



The If Works

by James Coglan

[Open source](#)

[Books](#)

[Conference talks](#)

[Podcast interviews](#)



[Buy my book, *Building Git*](#)

The Myers diff algorithm: part 1

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

As a programmer, you probably use a version control system such as [Git](#), and spend an awful lot of your time looking at diffs. You use them to check over your uncommitted work in progress, to look at what changed in a single commit, to compare two branches before performing a merge, and so on. Diffs are the language through which you understand how things have changed in your software.

But as well as being read by people, diffs are used by your version control system to automate changes. You can email a diff to someone and they can use the `patch` or `git apply` commands to merge it into their working copy. `git merge` has to reconcile and merge two or more change histories to produce a single tree, often reconciling changes within the same file. `git add --patch` lets you select individual changes from a working copy file rather than adding the whole file to the index, and that involves both you the user reading the diffs, and `git` selectively applying them to the indexed version of a file. And some version control systems use the differences between versions as their primary way to store the project history, rather than storing a snapshot of all the code for each commit.

So diffs are central to version control, but you might not have thought much about how they're generated. Often when you read a diff, it seems obvious to you which things should be marked as changes. You have an intuitive mental model of what it means to insert a new function into a file, or to delete a redundant one, or to rewrite a section. However, there's an awful lot more to diffing than meets the eye, and there are many ways to do it that produce different results.

Think for a moment about how you'd calculate a diff, and how you'd write a function to do it. You might have noticed that diff programs only show you what has changed, not what has stayed the same. How would you determine which parts of a file have not changed? Once you've found a difference between them, how would you find the next line in each version where the text matches up again? It's more complicated than it looks!

In this series of articles, I'd like to walk you through the default diff algorithm used by Git. It was developed by Eugene W. Myers, and the [original paper](#) is available online. While the paper is quite short, it is quite mathematically dense and is focussed on proving that it works. The explanations here will

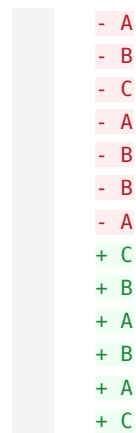
be less rigorous, but will hopefully be more intuitive, giving a detailed walk-through of what the algorithm actually does and how it works.

In this first article, we'll lay out the basic model of what the algorithm is trying to achieve and go through an example of how it works out the simplest set of edits to get from one version to another.

To use the example from the paper, say we want to calculate the difference between two strings:

- $a = \text{ABCABBA}$
- $b = \text{CBABAC}$

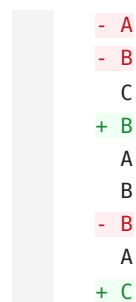
By “difference”, we mean a sequence of edits that will convert string a into string b . One possible such sequence is to simply delete each character in a , and then insert each character in b , or to use common diff notation:



```
- A
- B
- C
- A
- B
- B
- A
+ C
+ B
+ A
+ B
+ A
+ C
```

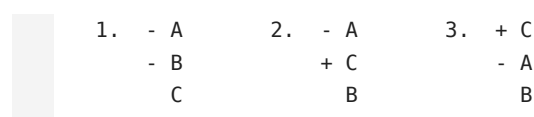
However, we wouldn't consider this a good-quality diff since it doesn't tell us very much. Changes to source code typically leave much of a file unmodified and we really want to see sections of code that were inserted or deleted. A diff that shows the entire file being removed and replaced with the new version isn't much use to us.

A better diff of these two strings would be:



```
- A
- B
C
+ B
A
B
- B
A
+ C
```

This makes the smallest possible number of changes to a in order to produce b , so it's a better visualisation of what really changed. It's not the only possible solution, for example these are also valid:



1.	- A	2.	- A	3.	+ C
	- B		+ C		- A
	C		B		B

- A	- C	- C
B	A	A
+ A	B	B
B	- B	- B
A	A	A
+ C	+ C	+ C

However, they are all *minimal*: they make the **smallest number of edits possible**, which in this case is five. What's **interesting about them** is they **differ in which sections they consider to be the same between the strings**, and **which order they perform edits in**. From **looking at diffs**, you **probably have an intuitive idea that diffs only show the things that changed**, but **these examples show that there are many possible interpretations of the difference between two files**.

So, the **purpose of diff algorithms** is to **provide a strategy for generating diffs**, where the **diffs have certain desirable properties**. We **usually want diffs to be as small as possible**, but **there are other considerations**. For **example, when you change something**, you're **probably used to seeing deletions followed by insertions**, not the **other way round**. That is, you'd **rather see solution 2 than solution 3 above**. And, **when you change a whole block of code**, you'd like to see the **whole chunk being deleted followed by the new code being inserted**, rather than **many deletions and insertions interleaved with each other**.

Good:	- one	Bad:	- one
	- two		+ four
	- three		- two
	+ four		+ five
	+ five		+ six
	+ six		- three

You also **probably want to see deleted or inserted code that aligns with your idea of the code's structure**, for **example if you insert a method**, you'd like that **method's end to be considered new**, rather than the **end of the preceding method**:

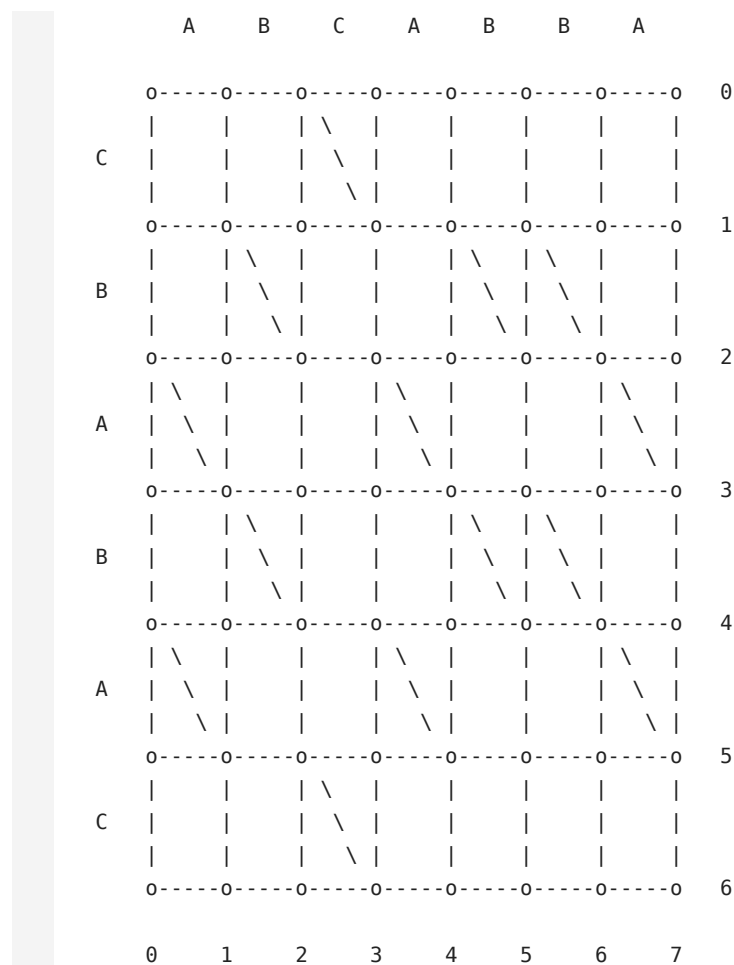
Good:	class Foo	Bad:	class Foo
	def initialize(name)		def initialize(name)
	@name = name		@name = name
	end		+ end
	+		+
	+ def inspect		+ def inspect
	+ @name		+ @name
	+ end		end
	end		end

Myers' algorithm is just one such **strategy**, but it's **fast** and it **produces diffs that tend to be of good quality most of the time**. It **does this by being greedy**, that is **trying to consume as many lines that are the same before making a change** (therefore **avoiding the "wrong end" problem**), and also by **preferring deletions over insertions when given a choice**, so that **deletions appear first**.

The **Myers paper** is based on the idea that finding the *shortest edit script* (SES) can be modelled as a **graph search**. Let's take our two strings, $a = \text{ABCABBA}$ and $b = \text{CBABAC}$, and build a graph of all the ways we can get from a to b .

The (x, y) co-ordinates in the grid shown below correspond to steps in the editing process; at $(0,0)$ we have string a , that is, we have not started editing. Moving rightward (increasing x) corresponds to deleting a character from a , for example moving to $(1,0)$ means we've deleted the first A from a . Moving downward (increasing y) corresponds to inserting a character from b , for example if we now move from $(1,0)$ down to $(1,1)$, we insert the first C from b , and our edited string is thus CBCABBA . At position $(4,3)$, we have converted ABCA into CBA , but we still need to convert BBA into BAC . The bottom-right position $(7,6)$ corresponds to converting string a fully into string b .

As well as moving rightward and downward, in some positions we can also move diagonally. This occurs when the two strings have the same character at the position's indexes, for example the third character in a and the first character in b are both C, and so position $(2,0)$ has a diagonal leading to $(3,1)$. This corresponds to consuming an equal character from both strings, neither deleting nor inserting anything.



The idea behind the Myers algorithm is quite simple: we want to get from $(0,0)$ to $(7,6)$ (the bottom-right) in as few moves as possible. A "move" is a single step rightward (a deletion from a) or downward (an insertion from b).

The **most number of moves we could take to get from a to b** is 13: the **combined length of the two strings**.

However, **walking diagonal paths is free since they don't correspond to making *changes*, thus we want to maximise the number of diagonal steps we take and minimise the number of rightward/downward moves**. The **examples above show that we can actually get from a to b making only five edits**, and **Myers provides a strategy for finding that pathway**.

To **develop an intuition for how the algorithm works**, let's **start exploring the graph**. To **try to find the shortest path to the bottom-right position**, we'll **explore every possible path from $(0,0)$ in tandem until we find a path that reaches the end**. I **recommend keeping the above grid handy while you follow this**.

Let's **start by recording our initial position at $(0,0)$** .

```
0,0
```

We have two options from this position: we can **move downward and reach $(0,1)$ or move rightward and reach $(1,0)$** .

```
0,0 --- 1,0
|
|
0,1
```

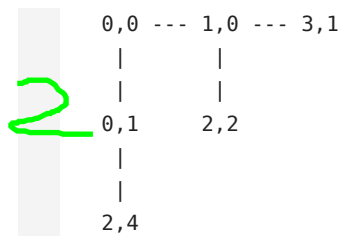
Now let's consider $(0,1)$. If we **move downward from here we reach $(0,2)$, but there is a diagonal from there to $(1,3)$, and from $(1,3)$ to $(2,4)$, and since diagonal moves are free we can say that moving downward from $(0,1)$ gets us to $(2,4)$ at the cost of only one move. Therefore we'll mark the move from $(0,1)$ to $(2,4)$ as a single step in our walk**.

Moving rightward from $(0,1)$ takes us to $(1,1)$ and again there is a diagonal from there to $(2,2)$. Let's mark both these moves on our walk.

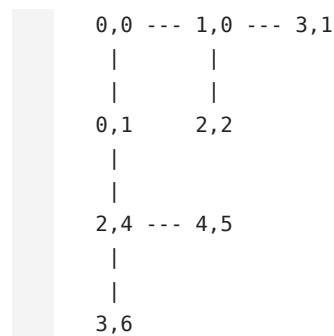
```
0,0 --- 1,0
|
|
0,1 --- 2,2
|
|
2,4
```

Now let's consider the **other branch we took from $(0,0)$, moving rightward to $(1,0)$. Moving downward from $(1,0)$ takes us to $(1,1)$, which as we just found out gets us to $(2,2)$. Moving rightward from $(1,0)$ takes us to $(2,0)$, which has a diagonal to $(3,1)$. Again, we'll record both these steps**.

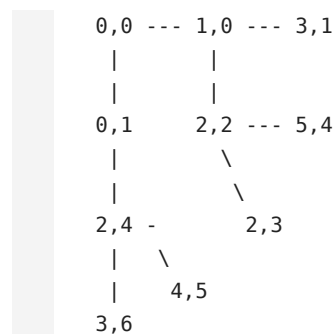
I'm recording $(2,2)$ as being visited via $(1,0)$ rather than $(0,1)$ for reasons that will become clear a little later. For intuition, consider that making a rightward move first means performing a deletion first, and we generally want deletions to appear before insertions.



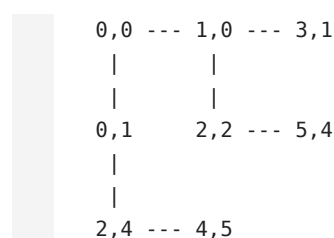
We've now fully explored the graph to two moves deep and we can begin on our third move. Moving downward from (2,4) gets us to (2,5), and from there is a diagonal to (3,6). Moving rightward from (2,4) takes us to (3,4), where again a diagonal takes us to (4,5).



Next, we consider (2,2). Moving rightward from there is as we've seen before: we move to (3,2), and follow the diagonals from there to (5,4). Moving downward introduces a new situation, however: this move gets us to (2,3) and there is no diagonal from there. Now, if we were doing a general-purpose graph search, we'd want to record both the result of moving rightward from (2,4) and the result of moving downward from (2,2), that is:



However, the structure of the particular graphs we're examining means that it's sufficient to just store the **best position** you can reach after a certain set of edits. The above record shows us that making two insertions then a deletion (down twice, and then right) gets us to (4,5), whereas making the deletion first, and then the two insertions, gets us to (2,3). So, we'll just keep the (4,5) result and throw the (2,3) away, indicating (4,5) is the best position reachable after one deletion and two insertions in any order.



```

|
|
3,6

```

Finally in our depth-2 scan, we visit (3,1). Moving downward from there goes to (3,2), which leads diagonally to (5,4), and so we'll again record this as a move downward from (3,1) rather than rightward from (2,2). Moving rightward from (3,1) gives (4,1), which has a diagonal to (5,2).

Here's the completed record after **three moves**:

```

0,0 --- 1,0 --- 3,1 --- 5,2
|         |         |
|         |         |
0,1      2,2      5,4
|         |         |
|         |         |
2,4 --- 4,5
|         |
|         |
3,6

```

You're probably getting the hang of this by now so let's rattle through the remaining moves. We can't move downward from (3,6), and moving rightward from there gives (4,6), which is also reachable downward from (4,5), so we'll mark it as such. Rightward of (4,5) is (5,5).

```

0,0 --- 1,0 --- 3,1 --- 5,2
|         |         |
|         |         |
0,1      2,2      5,4
|         |         |
|         |         |
2,4 --- 4,5 --- 5,5
|         |         |
|         |         |
3,6      4,6

```

(5,5) is also downward of (5,4) so we'll mark that, and moving rightward from (5,4) gives (6,4), with a diagonal leading to (7,5).

```

0,0 --- 1,0 --- 3,1 --- 5,2
|         |         |
|         |         |
0,1      2,2      5,4 --- 7,5
|         |         |
|         |         |
2,4 --- 4,5      5,5
|         |         |
|         |         |
3,6      4,6

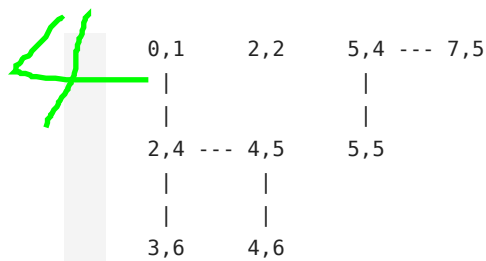
```

Downward from (5,2) also leads to (7,5), and moving rightward from (5,2) leads to (7,3), thus completing the **fourth row of the scan**.

```

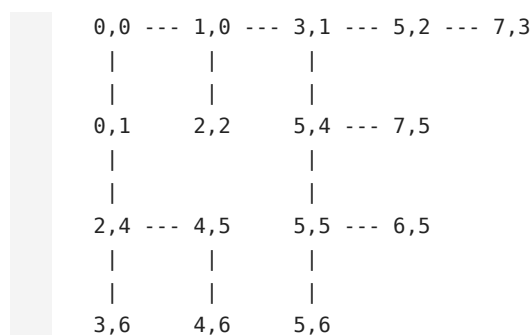
0,0 --- 1,0 --- 3,1 --- 5,2 --- 7,3
|         |         |
|         |         |

```

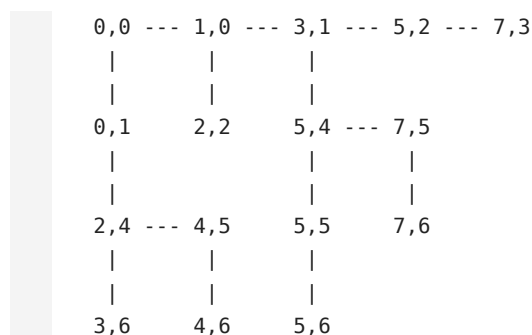


Now we **begin** the **fifth** row. Since we know there are **diffs** from **a** to **b** requiring only **five** edits, we expect this row of the scan to find the **bottom-right position**, (7,6).

There is **nothing** downward from (4,6), and **rightward** of that is (5,6), which is also downward from (5,5). **Rightward** of (5,5) is (6,5).



Finally, **moving downward from (7,5) gives (7,6) – the final position!** This is **certainly better** than (6,5), which we reached by going **right, right, down, down, right**, and so we **replace** it in our **trace** of the moves.



So that's the **basic idea** the **algorithm** is **based on**: **given** two **strings**, find the **shortest path** through a **graph** that **represents** the **edit space** between the two. We **explore** every **possible route** through the **graph** **breadth-first**, and **stop** as soon as we **reach** the **final position**.

In the **next article**, we'll look at how **Myers** **actually** **represents** this **process**, and **start** to look at **implementing** it in **code**.

Posted on [February 12, 2017](#). This entry was posted in [Ruby](#), [Git](#). Bookmark the [permalink](#).

Theme: Publish by Konstantin Kovshenin.