SLITHERLINK GAME AND SOLVER

PROJECT INTELLIGENCE REPORT

Kushagra Udai

CISE Department, University of Florida.

UFID 0937-7483

1 INTRODUCTION

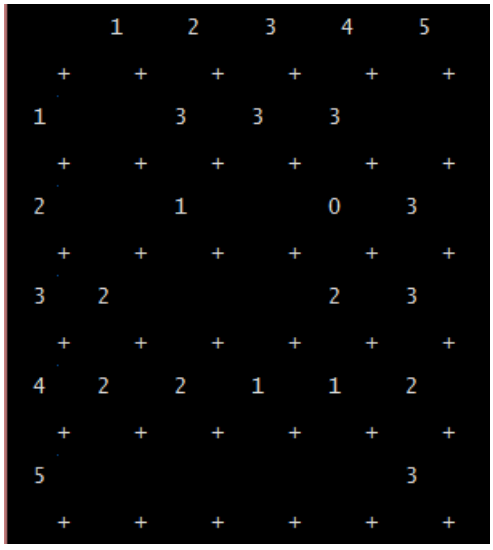
Slitherlink (also known as Fences, Takegaki, Loop the Loop, Loopy, Ouroboros, Suriza and Dotty Dilemma) is a popular puzzle which is comprised of a board of numbers which is aactualy a rectangular lattice of dots. More specifically, the board given to us consists of an m by n grid of numbers or blank spaces, with each cell of the grid surrounded by 4 dots on the four corners of that cell. Lines can be placed along any two adjacent dots horizontally or vertically (but not diagonally). Cells can either be empty or filled with number ranging from 0 to 3. The aim of the game is to create a single continuous loop using the lines such that all numbers on the board have exactly as many lines around them as the number itself.

2 IMPLEMENTATION APPROACH

The game can be played by a human as outlined below or it can be solved by the program itself. At the start of the program, the user is asked if he wants to play. If the answer is affirmative, the user is asked if they want to load a default board or if they would like to input a filename for their own board. The format to be used for the input file is a set of lines of text, with each line being the horizontal row of the board. Each number must be entered as is and each blank space MUST be entered as a space in the file. Using this scheme, each row will have the same number of columns filled with either numbers or spaces.

After the file has been successfully read by the program, the rules of the game are displayed to the user, who is then be asked to make a move. The user can enter moves by specifying the row, column and the side of the cell that he wants to place a line at in the format (row column side). If the user wishes to find the solution of the puzzle, they can simply type “solve” at the move input prompt to force the progam to attempt to solve the puzzle via a brute force depth first search after determining all (most of?) the deterministic lines that can be found from the specific input that is the board. When the game finishes, the list of moves made are displayed to the user, if the user solved the puzzle. Otherwise, the board is displayed with the deterministic lines and then the solved board is displayed. Also, the user can quit a particular board by simply typing “q” at the input prompt.

The program maintains a 2D array which will hold the values for the board. The size of this array is (2xm+1) by (2xn+1) for a m by n Slitherlink grid. Every time the user makes a move, the values in the array are updated and the updated board is displayed. The format for showing the board is shown below, for a sample 5x5 board. It’s worth noting that we display the column number on the top of the board before the first row and the row number on the left before the first column of the board to aid the user in finding the correct co-ordinate for move input.



Manual Play (Lines Input by the User) –

Input for the moves are made as (row column side) where row is the row number of the cell, column is the column number of the cell and side can be t, b, l or r which indicate the top line, bottom line, left line and right line of the cell. A board check condition is called after each user input to figure out if the game board is complete (solved). The check is described below:

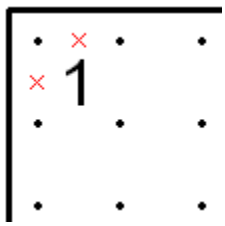
Check condition: The condition first scans through the board for numbers and if it finds a number, it checks if the number of lines around it are the same as the number itself. As soon as it finds a number that does not satisfy this condition it returns from the check condition with a negative result. After that, it goes through every dot (vertex) on the board and checks if there is any dot with a degree that is not 0 or 2. As soon as it finds such a vertex, it returns from the check condition with a negative result. And finally, if both these conditions are satisfied, it scans through the board to find a line, and using that line, finds the loop that contains that line by following it in both directions. Thereafter, it scans the board again and checks every line on the board against the loop obtained in the previous step. If we find any lines which do not belong to the list of lines obtained in the previous step, we return with a negative result because that indicated that there are two loops on the board. If none of the lines on the board fail this condition, we return with a true from the check condition.

AI Play (Automatic Solver) –

If the user types “solve”, the program’s solver is initiated. The implementation used here is described below:

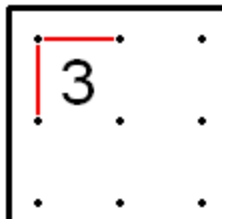
1. First, we find all deterministic lines from the given board. This is done using the following given conditions:

- **Corners**
  - If *a 1 is in a corner*, the actual corner's lines may be X'd out, because a line that entered said corner could not leave it except by passing by the 1 again. This also applies if two lines leading into the 1-box at the same corner are X'd out.



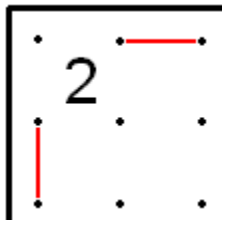
A 1 in a corner.

- If *a 3 is in a corner*, the two outside edges of that box can be filled in because otherwise the rule above would have to be broken.



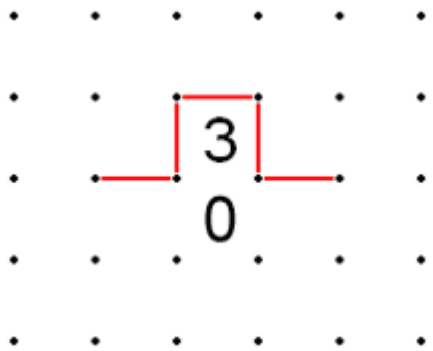
A 3 in a corner.

- If *a 2 is in a corner*, two lines must be going away from the 2 at the border.



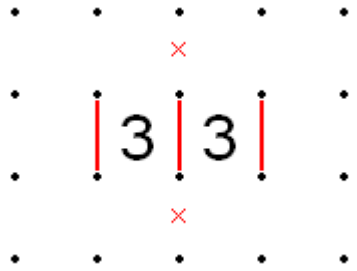
A 2 in a corner.

- **Rules for squares with 3**
  - If *a 3 is adjacent to a 0*, either horizontally or vertically, then all edges of that 3 can be filled except for the one touching the 0. In addition, the two lines perpendicular to the adjacent boxes can be filled.



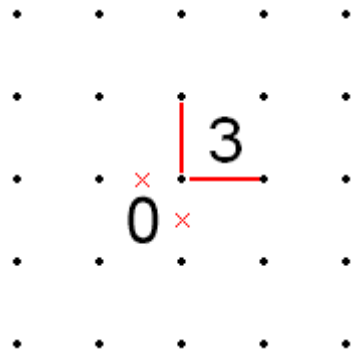
A 3 adjacent to a 0.

  - If *two 3s are adjacent* to each other horizontally or vertically, their common edge must be filled in, because the only other option is a closed oval that is impossible to connect to any other line. Second, the two outer lines of the group (parallel to the common line) must be filled in. Thirdly, the line through the 3s will always wrap around in an "S" shape. Therefore, the line between the 3s cannot continue in a straight line, and those sides which are in a straight line from the middle line can be X'd out.



Two adjacent 3s.

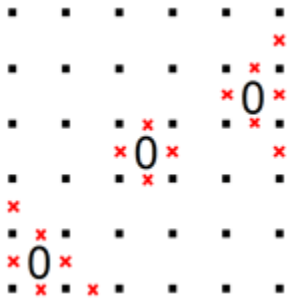
  - If *a 3 is adjacent to a 0 diagonally*, both sides of the 3 that meet the 0's corner must be filled. This is because if either of those sides were open, the line ending in the corner of the 0 would have no place to go. This is similar to the 3-in-a-corner rule.



3 diagonally next to 0.

- **No lines around a 0**

The 0 means it is not surrounded with any lines so we can place four X's around it to show all four links are excluded. This example also shows a 0 in a corner and a 0 on a side. In both cases, two additional X's are marked to show that lines are not allowed because they cannot be continued.



3 CODE LISTING

Functions Used (With Parameters) –

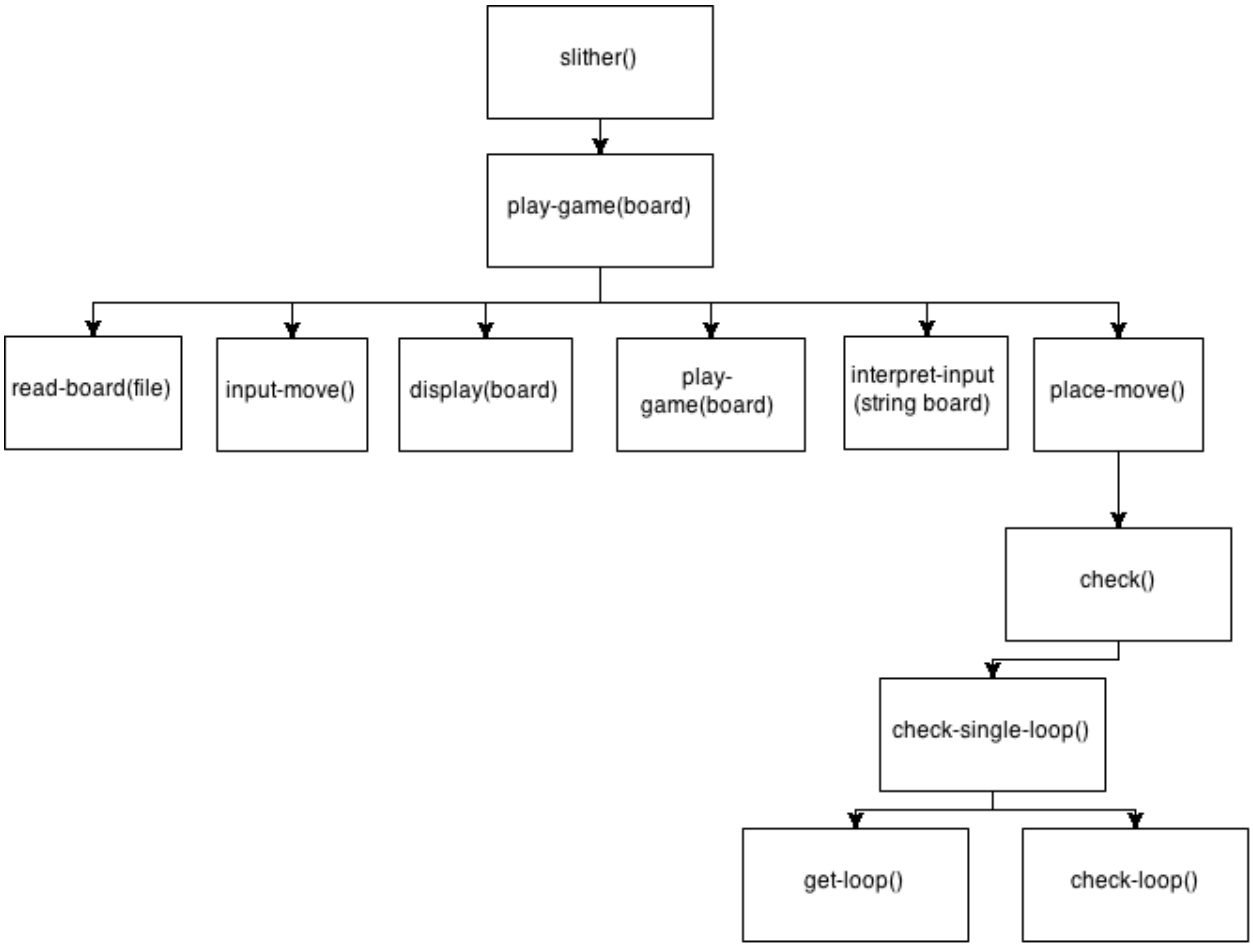
- slither() – Main method. Responsible for the start of the game
- read-board (file) – Used to read the board input from a file.
- loop-for-valid-file() – Recursively used to loop until a valid file name is given.
- play-game (board) – Used to take input, allow user to quit and solve a board.
- input-move – Used to take move input from the user for a move.
- print-winning-moves (moves) – Print all moves made by the user when the game ends.
- display (board) – Used to display the current state of the game. Called after every move.
- convert-board (strings) – Used to convert the read strings from file into an array to be used for as the board for playing the game.
- vertex-check (x y) – Checks if coordinate (x, y) is a vertex (dot).
- face-check (x y) – Checks if coordinate (x, y) is a face (number or blank)
- line-check (x y) – Checks if coordinate (x, y) is a possible line (array space which is not a vertex or a face).
- make-array-board (dimensions) – Called by convert-board to create an array of appropriate size.
- actual-size (num) – Called by make-array-board to calculate the size of the array.
- interpret-input (string board) – Used to convert user input to coordinates for the array and check validity.
- side-check (char) – Checks if the side entered by the user is valid (t, b, l or r).
- row-check (x board) – Checks if the user entered row number in input is valid for the current board.
- column-check (y board) - Checks if the user entered column number in input is valid for the current board.
- rowp (x board) – Checks if x is a valid row in the board array.
- columnp (y board) – Checks if y is a valid column in the board array.
- place-move (to-place board) – Places the user input move from the interpreted input.
- split (input) – Separates the 3 characters entered by the user as a move to parse.
- check (view) – Checks for the winning condition as described above.
- check-space (view x y) – Checks if the cell at (x, y) is satisfied i.e. if it is a number, it has that many lines around it.
- edge-count (board x y) – Counts the number of lines around a cell or a vertex.
- x-count (board x y) – Count the number of x's around a cell or a vertex.
- possible-edge-count(board x y) – Counts possible number of lines that can be placed around a vertex (this varies from 4 at the edges of the board)
- is-line (view x y) – Checks if the coordinate (x, y) is a line.
- is-x (view x y) – Checks if the coordinate (x, y) is an X.
- is-space (view x y) – Checks if the coordinate (x, y) is an empty element which can hold a line.

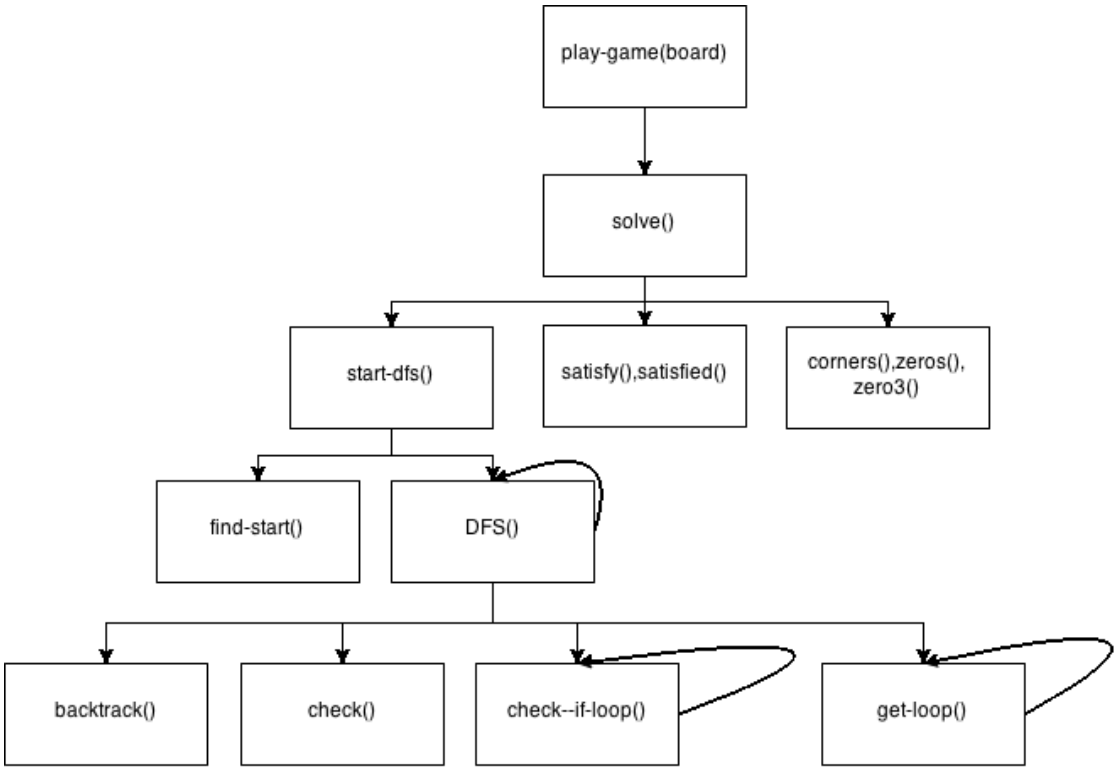
- check-single-loop(board) – Checks if the board has exactly one complete loop.
- check-loop(loop-list to-check) – Checks if coordinate to-check belongs to the looplist.
- get-loop(board loop-list x y &optional (from-x -1) (from-y -1)) – Recursively generates looplist which the list of coordinates of a loop by following the line on (x, y) on both sides starting from (x, y).
- solve(board) – Cleans the board of all user input lines, calls all constraints and calls start-dfs to solve the board, culminating with a call to cleanX to remove all Xs from the board and finally displaying the solved board.
- clean(board) – Cleans all user input lines to start solving. Called by solve.
- cleanX(board) – Cleans all Xs from the board after finding a solution using DFS.
- zeros(board) – Applies all constraints for 0s on the board.
- corners(board) – Applies all constraints for corners on the board.
- zero3(board) – Applies all constraints for 3s and 3s around 0s on the board.
- satisfied(board) – Applies all constraints for cells and vertices which are already satisfied.
- satisfy(board) – Applies all constraints for cells and vertices which are need only one more element in the array to be added to be satisfied.
- find-start(board) – Finds a vertex with degree 2 to pass to DFS to start solving. Called by solve.
- start-dfs(board start-x start-y) – Initializes all variables required for dfs.
- dfs(x y &optional (from-x -1) (from-y -1)) – Recursively finds a solution by exploring all possible solution spaces after all constraints have been applied and all deterministic lines have been found.
- backtrack() – Backtracks the last added line in the stack al.
- check-if-loop(looplist) – Checks if the first and last coordinate in the looplist are adjacent to figure out if looplist is actually a loop.

Major Variables Used –

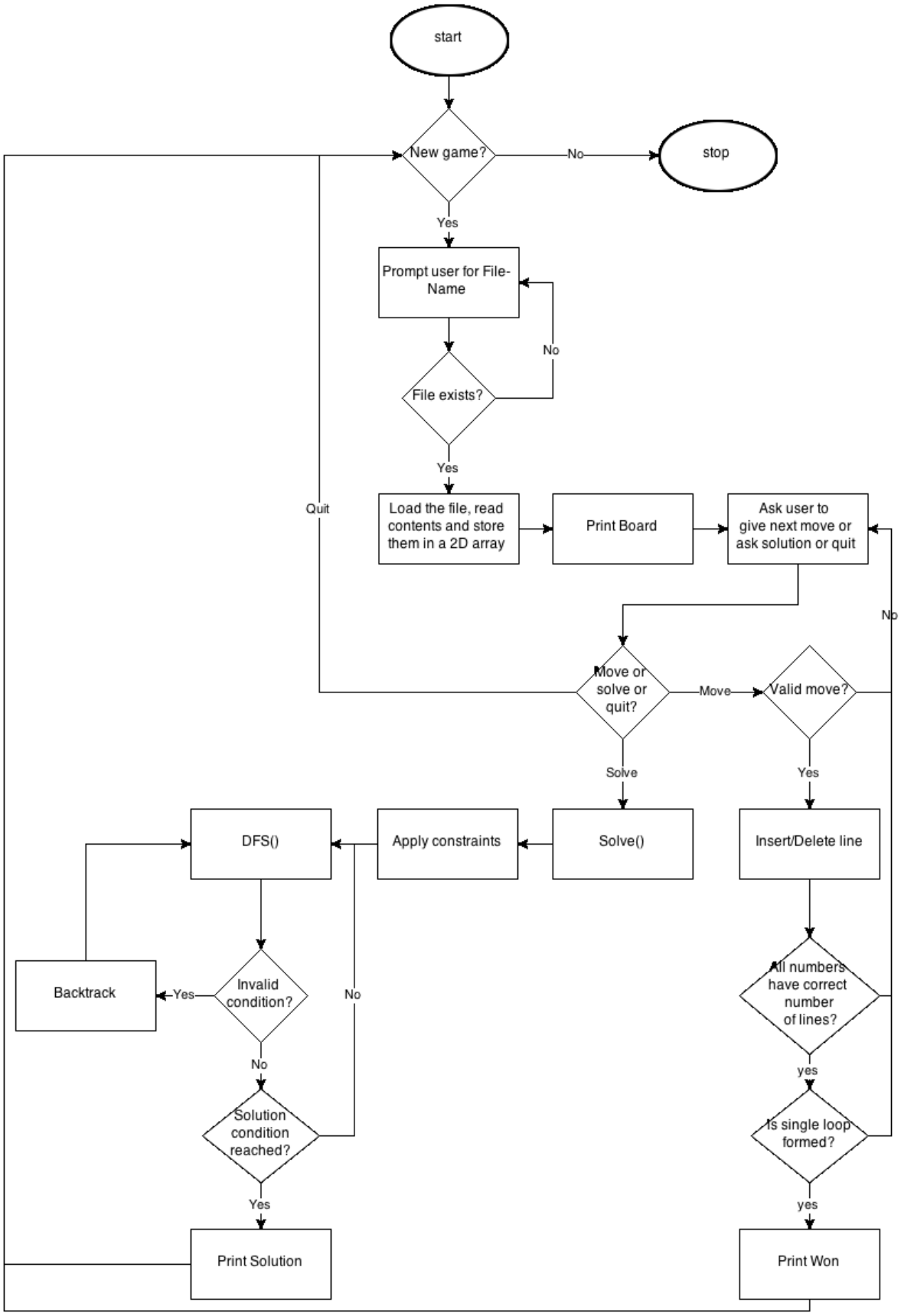
- board – Used to store the array which is the current game board.
- file – Used to store the filename from which the board will be read.
- move – Stores the current user move.
- moves – Stores the list of moves made by the user.
- to-place – Stores the parsed move as a list.
- input – Stores the input as a string to be parsed.
- view – Local variable used to store the board.
- edge – Used to count number of edges around a vertex or face.
- xc – Used to count xs around a vertex or edge.
- remains – used to count empty spaces around a vertex or face.
- loop-position-list, looplist, loop-list – Used to store the loop found from getloop()
- flag – Used to check if the any constraints can still be applied to the board.
- start-x, start-y – Used to store start coordinates for the DFS.
- won – Used to store the board state. T if solved and nil if not solved.
- al – List of coordinates used as a stack used for DFS to store all lines added by DFS.
- (x, y) – Used everywhere to store coordinates.

4 CALLING HIERARCHY





5 FLOWCHART


start
New game?
No
stop
Yes
Prompt user for File-Name
No
File exists?
Yes
Quit
Load the file, read contents and store them in a 2D array
Print Board
Ask user to give next move or ask solution or quit
No
Move or solve or quit?
Move
Valid move?
Solve
Yes
DFS()
Apply constraints
Solve()
Insert/Delete line
Backtrack
Yes
Invalid condition?
No
All numbers have correct number of lines?
No
Solution condition reached?
yes
Is single loop formed?
Yes
yes
Print Solution
Print Won

6 INTELLIGENCE IMPLEMENTED / LEFT-OUT

The following were the goals at the time of the initial report submission –

1. Apply all kinds of constraints to find the deterministic lines for any specific board input.
2. Eliminate all illegal lines from the board.
3. Use an A* search to solve the board. Additionally, use Jordan curves to facilitate the solution using A*.

Final Implementation: Coming up with a viable A* heuristic which would be general enough for all kinds of board inputs proved to be challenging and after multiple failures in solution generation from different A* implementation, I finally dropped the idea and resorted to a simpler, more inefficient but guaranteed solution technique using DFS. While the solution is more time consuming, it is guaranteed.
Also, multiple constraint were applied to the board before solving via DFS to prune the solution space and make the program faster, as originally planned. Furthermore, all illegal moves that could be found on the board from the given input were eliminated as planned in the initial report. However, having to drop the A* idea did not give me enough time to think about a heuristic based on Jordan curves that could help DFS choose the most optimal path first. A guiding heuristic would serve as a locally greedy approach which would solve the board faster by making better choices instead of brute force solving of all possible boards.

7 THINGS THAT COULD HAVE BEEN DONE DIFFERENTLY

1. In hindsight, I should have spent more time optimizing DFS. I managed to come up with a good heuristic for the solution using Jordan curves. The conclusion from the Jordan Curve Theorem is that any row of the grid must have an even number of vertical lines and any column must have an even number of horizontal lines. When only one potential line segment in one of these groups is unknown, you can determine whether it is part of the loop or not with this theorem. Using this, DFS's solution space could have exponentially reduced by a factor of 4!

2. I did not spend enough time exploring LISP in the beginning of the semester or until the last phase of the project. This led me to find some of the cooler features of the language much later than I should have and hence, I could not use them. I could have made my code much smaller using macros and functional currying to make more generic functions which return specific functions and hence have a single function replace several others. I would ideally like to refactor the entire code and clean it up significantly. This looks far too botched together for my liking.

Submitted by
Kushagra Udai
UFID: 0937-7483.