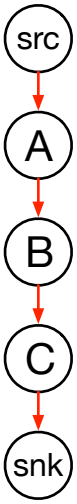
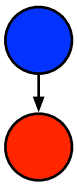
There is a single commit that created this file. The edges in the file graph are colored to indicate that they came from this commit.

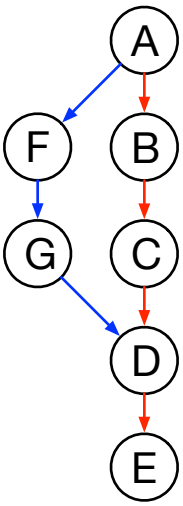


A file graph consists of nodes and edges. Each node contains a line of text (or a chunk of binary data). Each commit adds edges, and optionally nodes. In this case the red commit created the file ‘ABC’.

The first node of a file is always the src node, the last is the snk node. In general we won’t show these unless we need to.



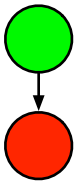
The blue commit depends on the red commit, but we’ll usually say that the blue commit dominates the red commit.



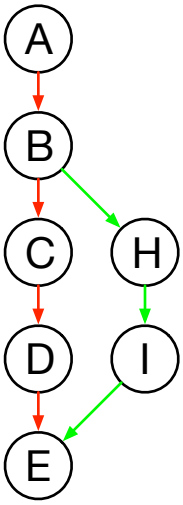
The *cogent path* is the path from the src node to the snk node that traverses all edges from each commit X unless another commit Y, which depends on X, has routed the cogent path around those edges.

If the cogent path exists then the nodes in the cogent path contain the data of the file and can be found trivially by always taking the outgoing edge that came from the most dominant commit.

In this case the red commit created the file ‘ABCDE’, and the blue commit replaced ‘BC’ with ‘FG’ to create ‘AFGDE’.

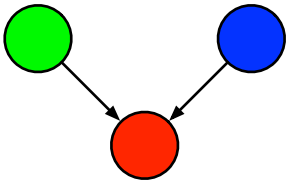


The green commit depends on the red commit.

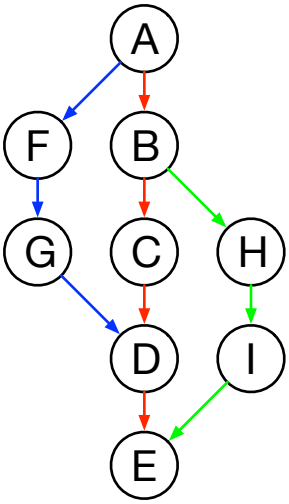


This is an example of a different commit applied to the same initial commit. This could have been a commit created by a different contributor that hasn’t tried to merge the blue commit into her repository.

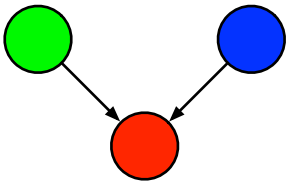
The cogent path here is ‘ABHIE’.

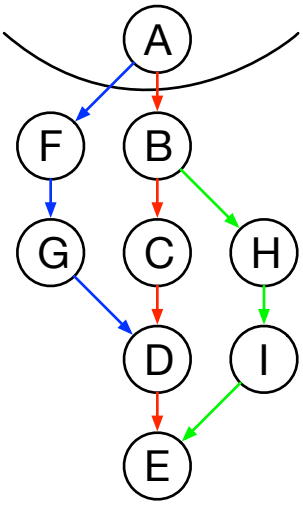


The blue and green commits are co-dominant, meaning neither dominates the other. Both dominate the red commit.



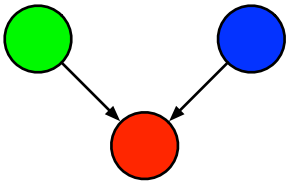
When these three commits are all merged together we find that there is no cogent path. Blindly following edges from the highest commit in the dependency graph will yield 'AFGDE', but it ignores the data from the green commit. Logically this must be a conflict, since data from the green commit has been deleted implicitly, and all modifications must be explicit, but we need a better way to define and detect conflicts.

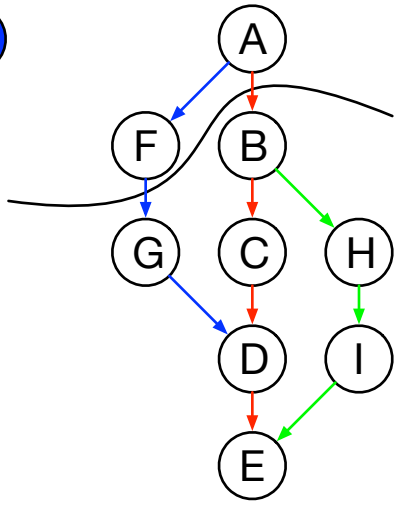




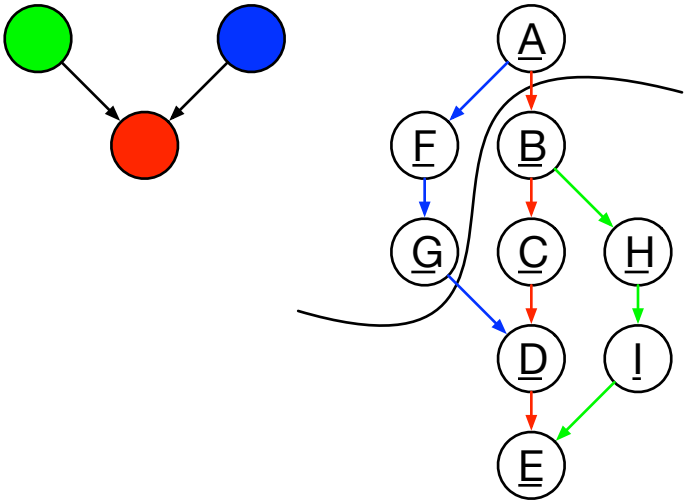
Let's define a cut through the graph called the *verge*. The verge starts containing no nodes and we advance it down the file one node at a time, only ever moving past a node if the verge has already passed all of that node's predecessors.

Given any position the verge is at we can always determine the next node rendered in the file, it will be the one we find by following the most dominant edge currently cut by the verge. If the cogent path exists then, by definition, this edge is on the cogent path. In this case the verge has passed A, and since the blue commit dominates the red commit, the next edge in the cogent path, if it exists, will be the edge from A to F.

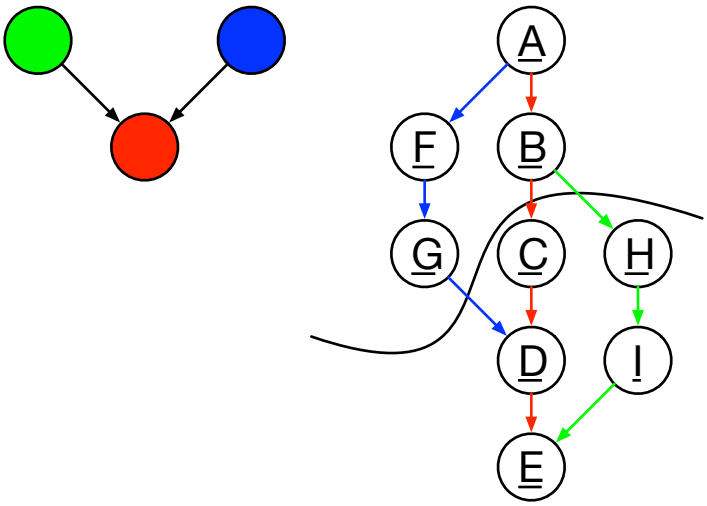




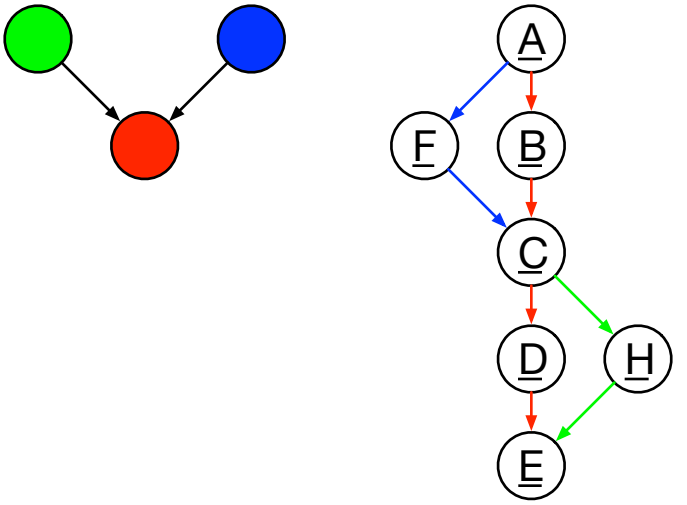
Here we've advanced the verge by moving it past F. Alternatively it could have been moved past B. We can see that the next edge in the cogent path, if it exists, will be the edge from F to G.



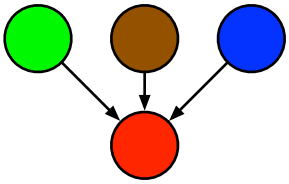
Again the verge has advanced, and again we could have chosen to advance past B but chose G instead.

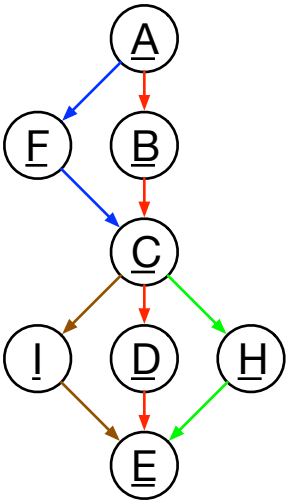


Things get interesting! We couldn't advance the verge past D because we hadn't also moved it past C, so we were forced to move it past B. Now the verge is cutting three edges, one from each commit. There is no way to decide between blue and green since the are co-dominant, so the cogent path does not exist and this file is in conflict.

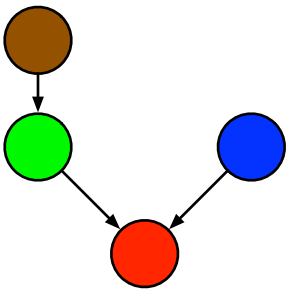


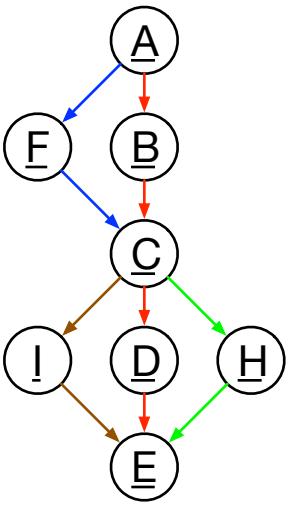
Here is an example with two co-dominant commits that don't conflict. There is no way to advance the verge such that it cuts a green and blue edge at the same time. Traversing the graph by always taking the edges from the most dominant commit yields 'AFCHE'.





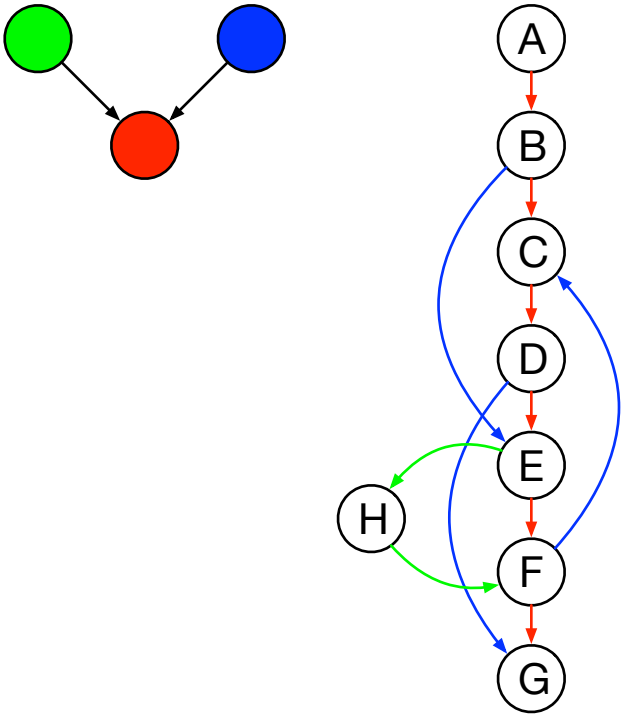
Here we've added an additional commit onto the graph above. Now we cannot advance the verge through the entire file without cutting two co-dominant edges: the edge from C to I and the edge from C to H. The cogent path can't exist because following the brown edges would avoid the green ones and vice versa.





This is like the above example except that the brown commit dominates the green commit. This means that the verge can't cut multiple independent edges, so the cogent path exists and the file reads 'AFCIE'.

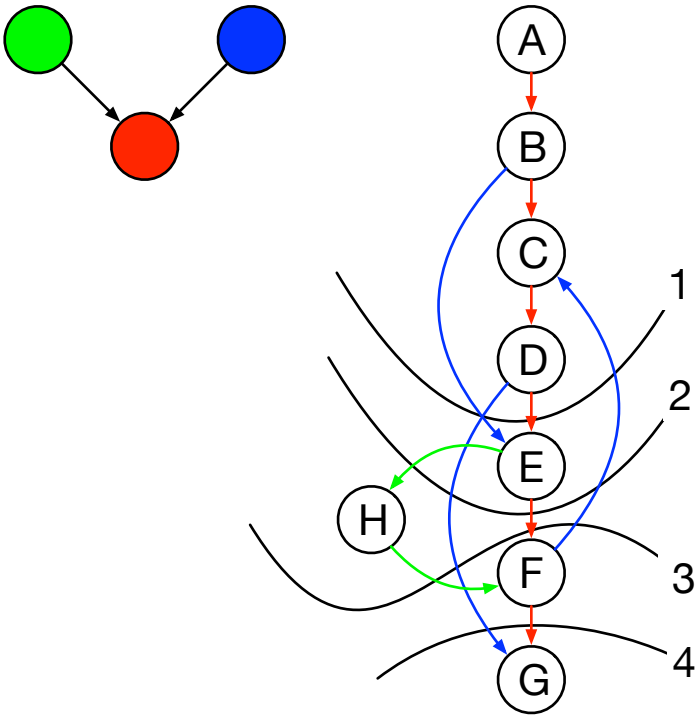
In simple terms this means that the person that made the brown commit knew about the green commit, and so the choice to avoid the green edges is intentional, whereas in the previous example it could not have been intentional.



This is an example of a move. The red commit alone yields ‘ABCDEFG’. Applying only the blue commit on top of the red commit yields ‘ABEFCDG’, it has moved the contents ‘EF’ in between B and C. Note that this is different than a deletion and an insertion because the co-dependent green commit gets moved along with it to yield ‘ABEHFCDG’.

The green and blue commits do not conflict, since a path (the cogent path) exists that traverses all of both of their edges. However our definition of the verge doesn’t work here since there are cycles. There is no way to advance the verge past B, because to advance to C we’d need to have already advanced it past F, which is obviously impossible.

To remedy this we need to allow the verge to ignore pairs of edges from the same commit that are going in opposite directions. In this case it means that the verge can advance past C because the incoming blue edge can be ignored.



Here is how the verge advances through several nodes in this graph.

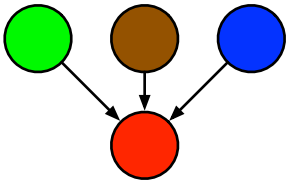
1. The verge is cutting one red and three blue edges, but we can ignore the reverse blue edge and one of the forward blue ones. Since the verge is only cutting blue and red edges there is no conflict.

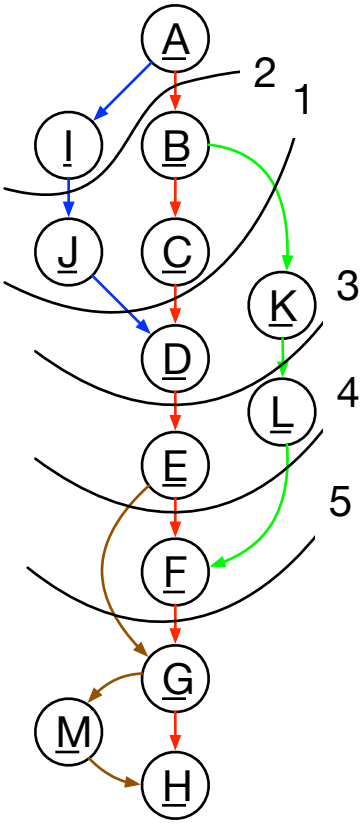
2. The verge is cutting one red, one green, and two blue edges. We can ignore both blue edges though since one is a reverse edge, so effectively it is only cutting red and green edges, so there is no conflict.

3. This is similar to 2, except the verge is cutting a different green edge this time.

4. A very ordinary cutting of a red and blue edge.

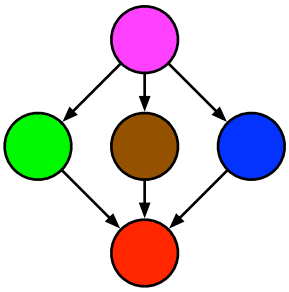
Note that from 2 we could not have advanced the verge past F since it requires that the verge passes H first.

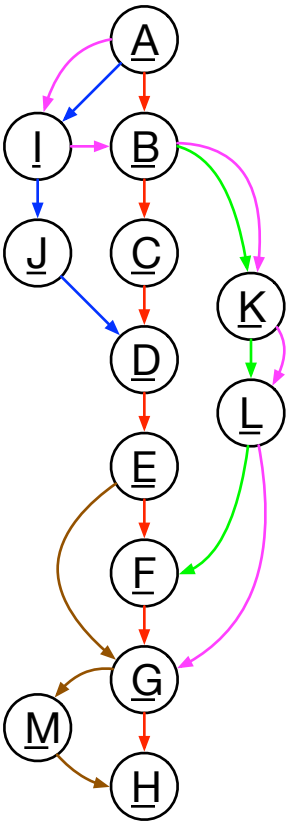




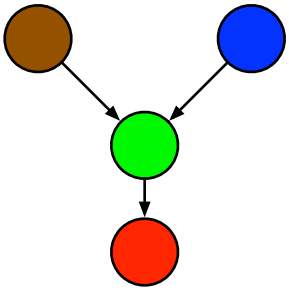
In this case the verge 1 has found two edges (blue and green) that conflict. Note that there are multiple verges that could have found this conflict, this is just one of them. To find the start of the conflict, logically, we want to scan backward until we find the last node at which green and blue agreed. When we do this eventually we'll end up with a verge like 2, where the green edge is no longer represented, but it's not clear yet where they agree. We continue representing the green commit now by tracking the red commit, since that is the most dominant edge from A to B in the green commit's transitive closure. We'll see later why this is necessary, but for now we're tracking blue and red. Eventually we'll advance the verge backward until it is cutting only the red and blue edges from A, at which point we know where our conflict begins, since our two conflicted commits are in agreement on a node.

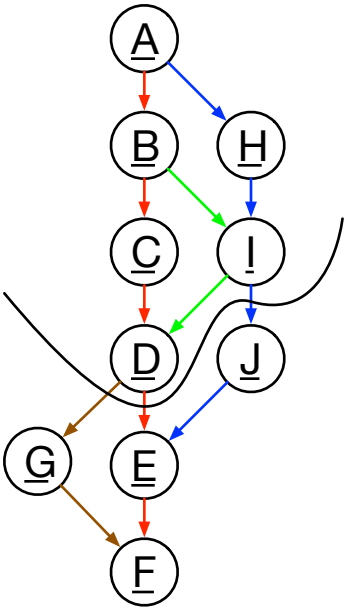
To find the end of a commit we advance the verge forward in a similar manner. Note that at verge 3, like at verge 2, it looks like there is no conflict, but that is because we are tracking red instead of blue. When we get to verge 4 we actually have a new conflict, brown. At verge 5 green has finally merged into red, and we can see that red and brown agree at G, so this is where our conflict ends. Note that advancing the verge further would cut another brown and red edge, so it's important to check if conflicted edges agreed on a node.





Here is one possible way of resolving the conflict in this graph. The magenta commit has incorporated part of the blue commit, all of the green commit, and also deleted the F, which was part of what the brown commit did. Now the cogent path clearly exists since there is a magenta path all the way from A to G, which was the range of our conflict.





This is an example that shows why we want to trace back using the most dominant edge in the transitive closure of an edge we're tracking. At the verge show here brown and blue are in conflict. When we move the verge backward past D, if we start tracking red in place of brown, we'll have to move the verge all the way back to A. If we track green instead we see that the conflict instead starts at I. This is a big difference, since we actually have to show these to users. In the first case we would have the users compare

ABIDGF vs AHIJEF

in the second case, where we're tracking the most dominant edge instead, we would show the user

IDGF vs IJEF