $max\_solutions$ solutions

```
declare $current$ as an $array[0..L]$ of bitmaps        // each one 40 bits long
declare $nc$ as an $array[0..L]$ of $integer$           // to store the number of conflicts
declare $next$ as an $array[1..L+1]$ of $integer$       // to store the choice points for backtracking
declare $result$ as an $array[1..L]$ of $integer$       // to store the selected sections
$ns \leftarrow 0$                                       // number of solutions generated so far
for $c \leftarrow 0$ to $C$
    $i \leftarrow 1$                                    // the next course to process
    $next[1] \leftarrow 1$                              // section 1 of course 1 to be processed first
    $current[0] \leftarrow (000000\ldots)$              // bitmap of forty zeroes
    $nc[0] \leftarrow 0$                                // initial node contains no conflicts
    loop
        if $ns > max\_solutions$ then exit program end if
        if $i = 0$ then exit loop end if                // traversed whole tree for current value of $c$
        if $next[i] > K$ then
            $i \leftarrow i - 1$                        // backtrack to previous course
            continue loop
        end if
        if $i = L + 1$ then                             // processed all courses
            if $c = nc[i-1]$ then                       // check for exact number of conflicts
                print the $result$ array
                $ns \leftarrow ns + 1$                  // update number of solutions found
            end if
            $i \leftarrow i - 1$                        // backtrack
            continue loop
        end if
        $new\_conflicts \leftarrow count\_ones(meeting\_times(i, next[i]) \bigwedge current[i-1])$
        if $(new\_conflicts + nc[i-1]) \leq c$ then     // move forward
            $nc[i] \leftarrow nc[i-1] + new\_conflicts$
            $current[i] \leftarrow current[i-1] \bigvee meeting\_times(i, next[i])$
            $result[i] \leftarrow next[i]$              // store section
            $next[i] \leftarrow next[i] + 1$            // prepare next section for backtracking
            $next[i+1] \leftarrow 1$                    // start at section 1 for next course
            $i \leftarrow i + 1$                        // go on to the next course
            continue loop
        end if
        $next[i] \leftarrow next[i] + 1$                // else try next section for current course
    end loop
end for
```

Fig. 1. Backtracking algorithm for generating course-section assignment schedules in increasing order of the number of conflicts

next section to try for *course*$_i$ upon backtracking, or when going forward "down the tree." The function $meeting\_times(i, j)$ returns a bitmap of 40 bits which depicts the meeting times during the week of section j of *course*$_i$. $nc[i]$ contains the number of conflicts in the schedule up to and including *course*$_i$. The function $count\_ones(bitmap)$ counts the number of "1"s in its argument.

The algorithm assumes that each section meets the same number of times (i.e. K) during the week. This simplification does not affect the core of the algorithm and makes the ensuing mathematical analysis tractable.

4 Time-Complexity of the Algorithm

In the following discussion, let us assume that a student takes N courses, each course has K sections (for uniformity), and each section meets R times per week. Let us also ignore the maximum number of results requested by the user, as this can only improve the performance of the algorithm.

The number nodes that are "generated" in the implicit tree is an accurate measure of the time complexity of the algorithm.

4.1 Worst-Case Time-Complexity Analysis

If the root has level 0, then the level of the leaf nodes in the full implicit tree is N, the branching factor of each inner node is K and the full tree has

$$\sum_{i=0}^{N} K^i \ . \tag{1}$$

nodes. In the *worst case*, all these nodes are visited. Furthermore, if C is the maximum number of conflicts that are tolerated, then the algorithm makes $C+1$ passes over the tree (although each distinct solution is printed exactly once), and the number of generated (and re-generated) nodes becomes

$$\sum_{j=0}^{C} \sum_{i=0}^{N} K^i \ . \tag{2}$$

The above formula is an upper bound on the number of nodes that are visited. However, if a node contains more conflicts than can be tolerated, it is not visited. In a sense, the tree is pruned. We explore that case below.

4.2 Average-Case Time-Complexity Analysis

For a specific number of conflicts that we can tolerate, we can compute the probability that a node in the implicit tree will be visited by the algorithm. Let $P(Y@L_b)$ denote the probability that a node at level b with *exactly* Y number of conflicts will be visited. Let c' be the maximum number of conflicts we can tolerate in a specific iteration of the algorithm. Then, the *expected* number of visited nodes for c' *or less* number of conflicts in an iteration of the algorithm is given by:

$$\sum_{j=0}^{c'} \sum_{i=0}^{N} K^i P(j@L_i) \; . \tag{3}$$

However, the algorithm, in order to list solutions in increasing number of conflicts (i.e. those solutions with no conflicts, followed by those with exactly one conflict, followed by those solutions with exactly two conflicts etc.), makes multiple passes of the virtual tree. For example, the leaf node that represents a solution with 0 conflicts will be generated (visited) three times if we can tolerate 2 conflicts. The average time complexity of the algorithm in that case is:

$$\sum_{c'=0}^{C} \sum_{j=0}^{c'} \sum_{i=0}^{N} K^i P(j@L_i) \; . \tag{4}$$

Computing $P(j@L_i)$ The root of the implicit tree is always visited, so are the nodes at level 1, since there cannot be any conflicts with no courses selected, or with one course selected only. Thus,

$$\begin{array}{l} P(0@L_0) = 1 \; . \\ P(0@L_1) = 1 \; . \\ P(j@L_0) = 0 \quad for \; j \geq \; 1 \; . \\ P(j@L_1) = 0 \quad for \; j \geq \; 1 \; . \end{array} \tag{5}$$

For nodes at level 2 we have

$$P(j@L_2) = \frac{\binom{40}{j}\binom{40-j}{R-j}\binom{40-R}{R-j}}{\binom{40}{R}^2} \; . \tag{6}$$

This formula can be justified as follows. $\binom{40}{R}^2$ is the total space of possibilities for the slots that can be taken by any two distinct courses (each course takes R slots). j slots are common to both courses, and these j slots can be taken in $\binom{40}{j}$ ways. That leaves the first course $\binom{40-j}{R-j}$ ways to choose its remaining $R-j$ slots, and the second course $\binom{40-R}{R-j}$ ways to choose *its* $R-j$ slots.

For values of i greater than 2, we need a recursive definition of $P(j@L_i)$. Let the notation $L_{i-1} \overset{k}{\Longrightarrow} L_i$ mean that k new conflicts are introduced by the move from a node at level $i-1$ to a node at level i, and $P(L_{i-1} \overset{k}{\Longrightarrow} L_i, e)$ denote the probability that k new conflicts are introduced on the move from a node at level $i-1$ to a node at level i, if at node $i-1$ we already have e conflicts. Then, for $i > 2$,

$$P(j@L_i) = \Sigma_{e=0}^{j} P(e@L_{i-1}) P(L_{i-1} \overset{j-e}{\Longrightarrow} L_i, e) \; . \tag{7}$$

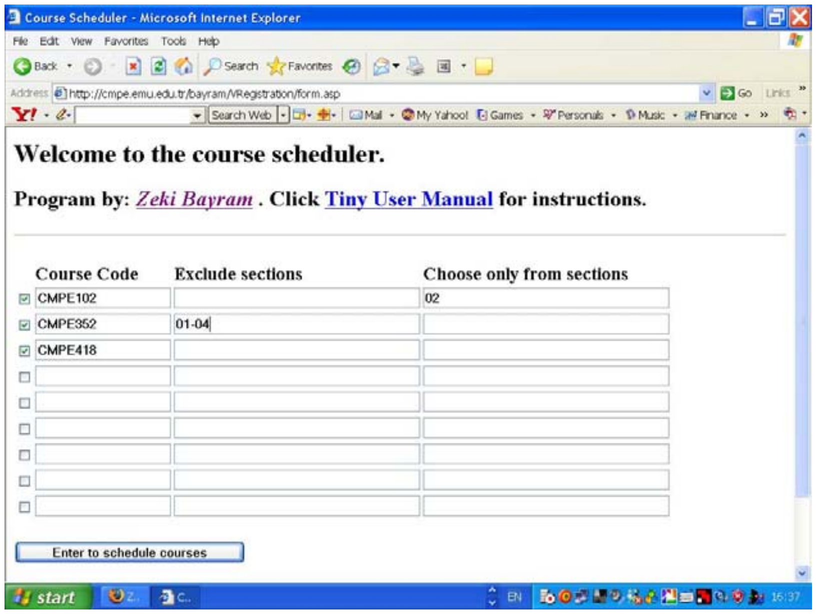
where

$$P(L_{i-1} \overset{k}{\Longrightarrow} L_i, e) = \frac{\binom{(i-1)R-e}{k}\binom{40-((i-1)R-e)}{R-k}}{\binom{40}{R}} \; . \tag{8}$$

The justification for Formula (8) is as follows. At level $i-1$ we have made assignments to $i-1$ courses, and since they have e conflicts, they use $(i-1)R-e$ slots. If we introduce k new conflicts, then surely these conflicts should be caused by the slots already taken up, hence the term $\binom{(i-1)R-e}{k}$. The remaining $R-k$ slots should come from slots not already taken up, which is $40-((i-1)R-e)$, hence the term $\binom{40-((i-1)R-e)}{R-k}$. $\binom{40}{R}$ is just all possible ways of selecting R slots out of 40 slots.

5 The Implemented Solution

The implemented solution is a WEB application with a three tier architecture. The user of the application (the student advisor) fills out an HTML form concerning the courses to be taken. He has the option of specifying which sections of a course to choose from, or alternatively which sections of a course to exclude (for example, a student might insist on taking a section which is taught by his/her favorite instructor, or s/he might insist on not taking a section taught by a disliked instructor). An example form is shown in figure 2. The form is sent to


Course Scheduler - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://cmpe.emu.edu.tr/bayram/VRegistration/form.asp
Go Links
Search Web Mail My Yahoo! Games Personals Music Finance
start
16:37

Welcome to the course scheduler.

Program by: *Zeki Bayram* . Click Tiny User Manual for instructions.

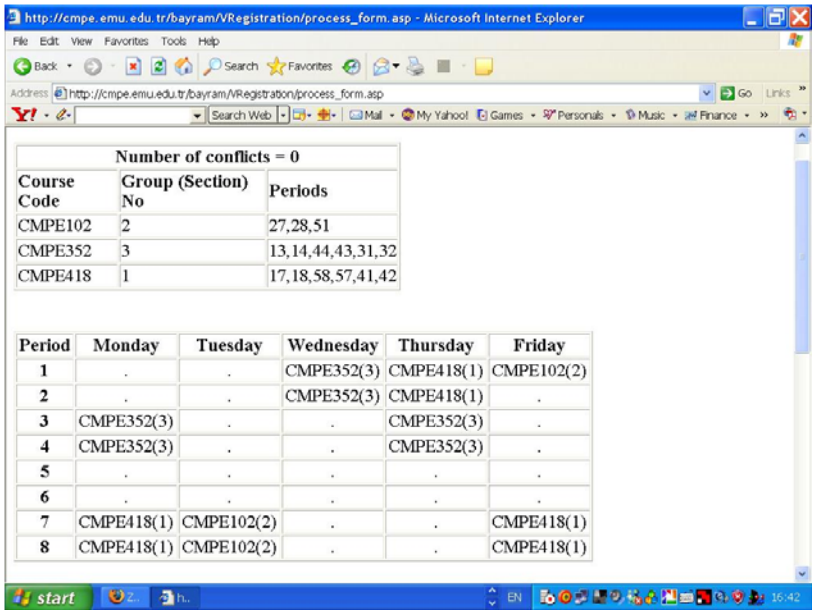
	Course Code	Exclude sections	Choose only from sections
✓	CMPE102		02
✓	CMPE352	01-04	
✓	CMPE418		
☐			
☐			
☐			
☐			
☐			
☐			

Enter to schedule courses

Fig. 2. The form for choosing courses

an Active Server Pages (ASP) application on the server, which, after querying a *courses* database, generates and sends back an HTML page with the necessary course information and the scheduling algorithm as a JavaScript program. The course information is hard-wired into the algorithm, so that each time a user makes a request with different courses, a different JavaScript code is sent to him/her. The JavaScript code then runs on the client, generating schedules (up to a “reasonable” maximum number hard-coded in the program), in order from the least number of conflicts (ideally 0) to the most number of conflicts.

Figure 3 shows part of the result returned by the scheduler. The scheduler WEB application is available at [3].



http://cmpe.emu.edu.tr/bayram/VRegistration/process_form.asp - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://cmpe.emu.edu.tr/bayram/VRegistration/process_form.asp
Go Links
Search Web Mail My Yahoo! Games Personals Music Finance
start
EN
16:42

Number of conflicts = 0		
Course Code	**Group (Section) No**	**Periods**
CMPE102	2	27,28,51
CMPE352	3	13,14,44,43,31,32
CMPE418	1	17,18,58,57,41,42

Period	Monday	Tuesday	Wednesday	Thursday	Friday
1	.	.	CMPE352(3)	CMPE418(1)	CMPE102(2)
2	.	.	CMPE352(3)	CMPE418(1)	.
3	CMPE352(3)	.	.	CMPE352(3)	.
4	CMPE352(3)	.	.	CMPE352(3)	.
5	.	.	.	.	.
6	.	.	.	.	.
7	CMPE418(1)	CMPE102(2)	.	.	CMPE418(1)
8	CMPE418(1)	CMPE102(2)	.	.	CMPE418(1)

Fig. 3. Part of the result returned by the scheduler

6 Related Work

There have been a few major venues of attack against course scheduling (sometimes called “timetabling”) problems. These include constraint logic programming, genetic algorithms, simulated annealing, tabu search or some combination of these.

In [4] the authors use the Eclipse constraint logic programming system to construct optimum timetables for university courses. The application of combined deductive and object-oriented technologies to a “complex scheduling problem” which emphasizes local propagation of constraints performed with deductive rules is presented in [5].

Use of genetic algorithms for the solution of the timetabling problem is investigated in [6].

In [7] the authors investigate a variety of approaches based on simulated annealing for the course scheduling problem, including mean-field annealing, simulated annealing with three different cooling schedules, and the use of a rule-based preprocessor to provide a good initial solution for annealing.

An expert system solution to the timetabling problem is given in [8]. The expert system is written using a CLP(FD) system that extends the CHIP constraint logic system with heuristics.

In [9] the authors formulate course scheduling as a constraint satisfaction problem and apply various optimization techniques to solve it. A similar approach is taken in [10] where the potential of constraint satisfaction techniques to handle deterministic scheduling problems is investigated.

In [11], the authors present graph colouring and room allocation algorithms and show how the two can be combined together to provide the basis of a "widely applicable" timetabling system.

Tabu search algorithms for solving scheduling problems in school environments are investigated in [1] and [12].

Our literature search has not revealed any direct investigation of the course-section assignment problem, the way we have presented it here. Our approach used in the mathematical analysis of the average-case time complexity of the backtracking tree-search algorithm also appears to be novel.

7 Conclusion and Future Research Directions

We formally defined the course-section assignment problem, presented an algorithm that solves instances of it by performing a depth-first search of the implicit search space defined by the problem, and analyzed the time complexity of the algorithm using a probabilistic approach.

The search algorithm for a solution to the course-section assignment problem can be generalized to solve other kinds of scheduling problems. The mathematical approach we used here can be applied to analyze those algorithms also. Future work might include the extension of the presented framework to handle such more general scheduling problems.

References

1. Schaerf, A.: Tabu search techniques for large high-school timetabling problems. In: Proceedings of the Fourteenth National Conference on Artificial Intelligence, Portland, Oregon, USA (1996) 363–368
2. Silberchatz, Korth, S.: Database System Concepts. 4 edn. McGraw Hill (2002)
3. Bayram, Z.: Course scheduling web application, available at web site. http://cmpe.emu.edu.tr/bayram/VRegistration/form.asp (2003)
4. Frangouli, H., Harmandas, V., Stamatopoulos, P.: UTSE: Construction of optimum timetables for university courses - A CLP based approach. In: Proceedings of the

3rd International Conference on the Practical Applications of Prolog PAP'95, Paris (1995) 225–243

5. Y., C., Guillo, P., Levenez, E.: A deductive and object-oriented approach to a complex scheduling problem. In: Proceedings of Deductive and Object-Oriented Databases: Third International Conference, Phoenix, Arizona, USA (1993) 67–80
6. Colorni, A., Dorigo, M., Maniezzo, V.: Genetic algorithms - A new approach to the timetable problem. Lecture Notes in Computer Science - NATO ASI Series , Combinatorial Optimization, (Akgul et al eds) **F 82** (1990) 235–239
7. Elmohamed, M.A.S., Coddington, P., Fox, G.: A comparison of annealing techniques for academic course scheduling. Lecture Notes in Computer Science **1408** (1988) 92–114
8. Azevedo, F., Barahona, P.: Timetabling in constraint logic programming. In: Proceedings of the 2nd World Congress on Expert Systems, Lisbon, Portugal (January 1994)
9. Blanco, J., Khatib, L.: Course scheduling as a constraint satisfaction problem. In: Proceedings of the Fourth International Conference and Exhibition on The Practical Application of Constraint Technology, London, England (1998)
10. Dignum, F.W.N., Janssen, L.: Solving a time tabling problem by constraint satisfaction. Technical report, Eindhoven University of Technology (1995)
11. Burke, E.K., Elliman, D.G., Weare, R.F.: A university timetabling system based on graph colouring and constraint manipulation. Journal of Research on Computing in Education **27**(1) (1994) 1–18
12. Gaspero, L.D., Schaerf, A.: Tabu search techniques for examination timetabling. Lecture Notes in Computer Science **2079** (2001) 104–108