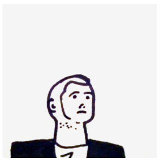

The If Works

by James Coglan

Buy my book, *Building Git*

Myers diff in linear space: theory

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

—

The interesting thing about diff algorithms is that they're a mix of computer science and human factors. There are many equally good diffs between two files, judging them by the length of the edit sequence, and choosing between them requires an algorithm that can best match people's intuition about how their code has changed.

As luck would have it, as soon as I had finished up my previous articles on Myers diffs and gone back to make progress on another project, I stumbled into a case where Git produced a confusing diff for a file I'd just changed, and I had to know why. Here's the portion of code I had been working on. It's a couple of C functions that copy bytes from one buffer to another, checking the sizes of the requested regions to make sure they're within the buffer. (This is not literally the code I was working on; I've removed a few things to make the example smaller.)

```
void Chunk_copy(Chunk *src, size_t src_start, Chunk *dst, size_t dst_start, siz
{
    if (!Chunk_bounds_check(src, src_start, n)) return;
    if (!Chunk_bounds_check(dst, dst_start, n)) return;

    memcpy(dst->data + dst_start, src->data + src_start, n);
}

int Chunk_bounds_check(Chunk *chunk, size_t start, size_t n)
{
    if (chunk == NULL) return 0;

    return start <= chunk->length && n <= chunk->length - start;
}
```

The change I made involved swapping the order of these two functions, that is:

```
int Chunk_bounds_check(Chunk *chunk, size_t start, size_t n)
{
    if (chunk == NULL) return 0;

    return start <= chunk->length && n <= chunk->length - start;
}

void Chunk_copy(Chunk *src, size_t src_start, Chunk *dst, size_t dst_start, siz
{
    if (!Chunk_bounds_check(src, src_start, n)) return;
    if (!Chunk_bounds_check(dst, dst_start, n)) return;

    memcpy(dst->data + dst_start, src->data + src_start, n);
}
```

Looking at this change, one would expect that a diff for it should show the shorter function `Chunk_bounds_check` being moved to appear before `Chunk_copy`, i.e.:

```
+int Chunk_bounds_check(Chunk *chunk, size_t start, size_t n)
+{
+    if (chunk == NULL) return 0;
+
+    return start <= chunk->length && n <= chunk->length - start;
+}
+
 void Chunk_copy(Chunk *src, size_t src_start, Chunk *dst, size_t dst_start, si
 {
     if (!Chunk_bounds_check(src, src_start, n)) return;
     if (!Chunk_bounds_check(dst, dst_start, n)) return;

     memcpy(dst->data + dst_start, src->data + src_start, n);
 }
-
-int Chunk_bounds_check(Chunk *chunk, size_t start, size_t n)
-{
-    if (chunk == NULL) return 0;
-
-    return start <= chunk->length && n <= chunk->length - start;
-}
```

And indeed, if you try running these two code snippets through our previous Myers implementation, this is exactly what you get. However, when you ask Git to compare these versions, here’s what happens:

```
-void Chunk_copy(Chunk *src, size_t src_start, Chunk *dst, size_t dst_start, si
+int Chunk_bounds_check(Chunk *chunk, size_t start, size_t n)
 {
-    if (!Chunk_bounds_check(src, src_start, n)) return;
-    if (!Chunk_bounds_check(dst, dst_start, n)) return;
+    if (chunk == NULL) return 0;

-    memcpy(dst->data + dst_start, src->data + src_start, n);
+    return start <= chunk->length && n <= chunk->length - start;
 }

-int Chunk_bounds_check(Chunk *chunk, size_t start, size_t n)
+void Chunk_copy(Chunk *src, size_t src_start, Chunk *dst, size_t dst_start, si
 {
-    if (chunk == NULL) return 0;
+    if (!Chunk_bounds_check(src, src_start, n)) return;
+    if (!Chunk_bounds_check(dst, dst_start, n)) return;

-    return start <= chunk->length && n <= chunk->length - start;
+    memcpy(dst->data + dst_start, src->data + src_start, n);
 }
```

This diff is *correct*; that is, it is a legal and minimal edit sequence that transforms the first version into the second, and it contains exactly as many changes as the “expected” diff. However, it is a *poor quality* diff in that it fails to convey the high-level meaning of the change to a programmer. It presents the change as a series of arbitrarily interleaved deletions and insertions, whereas the expected diff clearly shows the two functions switching places.

So, why is it that Git produces this diff rather than the one we’d expect? We can begin to answer this by drawing out the edit graph for this change, just as we did when we first investigated the Myers algorithm.

```
    0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
 0  o---o---o---o---o---o---o---o---o---o---o---o---o---o---o
    |   |   |   |   |   |   |   |   | \ |   |   |   |   |   |
 1  o---o---o---o---o---o---o---o---o---o---o---o---o---o---o
    |   | \ |   |   |   |   |   |   |   | \ |   |   |   |   |
 2  o---o---o---o---o---o---o---o---o---o---o---o---o---o---o
    |   |   |   |   |   |   |   |   |   |   | \ |   |   |   |
  3  o---o---o---o---o---o---o---o---o---o---o---o---o---o---o
     |   |   |   |   | \ |   |   | \ |   |   |   | \ |   |   |
  4  o---o---o---o---o---o---o---o---o---o---o---o---o---o---o
     |   |   |   |   |   |   |   |   |   |   |   |   | \ |   |
  5  o---o---o---o---o---o---o---o---o---o---o---o---o---o---o
     |   |   |   |   |   |   | \ |   |   |   |   |   |   | \ |
  6  o---o---o---o---o---o---o---o---o---o---o---o---o---o---o
     |   |   |   |   | \ |   |   | \ |   |   |   | \ |   |   |
  7  o---o---o---o---o---o---o---o---o---o---o---o---o---o---o
     | \ |   |   |   |   |   |   |   |   |   |   |   |   |   |
  8  o---o---o---o---o---o---o---o---o---o---o---o---o---o---o
     |   | \ |   |   |   |   |   |   |   | \ |   |   |   |   |
  9  o---o---o---o---o---o---o---o---o---o---o---o---o---o---o
     |   |   | \ |   |   |   |   |   |   |   |   |   |   |   |
 10  o---o---o---o---o---o---o---o---o---o---o---o---o---o---o
     |   |   |   | \ |   |   |   |   |   |   |   |   |   |   |
 11  o---o---o---o---o---o---o---o---o---o---o---o---o---o---o
     |   |   |   |   | \ |   |   | \ |   |   |   | \ |   |   |
 12  o---o---o---o---o---o---o---o---o---o---o---o---o---o---o
     |   |   |   |   |   | \ |   |   |   |   |   |   |   |   |
 13  o---o---o---o---o---o---o---o---o---o---o---o---o---o---o
     |   |   |   |   |   |   | \ |   |   |   |   |   |   | \ |
 14  o---o---o---o---o---o---o---o---o---o---o---o---o---o---o
```

You’**ll** **not**ice **th**is **gra**ph **cont**ains **t**wo **maj**or **diago**nals, **o**ne **o**f **len**gth 7 **beginn**ing **a**t (0,7) **a**nd **end**ing **a**t (7,14), **a**nd **o**ne **o**f **len**gth 6 **beginn**ing **a**t (8,0) **a**nd **end**ing **a**t (14,6). **The**se **diago**nals **repre**sent **t**he **t**wo **funct**ions **i**n **t**he **co**de **th**at **a**re **n**ot **modi**fied **b**y **t**he **cha**nge, **th**ey **sim**ply **tra**de **pla**ces. **Apa**rt **fr**om **the**se, **the**re **a**re **ma**ny **scatt**ered **sho**rt **diago**nals **th**at **corre**spond **to** **bra**ces **a**nd **bla**nk **lin**es **th**at **app**ear **i**n **bo**th **vers**ions. **The**se **typ**es **o**f **lin**es **app**ear **mult**iple **tim**es **i**n **t**he **te**xt, **whe**reas **a**ll **t**he **act**ual **lin**es **o**f **co**de **a**re **uni**que **a**nd **a**re **matc**hed **ju**st **on**ce **betw**een **vers**ions.

Now, **t**he **pa**th **correspond**ing **t**o **t**he **expe**cted **di**ff **i**s **th**is **o**ne, **whe**re **we** **fir**st **tra**vel **fr**om (0,0) **to** (0,7), **insert**ing **t**he `Chunk_bounds_check` **func**tion, **th**en **fr**om (0,7) **t**o (7,14) **t**o **displ**ay **t**he `Chunk_copy` **func**tion **unmod**ified, **th**en **fr**om (7,14) **t**o (14,14) **t**o **del**ete **t**he `Chunk_bounds_check` **fr**om **i**ts **prev**ious **loca**tion.

```
     0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
  0  o
     |
  1  o
     |
  2  o
     |
  3  o
     |
  4  o
     |
  5  o
     |
  6  o
     |
  7  o
       \
  8      o
           \
  9          o
               \
 10              o
                   \
 11                  o
                       \
 12                      o
                           \
 13                          o
```

```
                              \
14                              o---o---o---o---o---o---o---o
```

This path contains 7 diagonals, and recall that the aim of the Myers algorithm is to maximise the number of diagonals taken – that is, to find the longest common subsequence – so as to make as few changes as possible. However, it turns out that this path is not unique in containing 7 diagonals, for example, here is another path containing the same number:

```
    0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
 0  o---o
        |
 1      o
          \
 2          o---o---o
                    |
 3                  o
                      \
 4                      o---o
                            |
 5                          o
                              \
 6                              o
                                  \
 7                                  o---o
                                        |
 8                                      o
                                          \
 9                                          o---o
                                                |
10                                              o
                                                |
11                                              o
                                                  \
12                                                  o---o
                                                        |
13                                                      o
                                                          \
14                                                          o
```

If you compare this with the diffs above, you'll see that this is the path taken by Git. (There are many other possible paths through these diagonals that you can get if you vary the order of rightward and downward moves, this is just the one Git happens to take.) Because the number of matches between braces and blank lines – tokens that happen to be the same but aren't significant lines of program text – is equal to the number of lines in the entire matched `Chunk_copy` function, both paths are equally legitimate in terms of minimising the number of edits. However, it's not a useful diff to a human reader, and it will produce more merge conflicts than the expected diff because it doesn't do as good a job of separating sections of changes from each other.

But before we can look at how to fix these problems, we need to understand what causes them. Given the Myers algorithm we've seen so far does not choose this path, why does Git choose it?

The answer is that Git doesn't use the algorithm we implemented, it uses a variation of it. The original Myers algorithm takes a quadratic amount of space to calculate the edit sequence. The central trick in the Myers algorithm is that rather than requiring a general-purpose graph search algorithm, which would explore and store all possible branches, we can find the length of the longest common subsequence using only a single array of size proportional to N + M, the sum of the lengths of the two strings.

However if we want not just the length but the actual edit sequence, this requires storing a copy of the array as we move through the graph so we can backtrack at the end. In the worst case, it will take N + M moves to traverse the graph, and so the algorithm requires space proportional to $(N + M)^2$.

Requiring quadratic space is generally not a good property for an algorithm to have, since it rapidly becomes unfeasible to run the algorithm in memory as the input size increases. But fortunately, in the same paper where the original algorithm appears, Myers presents a modified version of it that runs in linear space, that is, space proportional to the sum of the string lengths. It is explained briefly in section 4b, but I'll give a worked example for our case here.

Whereas the original algorithm deterministically walks from the top-left to the bottom-right of the graph in a single pass, the linear space version uses a divide and conquer approach. The edit sequence is made up of a series of what Myers calls *snakes*, that is a rightward or downward step followed by zero or more diagonal ones. The linear-space version works by finding the *middle snake* of a possible edit path, that is a snake that crosses the halfway distance from top-left to bottom-right, and using the endpoints of that to divide the original graph into two smaller regions. It then works recursively on these regions until they're so small that no further work is required.

Here's an example: suppose that you were somehow able to determine that there exists a snake from (6,5) to (9,7), halfway between (0,0) and (14,14) in our example graph. (We will examine how to find this middle snake later.) This divides the original problem of getting from (0,0) to (14,14) into two smaller problems: getting from (0,0) to (6,5), and from (9,7) to (14,14). We can visualise this split like so:

```
       0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
   0   o---o---o---o---o---o---o
       |   |   |   |   |   |   |
   1   o---o---o---o---o---o---o
       |   | \ |   |   |   |   |
   2   o---o---o---o---o---o---o
       |   |   |   |   |   |   |
   3   o---o---o---o---o---o---o
       |   |   |   |   | \ |   |
   4   o---o---o---o---o---o---o
       |   |   |   |   |   |   |
   5   o---o---o---o---o---o---o
                                 \
   6                               @
                                     \
   7                                   @---o---o---o---o---o---o
                                           |   |   |   |   |   |
   8                                       o---o---o---o---o---o
                                           | \ |   |   |   |   |
   9                                       o---o---o---o---o---o
                                           |   |   |   |   |   |
  10                                       o---o---o---o---o---o
                                           |   |   |   |   |   |
  11                                       o---o---o---o---o---o
                                           |   |   | \ |   |   |
  12                                       o---o---o---o---o---o
                                           |   |   |   |   |   |
  13                                       o---o---o---o---o---o
                                           |   |   |   |   | \ |
  14                                       o---o---o---o---o---o
```

I have marked with a `@` symbol those points that have been removed from the problem and are not part of any remaining sub-region of the graph.

Having split the problem in two like this, we can apply the same technique to the sub-regions. In the (0,0)–(6,5) region there is a middle snake from (3,2) to (4,2), splitting the box into (0,0)–(3,2) and (4,2)–(6,5). In the (9,7)–(14,14) region the middle snake is from (11,10) to (11,11), splitting the box into (9,7)–(11,10) and (11,11)–(14,14).

```
    0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
 0  o---o---o---o
    |   |   |   |
 1  o---o---o---o
    |   | \ |   |
 2  o---o---o---o---o---o---o
                    |   |   |
 3                  o---o---o
                    | \ |   |
 4                  o---o---o
                    |   |   |
 5                  o---o---o
                              \
 6                              @
                                  \
 7                                  @---o---o---o
                                        |   |   |
 8                                      o---o---o
                                        | \ |   |
 9                                      o---o---o
                                        |   |   |
10                                      o---o---o
                                                |
11                                              o---o---o---o
                                                | \ |   |   |
12                                              o---o---o---o
                                                |   |   |   |
13                                              o---o---o---o
                                                |   |   | \ |
14                                              o---o---o---o
```

Applying the process yet again to these four boxes gives the following splits:

- (0,0)–(3,2) → (0,0)–(1,0), (2,2)–(3,2)
- (4,2)–(6,5) → (4,2)–(5,4), (6,4)–(6,5)
- (9,7)–(11,10) → (9,7)–(10,9), (11,9)–(11,10)
- (11,11)–(14,14) → (11,11)–(13,12), (13,13)–(14,14)

Notice that some of the resulting boxes have zero area, and some have corner points in common. This is perfectly normal, for example in the space from (4,2) to (6,5) below there are two resulting sub-regions: a 1-by-2 box (4,2)–(5,4), and a 0-by-1 box (6,4)–(6,5).

```
    0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
 0  o---o
        |
 1      @
          \
 2          o---o---o---o
                    |   |
 3                  o---o
                    | \ |
 4                  o---o---o
                            |
 5                          o
```

```
                              \
 6                              @
                                  \
 7                                  @---o---o
                                        |   |
 8                                      o---o
                                        | \ |
 9                                      o---o---o
                                                |
10                                              o
                                                |
11                                              o---o---o
                                                | \ |   |
12                                              o---o---o
                                                        |
13                                                      o---o
                                                        | \ |
14                                                      o---o
```

One **fin**al **pa**ss **a**nd **y**ou **c**an **s**ee **the**re **a**re **ve**ry **f**ew **poi**nts **le**ft **t**o **vis**it, **a**nd **a**ll **o**f **th**em **a**re **pa**rt **o**f **ze**ro-**ar**ea **o**r **un**it **box**es **whe**re **t**he **pa**th **fr**om **o**ne **e**nd **t**o **t**he **oth**er **i**s **obvi**ous. **Wh**at **w**e’re **le**ft **wi**th **i**s **t**he **pa**th **foll**owed **b**y **G**it’s **di**ff.

```
    0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
 0  @---@
        |
 1      @
          \
 2          @---@---o
                    |
 3                  @
                      \
 4                      o---@
                            |
 5                          @
                              \
 6                              @
                                  \
 7                                  @---o
                                        |
 8                                      @
                                          \
 9                                          o---@
                                                |
10                                              @
                                                |
11                                              o---o
                                                | \ |
12                                              o---o---o
                                                        |
13                                                      @
                                                          \
14                                                          @
```

This **i**s a **ne**at **tri**ck, **a**nd **i**t’s **actu**ally **t**he **rea**son **th**is **vari**ant **o**f **t**he **algor**ithm **c**an **disc**over **mult**iple **pat**hs **thro**ugh **t**he **gra**ph, **a**s **we**’**l**l **s**ee **shor**tly. **B**ut **t**o **g**et **i**t **t**o **wo**rk, **w**e **ne**ed **t**o **kn**ow **h**ow **t**o **fi**nd **tho**se **mid**dle **sna**kes. **Th**is **see**ms **li**ke a **chic**ken-**a**nd-**e**gg **prob**lem: **i**n **ord**er **t**o **ma**ke a **work**ing **di**ff **algor**ithm, **w**e **fir**st **ne**ed **someth**ing **th**at **c**an **fi**nd **opti**mal **ed**it **pat**hs, **whi**ch **i**s **ju**st **wh**at **t**he **orig**inal **Mye**rs **algor**ithm **do**es!

In **fa**ct, **th**is **i**s **exac**tly **wh**at **happ**ens. **T**o **fi**nd **t**he **mid**dle **sna**ke, **w**e **r**un **t**he **orig**inal **Mye**rs **algor**ithm **bo**th *forward* **fr**om **t**he **t**op-**le**ft, **a**nd *backward* **fr**om **t**he **bot**tom-**rig**ht, **unt**il **w**e **fi**nd a **poi**nt **a**t **whi**ch **val**ues **o**n **t**he **sa**me **diag**onal **me**et **ea**ch **oth**er. **Th**at **prob**ably **sou**nds a **b**it **cryp**tic **s**o **l**et’s **lo**ok **a**t a **sma**ll **exam**ple.

Say we’re working on the (0,0)–(3,2) box. Let’s reproduce it here, labelled with its (*x*, *y*) values, and also with each diagonal labelled with its value of *k*. Recall that *k* = *x* − *y* and it either increases or decreases by 1 each time we make a move. Similarly, when working on the backward scan, we will label diagonals with a value *c* which is zero at the bottom-right corner, increases with each step upward, and decreases with each step leftward. The relationship between *k* and *c* is:

k = *c* + ***delta***, where ***delta*** = ***width*** − ***height***, the difference between the box’s dimensions.

```
      x                       k
                                0     1     2     3
      0     1     2     3         \     \     \     \
y  0  o-----o-----o-----o           o-----o-----o-----o
      |     |     |     |      -1   |     |     |     | \
      |     |     |     |         \ |     |     |     |   2
   1  o-----o-----o-----o           o-----o-----o-----o
      |     | \   |     |      -2   |     | \   |     | \
      |     |   \ |     |         \ |     |   \ |     |   1
   2  o-----o-----o-----o           o-----o-----o-----o
                                      \     \     \     \
                                      -3    -2    -1      0
                                                            c
```

We run the original algorithm forward from (0,0) and backward from (3,2). Let’s begin exploring the graph, marking our progress as we go. Just like [our previous graph-walking explorations](), these diagrams place *d*, the number of moves taken, on the horizontal axis, and *k*, the number of the current diagonal, on the vertical axis, and record (*x*, *y*) points as we walk the graph. But unlike before, we also have the vertical axis labelled with *c*, which I’ve displayed on the right hand side, and we’ll include the points in the backward scan beginning from this side. Points on the same horizontal line in the forward and backward scan are on the same diagonal in the edit graph, and the forward and backward scans will overlap when a point in the forward scan has a greater *x* value than a point on the same diagonal in the backward scan.

We begin by marking our starting positions at opposite ends of a grid. (0,0) appears at (*d*, *k*) = (0,0), and (3,2) at (*d*, *c*) = (0,0).

```
    |   0       1       2       1       0   |
----+---------------------------------------+----
  k |                                       |  c
    |                                       |
  3 |                                       |  2
    |                                       |
  2 |                                       |  1
    |                                       |
  1 |                                  3,2  |  0
    |                                       |
  0 |  0,0                                  | -1
    |                                       |
 -1 |                                       | -2
    |                                       |
 -2 |                                       | -3
```

Next we explore the graph to one level deep, that is we move to (*d*, *k*) = (1,−1) and (1,1) and similarly for (*d*, *c*).

Moving downward from (0,0) lands us at (0,1), and moving rightward at (1,0). Likewise, moving leftward from (3,2) takes us to (2,2), which has a diagonal leading to (1,1), and moving upward takes us to (3,1).

```
    |   0       1       2       1       0   |
----+---------------------------------------+----
  k |                                       |  c
    |                                       |
  3 |                                       |  2
    |                                       |
  2 |                          3,1          |  1
    |                               \       |
  1 |          1,0                     3,2  |  0
    |       /                       /       |
  0 |  0,0                     1,1          | -1
    |       \                               |
 -1 |          0,1                          | -2
    |                                       |
 -2 |                                       | -3
```

So far, there is no opportunity for the forward and backward parts to meet; the forward path is on *k* diagonals −1 and 1 while the backward path is on *k* diagonals 0 and 2.

Let’s take one more move in the forward direction. We can move downward from (0,1) to reach (0,2), and from (1,0) to (1,1), taking a diagonal move to (2,2). We can move rightward from (1,0) to (2,0).

```
    |   0       1       2       1       0   |
----+---------------------------------------+----
  k |                                       |  c
    |                                       |
  3 |                                       |  2
    |                                       |
  2 |                  2,0     3,1          |  1
    |               /               \       |
  1 |          1,0                     3,2  |  0
    |       /       \               /       |
  0 |  0,0             2,2     1,1          | -1
    |       \                               |
 -1 |          0,1                          | -2
    |               \                       |
 -2 |                  0,2                  | -3
```

Now, we’ve found an overlap: the point (2,2) was found at (*d*, *k*) = (2,0), and the point (1,1) at (*d*, *c*) = (1,−1), which given the relationship between *k* and *c* means they live on the same diagonal. And, (2,2) on the forward path in (*x*, *y*) space is to the right of (1,1) on the backward path, and so these paths have overlapped. This means we’ve found our middle snake.

When ***del**ta* is even, the optimal path length will also be even, and so we will find an overlap after taking both a forward and a backward move on each iteration of *d*. But when it is odd, we will find the overlap after just taking a forward move on the final iteration, after several turns of both forward and backward moves. And so by convention, when ***del**ta* is odd, we consider the forward move we just made to be the middle snake, whereas when it’s even, we use the last backward move.

In this case, ***del**ta* is odd and so the middle snake is the last forward move, from (1,0) to (2,2). Therefore we can split our original box into (0,0)–(1,0) and (2,2)–(3,2).

```
      x

      0     1     2     3
y  0  o-----o
```

```
   1



   2              o-----o
```

Notice **th**at, **alth**ough I'**v**e **dra**wn **t**he **lin**es **o**f **mot**ion **o**n **t**he **tra**ces **abo**ve, **th**at's **pur**ely **t**o **he**lp **visua**lise **wh**at's **go**ing **o**n. **Sin**ce **w**e'**r**e **on**ly **do**ing **t**he **forw**ard **sc**an **o**f **t**he **orig**inal **algor**ithm **a**nd **w**e're **n**ot **go**ing **t**o **backt**rack **a**t **t**he **e**nd **t**o **calcu**late **t**he **fu**ll **pa**th, **w**e **d**on't **ne**ed **t**o **ke**ep **cop**ies **o**f **t**he **hist**ory **o**f **t**he **wa**lk **a**s **w**e **g**o - **w**e **on**ly **ne**ed **t**o **ke**ep **t**he **curr**ent **sta**te. **Th**at's **wh**at **all**ows **th**is **vari**ant **t**o **wo**rk **i**n **lin**ear **spa**ce; **w**e **on**ly **ne**ed **t**o **allo**cate **t**wo **arr**ays **o**f **si**ze ***wid**th* + ***hei**ght* + 1, **ha**lf **t**he **si**ze **o**f **t**he **arr**ay **i**n **t**he **orig**inal **vers**ion **sin**ce **w**e're **on**ly **go**ing **t**o **sc**an **half**way. **W**e **u**se **o**ne **arr**ay **f**or **t**he **forw**ard **sc**an **a**nd **o**ne **f**or **t**he **back**ward **sc**an.

Let's **d**o **o**ne **mo**re **exam**ple; **th**is **wi**ll **b**e **i**n **t**he **mid**dle **o**f **t**he **gr**id **a**nd **al**so **sho**ws **u**p a **situa**tion **whe**re **t**he **algor**ithm **c**an **ma**ke **choi**ces **betw**een **equa**lly **go**od **pat**hs. **W**e'**ll** **ta**ke **t**he (4,2)–(6,5) **b**ox.

```
      x                     k
                              0     1     2
      4     5     6             \     \     \
y  2  o-----o-----o               o-----o-----o
      |     |     |          -1   |     |     | \
      |     |     |             \ |     |     |   3
   3  o-----o-----o               o-----o-----o
      | \   |     |          -2   | \   |     | \
      |   \ |     |             \ |   \ |     |   2
   4  o-----o-----o               o-----o-----o
      |     |     |          -3   |     |     | \
      |     |     |             \ |     |     |   1
   5  o-----o-----o               o-----o-----o
                                    \     \     \
                                    -2    -1      0
                                                    c
```

Just **a**s **bef**ore, **w**e **ma**rk **o**ur **start**ing **poi**nts **i**n **approp**riate **posit**ions **i**n (*d*, *k*) **spa**ce.

```
    |   0       1       2       1       0   |
----+---------------------------------------+----
  k |                                       |  c
    |                                       |
  2 |                                       |  3
    |                                       |
  1 |                                       |  2
    |                                       |
  0 |  4,2                                  |  1
    |                                       |
 -1 |                                  6,5  |  0
    |                                       |
 -2 |                                       | -1
    |                                       |
 -3 |                                       | -2
```

We **fol**low **t**he **gra**ph **o**ne **mo**ve. **The**se **mov**es **a**re **a**ll a **sin**gle **st**ep, **exc**ept **f**or **t**he **mo**ve **down**ward **fr**om (4,2) **whi**ch **c**an **ta**ke a **diag**onal **t**o (5,4). **S**o **f**ar, **n**o **over**laps **a**re **poss**ible.

```
    |   0       1       2       1       0   |
----+---------------------------------------+----
  k |                                       |  c
    |                                       |
  2 |                                       |  3
    |                                       |
  1 |          5,2                          |  2
```

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

Now one more move. After this round, there are two overlaps: on the *k*=−2 diagonal both paths have found (5,5), and on the *k*=0 diagonal both have found (6,4). We now need to make a choice about which move we're going to consider our middle snake.

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

We know from the eventual outcome that Git picks (6,4) as the midpoint here, but why? To find out, I went looking in xdiffi.c in the Git source tree. This is actually part of LibXDiff, originally developed by Davide Libenzi and then imported and modified by the Git authors.

It actually boils down to the order in which Git visits graph locations: it iterates over *k* (or *c* in the backward direction) in descending order, from *d* to −*d*. That means in this final round it generates point (6,2) in the forward direction, and then (6,4), and then it stops since it's found an overlap. It never finds point (5,5) in the forward scan. If you change it to iterate from −*d* to *d* in ascending order, then it generates the same diff as the original Myers method. It's not making a decision between options as such, it's just an accident of implementation which one it picks, although this iteration order may have been chosen to prefer paths on the upper diagonals.

Having chosen (6,4) as the split point, that gives the following boxes after removing the middle snake.

```
      x

      4     5     6
y  2  o-----o
      |     |
      |     |
   3  o-----o
      | \   |
      |   \ |
   4  o-----o     o
                  |
                  |
   5              o
```

A similar choice presents itself on the top-level scan of the whole graph. As well as the snake from (6,5) to (9,7), there is one from (0,7) to (7,13), which is found via a path that took the diagonal from (14,14) to (13,13), but Git choses the first simply because it iterates over diagonals in descending order.

This process shows why it is that Git can choose a different path to the obvious one: rather than the original algorithm which is searching for a single point: the bottom-right, this algorithm might find many possible candidates for a midpoint and faces the problem (or opportunity!) of choosing between them. A lot of subtle implementation details affect the decisions it makes: what order we iterate diagonals in, whether we optimise for *x* or *y* in the forward and backward scans, which direction we pick when the two previous points have the same *x* value. We'll see some of these choices in the next article when we come to implement this algorithm.

Posted on March 22, 2017. This entry was posted in Ruby, Git. Bookmark the permalink.

← The Myers diff algorithm: part 3

→ Myers diff in linear space: implementation