Student Course Sequence: A CSP Solution

CS 7750: Artificial Intelligence I
Graduate Student Project
December 14, 2016

Shashank Avusali, Sue Brownawell, Quenton LaRoe

Abstract

Westminster College, small liberal arts college in Fulton, Missouri, desires to more accurately plan their course offerings to correctly anticipate staffing needs. In order to project course demand from current students, we develop an algorithm to suggest individual student plans to fulfill all graduation requirements within four, two-semester years including selected major(s) and minor(s) is desired. Constraints include maximum and minimum number of credits per semester, prerequisite and co-requisite courses, and miscellaneous other requirements. We assume students have pre-selected the courses they desire to take to meet elective requirements. We generate an initial state in a pre-processing step by assigning courses from the required and elective course lists to the eight semester lists of courses. Then we run a simulated annealing algorithm to maximize the number of constraints met. A simplified version of the problem for one major and the general education requirements and using placeholder courses for elective requirements was used to test the time performance and goodness of the solution. The algorithm was run with a starting temperature of 500 degrees and cooling rates of 1% and 5%. Both rates of cooling gave optimal solutions, meeting all constraints. The 1% cooling rate executed in 30ms and the 5% cooling rate executed in 28ms. The algorithm appears well suited to solve the complete problem and the next steps are to adapt the solution to include multiple majors and minors and to replace the placeholder courses with specific courses for the elective requirements.

1 INTRODUCTION

Westminster College, small liberal arts college in Fulton, Missouri, desires to more accurately plan their course offerings to correctly anticipate staffing needs. This would be greatly facilitated by more accurately projecting demand from current students. Additionally, students commonly major and minor in multiple disciplines and often run into scheduling difficulties, sometimes delaying graduation. To meet these business needs, an algorithm to suggest individual student plans to fulfill all graduation requirements within four, two-semester years including selected major(s) and minor(s) is desired.

Variations on the academic scheduling problem are common in the literature. This may be scheduling courses, rooms, professors and students in a given semester or some combination thereof, or scheduling courses for one student over the duration of their enrollment as in our problem. For this paper, we look at creating a schedule of courses for a single student, rather than setting a semester of courses assigned to different professors, rooms and times. While our problem has some differences, it has common

elements and shares how complex this type of problem can become as we add in the possibilities of having multiple majors for a single student.

Looking at paper [1], we considered how their problem is setup as well as some restrictions that they faced while when working on this algorithm. Their objective was to assign a professor to a course each semester for different classrooms and time slots for each professor and course section pairing, including lab sections [1]. They approached the problem through using a three-step process. The first step was assigning courses, time periods and professors based on the preferences that each professor has on what course they want to teach, as well as the time periods that the professor prefers to teach. After this they divided the problem into two smaller problems. The first problem to be solved is assigning each professor to a time, such that no two professors are assigned to the same time slot. The second problem deals with assigning the course sections based on the solution of the first problem. The final step is to use simulated annealing to then improve upon the solution that is met from the second phase which finds the fewest conflicts, as well as professors being assigned to their preferred courses and time slots [1].

The restrictions within [1] are discussed in further depth within [2], as they go over the mathematical equivalent of each restriction and how each set of professors, courses and time slots are satisfied. Aycan and Ayav [3] also chose simulated annealing for a similar problem.

2 Solution Design

2.1 Language of implementation

We have chosen Java for implementation for the ability to use objects and because all variables are allocated on the heap. Additionally, the majority of team members have more experience with Java than other alternatives.

2.2 Problem Definition: Variables and Domains

For our particular problem, we have a list of course requirements that students must fulfill to graduate. These course requirements fulfill either general education requirements or the requirements for their chosen major(s) or minor(s). Some of these requirements can be fulfilled by one specific course, and sometimes there is a list of courses that a student may choose from. In addition, there are requirements for a minimum and maximum number of credits per semester, and a minimum number of total credits needed for graduation. We require that the course sequence be completed in four, two-semester years. There are additional constraints on which semesters a given course is offered, and courses may have pre-requisite and co-requisite lists.

At first glance, there seem to be two types of variables in our problem. The first is the eight *semesters*, which have a domain of integers for the number of credits assigned per *semester*. Conceptually, it is natural to think of the second type of variable as a *course*, which has a domain of the eight semesters in our problem. However, we have chosen to reverse the relationship between courses and semesters. We have *semester course slots* as variables for which the domain is the list of courses. At this stage, we can think of this domain as the list of courses in the catalog (but we will adjust this below.) But these two types of variables are not independent; the number of credits in a *semester* is totally dependent on the courses that are assigned to that semester. So the only variables of consequence in problem formulation

are the *semester course slots.* These *semesters course slots* are distributed among the semesters during a pre-processing step.

Because students may choose different courses to fulfill some of the requirements, we assume that a student chooses the courses to fulfill requirements before processing begins. By doing this, the total number of credits are determined beforehand. Even if this was not the case, *total credits* would not be a variable because it too is dependent on the courses selected.

The decision to require that students indicate their course selections before processing begins also avoids this scheduling problem becoming a dynamic constraint satisfaction problem where the particular courses we are considering change while the problem is being solved [4]. If we allowed the courses chosen to fulfill a requirement to change, this would also imply that requirements themselves are variables. This is even more complicated in that sometimes a requirement is fulfilled by a certain number of courses and sometimes by a certain number of credits. Having additional variables for requirements and dynamically changing the course variables and even potentially the number of course variables would complicate the problem significantly. Additionally, this is not practical as students have varying reasons for choosing a given course over another and an algorithmic choice may not predict true student demand.

We have chosen data structures that reflect the dependency of semester credits on the courses assigned to *semester course slots*. A **semester** object holds a list of **course** objects, each place in the list being the *semester course slot*. **Course** objects have a property of number of credits, which are summed from the semester list of courses to calculate the *semester credits*.

The below code snippet shows our semester object properties.

```
public class Semester implements Cloneable {
      private ArrayList<Course> courses = new ArrayList<Course>();
      private int totalCredits = 0;
```

2.2.1 Problem simplifications and adaptations

We expected to receive files from the school containing a list of courses offered and the requirements for the general education program and all majors and minors, but were unable to do so in the time available. Additionally, we were unable to obtain the course lists for certain requirements such as for Writing Intensive courses. Because of these problems, we made three decisions. First, since we needed to hand-enter requirements based on the information in the Academic Catalog, we would hard code our courses and requirements and limit our initial solution to the general education requirements and one major (computer science). Second, for requirements with course lists and for the electives necessary to raise the overall credit total to the required minimum, we would use placeholder courses instead of real courses. For instance, for the two writing intensive courses that are required as part of the general education requirements, we would create two placeholder courses named WI_1 and WI_2 that do not have pre-requisites or co-requisites. These two placeholder courses would be treated like actual courses in our algorithm. Third, we would ignore when a course is typically offered in our constraint checking.

There are a few other constraints that are ignored due to our decision to use placeholders. For instance, there is a general education requirement for an upper level course outside of the major division. Since we are using a placeholder for this requirement, we assume the course is outside of the major division.

Also, since we are using placeholders, there are only five courses that have prerequisite and corequisite lists. This greatly reduces the constraint checking.

2.3 Problem Definition: Constraints

By assigning courses that fulfill requirements during problem set-up, we assure that those requirements are met and that the minimum number of total credits is met. Additional constraints remain, however. There are minimum and maximum numbers of credits that can be taken in a semester, there are certain general education and major courses required to be taken in certain semesters or years, and pre-requisite and co-requisites must be met.

In addition to an identifier for the course and a property for the number of credits, our course object also contains lists of pre-requisites and co-requisites as can be seen in the below code snippet. This is for ease of implementation over the alternative of maintaining separate data structures to look up the needed information.

```
public class Course {
	private String courseId;
	private int credits;
	private ArrayList<String> prereqs = new ArrayList<String>();
	private ArrayList<String> coreqs = new ArrayList<String>();
```

We have implemented constraint checking in a separate class for ease of management. This class enables us to easily calculate and track the number of constraints fulfilled and the total number of constraints. A method RunAll() runs a set of methods to check individual constraints or groups of similar constraints as can been seen in the following code snippet.

```
public int runAll(ArrayList<Semester> semester) {
	this.constraintsTotal = 0;
	this.constraintsFullfilled = 0;
	this.semesterHours(semester) ;
	this.firstSemester(semester.get(0));
	this.freshman(semester);
	this.upperClass(semester);
	this.upperClassFall(semester);
	this.pre_co_req(semester);

	return this.constraintsFullfilled;
}
```

Each of the methods counts the constraints that are being checked and adds them to the total constraints, and counts the constraints that are fulfilled and adds them to the fulfilled constraints. The constraints checked by each method are as follows:

1. semesterHours(ArrayList<Semester>semester)
   This method checks if the credits for each semester are at least 12 and no more than 19. Twelve credits are required to be a full-time student, and credits in excess of 19 incur additional charges.

2. firstSemester(Semester)
   All freshman take WSM_101 Westminster Seminar and LST_101 The Leader Within during their first semester, which are part of their general education requirements.

3. freshman(ArrayList<Semester>semester)
   In addition to the two courses checked by the firstSemester() method, freshman take a course to fulfill their General Education Tier 1 math requirement and Tier 1 foreign language requirement in either their first or second semester.

4. upperClsss(ArrayList<Semester>semester)
   There are two general education requirements that would need to be taken in the junior or senior year. This would typically be satisfied by checking pre-requisites on the selected courses, but since we are using placeholders we implemented this constraint instead. These requirements are for a class that is an upper level outside the division of the major and a multi-disciplinary upper level course.

5. upperClassFall(ArrayList<Semester>semester)
   The Computer Science department has designed the program for students to typically take a suite of classes together in either their junior or senior fall semester. Although the co-requisite lists would take care of most of these courses being scheduled together, we have this method to assure that they are taken during the correct semester.

6. pre_co_req(ArrayList<Semester>semester)
   This method checks each individual course for pre-requisite and co-requisite requirements, and then determines if those courses are scheduled for an appropriate semester. Each pre- or co-requisite class is a separate requirement.

2.4 Problem State Space

In our simplified problem, the total number of courses that are assigned among the eight semesters is 41. If we had adopted the view that courses are variables and they are assigned the value of any one of the eight semesters with no restrictions, the possible number of states p where s is the number of semesters and c is the number of courses would be:

$$p = s^c = 8^{41} = 1.06 \times 10^{37}$$

In our view of the problem, however, we have reversed that relationship so that semester course slots are assigned the value of one of the 41 courses. This would give us the number of possible states p where c is the number of courses as:

$$p = c! = 41! = 3.35 \times 10^{49}$$

This significantly increases the state space over the previous arrangement. But we distribute the semester course slots among the semesters during pre-processing so the number of courses per semester is pre-determined. Our algorithm distributes them to be 7 semesters of 5 courses and 1 semester with 6 courses. There are 41 course slots that can have any arrangement of the 41 courses. It is irrelevant, however, what order the courses appear in a given semester course list. If, in our algorithm, we do not allow a course to be assigned to more than one semester list, the possible number of states p where s is the number of semesters and c is the number of courses is given by:

$$p = \prod_{n=1}^{s-1} \frac{\big(c - \big(5(n-1)\big)\big)!}{5!\,(\big(c - (5n)\big)!)} = 1.30{\times}10^{32}$$

That is, for each semester we take number of combinations of c course objects taken 5 courses at a time, recognizing that after we assign the courses for one semester there are 5 fewer course objects to choose from the next semester. Then we multiply these combinations together. The last semester has six course objects for six semester course slots, so there is only one possible combination for that semester.

The combination of pre-processing with our chosen variables reduces the size of the problem state space somewhat, but it is still immensely large.

2.5 ALGORITHM

2.5.1 Problem break-down

When looking at the problem, we had to look at our constraints and try to create an algorithm that would be able to reach a satisfactory solution. With this thought, we looked at a couple of papers that solved a similar problem. What we found is that other researchers used simulated annealing after the use of a greedy sorting algorithm to come to an optimal solution [1] [2] [3]. When we dug a little further into the papers on why this was the most common path taken, we saw that it was due to time constraints, as well as how complex the problem can be. Backtracking search looks for a solution that satisfies all constraints and returns null if nothing can be found, which can be complicated with a large number of constraints. The time for backtracking search is also exponential b^d. For our variable set up, this would be a maximum of 41^{41} (if we did not reduce the domain of courses for each semester course slot as assignments were made). This is not ideal as we need a schedule to print out within a timely manner. We are also willing to accept a solution that is close to one that fulfills all constraints and allow a student and their advisor to make adjustments by hand. Simulated annealing meets our criteria of being able to find a good solution as it moves from state to state sequentially taking the best states that it finds, and occasionally taking a bad step as well. This is in comparison to backtracking search, which goes from state to state recursively, allowing them to go back if there are no good options left to move forward from. With these algorithms considered and simulated annealing chosen, we started to look at how to pre-populate the initial state, and setting a goal to minimize the number of constraints not satisfied.

2.5.2 Pre-Processing

Our initial setup for our problem is to randomly select courses and place them within each semester. We also decided to fulfill a few constraints during pre-processing. As we assign courses to semesters, we satisfy the number of credit hours being between twelve and nineteen credit hours per semester. We also assure that courses that must be taken in particular semesters are placed within the semesters that they need to be in. For instance, we had to ensure that each student had their general education math requirement done within either semester of their first year. We place the semester-constrained courses first and then randomly assign the remaining courses to semesters and pass these results to our simulated annealing algorithm.

2.5.3 Simulated Annealing

The goal that we have set up for this portion of the algorithm is to not to find the correct answer, but to find a good solution as defined by maximizing the number of constraints met out of the total. We start

our simulated annealing process by generating a neighbor state by exchanging a random course from two separate random semesters. After this, we calculate the energy (the number of fulfilled constraints) of both states and compare their energy states. If the energy within our neighbor is greater than our current state, we take the neighbor state. (Simulated annealing would normally look for a lower energy state, but our implementation is looking for more fulfilled constraints.) If the second state is the same, or worse, then we need to randomly pick one of these two. To do this, first we need to define a successor function that would be used for calculating the distribution of our two states. We use a modified version of the Boltzmann Distribution as our successor function, where we remove the Boltzmann constant, as seen within this equation:

$$p = e^{\Delta E/T},$$

Where T is our temperature, ΔE is the difference within the energy of the two states that we have passed, and p is the probability from our modified Boltzmann Distribution. We then take the second state if and only if the probability we found with the modified Boltzmann is larger than a random value that is generated within the range of zero to one. We repeat this process within a loop until we have reached a time limit, a goal state is found or if our temperature is below a set threshold.

A pseudocode representation of our simulated annealing algorithm is below.

```
Generate initial Schedule
Initialize time, maxTime, Temperature //0, 1000, 500
Energy1 = fulfilled constraints on Schedule
If Energy1 != total constraints
    While time < maxTime
        TempSchedule = Schedule
        Randomly switch courses in TempSchedule
        Energy2 = fulfilled constraints on TempSchedule
        If Energy2 == total constraints
            break
        If Energy2 > Energy1 //new state is better
            Schedule = TempSchedule
            Energy1 = Energy2
        Else
            Probability = e^((Energy2-Energy1)/(Temperature))
            If Probability > random number between 0 and 1
                Schedule = TempSchedule
                Energy1 = Energy2
    Reduce Temperature // Temperature*0.99 for example
```

3 Implementation and Results

The project is implemented in Java using Eclipse Mars.2 (Release 4.5.2), with Java execution environment Java SE 8 (1.8.0_77) on MacOS Sierra version 10.12.1, 2.2 GHz Intel Core i7, with 16 GB primary memory. We ran our simulated annealing implementation with a start temperature as 500 and two different cooling rates, 1% and 5%. At lower cooling rates, the system accepted moves that result in a high error for a longer duration of time before starting to accept only the moves that reduce error. In our case the

error is calculated as the percentage of constraints violated. At the 5% cooling rate, the system accepts bad moves only for a short duration before starting to accept only the moves that reduce the number of constraints violated. Figure.1a depicts the variation of error against the temperature for the 1% cooling rate. Figure 1.b depicts the same for the 5% cooling rate.

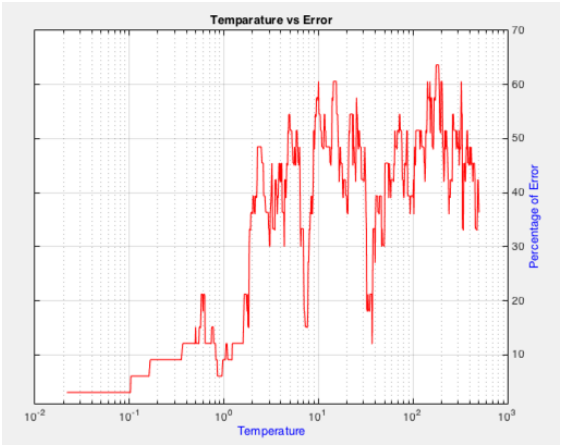

Figure a. 1% cooling rate

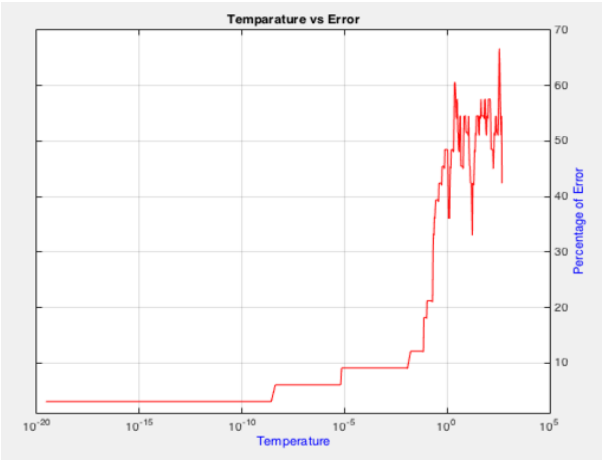

Figure b. 5% cooling rate

Figure 2 Temperature vs Error plot for different cooling rates

In general, the lower the cooling rate, the more probable that the algorithm reaches the global optimum. For high cooling rates, the algorithm may get stuck in a local optimum as the probability of accepting bad moves decreases at a high rate. But in our case the system returns a solution that satisfies all 34 constraints at both the 1% cooling rate and 5% cooling rate. The system took an average of 30 milliseconds and 28 milliseconds for the 1% cooling rate and 5% cooling rate respectively.

Figure 2 contains the snapshots of the solutions returned by the system at 1% cooling rate and 5% cooling rate. Both of the solutions satisfy all the constraints namely, the minimum and maximum number of credits per semester, prerequisites and co-requisites of courses etc.

4 SOLUTION REFINEMENT

Our implementation is a simplification of the real problem, and those simplifications must be removed to meet the needs of the institution if the college so desires. These changes do not affect the variables or the general algorithm, but will alter the list of courses in the domain and the constraints and supporting data structures. After obtaining machine-readable files of the list of courses and the general education, major, and minor requirements, the solution can be changed to replace place-holder courses with actual courses. Additional constraints that have been ignored such as semester typically offered can be added to the constraint checker, in most cases necessitating additional data elements or objects. Objects for majors and minors will be needed to allow the selection of individual requirement lists and other properties such as the division a major is in. Overall, these changes will require significant refactoring of the constraint checker.

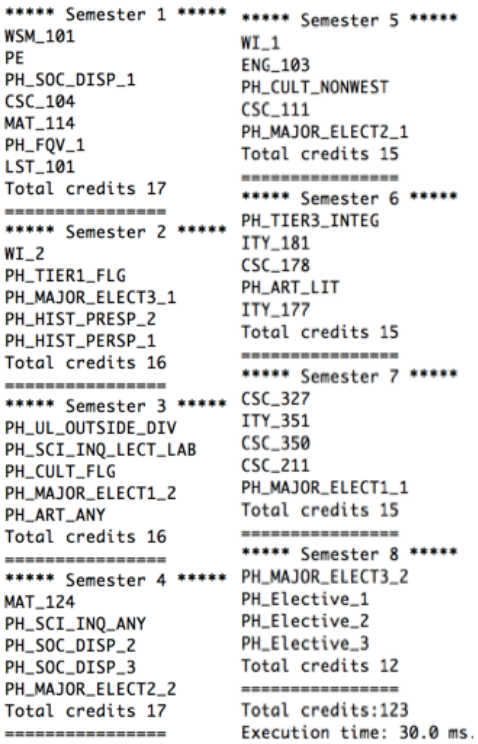

Figure c. At 1% cooling rate

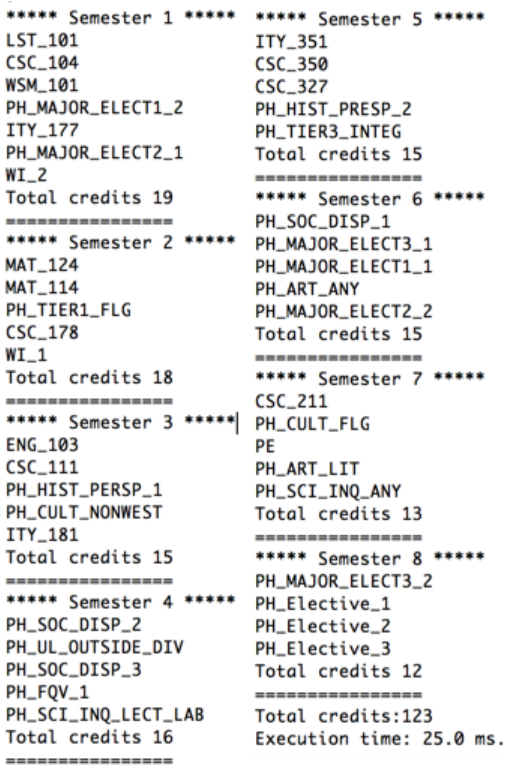

Figure b. At 5% cooling rate

Figure 2 Snapshots of solutions obtained at different cooling rates

We were able to use simple lists for the prerequisite and co-requisite lists in our simplified solution, but the full problem would require a more complicated structure to handle not just a list of courses that are all required, but lists of courses that can be selected between. This can be done with a nested list structure, a list of lists. Each item in the outer list is an individual course prerequisite or a list of courses that can be chosen among to satisfy the prerequisite (the inner list).

The algorithm can then be run with different numbers of majors and minors (limited to combinations of majors and minors that typically occur). For evaluating the altered algorithm, courses to fulfill given elective requirements can be randomly assigned in lieu of student selections.

If it is found that all constraints cannot usually be met, weighting of constraints can be used to preferentially assure that more important constraints have a higher likelihood of being met. This could be accomplished by changing total constraints and fulfilled constraints to total constraint weight and fulfilled constraints weight and making the adjustments in the constraint checker methods.

5 SUMMARY AND CONCLUSIONS

Looking back at the problem that was presented to us from Westminster College to design an algorithm that would be able to plan a four-year course sequence for a student, we can conclude this paper. The

simplified version of the problem for basic graduation requirements, one major, the general education requirements, and using placeholder courses for elective requirements proved to be quite feasible. By using a greedy algorithm to generate an initial state with semester-constrained courses already assigned to the proper semester and randomly sorting the remaining courses into semesters only considering the semester credit hours, we benefit on time by beginning with a partially good solution. Simulated annealing goes in afterwards and improves the areas that were overlooked by the greedy sort, including especially the prerequisites and co-requisites constraints. With this two-step approach, we were able to see a time of 30ms for a cooling rate of 1%, and 25ms for a cooling rate of 5% with a starting temperature of 500 degrees. These rates allowed most or all of our constraints to be met for our simplified problem. Further work on this project can lead to adding in multiple majors as well as minors and removing the place holders used for the specific courses that would be used for electives. With these additions, the problem becomes more complicated and will require more time for constraint checking. The state space will also increase as additional courses are required.

6 REFERENCES

[1] A. Gunawan, K. Ming Ng and K. Leng Poh, "Solving the Teacher Assignment-Course Scheduling Problem by a Hybrid Algorithm," *International Journal of Mechanical, Aerospace, Industrial, Mechatronic and Manufacturing Engineering,* vol. 1, no. 9, pp. 491 - 496, 2007.

[2] A. Gunawan, K. Ming Ng and K. Leng Poh, "An Improvement Heuristic for the Timetabling Problem," *International Journal of Computational Science 1992-6669 (Print) 1992-6677 (Online),* vol. 1, no. 2, pp. 162-178, 2007.

[3] E. Aycan and T. Ayav, "Solving the Course Scheduling Problem Using Simulated Annealing," in *2009 IEEE International Advance Computing Conference (IACC 2009)*, Delhi, 2009.

[4] K. N. Brown and I. Miguel, "Chapter21 Uncertainty and Change," in *Handbook of Constraint Programming*, Amsterdam, Elsevier, 2006, pp. 731-760.