CS 2800 Midterm Project Report

CLASS SCHEDULER USING A SAT SOLVER

Robert Sanders
Ohm Nabar

General Problem Area

The objective of this Class Scheduler program is to determine if there is an satisfiable schedule that could be determined and what exactly that schedule is. Our main curiosity in this endeavour results from spending lots of time in the past trying to derive our own schedules efficiently. Sometimes problems can arise, especially because classes can become full right before selecting courses. For these reasons, we decided to design a satisfiability problem around ensuring the available classes and sections are taken. To determine this valid schedule, we considered many factors such as the time of each class section, required courses, required subjects, and gen-ed requirement categories. This is a really interesting project, as every student at Northeastern selects their own schedule manually. They may be able to guess what sections to take, but most of the time students struggle to find a valid schedule. We designed our program to determine a satisfiable configuration of classes and sections for a student given the context of their courses and desired degree path.

Approach

Improvements on Traditional Approaches

Traditional approaches to finding a satisfiable schedule include ranking preference of classes, and guessing different combinations of sections to work in the students limited schedule. We realize that this randomized brute force section selection could lead to many invalid schedules being selected. This approach also includes pre-determining which classes to take in the semester. Our approach works better because considering the students' preference in conjunction with the timing of each section will help to optimize their schedule best.

Specificities of our Approach

We fully encode our problem as an input to a Boolean Satisfiability Solver. Our program is coded in Python and uses a Python library for the SAT solver itself. We know that our program works because the SAT solver appropriately declares a situation as being unsatisfiable or provides a sample schedule that a student could take to satisfy all the constraints given.

Goals

Our goals for this project were to gain more knowledge in the field of SAT solving by going through the process of translating a real-life problem into a SAT problem and learning how to use a real SAT solver. In addition, because we are currently college students who have to schedule classes every semester, this topic matter is of interest to us and depending on where this project takes us, it may potentially have real world applications to how we schedule classes in the future. To be of the most use to us, we intended this project to sufficiently outline a satisfiable schedule based on certain subject/class/time constraints. Overall, this project creates an interesting foray into the world of SAT solving.

Methodology

First, we have to generate mock class and section data for our program. We represent both of these as Python objects. Classes have a subject, a class number, and an associated NUPath requirement. Sections have a section id, a class name - which is a combination of a class's subject and number (e.g. "CS 2800"), and a start time. A class name can uniquely identify a class, and a section id can uniquely identify a section. The section ids are positive integers, which will allow them to be used as boolean variables in the SAT solver, where the variable "s" means the section with id "s" is being taken and the variable "-s" means the section with id "s" is not being taken. For the sake of simplicity, we choose to have all classes starting on the hour and lasting less than an hour so that one can tell if two classes overlap simply by comparing their start times. We create csv files representing 22 classes and 82 sections and read from those files in Python to create our data. We store the sections and classes in dictionaries whose keys are the identifying attributes for their values (class name for classes and section id for sections). Next, we must encode our built-in schedule constraints. A student may not take more than 4 sections, more than 1 section of the same class, or more than 1 section at the same time. These kinds of constraints are a particular type of "cardinality constraint", which is a constraint on the number of true variables from a list of variables. In this case, the relationship between the cardinality and the variables is <=, as in the number of true variables must be <= the given cardinality. Let us examine the naïve approach to this problem with our most complicated constraint, which limits the total number of sections to 4. This would require clauses (not converted to CNF for simplicity) like "$(s1 \wedge s2 \wedge s3 \wedge s4 \wedge -s5 \wedge -s6 \wedge ...) \vee (-s1 \wedge s2 \wedge s3 \wedge s4 \wedge s4 \wedge -s6 \wedge \ldots) \vee \ldots$". One can see that this would take n choose 4, or quartic, time. More troublingly, this would generate a quartic amount of clauses, which could slow down the SAT solver if the number of sections is high. We instead decide to use sequential counter encoding to enforce cardinality constraints. Given a set of variables (p1, p2, ..., pn) and a cardinality k, we generate nk new auxiliary variables such that variable cij represents whether the total number, or count, of true variables has reached j by pi. We require k clauses for our base case of p1. The first is $p_1 -> c_{11}(CNF: \ p_1 v s_{11})$, which represents the fact that if p1 is true, the count has reached 1 by p1. For all j such that 1 < j <= k, a clause s1j is created, which represents the fact that the count cannot be greater than 1 at the first variable. For all pi where i > 1, we require 2k + 1 clauses. If the count has reached 1 at the previous variable, or the current variable is true, then the count must have reached 1 at the current variable. This is represented by $\pi \vee (c(i-1)1) -> c_{i1}(CNF: (\pi \vee c_{11}) \wedge (c_{(i-1)1} \vee c_{i1}))$. Additionally, if the count has reached k by the previous variable, then the current variable cannot be true. This is represented by $s_{(i-1)k} -> \pi(CNF: \ c_{(i-1)k} \vee \pi)$. Finally, we add a constraint for all j such that $1 < j <= k: ((\pi \wedge c_{(i-1)(j-1)}) \vee c_{(i-1)j}) -> c_{ij}(CNF: (c_{ij} \vee \pi \vee \ c_{(i-1)(j-1)}) \wedge (c_{ij} \vee \ c_{(i-1)j})$. This states that if the current variable is true and the count has reached j-1 by the previous variable, or the count has reached j by the previous variable, then the count has reached j by the current variable. In total, we create k + (n-1)(2k+1) clauses. Given that we are dealing with very low cardinalities, this is effectively linear with regard to the number of variables. Now, we handle our user-defined constraints: section constraints, class constraints, subject constraints, NUPath constraints, and time constraints. A section constraint requires that a particular section be included in the schedule and creates a single variable clause s, where s is the section id of the desired section. A class constraint requires that a particular class be taken, and is encoded as $s1 \vee s2 \vee \ldots \vee sn$, where all si are sections of the desired class. A subject constraint is encoded the same way as a class constraint, except that all the sections' classes are of the desired subject. A time constraint takes in a range of acceptable times for sections to be scheduled and creates clauses $\ s1 \wedge \ s2 \wedge \ldots \wedge \ sn$, where all si are sections not in the specified range. When the user is ready, they ask the program to determine the satisfiability of their schedule constraints. If the schedule is satisfiable, the program will say so and give a possible assignment of classes; otherwise, it will say that the schedule is not satisfiable.

Results

Satisfying Goal

It was very interesting to analyze the output of our Class Scheduler along different constraints, and combinations of classes. We set out a goal to find an available schedule when provided with a dataset of classes. We successfully completed our goal, as will be demonstrated here. The sat solver finds a valid schedule based on the users desired sections, subjects, classes, timeframe, and nupath constraints.

Examples

Here we'll walk through some examples of the class scheduler to see the behavior it follows.

```
>>> c.add_class_constraint('CS 2500')
>>> c.solve()
Scheduler not running! Start scheduler with go()
>>> c.go()
>>> c.solve()
Satisfiable!
CS 2500 at 7:00, section_id: 1
>>> c.add_class_constraint('CS 1800')
>>> c.solve()
Satisfiable!
CS 2500 at 7:00, section_id: 1
CS 1800 at 9:00, section_id: 7
>>> c.add_class_constraint('MATH 2331')
>>> c.solve()
Satisfiable!
CS 2500 at 7:00, section_id: 1
CS 1800 at 9:00, section_id: 7
MATH 2331 at 8:00, section_id: 21
>>> c.add_class_constraint('PHIL 1111')
>>> c.solve()
Satisfiable!
CS 2500 at 7:00, section_id: 1
CS 1800 at 9:00, section_id: 7
MATH 2331 at 8:00, section_id: 21
PHIL 1111 at 15:00, section_id: 45
>>> c.add_class_constraint('ECON 1116')
>>> c.solve()
Unsatisfiable :(
>>>
```

Figure 1: Evaluating the satisfiability of adding class constraints

Adding Class Constraints

In this example we add a full schedule of classes, choosing sections with different starting times, and of different classes, and as we see The full schedule is satisfiable. Then we add 1 more class (ECON 1116), which shouldn't work because we can only take 4 classes, and as we see the Class Scheduler returns that it is unsatisfiable!

```
>>> from sat_solver import ClassScheduler; c = ClassScheduler(); c.go()
>>> c.add_section_constraint(1)
>>> c.solve()
Satisfiable!
CS 2500 at 7:00, section_id: 1
>>> c.add_section_constraint(6)
>>> c.solve()
Unsatisfiable :(
>>>
```

Figure 2: Evaluating combination of sections to unsatisfy the schedule

Adding Section Constraints

This example, adds the first section for 7:00, and then adds the 6th section which is also at 7:00. This is a contradiction, as we cannot take two classes at the same time, and appropriately so the Class Scheduler informs the user that this combination of classes is unsatisfiable

```
>>> from sat_solver import ClassScheduler; c = ClassScheduler(); c.go()
Use help() for help operating the scheduler
>>> c.add_subject_constraint('CS')
>>> c.solve()
Satisfiable!
CS 2500 at 7:00, section_id: 1
>>> c.add_subject_constraint('PHIL')
>>> c.solve()
Satisfiable!
CS 2500 at 7:00, section_id: 1
PHIL 1111 at 15:00, section_id: 45
>>> c.add_subject_constraint('MATH')
>>> c.solve()
Satisfiable!
CS 2500 at 7:00, section_id: 1
MATH 3400 at 13:00, section_id: 39
PHIL 2303 at 14:00, section_id: 56
>>> c.add_subject_constraint('ECON')
>>> c.solve()
Satisfiable!
CS 2500 at 7:00, section_id: 1
MATH 3400 at 13:00, section_id: 39
PHIL 2303 at 14:00, section_id: 56
ECON 1000 at 9:00, section_id: 60
>>> c.add_subject_constraint('THTR')
>>> c.solve()
Unsatisfiable :(
>>>
```

Figure 3: Evaluating combinations of subjects to unsatisfy the schedule

Adding Subject Constraints

This example evaluates adding in constraints for each subject. The Scheduler finds a minimum satisfiable section to fit from that subject. As we see trying to add in a 5th subject would not work as you can't take 5 classes.

```
>>> from sat_solver import ClassScheduler; c = ClassScheduler(); c.go()
Use help() for help operating the scheduler
>>> c.add_nupath_constraint('1')
>>> c.solve()
Satisfiable!
CS 2500 at 7:00, section_id: 1
>>> c.add_nupath_constraint('2')
>>> c.solve()
Satisfiable!
CS 2500 at 7:00, section_id: 1
CS 3200 at 9:00, section_id: 18
>>> c.add_nupath_constraint('3')
>>> c.solve()
Satisfiable!
CS 2500 at 7:00, section_id: 1
CS 3200 at 9:00, section_id: 18
THTR 1000 at 15:00, section_id: 80
>>> c.add_nupath_constraint('4')
>>> c.solve()
Satisfiable!
CS 2500 at 7:00, section_id: 1
CS 3200 at 9:00, section_id: 18
THTR 1000 at 15:00, section_id: 80
THTR 1433 at 10:00, section_id: 81
>>> c.add_nupath_constraint('5')
>>> c.solve()
Unsatisfiable :(
>>>
```

Figure 4: Evaluating nupath constraints added to the schedule

Adding NUPath Constraints

This example adds various nu path constraints, with the scheduler choosing the minimum possible class to create a satisfiable schedule. Here adding a 5th nupath constraint causes an error, as a student can't take 5 classes.

Adding Time Constraints

In this example we evalutate time constraints, as this student wants to only take classes between 10:00 and 16:00. We try selecting section number 1 for them, which is a 7:00 class, and doesn't fit in their time constraints. Therefore this schedule is unsatisfiable.

```
>>> from sat_solver import ClassScheduler; c = ClassScheduler(); c.go()
Use help() for help operating the scheduler
>>> c.add_time_constraint(10, 16)
>>> c.solve()
Satisfiable!
>>> c.add_section_constraint(1)
>>> c.solve()
Unsatisfiable :(
>>>
```

Figure 5: Evaluating adding section not within the time constraints

```
>>> from sat_solver import ClassScheduler; c = ClassScheduler(); c.go()
Use help() for help operating the scheduler
>>> c.add_time_constraint(10, 16)
>>> c.add_section_constraint(8)
>>> c.solve()
Satisfiable!
CS 1800 at 11:00, section_id: 8
>>> c.add_section_constraint(13)
>>> c.solve()
Satisfiable!
CS 1800 at 11:00, section_id: 8
CS 2800 at 15:00, section_id: 13
>>> c.add_class_constraint('PHIL 1111')
>>> c.solve()
Satisfiable!
CS 1800 at 11:00, section_id: 8
CS 2800 at 15:00, section_id: 13
PHIL 1111 at 10:00, section_id: 42
>>> c.add_class_constraint('ECON 1000')
>>> c.solve()
Satisfiable!
CS 1800 at 11:00, section_id: 8
CS 2800 at 15:00, section_id: 13
PHIL 1111 at 12:00, section_id: 43
ECON 1000 at 10:00, section_id: 61
>>>
```

Figure 6: Evaluating adding time and class constraints

Adding Time and Class Constraints

In this example we evaluate a full schedule of a student with strict time constraints. The desired schedule times are from 10:00 to 16:00. We add the sections 8 and 13. Then we want to take two more classes, so we choose those class constraints. The scheduler chooses sections so as to not interfere with the prior sections 8 and 13.

Adding Time and Subject Constraints

In this schedule, we set the same time constraints of 10:00 to 16:00, and add 3 exact class constraints. We then add Philosophy as a subject to find a section of philosophy to fit within those time constraints and it successfully chooses a generic philosophy class this student could fit into their schedule.

```
>>> from sat_solver import ClassScheduler; c = ClassScheduler(); c.go()
Use help() for help operating the scheduler
>>> c.add_time_constraint(10, 16)
>>> c.add_class_constraint('CS 2500')
>>> c.solve()
Satisfiable!
CS 2500 at 11:00, section_id: 3
>>> c.add_class_constraint('CS 1800')
>>> c.solve()
Satisfiable!
CS 2500 at 11:00, section_id: 3
CS 1800 at 13:00, section_id: 9
>>> c.add_class_constraint('CS 2800')
>>> c.solve()
Satisfiable!
CS 2500 at 11:00, section_id: 3
CS 1800 at 13:00, section_id: 9
CS 2800 at 12:00, section_id: 12
>>> c.add_subject_constraint('CS')
>>> c.solve()
Satisfiable!
CS 2500 at 11:00, section_id: 3
CS 1800 at 13:00, section_id: 9
CS 2800 at 12:00, section_id: 12
>>> c.add_subject_constraint('PHIL')
>>> c.solve()
Satisfiable!
CS 2500 at 11:00, section_id: 3
CS 1800 at 13:00, section_id: 9
CS 2800 at 12:00, section_id: 12
PHIL 1115 at 15:00, section_id: 49
>>>
```

Figure 7: Evaluating adding time and subject constraints

Adding Limited Time Constraint

In this schedule, we show how a schedule with only 3 hours of time constraints is unsatisfiable when choosing 4 classes, as we couldn't choose 4 sections without time overlap.

Generalization of Project

The class scheduler can be generalized really well. We could feed in a real schedule, such as Northeastern's Spring 2021 schedule, to find a satisfiable schedule of sections we could take, based on which classes we need, which time constraints we have, or even which subject we might need another class in.

Lingering Thoughts

There are some other factors we would have liked to consider for the Class Scheduler. To be more helpful for upper class students, we could add a way to exclude certain classes from being selected when adding in a subject constraint. We wouldn't want the Scheduler to tell

```
>>> from sat_solver import ClassScheduler; c = ClassScheduler(); c.go()
Use help() for help operating the scheduler
>>> c.add_time_constraint(10, 16)
>>> c.add_class_constraint('CS 2500')
>>> c.solve()
Satisfiable!
CS 2500 at 11:00, section_id: 3
>>> c.add_class_constraint('CS 1800')
>>> c.solve()
Satisfiable!
CS 2500 at 11:00, section_id: 3
CS 1800 at 13:00, section_id: 9
>>> c.add_class_constraint('CS 2800')
>>> c.solve()
Satisfiable!
CS 2500 at 11:00, section_id: 3
CS 1800 at 13:00, section_id: 9
CS 2800 at 12:00, section_id: 12
>>> c.add_subject_constraint('CS')
>>> c.solve()
Satisfiable!
CS 2500 at 11:00, section_id: 3
CS 1800 at 13:00, section_id: 9
CS 2800 at 12:00, section_id: 12
>>> c.add_subject_constraint('PHIL')
>>> c.solve()
Satisfiable!
CS 2500 at 11:00, section_id: 3
CS 1800 at 13:00, section_id: 9
CS 2800 at 12:00, section_id: 12
PHIL 1115 at 15:00, section_id: 49
>>>
```

Figure 8: Evaluating not enough time constraint

a student to take a CS class they've already taken, so it would be useful for the sat solver to consider already taken classes. These are things that would have required expanding our cardinality constraints, which would have been tough given the time frame of our project. Another Aspect of this project we wanted to expand upon was to consider the optimal schedule, not just a minimum satisfiable schedule. There are multiple satisfiable schedules based on class constraints, subject constraints, etc. so it would be amazing to expand the scope of our project to consider other factors or even allow for the user to select an optimal schedule out of the satisfiable schedules. We could have considered closeness of classes, and some favorability rankings to determine an optimal schedule out of a series of satisfiable schedules.

Summary

As we can see from the provided examples the class scheduler SAT solver provides a solid framework for determining a valid schedule for a student considering some constraints. The

Scheduler proved to fulfill many uses for students. Some of these include but are not limited to needing to find a last section to fill in their schedule to fit within a certain time constraint, a section to fit within a certain subject constraint or a student who needs to take classes but doesn't know which times to pick for any of them. The Scheduler allows for any of these uses, plus many more. For the students of Northeastern, this can provide a useful tool for generating a valid schedule, or choosing a class they may need to fulfill a requirement. The problem of determining a most optimal schedule out of a series of satisfiable schedules is an interesting problem that was made easier by the completion of this satisfiability problem. We now understand that we can utilize cardinality constraints to solve a Class Schedule across a variety of fields. We can leverage this to find many satisfiable schedules, and thus find the most optimal schedule over these satisfiable schedules using several optional parameters.

References

1

Ignatiev, A., Morgado, A., amp; Marques-Silva, J. (2018). SAT technology in Python¶. Retrieved November 02, 2020, from https://pysathq.github.io/index.html

2

Reasoning, A. (2020, September 03). Lecture 08-2 Encoding cardinality constraints. Retrieved November 02, 2020, from https://www.youtube.com/watch?v=PKbAICDB9tM