CPE593 Final Report: Patch Generation with Myers Diff

Jack Mavorah, Logan Smith

May 19, 2021

Source Code: https://github.com/JMavorah/BinaryDiff/

Abstract

Differencing and patching are critical components of software development. Developers often need to update software that may be in use by millions of users. If the update is a small bug fix, security update, or even a very large brand new version, deleting the old version and downloading the new one is a horribly inefficient solution. Rather, a patch file can be constructed which contains instructions to transform the old version into the new one. That patch can then be sent, and every user's copy of the software can update itself to the new version in a much faster and more efficient manner. This implementation uses the highly effective and very popular Myers Differencing algorithm to find shortest edit scripts between files and generate an appropriate patch.

Introduction

A literature review was conducted to determine the ideal differencing/patching algorithm. Important criteria were time and technical constraints, effectiveness of the algorithm, modern usage, amount of available reference material, and more. While many algorithms exist, including BSDiff, XDelta, and more, Myers is very efficient and sees ubiquitous use. It is used at Google and is the default differencing algorithm in Git, which is a system that relies heavily on efficient differencing. It is also not too complicated to implement and has a large amount of available reference material.

Methodology

Myers Diff - Edit Graph Introduction

The algorithm selected was the Myers Differencing algorithm. Myers Diff was published in 1986 by Eugene W. Myers. Myers Diff uses an "edit graph" to represent each file (A and B), and find the shortest edit script (SES) required to change the old file into the new one. An example edit graph, for changing the string A = "ABCABBA" into B = "CBABAC" is shown below. The values N and M represent the sizes of A and B respectively; here N = 7 and M = 6.

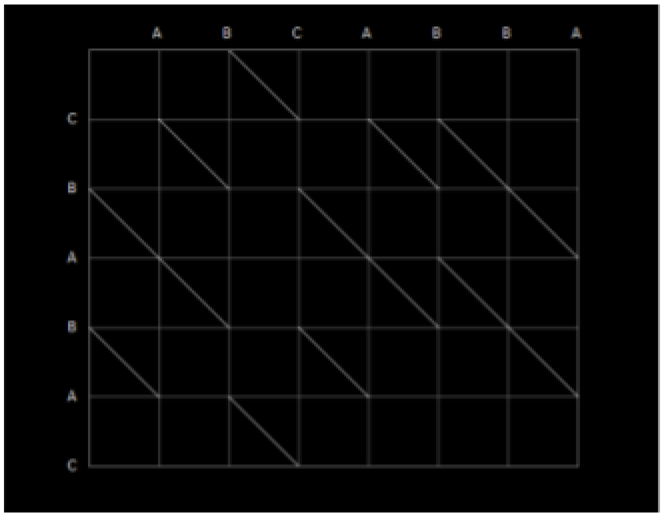


Figure 1: Basic Edit Graph [3]

The old file, A = "ABCABBA" in this case, is represented by the x-axis, while the new file, B = "CBABAC" here, is represented by the y-axis. The point (0,0) represents the start of the differencing process, and the point (N,M) [(7,6) in this example], represents the end of the differencing process, when file A has been transformed into file B. Vertical (downward) movements on the graph represent the deletion of data from file A and horizontal movements (to the right) represent the insertion of data from file B. Diagonals in the graph (seen in the figure above), represent matches in the files; no additions or deletions are necessary. The greatest number of changes needed is equal to N+M, in the case where there are no matches at all between A and B, and all data in A must be deleted and all data in B inserted. This is obviously not a common occurrence. The SES is the path from (0,0) to (N,M) containing the most diagonal movements, e.g., the fewest changes, resulting in the smallest patch. It is important to note that there can be many shortest edit scripts of the same distance, and Myers Diff will use the first one that it finds.

Before describing the algorithm itself, it is important to define a few more components of edit graphs. The first is k-lines, shown here as the brown (odd) and yellow (even) diagonal lines. K-lines begin on the axes and are parallel to the diagonals. The k-line through (0,0) has k = 0, and k increases to the right and decreases downward, so the k-line through (1,0) has k = 1 and the k-line through (0,1) has k = -1. Another important feature of k-lines is that the y value of a point on a known k-line can be calculated with the equation y = x - k, which means that the algorithm will not have to store y values if it knows

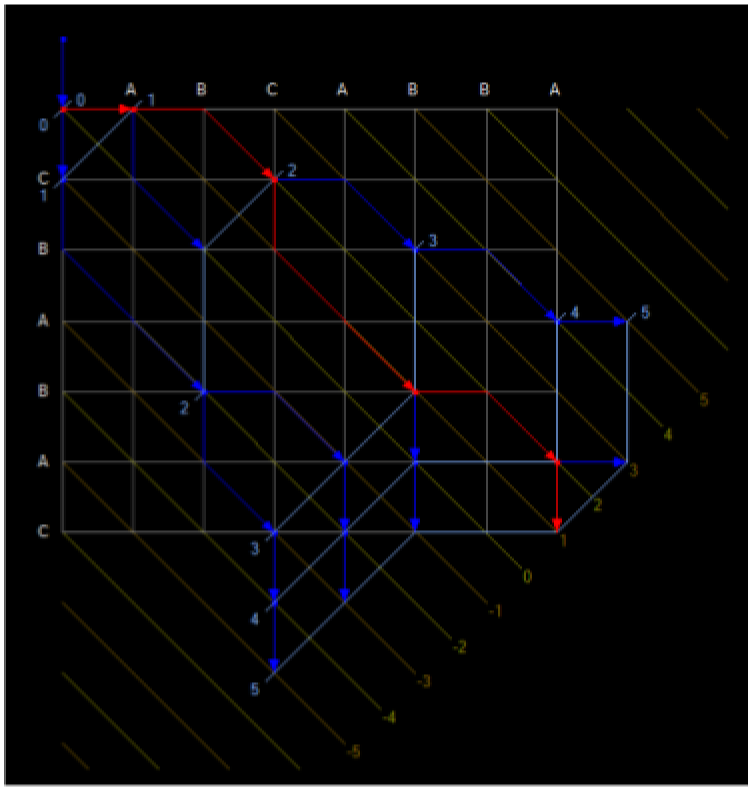


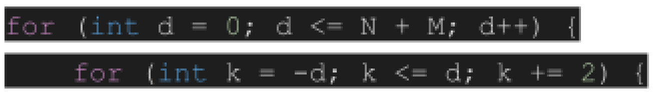
Figure 2: Edit Graph Components [3]

x and k. The next component is snakes, shown in dark blue on the graph above. Snakes are a single change (horizontal or vertical lines) followed by zero or more diagonal lines. Snakes have a start point at the start of the change, a midpoint at the end of the change and start of the diagonals, and an endpoint at the end of the diagonals (the midpoint and endpoint will be the same point if there are no diagonals following the horizontal/vertical line). Traces are multiple connected snakes, and are described by the value d, which defines the number of differences (horizontal or vertical lines) in the trace. Solution traces are traces with minimum d that lead from (0,0) to (N,M). There can be multiple solution traces for given A and B.

Myers Diff - Basic Greedy Algorithm

Loop Setup

The basic greedy version of Myers Diff finds a solution trace by finding the farthest reaching trace for increasing values of d. Beginning with d = 0 (there will only be a solution trace here if A and B are identical), the algorithm checks the farthest paths on all k-lines for each d value. The interval for k is [-d , d], k will equal -d if all movements are downward and k will equal d if all movements are to the right. For a combination of both movements, k will be found in the interval (-d, d). An important optimization is that for even d values, only even k-lines must be checked, and for odd d values, only odd k-lines must be checked. This is analogous to how a bishop on a chess board cannot travel to a square of the other color. Values of k are incremented by 2 from -d to d, and d is incremented until a trace reaches (N,M). When a solution trace is found, it is guaranteed to be an SES since all lower values of d have been checked.



```
for (int d = 0; d <= N + M; d++) {
    for (int k = -d; k <= d; k += 2) {
```

Determining Movement Direction

The next step is to determine a method for finding the farthest reaching path for each k-line. There are two ways to travel from one k-line to another: moving down from a higher k-line (shown left in magenta), or moving right from a lower k-line (shown left in green). At any point in the edit graph, there are three cases for k relative to d: k = -d, k = d, and k = (-d, d). When k = d, the farthest point obtained by moving right, when k = -d, the farthest point is obtained with a downward move, and when k is in (-d, d), the farthest point is obtained by moving from the k-line with the greatest x value. This can be determined by the following boolean expression:

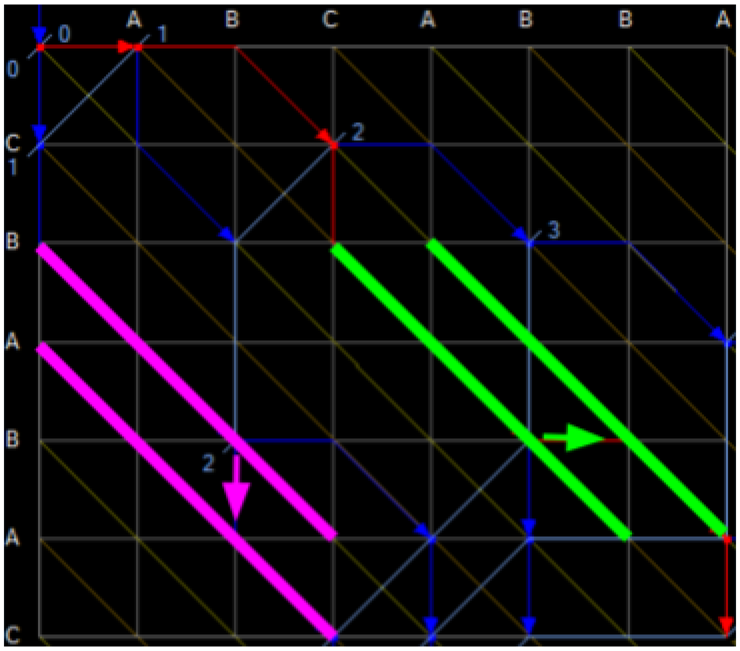
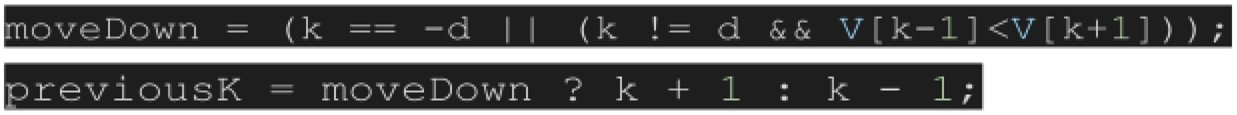


Figure 3: K-line Traversal



```
moveDown = (k == -d || (k != d && V[k-1]<V[k+1]));
previousK = moveDown ? k + 1 : k - 1;
```

After the direction of movement is determined, the start, mid, and end points of the snake are initialized, and any diagonals have to followed to form the full snake, and the endpoints of the snakes are stored in V. When (x,y) reaches (N,M), a solution has been found. Below is Myers' original pseudo code, and our implementation.

```
Constant MAX ∈ [0,M+N]

Var V: Array [−MAX .. MAX] of Integer

1.  V[1] ← 0
2.  For D ← 0 to MAX Do
3.      For k ← −D to D in steps of 2 Do
4.          If k = −D or k ≠ D and V[k−1] < V[k+1] Then
5.              x ← V[k+1]
6.          Else
7.              x ← V[k−1]+1
8.          y ← x−k
9.          While x < N and y < M and a_{x+1} = b_{y+1} Do (x,y) ← (x+1,y+1)
10.         V[k] ← x
11.         If x ≥ N and y ≥ M Then
12.             Length of an SES is D
13.             Stop
```

```
    for (int d = 0; d <= N + M; d++) {
        for (int k = -d; k <= d; k += 2) {
            moveDown = (k == -d || (k != d && V[k - 1] < V[k + 1]));
            previousK = moveDown ? k + 1 : k - 1;
            xStartPoint = V[previousK];
            yStartPoint = xStartPoint - previousK;
            xMidPoint = moveDown ? xStartPoint : xStartPoint + 1;
            yMidPoint = xMidPoint - k;
            xEndPoint = xMidPoint;
            yEndPoint = yMidPoint;
            while (xEndPoint < N && yEndPoint < M && A[xEndPoint] == B[yEndPoint]) {
                xEndPoint++;
                yEndPoint++;
            }
            V[k] = xEndPoint;
            if (xEndPoint >= N && yEndPoint >= M) {
                cout << "solution found" << endl;
                return d;
            }
        }
    }
```

Because V only stores the endpoints for each iteration of the inner loop, V must be copied and stored after each iteration to obtain the full list of points in the solution trace. Another loop then starts at the end of the edit graph (N,M) and travels back to (0,0), determining the direction of each movement using the current and previous x and y values (x changes in a horizontal move and y changes in a vertical move). Those movement directions are then used to construct a patch:

```
   for (int d = D; d > 0;  d--){ //increment from -D to D to retrace solution trace
        //y = x - k ==> k = x - y
        k = x - y;
        // cout << "Before prev init: k, x, y, = " << k << ", " << x << ", " << y << "\n";
        // cout << "After init: prevK, prevX, prevY, = " << prevK << ", " << prevX << ", " <<
prevY << "\n\n";

        //init prev k, x, y
        prevK = (k == -d || (k != d && V[k-1] < V[k+1])) ? k + 1 : k - 1;
        prevX = V[prevK];
        prevY = prevX - prevK;
        // cout << "After prev init: k, x, y, = " << k << ", " << x << ", " << y << "\n";
        // cout << "After init: prevK, prevX, prevY, = " << prevK << ", " << prevX << ", " <<
prevY << "\n\n";

        //follow any diagonals:
        //if x and y are both greater than previous ==> diagonal move
        while (x > prevX && y > prevY){
            x--;
            y--;
        }
        // cout << "After while: k, x, y, = " << k << ", " << x << ", " << y << "\n";
        // cout << "After init: prevK, prevX, prevY, = " << prevK << ", " << prevX << ", " <<
prevY << "\n\n";

        //construct patch:
        if(x == prevX) { //same x ==> change in y ==> down move - insertion
        cout << "down move - insertion\n";
        patch = to_string(prevX) + "I" + B[prevX] + " " + patch;
        }
```

Patches

Patches can be formatted in many different ways, a simple form being an index (of A), and an operation (deletion, or insertion (with character to insert)):

Patch from A ⇒ B

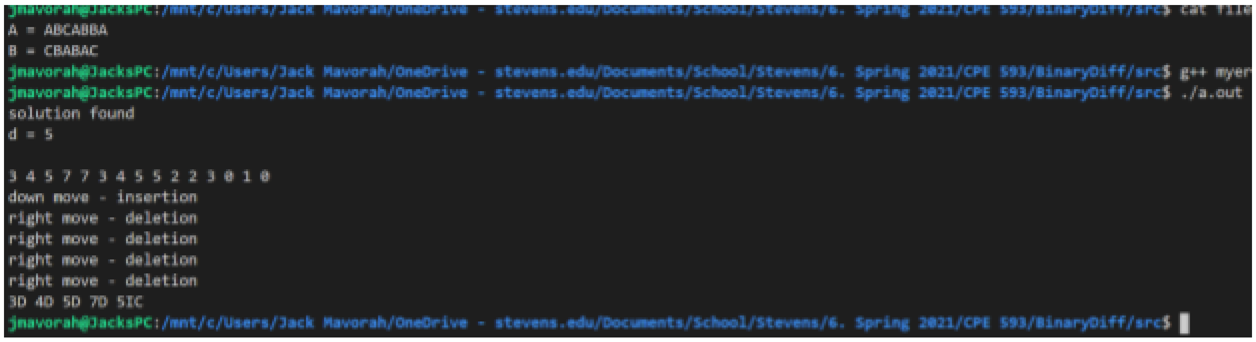
- A = ABCABBA
- B = CBABAC

Patch: 0D 1D 3IB 5D 7IB

⇒ ~~AB~~CBAB~~B~~AC ⇒ CBABAC = B

Results and Future Work

The differencing portion Myers Diff was successfully implemented, finding the length of the SES capable of transforming one file into another (demonstration below).



```
jmavorah@JacksPC:/mnt/c/Users/Jack Mavorah/OneDrive - stevens.edu/Documents/School/Stevens/6. Spring 2021/CPE 593/BinaryDiff/src$ cat file
A = ABCABBA
B = CBABAC
jmavorah@JacksPC:/mnt/c/Users/Jack Mavorah/OneDrive - stevens.edu/Documents/School/Stevens/6. Spring 2021/CPE 593/BinaryDiff/src$ g++ myer
jmavorah@JacksPC:/mnt/c/Users/Jack Mavorah/OneDrive - stevens.edu/Documents/School/Stevens/6. Spring 2021/CPE 593/BinaryDiff/src$ ./a.out
solution found
d = 5

3 4 5 7 7 3 4 5 5 2 2 3 0 1 0
down move - insertion
right move - deletion
right move - deletion
right move - deletion
right move - deletion
3D 4D 5D 7D 5IC
jmavorah@JacksPC:/mnt/c/Users/Jack Mavorah/OneDrive - stevens.edu/Documents/School/Stevens/6. Spring 2021/CPE 593/BinaryDiff/src$ 
```

Figure 4: Myers Diff Demonstration

The algorithm successfully found that the minimum number of changes to turn ABCABBA into CBABAC is 5. The patch generation is still buggy, due to how the endpoints of snakes are stored in the data structure V. Currently, the endpoints are stored in a vector, which is then pushed back into another vector after each iteration. However, a different data structure is needed because V must be indexed not from 0 to 2d, but from -d to d. Future work involves building data structures that can accomplish this task, as well as the storage of snapshots after each iteration. The current algorithm is $O((N+M)*D)$ in time and $O(N+M)$ in space, with N and M being the sizes of files A and B and D being the number of changes required for patching. Additional future work includes implementing Myers' recursive linear space optimization which reduces space complexity to $O(\min(N,M))$ and time complexity to $O(\min(N,M)*D)$, and implementing a patcher capable of executing the patch, actually transforming file A into file B.

References

1. 262588213843476. “Basic Myers Diff Js Implementation.” *Gist*, gist.github.com/Quasar-Kim/cafb5415ed111e47716c403de2490007.

2. Burns, R.C., and D.D.E. Long. “A Linear Time, Constant Space Differencing Algorithm.” *1997 IEEE International Performance, Computing and Communications Conference*, doi:10.1109/pccc.1997.581547.

3. Butler, Nicholas. “Investigating Myers' Diff Algorithm: Part 2 of 2.” *CodeProject*, CodeProject, 19 Sept. 2009, www.codeproject.com/Articles/42279/Investigating-Myers-Diff-Algorithm-Part-1-of-2.

4. Loce, Robert P. “Optimal Binary Differencing Filters: Design, Logic Complexity, Precision

5. Analysis, and Application to Digital Document Processing.” *Journal of Electronic Imaging*, vol. 5, no. 1, 1996, p. 66., doi:10.1117/12.228418.

6. Myers, Eugene W. “AnO(ND) Difference Algorithm and Its Variations.” *Algorithmica*, vol. 1, no. 1-4, 1986, pp. 251–266., doi:10.1007/bf01840446.

7. psionic12. “psionic12/Myers-Diff-in-c-.” *GitHub*, github.com/psionic12/Myers-Diff-in-c-.

8. Ristad, E.S., and P.N. Yianilos. “Learning String-Edit Distance.” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 5, 1998, pp. 522–532., doi:10.1109/34.682181.

9. *The If Works*, 12 Feb. 2017, blog.jcoglan.com/2017/02/12/the-myers-diff-algorithm-part-1/.