Count Luck

Vincent Foley — vfoley@gmail.com

July 26, 2016

1 Preface

This file is a literate program solving the Count Luck problem from HackerRank. The solution is written using Org mode and Rust.

- The PDF version of this program is available here
- The Org mode version of this program is available here
- The Rust version of this program is available here

I have known about literate programming for a few years, but recent events have made me want to look at it again: Howard Abrams posted a video on Literate Devops, Donald Knuth gave a keynote at UseR! on literate programming, John D. Cook wrote a blog post on literate programming, and I saw some people mention it in discussions on Hacker News and Lobsters. This has made me want to gain a deeper insight about the subject: the main goal of this program (essay?) is as a vehicle to explore this style of programming, to get some practical experience, and to understand some of the pros and cons.

I must mention that this is only my second attempt at writing a literate program, and as a reader, you will notice that the style of both the prose and the code is a little bit awkward. In addition, the structure of the document is most likely going to be biased towards a regular programming style.

With that caveat out of the way, let's look at the problem.

2 Overview

(I assume that you've skimmed the problem at this point.) The problem is set in the Harry Potter universe: Hermione and Ron are in the woods, and they must make their way to a portkey. They can only move up, down, left, or right (no diagonal moves) and they cannot move through trees, nor off the map. I have replicated a sample scene below. This scene is one of the inputs provided to our program: the X's represent trees, the periods are the ground that Hermione and Ron can walk on, the M is their starting location and the asterisk is the portkey.

```
.X.X......X
.X*.X.XXX.X
```

```
.XX.X.XM...
......XXXX.
```

Hermione and Ron walk along the road until they reach a fork: at that point, Hermione zaps her wand to indicate which path they should pick to reach the portkey. In the example above, she will zap her wand 3 times. The diagram below shows why: when Hermione and Ron are at (2,9), they can either go up to (1,9) or right to (2,10); Hermione zaps her wand and it indicates that they should go to (1,9), and that is the first usage. (Note that only the unexplored roads were considered: going back to (2,8) was not an option, otherwise Hermione would need to zap her wand at every step.) The second and third zaps of the wand occur at the points (0,5) and (3,3).

2.1 Solution outline

This is clearly a graph traversal problem. As I see it, we must solve two sub-problems:

1. We must find the path from M to $*$ (the problem statement guarantees that only a single path from M to $*$ exists);
2. we must count how many times Hermione and Ron arrive at a fork in the road.

The first order of business will be to create a graph representation of the forest. If a point is a road, the start, or the portkey, it is a node and it will be represented by a tuple of two `usize` values representing the point's x and y coordinates. An edge connects two nodes N_1 and N_2 if the two nodes are adjacent, i.e., are left/right or up/down of one another.

With a graph in hand, we can solve (1) by applying a breadth-first search algorithm. BFS will find the shortest path from M to $*$. (Because there is only one path, a DFS traversal would also find the shortest path, however I have less experience writing breadth-first traversals, so I'll use the opportunity to practice a little bit.) We want the result of BFS to be the sequence of points to visit; in the example above, the first element of that sequence is $(2, 7)$, followed by $(2, 8)$, and the last element is $(1, 2)$.

To solve (2), we will use the sequence computed in (1) and basically follow it step-by-step. When we reach a node that has more than one neighbor (excluding the node we came from), we have a fork and Hermione will use her hand there.

3 Imports

We begin our program by importing a few data structures from Rust's standard library that will be useful for our solution:

VecDeque for a breadth-first traversal, we use a queue to keep track of the nodes that are yet to be visited.

HashSet a set will be used to hold the neighbors of a node in the graph.

HashMap a map will be used for two purposes: (1) to represent the graph (we use the adjacency list model), (2) to track child/parent relationships during BFS.

```
use std::collections::VecDeque;
use std::collections::HashSet;
use std::collections::HashMap;
```

Next, we import the `std::io` module which contains the `stdin()` function.

```
use std::io;
```

4 Graph

In this section, we will create the graph data structure and we'll implement the three functions of our solution.

4.1 Structure

The graph data type is defined below. It is a regular Rust struct (product type); it has three fields, the adjacency list (the `connections` field) and the start and end nodes.

A common practice in Rust is to derive the `Debug` trait on types; this allows us to print a graph on the console during development.

```
#[derive(Debug)]
struct Graph {
    start: (usize, usize),
    end: (usize, usize),
    connections: HashMap<(usize, usize), HashSet<(usize, usize)>>
}
```

4.2 Input

We now come to our first big function, `read_graph`. Because of its size, I will split it into "chunks": this is not strictly necessary, however it is one of those LP things that I want to experiment with and see if they do aid with comprehension. A chunk is a named snippet of code that we can refer

to; they are used primarily to split a long piece of code into understandable parts, and to present the parts in an order that may be easier or more natural to a human.

In the weaved document (which you are likely reading at the moment), a chunk appears in double angle brackets, e.g., `<<connect-nodes>>`, while in the tangled program they are expanded inline.

The function `read_graph` receives the dimensions of the scene, reads the description of the scene from stdin, and returns the corresponding graph. Because this is a simple program and because the inputs provided by the HackerRank system are all well-formed, we don't do extensive error handling. For example, we don't check that a start and end point exist, and if there are invalid characters in the input, we just panic.

```
fn read_graph(rows: usize, columns: usize) -> Graph {
    let stdin = io::stdin();
    let mut buf = String::new();
    let mut nodes = HashSet::with_capacity(rows * columns);
    let mut g = Graph {
        start: (0, 0),
        end: (0, 0),
        connections: HashMap::new(),
    };

    <<read-scene>>

    <<connect-nodes>>

    return g;
}
```

We start by initializing a few local variables:

- `stdin` and `buf` are used for reading from the terminal. `stdin` is a structure that implements the `Read` trait, which provides the `read_line` method. This method reads characters up to, and including, the newline character into `buf` and returns either the number of bytes that were read, or an error. We assume that no errors occur during input.
- `g` is the graph we want to return. We initialize the coordinates of the start and end nodes to `(0, 0)`. In a program where we cared more about robustness, we'd probably declare those fields as `Option<(usize, usize)>` and initialize them to `None`; however, as we mentioned, we don't expect input without a start or an end, and we therefore keep the types simpler.
- `nodes` is the set of all the nodes (floor tiles, start and portkey) in the scene. We use this extra local variables rather than storing the information directly in the graph's HashMap due to Rust's borrow restrictions (more on this later).

Once the local variables are initialized, we read the whole scene. We will explain in detail how this process works in the next sub-section. At the end of the `<<read-scene>>` chunk, the `start` and `end` fields of `g` will have been appropriately set. After `<<connect-nodes>>`, the adjacency list will be populated and we can return `g`.

4.2.1 Reading from stdin

Reading the graph from stdin is rather straight-forward, thanks to the simplicity and regularity of the format. We know that the scene is $N \times M$ (the variable `rows` corresponds to N, and `columns` corresponds to M) and only five characters are allowed (period, asterisk, 'M', 'X', newline).

After each line (which occurs after having read `columns` characters), we must manually empty the input buffer as `read_line` doesn't clear it, but appends to it. I've had bugs in his past programs due to this behavior! We also use the `let _ = ...` pattern to explictly ignore the result of the `read_line` method; if we didn't do this, the Rust compiler would warn us about an unused result which must be used.

We read a character by using the `nth` method of the `Chars` iterator; `nth` returns an `Option<char>` since the index could be out of bounds, but because we know that this won't happen in our program, we boldly call `unwrap` to extract the underlying character. The case analysis of the character is pretty straight-forward: if we see the characters '.', '*', or 'M', we add the pair `(row, col)` to the set of nodes; in the case of '*' and 'M' we also update the `start` and `end` variables, respectively. We completely ignore the character 'X'. If any other character is read, we panic with an error message; we cannot leave out this case, as the Rust compiler makes sure that our case analysis is total.

```
for row in 0..rows {
    buf.clear();
    let _ = stdin.read_line(&mut buf);
    for col in 0..columns {
        let c = buf.chars().nth(col).unwrap();
        match c {
            '.' => { nodes.insert((row, col)); }
            '*' => {
                nodes.insert((row, col));
                g.end = (row, col);
            }
            'M' => {
                nodes.insert((row, col));
                g.start = (row, col);
            }
            'X' => { }
            _   => {
                panic!("invalid character");
            }
        }
    }
}
```

4.2.2 Connecting the nodes

The set `nodes` now contains the coordinates of all the points that Hermione and Ron can walk on. The next logical step is to connect the neighbors together.

For each point (x, y) in `nodes`, we create a set of neighbors. We can then populate that set by verifying if the points $(x-1, y)$, $(x, y-1)$, $(x+1, y)$, and $(x, y+1)$ exist in `nodes` and adding them if they do. A note about Rust: it will panic in debug builds if an integer overflow occurs; it is therefore important to compile the program with `cargo build --release` or `rustc -O`. We could check that the arithmetic operations would not cause an overflow, or we could use the unchecked arithmetic methods (e.g. `unchecked_add`), but I am feeling a bit lazy.

Once the neighbors of (x, y) have been found, we insert the key-value pair $\langle (x, y), neighbors \rangle$ in the graph's adjacency list.

```
for &(x, y) in nodes.iter() {
    let mut neighbors = HashSet::new();

    if nodes.contains(&(x-1, y)) {
        neighbors.insert((x-1, y));
    }
    if nodes.contains(&(x, y-1)) {
        neighbors.insert((x, y-1));
    }
    if nodes.contains(&(x+1, y)) {
        neighbors.insert((x+1, y));
    }
    if nodes.contains(&(x, y+1)) {
        neighbors.insert((x, y+1));
    }

    g.connections.insert((x, y), neighbors);
}
```

We mentioned before that we used the `nodes` variable to side-step an issue with Rust's borrow checker. In the `<<connect-nodes>>` code, if we had iterated over `g.connections`, `g` would be borrowed, and we couldn't insert a new key-value pair at the end of the loop's body (the `insert` method would require a mutable borrow, which couldn't be obtained).

4.3 BFS

Now that we have our Graph data type, and that we have a function to create it from a textual description, it is time to create the traversal function.

(Note: in the preceding section, I split `read_graph` into three separate, logical chunks. I have decided not to split the `bfs` function in this section. I want to see how chunk vs. no-chunk affects clarity and comprehension; if you have comments about using chunks, do let me know.)

The `bfs` function has three logical phases.

Initialization

In this phase, we create two data structures: a queue that contains the nodes to visit (a BFS traversal uses a FIFO structure to determine the next node to visit while a DFS traversal uses a LIFO structure), and a map to keep track of the nodes visited so far and their parent. A parent is represented with an Option type. The start node has no parent and so we represent its lack of a parent in the map with the value `None`. The parent of the other nodes is `Some((x, y))`. We begin the procedure, by putting the start node in the queue, and marking it as visited.

Traversal

Our next step is to traverse the whole graph. We iterate until the queue of nodes left to process is empty. We pop the oldest node from the queue and look at its neighbors. If a neighbor has already been visited, we skip it; if a neighbor has yet to be visited, we add it to the queue and record its parent in the map. We could terminate the traversal early by breaking from the loop when we hit the end node, but to keep the code shorter, we have omitted this small optimization.

Path finding

Once the `parent` map has been populated, finding the path from beginning to end is simple. Starting with the end node, we use the map to find its parent, and then find that parent's parent, and so on, until we reach the start node. All the while, we push the coordinates of the nodes inside a vector. After the loop, the vector contains the coordinates, but from end to beginning; we reverse the path, and return it.

```
fn bfs(g: &Graph) -> Vec<(usize, usize)> {
    let mut q = VecDeque::new();
    let mut parent: HashMap<(usize, usize), Option<(usize, usize)>> = HashMap::new();

    q.push_back(g.start);
    parent.insert(g.start, None);

    while !q.is_empty() {
        let curr = q.pop_front().unwrap();

        for neighbor in g.connections.get(&curr).unwrap() {
            if !parent.contains_key(neighbor) {
                parent.insert(*neighbor, Some(curr));
                q.push_back(*neighbor);
            }
        }
    }

    let mut path = Vec::new();
    let mut curr = Some(g.end);
    while let Some(x) = curr {
        path.push(x);
```

```
        curr = *(parent.get(&x).unwrap());
    }
    path.reverse();
    return path;
}
```

4.4 Zapping the wand

We finally get to the last part of the problem, counting how many times Hermione needs to zap her wand for her and Ron to find their way to the portkey. Hermione will use her wand when she and Ron are on a node that has two or more *unvisited* neighbors. When they are standing on the start node, that means two or more neighbors, and when they are standing on other nodes, that means three or more neighbors (we do not count the node we came from.) Additionally, if Hermione and Ron are standing on the portkey node, she won't need to zap her wand there, even if there is a fork, because they have reached their destination.

```
fn count_zaps(g: &Graph, path: &Vec<(usize, usize)>) -> usize {
    let mut zaps = 0;
    for p in path {
        let neighbors = g.connections.get(p).unwrap();
        if *p == g.start && neighbors.len() > 1 {
            zaps += 1;
        } else if *p != g.end && neighbors.len() > 2 {
            zaps += 1;
        }
    }
    return zaps;
}
```

5 Main function

We can now implement the main function and finish our program. The first thing I should mention is that up until now, I have failed to mention a small detail about the problem: Ron tries to predict how many times Hermione will need to zap her wand. If he gets it right, the program should display "Impressed", and if he gets it wrong, it should display "Oops!". This is not hard to handle: we'll simply compare his prediction with the return value of `count_zaps`.

Before we get into the implementation of the main function, let's talk about the input format. The first line of input is an integer that specifies how many problem instances we have to solve. Each problem instance has three inputs:

- The width and height of the scene
- The scene itself
- Ron's prediction

(Doing terse I/O is definitely not Rust's strong suit, as will be apparent from the upcoming source block.)

```
fn main() {
    let stdin = io::stdin();
    let mut buf = String::new();

    // Read number of instances
    let _ = stdin.read_line(&mut buf);
    let iters: usize = buf.trim().parse().unwrap();

    // Read each instance and solve it
    for _ in 0 .. iters {
        buf.clear();
        let _ = stdin.read_line(&mut buf);
        let (width, height) = {
            let mut dims = buf.trim().split_whitespace();
            let width: usize = dims.next().unwrap().parse().unwrap();
            let height: usize = dims.next().unwrap().parse().unwrap();
            (width, height)
        };
        let g = read_graph(width, height);
        buf.clear();
        let _ = stdin.read_line(&mut buf);
        let prediction: usize = buf.trim().parse().unwrap();

        let path = bfs(&g);
        if prediction == count_zaps(&g, &path) {
            println!("Impressed");
        } else {
            println!("Oops!");
        }
    }
}
```

6 Running the program

Let us test our program with the following example input from HackerRank.

```
3
2 3
*.M
.X.
1
4 11
```

```
.X.X......X
.X*.X.XXX.X
.XX.X.XM...
......XXXX.
3
4 11
.X.X......X
.X*.X.XXX.X
.XX.X.XM...
......XXXX.
4
```

Running the program gives the following output:

```
$ ./target/release/count-luck <example.txt
Impressed
Impressed
Oops!
```

Which is the expected output! In fact, submitting the tangled source code to HackerRank tells us that our program produces the correct answers for all their test cases.

Mission accomplished!

7 Epilogue

The goal of this program was to try literate programming, and experience first-hand the pros and cons of this programming methodology.

The first thing I should mention is that this has been one of the hardest program I've written in my life, and not because the problem itself was very difficult. If I only had to write the code, I would've likely finished in an afternoon. Instead, it took me three evenings to complete the whole program. It was the "literate" part that was very hard for me. It wasn't always clear what details were relevant, and which weren't and so a lot of text was written, then discarded. I was also rarely satisfied with my first drafts, and most of the sentences in this document have been re-written multiple times. I estimate that the ratio of time spent on text vs. code was around 4:1, maybe even 5:1. Surely, as one gains experience with literate programming, that ratio improves.

I don't know exactly how other programmers approach literate programming, but in my case, it was mostly this:

1. Get a general feeling for the problem and potential solutions by scribbling with pencil and paper;
2. Write out in org-mode the very general idea (mostly as a reminder);
3. Write the Rust code;

4. Comment the Rust code by adding the prose, making sure to mention the parts that are tricky or not obvious.

I suspect that, ideally, steps (3) and (4) should be reversed: I wrote the code to guide my prose, rather than using the text as a thought and design tool, which then informs the code.

7.1 Pros

Last summer, a number of people from my lab at McGill were going on interviews. To practice for those dreaded code challenges, we took problems from sites such as CareerCup and worked as a team to solve them, making sure to act as if we were in front of an interviewer. I remember those moments very fondly: in addition to the wonderful camaraderie of those sessions in front of a blackboard, it was an amazing opportunity to develop problem solving skills, and also *solution explaining* skills. Everyone who participated were on top of their game and presented their ideas in a clear and logical fashion. We'd sketch out example problems, see how we'd solve them, and try and find the corner cases that were lurking. We'd ask questions about the choice of proper data structures and algorithms, what assumptions we made about the problem, what invariants our solution needed to provide. We'd ask questions about running time, memory usage, we'd poke at details that seemed a little too hand-wavy. It was a very good way to improve as a programmer, and a fantastic way to spend quality time with excellent friends.

I feel that literate programming captures this spirit very well. I have tried my best to write down in the text the ideas that I would want to convey to those lab mates. One advantage of LP over doing it over at the board is that I have more time to think about what I want to say and how I want to say it.

Another obvious advantage of literate programming is how approachable it can make a new code base. We are often thrown into a foreign project, and asked to understand it, which is not always easy if the structure of the program is not documented or if the program is complex. In that context, this core idea of LP that you can open a program at page one, start reading, and begin to understand the program is a powerful one.

7.2 Cons

I have also found a number of ways by which literate programming was not as good as traditional programming.

Tool support

One of the biggest difficulty during the development of CountLuck was the disconnect between the literate programming document (the org file) and the different programming tools that I'm used to.

- I often forgot to tangle my document after making a change to the code.
- When the Rust compiler reports errors, it (of course) gives the line numbers in the tangled source file, not the org document. It's harder and longer to locate the cause of an error.

- Useful Emacs features such as `find-tag` or `imenu` that I use liberally to navigate in a project don't work.

I think one of the biggest challenges that literate programming faces if it is to gain wider-spread acceptance is to offer a set of tools that make using LP as convenient as regular programming. (I realize that I have only used one tool, org mode, and haven't tried *noweb* or *CWEB*; maybe they have solutions to those issues.)

Chunks

I don't know how I feel about chunks. On the one hand, they are great at splitting code into logical parts, and even allowing those parts to come in the order that makes the most sense for the reader. However, I feel that if I use chunks too much, I lose the mental picture of what the function is supposed to look like. Also, when should one use chunks vs. creating a new helper function? I would love to hear arguments for or against chunks.

Project organization

I am not sure how a literate programming project should be organized. Where should the literate program be put? In the `src` directory or somewhere else? Should the tangled source files be included in version control or should they be considered build artifacts? Should a literate program be a single file (`tex.web` is a single 25,000 line file) or is splitting encouraged? How does that affect the ability of a team to work together?

Language barrier

I am a native French speaker, and although I don't have difficulty reading and writing English, it seems that literate programming could make is harder for people who don't speak English fluently to contribute to a project if they are expected to clearly document the code that they write.

7.3 Editing style

One aspect of this literate program that is not apparent from the weaved document or the tangled program is how I've formatted the text inside the org file.

A few weeks ago, I read an old blog post by John Regehr where he suggests making the structure of sentences apparent by using newlines generously. All my sentences are on their own line (or multiple lines if they are long and the right margin) and are separated by a blank line (I use a # to make an org comment to prevent a paragraph break). This definitely makes the source text a little weird to look at, however sentences are a lot easier to move around, I can easily see how long a sentence is, and it's easier to notice if the same syntactic structure is used or not. In addition, this is friendlier for version control systems like Git since adding or removing a fewof words from a sentence only affects a line or two. If the text was re-filled, the entire paragraph might be in the diff block, thus making the changes harder to spot.

I have found that the hardest part about this style is not writing the text, but rather to avoid the temptation to hit `M-q` as soon as I hit the period key. (In Emacs, `M-q` invokes the `fill-paragraph` command.)