Review of the Shortest Edit Script Problem And Difference Algorithms

Haotian Xue
xueha@kean.edu
Wenzhou-Kean University
Wenzhou, Zhejiang, China

Abstract

Aiming to examine the detailed algorithm implementation behind the popular software utilities, such as Git and Diff, the study investigated the Shortest Edit Script (SES) problem as the underlying challenge, examined the different algorithms as solutions to the problem, and compared their performance under different scenarios. The study also provides insight into the Longest Common Subsequence (LCS) problem for a better understanding of the SES problem. For the performance test, the source files in the Linux kernel were selected to simulate the real-world use cases of the relevant development tools. In conclusion, the study found a significant performance gap between the different algorithms and implementations in different conditions.

Keywords: Algorithm, Difference, Longest Common Subsequence, Shortest Edit Script, Dynamic Programming

1 Introduction

With the rapid development of software engineering and the explosion of data in recent years, the operation of getting differences between different data sources is becoming increasingly significant in today's context as they can be used to extract information on changes from the comparisons. By analyzing the changes, people can easily trace the history of modification, shift in the public interest, compare the similarities, and detect the potential key variables in a multifactor environment, which contributes to its significance in applications like software source code management, genetic engineering, and big data analysis, where the changes are important. [5, 6, 12, 14, 16] The nature of getting data differences makes the difference algorithms the solutions to the Shortest Edit Script (SES) problem [12–14], which aims to find the shortest sequence of edition operations needed to convert one file to another, as the editions needed are simply the differences and the target to find the shortest script helps the differences to focus only on the key changing part of the data.

In the past few decades, extensive research has been done on the SES problem making it efficient and practical for its various use cases. [5, 6, 12, 14] With the application of new optimization techniques, the diff algorithms tend to have lower time complexity and space complexity, and some of them are still used today and integrated into some commonly used tools. For example, proposed by Myers in 1986 [14], the Myers Diff Algorithm and its refinement are still used as the default diff algorithm of Git and the Unix Diff Utility thanks to its outstanding performance in comparing code snippets. In addition, the Basic Diff Algorithm [17] also got extensive use in small programs and scripts for processing small amounts of data due to its simplicity.

Although the diff algorithms have been well-developed today and derived various variations for all kinds of use cases [11], it is still essential to look into the design logic and implementation details of the algorithms to understand the unique features of each for selection and help explain the behavior of some commonly used software utilities to make the output predictable and help maximize productivity with the assistance of the software tools, which is important in the modern software development practices as the version control systems based on the algorithms [16] have become a necessity in software projects on a large scale. Furthermore, the SES problem and diff algorithms are also crucial components of some more complicated programming problems, making the understanding of diff algorithms extensible and reusable in understanding and solving other programming challenges.

Starting with the Longest Common Subsequence (LCS) problem, which is computationally equivalent to the SES problem [2, 12–15] as the sequence excluded by the longest common subsequence is simply part of the document needing to be inserted or deleted defined by the shortest edit script (SES), and its solutions, this study then expand this method to solving the SES problem forming the basic diff algorithm [17] and finally introduces the further optimized Myers Diff Algorithm[14] and its variant with linear space complexity [14] as the most commonly used algorithm solution for this problem. A performance test on different situations was also conducted among the three Diff algorithms (basic, Myers, and linear-space Myers Algorithms) to investigate the performance influence of the different logic and concepts the algorithms are based on and their suitable use cases.

2 The Longest Common Subsequence (LCS) Problem

Before discussing the Shortest Edit Script (SES) problem, it is worthwhile to mention the longest common subsequence (LCS) problem first as a simpler equivalence of it. Thanks to

the similarity of the two problems, the solutions for the LCS problems can then be modified to be a valid solution for the SES problem as is shown in Section 3. It is also necessary to mention the LCS problem not only because it is another classic algorithmic problem that has a direct relationship with the diff algorithm but also due to its inspiration for further optimization of the diff algorithm.

2.1 LCS Problem Definition

As a common algorithmic problem [3], the Longest Common Subsequence (LCS) problem is finding the longest subsequence shared in the two input sequences where the subsequence does not need to be continuous in either input sequence distinguishing it from the Longest Common Substring problem [3]. The LCS of two input sequences follows the following rules:

- For two same input sequences, the LCS should be the same, containing all elements in the input sequence.
- For two different input sequences sharing the same elements, there exists LCS with its length greater or equal to one.
- For two completely different input sequences sharing no elements, LCS does not exist.

Notably, for the second scenario above, the most common one, there may exist more than one valid LCSes, which have the same length but different elements or order. Therefore, it is normal for different algorithms and implementations to have different solutions to the same LCS problem. [2, 7, 8, 12–15]

For convenience, define the length of the first input sequence (origin) as M and the length of the second input sequence (target) as N, which will be used in the rest of the paper for complexity calculation.

2.2 LCS Brute Force Solution

Since the longest common subsequence must first be a common subsequence of both input sequences, which is also one of the possible subsequences of one specific input sequence, consisting of a series of elements following the order of the original sequence, it is straightforward to try out every possible subsequence according to one of the inputs and choose the valid one with longest length as the result following the brute force method to solve the LCS problem. [2, 7, 15] *Algorithm 1* lists the essential steps of the brute force algorithm for the LCS problem.

Although the brute force solution is simple in logic, it has an unacceptably large time complexity of

$$O(2^{min(M,N)} \cdot max(M, N))$$

made up of the $O(2^{min(M,N)})$ need to traverse all possible subsequences and the $O(max(M, N))$ operations to check the validity of each proposed subsequence by traversing the other string.

Algorithm 1: LCS Brute Force Solution

```
Data: X, Y ∈ sequence ∧ len(X) ≤ len(Y)
Result: lcs = the LCS of X and Y
i ← 2^len(X) − 1 ;
lcs ← empty sequence ;
while i > 0 do
    sub ← binaryMask(X, i) ;
    if sub is a subsequence of Y AND
     len(sub) > len(lcs) then
        lcs ← sub ;
    end
end
```

One primary factor that caused the expansion of algorithm time complexity is the traversal of all possible subsequences as candidates of the LCS, though in *Algorithm 1* the shorter sequence in the inputs is used to generate the possible subsequences to reduce the number of iterations. In addition, the traversal operations are also a waste of computational power since the LCS must consist of several shorter common subsequences, which are traversed multiple times in search of the longest while their results are largely discarded in the brute force solution. Based on these drawbacks, the brute force algorithm for the LCS problem is not commonly used in practical use cases. [2, 15]

2.3 LCS Recursive Solution

To reduce the time complexity and make the algorithm applicable for practical use cases, the recursive solution of the Longest Common Subsequence (LCS) problem further reduces the number of iterations needed by utilizing the relationship between the LCS and its subsequences, which are shorter common subsequences between the two inputs. [2, 8, 15]

Because each element in the LCS is a common element between the two input sequences, when finding the first element of both inputs the same element, it can be directly appended to the LCS sequence, which represents the longest common sequence so far reading both inputs. [2] Otherwise, it indicates at least one of the two heading elements must not included in the LCS. In this situation, by discarding the first element of one sequence and continuing the evaluation respectively for each input sequence until one of them becomes empty, the LCS can be obtained by selecting the longer subsequence at each time of splitting the situations. [2]

This relationship reveals the recursive nature of the solution of the LCS problem that, knowing the LCS of the first i elements and j elements of the two input sequences, the complete LCS is simply the concatenation of the knowing LCS and the LCS of the leftover part of the sequences. To get

the LCS of two strings, it can be seen as finding the first element in the LCS and then appending the LCS of the leftover part to it. The method to find the first element is described above, forming a tree-like recursive structure for the finding of the complete LCS. Staring the divide-and-conquer process from the first element of the two input sequences and stopping when one of them has no more element for evaluation ensures the validity of the obtained LCS, with each iteration either determining one element in the LCS or branching the evaluation into two separate situations.

The mathematical representation of the logic is shown in *Equation 1* where $f(i, j)$ represents the LCS from the i^{th} elements of the first input sequence (origin) and the j^{th} element of the second input sequence (target) til the end of each sequence.

$$f(i,j) = \begin{cases} \textit{empty sequence}, & i > M \vee j > N \\ X[i] + f(i+1, j+1), & X[i] = Y[i] \\ longer(f(i+1, j), f(i, j+1)), & X[i] \neq Y[i] \end{cases} \tag{1}$$

Based on this recurrence relation, the recursive solution of the LCS problem is designed and implemented by checking a pair of elements in each iteration and entering the next iteration with either a new element appended to the current LCS or two separate situations to evaluate. Encountering its base case when there are no more element pairs to check, the algorithm will finally stop recurring after traversing one of the input sequences to the end in all situations. The essential steps of the algorithm are listed in *Algorithm 2*.

Algorithm 2: LCS Recursive Solution

```
Data: X, Y ∈ sequence
Result: lcs = the LCS of X and Y
Function LCS(i, j):
    if i > len(X) OR j > len(Y) then
        return empty sequence ;
    else if X[i] = Y[j] then
        return X[i] append LCS(i+1, j+1) ;
    else
        if len(LCS(i+1, j)) > len(LCS(i, j+1)) then
            return LCS(i+1, j) ;
        else
            return LCS(i, j+1) ;
        end
    end
end
```

By reusing the intermediate result of partial LCSes, the application of the recursive method in solving the LCS problem can significantly reduce the computation cost by dividing the big problem of finding LCS in the range of two complete sequences to many small problems of finding the next element of the LCS, which only includes a simple comparison and sequence concatenation. The recursive structure naturally supports the branching operations needed in the algorithm, making its code presentation simple and concise.

The recursive solution works well in the common use cases where there are only a small number of differences in the two input sequences. In such cases, the same element can be directly appended to the LCS, and a few branches are made since they are only needed when a difference is detected. [2] However, when the two input sequences are largely or entirely different, the recursive algorithm can hardly provide any improvement or even results in a worse time complexity of $O(2^{max(M,N)})$ compared to the Brute Force solution. [2, 8] Though reducing the overhead of checking the candidate subsequence in each iteration, the recursive algorithm may produce an even larger number of branches in the worst case, with many of them overlapping each other and repeatedly computing the same thing, causing a waste of computational power and memory space. The maximum number of branches satisfies the complexity of $O(2^{max(M,N)})$ in the worst case according to the $max(M, N)$ layer binary-tree-like structure made by recursion.

2.4 LCS Dynamic Programming Solution

Having the most significant problem of the recursive solution repeated computation, it is natural to apply the dynamic programming techniques to the algorithm as it allows the reuse of the computational result of every single step making no waste of computational power on overlapping branches. [1, 4, 9, 15] The existence of the explicit recurrence relation in finding the LCS makes the dynamic programming solution practical and clear by constructing a reversed search tree in a 2D space. [2, 15]

As is described in *Section 2.3*, the LCS can be obtained by constantly finding the first element of the LCS of the current segments and appending the LCS of the leftover part of it until there is no more elements or branches to examine. A similar logic is followed in the dynamic programming solution by first exploring all possible branches and constantly selecting the situations with longer LCSes by merging them during the traversal of input sequences. When the traversal reaches the end of both input sequences, there will be only one branch left, which, after all the comparisons, presents the final LCS of the problem. [2, 7, 15]

Though similar in the recurrence relation used and representation of data, the dynamic programming solution of the LCS problem has a different algorithm design and implementation compared to the recursive solution to adapt to the dynamic programming method. Defining $LCS(i, j)$ as the **length** of the LCS of the first i elements of the first input sequence (origin) and the first j elements of the second input sequence (target), the calculation of $LCS(i, j)$ follows

Equation 2 according to the recurrence relation mentioned above where $f(i, j)$ is the same as $LCS(i, j)$. [10]

$$f(i,j) = \begin{cases} 0, & i = 0 \vee j = 0 \\ f(i-1, j-1) + 1, & X[i] = Y[j] \\ max(f(i-1, j), f(i, j-1)), & X[i] \neq Y[j] \end{cases} \quad (2)$$

When one of the sequence segments is empty, the LCS of them is also empty ($i = 0 \vee j = 0$). When the next elements of the two sequences are the same ($X[i] = Y[j]$), add the element to the LCS, and the length of the LCS increases. When the next elements of the two sequences are not the same ($X[i] \neq Y[j]$), select the longer length of the nearest two branches as the length of the current LCS, as no element is added to the LCS. [7] According to this relationship, a *2D* integer array of size $(M+1) \times (N+1)$ can be created with each location indicating the intermediate result for a specific pair of i and j. By traversing each location in ascending order covering all pairs of i and j, the arrays can be filled, and the final LCS can be read from $dp[i][j]$ as it is the optimized result after merging all branches and entirely cover the two input sequences. Describing the processing in programming logic, the essential step of the dynamic programming solution of the LCS problem is listed in *Algorithm 3*.

Algorithm 3: LCS Dynamic Programming Generation

```
Data: X, Y ∈ sequence
Result: length = the length of the LCS of X and Y
Function DP(X, Y):
    dp ← int[M + 1][N + 1] ;
    Fill the row i = 0 and column j = 0 with 0 ;
    for i ← 1 to M do
        for j ← 1 to N do
            if X[i − 1] = Y[j − 1] then
                dp[i][j] = dp[i − 1][j − 1] + 1 ;
            else
                dp[i][j] = max(dp[i-1][j], dp[i][j-1]) ;
            end
        end
    end
    return dp[M][N] ;
end
```

By performing a const time operation of each location of the dynamic programming array, the dynamic programming solution of the LCS problem generates the intermediate results and gets the length of the LCS in a $O(MN)$ time and space complexity since the dynamic programming has $(M + 1) \times (N + 1)$ locations in total. [2, 7, 8, 15] Occupying some extra memory space, the dynamic programming techniques greatly improved the performance of the algorithm compared to the brute force and recursive solutions. However, only the length of the LCS can be obtained using dynamic programming, which requires additional steps to regenerate the path of increment, the LCS, in other words. [2]

To generate the path, *Equation 2* will be used again but in the reverse direction, as the target is to find the previous array location according to the intermediate results and input sequences. The mathematical relationship used in the process is shown in *Equation 3* where *maxDp* returns the (i, j) pair that has the larger value in the dynamic programming array obtained in *Algorithm 3*.

$$prev(i,j) = \begin{cases} stop, & i = 0 \vee j = 0 \\ (i-1, j-1), & X[i] = Y[j] \\ maxDp((i-1, j), (i, j-1)), & X[i] \neq Y[j] \end{cases} \quad (3)$$

Starting from the (M, N) location of the dynamic programming array indicating the end of both input sequences and backtracking following the rules in *Equation 3*, the LCS can be obtained in reversed order as the elements checked to satisfy $X[i] = Y[j]$ during the process until reaching the stopping condition. A stack data structure can be used to store the path and can provide the LCS in a positive order when popping the elements out. The essential steps of the backtracking process are listed in *Algorithm 4*.

Algorithm 4: LCS Dynamic Programming Backtracking

```
Data: X, Y ∈ sequence, dp = the DP array
Result: path = the stack containing the LCS
        elements
Function backtrack(X, Y, dp):
    result ← empty stack ;
    i ← M ;
    j ← N ;
    while i > 0 ∧ j > 0 do
        if X[i] = Y[j] then
            result.push(X[i]) ;
            i − − ;
            j − − ;
        else if dp[i − 1][j] > dp[i][j − 1] then
            i − − ;
        else
            j − − ;
        end
    end
    return result ;
end
```

Moving horizontally, vertically, and diagonally in the dynamic programming array, the backtracking process will

have a maximum of $M + N$ iterations, with each doing a constant time operation. Therefore, the time complexity of the backtracking procedure is $O(M + N)$, and the time complexity of the entire process of finding the LCS is $O(MN) = O(MN) + O(M + N)$, which is significantly lower than the brute force and recursive solution introduced previously, proving a superior performance. [7] Combining the two processes, the complete dynamic programming solution of the LCS problem is shown in *Algorithm 5*.

Algorithm 5: LCS Dynamic Programming Solution

```
Data: X, Y ∈ sequence
Result: path = the stack containing the LCS
        elements
Function LCS(X, Y):
    /* Generate DP Array                    */
    dp ← int[M + 1][N + 1] ;
    Fill the row i = 0 and column j = 0 with 0 ;
    for i ← 1 to M do
        for j ← 1 to N do
            if X[i − 1] = Y[j − 1] then
                dp[i][j] = dp[i − 1][j − 1] + 1 ;
            else
                dp[i][j] = max(dp[i-1][j], dp[i][j-1]) ;
            end
        end
    end
    /* Backtracking                         */
    result ← empty stack ;
    i ← M ;
    j ← N ;
    while i > 0 ∧ j > 0 do
        if X[i] = Y[j] then
            result.push(X[i]) ;
            i − − ;
            j − − ;
        else if dp[i − 1][j] > dp[i][j − 1] then
            i − − ;
        else
            j − − ;
        end
    end
    return result ;
end
```

Among the three solutions to the LCS problem introduced in this section, the dynamic programming solution offers a balance between time and space with its $O(MN)$ time and space complexity. It is also commonly considered the only practical LCS algorithm among the three.

3 The Shortest Edit Script (SES) Problem

Due to the time limit, the leftover part of the report is not finished yet.

TO BE CONTINUED.

3.1 SES Problem Definition

TO BE CONTINUED.

3.2 The Myers Algorithm

TO BE CONTINUED.

3.3 The Linear Space Complexity Variant of The Myers Algorithm

TO BE CONTINUED.

4 Performance Test

TO BE CONTINUED.

5 Conclusion

TO BE CONTINUED.

Acknowledgments

To Dr. Winoto, for teaching me throughout the semester and introducing me to the study and research of algorithms and their applications. To Dr. Chiu, for instructing me on Latex writing and pseudocode composition, which helped me a lot in writing the report. To the creators of online algorithm visualization tools, your work greatly assisted me in learning the concepts and provided me with easy ways to create figures for the report. Thanks to your help, I was able to complete the individual research and write this report.

References

[1] Richard Bellman. 1966. Dynamic Programming. *Science* 153, 3731 (July 1966), 34–37. https://doi.org/10.1126/science.153.3731.34

[2] L. Bergroth, H. Hakonen, and T. Raita. 2000. A Survey of Longest Common Subsequence Algorithms. In *Proceedings Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000.* 39–48. https://doi.org/10.1109/SPIRE.2000.878178

[3] Maxime Crochemore, Costas S. Iliopoulos, Alessio Langiu, and Filippo Mignosi. 2017. The Longest Common Substring Problem. *Mathematical Structures in Computer Science* 27, 2 (Feb. 2017), 277–295. https://doi.org/10.1017/S0960129515000110

[4] Sean R. Eddy. 2004. What Is Dynamic Programming? *Nature Biotechnology* 22, 7 (July 2004), 909–910. https://doi.org/10.1038/nbt0704-909

[5] James Gosling. 1981. A Redisplay Algorithm. *ACM SIGPLAN Notices* 16, 6 (April 1981), 123–129. https://doi.org/10.1145/872730.806463

[6] Dov Harel and Robert Endre Tarjan. 1984. Fast Algorithms for Finding Nearest Common Ancestors. *SIAM J. Comput.* 13, 2 (May 1984), 338–355. https://doi.org/10.1137/0213024

[7] D. S. Hirschberg. 1975. A Linear Space Algorithm for Computing Maximal Common Subsequences. *Commun. ACM* 18, 6 (June 1975), 341–343. https://doi.org/10.1145/360825.360861

[8] Daniel S. Hirschberg. 1977. Algorithms for the Longest Common Subsequence Problem. *J. ACM* 24, 4 (Oct. 1977), 664–675. https://doi.org/10.1145/322033.322044

[9] Ronald A. Howard. 1966. Dynamic Programming. *Management Science* 12, 5 (Jan. 1966), 317–348. https://doi.org/10.1287/mnsc.12.5.317

[10] Tao Jiang and Ming Li. 1995. On the Approximation of Shortest Common Supersequences and Longest Common Subsequences. *SIAM J. Comput.* 24, 5 (Oct. 1995), 1122–1139. https://doi.org/10.1137/S009753979223842X

[11] Sanjeev Khanna, Keshav Kunal, and Benjamin C. Pierce. 2007. A Formal Investigation of Diff3. In *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science*, David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, V. Arvind, and Sanjiva Prasad (Eds.). Vol. 4855. Springer Berlin Heidelberg, Berlin, Heidelberg, 485–496. https://doi.org/10.1007/978-3-540-77050-3_40

[12] P. Prakash Maria Liju. 2022. Algorithm to Derive Shortest Edit Script Using Levenshtein Distance Algorithm. https://doi.org/10.48550/arXiv.2208.08823 arXiv:2208.08823 [cs]

[13] William J. Masek and Michael S. Paterson. 1980. A Faster Algorithm Computing String Edit Distances. *J. Comput. System Sci.* 20, 1 (Feb. 1980), 18–31. https://doi.org/10.1016/0022-0000(80)90002-1

[14] Eugene W. Myers. 1986. AnO(ND) Difference Algorithm and Its Variations. *Algorithmica* 1, 1 (Nov. 1986), 251–266. https://doi.org/10.1007/BF01840446

[15] Mike Paterson and Vlado Dančík. 1994. Longest Common Subsequences. In *Mathematical Foundations of Computer Science 1994 (Lecture Notes in Computer Science)*, Igor Prívara, Branislav Rovan, and Peter Ruzička (Eds.). Springer, Berlin, Heidelberg, 127–142. https://doi.org/10.1007/3-540-58338-6_63

[16] Marc J. Rochkind. 1975. The Source Code Control System. *IEEE Transactions on Software Engineering* SE-1, 4 (Dec. 1975), 364–370. https://doi.org/10.1109/TSE.1975.6312866

[17] Robert A. Wagner and Michael J. Fischer. 1974. The String-to-String Correction Problem. *J. ACM* 21, 1 (Jan. 1974), 168–173. https://doi.org/10.1145/321796.321811

A Artificial Intelligence (AI) Usage

A.1 OpenAI ChatGPT

The author has used OpenAI ChatGPT for literature review and Latex layout & syntax consulting purposes, including summarizing paper contents, extracting research outcomes, seeking help on the *algorithm2e* package, etc.

A.2 GitHub Copilot

The author has used GitHub Copilot for coding writing. The use of AI in code writing significantly reduced the time spent on repeating similar structures and can help analyze the error message for debugging.

B Additional Resources

The LaTeX source files and source code of the algorithms discussed and analyzed in the report can be accessed on GitHub. Link: https://github.com/xht308/Diff_Algo_Demo

Written in Go language, the coding part of the project also provides a command-line utility for text file comparison. The executable binary for Windows x64 is currently provided. If needing the executable for other platforms, please follow the instructions in the GitHub repository to build the binary file.