CSC505 Jennings

Homework 3, Spring 2020

Please read, implement, and answer all 4 sections below.

1. Names of team members:

Bhoomi Shah (bshah2)

Poorva Kulkarni (pnkulkar)

Rohan Pillai (rspillai)

1. Overview:

In this assignment, you will design algorithms for solving string-related problems. You will implement some of your designs, as indicated in the next section. (The Analysis Questions section prompts further design activity.)

You may also need to write some code (or use existing programs) to analyze the provided sample data.

You must provide (via NCSU GitHub) to the teaching staff any source code you use and scripts that can be used to build any code and run any/all steps needed to reproduce your results.

And, as always, you must cite the origin of all source code, which may be original to your team or individual team members, or may be appropriately obtained from elsewhere. These programming languages are acceptable for any source code used in this assignment:

Java 12

Python 3.7

C (C11 or ANSI)

C++ 17

1. Implementation Task:

Using the data files in the HW3 directory of https://github.ncsu.edu/jajenni3/csc505 as sample input files, design and implement solutions for the following problems. Your solution should have a command line interface so that automated tests may be easily scripted.

**DIALOG:** Given a list of input files similar to the sample files (but not previously known), extract all the dialog into a separate output file. In the output, retain the quotation or other marks that separate the dialog from the narrative.

**DIALOG SEARCH:** Given one input file similar to the sample files (but not previously known), and a search string, determine if the search string appears within dialog in the sample file. If it does, then output each Chapter number/title (equivalently for anthologies, the number/name of the individual work) in which the search string appears.

You may require additional parameters to be supplied on the command line, or from a configuration file. In that case, document what each one does, and provide examples of their usage.

Put your code and data into a GitHub repository on github.ncsu.edu in a branch called “HW3” (upper case). Give read access to the teaching staff (3 TAs and Dr Jennings – email addresses on Piazza).

1. Analysis Questions:
   * + 1. **Give the URL to an NCSU GitHub repository containing your code, scripts, and data.**

**Answer:** <https://github.ncsu.edu/rspillai/CSC505-HW3>

* + - 1. **Provide citations for your source code and any data processing programs used. For command line utilities like grep, indicate the version (e.g. using grep --version). For any installed software, like a Python package, give the version and the command needed to install it.**

**Answer:**

**Citations:**

**Source Code: None as we have implemented the code on our own.**

**Input Files:** [**https://www.gutenberg.org/**](https://www.gutenberg.org/)

**Regex Testing:** [**https://regex101.com/**](https://regex101.com/)

1. Module: re - Regular Expression Module

Version: 2.2.1

Command to Install : pip install re

1. Module: sys - System-specific parameters and functions Module

Version: 3.7.6

Comes pre-installed with Python

* + - 1. **Demonstrate that your implementations work by providing a script for the DIALOG problem and another script for the DIALOG SEARCH problem.**
         1. The script for DIALOG should run your program on both sample input files, producing output files that you should commit to your repository.

**Answer:** The script has been uploaded on GitHub and it has to be executed using the following command:

**python dialog.py [--input\_filename] [--output\_filename]**

--input\_filename name of the input file along with its path

--output\_filename name of the file you want to store your output in.

* + - * 1. The script for DIALOG SEARCH should run your program multiple times, on a variety of search strings, using each of the sample files.

**Answer:** The script has been uploaded on GitHub and it has to be executed using the following command:

**python dialogSearch.py [--input\_filename] [--search\_string] [--output\_filename]**

--input\_filename name of the input file along with its path.

--search\_string string to be searched in the input file.

--output\_filename name of the file you want to store your output in.

* + - 1. **Documenting your approach to DIALOG and to DIALOG SEARCH. I.e. explain in words (and/or diagrams) how your algorithm finds the needed pieces of text. Importantly, this explanation documents (1) all assumptions you make about the input data, and (2) any additional parameters needed for your programs to run.**

**Answer:**

**Approach for DIALOG**

The script for DIALOG seeks to fulfil the function of extracting dialogues i.e all strings which lie between two quotation marks. We first create a regex as "[\"“]([^\"”]\*)[\"”]" which scans the text for expressions which have occurrences of the pattern as described by the regex using re.findall(). Further, we accumulate all these matched patterns in a list. We then open an output file and write the dialogues in it. In the output file, quotation marks and other punctuations pertaining to the relevant dialogues are retained.

**Approach for DIALOG SEARCH**

In order to search text in dialogues, we first extract the chapter names and numbers. Then for every chapter, dialogues are extracted using the approach mentioned in DIALOG. Once, dialogues for a chapter are extracted and the search string is present in this list of extracted dialogues, then the chapter name and number is displayed, and if it does not contain then the code iteratively goes on checking this for all the other chapters. If no such match is found in any of the chapters then a no match message is the output of the script.

**Assumptions about input data:**

1. There will be a single style of enumerating the chapters. In a given text file, there will never be more than 1 manner in which chapter numbers/titles are written. For example: only one of the following will be valid for a single input file- either **CHAPTER IV** or **IV.** So the files will either be in one of the two specified formats:

**I. A SCANDAL IN BOHEMIA**

(Chapter Number appears along with chapter name)

OR

**CHAPTER I**

**JONATHAN HARKER'S JOURNAL**

(Chapter Number appears on one line and chapter names appear on the following lines)

1. The unseen input files will have chapter and title name formatting similar to those in dracula.txt or sh.txt
2. The two types of quotation marks we are considering as a valid input are “ ” and " ".
3. We are not considering the lines which have sub-dialogs ie. a structure similar to “ ‘ ‘ “ as the program categorizes a line as a dialog only if it is present between two valid quotation marks.
4. Some paragraphs follow the archaic practice where a quotation is used at the beginning of paragraph. Our code does not handle any such formats.
   * + 1. **How can a user determine whether a new input file (never seen before) will be acceptable to your programs? If you used any additional parameters, how does the user determine what values to provide?**

**Answer:** In the implementation, we have made certain assumptions about the type and style of formatting applicable to a given input file. In the text files - dracula.txt and sh.txt, there were 2 distinct patterns of chapter/title formatting observed as follows:

1. **CHAPTER IV** **(in dracula.txt)**
2. **IV. (in sh.txt)**

We have implemented the regular expressions for identifying chapters given in the above formatting style appropriately. As long as the new input files have the same format of chapters, there will not be any issues in the solution and will be acceptable.

For new inputs having a different formatting style, a new regular expression needs to be generated to satisfy the criterion of that input file.

* + - 1. **Consider the two sample input files, and the output of your DIALOG solution. How did you verify that your implementation works correctly?**

**Answer:**

The code for DIALOG writes the lines involving dialogs in an output file. We created test cases for the following scenarios:

1. Not a dialog (plain-text).
2. Dialog within opening and closing quotes.
3. Mixed text i.e having characters interspersed between quotations.

**Verification:**

On running the test cases against the expected behavior of the program, we tried analyzing the appropriate outputs for each of the scenarios and verified the solution.

* + - 1. **Consider the two sample input files, and the test cases you scripted for DIALOG SEARCH. How did you verify that your implementation works correctly?**

**Answer:**

**Verification:**

Apart from the two given text files, we tried our solution on some text files from the Gutenberg library which are ***Living the Radiant Life A Personal Narrative, Picturesque Pala, Scenic Mount Lowe and Its Wonderful Railway, The Grand Canyon of Arizona*** which have the same formatting structure as dracula.txt or sh.txt. The code works correctly for these files as well. This verifies that the implementation will work for files having structure similar to given files.

* + - 1. **What is the worst-case asymptotic complexity of your DIALOG solution as a function of the m characters of total input? And for DIALOG SEARCH, as a function of m characters of input text and an n-character search string?**

**Answer:**

**Complexity of DIALOG:**

The complexity of the dialog program depends upon how python uses the regular expression matching. There are two approaches discussed after a perusal of the information given at <https://swtch.com/~rsc/regexp/> which are discussed below.

**Approach 1:**

We use re.findall() to find the occurrences of a given regular expression in the m characters of input. Regular expressions, when implemented as a finite state automaton can work on the input string in a time complexity of O(input character size). Here, we are given an input file of m characters and hence would require to iterate once over every character to find whether it matches the regex criterion. Hence, for finding dialogs in quotes, we require a worst case asymptotic time complexity of O(m).

Further, for every string which is a part of the dialog, we need to write to an output file. Let d be the total number of characters identified as part of the dialog. Hence, there will be (m-d) less iterations for writing to an output file. Hence, total worst case asymptotic time complexity is:

**= O(m) + O(d)** where d is the number of identified dialogs

**= O(m)**

**Approach 2:**

Considering the approach stated by Russ Cox, in which he elaborates that regular expressions matching is slow in languages which do not use automata for matching. Such approaches use the standard backtracking implementation which causes the worst case time complexity of upto **O(2^m)** where m is the number of input characters.

**Complexity of DIALOG SEARCH:**

From the follow-up of the discussion of complexity of DIALOG is **O(m)**. Further, we have a ‘p’ number of chapters in which we have to search for a string of input length n. This requires us to iterate through **p** chapters for searching a **n** input string. Hence, the total worst case asymptotic time complexity is : **O(m) + O(n\*p)**

* + - 1. **DIALOG design question: How would you change your solution to DIALOG if you expected the same input file to be given multiple times over the course of days, weeks, or even years of people using your program? Consider how you would identify the input file, and the fact that persistent storage is always limited. Give the benefit and the cost of your approach**.

**Answer:**

While analyzing the input file and finding patterns, preprocessing is an important step which is often expensive and thus involves preprocessing operations every time the file is given as input. We could avoid the extra pre-processing cost everytime by storing the already preprocessed input file once and further using this for dialog searching. In this manner, there will be no need to have the file in memory every time it is given to the program.

Considering we have the same input file, we could store the beginning characters/words in a prefix tree or trie. As and when there will be new characters, we could add nodes to the trie. Whenever an input file is given, we can easily match the first few characters to identify if the given file has been encountered before and thus reduce the preprocessing costs.

* + - 1. **DIALOG SEARCH design question: How would you change your solution to DIALOG SEARCH if you knew in advance the collection of files (the “corpus”) that your program would have to search? Consider how the user would identify which file they wanted to search. How could additional files be added to the corpus? Give the benefits and costs of your approach.**

**Answer:**

As observed in the current sample files, we can identify the different manners in which chapter titles are formatted and store it as a key in a dictionary using the characteristic pattern of the text. This key would have a corresponding value as a regular expression which successfully retrieves the chapter/title name. The code for searching the input string in the given dialogs of an arbitrary chapter will be the same.

Considering we know the corpus in advance, we can memoize the characteristic pattern and use the corresponding regular expression for extracting chapter number from strings encountered in the future. When the pattern is matched, the given file can be identified as having congruent structure to that as another key in the pattern dictionary. In case a new type of pattern occurs, we can create a new entry in the pattern dictionary.

**Benefits**: This kind of approach has the benefit that given every pattern is mapped in advance, the extra costs of trying and testing every regular expression is avoided.

**Costs**: As dictionaries provide a constant search time due to key-based indexing, every pattern would be checked exactly once, thus reducing the costs of trying every combination of regex to retrieve the chapter name/title.