

EN 600.425 DECLARATIVE METHODS

- TERM PROJECT WRITEUP

BETTER `diff`

Guoye Zhang, Qiang Zhang

May 6, 2017

1 Introduction

The utility `diff` is commonly used in all scenarios, whether for programming purpose or in general text editing circumstances. The website Github integrated `diff` into their version control system, facilitating ubiquitous knowledge of the utility. This legacy utility is based on the renowned *Edit Distance* algorithm, which in essence is a Dynamic Programming algorithm.

Aside from the application domain of version control, `diff`'s algorithm in itself is interesting and instructive enough for most computer science students to delve into. In general sense, the concepts of finding similar text and displaying minimum edit distance can be embodied in various domains and categories of applications. Our project aims at extending `diff` to provide firstly the feature of multi-file editing source referencing, accompanied with visualization, that can display the chain of editing among more than two files placed in the same folder, together with a better algorithm that fixes a certain scenario where the normal `diff`'s Edit Distance algorithm would perform suboptimally, which is a case we already described in the proposal.

1.1 Algorithmic Optimization

As a reminder, recall the scenario where normal `diff` would perform suboptimally. In this code snippet:

```
for(int i=0; i<10; i++) {  
    System.out.println("First_line");  
}  
for(int i=0; i<10; i++) {  
    System.out.println("Second_line");  
}  
for(int i=0; i<10; i++) {  
    System.out.println("Third_line");  
}
```

from which we will delete the second `for` block. And if we use the traditional `diff` on the old and new versions of this snippet, we would get:

```
for(int i=0; i<10; i++) {  
    System.out.println("First_line");  
}  
for(int i=0; i<10; i++) {  
-    System.out.println("Second_line");  
- }
```

```
- for(int i=0; i<10; i++) {
    System.out.println("Third_line");
}
```

Instead of what would be expecting as follows:

```
for(int i=0; i<10; i++) {
    System.out.println("First_line");
}
- for(int i=0; i<10; i++) {
-     System.out.println("Second_line");
- }
for(int i=0; i<10; i++) {
    System.out.println("Third_line");
}
```

This case is fixed with our improved version of `diff` algorithm, which we shall call by `MDiff` in the rest of this document. In fact, there are much more scenarios where the traditional `diff` algorithm would disappoint you. For example, consider the instance of comparison below (results are hand-generalized from Github's webpage highlighting):

```
<a><b>
----

<b><a>
++++
```

The notations used here should be self-explanatory. When comparing the two strings of `<a>` and `<a>`, traditional `diff` would return the results as shown above, which means, in a 0-based indexing convention, deleting the 1-4 of the first string, and adding the 4 characters at index 1-4 of the second string. But this answer of editing distance is clearly suboptimal. The optimal solution we expect would be:

```
<a><b>
- -

<b><a>
+ +
```

which is the answer that `MDiff` would return.

1.2 Multi-file Referencing

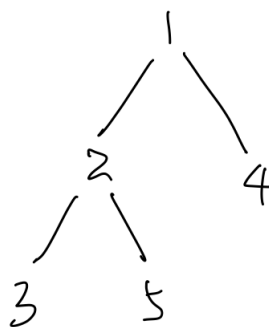
In this part, we extended `diff` to another domain of application of interest. To most eyes, `diff` is generally used to comparing the editing distance between two files. It is a utility used widely in use. Indeed, it is a utility, in that it does a job well, but it is not typically tailored for any domain. We implemented an application that embodies `MDiff` in its core, but meanwhile provides far more convenient and appealing functionalities.

For oft, we are interested not only to know the difference between two text files. Rather, we want to be able to look at some paragraph of this text file, and find out from which other paragraph in other file this paragraph derives. Text reusing is so common nowadays that this is a feature of no trivial productivity significance. In fact, to fulfill such a task, of finding the predecessor of a paragraph in this text file, most people would resort to nothing better than a brute-force file-by-file *Open&Search* routine.

To implement this feature, the optimized algorithm of `MDiff` alone is not quite enough. In practical scale, even comparing only two files with a `diff`-based algorithm can be undesirably slow. Not to mention that we are trying to determine edit chains between multiple files. For the application to be fast enough to be useful in practice, certain techniques are used to structure the implementation of its functionality.

The use case we are trying to address here is a scenario where the folder contains, rather than multiple files, multiple versions of a file. For example, `file1` could be the original file of an article in craft. Then after certain modifications, `file2` results. In the end, the folder contains multiple files, each pair of which, albeit being different, shares an overall similarity larger than 50%. Aside from that, we make the below assumptions:

1. File name indicate each file's last-modified-by time in an increasing order. Thus if there is a file i and a file j with $i < j$, we can convince ourselves that file i is lastly modified earlier than file j and file j is thus possible a successor of i : meaning we should include file i 's texts when we try to search for the predecessor of any text paragraph in file j . To be honest, this modelling is not perfectly accurate. For example, in the case below:



File 5 is modified later than file 4 (since 5 is larger than 4, and thus file 5 has a later last modification timestamp), but file 5 actually should not consider file 4 as one of its predecessor, because file 5 comes from modifications of file 2, and is on a editing path branch disjoint from that of file 2.

But overall, our approach should work fine.

2. The author of the files knows to separate each section, or paragraph of his files by an empty line. Like:

<paragraph1>

<paragraph2>

<paragraph3>

This is similar to how most Latex users format their documents.

When we open a certain file i , we want to be able to know:

1. **Objective 1:** Amongst all the paragraphs of all files in this folder, which paragraph is most likely to be the direct predecessor of this paragraph? Or, which paragraph is most likely to be the direct source of modification for this paragraph?
2. **Objective 2:** After finding the most likely predecessor of this paragraph, what are the minimum amount of edits required to change from the predecessor paragraph to the paragraph under inspection?

Objective 2 as we described in the last subsection, is addressed by an optimized version of `diff` algorithm, which we call `MDiff`. Objective 1's on the other hand, is key to speeding up the application to a speed suitable for real-life utilization.

By matching texts in the unit of paragraph first, we largely reduced the computational effort required. Yet this scheme does not violate any practical editing convention to render the application infutile. Normally, when a user edits a file, the edits, however many there are at a time, can be each bucketted into its own paragraph. By matching paragraphs first, we limit the scope to apply `MDiff` on to only a paragraph a time. This is analogous to an idea in 3D game designing that you don't have to render the entire 3D scene and maintain it at all time. You just have to render as large as the player's camera can capture and only when the camera does captures it. This optimization can reduce the computational cost of the application by a factor as large as several hundreds inpractice.

The key idea behind implementing Objective 1, and the paragraph matching explained above, is determine textual similarities. There is a well-established technique called Min-hash which is used to calculate a text document's signature, which can in turn be used to compare with another document's signature to determine the similarity between the two documents. Of course, we here are looking at the similarity between two documents rather than two paragraphs. We used a third-party library [1] which can return the similarity of two text strings based on the Min-hash algorithm.

There are more subtleties to be clarified in this scheme. But we are leaving further elaboration for later part of this document.

1.3 Running Instruction

By this point, enough information is provided to understand what this project hopes to deliver. In this section, we provide step by step instructions so that you can run the application now and see the results before going further.

2 First Step: Finding the Most Similar Paragraph

2.1 Modelling

As described in the previous section, our first goal is to be able to inspect a certain file i in the folder, and immediately locate the most similar paragraph amongst all files in the folder, under assumptions:

1. File name indicate each file's last-modified-by time in an increasing order.
2. The author of the files knows to separate each section, or paragraph of his files by an empty line.

The second assumption is established so that there is a valid way to define the notion of ancestry. If we treat an entire document as one single paragraph, then there is no point in saying *finding the predecessor of this entire document*. Because in practice, most modifications between files are minor and of comparable degree, which means different successors of the same predecessor document may end up looking very much alike. Even though we devise an algorithm to locate such an predecessor document for the document being inspected, the output is most likely to be inaccurate and uninformative.

After dividing a document into paragraphs, the notion of ancestry and the concept of reusing now makes sense. Think it this way: when you write code, if you say *reuse*, you usually refers to reusing a part of a code file, usually a function, or block. But in the case you want to reuse the code file in its entirety, you just `import` or `include` it. Such file-wise reusing operations would normally indicate the source of reusing directly and an dedicated algorithm to solve the ancestry relationship would be unnecessary. The same analysis applies to the case of general text files. We are only interested in finding the predecessor of a snippet of a text file, which is a reused part. We conveniently define such a concept of reused part as a paragraph. There can be other choices of delimitation of reused parts, but obsession over this definition is moot per se.

Now, to find the most likely predecessor of a paragraph, is to find a paragraph that:

1. is contained in a file that is lastly modified earlier than the file that contains the paragraph under inspection.
2. shares, amongst all paragraphs that satisfies the previous condition, a highest similarity with the paragraph under inspection.
3. owns a similarity with the paragraph that satisfies the previous condition that is higher than a lower threshold.

And to find such a paragraph that satisfies the above three conditions, we use the theory of textual similarity.

2.2 Textual Similarity and Textual Similarity

Since this is only a subroutine we resort to in this project, rather than the project itself's primary focus, the introduction here is brief. Minhash is a hashing scheme that can hash each string or piece of text to a value that is necessarily short but informative enough to capture the uniqueness of the text. This value is called the *signature* of this text. Minhash is designed to be able to hash similar texts to similar signatures. And by a corresponding comparison function of two texts' respective signature, the two texts' similarity can be determined.

2.3 Implementation

Back to the implementation of the application. In particular, we implement the process as follows:

1. Read in all files in the folder in the order of increasing lastly-modification time, which is determined by the filenames. Create two string arrays, one to store the filenames of all the files in question and one to store each file's content of all files.
2. For each content string of the file content array, break up the content string into an string array of storing this file's lines. By doing this, we have a 2-dimensional array of `filelines[m][n]` where `m` stands for the number of files involved, and `n` stands for each file's number of lines in its content. Incidentally, we trim out the whitespaces at the head and tail of each line.
3. Using this 2-dimensional array `filelines[m][n]`, we scan all the lines of all files in the order of an intuitive 2-dimensional iterative order:

```
for (int i = 1 .. m) {  
  for (int j = 1 .. n) {  
    <process>  
  }  
}
```

In this scanning, we break up all the lines into paragraphs by the delimiter of *emptyline* as mentioned before. For each paragraph, we iterate through all paragraphs discovered before it, and find the one paragraph with the highest similarity to it, which has to be above 80%. We determine this paragraph as the **predecessor** of current iteration's paragraph. Relevant information are all stored.

4. Once we know the predecessor for each of all paragraphs of all files in the folder, we also maintain the edit distance produced by the improved `MDiff` algorithm with each paragraph. More specifically, this edit distance is calculated against each paragraph's predecessor with itself. By now, we have all the information we need to implement the application layer features in the future. For each paragraph, the information, in summary, typically consists of:
 - The filename of the file that contains this paragraph. Note that we are being a little simplistic here: we assume that each file has a filename in the format of $i. <extension> \forall i$. This assumption does not harm the generality of application for the project and can be easily extended if necessary.
 - The start and ending line numbers of this paragraph in the file that contains it. With these line numbers and the filename stored in the above entry, the paragraph itself now can be uniquely located.
 - A pointer to the paragraph's **predecessor**, which is determined by a concept of *maximum similarity so far* as described above.
 - Annotation to represent the result of running `MDiff` with this paragraph's predecessor as the first argument and this paragraph itself as the second argument. Note that although we are trying to accomplish something that looks like multi-file `diffing` here, the `MDiff` algorithm itself only takes two strings at a time. The annotation consists of three parts. We will illustrate each part's meaning with this example's help:

```
<a><b>  
- -
```

```
<b><a>  
+ +
```

- The `boolean same` value, which is `true` if the two string arguments are the same, which means the edit distance is 0. In the case of the example shown here, this flag should be `false`.
- The `boolean[] delete` flag array. The explanation is left to below.
- The `boolean[] add` flag array. Remind yourself that although we are introducing the information maintained for each paragraph, the `delete` and `add` flag arrays actually apply to this paragraph's predecessor and this paragraph itself. Since for each paragraph, its predecessor can be uniquely identified.

The `delete` boolean flag array corresponds to the predecessor string, which is `<a>` in this case. The flags in this array indicates whether each character in the string would be kept so that the predecessor string could be edited into the target successor string with minimum editing distance. In particular, each flag is `false` if the corresponding character in this paragraph's predecessor paragraph should be deleted in min-distance editing. In this case, the `delete` array should be `{true, false, true, true, false, true}`. Note that this information is stored with the paragraph `<a>`, rather than the paragraph (string) `<a>`.

Similarly, `add` boolean flag array corresponds to the new, or successor string/paragraph of the pair. If `add[i]` is `true`, then the character at 0-based position i will be seen as an unedited character (certainly not deleted, yet is not also added to the new paragraph) when trying to edit this paragraph's predecessor into this paragraph itself with minimum distance. If `add[i]` is `false`, then the character at position i will be considered newly added during the min-distance editing. In this example, the `add` array, of a value `{true, false, true, true, false, true}`, should be maintained with the paragraph `<a>`.

References

- [1] `codelibs/minhash`: A third-party Java-based library for b-bit MinHash algorithm