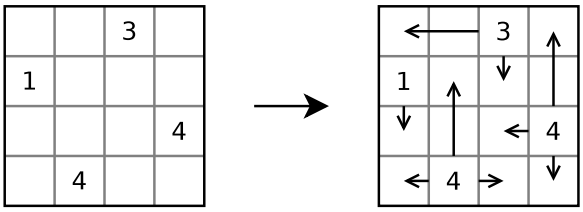
Lab 8
Lijnenspel

27 October 2015

The drill this week has two semi-separate parts: one to make sure you understand the surface task (a puzzle game called Lijnenspel), and the other part to get you started on using the `Matrix` stuff we've been talking about in class. You can do the drill bits (ha!) in either order.

Drill 1: Introducing Lijnenspel

Lijnenspel[1] is a puzzle played on a grid (similar to Sudoku or a crossword puzzle). In Lijnenspel, the initial grid contains numbers in some of the squares (as on the left below), and the puzzle solver's job is to draw horizontal and vertical arrows extending out from the numbers to fill the rest of the grid. The total length in squares of all the arrows emanating from a numbered square should add up to that number, as in the solved example on the right:[2]

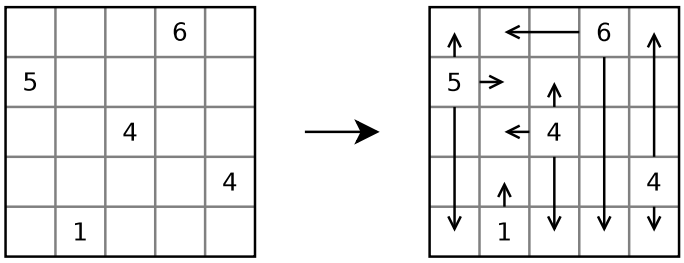


To be a proper Lijnenspel puzzle, the solution should be unique; and indeed any such puzzle with a unique solution is possible to solve without guessing (though the logic may require a bit of work). There are various tactics that can be applied. For instance, if a particular square is only "reachable" from one numbered square, there has to be an arrow connecting them; in the following example, the second square in the top row and the fourth square in the bottom row are *only* reachable from the 6—so we could immediately "spend" six to draw arrows from the 6 to those two squares. That makes

[1] Also known as "Line Game", but the authors of the site I pulled examples from are Dutch, and so publish it under both names. "Lijnenspel" looks cooler, no? It's pronounced "LINE-en-spell".

[2] This example comes from the description page at `http://www.puzzlepicnic.com/genre?lijnenspel` .

the right column unreachable except from the 4; and this sort of logic can continue through to complete the puzzle.[3]



To confirm you understand how the Lijnenspel puzzles work, work out at least the first two on the handout I gave out in class. Compare notes with other students!

Drill 2: Matrix

This part is more like the drills you've done in the past.

1. Start a program (`.cpp` file) that `#include`s the two `Matrix` headers we learned about this week, and also the `using namespace` line. See the code from `/home/shared/160-3/1026/use_matrix.cpp` if you don't remember how! For now, have it create a 4×4, 2D matrix of `int`, and print that out to the screen. Make sure this much compiles. (As *always*, try to compile as often as is reasonable, and start running your code as soon as it's feasible to do so.)

2. Before creating the matrix, read a single positive integer from the user; modify the matrix creation so that it's a square of that size. (For instance, if the user typed 3, the matrix would be a 3×3 grid.)

3. Change the contents of the matrix from `int` to `char`, add code to read in the grid from `cin` after reading the intended size of the grid, and build two test files with input appropriate to this format (a size, and then a curly-bracket-delimited matrix of characters).

4. Edit your examples to correspond to either completed or non-completed Lijnenspel puzzles. In a starting-position puzzle, each char will be ei-

[3] This puzzle by Zack Butler of RIT, who also provided the inspiration for this lab.

ther a digit from 1 to 9, or a period '`.`' for an open square. In a completed puzzle, all the periods will have been replaced with one of '`<`', '`>`', '`^`', or '`v`', depending on which direction their arrow was going. (Displaying and storing a length-3 arrow as "`>>>`" instead of "`-->`" will make our life easier later, but will still be easy to interpret visually.)

5. Write a function `count_numsquares` that takes a 2D grid (that is, a `Matrix<char,2>`) and counts and returns how many of the squares in the given grid are number squares. Two notes: first, to check if a character is a digit, you'll compare it to `'0'` (or `'1'`) and `'9'`—note the single quotes. Second, in this function you'll need one `for` loop to go through the rows, and another `for` loop *inside* it to go through each element in each row. In your `main` function, after you've printed the grid itself, print how many of its squares are number squares (i.e. the result of this function).

6. Write a second function `sum_numsquares` that takes a 2D grid and computes the total of all the number squares in the given grid. (So, for the Lijnenspel on the front page of this handout, it should return 12: 1+3+4+4.) Note that since we store a number square as a character rather than a number, you'll have to adjust it before doing math with it—if you remember in Lab 5 that `'a' + 3` yielded `'d'`, you might not be surprised to find that `'3' - '0'` yields the actual int value 3. Make use of this fact when computing your sum! Then, in your `main` function, also print the result of this function.

Vim FOTD: "ex" mode

Open two terminal windows, and arrange them side-by-side. The one on the left I will designate the "edit" window, and the one on the right the "other" window. In the edit window, edit a file named `dummy.txt`, type a few lines into it, and save and quit. (The content doesn't matter at all.)

In the other window, `cat` the file, that is, type

```
cat dummy.txt
```

You should see the exact contents you just typed in. If not, make sure both windows are in the same directory (your home directory is fine) and try again.

Return to the edit window, again edit the `dummy.txt` file, and add a couple more lines. This time, don't save and quit. Back in the other window, `cat` the file; since you haven't saved yet, what should you see?

Return to the edit window again. Make sure that you are in command mode (hit escape), and this time, instead of save-and-quit (with `:wq`), just save: if you type `:w` by itself and hit enter, this writes the file without quitting. Now in the other window, `cat` the file again—you should now see the updated version.

What you are seeing here is (more of) a third mode of Vim besides insert and command mode, called "Ex mode". Ex was an editor back in the day, a precursor to vi (which was itself the basis for vim). All interactions with the ex editor were done through commands typed on a line that started with a colon, and vestiges of this survive in the modern Vim editor; from ex mode you control file access operations, among other things. The `:w` command that you have just seen simply saves ("writes") the current file.

Another command is to save to a different file. After you issued the `:w` command a moment ago you were returned to command mode, so press the colon key again to return to ex mode and type

```
w dummy2.txt
```

This creates a brand new file, named `dummy2.txt`, with the current contents of the edit buffer. But unlike "Save As..." in a typical modern word processor, it doesn't change the default name of the file, so that if you type `:w` again, it will save it under the original name (`dummy.txt`).

You can use ex mode to edit a different file. Type

```
:e newfile.txt
```

and the `dummy.txt` file will disappear from the window, to be replaced by an empty buffer. If you type text in here and then hit `:w`, it will be saved under the name `newfile.txt` (as you can verify by using `ls` and `cat` to look at the file from the other window).

Ex mode can be used to quit Vim by typing `:q` and hitting enter. Note that this does *not* include writing out the file first. If you type `:q` before saving (and you can tell that the file has been edited by the "`[+]`" after the filename in the status bar), Vim will warn you that you haven't saved the file. You can then type

```
:q!
```

to say, no really, I mean it, just quit (don't save).

There are other ex mode commands that we'll learn eventually, but these file-related commands enable a particularly useful interaction style: you can now leave your source code open in one window, save it, and run the compiler in the other window. This is useful so that you can keep any compiler errors on the screen while you scan the code for the problem; in fact, from now on you should get in the habit of having at least two windows open when you're programming: one for editing, and one for compiling and testing. (Some of you have already been doing this, but it's even more streamlined now.)

Back to the task: more Lijnenspel

In the latter part of the drills, you wrote some functions that technically don't rely on the fact that the grids are meant to be Lijnenspel puzzles—just that some of the grid squares would have number characters in them. For the rest of the lab, we'll actually encode some of the logic of the puzzles themselves. While we won't quite write a solver ourselves, we will do a few things that would help out a human trying to solve them.

Specifically, you'll write functions to validate the puzzles themselves; to check if there are any open squares; to verify whether a particular number-square is "completed" (all its arrows drawn); and finally to check if an entire Lijnenspel puzzle is completed. You'll be able to use the two puzzles given above and the two you worked on as your test cases.

Remember to make use of functions you've already written!

valid_puzzle should take a grid and determine whether the given grid might be a valid Lijnenspel puzzle: in any valid puzzle, the total number of number-squares, plus the sum of all those numbers, should add up to the number of squares in the grid. If not, it's not even a valid puzzle. (Make sure to have an example that *isn't* valid when you test this!)

has_open should take a grid and determine whether any of the given grid's squares are open—represented by a period (rather than a number or arrow).

count_arrows_east should take a grid and a row and column index for a number-square at that position, and it should count how many `'>'` characters in a row appear immediately to the east (right) of the given number-square. Note that you have to be careful not to "fall off the edge" of the grid if the arrows go all the way to the rightmost column.

count_arrows_west, count_arrows_north, count_arrows_south should be pretty much like `count_arrows_east` but modified to look for their respective arrows in their respective directions.

is_numsquare_completed should take a grid and a row and column index for a specific number-square in that grid; and should determine whether the number-square at that position is "completed". A number-square is completed if the count of the arrows in all four directions adds up to the number in the square—neither too many (invalid) nor too few (incomplete).

is_puzzle_completed should take a grid and determine whether the given grid is fully and correctly filled in. To do so, it must be a valid puzzle, with no open squares, and for which every number-square is completed.

Handing in

As usual, use the `handin` program. Designate this as `lab8`. Hand it in by 4pm on Monday, 2 November.

RUBRIC (tentative)

General

1 Attendance at lab with drill done or question written down

½ Readme, evidence of testing

Drill

1 1–3: Reads and writes a 2D matrix of `char` with user-spec size

1 4: examples of Lijnenspel, in spec'ed format, in `.in` files

1 5: `count_numsquares`

1 6: `sum_numsquares`

Lijnenspel

½ `valid_puzzle` uses other functions & works

1 `has_open`

1 `count_arrows_east`

½ ...and west, north, south

½ ...and `is_numsquare_completed`

1 `is_puzzle_completed`