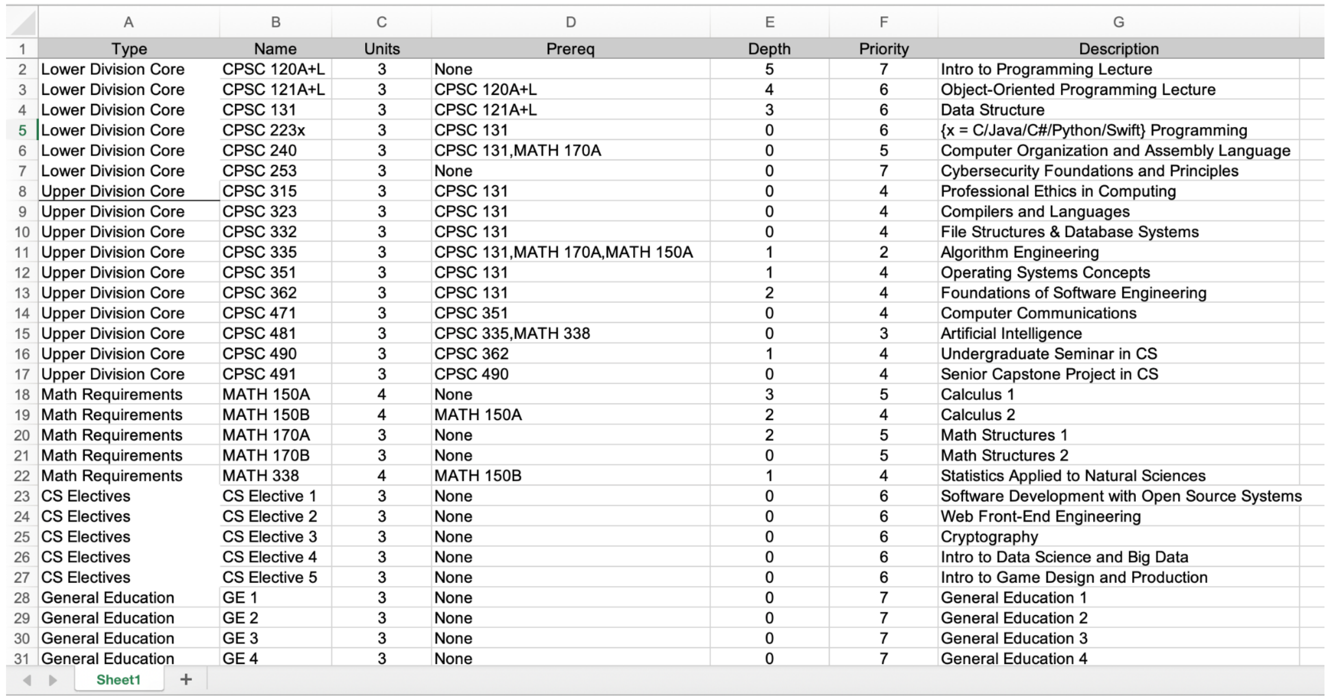
Project 1
CPSC 481-05
Fall Semester

Samad Ahmad
Mike Ball
Bryan De Los Santos
Randall Frye
Lillian Hoang
Rosa Silva

GITHUB REPO LINK:https://github.com/grafcar/CPSC-481-Project-1

Collect Data

In order to work on the project, we must look into the courses that are required for computer science majors. We collected the list of courses from the PDF file called "CS Core Courses" on the Department of Computer Science website. The CS Core Courses file contains a list of "Lower Division", "Upper Division", "Math Requirements", and "CS Electives" courses. We created an Excel spreadsheet to input the courses from the PDF File. In the spreadsheet, we created 7 different columns: Type, Name, Units, Prerequisites, Depth, Priority (we didn't end up using this one at all), and Description. Then, we entered the information from each course into their respective columns. Units and prerequisites were taken from the 2023-2024 University Catalog for Computer Science B.S and the CS Course Plan.



	A	B	C	D	E	F	G
1	Type	Name	Units	Prereq	Depth	Priority	Description
2	Lower Division Core	CPSC 120A+L	3	None	5	7	Intro to Programming Lecture
3	Lower Division Core	CPSC 121A+L	3	CPSC 120A+L	4	6	Object-Oriented Programming Lecture
4	Lower Division Core	CPSC 131	3	CPSC 121A+L	3	6	Data Structure
5	Lower Division Core	CPSC 223x	3	CPSC 131	0	6	{x = C/Java/C#/Python/Swift} Programming
6	Lower Division Core	CPSC 240	3	CPSC 131,MATH 170A	0	5	Computer Organization and Assembly Language
7	Lower Division Core	CPSC 253	3	None	0	7	Cybersecurity Foundations and Principles
8	Upper Division Core	CPSC 315	3	CPSC 131	0	4	Professional Ethics in Computing
9	Upper Division Core	CPSC 323	3	CPSC 131	0	4	Compilers and Languages
10	Upper Division Core	CPSC 332	3	CPSC 131	0	4	File Structures & Database Systems
11	Upper Division Core	CPSC 335	3	CPSC 131,MATH 170A,MATH 150A	1	2	Algorithm Engineering
12	Upper Division Core	CPSC 351	3	CPSC 131	1	4	Operating Systems Concepts
13	Upper Division Core	CPSC 362	3	CPSC 131	2	4	Foundations of Software Engineering
14	Upper Division Core	CPSC 471	3	CPSC 351	0	4	Computer Communications
15	Upper Division Core	CPSC 481	3	CPSC 335,MATH 338	0	3	Artificial Intelligence
16	Upper Division Core	CPSC 490	3	CPSC 362	1	4	Undergraduate Seminar in CS
17	Upper Division Core	CPSC 491	3	CPSC 490	0	4	Senior Capstone Project in CS
18	Math Requirements	MATH 150A	4	None	3	5	Calculus 1
19	Math Requirements	MATH 150B	4	MATH 150A	2	4	Calculus 2
20	Math Requirements	MATH 170A	3	None	2	5	Math Structures 1
21	Math Requirements	MATH 170B	3	None	0	5	Math Structures 2
22	Math Requirements	MATH 338	4	MATH 150B	1	4	Statistics Applied to Natural Sciences
23	CS Electives	CS Elective 1	3	None	0	6	Software Development with Open Source Systems
24	CS Electives	CS Elective 2	3	None	0	6	Web Front-End Engineering
25	CS Electives	CS Elective 3	3	None	0	6	Cryptography
26	CS Electives	CS Elective 4	3	None	0	6	Intro to Data Science and Big Data
27	CS Electives	CS Elective 5	3	None	0	6	Intro to Game Design and Production
28	General Education	GE 1	3	None	0	7	General Education 1
29	General Education	GE 2	3	None	0	7	General Education 2
30	General Education	GE 3	3	None	0	7	General Education 3
31	General Education	GE 4	3	None	0	7	General Education 4

Note: GE = General Education, SME = Science/Math Electives, GRE = Graduation Requirement

Formulate the Problem

We will implement an A* search algorithm that returns the shortest path to graduation, grouped by semesters.

1) **Initial state:** a set of courses that the student has already completed successfully (for first-time freshmen, this set may be empty).
2) **Actions:** the program considers the actions of taking a set of available courses within a semester, ensuring that the number of units does not exceed 17. These courses are the following types: “Lower Division Core,” “Upper Division Core,” “Math Requirements,” “CS Electives,” “General Education,” and “Graduation Requirement”. The program also keeps track of the number of units required for each type of course:
    a) Core CS course = 66 units
    a) CS Electives = 15 units
    b) Science/Math Electives = 12 units
    c) General Education = 24 units
    d) Graduation Requirement = 3 units Total = 120 units

Note that General Education Courses are not specified, but will reserve a spot within the student’s schedule, especially since taking only Computer Science classes is unrealistic.

3) **Transition model:** the resulting state will be the set of courses a student will take within a single semester.
4) **Goal state:** the completion of all required courses (120 units).
5) **Action Cost:** the number of units associated with a course.

Define heuristics

1) **Depth heuristic:** depth refers to the number of classes a particular course serves as a prerequisite for, assigning a higher priority to courses that are the beginning prerequisite. For example, CPSC 120A is a prerequisite to 5 other courses, meaning it has a depth value = 5.
2) **Semester Size heuristic:** semester size prioritizes semesters that are as close to 17 units as possible
3) **Goal Heuristic:** checks if the current state matches the goal state. If it does, it assigns a high negative value of -100, thus favoring this search. Otherwise it returns a 0, indicating that the goal has not been reached and that there's no advantage to this state.
4) **Balance heuristic:** the balance heuristic should balance the number of CS Core courses with other courses. If the number of units of CS Core Courses is less than 12, the cost decreases by 3, otherwise, it increases by 3.
5) **Distance from graduation heuristic:** the distance refers to the number of classes left per type of course.

We combine these five heuristics and then add them to the actual cost of the path so far, resulting in the estimated cost of the best path that continues from node n to the goal:

$$\mathbf{f(n) = g(n) + h(n)}$$

This function is the foundation of A* search. Unlike greedy best-first search, where f(n) = h(n), A* search always provides a cost-optimal solution given that it's an admissible heuristic. In other words, as long as we do not overestimate the cost of reaching the goal, we will find the shortest path to graduation.

Implement the A* search

This program creates all possible combinations of classes to take for a single semester, puts them in a priority queue, and then chooses the semester node with the best/lowest f-value from that priority queue. The program then expands that chosen semester node, creates all possible combinations for the next semester, puts them in the same original priority queue, and then selects the best node from the priority queue again. This process repeats until we've reached graduation.

1) Initialize the priority queue and store the states within it, assigning the highest priority to states with the lowest f-value (Recall: $f(n) = g(n) + h(n)$).
2) Define the initial state and add it to the priority queue.
3) While the priority queue is not empty:
    a) Pop the state with the lowest f-value from the priority queue
    b) Check if it's the goal state. If it is, we've found the shortest path to graduation and will stop the search. Otherwise, we expand the current state and generate all possible combinations of courses for the next semester. These are known as the child states.
    c) We calculate the g-cost (actual cost of the path so far) and h-cost (heuristic estimate) for each child state.
    d) We add these child states back into the priority queue along with their associated f-values.
    e) Continue doing this until we reach the goal state.

History of the Project
We started on Wednesday, September 20th. Mike, Bryan, Samad, and temporarily Randall worked on coding the search while Lillian, Rosa, and later on, Randall focused on the documentation. We programmed in Python and uploaded our code to the Github Repository, using the data of courses within the Excel spreadsheet. We began working on our own branches before combining our findings together.

When defining the possible heuristics, we thought of multiple possibilities:
- **Heuristic #1:** Does this semester plan get me closer to graduation? (The closer to graduation, the higher the heuristic)
- **Heuristic #2:** Does this semester plan have too many or too few CPSC Core courses? (If within a reasonable number, the heuristic is higher)
- **Heuristic #3:** Does this semester plan optimize courses that are in a long chain of prerequisites? (the longer the chain, the higher the heuristic)
- **Heuristic #5:** Does this semester have too many difficult classes? (Rank classes from 1 to 5 based on predictions and experiences)
- **Heuristic #6:** Does this semester plan emphasize Upper Core and Math Courses above others? (Upper Core, Math = +1, CS Electives = +2, Lower Core, GE, Science/Math Electives = +3)

In the end, we decided on the depth, semester size, balance, and distance from graduation as our heuristics.
Bryan used a site called Notion so that we could organize our notes and view resources from there. We tried to understand the A* search by watching videos like "A* (A Star) Search Algorithm" by Computerphile or by reading blogs such as "Implementing the A* Algorithm in Python: A Step-by-Step Guide" by SaturnCloud, but we still struggled

nonetheless. In the beginning, we thought we had to create our own A* search, but then we discovered the “class Problem” and “class Node” framework in the textbook’s GitHub repository. From there, we all created our own branches on Github, some working on their own branch while others pulled from Mike’s branch to adapt to his code. Since we were running low on time, Bryan took Mike’s code and completely rewrote it into a simple stripped-down version. This version’s goal was for the student to complete 120 units regardless of the classes’ type or difficulty, with the only condition being that the student could not take a class if it had prerequisites. This was to ensure we had something to submit in the event that the main code, Mike’s, turned out to not work. In the end, we were able to finish the main code and submitted it.

Encountered Issues:
There was another problem where one class would not show up as an action that could be taken. This issue stemmed from the leading space in front of class names within the Excel spreadsheet. Unfortunately, that caused about an hour and a half of headaches before it was finally solved. At first, we were adding individual classes to our priority queues, until we found out that our priority queues had to have a list of classes instead. This meant we had to replace our action and transition model, which used to be the action of taking an individual class to get the resulting state, to now taking a list of classes to produce all possible combinations of classes within a semester. Finally, we were able to print out the list of classes taken in the first semester, but grappled with printing out the remaining semesters. This is most likely due to having problems expanding the priority queue while making sure to include the previous nodes. So we tried concatenating the old and new priority queues together, but we couldn’t do this because they would not be sorted properly. Then we tried to merge the priority queues together instead.

Instructions to run the code: Read the README.file

General Structure of Program:

We created multiple files:

- Superclass.py
- Graduation_Subclass.py
- Main.py
- Courses_to_units.py
- download_data.py

Superclass.py:
# This file defines the abstract class Problem and class Node.

- **Problem class:**
    - **__init__(self, initial, goal, kwds**):** a constructor to create an instance of the class as "Problem(object)". The variable "initial" specifies the initial state and the "goal" specifies the goal state.
    - **actions(self, state):** returns all the possible actions the program can execute in the given "state".
    - **result(self, state, action):** returns the resulting state if "action" is taken in the "state".
    - **is_goal(self, state):** returns "True" if it is a goal state, "False" otherwise.
    - **action_cost(self, s, a, s1):** returns the cost of the path that arrives at state "s1" as a result of taking action "a" from state "s".
    - **h(self, node):** returns the heuristic estimate of this node.
    - **__str__(self):** returns the object as a string.

These methods described in the Problem class will be overridden in "Graduation_Subclass.py"

- **Node class:**
    - **__init__(self, total_courses, state_courses, g_value, h_value, parent=None):** creates a node. "Total_courses" is the total number of courses. "State_courses" is the course(s) contained within this node. "G_value" is the cost to reach the current node from the root node. "H_value" is the heuristic estimate associated with each node.

The remaining 3 methods override standards Python functionality for representing an object as a string:

- **__repr__(self):** returns the state of this node.
- **__len__(self):** returns the depth of the node.
- **__lt__(self, other):** defines comparison based on the f-value of two nodes.

This file also defines two other methods outside the classes:

- **expand(problem, node):** expands a node and generates all the children nodes.
- **path_actions(node):** returns the sequence of actions to get to this node.

Graduation_Subclass.py:
# Inherits from the Problem class and overrides their methods.

- **__init__(self, initial, goal, prerequisites, course_units, course_type, course_depth):** constructs a problem object and defines its attributes.
- **actions(self, state):** generates all possible actions of courses to take, as long as they're not already taken and all prerequisites are satisfied.
- **possible_semesters(self, actions):** returns all possible combinations of courses in a semester, given possible courses to take. A semester must have at

least 12 units but no more than 17. The semesters will be represented as a list of lists of courses.

- **result(self, state, action):** returns the resulting state from taking a course.
- **is_goal(self, state):** returns “True” if the total number of units taken is 120.
- **action_cost(self, a):** returns the number of units in this course
- **depth_heuristic(self, semester):** returns the number of courses this course serves as a prerequisite for.
- **semester_size_heuristic(self, semester):** semester size prioritizes semesters that are as close to 17 units as possible
- **goal_heuristic(self, courses_so_far):** returns -100 if the length of the courses taken so far equals the length of the goal state. Otherwise, it returns a 0.
- **balance_heuristic_short_term(self, semester):** checks for balance within the semester
- **distance_from_graduation(self, total_courses):** returns the remaining classes left per course type.

Main.py:

#This runs the program.

- **initial_state_courses _new_student:** empty list of courses initially taken by a new student. It is used to test the program as we worked on it. It would also be used as the initial state when planning a student with no classes.
- After creating a complete program, we added **transfer_student_course_load** and **mike_ball_course_load**, which fulfill the requirements for the other 2 initial course loads
- **initial_state_unit_dict:** dictionary that stores the initial state of the amount of units for each type of classes a new student will have.
- **final_state_units_dict:** dictionary that stores the number of units each type of classes must reach before a student can graduate.
- **final_state_courses:** variable initialized with download_data’s course_names_list.
- **problem:** variable where GraduationPathProblem from Graduation_Subclass.py is initialized, taking initial_state_courses _new_student, final_state_courses, prereq_dict, units_dict, type_dict as inputs.
- **currentNode:** node the program is currently at. It is initialized as a node using the Node class from Superclass.py taking in initial_state_courses_new_student, and initial_state_courses_new_student as inputs. All other values are either set to 0 or none.
- **actions:** variable that calls the problem variable’s actions function, taking currentNode’s state_courses as an input

- **expanded_nodes:** variable that calls the expand function from superclass.py that takes problem and currentNode as inputs.
- **for node in expanded node:** prints out possible semester courses, f-value, path-cost, and parent of node.
- **priorityQueue:** function that creates a priority queue. It takes expanded_nodes as an input and uses heappush to put a node into a heap called priority_queue. The key for each node is (node.g_value + node.h_value, node). It returns the priority_queue heap ranking the classes that need to be taken.
- **priority_queue:** list in priorityQueue function that is turned into a heap. The output that is returned from the priorityQueue function.
- **next_semester:** variable that takes the class with lowest priority from a heap created by the priorityQueue function.

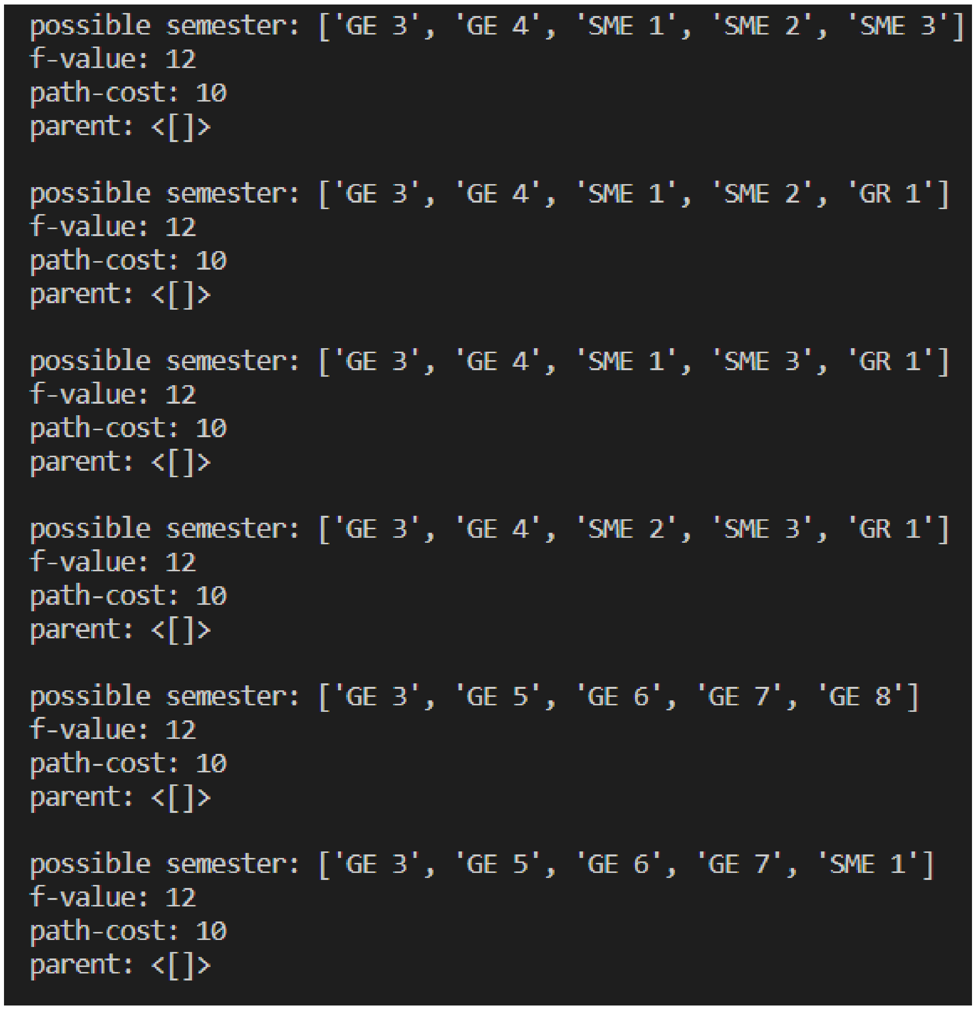
Courses_to_units.py:
#Basic Unit Category Heuristic
From **download_data.py**, **units_dict** & **type_dict** are imported

- **courses_to_units(courses):** function that takes courses as an argument.
    - **Unit_dict :** dictionary inside the function that has five keys: “CS Core Courses”, “CS Electives”, “Science/Math Electives”, “General Education”, and “Graduation Requirements”. The keys are set to 0 and each key tracks the number of units.
    - **For course in courses:** For loop that loops through every item in the list courses
        - Inside the loop, type_dict is used to determine the type of each course. Depending on the type, it updates the correct value in the unit_dict. If it's not recognized, it prints out an error message.
    - Function returns unit_dict, which contains the total number of units for each category.
- **Subtract_unit_dicts:** A function that takes dict1 & dict2 as arguments. Performs subtraction between the corresponding values in these dictionaries. Dist1 should be the larger dictionary
    - **Unit_dict**: dictionary inside the function that has five keys: “CS Core Courses”, “CS Electives”, “Science/Math Electives”, “General Education”, and “Graduation Requirements”. The keys are set to 0 and each key tracks the number of units.
    - Calculates the difference between the corresponding values in **dict1** & **dict2** for each course category and stores it in unit_dict
    - Function returns **unit_dict** with the differences between the two dictionaries

Download_data.py:
#Loads information from Excel Spreadsheet
Imports pd from the pandas library - Used for working with data sets
Imports pprints - A module in python that is used to print data structures in a readable way

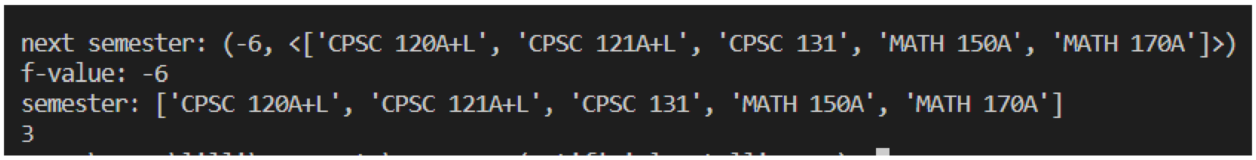
- **x1** variable loads the spreadsheet into the file
  - x1 = pd.ExcelFile('./data.xlsx')
- **df** & **df** variables load a single sheet from the Excel spreadsheet into the Dataframe by its name
  - df = x1.parse('Sheet1')
  - df = df.fillna('None')
- **Prereq_dict** variable creates a dictionary for the prerequisites of each course
  - Prepreq_dict = df.set_index('Name')['Prereq'].str.split(',').to_dict()
- **Units_dict, type_dict, & depth_dict** variables create a dictionary for each of the units
  - units_dict = df.set_index('Name')['Units'].to_dict()
  - type_dict = df.set_index('Name')['Type'].to_dict()
  - depth_dict = df.set_index('Name')['Depth'].to_dict()
- **pp** variable specifies the amount of indentation added
  - pp = pprint.PrettyPrinter(indent=4)
- **Course_names_list** variable creates a list and returns the preprq_dict keys
  - Course_names_list = list(prereq_dict.keys())

Demonstrate the Results:



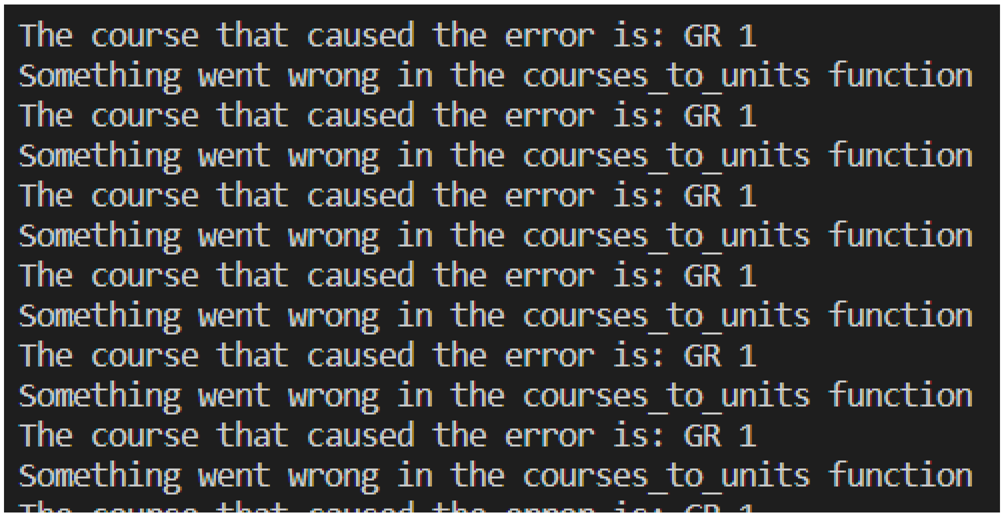
```
possible semester: ['GE 3', 'GE 4', 'SME 1', 'SME 2', 'SME 3']
f-value: 12
path-cost: 10
parent: <[]>

possible semester: ['GE 3', 'GE 4', 'SME 1', 'SME 2', 'GR 1']
f-value: 12
path-cost: 10
parent: <[]>

possible semester: ['GE 3', 'GE 4', 'SME 1', 'SME 3', 'GR 1']
f-value: 12
path-cost: 10
parent: <[]>

possible semester: ['GE 3', 'GE 4', 'SME 2', 'SME 3', 'GR 1']
f-value: 12
path-cost: 10
parent: <[]>

possible semester: ['GE 3', 'GE 5', 'GE 6', 'GE 7', 'GE 8']
f-value: 12
path-cost: 10
parent: <[]>

possible semester: ['GE 3', 'GE 5', 'GE 6', 'GE 7', 'SME 1']
f-value: 12
path-cost: 10
parent: <[]>
```

(Picture from an older version of the code; not the current one)
You can see above that we had multiple lists of courses full of GEs, SMEs, and GR. This printout showed the list of all possible semester combinations, barring those that had more than 8 courses, and had more than 17 units.



```
next semester: (-6, <['CPSC 120A+L', 'CPSC 121A+L', 'CPSC 131', 'MATH 150A', 'MATH 170A']>)
f-value: -6
semester: ['CPSC 120A+L', 'CPSC 121A+L', 'CPSC 131', 'MATH 150A', 'MATH 170A']
3
```

(Picture from an older version of the code; not the current one)
The above picture shows the best node chosen from the priority queue, containing the lowest f-value of 6. Our algorithm stopped at one semester, and we struggled with getting the remaining semesters.

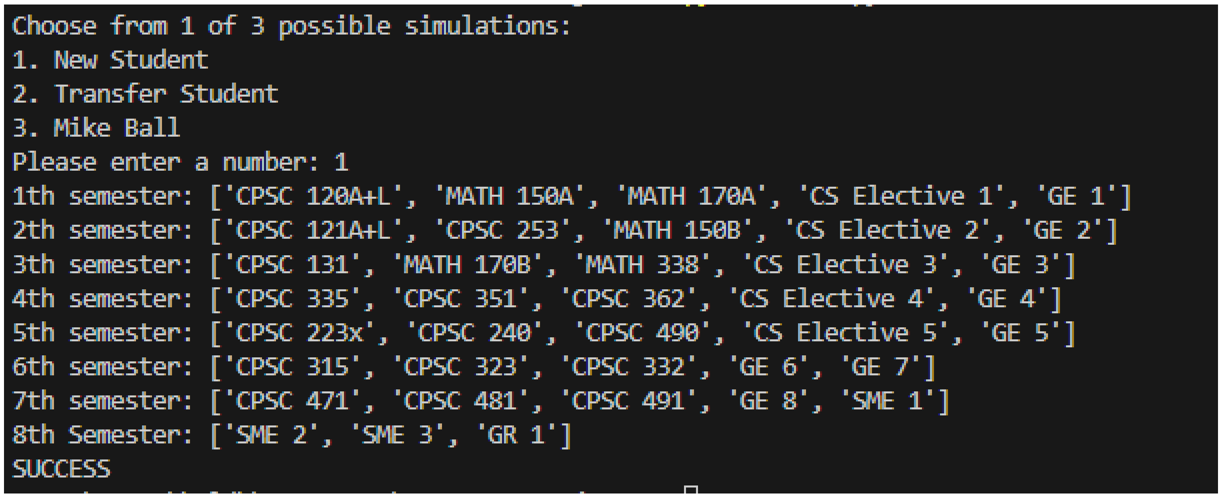


```
The course that caused the error is: GR 1
Something went wrong in the courses_to_units function
The course that caused the error is: GR 1
Something went wrong in the courses_to_units function
The course that caused the error is: GR 1
Something went wrong in the courses_to_units function
The course that caused the error is: GR 1
Something went wrong in the courses_to_units function
The course that caused the error is: GR 1
Something went wrong in the courses_to_units function
The course that caused the error is: GR 1
Something went wrong in the courses_to_units function
```

(Picture from an older version of the code; not the current one)
GR 1 had a leading space in its type column, meaning that the courses_to_units function did not recognize it. This was very frustrating, and I did not learn my lesson for the second time around this issue occurred.

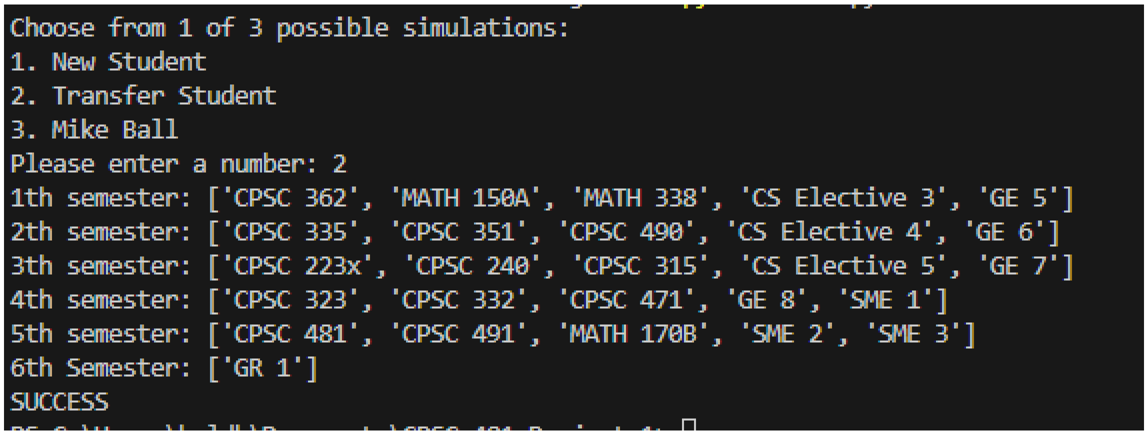
One of the final issues we ran into was an issue where the code kept looping forever. We implemented a band-aid solution, in which if the number of possible actions was less than 5, it would automatically take the remaining courses and put them into a node for the final semester.

Here is the final output:
Plan for a student who has taken no classes.



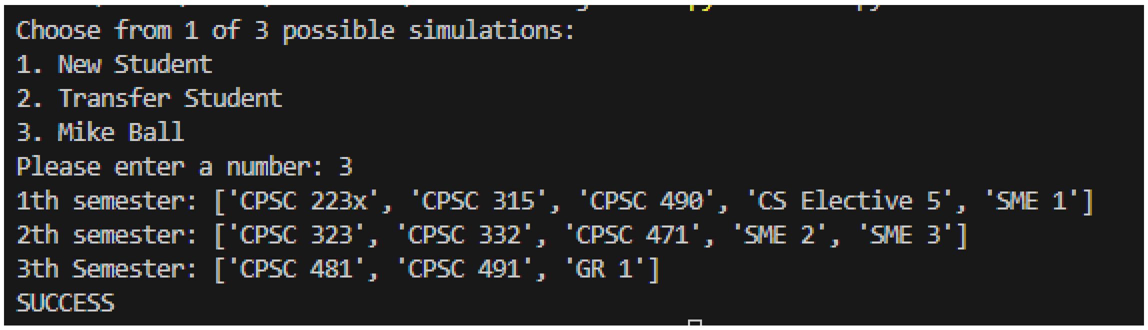
```
Choose from 1 of 3 possible simulations:
1. New Student
2. Transfer Student
3. Mike Ball
Please enter a number: 1
1th semester: ['CPSC 120A+L', 'MATH 150A', 'MATH 170A', 'CS Elective 1', 'GE 1']
2th semester: ['CPSC 121A+L', 'CPSC 253', 'MATH 150B', 'CS Elective 2', 'GE 2']
3th semester: ['CPSC 131', 'MATH 170B', 'MATH 338', 'CS Elective 3', 'GE 3']
4th semester: ['CPSC 335', 'CPSC 351', 'CPSC 362', 'CS Elective 4', 'GE 4']
5th semester: ['CPSC 223x', 'CPSC 240', 'CPSC 490', 'CS Elective 5', 'GE 5']
6th semester: ['CPSC 315', 'CPSC 323', 'CPSC 332', 'GE 6', 'GE 7']
7th semester: ['CPSC 471', 'CPSC 481', 'CPSC 491', 'GE 8', 'SME 1']
8th Semester: ['SME 2', 'SME 3', 'GR 1']
SUCCESS
```

Plan for a student who is halfway



```
Choose from 1 of 3 possible simulations:
1. New Student
2. Transfer Student
3. Mike Ball
Please enter a number: 2
1th semester: ['CPSC 362', 'MATH 150A', 'MATH 338', 'CS Elective 3', 'GE 5']
2th semester: ['CPSC 335', 'CPSC 351', 'CPSC 490', 'CS Elective 4', 'GE 6']
3th semester: ['CPSC 223x', 'CPSC 240', 'CPSC 315', 'CS Elective 5', 'GE 7']
4th semester: ['CPSC 323', 'CPSC 332', 'CPSC 471', 'GE 8', 'SME 1']
5th semester: ['CPSC 481', 'CPSC 491', 'MATH 170B', 'SME 2', 'SME 3']
6th Semester: ['GR 1']
SUCCESS
```

Plan for Mike Ball



```
Choose from 1 of 3 possible simulations:
1. New Student
2. Transfer Student
3. Mike Ball
Please enter a number: 3
1th semester: ['CPSC 223x', 'CPSC 315', 'CPSC 490', 'CS Elective 5', 'SME 1']
2th semester: ['CPSC 323', 'CPSC 332', 'CPSC 471', 'SME 2', 'SME 3']
3th Semester: ['CPSC 481', 'CPSC 491', 'GR 1']
SUCCESS
```

Resources we used:

- CS Core Courses:
  https://www.fullerton.edu/ecs/cs/_resources/pdf/course_plan/bs-cs-prerequisite-2023-2024.pdf
- Department of Computer Science Page
  https://www.fullerton.edu/ecs/cs/
- Academic Advisement
- (2023-2024) University Catalog for the Computer Science B.S.
  https://catalog.fullerton.edu/preview_program.php?catoid=80&poid=38156&returnto=11049
  https://www.fullerton.edu/ecs/cs/resources/advisement.php
- Video to understand A* search:
  https://www.youtube.com/watch?v=ySN5Wnu88nE&t=434s&ab_channel=Computerphile
- Blog to understand A* search:
  https://saturncloud.io/blog/implementing-the-a-algorithm-in-python-a-stepbystep-guide/

- Textbook Code: https://github.com/aimacode/aima-python/blob/master/search4e.ipynb
- Notion: https://awake-daphne-fa5.notion.site/Project-1-2fd9fd469bbe4645aac0b89e01f95929
- In the development of this code, we utilized ChatGPT as a collaborative tool. Three functions, specifically annotated with the comment “This was pretty much ChatGPT”, were primarily generated by ChatGPT. The remainder of the code was adapted and authored by our team.
    - https://chat.openai.com/share/ee4de5b5-e688-4ced-9eb6-fe199938f966
    - https://chat.openai.com/share/19e90bd4-f3eb-43e9-99a7-1fb0ca0dd161