Galou is Back

Selected Fun Problems of the ACM Programming Contest

Luca Dreiling
Universität Tübingen
Tübingen, Germany
luca.dreiling@student.uni-tuebingen.de

ACM Reference Format:
Luca Dreiling. 2020. Galou is Back: Selected Fun Problems of the ACM Programming Contest. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 THE PROBLEM

1.1 Problem Description

Zak Galou a retired witch needs help. His tractor has been sabotaged. There have been gears added and some have been removed. After these modification the engine is not running. Now some gears are blocked and some do not spin at all and starting the engine would break it. The task is to find out how the gears inside the engine will spin without starting it.
An engine contains of multiple grids. Based on a checkerboard pattern each position(or tile) in the grid can be indexed using (x,y)-coordinates with the origin on the top left. Additionally each line is offset by half a tile compared with the line above.

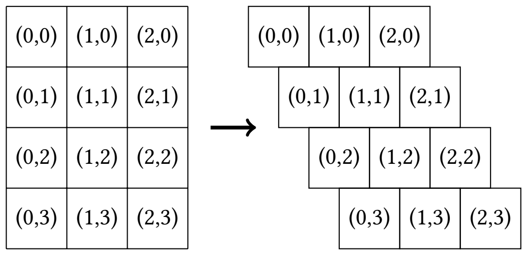

Figure 1: Positions within a 3x4 grid

Each tile inside the grid can contain a gear but can also be empty. Gears have a state that determines their behavior. They can be blocked, turning (counter-)clockwise or not turn at all. Some gears are initial powered and will always turn clockwise once the tractor starts.
Turning an already turning gear in the same direction will not affect the gears' turing direction. Trying to turn a gear in two opposite direction (clockwise and counterclockwise) will block the gear. Blocked gears will automatically block all other connected gears.

After the effect of all initial gears have been calculated the final state of the board should be printed to Standard Output.

1.2 The Language: Godot/GDScript

Godot is a free open-source game engine. Everything in Godot is based on Scenes which mimic a tree-like structure.
With a parent and the ability to sotre multiple children each Scene acts like a tree node. Scenes can have other properties depending on their purpose.
Children of a scene act relative to their parent, e.g. if the parent moves they will move too.
Godots' build in IDE supports most common languages but mainly uses its own python based script language: GDScript.
The language has been optimized for godots' rendering engine and game development in general. That is done by removing some less-important concepts (for game development) but adding e.g. primitive 2D and 3D Vector types for in game positioning.
Variables have to be declared using the keyword **`var`** improving the readability.

1.3 Input Data Description

The data is distributed as a well-formed string so an input is always in the desired format. Therefore checking for a valid input is not neccessary.
The problem description does specify that the data must be read from standard input. However Godot has no standard input. Therefore the data gets stored in a seperate text file named `data.txt`. This file will be loaded and parsed as the program starts.
The input consists of a description of multiple grids.
Each grid starts with a line consisting of two integers. The first represents the width (or x-coordinate) of the board and the second one the height (or y-coordinate).
This line is followed by y lines, consisting of x characters representing a row of gears inside the grid.
These characters can initially be:

.	no gear at this location
F	a gear
I	an initial powered gear (turning clockwise)

After solving the grids state the gears addiditonaly can be

(	a gear turning clockwise
)	a gear turning counterclockwise
B	a blocked gear

Note: There can not be an initial gear in a solved grid because it will turn initally.

The end of the input is marked by a single line after a complete grid. The line marking a boards size of (0, 0), e.g. the line `0 0`.

An input of a 3x4 and a 1x1 grid might look like this:

```
3 4
. . .
. F F
. I .
. . F
1 1
I
0 0
```

Figure 2: Example Input

1.4 Output Data Description

The output of the program should be printed similar to the input with one exception. All lines representing board sizes are replaced with empty lines including the last line with a board size of (0,0).

The output gets printed to Godots' standard output. Using the input from Example Input the output should be:

```

. . .
. B B
. B .
. . F

(

```

Figure 3: Output of the example Input

2 SOLVING THE PROBLEM

2.1 Coding helper

To enusure readability of the code snippets all gearstates are stored as constants. These variables contain the Strings that represent the state.
Note that the states for clockwise and counterclockwise turning have been named `GEAR_RIGHT` and `GEAR_LEFT` respectly to keep the names short.

```
const GEAR_INITIAL = "I"
const GEAR_FREE = "F"
const GEAR_LEFT = ")"
const GEAR_RIGHT = "("
const GEAR_BLOCKED = "B"
const GEAR_NONE = "."
```

Figure 4: Output of the example Input

2.2 Analyzing the Problem

The first task is to realize how two gears interact with each other. Gear *A* might try to turn gear *B*. That will be refered to a *A* applies a force to *B*.

The forces applied can be derived from the state of gear *A*.
A turning gear turns the connected gears in the opposite direction. An initial gear will turn clockwise, so it will turn the connected gears counterclockwise and blocked gears applies a blocked force. Nonexisting gears do not apply a force.
In conclusion forces can be turning (counter-)clockwise or blocked.

Using the behavior of real gears and the restrictions of the problem the transition can be viewed with following table:

Table 1: Results of forces applied

		Force		
		(	)	B
Gear	I	)	(	B
	F	)	(	B
	(	B	(	B
	)	)	B	B
	B	B	B	B
	.	.	.	.

Note: Initial gears can use the same behavior as clockwise gears since they will be converted in one anyway. This method was not choosen for better comprehensibility as discussed in section Step by step.

The table above can be minimized and expressed with if-else-statements in GDScript:

```
func calc_force(_gear, _force):
  if _gear == GEAR_NONE:
    return GEAR_NONE
  elif ((_gear == GEAR_FREE
      or _gear == GEAR_RIGHT)
      and _force == GEAR_LEFT)
      or _gear == GEAR_INITIAL:
    return GEAR_RIGHT
  elif (_gear == GEAR_FREE
      or _gear == GEAR_LEFT)
       and _force == GEAR_RIGHT:
    return GEAR_LEFT
  else:
    return GEAR_BLOCKED
```

Figure 5: Output of the example Input

2.3 Which algorithm should be used?

Every turing or blocked gear applies a force to all other gears it is connected to. The new gears also apply a force to their connected gears until all gears have been visited. This is similar to the behavior of a tree traversion. Depth first search is used for the tree traversion due to it's simple recursive implementation. The algorithm is complete for a cluster of connected nodes (in this case gears). All clusters containing a initial gear must be traversed, the others will not be turned and stay free. Considering this the search will be complete for the whole board.
Visiting every gear only once will not calculate the final state of the board. A gear must already be turning to get blocked otherwise it will turn according to the applied force. Another problem is that in case of a blockade it will not affect the state of the visited gears within the cluster.
To compensate this behavior the simplest solution is to let the depth first search visit a gear multiple times. Since visiting a gear once was the previous break condition of the recursion the search would carry on indefinitely. To compensate that the search should only continue if an applied force changes the state of a gear [1], e.g. applying a clockwise force to a free gear.

Since the grid is shifted a gear can interact with a different set of gears compared to a simple checkerboard pattern (e.g. above, below, left, right). With a gear placed at (0,0) the offsets can be used to illustrate which gears are connected.

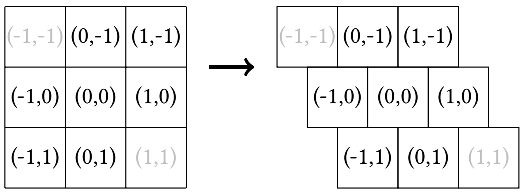

Figure 6: Offsets of connected gears

This results in the offsets $\{(0,-1),(1,-1),(-1,0),(1,0),(-1,1),(0,1)\}$.

These offsets can be used to calculate the position of every neighbor by adding it to the position of the current gear applying the force. The new positions must be checked whether or not they are on the board. If so the search will continue recursively until the final state (no changes) of the board has been calculated.

3 TWEAKS & IMPROVEMENTS

3.1 Step by step

Depth first search iterates over the board visiting a gear at a time. For better understanding of the algorithm a step by step mode has been implemented. It enables the user to retrace every interaction between gears that created the final solution.
Solving a step at a time and restarting the recursion at the same state would overcomplicate the recursion.
The solution is simple: Information about every step (that changes a gears' state) get appended to a list. For each step the coordinates and new state get stored.
After the board has been solved completely the steps will be diplayed using the information gathered in the list. Each step pops the first element of the list and changes the gear at the position to the stored state.
If a force has no effect on a gear the step will be skipped. These steps have no effect on the final solution and can be neglected.

Table 2: Generated steps for the Example Input

List index	Grid 1	Grid 2
0	$[(1,2),(]$	$[(0,0),(]$
1	$[(1,1),)]$	
2	$[(2,1),(]$	
3	$[(1,2),B]$	
4	$[(1,1),B]$	
5	$[(2,1),B]$	

3.2 Reevaluating the Transition Function

Taking advantage of the clusters within the grid the runtime can be boosted by just a tiny amount.
Taking the fact that a single initial gear will cause the whole cluster to be explored. Any more initial gears in that cluster will be overwritten since they were treated like free gears during the traversion. The overwritten gear then must be reevaluated by starting another search on that gear turing it initially clockwise.
Instead of treating an initial gear like a fee gear and then turning it clockwise the initial gear can be treated like a turing gear right away. Turning an initial gear clockwise will now result in a blocked gear instead of a counterclockwise turing one. The rest of the transition function does not change as seen in Table 3.

Table 3: Reevaluated transition function (Changes marked bold)

		Force		
		(	)	B
Gear	I	**B**	(	B
	F	)	(	B
	(	B	(	B
	)	)	B	B
	B	B	B	B
	.	.	.	.

Using this modified function a simple iteration over the grid solves it. This reduces the number of depth searches to the number of clusters in the grid which are less or equal to the initial gears.

In this implementation the function is not used because the effects of every initial gear can be seen more clearly. Retracing the modified function without the knowing about this behavior might lead to false conclusions.

3.3 Modifying the Board

The algorithm can detect blockages and free gears. With that knowledge the user might try to fix the grid and check if the modifications will remove any blockages. This can be done by adding and removing gears from the grid. Initial gears are directly connected to the engine and removing them prevent the engine from applying its force on the gears. There is no other information on torques applied by the engine. Both placing and removing initial gears will be prohibited.
Adding and removing will be realized by a simple UI that lets the user change the grid by dragging a new or an empty gear on a position.
Godot/GDScript comes with various built in Scenes one of which is a TileMap. TileMaps are used in most 2D games to create a big background from multiple small images (or tiles).
For each gear a color-coded tile is created forming the input grid. Blocked gears are red, initial gears green, free yellow. Turing gears are coded white and have been animated to turn in (counter-)clockwise. An offset between the rows of can be defined and is set to half a Tiles' size to mimic the shifted grid.
The origin of the TileMap is on the top left but the first full tile of each row has the x-index 0. For the Tiles to be set correctly indices must be converted between TileMap- and Grid-coordinates.

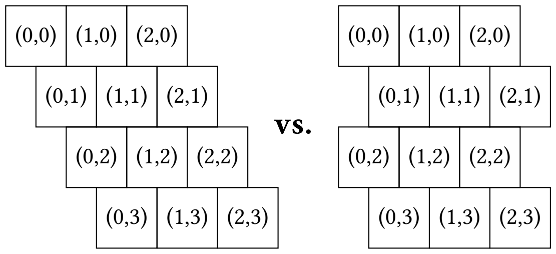

Figure 7: TileMap vs. Grid coordinates

Observing these patterns the y-coordinates are the same but the x-coordinates differ. Using formulas their index can be described by $f_{TileMap} = \frac{y \mod 2}{2}$ and $f_{Grid} = \frac{y}{2}$. Both function are similar and only differ in the nominator. The difference between modulo and a normal division is the divisor in this case $\frac{y}{2}$ (using integer division). The conversion can be done by adding or subtracting this difference from an index.

```
func as_grid_pos(_pos):
  return Vector2(_pos.x
      + int(_pos.y / 2), _pos.y)

func as_board_pos(_pos):
  return Vector2(_pos.x
      - int(_pos.y / 2), _pos.y)
```

Figure 8: Functions for converting indices

4 RUNTIME

Let there be a grid containing n gears. With $O(n)$ the runtime of a complete depth first search serves as a base.
Let there be $|I| \leq n$ initial gears. A search is started on every one of them. In order to find all initial gears the whole grid must be searched resulting in a runtime of $t_I = O(x \cdot y)$ (x, y are width and height of the grid).

With the implemented transition function all initial gear are treated as free. If an applied force does not change the state of a gear the recursion returns. So the state of a gear can change twice at most, e.g. Free and initial gears can change their state to turing and then to blocked.
Each call (for every initial gear) might traverse the whole grid, so the runtime will be $t_I + |I| \cdot 2t_{DFS} = t_I + |I| \cdot 2O(n) = t_I + |I| \cdot O(n)$.

Treating initial gears as discussed in Reevaluating the Transition Function improves the runtime of the overall runtime just a little bit:
There are $j \leq |I|$ clusters on which forces are applied. Any other Cluster initial gears can be ignored since they will never be reached by the depth first search anyway. All other inital gears found while searching are automatically treated as if they were turning already. Therefore only one search per cluster is enough. The overall runtime then would be $t_I + j \cdot O(n)$.

5 GIT REPOSITORY

A repository with the full code can be found at:
https://github.com/eisclimber/ACM_Solution_Galou_is_Back.