Benjamin Aasheim

Humer

CSC202

12/6/18

2.1

- A graph
    - A tree would be effective in most cases but because some functions may call each other, an unordered graph would work best to represent various functions and statements. Once all functions are read in lines of code are read in so that if statements and other optional statements/lines of code may become branches of the tree.
    - As the lines of code are read the program treats functions like vertices and forms and adjacency list. If a function is called inside a function, the lines that must be passed through to reach it are listed as the cost. If no other functions are called inside the above, all of the lines of code are listed in the cost of a loop vertice to itself. Alternatively if each statement is a vertice then there are no costs, just links from function call statement-vertices to the first line of the functions that they call. Each vertice would have an attribute visited initialized to False, a line number initialized to the line number read in (+1 from name.readlines index). When the program evaluates coverage it does a depth first search starting from wherever is appropriate after each test and changing the attr to True. The program would then return the total number of true statements over the total number of

statements as the coverage score, and if the " -m" prefix is added to the command, it will create an empty list and append the line number of all vertices where self.visited == False.

- A queue
    - For a breadth first search of the functions and statements
    - This would be the fastest way to see if there are any syntax errors or anything else that would preclude the coverage from even being needed to be run.
- A stack
    - For a depth first search of the graph.
    - Based on a given function call to test which lines are given, the graph will be searched depth first.

2.2

The project is mostly source code and the average execution would call items in what would appear to be a depth first traversal. There are useful README.txt's in almost every directory along with the files: setup.cfg, source.rst, subprocess.rst and others. Very little of the source code is for testing, most is for handling edge cases and preventing errors. Most of the code is well commented although the shorter the file and the more import or other file calls it contains, the less explanation is given.

2.3

- Data.py

    - Data.py Collects all of the available statements and makes what is essentialy an adjacency list for each of them.
    - It stores the data in a Json file. This consists of dictionaries for line, arc, file_tracer, and run data. Mapping the lines executed in each file, the line number pairs, file names with plug in names, and a list of dictionaries with information from each run. It is essentially a traversal of the tree.
    - The code is virtually unreadable at my level. The code calls other functions and implicit methods and offers no input/return explanations in the comments. Attempting to update, maintain, or adapt this code to other formats would be a nightmare.
- Collector.py
    - Collector keeps track of all the statements that were executed and their paths/calls.
    - To do this it Uses a class object called a Collector. The collector appears to use the adjacency lists stored in the dictionaries from data.py
    - Collector is much more thoroughly commented and has more clearly named variables and functions.
- Results.py
    - Results analyses the information compiled by the previous two python files and formats them for a print statement.

    - Results does this using an Analysis class object. Using the data from the above it traverses the tree in a breadth order fashion and keeps track of what statements are called and which are not.
    - The commentary and spacing are much better than in Collector and especially Data, however the variable names are very confusing. I think that ensuring that the correct values are passed through the file and function calls in this particularly as well as Summary would certainly pose a challenge to avoid type or attribute errors.
- Summary.py
    - Takes the analysis formatted by Results.py and prints it for the users consumption.
    - The Summary Reporter class object is essentially just a bundle of functions tied into the __init__ function. The report function is what actually prints the information.
    - Like Collector Summary uses very clear spacing and commentary. The functions could be much more effectively divided into helpers, but other than that it is not that bad. However because of the number of imports in this and Results trying to alter this code and maintain any kind of useability or cross-platform adaptability would be a terrifying prospect.