The Drunken Sailor’s Challenge

Tucker Balch

April 18, 1995

1 Introduction

Ahhh, life in the Navy. It’s not just a job, it’s a robotics problem! You wake up to the sunrise after your last night of liberty in Singapore. You probably drank too much, but it was worth it. Anyhow, you find yourself amidst hundreds of cargo containers bound for Malaysia. Your watch reads seven forty-five. Uh oh, you’ve only got half an hour to reach your ship before it leaves port. Good thing that from here you can see the mast. Unfortunately, you still must negotiate the maze of cargo between here and there. Which brings us to the problem: how many unanesthetized brain cells does it take to reach your berth?

The challenge is for you to write an efficient C routine, `navigate()`, for navigation across a two dimensional field with obstacles. Before you get worried that you’ll have to write a complicated program, I’ll tell you right now that it’s easy; all the hard stuff is done for you. We will splice your function into the sailor’s brain, `sailor.c`, and see if he makes it home. As input, your function will get a list of nearby obstacles, and an approximate heading to the ship. It should return a value indicating which way to go. Your function doesn’t necessarily have to keep track of where the sailor is, the main program will do this. `main()` will repeatedly call your function and move the sailor in whatever direction you say until he reaches his ship.

If your function guides our sailor home, we will measure its efficiency in three ways:

1. **Path length**: how far did the sailor have to walk?
2. **Brain cells**: how large is your compiled function?
3. **Time**: how long did it take? Remember, it hurts to think with a hangover.

2 Nitty Gritties

Here are more details on the rules of the game:

For simplicity, this game takes place on a rectangular grid, so there are only a finite number of points the sailor can occupy. In fact, the grid is 23 by 80
cells, a convenient size for ASCII character animation. At each step, the sailor may move in any of eight compass directions: north, northeast, east, etc. (from here on out we'll abbreviate these with capital letters). Moves E, W, N or S cost 1 path unit, while diagonal moves (NE, SE, SW or NW) cost \( \sqrt{2} \) units.

Each time your function, `navigate()`, is called it receives a list of nearby obstacles and the direction to the ship. The obstacle list is an array of nine integers set to `EMPTY` or `OCCUPIED` depending on whether or not a cargo container is blocking the way. Here is how the obstacle array is indexed:

<table>
<thead>
<tr>
<th>NW</th>
<th>N</th>
<th>NE</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>SAILOR</td>
<td>E</td>
</tr>
<tr>
<td>SW</td>
<td>S</td>
<td>SE</td>
</tr>
</tbody>
</table>

These symbols have been defined for you in `sailor.h`, so the result of an expression like `if (obstacles[NW] == OCCUPIED)` would tell you if there is an obstacle to the NW of the sailor. The `SAILOR` element is always `EMPTY`. Note that if you ever command the sailor to move over an obstacle your command will be ignored. The direction to the ship is also given as one of the eight compass directions.

If you'd like to keep track of the sailor's location, you'll need to declare some static variables on your own (you don't need state information, do you?).

### 3 Example Navigator

Okay, on to a concrete example. This `navigate()` function is automatically compiled with the distributed code. It uses a rather simple approach: first, it attempts to go towards the ship, but if that direction is blocked, it finds the next open direction.

```c
#include "sailor.h"

int navigate(int obstacles[9], int ship_direction)
{
    int i;

    if (obstacles[ship_direction] == EMPTY)
        return(ship_direction);
    else
    {
        for(i = NW; i <= SE; i++)
            if ((obstacles[i] == EMPTY) && (i != SAILOR)) break;
        return(i);
    }
}
```

2
This example is in examples/example1.c. If you’ve already unpacked the
distribution, you can see this one run just by typing demo. When you do that
you’ll see the following picture on your screen:

```
O O O O O O O O O O O O O O O O O O O O O O O O O O O O O O O O
O O O O O O O O O O O O O O O O O O O O O O O O O O O O O O O O
O THE SHIP X O O O O O O O O O O O O O O O O O O O O O O O O O
O O O O O O O O O O O O O O O O O O O O O O O O O O O O O O O O
O O O O O O O O O O O O O O O O O O O O O O O O O O O O O O O O
O O O O O O O O O O O O O O O O O O O O O O O O O O O O O O O O
O O O O O O O O O O O O O O O O O O O O O O O O O O O O O O O O
O O O O O O O O O O O O O O O O O O O O O O O O O O O O O O O O
O O O O O O O O O O O O O O O O O O O O O O O O O O O O O O O O
O O O O O O O O O O O O O O O O O O O O O O O O O O O O O O O O
```

The screen will look like this for a few seconds, then the * will move around
and eventually reach the X. A trail of dots will trace the sailor’s path.

This test scenario is in the test_worlds/test1. Obstacles are repre-
sented by Os, the goal is an X, and the sailor is an *. Other characters are
ignored but will be printed on the screen. You can make the problem more
difficult by editing test_worlds/test1, then typing demo again. If you seal off
the path to the left of the sailor, you’ll find he gets trapped!

## 4 How to Get the Code

So you think you know a better way? First get the code distribution by
anonymous ftp at cc.gatech.edu/people/tucker/sailor.tar.Z. Uncompress
it with the command `uncompress sailor.tar`, then un-tar it: `tar -xmf sailor.tar`. You
should now have a directory called sailor. Under that you will find a doc
subdirectory which contains this paper and a src directory which contains the
code. At Georgia Tech, you can also find this stuff in `~tucker/sailor/sailor.tar.Z`

If you are on a Sparc, you should be able to test the program by just typing
demo. Otherwise you will need to recompile. To do that, move into the src
directory and type `make sailor`. You will need the curses library and the gcc
compiler. These are available on most Unix systems.
5 How to Test Your Algorithm

Just edit src/navigate.c to your satisfaction, then recompile with make sailor. You shouldn’t have to do anything else (other than debug your code).

6 Super Challenges

Can you write a navigation function that uses no local variables? For those unfamiliar, this type of solution would be called “purely reactive” in robotics. Are there any obstacle/goal/sailor configurations which your function cannot solve? Try some of the other test scenarios like test_worlds/arkintrap. If the size of the playing field is not bounded, is it even possible to develop a purely reactive navigation function that will always work? If not, how much state (or how many local variables) do you need?

7 How to Submit Your Algorithm

I’m interested in your algorithm. Please send your navigate.c file to me by e-mail at tucker@cc.gatech.edu. I’m also interested in hearing your suggestions and bug reports.

I expect at some later date to hold a “Navigation Showdown” between the algorithms I receive. The metrics listed up top (path length, code size, and time), will be used to rank each submitted function. I’ll be glad to inform you of how yours fared. Good luck, sailor.