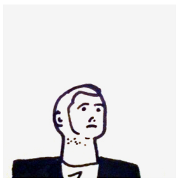

The Myers diff algorithm: part 2

If you enjoy this article, I have published a book explaining the internals of Git through implementation: Building Git.

—

In part 1 of this series, we saw how the diff between two strings is modelled as a graph search problem. We worked through the shortest edit script between two strings:

- *a* = ABCABBA
- *b* = CBABAC

We saw that the edit graph for these strings looks like this:

```
       A     B     C     A     B     B     A

    o-----o-----o-----o-----o-----o-----o-----o   0
    |     |     | \   |     |     |     |     |
C   |     |     |  \  |     |     |     |     |
    |     |     |   \ |     |     |     |     |
    o-----o-----o-----o-----o-----o-----o-----o   1
    |     | \   |     |     | \   | \   |     |
B   |     |  \  |     |     |  \  |  \  |     |
    |     |   \ |     |     |   \ |   \ |     |
    o-----o-----o-----o-----o-----o-----o-----o   2
    | \   |     |     | \   |     |     | \   |
A   |  \  |     |     |  \  |     |     |  \  |
    |   \ |     |     |   \ |     |     |   \ |
    o-----o-----o-----o-----o-----o-----o-----o   3
    |     | \   |     |     | \   | \   |     |
B   |     |  \  |     |     |  \  |  \  |     |
    |     |   \ |     |     |   \ |   \ |     |
    o-----o-----o-----o-----o-----o-----o-----o   4
    | \   |     |     | \   |     |     | \   |
A   |  \  |     |     |  \  |     |     |  \  |
    |   \ |     |     |   \ |     |     |   \ |
    o-----o-----o-----o-----o-----o-----o-----o   5
    |     |     | \   |     |     |     |     |
C   |     |     |  \  |     |     |     |     |
    |     |     |   \ |     |     |     |     |
    o-----o-----o-----o-----o-----o-----o-----o   6


    0     1     2     3     4     5     6     7
```

And, we recorded a trace through the graph to find the shortest path from (0,0) to the bottom-right corner (7,6).

```
0,0 --- 1,0 --- 3,1 --- 5,2 --- 7,3
 |       |       |
 |       |       |
0,1     2,2     5,4 --- 7,5
```

```
 |               |       |
 |               |       |
2,4 --- 4,5     5,5     7,6
 |       |       |
 |       |       |
3,6     4,6     5,6
```

Now, having seen how the graph search works, we’re going to change the representation slightly to get us toward how the Myers algorithm actually works. Imagine that we take the above graph walk and render it rotated by 45 degrees.

```
    |      0     1     2     3     4     5
----+--------------------------------------
    |
 4  |                             7,3
    |                           /
 3  |                       5,2
    |                     /
 2  |                 3,1         7,5
    |               /     \     /     \
 1  |           1,0         5,4         7,6
    |         /     \           \
 0  |     0,0         2,2         5,5
    |         \                       \
-1  |           0,1         4,5         5,6
    |               \     /     \
-2  |                 2,4         4,6
    |                     \
-3  |                       3,6
```

The number along the horizontal axis, *d*, is the depth we’ve reached in the graph, i.e. how many moves we’ve made so far, remembering that diagonal moves are free. The number along the vertical axis we call *k*, and notice that for every move in the graph, $k = x - y$ for each move on that row.

Moving rightward increases *x*, and so increases *k* by 1. Moving downward increases *y* and so decreases *k* by 1. Moving diagonally increases both *x* and *y*, and so it keeps *k* the same. So, each time we make a rightward or downward move followed by a chain of diagonals, we either increment or decrement *k* by 1. What we are recording is the furthest through the edit graph we can reach for each value of *k*, at each step.

Now, here’s how the algorithm proceeds. For each *d* beginning with 0, we fill in each move for *k* from $-d$ to $+d$ in steps of 2. Our aim at each (d, k) position is to determine the best move we can make from the previous position. The best move is the one that gives us the highest *x* value; maximising *x* rather than *y* means we prioritise deletion over insertion.

To discover the best move, we need to decide whether we should pick a downward move from $(d - 1, k + 1)$, or a rightward move from $(d - 1, k - 1)$. If *k* is $-d$ then the move must be downward, likewise if *k* is $+d$ then we must move rightward. For all other values of *k*, we pick the position with the highest *x* from the two adjacent *k* values in the previous column, and determine where that move leads us.

For example, consider the move at (d, k) = (2,0). We can either move rightward from (d, k) = (1,−1) where (x, y) = (0,1), or downward from (d, k) = (1,1) where (x, y) = (1,0).

```
    |      0     1     2
----+----------------------
    |
 1  |            1,0
    |          /     \
 0  |      0,0         ( 2,2 )
    |          \
-1  |            0,1
```

(1,0) has a higher x value than (0,1), so we pick a move downward from (1,0) to (1,1), which leads us to (2,2) diagonally. Therefore we record (x, y) = (2,2) for (d, k) = (2,0). This explains why we recorded the move via this path when (2,0) is also reachable by going rightward from (0,1); picking the previous position with the highest x value means we try to maximise the number of deletions we make before trying insertions.

In some situations, the two previous positions will have the same x value. For example, consider the move at (d, k) = (3,−1), where we can move downward from (x, y) = (2,2) or rightward from (x, y) = (2,4). Moving rightward will increase x, so we move from (2,4) to (3,4) and then diagonally to (4,5).

```
    |      0     1     2     3
----+----------------------------
    |
 2  |                  3,1
    |                /
 1  |            1,0
    |          /     \
 0  |      0,0         2,2
    |          \
-1  |            0,1         ( 4,5 )
    |                \     /
-2  |                  2,4
```

There are a few final simplifications that get us to the algorithm as presented in the paper. The first is that, since we're storing each (x, y) position indexed against k, and $k = x − y$, we don't need to store y since it can be calculated from the values of k and x. The second is that we don't need to store the direction of the move taken at each step, we just store the best x value we can achieve at each point. The path will be derived *after* we've completed this process to find the smallest d that gets us to the bottom-right position; once we know where the final position shows up we can backtrack to find which single path out of the many we've explored will lead us there.

Removing those details leaves us with this information:

```
    |      0     1     2     3     4     5
----+--------------------------------------
    |
 4  |                              7
    |
 3  |                        5
    |
 2  |                  3           7
    |
 1  |            1           5           7
    |
 0  |      0           2           5
    |
-1  |            0           4           5
    |
-2  |                  2           4
    |
-3  |                        3
```

The final simplification is that the *x* values in the *d*th round depend only on those in the (*d* − 1)th round, and because each round alternately modifies either the odd or the even *k* positions, each round does not modify the values it depends on from the previous round. Therefore the *x* values can be stored in a single flat array, indexed by *k*. In our example, this array would evolve as follows with each value of *d*:

```
      k |   -3    -2    -1     0     1     2     3     4
--------+-----------------------------------------------
        |
  d = 0 |                      0
        |
  d = 1 |                0     0     1
        |
  d = 2 |          2     0     2     1     3
        |
  d = 3 |    3     2     4     2     5     3     5
        |
  d = 4 |    3     4     4     5     5     7     5     7
        |
  d = 5 |    3     4     5     5     7     7     5     7
```

The iteration stops when we discover we can reach (*x*, *y*) = (7,6) at (*d*, *k*) = (5,1).

We've now arrived at the representation of the problem used in the algorithm, and we can translate this into working code. We create a function that takes two lists, a and b, which will contain `Diff::Line` objects that represent lines in a file:

```ruby
module Diff
  Line = Struct.new(:number, :text)

  def self.lines(document)
    document = document.lines if document.is_a?(String)
    document.map.with_index { |text, i| Line.new(i + 1, text) }
  end
end
```

Our diff code will mostly only rely on the `text` field of these objects, but it's useful to store the original line numbers as it will make printing and other operations easier later. Since all the algorithms we'll look at will need to break a string into lines, let's make a utility function for doing that and then running the resulting lists through a diff algorithm of our choice.

```
module Diff
  def self.diff(a, b, differ: Myers)
    differ.diff(lines(a), lines(b))
  end
end
```

Now we'll begin writing the `Myers` class that will implement the algorithm we've been discussing. To start with, we'll just make some boilerplate for storing the two strings as instance variables on an object that implements the `diff` method, which we'll define later once all the building blocks are in place.

```
class Myers
  def self.diff(a, b)
    new(a, b).diff
  end

  def initialize(a, b)
    @a, @b = a, b
  end

  def diff
    # TODO
  end
end
```

To return a diff, we need to find the shortest edit path. We begin by storing `n` as the size of `@a` and `m` as the size of `@b`, and `max` as the sum of those; that's the most number of moves we might need to make.

```
  def shortest_edit
    n, m = @a.size, @b.size
    max  = n + m
```

Then we set up an array to store the latest value of x for each k. k can take values from `-max` to `max`, and in Ruby a negative array index is interpreted as reading from the end of the array. The actual order of the elements doesn't matter, we just need the array to be big enough so that there's space for the positive and negative k values.

We set `v[1] = 0` so that the iteration for $d = 0$ picks $x = 0$. We need to treat the $d = 0$ iteration just the same as the later iterations since we might be allowed to move diagonally immediately. Setting `v[1] = 0` makes the algorithm behave as though it begins with a virtual move downwards from $(x, y) = (0,−1)$.

```
    v     = Array.new(2 * max + 1)
    v[1] = 0
```

Next, we begin a nested loop: we iterate d from `0` to `max` in the outer loop, and k from `-d` to `d` in steps of 2 in the inner loop.

```
      (0 .. max).step do |d|
        (-d .. d).step(2) do |k|
```

Within the loop, we begin by choosing whether to move downward or rightward from the previous round. If k is -d, or if it's not d and the k + 1 value is greater than the k - 1 value, then we move downward, i.e. we take the x value as being equal to the k + 1 value in the previous round. Otherwise we move rightward and take x as one greater than the previous k - 1 value. We calculate y from this chosen x value and the current k.

```
          if k == -d or (k != d and v[k - 1] < v[k + 1])
            x = v[k + 1]
          else
            x = v[k - 1] + 1
          end

          y = x - k
```

Having taken a single step rightward or downward, we see if we can take any diagonal steps. As long as we've not deleted the entire @a string or added the entire @b string, and the elements of each string at the current position are the same, we can increment both x and y. Once we finish moving, we store off the value of x we reached for the current k.

```
          while x < n and y < m and @a[x].text == @b[y].text
            x, y = x + 1, y + 1
          end

          v[k] = x
```

Finally, we return the current value of d if we've reached the bottom-right position, telling the caller the minimum number of edits required to convert from a to b.

```
          return d if x >= n and y >= m
        end
      end
    end
```

This minimal version of the algorithm calculates the smallest number of edits we need to make, but not the what those edits are. To do that, we need to back-track through the history of *x* values generated by the algorithm. We'll see how to do this in the next and final article in this series.

Posted on February 15, 2017. This entry was posted in Ruby, Git. Bookmark the permalink.

← The Myers diff algorithm: part 1

→ The Myers diff algorithm: part 3