Optimizing Scheduling of UW Undergraduate Classes

Caroline Cannistra, Ye Cao, Cole Chamberlin, Li-An Chu

3 February 2017

Introduction

As college students at a large university, we understand the challenges that come with finding a class schedule for the new quarter. There are many considerations when designing a schedule. Probably most importantly, students need to pick classes that will help them progress toward degree completion. But that still leaves plenty of room for variation in schedules. A second factor on such a large campus is travel time between classrooms. Passing period is 10 minutes for most classes, and as most UW students know, it's no fun to be sprinting across The Quad to make it to your next class on time.

The goal of our model is to generate class schedules based on common criteria to minimize the total travel distance on campus and avoid those hasty jaunts through Red Square. Using real data from the University of Washington, we believe that we can provide valuable information to students and model a scenario that we're all too familiar with. Our model relies on the Linear Programming (LP) technique of formulating the problem into decision variables and an objective function. The decision variables form constraints on the objective function, and the function is maximized or minimized within these constraints. In the context of our problem, the decision variables correspond to classes, number of credits, and distances between them. These model the criteria that a student would typically use to determine his or her schedule.

Background

We decided to model a scheduling problem based on our mutual experiences as college students. Considering our familiarity with the data, UW courses were a convenient subject for our project. We started formulating our initial project ideas: build up a student course schedule that will maximize desirable factors and minimize adverse ones. After much deliberation, we decided to maximize total travel distances between classes because as UW students, we spend a lot of time walking to classes every day. Buildings on campus are spread out and some buildings are too far away from each other to walk between in a 10 minute passing period. Minimizing travel distances between classes has immediate value to our daily school life. In addition, since many freshman students are still unfamiliar with building locations and routes around the UW campus, this model will provide a good resource to help them pick a feasible schedule.

Doing some background research, we found similarities between our problem and the traveling salesman and class scheduling problems. A traveling salesman problem (TSP) seeks out the shortest possible route between a list of cities with pairwise distances, such that each city is visited exactly once and the salesman ends up in the city that they started in. It was first expressed as an integer linear program and solved by the cutting plane method by George Dantzig, Delbert Ray Fulkerson and Selmer M.Johnson from the RAND Corporation in the 1950s and 1960s (E.L.Lawler et al., 1985). The TSP and our problem share the similarity of finding the shortest path between a list of locations. Our problem's

objective function resembles that of the TSP:

$$\sum C_{ij}X_{ij}$$

where $X_{ij} = 1$ if the path goes from city i to city j, $X_{ij} = 0$ otherwise

and C_{ij} is the distance from city i to city j

However, in our problem, we don't need to visit all the destinations and we don't need to return to the original city. We only need to visit a subset of the destinations that minimize the total distances and also meet our other constraints. This variation introduces the challenge of deciding which distances are chosen when calculating the total distances.

The class scheduling problem, especially student class scheduling is another reference point for our project. The class scheduling problem is a common topic and has been modeled and solved by many mathematician and scholars, such as in the paper *An Optimization Model for Scheduling Classes in a Business School Department* by Robert Saltzman (Saltzman, R.M., 2009), and *Ohio University's College of Business uses an integer-programming to Schedule Classes* by Martin C.H. (Martin, C.H., 2004). These papers use integer programming to solve the problem of assigning classrooms to buildings. However, student class scheduling is a different challenge that isn't addressed. This challenge was first introduced and solved by Winch, Janice K. and Jack Yurkiewicz in a student class scheduling problem with linear programming in 2013 (Janice K et al., 2013). The authors collect a list of courses and determine a course schedule to maximize the class rating. It is similar to our problem in the type of constraints required. For example, our model disallows taking more than one

class during a given time slot or during any set of overlapping time slots. However, instead of maximizing the total rating of classes, our model minimizes the total distances traveled between classes.

The Model

Problem Statement

The goal of the project was to generate a timetable by choosing from a specific subset of 100 and 200 level classes designed for freshman and minimizing the total travel distances between them. The timetable only considered Monday classes and the number of classes was fixed at 3.

Time Slot	**Time**
1	8:30 - 9:20
2	9:30 - 10:20
3	10:30 - 11:20

We only chose one hour lectures that can fill in one of these time blocks on Monday. We ignored scheduling of any associated quiz and lab sections. In total, there were 526 courses to choose from. A truncated table of all courses is listed in the appendix under H.

Since each class corresponds to a building, If one class takes place after another, then a distance exists between them. For example, if a student takes PHYS 115 in Kane Hall in the first time block and takes FRENCH 101 in Thomson Hall in the second time block, then we need to walk from Kane Hall to Thomson Hall. We used total walking time measured

in minutes to represents distances. The walking time data came from the UW Map site (University of Washington). In our example, we need 4 minutes to walk from Kane hall to Thomson hall. If we then choose MATH 126 in Electrical Engineering Building in the third time block, there will be a 5 minute walk from Thomson Hall to Electrical Engineering Building, and the total travel time of the day is 9 minutes. In addition, the combination classes should meet some other constraints: Total credits of the classes should between 12 credits and 18 credits. Take no more than one class at a time, which means either taking 0 class or 1 class in each time block. If a class has different sections, take only one of them.

The resulting model solves the problem of choosing exactly three classes while minimizing the sum of the distances from class one to two and from two to three.

Mathematical Model

We formulated the model for one day and taking exactly 3 classes. Classes were chosen from 3 time slots: 8:30-9:20, 9:30-10:20, 10:30-11:20. Suppose there are m classes in each time slots.

1. Decision Variables

   - $X_{i,j}$ is a binary variable where $X_{i,j} = 1$ if and only if class j in time slot i is taken. $i = 1, 2, ...k$. $k = 3$ in this case, $j = 1, ..., m$ If class j has more than 1 section, we will have X_{i,j_n} where j_n denotes the nth section of class j.
   - $V_{i,j}$ denotes the credits associated with class j in time slot i, $V_{ij} = \{1, 2, 3, 4, 5\}$.
   - $Y_{a,b,i}$ is a binary variable denotes whether we need to travel between class a and class b, with $Y_{a,b,i} = 1$, if and only if we need to travel between class a and b

where class a is at time slot i. $Y_{a,b,i}$ exists if and only if building of $a \neq$ building of b.

- Let $D_{a,b}$ denotes the walking time (minutes) between class a and b, $a \neq b$.

2. Objective Function The objective function minimize total traveled distance between classes in a day.

$$\min \sum_{i=1}^{k-1} \sum_{a,b}^{n} Y_{a,b,i} D_{a,b}$$

3. Constraints

- Constraint (1): sum all the class variable for each specific time slot i so that in each time slot can take exactly one class.

$$\sum_{j=1}^{n} X_{i,j} = 1 \qquad \forall i \in \text{time slot}$$

- Constraint (2): Can only take one section or zero section from each course. For example, student cannot take both MATH 124A and MATH 124B.

$$\sum_{i=1}^{k} \sum_{m=1}^{n} X_{i,j_m} \leq 1$$

- Constraint (3): The total credit for the chosen class have to be less than or equal to 18 and greater or equal to 12, which this constraint satisfies the credit requirement for each quarter in University of Washington.

$$\sum_{i=1}^{k} \sum_{j=1}^{n} V_{i,j} X_{i,j} \leq 18$$

$$\sum_{i=1}^{k}\sum_{j=1}^{n} V_{i,j}X_{i,j} \geq 12$$

- Constraint (4): Set a constraint on the travel time variable. $Y_{a,b,i} = 1$ if and only if $X_{i,a} = 1$ and $X_{i+1,b} = 1$. Meaning that the travel time between class a, b for i is set to 1 if and only if class a is chosen at time slot i and class b is also chosen at time slot $i + 1$.

$$\left.\begin{aligned} Y_{a,b,i} &\leq X_{i,a} \\ Y_{a,b,i} &\leq X_{i+1,b} \\ Y_{a,b,i} &\geq X_{i,a} + X_{i+1,b} - 1 \end{aligned}\right\} \forall i \in \text{first } (k-1) \text{ time slot}$$

- Constraint (5): The variable $Y_{a,b,i}$ exists if and only if class a and class b are in different building for all first $(k - 1)$ time slot i. Otherwise we set a constraint that $X_{i,a}$ and $X_{i+1,b}$ cannot both be taken, because we want to have at least some walking time between classes. So if class a and class b are in the same building for time slot i and $i + 1$ we set the following constraints:

$$X_{i,a} + X_{i+1,b} \leq 1$$

Text file of objective function, all constraints and decision variables is attached in appendix *I*.

Solution & Techniques

The first step in formulating our model was sourcing the data. The University of Washington website contained all the information we needed about building locations and class schedules. Unfortunately, they don't make it easily accessible for data ingestion. The building names were scraped from the Guide to Classroom Locations page which contained a list of every building on campus. These were collected into a csv with each row containing building name and its three letter abbreviation. We manually filtered through this list to eliminate buildings without classrooms, such as libraries, administration, and dormitories. A truncated table of this data is visible in Appendix F. The code used to generate the data is in Appendix A.

Getting the distance between each pair of buildings was not trivial. We needed to build an adjacency matrix of travel distance between every possible pairing of buildings, which meant finding $n(n-1)/2$ distances. Our first idea was to feed each pair of building locations into a Google Maps API which would provide walking directions. This proved troublesome, as the API didn't pinpoint the buildings as accurately as necessary. Many of the query results just found the Seattle campus and didn't drill down to the buildings themselves. Seeking alternative sources for the data, we came across UW's own campus map service (University of Washington). This service is provided for students and visitors to find buildings on campus, or navigate them between buildings, the exact task that we needed to accomplish. But the task of entering two building names and reading back the travel time was very manual. We found a Python library called Selenium that allowed for browser automation, by interacting with the HTML of a website. Creating a script, we were able to feed all $n(n-1)/2$ pairs of buildings into the service and output the adjacency matrix of distances. We captured

physical distance in miles, as well as pedestrian travel time in minutes. This introduced some problems. Some of the building codes found from the previous source didn't directly correspond to the codes used for the map service. For example, the Electrical Engineering Building is abbreviated as EEB in our first source, but EE1 on the map. After working out the inconsistencies, we were able to collect the building data. A truncated table of this data is visible in Appendix G. The code used to generate the data is in Appendix B.

The final step of data collection was collecting class information. The constraints of our problem relied on number of credits, class location, start and end time, and class sections. This data was readily available from the UW time schedule (University of Washington, 2016). It was formatted in tables, but the web designer made poor use of the table row structure, so extracting the correct attributes required some regular expression splitting and joining. We chose to use the Winter Quarter time schedule for our purposes, but the model and data collection would be identical for any other quarter. The initial pass on the website collected 6000 courses from all the departments on campus. We excluded any classes that were not scheduled or were missing any one of the attributes listed. Then we paired down the data for our model limitations. All classes with a 300 or higher course number were eliminated. This was to mitigate the chance of a schedule being picked with classes that had large numbers of pre-requisites. This model is designed with the college freshman in mind, after all. A truncated list of this data is visible in Appendix H. The code used to generate the data is available in Appendix C.

We then paired down the list even further by eliminating quiz and lab sections. We also considered only classes that had Monday sections and ignored conflicts on other days of the week. These simplifications begin to limit the usefulness of the model since an actual scheduling tool would need to take those constraints into consideration, but for the sake of simplicity, it was necessary. The pared down model left around 500 classes to choose from, which we binned into their respective time blocks. All classes starting at 8:30 and ending at 9:20 were put into one block and so on. The original model is restricted to picking a single class from each of three time blocks and calculating the distance between adjacent pairs.

After collecting the finalized data set into a csv file, we wrote a Java program to output the input text file that was used as the input for LPSolve. The entire Java program consists of two files **Course.java** and **ProjectMain.java**. Both files can be referenced in Appendix D and E respectively.

The purpose of Course.java class is to store information for a specific course containing the following information: credit amount, start time, end time, department, section, course number, building, course name, and SLN. Some of the information is not used when writing the input program, but it is later used to identify the class once we obtain the result from solving the IP.

The major portion of the Java code is in the file ProjectMain.java. This class contains a runnable main function that builds the input file when run. The main function run in the following order:

1. Read in building csv file and store traveling time between distance as a 2D array.

The building name itself is stored in a Java List object, and therefore can be later used to get the index of each building and find the corresponding traveling time. For example, we want to find the distance between Bagley and Kane Hall. In the list of buildings, the index for Bagley and Kane Hall are 61 and 7 respectively. We can use the index to get the distance stored in the 2D array by calling distances[7][61] or distances[61][7], which is initialized when reading in the building information. In appendix E it shows a general content of the building file. Note that the there are two different variables for each pair of buildings, one identifying the distance between buildings and one identifying the traveling time. In our model we are minimizing the traveling time so the code only stores the second variable.

2. Read in the courses csv file and store it into a 2D array list. The array list will contain i array where i is the number of time slots. Then each time slot array stores all the course information in a course object.

3. Write the input file for LPSolve. This step uses the information from step 1 and 2 above and writes the objective function and the constraints for the program.

After running the program, a file called **input.txt** is generated. A truncated version of the content is in Appendix I. We then use the command

```
lp_solve input.txt > output.txt
```

to write the program result to a text file. After getting the result, we then referenced the Java code to match the solution generated by the LPSolve to actual names and course numbers.

Commentary on Solution

Our first run of LPSolve with our constraints produced an unsurprising result. The model chose three classes all in the same building. The schedule looked like this:

Class	**Building**
CHEM 110	Bagley
MATH 124	Bagley
CHEM 142	Bagley

This minimized the objective function to zero. While this was a valid solution to our problem, it wasn't a very interesting one. The result obtained is a bit unrealistic considering that all three classes are taken in the same building, which is highly unlikely. But considering buildings like Bagley are popular lecture halls, it's unsurprising that courses in our subset are scheduled during all the time slots we chose. The solution found has the student stay in the same building in order to minimize the traveling time. So we decided to change a few constraints to improve the usefulness of the model so that the output is more similar to an actual student's schedule.

The new condition we included was to impose that two consecutive classes taken by the student cannot be in the same building. Therefore the traveling time is a non-zero integer.

After adding the new constraint we get non-zero results from the model. However, since the classes chosen from for a particular time slot are so numerous, we were not able to get a good result. To fix this, we pared down the choice of the courses to select from, based on certain class themes. The modification seemed to give better results. The table below is the

result obtained after comparing the variations of solutions. We think that this schedule best reflects an actual student schedule. In the case below we use only STEM courses and the 1 hour time slot from 8:30 to 11:30.

Time Slot	**Class**	**Building**
8:30 - 9:20	CHEM 110	Bagley
9:30 - 10:20	MATH 126	Electrical Engineering Building
10:30 - 11:20	BIOL 118	Guggenheim Hall

The total walking time between the three time slots is $2 + 1 = 3$ minutes.

In this solution, a student takes CHEM 110, MATH 126, BIOL 118, making a 13 credit schedule. Students taking this schedule need to walk 2 minutes from Bagley to Electrical Engineering Building and then walk 1 minute from Electrical Engineering Building to Guggenheim Hall. This class schedule makes sense for freshman students, since the classes are all entry level and popular among students applying to STEM majors.

Variations of Model

Variation 1:

First, we required that no two consecutive classes were in the same building. This attempted to replicate the unlikeliness that a student would have multiple classes in the same building. It also gave us a non-zero solution to the objective function. Our updated schedule looked like this:

Class	**Building**
CHEM 110	Bagley
ENGL 299	Chemistry Library
CHEM 162	Bagley

This time, the travel time was 2 minutes. **Variation 2:**

In this variation, we limit the choices for the LP to have STEM courses only, having three time slots of 8:30 - 9:20, 9:30 - 10:20 and 10:30 - 11:20 respectively. By using LPSolve we obtained the following courses:

Class	**Building**
CHEM 110	Bagley
MATH 126	Electrical Engineering Building
BIOL 118	Guggenheim Hall

The travel time obtained is 3.

Variation 3:

In this variation, we limited the courses to only STEM (Biology, Physics, Mathematics, Chemistry). We also bumped up the schedule to 4 time slots instead of 3. The time slots are 8:30 - 9:20, 9:30 - 10:20, 10:30 - 11:20, and 11:30 - 12:20. We obtain the following classes:

Class	**Building**
CHEM 110	Bagley
PHYS 122	Physics-Astronomy Auditorium
MATH 126	Electrical Engineering Building
BIOL 220	Guggenheim Hall

The total travel time is 8.

Variation 4:

In this variation we wanted to see if modifying the objective function would significantly change the result. There can be multiple combinations of classes that result in the same travel times. For example 1 + 6 and 3 + 4 both give a travel time of 7. We minimized the square of the time to account for the scenario where one travel distance is significantly larger than others. Considering students only have 10 minutes travel time between classes, minimizing each individual travel time is also important. For example, it is better to have a 3 and 4 minute travel time instead of 1, 6 because slack time from one passing period doesn't roll over to the next. Squaring the travel times favors the more balanced schedule where the values are closer together. The objective function of a solution with travel times of 1, 6 would have value 37 and 3, 4 would have value 25.

Class	**Building**
PHYS 225	Physics-Astronomy Auditorium
CHEM 162	Bagley
BIO 220	Guggenheim Hall
MATH 124	Thomson Hall

The total travel time is 2 + 3 + 4 = 9

Notice that the traveling time for both results are 9. But if consider the square sum for both case $6^2 + 1^2 + 2^2 = 41$ and $2^2 + 3^2 + 4^2 = 26$. The travel distance for the second schedule is more evenly distributed than the first schedule because the sum of squares is minimized. The second schedule is considered to be a better schedule than the first one because we

don't really want to have a traveling time that exceeds 5 minutes in a 10 minutes gap, since a student may need time to use the bathroom or fill up their water bottle.

Variation 5: Similar to variation 3, this variation keeps the same pool of STEM courses. The only change made is to shift the time slot by an hour. So the resulting time slots are 9:30 - 10:20, 10:30 - 11:20 11:30 - 12:20 and 12:30 - 13:20. The following results are obtained:

Class	**Building**
PHYS 225	Physics-Astronomy Auditorium
BIOL 118	Guggenheim Hall
MATH 120	Electrical Engineering Building
CHEM 142	Bagley

The total travel time is $6 + 1 + 2 = 9$

Conclusion

Conducting this project, we realized that there is a multitude of factors that go into picking a schedule. Every student has different considerations and priorities as well, which makes it difficult to create a general model. Our objective function minimizes just one of these factors with little regard for the others. The bottom line in this travel optimization problem is to keep travel time under 10 minutes, since any longer results in tardiness. Some students will be willing to sacrifice a few minutes in order to take a particular class. Students with less mobility may need to factor in elevator access and other restrictions.

Another modification worth considering would be allowing for gaps in classes throughout the day. Rarely does a schedule work out so nicely that the student has a contiguous block of classes. This would require significant modifications to the model, since minimizing distance between classes separated by an hour may no longer be the objective. The biggest issue with the model is that it under-fits the problem. In order to be a valuable tool, it would need to be much more configurable to meet the needs of individual students. In implementation, students should be able to filter the class list down further to classes that they are already interested in. This would be a trivial extension that would go a long way in improving the usefulness of the model. Another beneficiary of the model would be teachers and administrators, during assignment of teaching schedules. This is another direction for our future research.

If the project were to be continued, and the model made more complex (considering all days of the week, quiz/lab sections, and prerequisite limitations), a linear programming approach may no longer be suitable. The sheer number of decision variables and constraints starts to turn the model into an unwieldy and poorly extensible mess. We will keep researching and find an appropriate technique for addressing these challenges. Regardless, we still managed to obtain some useful solutions from our model.

Appendix

A. building_scraper.py

```
# building_scraper.py scrapes buildings from a UW website, generating a text
# file with a list of all buildings and their 3 letter codes

import bs4
from bs4 import BeautifulSoup as bs
from urllib.request import urlopen
import re

# open UW buildings list site
url = "https://www.washington.edu/students/reg/buildings.html"
site = bs(urlopen(url).read(), "html.parser")

# get all div tags with class 'container uw-body', get all code tags inside
codes = site.find('div', {'class', 'container uw-body'}).findAll('code')
# iterate through all codes, grab building code and name to print
for c in codes:
        # try finding building
    try:
        building = site.find(href='/maps/?l=' + c.text.lower())
        print(c.text + " " + building.text)
        # if building not found, print code with error
    except:
        print(c.text + ' error')
```

B. map_scraper.py

```
# map_scraper.py queries a UW website for distances between pairs of buildings
# generating a matrix of travel distances between every pair of buildings

from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.common.keys import Keys
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
import time
import re

# read buildings from file
with open('buildings.txt', 'r') as file:
    buildings = file.readlines()
        # get just building code
    buildings = [b.split(' ')[0] for b in buildings]
    error = 0
        i = 0 # position in building list
    # write csv header to output file
    with open('distances.txt', 'w') as out:
        # write header
        out.write(',' + re.split('\[|\]', str(buildings))[1] + '\n')
        print(',' + re.split('\[|\]', str(buildings))[1] + '\n')
    # iterate through all buildings
    for b in buildings:
        i = i + 1
        # open building file to write adjacency matrix of
        # distances between buildings
        with open('distances.txt', 'a') as out:
            # open browser and configure selenium
            # browser was reset each time to avoid sluggishness after many
                searches
            browser = webdriver.Chrome()
            browser.get('https://www.washington.edu/maps/')
            browser.implicitly_wait(10)

            # click on directions
            directions = browser.find_element_by_id('switcher-directions')
            directions.click()

            # initialize elements to send to-from info to
            search_current = browser.find_element_by_id('current')
            search_destination = browser.find_element_by_id('destination')

            prev_details = ''
            row = b + ','
            j = 0
```

```
            # iterate through all buildings again
            for b2 in buildings:
                j = j + 1
                # only check for distance one way
                if i < j and b != b2:
                    # search for building b, select it
                    search_current.clear()
                    search_current.send_keys(b + Keys.RETURN)
                    result = browser.find_element(By.CLASS_NAME, 'search-result')
                    result.click()
                    # search for building b2, select it
                    search_destination.clear()
                    search_destination.send_keys(b2 + Keys.RETURN)
                    result = browser.find_element(By.CLASS_NAME, 'search-result')
                    result.click()
                    # wait for travel distance to load on page
                    time.sleep(0.8)
                    details = browser.find_element(By.CLASS_NAME, 'print').text
                    # error handling in case site doesn't refresh query
                    if details == prev_details:
                        # wait longer
                        time.sleep(1.6)
                        # search again for b and b2
                        search_current.clear()
                        search_current.send_keys(b + Keys.RETURN)
                        result = browser.find_element(By.CLASS_NAME,
                            'search-result')
                        result.click()
                        search_destination.clear()
                        search_destination.send_keys(b2 + Keys.RETURN)
                        result = browser.find_element(By.CLASS_NAME,
                            'search-result')
                        result.click()
                        time.sleep(0.8)
                        details = browser.find_element(By.CLASS_NAME, 'print').text
                    # if details are still the same after refresh
                    if details == prev_details:
                        error = error + 1
                    prev_details = details
                    output = details.split(' ')
                    # try grabbing miles and walk time
                    try:
                        output = output[0] + ' ' + output[4]
                    except:
                        output = 'error'
                    row = row + output + ','
                else:
                    row = row + ","
                search_current.clear()
                out.write(row[:-1] + '\n')
```

```
                print(row[:-1] + '\n')
                browser.close()
print('error' + error)
```

C. class_scraper.py

```
# class_scraper scrapes all classes for a particular quarter from a UW
# website, generating a csv with columns: name, number, dept, sln, start time,
    etc.

import bs4
from bs4 import BeautifulSoup as bs
from urllib.request import urlopen
import re
# -*- coding: utf-8 -*-

# get all classes for a department, accept url to department's schedule
def get_classes(url):
    subsite = '' # store website
    error = 0 # track number of failures on page
    try:
        subsite = bs(urlopen(url), "html.parser")
    # if site can't be found, exit
    except:
        print('error: ' + str(url))
        return

    # open courses to append output to
    with open('courses.csv', 'a') as file:
        # store fields of each class
        department = ''
        class_number = ''
        class_title = ''
        courses = subsite.find_all('table')
        header_flag = False
        # iterate through all courses on page
        for c in courses:
            # skip first table on page
            if c.has_attr('bgcolor') and c.get('bgcolor') != '#d3d3d3':
                header_flag = True # mark course header as found
                # extract class title information
                title = c.find('a')
                title_text = title.contents[0]
                title_text = title_text.replace(u'\xa0', u'').strip()
                department = ' '.join(title_text.split(' ')[:-1])
                class_number = title_text.split(' ')[-1]
                try:
                    class_title = title.find_next_sibling().contents[0]
                    class_title = re.split('<|>', class_title)[0].strip()
                except:
                    pass
                # skip junk data
                if title_text != '?' or class_title != '?':
```

```
            pass
        else:
            department = ''
            class_number = ''
            class_title = ''
    # once header is found, fill in rest of the fields
    elif header_flag:
        sections = c.find_all('pre')
        # read every section of class
        for section in sections:
            # fields to store metadata
            sln = ''
            sect = ''
            cred = ''
            class_no = ''
            days = ''
            start = ''
            end = ''
            building = ''
            room_no = ''
            instructor = ''

            out = section.contents
            # iterate through metadata
            for i, o in enumerate(out):
                # if first chunk of text is not valid
                if i == 0 and isinstance(o, bs4.element.NavigableString):
                    continue
                # set sln
                if o.name == 'a' and sln == '':
                    sln = o.contents[0]
                # set building
                elif o.name == 'a':
                    building = o.contents[0]
                # skip classes that haven't been scheduled
                elif 'to be arranged' not in o and sect == '':
                    # set section, credits, days, start time, end time
                    try:
                        line = re.split(' {2,}', str(o))
                        sect = line[0].strip()
                        cred = line[1].strip()
                        days = line[2].strip()
                        if len(days) > 6:
                            sp = re.split(' |-', days)
                            days = sp[0]
                            start = sp[1]
                            end = sp[2]
                        else:
                            start = line[3].split('-')[0]
                            end = line[3].split('-')[1]
```

```
                        except:
                            pass
                    # if all fields are valid, append data to file
                    if all([department, class_number, class_title, sln, sect,
                        cred, days, start, end, building]):
                        # build string for row in csv
                        output = department + ',' + class_number \
                            + ',' + class_title + ',' + sln + ',' + sect \
                            + ',' + cred + ',' + days + ',' + start + ',' \
                            + end + ',' + building + '\n'
                        # output result to console and file
                        print('writing: ' + output)
                        file.write(output)
                    else:
                        error = error + 1
    # view total number of errors
    print("errors: " + str(error))

# location of all department schedules
url = "https://www.washington.edu/students/timeschd/WIN2017/"
site = bs(urlopen(url).read(), "html.parser")

# find all links on the page to individual department schedules
links = site.findAll('a')

# iterate through links, getting classes for each department
for link in links:
    link = str(link).split('\"')[1]
    # check that link is of correct format
    if '/' not in link and 'html' in link:
        print(link)
        get_classes(url + link)
```

D. Course.java

```
/*
    This is a Class that stores all the information needed to define a course.
    It contains a constructor that builds a course and the getter function for
    all information if needed.
*/
public class Course {

    private int credit;
    private int startTime;
    private int endTime;
    private String department;
    private String section;
    private int courseNum;
    private String building;
    private String courseName;
    private int SLN;

    // constructor
    public Course(String department, int courseNum, String courseName, int SLN,
        String section, int credit, int startTime, int endTime, String building) {
        this.credit = credit;
        this.startTime = startTime;
        this.endTime = endTime;
        this.department = department;
        this.section = section;
        this.courseNum = courseNum;
        this.building = building;
        this.courseName = courseName;
        this.SLN = SLN;
    }

    // getter functions
    public int getCredit() {
        return credit;
    }

    public int getStartTime() {
        return startTime;
    }

    public int getEndTime() {
        return endTime;
    }

    public String getDepartment() {
        return department;
    }
```

```
    public String getSection() {
        return section;
    }

    public int getCourseNum() {
        return courseNum;
    }

    public String getBuilding() {
        return building;
    }

    public String getCourseName() {
        return courseName;
    }

    public int getSLN() {
        return SLN;
    }
}
```

E. ProjectMain.java

```
import java.io.*;
import java.util.*;

/*
    This is a class that contains a runnable main function that builds
    up the integer program system by reading in relevant information
    (buildings, courses) and output the result into a specified text
    file
*/
public class ProjectMain {

    private static final String OUTPUT_FILE = "input.txt";
    private static final String BUILDING_FILE = "building_distances.csv";
    private static final String COURSE_FILE = "stem.csv";
    // specified the number of time slots
    private static final int NUM_TIME_SLOT = 3;

    // data structures that use to store all the relative information
    private static int[][] distances;
    private static ArrayList<String> buildingList;
    private static ArrayList<ArrayList<Course>> courseList;
    private static Map<String, Set<String>> courseVariable;

    /*
        This is the main function that runs the program
    */
    public static void main(String[] args) throws IOException {

        // initialize the buffer reader and writer
        BufferedReader br = null;
        FileWriter fw = null;
        BufferedWriter bw = null;
        String csvSplitBy = ",";

        try {
            // reads the building input file and store it in memory
            br = new BufferedReader(new FileReader(BUILDING_FILE));
            readBuildingAndDistance(br, csvSplitBy);

            // reads the course input file and store it in memory
            br = new BufferedReader(new FileReader(COURSE_FILE));
            readCourses(br, csvSplitBy);

            // organize all courses using their department and course number
            buildCourseVariable();

            // write the objective function and constraints
```

```
            fw = new FileWriter(OUTPUT_FILE);
            bw = new BufferedWriter(fw);
            writeInputFile(bw);

        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } finally {
            if (br != null) {
                br.close();
            }
            if (bw != null) {
                bw.close();
            }
            if (fw != null) {
                fw.close();
            }
        }
    }

    /*
        This is a private helper function that takes in a BufferedReader for
        file reading and a String that helps split the csv line into array.
        This function reads the building file input and stores all the
        distance between buildings in memory
    */
    private static void readBuildingAndDistance(BufferedReader br, String
        csvSplitBy) throws IOException{
        String line = "";
        buildingList = new ArrayList<>();

        String[] building = null;
        if ((line = br.readLine()) != null) {
            building = line.split((csvSplitBy));
        }

        distances = new int[building.length-1][building.length-1];

        int count = 1;
        while ((line = br.readLine()) != null) {
            String[] info = line.split(csvSplitBy);
            buildingList.add(info[0]);
            for (int i = count + 1; i < info.length; i++) {
                String distTime = info[i];
                String[] split = distTime.split(" ");
                // only taking the second variable, which is the traveling
                // time
                int time = Integer.parseInt(split[1]);
                distances[count-1][i-1] = time;
                distances[i-1][count-1] = time;
            }
```

```
            count++;
        }
        // Electrical Engineering Building appears in building file as EE1 but EEB
        // in course file so replace EE1 with EEB
        buildingList.set(buildingList.indexOf("EE1"), "EEB");
    }

    /*
        This is a private helper function that takes in a BufferedReader
        for file reading and a String that helps split the csv line into
        array. This function reads the course file input and stores all
        the course information into the correct time slot in memory
    */
    private static void readCourses(BufferedReader br, String csvSplitBy) throws
        IOException{
        String line = "";
        courseList = new ArrayList<>();
        for (int i = 0; i < NUM_TIME_SLOT; i++) {
            courseList.add(new ArrayList<>());
        }

        if ((br.readLine()) != null) {
            while ((line = br.readLine()) != null) {
                String[] courseInfo = line.split(csvSplitBy);
                int startTime = Integer.parseInt(courseInfo[8]);
                int endTime = Integer.parseInt(courseInfo[9]);
                String courseBuilding = courseInfo[10];
                if ((br.readLine()) != null) {
                    while ((line = br.readLine()) != null) {
                        String[] courseInfo = line.split(csvSplitBy);
                        int startTime = Integer.parseInt(courseInfo[8]);
                        int endTime = Integer.parseInt(courseInfo[9]);
                        String courseBuilding = courseInfo[10];
                        if (buildingList.indexOf(courseBuilding) != -1) {
                            if (startTime >= 830 && endTime <= 920) {
                                addCourse(0, courseInfo);
                            } else if (startTime >= 930 && endTime <= 1020) {
                                addCourse(1, courseInfo);
                            } else if (startTime >= 1030 && endTime <= 1120) {
                                addCourse(2, courseInfo);
                            }
                        }
                    }
                }
            }
        }
    }

    /*
        This is a private helper function that takes in a BufferedWriter
```

```
    for file writing. This function writes the entire program in a
    text file that is later used to feed into lp_solve. The function
    writes the objective function and the implicit and explicit
    constraints
*/
private static void writeInputFile(BufferedWriter bw) throws IOException {
    // write objective function
    String objective = "min:";
    for (int beginSlot = 1; beginSlot < NUM_TIME_SLOT; beginSlot++) {
        for (int i = 1; i <= courseList.get(beginSlot-1).size(); i++) {
            Course prevCourse = courseList.get(beginSlot-1).get(i-1);
            for (int j = 1; j <= courseList.get(beginSlot).size(); j++) {
                Course nextCourse = courseList.get(beginSlot).get(j-1);
                int dist = getDistance(prevCourse.getBuilding(),
                    nextCourse.getBuilding());
                if (dist > 0) {
                    int squareDist = (int) Math.pow(dist, 2);
                    objective += " + " + squareDist + "y_" + i + "_" + j + "_"
                        + beginSlot;
                }
            }
        }
    }
    objective += ";\n";
    bw.write(objective);

    // write constraint 1, time slot constraints, each time slot need to be
    // filled with exactly one course
    String timeSlotCons = "";
    for (int i = 1; i <= courseList.size(); i++) {
        for (int j = 1; j <= courseList.get(i-1).size(); j++) {
            timeSlotCons += "+ x_" + i + "_" + j + " ";
        }
        timeSlotCons += "= 1;\n";
    }
    bw.write(timeSlotCons);

    // write constraint 2, cannot take multiple sections for the same course
    String sectionCons = "";
    for(Map.Entry<String, Set<String>> entry: courseVariable.entrySet()) {
        sectionCons = "";
        for(String var: entry.getValue()) {
            sectionCons += "+ " + var + " ";
        }
        bw.write(sectionCons + "<= 1;\n");
    }

    // write constraints 3, credit constraints, the total credit for
    // the chosen class need to be greater than or equal to 12 and
    // less than or equal to 18
```

```
String creditCons = "";
for (int i = 1; i <= courseList.size(); i++) {
    for (int j = 1; j <= courseList.get(i-1).size(); j++) {
        Course course = courseList.get(i-1).get(j-1);
        creditCons += "+ " + course.getCredit() + "x_" + i + "_" + j + " ";
    }
}
bw.write(creditCons + ">= 12;\n");
bw.write(creditCons + "<= 18;\n");

// write constraint 4 and 5, travel time constraints
for (int beginSlot = 1; beginSlot < NUM_TIME_SLOT; beginSlot++) {
    for (int i = 1; i <= courseList.get(beginSlot-1).size(); i++) {
        Course prevCourse = courseList.get(beginSlot-1).get(i-1);
        for (int j = 1; j <= courseList.get(beginSlot).size(); j++) {
            Course nextCourse = courseList.get(beginSlot).get(j-1);
            int dist = getDistance(prevCourse.getBuilding(),
                nextCourse.getBuilding());
            int nextSlot = beginSlot + 1;

            String prevCourseVar = "x_" + beginSlot + "_" + i;
            String nextCourseVar = "x_" + nextSlot + "_" + j;
            String distVar = "y_" + i + "_" + j + "_" + beginSlot;
            String distCons = "";

            if (dist == 0) {
                // constraint 5
                // if the distance is 0 then there are no distance
                // between classes instead set a constraint that the
                // two given class cannot both be taken
                distCons += prevCourseVar + " + " + nextCourseVar + " <=
                    1;\n";
            } else {
                // constraint 4
                // if the distance is not 0 then write the
                // travel time constraints
                distCons += distVar + " <= " + prevCourseVar + ";\n";
                distCons += distVar + " <= " + nextCourseVar + ";\n";
                distCons += distVar + " >= " + prevCourseVar + " + " +
                    nextCourseVar + " - 1;\n";
            }
            bw.write(distCons);
        }
    }
}

// write constraint 6, binary constraints
String binaryCons = "bin ";
for (int beginSlot = 1; beginSlot < NUM_TIME_SLOT; beginSlot++) {
    for (int i = 1; i <= courseList.get(beginSlot-1).size(); i++) {
```

```
            for (int j = 1; j <= courseList.get(beginSlot).size(); j++) {
                binaryCons += "y_" + i + "_" + j + "_" + beginSlot + ", ";
            }
        }
    }

    for (int i = 1; i <= NUM_TIME_SLOT; i++) {
        for (int j = 1; j <= courseList.get(i-1).size(); j++) {
            binaryCons += "x_" + i + "_" + j + ", ";
        }
    }
    binaryCons = binaryCons.substring(0, binaryCons.length()-2) + ";\n";
    bw.write(binaryCons);
}

/*
    This is a private helper function that gets the distance between two
        buildings
*/
private static int getDistance(String b1, String b2) {
    int b1Index = buildingList.indexOf(b1);
    int b2Index = buildingList.indexOf(b2);
    return distances[b1Index][b2Index];
}

/*
    This is a private helper function that matches each course
    variable x_i_j to the course number so that this can be used
    to set up the constraint to differentiate different sections
*/
private static void buildCourseVariable() {
    courseVariable = new HashMap<>();
    for (int i = 1; i <= courseList.size(); i++) {
        for (int j = 1; j <= courseList.get(i-1).size(); j++) {
            Course course = courseList.get(i-1).get(j-1);
            String var = "x_" + i + "_" + j;
            String key = course.getDepartment();
            if (!courseVariable.containsKey(key)) {
                courseVariable.put(key, new HashSet<>());
            }
            courseVariable.get(key).add(var);
        }
    }
}

/*
    This is a private helper function that takes in the index
    as the time slot number and a String array that contains
    the course information. The function will store the given
    course information into an array in the given time slot
```

```
        as a Course Object
    */
    private static void addCourse(int index, String[] courseInfo) {
        courseList.get(index).add(new Course(
                courseInfo[1],
                Integer.parseInt(courseInfo[2]),
                courseInfo[3],
                Integer.parseInt(courseInfo[4]),
                courseInfo[5],
                Integer.parseInt(courseInfo[6]),
                Integer.parseInt(courseInfo[8]),
                Integer.parseInt(courseInfo[9]),
                courseInfo[10]));
    }
}
```

F. Table of building names and codes

code	**name**
ACC	John M. Wallace Hall
AERB	Aerospace & Engineering Research Building
ALB	Allen Library
AND	Anderson Hall
ARC	Architecture Hall
ART	Art Building

G. Truncated matrix of building distance (walking distance in miles, walking time in minutes)

	ACC	AERB	ALB	AND	ARC	ART
ACC		0.5 12	0.5 12	0.4 9	0.3 7	0.7 16
AERB			0.2 4	0.2 4	0.3 5	0.4 8
ALB				0.3 6	0.2 4	0.3 6
AND					0.3 5	0.5 12
ARC						0.4 10
ART						

H. Table of classes

dept	num	sln	sect	cred	day	start	end	build
HONORS	205	15407	A	5	MWF	930	1050	MGH
ESS	100	14657	A	2	MW	1230	1320	KNE
CHSTU	101	12573	A	5	MTWTh	1230	1320	SMI
GERMAN	101	15229	D	5	MTWThF	1230	1320	DEN
PSYCH	101	19399	C	5	MTWThF	1230	1320	KNE
DRAMA	101	13519	A	5	MWF	1230	1320	KNE
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
AIS	275	22330	B	5	MW	1730	1920	BNS
MUSEN	100	17730	B	1	M	1830	2020	MNY
ECON	200	13764	G	5	MW	1830	2020	ARC

I. general content of input.txt (file used as input for LPSolve)

```
min: + 25y_1_1_1 + 25y_1_2_1 + 25y_1_3_1 + 9y_1_5_1 + 100y_1_7_1 + 4y_1_8_1 +
    25y_1_9_1 + 4y_1_10_1 + ... (about 200 similar variables)

+ x_1_1 + x_1_2 + x_1_3 + x_1_4 + x_1_5 + x_1_6 = 1;
(2 more similar lines)

+ x_3_7 + x_2_4 + x_1_3 + x_1_1 + x_1_2 <= 1;
(about 3 more similar lines)

+ 3x_1_1 + 5x_1_2 + 5x_1_3 + 5x_1_4 + ... (about 20 more variables) >= 12;
+ 3x_1_1 + 5x_1_2 + 5x_1_3 + 5x_1_4 + ... (about 20 more variables) <= 18;

y_1_1_1 <= x_1_1;
y_1_1_1 <= x_2_1;
y_1_1_1 >= x_1_1 + x_2_1 - 1;
(about 30 more similar combination of lines)

x_1_1 + x_2_4 <= 1;
(about 10 more similar lines)

bin y_1_1_1, y_1_2_1, y_1_3_1, y_1_4_1, y_1_5_1, y_1_6_1, y_1_7_1, y_1_8_1,
    y_1_9_1, y_1_10_1, ... (about 300 similar variables)
```

References

"Guide to Classroom Locations." University of Washington, Nov. 2016. Web. <https://www.washington.edu/students/reg/buildings.html>.

"University of Washington Seattle Time Schedule." University of Washington, Nov. 2016. Web. <https://www.washington.edu/students/timeschd/>.

"Campus Maps." Campus Maps. University of Washington, n.d. Web. <https://www.washington.edu/maps/>.

E.L.Lawler., Eugene L The Traveling salesman problem : a guided tour of combinatorial optimization. Wiley, Chichester [West Sussex] ; New York, 1985.

Saltzman, R. M., An Optimization Model for Scheduling Classes in a Business School Department, California Journal of Operations Management, Vol. 7, No. 1, pp. 95-103, Feb. 2009.

Martin, C. H., Ohio Universitys College of Business Uses Integer Programming to Schedule Classes, Interfaces, Vol. 34 (6), 2004, 460-465

Winch, Janice K., and Jack Yurkiewicz. "STUDENT CLASS SCHEDULING WITH LINEAR PROGRAMMING." Proceedings for the Northeast Region Decision Sciences Institute; 2013, p374, Apr. 2013.