2020

CISC/CMPE 204: Modelling Project {22}

ACADEMIC PLANNER

ELLEN BARSS
ZIHANG CHEN
DANIIL PAVLOV
RUEBAN RASASELVAN

Table of Contents

Project Summary

Most Queen's students can agree that course registration is always a bit of a bumpy road – from figuring out what courses can actually be taken based on prerequisites to fitting all these intended courses into a manageable schedule. These pitfalls of the current system in place have been the project's inspiration; to create a model for students to plan out their courses based on their respective degree plans. When developing this project, it was crucial to explore various sets of propositions and their constraint relations to encode an algorithm that determines course requirements in graduation.

Throughout the project, various propositions were explored; a set of courses required, a set of courses already taken, a set of courses that can be taken, a list of time slots, credit requirements, and many more. As the model's scope was streamlined, not all of these propositions made it into the final algorithm. Instead, the focus of the project shifted to looking at a singular degree plan for a student pursuing a Bachelor in Fundamental Computing (refer to Appendix I) and developing a specific list of courses that are related through predicate logic to represent the hierarchical form of their academic journey in the degree path. Specifically, the model now determines what class a student can take with their degree plan and then at what pace they will finish their Bachelor. Finally, the model looks at the student's list of courses and determines whether they are eligible to graduate.

The user interface of the model walks through each hypothetical year and semester for the student. It provides a list of possible courses that can be taken, allows the student to "take" the course, and gives the flexibility to plan the consecutive academic semester. Finally, at year 4, semester 2, the student can see if they can graduate.

Propositions

Using the Fundamental Computing Degree Map, all the courses required for graduation were mapped into a full course list (see Appendix II). This list formed the basis for most propositions of the model. The following are the final propositions that are included in the course selection model:

z_i – The course list, consisting of all courses Zi required for graduation. The original intention was that Zi would evaluate to true when a course is required for a specific degree plan. Due to the narrowing of the project scope, this does not have much use in the current model. However, it will be essential to expand the model to encompass more degree plans. Further, it was used to develop the concept of graduation in the model. Examples of zi can be referred directly to the master course list in the appendix.

x_i – A subset of the course list consisting of entries, xi, which represent specific courses that have been "taken" by the student. xi evaluates to true when a course with the code *i* is already taken by the student. Some examples of xi could be x_{cs101}, x_{cs121}, x_{mt121}, etc.

y_i – A subset of the course list, consisting of courses y_i, available for students to take. The algorithm curates this list based on proposition logic that reflects conflicts regarding prerequisites or mandatory choices between courses. y_i evaluates to true when a student can take a course with code *i* at the study's current stage. Some examples of y*i* could be y_{cs101}, y_{cs110}, y_{mt121}, etc.

$w_{i,y,s}$ – Is represented by a 4 x 2 time table (4 years and 2 semesters each year). The variable evaluates to true when a course is taken at that specific time (year, semester). An example of $W_{i,y,s}$ is $W_{cs101,1,1}$ to

represent CISC101 is taken in the first semester of the first year. *Note*: in the proposition number, year and semester are directly represented in y and s. However, in application code, year and semester are represented as y-1, and s-1 compared to the proposition. Ex. $w_{cs101,0,0}$ for the first semester of the first year.

g – A Boolean statement that evaluates to true or false. To evaluate to true, the student must have met all the requirements to graduate from the degree plan, meaning that they must complete all the required courses.

Constraints

There are three main categories of constraints for the academic planner algorithm: when students must choose one course or another, when students must complete the prerequisites of a course before they are eligible to take the course, and constraints to map a path to graduation.

In the first class, constraints for when students must choose one course or another, is modelled directly from the program map. It is common in the school of computing for students to choose between mandatory prerequisite classes, especially for math, statistics, and introductory coding classes. For example, students must choose between taking CISC 101 or CISC 110. The student must choose and take only one course of the pair. This relation is modelled propositionally as:

$$x_{cs101} \oplus x_{cs110} \text{ which is equivalent to } (x_{cs110} \wedge \neg x_{cs101}) \vee (\neg x_{cs110} \wedge x_{cs101})$$

Encoding this constraint directly into the algorithm is a rather long process, so to save time and improve efficiency, a xor function was created directly in the code to represent this constraint. Refer to Appendix III for a screenshot of the function in the run.py file. Note that this constraint was also proved in Jape and will be discussed later in this report.

The second category, constraints that determine if a student can take a course if the prerequisites have been taken, is also modelled directly from the program map. The algorithm for this constraint directly represents the hierarchical relationship between courses and their prerequisites. Students can take a course if they finished all prerequisites, which determines that x_i is materially equivalent to y_i. In this algorithm, once students complete are prerequisites following courses will automatically become available; therefore, when y_i is true x_i has to be true, and vice versa. For example, CISC 221 cannot be completed if and only if its prerequisite, CISC 124 is completed first. This can be modelled propositionally as:

$$(\neg x_{cs124} \vee y_{cs221}) \wedge (x_{cs124} \vee \neg y_{cs221})$$

Again, encoding this constraint multiple times into the program proved tedious, so an iff function was created to represent the material implication relationship (Appendix IV). This iff relationship is also proved in Jape later on in this report.

The final category is constraints that model the path to graduation. Given the program map of a Fundamental Computing degree, there are several ways that a student can reach graduation. This variance of possible course combinations is handled through constraints that are a mix between material implication and the previous xor function. For example, a student can take CISC 101 or CISC

110, both of which (with the right combination of additional courses) will lead to graduation equaling True. This relationship can be propositionally modelled as:

$$y_{cs101} \vee y_{cs110}$$

$$(\neg y_{cs101} \vee y_{cs110}), (y_{cs110} \vee \neg y_{cs101}) \wedge (\neg y_{cs110} \vee y_{cs101})$$

In the code, this constraint uses some simple 'or' statements and the xor function represents the logic. For degree planning purpose, even though courses can be taken multiple times at university, the degree planner would not allow that to happen since repeated courses taken are not recommended. Therefore, courses are only taken once for the planning.

$$w_{cs101,1,1} \oplus w_{cs101,1,2} \oplus w_{cs101,2,1} \oplus w_{cs101,2,2} \oplus w_{cs101,3,1} \oplus w_{cs101,3,2} \oplus w_{cs101,4,1} \oplus w_{cs101,4,2}$$

Jape Proofs

As previously mentioned, Jape was used to prove the logic of xor and iff functions used in the model. Additionally, Jape was also used to prove the relationship between completing courses and graduation. The final three theorems used in the model were:

1. P, (P∨¬Q)∧(¬P∨Q)⊢Q
2. P, (P∧¬Q)∨(¬P∧Q)⊢¬Q
3. ∀x.P(x), ∀x.Q(x) ⊢∀x.((Q(x)→P(x)) ∧ (P(x)→Q(x)))

The first proof represents the iff function's logic, where P could be the course that is intended to be taken (a y_i proposition), and Q is a course that has been taken already (a x_i proposition). P or not Q must be true, and not P or Q must be true. This relationship represents that P cannot be taken unless Q has already been taken.

The second sequent represents the xor function, an exclusive disjunct, where P represents the first option of a course that has to be taken (such as x_{cs101}) and Q represents the second option of a course (such as x_{cs110}). P and not Q or not P and Q must be true, meaning that either only P or Q can be taken, but not both. Therefore, if P is true (and thus taken), then not Q must be true.

The third Jape sequent reflects the logic of the number of courses taken and the number of courses needed for graduation. P can represent the set of total courses taken by a student, and Q can represent the set of complete courses required for a Bachelor of Computing. Given both sets, all Q must imply P, and P must imply Q. This means that both sets must be equal for graduation.

Please refer to Appendix V for a screenshot of the final theorems and their respective proofs.

Model Exploration

Model 1: Proposal model

Initial Summary

We are creating a modelling problem to help students, or the university determine a student's schedule and degree plan. We plan to create an algorithm that checks if a student can or cannot take a specified course (or does not qualify for graduation in a specific plan). This problem is modelled around the system that Queen's currently uses with solus - an academic planner.

At this stage, we had not started a python implementation yet. We planned to have a student select their degree plan then build a schedule based on that plan's requirements. The model would accept a student's course selection for every semester then return whether or not graduation is satisfied. Within each semester, the model would check whether or not the courses conflicted with each other in their time slots and whether the required prerequisites and exclusions were met for each selection. After eight semesters, the model would check whether the core courses and credit requirement were met.

Propositions

Our propositions: {degree_plan, time_conflict, credit_requirement, degree_requirement, course_taken, course_to_take}

d_i: *Degree_plan* - T or F depending on program student is in - T if the plan is Bachelor of Computer Science (Honours)

- Where *i* is different degree plans (ex. Cognitive Science, Game Development, Computing, etc.)

t_k: *Time_conflict* - T or F - F there is a selected course take place in the time slot

- Where *k* is the time slot it occupies

c_i: *Credit_requirement* - T or F - T if the student has enough credits (ex. Credits == 120)

- Where *i* refers to the credit requirement of specific degree plans (i.e. c1 refers to credit requirement of d1)

r_i: *Degree_requirement* - T or F - T if the course goes towards a student degree plan

- Where *i* refers to degree plans.
- $(X_j \wedge ri)$

x*j*: Course_taken - T if the course has been taken by the student.

- Where j is a specific course taken (ex. 203, 204, 220, 221)

y_j: Course_to_take - T if a student can take this course. F if a student did meet the requirements of the course.

Constraints

Courses cannot be taken during the same timeslot.

$$(y_{cisc203} \wedge t_1) \oplus (y_{cisc204} \wedge t_1)$$

$$(x_{cisc203} \curlywedge t_1) \wedge (x_{cisc204} \curlywedge t_1) \rightarrow \bot$$

Courses with an unfulfilled prerequisite cannot be taken.

$$x_{cisc121} \rightarrow \neg y_{cisc124}$$

Must choose one course or another – exclusions.

$$(x_{math110} \oplus (x_{math111} \wedge x_{cisc102}))$$

Courses with corequisites must be taken in the same semester.

$$y_{cisc121} \wedge y_{cisc102}$$

Prerequisite - students must complete prerequisites to take a course.

$$x_{cisc101} \wedge x_{cisc121} \leftrightarrow y_{cisc124}$$

Did students take all required courses (Graduation)?

$$(x_{cisc203} \wedge x_{cisc204} \wedge x_{cisc221} \wedge ....) \wedge c_{120} \rightarrow d_{bach\_comp}$$

Model 1 Takeaways

Although this stage of the project did not deep dive into developing a complete list of constraints and encoding them, it was crucial to fully understanding the problem. The group sat down and discussed their experiences with course registrations, what aspects they thought were the most important to a functional system, and ways that it can improved. At this point they were able to explore different types of Bachelor Program's offered by Queen's and how the required courses interacted with each other. Doing this allowed the group to develop a rather large list of propositions and constraints that proved to be useful to refer to at later stages.

Model 2: Draft model

Based on feedback from the professor, we decided to simplify our model for the draft. Instead of having a separate variable for a degree plan, we decided on a specific plan to focus on. Based on this, we removed the following propositions: "Degree Plan", "Time Conflict", "Credit Requirement", "Course Requested", then added, "Courses that can be taken". This version of the model reduced our propositions down to three: the courses that were required, the courses that have already been taken, and the courses that can be taken in the future. These propositions were then used to implement exclusions and prerequisites courses as constraints on our algorithm. At this point, we had not yet looked at the first-order extension of the model.

At this stage, we also began to develop the coded version of our model. In terms of our algorithm's python implementation, we had this version encoded more in line with the suggested variable encoding from 204. We hardcoded the encoding of x, y, and z propositions for every single course to calculate each variable's satisfiability and relationship (refer to Appendix VI). The variable's relationship was then represented by encoded constraints (Appendix VII). At this stage, our model was still dependent on the specific courses in our selected degree plan as opposed to being quickly replaced. While this made the code shorter, it did not allow for scalability when more degree plans would be added. Furthermore,

encoding the constraints required a significant amount of time to write the if and only relationships. We implemented the whole algorithm within "run.py".

Propositions

Our propositions: {courses_taken, course_to_take, courses_required}

> y_i: Course required - Where i is a specific course required for graduation (ex. $y_{cisc101}$, $y_{cisc110}$, $y_{cisc121}$..., $y_{math121}$)
>
> x_i: Course taken - Where i is a specific course (ex. $x_{cisc101}$, $x_{cisc110}$, $x_{cisc121}$..., $x_{math121}$)
>
> z_i: Course can be taken – Where i is a specific course (ex. $z_{cisc101}$, $z_{cisc110}$, $z_{cisc121}$..., $z_{math121}$)

Constraints

Courses with an unfulfilled prerequisite cannot be taken.

$$x_{cisc101} \not\wedge x_{cisc121} \longleftrightarrow y_{cisc124}$$

Prerequisite - Students must complete prerequisite to take a course.

$$(x_{math110} \oplus (x_{math111} \wedge x_{cisc102})) \wedge x_{cisc121} \rightarrow z_{cisc204}$$

$$(x_{cisc110} \oplus x_{cisc101}) \wedge x_{cisc121} \rightarrow z_{cisc121}$$

$$x_{cisc101} \wedge x_{cisc121} \rightarrow z_{cisc121}$$

Etc. for each course

Students must choose one course or another.

$$x_{math110} \oplus (x_{math111} \wedge x_{cisc102})$$

$$x_{cisc110} \oplus x_{cisc101}$$

$$x_{cisc332} \oplus x_{cisc326}$$

Model 2 Takeaways

After gaining a better understanding the model from the last stage, we began to get more familiar with translating these ideas into python. However, we also realized the detail and commitment that came with this, the course list contains more than 20 entries that will need to be encoded multiple times in accordance with each proposition. Further, these variables will all then need to be connected through constraints. At this stage in the model, we realized that we would need to create more functions to support the implementation of the constraints and streamline their operation with each other.

Model 3: Final model

Based on our TA and peer feedback, we realized that our simplification of the model went too far, so we reintroduced some constraints. We started developing the final model by encoding every parameter and constraints for each course into an organized data set. This allowed us to store and calculate the relationship and satisfiability of each course. In the draft model, we had hardcoded most of our

functionality, but in this version, we developed sets and functions in order to combine multiple propositions.

Grouping these constraints allowed us to implement the different steps in course selection. It allowed us to determine what courses would be available in specific years, include different classes within the same timetable, and determine how long it would take a student to complete all their required courses to satisfy graduation.

As mentioned previously, our implementation in the draft design was not flexible enough to work with other degree programs. In this version, we have increased the use of general variable representations. These allowed us to re-use the same variables for many uses besides accommodating many sets of data. The way we store propositions now also keeps track of what courses have already been taken and reassesses courses available in upcoming semesters. Through this, we can keep track of the students' progress toward graduation.

By creating replaceable data sets, we open up the algorithm to be used with more degree plans in the future without modifying the algorithm. We would be able to simply replace the list of courses and prerequisites unique to each degree. We could also add different sets of degree plans and graduation requirements to develop degree planners' current concept.

One notable addition to the model at this stage, was the re-introduction of time constraints. Throughout the development of the Academic Planner, the group's goal was to model a student's ability to take courses in conjunction with time slots and scheduling conflicts. Unfortunately, implementing this with the other constraints proved to be too complicated. However, this functionality was considered crucial to the scope of the problem, so we developed a simplified structure to implements. For the final model, scheduling conflicts were limited to a specific set of rules:

- A maximum of five courses were allowed each semester (totalling ten for a year of study)
- The same courses cannot be taken twice in a semester.

These rules supported the creation of a user interface for the model and added a degree of complexity. The user interface and main testing function went through several iterations. Originally running the program just showed a list of possible courses that could be taken, where the user would have to select the corresponding index (Appendix VIII). The output was formatted to be easier for the user to add courses to their “basket” and then would build a schedule accordingly. Finally the program will output whether the schedule is feasible with course constraints, and if the student can graduate within four years (Appendix IX).

In summary, this final model added functionalities that were not available in our draft model while simplifying the code to deal with the repetitive nature of the model's goal. The whole model was implemented within "run.py," but the plan was to have separate files for different degree plans. These would contain a list of prerequisites, exclusions, timeslots, and course requirements.

Final Model Takeaways

At this stage in the project, the model really began to come together. We first realized the most crucial propositions and constraints of the model; lists of courses taken, courses that can be taken, and courses that must be taken for graduation. The idea of timeslot conflict was re-introduced for the functionality of the program and the relationship that determines graduation was finally encoded.

Additionally, we realized the importance of creating a testing program in the run.py. This began the process of developing a user interface that would be an essential reflection of the model's logic and would allow for testing it.

First-Order Extension

As the project is based on course selection, the project's universe of discourse can easily be represented as the course set that the university provided to students (or limit it to smaller lists for different purposes such as faculty specific requirements). Every single course can have four different properties for a student:

1. A course the student has already taken
2. A course student can take now,
3. A course that the student cannot take (due to prerequisites not met or they do not have access to those courses)
4. A course student must take for graduate.

The courses that cannot be taken by the student will not be taken into consideration while doing degree planning. Therefore, there are only three predicates to look at. During the progress of a student studying at university, the student will begin to satisfy prerequisites and, in the end, must be able to satisfy the graduation requirement to be able to graduate.

Universe of Discourse:

The universe of discourse should be the whole course list provided by the university.

A: set all courses are provided by the university. Using code i to determine courses.

Predicates:

There are four predicates to define course status.

X_i: Courses already taken by the student. Course i is in set X if the course is taken.

Y_i: Course available for students to take. Course i is in the set Y if the course is available for the current/future semester.

Z_i: A set of courses for a degree plan. Also can have multiple different degree plans with a different set of courses.

$W_{y,s(i)}$: Multiple set, y stand for year and s stand for semester. Course i is in the set $W_{y,s}$ whenever the course is taken at (y, s) time slot of the 4-year undergraduate study.

Constraints:

Formulas can be satisfied in the model.

Prerequisites: Every single course has its prerequisites (can have 0 prerequisite). Whenever a course's prerequisite is satisfied, the course can be taken for current or future semesters. A course can become available if and only prerequisites are satisfied(Y(i) cannot map to true if the corresponding X(i)s are not true).

Ex. X(cs101)⊕X(cs110) ↔ Y(cs121), X(cs121) ↔ Y(cs124)

Course Conflict: Some courses will not take with some other courses, even if prerequisites are satisfied. For a course that conflicts with some other courses already taken, this course's prerequisite cannot be satisfied. Moreover, two-course conflict with each other can't be both taken.

Ex. (X(cs101)⊕X(cs110)) ∧ (X(cs101)⊕Y(cs110)) ∧ (Y(cs101)⊕X(cs110))

Graduation: Graduation is a formula that should be satisfiable in any given Z(i) set(Graduation requirement for a degree plan). The graduation formula changed with different degree plan sets. Every single course in the graduate requirement should be taken, except for courses conflict with other courses; only one needs to be taken to satisfy the requirement.

Ex. (Z(cs101)⊕Z(cs110) → X(cs101)⊕X(cs110)) ∧ (Z(cs121) → X(cs121)) ∧ …

Course Retake: Even though retake a course is an option for study; however, to the degree planning propose, this will not be allowed in the model. Therefore, every course student can only take once; then, there will not be allowed to have a single course i in multiple different sets.

Ex. W1,1(cs110) ⊕ W1,2(cs110) ⊕ W2,1(cs110) ⊕ W2,2 (cs110) ⊕ …

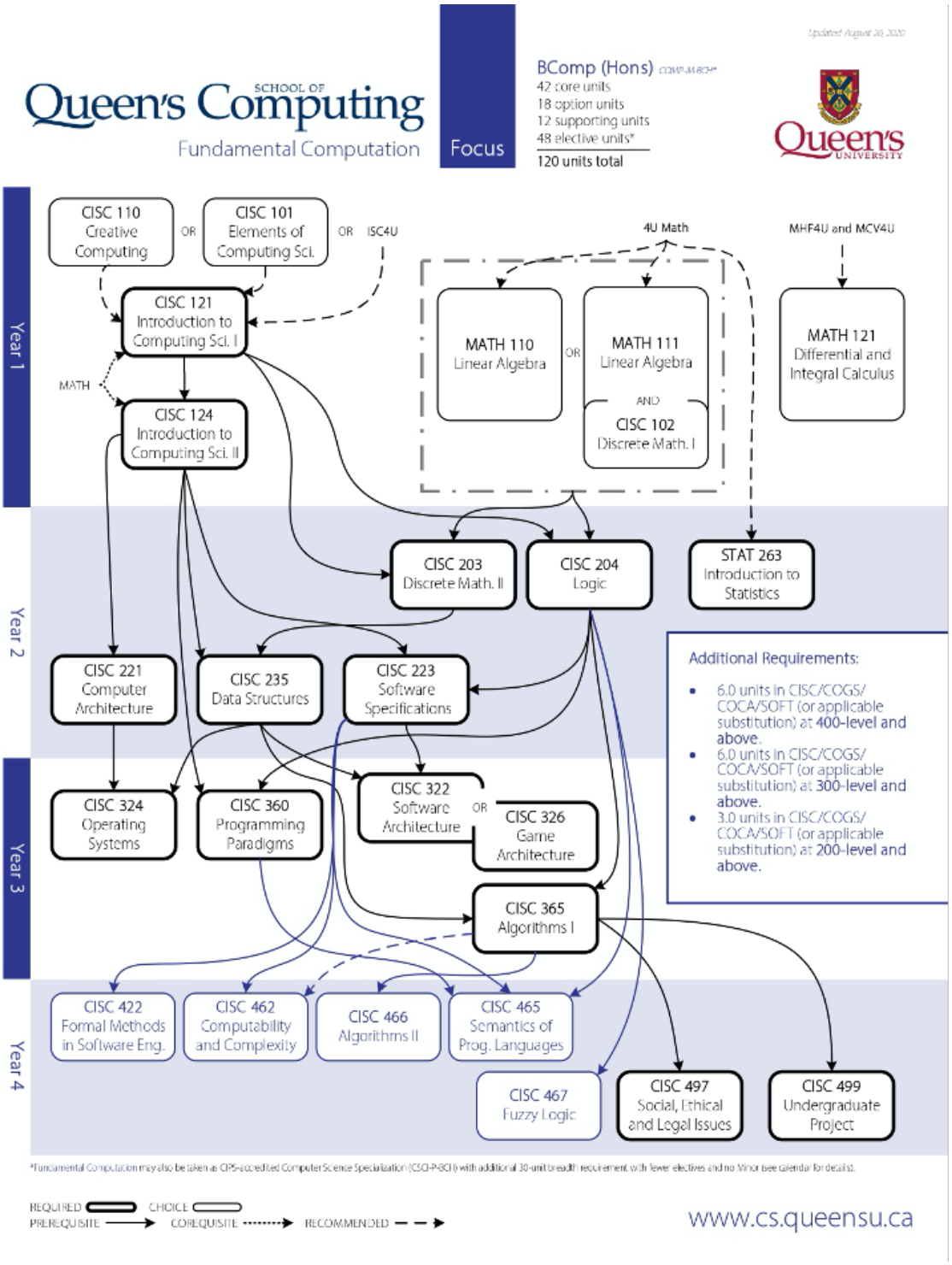
Note: There could be predicate defined if a course is elective or not, but it was not included as it was not the primary scope of this project.

Conclusion

Overall, through much trial and error, our group has been successful in creating a model that reflects the hierarchy of student completing a degree at Queen's. It considers the propositions crucial to success, namely the set of lists of courses that result from courses the student has taken, is eligible to take, and must take to graduate. The relations of these lists are encoded into the final model through various constraints. Finally, the model is presented in an interface that does all the work for a student: displays a list of possible courses for a student, allows them to select courses for the semester, sets a maximum number of courses allowable per semesters and then repeats this process for each year and semester. Finally the program puts these choices into a schedule based on satisfiability and determines graduation.

Appendices

Appendix I - Fundamental Computing Program Map



Appendix II - Course Master List

Course list =

{"cs101", "cs110", "cs121", "cs102", "cs124", "cs203", "cs204", "cs221", "cs223", "cs235", "cs324", "cs360", "cs322", "cs326", "cs365", "cs422", "cs462", "cs466", "cs465", "cs467", "cs497", "cs499", "st263", "mt110", "mt111", "mt121"}

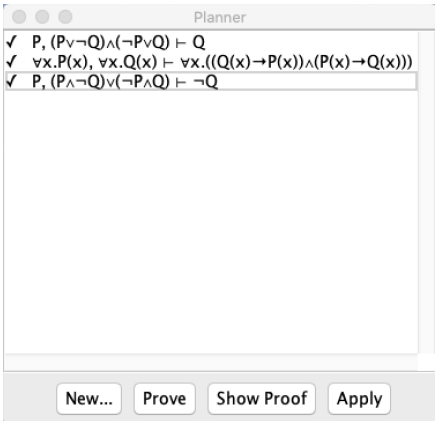
Appendix III - xor function, for exclusive disjunct in run.py

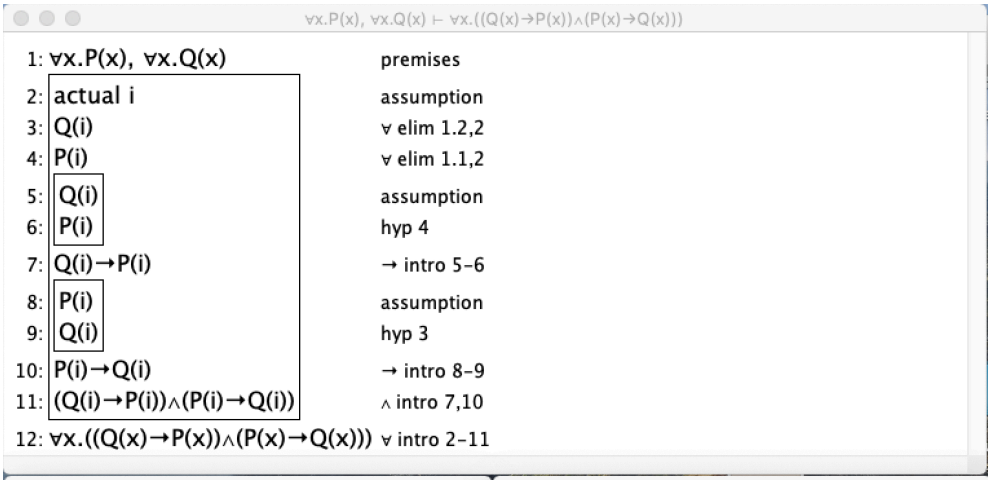
```
def xor(left, right):
    return((left.negate() & right) | (left & right.negate()))
```

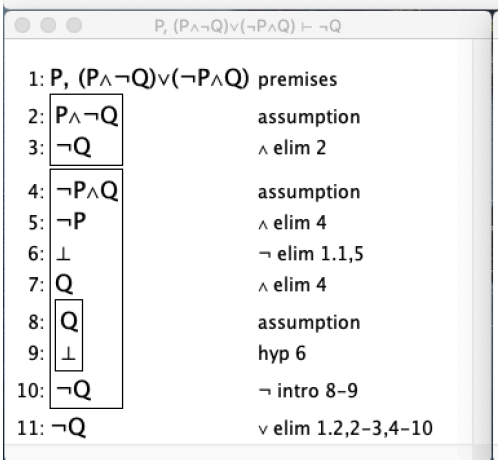
Appendix IV - iff function, for material equivalence in run.py

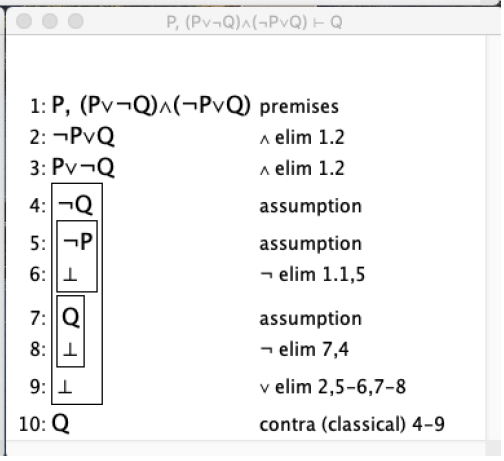
```
def iff(left, right):
    return(left.negate() | right) & (right.negate() | left)
```

Appendix V - Jape Theorems and Proofs









Appendix VI – Draft run.py Variable Encoding

```
# y variable courses are courses required for the degree
ycs101 = Var('ycs101')
ycs110 = Var('ycs110')
ycs121 = Var('ycs121')
ycs102 = Var('ycs102')
ycs124 = Var('ycs124')
ycs203 = Var('ycs203')
ycs204 = Var('ycs204')
ycs221 = Var('ycs221')
ycs223 = Var('ycs223')
ycs324 = Var('ycs324')
ycs235 = Var('ycs235')
ycs360 = Var('ycs360')
ycs322 = Var('ycs322')
ycs326 = Var('ycs326')
ycs365 = Var('ycs365')
ycs422 = Var('ycs422')
ycs462 = Var('ycs462')
ycs466 = Var('ycs466')
ycs465 = Var('ycs465')
ycs467 = Var('ycs467')
ycs497 = Var('ycs497')
ycs499 = Var('ycs499')
yst263 = Var('yst263')
ymt110 = Var('ymt110')
ymt111 = Var('ymt111')
ymt121 = Var('ymt121')

# x variable courses are courses taken already
xcs101 = Var('xcs101')
xcs110 = Var('xcs110')
xcs121 = Var('xcs121')
xcs102 = Var('xcs102')
xcs124 = Var('xcs124')
xcs203 = Var('xcs203')
xcs204 = Var('xcs204')
xcs221 = Var('xcs221')
xcs223 = Var('xcs223')
xcs324 = Var('xcs324')
xcs235 = Var('xcs235')
xcs360 = Var('xcs360')
xcs322 = Var('xcs322')
xcs326 = Var('xcs326')
xcs365 = Var('xcs365')
xcs422 = Var('xcs422')
xcs462 = Var('xcs462')
xcs466 = Var('xcs466')
xcs465 = Var('xcs465')
xcs467 = Var('xcs467')
xcs497 = Var('xcs497')
xcs499 = Var('xcs499')
xst263 = Var('xst263')
xmt110 = Var('xmt110')
xmt111 = Var('xmt111')
xmt121 = Var('xmt121')
```

Appendix VII – Draft run.py Constraint Encoding

```
def example_theory():
    E = Encoding()
    # Courses choices
    E.add_constraint((xmt110 & (xmt111 & xcs102).negate) | (~xmt110 &(xmt111 & xcs102)))
    E.add_constraint((xcs110 & ~xcs101) | (~xcs110 & xcs101))
    E.add_constraint((xcs322 & ~xcs326) | (~xcs322 & xcs326))

    # Prerequisite constraints
    E.add_constraint((xcs110 | xcs101).negate() | zcs121)
    E.add_constraint((((xcs110 & ~xcs101) | (~xcs110 & xcs101)) & xcs121).negate() | zcs124)
    E.add_constraint((((xmt110 & (xmt111 & xcs102).negate) | (~xmt110 &(xmt111 & xcs102))) & xcs121).negate() | zcs204)
    # More to come
```

Appendix VIII – Initial User Interface for run.py

```
Is guaduation satisfiable?:  True
{'xcs462': True, 'xcs124': True, 'xcs465': True, 'xcs360': True, 'xcs324': True, 'xcs204': True, 'xcs497': True, 'xcs46
6': True, 'xcs101': True, 'xcs110': False, 'xcs221': True, 'xcs365': True, 'xcs223': True, 'xcs326': True, 'xcs322': Fa
lse, 'xmt121': True, 'xcs203': True, 'xcs102': True, 'xmt111': True, 'xmt110': False, 'xcs467': True, 'xst263': True, '
xcs121': True, 'xcs422': True, 'xcs499': True, 'xcs235': True}
Year 1 Semester 1
Course Currently Avaliable:  ['cs101', 'cs110', 'cs102', 'st263', 'mt110', 'mt111', 'mt121']
Input array index to put a course into course cart:
Or input x to end current semester's planning.
0
['cs101']
Year 1 Semester 1
Course Currently Avaliable:  ['cs101', 'cs102', 'st263', 'mt110', 'mt111', 'mt121']
Input array index to put a course into course cart:
Or input x to end current semester's planning.
1
['cs101', 'cs102']
Year 1 Semester 1
Course Currently Avaliable:  ['cs101', 'cs102', 'st263', 'mt111', 'mt121']
Input array index to put a course into course cart:
Or input x to end current semester's planning.
3
['cs101', 'cs102', 'mt111']
```

Appendix IX – Final User Interface for run.py

Selecting Courses

```
Year 1 Semester 1
Courses Currently Avaliable:
Index    Course Code
  0         cs101
  1         cs110
  2         cs102
  3         st263
  4         mt110
  5         mt111
  6         mt121
Input array index to put a course into course cart:
Or input x to end current semester's planning.
0
Courses this semester:  ['cs101']


Year 1 Semester 1
Courses Currently Avaliable:
Index    Course Code
  1         cs102
  2         st263
  3         mt110
  4         mt111
  5         mt121
Input array index to put a course into course cart:
Or input x to end current semester's planning.
1
Courses this semester:  ['cs101', 'cs102']


Year 1 Semester 1
Courses Currently Avaliable:
Index    Course Code
  2         st263
  3         mt111
  4         mt121
Input array index to put a course into course cart:
Or input x to end current semester's planning.
3
Courses this semester:  ['cs101', 'cs102', 'mt111']
```

Schedule building, and graduation determination

```
Student failed to graduate in time with current's schedule.


Failed course schedule:
Year 1 Semester 1:
['cs101', 'cs102', 'mt111', 'mt121', 'st263']
Year 1 Semester 2:
['cs121']
Year 2 Semester 1:
['cs124', 'cs203', 'cs204']
Year 2 Semester 2:
['cs221', 'cs223']
Year 3 Semester 1:
[]
Year 3 Semester 2:
[]
Year 4 Semester 1:
[]
Year 4 Semester 2:
[]
Unable to create time table.
```