CS 2800 MIDTERM PROJECT REPORT

CLASS SCHEDULER USING A SAT SOLVER

Robert Sanders
Ohm Nabar

General Problem Area

The objective of this program will be to determine if there is an optimal schedule that a student can have, and what exactly that optimal 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 realized how unique it would be to design a satisfiability problem around ensuring the best available classes are taken. To determine the optimal schedule, we will consider 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 an optimal schedule. We want to utilize our program to determine an optimal 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 an optimal 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 unsatisfiable schedules being selected. This approach also includes pre-determining which classes to take in the semester. Our approach will work 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 will need to fully encode our problem to be provided as an input to a Boolean Satisfiability Solver. Our program will be coded in Python and will use a Python library for the SAT solver

itself. We will know if our program works if the SAT solver terminates in a reasonable amount of time and either declares a situation as being unsatisfiable or provides a sample schedule that a student could take to satisfy all the constraints given.

Intended Goals

By doing this project, we will be gaining 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 intend this project to sufficiently outline a satisfiable schedule based on certain subject/class/time constraints. Overall, this project should be an interesting foray into the world of SAT solving.

Methodology

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.

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,

```
>>> 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

```
>>> 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

and doesn't fit in their time constraints. Therefore this schedule is unsatisfiable.

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.

```
>>> 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

```
>>> 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 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 8: Evaluating not enough time constraint

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 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.

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.

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