**Jasons mapping instructions**

theres just 1 super tedious step hardcoding the map

the important part is this map as in the gameboard which u can technically hardcode

well basically **for every edge from A to B u also store what left right and back** is

no need to orientation, as you u just store a different edge for the two directions

YOU NEED A STRUCT FOR YOUR EDGES AND VERTEX’S but basically u write a function **addEdge** which can take in:

**To = vertex it goes to  
from = vertex it comes from  
left = edge if you turn left  
right = edge if u turn right  
back = edge if u go back**

and then **store the edge**, and you get these values from literally looking at the map and doing it by hand.

making the map, then becomes, super tedious also super ez to get wrong and hard to debug

------

ill tell u what im doing im implementing it as a linked list but using arrays instead of pointers

you can think of the index of the array as the ID of the thing so for example:

**Node[MAX\_Node]**

**struct node { int value; int next; }**

and then lets say the ID of the head of the list is 1 therefore first node is node[1]

now node[1] is a node lets say

it has the value 5 and next = 4 then u can look up node[4] and repeat till next = INVALID  
**#define INVALID -1**

not null because it's an int, not a pointer

**node.next = 4**

node.next sorts the ID of the next thing in the 'list'. The order of the list doesn't have to be

1 2 3 4 5 67

like think of it this way, u have an array 🡪 the array has values 🡪 and you store information about how it's joined and vertex[4].next could be 90999 the bad thing about storing them as arrays is you need to specify the size of the array

if you didnt know the maximum size then it's better to use pointers because whenevr you need a new node u can just make one whereas in this case you only have 54 nodes if u have a 55th ur screwed so thats one time you would prefer to use pointers

ok so with pointers, your strcut would look like:

**struct { int value; node\* next; } \*node**

you create it with

**(\*node) malloc(sizeof(\*node));**

now to go thru the list u go

**node = node->next**

so when you create the node it's also good (always do this basically) to set **node->next = NULL**

and theninserting nodes you just make a new node**,** and with your previous nodego **node->next = newNode**

and the most basic thing you can do with this list

is sort through it with

**while (node != NULL) { node = node->next; }**which sayswhile im not at the end of the list**,** set my current node to the next node

-----------------------------------------------------------------------------------------------------------------------------------------------------------

ok so basically what i told you to do is translate the edges, when u do it by hand, to code, so u know how the pathing works.

u read in L, that means go left

you also know the map always looks the same therefore when the path says L you know EXACTLY what that means  
you know which path it is, which edge, so technically when you build your board you can store

**'if im at this edge'**

**'and it tells me to go left'**

**'go to this edge'**

by just storing what left is from that edge so like

**struct edge { int left; }**

then when u do the path finding u can just go **edge.left** and be on the next edge

the super annoying part about this is you need to actually type out every single edge

so basically your AI needs to know what the map 'looks' like aka you need to store that

the good news is because the map structure always looks the same you can hardcode it addEdge(vertex1, vertex 2, leftedge, rightEdge, backEdge) 72 times

so when you're at a vertex, and i tell you to go left, how do you figure out which path to take

--------

In the edge, store where you came from + where youre going to. Once you know where you came from

-----

E.g. when you want to go to EDGE [15 – 9]

-----

Label each region according to the discipline arrays

Write out vertexes

Write out regions (+ what vertices inside)

e.g. if AI\_ENEMY wanted to go 1 🡪 4 🡪 9 🡪 10 🡪 5 🡪 2, from would be 5, to would 2, next would be none.

Have edges array

Edge e;

Ed.to = 1

E.from = 0

Edge[k++;] (k = how many edges I have currently)