The If Works

by James Coglan

Open source

Books

Conference talks

Podcast interviews

Myers diff in linear space: implementation

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

—

In the last article we explored a linear space variant of the Myers diff algorithm, which is the version of the algorithm actually used by Git. We saw how it achieves its space-saving trick and how it can lead to multiple possible edit paths since it can discover many possible optimal edit paths.

Now I'd like to step through an implementation so you can see this working in code. We'd like a function `MyersLinear.diff(a, b)` to take two arrays of `Diff::Line` values and return an array of `Diff::Edit` objects that record the line number and content of each line and whether it's a deletion, insertion, or unchanged line.

Let's start off by creating a class. Many of the functions here will need to refer to a and b and so rather than pass those around everywhere, I'm just going to store them off as instance variables on this object.

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

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

  def diff
    # TODO
  end
end
```

Now, recall that this algorithm works by recursively splitting the edit graph into smaller and smaller sub-regions. As we implement this recursive process, we're going to need a way to represent the region we're currently working on. We could pass the co-ordinates or width and height around, but I'd like to encapsulate this idea as a single value. The following `Box` class represents a region by its top-left and bottom-right co-ordinates and provides some convenience methods that will clarify the subsequent code. In Ruby, `Struct` just creates a class whose constructor assigns its arguments to the given names.

```
Box = Struct.new(:left, :top, :right, :bottom) do
  def width
    right - left
  end

  def height
    bottom - top
  end

  def size
    width + height
  end

  def delta
    width - height
  end
end
```

Our implementation will work in two phases: first, we run the algorithm described in the previous article, that is the recursive procedure for finding the series of snakes that form the midpoint of each region. Then, we will use this series to reconstruct the line-by-line diff.

Let's introduce a method in the `MyersLinear` class for the first phase; this is the core of the recursive algorithm. Given any set of co-ordinates `left`, `top`, `right` and `bottom`, we find the middle snake within that box. If we don't find a snake, it means this region is already as small as possible (it's a single point) and we return `nil` to indicate this yielded no new moves. But if we do find a snake, we take its start and end points and construct two new regions from the top-left of our original box to the start of the snake, and from the end of the snake to the bottom-right of our box, and recursively find the edit path for each of them. Finally, we combine the edit paths for these two regions into a single list and return it.

```
def find_path(left, top, right, bottom)
  box   = Box.new(left, top, right, bottom)
  snake = midpoint(box)

  return nil unless snake

  start, finish = snake

  head = find_path(box.left, box.top, start[0], start[1])
  tail = find_path(finish[0], finish[1], box.right, box.bottom)

  (head || [start]) + (tail || [finish])
end
```

The `midpoint` method will return a representation of a snake, which is a pair of points for the start and end of the snake, for example `[[6, 5], [9, 7]]`. Each recursive call to `find_path` will either return a list of points or `nil`, and when it does return `nil` we use one of the snake's points in its place. For example, when `head` is `nil`, that means the region from the top-left of the box to the start of the middle snake is just a single point, so we can use the start of the snake to represent it. In this way, the method ends up building a list of the ends of all the middle snakes it found as it divided the graph into smaller boxes.

Having defined the basic outline of the algorithm, we now need to implement the method for actually finding the midpoint for each region. Our `midpoint` method shown below will take a `Box` structure and try to return a pair of points representing the middle snake, in the form `[[left, top], [right, bottom]]`. To do this, it runs the original version of the algorithm for finding the shortest edit, but it runs it simultaneously in both directions, one beginning at the top-left and one at the bottom-right.

Therefore, rather than a single `v` array for storing the best position for each diagonal, we need two: `vf` for the forward direction and `vb` for the backward one. As before, I'm seeding the forward-scan with `box.left`, i.e. the *x* co-ordinate of the start of the search. However, for the backward scan I'm starting with `box.bottom`, rather than `box.right`. When we scan forward, we're looking to maximise *x*, so as to prioritise deletions over insertions. But when we scan in the reverse direction, we'd like the opposite: we want prioritise insertions, so that those appear last in the final output. So, I'm basing the backward scan around minimising *y* rather than *x*, and have therefore used `box.bottom` as my starting value. This is an entirely arbitrary decision, and you can just as well use *x* for both scans and experiment to see what edit paths you end up with.

With the starting arrays set up, we then iterate *d* as before, but on each iteration we fill in a row of values in both the forwards and the backwards direction. I've implemented this as two method calls, `forwards` and `backward`, that perform the scan and yield any potential snakes they found.

To do this, they need to take both `vf` and `vb` so they can detect whether they've found any overlapping paths.

As soon as `forwards` or `backward` yields a snake, we return it. This mirrors the behaviour of Git, which stops on the first overlap it finds. However you could experiment with different results by storing off all the snakes yielded and then implementing a scoring method to choose between them.

```
def midpoint(box)
  return nil if box.size == 0

  max = (box.size / 2.0).ceil

  vf    = Array.new(2 * max + 1)
  vf[1] = box.left
  vb    = Array.new(2 * max + 1)
  vb[1] = box.bottom

  (0 .. max).step do |d|
    forwards(box, vf, vb, d) { |snake| return snake }
    backward(box, vf, vb, d) { |snake| return snake }
  end
end
```

The `forwards` and `backward` methods themselves will look quite familiar from the original Myers algorithm. To refresh your memory, here's our original `shortest_edit` method:

```
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
```

Our new methods (see below) are just performing the inner loop over k, but there are a few modifications we've had to make. The first is that, because we're trying to return the start and end points of the snake, not just the length of the edit sequence, we need to store the previous value of x before we make our next move. I've called this value px and it's the same as x if we take a downward step, but one less than x if we step rightward.

The second change is that rather than simply saying `y = x - k`, we need to adjust our x and y values relative to the box's top-left corner, so this becomes `y = box.top + (x - box.left) - k`. Then we must also calculate py, which will be one less than y if we moved downward (i.e. x did not change) and equal to y otherwise.

Then, as before, we follow any possible diagonal steps, taking care not to overshoot `box.right` or `box.bottom`, and we store off the final value of x. The final change is that at the end of this method, we check for overlaps

between the forward and backward scans, here comparing our current y value to the value found in the corresponding c diagonal in the vb array. If our y value is greater than or equal to the corresponding `vb[c]` value, the scans have overlapped and we can yield the move we just took back to the caller as a potential middle snake. Mirroring Git again, I'm iterating k in descending order, so we pick the overlap on the uppermost diagonal possible.

```
def forwards(box, vf, vb, d)
  (-d .. d).step(2).reverse_each do |k|
    c = k - box.delta

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

    y  = box.top + (x - box.left) - k
    py = (d == 0 || x != px) ? y : y - 1

    while x < box.right and y < box.bottom and @a[x].text == @b[y].text
      x, y = x + 1, y + 1
    end

    vf[k] = x

    if box.delta.odd? and c.between?(-(d - 1), d - 1) and y >= vb[c]
      yield [[px, py], [x, y]]
    end
  end
end
```

The `backward` method is much the same, except that it tries to minimise y rather than maximise x, and values in the `vb` array are indexed by c rather than k.

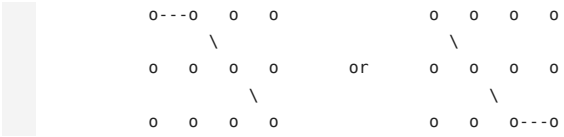
```
def backward(box, vf, vb, d)
  (-d .. d).step(2).reverse_each do |c|
    k = c + box.delta

    if c == -d or (c != d and vb[c - 1] > vb[c + 1])
      py = y = vb[c + 1]
    else
      py = vb[c - 1]
      y  = py - 1
    end

    x  = box.left + (y - box.top) + k
    px = (d == 0 || y != py) ? x : x + 1

    while x > box.left and y > box.top and @a[x - 1].text == @b[y - 1].text
      x, y = x - 1, y - 1
    end

    vb[c] = y

    if box.delta.even? and k.between?(-d, d) and x <= vf[k]
      yield [[x, y], [px, py]]
    end
  end
end
```

We've now seen all the methods that find the shortest edit path: `find_path` uses `midpoint` to find the middle snake, which in turn uses `forwards` and `backward` methods to scan the current box. `find_path` then recurses into the two smaller boxes defined by the middle snake. When we run this on the C code in the previous article, it generates this sequence of points:

```
[ [0, 0],   [1, 0],   [2, 2],   [3, 2],   [4, 2],
  [5, 4],   [6, 4],   [6, 5],   [9, 7],   [10, 9],
  [11, 9],  [11, 10], [11, 11], [12, 12], [13, 12],
  [13, 13], [14, 14] ]
```

Notice that not every point in this sequence is a single step from the one before it. For example, while the difference between `[6, 4]` and `[6, 5]` is a single downward step, the gap between `[6, 5]` and `[9, 7]`, or between `[9, 7]` and `[10, 9]`, is composed of multiple steps. To construct the complete diff, we need to work out all the single steps that make up the jumps between these points.

Take that snake we found at the first iteration of the algorithm, from `[6, 5]` to `[9, 7]`. We know a snake is composed of a single rightward and downward step followed by zero or more diagonal ones. Because of snakes found by scanning backward, they can also be a sequence of diagonal steps followed by a single rightward or downward one (leftward or upward in reverse direction). So, this snake could correspond to one of these sets of steps:



We know this move includes a rightward step rather than a downward one, because the change in the *x* co-ordinate from 6 to 9 is greater than the change in *y* from 5 to 7. A greater change in *y* than *x* would indicate a downward move.

As well as deciding which direction this step is in, we also need to decide whether it appears at the beginning or the end of the snake, that is, do we have the sequence on the left in the above figure, or the one on the right? To figure this out we can compare the lines at the starting point of the snake. If line 6 in the old version is equal to line 5 in the new one, then the first step is a diagonal and the rightward step comes at the end. Otherwise, it comes at the start.

Putting all this together, we can write a method the uses the output of `find_path` to to reconstruct the complete diff. The `walk_snakes` method below takes each consecutive pair of points in the `find_path` output and derives the complete path between them. For each pair of points, it first tries to walk a diagonal from the first point by checking the lines in the @a and @b strings. From there, it compares the change in *x* to the change in *y*; if it's negative then the *y* change is greater and we yield a downward step, but if it's positive then we yield a rightward step. Note that this code explicitly does nothing if the changes in *x* and *y* are equal. Some snakes will consist only of diagonals so there is no rightward or downward step to yield. Finally, we try to walk diagonals again to handle the case where the rightward/downward move happens first.

```
def walk_snakes(&block)
  path = find_path(0, 0, @a.size, @b.size)
  return unless path

  path.each_cons(2) do |(x1, y1), (x2, y2)|
    x1, y1 = walk_diagonal(x1, y1, x2, y2, &block)

    case x2 - x1 <=> y2 - y1
    when -1
      yield x1, y1, x1, y1 + 1
      y1 += 1
    when 1
      yield x1, y1, x1 + 1, y1
      x1 += 1
    end

    walk_diagonal(x1, y1, x2, y2, &block)
  end
end

def walk_diagonal(x1, y1, x2, y2, &block)
```

```
    while x1 < x2 and y1 < y2 and @a[x1].text == @b[y1].text
      yield x1, y1, x1 + 1, y1 + 1
      x1, y1 = x1 + 1, y1 + 1
    end
    [x1, y1]
  end
```

Now we can loop back to where we started and implement the `diff` method. All this does is call `walk_snakes` and consumes the pairs of points yielded by that method. For each pair of points, if *x* is unchanged then we have a downward move, which corresponds to an insertion; if *y* is unchanged then we have a deletion; and otherwise we have a diagonal move and the lines are equal.

```
  def diff
    diff = []

    walk_snakes do |x1, y1, x2, y2|
      if x1 == x2
        diff << Diff::Edit.new(:ins, nil, @b[y1])
      elsif y1 == y2
        diff << Diff::Edit.new(:del, @a[x1], nil)
      else
        diff << Diff::Edit.new(:eql, @a[x1], @b[y1])
      end
    end

    diff
  end
```

Putting everything together, when we run `Diff.diff(a, b, differ: MyersLinear)` where a and b are the two versions of the C code in the last article, we indeed get the diff that Git produces:

```
-    1         void Chunk_copy(Chunk *src, size_t src_start, Chunk *dst, size_t dst_st
+         1    int Chunk_bounds_check(Chunk *chunk, size_t start, size_t n)
     2    2    {
-    3              if (!Chunk_bounds_check(src, src_start, n)) return;
-    4              if (!Chunk_bounds_check(dst, dst_start, n)) return;
+         3        if (chunk == NULL) return 0;
     5    4
-    6              memcpy(dst->data + dst_start, src->data + src_start, n);
+         5        return start <= chunk->length && n <= chunk->length - start;
     7    6    }
     8    7
-    9         int Chunk_bounds_check(Chunk *chunk, size_t start, size_t n)
+         8    void Chunk_copy(Chunk *src, size_t src_start, Chunk *dst, size_t dst_st
    10    9    {
-   11              if (chunk == NULL) return 0;
+        10        if (!Chunk_bounds_check(src, src_start, n)) return;
+        11        if (!Chunk_bounds_check(dst, dst_start, n)) return;
    12   12
-   13              return start <= chunk->length && n <= chunk->length - start;
+        13        memcpy(dst->data + dst_start, src->data + src_start, n);
    14   14    }
```

The ability of this algorithm to make choices about which snake to pick means you can experiment with it. Try iterating k in ascending order to see how that changes the diffs you get. Try making the `backward` method work using *x* rather than *y* co-ordinates. Allow snakes to be selected from either direction, regardless of whether `delta` is even or not. Invent a scoring method for selecting between multiple candidate snakes. There are endless tweaks you can make to this algorithm, and getting it "right" is a question of observing how it works on real-world inputs. Why not have a look at Git's source code to see how it does things, and try out your own ideas for improving its output.

Posted on April 25, 2017. This entry was posted in Ruby, Git. Bookmark the permalink.

← Myers diff in linear space: theory

→ Merging with diff3