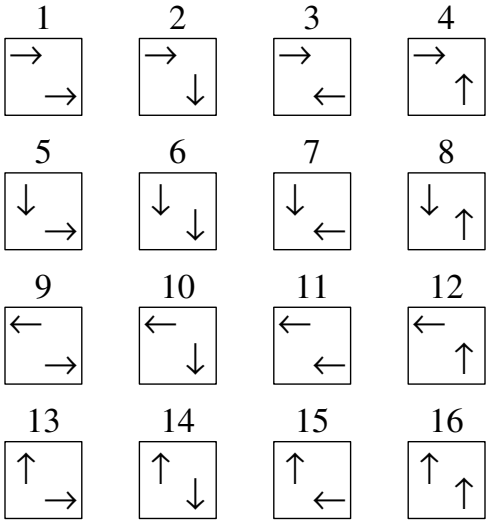
A STACKLESS FLOODFILL AUTOMATON

Nils M Holm, Jan 2023

1. The Algorithm

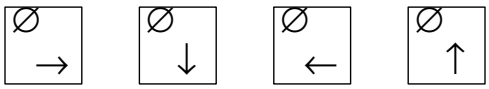
Floodfill is an algorithm that, given any point within a pixel-based representation of any shape, fills the entire shape with a given color. The algorithm is recursive by nature, because at any given point it has to try filling four adjacent points and then return to the origin, so the coordinates of each origin pixel are normally kept on a stack.

If you can spare 16 colors, though, no stack is needed.† Using those 16 otherwise unused color values, the following states of a finite state automaton can be encoded:

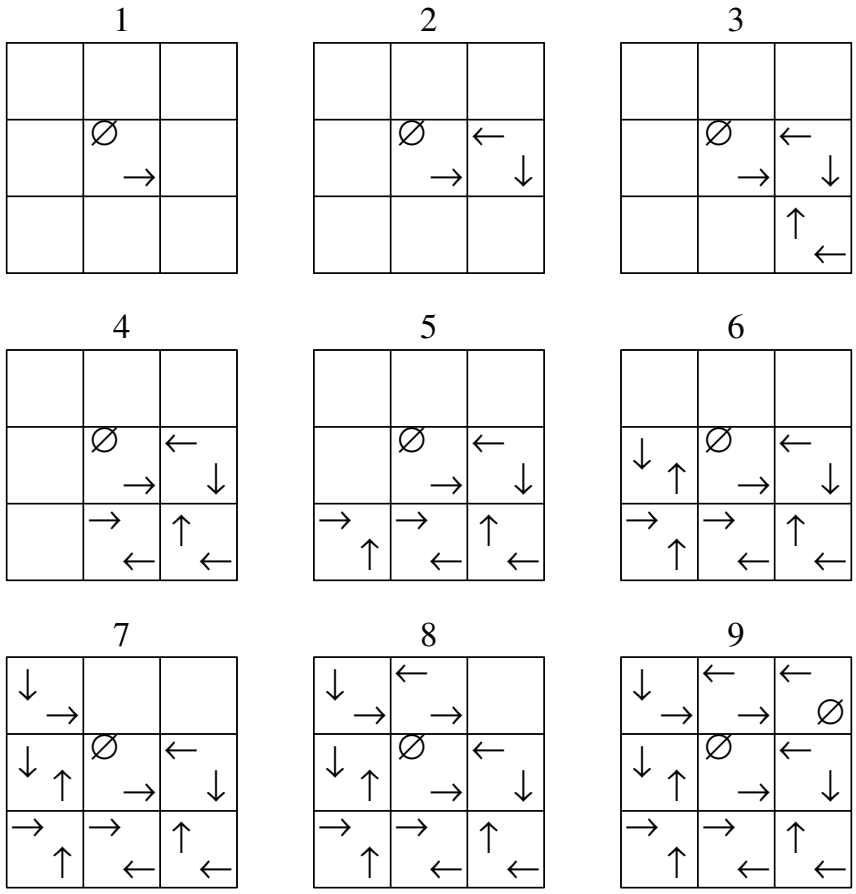


The upper left arrow in a state X denotes the direction in which the origin pixel lies, i.e. the pixel from which pixel X was tried. The lower right arrow denotes the direction to try next, using X itself as the origin. State pixels are stored directly in the image.

If another 4 values can be spared, the following four states can be added, which indicate the pixel at which the floodfill started, so there is no pixel from which this state originated. These states are not strictly necessary, though, because the coordinates of the starting point can be stored and used to identify this point.



† Nor any other dynamic data structure. I have created the stackless variant for use in a text-based minesweeper game on a BASIC system with only 7K bytes of program memory and 30 levels of subroutine nesting.



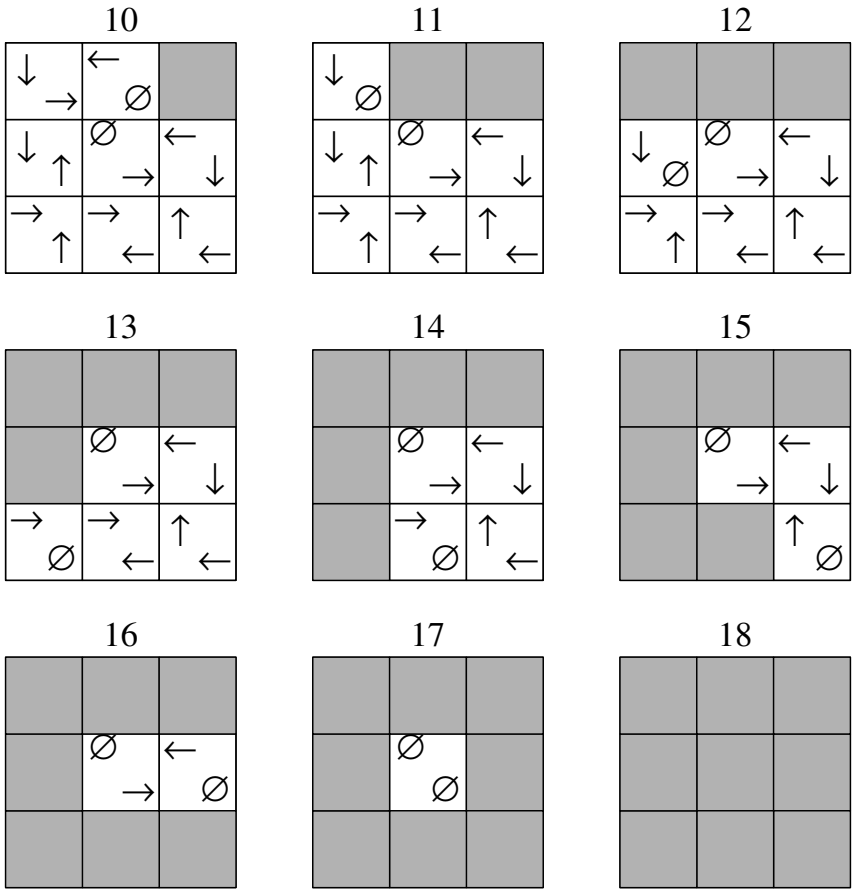
The above panels illustrate how the stackless floodfill works. It starts at the center (1) and tries the right direction first. There is no boundary condition found in the pixel to the right. A boundary exists when the pixel to be tried is already filled, either by a state or the fill color, or when another condition is met (such as reaching the edge of the image).

Because the right pixel can be filled, it is filled with a state indicating that its origin is to the left (2). There is no further pixel to the right, though, so the downward direction is tried next, which works.

In panel (3) the new state refers back to the pixel on the top. There is no unfilled pixel to the right nor at the bottom, so the left direction is tried next (and works).

In panel (5) the upward direction has to be used, because no unfilled pixel is on the right, left, or at the bottom.

The process continues until the upper right corner is reached in panel (9). Because all pixels are filled with state information at that point, there is no direction to turn to from the upper right pixel. The direction arrows (lower right) point all the way from the starting point to the upper right corner, and the origin arrows (upper left) point all the way back.



When there is no direction to go to, as indicated by the ∅ symbol in the place of a direction arrow, the current pixel is filled with the fill color and the origin arrow is used to return to the origin pixel of the current pixel.

The algorithm then backtracks all the way to the starting pixel and replaces all state pixels without a direction with the fill color. Of course this example is simplified. In a more complex shape, there may be multiple directions to try at a given point, so when returning to a pixel, a new branch may be started before running out of choices.

In any case, though, the algorithm will eventually end up at the starting point, which is identified either by a state with an origin of ∅ or by its coordinates. The starting point is then filled with the fill color and the algorithm terminates.

Like recursive floodfill the algorithm has linear $O(n)$ complexity, where n is the number of pixels to fill ($O(4n)$ to be precise, because it will try all four directions for each pixel). It can be extended to hex grids or voxel spaces, but then you will have to make use of 36 spare colors.

2. An Implementation

Here is an implementation in a minimal BASIC that I have created in the early 1990's. There should not be any surprises. a/b is the floored quotient of a and b, and $a\backslash b$ is the remainder of a/b.

```
500 LET N = 1
505 IF X < 0 GOTO 580
510 IF X >= Xmax GOTO 580
515 IF Y < 0 GOTO 580
520 IF Y >= Ymax GOTO 580
530 IF Z ( Y * Xmax + X ) <> EmptyColor GOTO 580
535 REM 'More boundary conditions here'
540 LET Z ( Y * Xmax + X ) = N
545 IF N \ 100 = 5 LET Z ( Y * Xmax + X ) = FillColor
550 LET D = N \ 100
555 IF D = 1 LET X = X + 1 : LET N = 301 : GOTO 505
560 IF D = 2 LET Y = Y + 1 : LET N = 401 : GOTO 505
565 IF D = 3 LET X = X - 1 : LET N = 101 : GOTO 505
570 IF D = 4 LET Y = Y - 1 : LET N = 201 : GOTO 505
580 LET P = N / 100
585 IF P = 0 RETURN
590 IF P = 1 LET X = X + 1
595 IF P = 2 LET Y = Y + 1
600 IF P = 3 LET X = X - 1
605 IF P = 4 LET Y = Y - 1
610 LET N = Z ( Y * Xmax + X ) + 1
620 GOTO 540
```

States are represented by numbers of the form $n = 100p + d$, where p is the origin (or ''parent'') and d is the direction. Right=1, down=2, left=3, up=4, ''out of choices''=5, and ''starting pixel''=0.

EmptyColor is the color to be replaced and *FillColor* is the color with which the shape shall be filled.

Z is a single-dimension array with a size of $X_{\max} \times Y_{\max}$. When filling visual shapes this would be an image or the video memory.

The boundary conditions in lines 505–520 make sure that the algorithm does not leave the array Z if the shape to fill intersects with its border. This is not necessary when the shape, including its border, lies entirely within Z.

Line 610 selects the next direction using $Z(Y \times X_{\max} + X) + 1$, which means that the direction codes in the states have to have successive values.