

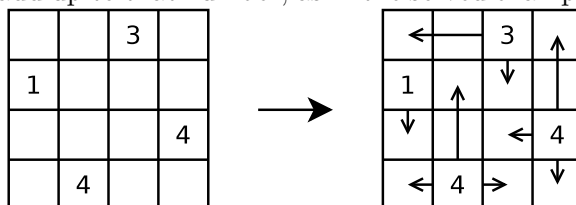
Lab 12 Part 1

Lijnenspel

19 November 2024

Setup 1: Introducing Lijnenspel

Lijnenspel¹ 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

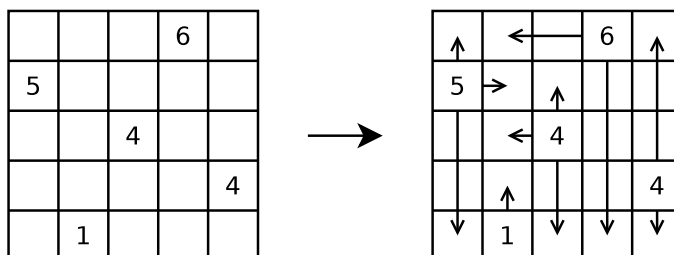


To be a proper Lijnenspel puzzle, the solution should be unique; and indeed any such puzzle with a unique solution is possible to solve without guessing (though the logic may require a bit of work). There are various tactics that can be applied. For instance, if a particular square is only “reachable” 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 the right column unreachable except from the 4; and this sort of logic can continue through to complete the puzzle.³

¹Also 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”.

²This example comes from the description page at <http://www.puzzlepnic.com/genre?lijnenspel>.

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



To confirm you understand how the Lijnenspel puzzles work, work out at least the first two on the lab worksheet linked from the course page. Bring them with you to lab. Compare notes with other students!

Setup 2: 2D data

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

1. Start a program (.cpp file) with a `main` function that reads a single `int` from the user. This will be the size of the Lijnenspel grid (which is square). Make sure this much compiles. (As *always*, but especially this week, try to compile as often as is reasonable, and start running your code as soon as it's feasible to do so.)
2. Create a 2D vector value that will store the grid. We will represent individual cells in the grid with character values, so this will be a vector of vectors of `char`. Note that it is initially empty (and you can explicitly say so!).
3. Write loops to read in $n \times n$ characters from `cin` and add them to your grid appropriately. You will almost certainly want this as a nested pair of loops, the outer one working with the whole row and the inner one going character-by-character to build the row.
4. To help you debug and understand your code, write a function to take a given grid and print it to `cout`. This is a rare (for me) example of a function where you actually do want to use `cout` inside the function. In your `main`, go ahead and call this function to print out the board that has been read in from the user.
5. Write example files that correspond to either completed or non-completed Lijnenspel puzzles. In a starting-position puzzle, each char will be either 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.)

6. Write a function `count_numsquares` that takes a given 2D grid (that is, a `vector<vector<char>>`) and counts and returns how many of the squares in the given grid are number squares. Two notes: first, remember that there is a builtin function to determine whether a character is a digit. 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).
7. (How might you test that function?)

The task: more lijnenspel

In the second part of the setup, 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 the basis for your test cases.

Remember to make use of functions you've already written! In some cases you will need to do so in order to get full credit.

`sum_numsquares` computes the total of all the number squares in a 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 that `'a' + 3` yields `'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!

valid_puzzle determines whether a given grid *might* be a valid Lijnenspel puzzle, according to the following rule: 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 determines whether any of a given grid's squares are open—that is, represented by a period (rather than a number or arrow).

count_arrows_east considers a given grid starting at a given row and given column position; and if that position is a number-square, it counts how many `'>'` characters *in a row* appear *immediately* to the east (right) of that position. (If the position is not a number-square, it just returns zero.)

This function is simple in the basics but has several edge cases you need to think about:

- The position may not be a number-square.
- The square to the immediate east of the position may not have a `'>'` character in it.
- There might not *be* a square to the immediate east of the position.
- The arrows may continue all the way to the edge of the grid, and your program should neither hang nor crash in this case.

If you are not sure what would be an appropriate header for this function, you can use this one:

```
int count_arrows_east (vector<vector<char>> grid, int row, int column);
```

(There are other possible correct headers; see me if you'd like to talk about them.)

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 also consider a given grid at a given row and given column position; 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 `lab12`. Hand it in by 4pm on Monday, 24 November.

RUBRIC (tentative)

General

- 1 Attendance at lab with preview done or question written down
- 1/2 Readme, evidence of testing

Setup

- 1 1–4: Reads and writes a 2D grid of `char` with user-spec size
- 1 5: examples of Lijnenspel, in spec’ed format, in `.in` files
- 1 4/6: Function takes 2D grid parameter, iterates over it
- 1/2 6: `count_numsquares` correct

Functions

- 1/2 `sum_numsquares`
- 1/2 `valid_puzzle` uses other functions & works
- 1 `has_open`
- 1 `count_arrows_east`
 - 1/2 Starts at square, counts east, stops if end of grid
 - 1/2 Returns correct answer
- 1/2 ...and west, north, south
- 1/2 `is_numsquare_completed` uses other functions & works
- 1 `is_puzzle_completed` uses other functions & works

AI policy and frequent submission

(no substantive change from the Lab 3 version of this policy)

Some use of generative AI is fine, but you a) should not paste this assignment or type it verbatim into the AI prompt, and b) should not be asking the AI for the whole program all at once. (Just like you can ask for help from a human, but should not have them write the whole program for you!) If you get help from an AI chat OR from a person, you should note that in a program comment near whatever you got from them.

Relatedly, I expect that you'll run the `handin` program relatively often, and I *require* that you do so at least 2–3 times over the course of working on the lab. As a rule of thumb, hand it in after completing each 1–2 points on the rubric. Submissions that jump straight to a final, (near-)correct version with no intervening submissions along the way *will receive little or no credit* for that part.

This is a new policy that I'm experimenting with; let me know if you have any feedback.

This document was written and prepared without the use of generative AI.