Student - Class Assignment Optimization Using Simulated Annealing

Krzysztof Ciebiera[1], Marcin Mucha[1]

[1]Faculty of Mathematics, Informatics, and Mechanics, University of Warsaw, Banacha 2, 02-097 Warszawa, ciebie@mimuw.edu.pl, mucha@mimuw.edu.pl

Keywords
class registration; local search; simulated annealing; University Study-Oriented System (USOS);

1. ABSTRACT

The University Study-Oriented System (USOS) [1] is an integrated student management information system for handling student affairs at Polish universities. Its development and deployment is coordinated and supported financially by the consortium of Polish higher education institutions.

Course registration is one of the most important functions of the system. Timetabling problem is broadly studied subject leading to many theoretical and practical results [2, 5]. Part of the registration problem is an assignment of students to classes after schedule has been published and students have been registered to courses. We have experimented with different strategies like “first in first served” or “greedy assignment method”' however they seem to be either unfair or results are not satisfactory. Too many students had conflicts (two or more classes at the same time) or were unable to register to any class.

We have created computer program based on simulated annealing local search method, that allowed us to reduce number of students having conflicts by 20%. First students define their preferences using self-service web based system, then the registration system is taken to read-only mode and assignments are generated using our engine (SA-RDG), and at the end of a process, results are imported into web system where they are available for students.

In this paper we describe algorithm details, technical solution method and we present empirical results of the system performance at Mathematics, Computer Science and Mechanics Department of University of Warsaw in years 2010-2013. SA-RDG is default registration method for USOS installed in 40 higher education institutions in Poland.

Research was supported by the ERC StG project PAAl no. 259515.

2. INTRODUCTION

Course and class registrations are one of the most important functions of any student management information systems. Popular methods supported by USOS are:

- Manual registration by dean's office. Students are not given any choices. Both course and cclass assignments are made by dean's office.
- Registration outside of the USOS. Students register to courses and classes in real-world and then registration results are imported into the system.
- Token based registration [3]. Every student receives a number of tokens of different kinds and registers to classes using Internet based system paying with these tokens. After reaching class limits registration to this class is suspended.
- Preferences based registration. Students define their preferences and then computer system tries to meet them preserving other requirements.

In this paper we describe an application of Simulated Annealing algorithm to preferences based registration. Our solution does not require students to be available during registration at the same time or at the same place. They define their preferences via web based system during phases defined by system administrators (usually a few days).

2.1 Course schedule constraints

In this section we describe course schedule constraints, defined by system administrator.

Limits. Course schedule consists of many courses, with every course there are associated some classes. Each class has its own schedule and a limit on number of students. After registration to a course student must be assigned to one and only one of its classes. Different classes of a course may have different schedules.

Conflicts. Classes are in conflict one with another if their schedules overlap. We use different type of schedules (e.g. every second Monday 8:30am-10:00am, starting on 10th February 2014, seven occurrences) but the only thing that matters to our algorithm is whether two schedules overlap or not.

Exclusions. Some pairs of classes are excluded. If a pair of classes (X, Y) is excluded it means that student can not be assigned to both X and Y at the same time. Administrators of the system use exclusion mechanism to join classes of different courses into sets that must be assigned together. Instead of defining which classes must be assigned together they define classes that student can not be assigned to in the same time. *For example consider two courses A (classes A1, A2) and B (classes B1, B2, B3, B4). If administrator wants to join A1 with B1, B2 and A2 with B3, B4, he says following pairs: (A1, B3), (A1, B4), (A2, B1), (A2, B2) are excluded.*

2.2 Two-phase registration model

SA-RDG supports two-phase registration model (Figure 1).

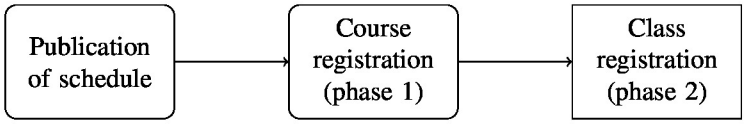

Figure 1. Figure shows two-phase registration model. SA-RDG performs automatically class assignments during phase 2.

During the first phase students either are registered by dean's office or register themselves to courses. During the second phase students are registered to classes by SA-RDG based on their preferences and other constraints.

Phase 2 consists of following steps:

1. Students define their preferences (it usually takes a few days) using registration system.
2. System is taken to read-only mode.
3. SA-RDG engine performs student-class assignments (usually 5-10 minutes).
4. Assignment results are shown to students.

2.3 Students preferences

In this section we describe preferences that can be defined by students.

Course importance. Using web interface each student define their preferences. Since the most important parameter for the system is number of conflicts, student may mark some courses as not important for him. For the system it means that student doesn't want to attend classes of this course

so it will not generate any conflicts for him. All courses not marked as unimportant are considered important.

Preference. We say that two classes are in conflict whenever their schedules overlap, so if student was assigned to 3 classes (A1, B1, C1) all of them overlapping we say the student has 3 conflicts (A1-B1, A1-C1, B1-C1).
Students may prioritize classes of all his important courses. They define sequence of subsets of all classes (Figure 2).

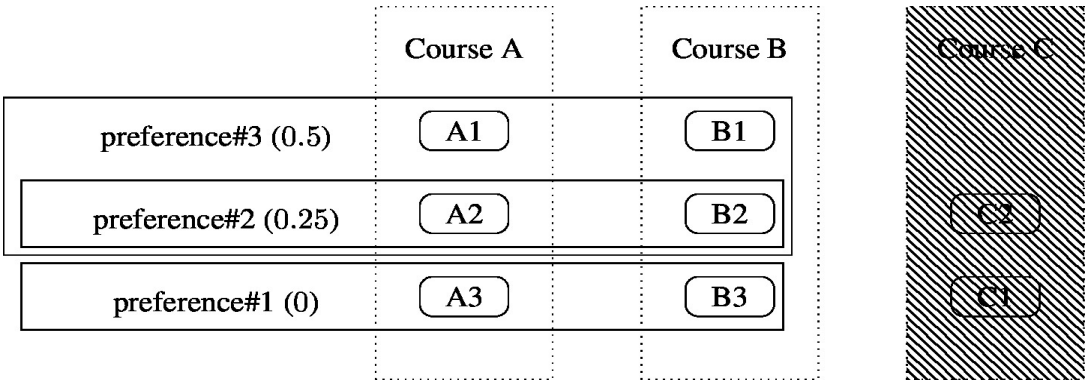

Figure 2. Example of students preferences. Student ignores course C. He wants classes (A3, B3) the most (priority 0), classes (A2, B2) at the second place and any pair of classes (A1, A2, B1 ,B2) at the last place.

The first subset gets priority 0 (lower values are better), the last subset (if there are more than one) gets 0.5, and other subset are prioritized linearly between 0 and 0.5. Subset may contain more than one class of a course, system will assign student to only one of those classes.

3. GREEDY SOLUTION

Solution used by USOS before SA-RDG engine was a greedy one [8]. It was implemented in form of a monolithic Java based application, that was performing greedy registration. It ordered students either randomly or according to some criteria (disabilities, GPAs, etc.) and then assigned them to classes one after another using Algorithm 1.

Data: students priorities, plan
Result: assignment of students to classes
order students according to some criteria or permute them randomly;
for all students **do**
  for all ordered preferences of current student **do**
   if matching preference does not exceed limits **then**
    assign student using preference;
    take next student;
   end
  end
  register student to least crowded classes without breaking exclusions;
end

Algorithm 1. Greedy class registration

It may seem to be quite reasonable solution, however, it had some significant drawbacks: although lucky students (at the beginning of the list) get their priorities fulfilled without any conflicts, many unlucky students have a lot of conflicts, some configurations can lead to broken class limits.

Unlike in SA-RDG, students preference sets contained only one class for each course. To get more expressive power some students created their own programs that defined automatically thousands of class subsets.

Because of the algorithm properties, in its later phases two type of constraints could become conflicting: exclusions of classes and class limits. Algorithm preferred to break class limits, although SA-RDG could find solution that met all schedule constraints.

4. TECHNICAL SOLUTION

SA-RDG environment consists of a few parts: web interface for students for preferences input, programs for converting database preferences into text files, and engine finding good solution. Figure 3 shows architecture of the system.

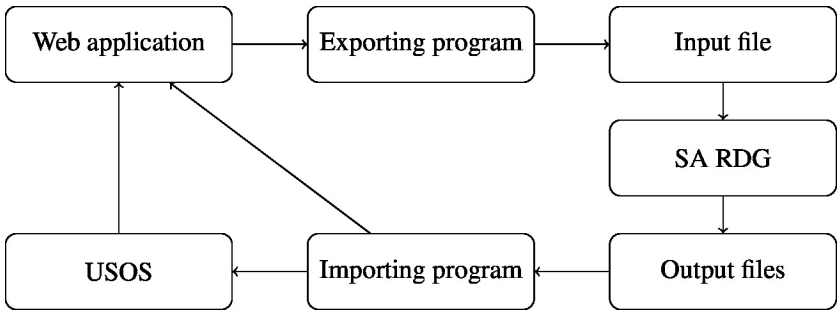

Figure 3. Model of SA-RDG architecture.

Although architecture seems redundant (we could integrate exporting and importing program with engine), we find this design useful and robust. It is quite easy to debug problems (both bugs and design related) having separated engine and database logic. During system testing we almost instantly knew which part of the system contained a bug.

Input file is in a human-readable text format and contains list of courses and classes with their attributes (limits, schedule, dependencies), list of students and their preferences. It may contain students personal information in form o comments. Output file contains only list of students and their classes. Program also writes a log file and a report file, which are useful for evaluating algorithm performance.

SA-RDG is run off-line after the on-line registration module is switched to read-only mode. After results are examined by an administrator the results are imported into the USOS and are shown to students. SA-RDG is written in C++ using Standard Library.

5. ALGORITHM DESCRIPTION

We use Simulated Annealing local search method [4] as shown on Algorithm 2. Algorithm starts with a random solution (we randomly assign students to classes) and tries to improve it, reducing its penalty, with a sequence of small moves. In our case a single move is to move a random student from one class to another. We start with a random solution, that often breaks some constraints, however, experiments have shown that the final result always met those constraints.

Data: students priorities, plan
Result: assignment of students to classes

```
temperature ← starting temperature; // default: 1000
finish_temperature ← constant;     // default 0.000001
cooling_coefficient ← constant;    // default: 0.999999
solution ← random assignment of students to classes;
solution_penalty ← penalty(solution);
while temperature > finish_temperature do
  s ← random student;
  c ← random class of a student s;
  Δ ← penalty(solution with student s in class c) - current penalty;
  if Δ <= 0 or Δ should be accepted at current temperature then
   move student s to class c;
  end
  temperature ← temperature * cooling_coefficient;
 end
```

Algorithm 2. Simulated annealing class registration

In order to compare different solutions we define function *penalty*. Better solutions should have smaller penalties. Details of penalty function are described in Section 5.1. In our case when penalty reaches 0 no students have conflicts, all preferences are met and no constraints are broken. For performance reasons, instead of evaluating penalties of the whole solutions, we only evaluate Δs of moves.

We could use Hill Climbing [6] local search technique that always improves solution. However, in our case we often solve problem instances that have a lot of local minima (e.g. number of students registered to a course equals sum of course's class limits, so moving a single student from one class to another is impossible, since it increases penalty). That is why we use Simulated Annealing that sometimes performs moves that increases penalty of solution, but it prevents program from becoming stuck at a local minimum.

If Δ is lower than 0, we accept moves since it improves quality of the solution. If Δ is higher than zero we accept moves with probability $\exp(-\Delta/T)$. Probabilities of moves acceptance decrease with decreasing temperature.

5.1 Penalty

Penalty function is calculated as weighted sum of following:

- total number of broken class exclusions (weight 4),
- total number of broken class limits penalties (weight 2),
- total number of all conflicts (weight 1),
- total sum of met students priorities (weight 0.2).

Since we allow intermediate solutions to break constraints, the highest weights are assigned to features describing those constraints. Our program is flexible enough, so it can handle cases when some constraints can't be met (e.g. there are too many students than places in a class). We compute penalty for a broken class limit as 0 if the number of students assigned to class is lower or equal to limit or square of difference between number of assigned students and class limit, so in cases when limits cannot be met we will have similar overload for each class.
Penalty can be divided in two parts, class induced (exclusions and limits) and students induced (conflicts and preferences). We allow to assign a scaling parameter independently for each student, so some students conflicts and preferences are more important for penalty function that others (e.g. students with disabilities). We multiply number of students conflicts and their priorities by this scaling factor.

5.2 Program performance

It is important to be able to evaluate as many moves as possible, so we use some performance optimizations.

Δ *computation*. Instead of computing two penalties of current and evaluated solution we only calculate Δ of a single move. In order to do it program needs to read from memory states of two classes of one course, and assignments-and-preferences of a single student.

Class conflicts. Instead of calculating if classes schedules overlap during Δ evaluation for each student we keep a small two dimensional bit-array of conflicts that are calculated once during program initialization. Program supports irregular schedules (e.g. 2014-01-10 10:15-11:45 and every second Monday 13:45-15:10) without any significant performance degradation.

Intermediate results cache. Whenever possible (number of students assigned to class, priority of met preference, etc.) we store the computation result in an object cache.

C++ optimizations. We avoid using virtual functions to make it possible for code optimizer to generate the fastest code possible [7]. We use as sparse data structures as possible to make it possible for processor to fit data into cache.

6. RESULTS

We have compared SA solution with the greedy solution which was used before. We used registration results at Mathematics, Computer Science and Mechanics Department of University of Warsaw during years 2010-2013. Three of registrations were small (about 150 students, 10, courses 20 classes) and the other 7 were normal (900-1100 students, 210-270 courses and 380-520 classes).

We compared three parameters:

- total number of conflicts (Figure 4),
- number of students having conflicts (Figure 5),
- number of students that have set some priorities and the system could not met any of them (Figure 6).

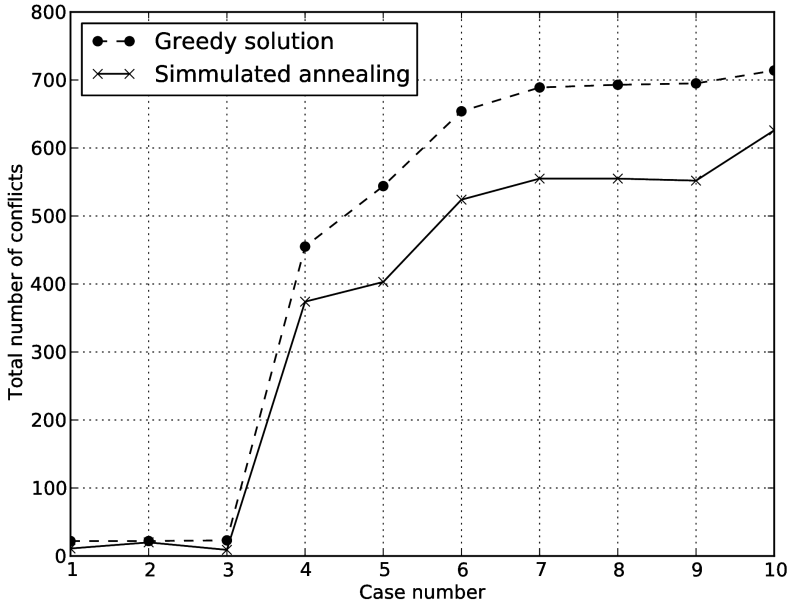

Figure 4. Comparison of total number of conflicts using assignments generated by two algorithms Simulated Annealing and Greedy Solution

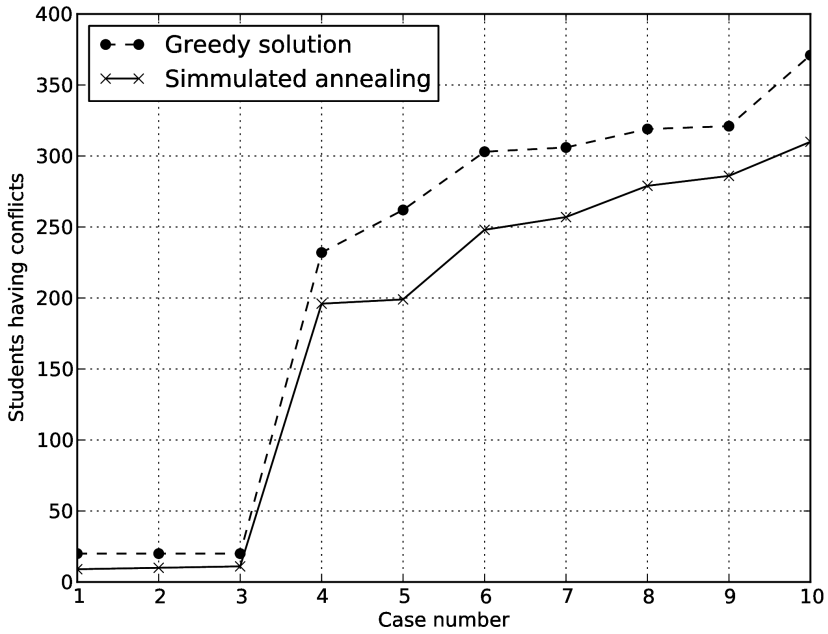

Figure 5. Comparison of total number of students having conflicts using assignments generated by two algorithms Simulated Annealing and Greedy Solution

Total number of conflicts and total number of students not having conflicts dropped by 20%. It may seen that total number of conflicts is huge (over 600 conflicts of 1100 students) however we manually checked schedules of students with the highest number of conflicts and those conflicts were inevitable (student registered to courses that have only conflicting classes).

In two cases number of students that have no priorities met was a little bigger (5%) than in a case of the greedy solution.

7. REFERENCES

[1] USOS website. University Study Oriented System. Retrieved in February 2014 from: http://usos.edu.pl.

[2] Post, G., Di Gaspero, L., Kingston, J., McCollum, B., Schaerf, A. (2013). The Third International Timetabling Competition *Annals of Operations Research*

[3] Ciebiera, K., Mincer-Daszkiewicz, J., Waleń, T. (2004). New Course Registration Module for the University Study-Oriented System. *EUNIS 2004, 247-252*

[4] Kirkpatrick, S., Gelatt Jr, C. D., Vecchi, M. P. (1983). Optimization by Simulated Annealing. *Science 220,* 671–680.

[5] Bellio, R, Di Gaspero, L, Schaerf A. (2012). Design and statistical analysis of a hybrid local search algorithm for course timetabling. *Journal of Scheduling*, 49-61.

[6] Russell, S, Norvig, P (2003), Artificial Intelligence: A Modern Approach (2nd ed.), *Prentice Hall*, 111–114

[7] Driesen, K., & Hölzle, U. (1996). The direct cost of virtual function calls in C++. In *ACM Sigplan Notices* 306-323.
[8] Cormen, Leiserson, Rivest (1990) Introduction to Algorithms, Chapter 17 "Greedy Algorithms" p. 329.

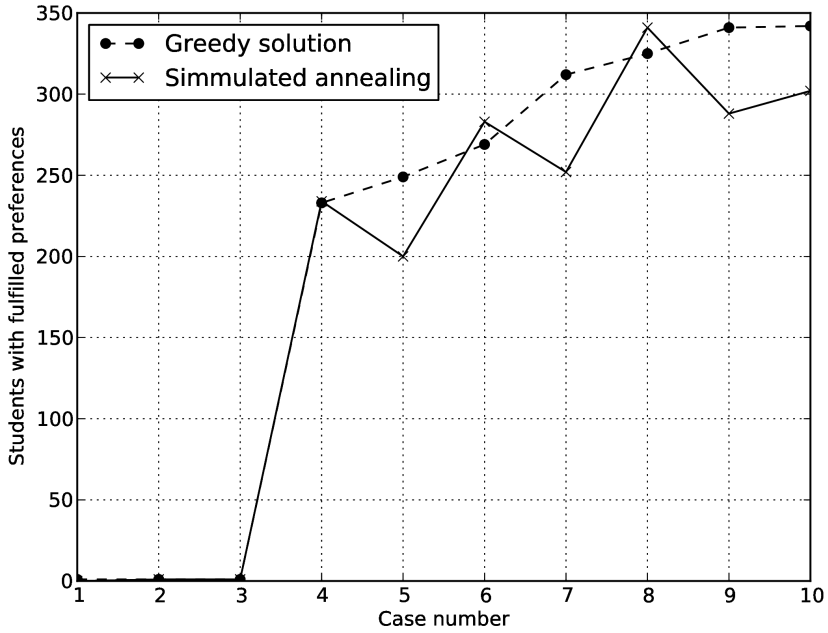

Figure 6. Comparison of total number of students having any of preferences met using assignments generated by two algorithms Simulated Annealing and Greedy Solution

8. AUTHORS' BIOGRAPHIES

Krzysztof Ciebiera is a programmer at the Institute of Informatics, University of Warsaw, member of the Algorithms Group.

He worked on registration module of USOS system.

Marcin Mucha is an assistant professor at the Institute of Informatics, University of Warsaw, member of the Algorithms Group.

He works on graph algorithms, approximation algorithms and on-line algorithms.