CSCI-1200 Data Structures — Fall 2018
Homework 6 — Inverse Slitherlink

In this homework we will solve and generate Slitherlink puzzles, a grid-based fence enclosure game developed by Nikoli. You can read more about the game and play online versions here:

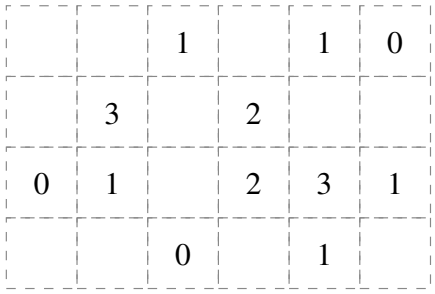
https://en.wikipedia.org/wiki/Slitherlink
https://www.puzzle-loop.com/
https://krazydad.com/tablet/slitherlink/
https://www.conceptispuzzles.com/index.aspx?uri=puzzle/slitherlink/techniques

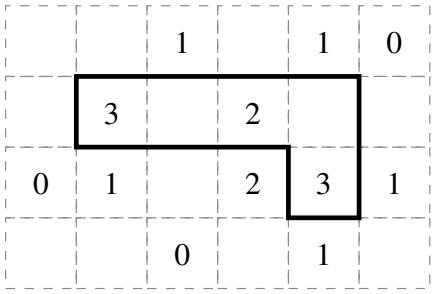
You may not search for, study, or use any code related to existing solvers for this puzzle game. *Please carefully read the entire assignment and study the examples before beginning your implementation.*

Slitherlink - How to Play

Slitherlink is a game played on a rectangular grid with w x h cells. Each cell of the grid is labeled with a number, 0, 1, 2, 3, or 4, or the cell may be unlabeled. A solution to a Slitherlink board is a single *fence* or closed loop of edges between the cells such that the label of every cell matches the number of edges that touch the cell. Unlabeled cells can have any number of edges touching them. In our simplified version of the game we will allow solutions that have multiple disconnected closed loops of edges. Here is a small example:

```
..1.10
.3.2..
01.231
..0.1.
```






```
......
.----.
.---..
....-.
......

.......
.|...|.
....||.
.......
```

Command Line Arguments

Your program will accept one or more command line arguments. The first argument is the name of a puzzle board file similar to the file on the upper left. We use '.' to represent an unlabeled cell in the input. Your task is to compute a valid closed loop path as illustrated in the middle diagram above.

By default, your program should print to `std::cout` a single ASCII art solution (if one exists). Each solution should begin with the string "`Solution:`". See the provided code and sample output. If the optional command line argument `--all_solutions` is specified, your program should output all valid solutions and also print a message with the total number of solutions.

By default, your program should allow solutions that consist of multiple disconnected closed loops. If the optional command line argument `--loop_analysis` is specified, solutions that consist of multiple loops will instead be labeled "`Multi-Loop Solution:`" and the message at the bottom will separately list the count of single vs multi-loop solutions. Finally, if the optional command line argument `--single_loop` is specified the multi-loop solutions will not be printed or counted.

Alternately, if the command line argument `--inverse` is specified, the input file will instead encode a representation of the desired path, as show above on the right. The inverse input first encodes the horizontal portions of the path with a w x $h+1$ grid of characters, where '`-`' indicates the edge is part of the path and '`.`' indicates the edge is not part of the path. After a blank line the file continues with the vertical portions of the path as a $w+1$ x h grid of characters where '`|`' is a vertical edge. For the inverse Slitherlink input

your job is to output an *interesting* puzzle board with a single unique solution that matches the specified fence pattern. Simply outputting the complete grid with all cells labeled with their edge count may be a valid Slitherlink puzzle, but it is not very *interesting*. Why not? A grid with more labeled cells is usually easier for a human (and presumably a computer) to solve. We will say that a puzzle with fewer labeled cells is more *interesting*. For a given fence pattern, there are possibly (probably?) many different input Slitherlink boards that are equally (or near equally) *interesting*. Your task is to output one of them. We will award partial credit if you produce a good puzzle, but it contains more labeled cells than the optimal *interesting* solution for that puzzle.

Output Formatting

We provide starter code to read and print both the regular input file and the inverse input files. You may use or modify any or all of this provided code.

Your program will print the solution(s) to `std::cout`. Study the provided sample output files carefully. If there are multiple solutions and you are not asked to find all of them you may output any solution for full credit. If you are asked to output all solutions, they may be printed in any order for full credit. You may also find the `--no_ascii_art` command line option helpful for testing the output of your inverse puzzle generator with your puzzle solver and vice versa.

Additional Requirements: Recursion & Big O Notation

We suggest you *DO NOT* spend too much time implementing the special patterns or rules for Slitherlink that you will find online that help humans solve Slitherlink puzzles. Implementing and debugging those patterns will take a lot of time. Instead focus on implementing a recursive method to *guess* at the placement of a segment of fence path, and then check if a valid solution can be built from the remaining board. The TAs will be grading the elegance of your recursive solution. Once you have a working solution you can consider adding additional rules or patterns to improve the performance of your program.

You must use recursion in a non-trivial way in your solution for both the normal Slitherlink puzzle solver and also the inverse Slitherlink puzzle generator. As always, we recommend you work on this program in logical steps. Partial credit will be awarded for each component of the assignment. *IMPORTANT NOTE: This problem is computationally expensive, even for medium-sized puzzles! Be sure to create your own simple test cases as you debug your program.*

Once you have finished your implementation, analyze the performance of your solver and generator algorithms using big O notation. What important variables control the complexity of a particular problem? The dimensions of the board (w and h)? The number of labeled cells in the puzzle (n)? The number of edges in closed loop fence path (e)? In your `README.txt` file write a concise paragraph (< 200 words) justifying your answer. Include a table summarizing the running time and number of solutions found by your program on each of the provided examples. *Note: It's ok if your program can't solve the biggest puzzles in a reasonable amount of time.*

You must do this assignment on your own, as described in the "Collaboration Policy & Academic Integrity" handout. If you did discuss this assignment, problem solving techniques, or error messages, etc. with anyone, please list their names in your `README.txt` file.

Homework 6 Inverse Slitherlink Contest Rules

- All students are required to submit their program to the contest. Extra credit will be awarded for programs that have a strong performance in the contest.
- Contest submissions are a separate homework submission. Contest submissions are due Tuesday, October 30th at 11:59pm. You may not use late days for the contest. (The regular homework deadline

is Thursday, Oct 25th at 11:59pm and late days are allowed for the regular homework submissions.)

- You may submit the same code for both the regular homework submission and the contest. Or you may make a small or significant change for the contest.
- Contest submissions *do not* need to use recursion.
- We will compile your code with optimizations enabled (`g++ -O3 *.cpp`) and run all submitted entries on the homework server.
- Programs must be single-threaded and single-process.
- We will run your program by *redirecting* `std::cout` to a file and measure performance with the UNIX `time` command. For example:

```
time slitherlink.out puzzle4.txt --all_solutions > out_puzzle4_all.txt
```

- You may want to use a *C++ code profiler* to measure the efficiency of your program and identify the portions of your code that consume most of the running time. A profiler can confirm your suspicions about what is slow, uncover unexpected problems, and focus your optimization efforts on the most inefficient portions of the code.
- We will be testing with and without the optional command line arguments `--all_solutions`, `--loop_analysis` `--single_loop`, and `--inverse` options and will highlight the most correct and the fastest programs.
- You may submit a new Slitherlink puzzle board file and/or a new inverse Slitherlink path file for possible inclusion in the contest. Name these tests `puzzle_smithj.txt` and `inverse_smithj.txt` (where `smithj` is your RCS username). Extra credit will be awarded for interesting test cases that are used in the contest. Caution: Don't make the test cases so difficult that your program cannot solve them in a reasonable amount of time!
- In your `README_contest.txt` file, describe the optimizations you implemented for the contest, describe your new test cases, and summarize the performance of your program on all test cases.