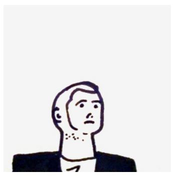

The Myers diff algorithm: part 3

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

—

This is the final installment in my series on the Myers diff algorithm, the default diff algorithm used by Git. In part 1 we established a model for the algorithm as finding the shortest path through a graph representing all the possible edit sequences to convert one string into another. In part 2 we changed the representation of this graph walk into a different co-ordinate space that allows a few optimisations, and from there we got to a working implementation to find the smallest number of edits required:

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

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

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

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

      v[k] = x

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

The final step to complete the algorithm requires us to backtrack. The graph search lets us explore all possible edit paths until we find the smallest path necessary to reach the end. Once we've reached the end, we can look at the data we've recorded in reverse to figure out which single path led to the result.

Let's look at our example again. Recall that the `shortest_edit` function records the best value of *x* we can reach at each (*d*, *k*) position:

```
    |      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
```

We **kn**ow **t**he **fin**al **posi**tion **is** **a**t $(x, y) = (7,6)$, **s**o $k = 1$, **a**nd **we** **fou**nd **it** **aft**er $d = 5$ **ste**ps. **Fr**om $(d, k) = (5,1)$, **we** **c**an **tra**ck **back**ward **t**o **eit**her (4,0) **o**r (4,2):

```
    |     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
```

We **s**ee **th**at (4,2) **cont**ains **t**he **hig**her x **val**ue, **a**nd **s**o **we** **mu**st **ha**ve **reac**hed (5,1) **v**ia a **down**ward **mo**ve **fr**om **the**re. **Th**is **tel**ls **u**s **th**at **t**he (x, y) **posi**tion **bef**ore (7,6) **w**as (7,5).

```
    |     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
```

Via a **simi**lar **argu**ment, **t**o **mo**ve **ba**ck **fr**om (*d*, *k*) = (4,2) **we** **mu**st **ha**ve **co**me **fr**om (3,1) **o**r (3,3). (3,3) **do**es **n**ot **ha**ve a **grea**ter *x* **th**an (3,1), **s**o **we** **kn**ow **th**is **w**as a **right**ward **mo**ve **fr**om (3,1), **or** **fr**om (*x*, *y*) = (5,4) **t**o (7,5).

```
    |      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
```

Before **th**at, **we** **ha**ve a **cho**ice **betw**een (*d*, *k*) = (2,0) **a**nd (2,2), **a**nd (2,2) **h**as **t**he **hig**her *x* **val**ue.

```
    |      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
```

After **th**is, **we'r**e **o**n **t**he **ed**ge **o**f **t**he **gr**id **a**nd **we** **c**an **on**ly **ha**ve **ma**de **right**ward **mov**es.

```
    |      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
```

We’ve walked all the way back to the start, and now we know that the (*x*, *y*) positions of each move are:

```
(0,0) -> (1,0) -> (3,1) -> (5,4) -> (7,5) -> (7,6)
```

These positions are enough to figure out the diagonal moves between each position: we decrement both *x* and *y* until one of them is equal to the values in the previous position, and then we end up a single downward or rightward move away from that position. For example, to get from (5,4) back to (3,1), we observe that 5 > 3 and 4 > 1, and so we can step diagonally back one step to (4,3). This still has neither co-ordinate equal to (3,1) so we take another step back to (3,2). Now, we’ve reached the same *x* value as (3,1), so we must make an upward step from (3,2) to (3,1) to complete the move.

To perform this backtracking, we need to make a small adjustment to `shortest_edit`. Rather than just recording the latest *x* value for each *k*, we need to keep a trace of the array before each turn of the outer loop.

We add the variable `trace` to store these snapshots of `v`, and push a copy of `v` into it on each loop. Rather than just returning the number of moves required, we now return this trace to the caller.

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

    (0 .. max).step do |d|
      trace << v.clone

      (-d .. d).step(2) do |k|
        # calculate the next move...

        return trace if x >= n and y >= m
```

At the end of the function, `trace` will contain enough information to let us reconstruct the path that leads to the final position. It essentially contains the best *x* value that we found at each (*d*, *k*), as we saw earlier. Each copy of `v` that we stored corresponds to a column from our search history.

Now we can implement the backtracking in code. We can write a function that takes a trace produced by the `shortest_edit` function, and the target final (*x*, *y*) position, and works out the backtrace. For each move, it yields the *x* and *y* values before and after the move.

The function iterates over the trace in reverse order. `each_with_index` generates a list that pairs each copy of `v` in the trace up with its corresponding `d` value, and then `reverse_each` iterates over that list backwards.

```
  def backtrack
    x, y = @a.size, @b.size
    shortest_edit.each_with_index.reverse_each do |v, d|
```

At each step of the trace, we calculate the k value, and then determine what the previous k would have been, using the same logic as in the `shortest_edit` function:

```
      k = x - y

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

From that previous k value, we can retrieve the previous value of x from the trace, and use these k and x values to calculate the previous y.

```
      prev_x = v[prev_k]
      prev_y = prev_x - prev_k
```

Then we begin yielding moves back to the caller. If the current x and y values are both greater than the previous ones, we know we can make a diagonal move, so we yield a move from `x-1, y-1` to `x, y` and then decrement x and y as long as this condition holds.

```
      while x > prev_x and y > prev_y
        yield x - 1, y - 1, x, y
        x, y = x - 1, y - 1
      end
```

Finally, we yield a move from the previous x and y from the trace, to the position we reached after following diagonals. This should be a single downward or rightward step. If d is zero, there is no previous position to move back to, so we skip this step in that case. After yielding this move, we set x and y to their values in the previous round, and continue the loop.

```
      yield prev_x, prev_y, x, y if d > 0

      x, y = prev_x, prev_y
    end
  end
```

Running this on our example inputs, this function yields the following pairs of positions, which you should verify correspond to the full path through the edit graph, in reverse:

```
(7, 5) -> (7, 6)
(6, 4) -> (7, 5)
(5, 4) -> (6, 4)
(4, 3) -> (5, 4)
(3, 2) -> (4, 3)
(3, 1) -> (3, 2)
(2, 0) -> (3, 1)
(1, 0) -> (2, 0)
(0, 0) -> (1, 0)
```

To glue everything together, we need a function that takes two texts to diff, passes them as lists of lines into `shortest_edit` to get a trace, and then runs that trace through `backtrack` to generate a diff: a sequence of lines marked as either deletions, insertions, or equal lines.

As `backtrack` yields each pair of positions in reverse, this function adds a diff line to the front of a list. If the *x* values are the same between positions, we know it's a downward move, or an insertion; if the *y* values are the same then it's a deletion, otherwise the lines are equal. To build the diff, we pop a line off the end of each file as appropriate, popping a line off both files if the lines are equal.

```
    def diff
      diff = []

      backtrack do |prev_x, prev_y, x, y|
        a_line, b_line = @a[prev_x], @b[prev_y]

        if x == prev_x
          diff.unshift(Diff::Edit.new(:ins, nil, b_line))
        elsif y == prev_y
          diff.unshift(Diff::Edit.new(:del, a_line, nil))
        else
          diff.unshift(Diff::Edit.new(:eql, a_line, b_line))
        end
      end

      diff
    end
```

The `Diff::Edit` class used in this function is a simple structure that records the type of diff edit (deletion, insertion, or equal), and the line from each file as appropriate. We'll also add a few convenience methods that will make it easier to print these objects out.

```
  module Diff
    Edit = Struct.new(:type, :old_line, :new_line) do
      def old_number
        old_line ? old_line.number.to_s : ""
      end

      def new_number
        new_line ? new_line.number.to_s : ""
      end

      def text
        (old_line || new_line).text
      end
    end
  end
```

So `diff` produces a list of `Diff::Edit` objects, which we can then use for various purposes. The simplest application of this is to print it to the terminal, colouring the lines red and green as appropriate if the stream it's writing to is a TTY (e.g. the program's standard output if being written to a terminal).

```
  module Diff
    class Printer

      TAGS = {eql: " ", del: "-", ins: "+"}
      COLORS = {
        del:     "\e[31m",
        ins:     "\e[32m",
        default: "\e[39m"
      }

      LINE_WIDTH = 4

      def initialize(output: $stdout)
        @output = output
        @colors = output.isatty ? COLORS : {}
      end

      def print(diff)
        diff.each { |edit| print_edit(edit) }
      end

      def print_edit(edit)
        col   = @colors.fetch(edit.type, "")
        reset = @colors.fetch(:default, "")
        tag   = TAGS[edit.type]

        old_line = edit.old_number.rjust(LINE_WIDTH, " ")
        new_line = edit.new_number.rjust(LINE_WIDTH, " ")
        text     = edit.text.rstrip

        @output.puts "#{col}#{tag} #{old_line} #{new_line}    #{text}#{reset
      end

    end
  end
```

Here’s the result of running `Diff::Printer.print Diff.diff(a, b)` on our two example strings, just as we expect:

```
-    1         A
-    2         B
     3    1    C
+         2    B
     4    3    A
     5    4    B
-    6         B
     7    5    A
+         6    C
```

You can make further refinements to this, for example to only show regions that have changed, keeping a certain amount of unchanged context around them, formatting them as `git diff` does, and so on. Although `git diff` doesn’t show numbers for each line, it does include a header before each change section that includes the offsets of that section, and the line numbers are needed to calculate that. The offsets help `git apply` find the right place to apply each change.

Git does not just use diffs to show information to the user via `git diff` and `git show`. It also diffs internally to implement other operations, such as `git merge`. Retaining the line numbers is important for the merge algorithm to correctly incorporate multiple changes within a file, and I’ll be looking at that in a future article.

Posted on February 17, 2017. This entry was posted in Ruby, Git. Bookmark the permalink.

← The Myers diff algorithm: part 2

→ Myers diff in linear space: theory