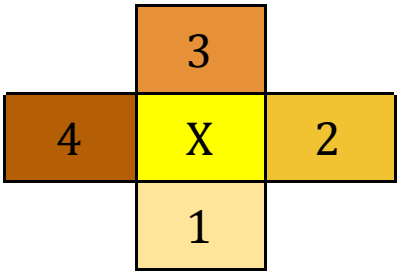
Project 3:

Bender

Pau Bestard Satorra
29/2/2020
1st DAW - Programming

Goals

This projects goal is to make a brute-force pathfinding algorithm and a heuristic calculator. The pathfinding algorithm is based on the following priority list, where X is the current position:



This algorithm has to choose a direction and move until it finds a wall or the goal. In the case that it finds a wall it must try all directions in the pre-defined order.

As said above walls are a special type of cells our map can have, the list is as follows:

- # Wall: These cells cannot be walked through, when the algorithm finds one it is forced to adjust its path.
- X Starting point: This cell defines the starting point of the algorithm
- $ Goal: This cell is the target we must reach.
- T Teleporters: The teleporter cells are inter-connected each other, when you enter one you exit through the closest teleporter.
- I Inverter: Whenever we walk on a inverter the priority list will be changed from **S-E-N-W** to **N-W-S-E** and vice-versa.

It is easy to see how this simple algorithm can and **will** get stuck on infinite loops. To resolve this issue we must add a loop-detecting function.

Pau Bestard Satorra
29/2/2020
1st DAW - Programming

Map

In a pathfinding algorithm such as this one the correct creation of the map is a crucial step, we have to make sure the map is a light as possible, as a big enough map would **destroy** the processing speed, and contains all the necessary information. To do the map we will use Javas Array, in specific a bidimensional array (Array[][]) of our custom class **Cell**.

Cells

Our map will consist in Cells, a Cell is a custom class we will created with the following attributes:

- **Type**: This defines what the cells function is.
- **X**: This defines the column of the map in which the cell is located (map[ ][ ])
- **Y**: This defines the row of the map in which the cell is located (map[ ][ ])
- **Heuristic**: This is the distance from the cell to the goal, this is used only in calculating the best path
- **Visits**: This stores the amount of times this Cell has been the **current Cell**.

For the class we will make two constructors, the default empty one, used for instantiating empty cells, and the main constructor which will receive as input a character and two integers. The character is the representation of the cell (#,T,I,X,$) and is used to assign the **Type** attribute. The integers represent the **X** and **Y** attributes respectively. As the other attributes are not necessary for all the methods we don't include them in the constructor.

Building the map

Now we have defined what the cells are we need to distribute them in the map. The map array will be created from a input string, like this one:

```
#######
# X     #\n
#       #\n
#       #\n
#       #\n
# $     #\n
#       #\n
#######
```

Although it is only one string we can see that the line-jump(\n) is used to represent different rows of the map. So we will start by creating the rows by using splitting the string into a array based on the line jumps. Now we will simply separate the rows into columns with a loop based on the split array. Apart from creating the array we also will store the **Goal**, **Start** and all the **Teleporters**.

The data flow structure.

Finding the path

Now that we have a map we have to create the path from the Start cell to the Goal cell. To do this we start by defining our current position as the starting point. Then we start a loop that will run until it finds the goal or a infinite loop is detected. This loop is as follows:

1. Check if we are at the goal
    a. If we are:
        i. Return the path
    b. If we aren't:
        i. Continue the loop
2. **Check if there is a wall**
    a. If there is:
        i. Move
        ii. If we landed in a teleporter, **teleport**
        iii. If we landed in a inverter, **invert**
    b. If there isn't:
        i. **Turn** around until we find a non-wall
3. Then we **check for loops**
4. Reset the loop

Here is the flow of the Path creating loop

This is the loop by which we find the path, now here is the explanation for each method.

Pau Bestard Satorra
29/2/2020
1st DAW - Programming

Turn

The method turn will change the direction so that we are left facing something that isn't a wall. To do this it will use the predefined direction priorities and turn around calling the **Check for Wall** function. If it finds a non wall it will stop.
The flow of turning

Check for Wall

In order to check for a wall we must look at the next cell over in the current **direction** and get its cell **type**. Then we simply return true if it is a wall and false otherwise.
The flow of finding walls

Move

The move function simply changes the **current** cell with then next cell over based in the **direction**. It also stores the direction in which it moved in the **path.**
The flow of moving

Teleport

The teleport function changes the current cell to the closest teleport whenever it is called. To do this it compares all the teleporters on the **teleporter list** and gets the one with the shortest distance. In the case that two teleporters are at the same distance it uses the following priorities:

8	1	2
7	X	3
6	5	4

This logic is easier to understand if we see it as angles on a circle:

315º	0º	45º
270º	X	90º
225º	180º	135º

The code to get the shortest one is simply a loop through the **teleporter list** where we got storing the shortest distance, and in the case that they are equal we compare their angles.

[The flow of teleporting]
[The flow of obtaining teleport priorities]

Invert

The invert function simply has to change the priority order based on if it is inverted or no. The priority lists are:

Default: S-E-N-W - **Inverted**: N-W-S-E

Check for Loops

To check for loops we can simply store the amounts of visits a cell has. We can calculate the maximum possible amount of non-loop visits by determining that:

4-directions(NSEW) * 2-states(default-inverted)
=
8 possible directions

So if we pass over the same cell more than 8 times we can determine that we are in a infinite loop. In the case that there were more directions or states we simply would have to change the operation.

Shortest possible path

To find the shortest possible path we explore the map starting from the end point until we find the starting point, whenever we do a movement we increase the counter, then when we find the starting point we simply return the number as it is at the time, this number represents the minimum length our path can possibly have, with the logic:

#	#	#	#	#	#	#	#	#	#	#	#	#
#			13			#	9	10	11	12	13	#
#		X13	12	13		#	8	9	10	11	12	#
#	13	12	11	12	13	#	7	8	9	10	11	#
#	12	11	10	11	12	#	6	7	8	9	10	#
#	11	10	9	10	11	#	5	#	#	#	#	#
#	#	9	8	#	#	#	4	3	2	3	4	#
#	9	8	7	6	5	4	3	2	1	2	3	#
#	10	#	#	5	4	3	2	1	$0	1	2	#
#	9	8	7	6	#	4	3	2	1	2	3	#
#	10	9	8	7	#		4	3	2	3	4	#
#	#	#	#	#	#	#	#	#	#	#	#	#

To build this logic into Java we will use two lists of cell and use a neighbour based system where we store the cells that have already had their value assigned and we store their non-assigned neighbours, then every time we assign a neighbours value we

Pau Bestard Satorra
29/2/2020
1st DAW - Programming

remove that neighbour from the neighbour list and added to the closed one. Until we find the starting point.

The flow of assigning Heuristics
The flow of obtaining Neighbours