CITS1001 Project 2, Semester 1 2019: Slither Link

Version 1.20. (Added to Hints & Tips.)
Check the LMS to ensure that you have the latest version.
If you have any questions about any aspect of the project, submit them to *help1001*.

Project Rules

- Submission deadline: **5pm Friday 24 May 2019**
- Submit via ***cssubmit***
- Value: **15% of CITS1001**
- Project work to be done **in pairs** or **individually**; groups of more than two are ***not allowed***
- **Make sure that all students involved are clearly identified in all of your classes**
- Project published: 5 May 2019

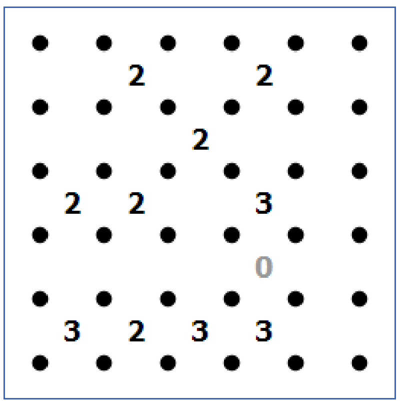
The project task is to construct a Java program containing your solution to the following problem. You must submit your program via *cssubmit*. No other method of submission is allowed.

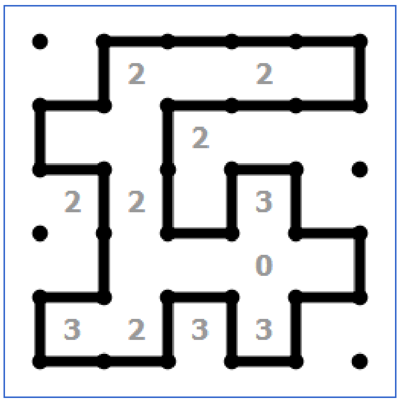
You are expected to have read and understood the UWA Policy on Academic Conduct. In accordance with this policy, you may discuss with other students the general principles required to understand this project, but the work you submit must be the result of your own effort.

You must submit your project before the submission deadline above. UWA's Policy on Late Submission will apply.

Project Overview

Slither Link is a modern puzzle game developed by Nikoli. The player is required to find a path around a grid of dots while satisfying various constraints imposed by the puzzle. A typical starting position for a small instance of Slither Link is shown in Fig. 1(a), along with its solution in Fig. 1(b). (Fig. 1(a) corresponds to the file *eg5_2.txt* in the project folder, discussed below.)





Figures 1(a) and 1(b).

A solution to a Slither Link puzzle satisfies two requirements.

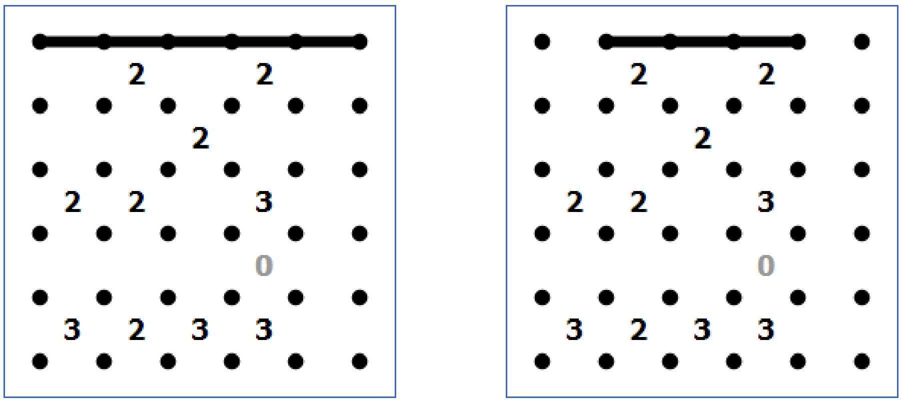
- All numbers in the puzzle have the right number of line segments surrounding them.
- All line segments combine to form a single closed loop.

You can play Slither Link online, and a free app is available from the usual sources. There is also a Wikipedia page dedicated to Slither Link, plus many other pages. We will use a somewhat restricted form of the game; make sure you understand the restrictions and stick to them.

In this project you will write an implementation of Slither Link to allow someone to play the game.

The Game of Slither Link

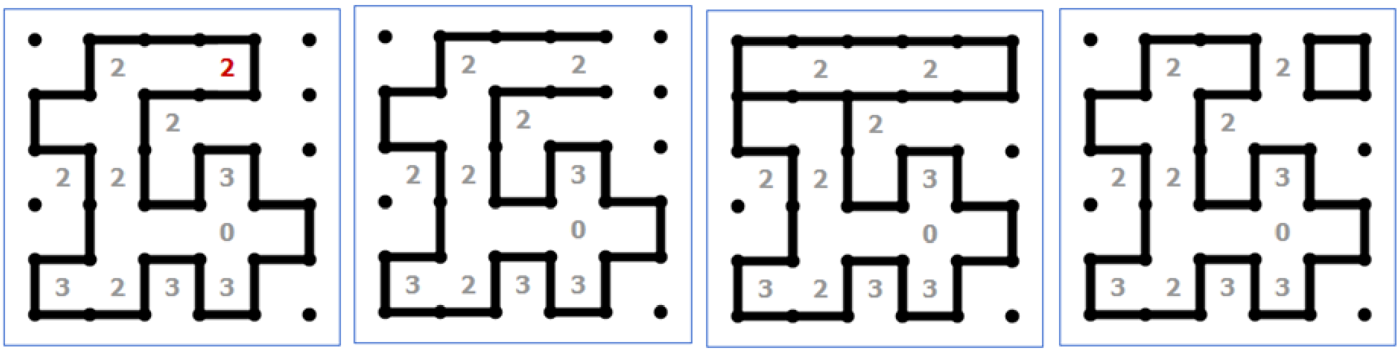
During a game of Slither Link, the player has only one action available: to click the mouse between two dots in the grid, causing a line segment between those two dots to toggle on and off. For example given the starting position shown in Fig. 1(a), if the player performs five clicks, one between each adjacent pair of dots in the top row, the position will be as shown in Fig. 2(a). If they then click between the two leftmost dots and again between the two rightmost dots, those line segments are removed and the position will be as shown in Fig. 2(b). Diagonal line segments are not allowed. Try it out at puzzle-loop.com.



Figures 2(a) and 2(b).

The player's goal is to generate a set of line segments that satisfy two requirements: collectively they must form a single closed loop, with no extraneous segments; and every number in the original puzzle must be surrounded by the correct number of segments. Squares that have no number can have any number of line segments surrounding them. The (unique) solution to the example puzzle is shown in Fig. 1(b).

At any stage of the game, the player can ask the system whether the current situation constitutes a solution to the puzzle. The system will respond either in the positive, or with a message reporting one of a set of possible failure modes. Figs. 3(a)-3(d) enumerate the ways in which a situation can *fail* to be a solution to a puzzle.



Figures 3(a), 3(b), 3(c), and 3(d).

- 3(a) fails because one of the numbers (the red 2) has the *wrong number* of line segments.
- 3(b) fails because the line has *dangling ends*.
- 3(c) fails because the line has *branches*.
- 3(d) fails because it has *disconnected loops*.

Obviously it is possible for a puzzle situation to have multiple failure modes, even all four at once! The system needs to check the current situation on demand, and to report either success or an appropriate error message.

Slither Link puzzles come in various sizes, and even with differently-shaped cells, e.g. triangles or hexagons. In this project we will stick to square puzzles of the type shown here, with size at least 2x2.

Testing for a Solution

Testing to see whether the current board situation constitutes a solution to a puzzle is quite tricky. We need to check for all four failure modes and to return an error message if any of them applies. This is basically what the class `AnalyzeSolution` (introduced below) does, with several helper methods as described in the code skeleton.

Use the following algorithm to test a solution. Remember that the puzzle is finished iff all squares are individually correct, and all line segments form a single closed loop.

1. Check the number of line segments around each square. If any of them are wrong, return *Wrong number*.
2. Find any line segment on the board, and record the dot where it is located. Trace that line around the board, counting the number of steps as you go. If at some point the line ends, return *Dangling end*. If at some point the line branches, return *Branching line*. Otherwise eventually the line must go back to the starting dot, and thus it forms a closed loop. Let the number of line segments in the loop be `n`.
3. Count the total number of line segments on the board; let this number be `t`. If `t > n`, the loop does not contain all of the line segments on the board; return *Disconnected lines*.
4. Otherwise `t = n`, i.e. all of the line segments are in the loop, and the puzzle is finished! return *Finished*.

The methods in `AnalyzeSolution` collectively implement this algorithm. Tackle them in the order suggested.

Coordinate System

To model a game of this type, we need a 2D coordinate system to refer to positions on the board. Given an *nxn* puzzle, note the following.

- Squares: the board will have *n* squares across and down.
- Dots: the board will have *n+1* dots across and down.
- Horizontal lines: the board can have *n* horizontal lines across, but *n+1* rows of these down.
- Vertical lines: the board can have *n+1* vertical lines across, but only *n* rows of these down.

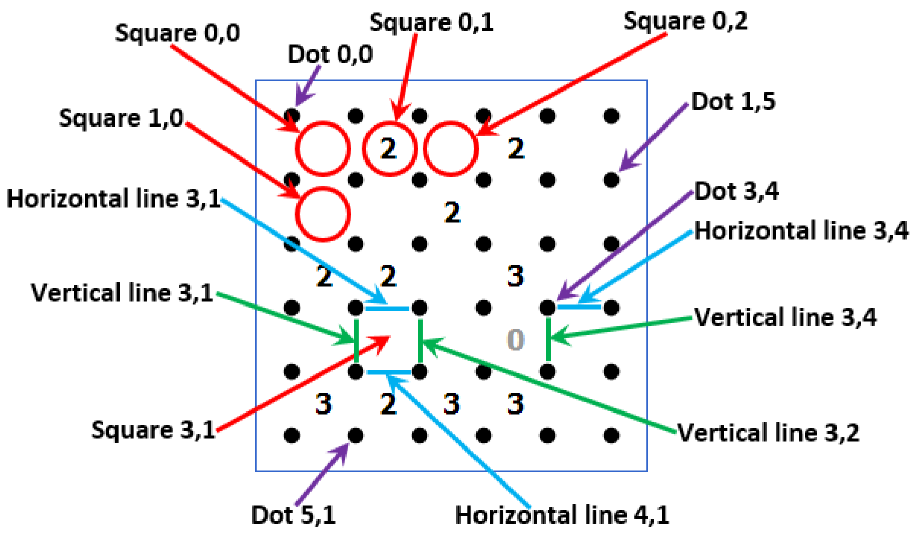


Figure 4.

Squares, dots, and lines are all indexed by two numbers `r,c`, short for row and column. In computer graphics, it is usual to index from the top-left corner as shown in Fig. 4, so

- the top-left square is index `0,0`, the bottom-left square is `n-1,0`, the top-right square is `0,n-1`, and the bottom-right square is `n-1,n-1`;
- the top-left dot is index `0,0`, the bottom-left dot is `n,0`, the top-right dot is `0,n`, and the bottom-right dot is `n,n`;
- the horizontal line at the top of Square *r,c* has index *r,c*, and the horizontal line at the bottom has index *r+1,c*;
- the vertical line at the left of Square *r,c* has index *r,c*, and the vertical line at the right has index *r,c+1*;
- the dot at the top-left of Square *r,c* has index *r,c*, and the dot at the bottom-right has index *r+1,c+1*;
- the horizontal line that goes right from Dot *r,c* has index *r,c*, as does the vertical line that goes down from that dot.

This gives us the three fields in the `Puzzle` class used to represent a puzzle and its current situation. (We do not need to represent the dots; they are always the same.)

```
private int[][] puzzle; // -1 if no number, otherwise 0-3
private boolean[][] horizontal; // true if on, false otherwise
private boolean[][] vertical; // ditto
```

`puzzle` represents the problem; the pattern of numbers and empty squares. `horizontal` and `vertical` represent the current solution; the pattern of line segments which are present on the board.

A 5x5 puzzle like Fig. 4 would have arrays with dimensions 5x5, 6x5, and 5x6, respectively. In each case, the first sub-array contains the information for the top row, the second sub-array contains the information for the second row, etc.

Project Materials

Download the folder *Project2.zip* from the LMS Project 2 page. This folder contains the following.

1. Skeletons for three Java classes.
    - `Puzzle` maintains the internal data structures representing a puzzle and its current board situation.
    - `SlitherLink` displays the puzzle and its current situation on the screen, and allows the game to be played using mouse input.
    - `AnalyzeSolution` implements the methods for testing whether the line segments currently on the board constitute a complete solution to the puzzle.

    Collectively, these classes implement the game of Slither Link. You are required to complete the constructors and methods for all three classes. Where the body of a constructor or method contains a comment `COMPLETE THIS`, delete the body and replace it with code which implements the required functionality, as described in the associated program comment. The numbers in these comments describe a suggested order for you to perform these tasks.

2. Two fully-written classes.
    - `FileIO` will help you to perform file input operations in `Puzzle`.
    - `SimpleCanvas` will help you to perform screen output and mouse input in `SlitherLink`.

    You do not need to change or submit these classes. Especially do not change `SimpleCanvas`; during marking we will use **our version** of this class.

3. Three JUnit test classes, which are provided for you to check your code. Note that the test cases are not complete; a method that gets all green ticks is not guaranteed to be completely correct. Additional test cases will be used for marking and it is your responsibility to thoroughly test your code. For obvious reasons, JUnit cannot be used to test your screen display.

4. Several example input files *eg*.txt*, for use with the main `Puzzle` constructor.

Project Management Tips

Before starting the project, students are expected to have

- studied the lectures and the relevant chapters of the text;
- completed the assigned labs during Weeks 2-9;
- read and understood the whole of this project description;
- and read and understood all relevant UWA policies.

Note especially that The Game of Life lecture and the Fifteen Puzzle lab will set you up well for this project. Submit any questions about any of this to *help1001*.

It is recommended that you tackle the project tasks in the order indicated; that you compile frequently; and that you test and run the code after completing each method, to ensure your code behaves as you think it ought, and does not fail any tests. You can gain good marks even if you do not complete all the methods, so long as the code you have written compiles and runs correctly. But if you submit a large body of code that does not compile or that crashes, then few marks can be awarded.

Hints and tips about the various methods may be uploaded here from time to time. Whenever that happens, the document version number will be updated.

1. For parsing input files, you will find the `String` method `split` useful. Look it up. Also `Integer` contains a method for turning a `String` into an `int`.
2. You can drive the game solely using the methods in `Puzzle`, but it's not much fun :). It's useful for testing purposes though.
3. With mouse input, the tricky bit is figuring out where the player needs to click to toggle a particular line segment. Drawing a picture will help, as will experimenting at puzzle-loop.com.

4. Having a way to ask if the puzzle is finished by using the mouse is a useful thing. As is having a way to clear the puzzle of all line segments. Other design features are up to you...
5. To clarify the required output for `tracePath`:
   - if there are no links to the initial dot, you should return `"No path"`;
   - if there is a closed loop with no branching, you should return the length of the loop as a `String`, e.g. `"32"`;
   - if the line from the initial dot ends, you should return `"Dangling end"`;
   - if the line branches, you should return `"Branching line"`;
   - if it does both of these things, you can return either of these results.

   You can also append any other content you like to these results, and the testing/marking code will still be happy.
6. Similarly, for `finished`:
   - if the puzzle solution is complete, return `Finished`;
   - if there is a dangling end anywhere on the board, you can return `Dangling end`;
   - if there is a branching line anywhere on the board, you can return `Branching line`;
   - if there is a square with the wrong number of line segments anywhere on the board, you can return `Wrong number`;
   - if there are disconnected lines anywhere on the board, you can return `Disconnected lines`;
   - if there are multiple failure modes anywhere on the board, you can return any of the relevant results.

   Again, you can append anything to these results, and the testing/marking code will still be happy.

Project Submission

Submit your three completed **.java source files** by the submission deadline above via *cssubmit*. Do not submit anything else. No other method of submission is allowed. **Make sure that all students involved are clearly identified in all of your classes.**

The file names, class names, and method signatures in your submitted code must match the original specifications exactly. The marking process is partly automated, and any changes you make which complicate this process will be penalised. If your code cannot be compiled with the original JUnit test cases provided, your submission will be penalised. It is ok to add other methods if you like, so long as you do not change the signatures of existing methods.

Common mistakes are to submit .class files in place of .java files, or to submit the test classes in place of your code. If you do one of these (or something similar), you will be notified by email as soon as we become aware, but you will be due for any applicable late penalty up to the time you re-submit. *cssubmit* makes it easy to check your files after you have submitted them - do it!

Project Assessment

Your submission will be assessed on

- completeness: how many of the methods you have written;
- correctness: whether your methods implement the specifications exactly;
- clarity: whether your code is clear to read and well-constructed.

JUnit testing classes are provided to help you check your program before submission, but the correctness of your program remains your responsibility.

Completeness and Correctness guidelines (/40)

The marks available for each constructor and method are as follows. These numbers give some indication of how difficult we believe each component is.

`Puzzle:`	constructor (1), `parseFile` (3), `horizontalClick` (1), `verticalClick` (1), `clear` (1)
`SlitherLink:`	constructor (2), `displayPuzzle` (3), `horizontalClick` (2), `verticalClick` (2), `mousePressed` (5)
`AnalyzeSolution:`	`linesAroundSquare` (2), `badSquares` (3), `getConnections` (3), `lineSegments` (3), `tracePath` (5), `finished` (3)

Methods will be assessed independently, so e.g. even if your `getConnections` is faulty, other methods that use `getConnections` could still get full marks.

Completeness and correctness will be evaluated (partly) automatically using JUnit tests. You should, therefore, try to ensure the following.

- The code passes the tests as per the specification.
- Validation should be **exactly** what is required by the assignment specification.
- Incorrectly rejecting a parameter is incorrect and will be penalized, so do not add extra validation that is not asked for.
- Do not print things to `System.out` or `System.err` that are not asked for. Use the BlueJ debugger rather than `print/ln` statements for debugging your code.
- Do not change the given file names or class names, or modify the signatures of existing methods.

Clarity guidelines (/10)

- Ensure you fill in the @author and @version fields in the header comments with your details for each submitted class.
- All code should be neatly laid out and indented, with lines no longer than 80 characters.
- Variables should be given appropriate names.
- Select appropriate programming constructs for your implementations.
- Keep code as simple as possible for the job it is required to do.
- If the logic you have used in a method is complex, a brief comment should be added explaining the strategy you have adopted; but otherwise, code should not be commented unnecessarily.
- Feel free to add fields *where appropriate* in the `SlitherLink` class; especially make good use of `final` variables for your display code.

Help!

The quickest way to get help with the project (unless you are sitting in a lab session!) is via *help1001*. You can ask questions, browse previous questions and answers, and discuss all sorts of topics with both staff and other students.

Please read and follow these guidelines.

- Do not post project code on *help1001*. For obvious reasons, this behaviour undermines the entire assessment process: as such you will be liable for punishment under the University's Policy on Academic Conduct, even if the behaviour is unintentional.
- Before you start a new topic, check first to see if your question has already been asked and answered in an earlier topic. This will be the fastest way to get an answer!
- When you do start a new topic, give it a meaningful subject. This will help all students to search and to make use of information posted on the page.
- Feel free to post answers to other students' questions. And don't be afraid to rely on other students' answers: we read everything on *help1001* and we will correct anything inaccurate or incomplete that may be posted from time to time.
- Be civil. Speak to others as you would want them to speak to you. Remember that when you post anonymously, this just means that your name is not displayed on the page; your identity is still recorded on our systems. Poor behaviour will not be tolerated.
- Do not post project code on *help1001*, or anywhere else online. This one is worth saying twice.

If you have a question of a personal nature, do not post to *help1001*: please email Lyndon instead.

Project Extension

An extension to the project is available by which you can earn extra marks in the assessment.

- Up to 5 marks can be earned this way, which can replace marks lost in the main assessment.
- It is not possible to be awarded a total mark greater than 50 for the project.
- Late submissions will not be awarded any extension marks.
- If you attempt the extension, submit in the normal way and email Lyndon before the deadline to let me know what you have attempted. No extension marks will be awarded without such notification.

The best extended project will be awarded a prize*.

Extension details

While you can get full marks in the project if you submit a correct program with no clarity issues and with a basic mouse-driven display, there are many ways in which the game might be improved. Use whatever sources you find on the Internet and your own imagination to come up with ideas for extending the player's experience, and implement those ideas to earn bonus marks. Broad ideas include

- enhancing the board display;
- reporting solution failures in more detail;
- using the mouse in more interesting ways;
- introducing some sort of scoring system;
- or whatever else you might think of.

Go for it!

---

*All prizes are awarded or not solely at Lyndon's discretion.

UWA Department of Computer Science & Software Engineering