GRP_5: Schedule Sensei

Corey McCann (21CCM12)

Mattias Khan (22MHK)

Hayden Jenkins (21HJDM)

Vasilije Petrovic (21VMP8)

Course Modelling Project

CISC/CMPE 204

Logic for Computing Science

December 7, 2023

Abstract

Our project introduces an innovative logical model designed to assist students in navigating the complexities of course schedule planning. This tool is structured around customization; students can input offered courses and their respective sections, as well as specific course requirements. A student can then specify the courses they wish to take and with whom. The model will then evaluate key factors such as program-specific requirements, course prerequisites, and potential time-slot conflicts. Leveraging the principles of natural deduction, our model attempts to produce a feasible schedule.

The current model contains predefined test cases, each representing typical scheduling scenarios students might encounter. These cases are specifically chosen to showcase the model's proficiency in handling a variety of scheduling-related challenges.

Our model and its modern approach offer students a practical and insightful tool for academic timetable planning. Interacting with this model allows students to gain valuable insight into complex decision-making and planning-related problems. Experimenting with how course requests, friendships, and previously completed courses affect and interact with constraints. In essence, our project serves as a connection between theoretical course planning and real-world academic scheduling, providing a highly customizable platform for students to understand the complex reality of their educational journey.

Propositions

StudentEnrolledCourse_x_y: True if student x is enrolled in course y

StudentEnrolledCourseTerm_x_y_z: True if student x is enrolled in course y in term z

StudentCourseRequiredTerm_x_y_z: True if a student x must take a course y in term z in order to satisfy some prerequisite(s)

StudentEnrolledCourseSection_x_y_z: True if student x is enrolled in course y in section z

CourseTermSectionTimeConflict_x_y: True if there is a time conflict between two courses x and y (courses are indexed by section and term as well)

CourseTermSectionAvailableCapacity_x_y_z: True if there are seats available for course x in term y in section z

Friendship_x_y: True if student x and y are friends

CheckCourseExclusionsExists_x_y_z: True if a course x (excluded_course) that is an exclusion for another course y (course) is present in a student z schedule

CourseExclusionRequirement_x_y: True if the exclusion requirement is satisfied for course x and student y

CheckCoursePrerequisitesExists_x_y_z: True if a prerequisite course x (required_course) for another course y (course) has been completed by a student z

CoursePrerequisiteRequirement_x_y: True if the prerequisite requirement is satisfied for course x and student y

CheckCourseCorequisitesExists_x_y_z: True if a corequisite course x (required_course) for another course y (course) is being taken concurrently by a student z

CourseCorequisiteRequirement_x_y: True if the corequisite requirement is satisfied for a course x and a student y

Constraints

Enrollment Rules

Single Term Enrollment: For every student and every course, if a student wishes to take a course they must take the course in one of the terms, i.e., "FALL", "WINTER", "SUMMER".

Single Term: For every student and every course that they wish to take, a student can be enrolled in a course during only one term ex: one of "FALL", "WINTER" or "SUMMER" depending on a courses offering.

Single Section: For every student and every course that they wish to take, a student can only be enrolled in exactly one section of a course.

Term Enrollment Consistency Check: For every student and every course that they wish to take, if a student is enrolled in a course during a specific term they must be taking that course during that term.

At Most 10: A student can be enrolled in at most 10 courses during a year.

Enrollment Restrictions

Time Conflict: For every student and every course that they wish to take, if any section of a course has a time conflict with a section of another course, the sections cannot both be taken.

Section Enrollment Capacity: A student can only enroll in the section of a course if there is capacity in that section.

Enroll Until Max Capacity: Enroll only as many Students in a section as there is room for.

Enrollment Requirements

Course Exclusion Rule: For every student and every course, if a course that a student wishes to take has an exclusion rule in its requirements, then such a course should not exist in the student's course history or in the list of courses they wish to take should.

Prerequisite Constraint: For every student and every course, if a course that a student wishes to take has a prerequisite rule in its requirements, then the prerequisite course must exist in the student's course history, i.e., have already been taken.

Course Corequisite Constraint: For every student and every course, if a course that a student wishes to take has a corequisite rule in its requirements, then the corequisite course must exist in the student's course history or the student must take the corequisite course at the same time or before the course.

Course Requirements Check: For every student and every course in a student wishlist, if a student is enrolled in a course then the course's exclusions, prerequisites, corequisites, and program requirements are all satisfied.

Friendship Constraints

Friendship Consistency Check: If two students are both friends, then they must have a friendship, i.e., they both agree that they are friends.

Restrictions Between Friends: If two students are friends and are enrolled in the same course, then there are no prior restrictions that affect them both.

Model Exploration

Original Plan

The project's initial goal was ambitious: allow students to input their desired course schedule into our model and assess its feasibility. The inital concept raised numerous questions as to how users would provide data and how such data would be managed and then used throughout the SAT solver. Rooted in complex data analysis, handling, and manipulation, the execution and success of the project depended on data.

Evolution of the Project

Post-proposal, we discovered an extensive data set related to courses and programs at Queen's University from 2019/20. The data sets used from QMULUS were:

Buildings: Contained information about buildings on the Queen's University campus.

Courses: Contained data related to the courses offered at the university during 2019/20.

Departments: Details about different departments within the university.

Sections: Data pertaining to specific sections, including details like timing, location, and instructors.

These data sets provided us with structured json data that we could use to populate and enhance custom student timetables. A large portion of the initial work on the project began with creating a robust and yet intuitive datalayer that managed all the handling and complex querying of the json data. This is the `datalayer.py`. Such a large portion of the initial project was dedicated to this module as we new that the easy and intuitive querying and manipulation of the data sets would create a clean and confusion free code for the main part of the project, the SAT solver.
Decisions were also made to break the project down into smaller modules to allow for greater modularity and maintainability. As a result, we split the entire codebase into several different modules, each responsible for the different aspects of the project. These modules included:

- **DataLayer Module:** Responsible for storing, handling and querying the JSON data. This module is the backbone of the entire project, it provides a seamless and intuitive layer to access the extremely sophisticated an extensive data set that we were able to obtain. This module is the `datalayer.py`.
- **SAT Solver Module:** This module uses the data provided by the DataLayer to create potential timetables based on student inputs and the constraints specified earlier. This is the main module of the project, `sat_solver.py`.
- **Visualization Module:** Also at the heart of the project, this module creates a modern and simplistic view of the solutions provided by the SAT solver. This module in reality is a combination of multiple different modules. `run.py`, `timetable.py` and `webapp_api.py`.

With this modular approach, we aimed to create a system where each part could be independently developed, tested, and improved upon without affecting the others. This structure not only facilitated easier debugging and maintenance but also allowed team members to specialize and focus on specific areas of the project.

Post-draft, a suggestion to create schedules for multiple students led us to explore the concept of "friendships" between students. This innovative approach expanded our model to accommodate the dynamics of group scheduling, resulting in the development of new constraints and propositions to facilitate this feature.

Current Status

As described above, the complexity of handling an extensive course dataset proved challenging, limiting our ability to fully implement user-driven schedule inputs. Instead, we pivoted to a more controlled approach, relying on the existing data from QMULUS:

- **Predefined Test Cases:** We developed specific test cases that users can select from. These cases are carefully designed to demonstrate the capabilities of our model in various realistic scenarios, including the new "friendship" feature among students. These predefined test cases are essential for ensuring that even inexperienced or non-technical users can properly engage with our project without confusion. By offering a selection of ready-to-use scenarios, users are spared the complexity of setting up and integrating new data.
- **Student Wishlist Integration:** Instead of a full schedule input from users, we shifted to a system where students have a "wishlist" of courses that they wish to enroll in, from which the model uses as a basis for testing feasibility.
- **Web Application Interface:** To enhance user experience and provide a clearer understanding of the solutions produced by our model, we developed a web application. This platform allows users to run selected test cases in the SAT solver and visually explore the course schedules for each term, in a calendar.
- **Scale** Recognizing the limitations imposed by our project's data complexity, we scaled down the test cases to include only a few students and courses. Despite this reduction, we believe these scenarios successfully illustrate our model's functionality and effectiveness.

Challenges and Solutions

- **Data Management:** The most obvious and largest problem we faced was managing and organizing the large datasets used by our model. As discussed earlier we addressed these issues by spending a large amount of time creating a robust and intuitive datalayer that controlled and managed all the JSON objects and any of their relationships. Focusing on smaller and more manageable test data sets/cases, allowed quicker customization of specific properties ensuring correct handling of all edge cases.

- **Constraint & Proposition Complexity:** The constraint and proposition implementations provided the largest challenges for obvious reasons. Several key issues and solutions are outlined below:

  For starters there was a significant learning curve, in adapting to the new concept of constraint-based modeling. Understanding the Bauhaus library and how to properly create propositions and constraints were a large portion of our issues early on in the development process.

```
#For every student and course, they can be enrolled in exactly one section
for student in students:
    for course in student.courses:
        for term in course.section:
            section_options = []
            for course_section in course.section[term].course_sections:
                section_options.append(StudentEnrolled(student, course, term,
                ↪  course_section))
            constraint.add_exactly_one(E, section_options)

#For every student and course, they can only be enrolled in a course during one term
for student in students:
    for course in student.courses:
        term_options = []
        for term in course.section:
            coursecode = course.section[term].courseid
            term_options.append(StudentCourse(student, coursecode, term))
        constraint.add_exactly_one(E, term_options)

#For every student and course, if they are not taking the course during a term they
↪  should not be enrolled in any of its sections
for student in students:
    for course in student.courses:
        for term in course.section:
            for course_section in course.section[term].course_sections:
                E.add_constraint(~StudentCourse(student,
                ↪  course.section[term].courseid, term) >> ~StudentEnrolled(student,
                ↪  course, term, course_section))


Satisfiable: True
         Solutions: 8
         Solution:
        {(Student in STAT-263 T: Winter): False,
        (Student in STAT-263 T: Fall): True,
        (Student in STAT-263 T: Fall -> 4490): True,
        (Student in STAT-263 T: Fall -> 4491): False,
        (Student in STAT-263 T: Winter -> 4036): True,
        (Student in STAT-263 T: Winter -> 4043): False}
```

In the code snippet outlined above you can see a few of our initial constraints and propositions. The first constraint outlines the rule that a student can be enrolled in exactly one section, the second that a student can only be enrolled in a course during one term, and the third that if a student isn't enrolled in a course during a given term then they shouldn't be enrolled in any of its sections. Given the propositions in our solution, you can see that this isn't working. The student has been enrolled in one term, "Fall" and has been enrolled in one section per term, but has also been enrolled in a section during the "winter" term. Surprisingly it took us quite a while to discover the issue. The issue we were encountering resulted from the interaction between `constraint.add_exactly_one` and our third constraint. `constraint.add_exactly_one` ensures that exactly one of the propositions in the given list is true, however, it doesn't allow other constraints or implications to have an effect of the choice, leading to contradictory scenarios and impossible solutions. This function works great for enforcing the enrolment of a single term but in this case, i is enforcing the enrolment of at least one section per term. Our solution at the time was to use `constraint.add_at_most_one` however this later caused more undesirable behavior.

The usage of `constraint.add_exactly_one` and `constraint.add_at_most_one` was feasible until we decided to implement the friendship functionality. The friendship constraints ensured that two friends were enrolled in the same section of a course during the same term. This became quite complicated as we had to consider the enrolment requirements for both students to choose which term to enroll them in. This meant that already enforcing a course term selection for a given student using `constraint.add_at_most_one` and then trying to enforce that the two students were in the same term and section would result in no solutions. Therefore we had to reinvent our original constraints without forcing propositions as true or false. We further broke down our constraints into even more basic rules.

```
#CONSTRAINT 0 - Course -> Term
    #For every student and course, if a student is taking a course they must be taking
    ↪  the course in one of the available terms.
    for student in students:
        if len(student.course_wish_list) != 0:
            for course in student.course_wish_list:
                offered_terms = course.sections.get_term_offerings()
                offerings = []
                for term in offered_terms:
                    offerings.append(StudentEnrolledCourseTerm(student, course, term))
                offerings = Or(offerings)
                E.add_constraint(StudentEnrolledCourse(student, course) >>
                ↪  (offerings))

    #CONSTRAINT 0.1 - One Term Per Course
    #For every student and course, they can be enrolled in the course during only one
    ↪  term.
    for student in students:
        for course in student.course_wish_list:

            offered_terms = course.sections.get_term_offerings()
            for term in offered_terms:
                other_terms = []
                for other_term in offered_terms:
                    if term != other_term:
                        other_terms.append(StudentEnrolledCourseTerm(student, course,
                        ↪  other_term))
                other_terms = Or(other_terms)
                E.add_constraint(StudentEnrolledCourseTerm(student, course, term) >>
                ↪  (~other_terms))
```

The first constraint enforces that if a student is enrolled in a course, they must be enrolled in a term. We accomplished this by looping over the the available terms for a course and appending them to a list and then separating the elements of this list with the Or function. However, this didn't enforce that a student could only be enrolled in one term. Therefore in our second constraint, we used a nested for loop to consider all possibilities of term enrollments. Building a list of all the other options, splitting this list with the Or function. This constraint enforced that if a student was enrolled in a term, then they couldn't be enrolled in any of the other terms. These two constraints accomplished what `constraint.add_exactly_one` and `constraint.add_at_most_one` were trying to accomplish but without forcing a proposition to be true or false at that stage in the model. Allowing the model to properly generate solutions and allow other constraints such as course requirements and friendships to influence the decision of term selection, without creating cases of no solutions. However, we did still use `constraint.add_none_of` in our final solution to enforce that certain propositions were false given certain outcomes, for example, if a prerequisite requirement wasn't satisfied.

Variable/Dynamic Criteria

The most challenging part of the model was implementing the course requirement constraints. Consider for example the requirements for CISC-271 as found in the JSON data set from QMULUS:

```
"requirements": "Prerequisites Level 2 and C- in (CISC101 or CISC121) and (MATH111 or
↪  MATH112 or MATH110) and (MATH120 or MATH121 or MATH122 or [MATH123 and MATH124] or
↪  MATH126).\nExclusion No more than 3.0 units from CISC271; MATH272; PHYS313."
```

The raw requirements had no obvious way to parse course exclusions, prerequisites, or corequisites. Which meant we had to create a new subset of data that could be used to test the requirement constraints. We create a data set for all CISC courses using the following schema.

```
{
    "id": "CISC-271",
    "requirements": [
      {
        "type": "PREREQUISITE",
        "criteria": "(CISC-101 OR CISC-121) AND (MATH-111 OR MATH-112 OR MATH-110) AND
        ↪  (MATH-120 OR MATH-121 OR MATH-122 OR (MATH-123 AND MATH-124) OR MATH-126)"
      },
      {
        "type": "EXCLUSION",
        "criteria": "MATH-272 OR PHYS-313"
      },
      {
        "type": "PROGRAM REQUIREMENT",
        "criteria": "NONE"
      },
      {
        "type": "APPROVAL",
        "criteria": "NONE"
      },
      {
        "type": "COREQUISITE",
        "criteria": "NONE"
      }
    ]
  }
```

The new formatted data allowed for easier parsing but still presented a significant challenge in implementing constraints due to its variability. The dynamic criteria for each requirement somehow needed to be parsed and executed as a constraint in the Bauhaus library. The following solution was achieved:

```
for student in students:
    for course in student.course_wish_list:
        prerequisite_courses = utils.extract_courses(prerequisite_rule)

        for check_course in prerequisite_courses:
            prerequisite_exists = CheckCoursePrerequisitesExists(student.name,
            ↪  course.id, check_course)

            #If a course that is in the prerequisite rule has  been taken, then the
            ↪  prerequisite rule has been satisfied
            if check_course in str(student.completed_courses):
                constraint.add_exactly_one(E,[prerequisite_exists]) #force the
                ↪  proposition to true, i.e an prerequisite is present

            else:
                constraint.add_none_of(E,[prerequisite_exists]) #force the proposition
                ↪  to false, i.e an prerequisite is not present

            prerequisite_rule = prerequisite_rule.replace(check_course,
            ↪  f"CheckCoursePrerequisitesExists('{student.name}', '{course.id}',
            ↪  '{check_course}')" )
            prerequisite_rule =
            ↪  prerequisite_rule.replace("AND","&").replace("OR","|").replace("NOT",
            ↪  "~")
            prerequisite_rule = f"({prerequisite_rule})"
            requirement_met = f"(CoursePrerequisiteRequirement('{student.name}',
            ↪  '{course.id}'))"

        new_constraint = f"E.add_constraint(({prerequisite_rule} & {requirement_met})
        ↪  | (~{requirement_met} & ~{prerequisite_rule}))"
        exec(new_constraint)
```

Using string manipulation and the `exec` function we were able to dynamically generate constraints that ensured that all course prerequisites were satisfied. The code loops over all the courses that a student wishes to take and fetchs the prerequisite criteria, extracting all the courses in the criteria. We then loop over each of these courses and create a CheckCoursePrerequisitesExists proposition representing the specific course that is being checked. If the course has already been taken then this proposition is set to true otherwise false. We then replace the course code in the actual string representation of the criteria with the literal Python code to create this proposition. Replacing the "AND", "OR" and "NOT" operators with their Bauhaus equivalences allows us to generate the string version of the Python code needed to execute this constraint. Adding the final part of the constraint (that being enrolled in this course means that the dynamic prerequisite criteria must be satisfied) gives us a complete constraint that we can execute using the `exec` function. This somewhat non-trivial solution allowed us to implement all the variable/dynamic criteria constraints.

Solution Display

As our project grew in size and constraints kept evolving it became increasingly harder to ensure that all of the test cases were being satisfied correctly, especially the timeslot constraints. Trying to make sense of the propositions that were being spat out into the console became impossible, especially with multiple students each enrolling in several courses. We decided to implement a Nextjs web app to simplify both the execution of test cases and the display of timetable solutions.

The Solutions from the SAT solver are passed to the `timetable.py` where the data is extracted from the propositions and added to new display objects which are then serialized and sent to the Nextjs web app over HTTP. The SAT solver was also wrapped in a flask app `webapp_api.py` so that the Nextjs web app could send solve requests for a particular test cases to the SAT Solver and have the solutions returned.

Through the use of the Nextjs web app we have simplified and modernized the overall user experience.

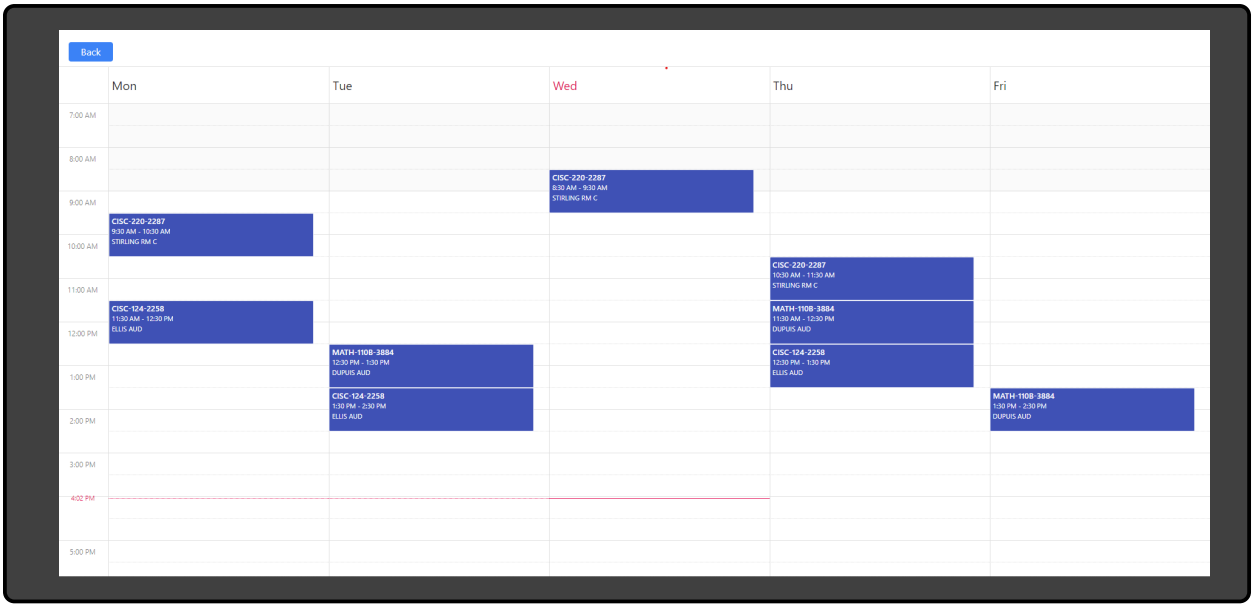

Figure 1: Example Timetable

Friendships

Post-draft, a suggestion to create schedules for multiple students led us to explore the concept of "friendships". This addition require revisiting and modifying almost all aspects of the model to accommodate multiple student. It also added more complexity in ensuring that each student's schedule not only met individual constraints by also satisfied the constraints required to have a friendship between students. Exploring this possibility and deciding to implement it introduced a complex multi-dimensional problem space that allowed us to explore how even small changes to a single students course whishlist or prerequisite completions could have drastic changes to another students schedule. The largest challenge was implementing the logic that could consider which classes would satisfy both the students varying course history, forcing a balance between collective preferences and individual needs.

```
#CONSTRAINT 12 - If students are friends and wish to be enrolled in the same course,
↪  they may. If and only if there are no prior restrictions that affect them both.
for student in students:
    if student.has_friends():
        for friend in student.friends:
            for course in friend.shared_courses:
                if student.is_reciprocal(friend, course): #if the friendship and
                ↪  course selection is mutual
                    friend = datalayer.Students.ALLSTUDENTS[friend.name]

                    term_options = [] #the term options 2 students can take a course
                    ↪  in
                    section_options = [] #the section options 2 students can take a
                    ↪  course in
                    for term in datalayer.Term:
                        term_offerings_course =
                        ↪  course.sections.get_term_collection(term)
                        term_options.append(StudentEnrolledCourseTerm(student, course,
                        ↪  term) & StudentEnrolledCourseTerm(friend, course, term))

                        for section in term_offerings_course:

                            ↪  section_options.append(StudentEnrolledCourseSection(student,
                            ↪  course, term, section) &
                            ↪  StudentEnrolledCourseSection(friend, course, term,
                            ↪  section))

                    common_term = Or(term_options)
                    common_section = Or(section_options)


                    E.add_constraint((StudentEnrolledCourse(student, course) &
                    ↪  StudentEnrolledCourse(friend, course) & Friendship(student1,
                    ↪  student2)) >> (common_section))
```

Above is a highlighted snippet for one of the friends constraints. This constraint enforces that if two friends are friends and they both wish to take the same course, and both satisfy all requirements to take such course, they should both be enrolled in the same section. This is a fairly straightforward constraint that would be trivial in friendship relationship. However where our model differs is in its ability to problem solve and create solutions for less than ideal models.

An unfortunate result of both our course requirement constraints and our friendship constraints were there dependence one one another. If a student didn't satisfy all the requirements for a course our

model wouldn't produce a solution, and thus the friendship wouldn't exists. This convinced us to explore other alternatives to avoid so many impossible solutions. Revisiting our course requirement constraints, prompted us to implement functionality to resolve the conflicts. Rather than not producing a solution because a student didn't satisfy a course perquisite or co-requisite, our model would attempt to enroll the student in the required course to satisfy the requirement rule.

```
#If a course that is in the prerequisite rule has  been taken, then the prerequisite
↪  rule has been satisfied
if check_course in str(student.completed_courses) or check_course+'A' in
↪  str(student.completed_courses) or check_course+'B' in
↪  str(student.completed_courses):
    constraint.add_exactly_one(E,[prerequisite_exists]) #force the proposition to
    ↪  true, i.e an prerequisite is present

#If a course that is in the prerequisite rule has not already been taken and is not
↪  being taken before the course in question, then the prerequisite rule has been
↪  broken, therefore the course must be taken at the same time or before.
elif check_course in str(student.course_wish_list):
    offered_terms = course.sections.get_term_offerings()
    for term in offered_terms:
        other_terms = []
        if term == datalayer.Term.FALL:
            if datalayer.Term.WINTER in offered_terms:
                other_terms.append(datalayer.Term.WINTER)
            if datalayer.Term.SUMMER in offered_terms:
                other_terms.append(datalayer.Term.SUMMER)
        elif term == datalayer.Term.WINTER and datalayer.Term.SUMMER in offered_terms
        ↪  and datalayer.Term.SUMMER:
            other_terms.append(datalayer.Term.SUMMER)
        if other_terms != []:
            options = []
            for o_term in other_terms:
                options.append(StudentEnrolledCourseTerm(student,
                ↪  student.course_wish_list[check_course], o_term))
            options = Or(options)
            E.add_constraint(StudentEnrolledCourseTerm(student, course, term) >>
            ↪  (options))
        else:
            E.add_constraint(StudentEnrolledCourseTerm(student,
            ↪  student.course_wish_list[check_course], term) >>
            ↪  ~(StudentEnrolledCourse(student, course)))
    constraint.add_exactly_one(E,[prerequisite_exists]) #force the proposition to
    ↪  true, i.e a prerequisite is present
```

The first if statement in the above code block checks if a students course prerequisite has been satisfied. If it has not then the model attempts to enroll the student in the course during a term that would satisfy the criteria. The constraints enforce that a student must take a prerequisite course in a term before the course in question. The actual constraint logic gets alot more complicated when we consider the availability of each of the courses and whether either or them is full year. The implementation of this is accomplished by compiling a list of possible enrollment terms for the prerequisite course given each of the possible enrolment terms for the main course. Enforcing that if a student is enrolled in the main course then he must be enrolled in the prerequisite course in a term prior to the main courses term.

First-Order Extension

Our extension to predicate logic has been carefully designed to incorporate the major aspects of our problem domain that are the backbone of our logical model.

PREDICATES
StudentProgram(x,y): student x is in program y
EnrolledCourseSection(w,x,y,z): student w is enrolled in course x in term y in section z
EnrolledCourse(x,y): student x is enrolled in course y
Prerequisite(x,y): course x is a prerequisite for course y
CourseTerm(x,y): course x is in term y
MandatoryCourse(x,y): course x is a mandatory course for program y

Types for Individual Objects
Student(x): x is a student
Course(x): x is a course
Term(x): x is a term
Section(x): x is a section
Program(x): x is a program
Time(x): x is a specific course time
assume that we have equality for objects $(x = y)$ or $\neg(x = y)$
We also need to assume that we can index the EnrolledCourse predicate by year for readability and conciseness

CONSTRAINTS

A student can only be enrolled in exactly one section of a course.

$$\begin{aligned}\forall v \forall w \forall x \forall y \forall z.\, ((&\text{Student}(v) \land \text{Course}(w) \land \text{Term}(x)\\ &\land \text{Section}(y) \land \text{Section}(z) \land \neg(y = z))\\ \rightarrow\ &(\text{EnrolledCourseSection}(v, w, x, y)\\ &\rightarrow \neg\text{EnrolledCourseSection}(v, w, x, z)))\end{aligned}$$

A student cannot take two courses simultaneously if those courses have a time conflict

$$\begin{aligned}\forall s \forall t \forall u \forall v \forall w \forall x \forall y \forall z.\, ((&\text{Student}(s) \land \text{Course}(t) \land \text{Course}(u) \land \text{Term}(v)\\ &\land \text{Section}(w) \land \text{Section}(x) \land \text{Time}(y) \land \text{Time}(z)\\ &\land \neg(u = t))\\ \rightarrow\ &((y = z) \rightarrow (\text{EnrolledCourseSection}(s, t, v, w)\\ &\lor \text{EnrolledCourseSection}(s, u, v, x))))\end{aligned}$$

If a student is enrolled in a program, then all mandatory classes for that program must be included in the student's schedule.

$$\begin{aligned}\forall x \forall y \forall z.\, ((&\text{Student}(x) \land \text{Course}(y) \land \text{Program}(z)) \\ &\rightarrow ((\text{StudentProgram}(x, z) \land \text{MandatoryCourse}(y, z)) \\ &\quad \rightarrow \text{EnrolledCourse}(x, y)))\end{aligned}$$

If a course C1 is a prerequisite for another course C2, then the student must take course C1 either in a previous year or before course C2 in the same year

$$\begin{aligned}\forall x \forall y \forall z.\, ((&\text{Student}(x) \land \text{Course}(y) \land \text{Course}(z) \land \neg(y = z)) \\ &\rightarrow (\text{prerequisite}(y, z) \\ &\rightarrow ((\text{EnrolledCourse}(x, y)_{\text{term1}} \land \text{EnrolledCourse}(x, z)_{\text{term2}}) \\ &\quad \lor (\text{EnrolledCourse}(x, y)_{\text{previous year}} \land \text{EnrolledCourse}(x, z)_{\text{current year}}))))\end{aligned}$$

Jape Proofs

To further explore our models framework it is important to simplify the model to its core elements. By doing so, we are not overwhelmed by the intricacies of such a large and complex system, but rather can meticulously analyze the interplay of all of its parts. Jape will serve as an environment to test and prove that our initial assumptions lead to valid conclusions. Our domain of discourse will be simplified to just a few courses and students..

NOTE: the actual jape proofs in the .j file will look different as jape restricts the use of certain characters (predicates used will just be P, Q, R). However, the sequents will still remain the exact same.

Proof 1:

P(x,y) = student x is in program y

E(x,y) = student x is enrolled in course y

M(x,y) = course x is mandatory for program y

This proof describes the following:

For all students, all programs and all courses. If all courses z in our domain are mandatory for program y, there exists a student x in program y and that student is enrolled in all courses in the domain. Then we should be able to deduce that there exists a student in any course z and that course z must be a mandatory class for program y.

$$\forall x.\forall y.\forall z.(P(x,y) \land M(z,y)), \quad \exists x.(P(x,y)), \quad \exists x.\forall z.(E(x,z)) \vdash \exists x.\forall y.\forall z.(E(x,z) \land M(z,y))$$

1	$\forall x.\forall y.\forall z.(P(x,y) \land M(z,y)), \exists x.P(x,y), \exists x.\forall z.E(x,z)$	premise
2	actual $i, \forall z.E(i,z)$	assumptions
3	$\forall y.\forall z.(P(i,y) \land M(z,y))$	∀E, 1. 1, 2. 1
4	actual $i1$	assumption
5	$\forall z.(P(i,i1) \land M(z,i1))$	∀E, 3, 4
6	actual $i2$	assumption
7	$P(i,i1) \land M(i2,i1)$	∀E, 5, 6
8	$M(i2,i1)$	∧E, 7
9	$E(i,i2)$	∀E, 2. 2, 6
10	$E(i,i2) \land M(i2,i1)$	∧I, 9, 8
11	$\forall z.(E(i,z) \land M(z,i1))$	∀I, 6–10
12	$\forall y.\forall z.(E(i,z) \land M(z,y))$	∀I, 4–11
13	$\exists x.\forall y.\forall z.(E(x,z) \land M(z,y))$	∃I, 12, 2. 1
14	$\exists x.\forall y.\forall z.(E(x,z) \land M(z,y))$	∀I, 1. 3, 2–13

Proof 2:

PR(x,y) = course x is a prerequisite for course y

D(x,y) = student x has completed course y

E(x,y) = student x is enrolled in course y

This proof describes the following:

If there exists a student z enrolled in course y and we say that if course x is a prerequisite for course y that implies the student has completed course x then we should be able to deduce that the student z is enrolled in course y which means either course x is not a prerequisite for course y or the student has already completed course x.

$$\exists x.\exists y.\exists z.(E(z,y) \land (PR(x,y) \rightarrow D(z,x))) \quad \vdash \exists x.\exists y.\exists z.(E(z,y) \land (\neg PR(x,y) \lor D(z,x)))$$

1	$\exists x.\exists y.\exists z.((E(z,y) \land (PR(x,y) \rightarrow D(z,x)))$	premise
2	actual $i, \exists y.\exists z.((E(z,y) \land (PR(i,y) \rightarrow D(z,i)))$	assumptions
3	actual $i1, \exists z.((E(z,i1) \land (PR(i,i1) \rightarrow D(z,i)))$	assumptions
4	actual $i2, (E(i2,i1) \land (PR(i,i1) \rightarrow D(i2,i)))$	assumptions
5	$PR(i,i1) \rightarrow D(i2,i)$	$\land$E, 4. 2
6	$E(i2,i1)$	$\land$E, 4. 2
7	$PR(i,i1) \lor \neg PR(i,i1)$	Theorem PR$\lor\neg$PR
8	$PR(i,i1)$	assumption
9	$D(i2,i)$	$\rightarrow$E, 5, 8
10	$\neg PR(i,i1) \lor D(i2,i)$	$\lor$I, 9
11	$\neg PR(i,i1)$	assumption
12	$\neg PR(i,i1) \lor D(i2,i)$	$\lor$I, 11
13	$\neg PR(i,i1) \lor D(i2,i)$	$\lor$E, 7, 8–10, 11–12
14	$E(i2,i1) \land (\neg PR(i,i1) \lor D(i2,i))$	$\land$I, 6, 13
15	$\exists z.(E(z,i1) \land (\neg PR(i,i1) \lor D(z,i)))$	$\exists$I, 14, 4. 1
16	$\exists z.(E(z,i1) \land (\neg PR(i,i1) \lor D(z,i)))$	$\exists$E, 3. 2, 4–15
17	$\exists y.\exists z.(E(z,y) \land (\neg PR(i,y) \lor D(z,i)))$	$\exists$I, 16, 3. 1
18	$\exists y.\exists z.(E(z,y) \land (\neg PR(i,y) \lor D(z,i)))$	$\exists$I, 2. 2, 3–17
19	$\exists x.\exists y.\exists z.(E(z,y) \land (\neg PR(x,y) \lor D(z,x)))$	$\exists$I, 18, 2. 1
20	$\exists x.\exists y.\exists z.(E(z,y) \land (\neg PR(x,y) \lor D(z,x)))$	$\exists$I, 1, 2–19

Proof 3:

E(x,y) = Student x is enrolled in course y

A(x,y) = there are seats available for student x in course y

T(x,y) = there is a time conflict between course x and course y

This proof describes the following:

If there are two classes and one student in our domain of discourse and that student is enrolled in both classes then you must be able to deduce that there are open seats in both classes for the student and there must not be a time conflict between the two classes.

$$\exists x.\exists y.\exists z.((E(x,y) \land E(x,z)) \land ((E(x,y) \land E(x,z)) \rightarrow (A(x,y) \land A(x,z) \land \neg T(y,z)))) \vdash \exists x.\exists y.\exists z.(A(x,y) \land A(x,z) \land \neg T(y,z))$$

1	$\exists x.\exists y.\exists z.((E(x,y) \land E(x,z)) \land ((E(x,y) \land E(x,z)) \rightarrow ((A(x,y) \land A(x,z)) \land \neg T(y,z)))$	premise
2	actual $i, \exists y.\exists z.((E(i,y) \land E(i,z)) \land ((E(i,y) \land E(i,z)) \rightarrow ((A(i,y) \land A(i,z)) \land \neg T(y,z)))$	assumptions
3	actual $i1, \exists z.((E(i,i1) \land E(i,z)) \land ((E(i,i1) \land E(i,z)) \rightarrow ((A(i,i1) \land A(i,z)) \land \neg T(i1,z)))$	assumptions
4	actual $i2, (E(i,i1) \land E(i,i2)) \land ((E(i,i1) \land E(i,i2)) \rightarrow ((A(i,i1) \land A(i,i2)) \land \neg T(i1,i2)))$	assumptions
5	$(E(i,i1) \land E(i,i2)) \rightarrow ((A(i,i1) \land A(i,i2)) \land \neg T(i1,i2)))$	$\land$E, 4. 2
6	$E(i,i1) \land E(i,i2)$	$\land$E, 4. 2
7	$(A(i,i1) \land A(i,i2)) \land \neg T(i1,i2)$	$\rightarrow$E, 5, 6
8	$\exists z.(A(i,i1) \land A(i,z)) \land \neg T(i1,z)$	$\exists$I, 7, 4. 1
9	$\exists z.(A(i,i1) \land A(i,z)) \land \neg T(i1,z)$	$\exists$I, 3. 2, 4–8
10	$\exists y.\exists z.(A(i,y) \land A(i,z)) \land \neg T(y,z)$	$\exists$I, 9, 3. 1
11	$\exists y.\exists z.(A(i,y) \land A(i,z)) \land \neg T(y,z)$	$\exists$I, 2. 2, 3–10
12	$\exists x.\exists y.\exists z.(A(x,y) \land A(x,z)) \land \neg T(y,z)$	$\exists$I, 11, 2. 1
13	$\exists x.\exists y.\exists z.(A(x,y) \land A(x,z)) \land \neg T(y,z)$	$\exists$I, 1, 2–12

Summary

The "Schedule Sensei", is an innovative and modern logical model designed to assist students in understanding course schedule planning. The model was built around being highly customizable while also being as simple and straightforward to use as possible. The pre-existing JSON data sets or test-cases as we call them, allow students to experiment with how different factors in course offerings, requirements and friendships can affect the models ability to generate feasible schedules.

The modular approach of our project created a system where each section could be independently developed, tested, and improved without affecting the others. Such sections include the JSON data handling, SAT solving and constraint implementation, visualization and user interactivity. Each of these separate sections also allowed us to explore various different aspect of our project in-depth. Overcoming the challenges relating to the implementation of friendships and course requirements also gave us valuable insight in working with a complex, dynamic and multi-dimensional problem space.

Our project addresses the complexity of academic course scheduling by breaking it down into smaller more manageable rules. This deconstruction allowed us to translate the broad problem into logical statements that we could implement using propositional encoding in Bauhaus. Each of the many rules that exist in our model can be grouped into distinct categories that each handle the many aspects of course scheduling, such as course prerequisites, student preferences, and friendship connections.

Despite facing significant challenges, 'Schedule Sensei' has been created as a tool that is able to offer valuable insights and practical solutions in the domain of academic course scheduling.