The ndiff File Comparison Utility

Michael Corley, R. Sekar
Department of Computer Science
Stony Brook University
Stony Brook, New York, 11794

1 Introduction

Computing the differences between two files is at the core of many applications. Perhaps one file is a newer version of the other file. Or maybe the two files started out as identical copies but were changed by different people. The tool of choice for comparing the content of two text files in Unix is the `diff` file comparison utility which shows the differences between two files by outputting differences line by line. With this output format, each newline character acts as a delimiter separating the files into blocks, where each block is a line. This means that `diff` works very well for files containing source code (which tends to have short lines, with one statement per line) but not so well for other file types.

By default, `diff` notices any differences between the two text files, no matter how small. This could be as simple as a space character being changed into a tab character from one file to the next. This unfortunately is also one of its biggest limitations. Take for example a technical paper or a book, organized into paragraphs. Here each paragraph is essentially a very long line and performing a file comparison with `diff` produces nearly useless output, since a change of a single word (or even letter) will cause it to detect and report a line difference. Examining two very long lines to spot a difference of just a single word or character is often a very difficult task.

An extension of this is to compare lines of unequal length. Imagine a file with only two lines: ***line A*** is composed of 1000 words, and ***line B*** is composed of 100 words. Now, if a single word in ***line A*** changes, `diff` will report that half the data in the file has changed (which is of course true, since half the lines are "different"). In reality however, the difference is far smaller (a single word in 1100).

This paper proposes `ndiff`, an alternative file comparison utility that applies to a broader range of files to produce high-quality and natural outputs. The underlying algorithm is analyzed and compared with GNU `diff`. An experimental evaluation on both algorithms is provided along with the source code to our implementation of the algorithm written in C++.

2 Design

2.1 Text segmentation

The first step of our file comparison algorithm is to perform text segmentation. This is the process of dividing the files into meaningful units that we can operate on and compare against. Comparing by individual characters produces the finest level of detail and gives us the most control, but takes the longest to execute the comparison due to the larger number of tokens. Comparing by word boundaries or line breaks is faster and produces fewer individual edits, but the total length of the edits is larger. The required level of detail

varies depending on the application. For instance, comparing source code is generally done on a line-by-line basis, whereas comparing an English document is generally done on a word-by-word basis, and binary data or DNA sequences is generally done on a character-by-character basis.

Most of the issues with the quality of output given to us by `diff` result from it using line breaks to divide up the files. So instead we create tokens using word boundaries. *Tokens* are strings of characters defined by regular expressions and are partitioned into words or operators. In order to convert the sequence of characters of the files we want to compare into a sequence of tokens, we define a general specification to work with the lexical analyzer generator `flex`. Each rule in the specification consists of a regular expression, and defines the set of possible character sequences that are used to form individual tokens.

A hash function or checksum can be used to reduce the size of the string values in tokens in the sequences leading to more efficient comparisons. Additionally, the randomized nature of hashes and checksums would guarantee that comparisons would short-circuit faster, as lines of source code will rarely be changed at the beginning.

There are three primary drawbacks to this optimization. First, an amount of time needs to be spent beforehand to precompute the hashes for the two sequences. Second, additional memory needs to be allocated for the new hashed sequences. However, in comparison to the naive algorithm we use, both of these drawbacks are relatively minimal. The third drawback is that of collisions. Since the checksum or hash is not guaranteed to be unique, there is a small chance that two different tokens could be reduced to the same hash. This is unlikely in source code, but it is possible. A cryptographic hash would therefore be far better suited for this optimization, as its entropy is going to be significantly greater than that of a simple checksum. However, the setup and computational requirements of a cryptographic hash may not be worth it for small sequence lengths.

We also consider identified tokens that are exceptionally long. If we identify a sequence of characters with a length greater than some δ, we want to break it up into multiple tokens, rather than have one large token. Unusually long tokens present the same issues we encounter when using line breaks as a delimiter. One subtlety we need to handle is maintaining uniqueness to these long tokens. That is, we should recognize the difference between a large token broken into two segments, and an individual token, that might have of the same text value as one of the newly created segments. To handle this, we can assign a unique sentinel character to each token we segment, and concatenate it to the text value of each newly created segment. This solution also allows for us to maintain the information of which segments are the parts to get back the whole.

2.2 Anchors

When comparing two files, `diff` finds sequences of lines common to both files, interspersed with groups of differing lines called *hunks*. Comparing two identical files yields one sequence of common lines and no hunks, because no lines differ. Comparing two entirely different files yields no common lines and one large hunk that contains all lines of both files. In general, there are many ways to match up lines between two given files. `diff` tries to minimize the total hunk size by finding large sequences of common lines interspersed with small hunks of differing lines.

This also seems like a natural approach which mirrors what we do as humans in trying to understand what are the relationships between regions of the two input files. We first identify those regions which remain unchanged between the two files, in order to give us a sort of pivot around which other changes are detected and classified. Within `ndiff`, we refer to these common regions as *anchors*.

Given two texts T_1 and T_2, the concatenation of these two texts is $T = T_1||T_2$. An anchor is a long substring in T which occurs twice. Suppose a is an anchor of length l in T which occurs at the starting positions i and j, i.e. $T_i \dots T_{i+l-1} = T_j \dots T_{j+l-1}$. To reduce redundancy, we restrict attention to maximal anchors. Anchor (l, i, j) is maximal if $(T_{i-1} \neq T_{j-1})$ and $(T_{i+1} \neq T_{j+l})$ (whenever these positions exist in T).

2.2.1 Finding Anchors

The problem of finding anchors between two files can be reduced to finding long substrings that are common to both files. One could use many different algorithms to search for long common strings in a text file. Candidates include using binary search trees, suffix trees, suffix arrays, or a hashing based approach such as Karp and Rabin's method of fingerprints. We have carefully looked into each approach and in the end decided to use suffix arrays.

Suffix arrays contain most of the same information in a suffix tree, but are a simpler and more compact data structure. Compared to binary search trees, suffix arrays allow for faster queries and utilize less storage, especially for texts with a large number of suffixes. Compared to hash tables, suffix arrays allow for faster worst case queries, and avoid problems caused by collisions and computing hash functions.

Let us store the suffixes of a text T in lexicographical order in an intermediate array. Then the suffix array is the array that stores the index corresponding to each suffix. For example, if our text is *banana*\$, the intermediate array is [\$,*a*\$,*ana*\$,*anana*\$,*banana*\$,*na*\$, *nana*\$] and the suffix array is $[6, 5, 3, 1, 0, 4, 2]$. When there are multiple texts in question, we can concatenate them with $\$_1, \$_2, \ldots, \$_n$. Since suffixes are ordered lexicographically, we can use binary search to search for a pattern P in $\mathcal{O}(|P| \, log \, |T|)$ time. We can reduce the search time with very little construction time overhead by coupling the suffix array with LCP's. By definition, $LCP(i, j)$ is the length of the longest common prefix of the suffixes specified in positions i and j in suffix arrays sorted order.

We compute the length of the longest common prefix between neighboring entries of the intermediate array. If we store these lengths in an array, we get what is called the LCP array. These LCP arrays can be constructed in $\mathcal{O}(|T|)$ time, a result due to Kasai et al. In the example above, the LCP array constructed from the intermediate array is $[0, 1, 3, 0, 0, 2]$. Using LCP arrays, we can improve pattern searching in suffix arrays to $\mathcal{O}(|P| + log \, |T|)$.

The suffix array used by the `ndiff` algorithm is constructed with the DC3 (Difference Cover 3) divide and conquer algorithm. It has a time complexity of $\mathcal{O}(|T| + sort(\Sigma))$, where $|T|$ is the size of the concatenated files. We closely follow the exposition of the paper by Karkkainen-Sanders-Burkhardt that originally proposed the DC3 algorithm. Here we give a description of DC3.

1. Sort the alphabet Σ. We can use any sorting algorithm, leading to the $\mathcal{O}(sort(\Sigma))$ term.
2. Replace each token formed from the text with its rank among the tokens from the text. Note that the rank of the token depends on the text. For example, if the text contains only one token, no matter what token it is, it will be replaced by 1. This operation is safe, because it does not change any relations we are interested in. We also guarantee that the size of the alphabet being used is no larger than the size of the text (in cases where the alphabet is excessively large), by ignoring unused alphabets.
3. Divide the set of tokens composing the text T into 3 parts and package triples of tokens into megatokens. More formally, form T_0, T_1 , and T_2 as follows:

$$T_0 = < (T[3i],\ T[3i+1],\ T[3i+2]) \text{ for } i = 0, 1, 2, \ldots >$$
$$T_1 = < (T[3i+1],\ T[3i+2],\ T[3i+3]) \text{ for } i = 0, 1, 2, \ldots >$$
$$T_2 = < (T[3i+2],\ T[3i+3],\ T[3i+4]) \text{ for } i = 0, 1, 2, \ldots >$$

   Note that T_i's are just texts with $n/3$ tokens of a new alphabet Σ^3. Our text size has become a third of the original, while the alphabet size has cubed.
4. Recurse on $< T_0,\ T_1 >$, the concatenation of T_0 and T_1. Since our new alphabet is of cubic size, and our original alphabet is pre-sorted, radix-sorting the new alphabet only takes linear time. When this recursive call returns, we have all the suffixes of T_0 and T_1 sorted in a suffix array. Then all we need is to sort the suffixes of T_2 , and to merge them with the old suffixes to get suffixes of T.
5. Sort suffixes of T_2 using radix sort.
6. Merge the sorted suffixes of T_0, T_1, and T_2 using standard linear merging.

We now briefly mention how we can compute all LCP values needed in linear time with a scan over the suffix array. Consider a suffix array SA$[1, \ldots, n]$. If we were to compute the LCP of SA[1] with SA[2], then SA[2] with SA[3], etc., we'd end up with an $\mathcal{O}(n^2)$ algorithm. Instead, we proceed in a different order, which allows for a linear-time algorithm.

Let $P[i]$ be the j such that SA$[j] = i$. That is, $P[i]$ is the rank of the ith suffix. This can be computed from SA in linear time. Now compute the LCP of SA$[P[1]]$ with SA$[P[1]-1]$ by token-by-token brute force. Suppose this is k. If $k = 0$, then this took constant time and we move on. Otherwise $k > 0$. Now compute the LCP of SA$[P[2]]$ with SA$[P[2]-1]$. We know that this LCP is at least $k-1$ so we skip these tokens and continue token-by-token as before. And continue with the third suffix and so on.

The total work spent on finding mismatched tokens is $\mathcal{O}(n)$, since each mismatched token pair moves us to the next LCP computation, and there are only $n-1$ LCP computations to perform. Thus the total computation time is $\mathcal{O}(n)$.

2.2.2 Discarding Anchors

We need to distinguish between when are we finding commonalties between the two files that exist because we're looking at such small substrings, the probability that they will repeat is high, rather than because these are two duplicate regions. We need to have a high confidence that those anchors selected are not coincidental matches. At the same time, we would like the set of anchors to also be sufficient, in that they provide a reasonably well coverage of the files in question. Another way to think about this is to decide where we can "draw the line".That is, we want to partition A, the set of all anchors we identify, into two disjoint sets, $A_{keepers}$ and $A_{throwbacks}$. Thresholding is one of the simplest methods of drawing a line in the set of anchors.

During the thresholding process, individual anchors are marked as "significant" if their value is greater than some threshold value (assuming an anchors to not exist by chance) and as "insignificant" otherwise. If an anchor is significant, we keep it, and if it is insignificant, we throw it back. This convention is known as threshold above. The key parameter in the thresholding process is the choice of the threshold value. Users can manually choose a threshold value, or a thresholding algorithm can compute a value automatically. A simple method would be to choose the mean or median anchor length, the rationale being that if an anchor is significant, it should also be more significant than the average. A more sophisticated approach might be to create a histogram of the anchor lengths and use the valley point as the threshold. The histogram approach assumes that there is some average lengths for both the significant and insignificant anchors, but that the actual anchor lengths have some variation around these average values. However, this may be computationally expensive, and histograms may not have clearly defined valley points, often making the selection of an accurate threshold difficult. One method that is relatively simple and does not require much specific knowledge of the anchors is the following iterative method:

1. An initial threshold (T) is chosen, this can be done randomly or according to any other method desired.
2. The anchors are segmented into significant and insignificant anchors as described above, creating two sets:

$$G_1 = a(l,i,j) : a(l,i,j) > T \text{ (significant anchors)}$$
$$G_2 = a(l,i,j) : a(l,i,j) \leq T \text{ (insignificant anchors)}$$

3. The average of each set is computed. = average value of

$$m_1 = \text{average value of } G_1$$
$$m_2 = \text{average value of } G_2$$

4. A new threshold is created that is the average of m_1 and m_2.

5. Go back to step two, now using the new threshold computed in step four, keep repeating until the new threshold matches the one before it (i.e. until convergence has been reached).

This iterative algorithm is a special one-dimensional case of the k-means clustering algorithm, which has been proven to converge at a local minimum which means however, that a different initial threshold may give a different final result.

Our approach combines some ideas from the histogram idea and iterative algorithm described above. We first introduce some terminology. Anchors can be partitioned into two classes: ***self-anchors*** and ***cross-anchors***. Self-anchors represent the common substrings found when considering a file with itself. They give us a way to express self similarity by providing a measure on the distribution of common substrings for that file. By getting an idea of how similar two files are with themselves, we can get a better feel for the statistical significance of cross-anchors (common substrings between two files). This will help single out any coincidental matches and give us a basis on where we can draw the line.

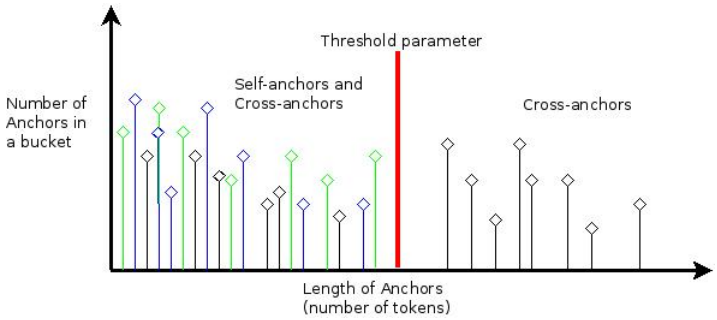

Figure 1: Histogram of anchors

The intuition behind this approach can be seen from Figure 1. Cross-anchors that live amongst the self-anchors we can say exist by chance. We discovered them only because of their short lengths and because there is a high probability they exist. The anchors that live just beyond this towards the tail, however are much more interesting. So by first finding all of these self-anchors, we set the stage for being able to safely discard cross-anchors, which are the anchors we really care about.

2.3 Difference Algorithm

Once the pre-processing of the files is complete, the remaining tokens are compared with a difference algorithm. A brute-force technique would take $\mathcal{O}(F_1 * F_2)$ to execute (where F_1 and F_2 are the lengths of each input). This approach sacrifices space efficiency for simplicity of implementation. Since this is clearly unscalable in practical applications where the text lengths are arbitrary, much research has been conducted on better algorithms which approach $\mathcal{O}(F_1 + F_2)$.

Any difference algorithm could theoretically process any input, regardless of whether it is split by characters, words or lines. However, some difference algorithms are much more efficient at handling small tokens such as characters, others are much more efficient at handling large tokens such as lines. The reason is that there are an infinite number of possible lines, and any line which does not appear in one file but appears in the other is automatically known to be an insertion or a deletion. Conversely, there are only 80 or so distinct tokens when processing characters (a-z, A-Z, 0-9 and some punctuation), which means that any non-trivial text will contain multiple instances of most if not all these characters. Different algorithms can exploit these statistical differences in the input texts, resulting in more efficient strategies. An algorithm which is specifically designed for line-by-line differences is described in J. Hunt and M. McIlroy's 1976 paper.

Arguably the best general-purpose difference algorithm is described in the papers *An O(ND) Difference Algorithm and its Variations* by Eugene W. Myers and in *A File Comparison Program* by Webb Miller and Myers. This algorithm solves the longest common subsequence (LCS) problem which is also what the

operation of `diff` is based on. In this problem, you have two sequences of tokens and you want to find a longest sequence of tokens that is present in both original sequences in the same order. That is, you want to find a new sequence which can be obtained from the first sequence by deleting some tokens, and from the second sequence by deleting other tokens. You also want this sequence to be as long as possible. From a longest common subsequence it's only a small step to get `diff`-like output. If a token is absent in the subsequence but present in the original, it must have been deleted. If it is absent in the subsequence but present in the second sequence, it must have been added in. This algorithm is $\mathcal{O}(ND)$ in time and space where N is the sum of the lengths of the two input files and D is the size of the minimum edit script for the two files. A refinement of the algorithm requires only $\mathcal{O}(N)$ space.

2.4 Post-processing Cleanup

2.5 Gap Penalty

During the development of this project the idea of using gap penalty as a way to avoid the problem of many small edits came up. In general, gap penalty is a technique used in some types of string-matching algorithms (mainly those that apply to DNA and protein sequences) to minimize the amount of gaps. This seemingly translates well to our problem, since having many small edits in a single line translates to having many small gaps.

Upon more research it became apparent that using the concept of gap penalty would not solve this. Basically, gap penalty is a construct used in string alignment - it minimizes the amount of gaps (not necessarily size) in a situation where multiple alignments are possible. Put another way, given a number of possible alignments, it favors the one with fewer gaps. This approach does not apply to our case, since we don't actually have multiple alignments. The LCS algorithm, when run on a large alphabet (ours is large since each word is a unique token, and `diff`'s is larger still because lines repeat less often than words), produces only a single alignment. For this reason a gap penalty approach would not yield anything.

In addition to this, we must remember that the gap penalty approach is a two-step method. This means that first sequences are determined, then in a post-processing phase alignment happens with taking gap penalty into account. In our case on the other hand, we are looking more for something that would take gaps into account as part of the main algorithm. Gaps detected via post-processing are not as useful to us, since if we intend to do any kind of post processing we can already do better than simply considering gap penalties (for instance the much simpler approach of counting the number of gaps on each individual lines and having a threshold for how many gaps are needed to classify the line as a new one as opposed to a changed one). Post-processing also allows us to do things like highlighting, and the more general case of making the output look like anything we want, which is much more powerful than what gap penalty can do.

The bottom line is gap penalties require post-processing, and we are able to do more useful things in post-processing than what gap penalties give us. Also, links are provided in the References section about gap penalties and how they are used.

3 Implementation

4 Evaluation

5 Conclusions

6 References