Git and the $O(NP)$ Myers diff algorithm

Razi Shaban
Swarthmore College
rshaban1@cs.swarthmore.edu

Abstract

This paper will introduce the **diff** algorithm used in Git , known as the $O(ND)$ Myers algorithm, and describe a more efficient implementation, the $O(NP)$ Myers algorithm. The $O(NP)$ algorithm, as well as the first **diff** algorithm, Hunt-McIlroy ($O(N^2)$), are implemented in an attached Python file. Our implementation of the Myers algorithm runs significantly faster than the implementation of Hunt-McIlroy for typical use cases.

1 Introduction

Most programmers use **diff** every day without recognizing the power of the tool they are using. Any sophisticated version control system will use a **diff** algorithm to find the difference between different versions of a file; this difference serves as an indicator of the difference between versions. Git, the version control system originally created to keep track of the development of the Linux kernel, works by keeping track of snapshots (commits) of a collection of files (a repository) at certain points in time. These snapshots summarize the state of each file in the folder at a certain version by keeping track of the differences between that file and the previous state of the file. The difference between two files is calculated by running a **diff** algorithm, which produces a patch file. This patch file contains an easily-parseable summary of the changes necessary to convert one version of a file to another. Git then uses the **patch** algorithm to apply the changes laid out in the patch to the old file, outputting the new file.

The **diff** program that ships with Git has four variant algorithms: **Myers**, **minimal**, **patience**, and **histogram**. The $O(ND)$ Myers algorithm is the original Git **diff** algorithm and remains the default, and in fact dates back to 1986 (Linus Torvalds first started developing Git in 2005). This paper will look at a later version of Myers' algorithm which improves the runtime to $O(NP)$.

2 Motivation

The **diff** algorithm is a critical function at the heart of Git and other version control programs, but it also has many other important uses. **diff** algorithms are used to determine the difference between different genomes (Myers is a bioinformatician). One particularly new development is the use of **diff** algorithms to detect the difference between a representative virtual Direct Object Model (DOM) and the rendered DOM in a web-page; using intelligent **diff** algorithms in this context allow for tremendous performance gains.

As such, it is essential that **diff** be a both accurate and efficient function. We will see below that there are several parameters for accuracy, but efficiency is clear: **diff** is a program that will be run many times by many other programs, so it must be as time efficient as possible without sacrificing accuracy.

3 Background

The Wagner-Fischer algorithm was the first algorithm widely used to calculate the edit distance between two sequences [Wagner and Fischer, 1974]. Wagner-Fischer is a dynamic programming solution that computes the Levenshtein edit distance between two strings by building up the edit distance between smaller subsequences of each of the strings, in typical dynamic programming style. Implemented in its original form, Wagner-Fischer runs in $O(N^2)$ time and space, as it builds an $N \times N$ array to compare every subsequence of increasing size with every other subsequence of increasing size.[1] While Hirschberg was able to improve Wagner-Fischer to run in linear space, his implementation still runs in $O(N^2)$ time [Hirschberg, 1975]. The Hunt-McIlroy algorithm modified the Wagner-Fischer algorithm to only look at

[1] An example of the Wagner-Fischer algorithm is implemented in the attached code.

candidates that are viable, thus employing a divide and conquer method to avoid filling out an entire $N \times N$ array [Hunt and MacIlroy, 1976]. This algorithm was shipped as the first **diff** algorithm with the fifth edition of Unix in 1974 [Zeidman, 2011], and remained the default **diff** algorithm until it was eventually replaced by the Myers **diff** algorithm.

Aho, Hirschberg, and Ullman proved via decision trees that any edit distance algorithm using equal-unequal comparisons has a $O(N^2)$ worst-case runtime, as it will require $O(N^2)$ equal-unequal comparisons [Ullman et al., 1976]. In the case of a more restricted alphabet, they further proved that $O(N \times S)$ comparisons are necessary, where S is the size of the alphabet used [Ullman et al., 1976]. Hirschberg went on to use information theory to prove that there must be at least $O(Nlog(N))$ comparisons in any decision tree that solves the lowest-common subsequence problem using less than-equal-greater than comparisons [Hirschberg, 1978].

Myers tried to work around this issue by focusing on the size of the longest common subsequence instead. In most use-cases of **diff** , the difference between the two sequences should be small relative to the sequences themselves. In a program, for example, the change could be several lines out of several thousand; in a DNA strand, the change could be a small mutation between instances. To accommodate this, Myers defines a parameter

$$D = 2(N - L),$$

where N is the length of the sequences being compared and L is the length of the longest common subsequence. As such, D is the length of the shortest edit script [Myers, 1986]. Given this, the expected runtime of Myers' diff algorithm is $O(N + D^2)$ and the worst-case is $O(ND)$. Myers claims that his algorithm runs two to four times faster than the Hunt-McIlroy algorithm, which was used as the Unix **diff** algorithm before it adopted the Myers algorithm [Hunt and MacIlroy, 1976].

In fact, Myers later published a paper improving the number of node accesses needed and slimming the algorithm from $O(ND)$ to $O(NP)$, where P is the number of deletions necessary [Wu et al., 1990]. Myers described the second algorithm as "nearly twice as fast as the $O(ND)$ algorithm", and "much more efficient when A and B differ substantially in length" [Wu et al., 1990]. This paper will examine the later algorithm.

4 Interface

The general **diff** algorithm can be conceived as follows:

Input: two sequences A and B

Output: an edit script consisting of a set of insertion and deletion commands that transform A into B.

Note that while the length of the two sequences need not be equal or similar, many of the **diff** algorithm variations are designed to optimize for sequences of similar length as that is the most common input. As such, all runtimes are described in terms of N, the length of the larger input sequence.

5 The Myers algorithm

The Myers algorithm is a greedy algorithm to compute the shortest edit script necessary to transform input sequence A into input sequence B. The algorithm works by dividing the two sequences, conceptually, into a longest common subsequence (LCS) and a shortest edit script (SES). The algorithm will find the shortest possible edit by finding the longest common subsequence.

An edit graph helps clarify both the difference between and interaction of the LCS and SES. Figure 1 below is the edit graph included in Myers' original $O(ND)$ paper, which illustrates the edit path from $A = abcabba$ to $B = cbabac$ [Myers, 1986].

In Figure 1 below, the different types of edges correspond to different edit behaviors. The path leading from $(0, 0)$ to any one point (x, y) represents the steps necessary to transform $A[: x]$ into $B[: y]$. A diagonal edge indicates a common subsequence, e.g. $a_x = b_y$. A horizontal edge represents the deletion of a symbol, and a vertical edge represents the insertion of a symbol. Thus the two horizontal edges from $(0, 0)$ to $(2, 0)$ corresponds to the deletion of the first two symbols in A. The diagonal edge from $(2, 0)$ to $(3, 1)$ corresponds to a common subsequence, c, in both A and B. The vertical edge from $(3, 1)$ to $(3, 2)$ corresponds to the insertion of a symbol, b, as this exists in B but not in A.

Thus we conceive of the diagonal edges as representing the common subsequences between A and B and the horizontal and vertical edges as representing the edit script. By maximizing the number of diagonal edges in our path from $(0, 0)$ to (m, n) (our total path) we minimize the number of

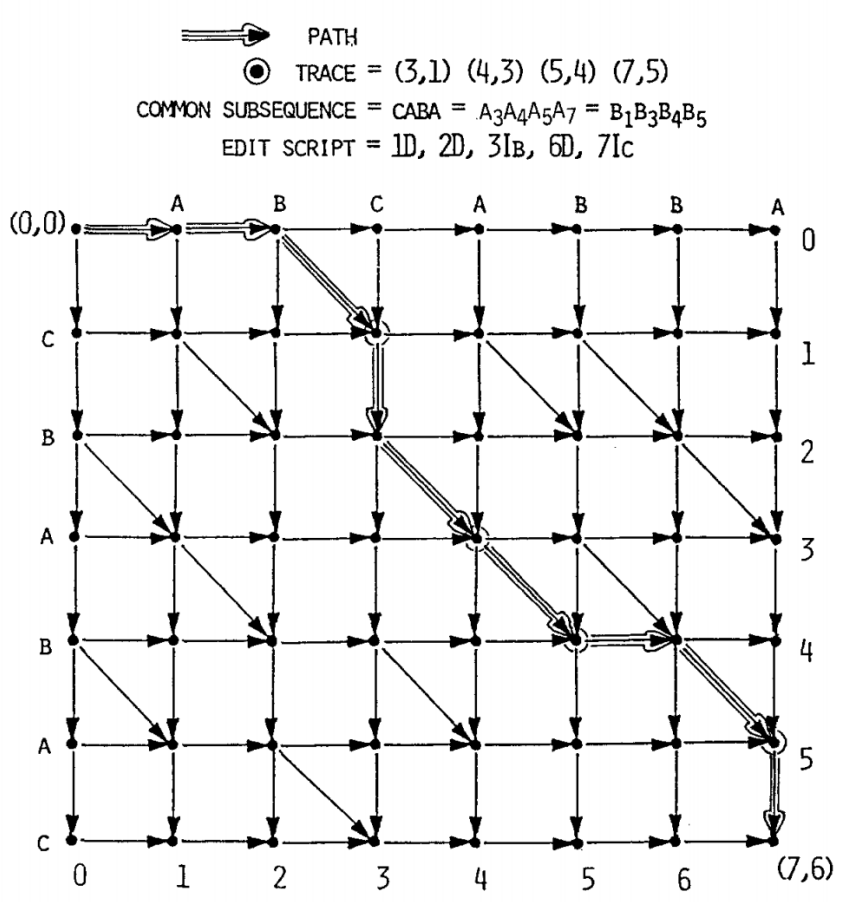

Figure 1: Edit graph from $A = abcabba$ to $B = cbabac$ [Myers, 1986]

vertical and horizontal edges; by extension, by maximizing the length of the common subsequence we are minimizing the length of the edit script.

The Myers algorithm thus attempts to find the total path with the most diagonal edges, thus ensuring the smallest possible edit script. In the original $O(ND)$ Myers algorithm, D was defined as the length of the shortest possible edit script. The second, $O(NP)$ Myers algorithm defines P as the number of deletions in the SES:

$$P = D/2 - (N - M)/2$$

.

The Myers algorithm defines the diagonal k to be the nodes on the edit graph where $k = y - x$. Thus the diagonals run from $k = -N$ to $k = N$, with $k = 0$ being the straight-shot diagonal from $(0, 0)$ to (N, N). A *snake* is a straight diagonal sub-section; the Myers algorithm includes a sub-function that simply traces a snake to its end-point. The algorithm expands between diagonals $-P$ and $\Delta - P$, where $\Delta = n - m$, iteratively increasing P until it reaches the end of the edit graph. By iteratively expanding the frontier of edit paths, the algorithm can find the shortest possible edit script. An illustration of exploring the two areas is in Figure 2 below [Wu et al., 1990].

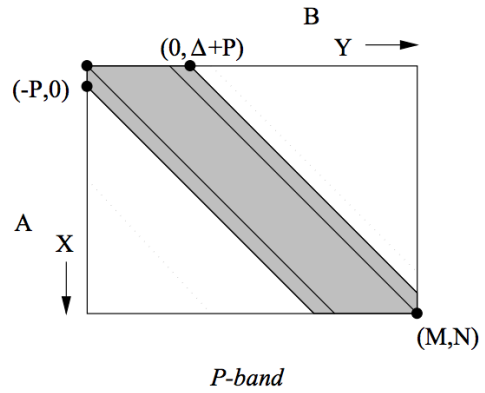

Figure 2: P-band for the $O(NP)$ Myers algorithm [Wu et al., 1990]

The value of P is defined as follows:

$$P(x, y) = \begin{cases} V(x, y), & \text{if } (x, y)\text{is below diagonal } \Delta \\ V(x, y) + (k - \Delta), & \text{if } (x, y)\text{is above diagonal } \Delta \end{cases}$$

where $V(x, y)$ is equal the number of vertical edges in the path from $(0, 0)$ to (x, y). Thus Myers derives the algorithm on the next page.

The expected running time of the Myers algorithm is $O(N + PD)$, while the worst-case running time is $O(NP)$ [Wu et al., 1990]. As P is less than half of D by definition, this is an improvement of at least a factor of two. Also, in the case that A is a subsequence of B (that is, the two sequences are identical), the running time of the algorithm is $O(N)$, which is a significant improvement over Hunt-McIlroy ($O(N^2)$ to confirm) and the first Myers algorithm ($O(ND)$ to confirm).

Algorithm 1 Myer's $O(NP)$ algorithm

```
1:  function MYERS
2:      fp[−(M + 1), (N + 1)] ← −1
3:      p ← −1
4:      repeat
5:          p ← p + 1
6:          for k ← −p to Δ − 1 do
7:              fp[k] ← snake(k, max(fp[k1] + 1, fp[k + 1]))
8:              for k ← Δ + p to Δ + 1 by -1 do
9:                  fp[k] ← snake(k, max(fp[k1] + 1, fp[k + 1]))
10:                 fp[] ← snake(, max(fp[1] + 1, fp[ + 1]))
11:             end for
12:         end for
13:     until fp[] = N
14:     return Δ + 2p
15: end function
16:
17: function SNAKE(k, y)
18:     x ← yk
19:     while x < M and y < N and A[x + 1] = B[y + 1] do
20:         x ← x + 1
21:         y ← y + 1
22:     end while
23:     return y
24: end function
```

6 Benchmarks

In a speed test run on sample data used to test Google **diff** algorithms, the Hunt-McIlroy algorithm ran in 0.02573 seconds while the Myers algorithm ran in 0.01149 seconds, less than half the time [Fraser, 2012]. This result is expected given the expected run-time of each.

The difference between the two was far more pronounced when diffing two identical files: the Hunt-McIlroy algorithm took 0.0307 seconds to detect that the files were the same, while the Myers algorithm terminated in 0.0000638 seconds, only slightly longer than it took Python to load the files.

When run against large, fairly random files that had almost no similarities (output logs from different iterations of a growing neural gas network), the algorithms actually ran in fairly similar time. The Hunt-McIlroy algorithm terminated in 34.870 seconds, while the Myers algorithm finished in 30.817. This reflects the fact that the Myers algorithm is optimized for small P values; when the total number of lines deleted approaches N, the Myers algorithm will approach $O(N^2)$.

Table 1: Actual runtime for algorithms on sample datasets, in seconds

Dataset	Hunt-McIlroy	Myers ($O(ND)$)
Google Test Set	0.0257	0.0114
Identical Files	0.0307	0.0000638
GNG logs	34.870	30.817

References

N. Fraser. Diff, match and patch libraries for plain text, Nov 2012. URL `https://code.google.com/p/google-diff-match-patch/`.

D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975.

D. S. Hirschberg. An information-theoretic lower bound for the longest common subsequence problem. *Information Processing Letters*, 7(1):40–41, 1978.

J. W. Hunt and M. MacIlroy. *An algorithm for differential file comparison.* Bell Laboratories, 1976.

E. W. Myers. Ano (nd) difference algorithm and its variations. *Algorithmica*, 1(1-4):251–266, 1986.

J. Ullman, A. Aho, and D. Hirschberg. Bounds on the complexity of the longest common subsequence problem. *Journal of the ACM (JACM)*, 23 (1):1–12, 1976.

R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Journal of the ACM (JACM)*, 21(1):168–173, 1974.

S. Wu, U. Manber, G. Myers, and W. Miller. An o (np) sequence comparison algorithm. *Information Processing Letters*, 35(6):317–323, 1990.

B. Zeidman. *The Software IP Detective's Handbook: Measurement, Comparison, and Infringement Detection.* Prentice Hall Professional, 2011.