Recursive Backtracking Explained with the Flood Fill Algorithm

Borrowed with modifications from: www.inventwithpython.com

This is a cat:

This is a normal human:

This is a normal human who has been turned into a flesh-eating zombie:

Zombies are lazy and will only bite things that are next to them. Humans that are bitten will then turn into zombies:

There is an interesting recursive principle here, because the humans that have turned into zombies will start to bite other humans that are next to them, which will make more zombies, who bite more adjacent humans, which will make more zombies, and so on and so on in a chain reaction:

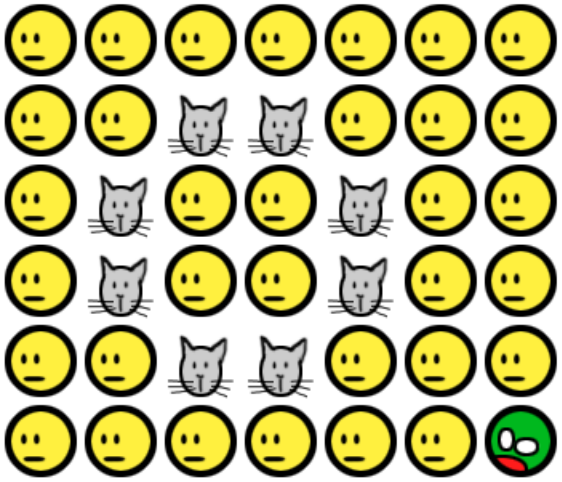
Zombies don't bite cats though. If you put a zombie next to a cat, you'll just end up with a zombie and a cat:

So as long as there is a cat between the human and the lazy zombie, the human is safe:

The same cannot be said of any humans who don't have a cat between them and a zombie:

So not only does this simple lazy-zombie principle cause a chain reaction of zombies, it also causes this chain reaction to stop when a cat is encountered.

Let's make a two-dimensional field of humans, cats, and zombies like this:

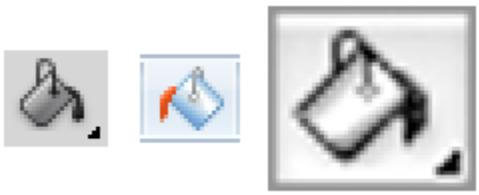


All we need is one starting zombie, and you'll see the infection spread until the entire human population is zombified. But there's hope. If there is an enclosed ring of cats blocking the initial zombie, then humanity is (slightly) saved. (In our example, zombies don't bite diagonally.)

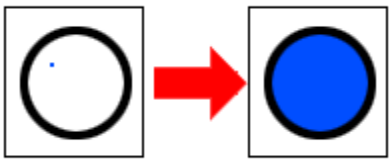
But if there is a gap in the cats, then the entire population is doomed:

This whole zombie thing is an elaborate and silly analogy for the ***flood fill algorithm***.

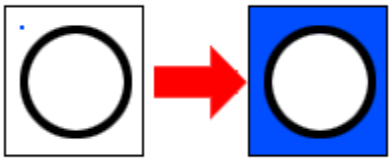
Flood filling is when you want to change the color of an area of color in an image. It's available in a lot of graphics programs and usually looks like a paint bucket being tilted over, like this:



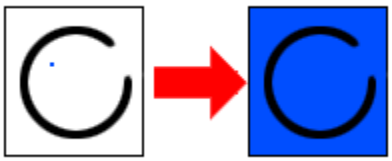
For example, if we flood fill a circle, the change will look like this (the blue dot indicates the starting location):



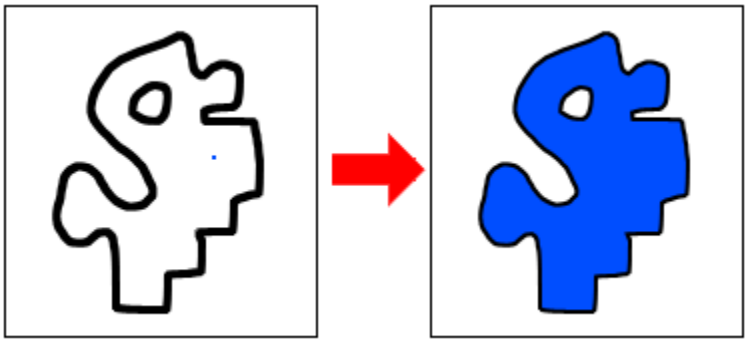
The blue flood filling starts at that blue point (which was originally white), so it will keep spreading as long as it finds adjacent white pixels. Imagine it as blue paint pouring on that dot and it keeps spreading until it hits any non-white colors (like the black circle). If we started the flood filling from the outside of the circle, then the entire outside area would be filled up instead:



Flood filling a circle that is not completely enclosed wouldn't work the same way. The flood filling color would "leak out" the open space in the circle and then fill up the outside space as well:



The really clever thing about flood fill is that it can fill up any arbitrary enclosed shape:



The image on your screen is nothing but discrete squares of color called pixels. In order to change the color of the pixels, our function will have to calculate the X- and Y-coordinates of each pixel that needs to be changed. It isn't really obvious how to do that. There are so many different possible shapes, so how can we write one function that will handle all the possible shapes there are?

Let's write some pseudo-code that does this:

```
void floodFill(x, y, oldColor, newColor):

    if location x, y is not oldColor OR is out of bounds //the base case

        return

    set pixel at x, y to newColor

    try expanding (filling) to the right, left, up, and down //the recursive calls
```

You call the `floodFill` function once with an X- and Y-coordinate, the color of the pixel you want to flood, and the new color that will flood the surface.

If the X- and Y-coordinate on the surface matches the old color, it changes it to the new color. Not only that, but it will then call `floodFill` on the pixel to its right, left, down, and up direction. It doesn't matter what order these pixels are called; the result will always be the same.

If those neighboring pixels are also the same as the old color, then the process starts all over again (just like a human turned into a zombie will begin biting all the neighboring humans). Those `floodFill` calls will make other `floodFill` calls, until they finally all return to the original `floodFill` call, which itself returns. The surface will then have the old color flooded with the new color.

This is the ***backtracking*** element; when a particular call reaches a "boundary" (a color that doesn't match the color to replace), it returns back to the location that called it, and tries all other directions *from that previous location*. These calls, in turn, try all other possible directions until they reach a boundary and return.

This technique, called recursive backtracking, has many other uses, but always follows a similar pattern: try all possible courses of action and, if failure occurs, return back (backtrack) to the parent and try all other courses of action.