Coursework 2 - Data Analysis of a Document Tracker
F20SC Industrial Programming

Name: Lachlan Woods (H00223249)
Due Date: Thursday 29th of November 2018

Table of Contents

Introduction

The purpose of this coursework was to develop a Python 3 program to analyze and display data that was stored in json format. The data, which was provided from issuu.com, gave details of user interactions with documents.
Data samples were often very large, which made code efficiency a very important aspect of this coursework. Many of my design considerations were made with code efficiency in mind. My program was tested with a three million line json file (1.7GB), and provided correct results in a reasonable timeframe.

My code has been designed with readability, maintenance and extendibility in mind. Code has been commented extensively, and every function and class has documentation comments. An object oriented programming style has been used throughout my program.

I am very pleased with the program that I developed. I am confident that I have met all requirements. I have tested my program with all test cases provided, and received the correct results for each case. I have used advanced programming concepts covered in this course, including; type hints, high order functions, list comprehensions, and assertions.

Development Environment

My program was developed on a laptop running ElementryOS (Linux distribution) with Python3 version 3.5.2 installed. My program was developed in the PyCharm IDE (2018.2.4 community edition). All code was also tested on the University's linux machines.

Assumptions

I made the following assumptions about this project before beginning development:

1. All json files used as input to our program will be formatted so each entry / document event will be on a new line. This will allow us to read large files line by line, rather than in blocks, or the entire file at once.
2. When generating "also likes lists", the order of documents that have the same number of readers do not need to be in the same order as the "also likes lists" provided as example test results (by Hans-Wolfgang Loidl). For example, The list of tuples ("docid, "number of readers") [("doc1, "2"), (("doc2, "1"), ("doc3, "2")] could be correctly sorted to either: [("doc1, "2"), (("doc3, "2"), ("doc2, "1")] or [("doc3, "2"), (("doc1, "2"), ("doc2, "1")]

Requirements Checklist

Task	Requirement	Achieved	Comment
2a	Given a document ID, generate a histogram displaying the countries of the document's readers	Yes	A dictionary is created that stores the country codes of users who have read the document as keys, and the value as the number of times a user from that country read the document. A histogram is generated from this dictionary using the Matplotlib library.
2b	Given a document ID, generate a histogram displaying the continents of the document's readers	Yes	This requirement was achieved using the same function as the previous requirement. A string operation function was passed in as a parameter (high order function), which allowed modifications to be made to the names of fields, before they were saved as keys in the dictionary. In this case, a dictionary from country codes to continent names was used. Therefore, read events were group together in the dictionary by continent, rather than country codes.
3a	Given a document ID, generate a histogram displaying the browser used by the document's readers	Yes	This was accomplished using the same function as the two previous requirements. The string "visitor_useragent" was passed in as a parameter, which specified the json field to be counted.
3b	Process the "visitor useragent" field, so that only the main browser names are displayed in the histogram.	Yes	Similarly to task 2b, the get_browser_name function was passed in as a parameter. This function uses a regular expression to find the main browser name. The strings returned from this function are

			then used as the dictionary keys.
4a	Implement a function that takes a document UUID and returns all visitor UUIDs of readers of that document.		Function get_document_readers takes a document ID as input, and returns a list of user ids for the people who read the document. This is achieved using a dictionary from document IDs to a list of reader IDs.
4b	Implement a function that takes a visitor UUID and returns all document UUIDs that have been read by this visitor.	Yes	Function get_user_read_documents takes a user ID as input, and returns a list of document IDs that have been read by the user. Similarly to task 4a, this is achieved using a dictionary from user IDs to a list of read documents.
4c	Using the two functions above (4a and 4b), implement a function to implement an “also like” functionality.	Yes	An also_likes function was created. This function returns a list of also liked documents, where each entry is a tuple in the form (docID, [list of readers])
4d	Use 4c to produce an “also like” list of documents, using a sorting function, based on the number of readers of the same document.	Yes	Function also_likes takes in a function as a parameter to sort the entries in the returned list. A function was defined to sort the “also likes list” so that documents that were read by the most other users were first.
5	Display an also likes graph using the results from the also_likes function.	Yes	The python Graphviz library was used to generate an “also likes” graph.
-	The application shall provide a command-line interface	Yes	The python argpass library was used to implement a command-line interface.
-	Develop a simple GUI that reads user inputs, and has buttons to process the data as required per task.	Yes	Tkinter was used to develop a gui, with input fields and buttons for each task. The GUI also has an output field to display debug

			messages.

Additional Functionality

A verbose flag (--verbose or -v) was defined. When the flag is set, additional debug messages will be printed. For example, when set, the also_likes function will return the also_likes list, and print the ID of the users who liked each document in the list. This additional functionality was very helpful in debugging the program. Users of the program may find this flag helpful if they require in depth information.

An output console was also created for the GUI. When the gui flag is set, console messages will also be displayed inside the GUI window. This additional functionality was created with program usability in mind.

Library Usage

The following libraries were used:
argpass[1]: Argpass was used to create a command line interface. The argpass library allows command line arguments to be defined, and assigned attributes, such as help text, full and short argument names, default values, required/optional arguments, and type information. It was suggested in lectures that the getopt library could be used for the same functionality, however I decided to use the argpass library after reading that it requires fewer lines of code and provides better error messages than the getopt library[2]. I briefly used the getopt library, and found the previous statement (made by Python Software Foundation) to be true.

graphviz[3]**:** A graphviz library exists for python. This library was used to generate and display the graph required for task 5. This library did not appear to be installed on the Linux machines, so had to be downloaded. I have included the Graphviz files in my code submission.

re: The re library can be used for regular expressions. Regular expressions were used to complete task 3b, as they provide an easy way to find the main browser name from a user agent string.

json: The json library was used to read data from json files line by line. I found this library sufficient for this coursework, as very little processing of the json data was required. I therefore did not use the pandas library.

tkinter.filedialog and tkinter.scrolledtext: These two additional tkinter libraries were imported and used during the development of my GUI. The filedialog library provides an easy to use file selector window, which allows users to change the input json file. The scrolledtext library was used to implement the console view of the GUI. This library provides a text view that can be scrolled once text no longer fits within the view.

Design Considerations

Class Design

An object oriented programming approach was used during the development of my program. Classes and methods were defined so that each class had one clear, well defined purpose. Python documentation comments have been written for all classes and methods. This design decision ensures my code is readable, extendable, and easily maintainable for other developers.

A ReturnStatus enum was created. This enum contains codes to indicate the success or failure of functions. A ReturnStatus value is returned by most functions in my program. For example, when loading a file using the load_json function, either a BAD_File or SUCCESS value will be returned. This enum provides more detail than True/False return types, and was defined to remove the need for exit() calls when bad input was provided. When a function is called, the return status can be checked, and the appropriate action can be made accordingly.

Data Structures

Dictionaries: Data read from json files are stored in two dictionaries:
user_id_to_documents is a dictionary from a user ID to a list of documents read by the user.
doc_id_to_readers is a dictionary from a document ID to a list of readers of the document.
These two dictionaries were used as they allow for efficient implementations of all task. For example, to find an also likes list for document D, D can be used as the key to doc_id_to_readers. This will return a list of all readers of document D, and has O(1) time complexity. The list can than be iterated, which has O(u) complexity, where u is the number of people who have read document D. For each reader, the user_id_to_documents dictionary can be used to build a list of also_likes documents, which also has an O(1) time complexity. This is the most efficient implementation that I could think of, as it removes the need to iterate over all read events.

Also likes list: The list returned from the also_likes function (task 4) contains tuples in the form (docID, [readers]). The list is structured in this way, as it allows for detailed debug messages, and simplifies task 5. Since the readers of each “also likes” document is stored in the list, there is no need to do additional docID to reader dictionary lookups when generating the also likes graphs.

Advanced Language Constructs

List comprehensions: List comprehensions were used to build lists from data stored within json objects. For example:

```
user_read_ids = set([d["env_doc_id"] for d in user_read_list])
```

Is a one line implementation to select the “env_doc_id” attribute from a list of all read events for a particular user (without duplicate entries, as the list is converted to a set). List contcatinations are very useful for constructing lists in very few lines.

Assertions: Assertions were particularly useful as Python is dynamically typed. My program made use of high order functions. I often used assertions to check if a parameter was a callable function. I also used assertions to ensure that empty dictionaries were not used to generate histograms.

High order functions: High order functions are functions that take other functions as input. I often used high order functions, so a single function could be used to complete multiple tasks. This design decision was important for minimising code duplication. My json_load function takes a filter function, so only relevant lines from json files are stored. This filter function was typically in the form of a lambda expression. For example:

```
data_filter = lambda x: True if input_doc == x["env_doc_id"] else False # only save read events for the requested doc
```

This design decision was made with performance in mind. Since very large json files were used as test data, it would be inefficient to store all data, when only a single document was being analysed.

Type hints: Since Python is dynamically typed, it can be difficult to determine the type assigned to variables. Type hints can be used to indicate the desired type of a variable. Some tools, such as the PyCharm IDE (which I used to develop my program) have built in support for type hints, and will provide dynamic time warnings for possible typing errors.

User Guide

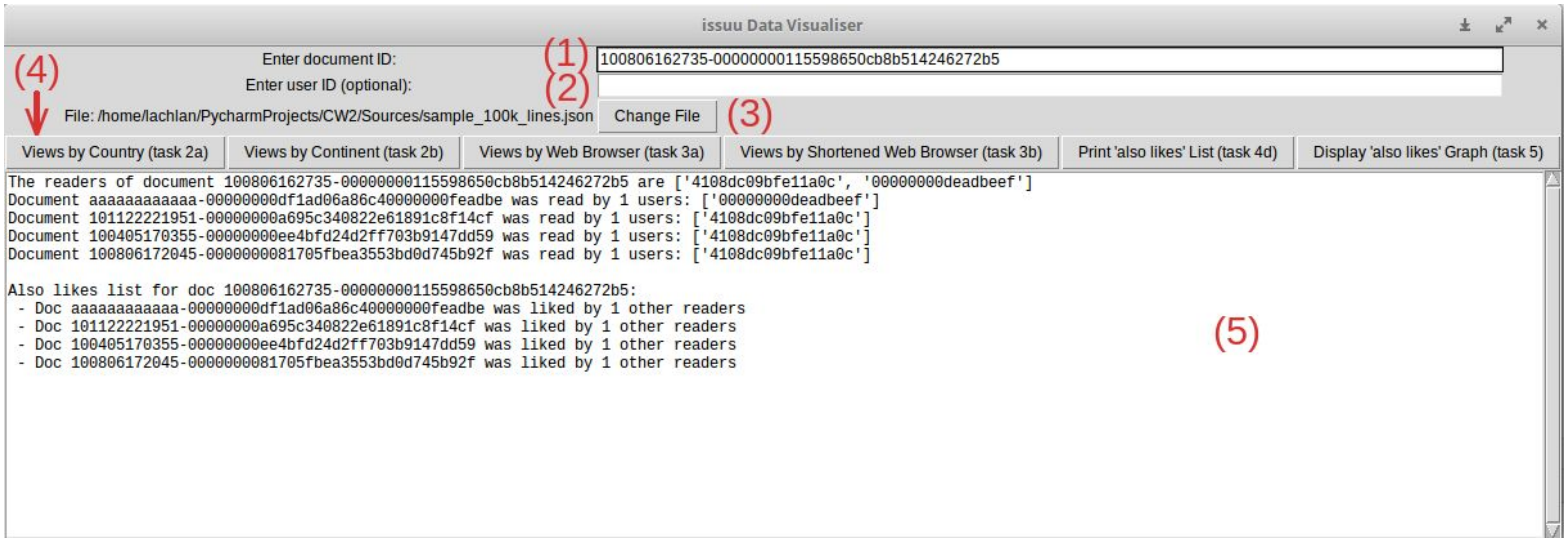
Command Line Interface

Type “./cw -h” or “./cw --help” for a list of command line arguments and usage information.

```
./cw2 -h
usage: cw2 [-h] [-u USERID] -d DOCID [-t {2a,2b,3a,3b,4d,5}] -f FILE [-g] [-v]

optional arguments:
  -h, --help            show this help message and exit
  -u USERID, --userID USERID
                        The id of the user you wish to display, specified by
                        the field: visitor_uuid
  -d DOCID, --docID DOCID
                        The id of the document you wish to display, specified
                        by the field: env_doc_id
  -t {2a,2b,3a,3b,4d,5}, --task {2a,2b,3a,3b,4d,5}
                        The task number to run
  -f FILE, --file FILE  A file path to a json file containing data from
                        issuu.com
  -g, --gui             set this flag to display a gui.
  -v, --verbose         set this flag to display detailed is likes data in the
                        console.
```

GUI



1) Document ID input field: Enter the ID of the document you wish to analyse. All graphs generated will revolve around the document entered.
2) User ID input field: This is an optional field. Entering a user ID will impact also likes lists and the also likes graph. If a user ID is specified, also likes lists will not include other documents read by that user.

3) File selector: Clicking this button will open a file selection window. You may select different json files from this window to analyse different data sets.
4) Task buttons: A button exists for each task. Clicking on a button will run the corresponding task with the values entered in the input fields.
5) Output console. Information, also likes lists and error messages will be printed to the output console. You may set the verbose flag with -v or --v when launching the program to view detailed messages.

Developer Guide

The documentation below details the classes and main functions of my program:

ProcessData.py

ProcessData is the class responsible for loading data from json files. Data is stored within two dictionaries, user_id_to_documents and doc_id_to_readers.

Reading json files

The load_json file will read a json file line by line, and stores data in two dictionaries. A filter function can be used to only store relevant lines.

```
load_json(self, file_path: str, filter_function, display) -> ReturnStatus:
```

Input:
file_path: The file path of the json file to read
filter_function: A function to filter the json entries to be saved (for performance reasons)
display: A DisplayData object (needed to print errors to gui console)
Returns: A ReturnStatus indicating if the file was loaded successfully (SUCCESS) or not (Bad_File)

Get the readers of a document

get_document_reader takes a document ID and returns a list of all users who have read the document.

```
get_document_readers(self, doc_id: str) -> list:
```

Input:
doc_id: The document to return the readers for
Returns: A list of user IDs for the people who have read a document

Get the documents read by a user

get_user_read_documents takes a user ID and returns a list of all documents read by the user.

```
get_user_read_documents(self, user_id: str) -> list:
```

Input:
user_id: The ID of the user
Returns: A list of Document IDs read by user_id

TaskManager.py

The TaskManager class is responsible for calling functions to perform each task. This class is called from the main method in cw2.py when the command line interface is used, and the GUIManager class when the GUI is used.

Run a specific task

The run_task function runs a specific task using the user input provided from the command line, or GUI input fields.

```
run_task(self, task: str, input_doc: str, input_user: str, file: str, display) -> ReturnStatus:
```

Input:
task: The task ID to run as a string
input_doc: The document ID to run the task for
input_user: The user ID to run the task for
file: The file path of the json data to analysed
display: A DisplayData instance. This is the object that will be used to carry out all tasks.
Returns: A ReturnStatus code specifying the result of the task

GUIManager.py

The GUIManager class is responsible for displaying and processing GUI events. The GUI Manager is only used if the -g or --gui command line argument is set.

Creating the GUI

To display the GUI, create a new GUIManager object. Calling the display_gui function will then display the GUI window.

```
gui = GUIManager(args.docID, args.userID, args.file, data_displayer)
gui.display_gui()  # show the gui
```

Input:
input_doc: The doc ID to display in the document ID input field on launch
input_user: the user ID to display in the user ID input field on launch
file: the file path of the json data to load
display: A DisplayData object. This will be used to complete all tasks

Display a message in the GUI console

The append_to_console function appends a string to the end of the console displayed in the GUI window.

```
append_to_console(self, message: str):
```

Input:
message: The message to display

ReturnStatus.py

ReturnStatus.py contains an enum of possible values that can be returned from a function. Values include:

```
BAD_ID = "Could not find any results with the specified user and document id"
BAD_File = "Could not load file"
NO_LIKES = "Could not find any 'also likes' documents"
Bad_Task = "Invalid task name. Please type -h for usage info"
SUCCESS = "Success"
```

cw2.py

Cw2.py contains the DisplayData class. This class contains functions to create histograms and 'also likes' graphs / lists.

Creating histograms

The create_histogram function will create and display a histogram using the matplotlib library.

```
create_histogram(self, data: ProcessData, input_doc: str, field_name: str, string_function, graph_title:
str) -> ReturnStatus:
```

Input:
data: The ProcessData object that was used to load a json file. This will contain the two dictionaries needed to populate the histogram.
input_doc: A string representing the document to be analysed.
field_name: The name of the json field that is to be plotted on the histogram. For example, to create a histogram of reader's countries, this string would be "visitor_country".
string_function: A function that will be applied to all strings stored in the json data, under the field specified by field_name. This function is used to modify strings before they are plotted on the graph. For example, a function could be passed in to convert country codes to continents.
graph_title: The title to be displayed above the histogram.
Returns: A ReturnStatus value. This will be SUCCESS if the histogram was successfully generated and displayed, or BAD_ID if no data could be found relating to the input document.

Creating also likes lists

The also_likes function returns a list of also likes documents. List elements are a tuple in the form (docID, [readers])

```
also_likes(self, data: ProcessData, sorting_function, input_doc: str, input_user:str = None) -> list:
```

Input:
data: The ProcessData object that was used to load a json file.
sorting_function: A sorting function to order the returned list
input_doc: The id of the document you want to find “also liked” documents for
input_user: (Optional) A user id to find also liked documents for.
Returns: An also likes list of tuples in the for (docID, [readers])

Draw an also likes graph

draw_relation_graph takes an also_likes graph, and displays it as an also likes graph. This function makes use of the Graphviz library.

```
draw_relation_graph(self, data: ProcessData, also_likes: list, input_doc: str, input_user: str = None)
-> ReturnStatus:
```

Input:
data:The ProcessData object that was used to load a json file.
also_likes: A list of also liked documents (from the also_likes functions)
input_doc: The ID of the root document, i.e. the document used to generate the also likes list.
input_user: The ID of the user used to generate the also likes list (optional)
Returns: A ReturnStatus value. This will be SUCCESS if the graph was successfully generated and displayed, or BAD_ID if the also likes list was empty.

Testing

Assertions were used throughout my program to perform runtime tests.
The following tests were conducted on my program. The expected and actual results have been recorded.

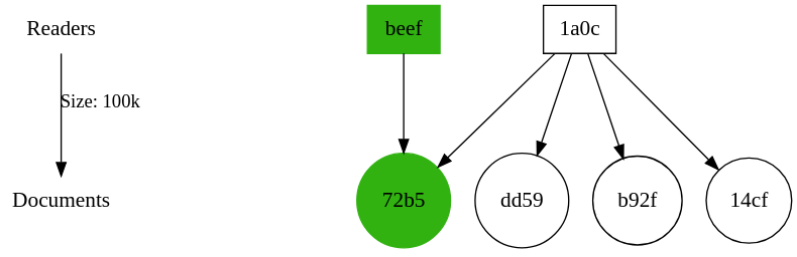
Action	Expected Result	Actual Result	Pass?
Provide an incorrect task name as a run time argument	A Bad_Task return status should be returned by the TaskManager class, and an error message displayed in the console.	A Bad_Task code was returned, and program usage information was displayed in the console.	Yes
Provide a file path that does not exist as a run time argument	A Bad_File status should be returned from the ProcessData class, and a warning message displayed in the console.	A Bad_File status was returned from the ProcessData class, and a warning message was displayed in the console.	Yes

Provide a file that does not contain valid json as a run time argument	A Bad_File status should be returned from the ProcessData class, and a warning message displayed in the console.	A Bad_File status was returned from the ProcessData class, and a warning message was displayed in the console.	Yes
Provide a document ID that does not exist in the json file loaded, and run task 2a.	A BAD_ID status should be returned, and an error message displayed in console.	A BAD_ID status was returned, and an error message was displayed in console.	Yes
Run task 2a with a valid json file and document ID	A histogram displaying the countries of readers of the document should be displayed.	A correct histogram was displayed.	Yes
Run task 4d with a user ID that does not exist in the loaded json file	A BAD_ID status should be returned, and an error message displayed in console.	A BAD_ID status was returned, and an error message was displayed in console.	Yes
Run task 4d with a valid json file, doc ID and user ID.	An also likes list should be displayed in the console.	A correct also likes list was displayed in the console	Yes

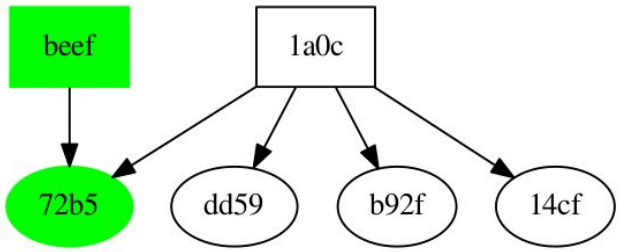
Testing against provided test cases

The following test cases were made available at:
http://www.macs.hw.ac.uk/~hwloidl/Courses/F21SC/Test_Data/

100K lines:
doc_uuid: 100806162735-00000000115598650cb8b514246272b5
user_uuid: 00000000deadbeef
Expected:



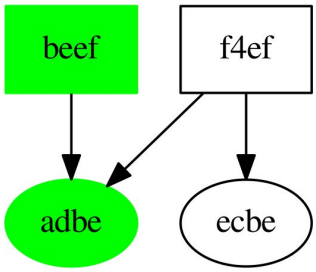
My output:



doc_uuid: aaaaaaaaaaaa-00000000df1ad06a86c40000000feadbe
user_uuid: 00000000deadbeef

Expected:

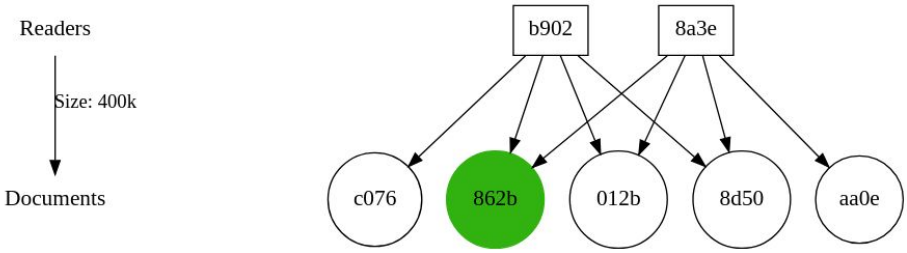
My output:



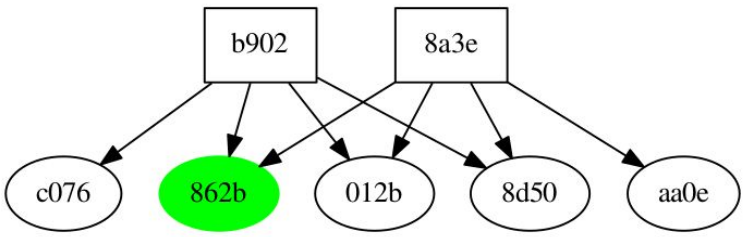
400K Lines:

doc_uuid: 140310170010-0000000067dc80801f1df696ae52862b

Expected:

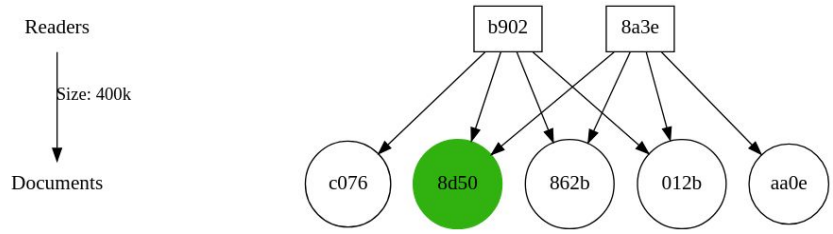


My Output:

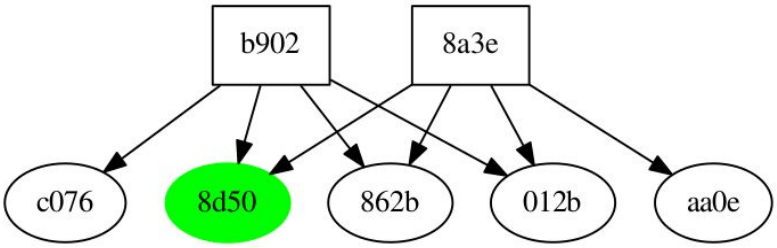


doc_uuid: 140310171202-000000002e5a8ff1f577548fec708d50

Expected:



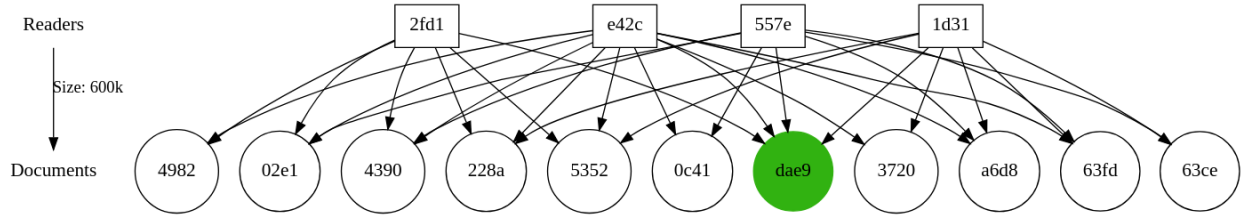
My Output:



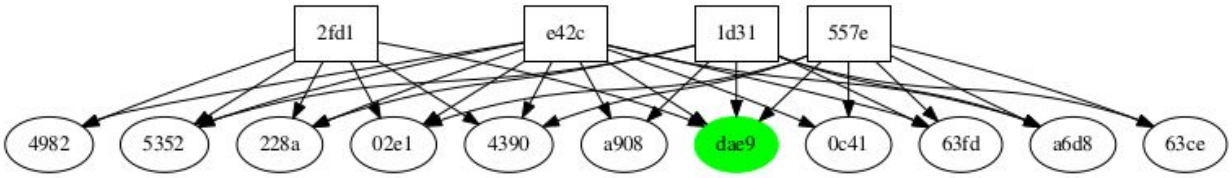
600K Lines:

doc_uuid: 140207031738-eb742a5444c9b73df2d1ec9bff15dae9

Expected:



My Output:

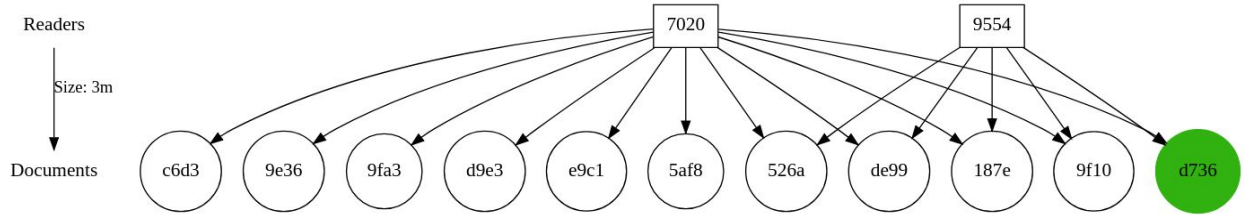


Note: This output is correct. Read assumption 2 for details.

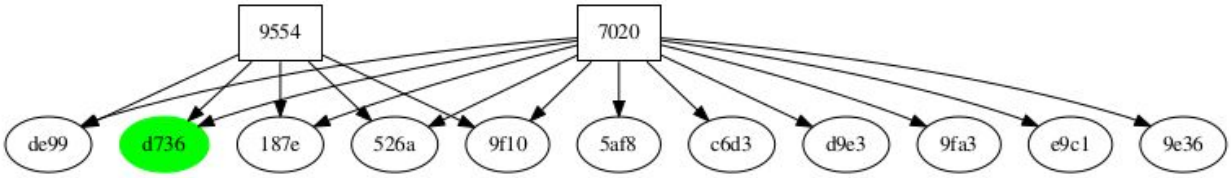
3M Lines:

doc_uuid: 140109173556-a4b921ab7619621709b098aa9de4d736

Expected:

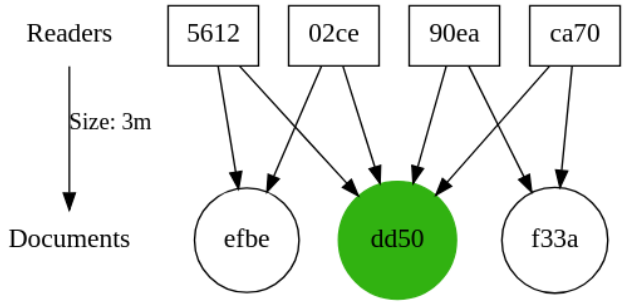


My Output:

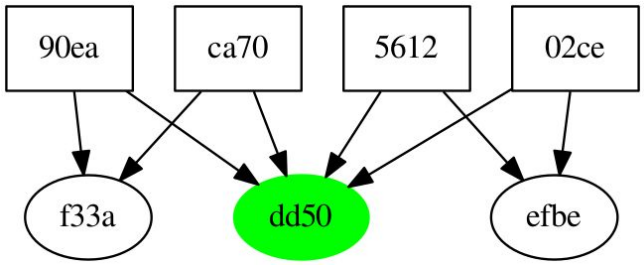


doc_uuid: 140213232558-bdd53a3a2ae91f2c5f951187668edd50

Expected:



My Output:



What did I learn from coursework one

The feedback I received on coursework one was very useful, and was taken into consideration when working on this coursework.

I was pleased to receive positive feedback on my coding style for coursework one. I have strived to continue this for coursework two. It was mentioned that in some cases, I did not separate GUI code from logical code. I understand this is important, as it should be possible to modify/extend GUI and logical code independently. To ensure code separation in this coursework, an independent GUIManager class was created. This class simply listens for user events, and makes calls to the TaskManager class. Therefore, all GUI code is entirely seperated from logical code.

My feedback also stated that my code was lacking a copyright license statement. This has now been included in all my classes.

It was stated that my developer guide should not include entire method bodies, but rather provide the interface to the method. This feedback has been taken into consideration for this report. My developer guide now provides details of parameters, return types, and a brief explanation of each function's purpose.

Feedback for coursework one also stated that my report should include citations in text, rather than just a list of references. This feedback was taken into consideration for this report.

The coding exercise given during the demo of coursework one provided a good example of coding questions that could be given during a job interview. I am happy with the answers I gave to the first two tasks, which asked for methods to count the depth of a tree, and find the total number of nodes. I struggled with the final two tasks, which required use of a delegate. I could not remember the correct way to define a delegate. For the upcoming demo, I will read over the class notes to refresh my memory on the syntax and use cases of advanced coding constructs.

Conclusions

I am confident that I have met all requirements of the coursework. My program provided correct outputs for all test cases provided. Emphasis was put into code efficiency. When developing functions, I always considered the Big-O complexity of my code. I believe my use of two dictionaries; doc_id_to_readers and user_id_to_documents, was a good design decision for data storage.

I am very happy with the quality of my code, and believe that software engineering best practices have been used throughout my program. Code has been written to be easily readable and maintainable. This has been achieved by using an object oriented style for programming, and commenting all code, methods and classes.

High order functions were frequently used in my program. High order functions were very useful, as they allowed a single function to be used for many tasks. This drastically reduced the number of lines of code, and helped prevent code duplication. Lambda functions were often written, and passed as parameters. Lambda functions were useful for writing one line functions to filter data.

If I were to repeat this coursework, I would like to implement some multithreading or parallelism into my program. I believe it would be possible to partition json files into smaller parts, and read them in parallel. This could speed up the process of reading in large files. I would also like to implement multithreading for the GUI, as currently the GUI will freeze while large files are being read, after clicking on a task button.

References

[1] argpass library: https://docs.python.org/2/library/argparse.html
[2] getopt library: https://docs.python.org/2/library/getopt.html
[3] graphviz library: https://pypi.org/project/graphviz/
[4] Test cases: Hans-Wolfgang Loidl. Available at:
http://www.macs.hw.ac.uk/~hwloidl/Courses/F21SC/Test_Data/

[In Code cw2.py line 22] Country codes to continents. Adapted from simple_histo.py by Hans-Wolfgang Loidl http://www.macs.hw.ac.uk/~hwloidl/Courses/F21SC/index.html

[In Code cw2.py line 313] Display a histogram using the matplotlib library. Adapted from simple_histo.py by Hans-Wolfgang Loidl
http://www.macs.hw.ac.uk/~hwloidl/Courses/F21SC/index.html

[In Code cw2.py line 334] Graphviz code adapted from graphvis' documentation. available at https://graphviz.readthedocs.io/en/stable/manual.html

[In Code cw2.py line 371] A regular expression to select the main browser name from a user agent string. Adapted from regex provided by ticky https://gist.github.com/ticky/3909462