Advent of Code 2017 Day 22
90 Degree Rotations

William John Holden
wjholden@gmail.com

December 23, 2017

Abstract

A programming challenge requires 90 degree rotations of a vector. I enjoyed this challenge and wanted to explore both tricky and mathematical programming approaches to solving this problem. The solution I prefer takes advantage of transformation matrices and results in a very simple algorithm.

This year I have enjoyed programming challenges at `http://adventofcode.com/2017` (AoC). Day 22 was particularly interesting to me and I thought I would share some findings. The challenge requires the programmer to model transformations in a two-dimensional space for a computer virus notionally infecting computing nodes arranged on a grid.

The virus moves to the adjacent node to the right or left of the current node depending on whether the current node was or was not infected (respectively). The virus toggles the infection state at each "burst," so infected nodes become cleaned and clean nodes become infected.

I modeled my rotation with an d_x direction and d_y direction. I wasn't being particularly rigorous about it, but intuitively (d_x, d_y) models a vector representing the direction. The position of the virus can be modeled with position values p_x and p_y.

By restricting the values of d_x and d_y to only integers in $\{-1, 0, 1\}$, an algorithm for modeling changes in position can be as simple as $p_x = p_x + d_x$ and $p_y = p_y + d_y$.

The tricky bit is to construct a simple algorithm for updating (d_x, d_y). One *could* use a proper rotation angle Θ and use trigonometry. This technique has advantages and disadvantages. The advantage is that anyone with a reasonable mathematical background could recognize and understand the code, making the program easily maintainable and verifiable. The disadvantages of this approach are performance and errors. Surely there must be a faster way than to call sine and cosine functions. In languages like Java and C one must be very cautious of small but important floating-point errors.

Let us be very clear about our assumptions for this problem and their implications. First, this application has exactly four valid values $(d_x, d_y) \in \{(0, 1), (1, 0), (0, -1), (-1, 0)\}$. The application is therefore a finite-state machine in some sense. Second, the application supports only left and right rotations. Given four values and two operations (left and right), there are eight possible state transitions:

$$
\begin{aligned}
(-1,0) \leftarrow (0,1) &\rightarrow (1,0) \\
(0,1) \leftarrow (1,0) &\rightarrow (0,-1) \\
(1,0) \leftarrow (0,-1) &\rightarrow (-1,0) \\
(0,-1) \leftarrow (-1,0) &\rightarrow (0,1)
\end{aligned}
$$

Finite-state machines are generally a good idea. Knowledge of all possible states and their transitions make program verification and error detection easier than other approaches to programming. With only four states, each with only two transitions, it might be reasonable to enumerate them all in ugly but bulletproof branching statements.

If your mind works like mine, the idea of a series of nested `if-else` statements makes your skin crawl. We want a clever generating function to produce the series $\{0, 1, 0, -1, 0, 1, 0, -1, ...\}$.

My first thought was to find some clever modulo arithmetic (perhaps `dx = (dx - 1) % 2`) but I could not get this to work.

In retrospect, a neat approach might have been to instantiate a circular array `int d[] = { 0, 1, 0, -1 }` and use an index pointer `p`. Values for d_x are given by `d[p % 4]` and d_y by `d[(p + 1) % 4]`. Increment `p` to turn right and decrement `p` to turn left. I have not tested this idea but it should work and it looks very fast.

Still, I wanted a way to get the next value (d_x, d_y) by rotating the current vector. I don't remember much from my linear algebra class, but a cursory search of the Internet reminds me of transformation matrices. To rotate a two-dimensional vector v around the origin by angle Θ, the rotated vector v' is the cross-product $R_\Theta \times v$ where

$$
R_\Theta = \begin{bmatrix} \cos(\Theta) & -\sin(\Theta) \\ \sin(\Theta) & \cos(\Theta) \end{bmatrix}
$$

For a left turn, $\Theta = \pi/2$ and

$$
R_{\frac{\pi}{2}} = \begin{bmatrix} \cos(\frac{\pi}{2}) & -\sin(\frac{\pi}{2}) \\ \sin(\frac{\pi}{2}) & \cos(\frac{\pi}{2}) \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}
$$

Applied to the AoC problem, the vector (d_x, d_y) becomes (d'_x, d'_y) by the cross-product

$$
\begin{bmatrix} d'_x \\ d'_y \end{bmatrix} = R_{\frac{\pi}{2}} \times \begin{bmatrix} d_x \\ d_y \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} d_x \\ d_y \end{bmatrix} = \begin{bmatrix} (0 \times d_x) + (-1 \times d_y) \\ (1 \times d_x) + (0 \times d_y) \end{bmatrix} = \begin{bmatrix} -d_y \\ d_x \end{bmatrix}
$$

This means that calculating $\pi/2$ counter-clockwise vector rotations (aka, "turn left") is very simple:

```
int tmp = dx;
dx = -dy;
dy = tmp;
```

Rotate to the right by $-\pi/2$ with

$$\begin{bmatrix} d'_x \\ d'_y \end{bmatrix} = R_{\frac{-\pi}{2}} \times \begin{bmatrix} d_x \\ d_y \end{bmatrix} = \begin{bmatrix} \cos(\frac{-\pi}{2}) & -\sin(\frac{-\pi}{2}) \\ \sin(\frac{-\pi}{2}) & \cos(\frac{-\pi}{2}) \end{bmatrix} \times \begin{bmatrix} d_x \\ d_y \end{bmatrix}$$
$$= \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \times \begin{bmatrix} d_x \\ d_y \end{bmatrix} = \begin{bmatrix} (0 \times d_x) + (1 \times d_y) \\ (-1 \times d_x) + (0 \times d_y) \end{bmatrix} = \begin{bmatrix} d_y \\ -d_x \end{bmatrix}$$

So to rotate a vector v by $\pi/2$ right (clockwise around the origin, effectively $-\pi/2$) we simply need:

```
int tmp = dx;
dx = dy;
dy = -tmp;
```

The advantage of this solution is that it is very fast and can be applied to any vector, not just the four useful in this AoC application. A disadvantage is that the code listings are not self-explanatory. With that said, bit-twiddling tricks are seldom obvious. I would recommend leaving explanatory comments anytime a program uses math tricks.