Cooper Mattern
Francisco Wilken

Lab 9 Writeup

2.1 Before Examining the Code:

Possible Data Structures Used:

- **Directed Graph** - code files are calling other files, functions are calling other functions, all of this movement could be tracked using a graph. A graph could provide information about the flow of the process for line by line tracking
- **Hash Table -** every single line of each file has to be tracked, which would grow in size extremely for big files. This O(1) performance independent of file size is very important, so that coverage checking doesn't take too long. The Hash Table is a good structure for this.
- **Stack -** Function calls need to be handled in a Last in, First out manner, as they need to return their calls to mother functions. This makes a stack, as LIFO data structure, very useful. It is possible the Python interpreter might do this instead, but if not a stack would be useful for coverage.

2.2 Initial Code Examination

The code has a descriptive README with links to the program's documentation in the doc folder, containing files that describe the main functions, classes, modules, etc. Most of the main directory is operating system boilerplate, not really source code, with most of the code being in the coverage folder. Each folder corresponds to a different aspect of the program, with tests being separated from source code and documentation. Most of the project is NOT source code, roughly 30%, with more being tests and documentation. The code is all commented, or at least has a

docstring for smaller functions. Overall the layout is easily navigable through proper naming and documentation.

2.3 Detailed Code Examination

- **Data.py**
  - The data.py file has multiple functions, including getting file paths to data, and updating that data into a Hash Table. It also takes care of parallel data, by combining and/or deleting it.
  - This function uses the Hash Table data structure, almost exactly as we predicted. It uses filenames as keys to store line count information, and also uses dictionaries in smaller functions (a form of a hash table).
  - The code is laid out well in docstring. There are small comments in the complex stuff. Some variables don't have amazing names but can be inferred in context.
- **Collector.py**
  - The collector is the main workhorse as far as getting the data on each file. It uses the collector class to initiate tracers on each thread, running through each thread and gathering the data, some of which is cached locally into temp files.
  - This file uses a Stack just like predicted, as it has to run through trace by trace and pops them when they are complete. This makes a stack extremely useful for both O(1) push and pop, since only data from the top tracer is ever needed.
  - This code has lengthy docstrings for every function, making the process behind each function very clear. However, the lengthy nature and deep understanding of thread order would make maintaining such a file a difficult process at our current skill level.

- **Results.py**
  - The results file has multiple functions in processing the already-measured data. It uses functions in the Analysis class to figure out what arcs (jumps in code from a start to a destination) are travelled or missed, and then uses the Numbers class to format data that will eventually go back to the user.
  - This file uses a data structure similar to that of a Directed Graph, and also uses some sorted lists. The graph component deals with branches and arcs, which have direction between two different lines of code. Most returns from the Analysis class methods are in the form of sorted lists.
  - This code is also well commented, with docstrings often being 5+ lines. The docstrings also mention specific variables in quotes, which is useful for understanding exactly what each variable stands for.
- **Summary.py**
  - The summary reporter class in this file handles calls of 'coverage report,' and does the actual string formatting behind the scenes of that report. It is more of a front end / UX focused file rather than number crunching.
  - No special data structures are really needed here, because summary is only responsible for outputting data rather than processing it. It simply uses normal python structures like lists and tuples.
  - The commenting is still good for this part. Lots of string formatting is not easy to read but just takes some time.

This code is overall better than our own, which looks severely uncommented in comparison.