Curriculum Graph Visualizer

Dana Oira Toribio

Dr. Kevin Wortman, Advisor

Dr. Kent Palmer, Reviewer

Department of Computer Science

California State University, Fullerton

May 18, 2016

Contents

List of Figures

List of Tables

Abstract – **The graph academic curricula is complex and can be difficult to visualize it. This program demonstrates the ability to take a set of course curriculum data and user inputs to provide study plan suggestions which are converted to a graph visualization using Python GraphViz.**

1   Introduction

1.1   Problem Description

In academia, a course curriculum is available which lists courses and electives required for completion of a degree program. Often times, a course curriculum is simply given as a list of courses.

Table 1. Computer Science Core Curriculum

Lower Division Core (18 units)

CPSC 120	Introduction to Programming[1]
CPSC 121	Programming Concepts[1]
CPSC 131	Data Structures Concepts
CPSC 223	Object-oriented Programming Language[2]
CPSC 240	Computer Organization and Assembly Language
CPSC 254	Software Development with Open Source Systems

Upper Division Core (28 units)

CPSC 311	Technical Writing for Computer Science
CPSC 315	Social and Ethical Issues in Computing
CPSC 323	Programming Languages and Translation
CPSC 332	File Structures and Database Systems
CPSC 335	Algorithm Engineering
CPSC 351	Operating Systems Concepts
CPSC 362	Software Engineering
CPSC 440	Computer System Architecture
CPSC 471	Computer Communications
CPSC 481	Artificial Intelligence

Shown in **Table 1** is the core curriculum for the Computer Science major at California State University Fullerton from the 2010 student handbook. What this format does not clarify with is visualizing and understanding the relationship between all the courses. Underlying information like prerequisites, co-requisites and the progression of the curriculum in a semester-to-semester basis is difficult to visualize from a simple list of courses.

Figure 1. Undergraduate Computer Science Curriculum Graph
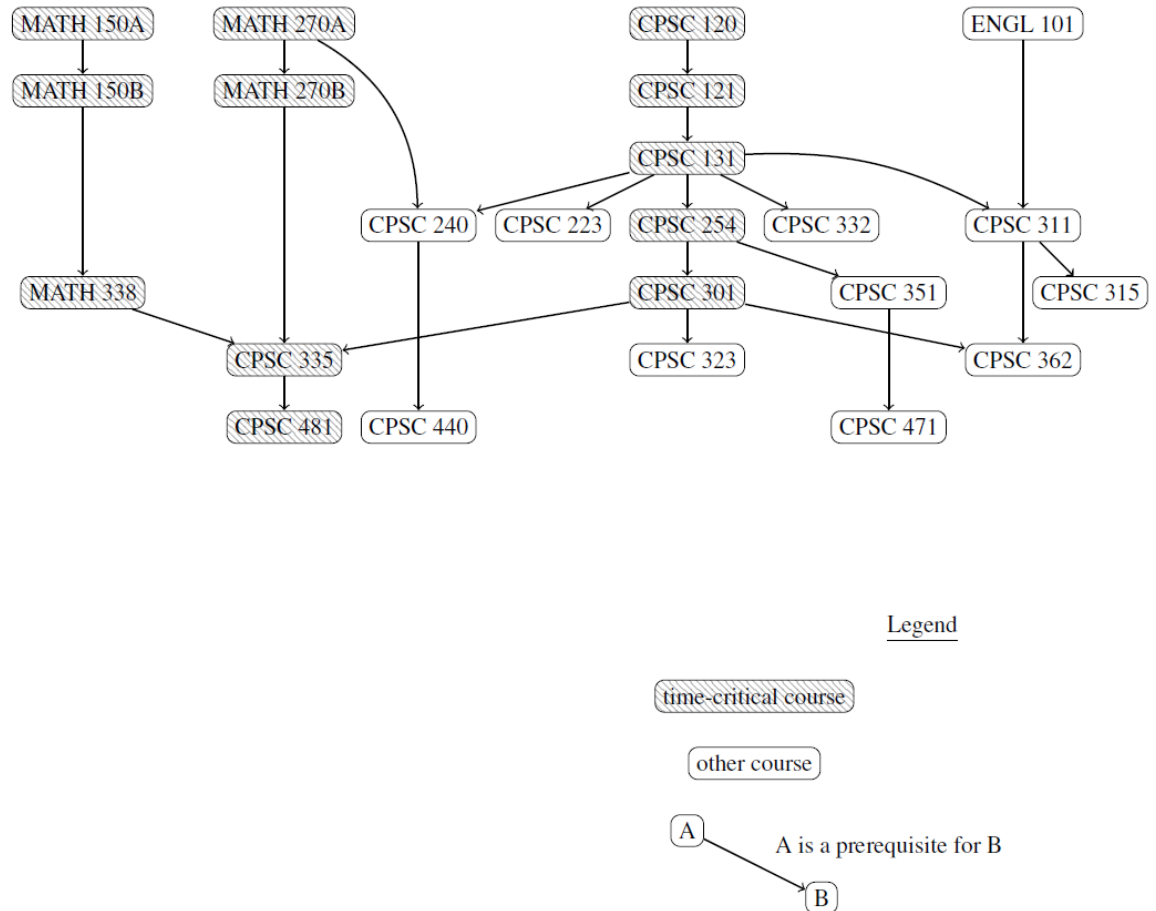
Figure 1 shows a graphical representation of the course curriculum in Figure 1. This figure is also known as a curriculum graph. The curriculum graph is able to show course prerequisites and co-requisites with a legend key to add clarity to the relationship between all the courses. Paired with the course list from Figure 1, the curriculum graph gives a better understanding of the course progression.

With the contrast of the course listing and the curriculum graph, it's evident that the graph of courses and their prerequisites is complex, and students and faculty can both struggle to visualize it.

1.2   Project Objectives

The objective of this project is to create a curriculum graph which will take the input of curriculum graph data of courses names, prerequisites and locations to output a graph drawing. The graph drawing will organize its output to minimize overlapping edges, organize the graph by suggesting courses by semester, the courses in the graph by tracks and to indicate completed courses.

The final results of the project will include an (1) Algorithm Design Document which detail the rationale behind the algorithm selection, included in this document, and (2) a command line application that is able to perform all the tasks aforementioned in the Objectives:

1. *Create a graph drawing.* The user will input course information which the program will use to generate a graph drawing that is readable. The graph drawing will include the following attributes below.

2. *Organize graph into tracks.* Since there are several core tracks as well as elective tracks within the major, the program will aim to generate the curriculum graph's output to be organized by these tracks. In general, the core tracks will be categorized based on its root node. Courses in the same track will be organized within the same cluster.

3. *Minimize overlapping edges.* Decreasing overlapping edges is a best practice for graph drawings. Due to the nature of course organization into strata and tracks, overlapping edges will likely be unavoidable. This project will prioritize strata and track organization before edge minimization.

4. *Indicate complete courses.* Indicating course completion may be done by coloring the completed courses' nodes in a different color. This feature will be helpful with tracking completed courses and tracking which courses are to be done.

The main goal of the final demonstration of the project is to efficiently show the program's full range of functionality.

1.3 Development Environment

Hardware

The hardware used to develop the project includes the following:

1. Central Processing Unit: Intel Core i5-3570K 3.4GHz
2. Random Access Memory: 8GB
3. Graphical Card: PNY NVIDIA GeForce 560 Ti Fermi 1GB
4. Operating System: Windows 10

Software

The software used in the development of this program includes the following:

1. Programming Languages: Python 3.5.1 (v3.5.1:37a07cee5696), DOTS
2. Graph Visualization: GraphViz 2.38
3. Python GraphViz Package: GraphViz 0.4.10
4. Editor: Sublime Text Build 3013

1.4 Operational Environment

Hardware

The hardware needed to run this program includes:

1. Central Processing Unit: 1.5GHz or faster processor
2. Random Access Memory: 1GB
3. Screen Resolution: 1024x768 pixels
4. Operating Systems: Windows 7/8/10, Linux, Mac OS X v10.9/10.10/10.11

Software

The software needed to run this program includes:

1. Programming Languages: Python 3
2. Graph Visualization: GraphViz 2
3. Graph Viewer: Adobe Acrobat Reader

2 Requirements Description

2.1 External Functions

1. Educational institution's curriculum specifications.

    a. *Unit limitations:* A maximum of 16 units is allowed per semester.
    b. Prerequisite requirements

2. GraphViz and Python GraphViz formatting requirements.

2.2 External Interfaces

User Interfaces

The user interfaces shall follow the following requirements:

1. The program shall be executed through the command line.

In the interest of time, a command line execution for the program suffices for the demonstration of the project's functions.

Hardware Interfaces

The hardware interfaces shall follow the following requirements:

1. The program shall respond to keyboard input.
2. The graphical card shall generate a graph to the screen.

No actual hardware was created in the development of this process. To simplify the project, the only hardware interaction required keyboard and mouse navigation to access the command line prompt.

Software Interfaces

The CGV utilizes several external interfaces:

1. *Programmatic Interface:* The CGV shall access functions of the external software through a library (Python GraphViz 0.4.10, Python os module, Python re module, Python sys module).

2. *Graph Drawing Interface:* The CGV shall access graph drawing capabilities through a visualization software (GraphViz 2.38).

3 Design Description

3.1 Architecture

The main vision of the project was to create a program that would simplify the way a user can view the course curriculum and receive suggestions on courses to take per semester. Simplicity was the main ideal during all phases of the software development life cycle.

The architecture of data flow is described in **Figure 2**:

1. *curr.txt -> cgv.py:* Course curriculum in curr.txt is passed to cgv.py.

2. *cgv.py -> studyplan.txt:* A new file that contains a revised list of courses and a suggestion of courses to take per semester for the user's study plan is created.

3. *studyplan.txt -> cgc.py:* Study plan is passed to cgc.py.

4. *cgc.py -> studyplan.py:* The translation of the study plan file to an executable Python GraphViz format is created.

5. studyplan.py -> curriculum graph visualization (studyplan.pdf, studyplan.gv): A PDF and DOT file of the study plan are created.

Figure 2. Data flow of the curriculum graph visualization program
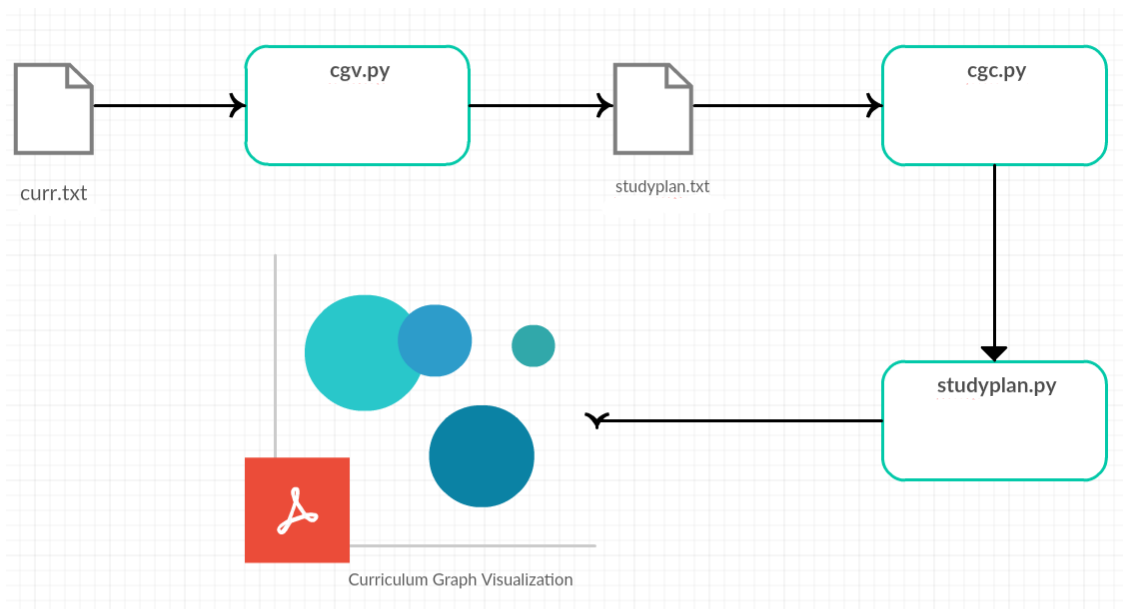
3.2 Internal Functions

There are two main files involved in the curriculum input (CIF) to graph generation:

1. *cgv.py, Curriculum Graph Visualizer:* This file guides the main events for the creation of the Study Plan list.

    a. Input: curriculum.txt
    b. Output: studyplan.txt

2. *cgc.py, Curriculum Graph Converter:* This file does the translation functions from the CIF to generate a Python GraphViz executable code.

    a. *Input:* studyplan.txt
    b. *Output:* studyplan.py

The following files are created by the user:

1. *curriculum.txt:* The CIF that has a major's entire curriculum, prerequisites and track electives.

The following describes the files generated by cgv.py and cgc.py:

1. *studyplan.txt:* An updated CIF which includes only required courses and elective track courses specified by the user.

2. *studyplan.py:* A translation of studyplan.txt to Python GraphViz, which launches the curriculum graph visualization.

Figure 3. Translation between studyplan.txt to studyplan.py
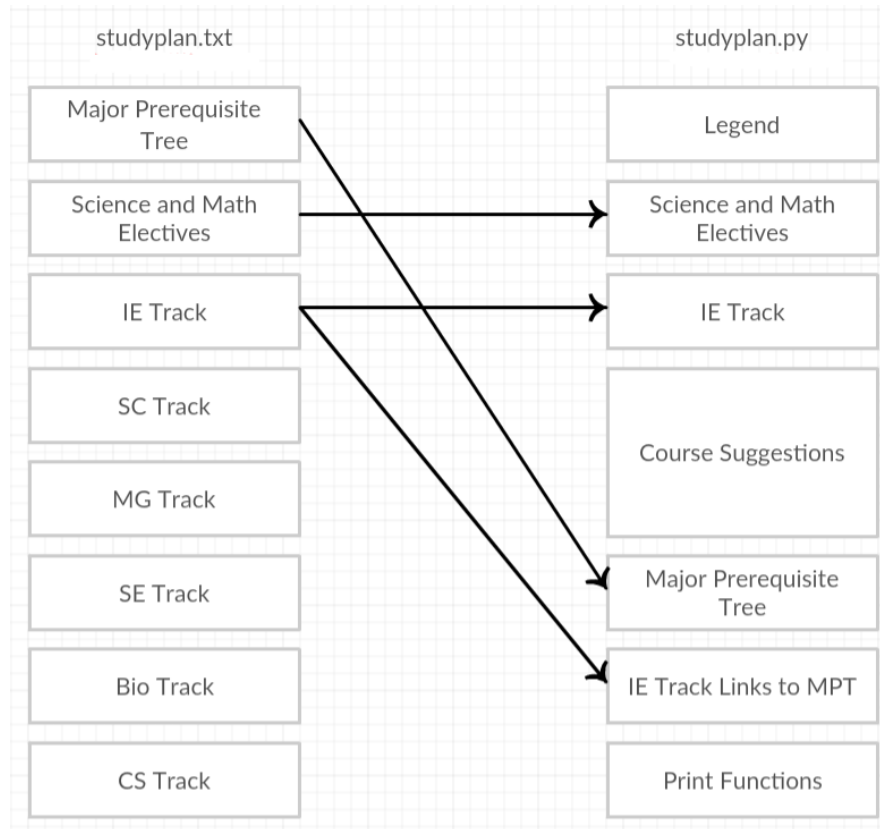
3.3 Internal Interfaces

1. *Deque:* From the Python collections library. Used to implement a more efficient deque function than using a regular list as a queue.

2. *OrderedDict:* From the Python collections library. Used to preserve the order of data added to a dictionary, rather than the default optimized sort.

3. *OS Module:* Python's built-in OS module used for executing Python files.

4. *Python GraphViz:* The Python GraphViz library which generates the graph drawings.

5. *Regular Expressions Module:* Python's built-in regular expressions module used for matching course patterns.

6. *System Module:* Python's built-in system module used to take in command line arguments.

4 Implementation

4.1 Organization of Source File Structure

cgv.py, Curriculum Graph Visualizer

The source file structure is located in cgv.py.

1. Read in Curriculum Input File (CIF).
2. Find elective tracks.
3. Print electives to console.
4. Read in one (1) user input for elective track.
5. Read courses in user's study plan, which include:
    a. Required courses
    b. Elective track courses
6. Writes courses to studyplan.txt file.
7. Create study plan tree of user's selected track.
8. Console prints list of courses in study plan.
9. Take in user input for completed courses.
10. Update study plan tree with completed courses.
11. Write suggestions to studyplan.txt.
12. Run cgc.py

The classes and functions found in cgv.py are listed below.

1. class Node:
    a. __init__(self, data):
        i. self.data = data
        ii. self.children = []
        iii. self.parents = []
        iv. self.units = 0
        v. self.priority = False
        vi. self.taken = False
        vii. self.studyplan = False
    b. add_child(self, obj)
    c. get_child(self)
    d. add_parent(self, obj)
    e. get_parent(self)
    f. set_priority(self)
    g. set_units(self)
    h. set_taken(self)
    i. set_studyplan(self)

2. main()
   a. bfs(root)
   b. create_studyplan(bfs)
   c. create_tree(file)
   d. get_major(file)
   e. get_tracks(file)
   f. handle_taken_input(user_input, course_list, taken_list):
   g. in_limit(course_units, unit_count)
   h. input_elective(elec_trk)
   i. input_taken_courses(course_list)
   j. prereqs_dne(course, term)
   k. print_courses(courses)
   l. print_tracks(elec_trk)
   m. priority(course)
   n. sp_to_studyplan(term, course)
   o. sp_to_queue(queue, course)
   p. sp_to_newterm(studyplan, term, course)
   q. sp_do_queue(queue, unit_count, studyplan, term)
   r. sp_end_queue(queue, unit_count, studyplan, term)
   s. test_print_bfs(bfs)
   t. test_print_sp_queue(queue)
   u. test_print_sp(result(studyplan)
   v. test_print_studyplan(studyplan)
   w. test_print_tree(tree)
   x. test_print_prereqs(tree)
   y. update_taken(tree, taken_list)
   z. write_core_elecs(infile, outfile, elective, courses)
   aa. write_suggestions(studyplan, outfile)

cgc.py, Curriculum Graph Converter

The source file structure for cgc.py:

1. Open studyplan.txt to read.
2. Open study plan.py to write.
3. Read studyplan.txt line by line.
4. Update the following attributes with the Python GraphViz translation.
   a. req_electives
   b. trk_electives
   c. completed_courses
   d. suggested_courses
   e. core_courses
   f. elective_prereqs
5. Write resulting translation to studyplan.py.
6. Run studyplan.py

4.2 Reference List of Files

A reference list of files in order of input/output in the process.

1. *curriculum.txt:* Curriculum Input File (CIF)
2. *cgv.py:* Curriculum Graph Visualizer
3. *studyplan.txt:* Modified CIF
4. *cgc.py:* Curriculum Graph Converter
5. *studyplan.py:* Translated studyplan.txt to Python GraphViz
6. *studyplan.gv:* DOT code generated by Python GraphViz
7. *test_results.txt:* Results of execution data

5 Test and Integration

5.1 Plan

Since the project was created by one person, there were no formal daily standup meetings. Daily progress was recorded internally as reference. Testing was done individually and bug fixes were part of the development phase.

The integration phase resulted in the deliverable of the completed project as a command line executable application.

Table 2 shows the proposed project schedule. The project began in early January through the duration of the semester, allowing approximately 308 hours of work over the course of 20 weeks.

Table 2. Project Schedule

2016	January				February				March				April				May				Summary	
Tasks:	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	Hours	Percent
Requirements	8	8																			16	5.2%
Research & Discovery	8	8	8	8	8	8															48	15.6%
Design			8	8	8	8	8	8													48	15.6%
Integrate & Test							8	8	20	20	20	20	20	20	8	8					152	49.4%
Write User's Manual															8	8					16	5.2%
Write Final Report																	8	8	8		24	7.8%
Demonstrate																				4	4	1.3%
Hours	16	16	16	16	16	16	16	16	20	20	20	20	20	20	16	16	8	8	8	4	**308**	100.0%

There is some overlap with the tasks in that the next task may be worked on as the preceding task is reaching completion. To comply with the SCRUM time-boxing of two-week long sprints, a new task typically starts every two (2) weeks aside from the Integration portion where the bulk of development will occur. There will be a total of five (5) iterations, beginning

the first week of every month. Each iteration had two (2) sprints each, giving a total of twenty (20) sprints for the entire duration of the project.

Requirements gathering was given 16 hours since many of the requirements were gathered before the time of writing the proposal. Extra time was used for research and discovery of algorithm patterns and testing since the Algorithm Phase proved to be the most difficult portion of the project.

The project's development was conducted in two phases. Phase 1 included the Algorithm Phase, to determine which graph drawing algorithms will be implemented into the program. The second phase was the Development Phase, in which the development of the program occurred.

Phase I – Algorithm

The Algorithm Phase for the first phase of the project includes research activities to decide which algorithms will be chosen for the implementation of the graph drawings. The two activities in the Algorithm Phase are Algorithm Discovery and Algorithm Testing.

Algorithm Discovery. The Algorithm Discovery portion of this activity will be continued research on graphing algorithms.

Algorithm Testing. This activity includes testing of graphing algorithms to determine their suitability against the project's objectives.

The resulting output of this Algorithm Phase are (1) selected algorithms implemented in Phase II and (2) an Algorithm Design Document which includes pseudocode brief description of the rationale behind the algorithm selection. The Algorithm Design Document is found in section **5.2 Results**.

Phase II – Development

The Development Phase follows the Algorithm Phase. This phase includes common software engineering development activities including software design, software development and quality assurance testing.

With the time constraint limiting the entire project to the duration of a semester, the development process will loosely follow the Agile methodology using SCRUM. The project was time-boxed into 2-week sprints beginning in January. Each sprint begins with a meeting with the project advisor to discuss the project progress and if the sprint's goals are reasonably appointed. The following meeting followed up on the project's progress and demonstrated any working samples from that sprint's development.

Testing

Testing was done prior to the development and occurred during the research phase. In order to understand the workflow of how the project would be executed, Ideal Results were created as part of the testing data. The Ideal Results were used to prototype the expected output of a proper graph based on several of the computer science elective tracks, shown in **Figure 3**.

Test cases are found in the test folder. Viewing the Python (.py) test case file in a programming editor will give the ideal Python GraphViz code needed to generate the ideal graph output. Running the Python file will generate the ideal graph output.

For this report, a case study using the Internet & Enterprise Computing Track will be provided as the example of all the steps of the project.

Figure 4. An ideal output of Internet and Enterprise Computing Track
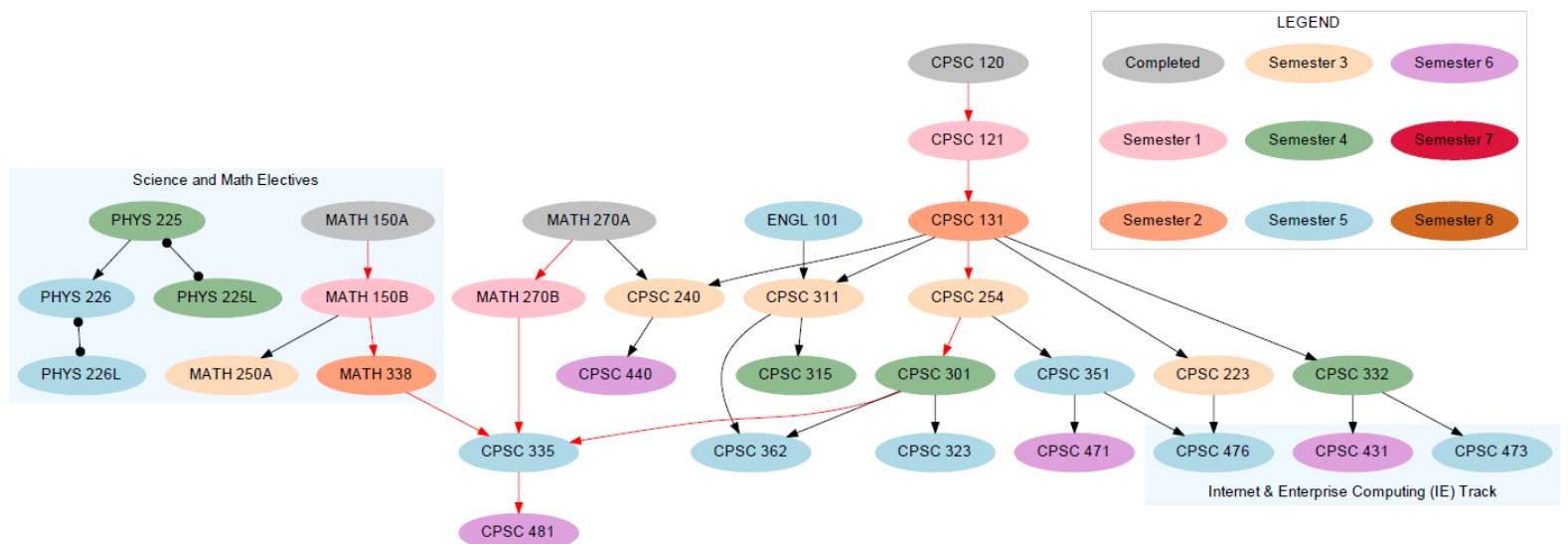
5.2   Results

Phase I – Algorithm

Algorithm Discovery

Tree Implementation

The create_tree(file) method was created to handle generation of the course tree in preparation for the breadth first search algorithm.

By default, data is stored into a Dictionary based on an efficient mapping scheme. This posed a potential issue with the tree and future suggestion algorithm since this program relied heavily on sequential data collection. An OrderedDict from the Python Dictionary class was used to maintain the order of the data.

Pseudocode

Input: File with modified curriculum of user-selected major and elective track
Output: An ordered tree of Node objects

1. Create empty tree as OrderedDict()
2. Create 'ROOT' node in tree
   a. Set its status in studyplan to True by default
3. For each line in file:
   a. Store RegEx matches of line in array (2 values maximum)
   b. If first value is not in tree:
      i. Create a Node object of that value
      ii. Add this node as a child of 'ROOT'
      iii. Set node's units
      iv. If '*' in line:
         1. Set first value as priority
   c. Try block to check if second value exists:
      i. If second value is not in tree:
         1. Create a Node object of that value
      ii. Add first value as parent node of second value
      iii. Add second value as childe of first value
      iv. Set units
      v. If '*' in line:
         1. Set first value as priority
         2. Set second value as priority
   d. Pass if no second value exists
4. Return the tree

Breadth First Search

A breadth first search algorithm was used to handle the first part of the study plan suggestion. Since a curriculum is a directed graph of nodes that have prerequisites to be met before moving on to the next node, breadth first search was a prime candidate for sorting the values in prerequisite order.

In (2), the Deque class from the Queue library was used to handle removing the first of the queue. A list could also have been implemented to handle this operation. Removing the first item in a list causes a lot of computational overhead with shifting values as a consequence for the removed first index. The Deque class was specifically created to handle the same operation more efficiently and thus decided to be used for this project
.

In (3b), the condition to remove a vertex that already exists in the 'visited' list and to append it at the end of the queue is done to remove duplicate nodes caused by a node having multiple prerequisites. If a node appears multiple times in the breadth first search, the preceding values are more likely to not have their prerequisites met. Removing previous entries and adding the latest to the end of the queue, ensures that the node's prerequisites precede it in the visited list.

In (3c), the root is skipped in the BFS result. The root node acts as a placeholder to organize the curriculum into a tree.

Pseudocode

Input: A Node object as the root of a tree
Output: A list of values in breadth first search order

1. Create empty list for visited nodes
2. Create queue as a deque with 'root' as first value of the array
3. While the queue has values:
   a. Pop the first value in the queue and assign it to vertex
   b. If vertex is in visited:
      i. Remove vertex from visited
   c. If vertex is not 'ROOT':
      i. Add vertex to visited
   d. For each value in vertex.children:
      i. Append value to queue
4. Return visited

Suggestion Algorithm

The suggestion algorithm is the final part of the study plan suggestion. It takes the breadth first search results and creates a hash map of semester keys and course list values. It primarily operates as a greedy algorithm by putting priority courses into the study plan at the earliest opportunity to satisfy critical path courses as quickly as possible.

This operation is handled by several conditional checks:

1. *course.priority:* Returns True if the course is priority.
2. *prepreqs_met(course):* Returns True if the course's prerequisite courses exist in the study plan.
3. *prereqs_dne(course, term):* Returns True if none of the course's prerequisites exist in the current term. Returns False if a prerequisite exists in the current term.
4. *in_limit(course_units, unit_count):* Returns True if the course's units are within the unit limit.

A truth table was used to handle all of the different conditional checks. Prior to using a truth table, a lot of duplication and incorrect actions were created during development. Going back to truth tables to handle all of the scenarios cleared up the difficulty of programming the suggestion algorithm as well as greatly improved reusability and readability.

Table 3. Truth Table for Suggestion Algorithm

Action	priority	prereqs_met()	prereqs_dne	in_limit
sp_to_studyplan	T	T	T	T
sp_do_queue	T	T	T	F
sp_do_queue	T	T	F	T
sp_do_queue	T	T	F	F
sp_to_queue	T	F	T	T
sp_to_queue	T	F	T	F
sp_to_queue	T	F	F	T
sp_to_queue	T	F	F	F
sp_to_queue	F	T	T	T
sp_to_queue	F	T	T	F
sp_to_queue	F	T	F	T
sp_to_queue	F	T	F	F
sp_to_queue	F	F	T	T
sp_to_queue	F	F	T	F
sp_to_queue	F	F	F	T
sp_to_queue	F	F	F	F

Conditional Functions

1. *sp_to_studyplan:* Adds the course to the study plan.
2. *sp_do_queue:* Occurs when a priority course cannot be added to the study plan. Items in the queue are populated into the study plan.
    a. *sp_to_newterm:* A new term is created and the priority course is added to it.
3. *sp_to_queue:* All other courses are added to the queue.
4. *sp_end_queue:* If queue is not empty at the end of scanning the breadth first search results, add courses in queue to study plan.

Pseudocode

Input: Breadth first search result list
Output: Hash map with course suggestions by semester

1. Create empty OrderedDict() for study plan
2. Create empty array for taken courses with key 'taken'
3. Create empty list as queue
4. Set term = 1
5. Set unit count = 0
6. Create empty array for study plan hash key 'term'
7. For each node in breadth first search list:
    a. If node is marked as a taken course:
        i. Append to list in study plan hash key 'taken'
        ii. Mark node as in the study plan
    b. Else if node (is priority) and (prerequisites met)   # T T T T
            and (prerequisites not in current term) and (in the unit limit):
        i. Add the node to the study plan hash
        ii. Update unit count
    c. Else if node (is priority) and (prerequisites met)   # T T T F
            and (prerequisites not in current term) and (not in the unit limit):
        i. Run study plan queue function to fill current term with queue values
        ii. Create new term
        iii. Update unit count
    d. Else if node (is priority) and (prerequisites met)   # T T F T
            and (prerequisites is in current term) and (in the unit limit):
        i. Same as c.i-iii
    e. Else if node (is priority) and (prerequisites met)   # T T F F
            and (prerequisites in current term) and (not in the unit limit):
        i. Same as c.i-iii
    f. Else if node (not priority):
        i. Add node to queue
8. For each node in queue:
    a. Do queue function to put all remaining queue values onto hash
9. Return study plan hash map

Algorithm Testing

The following results are from the Internet and Enterprise Computing Track example. These tests results are written onto test_results.txt after each corresponding process. The data collected used for debugging and tracing data to find any bugs in the code. A concise version of the test data is included in the following pages for the major algorithms.

Tree Implementation

The final output of the tree implementation. An asterisk preceding a line indicates that the first course before the (->) is critical path or a priority.

ROOT -> CPSC 120, ENGL 101, MATH 150A, MATH 270A, PHYS 225
* CPSC 120 -> CPSC 121
* CPSC 121 -> CPSC 131
* CPSC 131 -> CPSC 223, CPSC 240, CPSC 254, CPSC 311, CPSC 332
CPSC 223 -> CPSC 476
CPSC 240 -> CPSC 440
* CPSC 254 -> CPSC 301, CPSC 351
CPSC 311 -> CPSC 315, CPSC 362
CPSC 332 -> CPSC 431, CPSC 473
CPSC 440
* CPSC 301 -> CPSC 323, CPSC 335, CPSC 362
CPSC 351 -> CPSC 471, CPSC 476
CPSC 323
* CPSC 335 -> CPSC 481
CPSC 362
CPSC 315
CPSC 481
CPSC 471
ENGL 101 -> CPSC 311
* MATH 150A -> MATH 150B
* MATH 150B -> MATH 338, MATH 250A
* MATH 338 -> CPSC 335
* MATH 270A -> CPSC 240, MATH 270B
* MATH 270B -> CPSC 335
MATH 250A
PHYS 225 -> PHYS 225L, PHYS 226
PHYS 225L
PHYS 226 -> PHYS 226L
PHYS 226L
CPSC 431
CPSC 473
CPSC 476

Breadth First Search

The final output of the breadth first search with extra information on course units, priority and if the course was marked as taken by the user. This was helpful with tracing the suggestion algorithm's operations.

Breadth first search was done by order of its appearance in the curriculum input file and does not take into account course priority.

UNITS	COURSE	PRIORITY	TAKEN
3	CPSC 120	True	False
3	ENGL 101	False	False
4	MATH 150A	True	False
4	MATH 270A	True	False
3	PHYS 225	False	False
3	CPSC 121	True	False
4	MATH 150B	True	False
4	MATH 270B	True	False
1	PHYS 225L	False	False
3	PHYS 226	False	False
3	CPSC 131	True	False
4	MATH 338	True	False
4	MATH 250A	False	False
1	PHYS 226L	False	False
3	CPSC 223	False	False
3	CPSC 240	False	False
3	CPSC 254	True	False
3	CPSC 311	False	False
3	CPSC 332	False	False
3	CPSC 440	False	False
3	CPSC 301	True	False
3	CPSC 351	False	False
3	CPSC 315	False	False
3	CPSC 431	False	False
3	CPSC 473	False	False
3	CPSC 323	False	False
3	CPSC 335	True	False
3	CPSC 362	False	False
3	CPSC 471	False	False
3	CPSC 476	False	False
3	CPSC 481	False	False

Suggestion Algorithm

Final output from suggestion algorithm. Previously a complete trace of the suggestion algorithm which followed every operation node by node.

    taken ->
    1 -> CPSC 120, MATH 150A, MATH 270A, ENGL 101
    2 -> CPSC 121, MATH 150B, MATH 270B, PHYS 225, PHYS 225L
    3 -> CPSC 131, MATH 338, PHYS 226, MATH 250A, PHYS 226L
    4 -> CPSC 254, CPSC 223, CPSC 240, CPSC 311, CPSC 332
    5 -> CPSC 301, CPSC 440, CPSC 351, CPSC 315, CPSC 431
    6 -> CPSC 335, CPSC 473, CPSC 323, CPSC 362, CPSC 471
    7 -> CPSC 476, CPSC 481

Same output with course units in vertical format.

    cluster: taken   3 CPSC 254
                                      3 CPSC 223
    cluster: 1   3 CPSC 240
    3 CPSC 120   3 CPSC 311
    4 MATH 150A   3 CPSC 332
    4 MATH 270A
    3 ENGL 101   cluster: 5
                                      3 CPSC 301
    cluster: 2   3 CPSC 440
    3 CPSC 121   3 CPSC 351
    4 MATH 150B   3 CPSC 315
    4 MATH 270B   3 CPSC 431
    3 PHYS 225
    1 PHYS 225L   cluster: 6
                                      3 CPSC 335
    cluster: 3   3 CPSC 473
    3 CPSC 131   3 CPSC 323
    4 MATH 338   3 CPSC 362
    3 PHYS 226   3 CPSC 471
    4 MATH 250A
    1 PHYS 226L   cluster: 7
                                      3 CPSC 476
    cluster: 4   3 CPSC 481

Phase II – Development

The Development Phase follows the Algorithm Phase. This phase includes common software engineering development activities of software design, software development and quality assurance testing.

With the time constraint limiting the entire project to the duration of spring semester, the development process loosely followed the Agile methodology using SCRUM. The project was time-boxed into 2-week sprints beginning in January. Each sprint included a meeting with the project advisor to discuss the project progress and if the sprint's goals are reasonably appointed. The proceeding meeting followed up on the project's progress, demonstrated any working samples from the previous sprint's development and received guidance from the advisor.

Since the project was developed by one person, there were no formal daily standup meetings. Daily progress was recorded internally as reference for the developer. Testing was done individually with bug fixes as part of the development phase.

The development phase resulted in the deliverable of the completed project as a command line application.

The following pages demonstrate the resulting graphical outputs created by the Curriculum Graph Visualizer.

Development Results

Example 1

- Elective: Internet and Enterprise Computing
- Taken: None

Figure 5. Graph visualization of Internet and Enterprise Computing Track using CGV
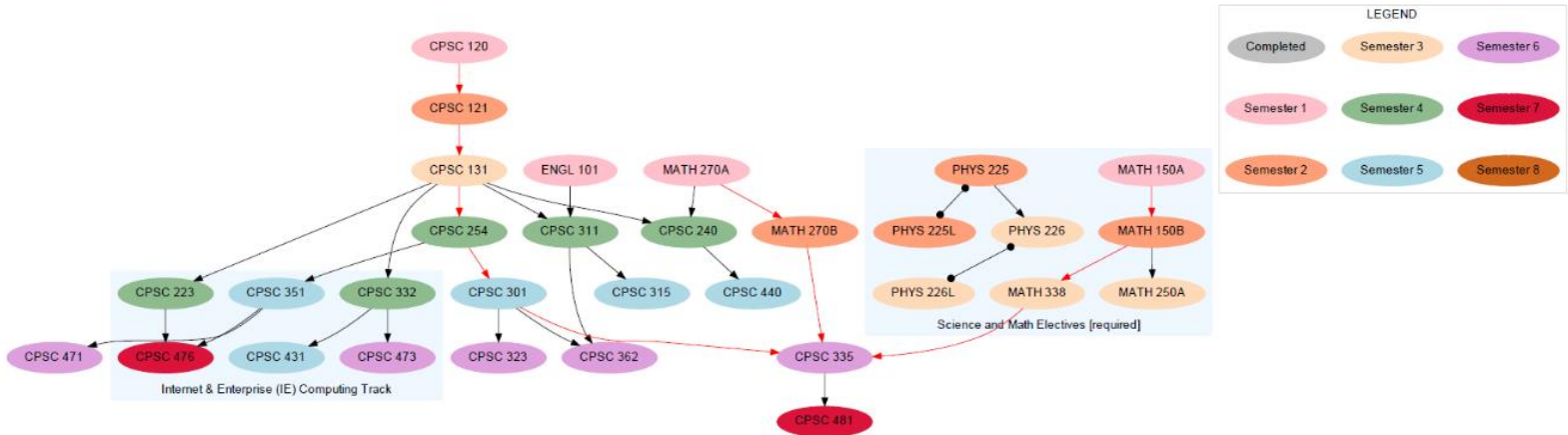
Example 2

- Elective: Internet and Enterprise Computing
- Taken Courses: MATH 150A, MATH 150B, ENGL 101, CPSC 120

Figure 6. Example of IE Track with courses marked as Taken.
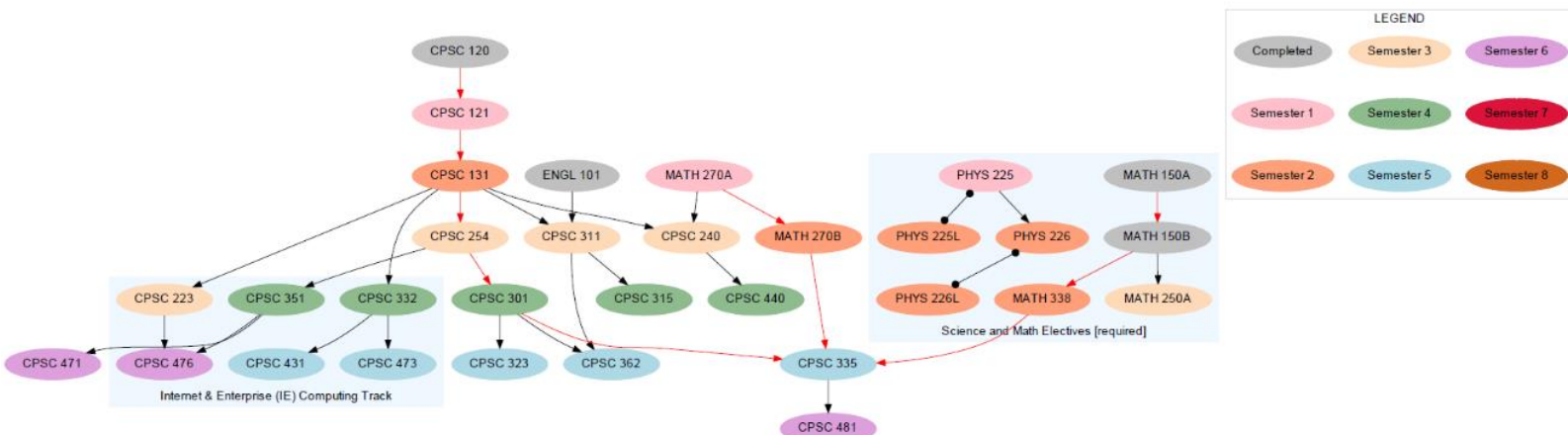
Example 3

- ▪ Elective: Scientific Computing Track
- ▪ Taken Courses: None

Figure 7. Scientific Computing track with no taken courses.
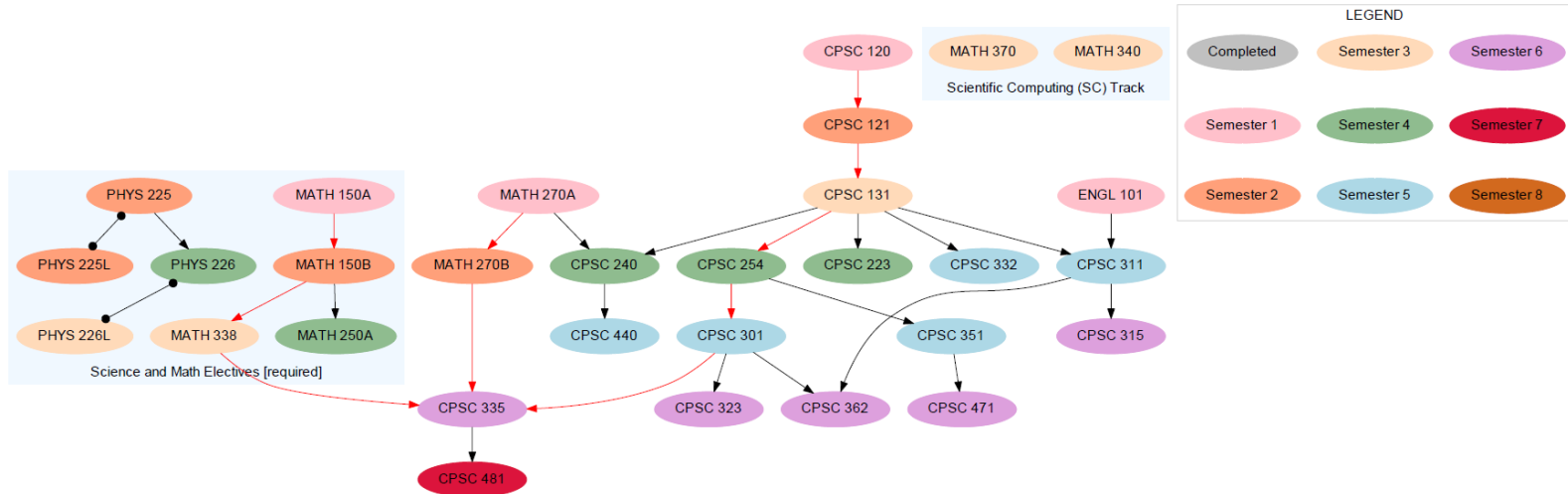
There is an error in **Figure 7** in the elective track courses for MATH 370 and MATH 340, which are not connected to the graph because a prerequisite for these courses were not indicated in curr.txt. Adding the prerequisite edge to MATH 338 causes a cluster conflict with MATH 338. Nodes in the Sciences and Math Electives and Scientific Computing Tracks are grouped into their own separated encapsulated Digraph object clusters using Python GraphViz. Adding a prerequisite edge to courses in a cluster to a course that already exists in a different cluster causes a conflict. This results in a runtime error since a course cannot exist in more than one cluster.

Potential solutions to this problem include:

1. Adding a list at the end of curr.txt which addresses the missing prerequisite edges, categorized by track.

2. Removing the Digraph cluster that is created for Required Electives. This results in one Digraph cluster only for the user's chosen Elective Track courses.

Example 4

- ▪ Elective: Scientific Computing Track
- ▪ Taken Courses: PHYS 225, MATH 270A, MATH 270B

Figure 8. Scientific Computing track with three courses taken
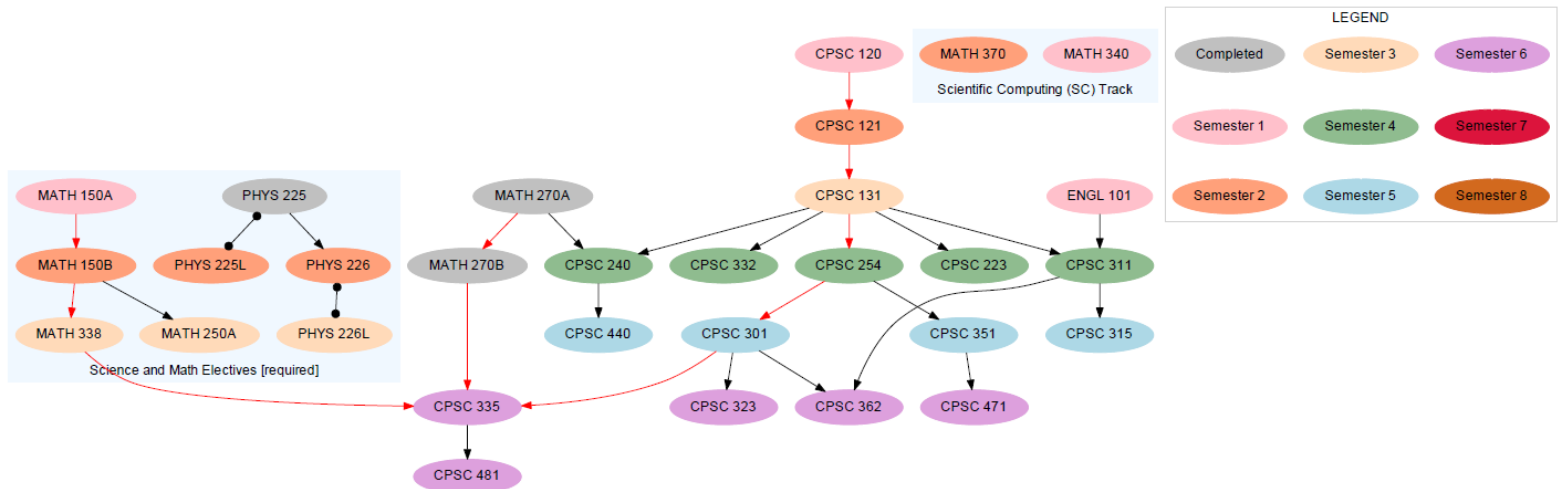
6 Installation Instructions

1. Download CGV package from GitHub.

Figure 9. CGV download on GitHub
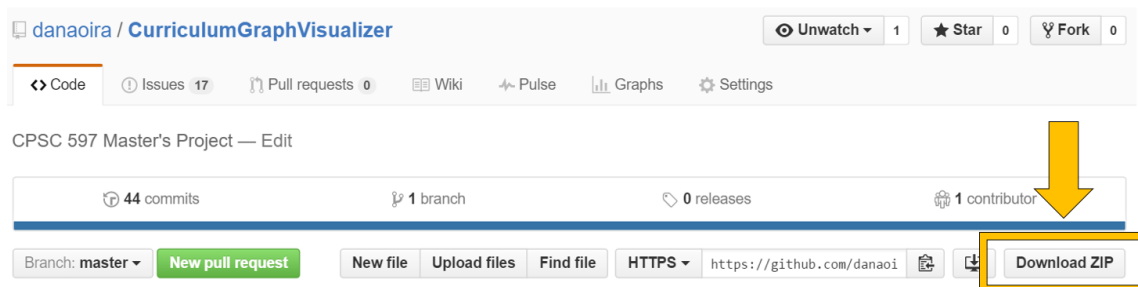
2. Extract CGV files into any folder.
3. Install Python.
4. Install GraphViz.

7 Operating Instructions

Creating a Curriculum Input File (CIF)

In the CGV folder, there will be a file called 'curr.txt'. This file will detail the formatting instructions a user must use in order for the CGV to work.

The CGV can read in any plain text file format as the CIF document.

Execute CGV to Generate a Study Plan

1. Create a CIF.
2. Open a command prompt.
   a. Press Windows key + R and type 'cmd'.

Figure 10. Run command prompt
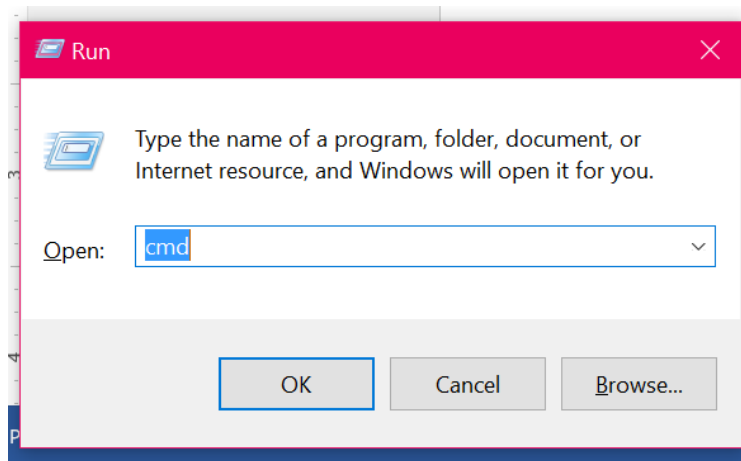
Figure 11. Successful command prompt launch
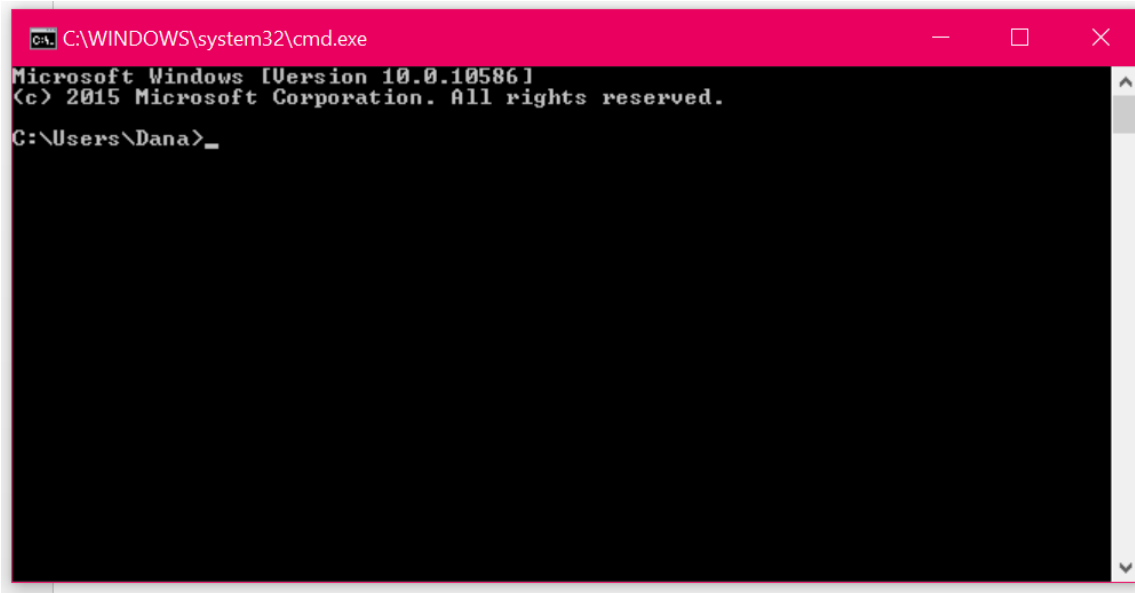
3. In console window, change directory to CGV directory location.
    a. Input string: cd [DIRECTORY PATH]
    b. Press Enter.

Figure 12. Successful directory change
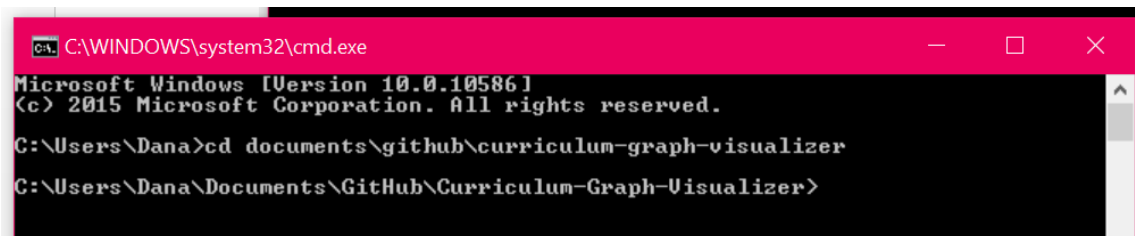
4. Type CGV execution command in console.
    a. Format: `python cgv.py [CIF file]`
        i. `python`: Command to run the file using Python.
        ii. `cgv.py`: Runs the CGV file.
        iii. `[CIF File]`: The input file for the course curriculum.
            1. For convenience, save CIF is in the same directory as `cgv.py`.

Figure 13. Run command for CGV in console
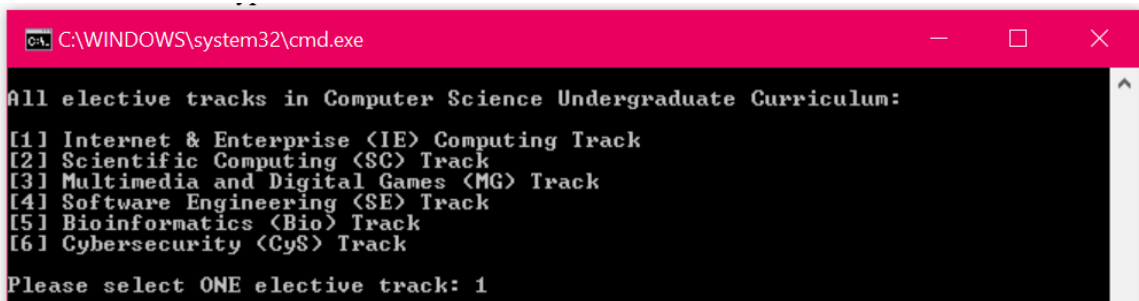
5. Select one (1) elective track.

Figure 14. Elective track selection for CGV in console menu
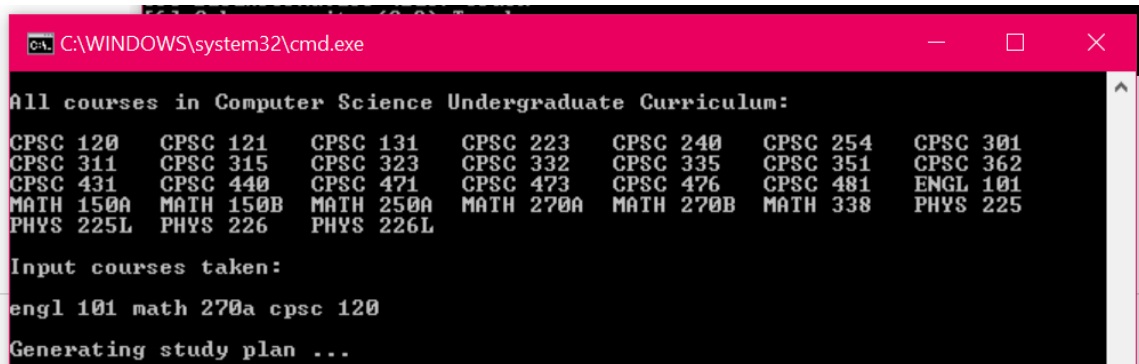
6. Input completed courses.
   a. Acceptable inputs:
       i. *Comma separated list:* CPSC 121, CPSC 131, MATH 120
      ii. *Single-line input*

Figure 15. User input of completed/taken courses in CGV

7. Press Enter twice to generate the study plan.

8 Recommendations for Enhancements

1. *Graphical User Interface:* A graphical interface to take the place of the command line program will give a user friendly experience.

2. *Web Application:* Creating the program as a web application will allow wider and easier access for users.

3. *File upload for CIF:* Having a file upload option instead of a command line input will simplify CIF import to the program.

4. *CIF tutorial GUI:* A tutorial on writing the CIF will help users in how to understand the formatting for the CIF.

5. *Weighted system for course load difficulty:* Allow a weighted scale (ex. 1-10 scale) for course level difficulty for each course and make suggestions based on a maximum number of difficulty. This will help to create suggestions that don't result in study plans where many courses with difficult course loads appear in the same term.

6. *Further refine study plan to GraphViz Python translation.* A majority of the cgc.py operations are hardcoded and can be improved with automation.

7. *Combine cgc.py and studyplan.py.* Since cgc.py is essentially a translator, it may help to remove the generation of studyplan.py and have all its operations done in cgc.py.

9 Acknowledgement

I would like to thank my mentors, Dr. Kevin Wortman (Advisor) and Dr. Kent Palmer (Reviewer), for the guidance and inspiration for the proposal portion of this project as well as the continued collaboration during the project's creation in the upcoming semester.

10 GitHub Repository

This project can be viewed on its GitHub repository which includes all created files, proposals and final deliverables for the CPSC 597 Project course.

For more information, visit:

https://github.com/danaoira/curriculumgraphvisualizer

11 Bibliography

[1] Agrawal, V.K.; Earnest, J.; Gupta, S.K.; Tegar, J.P.; Mathew, S.S., "Outcome based engineering diploma curriculum - 2012 Gujarat experiment," in *Frontiers in Education Conference, 2013 IEEE* , vol., no., pp.1864-1870, 23-26 Oct. 2013

[2] Azlan, A.; Hussin, N.M., "Implementing graph coloring heuristic in construction phase of curriculum-based course timetabling problem," in *Computers & Informatics (ISCI), 2013 IEEE Symposium on* , vol., no., pp.25-29, 7-9 April 2013

[3] Gansner, E.R.; Koutsofios E.; North S.C.; Vo K., "A Technique for Drawing Directed Graphs," in *IEEE Transactions on Software Engineering, 1993, vol. 19, no. 3, pp. 214 -230, March 1993*

[4] Gestwicki, P., "Work in progress - curriculum visualization," in *Frontiers in Education Conference, 2008. FIE 2008. 38th Annual* , vol., no., pp.T3E-13-T3E-14, 22-25 Oct. 2008

[5] Hosobe, H., "Analysis of a high-dimensional approach to interactive graph drawing," in *Visualization, 2007. APVIS '07. 2007 6th International Asia-Pacific Symposium on* , vol., no., pp.93-96, 5-7 Feb. 2007

[6] Kabicher, S.; Motschnig-Pitrik, R., "Coordinating Curriculum Implementation Using Wiki-supported Graph Visualization," in *Advanced Learning Technologies, 2009. ICALT 2009. Ninth IEEE International Conference* on , vol., no., pp.742-743, 15-17 July 2009

[7] McCreary, C.L.; Chapman, R.O.; Shieh, F.-S., "Using graph parsing for automatic graph drawing," in *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on* , vol.28, no.5, pp.545-561, Sep 1998

[8] Rodgers, P.J., "A graph rewriting programming language for graph drawing," in Visual Languages, 1998. Proceedings. 1998 IEEE Symposium on , vol., no., pp.32-39, 1-4 Sep 1998

[9] Rusu, A.; Crowell, A.; Petzinger, Bryan; Fabian, A., "PieVis: Interactive Graph Visualization Using a Rings-Based Tree Drawing Algorithm for Children and Crust Display for Parents," in *Information Visualisation (IV), 2011 15th International Conference on*, vol., no., pp.465-470, 13-15 July 2011

[10] Slim, A.; Heileman, G.L.; Kozlick, J.; Abdallah, C.T., "Employing Markov Networks on Curriculum Graphs to Predict Student Performance," in *Machine Learning and Applications (ICMLA), 2014 13th International Conference on* , vol., no., pp.415-418, 3-6 Dec. 2014