Project Proposal
Better `diff`

Guoye Zhang, Qiang Zhang

1 Introduction

`diff` is a commonly used utility. It has been widely adopted in spreading circumstances of file comparison and patch generation. Furthermore, `diff` is the basis of version control systems including `git`.

What `diff` solves is essentially the *longest common subsequence* problem. The algorithm `diff` employs is dynamic programming in spirit, targeted at finding least number of line changes that can lead from the old file to the newer version. Usually, `diff`, does a pretty good job in solving this problem and deserves all its compliments. But from time to time, `diff` does not always produce the perfect, or the more intuitively desirable solution. For lack of a better example, here is a case where `diff` only manages to find suboptimal solution:

```
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");
}
```

And when we try to delete the second block entirely, what we expect would be:

```
  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");
  }
```

But what we have instead is actually:

```
  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");
  }
```

The above solutions are equivalent in effect, but can be confusing regarding the purpose of revealing editing history. This is only a primitive example that we managed to find shortly that is reproducible. In real world applications, a similar counter-intuitive patching at a larger scale can be considerably frustrating and counterproductive.

We intend to devise a new `diff` strategy (including algorithms and implementation) that takes scoping issues into consideration. Also dynamic programming is not the best choice of solve in our opinion since dynamic programming is incapable of solving NP-complete problems. We intend to reduce the problem into a MAX-SAT problem for which numerous smart heuristics have been researched and implemented.

We also intend to extend the utility's functionality so that it can detect moving and copying.

The basic guideline of our design would be adding an intermediate layer between the original text file and the result of `diff`. We want our algorithm to be able to produce a solution of line changes that does not break the block-/scoping structure of the original file. Our utility would first recognize all the blocks, subject to different definitions regarding different kinds of text, in the file. We apply a hashing to each block and use the hash value as the basis of duplicate detection. Using hashing algorithms like locality-sensitive hashing, our solver would try to match a block in the old version to a block in the newer version with a probability that is highly positively associated with the similarity between the two blocks. By this heuristic, we will be more likely to find a line changes solution that preserves the file's block structure as much as possible.

Now that we are able to locate the newer version of some block of the older version file, we have to look into the original files and find the editing changes within the blocks on a line-by-line basis. There are currently many heuristics for this problem, including the one currently `diff` is using. Although we are not final on the choice of this, MAX-SAT seems very auspicious.

2 Dealing with Source Code

Source code files are relatively easier to process amongst all sorts of text files because block structure in code files are usually well defined and even well labeled. We need not calculate the hash for each block in this situation as we can just use some technique to extract the block name (like method name in Java) as the key for block matching.

One subtle issue would arise when overloading comes into place. There will be a situation where the user just changed a 2-arg method into a 3-arg method. There will also be a situation where a user did not change the 2-arg method, but rather just added a 3-arg one. Our algorithm should be able to distinguish between such situations. One plan would be to match blocks by block name first. Then compare the number of blocks of the same name in each file. If The numbers are equal, then we directly match the two blocks, without further differentiating more information about the two blocks. But if the number of blocks under the same name are not equal in the two files, then additional information are needed for matching the two files. We further take into consideration the entire signature/header line of the block, and treat the block in the new version file that has the same signature line as the modified version of the same block in the original file. The other block in the new version file that has the same name but different signature line shall be treated as additions.

We realize that there will be other issues that may come up and break our algorithm. Thus as tempting as it is to skip hashing the block body lines altogether for code files, we might still end up implementing some minimal version of hashing for the block body lines.

3 Dealing with Non-Code Files

Non-code files are regular files which could be an article, or letter. The tricky part in such situations is to define and implement an efficient but reasonable way to delimitating different blocks in the file. Natural block delimitation based on paragraphs seems reasonable, but sometimes paragraph information may not be available. We are still contemplating about the right way to `block-ize` the file in this situation.

Once proper block structuring scheme is established, the problem remaining is devising a good way to compare blocks in such files. Based on the previously mentioned LSH algorithms, we hope to develop a strategy to hash blocks so that blocks in different files are more likely to be matched together when their hash values are close.

Two blocks in two files having equal hash values should be interpreted as mere copying or moving.

4 Summary

Our primary goal is to improve `diff` in such a way that the algorithm no longer tries to match lines in different files on a basis of line positions (line numbers), but rather on their aggregate attribute, namely, on lines' relative relationship between each other. By doing so, our version of `diff` would also be able to recognize editing changes like copying and moving, which is transparent to current implementation of `diff`.

We are advised by Prof. Eisner that our algorithm should be implemented in a tool that can trace text migration and inheritance between different files. The tool should be able to display these changes/differences and the inheritance relationship in a friendly GUI. Our plan for such a tool's design is not ripe yet, but the first intuition is to other than showing the line difference results (including moving and copying) between the two different files, we also add some additional visuals to be able to display the temporal relationships between multiple files.

A naive idea would be to directly use the file modification timestamp as the basis of judging the temporal relationships between two blocks in two files that are recognized as matching. The problem with this idea is, block 1 in file 1 might be modified earlier than block 2 in file 2, but if we tried to modify some part of file 1 other than block 1 later after we finished editing block 2 of file 2, the file modification time would not be a solid basis for judging the temporal precedence between the two blocks.

A potential future direction is to tap into the full power of text editors or IDEs. By recording the undo stack of text editors, we will be able to understand exactly how user produces a text file. Whether a line is typed, copied, or moved is simple to distinguish. Of course this is out of scope for our current project.