The Myers Difference Algorithm for Version Control Systems

Daniel Hernandez

Minerva Schools at KGI

The Myers Difference Algorithm for Version Control Systems

Software engineering requires a lot of collaboration and iteration. A typical software development team will contain multiple engineers working on the same project at the same time, contributing new features, fixes, modifications, and documentation to the same code base. Version control system (VCS) software is one solution used to facilitate this widespread collaboration. VCS software such as git or mercurial allows for the creation of "branches", which stem from the original code base, and are a self-contained sandbox where an individual or a team can work independently on changes to the code base, without directly modifying the working code ("Git", 2019; "Mercurial SCM", 2019). Once the change is finished and thoroughly tested, the VCS allows for the "merging" of the branch with the original "master" branch, thus implementing the changes in the original code base. This allows for multiple teams to work on multiple features simultaneously without conflict. Furthermore, the states of each branch are saved each time code is committed to the branch, which allows teams to easily navigate between versions of the software – for example, to backtrack to an earlier version of an application where a bug was not yet present.

The "merge" operation, where one branch is merged with another branch, is an interesting computational problem. If branch A is being merged with branch B, where branch B contains modifications, the merge operation should be able to efficiently update branch A with the new information contained in branch B. Furthermore, if branch A has been merged with other branches since branch B was "branched out" from branch A, then the merge operation should also be able to tell whether the changes made in branch B would conflict with the changes that have been made to branch A since.

The solution to this is called the “diff” command, where “diff” stands for “difference”. The diff command can be used to compare two files and see the difference between both in terms of insertions and deletions (Hunt & Mcllroy, 1976). For example, if file A in branch A contained the lines “ABAB”, and file A in branch B (which we will call the new file B, for convenience) was modified to contain the lines “ABBAB”, then the difference command should be able to tell you the changes that were made to get from file A to file B. However, this requires some refinement, since there are many ways to list instructions that would get you from file A to file B: deleting all the lines in file A and then inserting all the lines in file B (delete “ABAB” and insert “ABBAB” for a total of 4 deletions and 5 insertions), deleting the second “A” from file A and inserting “AB” at the end of file B (for a total of 1 deletion and 2 insertions), inserting a “B” after the first “B” in file A (for a total of 1 insertion) – and the list could go on. The diff command finds the minimum number of insertions and deletions that are necessary to go from file A to file B – the “shortest edit script”. In our previous example, this would be the last case, where we insert a “B” after the first “B” in file A.

Using this command not only allows one to see the changes that were made to take file A to file B, but also to compare two different files to see if the changes that were made to them conflict with each other. This is what the “merge” operation between branches does. Furthermore, the command can also allow us to keep multiple versions of the same file without storing every single file that we have created, by only storing the edit scripts. If we wanted to go back from file B to file A, all we would need to do is apply the edit script that took us from file A to B in reverse. Storing edit scripts instead of entire files saves a lot of space and avoids duplicated information.

Thus, as we can see, the diff command has many interesting real-world applications. Furthermore, since it is used so often in the software development process, it needs a fast and efficient solution. The diff algorithm should be able to take two files as inputs, file A and B, and output an "edit script", composed of deletions and insertions, that take you from file A to file B. In this paper, we look at the Myers Difference Algorithm, an O(ND) algorithm that generates the shortest edit script between two files, and replicate the functionality of "git diff" in Python by implementing it.

The Myers Difference algorithm

In his paper, Myers describes an O(ND) difference algorithm, where N is the sum of the lengths of the strings being compared and D is the length of the edit script. He uses a technique reminiscent of the traditional dynamic programming solution to calculate edit distance but adds a greedy element to it, which drastically improves the complexity of the algorithm. He does this by modeling the difference problem between two files as a graph search problem (Myers, 1986).

Suppose we have two strings "abcabba" (string A) and "cbabac" (string B). If we input these into Myers' difference algorithm, we should obtain the shortest edit script that takes us from string A to string B. The traditional way to represent this problem is through an edit graph, such as the one below[1]:

[1] Figure taken from Myers' original paper, referenced at the end (Myers, 1986).

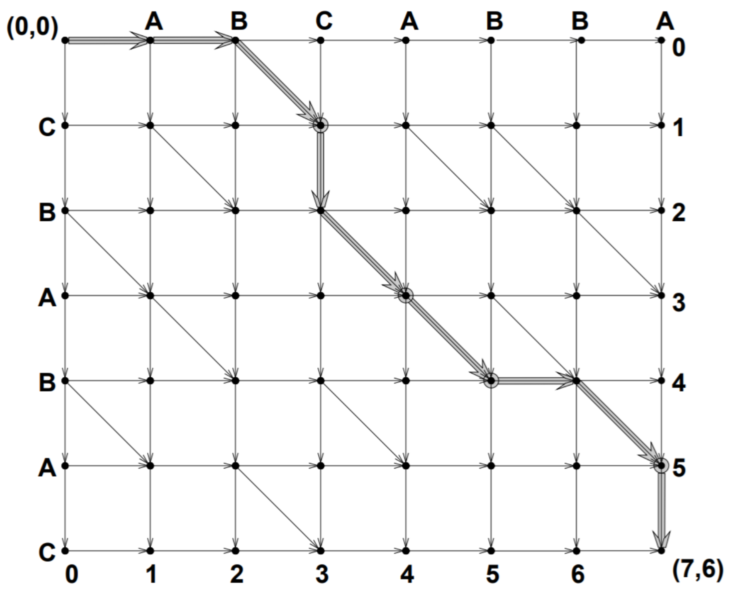


Figure 1. Edit graph of the strings “ABCABBA” and “CBABAC”.

We have string A at the top and string B down the side. Moving to the right in this graph represents a deletion. For instance, going from (0,0) to (1,0) represents deleting “A” from string A. Moving diagonally represents the fact that the two characters being compared are the same. For example, at (0,2), after deleting “A” and “B” from string A, we move diagonally to (3,1) because the “C” in string A is now in the same position as the first “C” in string B. Moving downwards represents insertion. When going from (3,1) to (3,2), we are inserting a “B” after our “C” in string A to match the “B” in string B. Once we reach the bottom-right corner of the graph, string A will be equal to string B.

Only horizontal and vertical moves are edits. Thus, if we minimize the number of vertical and horizontal moves (or equivalently, if we maximize the number of diagonal moves we make) we find the shortest edit script that takes us from string A to string B.

Here, Myers introduces the concept of “D-paths”. We define a D-path as a path starting at (0,0) with exactly D non-diagonal edges (thus, a 0-path consists of solely diagonal edges). Through simple induction, it follows that a D-path must consist of a (D-1)-path followed by a non-diagonal edge (thus making it a D-path), which is then followed by a (possibly empty) sequence of diagonal edges, termed a “snake”, since diagonal edges don’t add to the D-count.

Next, we number the diagonals in our edit graph k, where a diagonal k is defined as the diagonal for which every point (x,y) that lies on it satisfies the relationship $k = x - y$. Thus, the diagonal k = 0 is the one that goes from (0,0) all the way up to the bottom right corner, assuming we have a perfect square. Using this definition, it is possible to prove by induction that every odd D-path must end on an odd diagonal k, and every even D-path must end on an even diagonal k (Myers, 1986).

We can then define a furthest-reaching D-path on diagonal k if, of all the D-paths ending on diagonal k, it is the one that ends furthest from the origin, (0,0). It follows that furthest-reaching D-paths are obtained by extending furthest-reaching (D-1)-paths. If a D-path ends on diagonal k, it follows that the (D-1)-path ends on diagonal $k \pm 1$, depending on whether the connection between the D-paths is horizontal or vertical. The final “snake” of the (D-1)-path must be a maximal one, since if it was not, the D-path would not be furthest-reaching, and we could connect the D-path to a different further-reaching (D-1)-path with a horizontal or vertical move. Thus, every D-path can be decomposed into a maximal (D-1)-path, a vertical or horizontal move, and a possibly empty snake. This exhibits optimal substructure, and it also allows us to make the optimal “greedy” choice at each step, since we can greedily extend each snake at the end of a diagonal until we can no longer make diagonal moves to find our next maximal path.

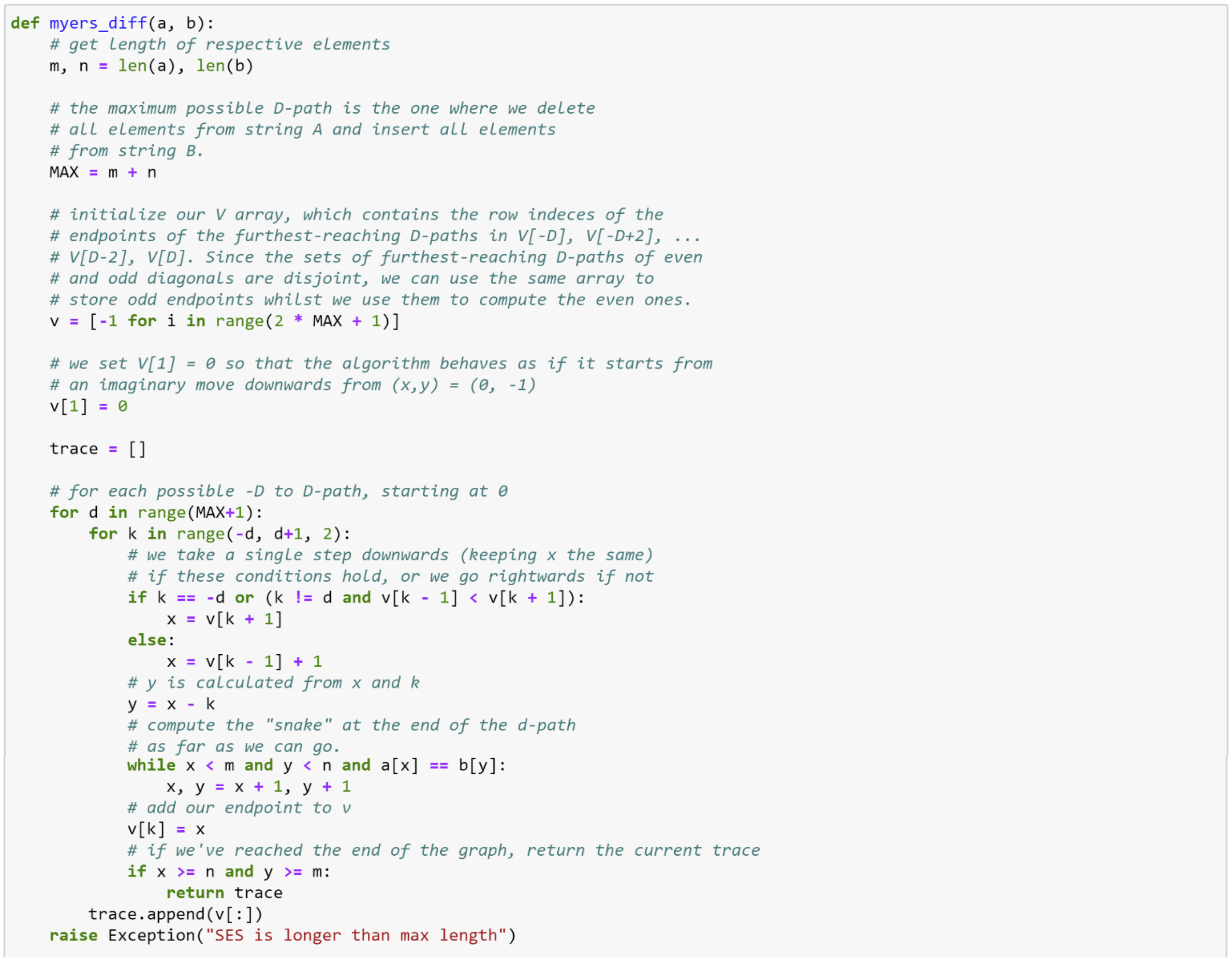
Thus, given the endpoints of the furthest-reaching (D-1) paths on diagonals $k \pm 1$, we can obtain the furthest reaching of the two, add an appropriate horizontal or vertical move to our current diagonal k, and extend the snake as far as possible. If we reach the end of the graph (bottom-right corner), we have found a solution for the shortest edit script. Additionally, since there are only D+1 diagonals on which a D-path can end, we can compute the endpoints of the relevant D+1 diagonals for increasing values of D (starting with the 0-path) until our furthest reaching path reaches the end of our graph, in a dynamic programming fashion.

Below is pseudocode for the implementation of Myers' Difference algorithm, where *N* and *M* are the lengths of our strings B and A, respectively (Myers, 1986):

```
For D ← 0 to M+N Do
    For k ← –D to D in steps of 2 Do
        Find the endpoint of the furthest reaching D-path in diagonal k.
        If (N,M) is the endpoint Then
            The D-path is an optimal solution.
            Stop
```

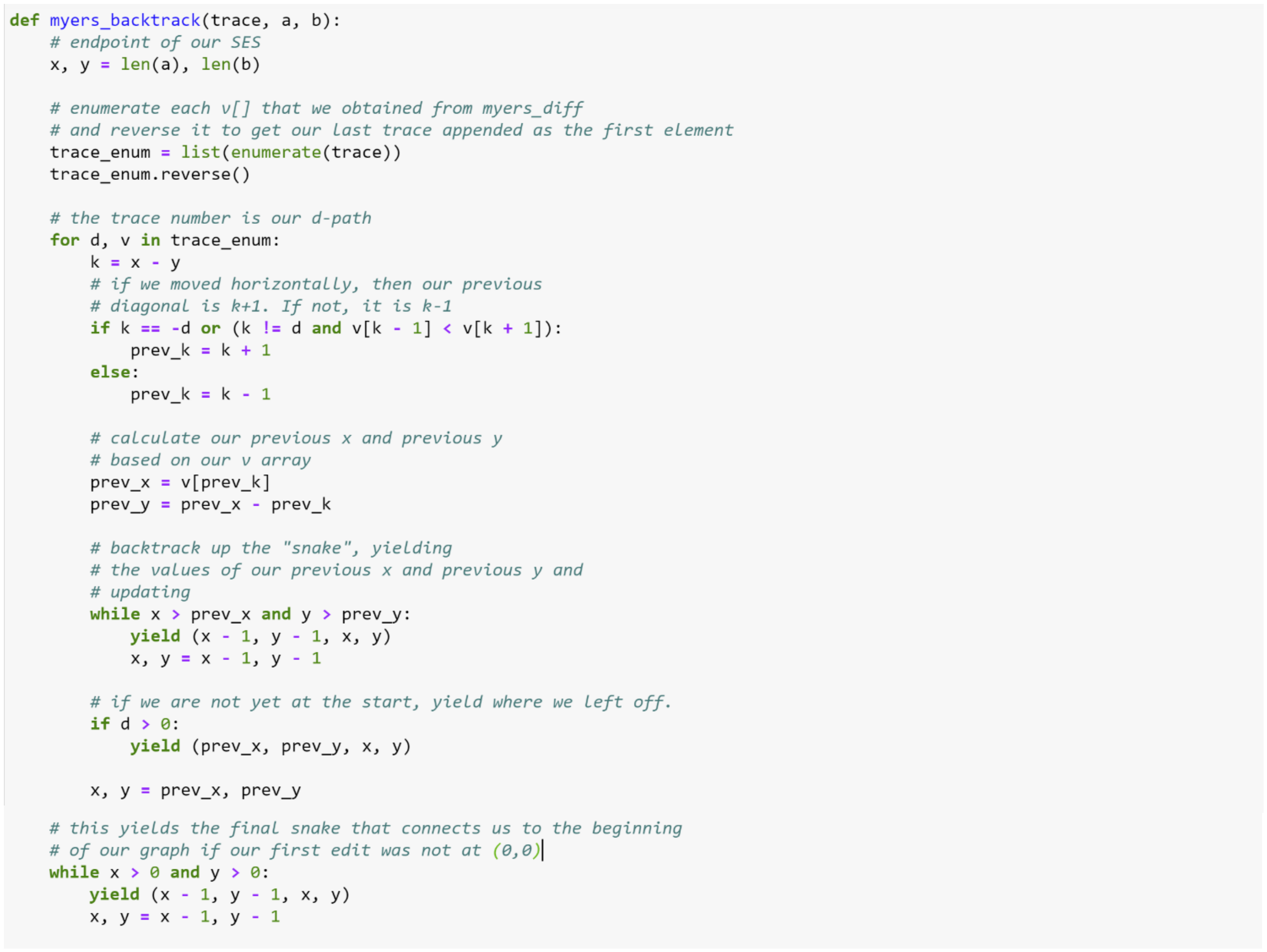
Python implementation

Since we have the pseudocode, implementing it in Python is relatively straightforward. Below is the implementation of the procedure $MYERS - DIFF$, which returns an array named “trace” that allows us to reconstruct the shortest edit script required to go from string A to string B. Trace will contain the x-coordinates of the endpoints of D-paths along diagonal k:



```python
def myers_diff(a, b):
    # get length of respective elements
    m, n = len(a), len(b)

    # the maximum possible D-path is the one where we delete
    # all elements from string A and insert all elements
    # from string B.
    MAX = m + n

    # initialize our V array, which contains the row indeces of the
    # endpoints of the furthest-reaching D-paths in V[-D], V[-D+2], ...
    # V[D-2], V[D]. Since the sets of furthest-reaching D-paths of even
    # and odd diagonals are disjoint, we can use the same array to
    # store odd endpoints whilst we use them to compute the even ones.
    v = [-1 for i in range(2 * MAX + 1)]

    # we set V[1] = 0 so that the algorithm behaves as if it starts from
    # an imaginary move downwards from (x,y) = (0, -1)
    v[1] = 0

    trace = []

    # for each possible -D to D-path, starting at 0
    for d in range(MAX+1):
        for k in range(-d, d+1, 2):
            # we take a single step downwards (keeping x the same)
            # if these conditions hold, or we go rightwards if not
            if k == -d or (k != d and v[k - 1] < v[k + 1]):
                x = v[k + 1]
            else:
                x = v[k - 1] + 1
            # y is calculated from x and k
            y = x - k
            # compute the "snake" at the end of the d-path
            # as far as we can go.
            while x < m and y < n and a[x] == b[y]:
                x, y = x + 1, y + 1
            # add our endpoint to v
            v[k] = x
            # if we've reached the end of the graph, return the current trace
            if x >= n and y >= m:
                return trace
        trace.append(v[:])
    raise Exception("SES is longer than max length")
```

Using the array of arrays that is returned (“trace”), it is possible to reconstruct the solution of edit scripts, since we know that horizontal moves represent a deletion, vertical moves represent an insertion, and diagonal moves mean that both characters are the same and no edits are needed. We do this via the $MYERS - BACKTRACK$ procedure:



```
def myers_backtrack(trace, a, b):
    # endpoint of our SES
    x, y = len(a), len(b)

    # enumerate each v[] that we obtained from myers_diff
    # and reverse it to get our last trace appended as the first element
    trace_enum = list(enumerate(trace))
    trace_enum.reverse()

    # the trace number is our d-path
    for d, v in trace_enum:
        k = x - y
        # if we moved horizontally, then our previous
        # diagonal is k+1. If not, it is k-1
        if k == -d or (k != d and v[k - 1] < v[k + 1]):
            prev_k = k + 1
        else:
            prev_k = k - 1

        # calculate our previous x and previous y
        # based on our v array
        prev_x = v[prev_k]
        prev_y = prev_x - prev_k

        # backtrack up the "snake", yielding
        # the values of our previous x and previous y and
        # updating
        while x > prev_x and y > prev_y:
            yield (x - 1, y - 1, x, y)
            x, y = x - 1, y - 1

        # if we are not yet at the start, yield where we left off.
        if d > 0:
            yield (prev_x, prev_y, x, y)

        x, y = prev_x, prev_y

    # this yields the final snake that connects us to the beginning
    # of our graph if our first edit was not at (0,0)
    while x > 0 and y > 0:
        yield (x - 1, y - 1, x, y)
        x, y = x - 1, y - 1
```

When we make a generator using the $MYERS - BACKTRACK$ procedure, we can determine whether we need to delete, insert, or do nothing based on the previous x and y values and the current ones, since we can determine whether we made a horizontal, vertical, or diagonal move. We then loop over the generator returned by the $MYERS - BACKTRACK$ procedure, as seen in the accompanying code, to do just that.

Note that Myers' algorithm can be extended to any kind of element that can be meaningfully compared. This means that to imitate the "git diff" command, all we have to do is implement a class that supports equality comparisons. For instance, if "a" and "b" in $MYERS - DIFF$ are arrays of Line objects, where each line can be compared to one another to test equality, then we could obtain an edit script for which lines were deleted and which lines were inserted to get from document "a" to document "b". This is the approach we take in the accompanying code.

Analyzing complexity

Time complexity

First, we will look at the complexity of the $MYERS - DIFF$ procedure, the algorithm itself. The inner for-loop, where we loop through each diagonal *k*, is repeated at most (D+1)(D+2)/2, since the outer for loop is repeated D+1 times and during the *k*th iteration the inner loop is repeated at most *k* times. All the lines inside the inner for-loop take constant time except the while loop, so, without counting the while loop, these lines are $O(D^2)$.

However, what reduces the complexity of the algorithm is the while loop. The while loop is iterated over once for *each diagonal traversed* in the extension of furthest reaching paths (the snakes). But at most $O((M + N)D)$ diagonals are traversed (where *M* and *N* are the lengths of strings A and B) because all D-paths lie within the diagonals -D and D, and there are at most $(2D + 1) * \min(M, N)$ points within this band of diagonals. Thus, the entire algorithm takes only $O((M + N)D)$ time, since the algorithm terminates after finding the shortest edit script once it has traversed the last diagonal.

For the $MYERS - BACKTRACK$ procedure, we only have one for-loop which iterates over the array "trace" that is returned by $MYERS - DIFF$. Thus, the complexity will be proportional to the length of the trace. Since we know that the complexity of the $MYERS -$

$DIFF$ procedure is $O((M+N)D)$, but that there is only one optimal D-path returned by the procedure, this means that the length of the trace is at most $M+N$, since we are appending to it at the end of each iteration of the outer for-loop. Thus, the procedure $MYERS-BACKTRACK$, which loops over the variable "trace", will have a complexity of at most $O(M+N)$. The while loop inside it does not add to the complexity since it simply follows the diagonal up to the next location in the trace. Another way to look at it is that all the backtracking procedure does is trace the optimal D-path from the start to the end, recording the type of movement that needs to be done at each step. Since we know that the worst possible D-path is the one that deletes the entire first string character by character and inserts the entire second string character by character, then backtracking will take at most $O(M+N)$, since the length of the worst possible D-path is $M+N$.

Space complexity

The space complexity of the $MYERS-DIFF$ procedure is essentially the complexity of storing the trace. We continuously copy the current state of the array V into the trace variable. However, we only do so D times, since the length of the optimal D-path will have D trace points associated with it (diagonals are not stored in the trace variable since we don't think of them as "turning points"). Since each V array has a length of $2*(M+N)$, the procedure has a space complexity of $O((M+N)D)$.

The $MYERS-BACKTRACK$ procedure simply uses the output variable "trace" but does not allocate any new space, so the space complexity of both procedures combined remains the same.

Comparing the Myers Difference algorithm to other existing solutions

The Myers algorithm is clearly superior than a naïve brute force approach where you enumerate each possible edit script and find the one that gives you the shortest possible script, since the number of possible scripts grows exponentially. It is also superior than the naïve comparison of two files line-by-line to find the difference between two files since, although such an algorithm would run very fast by only looping through each file once, it would not find the shortest possible edit script, only "an" edit script. This would not be very useful for version control systems or for the end user of such a program, since a lot of space would be wasted in the version control system and the output would be less comprehensible for a software engineer.

However, the Myers algorithm is also superior to other algorithms that can compute the shortest edit script. One such algorithm is the Longest Common Subsequence algorithm (Cormen, 2001), which also finds the insertions and deletions necessary to go from one string to another (although it does so more implicitly – it first calculates what the longest common subsequence between two strings is, and then the elements that are not part of that sequence must have been either inserted or deleted from string A to string B). However, the LCS algorithm has a time complexity $\Theta(N^2)$, if both strings are of the same length, or $\Theta(MN)$. This is because the LCS algorithm uses a dynamic programming approach but does not incorporate the greedy component of Myers' algorithm. It fills out the entire dynamic programming table that contains the longest common subsequence between the prefixes of strings and it backtracks the solution afterwards (so it still has some extra overhead when recreating the solution, just like Myers' algorithm).

Furthermore, Myers' algorithm prioritizes deletions over insertions. In the inner for-loop conditions, it explicitly checks if we can arrive at the current diagonal by moving downwards first, and, if not, by moving horizontally. This means that even if both paths are furthest-reaching, the algorithm will choose to move downwards instead of horizontally. This is of importance for the purpose of the algorithm, since it makes more sense when viewing the difference between two files to see the lines you have deleted *before* the lines you have inserted. This makes the algorithm more naturally suited to being used as a difference tool in version control systems.

It should also be noted that it is possible to specialize Dijkstra's algorithm, a graph-search algorithm where we know there are no negative cycles, to also find the shortest edit script between a string A and a string B in $O(ND)$ time, where N is the sum of the length of string A and string B (Myers, 1986). However, the modifications required are extensive enough to justify the definition of a new algorithm (Myers) which achieves the same purpose and has a simpler derivation and is thus more naturally suited to the problem. One could appeal to the concept of parsimony, since both algorithms achieve the same time complexity.

Additionally, one advantage of the Myers Difference algorithm is that its complexity depends on D, the length of the shortest edit script (D-path). This means that the algorithm runs very fast when the number of modifications between two files are small. In practice, this is often the case: engineers will commit code to the master repository multiple times a day, often in small chunks. In this regard, the fact that the Myers Difference algorithm has an *upper* bound of $O((M+N)D)$ but not a tight bound makes it vastly superior to the other algorithms discussed should they be employed for finding the difference between files, since the LCS algorithm, for example, has a tight bound of $\Theta(MN)$, meaning that the length of the shortest edit script does not make a difference to its complexity.

References

Cormen, T. H., & Cormen, T. H. (2001). *Introduction to algorithms*. Cambridge, Mass: MIT Press.

Git. (2019). Retrieved 19 December 2019, from https://git-scm.com/about

Hunt, J., & Mcllroy, D. (1976). *An Algorithm for Differential File Comparison*. Bell Laboratories.

Mercurial SCM. (2019). Retrieved 19 December 2019, from https://www.mercurial-scm.org/about

Myers, E. (1986). An O(ND) difference algorithm and its variations. Algorithmica, 1(1-4), 251-266. doi: 10.1007/bf01840446

Appendix

The accompanying code for this paper can be found online at https://github.com/DHDaniel/git-diff-clone.