

Created: May 2nd, 2023 11:28 PM

Tags:

Canvas: [Agent](#), [Environment](#)

## To do

- Add `goal_test()`
- Need a `set_goal()` function for the Environment.
  - It will be in `generate()`
- Have an `information()` function that can run anytime to print current information
- Complete `Environment` class's functions
  - `__init__()`
  - `generate()`
  - `percept()`
  - `update()`
  - `find_positions()`
  - `visualize()`
  - `transition_model()`
  - `goal_test()`
  - `place_agent_and_goal()`
  - `place_agent()`
  - `place_goal()`
  - `h_distance()`
- Make sure Agent can run without coding AI part
  1. `Initiate variables`
  2. `Place min agents with place_agents(percept)`
  3. `Save data with update_data()`
  4. `run()`
    - 1.

## General Notes

By the time I get to `update()`, I should have just one agent to worry about. I should choose the best agent based on `cost()`. But also, I want to keep track of the agents and slowly eliminate them as the code runs on and `transistion_model` slowly finds the illegal ones.

Calculate cost using Hermitian

The agent can destroy walls at a cost. Remember the wall is destroyed, and it would be destroyed physically so might as well update the Environment class's state as well.

For dead ends you could go back in your history in the DataFrame and backtrack to when there was more than 2 zeros. More than 2 zeros because that's where another option is.

For the dead end you just went down, you can create a virtual wall, or an X to not go down that path.

After exhausting all your path options, you can go back down the dead ends and destroy walls. So after trying all the `move()` you can then rely on `DESTROY()`

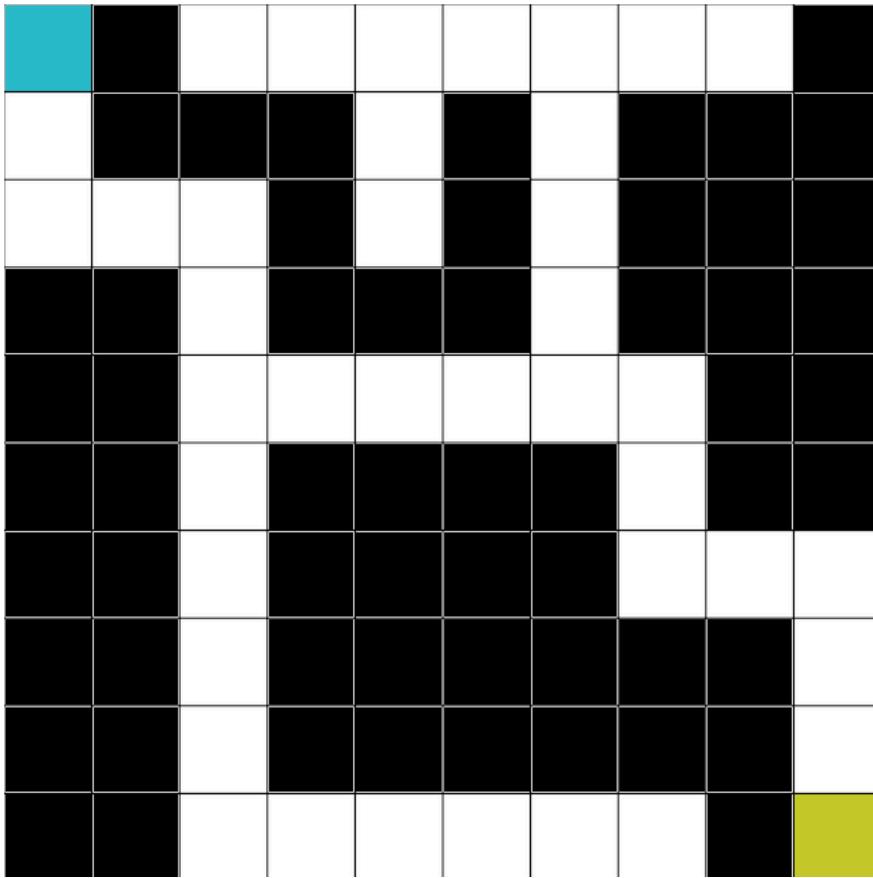
At the start there are multiple possible positions and since there are multiple possible positions we can slowly eliminate them throughout time.

## Things postponing

- `self.generate()`
- `self.valid_map(initial_state)`

## Journal

The map I'm planning to use



The blue is the starting point, and the yellow is the goal

Here is a the matrix in [Excel](#)

2	1	0	0	0	0	0	0	0	1
0	1	1	1	0	1	0	1	1	1
0	0	0	1	0	1	0	1	1	1
1	1	0	1	1	1	0	1	1	1
1	1	0	0	0	0	0	0	1	1
1	1	0	1	1	1	1	0	1	1
1	1	0	1	1	1	1	1	0	0
1	1	0	1	1	1	1	1	1	0
1	1	0	0	0	0	0	0	1	3

As a matrix:

```
initial_state = [
[2, 1, 0, 0, 0, 0, 0, 0, 0, 1],
[0, 1, 1, 1, 0, 1, 0, 1, 1, 1],
```

```
[0, 0, 0, 1, 0, 1, 0, 1, 1, 1],  
[1, 1, 0, 1, 1, 1, 0, 1, 1, 1],  
[1, 1, 0, 0, 0, 0, 0, 0, 1, 1],  
[1, 1, 0, 1, 1, 1, 1, 0, 1, 1],  
[1, 1, 0, 1, 1, 1, 1, 0, 0, 0],  
[1, 1, 0, 1, 1, 1, 1, 1, 1, 0],  
[1, 1, 0, 1, 1, 1, 1, 1, 1, 0],  
[1, 1, 0, 0, 0, 0, 0, 0, 1, 3]  
]
```

Okay I had this plan where the environment is created first, and then the information is sent to the agent

```
environment=Environment(initial_state)  
percept=environment.percept()  
agent=Agent(initial_state,percept)
```

But I think it'll be better if the environment will be inside the Agent because I'm not sure how I'll be referring to an object outside the class

Wait no it makes more sense to have the Agent inside the Environment. Logically you can have multiple agents in a environment, but not the other way around.

Use np.array for your matrix so you can use a simple tuple for your matrix's position instead of `matrix[position[0],position[1]]`

```
matrix = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])  
position = (0, 0)  
element = matrix[position]  
  
position=[0,0]  
value=matrix[position[0]][position[1]]
```

Maybe not since

```
if initial_state==None:
```

with

```
type(initial_state)=Array of int32  
  
2 1 0 0 0 0 0 0 0 1  
0 1 1 1 0 1 0 1 1 1  
0 0 0 1 0 1 0 1 1 1  
1 1 0 1 1 1 0 1 1 1  
1 1 0 0 0 0 0 0 1 1  
1 1 0 1 1 1 1 0 1 1  
1 1 0 1 1 1 1 0 0 0  
1 1 0 1 1 1 1 1 1 0  
1 1 0 1 1 1 1 1 1 0  
1 1 0 0 0 0 0 0 1 3
```

Just coded the `find_positions()` and it went pretty fast. Since I have 12 minutes left on my pomodoro I should set up GitHub

Okay it's [here](#)

Going to start working on the `percept()` function

Change `def find_position(state,initial=False)` to `def find_position(state,first=False)` to indicate finding the first mini agent since you can use finding the first mini agent multiple times throughout this code

```
Added a required state parameter to percept()
```

Got to figure out the math needed to find

```
[i-1,j]
[i,j-1] [i,j] [i,j+1]
[i+1,j]
```

Okay well the math seems pretty simple. If it's out of range (out of matrix), let's consider that as a wall.

Changing this from

```
initial_state = [
[2, 1, 0, 0, 0, 0, 0, 0, 0, 1],
[0, 1, 1, 1, 0, 1, 0, 1, 1, 1],
[0, 0, 0, 1, 0, 1, 0, 1, 1, 1],
[1, 1, 0, 1, 1, 1, 0, 1, 1, 1],
[1, 1, 0, 0, 0, 0, 0, 0, 1, 1],
[1, 1, 0, 1, 1, 1, 0, 1, 1, 1],
[1, 1, 0, 1, 1, 1, 0, 0, 0, 0],
[1, 1, 0, 1, 1, 1, 1, 0, 1, 0],
[1, 1, 0, 1, 1, 1, 1, 1, 1, 0],
[1, 1, 0, 0, 0, 0, 0, 0, 1, 3]
]
```

to

```
initial_state = [
[0, 1, 0, 0, 0, 0, 0, 0, 0, 1],
[0, 1, 1, 1, 0, 1, 0, 1, 1, 1],
[0, 0, 0, 1, 2, 1, 0, 1, 1, 1],
[1, 1, 0, 1, 1, 1, 0, 1, 1, 1],
[1, 1, 0, 0, 0, 0, 0, 1, 1, 1],
[1, 1, 0, 1, 1, 1, 0, 1, 1, 1],
[1, 1, 0, 1, 1, 1, 0, 0, 0, 0],
[1, 1, 0, 1, 1, 1, 1, 0, 1, 0],
[1, 1, 0, 1, 1, 1, 1, 1, 1, 0],
[1, 1, 0, 0, 0, 0, 0, 0, 1, 3]
]
```

for testing purposes, remember to change it back

This the code I came up with

```
percept+=str(state[position[0]-1][position[1]])
percept+=str(state[position[0]][position[1]]+1)
percept+=str(state[position[0]][position[1]]-1)
percept+=str(state[position[0]+1][position[1]])

print(percept)
```

It results in 0311 when it should be 0111 . I have no idea where the 3 came from. I gotta sleep anyways

Oh it was some silly error

```
print(position)
percept+=str(state[position[0]-1][position[1]])
percept+=str(state[position[0]][position[1]]+1)
percept+=str(state[position[0]][position[1]]-1)
percept+=str(state[position[0]+1][position[1]])

print(percept)
```

It works now: 0111  
Just got to add a

try:

with an out of range error exception check, and if it gets the error then we say the percept is next time I work on the code

Add a None to account for array type

```
def __init__(self, initial_state=None, agent_name="Pathfinder"):
```

Also see if a function like .isempty() exists for an array

💡 Instead of finding the mini agent's position every time, we could update a self.current\_position variable by adding a value to the i and j.

Making the borders of the matrix 5 for testing IndexError

```
initial_state = [
    [5, 5, 5, 5, 5, 5, 5, 5, 5, 5],
    [5, 1, 1, 1, 0, 1, 0, 1, 1, 5],
    [5, 0, 0, 1, 2, 1, 0, 1, 1, 5],
    [5, 1, 0, 1, 1, 1, 0, 1, 1, 5],
    [5, 1, 0, 0, 0, 0, 0, 0, 1, 5],
    [5, 1, 0, 1, 1, 1, 0, 1, 1, 5],
    [5, 1, 0, 1, 1, 1, 1, 0, 1, 5],
    [5, 1, 0, 1, 1, 1, 1, 0, 0, 5],
    [5, 1, 0, 1, 1, 1, 1, 1, 1, 5],
    [5, 1, 0, 1, 1, 1, 1, 1, 1, 5],
    [5, 5, 5, 5, 5, 5, 5, 5, 5]
]
```

No work

```
#N
try:
    percept+=str(state[position[0]-1][position[1]])
except IndexError or position[0]==0:
    percept+="1"
#E
try:
    percept+=str(state[position[0]][position[1]+1])
except IndexError or position[1]==len(state[0]):
    percept+="1"
#W
try:
    percept+=str(state[position[0]][position[1]-1])
except IndexError or position[1]==0:
    percept+="1"
#S
try:
    percept+=str(state[position[0]+1][position[1]])
except IndexError or position[0]==len(state):
    percept+="1"
```

Works but does not look as good

```
#N
[0,10]
try:
    if position[0]!=0:
        percept+=str(state[position[0]-1][position[1]])
    else:
        percept+="1"
```

```

except IndexError:
    percept+="1"
#E
try:
    if position[1]!=len(state[0]):
        percept+=str(state[position[0]][position[1]+1])
    else:
        percept+="1"
except IndexError:
    percept+="1"
#W
try:
    if position[1]!=0:
        percept+=str(state[position[0]][position[1]-1])
    else:
        percept+="1"
except IndexError:
    percept+="1"
#S
try:
    if position[0]!=len(state):
        percept+=str(state[position[0]+1][position[1]])
    else:
        percept+="1"
except IndexError:
    percept+="1"

print(percept)

return 1011

```

I'm pretty sure the return does not have to be an `int`. There shouldn't be situation where we would need the `percept` to be an `int`. It would more convenient as a `str`

```
return percept
```

When giving the `initial_state` to the Agent, we got to remove the `2` so it does not where it is initially. That's why we are giving the `percept`

```

#Make sure agent does not know it's position
for i in range(len(initial_state)):
    for j in range(len(initial_state)):
        if initial_state[i][j]==2:
            initial_state[i][j]=0

```

Can I code `self.generate()` in under an hour?

Each map will be 10x10

 At one point we will probably need a Heuristic function to find the direct path

I can have 2x2 pieces placed in a 16x16 map

Have rule book that checks if a piece is allowed to be next to another piece

Also, have a smaller matrix with just the name of the pieces, and then later substitute the matrixes

8x8 matrix needed to represent a 16x16 matrix with 2x2pieces

Can start 8x8 matrix with random pieces, and then go through all of it replacing pieces that are invalid. Though the problem will be replacing one piece might break a piece we just fixed

We should have a dictionary of each piece that says which pieces are allowed next to each other. We check what piece to replacing it with a random piece that both are a union

We will be more relaxed with the validation of the piece state. It will be okay for the agent to fail. If the agent tries 1000 moves, and does not get out, then the map is probably impossible. This allows us to use the feature where the agent can break a wall if we want to. Also, the randomness of the map will most likely have an opening most times. We will still use the rulebook for checking.

## Changing

```
piece_state=[[random.randint(1, 7) for i in range(8)] for i in range(8)]
```

to

```
piece_state=[[random.randint(1, 3) for i in range(8)] for i in range(8)]
```

```

[[3, 2, 2, 2, 3, 2, 3, 3],
 [2, 2, 1, 3, 2, 2, 2, 3],
 [3, 3, 1, 1, 2, 3, 2, 1],
 [3, 1, 3, 1, 2, 3, 2, 1],
 [1, 3, 2, 1, 1, 1, 3, 2],
 [1, 2, 2, 2, 2, 3, 3, 2],
 [1, 3, 2, 1, 3, 2, 2, 3],
 [2, 3, 3, 3, 2, 3, 2, 2]]
rulebook={0:[1,2,3],1:[1,2,3],2:[1,3],3:[2]}
[[3, 2, 2, 2, 3, 2, 3, 3],
 [2, 3, 0, 3, 2, 1, 2, 0],
 [3, 2, 3, 2, 1, 1, 1, 1],
 [3, 0, 2, 1, 3, 2, 1, 2],
 [1, 1, 1, 1, 2, 1, 2, 1],
 [1, 3, 2, 1, 1, 2, 1, 3],
 [1, 2, 1, 2, 3, 0, 2, 0],
 [2, 3, 2, 3, 2, 1, 1, 3]]

```

I got to make the rules for the rulebook. Actually, first I got to save the version of the code onto GitHub.

Okay I set up github and made a new branch called "Creating-rules-in-rulebook". Now will start thinking have the pieces will fit together

Piece 0 is an empty piece, so it can work for any piece.

```
0:[0,1,2,3,4,5,6,7]
```

To remind myself, the code checks piece A and piece B. Piece B is down one and to the left one. Then piece C one under piece A is replaced with a piece that works for A and B.

Assuming

```
rulebook=[0:[0,1,2,3,4,5,6,7],1:[3],2:[3],3:[],4:[],5:[],6:[],7:[]]
```

and we have the random state

```
[[], [7, 1, 7, 7, 7, 7, 7, 7], [2, 7, 7, 7, 7, 7, 7, 7], [7, 7, 7, 7, 7, 7, 7, 7, 7],  
[7, 7, 7, 7, 7, 7, 7, 7], [7, 7, 7, 7, 7, 7, 7, 7], [7, 7, 7, 7, 7, 7, 7, 7, 7]]
```

The result is

`[[7, 1, 7, 7, 7, 7, 7], [2, 3, 0, 0, 0, 0, 0], [7, 0, 7, 0, 6, 0, 1, 3], [7, 0, 0, 0, 0, 6, 0, 0], [7, 0, 6, 0, 7, 0, 7, 0], [7, 0, 0, 2, 0, 2, 0, 7], [7, 0, 1, 3, 0, 3, 0, 0], [7, 0, 3, 0, 3, 0, 0, 1]]`

To test, I made sure that in the rulebook, piece 1 and 2 intersection is 3

1:[3], 2:[3]

Then I set up a situation in the first two columns where 2 is down one and to the left one of 1

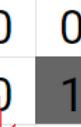
`[7, 1, 7, 7, 7, 7, 7], [2, 7, 7, 7, 7, 7]`

The result should be 3 "between" 1 and 2. Down one from 1

`[7, 1, 7, 7, 7, 7, 7], [2, 3, 0, 0, 0, 0, 0]`

Now I got to make the rule book. I'll use [Excel](#) to visualize the pieces.

Piece 1	0	0
	0	1
Piece 2	1	0
	0	0
Piece 3	1	1
	1	1
Piece 4	0	0
	1	1
Piece 5	1	1
	0	0
Piece 6	0	1
	0	0
Piece 7	0	0
	1	0

Let's define what it means when two pieces "work" together. A piece works with another piece if there is a path that transition between them. This path has to be present when the piece is above, and when on the side. For example,  works with  because you can enter  from above or from the left:

	0	0
	0	1
0	0	0
0	0	0
0	1	1

does not work with

because you can't enter from above

	0	0
	X	1
0	0	0
1	1	0

Let's start with all the combinations of

## Piece 1

```
piece1=[[0,0],  
        [0,1]]
```

	0	0
	0	1
0	0	
0	1	

Pieces that work are

1: [1, 4, 6, 7]

### Pieces that work >

1

	0	0
	0	1
0	0	0
0	1	0

Not 2

	0	0
	0	1
0	0	1
0	1	0

4

	0	0
	0	1
0	0	0
0	1	1

6

A	B	C	D
1		0 0	0 0
2		0 1	
3	0 0	0 1	
4	0 1	0 0	
5			
6			
7			

A	B	C	D
1		0 0	0 0
2		0 1	
3	0 0	0 0	0 0
4	0 1	1 0	0

## Piece 2

A	B	C	D
1		1 0	0 0
2	1 0		
3	0 0		

Pieces that work are

[2: \[1, 2, 4, 6, 7\]](#)

## Piece 3

Piece 3 might be used to spice up the map by placing it randomly. It can be used to up the difficulty

A	B	C	D
1		1 1	1 1
2	1 1		
3	1 1		

Pieces that work are

[3: \[\]](#)

## Piece 4

A	B	C	D
1		0 0	0 0
2		1 1	
3	0 0		
4	1 1		

Pieces that work are

4: []

Okay I did 2 hours for today. I'm not sure if I should add a code where it considers pieces like piece 4. It is a valid piece that works from the side, but not above. I guess it will be placed in randomly anyways from the code. But then the rulebook will replace it.

Continuing today for 4 hours

### Piece 5

		1	1
		0	0
1	1		
0	0		

Pieces that work are

5: [1, 2, 6]

### Piece 6

		0	1
		0	0
0	1		
0	0		

Pieces that work are

6: [1, 2, 6]

### Piece 7

		0	0
		1	0
0	0		
1	0		

Pieces that work are

7: [1, 2, 4, 7]

Okay I'm done figuring what pieces work with what, now will add the pieces' rules into the rulebook

Actually never mind, it seems I've done a mistake. In the code piece 7 is

```
piece7=[[1,0],  
        [1,0]]
```

but in Excel I've written it as

Piece 7	0	0
	1	0

```
piece7=[[0,0],  
        [1,0]]
```

But this is fine, because I realized I should re-customize my pieces. I should have it where most of these pieces fit together.

For example, pieces like piece5 will never work because the top is blocked off in all cases

	0	0
	1	0
0	0	1 1
1	0	0 0

Piece 5

but instead of a wall, we had a single corner, this would work for many more cases

	0	0
	1	0
0	0	1 0
1	0	0 0

New piece

So I think I'll re-design the pieces so they have a larger chance of working with each other.

This will also save memory and ram because now there won't be pieces stored and processed that are unusable

Okay these will be the new pieces

Piece 1	1	0
	0	0
Piece 2	0	1
	0	0
Piece 3	0	0
	1	0
Piece 4	0	0
	0	1
Piece 5	0	1
	0	1
Piece 6	0	0
	1	1
Piece 7	1	1
	1	1

I'm going to keep two wall pieces: `piece5` and `piece6`. The reason is to give the possibility of dead ends around the center of the map instead of just the edges (because the rulebook doesn't check the edges). Also I'll have `piece7` be all 1, but not sure if I'll use it.

New rulebook:

### Piece 1

	1	0
	0	0
1	0	
0	0	

1: [1, 2, 3, 4, 5, 6]

### Piece 2

	0	1
	0	0
0	1	
0	0	

2: [1, 2, 4, 5]

### Piece 3

		0 0
	1 0	
0 0		
1 0		

3: [1, 3, 4, 6]

#### Piece 4

		0 0
	0 1	
0 0		
0 1		

4: [2, 3, 4, 5, 6]

#### Piece 5

		0 1
	0 1	
0 1		
0 1		

5: []

#### Piece 6

		0 0
	1 1	
0 0		
1 1		

6: []

#### Piece 7

		1 1
	1 1	
1 1		
1 1		

7: []

I've realized that the walls are not really dead ends because there is still an opening.

For example with piece4 with piece5 (wall piece)

	0	0
	0	1
0	0	1
0	1	0

and piece4 with piece6 (other wall piece)

	0	0
	0	1
0	0	0
0	1	1

But at the same time they are a dead end, because piece5 / piece6 might work with other pieces, but other pieces won't work with piece5 / piece6

	0	1
	0	1
0	1	
0	1	

So this is good. But we will see how it really turns out once we test it in practice

Okay now for the complete rulebook

```
rulebook={0:[0,1,2,3,4,5,6,7],1:[1,2,3,4,5,6],2:[1,2,4,5],3:[1,3,4,6],4:[2,3,4,5,6],5:[],6:[],7:[]}
```

also updated the pieces

Okay now I need to expand the 8x8 piece\_state into a 16x16 state . I'm trying to remember what you ( Manuel) showed me

I tried looking for the papers you have gave me with the explanation until I realized it's at the bottom of a box under so much stuff in the basement, so I'll try to refigure this out from scratch.

What I remember is that we used range() in order to translate the 8x8 into a 16x16 so I'll start playing with that

Just created a [new Git repository](#) because I butchered [the last one](#) so badly

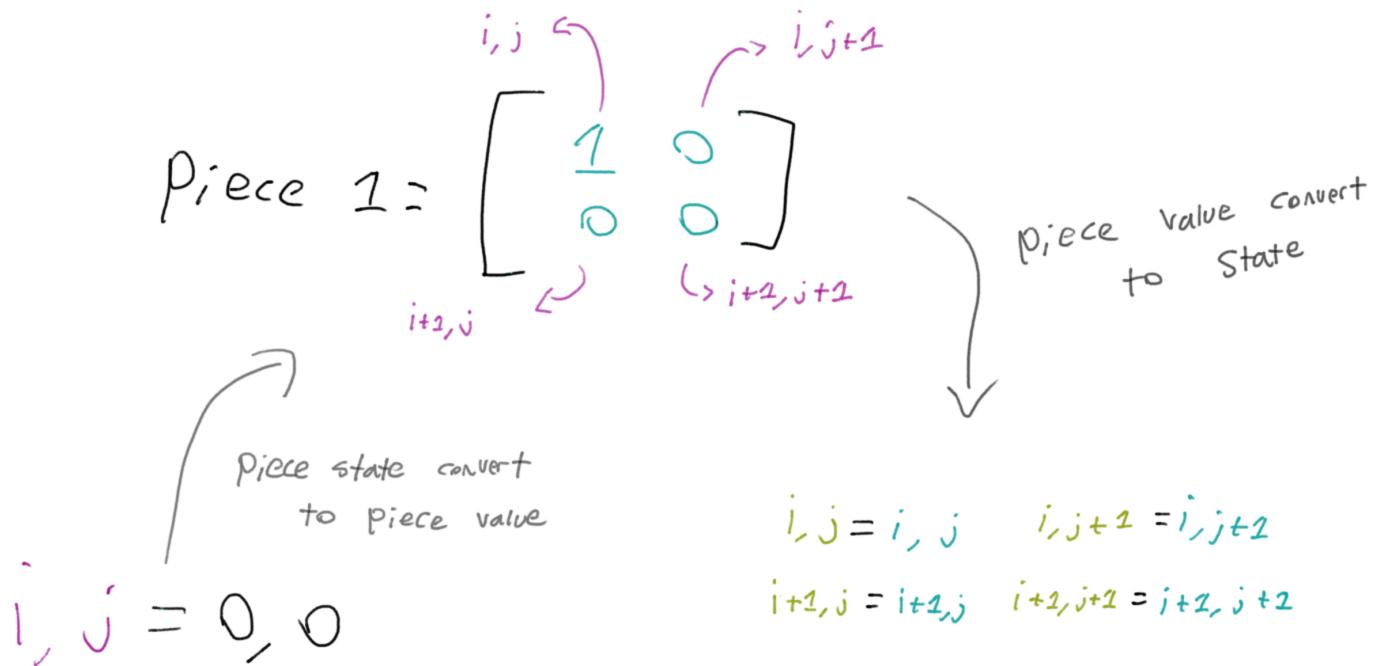
This is where I'm stuck so far because I know I need to have +2 somewhere to expand the code but not sure where or how

```
for i in range(len(piece_state)):
    for j in range(len(piece_state[0])):
        state[i][j]=piece_state[i][j]
```

Made a dictionary for the pieces to easily call them

```
pieces={0:piece0,1:piece1,2:piece2,3:piece3,4:piece4,5:piece5,6:piece6,7:piece7}
```

Trying to figure out how to convert them by noticing how to convert  $i, j$  of the piece\_state to the  $i, j$  of the state



Okay I got the code to print out the piece

```
pieces={0:piece0,1:piece1,2:piece2,3:piece3,4:piece4,5:piece5,6:piece6,7:piece7}
for i in range(len(piece_state)):
    for j in range(len(piece_state[0])):
        piece=pieces[piece_state[i][j]]
        print(piece)
```

Now the state values will be replaced with values of the piece. Now I just have to off-set  $i, j$  in order to replace the correct values in the 16x16 state

```
pieces={0:piece0,1:piece1,2:piece2,3:piece3,4:piece4,5:piece5,6:piece6,7:piece7}
for i in range(len(piece_state)):
    for j in range(len(piece_state[0])):
        # Get the piece
        piece=pieces[piece_state[i][j]]

        # Replace values in state with values of the piece
        state[i][j]=piece[i][j]
        state[i+1][j]=piece[i+1][j]
        state[i][j+1]=piece[i][j+1]
        state[i+1][j+1]=piece[i+1][j+1]
```

Fixed an error where I would go outside the piece range

```
# Replace values in state with values of the piece
state[i][j]=piece[0][0]
state[i+1][j]=piece[1][0]
state[i][j+1]=piece[0][1]
state[i+1][j+1]=piece[1][1]
```

Fixed error where  $i, j$  would go out of range of piece\_state. Fixed by moving  $i+=2$  to where `for i in range(len(piece_state))` loop is. And also

fixed by adding -2 to the length of the `piece_state`

```
print(piece_state)

pieces={0:piece0,1:piece1,2:piece2,3:piece3,4:piece4,5:piece5,6:piece6,7:piece7}
for i in range(len(piece_state)-2):
    if i!=0:
        i+=2
    for j in range(len(piece_state[0])-2):
        # Get the piece
        print(i,j)
        piece=pieces[piece_state[i][j]]

        #Off set i,j
        if i!=0:
            j+=2

        # Replace values in state with values of the piece
        state[i][j]=piece[0][0]
        state[i+1][j]=piece[1][0]
        state[i][j+1]=piece[0][1]
        state[i+1][j+1]=piece[1][1]
```

it resulted in this map:

which looks like this

There must have been an error somewhere because there are way too many zeros.

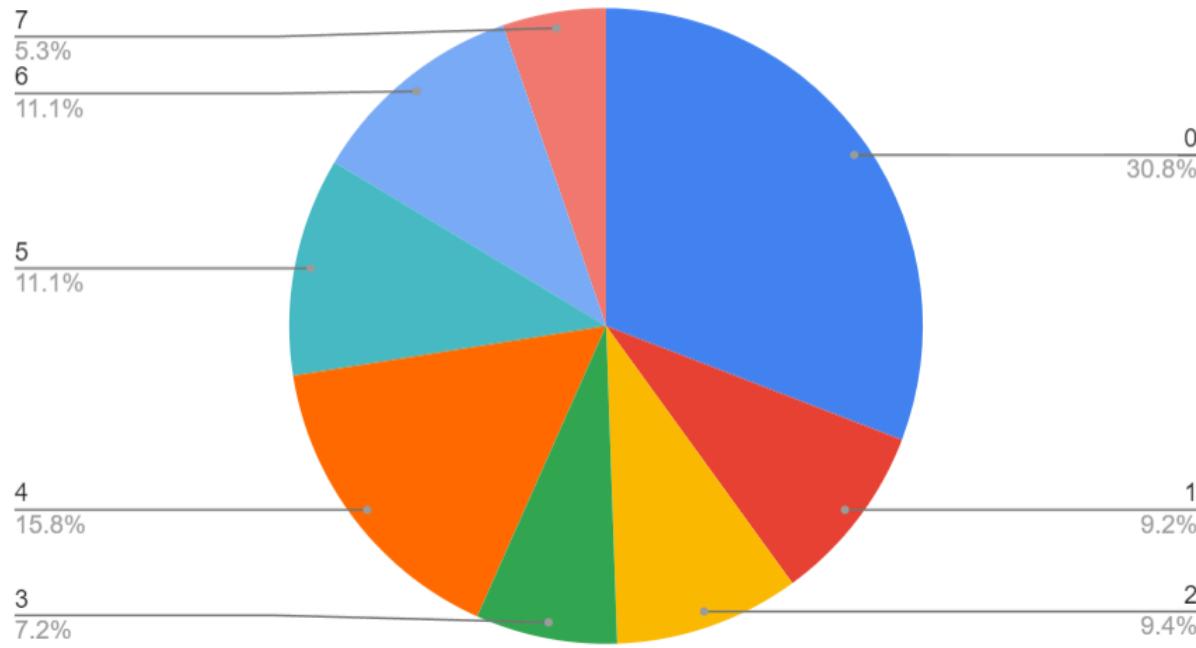
I think the error might be because of the 0 in the rulebook .

I'm going to sample the pieces, and then see how common each piece is

```
for j in range(len(piece_state[0])-2):
    # Get the piece
    piece=pieces[piece_state[i][j]]
    print(piece_state[i][j]) #Added this print
```

Using [Excel](#), I pasted the results of what pieces are being converted. I ran the code 10 times and gave the values. This is the pie chart

## Frequency vs. Piece



Yeah a third of the `piece_state` is the piece 0, so something is up. One theory is that the `check()` function is preferring `piece0` too often. This is because the situation can be like the butterfly effect. If one piece is 0, then the common intersection would be 0 or something like. I can see on the 16x16 map above that it starts off with walls, but as you reach the end, it just becomes empty space

My 4 hour timer went off, so that's all I'll do for today.

Starting on code once again. I'll make a new branch for testing errors. First I'll try to find the reason behind the zeros by playing around with the `i, j` indices in

```
for i in range(len(piece_state)-2):
    # Off set i
    if i!=0:
        i+=2
    for j in range(len(piece_state[0])-2):
        # Get the piece
        piece=pieces[piece_state[i][j]]
        print(piece_state[i][j])

        # Off set j
        if i!=0:
            j+=2

        # Replace values in state with values of the piece
        state[i][j]=piece[0][0]
        state[i+1][j]=piece[1][0]
        state[i][j+1]=piece[0][1]
        state[i+1][j+1]=piece[1][1]
```

and by playing around with the criteria.

I'll name the new branch called `zero-abundance-error-fix`, it's branching off the other branch `expanding-piece_state-to-state`

As a test to rule out the intersections being the error, I'll replace the rulebook from

```
rulebook={0:[0,1,2,3,4,5,6,7],1:[1,2,3,4,5,6],2:[1,2,4,5],3:[1,3,4,6],4:[2,3,4,5,6],5:[],6:[],7:[]}
```

to

```
rulebook={0:[1,2,3,4,5,6,7],1:[1,2,3,4,5,6,7],2:[1,2,3,4,5,6,7],3:[1,2,3,4,5,6,7],4:[1,2,3,4,5,6,7],5:[1,2,3,4,5,6,7],6:[1,2,3,4,5,6,7],7:[1,2,3,4,5,6,7]}
```

so all pieces interact with each other.

The resultant map is

1	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0
0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	1	1	1	0	0	0	0	0	0	0	0
0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0
0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	0
0	0	1	0	0	1	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

so nothing really has changed. The error is not related to the pieces intersecting

Actually let's focus on the how the `piece_state` is filtered after the `check()` function

```
print(piece_state, "\n\n\n\n")

# Checks validity of pieces using rulebook
for i in range(len(piece_state)-1):
    for j in range(1, len(piece_state[0])):
        piece_Q=check(piece_state[i][j], piece_state[i+1][j-1])
        piece_state[i+1][j]=piece_Q
print(piece_state)
```

if we kept the `rulebook` to the original rules

```
rulebook={0:[0,1,2,3,4,5,6,7],1:[1,2,3,4,5,6],2:[1,2,4,5],3:[1,3,4,6],4:[2,3,4,5,6],5:[],6:[],7:[]}
```

and if the original `piece_state` is

```
[[5, 3, 3, 3, 2, 6, 4, 6], [1, 6, 7, 7, 4, 3, 4, 3], [5, 1, 5, 7, 2, 6, 4, 4], [3, 5, 2, 6, 7, 6, 6, 7], [3, 1, 4, 4, 2, 6, 1, 2], [5, 3, 7, 1, 1, 5, 5, 4], [6, 7, 3, 4, 3, 2, 6, 7], [3, 7, 1, 6, 1, 4, 6, 4]]
```

the resultant `piece_state` would be

```
[[5, 3, 3, 3, 2, 6, 4, 6], [1, 1, 3, 4, 4, 0, 3, 0], [5, 0, 3, 4, 3, 4, 4, 3], [3, 4, 6, 0, 3, 4, 2, 1], [3, 3, 0, 6, 0, 5, 0, 3], [5, 0, 6, 0, 6, 0, 0, 4], [6, 0, 0, 3, 0, 6, 0, 3], [3, 1, 4, 4, 6, 0, 6, 0]]
```

There are some zero pieces, but only where needed.

But even if the 16x16 state did work perfectly, it wouldn't matter because there still would a crap ton of zeros. The `piece_state` from above would look like this

this sucks because as Manuel said everything is "small islands". And also, my entire point of making sure there is path seems to fail because the starting point is blocked.

I have to rethink the criteria

Okay this is the plan

1. Fix the code expansion without changing the pieces or rulebook. We won't care about the new criteria for now
  2. Think of a new criteria that follows the logic of pieces and a rulebook

Let's test the indices to make sure they work. To have a clear goal in mind I'll make the `piece_state` all 7 so the new 16x16 state should be only walls

The result right now with the broken code is

1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0
0	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0
0	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0
0	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0
0	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

I fixed a part of the index error by having the range go through all the indexes of the `piece_state`.

I removed the

```
if i!=0:  
    i+=2
```

and just have the range go through each element

```
for i in range(len(piece_state)):  
    for j in range(len(piece_state[0])):
```

so far results in

1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fixed the relation between the indices from piece\_state to state by multiplying the i,j by 2

```
state[i*2][j*2]=piece[0][0]
state[i*2+1][j*2]=piece[1][0]
state[i*2][j*2+1]=piece[0][1]
state[i*2+1][j*2+1]=piece[1][1]
```

The code seems to work now

Testing with another piece to make sure it work. Testing with `piece1`

the result is

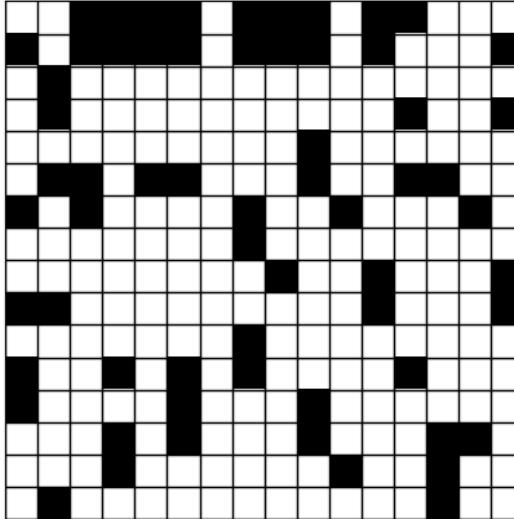
## Testing with piece5

results in

0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

So now that I fixed the expansion part of the code, I need to rethink the criteria. I'll push the branch first

Right now my criteria does not work, and a big error with it that the starting square is blocked



Creating a new branch called `improved-criteria`

I'm going to try to see how the map would look purely random. I'll remove `check()`

```
def check(piece_A,piece_B):  
    pieces=list(np.intersect1d(rulebook[piece_A],rulebook[piece_B]))  
    if pieces:  
        return random.choice(pieces)
```

```

else:
    return 0

```

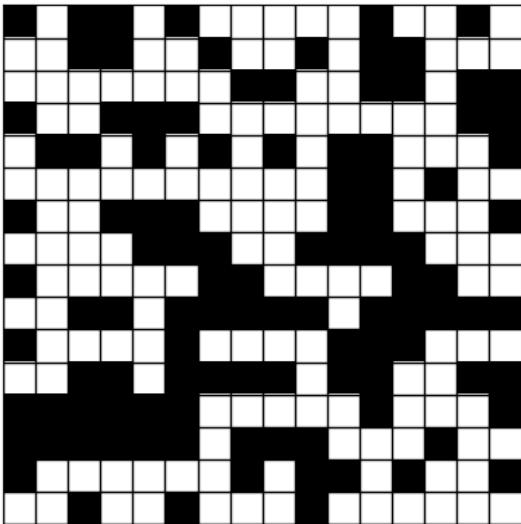
and the use of the rulebook

```

# Checks validity of pieces using rulebook
for i in range(len(piece_state)-1):
    for j in range(1,len(piece_state[0])):
        piece_Q=check(piece_state[i][j],piece_state[i+1][j-1])
        piece_state[i+1][j]=piece_Q
print(piece_state, "\n\n\n\n")

```

it looks like this



which is better than before, but could use more walls. And of course needs an algorithm to have a certain path

I have an idea. We can have a "mole" that starts at `i, j=0, 0` that goes for open squares next to it. It will move onto open squares, but also can destroy walls. If it's in a certain situation, it will destroy a wall. The chance of which wall to be destroyed can depend on the mole's position. For example if it's closer to the goal. We then can have multiple moles to make multiple paths, and some moles will make dead ends, but there will be at least one mole to make a path to a goal. The goal also can be placed by the mole instead of being always in one spot.

To start this, I would need to make pieces with more walls cause right now there is too much open space

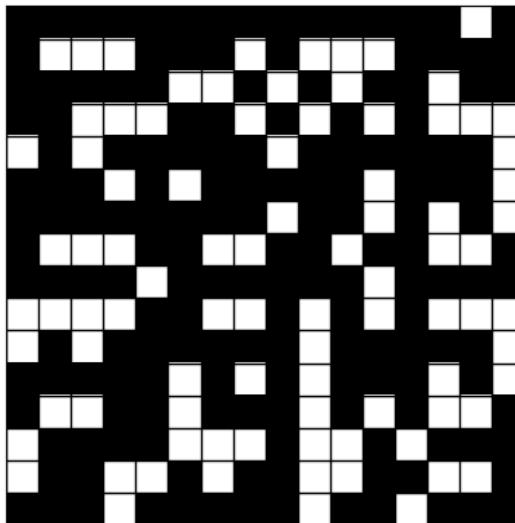
Another idea I could do is to instead split the `state` into larger pieces, like maybe 4 instead of 64. And I'll design each piece so no matter where they are positioned, they will always connect (though I can use the rulebook still). I'll actually try this idea first.

But I did a little bit of progress of my previous idea by changing the pieces to have more walls

```

piece0=[[0,0],
        [0,0]]
piece1=[[1,1],
        [0,0]]
piece2=[[1,0],
        [1,0]]
piece3=[[0,1],
        [1,0]]
piece4=[[1,0],
        [0,1]]
piece5=[[0,1],
        [1,1]]
piece6=[[1,1],
        [1,0]]
piece7=[[1,1],
        [1,1]]
```

which results in



So we could be using 4 pieces instead

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

This could be an example of 4 pieces

0	1	0	0	0	1	0	0	1	0	1	1	1	1	1	1
0	1	1	1	0	1	1	0	1	0	0	0	0	0	1	1
0	0	0	0	0	0	0	0	1	0	1	1	1	0	0	1
1	0	1	1	0	1	1	0	0	0	1	0	1	1	0	1
1	0	0	1	0	1	1	1	0	1	0	0	1	0	1	1
1	1	0	1	0	0	0	1	1	0	0	0	1	1	0	1
0	0	0	1	1	1	0	1	0	0	1	1	0	0	0	1
1	1	0	0	1	1	0	1	1	1	1	0	1	1	1	1
1	1	1	0	1	1	1	1	1	0	0	0	0	1	1	1
0	0	0	0	0	0	0	1	1	0	1	1	0	0	0	1
0	1	0	1	1	1	0	1	1	0	0	1	1	1	0	1
0	0	1	0	0	0	0	1	1	1	0	1	0	0	0	1
1	0	1	0	1	1	0	0	0	0	0	1	1	1	0	0
1	0	1	0	1	0	0	1	1	1	0	1	1	0	1	1
1	0	0	1	0	0	1	1	1	0	0	1	0	1	0	0
1	1	0	0	0	1	1	1	1	1	1	1	1	0	0	1

then I could have about 4 different pieces for each corner

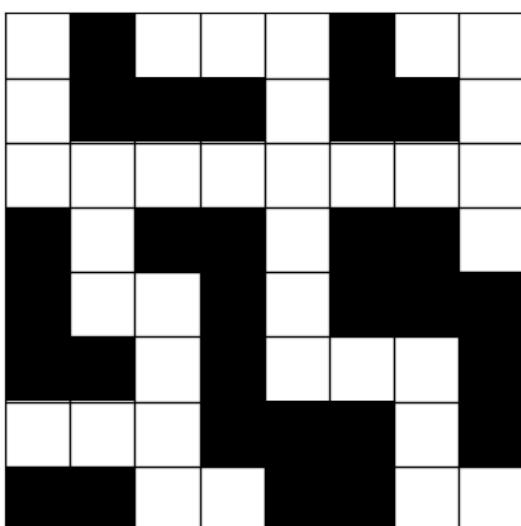
Okay it's the next day and I think I'll make 4 different pieces instead, but in order to have more possible maps, I'll need a little bit of math to know how variations to make for each side.

Knowing that each corner's variation can be combined with another corner's variation, then if I made 4 different variations for each side, the amount of different maps would be

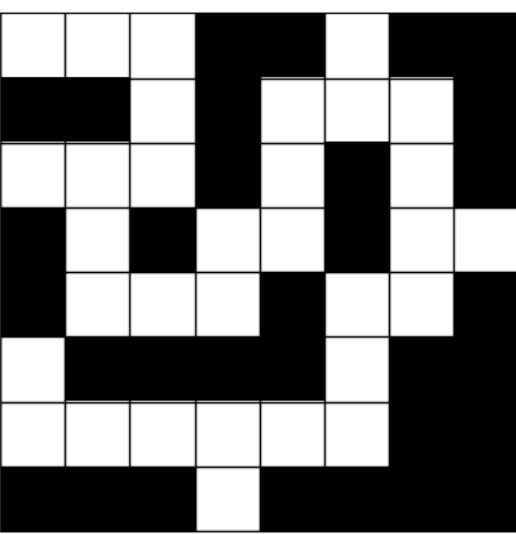
if I had 5 different variations for each side

so I'll start making the variations and see how many sides I feel to make

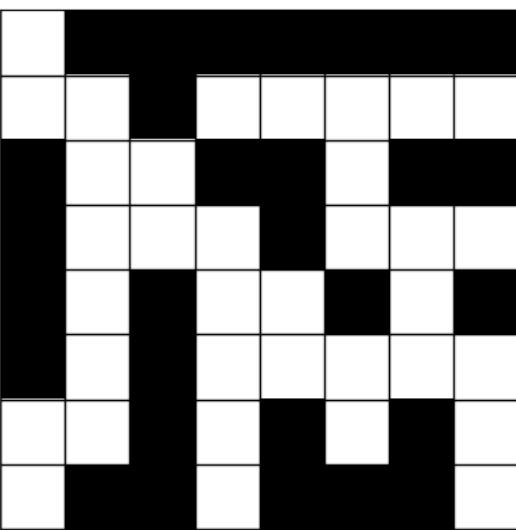
Variations for piece1



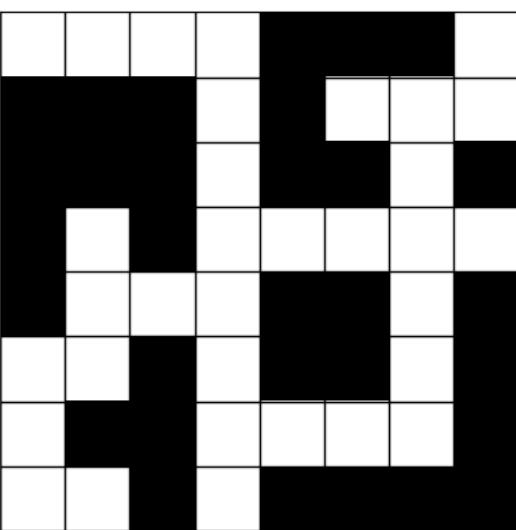
```
[[0, 1, 0, 0, 0, 1, 0, 0], [0, 1, 1, 1, 0, 1, 1, 0], [0, 0, 0, 0, 0, 0, 0, 0], [1, 0, 1, 1, 0, 1, 1, 0], [1, 0, 0, 1, 0, 1, 1, 0], [1, 1, 0, 1, 0, 0, 1, 1], [0, 0, 0, 1, 1, 1, 0, 1], [1, 1, 0, 0, 1, 1, 1, 0], [1, 1, 1, 0, 1, 1, 1, 0], [0, 0, 0, 0, 0, 0, 1, 1], [0, 1, 0, 1, 1, 1, 0, 1], [1, 0, 1, 0, 1, 1, 1, 0], [1, 0, 0, 1, 0, 0, 1, 1], [1, 1, 0, 0, 0, 1, 1, 1], [1, 1, 0, 0, 0, 0, 1, 1], [1, 1, 0, 0, 0, 0, 0, 1]]
```



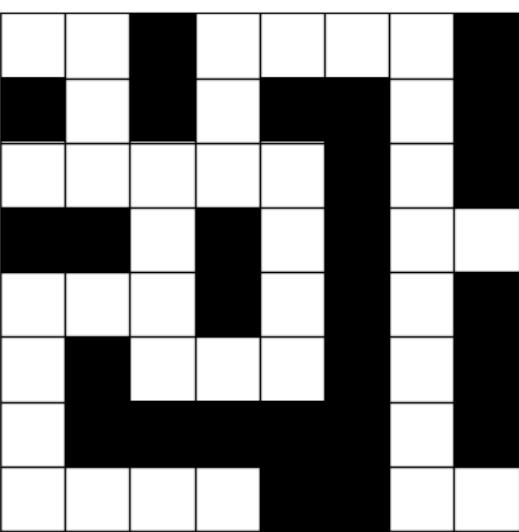
```
[[0, 0, 0, 1, 1, 0, 1, 1], [1, 1, 0, 1, 0, 0, 0, 1], [0, 0, 0, 1, 0, 1, 0, 1], [1, 0, 1, 0, 0, 1, 0, 0], [1, 0, 0, 0, 1, 0, 0, 1],  
[0, 1, 1, 1, 0, 1, 1], [0, 0, 0, 0, 0, 1, 1], [1, 1, 1, 0, 1, 1, 1, 1]]
```



```
[[0, 1, 1, 1, 1, 1, 1, 1], [0, 0, 1, 0, 0, 0, 0, 0], [1, 0, 0, 1, 1, 0, 1, 1], [1, 0, 0, 0, 1, 0, 0, 0], [1, 0, 1, 0, 0, 1, 0, 1],  
[1, 0, 1, 0, 0, 0, 0, 0], [0, 0, 1, 0, 1, 0, 1, 0], [0, 1, 1, 0, 1, 1, 1, 0]]
```

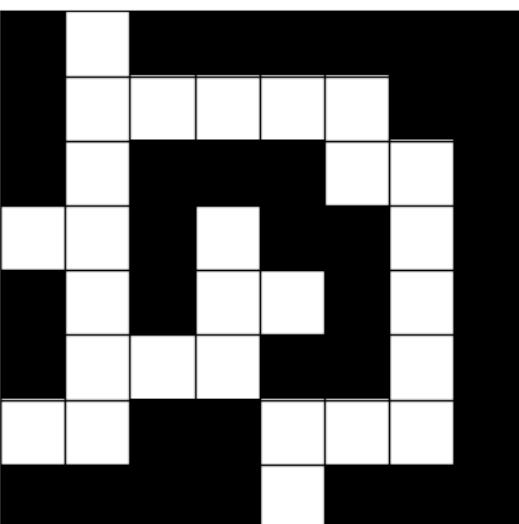


```
[[0, 0, 0, 0, 1, 1, 1, 0], [1, 1, 1, 0, 1, 0, 0, 0], [1, 1, 1, 0, 1, 1, 0, 1], [1, 0, 1, 0, 0, 0, 0, 0], [1, 0, 0, 0, 1, 1, 0, 1],  
[0, 0, 1, 0, 1, 1, 0, 1], [0, 1, 1, 0, 0, 0, 0, 1], [0, 0, 1, 0, 1, 1, 1, 1]]
```

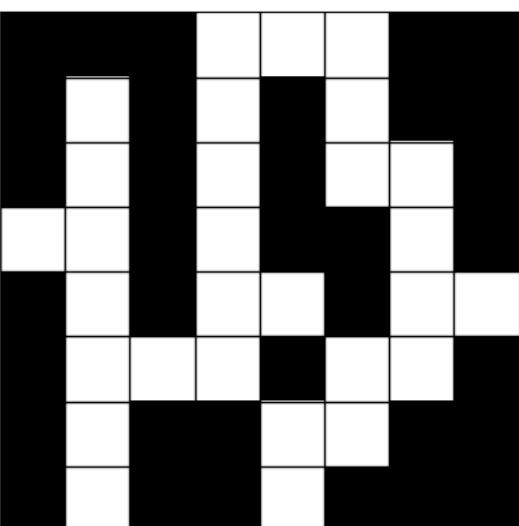


```
[[0, 0, 1, 0, 0, 0, 0, 1], [1, 0, 1, 0, 1, 1, 0, 1], [0, 0, 0, 0, 0, 1, 0, 1], [1, 1, 0, 1, 0, 1, 0, 0], [0, 0, 0, 1, 0, 1, 0, 1],  
[0, 1, 0, 0, 0, 1, 0, 1], [0, 1, 1, 1, 1, 0, 1], [0, 0, 0, 1, 1, 0, 0]]
```

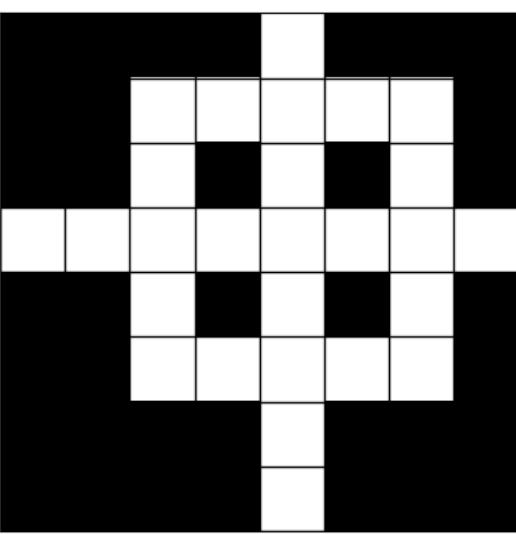
Variations for piece2



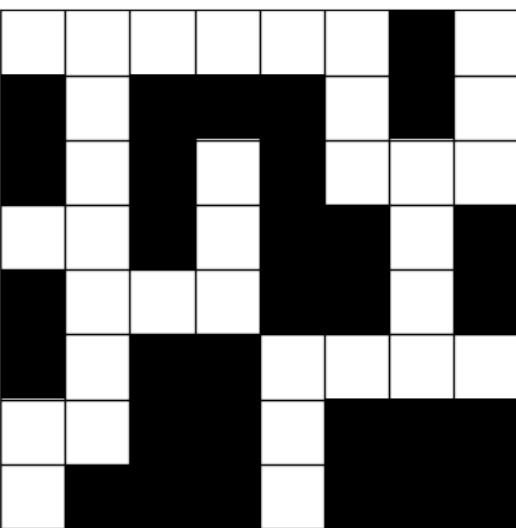
```
[[1, 0, 1, 1, 1, 1, 1, 1], [1, 0, 0, 0, 0, 0, 1, 1], [1, 0, 1, 1, 1, 0, 0, 1], [0, 0, 1, 0, 1, 1, 0, 1], [1, 0, 1, 0, 0, 1, 0, 1],  
[1, 0, 0, 0, 1, 1, 0, 1], [0, 0, 1, 1, 0, 0, 0, 1], [1, 1, 1, 1, 0, 1, 1, 1]]
```



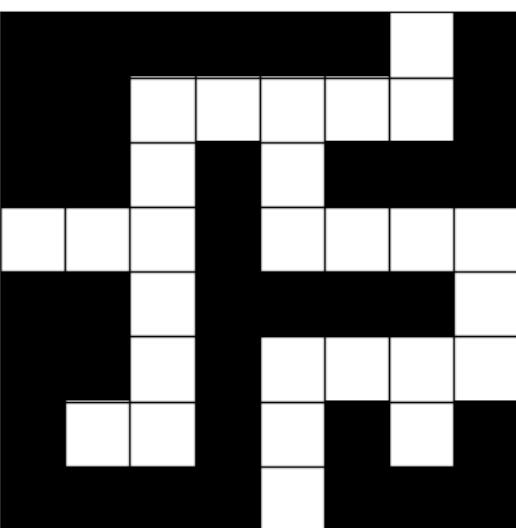
```
[[1, 1, 1, 0, 0, 0, 1, 1], [1, 0, 1, 0, 1, 0, 1, 1], [1, 0, 1, 0, 1, 0, 0, 1], [0, 0, 1, 0, 1, 1, 0, 1], [1, 0, 1, 0, 0, 1, 0, 0],  
[1, 0, 0, 0, 1, 0, 0, 1], [1, 0, 1, 1, 0, 0, 1, 1], [1, 0, 1, 1, 0, 1, 1, 1]]
```



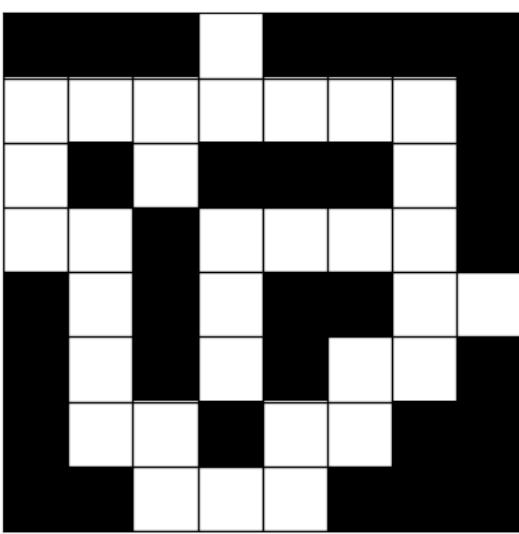
```
[[1, 1, 1, 1, 0, 1, 1, 1], [1, 1, 0, 0, 0, 0, 0, 1], [1, 1, 0, 1, 0, 1, 0, 1], [0, 0, 0, 0, 0, 0, 0, 0], [1, 1, 0, 1, 0, 1, 0, 1],  
[1, 1, 0, 0, 0, 0, 1], [1, 1, 1, 1, 0, 1, 1, 1], [1, 1, 1, 1, 0, 1, 1, 1]]
```



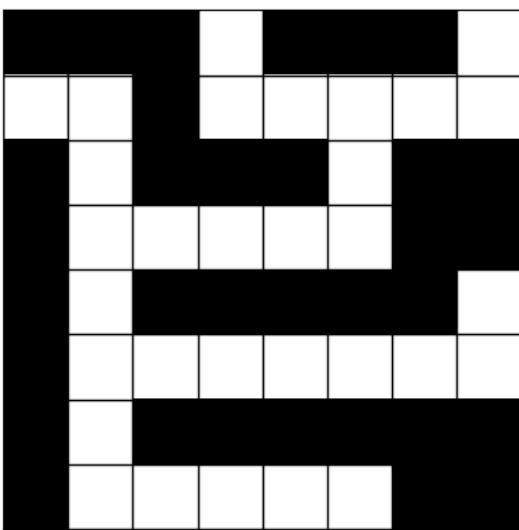
```
[[0, 0, 0, 0, 0, 1, 0], [1, 0, 1, 1, 1, 0, 1, 0], [1, 0, 1, 0, 1, 0, 0, 0], [0, 0, 1, 0, 1, 1, 0, 1], [1, 0, 0, 0, 1, 1, 0, 1],  
[1, 0, 1, 1, 0, 0, 0, 0], [0, 0, 1, 1, 0, 1, 1, 1], [0, 1, 1, 1, 0, 1, 1, 1]]
```



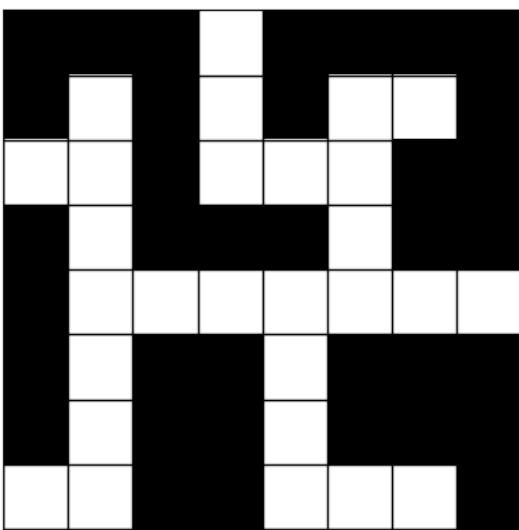
```
[[1, 1, 1, 1, 1, 1, 0, 1], [1, 1, 0, 0, 0, 0, 0, 1], [1, 1, 0, 1, 0, 1, 1, 1], [0, 0, 0, 1, 0, 0, 0, 0], [1, 1, 0, 1, 1, 1, 1, 0],  
[1, 1, 0, 1, 0, 0, 0, 0], [1, 0, 0, 1, 0, 1, 0, 1], [1, 1, 1, 1, 0, 1, 1, 1]]
```



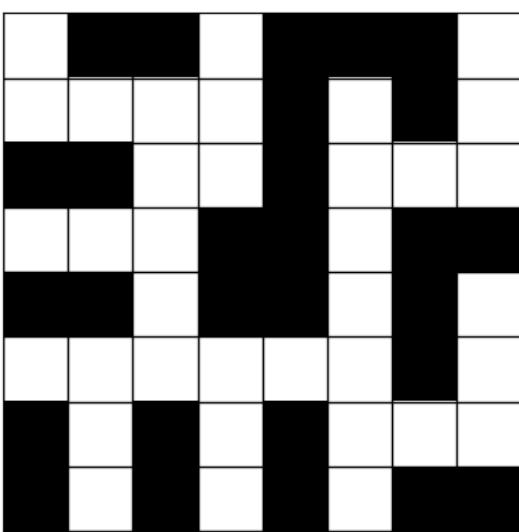
```
[[1, 1, 1, 0, 1, 1, 1, 1], [0, 0, 0, 0, 0, 0, 0, 1], [0, 1, 0, 1, 1, 1, 0, 1], [0, 0, 1, 0, 0, 0, 0, 1], [1, 0, 1, 0, 1, 1, 0, 0], [1, 0, 1, 0, 0, 0, 1], [1, 0, 0, 1, 0, 0, 1, 1], [1, 1, 0, 0, 0, 1, 1, 1]]
```



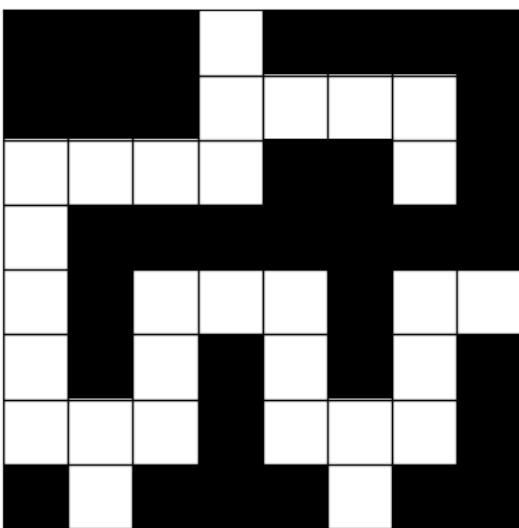
```
[[1, 1, 1, 0, 1, 1, 1, 0], [0, 0, 1, 0, 0, 0, 0, 0], [1, 0, 1, 1, 1, 0, 1, 1], [1, 0, 0, 0, 0, 0, 1, 1], [1, 0, 1, 1, 1, 1, 1, 0], [1, 0, 0, 0, 0, 0, 0, 0], [1, 0, 1, 1, 1, 1, 1, 1], [1, 0, 0, 0, 0, 0, 1, 1]]
```



```
[[1, 1, 1, 0, 1, 1, 1, 1], [1, 0, 1, 0, 1, 0, 0, 1], [0, 0, 1, 0, 0, 0, 1, 1], [1, 0, 1, 1, 1, 0, 1, 1], [1, 0, 0, 0, 0, 0, 0, 0], [1, 0, 1, 1, 0, 1, 1, 1], [1, 0, 1, 1, 0, 1, 1, 1], [0, 0, 1, 1, 0, 0, 0, 1]]
```

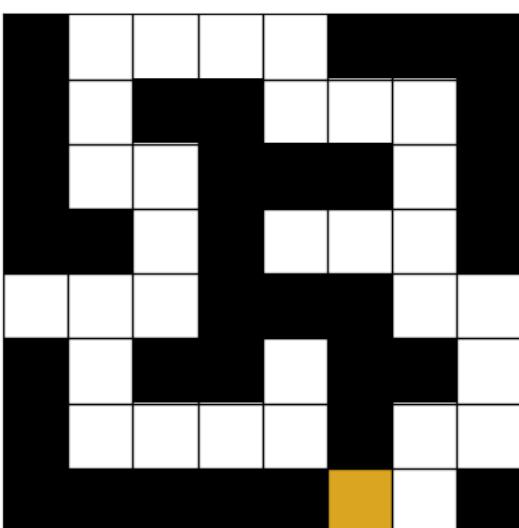


```
[[0, 1, 1, 0, 1, 1, 1, 0], [0, 0, 0, 0, 1, 0, 1, 0], [1, 1, 0, 0, 1, 0, 0, 0], [0, 0, 0, 1, 1, 0, 1, 1], [1, 1, 0, 1, 1, 0, 1, 0], [0, 0, 0, 0, 0, 1, 0], [1, 0, 1, 0, 1, 0, 0], [1, 0, 1, 0, 1, 0, 1, 1]]
```

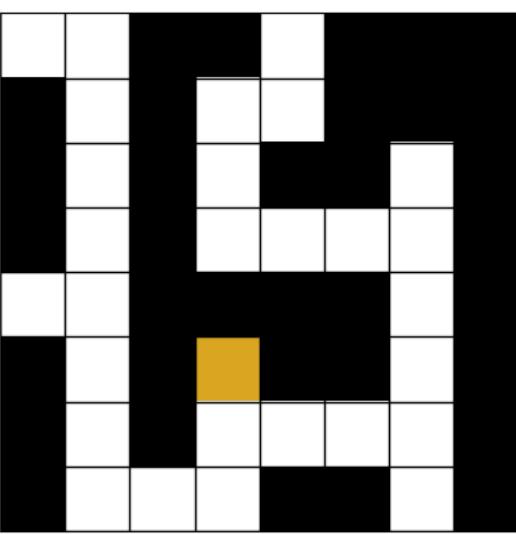


```
[[1, 1, 1, 0, 1, 1, 1, 1], [1, 1, 1, 0, 0, 0, 0, 1], [0, 0, 0, 0, 1, 1, 0, 1], [0, 1, 1, 1, 1, 1, 1, 1], [0, 1, 0, 0, 0, 1, 0, 0], [0, 1, 0, 1, 0, 0, 0, 1], [1, 0, 1, 1, 0, 1, 1, 1]]
```

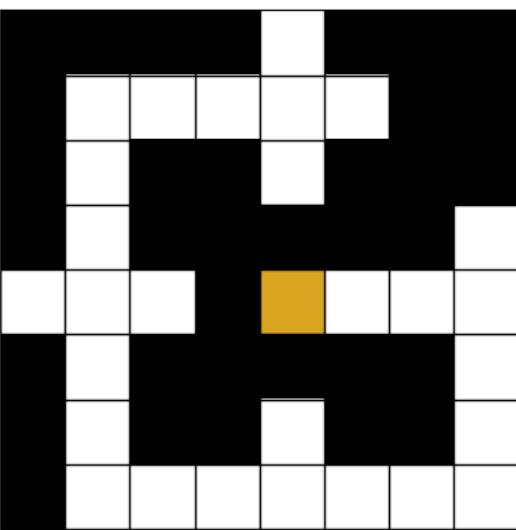
Variations for piece4



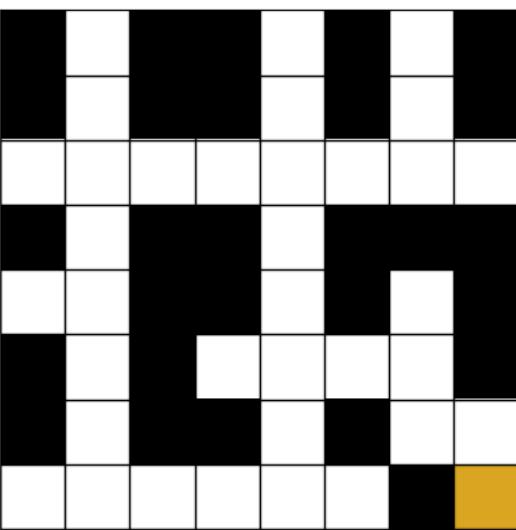
```
[[1, 0, 0, 0, 0, 1, 1, 1], [1, 0, 1, 1, 0, 0, 0, 1], [1, 0, 0, 1, 1, 1, 0, 1], [1, 1, 0, 1, 0, 0, 0, 1], [0, 0, 0, 1, 1, 1, 0, 0], [1, 0, 1, 1, 0, 1, 1, 0], [1, 0, 0, 0, 1, 0, 0], [1, 1, 1, 1, 1, 3, 0, 1]]
```



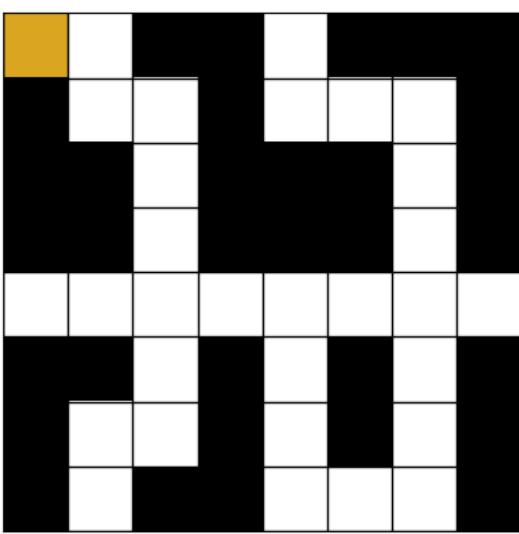
```
[[0, 0, 1, 1, 0, 1, 1, 1], [1, 0, 1, 0, 0, 1, 1, 1], [1, 0, 1, 0, 1, 1, 0, 1], [1, 0, 1, 0, 0, 0, 0, 1], [0, 0, 1, 1, 1, 1, 0, 1],  
[1, 0, 1, 3, 1, 1, 0, 1], [1, 0, 1, 0, 0, 0, 0, 1], [1, 0, 0, 0, 1, 1, 0, 1]]
```



```
[[1, 1, 1, 0, 1, 1, 1], [1, 0, 0, 0, 0, 1, 1], [1, 0, 1, 1, 0, 1, 1, 1], [1, 0, 1, 1, 1, 1, 1, 0], [0, 0, 0, 1, 3, 0, 0, 0],  
[1, 0, 1, 1, 1, 1, 0], [1, 0, 1, 1, 0, 1, 1, 0], [1, 0, 0, 0, 0, 0, 0, 0]]
```



```
[[1, 0, 1, 1, 0, 1, 0, 1], [1, 0, 1, 1, 0, 1, 0, 1], [0, 0, 0, 0, 0, 0, 0, 0], [1, 0, 1, 1, 0, 1, 1, 1], [0, 0, 1, 1, 0, 1, 0, 1],  
[1, 0, 1, 0, 0, 0, 0, 1], [1, 0, 1, 1, 0, 1, 0, 0], [0, 0, 0, 0, 0, 0, 1, 3]]
```



```
[[3, 0, 1, 1, 0, 1, 1, 1], [1, 0, 0, 1, 0, 0, 0, 1], [1, 1, 0, 1, 1, 1, 0, 1], [1, 1, 0, 1, 1, 1, 0, 1], [0, 0, 0, 0, 0, 0, 0, 0],  
[1, 1, 0, 1, 0, 1, 0, 1], [1, 0, 0, 1, 0, 1, 0, 1], [1, 0, 1, 1, 0, 0, 0, 1]]
```

Now will start the code that will apply the 4 side pieces together

I think I'll have a function called `choose_piece1` that will pick an option for `piece1`. I'll do this instead of having a variable like `piece1_options` that is a list of dictionary of `piece1`'s options. I'm choosing a function instead because the code will be easier to read.

Okay finished creating

```
choose_piece1()  
choose_piece2()  
choose_piece3()  
choose_piece4()
```

They all return a random option

```
option1  
option2  
option3  
option4  
option5
```

and the code I created to return is

```
return [option1,option2,option3,option4,option5][random.randint(0,4)]
```

The flaw with this return is that it is hardcoded though

Removing

```
piece_state=[[random.randint(1, 7) for i in range(8)] for i in range(8)]  
  
rulebook={0:[0,1,2,3,4,5,6,7],1:[1,2,3,4,5,6],2:[1,2,4,5],3:[1,3,4,6],4:[2,3,4,5,6],5:[],6:[],7:[]}  
  
for i in range(len(piece_state)):  
    for j in range(len(piece_state[0])):  
        # Get the piece  
        piece=pieces[piece_state[i][j]]  
  
        # Replace values in state with values of the piece  
        state[i*2][j*2]=piece[0][0]  
        state[i*2+1][j*2]=piece[1][0]  
        state[i*2][j*2+1]=piece[0][1]  
        state[i*2+1][j*2+1]=piece[1][1]
```

Still will use

```
pieces={0:piece0,1:piece1,2:piece2,3:piece3,4:piece4,5:piece5,6:piece6,7:piece7}
```

but will adapt it to the functions

```
pieces={1:choose_piece1(),2:choose_piece2(),3:choose_piece3(),4:choose_piece4()}
```

Okay so my approach to this problem was to scan through the entire state, and depending on what quadrant the  $i, j$  is in, the piece would switch. I used the module to adapt the  $i, j$  of the piece

```
# Place pieces in state

for i in range(len(state)):
    for j in range(len(state[0])):

        # Initiate pieces
        piece1=pieces[1]
        piece2=pieces[2]
        piece3=pieces[3]
        piece4=pieces[4]

        # if in Piece 1's range
        if i<=7 and j<=7:
            state[i][j]=piece1[i%7][j%7]

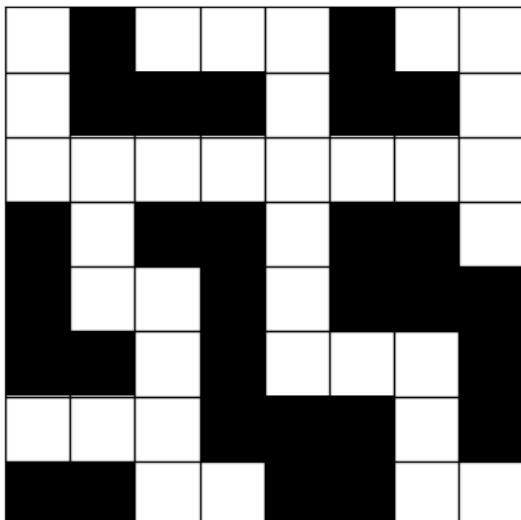
        # if in Piece 2's range
        if 7<=i and 7<j:
            state[i][j]=piece2[i%7][j%7]

        # if in Piece 3's range
        if 7>i and j<=7:
            state[i][j]=piece3[i%7][j%7]

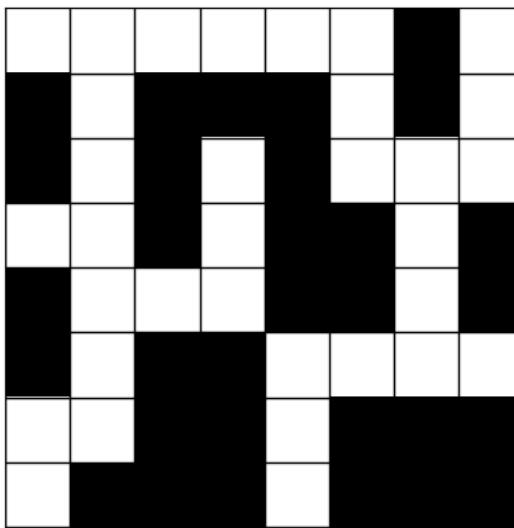
        # if in Piece4's range
        if 7>i and 7<j:
            state[i][j]=piece4[i%7][j%7]
```

Going to test if the map is correctly generated based on the pieces chosen

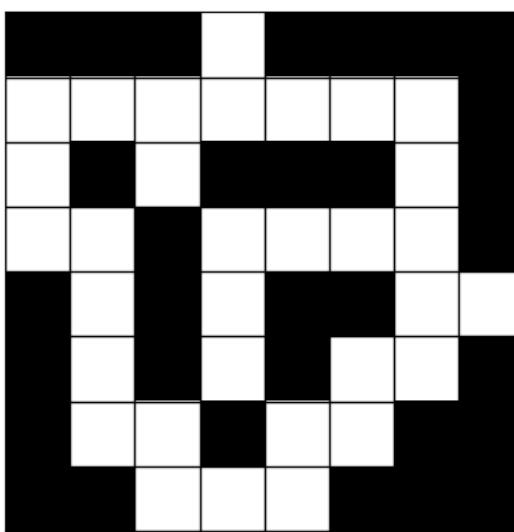
piece1 is



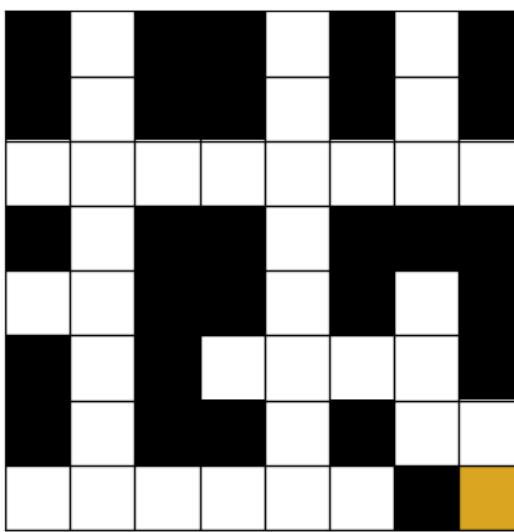
piece2 is



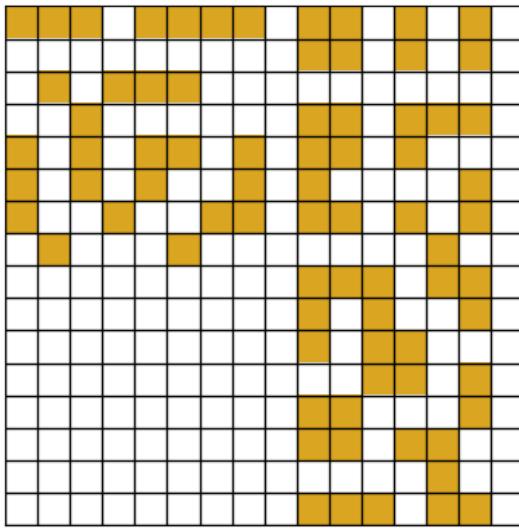
piece3 is



piece4 is



state is



Seems I have made typos here

```
# if in Piece 3's range
if 7>i and j<=7:
    state[i][j]=piece3[i%7][j%7]

# if in Piece4's range
if 7>i and 7<j:
    state[i][j]=piece4[i%7][j%7]
```

It should be `7< j` and `7< i` instead of `>`

Okay there are still errors because I cannot use the module because

and not , so we never get an index of 7

Pretty sure I can have a compact if statement that only uses the module if `i, j` does not equal to 7, and if it does, then use 7

```
i%7 if i!=7 else 7
```

Actually if I make the module 8, it would give me what I want

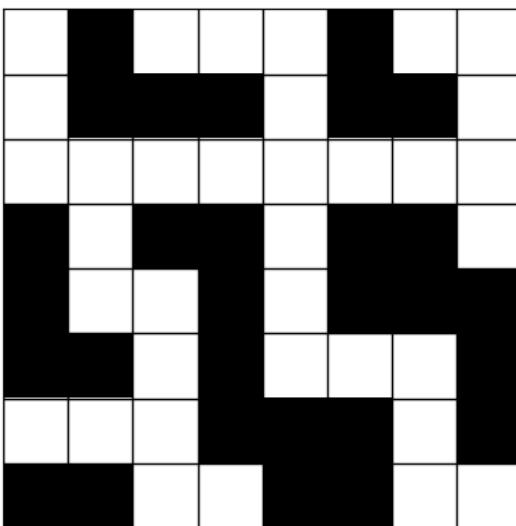
It would give 7 when `i=7`

It would give 0 when `i=8`

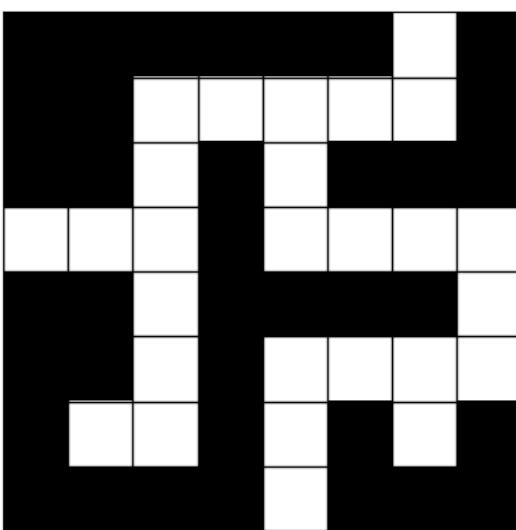
It would give 7 again `i` is at it's max index, `i=15`

Okay let's see if it works now

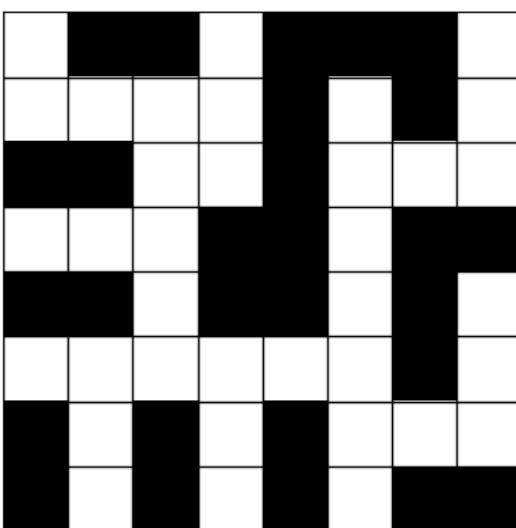
piece1



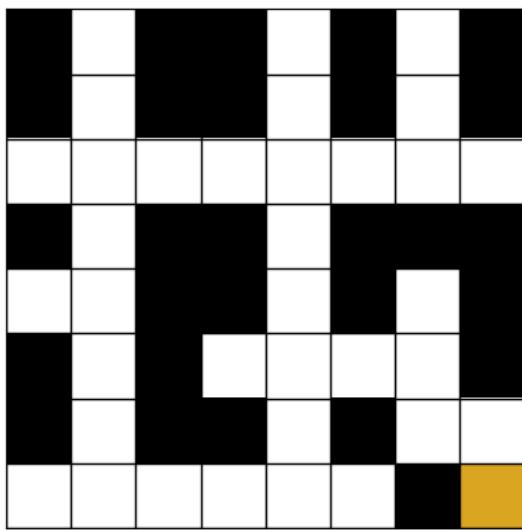
piece2



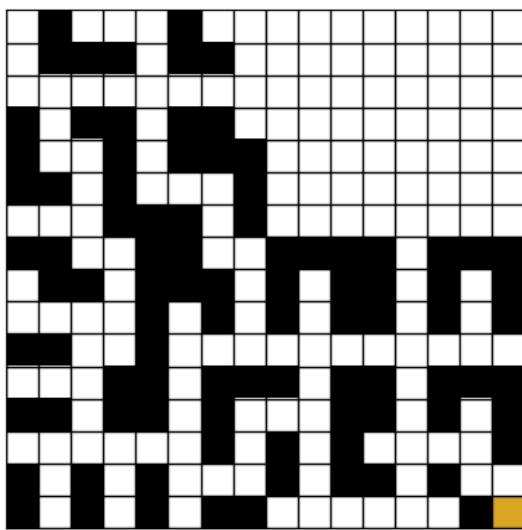
piece3



piece4



state



so close, piece2 is gone

Another typo

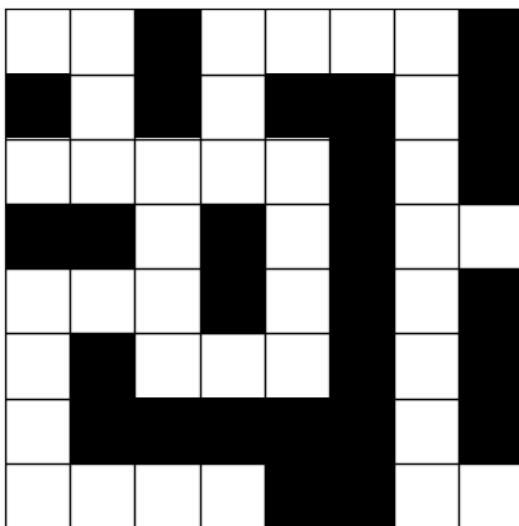
```
if 7<=i and 7<j:
```

It should be

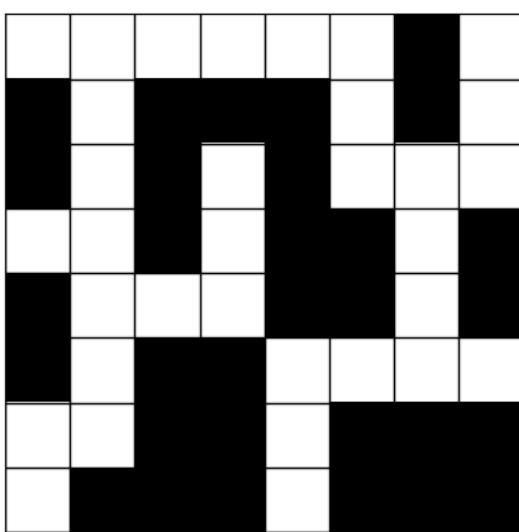
```
if i<=7 and 7<j:
```

Last test

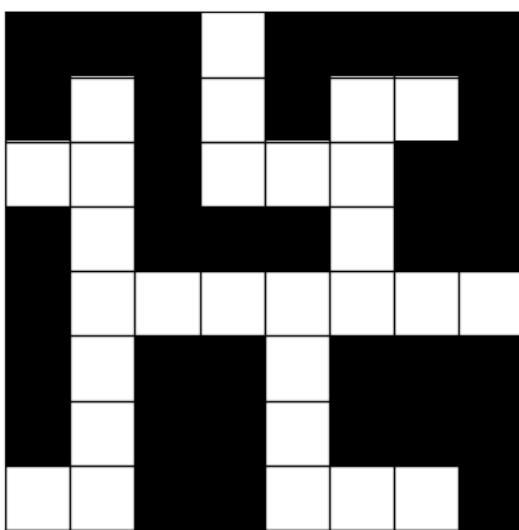
piece1



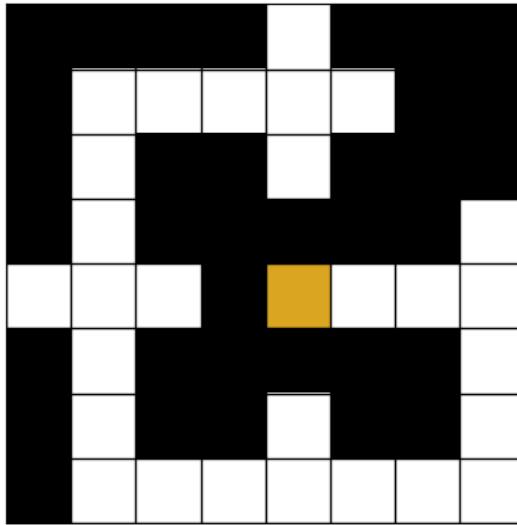
piece2



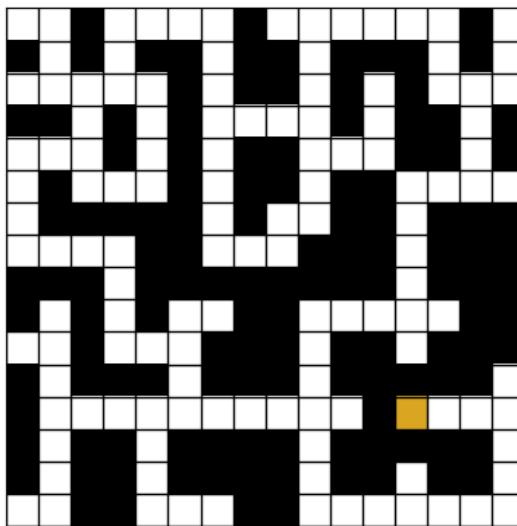
piece3



piece4



state



Alright it works!

Going to start recording the video

Actually first I will push it to GitHub

Awesome now I have unique maps!

That is a % chance of having the same map twice in a row

Okay, here we go again. First thing I will do is push the branch onto the main

Did it

Forgot I wanted to make `generate()` less hard coded

This is the new `choose_piece()` function format

```
def choose_piece3():
    options = [[[1, 1, 1, 0, 1, 1, 1, 1],
                [0, 0, 0, 0, 0, 0, 0, 1],
                [0, 1, 0, 1, 1, 1, 0, 1],
                [0, 0, 1, 0, 0, 0, 0, 1],
```

```

[1, 0, 1, 0, 1, 1, 0, 0],  

[1, 0, 1, 0, 1, 0, 0, 1],  

[1, 0, 0, 1, 0, 0, 1, 1],  

[1, 1, 0, 0, 0, 1, 1, 1]],  
  

[[1, 1, 1, 0, 1, 1, 1, 0],  

 [0, 0, 1, 0, 0, 0, 0, 0],  

 [1, 0, 1, 1, 1, 0, 1, 1],  

 [1, 0, 0, 0, 0, 0, 1, 1],  

 [1, 0, 1, 1, 1, 1, 1, 0],  

 [1, 0, 0, 0, 0, 0, 0, 0],  

 [1, 0, 1, 1, 1, 1, 1, 1],  

 [1, 0, 0, 0, 0, 0, 1, 1]],  
  

[[1, 1, 1, 0, 1, 1, 1, 1],  

 [1, 0, 1, 0, 1, 0, 0, 1],  

 [0, 0, 1, 0, 0, 0, 1, 1],  

 [1, 0, 1, 1, 1, 0, 1, 1],  

 [1, 0, 0, 0, 0, 0, 0, 0],  

 [1, 0, 1, 0, 1, 1, 1, 1],  

 [1, 0, 1, 1, 0, 1, 1, 1],  

 [0, 0, 1, 1, 0, 0, 0, 1]],  
  

[[0, 1, 1, 0, 1, 1, 1, 0],  

 [0, 0, 0, 1, 0, 1, 0, 1],  

 [1, 1, 0, 0, 1, 0, 0, 0],  

 [0, 0, 0, 1, 1, 0, 1, 1],  

 [1, 1, 0, 1, 1, 0, 1, 0],  

 [0, 0, 0, 0, 0, 0, 1, 0],  

 [1, 0, 1, 0, 1, 0, 0, 0],  

 [1, 0, 1, 0, 1, 0, 1, 1]],  
  

[[1, 1, 1, 0, 1, 1, 1, 1],  

 [1, 1, 0, 0, 0, 0, 0, 1],  

 [0, 0, 0, 1, 1, 0, 0, 1],  

 [0, 1, 1, 1, 1, 1, 1, 1],  

 [0, 1, 0, 0, 0, 1, 0, 0],  

 [0, 1, 0, 1, 0, 1, 0, 1],  

 [0, 0, 0, 1, 0, 0, 0, 1],  

 [1, 0, 1, 1, 1, 0, 1, 1]]]  
  

return options[random.randint(0,len(options)-1)]

```

The variables `option1`, `option2`, `option3`, `option4`, and `option5` are now the variable `options`. I used Spyder's Autopep8 to format it to look good  
 Format file or selection with Autopep8 Ctrl+Alt+I

The new return line is now `options` instead of a list of the old options variables

Now will apply the same format to the rest of the pieces

Done

Defining a variable called `pieces_n`, the user can write how pieces are being used

```
# Specify amount of variations per corner  
n=5
```

Actually I don't think I'll do the adaptability of the state, but I did make adding more variations flexible

# GIT CHEAT SHEET

presented by **TOWER** > Version control with Git - made easy



## CREATE

Clone an existing repository  
`$ git clone ssh://user@domain.com/repo.git`

Create a new local repository  
`$ git init`

## LOCAL CHANGES

Changed files in your working directory  
`$ git status`

Changes to tracked files  
`$ git diff`

Add all current changes to the next commit  
`$ git add .`

Add some changes in <file> to the next commit  
`$ git add -p <file>`

Commit all local changes in tracked files  
`$ git commit -a`

Commit previously staged changes  
`$ git commit`

Change the last commit  
*Don't amend published commits!*  
`$ git commit --amend`

## COMMIT HISTORY

Show all commits, starting with newest  
`$ git log`

Show changes over time for a specific file  
`$ git log -p <file>`

Who changed what and when in <file>  
`$ git blame <file>`

## BRANCHES & TAGS

List all existing branches  
`$ git branch -av`

Switch HEAD branch  
`$ git checkout <branch>`

Create a new branch based on your current HEAD  
`$ git branch <new-branch>`

Create a new tracking branch based on a remote branch  
`$ git checkout --track <remote/branch>`

Delete a local branch  
`$ git branch -d <branch>`

Mark the current commit with a tag  
`$ git tag <tag-name>`

## UPDATE & PUBLISH

List all currently configured remotes  
`$ git remote -v`

Show information about a remote  
`$ git remote show <remote>`

Add new remote repository, named <remote>  
`$ git remote add <shortname> <url>`

Download all changes from <remote>, but don't integrate into HEAD  
`$ git fetch <remote>`

Download changes and directly merge/integrate into HEAD  
`$ git pull <remote> <branch>`

Publish local changes on a remote  
`$ git push <remote> <branch>`

Delete a branch on the remote  
`$ git branch -dr <remote/branch>`

Publish your tags  
`$ git push --tags`

## MERGE & REBASE

Merge <branch> into your current HEAD  
`$ git merge <branch>`

Rebase your current HEAD onto <branch>  
*Don't rebase published commits!*  
`$ git rebase <branch>`

Abort a rebase  
`$ git rebase --abort`

Continue a rebase after resolving conflicts  
`$ git rebase --continue`

Use your configured merge tool to solve conflicts  
`$ git mergetool`

Use your editor to manually solve conflicts and (after resolving) mark file as resolved  
`$ git add <resolved-file>`  
`$ git rm <resolved-file>`

## UNDO

Discard all local changes in your working directory  
`$ git reset --hard HEAD`

Discard local changes in a specific file  
`$ git checkout HEAD <file>`

Revert a commit (by producing a new commit with contrary changes)  
`$ git revert <commit>`

Reset your HEAD pointer to a previous commit...and discard all changes since then  
`$ git reset --hard <commit>`

...and preserve all changes as unstaged changes  
`$ git reset <commit>`

...and preserve uncommitted local changes  
`$ git reset --keep <commit>`

I'll remove this function since we know our maps are always valid

```
def valid_map(self, state):
    return True
```

Before coding the `Agent` class, I'll first make sure the functions in `Environment` are complete.

## Complete Environment class's functions

- `_init_()`
- `-generate()`
- `-percept()`
- `update()`
- `-find_positions()`

I got to code `update()` before coding the `Agent`, I should also code `visualize()` to visualize a state.

Going to start coding `visualize()`

I imported `matplotlib.pyplot` and I know I have to use a colourmap and have a custom colour scale:

- 0 will be white for empty space
- 1 will be black for a wall
- 2 will be blue for the agent
- 3 will be yellow for the goal

Made a `cmap` according to <https://matplotlib.org/stable/tutorials/colors/colormap-manipulation.html>

```
def visualize(self,state):  
    cmap=ListedColormap(["white","black","blue","yellow"])  
    print(cmap)  
    pass
```

[Seems useful](#)

# matplotlib.axes.Axes.matshow #

`Axes.matshow(z, **kwargs) #`

[source]

Plot the values of a 2D matrix or array as color-coded image.

The matrix will be shown the way it would be printed, with the first row at the top. Row and column numbering is zero-based.

## Parameters:

`z : (M, N) array-like`

The matrix to be displayed.

## Returns:

`AxesImage`

## Other Parameters:

`**kwargs : imshow arguments`

## See also

`imshow`

More general function to plot data on a 2D regular raster.

Seems like I need to place the array in a M, N format

Forgot

```
return state
```

at the end of `generate()`

Right now `state` is `<class 'list'>`

Had to change

```
ax.matshow(state,cmap)
```

#to

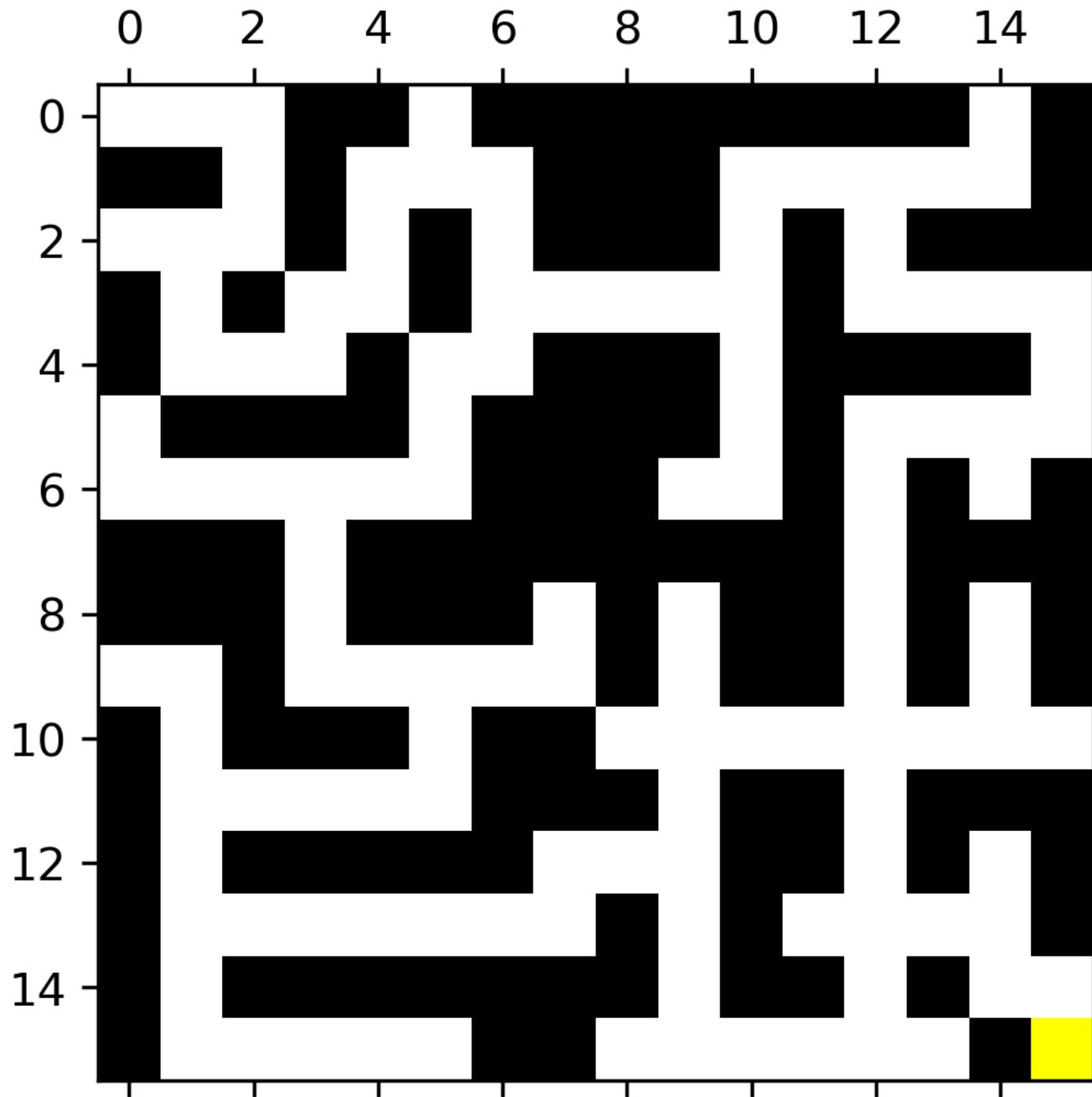
```
ax.matshow(state,cmap=cmap)
```

Got code to output an accurate map

```
def visualize(self,state):
    # Custom color map
    cmap=ListedColormap(["white","black","blue","yellow"])

    # Create plot
    fig,ax=plt.subplots()
    ax.matshow(state,cmap=cmap)
    fig.show()
```

pass



Changed colours to same as Google Sheets

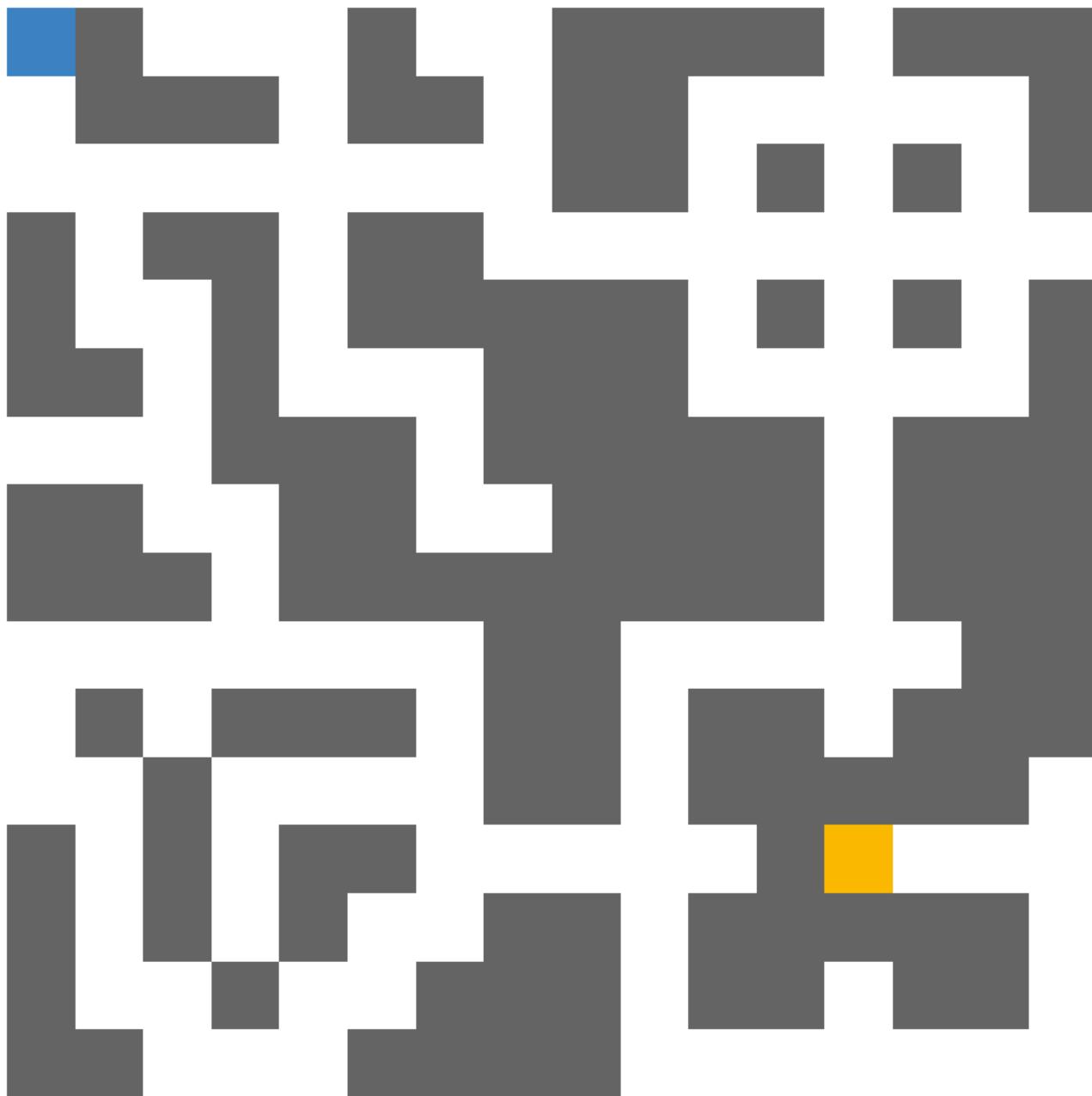
```
cmap=ListedColormap(["#ffffff", "#666666", "#3d85c6", "#fb8c04"])
```

Turned off axis

```
ax.axis("off")
```

Tested with 2 for Agent

```
state[0][0]=2
```



Not sure how to cut the borders, tried this but didn't work

```
ax.margins(0,0)
fig.subplots_adjust(top = 1, bottom = 0, right = 1, left = 0,
    hspace = 0, wspace = 0)
```

So the final code for `visualize()` for now is

```
def visualize(self,state):
    # Custom color map
    cmap=ListedColormap([ "#ffffff", "#666666", "#3d85c6", "#fbcc04"])

    # Create plot
    fig,ax=plt.subplots()
    ax.matshow(state,cmap=cmap)
    ax.axis("off")

    # Show plot
    fig.show()

    return None
```

Will push this code to main for having `visualize()`

Looks like there are these errors I'll deal with later, for now I really want to start on `update()`

```
You have unmerged paths.  
(fix conflicts and run "git commit")  
(use "git merge --abort" to abort the merge)  
  
Unmerged paths:  
(use "git add <file>..." to mark resolution)  
 both modified: Project 2.py
```

Alright now there are random stuff in my code

```
>>>>> 44b1ee2dd6a2e238d750a915b66c92175ccfdad3  
    for i in range(len(state)):  
        for j in range(len(state[0])):  
  
            # if in Piece 1's range  
            if i<=7 and j<=7:  
                state[i][j]=piece1[i%8][j%8]  
  
            # if in Piece 2's range  
            if i<=7 and 7<j:  
                state[i][j]=piece2[i%8][j%8]  
  
            # if in Piece 3's range  
            if 7<i and j<=7:  
                state[i][j]=piece3[i%8][j%8]  
  
            # if in Piece4's range  
            if 7<i and 7<j:  
                state[i][j]=piece4[i%8][j%8]  
  
<<<<< HEAD  
    # print(state)  
    return state  
=====  
    print(state)  
    sys.exit()  
    pass  
>>>>> 44b1ee2dd6a2e238d750a915b66c92175ccfdad3
```

`update()` will take an action from the Agent . An action will be

```
"N" #for North  
"E" #for East  
"W" #for West  
"S" #for South
```

`update()` will then move the agent (the 2 in the `state` ) in the direction asked by the Agent . A `transition_model()` will be used to make sure the agents don't go in walls or out of the map. If an agent tries to go in a wall, it just won't move

`update(self,action,state)` will ask for an action and a state

Actually first I'll do the transition model before `update()`

action and state will be needed

```
def transition_model(self,action,state)
```

I'll be testing going South

```
self.transition_model("S",state)

#Testing zone
state[0][0]=2
self.visualize(state)
```

Just to be future proof though I know there will just be one main "real" Agent on the map, I'll have the code verify for multiple positions

```
positions=self.find_positions(state)

for position in positions:
    print(position)
```

So far this code is working when checking for a wall

```
def transition_model(self,action,state):
    # Testing zone
    state[0][0]=2
    self.visualize(state)

    # Get positions
    positions=self.find_positions(state)

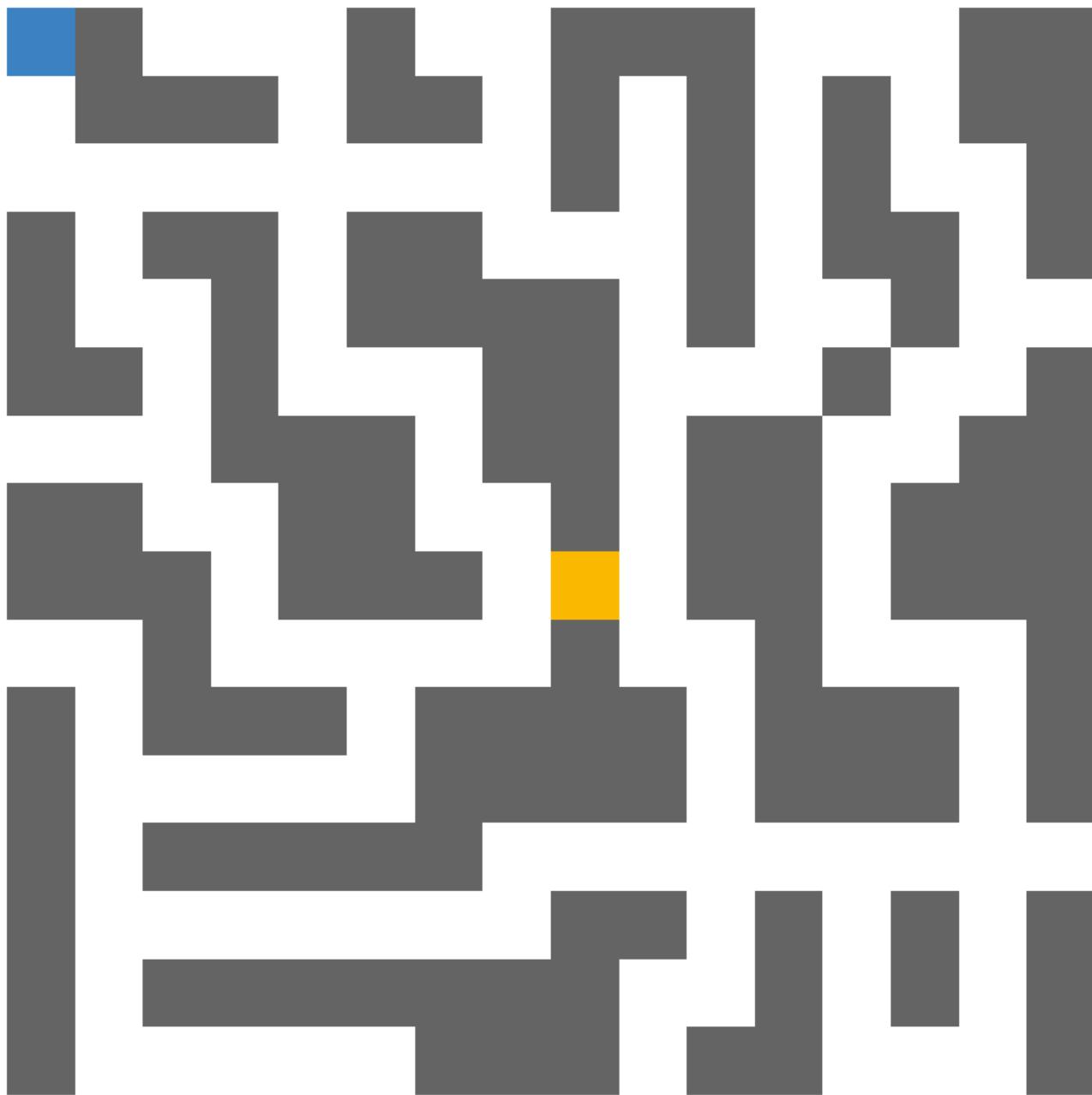
    # Initiate results
    result=[]

    for position in positions:
        # Position of agent
        i=position[0]
        j=position[1]

        # If action is South
        if action=="S":

            # Check one step South in state
            if state[i+1][j]==1:

                # Return False if there is a wall blocking agent
                result.append(False)
            else:
                result.append(True)
    print(result)
    return result
```



```
result=[True]
```



```
result=[False]
```

Actually I will do something different, I will use the `percept()` to know where the walls are instead. I won't recode the same thing by finding if there's a wall or not

```
def transition_model(self,action,state):
    # Testing zone
    state[0][0]=2
    self.visualize(state)

    # Get percept
    percept=self.percept(state)
    print(percept)

    # Initiate results
    result=[]
```

So far this works well

```
def transition_model(self,action,state):
    # Testing zone
    state[0][0]=2
    self.visualize(state)

    # Get positions
    positions=self.find_positions(state)

    # Initiate results
    result=[]

    for position in positions:
        # Find percept
        percept=self.percept(state,position)

        # If North
        if action=="N":
            # Check for a wall
            if percept[0]=="1":
                # False if there's a wall
                result.append(False)
            else:
                result.append(True)
```

results in when trying to go North



```
result=[False]
```

After finishing the code, let's see if it's right

```
def transition_model(self,action,state):
    # Testing zone
    state[random.randint(0,15)][random.randint(0,15)]=2
    state[random.randint(0,15)][random.randint(0,15)]=2
    state[random.randint(0,15)][random.randint(0,15)]=2
    self.visualize(state)

    # Get positions
    positions=self.find_positions(state)

    # Initiate results
    result=[]

    for position in positions:
        # Find percept
        percept=self.percept(state,position)
```

```

# If North
if action=="N":
    # Check for a wall
    if percept[0]=="1":
        # False if there's a wall
        result.append(False)
    else:
        result.append(True)

# If East
if action=="E":
    if percept[1]=="1":
        result.append(False)
    else:
        result.append(True)

# If West
if action=="W":
    if percept[2]=="1":
        result.append(False)
    else:
        result.append(True)

# if South
if action=="S":
    if percept[3]=="1":
        result.append(False)
    else:
        result.append(True)

print(result)
return result

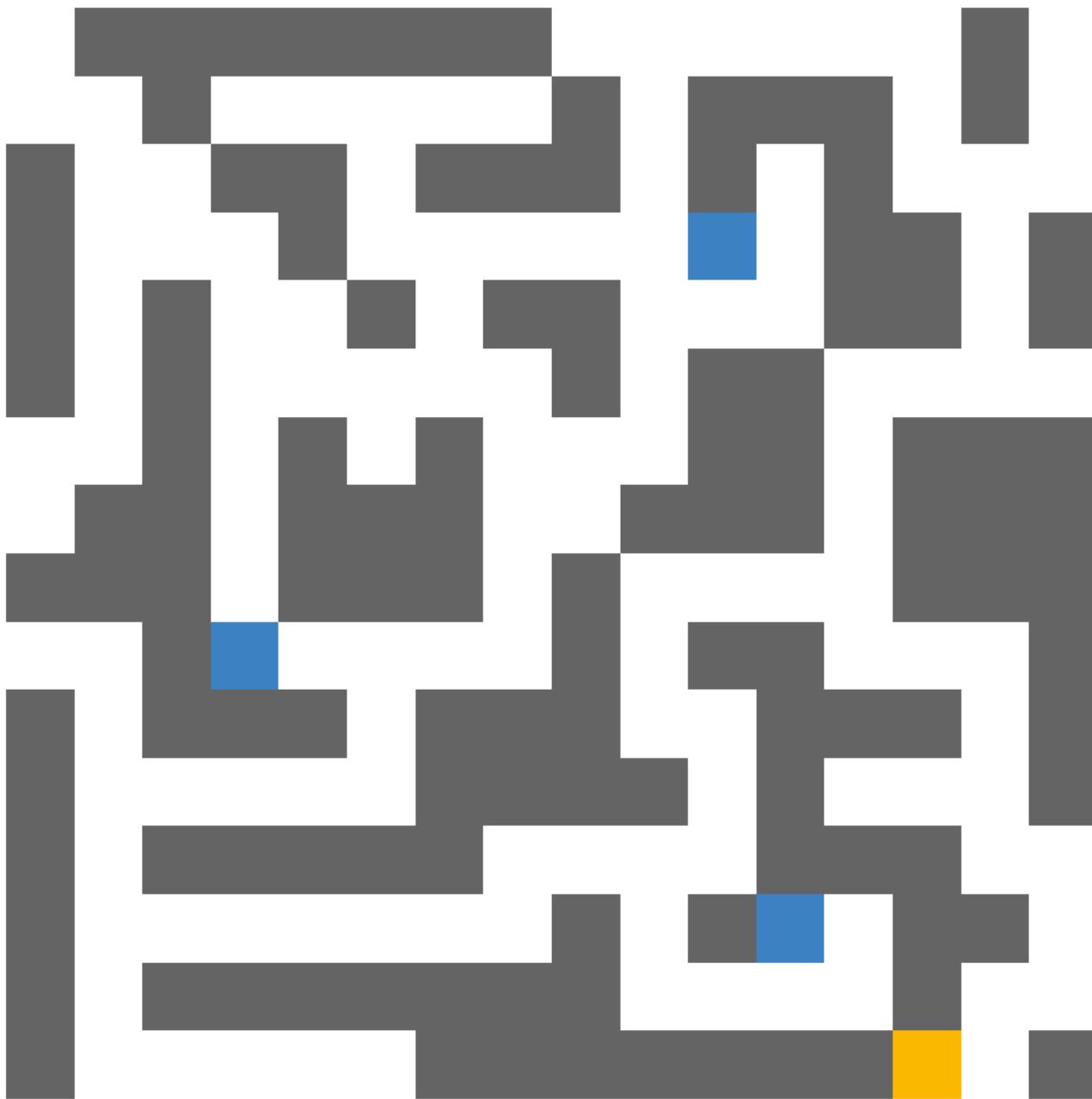
```

I added agents around randomly and testing for North

```

state[random.randint(0,15)][random.randint(0,15)]=2
state[random.randint(0,15)][random.randint(0,15)]=2
state[random.randint(0,15)][random.randint(0,15)]=2

```



```
result=[False, True, False]
```

- For position [3, 10] it's false because there is a wall above, so correct
- For position [9, 3] it's true because there is no wall above, so correct
- For position [13, 11] it's false because there is a wall above, so correct

So far everything seems correct with the code. If the user wanted only the first agent, they could do

```
result[0]
```

In case two agents are next to each other like this



I added in the code to return false if there's a wall or an agent in the way (testing for East)

```
# If East
if action=="E":
    if percept[1] in "12":
        result.append(False)
    else:
        result.append(True)
```

it results in

```
result=[False, True, False]
```

which is correct

Final code for `transition_model()`

```
def transition_model(self,action,state):
    # Get positions
    positions=self.find_positions(state)

    # Initiate results
```

```

result=[]

for position in positions:
    # Find percept
    percept=self.percept(state,position)

    # If North
    if action=="N":
        # Check for a wall
        if percept[0] in "12":
            # False if there's a wall
            result.append(False)
        else:
            result.append(True)

    # If East
    if action=="E":
        if percept[1] in "12":
            result.append(False)
        else:
            result.append(True)

    # If West
    if action=="W":
        if percept[2] in "12":
            result.append(False)
        else:
            result.append(True)

    # if South
    if action=="S":
        if percept[3] in "12":
            result.append(False)
        else:
            result.append(True)

return result

```

Pushed the code to the main with the commit Added transition\_model() and update()

Going to try to complete \_\_init\_\_()

How it looked before

```

def __init__(self,initial_state=None,agent_name="Pathfinder"):
    state=self.generate()
    self.transition_model("S",state)
    sys.exit()

    #Generate initial state if not given
    if initial_state==None:
        initial_state=self.generate()
    else:
        #If given, then check if it's valid
        if self.valid_map(initial_state):
            pass
        else:
            raise(Exception("Invalid map"))

    #Find positon of mini agent (there will be one)
    position=self.find_positions(initial_state,True)

    #Make sure agent does not know it's position
    initial_state_clueless=copy.deepcopy(initial_state)
    for i in range(len(initial_state_clueless)):
        for j in range(len(initial_state_clueless)):
            if initial_state_clueless[i][j]==2:
                initial_state_clueless[i][j]=0

```

```

print(initial_state)
#Get percept for Agent
percept=self.percept(initial_state,position)

#Give percept to Agent
self.Agent=Agent(agent_name,initial_state_clueless,percept)
print(self.Agent.data)
# print(self.Agent.data)
#Save data in DataFrame
self.data=pd.DataFrame(columns=["Initial state","State","Percept"])
self.data["Initial state"]=initial_state
self.data["State"]=initial_state
self.data["Percept"]=percept
# print(self.data)

```

Don't need this part since are not checking if the map is valid

```

else:
    #If given, then check if it's valid
    if self.valid_map(initial_state):
        pass
    else:
        raise(Exception("Invalid map"))

```

Added

```

# Place Agent in map
initial_state[0][0]=2

```

I don't think I'll give `Environment` a `DataFrame` because I don't really see any use for it. `Environment` is supposed to be the real time status of the `Agent`, and I don't see why we would need to go back in history. Maybe to record cost? But that will be recorded by the `Agent`. `Environment` is more of an interface for the user

Going to start on `update()`

Okay got the code working and did a little test run

```

# Initiate indicator that says if Agent moved
moved=False

# Get position of Agent
i=self.find_positions(state,True)[0]
j=self.find_positions(state,True)[1]

# If action is North, verify with transition model
if action=="N" and self.transition_model("N",state)[0]:

    # Place Agent one step North
    state[i-1][j]=2

    # Say Agent moved
    moved=True

if moved:
    state[i][j]=0

return state

```

First I made a variable that says if the agent moved, this is so later we know if we have to get rid of the old agent

```

# Initiate indicator that says if Agent moved
moved=False

```

Then we get the position of the agent (there only should be one)

```
# Get position of Agent
i=self.find_positions(state,True)[0]
j=self.find_positions(state,True)[1]
```

Then we check which action we are going, and verify if it's possible with the transition model

```
# If action is North, verify with transition model
if action=="N" and self.transition_model("N",state)[0]:
```

```
# Place Agent one step North
state[i-1][j]=2
```

```
# Say Agent moved
moved=True
```

If we pass the if statement, then we place where the new agent will be, and update moved to True

Later we can replace the old agent and return the state

```
if moved:
    state[i][j]=0

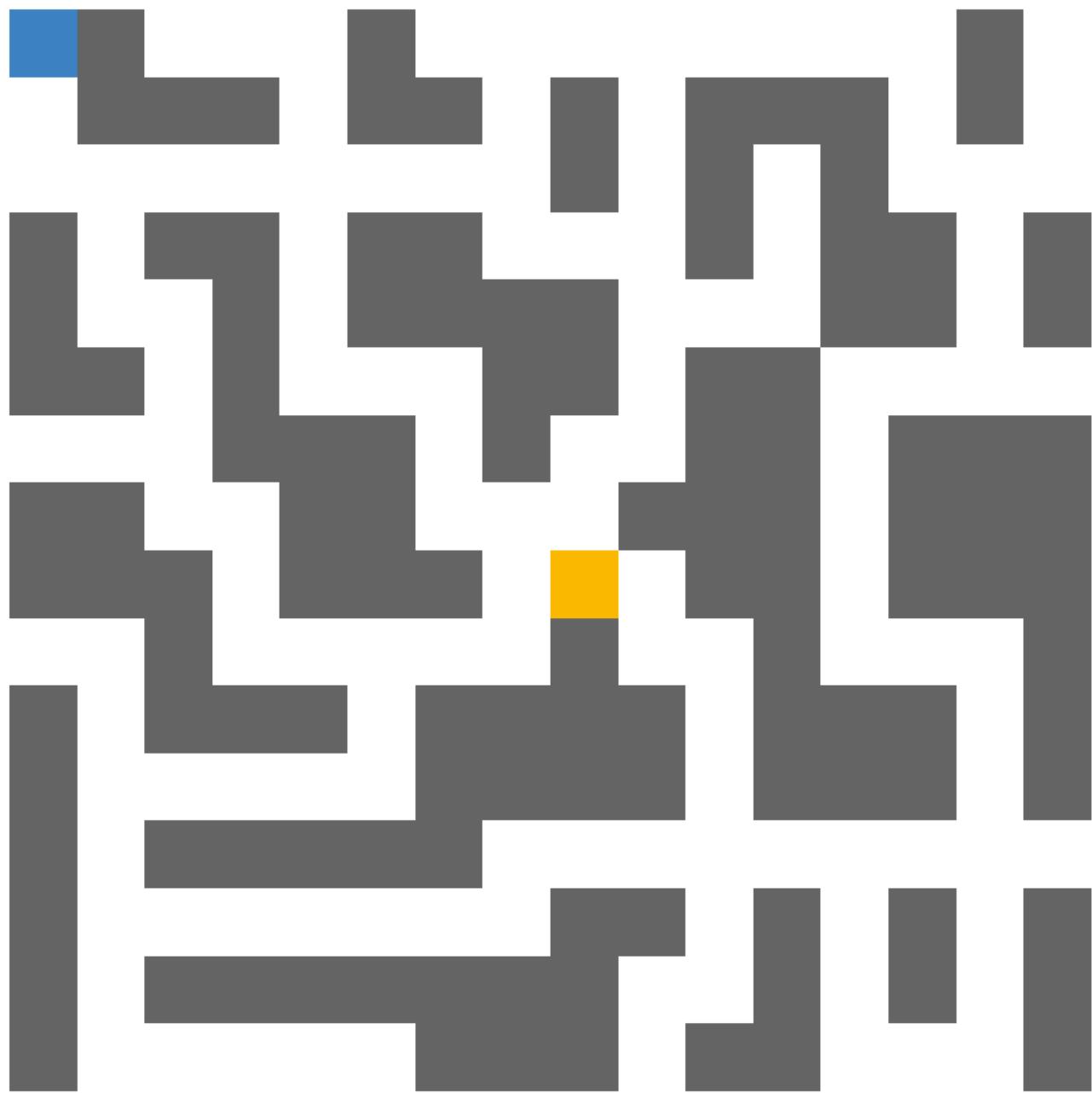
return state
```

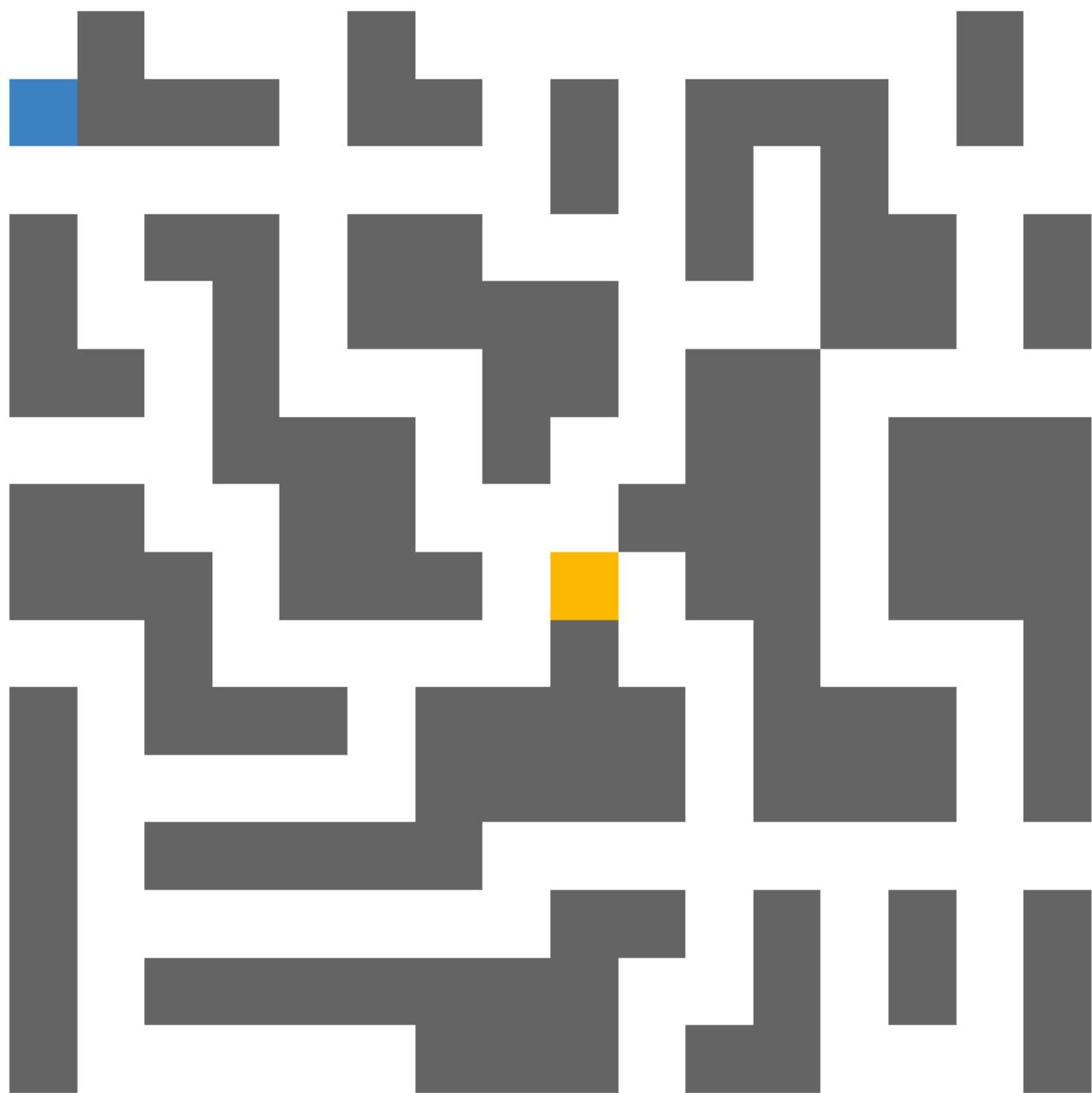
To test if this works, I made a little code that tries to get the agent to a specific place. I also tested the transition model by adding extras "W" at the end of the run

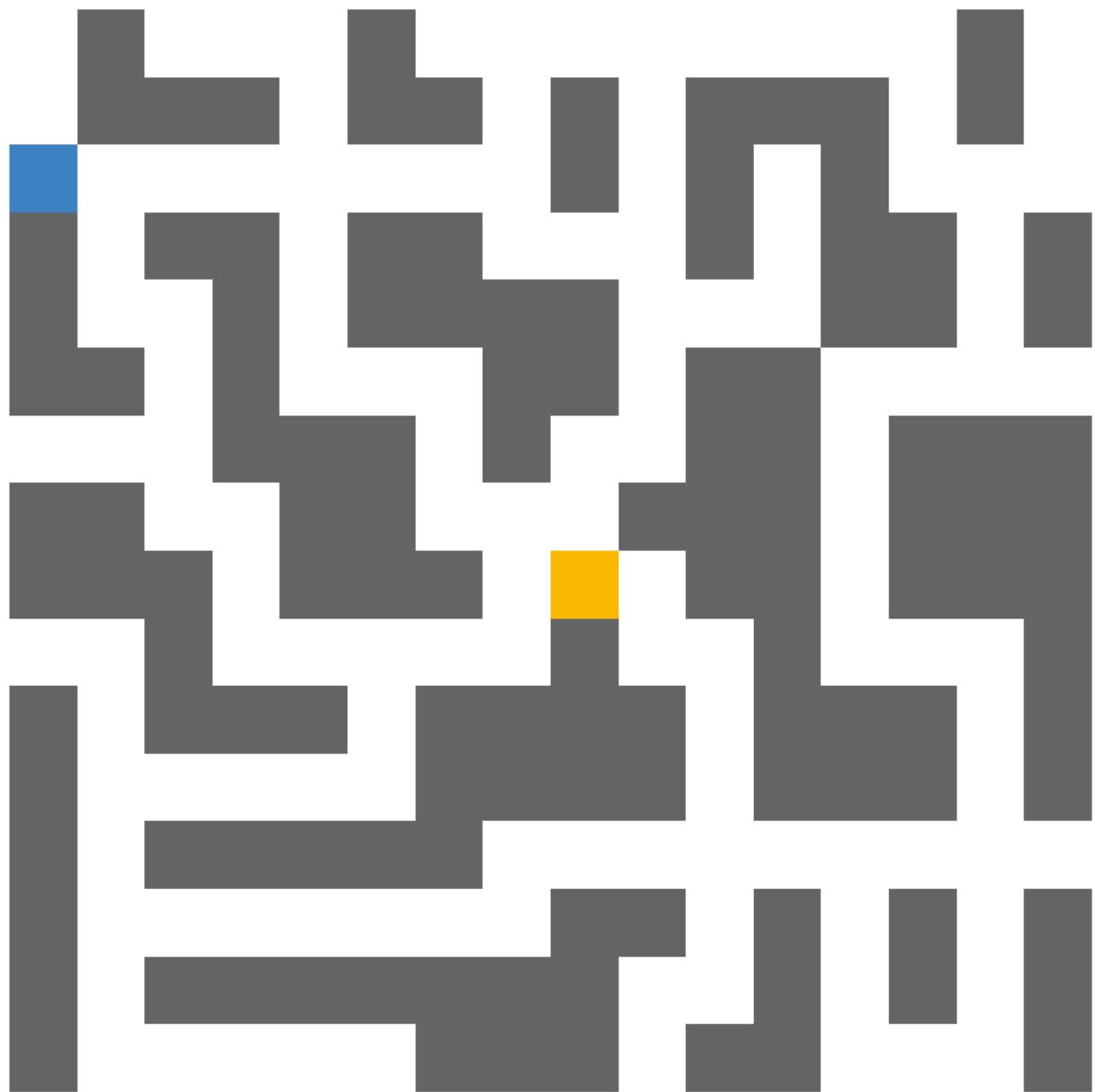
```
for direction in ["S","S","E","E","E","N","N","W","W","W","W"]:
    initial_state=self.update(direction,initial_state)
```

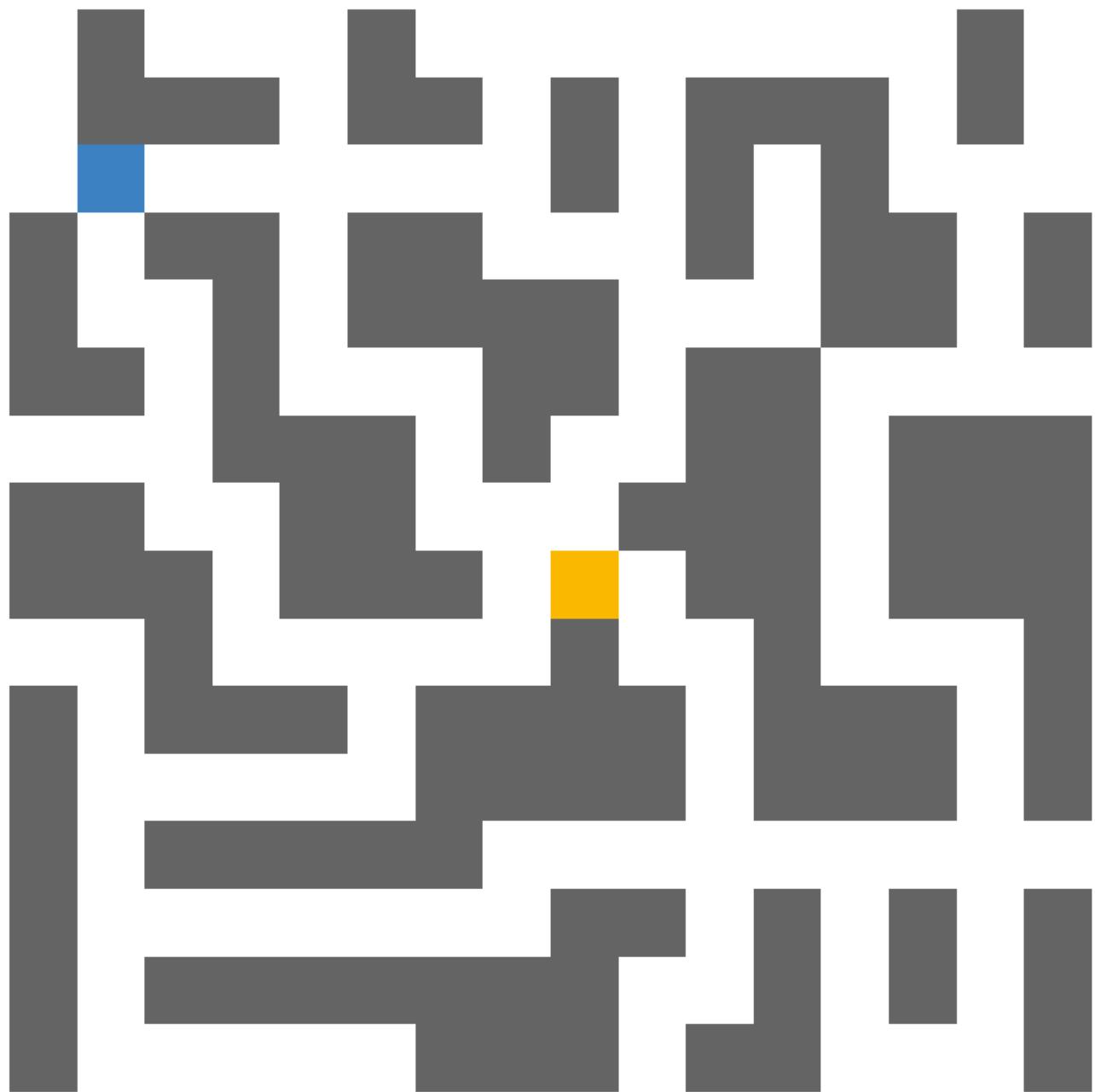
This is the result:

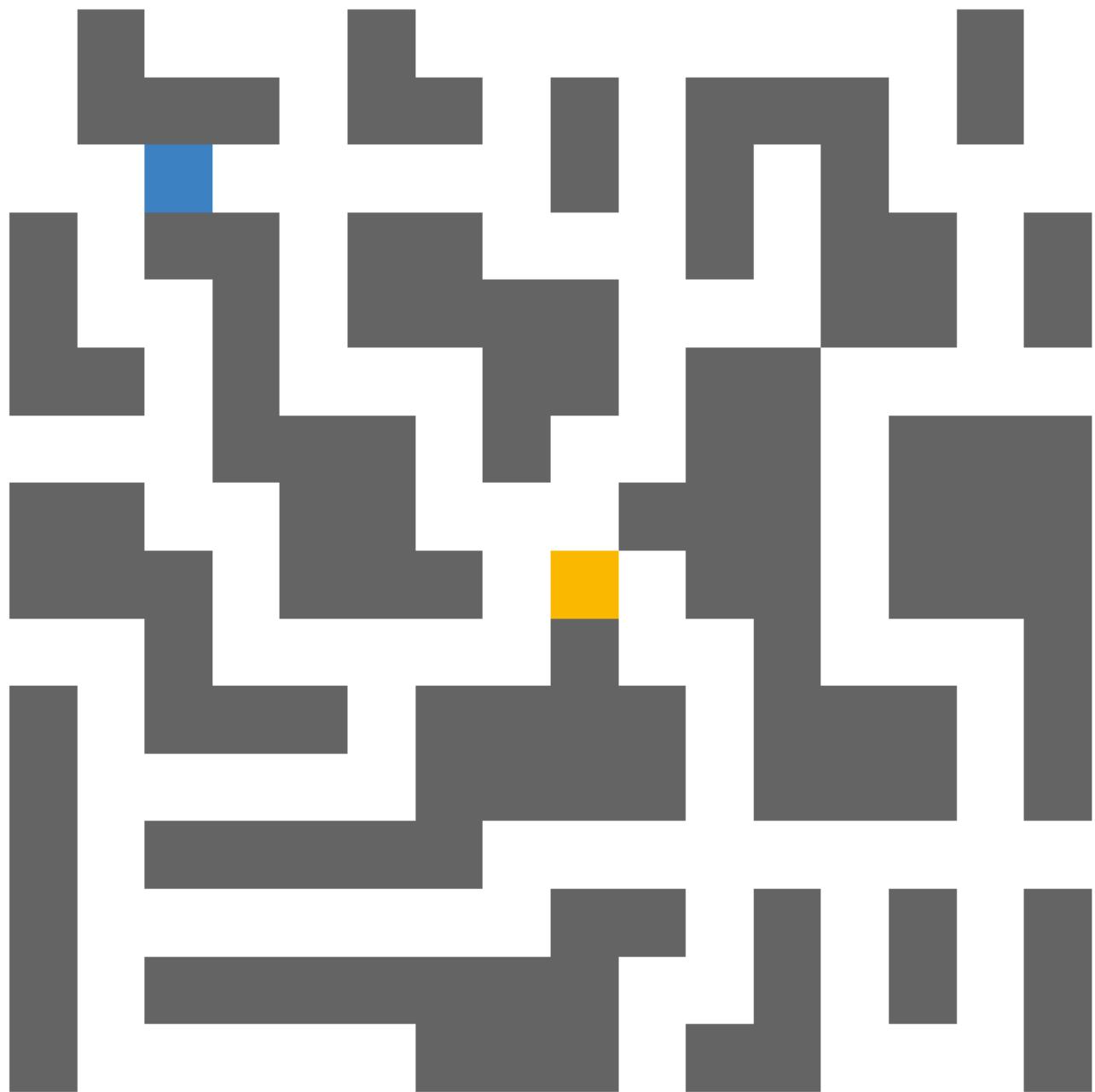
 Agent path >

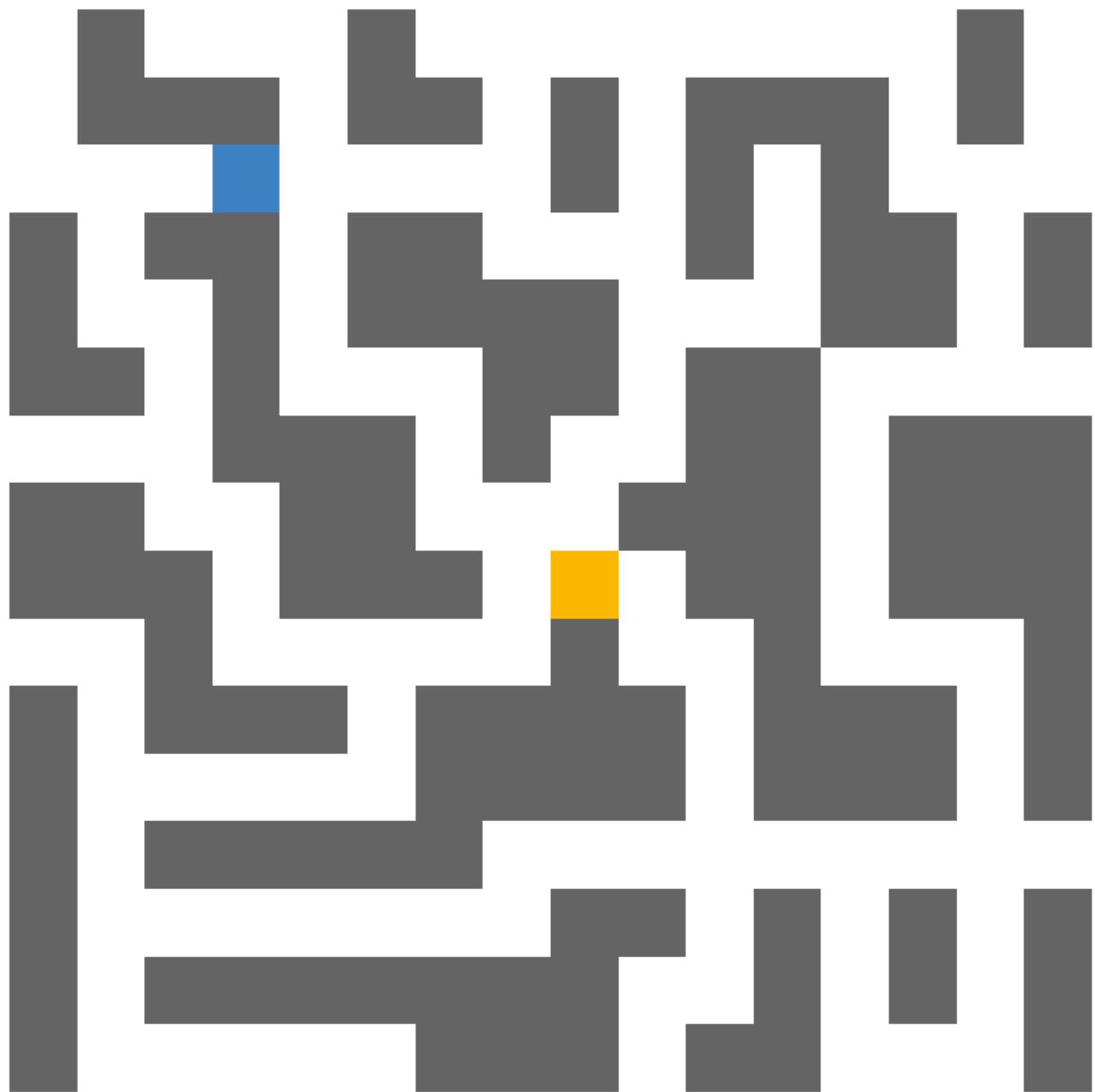


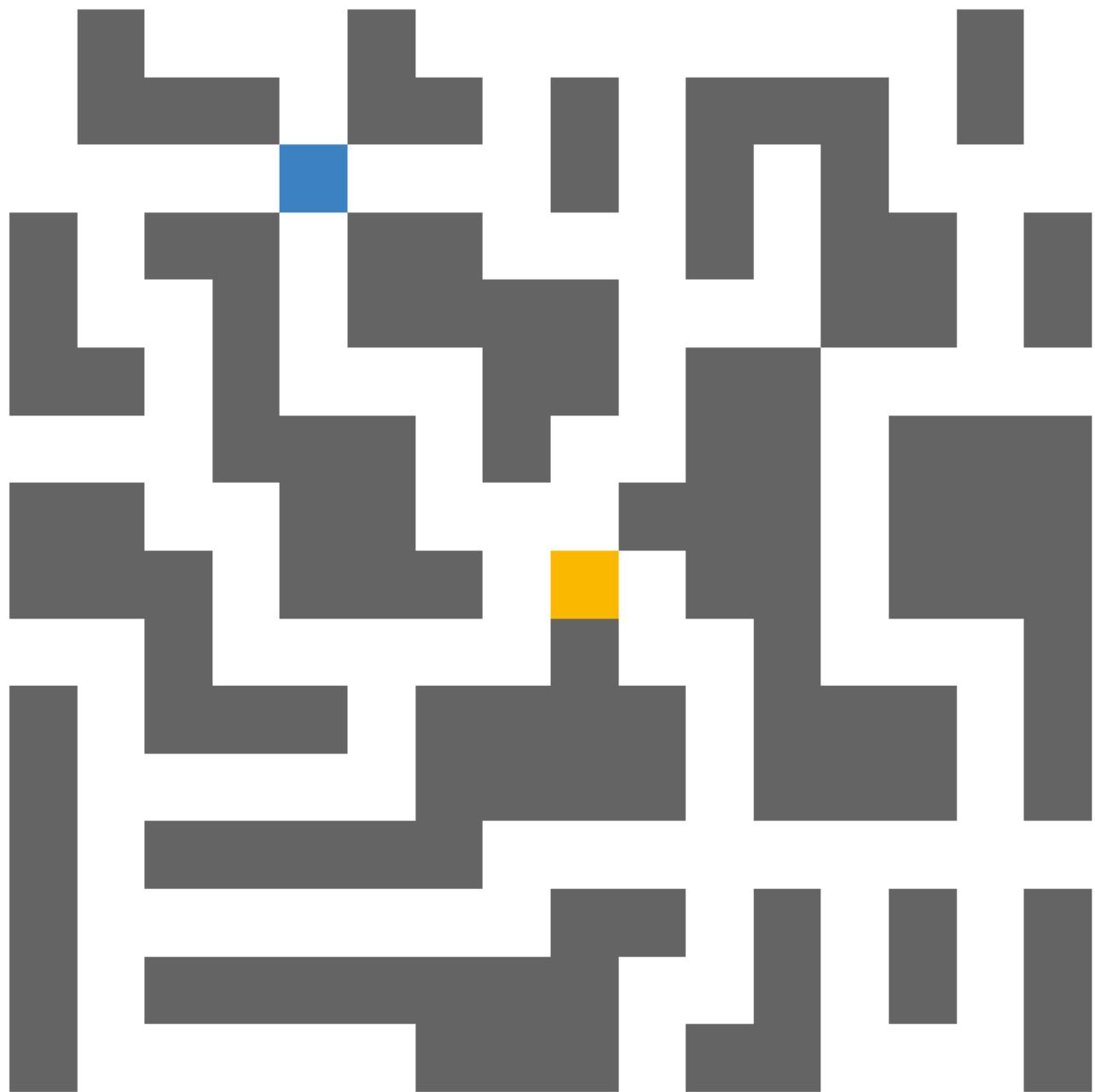


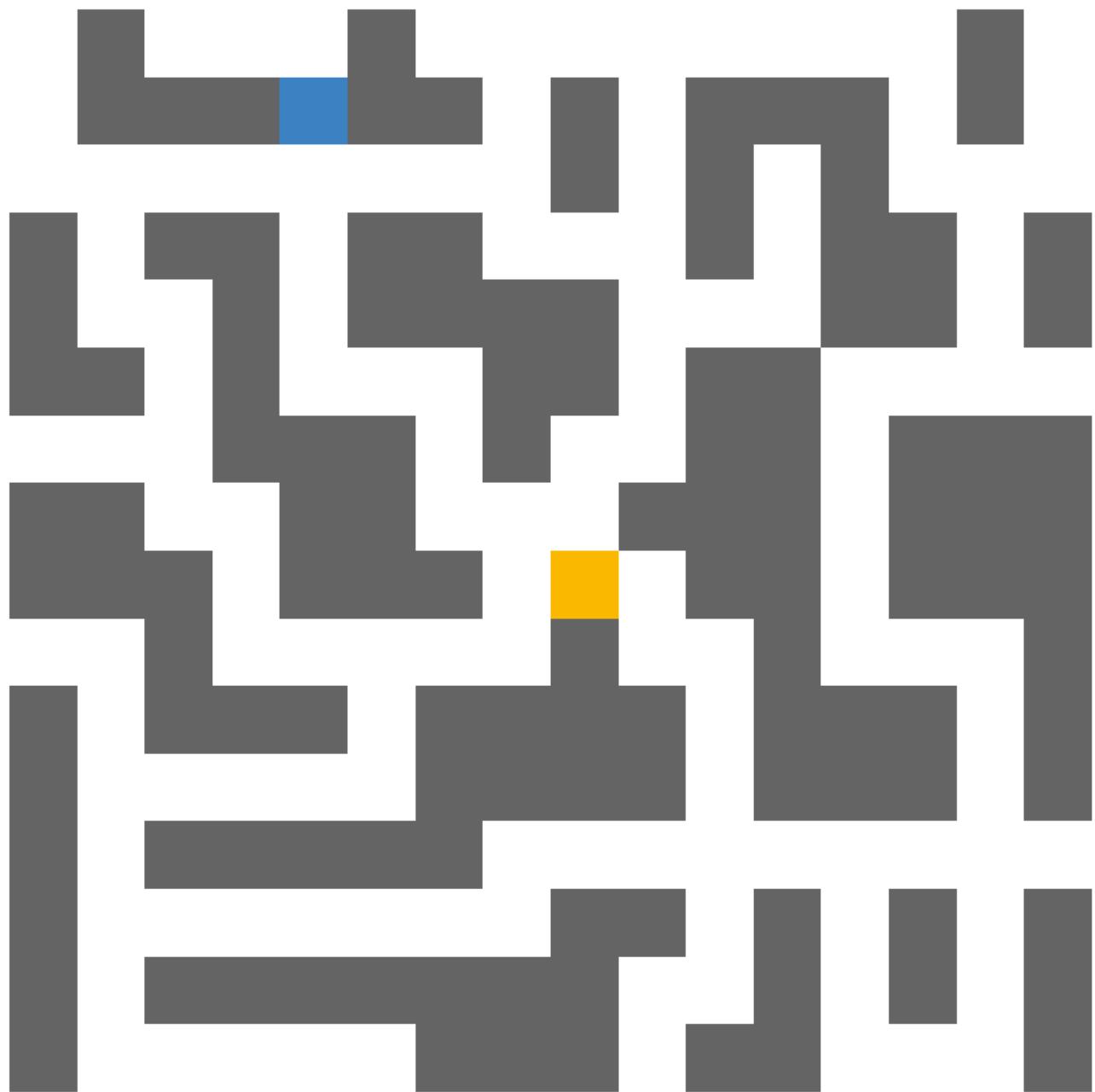


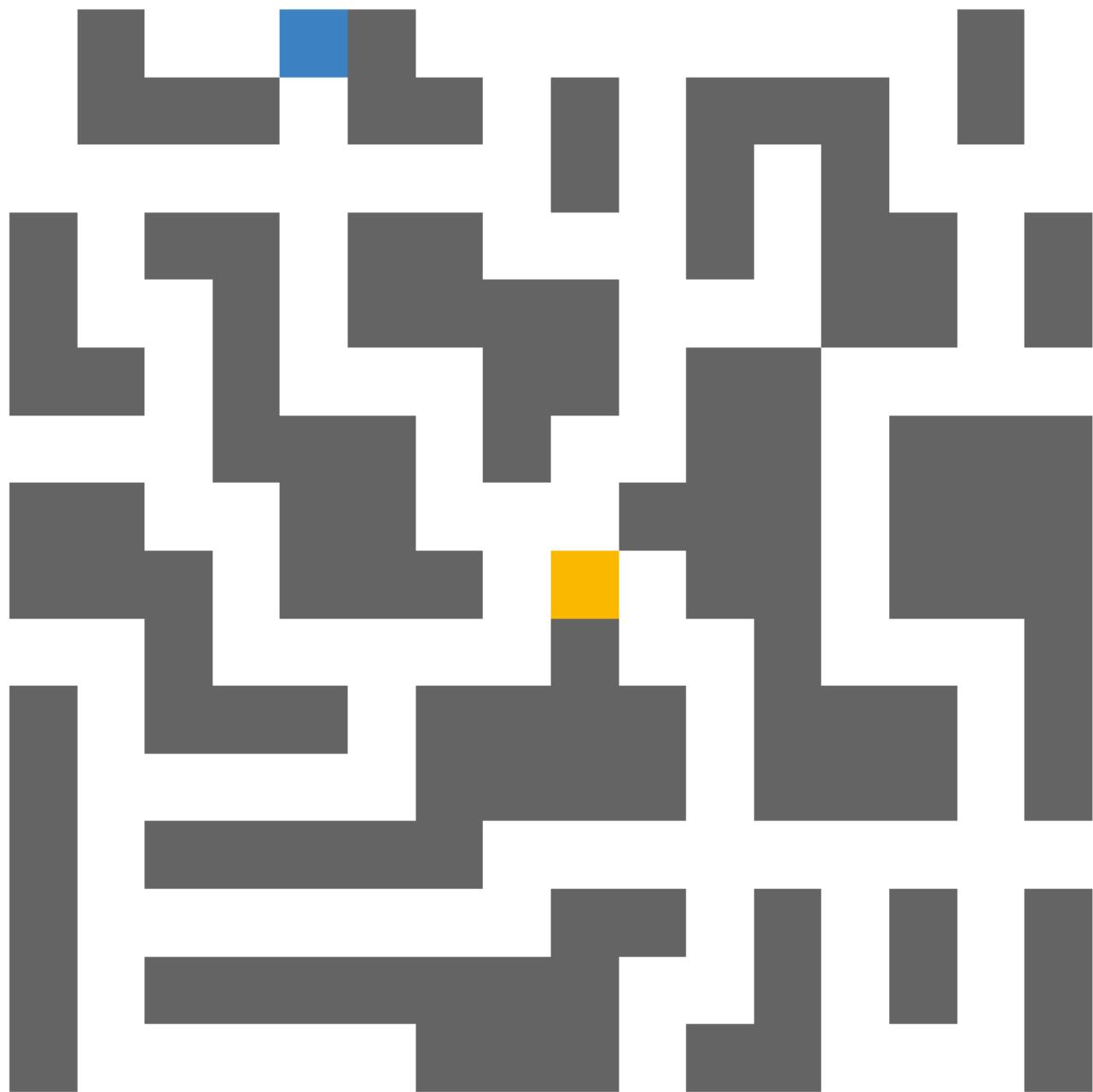


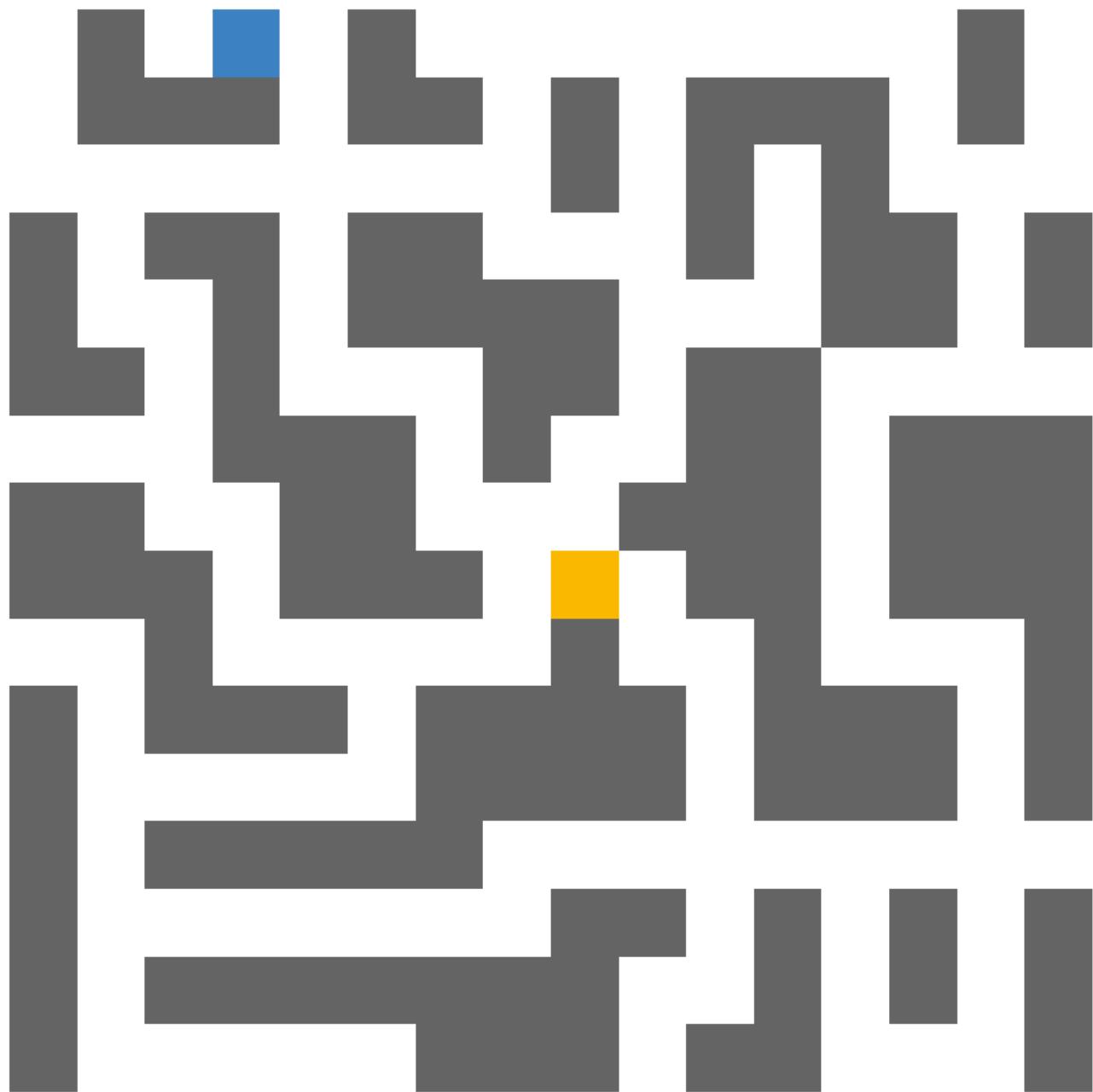


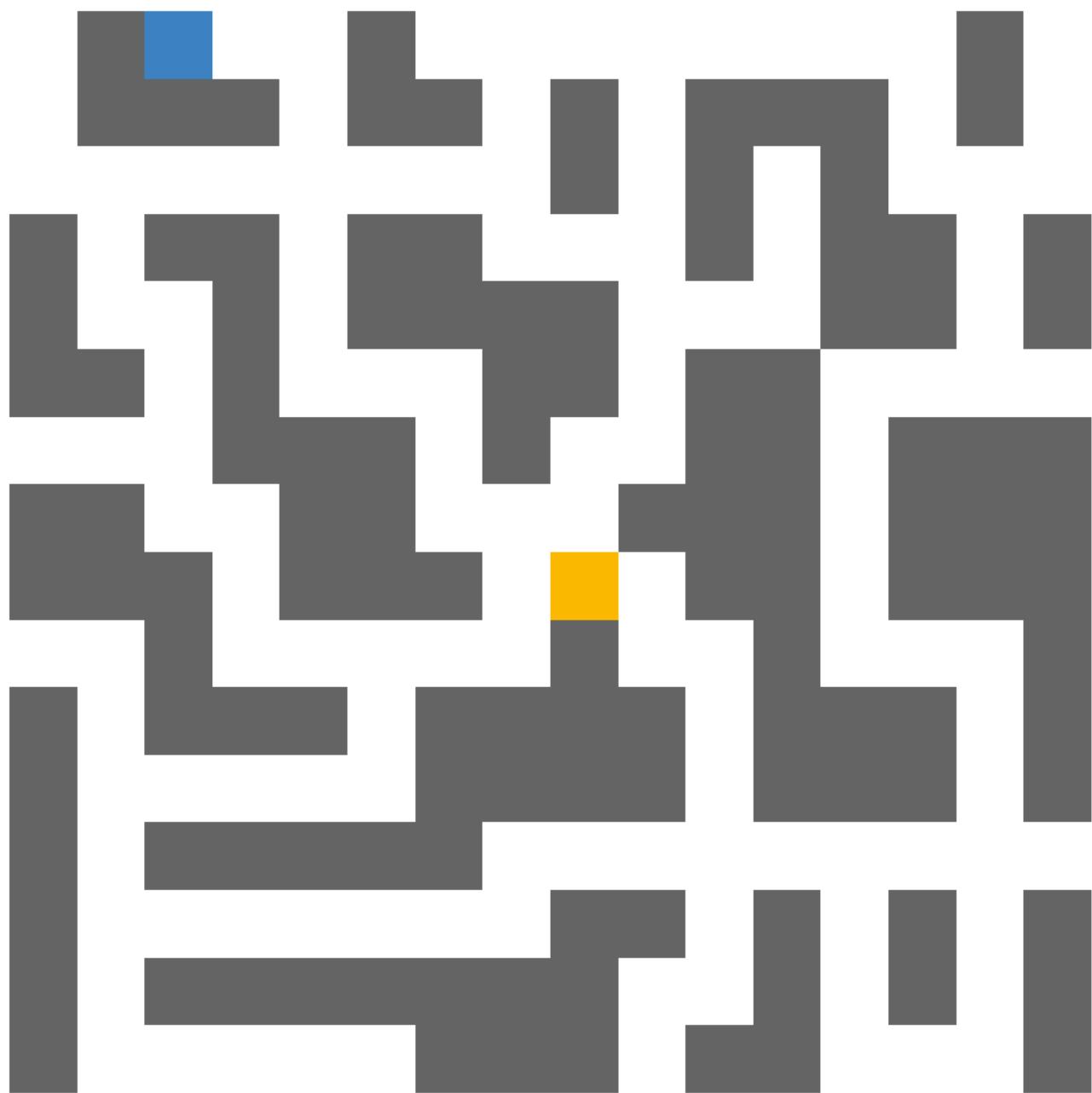


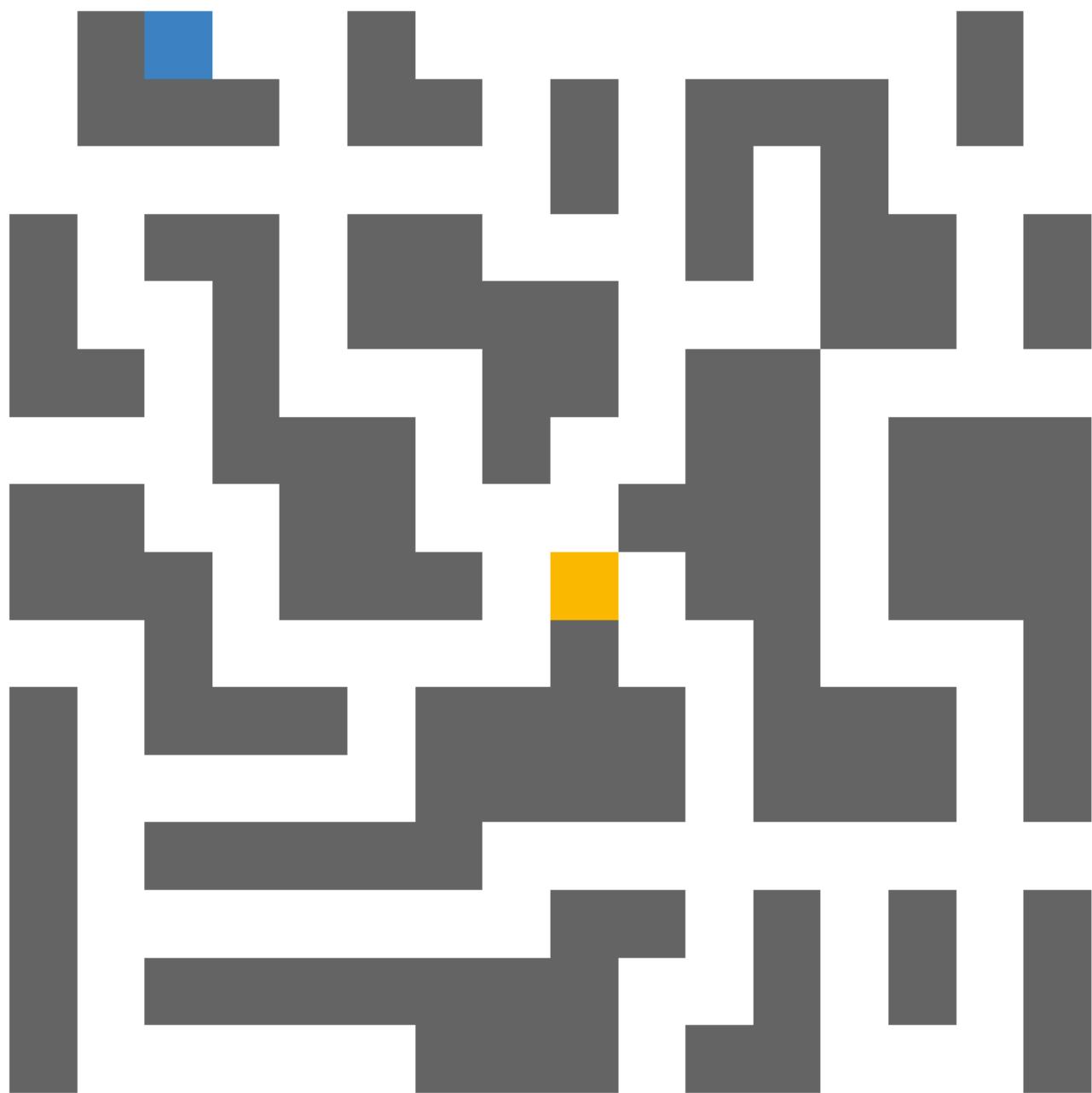


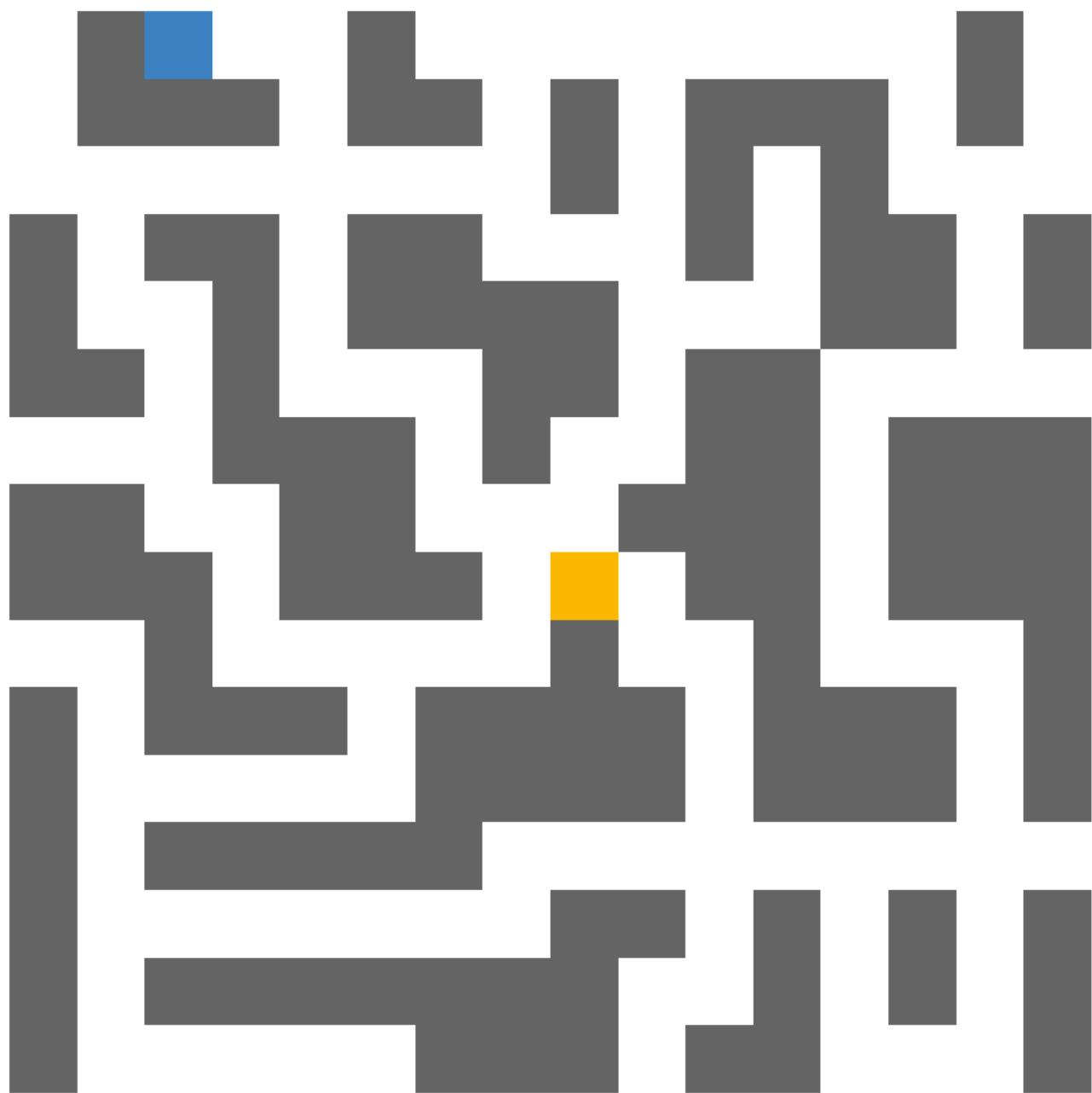


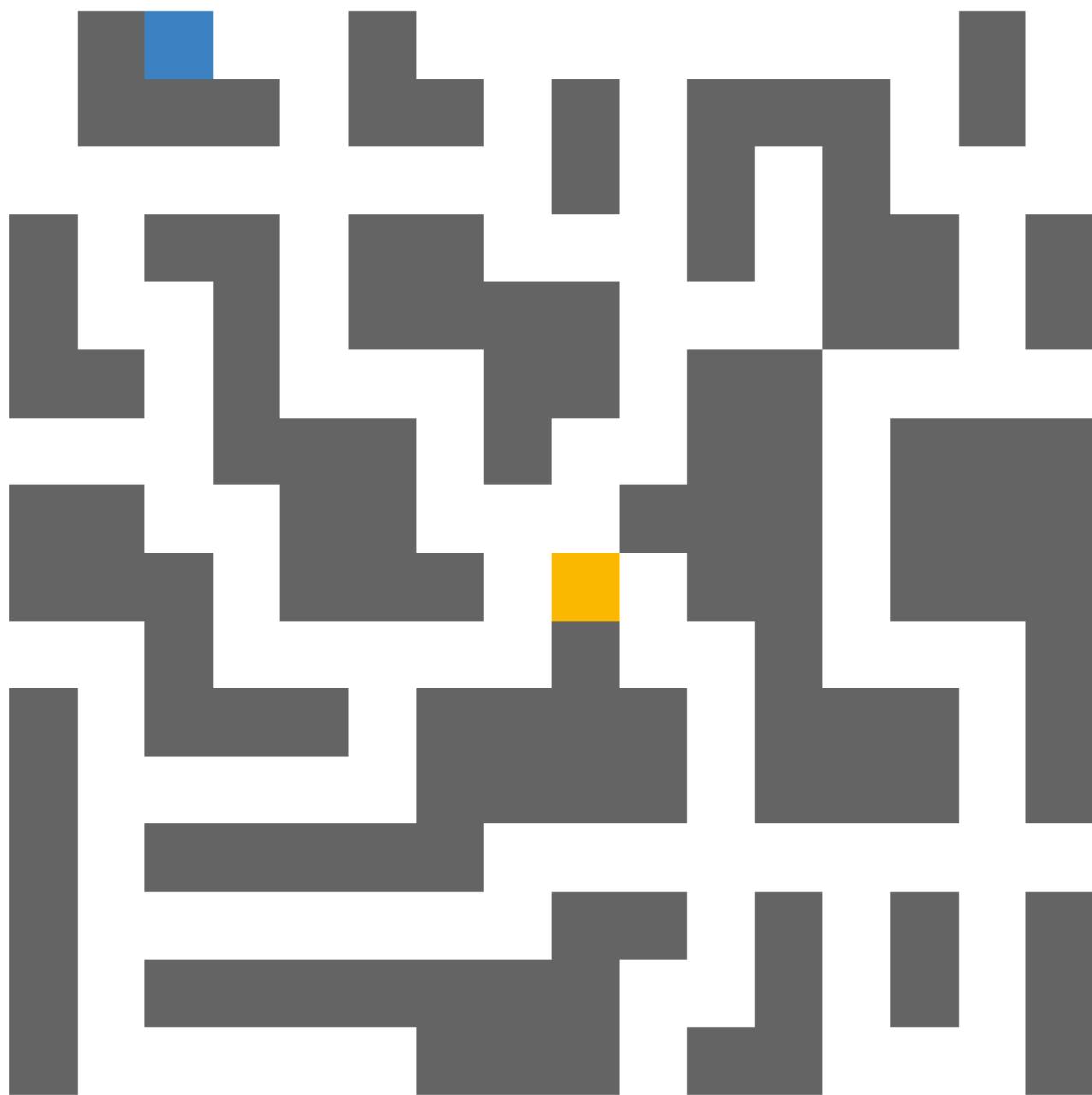












The path is correct and the agent does not move at the wall like it's supposed to do

I should add a `goal_test()` and add some `self.` variables in `__init__()` for `state` and `goal`. This would be better than having a `DataFrame`

Updated `__init__()` with

```
# Initiate variables
self.agent_name=agent_name
self.initial_state=initial_state
self.state=initial_state
self.goal=None

# Added self.state to also get the initial_state
# Generate initial state if not given, and save to self.initial_state and self.state
if self.initial_state==None:
    self.initial_state=self.generate()
    self.state=self.initial_state
```

```

# Find goal for self.goal
for i in range(len(self.initial_state)):
    for j in range(len(self.initial_state[0])):
        if self.initial_state[i][j]==3:
            self.goal=[i,j]

# Commented out DataFrame for Environment for now
#Save data in DataFrame
# self.data=pd.DataFrame(columns=["Initial state","State","Percept"])
# self.data["Initial state"]=initial_state
# self.data["State"]=initial_state
# self.data["Percept"]=percept

```

Added goal\_test()

```

def goal_test(self,state):
    # If the position of the agent is same as goal return True
    if self.find_positions(state,True)==self.goal:
        return True
    else:
        return False

```

Added a goal test in update() to check every time state is updated in case the Agent has reached the goal

```

if self.goal_test(state):
    print("The Agent has reached the goal!")

```

I think I'll add a `self.place_agent()` function to randomly place the agent in the second quadrant because the `Agent` is not supposed to know where it starts, but if it's always at `[0,0]` then that's no fun

Pretty easy to code

```

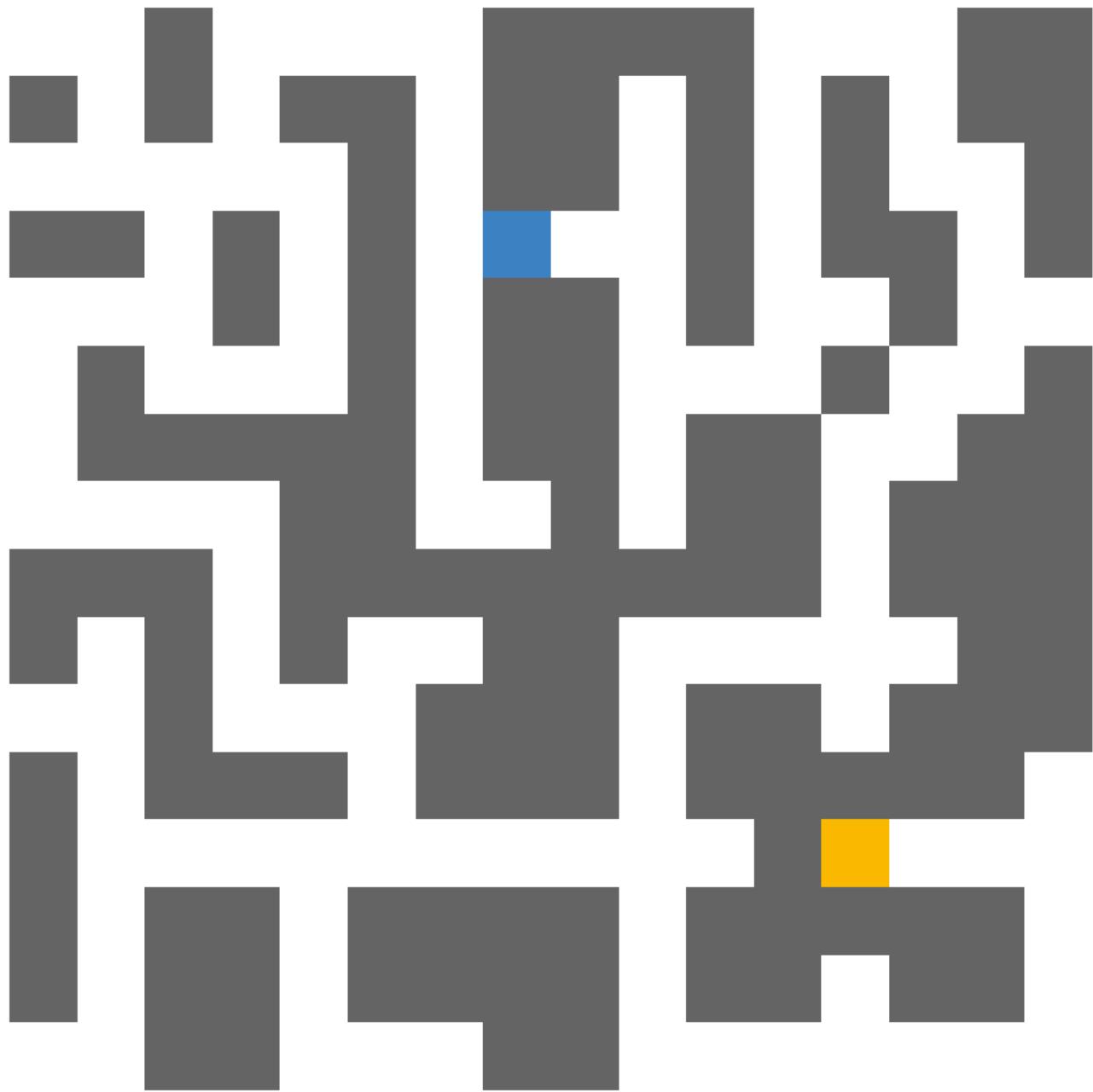
def place_agent(self,state):
    while True:
        i=random.randint(0,7)
        j=random.randint(0,7)

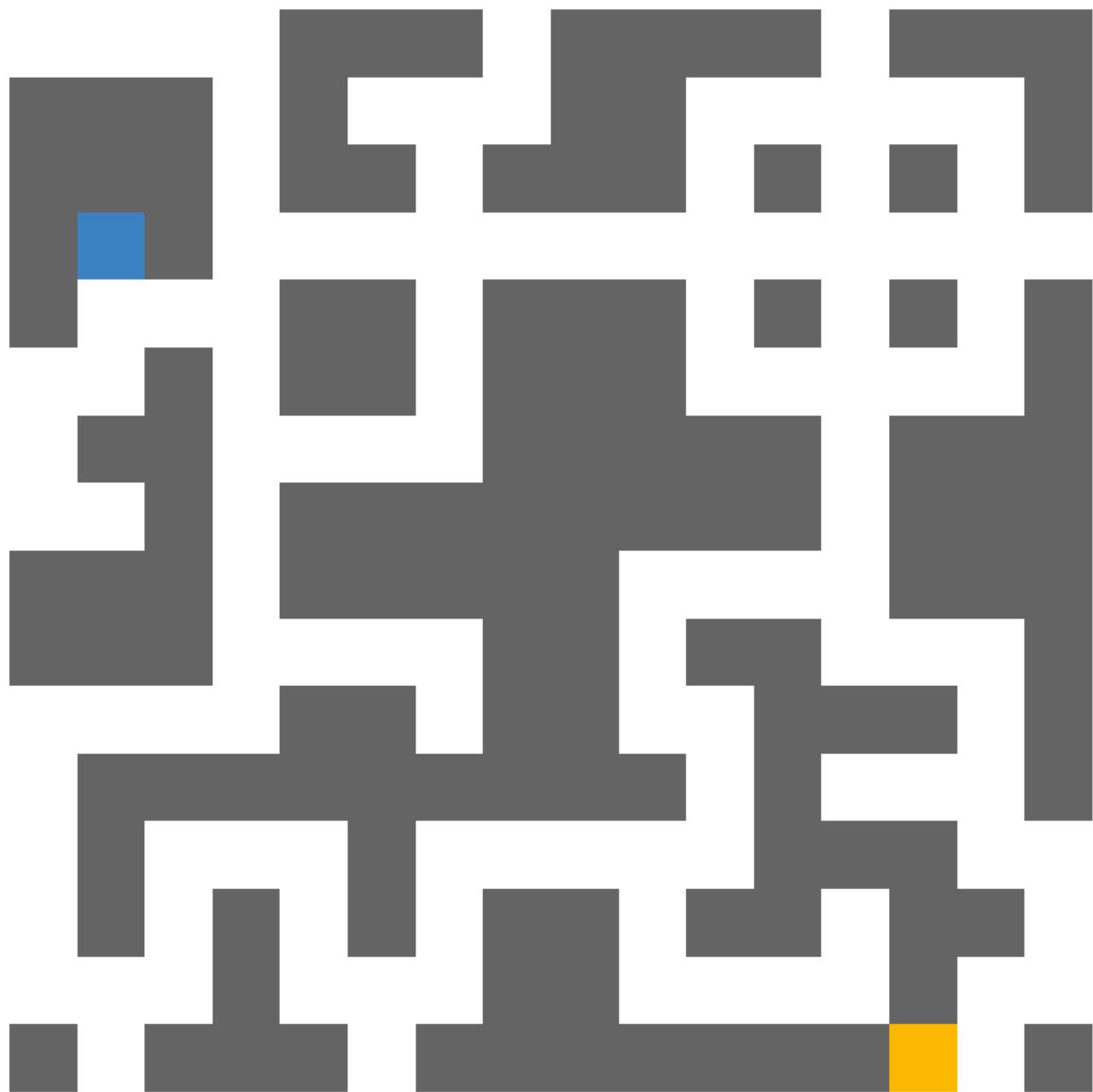
        # If there is a free space, place the agent
        if state[i][j]==0:
            state[i][j]=2
            return state

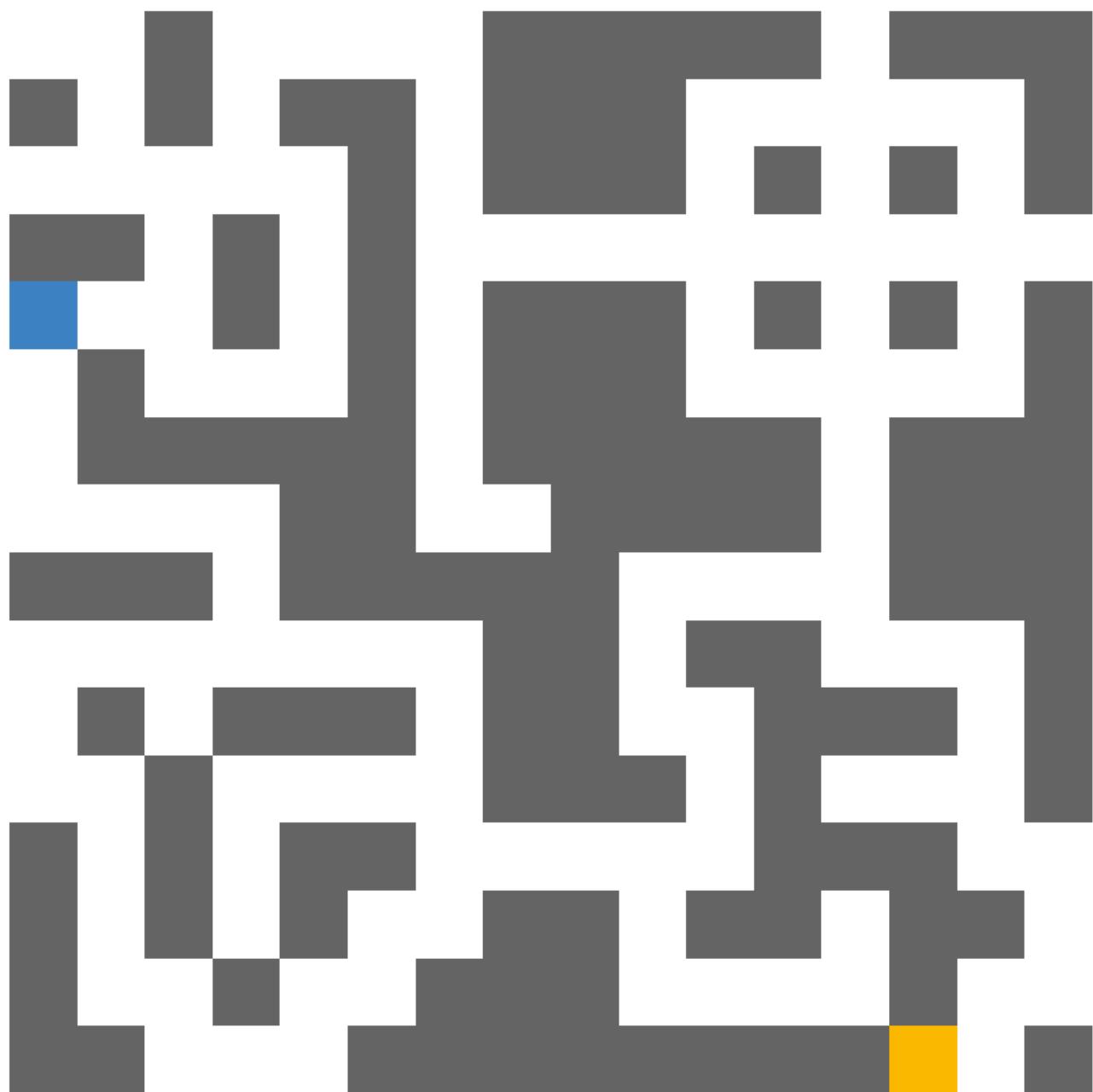
```

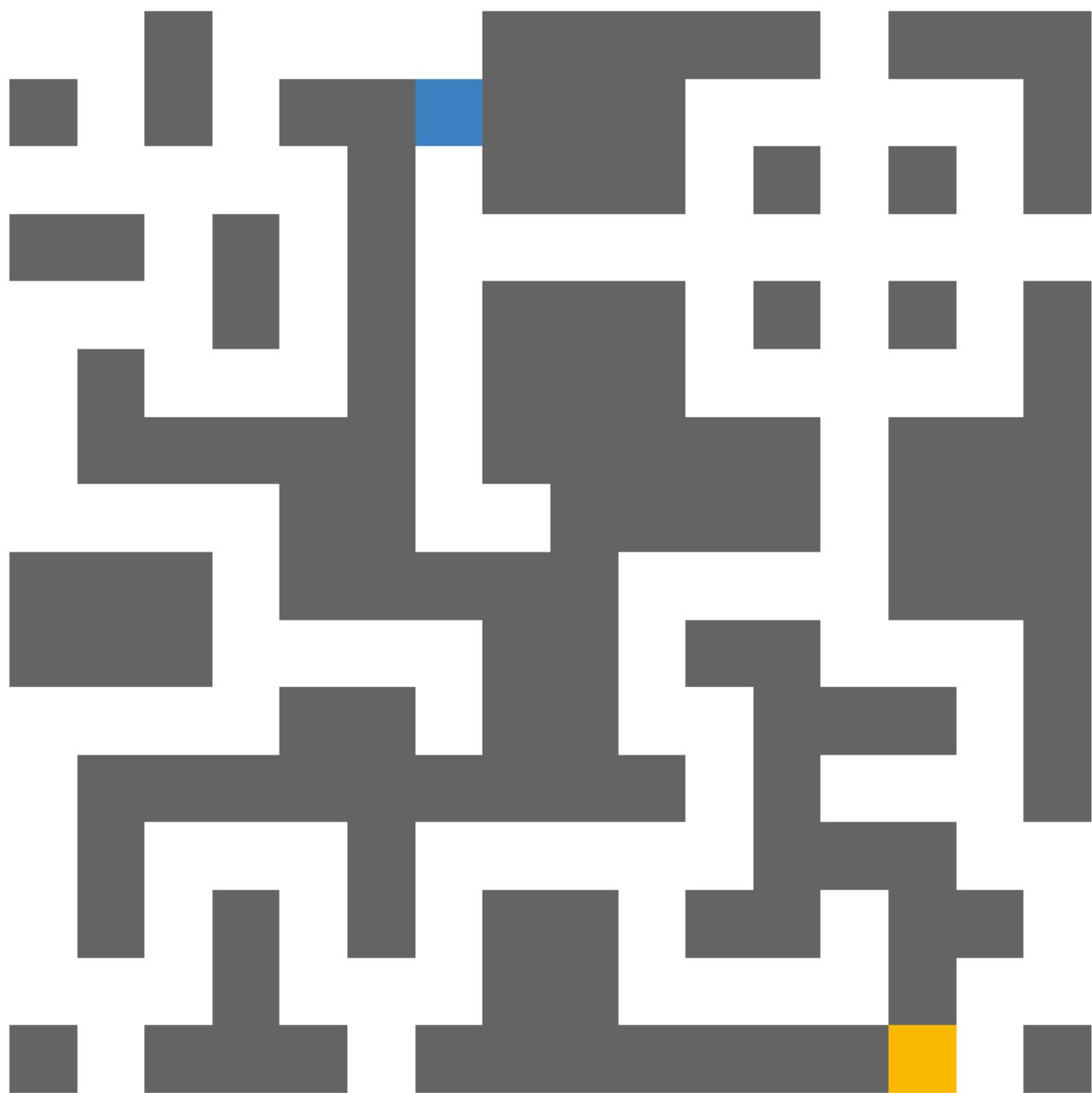
Now we got states like this

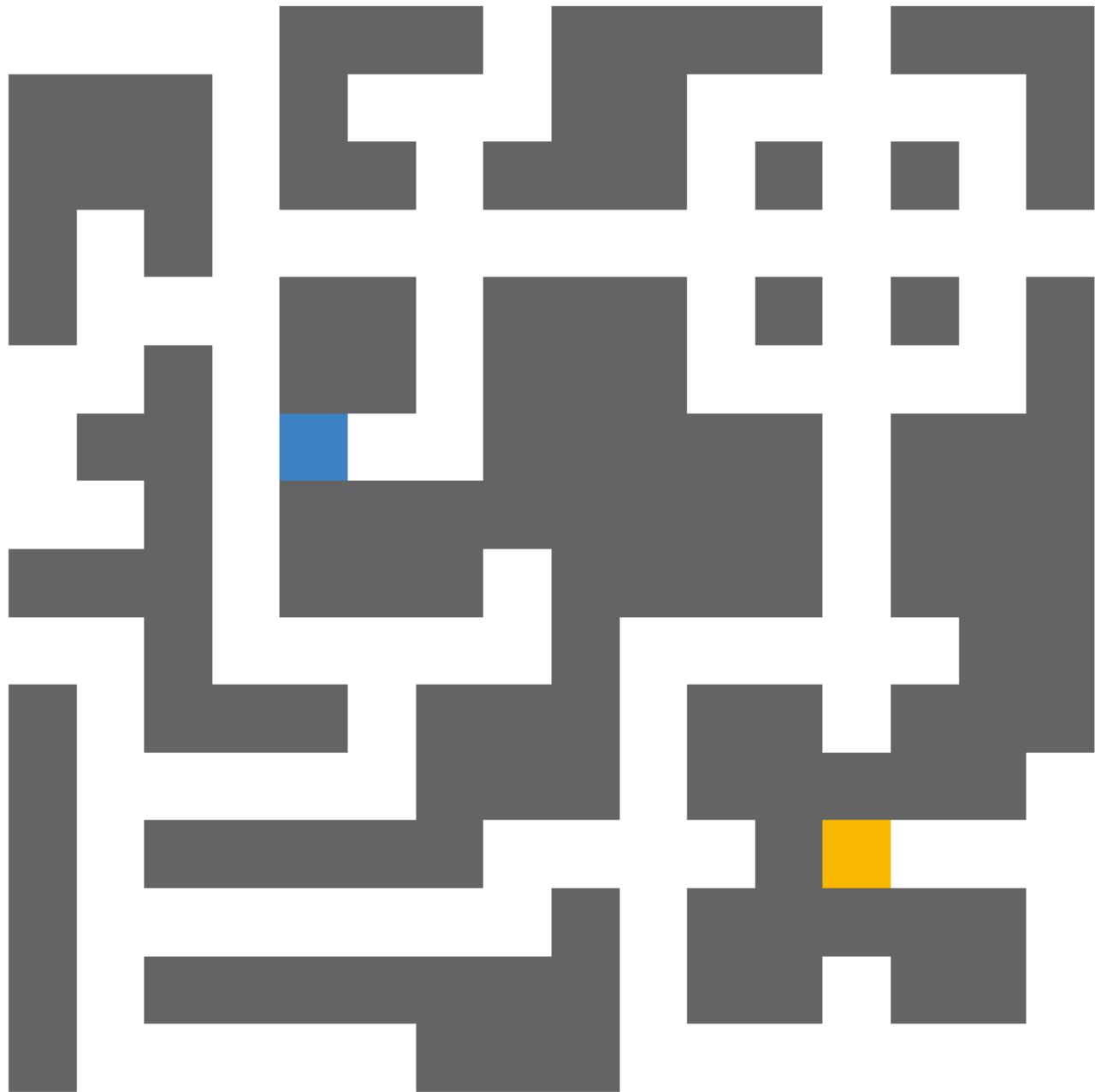
 Random Agent >











If I wanted to, I actually can do the same for the goal

Okay I want to make the goal random as well. I also want to make sure the agent and the goal are a certain distance apart, and in order to do that I would need to know the direct path between them, which means I would need to code the Hermitian. Which will be useful later on so might as well do it now. I'll have to use the [Manhattan distance](#).

First I will make a new branch called `Manhattan`

I'll replace the range for `place_agent()` to be the entire map

```
i=random.randint(0,15)
j=random.randint(0,15)
```

Copy and pasted to create `place_goal()`

```
def place_goal(self,state):
    while True:
```

```

i=random.randint(0,15)
j=random.randint(0,15)

# If there is a free space, place the goal
if state[i][j]==0:
    state[i][j]=3
    return state

```

Going to have a `place_agent_and_goal()` function that includes `place_agent()`, `place_goal()`, and `h_distance()`

The [Manhattan distance](#) formula is pretty simple, the formula is

so in Python it's

```
abs(position1[0]-position2[0])+abs(position1[1]-position2[1])
```

So `h_distance(self,position1,position2)` is

```
def h_distance(self,position1,position2):
    return abs(position1[0]-position2[0])+abs(position1[1]-position2[1])
```

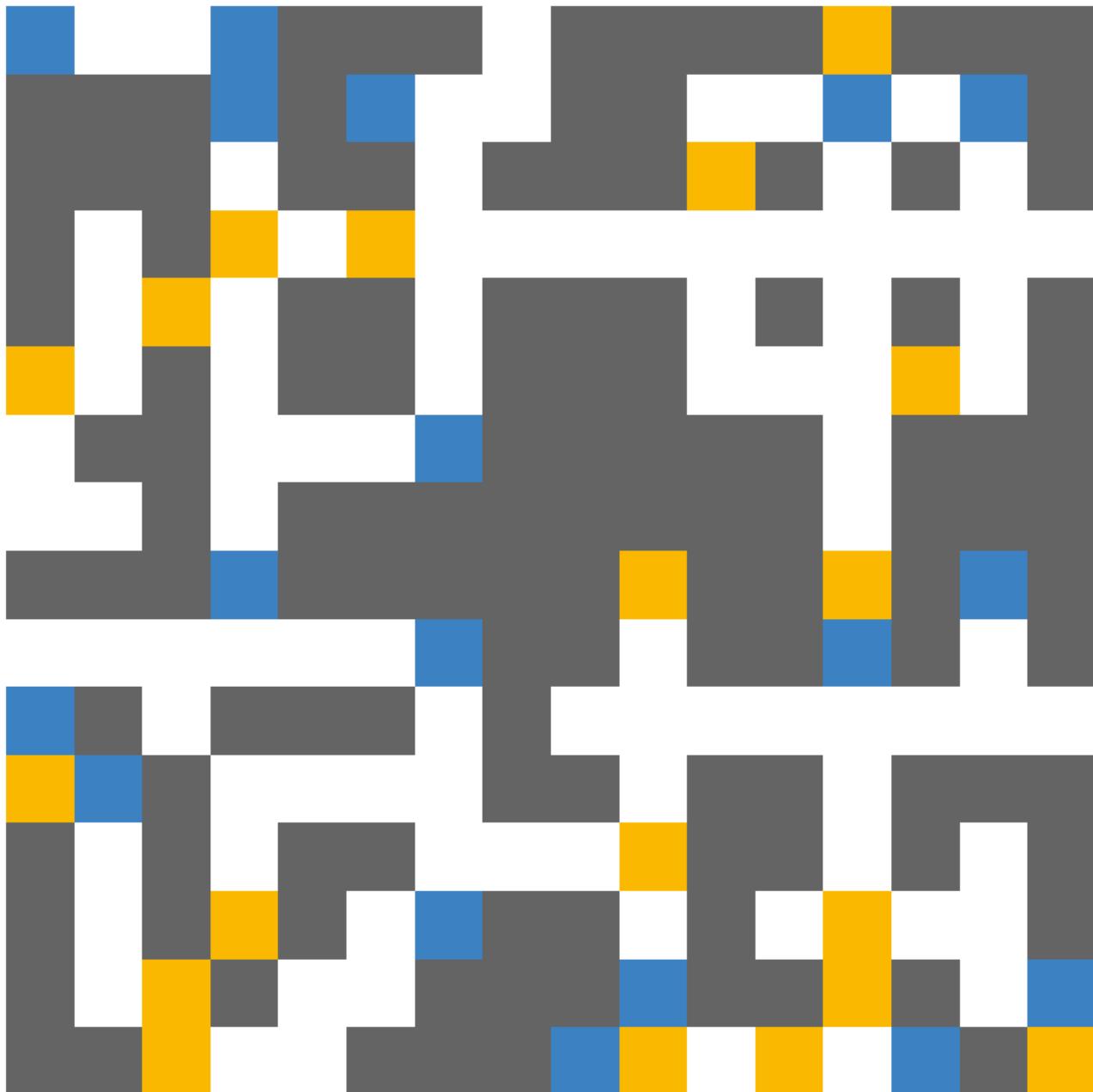
In order to know the goal or agent was placed, I will add an operation to `find_positions()` in order to find the goal as well if needed. Now there will be

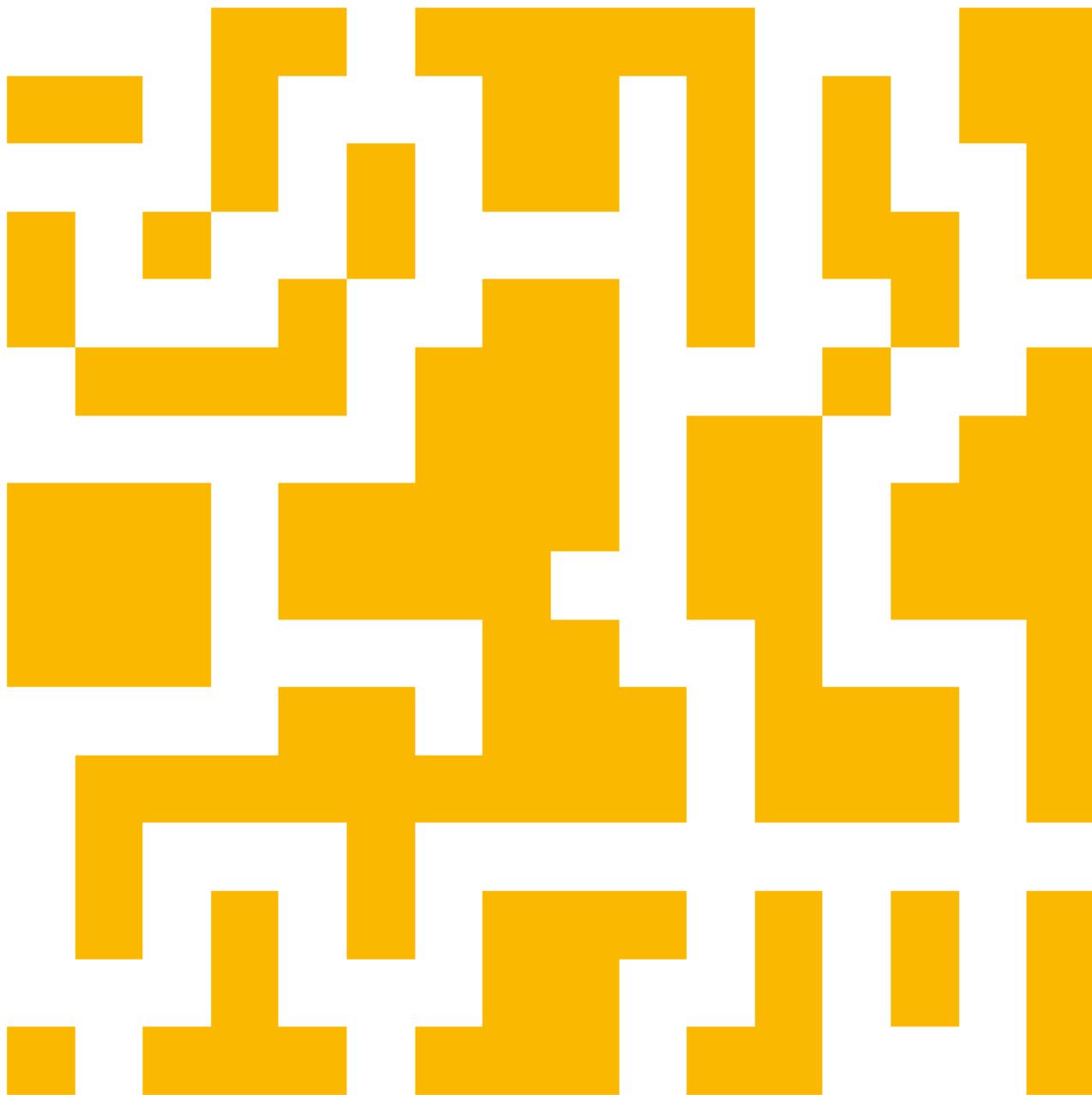
```
find_positions(self,state,first=False,find_goal=False)
```

with

```
if find_goal:
    piece_type=3
else:
    piece_type=2
```

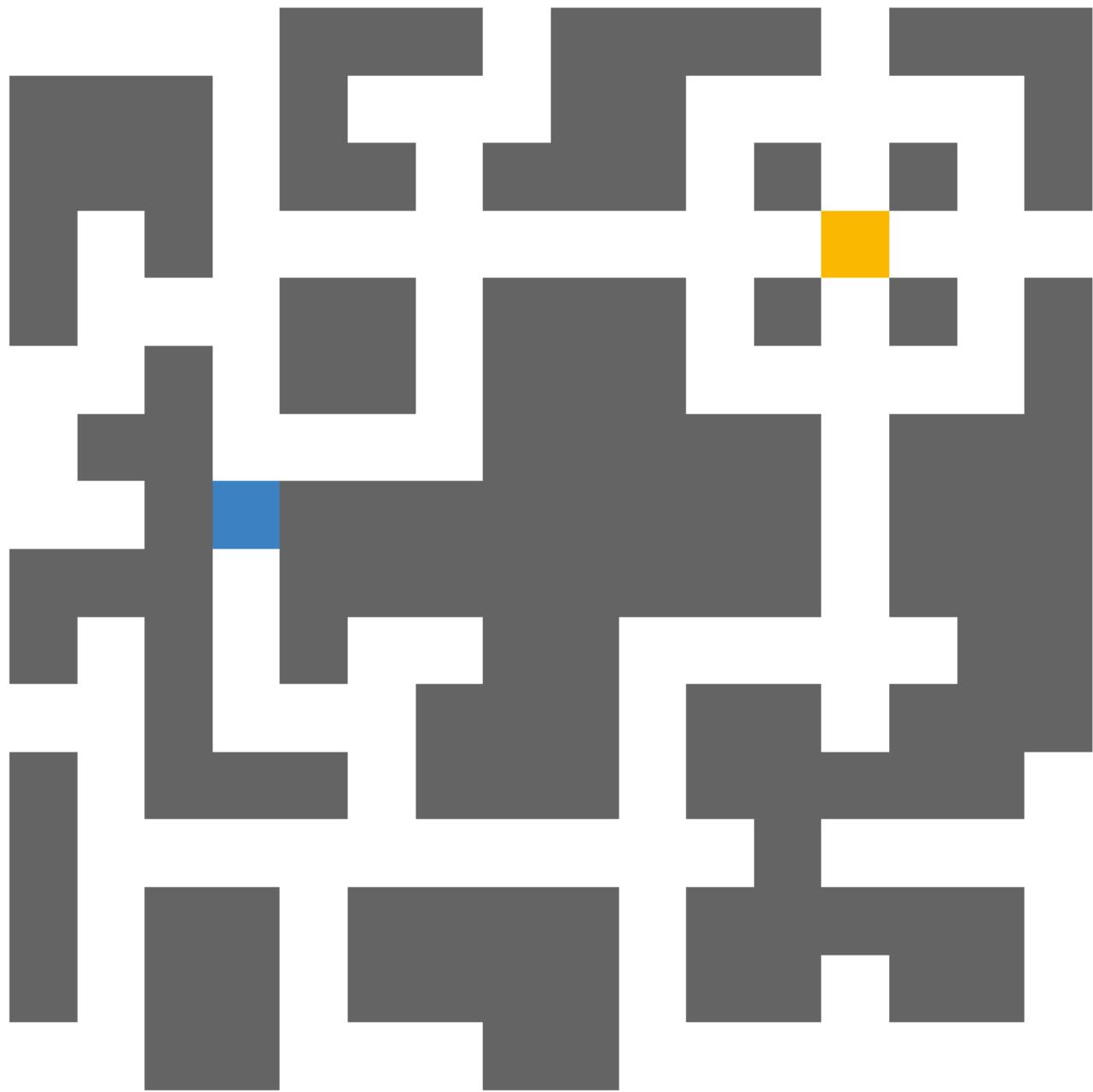
Errors were made



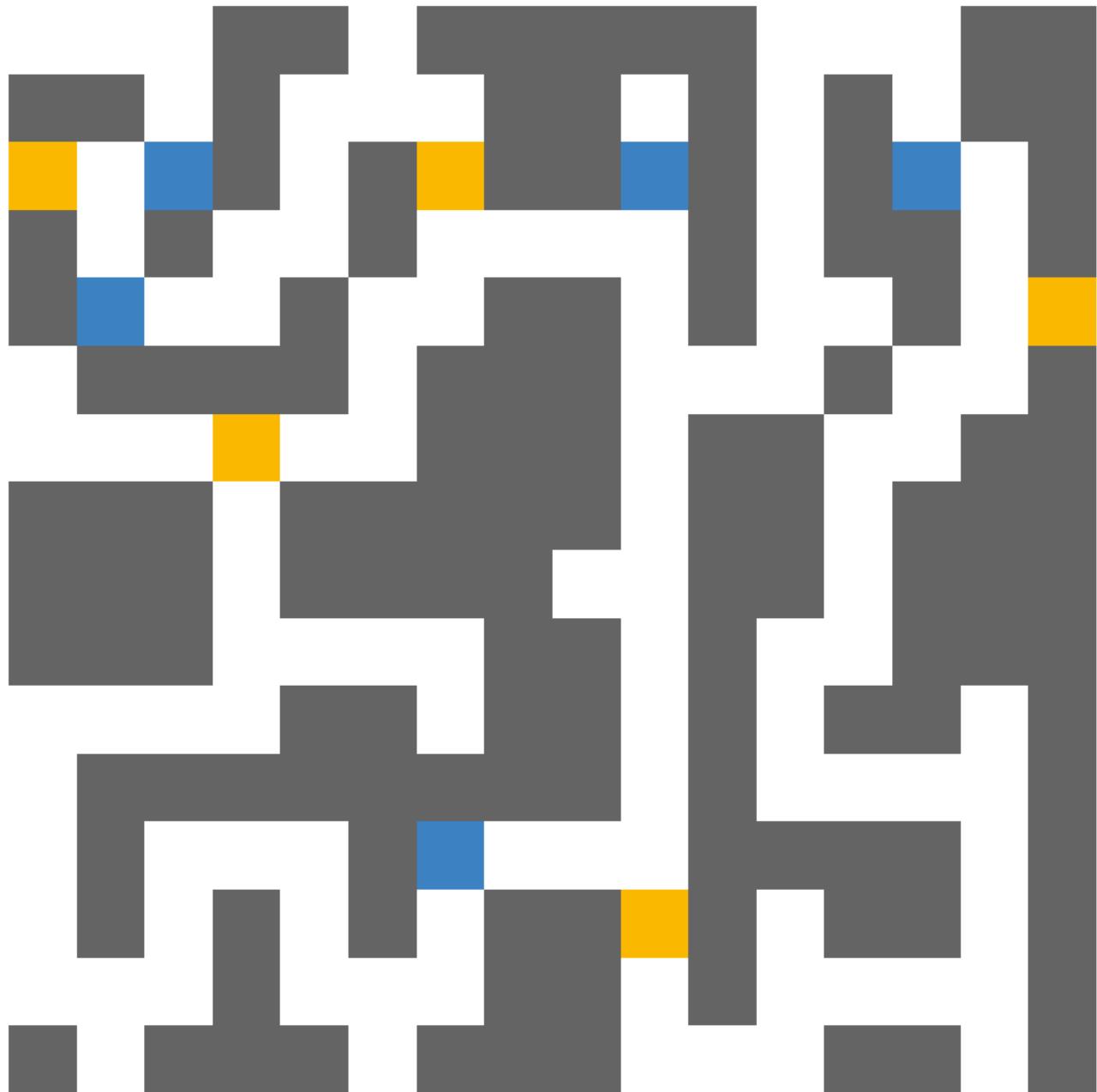


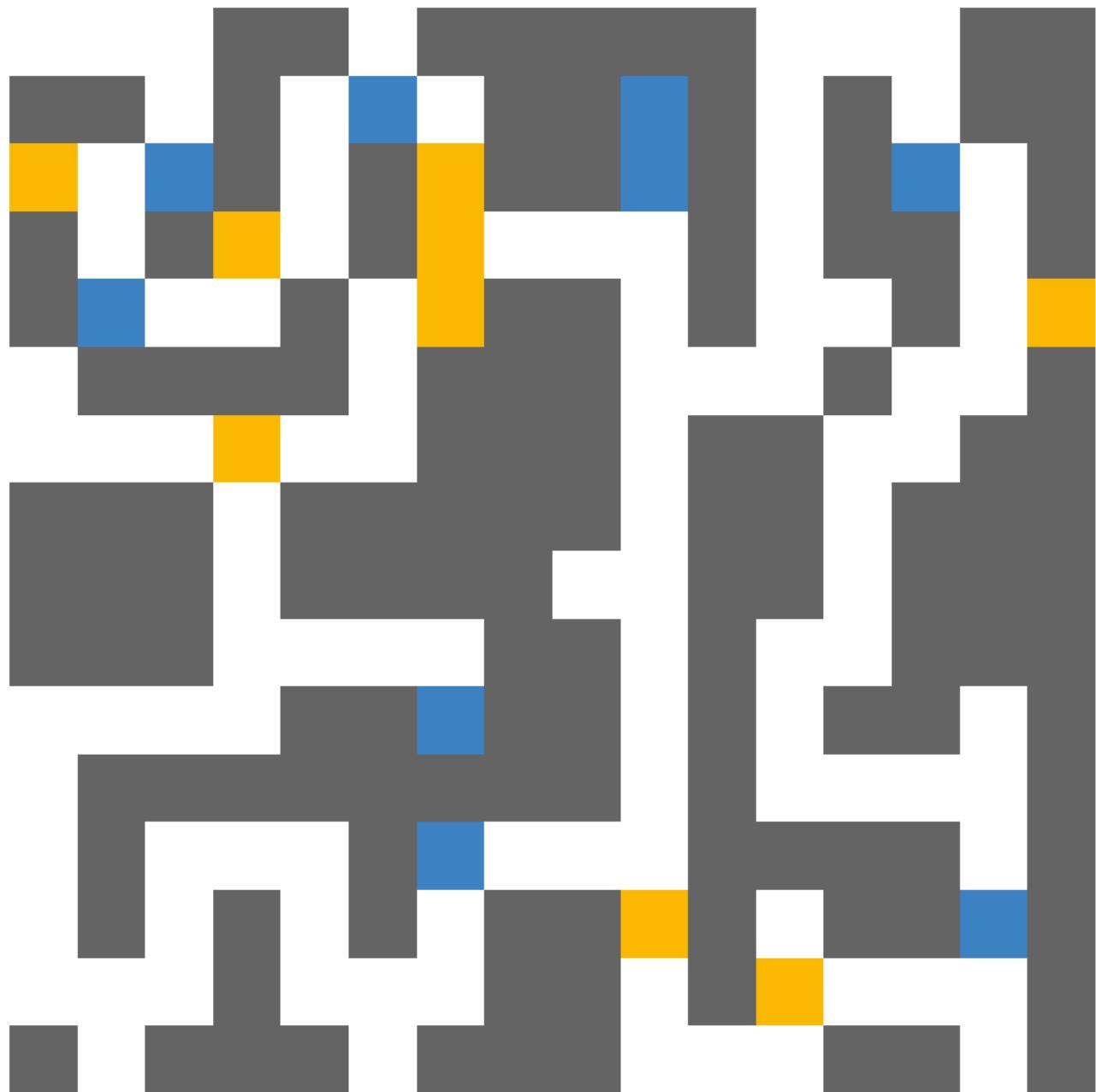
Interesting glitch

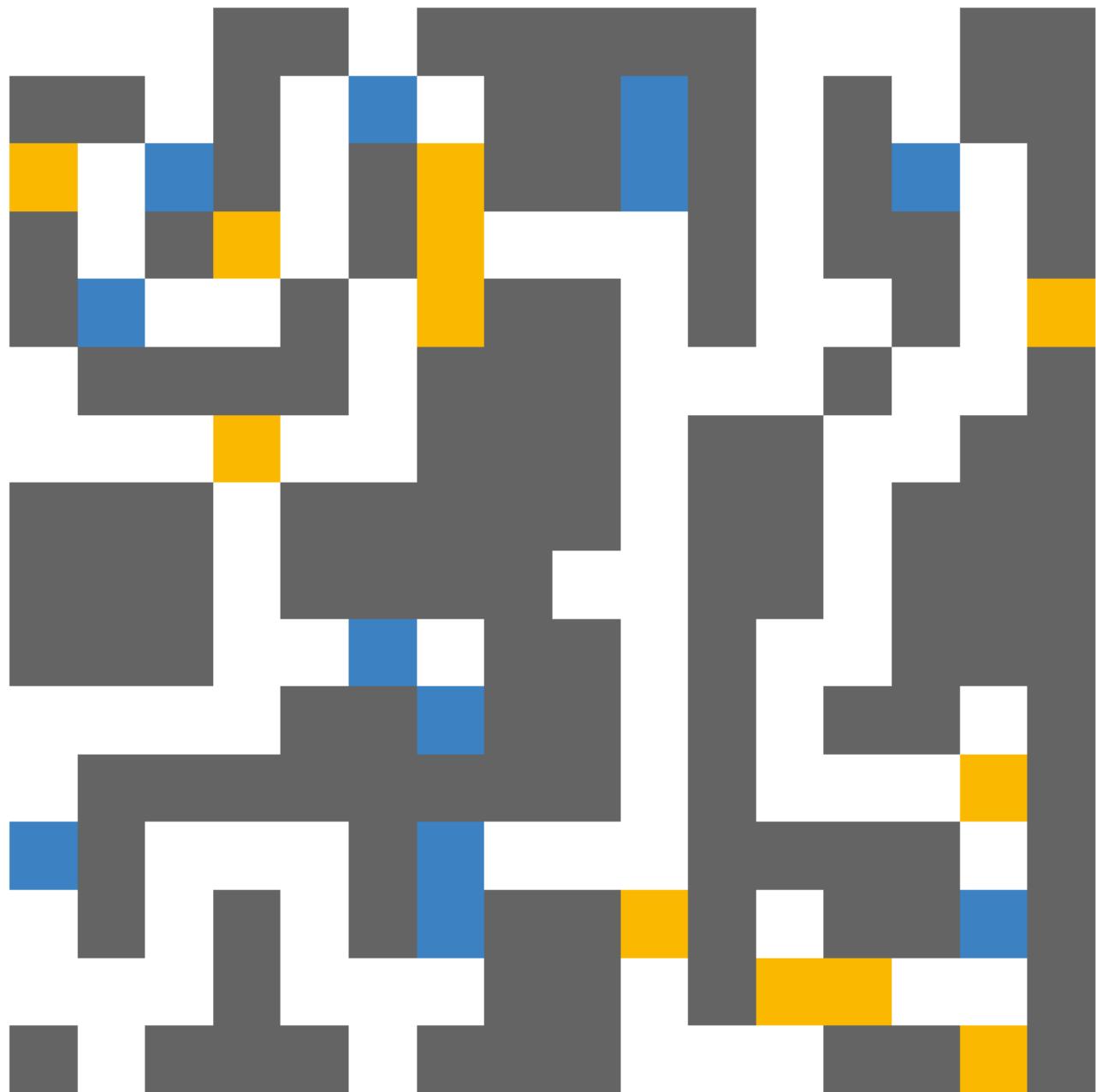
 [Accumulating agents and goals >](#)

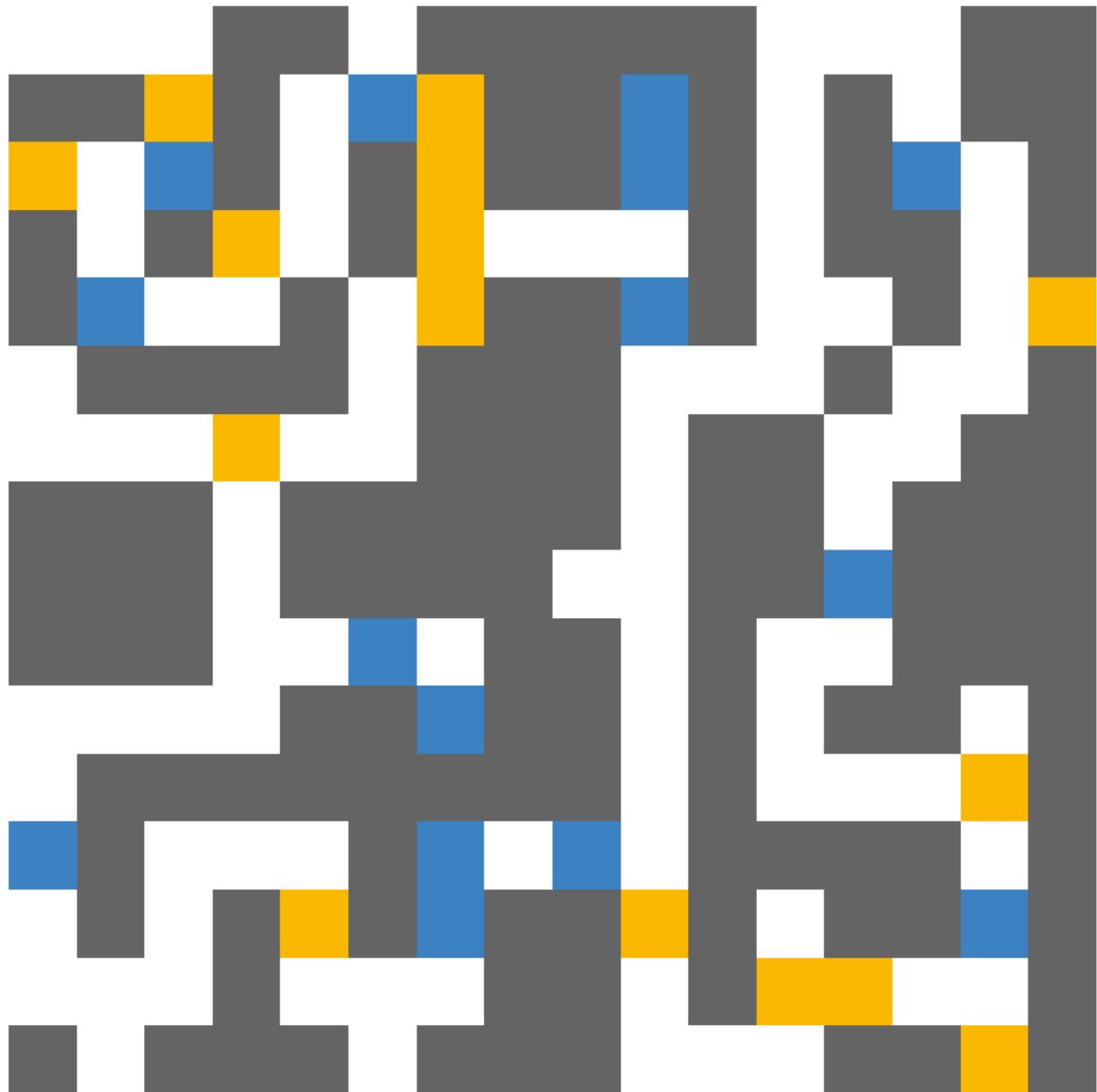


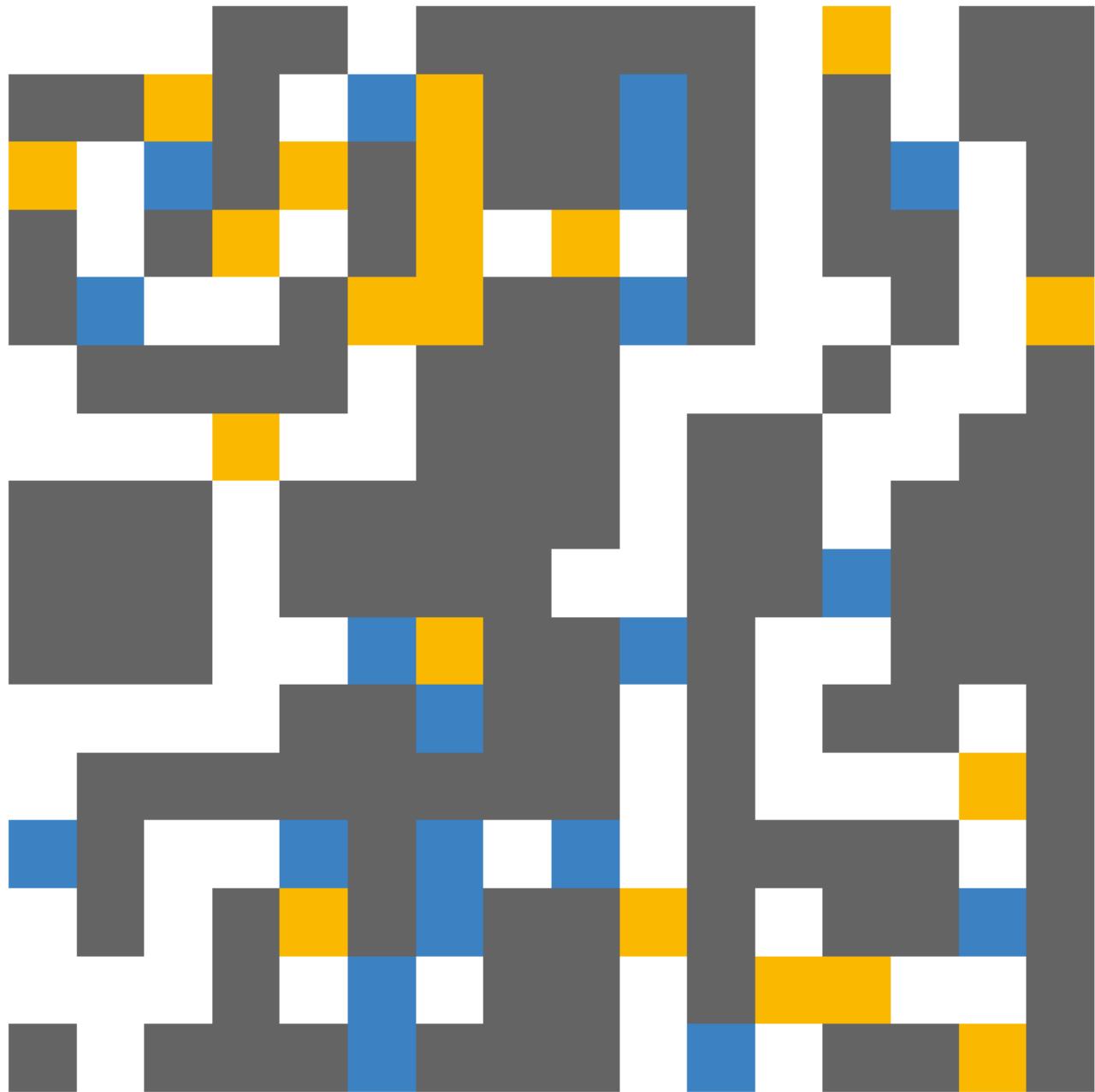


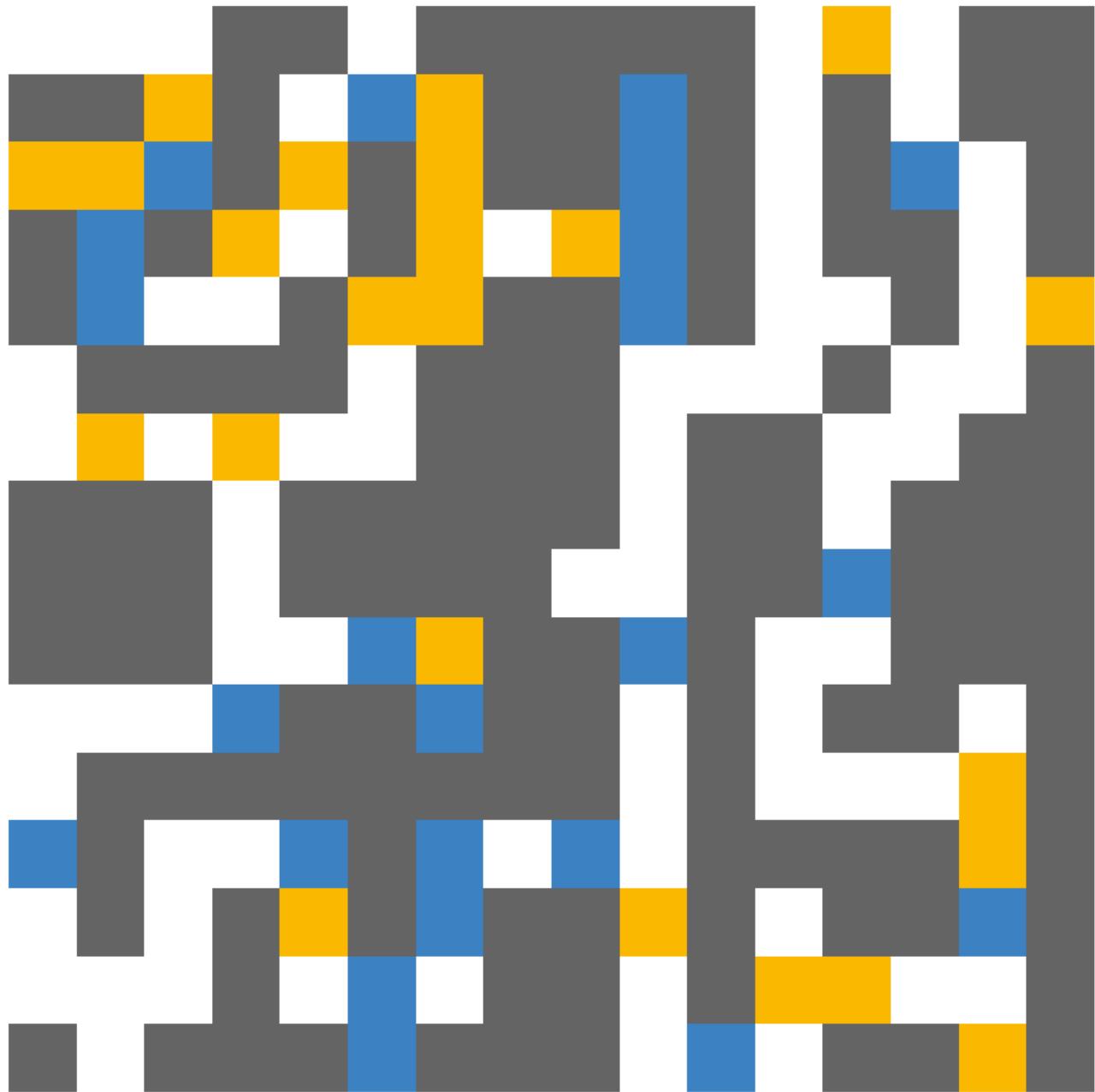


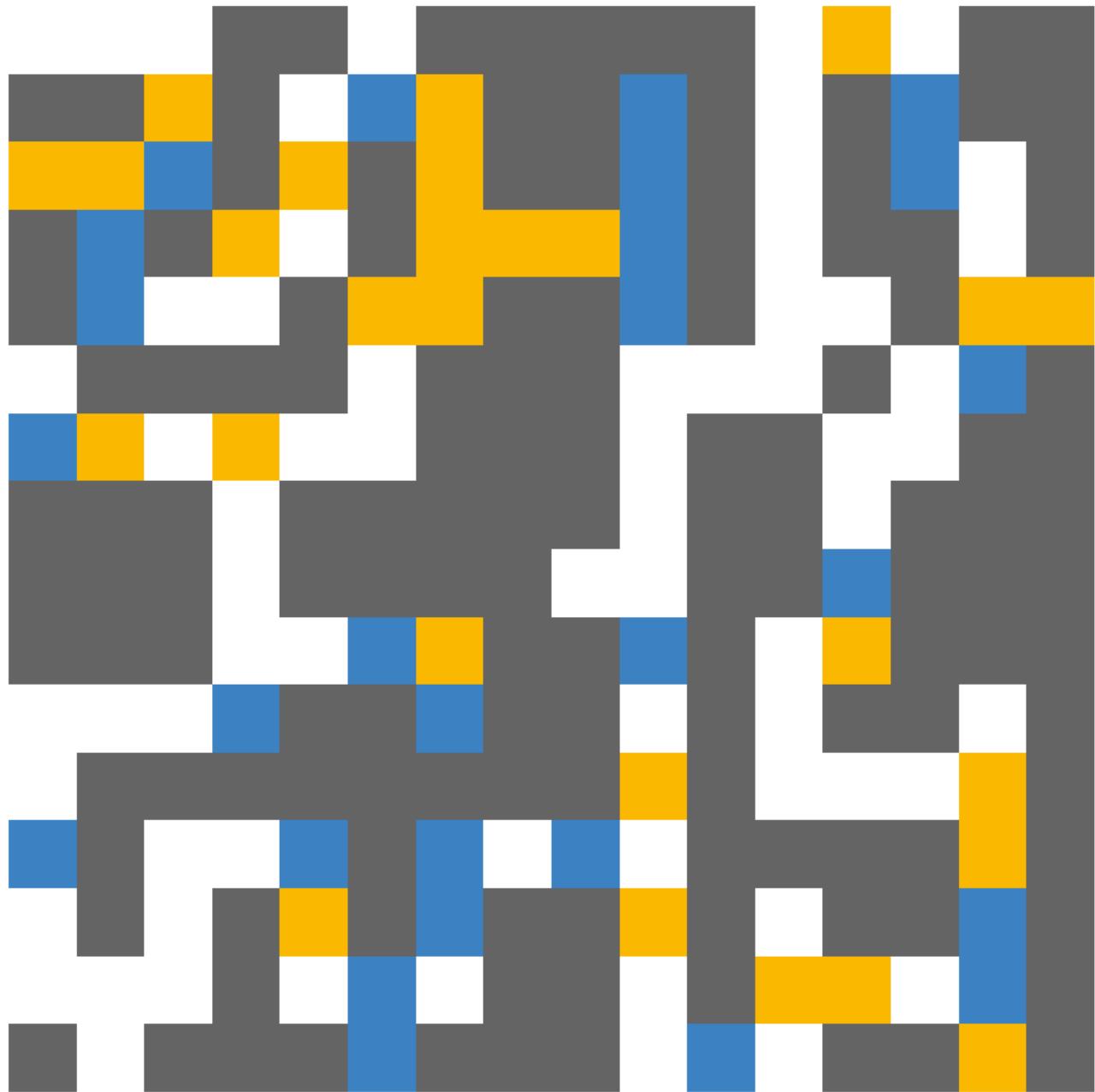


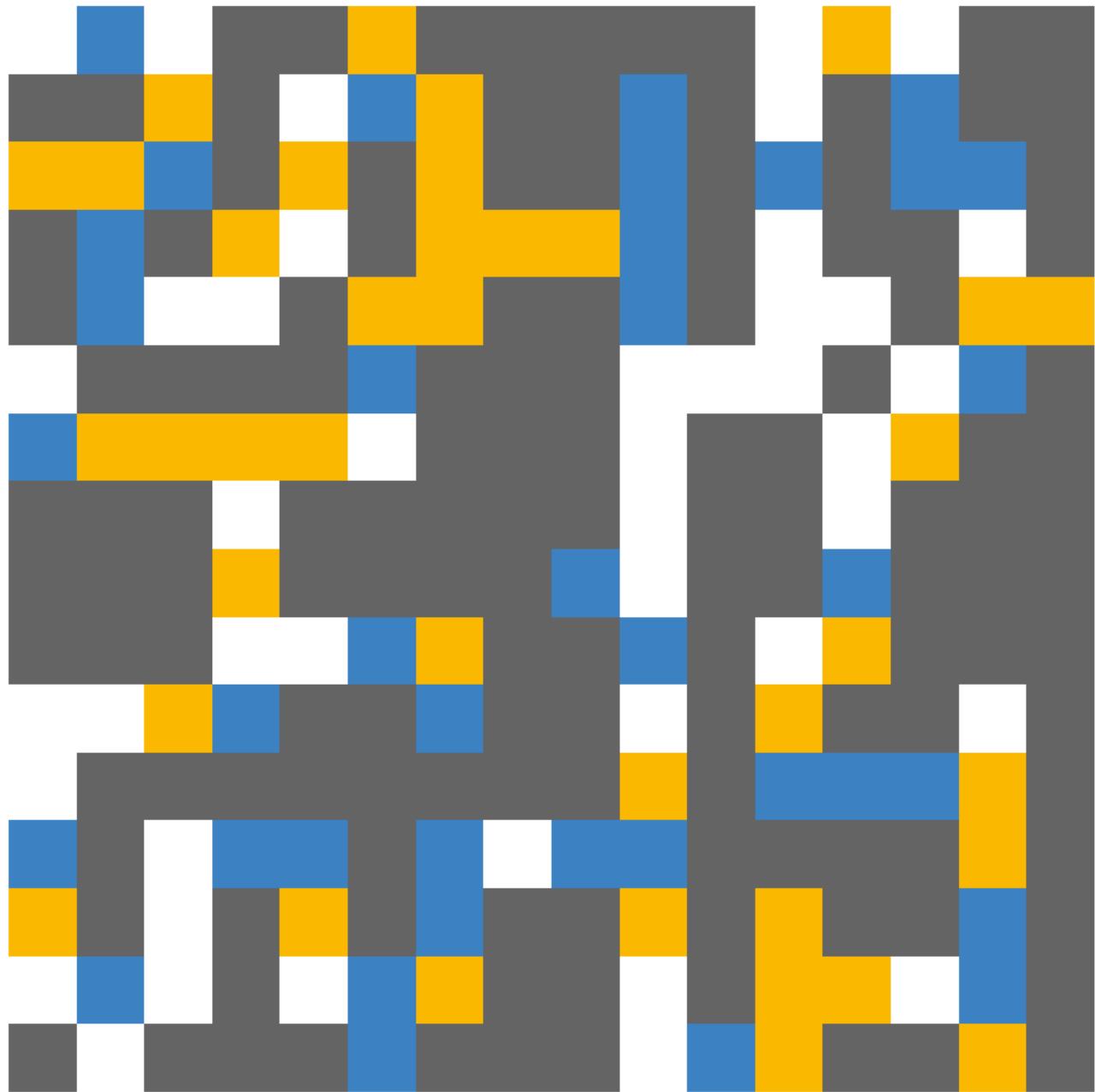


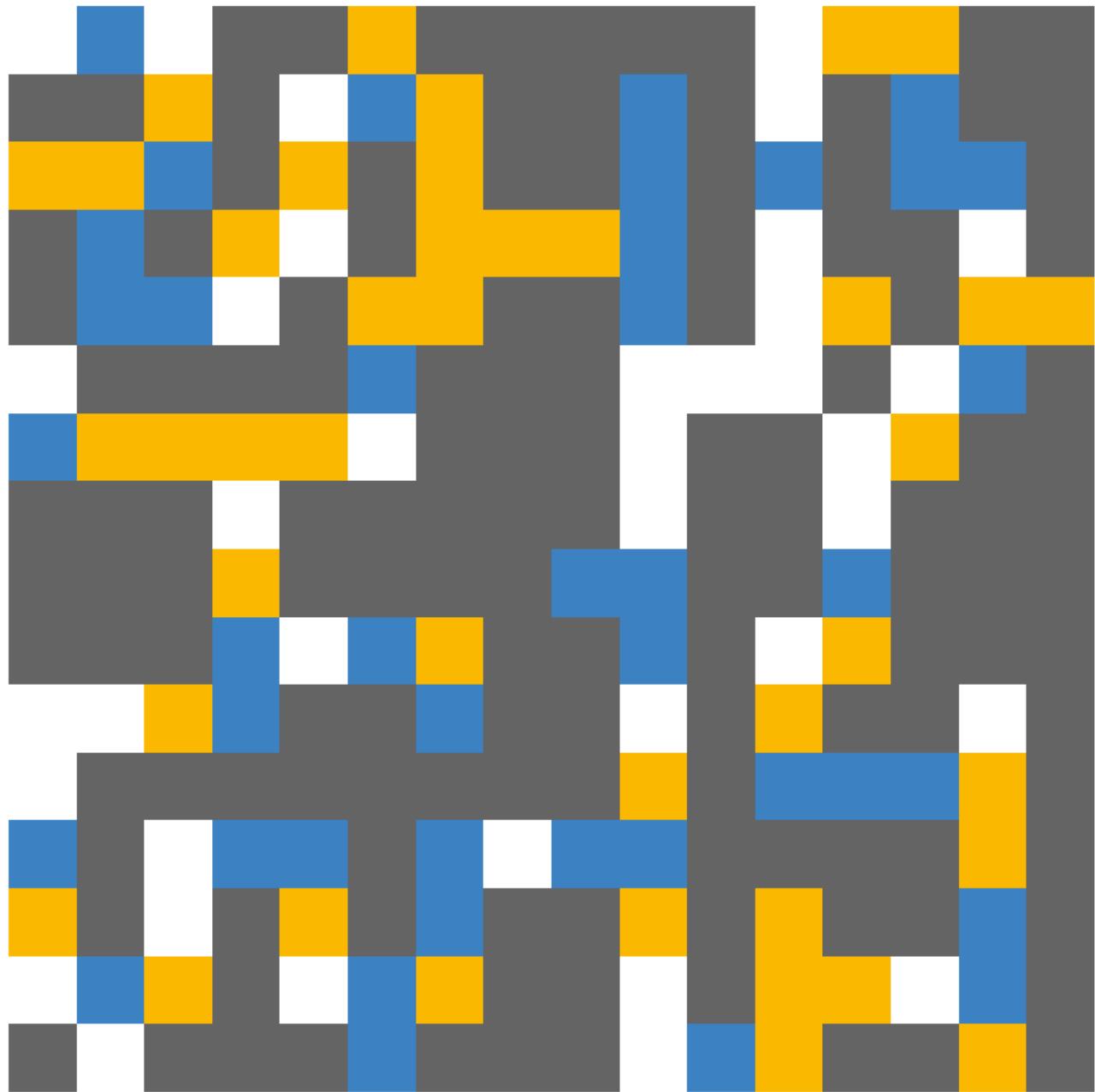


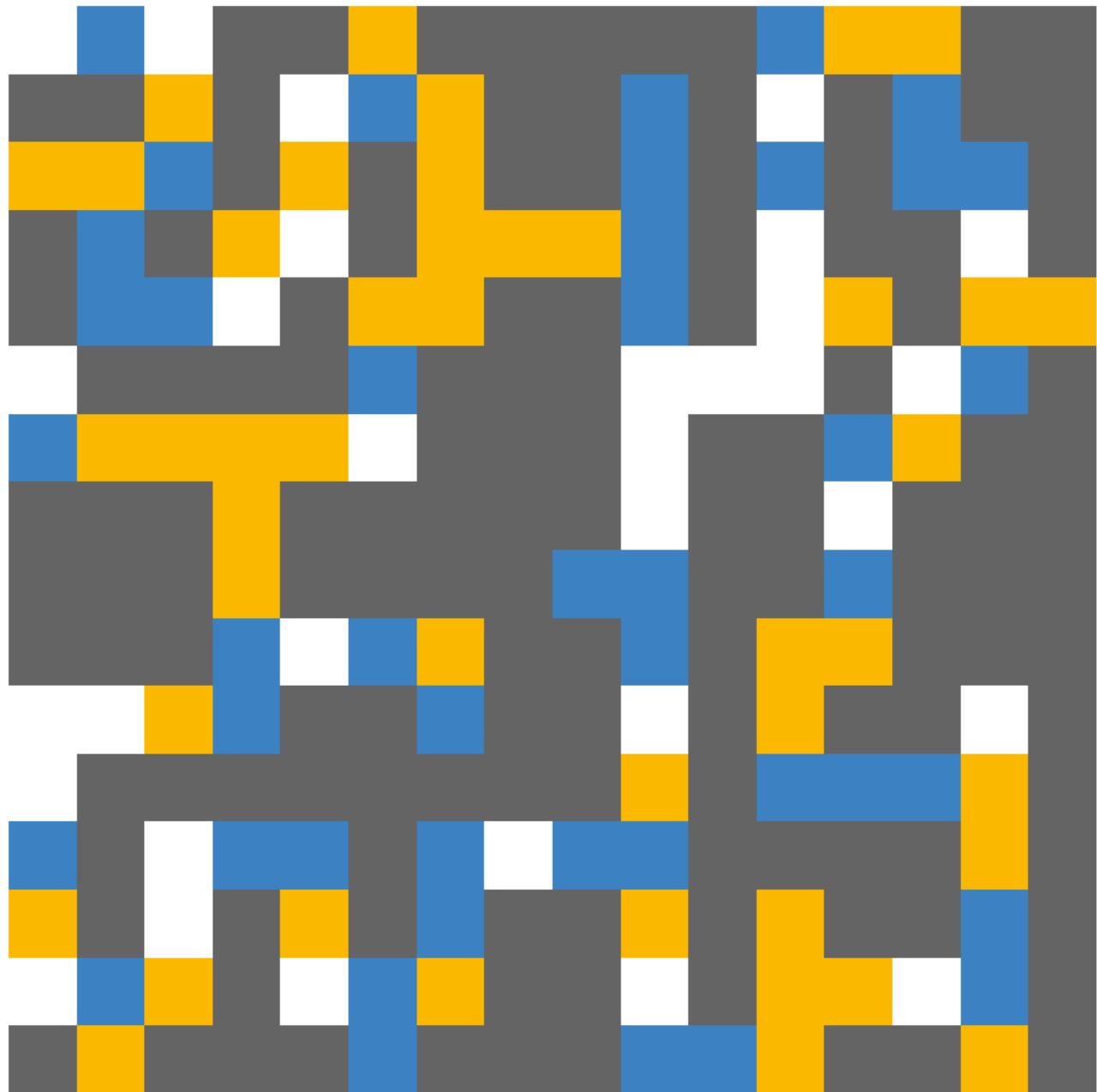


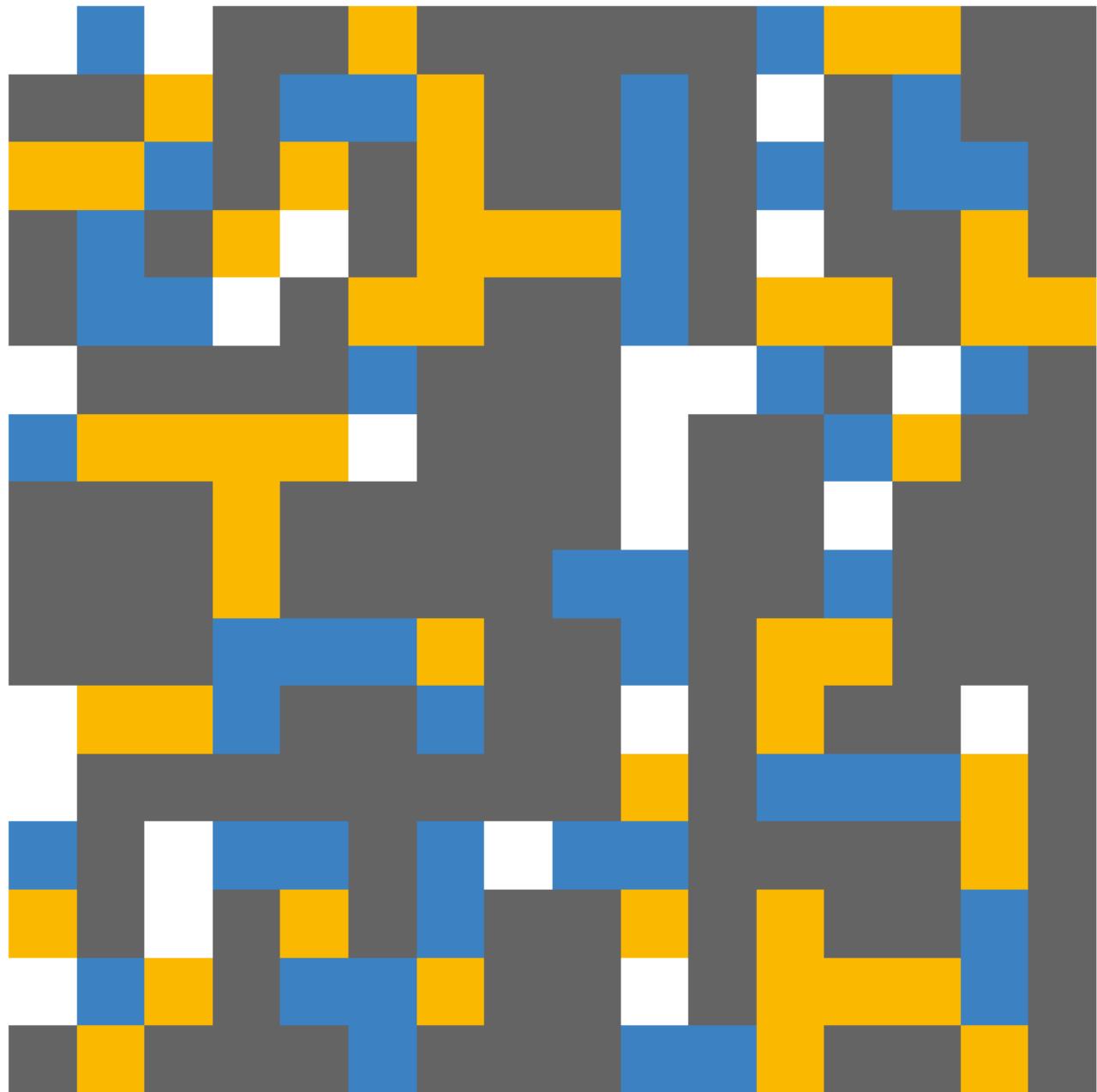


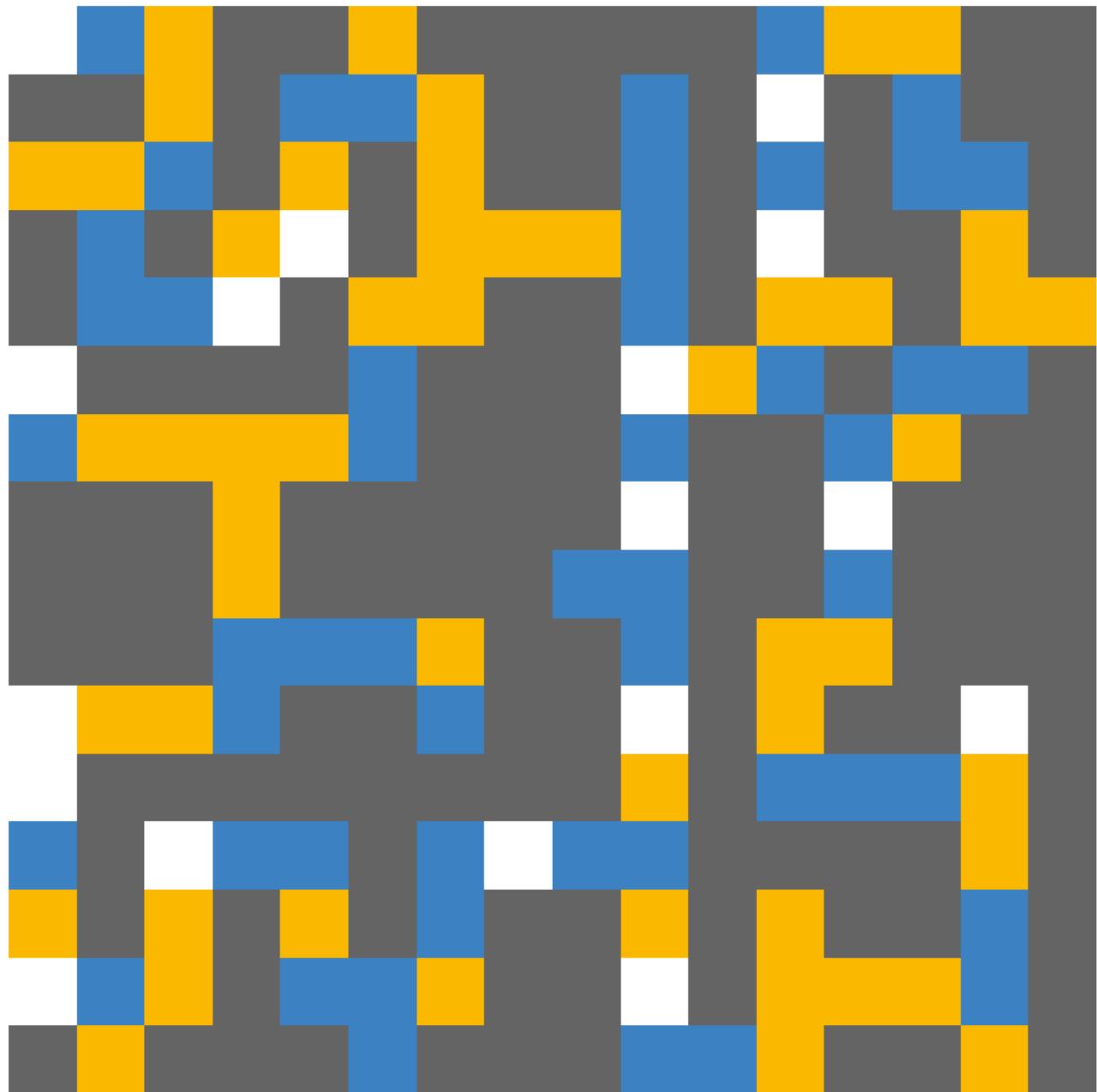


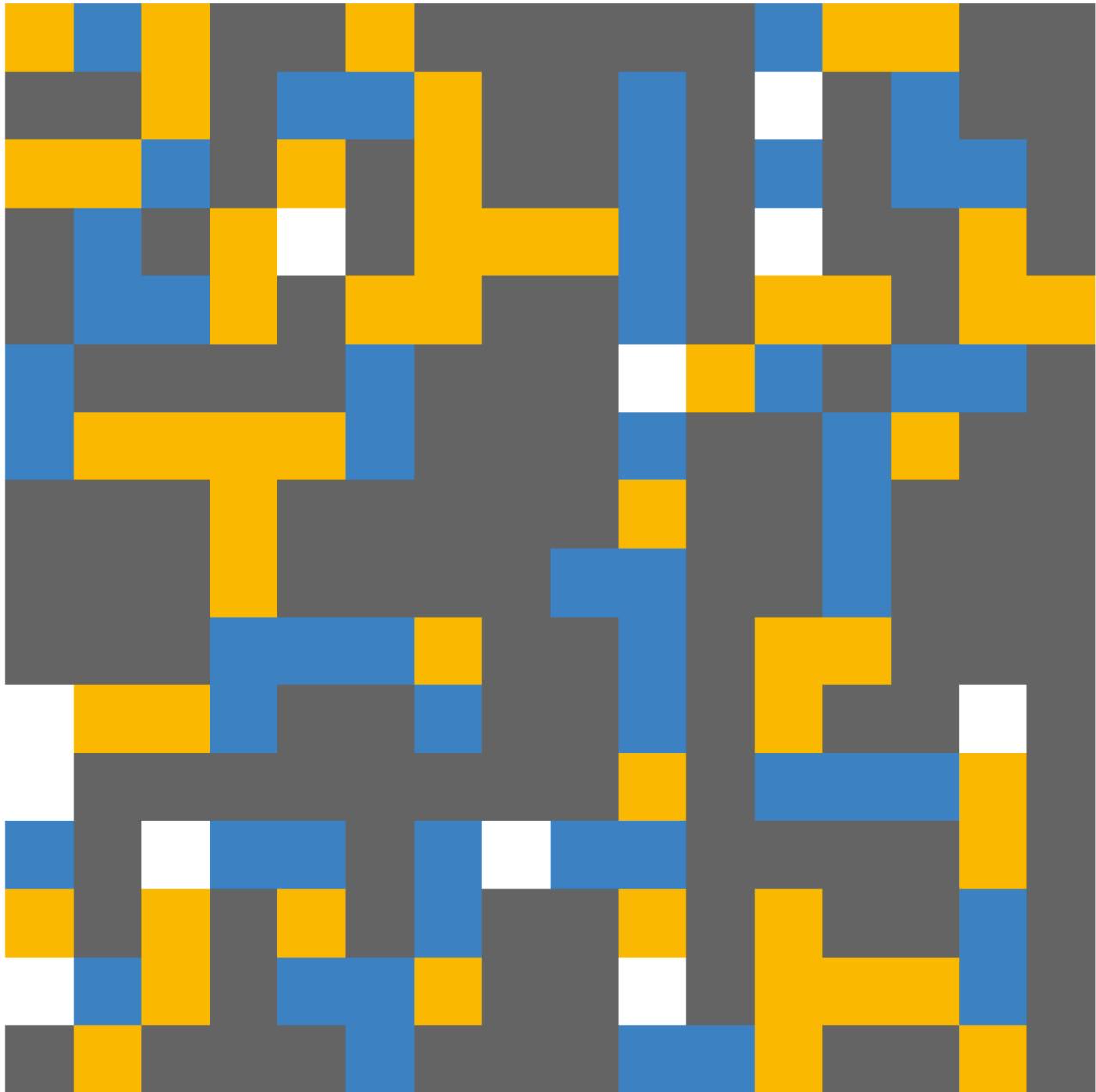


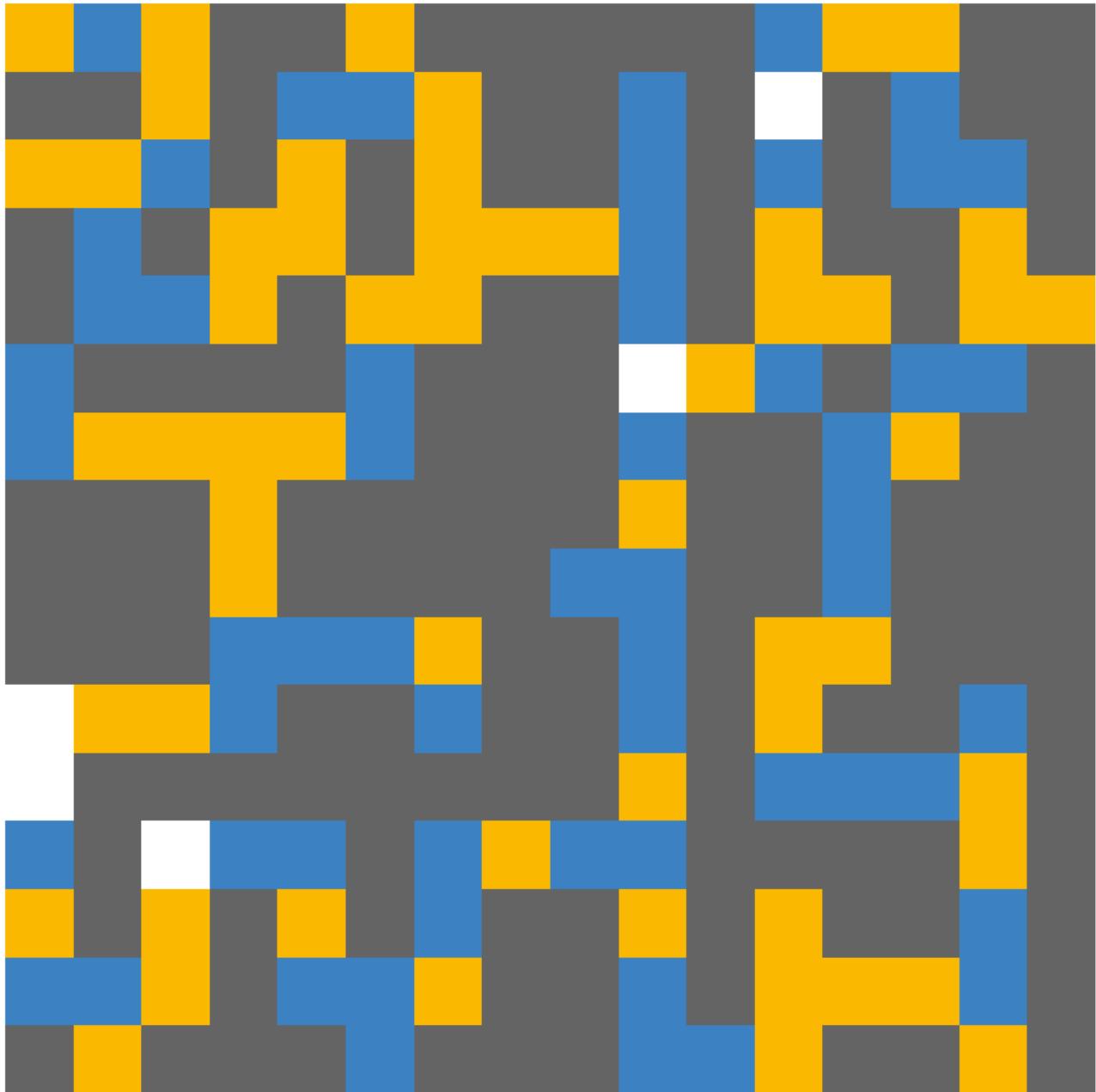


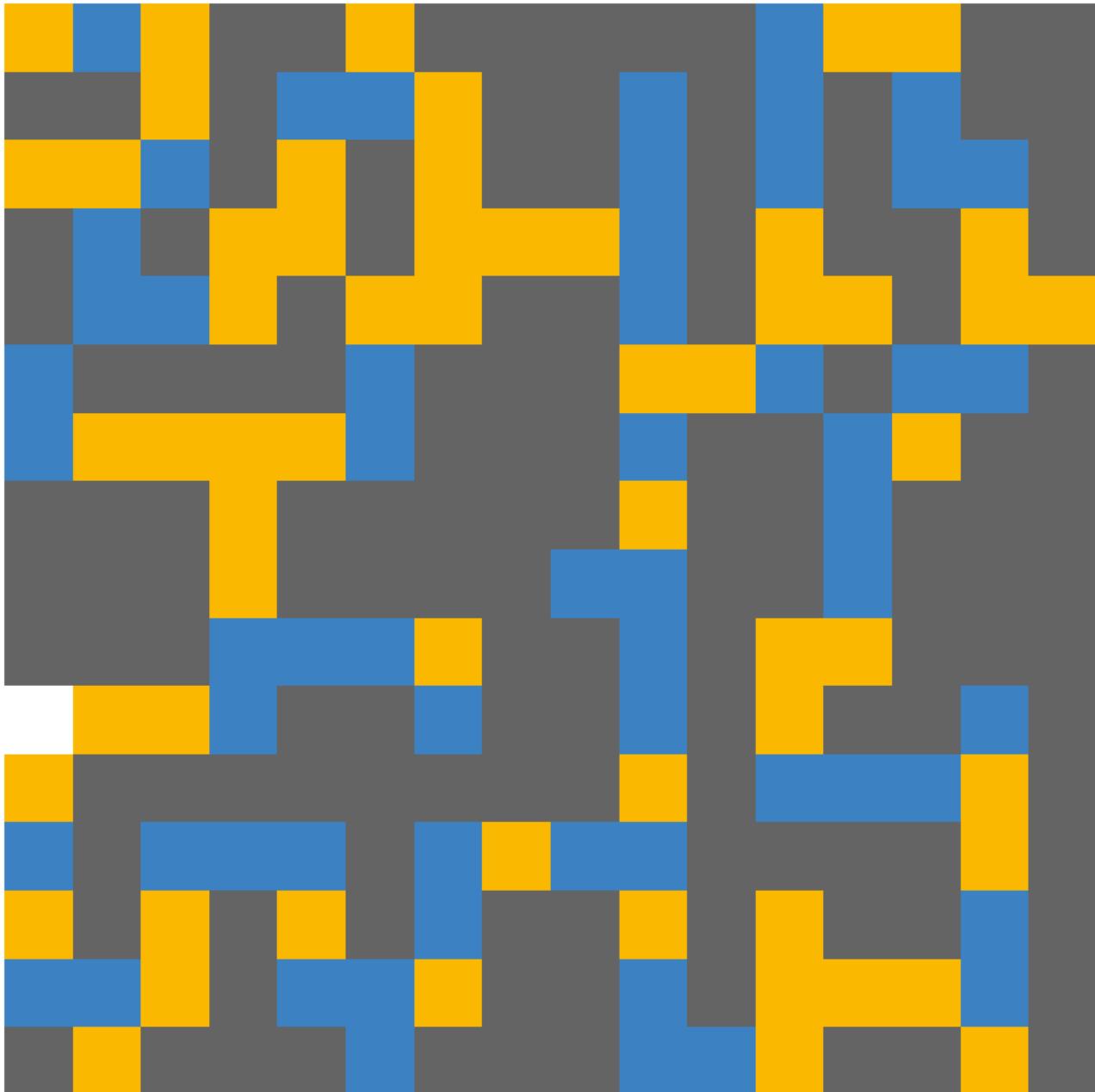












Trying to reset the state by saving an original copy of the state but it's not working, I'll try a different method

```
def place_agent_and_goal(self,state):
    original_state=state

    while True:
        # First place the agent
        state=self.place_agent(state)
        agent_position=self.find_positions(state,True)

        # Place goal
        state=self.place_goal(state)
        goal_position=self.find_positions(state,True,True)

        # self.visualize(state)
        print(state)

        # Check if Manhattan distance between them is far enough
        if self.M_distance(agent_position,goal_position)>=12:
            return state
        else:
```

```
# Else reset state
state=copy.deepcopy(original_state)
```

Nvm maybe there error was forgetting to remove the 3 from a variation of piece4

```
[[1, 0, 1, 1, 0, 1, 0, 1],
 [1, 0, 1, 1, 0, 1, 0, 1],
 [0, 0, 0, 0, 0, 0, 0, 0],
 [1, 0, 1, 1, 0, 1, 1, 1],
 [0, 0, 1, 1, 0, 1, 0, 1],
 [1, 0, 1, 0, 0, 0, 0, 1],
 [1, 0, 1, 1, 0, 1, 0, 0],
 [0, 0, 0, 0, 0, 1, 3]],
```

```
# Else reset state
for i in range(len(state)):
    for j in range(len(state[0])):
        if state[i][j]==2 or state[i][j]==3:
            state[i][j]=0
```

Nvm the problem is still there, but it's weird because there is always at most 4 colours when there is an error, so maybe it's the placement of where I reset the state

Okay that seemed to fix the problem

```
def place_agent_and_goal(self,state):
    # Save original state
    original_state=state

    while True:
        # Reset the state on each loop
        state=copy.deepcopy(original_state)

        # First place the agent
        state=self.place_agent(state)
        agent_position=self.find_positions(state,True)

        # Place goal
        state=self.place_goal(state)
        goal_position=self.find_positions(state,True,True)

        # self.visualize(state)

        # Check if Manhattan distance between them is far enough
        if self.M_distance(agent_position,goal_position)>=12:
            return state
```

I think I am going to add a difficulty option where it changes the distance between the Agent and the goal

After a bunch of tests, it seems like the farthest distance the agent and the goal can be from each other is over 28

The difficulty parameters:

- Easy:
- Medium:
- Hard:

Coded the parameters

```

# Set difficulty parameters
## If set to easy
if difficulty=="E":
    min_dist=5
    max_dist=9
## If set to medium
if difficulty=="M":
    min_dist=17
    max_dist=20

## If set to hard
if difficulty=="H":
    min_dist=18
    max_dist=30

```

Okay I think I am done coding the `Environment` class

Changing the functions to use `self`.

Finished adapted functions to `self.state` and pushed to main

Now to work on the `Agent` class, I'll copy and paste the functions that can be reused

- `scan()` (basically it's `percept()`)
- `update()`
- `transition_model()`
- `find_positions()`
- `visualized()`
- `goal_test()`
- `M_distance()`

Removing goal from Agent's DataFrame because I don't see a use for it

Might have a function for Agent called `update_data()`

To run the agent that uses the functions, will probably have something like this in `__init__`:

```

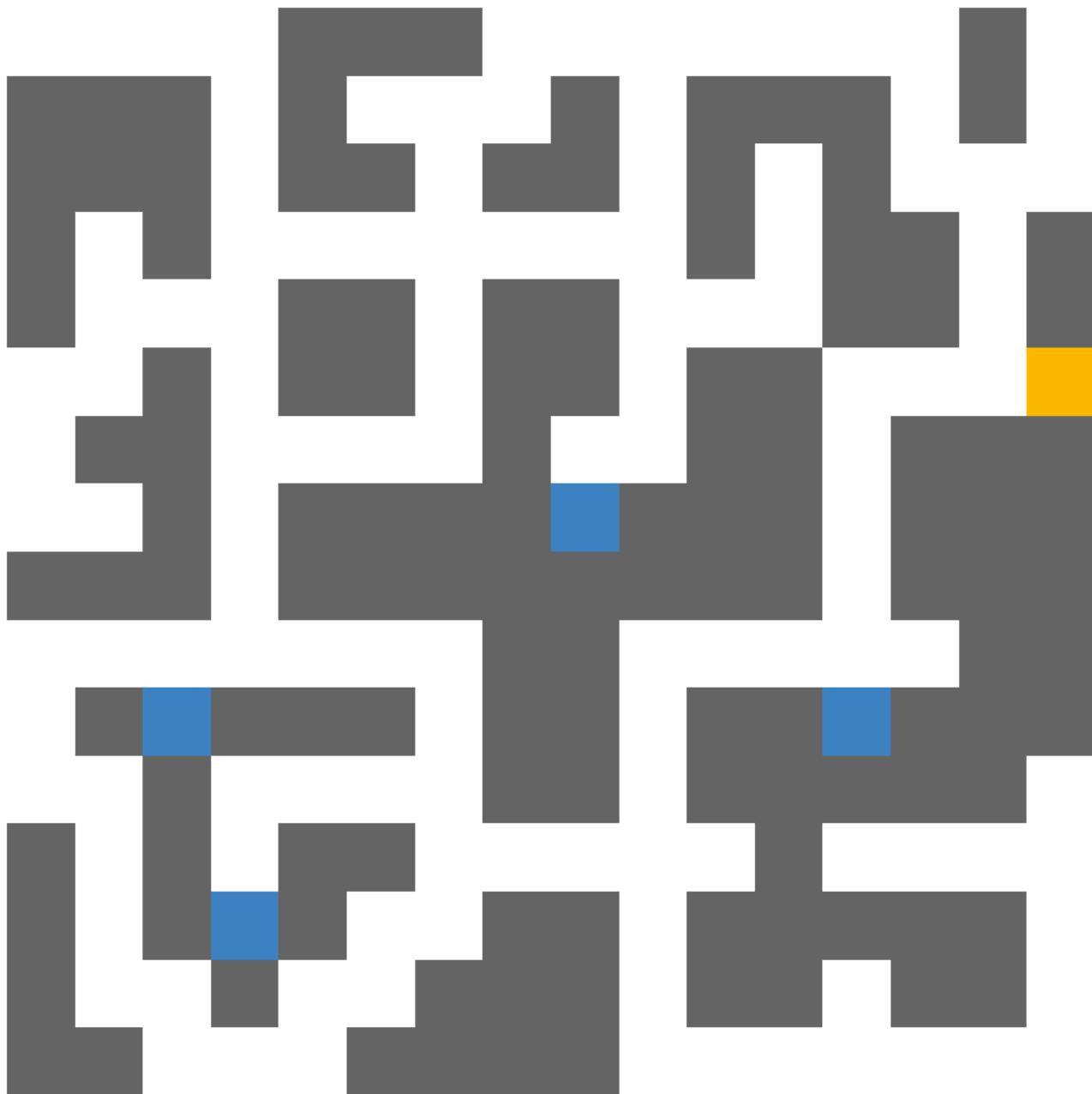
while True:
    self.run()

```

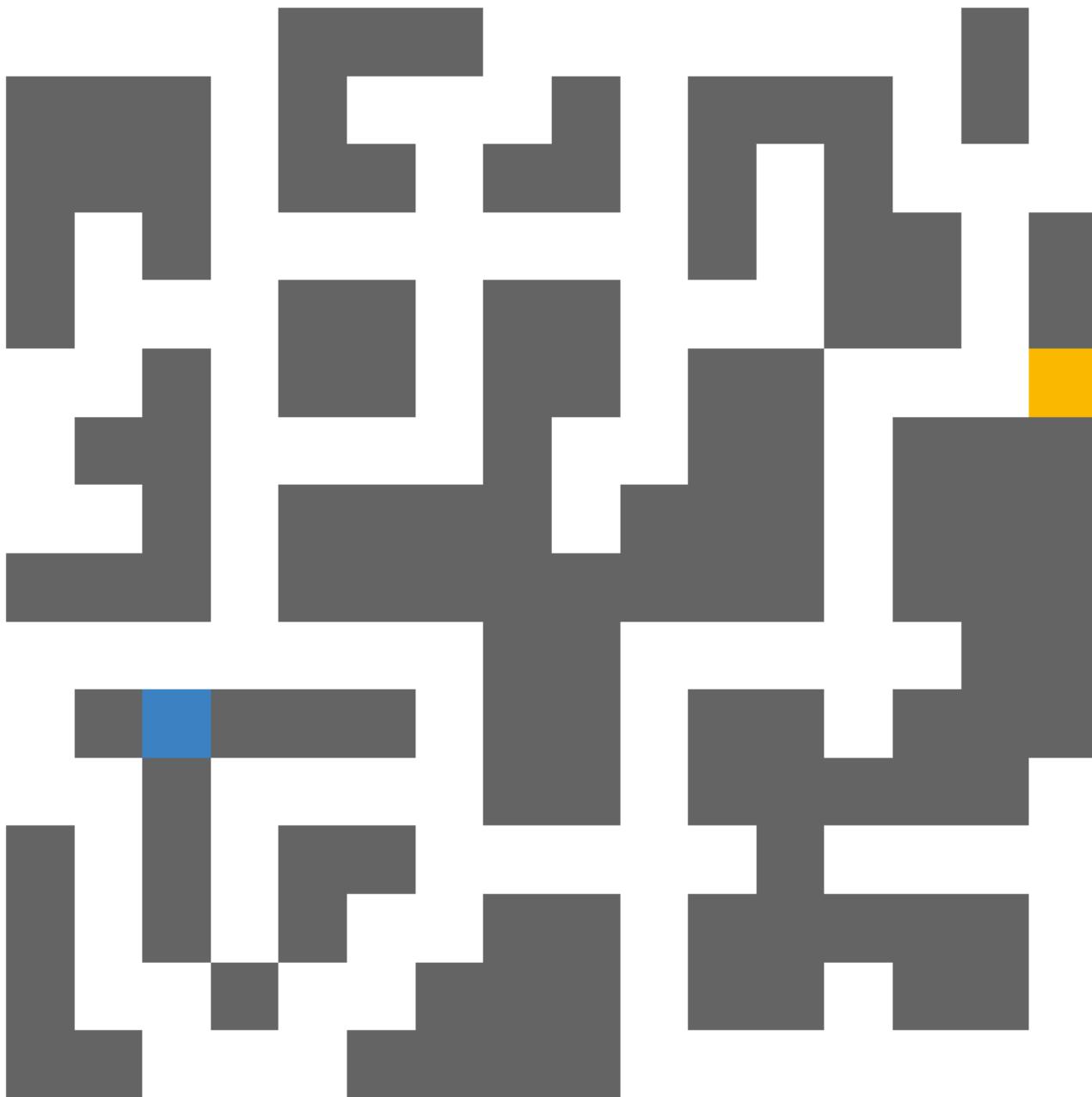
First we got to place the `Agent`'s mini-agents, so we got to place an agent on every square that's the correct percept. Will make a function called `place_agents()`, and for the list of functions I want to place in, I'll do it later as I go through the code

Saving initial state into a variable `self.initial_state` instead of copy and pasting it for each move in the DataFrame

This is the result



for



I'll also just remembered I need to import the Environment class, or do something so that the Agent class can get the percept from the Environment class

Was able to import the Environment class into Agent so it can use `percept()`

```
# In Environment class
self.Agent=Agent(self.agent_name,initial_state_clueless,percept,self)

# In Agent class
def __init__(self,name,initial_state,percept,Environment)
print(self.enviro.percept())
```

So I need to think how all the functions will run through each other:

1. Initiate variables
2. Place min-agents with `place_agents(percept)`
3. Save data with `update_data()`
- 4.

Created `update_data()`

```

def save_data(self):
    # Check if self.data exists
    if not "data" in dir(self):
        self.data=pd.DataFrame(data={"Belief state":[self.state],
                                     "Positions": [ self.find_positions()],
                                     "Percept":self.enviro.percept()},index=[0])
    return self.data

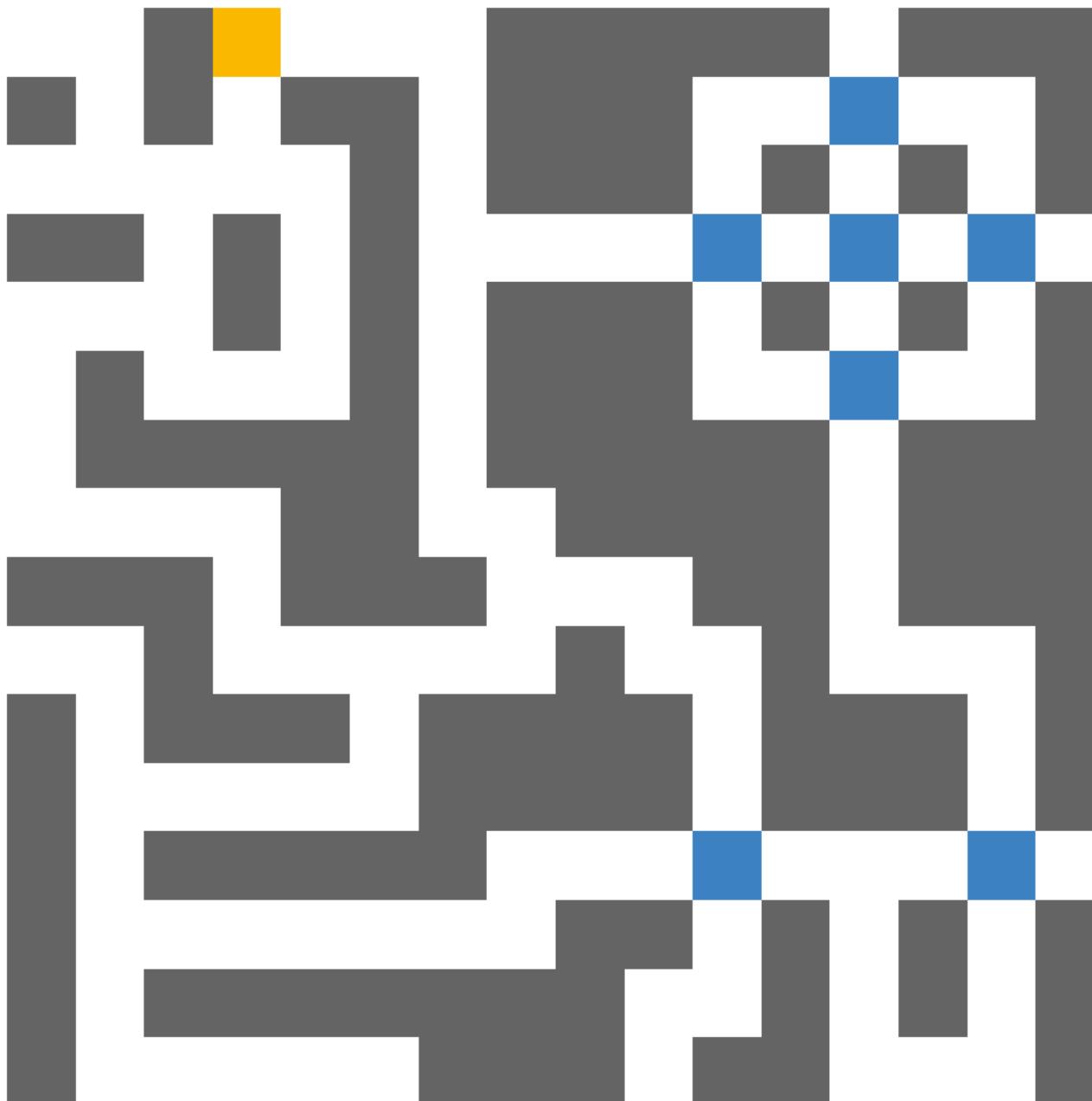
    # If it exists, then add on new data
new_data=pd.DataFrame(data={"Belief state":[self.state],
                           "Positions": [self.find_positions()],
                           "Percept":"self.enviro.percept()",index=[0]})

self.data=pd.concat([self.data,new_data])
return self.data

```

Okay I was thinking how `run()` would work but I already got a flowchart for it, despite I forgot I created `run()` before  
[Agent](#)

Cool Agent state



Coding update(percept)

Added "Move" to self.data

Okay finished coding update() , I'm pretty sure this is all update needs to do. It just needs to remove all mini-agents that are not corresponding with the new percept

```
def update(self,percept):
    # If the position is not the same as the percept, then remove mini-agent
    for position in self.find_positions():
        if position!=percept:
            self.state[position[0],position[1]]=0

    return None
```

Coding choose\_action()

Coding transition\_model()

Okay finished copy and pasting transition\_model() from Environment but instead of finding True or False for all of mini-agents, it's designed to find just one. This is because choose\_move() will already find all the positions. It's ironic because this would have been also useful to find just one position in Environment but I didn't cause I wanted to code it to work for Agent

Left off on coding choose\_action()

Finished choose\_action()

Created self.move() that uses self.action to know which direction to move

Going to have a loop like this that will keep running run until it reaches the goal

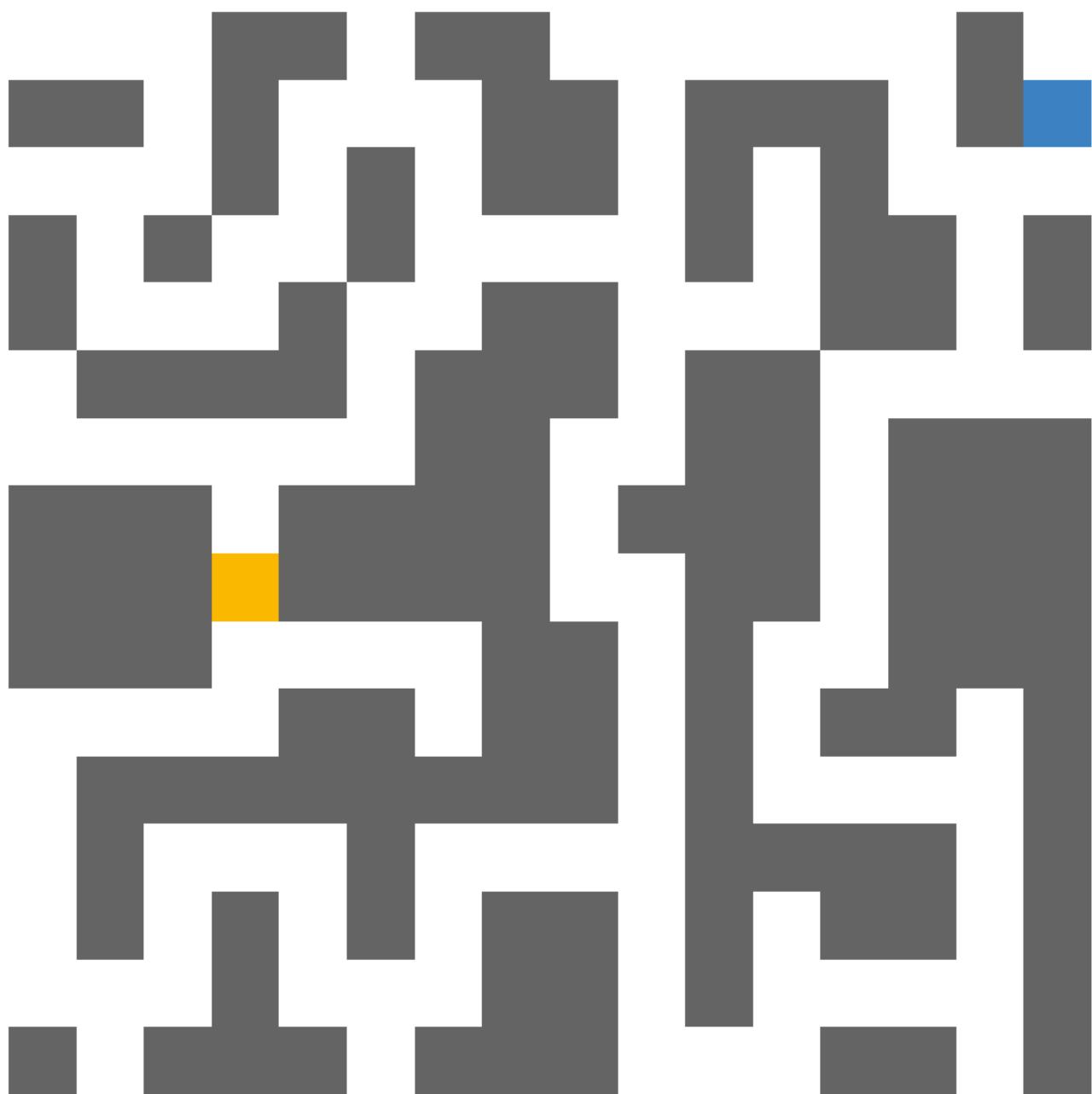
```
while not self.enviro.goal_test():
    self.run()
```

Forgot in self.move() that I need to move each mini-agent. Just had to basically copy and paste update() from Environment

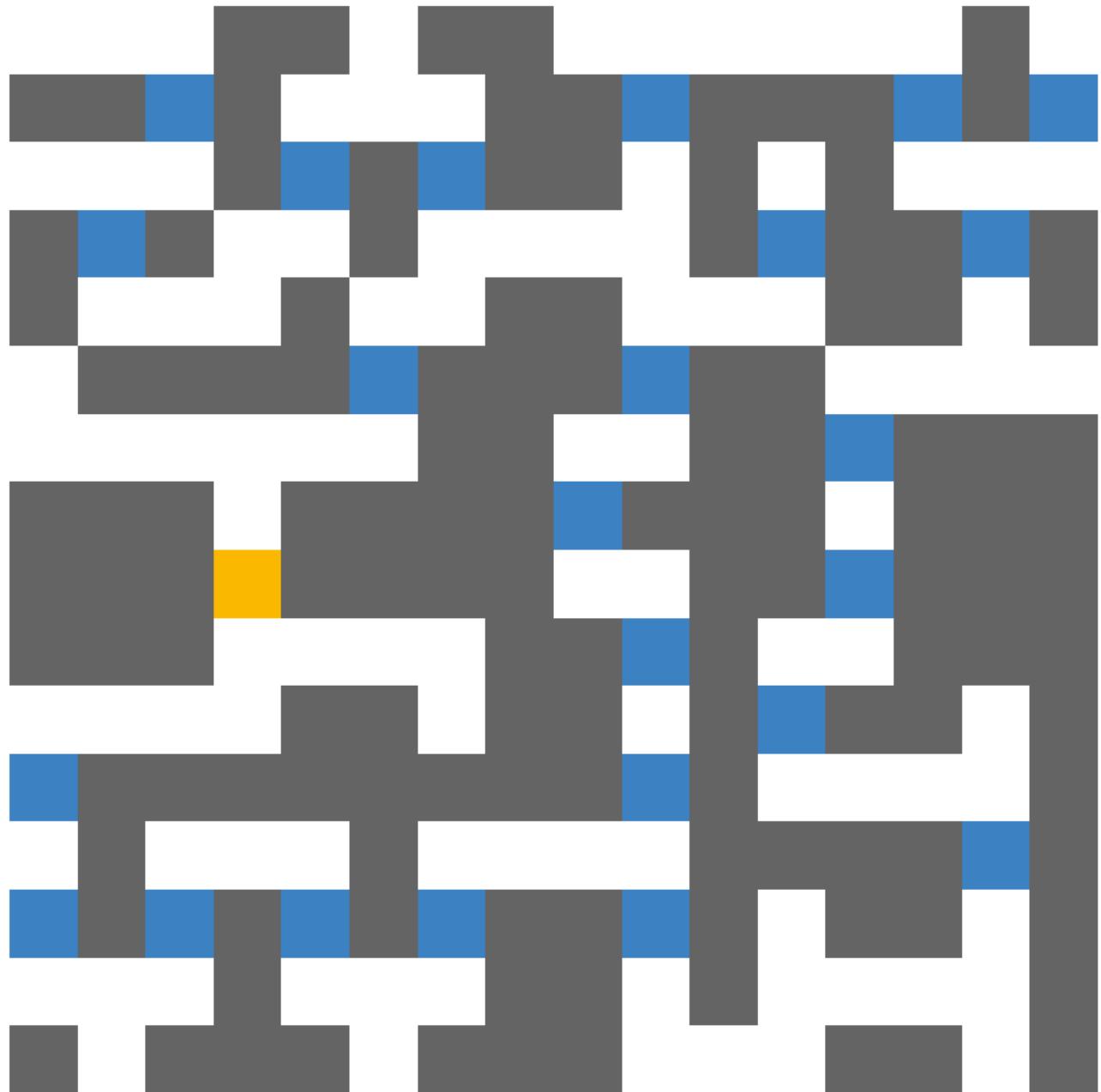
Okay so I ran run() a couple of times and it eliminated the real Agent

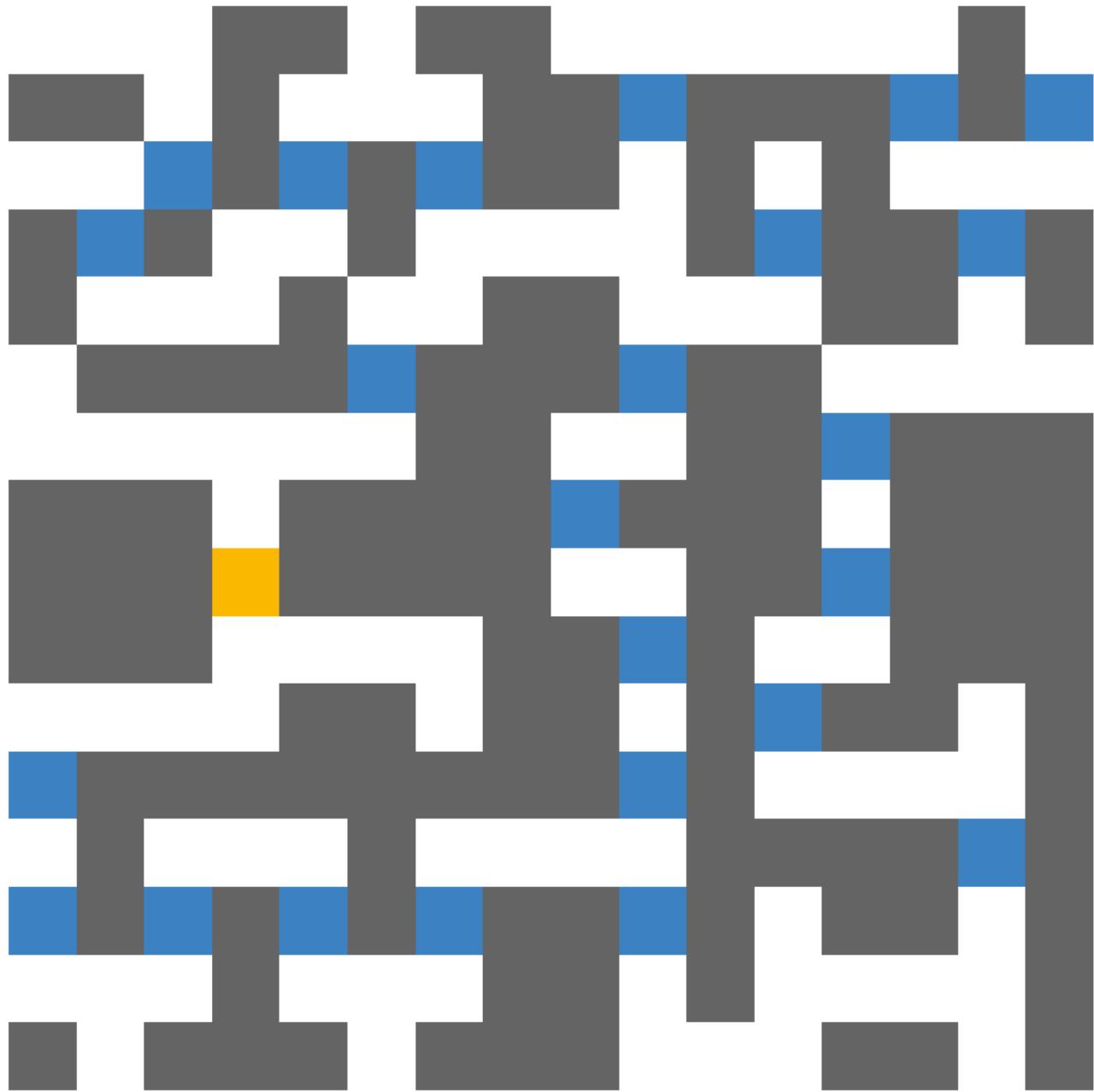
 [Eliminating the real Agent >](#)

The real Agent



The mini-agents





Okay well while pasting the images in I could see that a mini-agent went down as the rest didn't which is an error

Part of the error was fixed by taking

```
# Officially update the Environment state  
self.enviro.update(self.action)  
  
return None
```

out of the

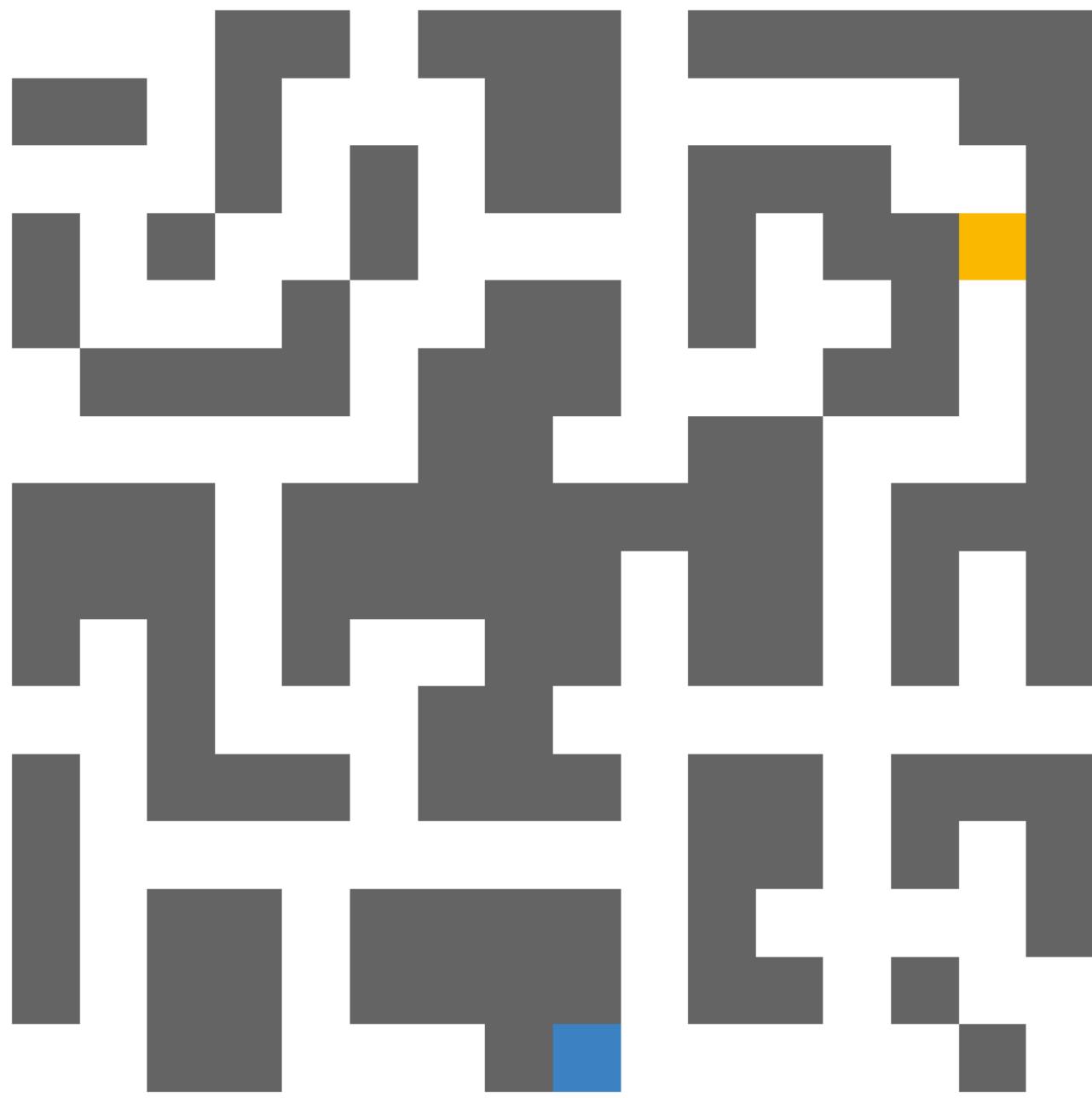
```
for position in self.find_positions()
```

loop in move()

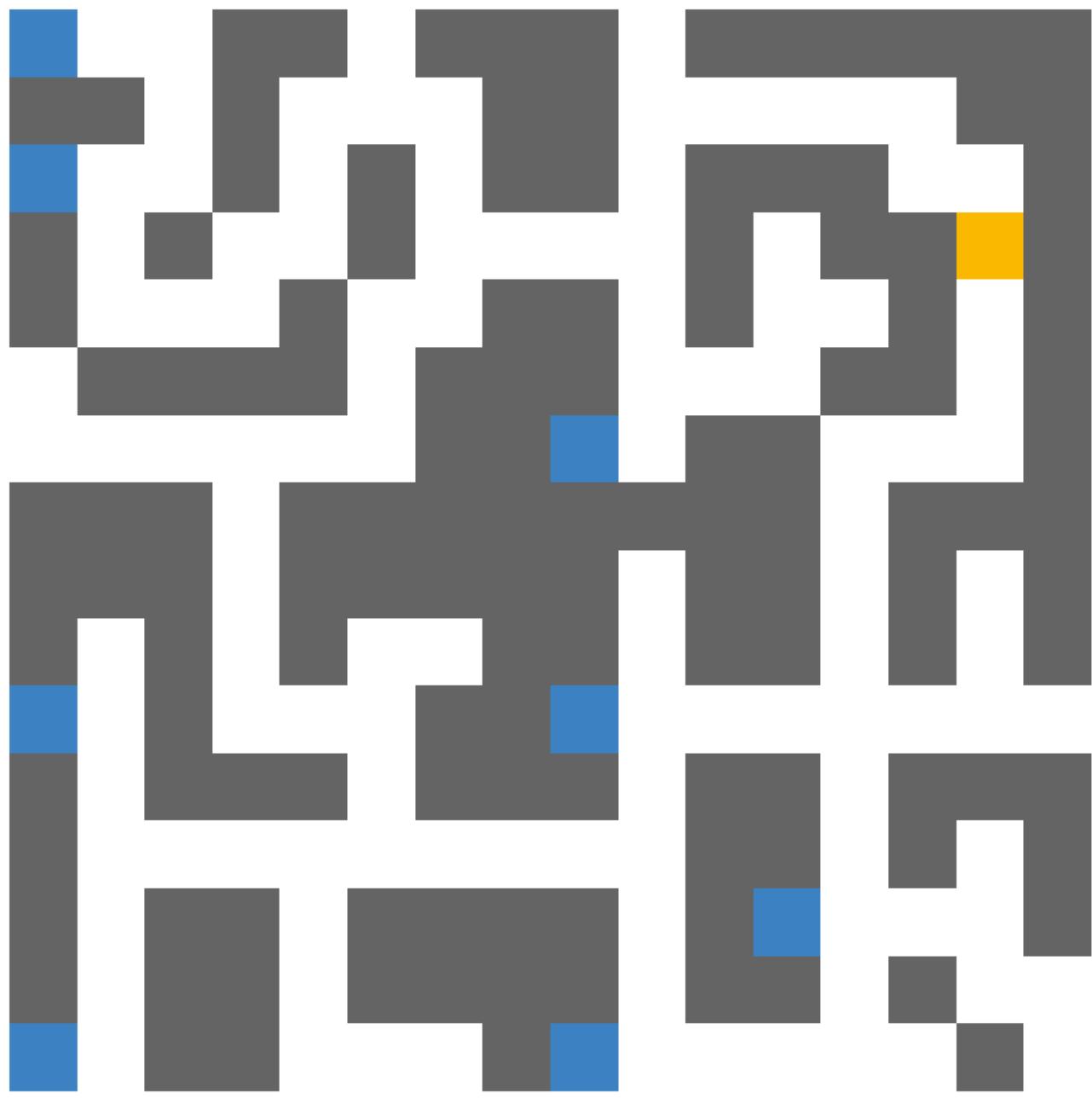
Now we have

Photo >

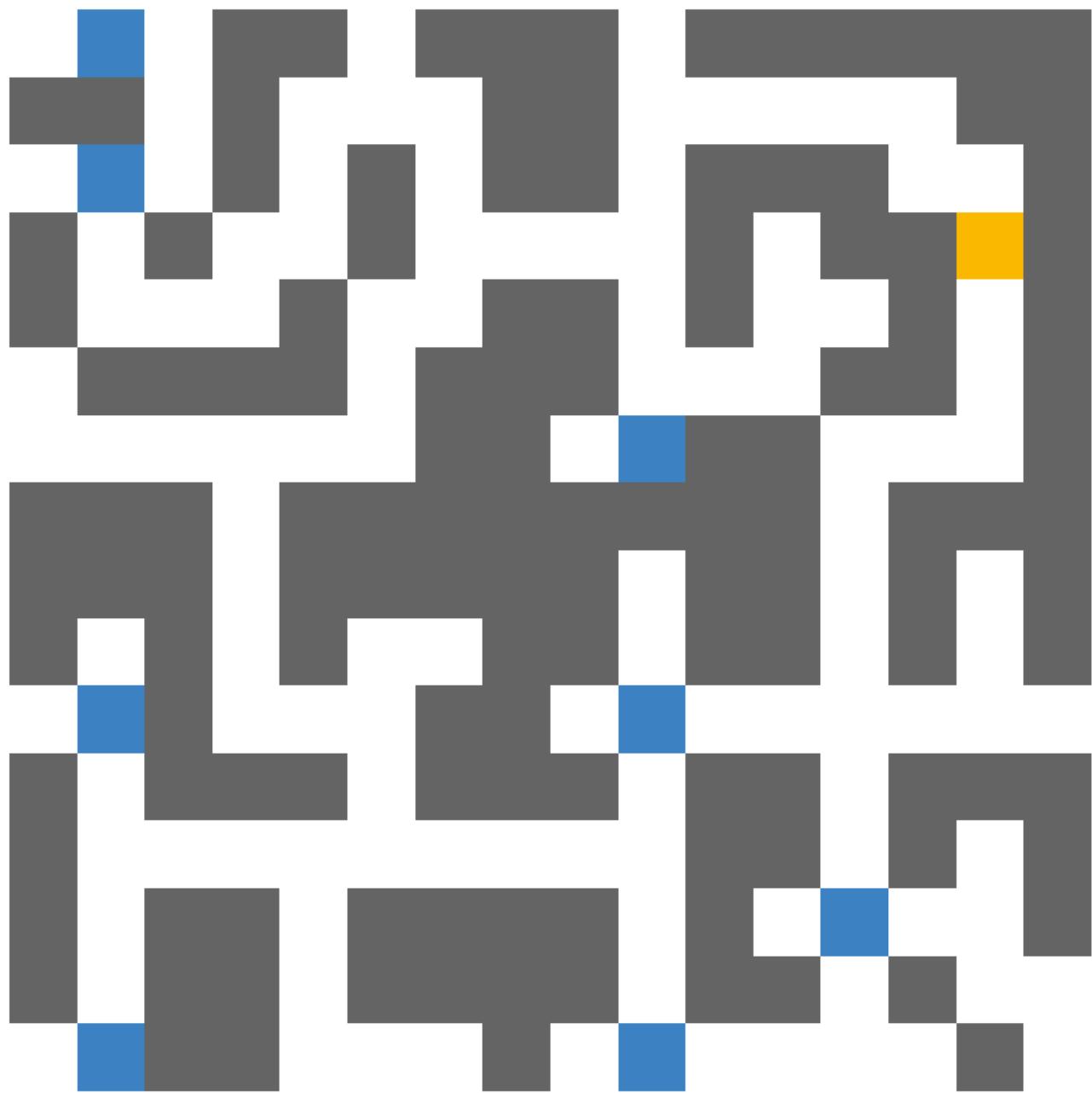
Real Agent



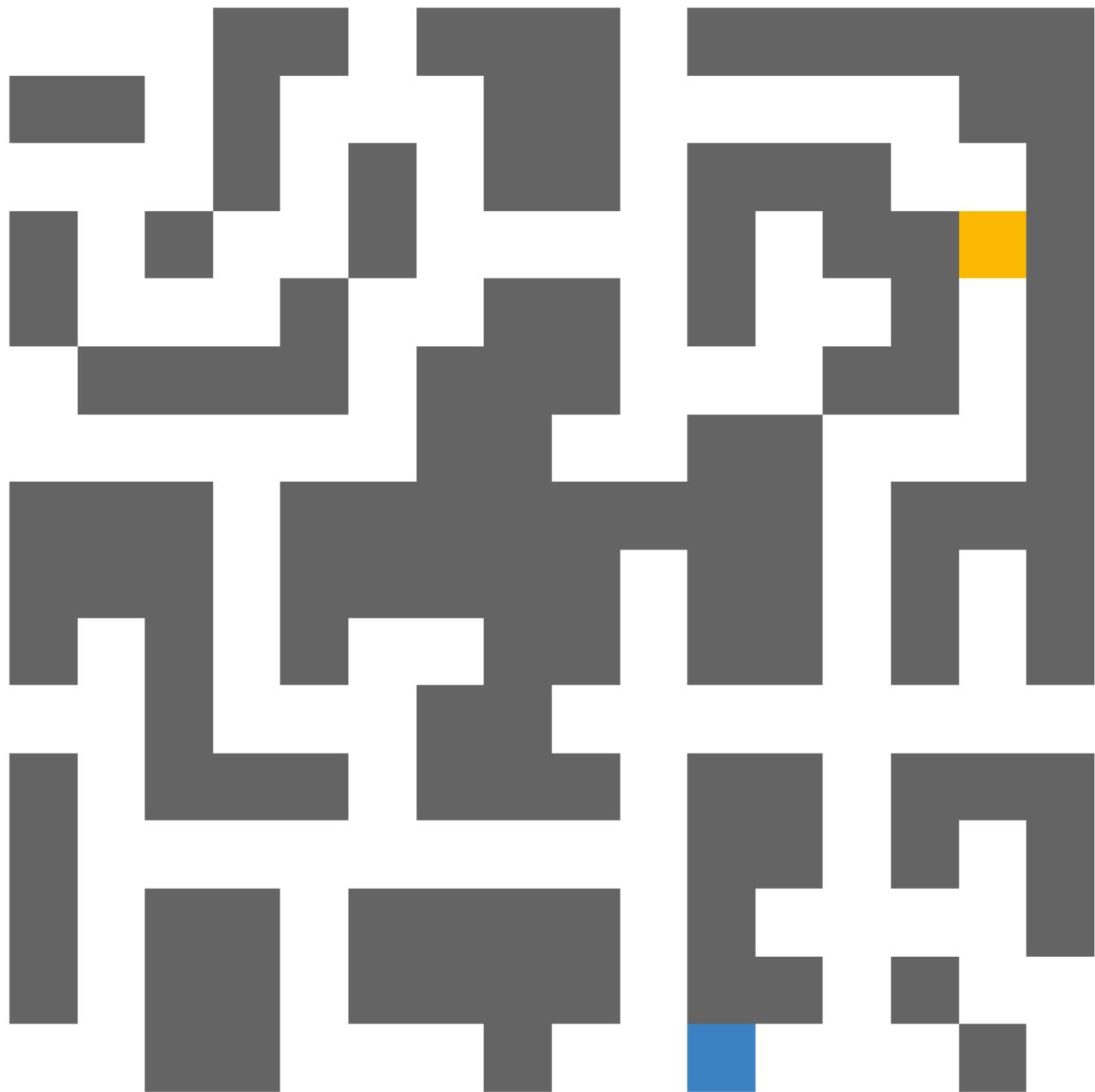
Mini-agents  
fine



everything moves East



everything moves East again but all other mini-agents get removed

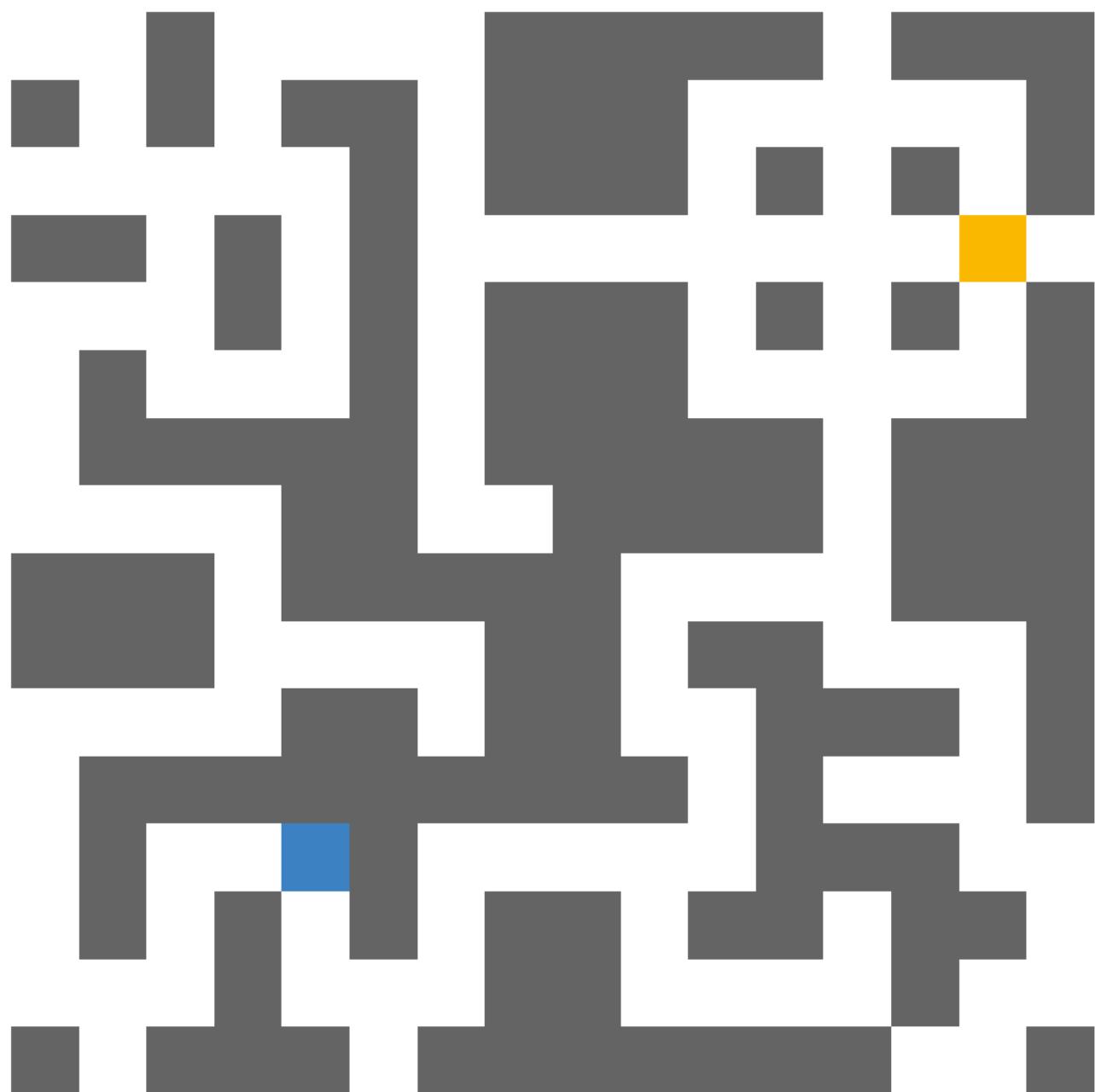


The problem must be related to `update()` since it removes mini-agents that are not in the right spot afterwards

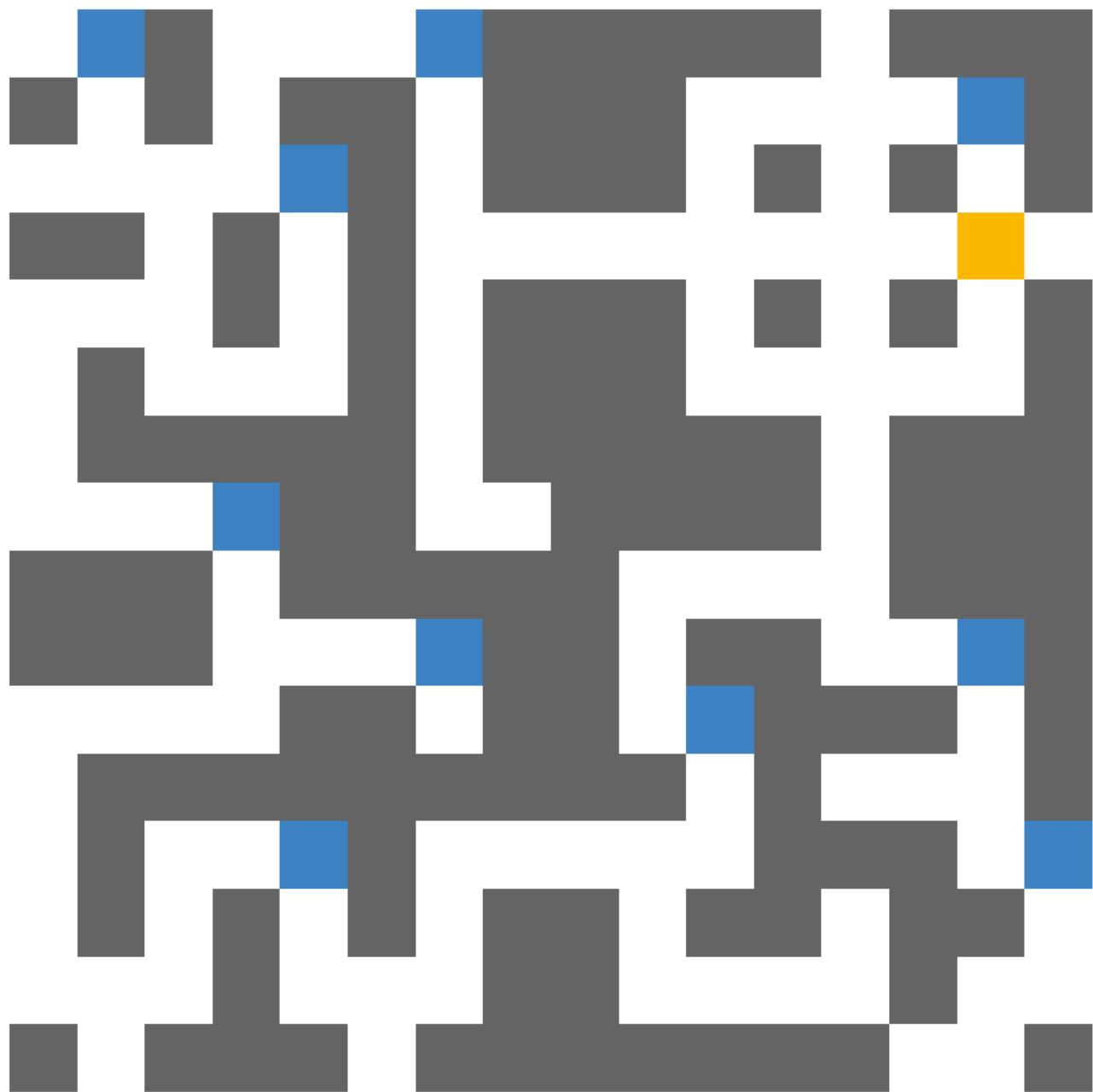
This is one makes sense with the `self.data`

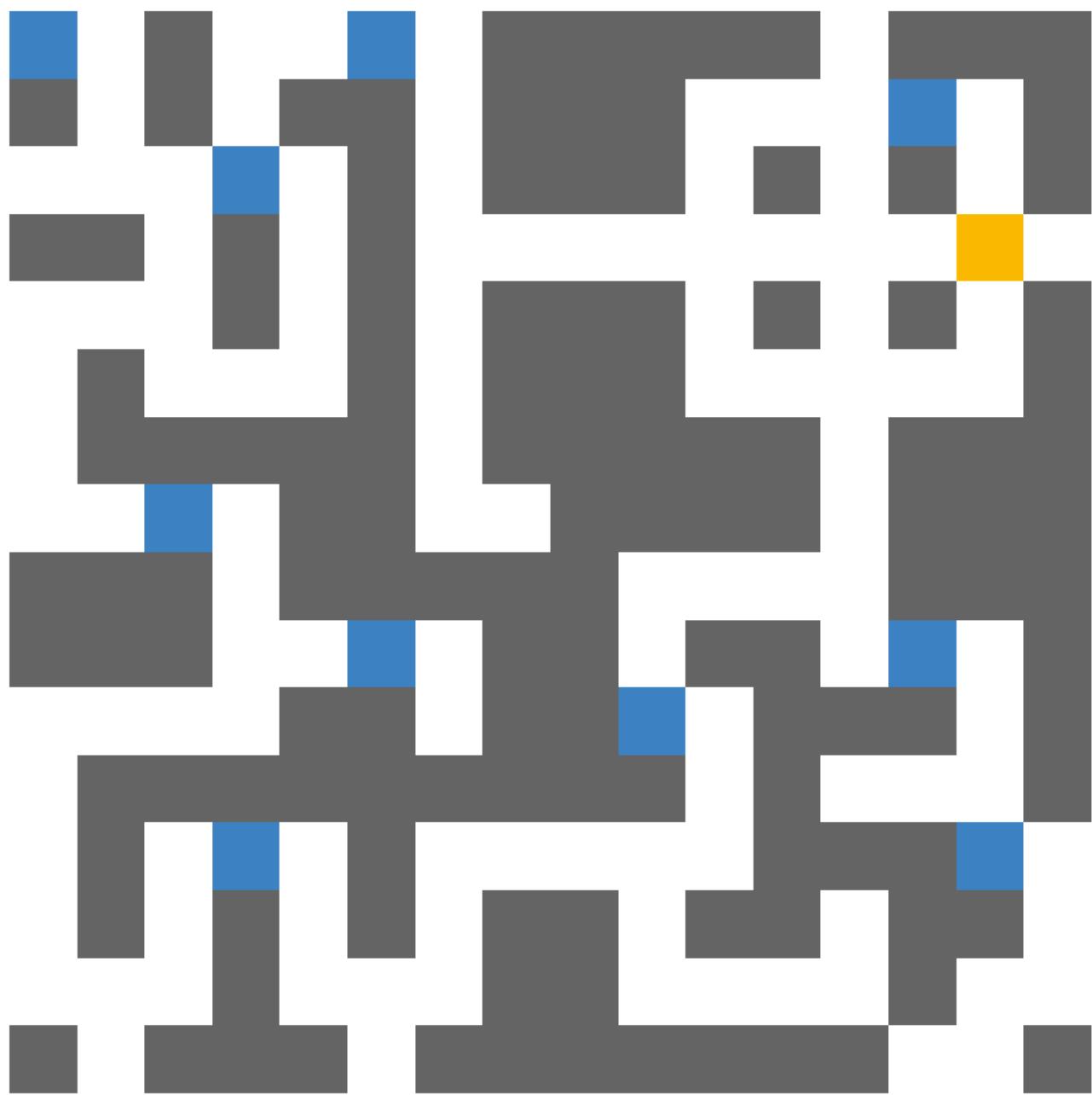
 Photo >

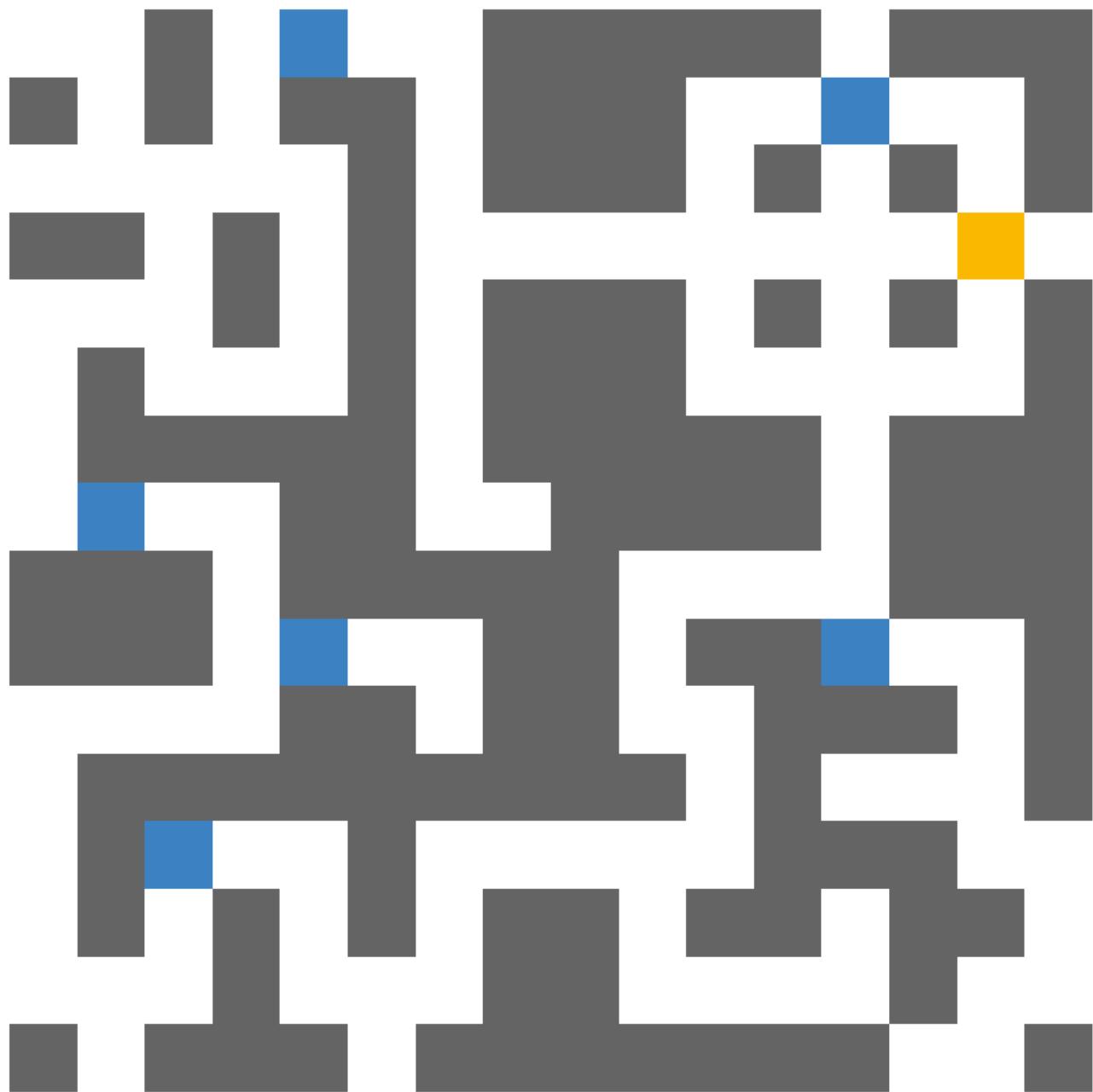
Agent

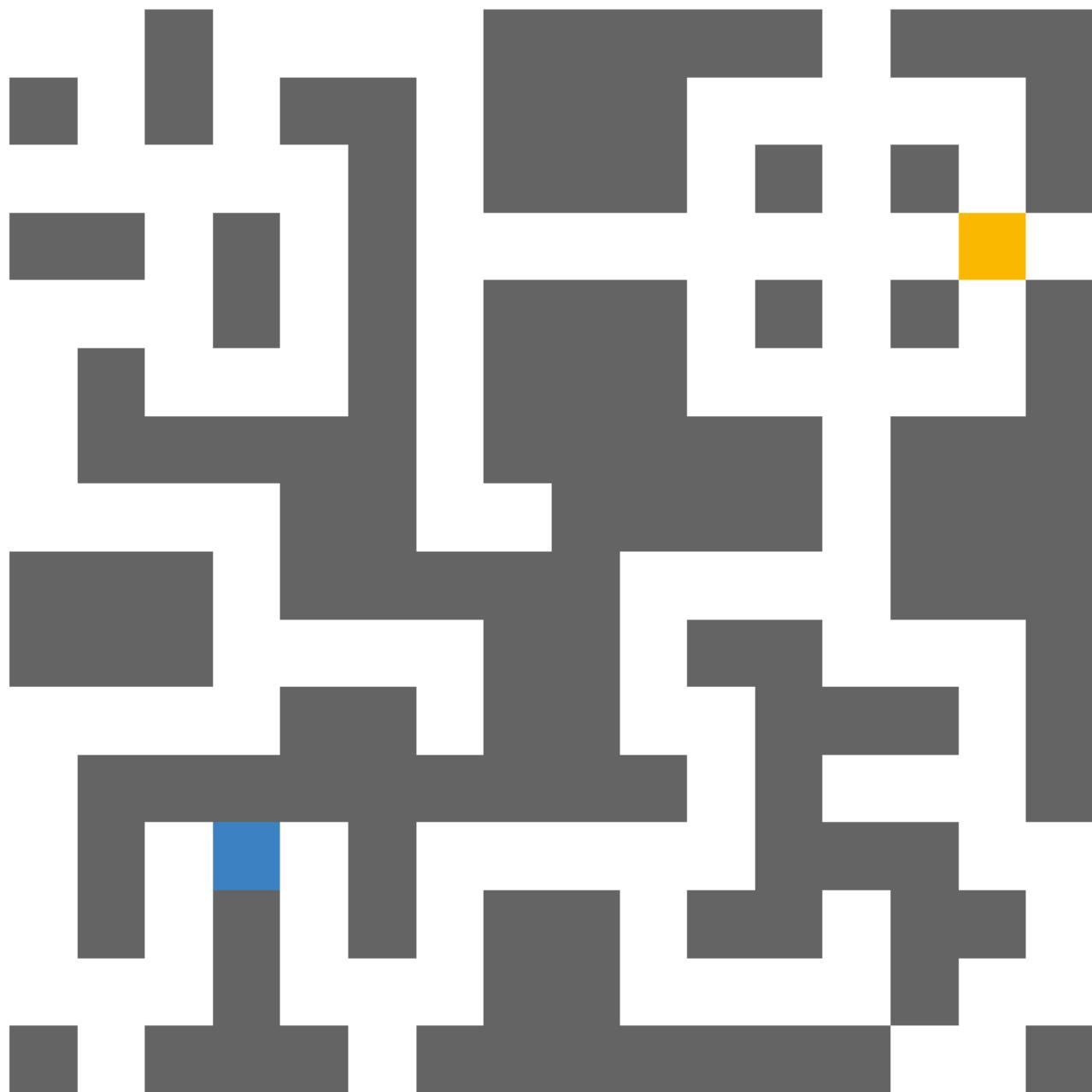


Mini-agents



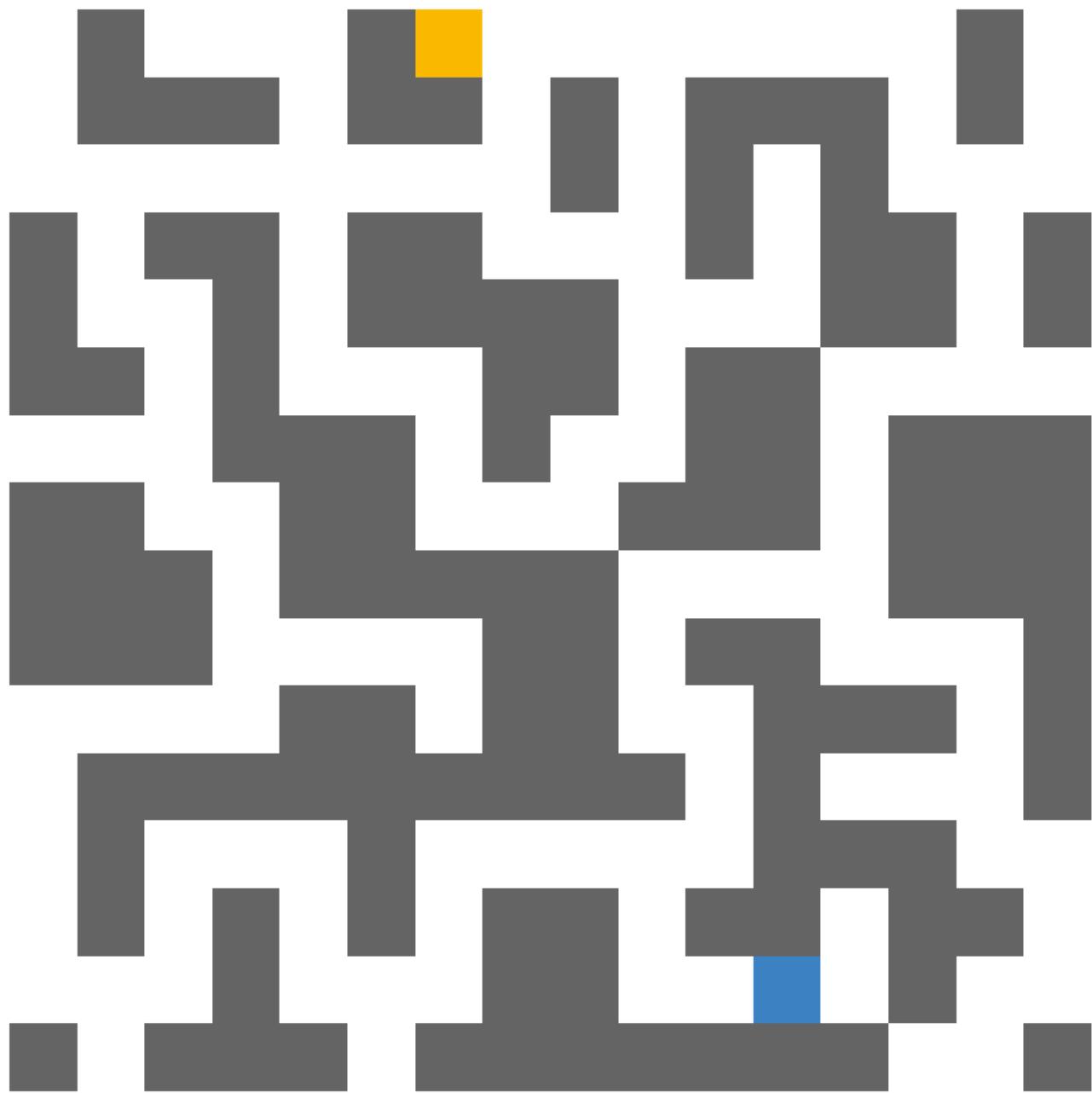


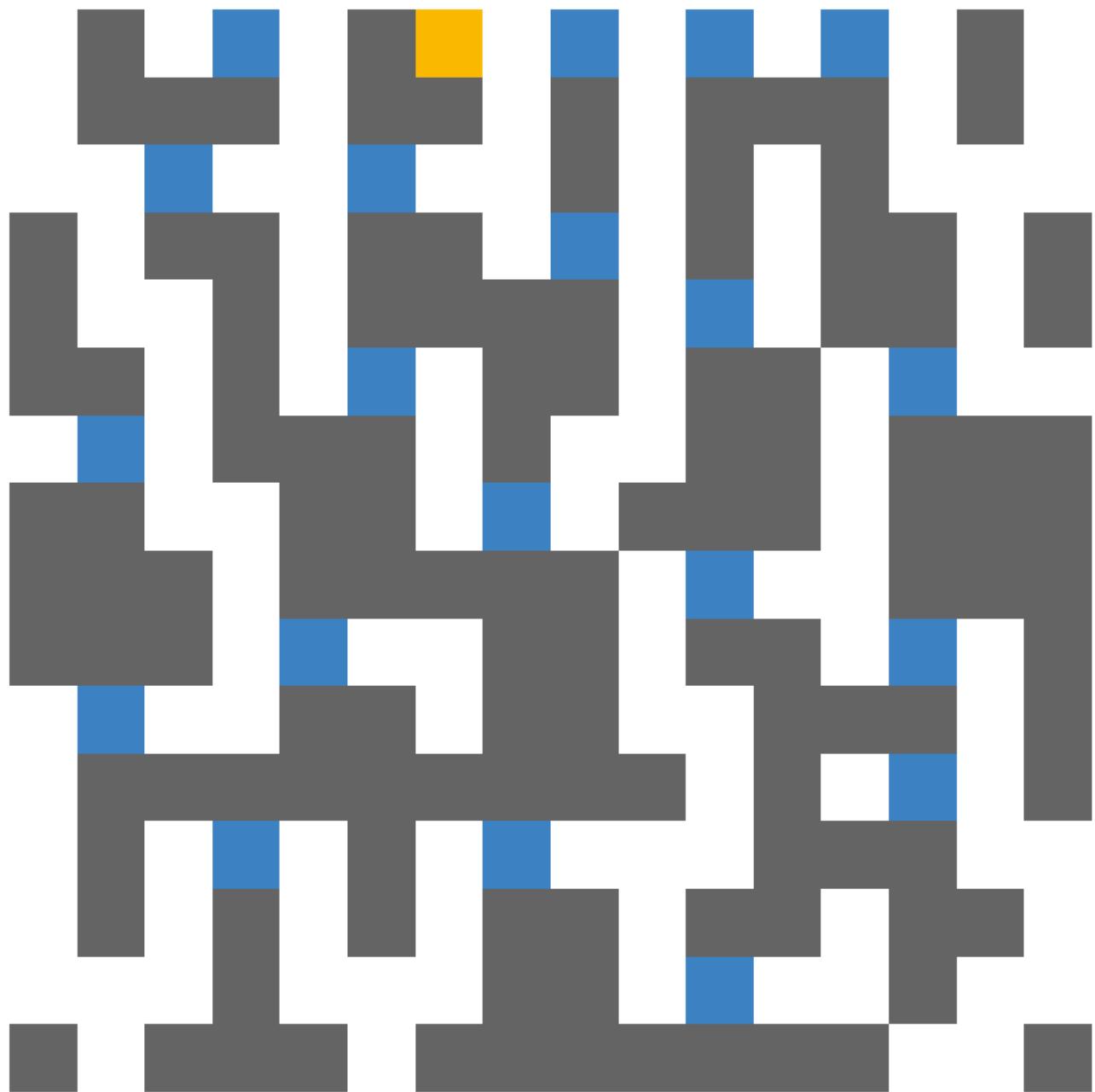


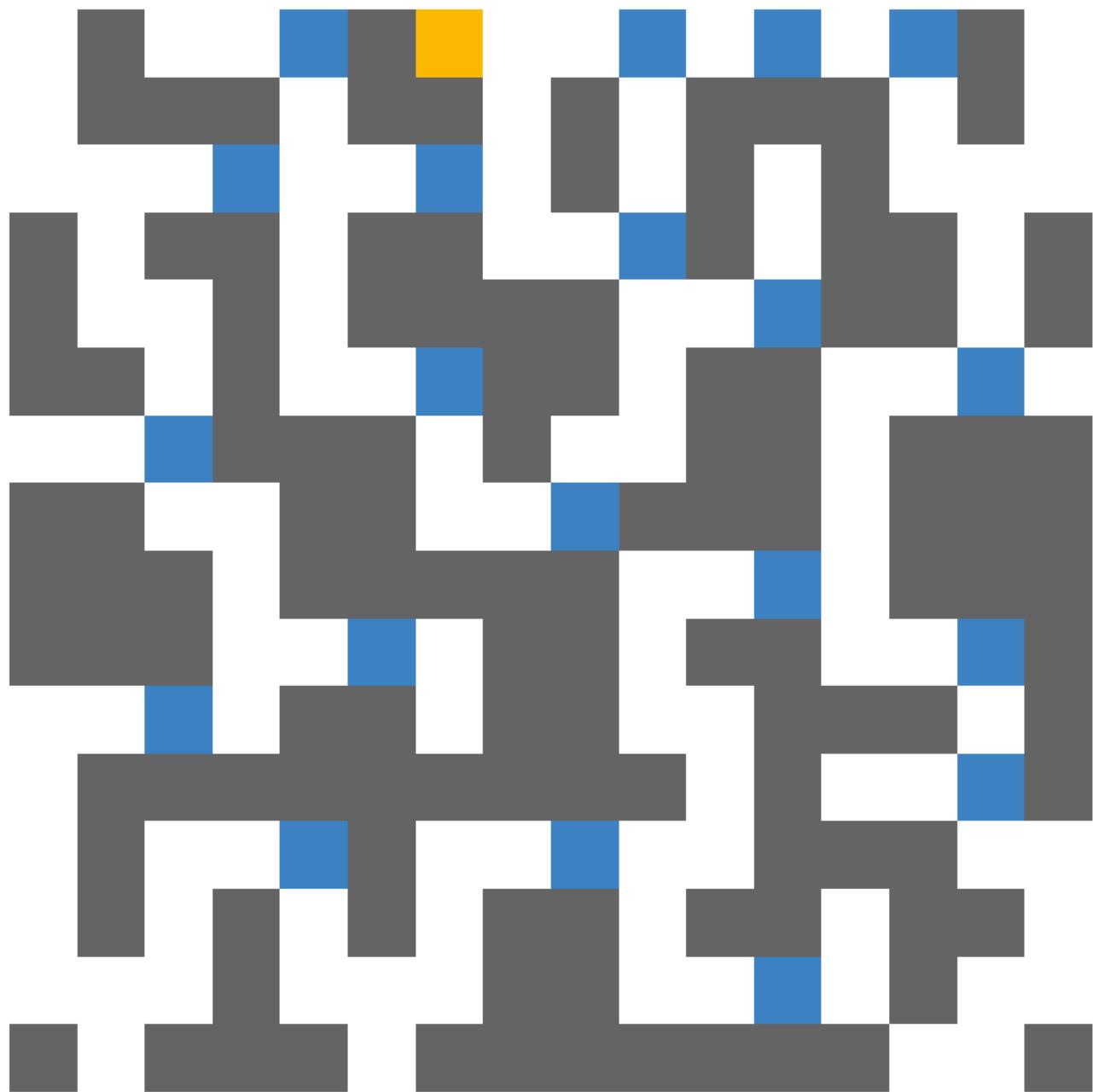


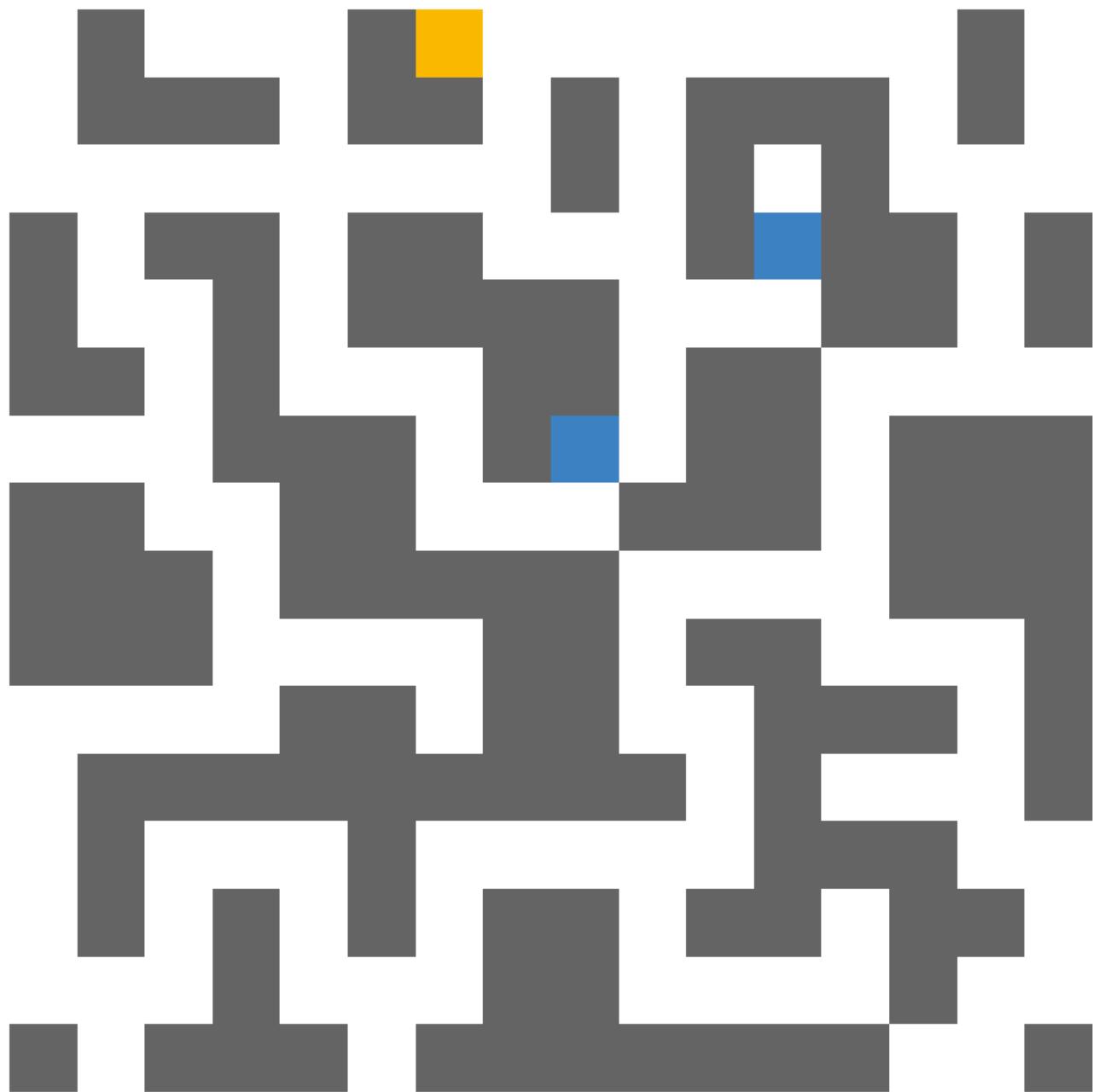
Okay it seems like most of the cases work, but there are some cases that obviously don't work because it gets rid of the main Agent, like this one

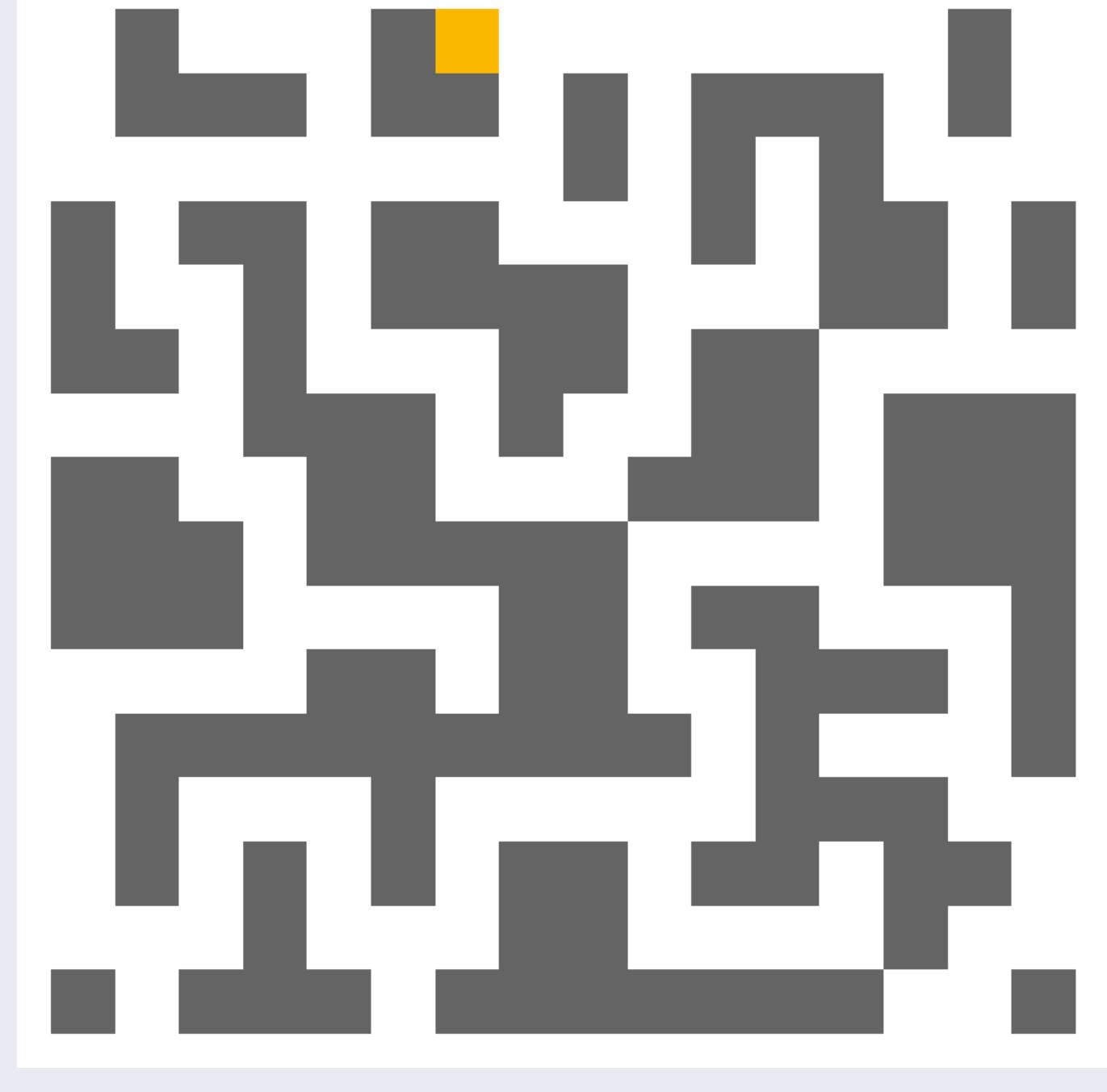
[Photo >](#)











It seems like it happened because the choose\_direction() choose North when that's not possible

Index	Belief state	Positions	Percept	Move
0	[[0, 1, 0, 0, 0, 1, 3, 0, 0, 0, 0, 0, 0, 0, ...	[[0, 3], [0, 8], [0, 10], [0, 12], [2, 2], ...	1001	None
1	[[0, 1, 0, 0, 0, 1, 3, 0, 0, 0, 0, 0, 0, 0, ...	[[0, 3], [0, 8], [0, 10], [0, 12], [2, 2], ...	1001	E
2	[[0, 1, 0, 0, 0, 1, 3, 0, 0, 0, 0, 0, 0, 0, ...	[[4, 11], [7, 8]]	0101	N
3	[[0, 1, 0, 0, 0, 1, 3, 0, 0, 0, 0, 0, 0, 0, ...	[]	1110	N
4	[[0, 1, 0, 0, 0, 1, 3, 0, 0, 0, 0, 0, 0, 0, ...	[]	1110	N
5	[[0, 1, 0, 0, 0, 1, 3, 0, 0, 0, 0, 0, 0, 0, ...	[]	1110	N
6	[[0, 1, 0, 0, 0, 1, 3, 0, 0, 0, 0, 0, 0, 0, ...	[]	1110	N
7	[[0, 1, 0, 0, 0, 1, 3, 0, 0, 0, 0, 0, 0, 0, ...	[]	1110	N
8	[[0, 1, 0, 0, 0, 1, 3, 0, 0, 0, 0, 0, 0, 0, ...	[]	1110	N
9	[[0, 1, 0, 0, 0, 1, 3, 0, 0, 0, 0, 0, 0, 0, ...	[]	1110	N
10	[[0, 1, 0, 0, 0, 1, 3, 0, 0, 0, 0, 0, 0, 0, ...	[]	1110	N

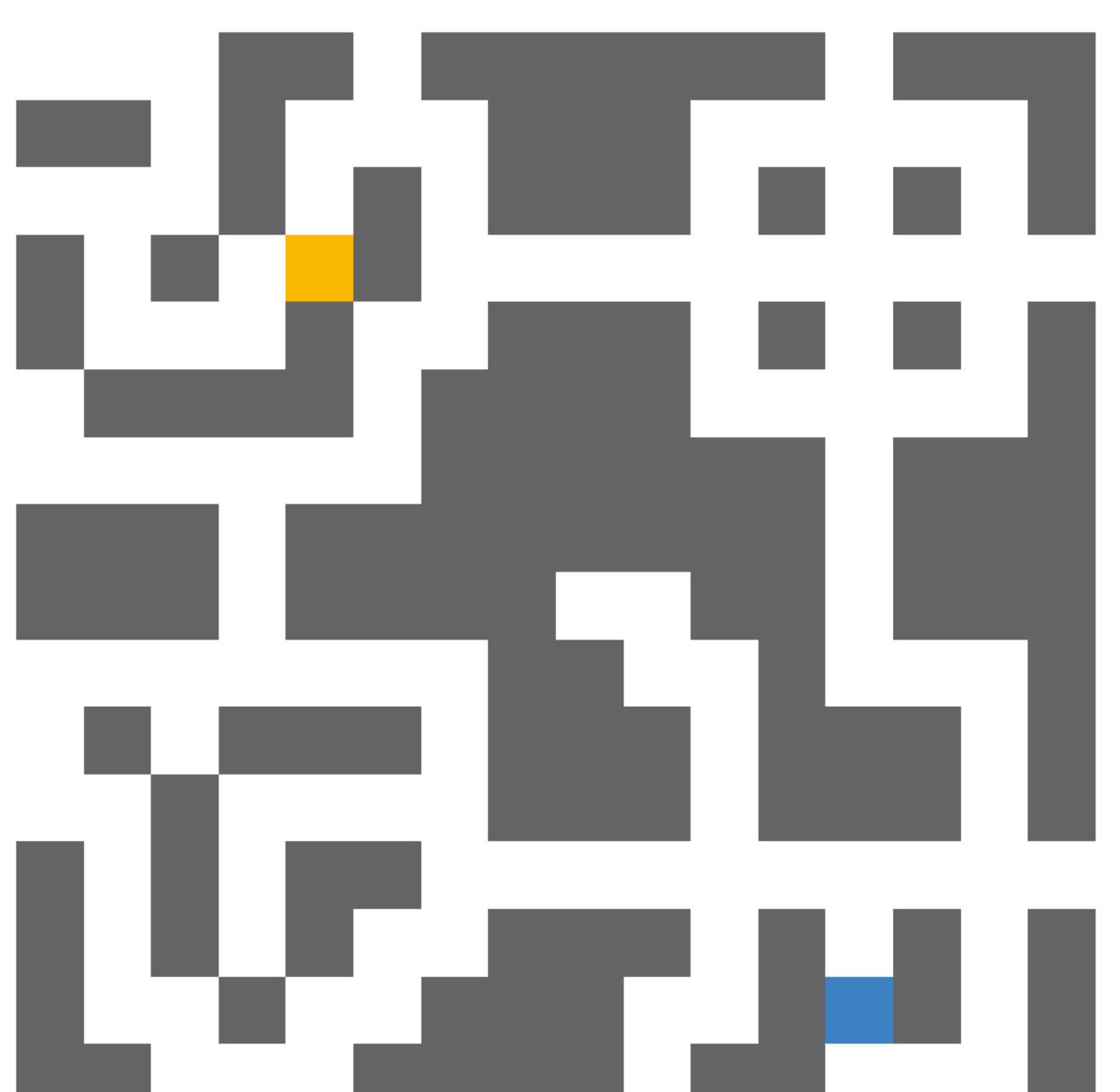
So it must be how I choose the random direction does not work, not really the code itself

Okay I found why it doesn't work, it's because of `place_agents()`. When a mini-agent is being placed in spots with valid percepts, there a chance another

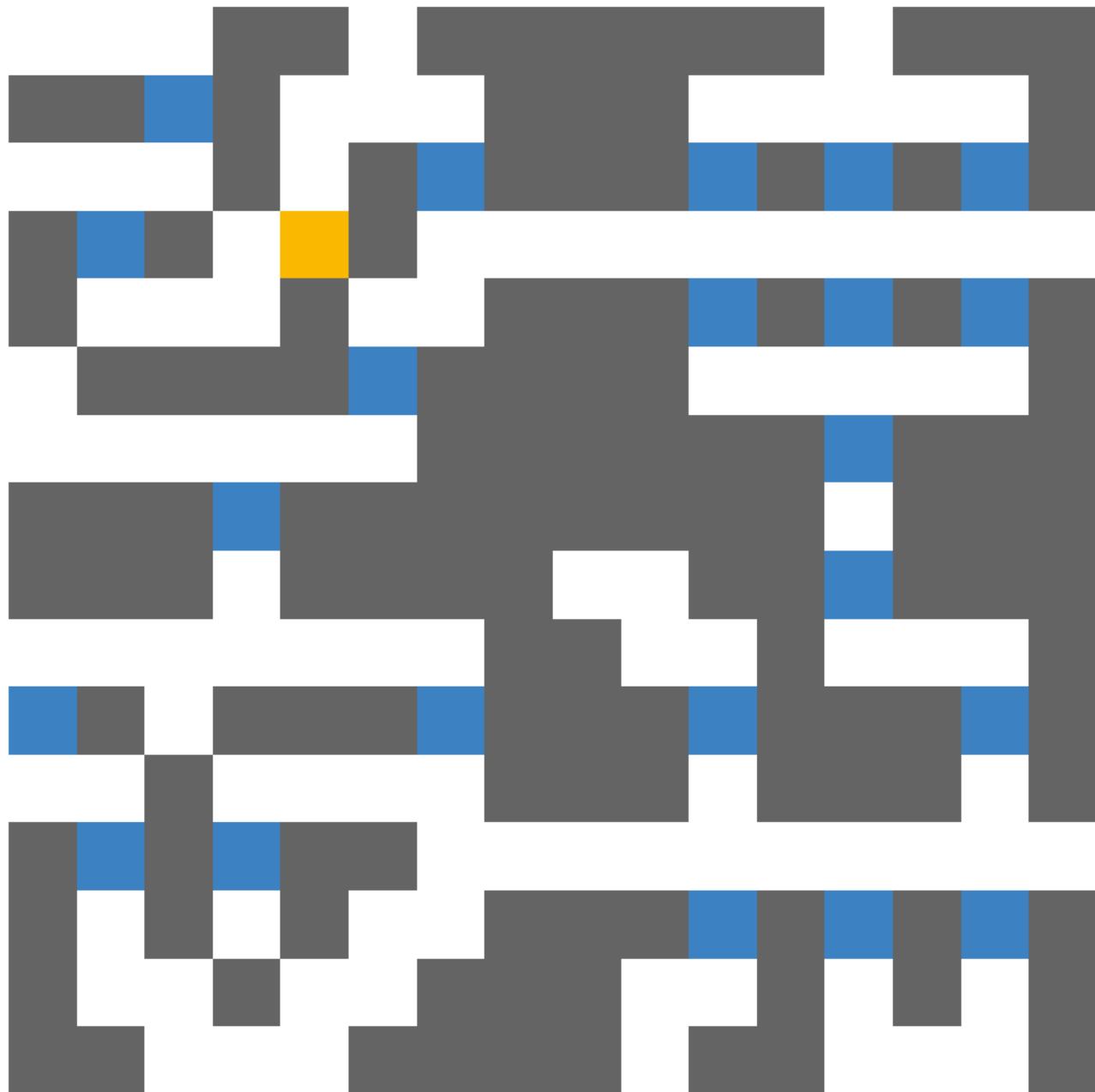
mini-agent could be next to a valid spot. As in, if there are two valid spots next to each other, only one will be picked because later the 0 becomes 2 means the percept is technically not the same as seen here

#### Example >

Real Agent



Mini-agents (you can see that the Real Agent moved)



So I got to fix that by having 0 and 2 both be interchangeable

I'm trying to set up some sort of combination thing to give all possible percepts if there is a 2 instead of a 0

```
for i in range(4):
    for j in range(3):
        for k in range(2):
            for l in range(1):
                print(i,j,k,l)
```

gives

```
0 0 0 0
0 0 1 0
0 1 0 0
0 1 1 0
0 2 0 0
0 2 1 0
1 0 0 0
1 0 1 0
```

```
1 1 0 0  
1 1 1 0  
1 2 0 0  
1 2 1 0  
2 0 0 0  
2 0 1 0  
2 1 0 0  
2 1 1 0  
2 2 0 0  
2 2 1 0  
3 0 0 0  
3 0 1 0  
3 1 0 0  
3 1 1 0  
3 2 0 0  
3 2 1 0
```

Okay I thought of a way easier solution

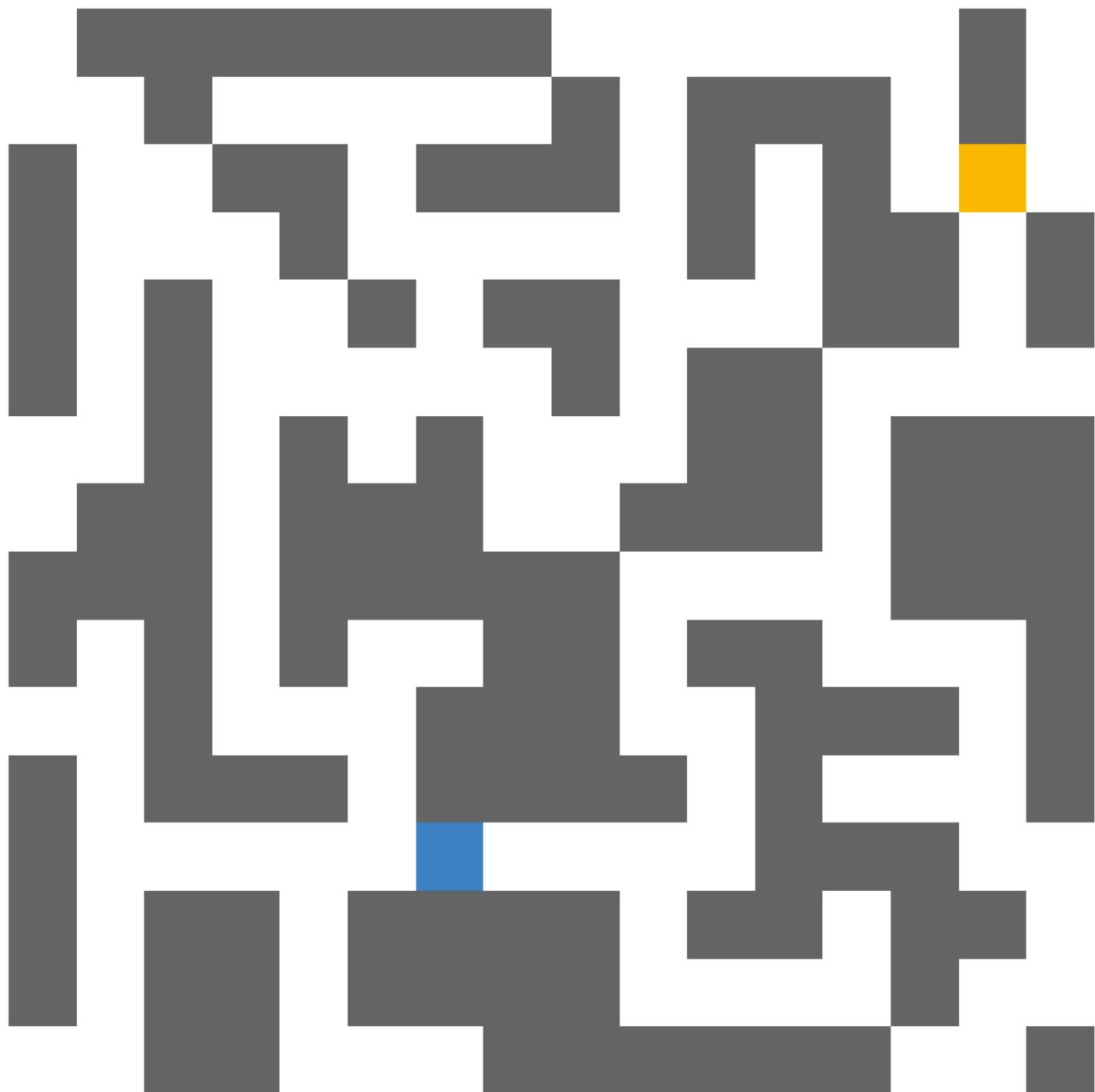
```
# Replace  
if percept==self.scan([i,j]) and self.state[i][j]==0:  
  
# With  
if percept==self.scan([i,j]).replace("2","0") and self.state[i][j]==0:
```

Now percept from self.scan() will ignore mini-agents (2)

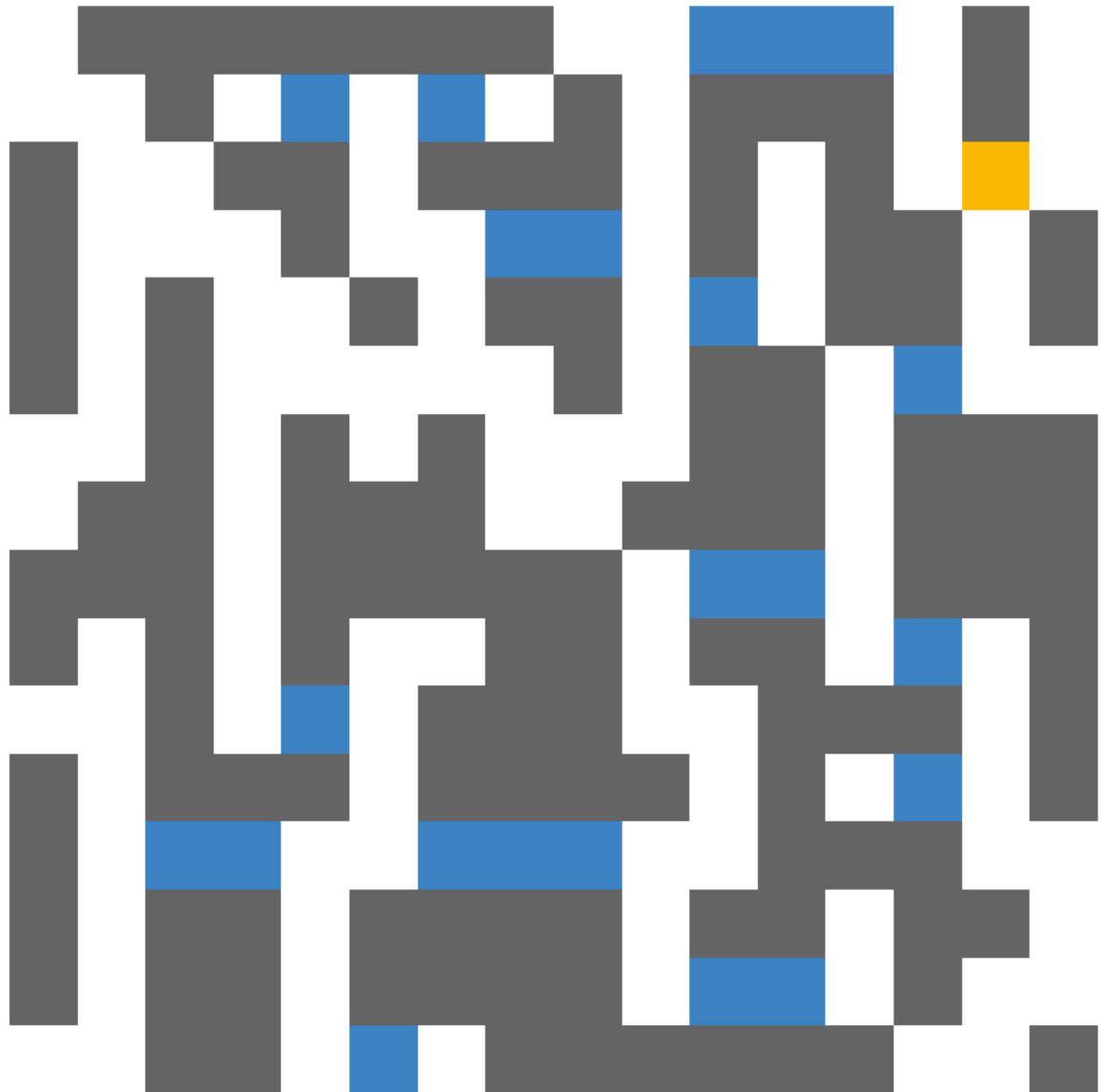
Got this situation, and the real Agent jumped way too far so there is still a problem

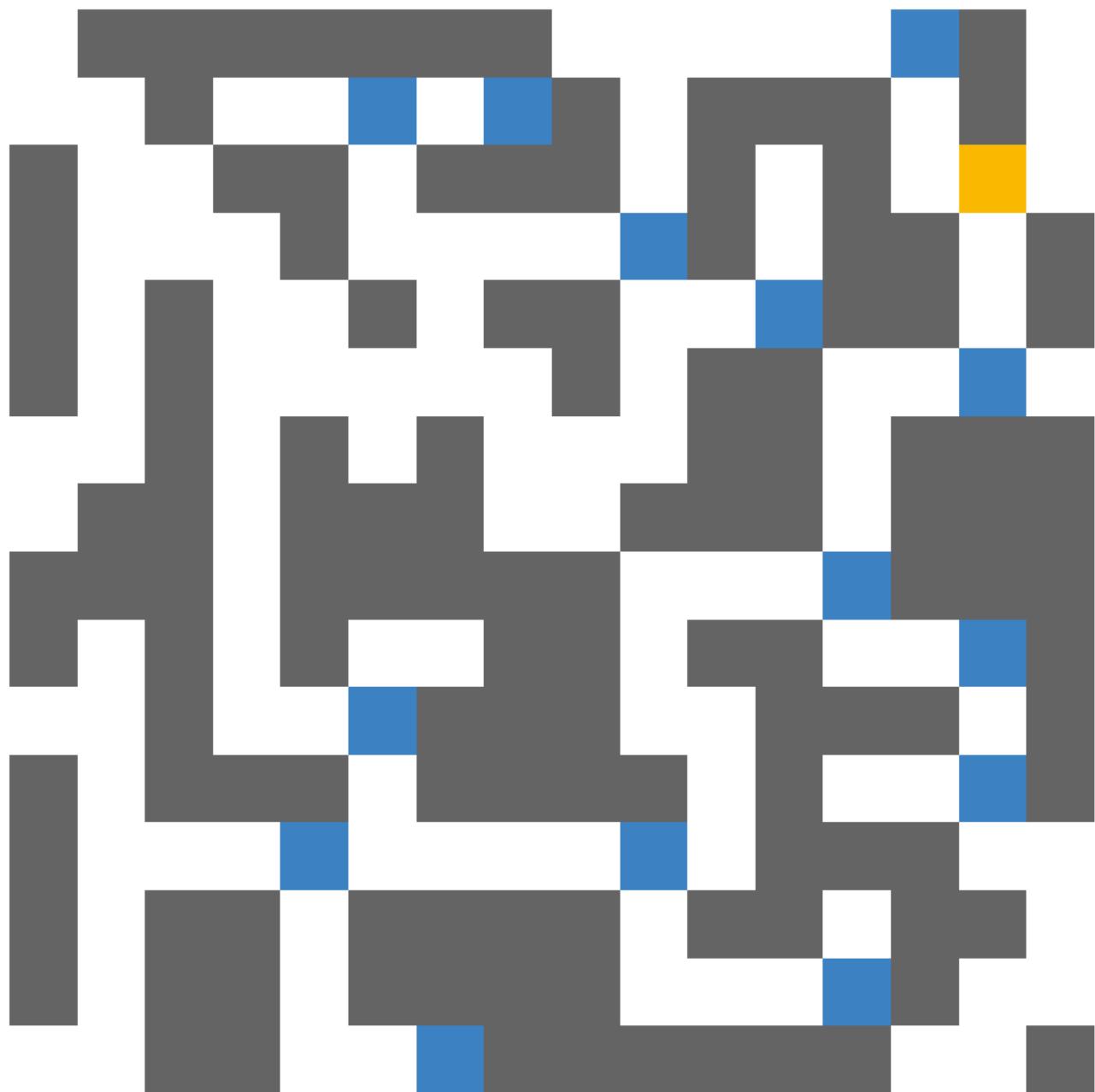
 [Still a problem >](#)

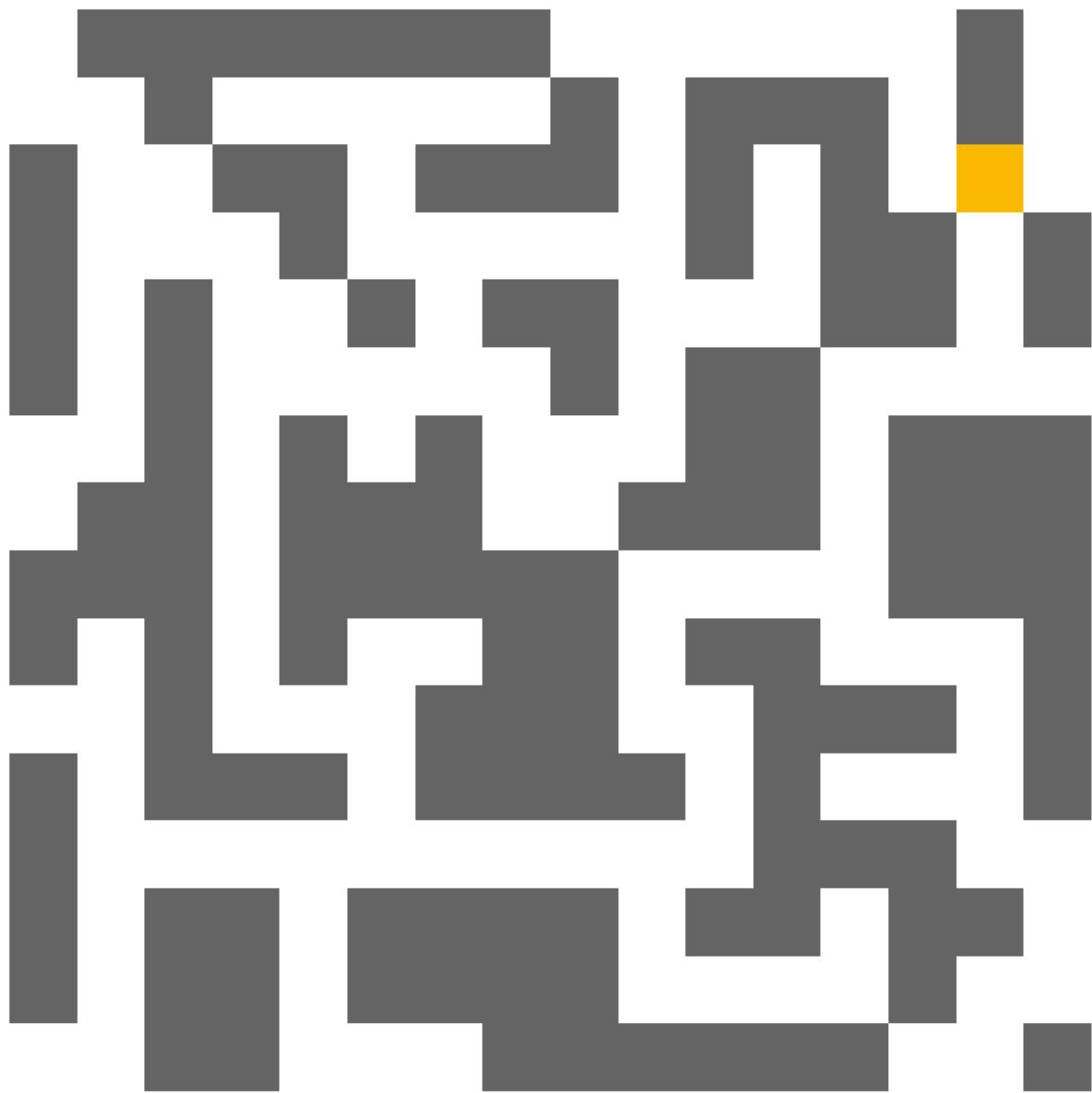
Real Agent



Mini-agents







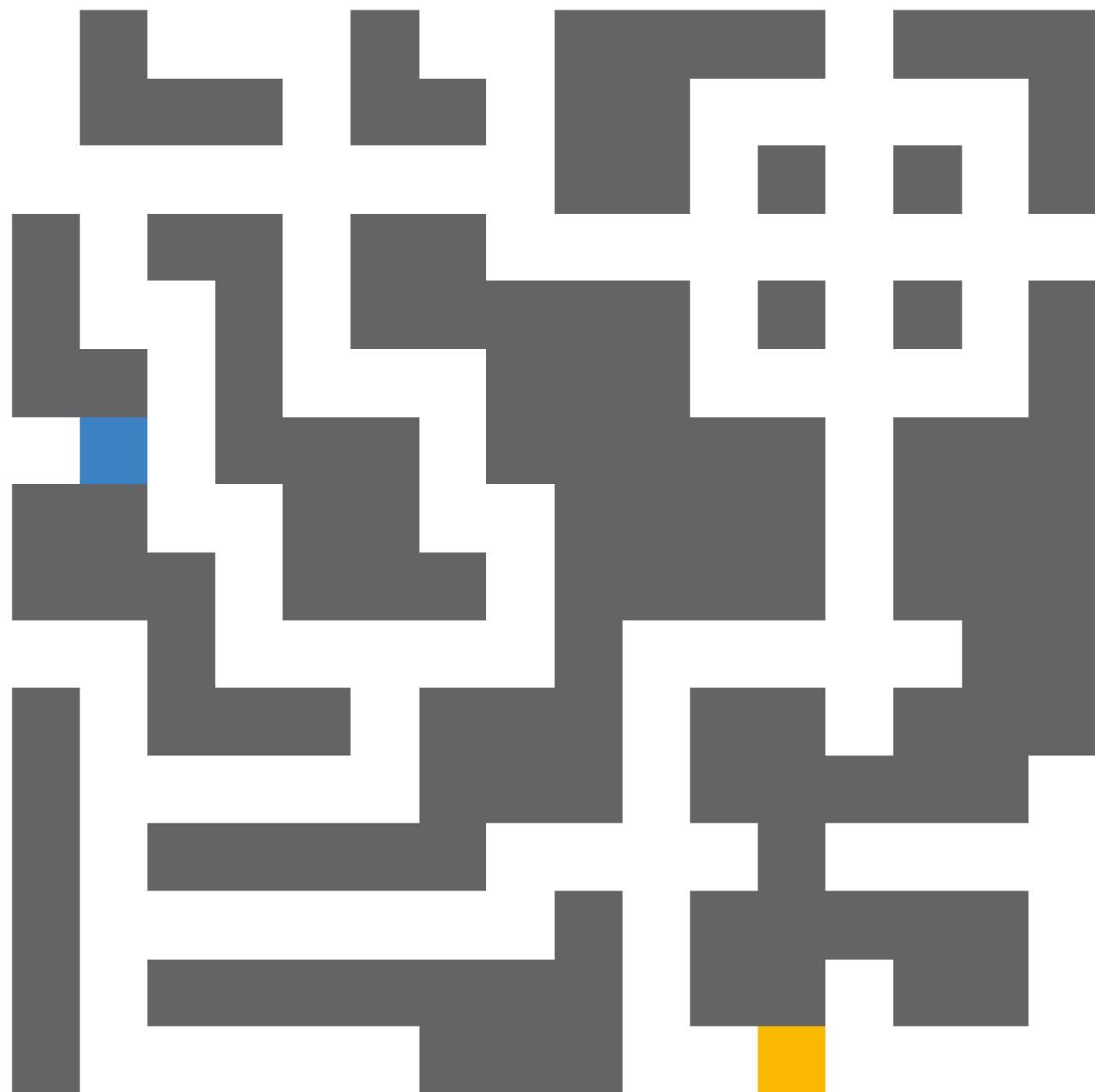
Trying exact solution but for `update()`

```
if self.scan(position)!=percept:  
# To  
if self.scan(position).replace("2", "0")!=percept:
```

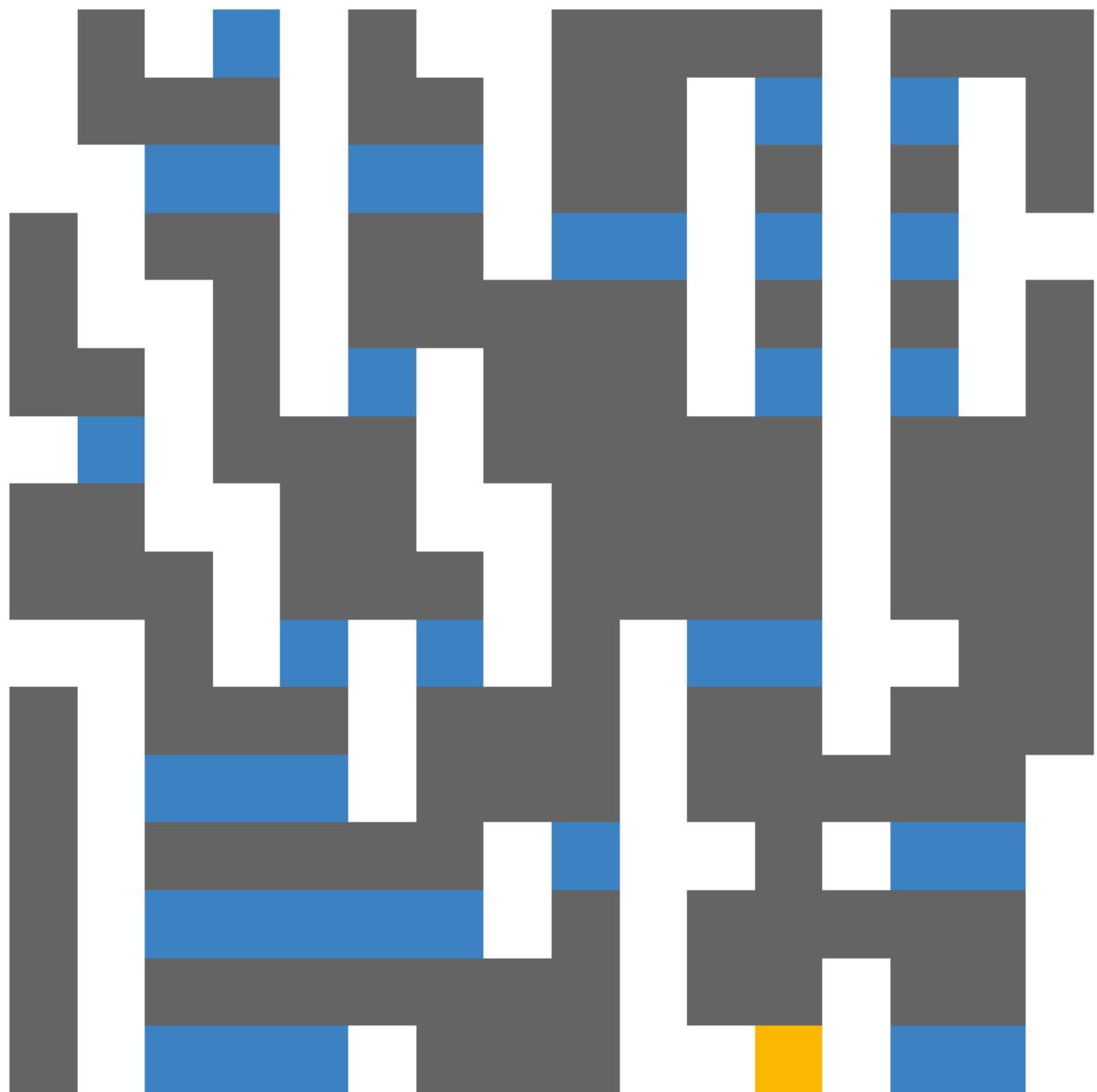
Well it seems it fixed it

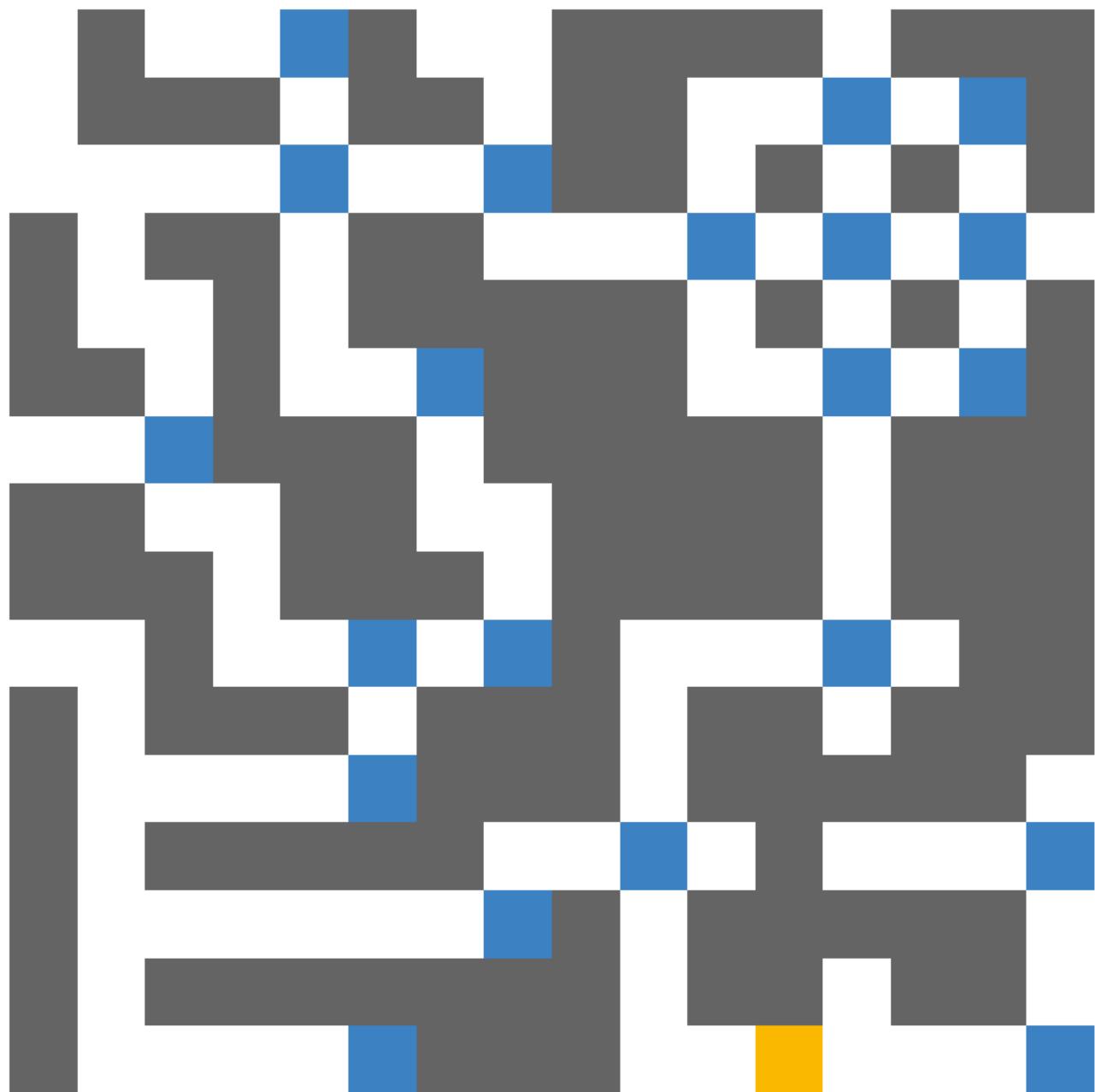
Fixed >

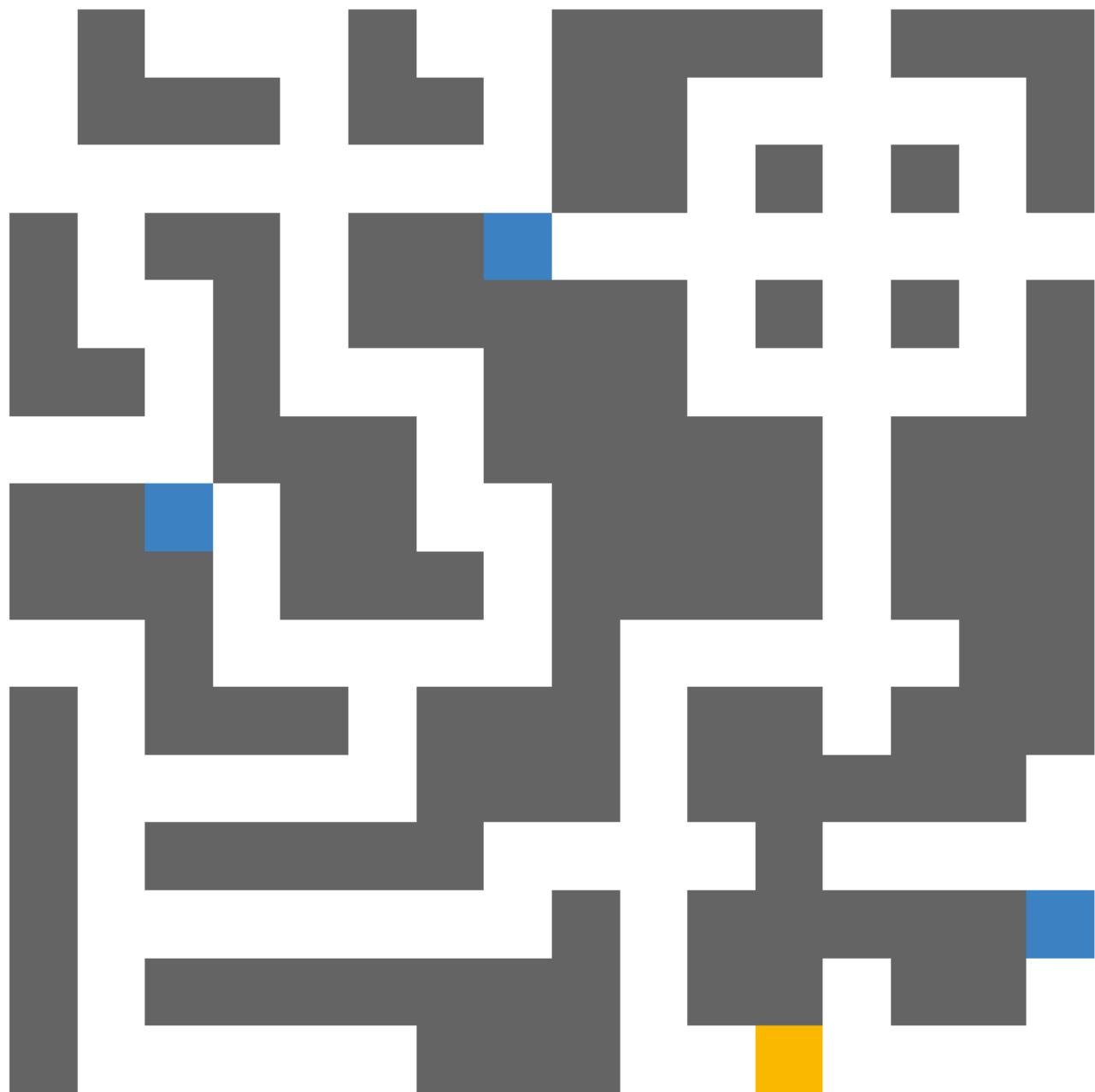
Real Agent

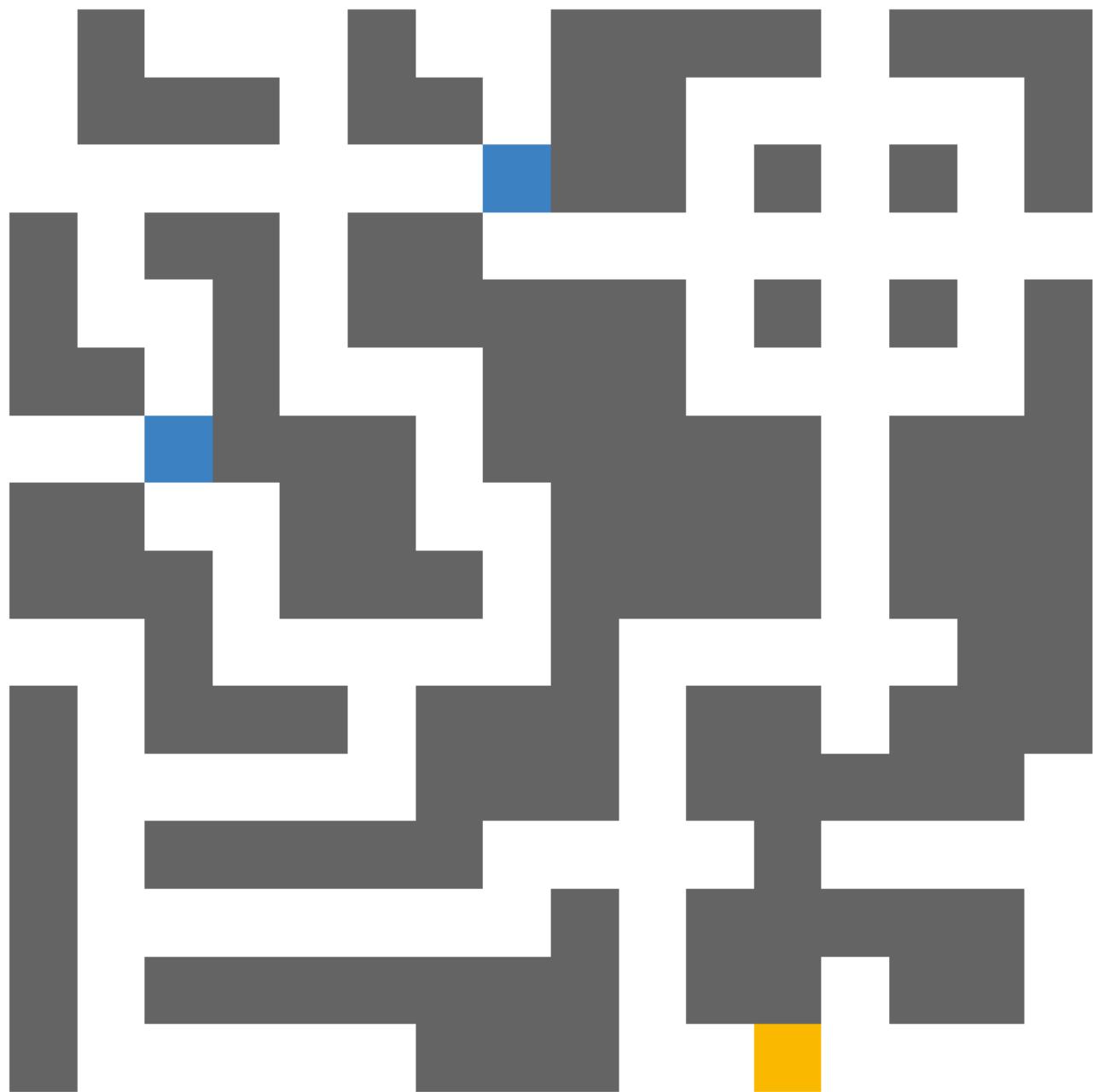


Mini-agents



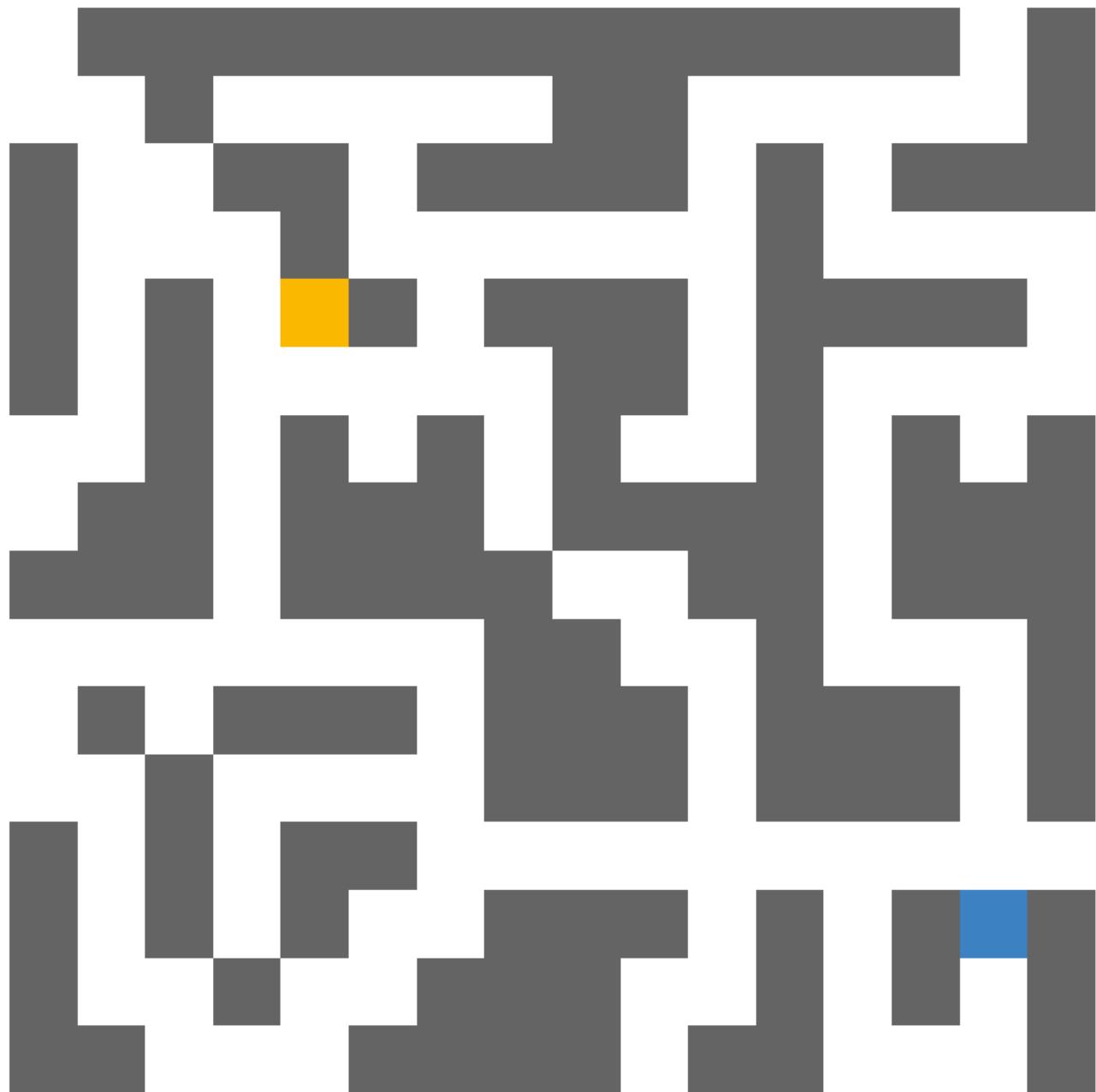


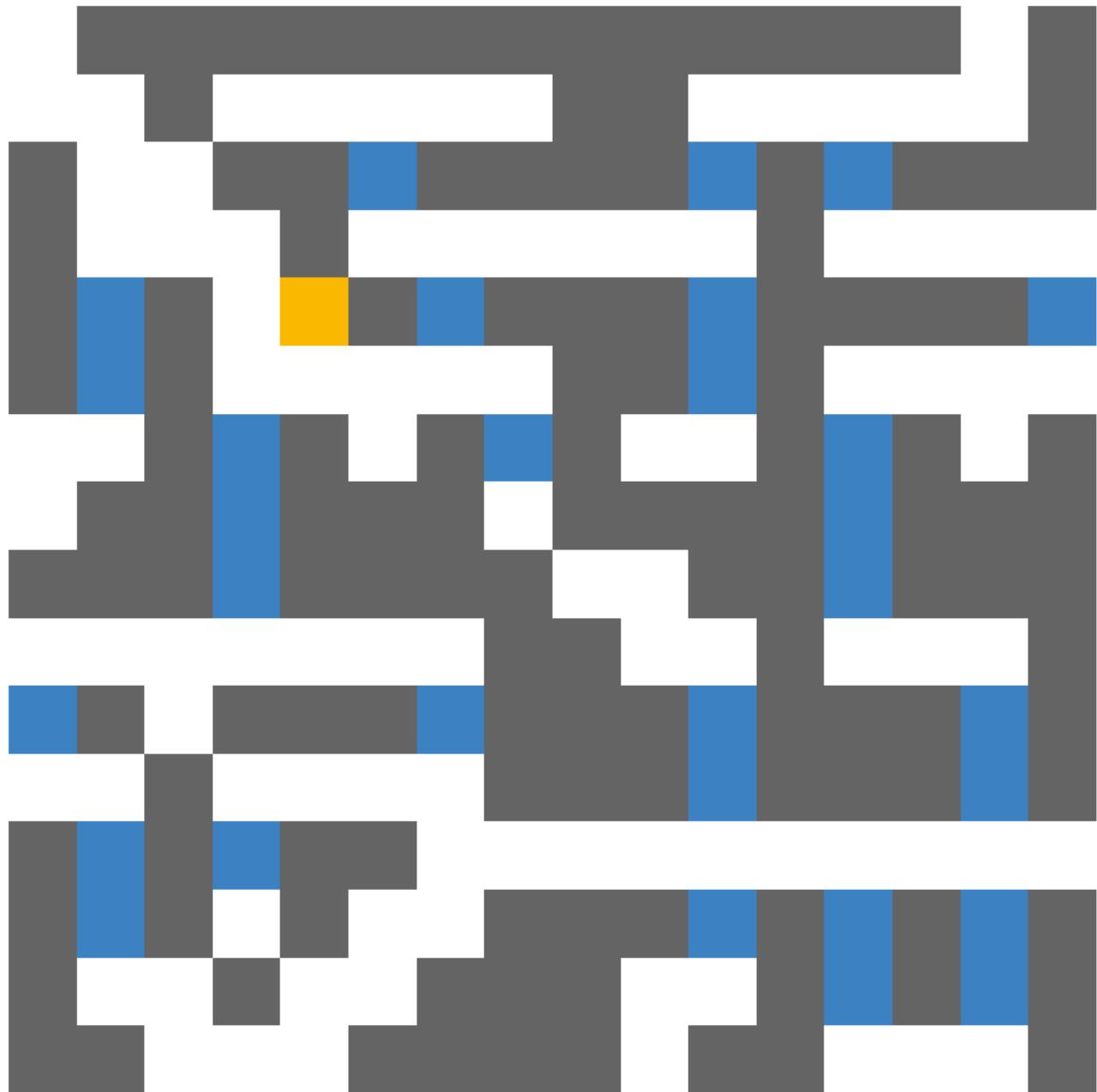


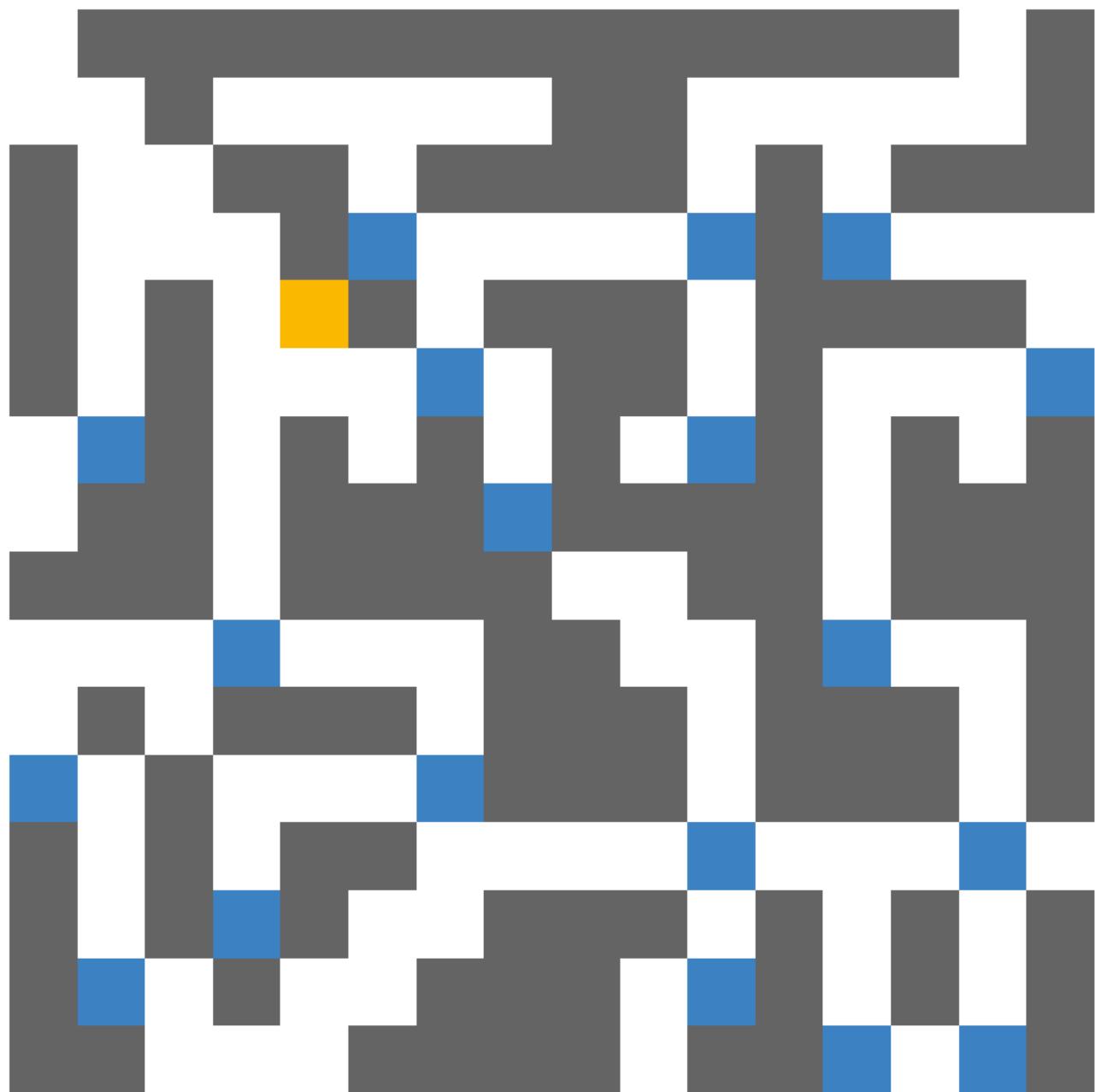


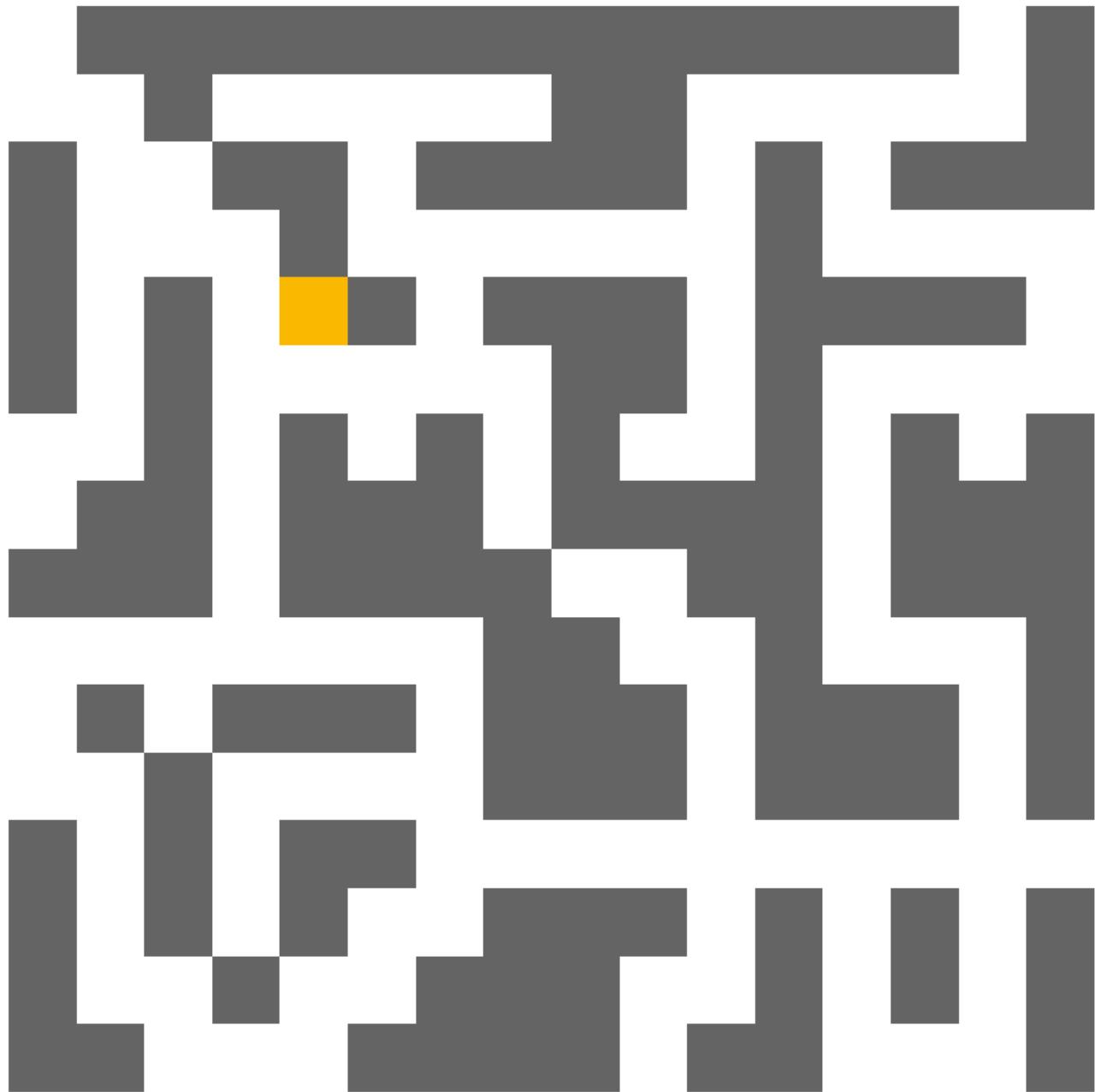
Never mind it didn't

**Not fixed** >









The problem must be that there is some if statement that checks if the square is empty. Or maybe not

Maybe in `move()` this line is the causing the error since it's replacing the old position that could possibly have a mini-agent to 0

```
# Set old position as 0 if Agent moved
if moved:
    self.state[i][j]=0
```

This might fix it

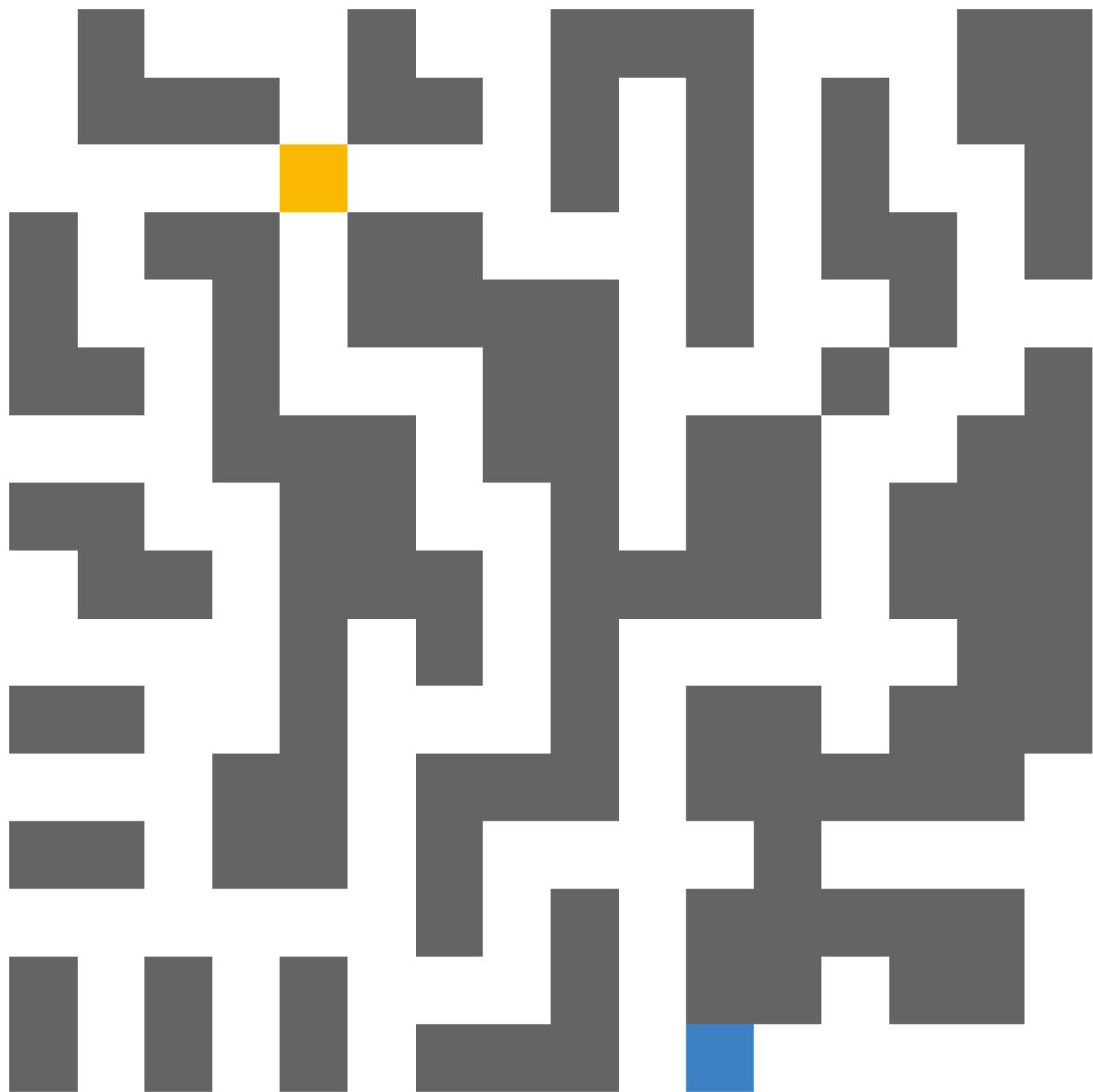
```
if moved and self.state[i][j]!=2:
    self.state[i][j]=0
```

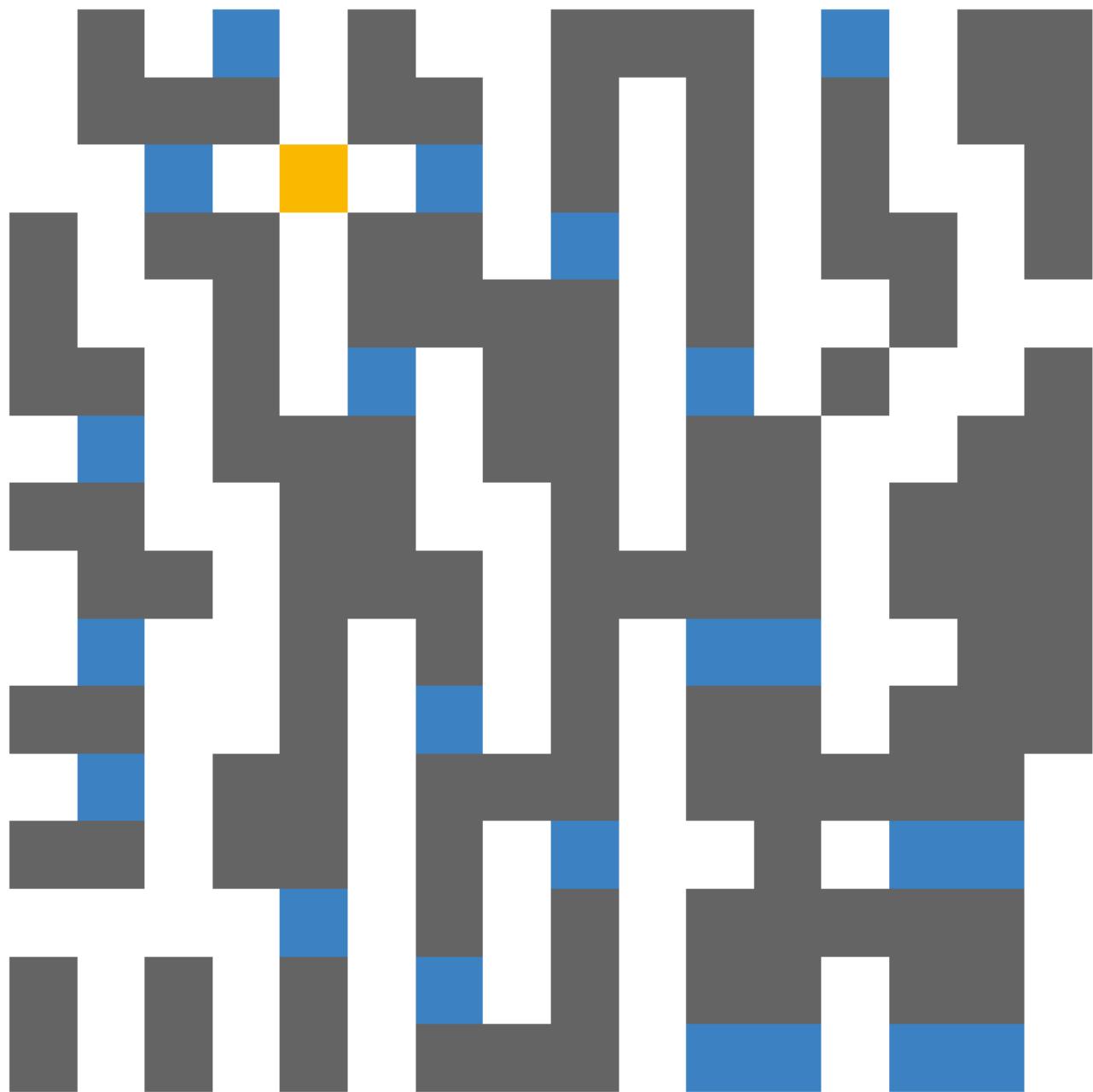
but it won't because the old position will always be a 2, that's the whole point.

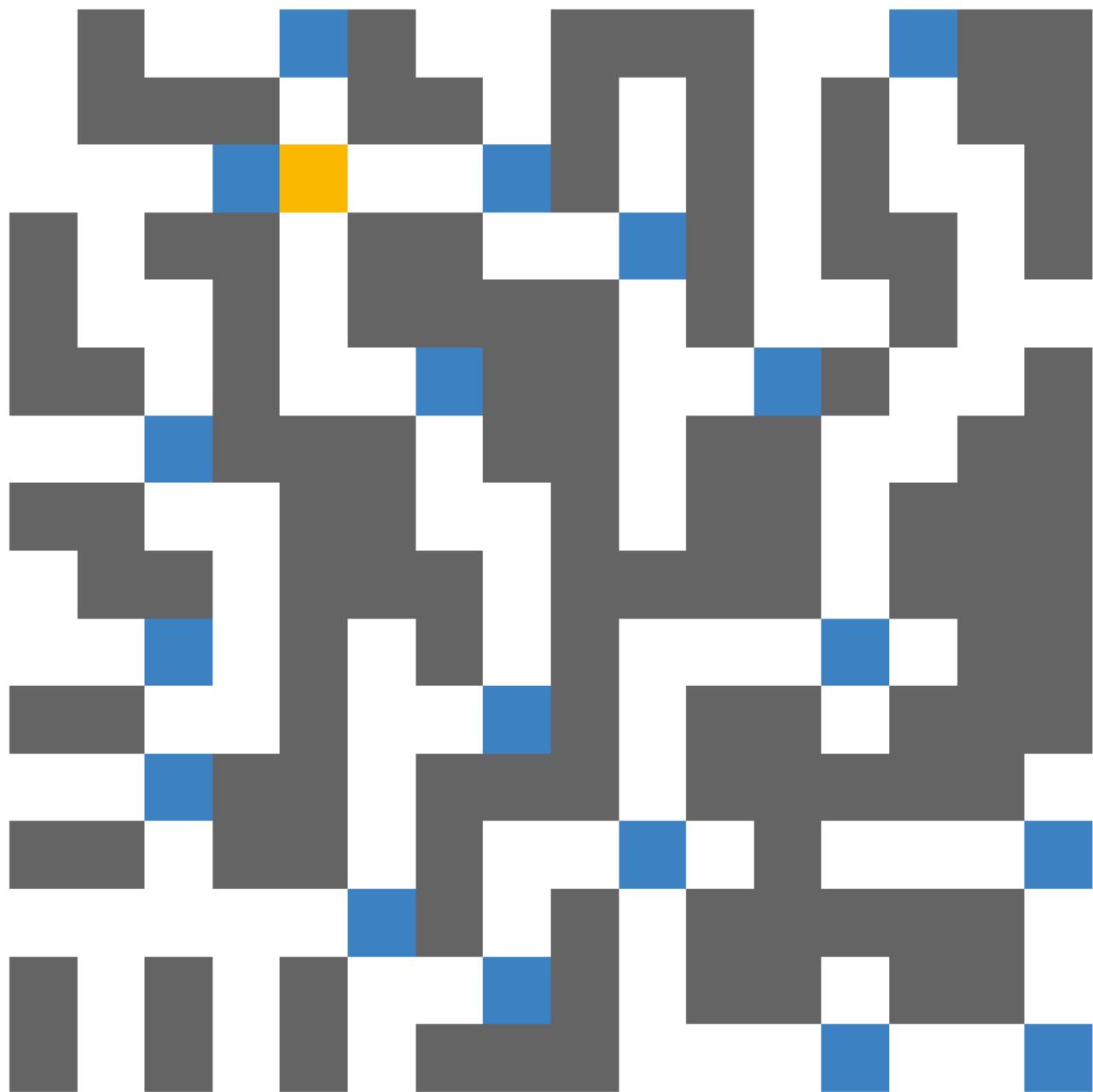
Best solution I can think of right now is to make a whole entire function that checks for this very specific situation

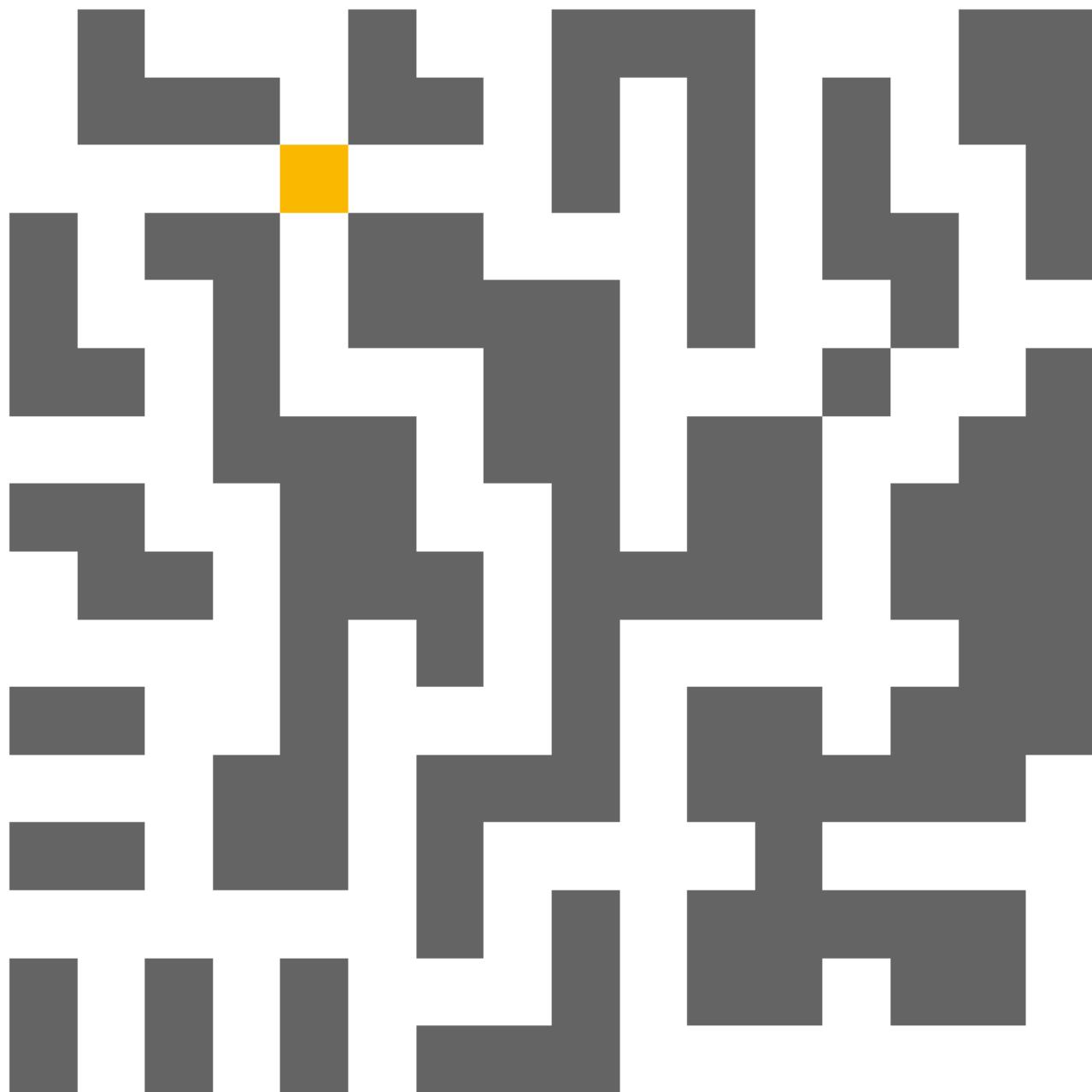
Probably can called the function `if_follow_up_agent()` that turns true or false. I think I'll do it tomorrow

 [Another situation >](#)









Starting on `if_followed()`

Fixed

Instead of scanning the entire board each time to find positions, there will be a list saved of all the positions. Then the operations will be applied to that list.

I don't think this is needed anymore because if we are only editing the list and adding new positions, it won't matter if the mini-agent is followed or not. The board is no longer scanning, just placing. There's no chronological order anymore. The only thing that can go wrong is two mini-agents overlapping each other, which won't be possible.

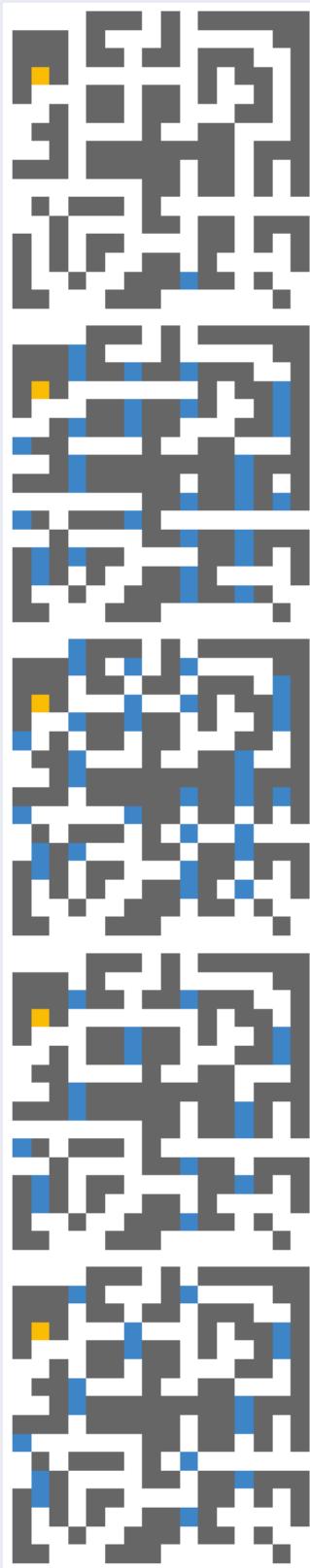
```
# Set old position as 0 as if Agent moved
if moved and not self.if_followed(position):
    self.state[i][j]=0
```

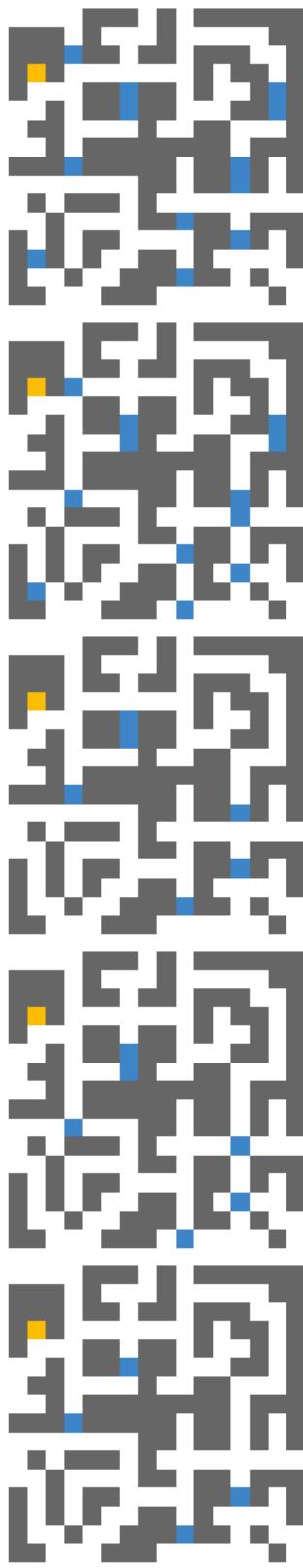
Going to apply [refactoring](#) and reuse the code `place_agents()` with a new parameter that would place specified positions from `self.positions`. The new parameter:

```
def place_agents(self,percept,place_positions=False)
```

Well it seemed to work pretty well. It eliminated the old problem where the moving all the mini-agents were not consistent. And I was able to get rid of the `if_followed()` function. Also was able to not check if the agent moved in `move()`. Yeah everything seems good.

[Working example >](#)







I'll remove `if_followed()`

```
def if_followed(self, position):
    i=position[0]
    j=position[1]

    percept=self.scan(position)

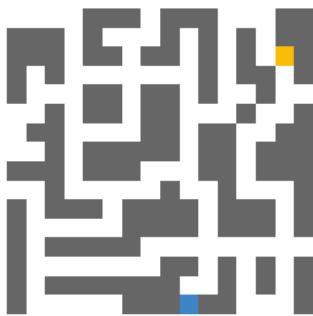
    # If N, check one step behind S
    if self.action=="N":
        if percept[3]=="2":
            return True
        else:
            return False

    # If E
    if self.action=="E":
        if percept[2]=="2":
            return True
        else:
            return False

    # If W
    if self.action=="W":
        if percept[1]=="2":
            return True
        else:
            return False

    # If S
    if self.action=="S":
        if percept[0]=="2":
            return True
        else:
            return False
```

Rare percept that applies to only one square



Going to clean code's comments and then upload to Git

Okay now I need to figure out the `choose_action()` and `cost()`

First I need to have the Agent to know where the goal is

For `choose_action()` I know

- The history of Agent
- Where the goal is
- The entire state
- The Manhattan distance

I don't know

- The real agent (for the first steps)
- The direct path to the goal

To come up with more ideas I'll refer to [Chapter 4](#)

Notes

- We are dealing with a [Conditional plan](#)
- Maybe could have random branches going out and the Agent selects the one that's closest
  - The Agent knows what quadrant the goal is in
    - The Agent knows what quadrant it's in and can try to go to the same quadrant as the goal. From there it tries different paths
  - The agent can have 4 branches that go directly N, E, W, or S, and they try their best to go that far for a certain amount of steps, and the agent sees the one with the lowest [Manhattan distance](#)
- The problem with AND-OR-GRAH-SEARCH code is that it has to return instructions based on a very specific state. All my states will always be different. It's uncountable

```
function AND-OR-GRAPH-SEARCH(problem) returns a conditional plan. or failure
    OR-SEARCH(problem.INITIAL-STATE, problem, [])
```

---

```
function OR-SEARCH(state, problem, path) returns a conditional plan. or failure
    if problem.GOAL-TEST(state) then return the empty plan
    if state is on path then return failure
    for each action in problem.ACTIONS(state) do
        plan ← AND-SEARCH(RESULTS(state, action), problem, [state | path])
        if plan ≠ failure then return [action | plan]
    return failure
```

---

```
function AND-SEARCH(states, problem, path) returns a conditional plan. or failure
    for each si in states do
```

which is way more efficient.

```
        plani ← OR-SEARCH(si, problem, path)
        if plani = failure then return failure →, if it finds a cycle, return failure instead of checking for a path repetition
```

- We are not dealing with a sensorless agent, which is Section 4.4 in [Chapter 4](#)
- Not using any predictions since we know what the next step will look like. We know what any step can look like
- The only uncertainty we have is what is the real agent.
  - Maybe the agent can move randomly or in a way that will eliminate as many mini-agents as possible, and then the path finding can start.
- My AI has non-deterministic actions cause it's not possible to guess what will happen. I guess it's not possible to know what mini-agents will stay or not.

- I have not really found a direct solution from my notes that could work for me. Though it's possible I missed it. I'll start a new branch and start coding my own way it'll find its path

I'll start a new branch and start coding a function that moves the agent until there's only one left

When I want to switch back to the branch I need to type

```
git checkout "Function-to-eliminate-mini-agents-to-only-realmini-agent"
```

had a typo with real-mini-agent

Another idea is how the mini-agent doesn't necessarily have to go to a square to know its cost, we can have the AI be able to check its surroundings and then go on the square with the least cost. Make a new matrix of the mini-agent's surroundings.

Maybe at one point while coding I need to make a function that forces the AI to move a sequence of actions

Also got to make a function that gives possible directions the agent can move

Okay I got `possible_actions()` coded but not sure if need use -2 instead of -1 so the visual matches the print. I'll have to check that. I think I can keep it to -1 and the `visualize()` is out of order

```
def possible_actions(self, position=None):
    # Initiate list
    possible_actions=[]

    # If the position is not given, find the percept. Else assume it's the most recent percept
    if position!=None:
        percept=self.scan(position).replace("2", "0")
    else:
        percept=self.data["Percept"].iloc[-1]

    # Find open squares
    for i,number in enumerate(percept):
        if number=="0" or number=="3":
            # N
            if i==0:
                possible_actions.append("N")

            # E
            if i==1:
                possible_actions.append("E")

            # W
            if i==2:
                possible_actions.append("W")

            # S
            if i==3:
                possible_actions.append("S")

    return possible_actions
```

It has been 3 weeks since I last have worked on this code. Going to refresh myself on the email I sent to Manuel before going on vacation

Important notes:

For more info on how to code the AI, use chapter 3

I'd focus on a deterministic resolution. So, chapter 3 is your best friend.

Will be using greedy best search in real time to move around

In one of your options, you describe something akin to a greedy best search or A\*. That could very well work! I'd say start with that, define the cost and instruct the agent to move where it is lowest, we'll see after if there should be more. To be sure, calculate cost in a separate function, so it is not bound to a specific algorithm, and can be easily reused.

Today will need to code agent isolating itself

Two ways could be used. As you are suggesting, you could either work out where exactly the true agent is by moving around a bit, then solve for that one agent. I think this would be the best option.

Would need to code a function that makes sure agent does not go back the path it came from. Could already be taken care of by the algorithm

However, I can very well see the agent running in circles or getting stuck. So, you'll need to keep in memory the sequence of squares visited, in order to compare with it if the same sequence of two squares is in it. If so, the agent is probably on a loop and should be forced out of it (maybe by putting a high cost, like we discussed?).

My minimum goal for this week

First thing I'll try to code after the `single_out_agent()` is the function for the greedy best search. It will be similar or same to what I have explained since it's calculating each next square in real-time. And I'll make sure to have the cost function be independent. It should be the accumulated steps + any other extra cost ideas I have down the road.

Let's stick to real time, no planning. It's all going to be the A\* algorithm at play doing the magic. I just got to code it.

As for real-time vs planning, I'd tend to stick to one or the other, but the choice is yours.

Starting on coding `single_out_agent()`

Realized I'm still working on `possible_actions()`

I can confirm `possible_actions()` function does find the most recent possible actions in the dataframe, yet, when the code is executed, there's an extra `self.visualized()` that's executed. It goes past the dataframe and the `possible_actions()`

Either way, the function works as long the data is updated. It needs to placed after the `self.save_data()`

Going to now start on coding `single_out_agent()`

We don't want to try to do as many directions as possible, we want to try to get as many unique percepts as fast as possible.

We would need to use history and scan

Simple code that finds the first instance a value is shown. Checks for repetition. Checks if a new value appears:

```
import random

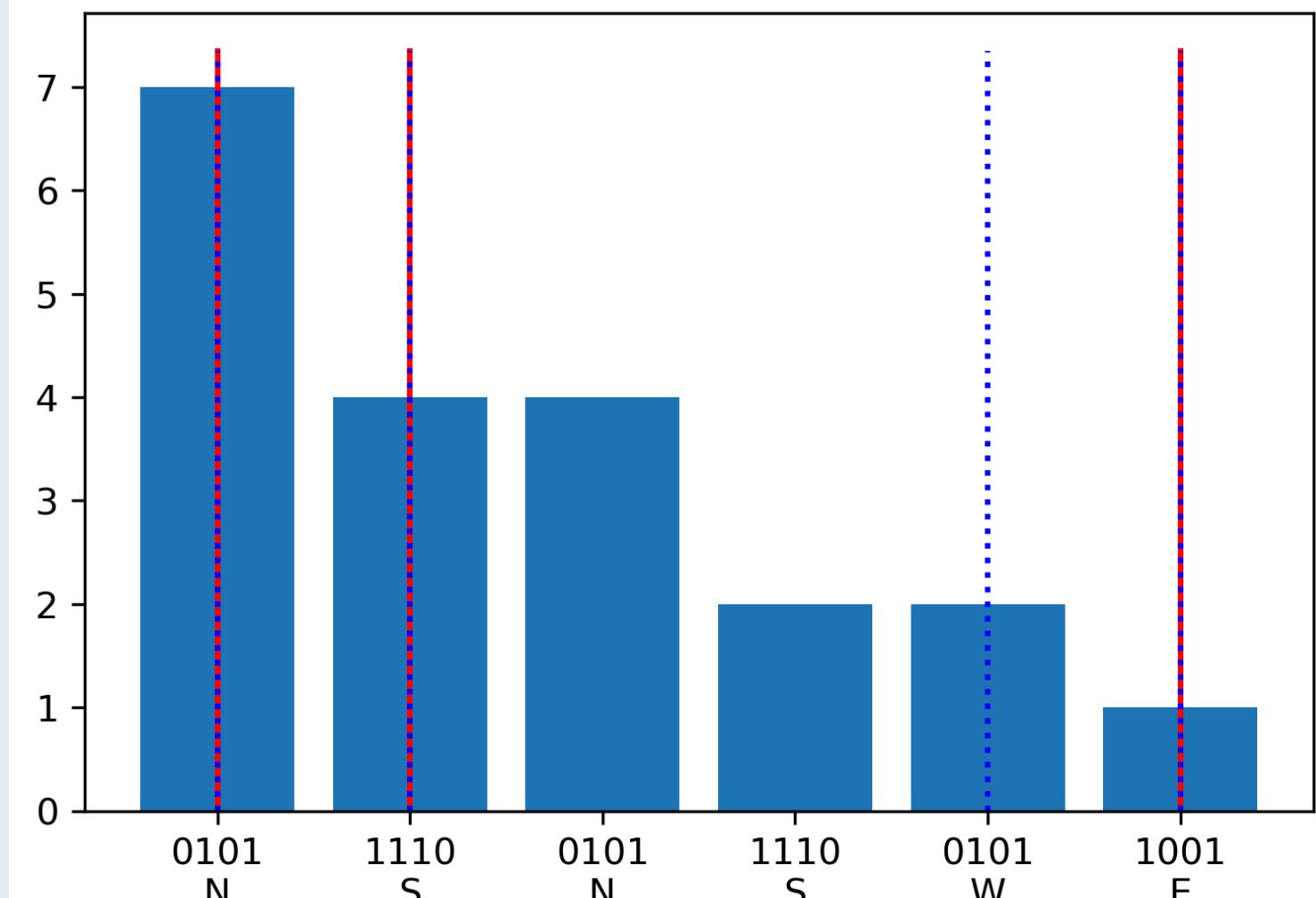
# Random list
l=[random.randint(0,5) for i in range(20)]

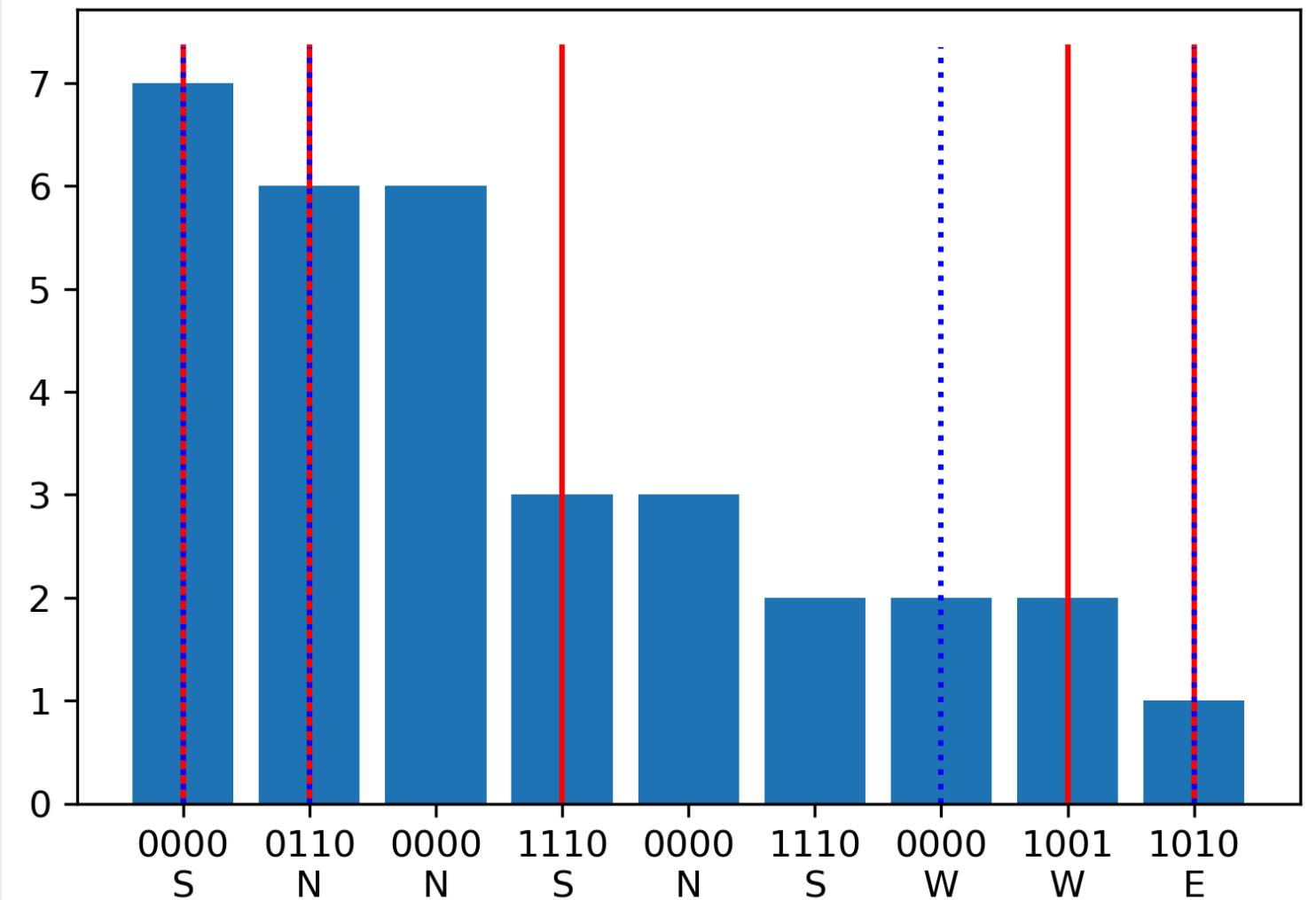
for i,number in enumerate(l):
    if number not in l[:i]:
        print(number)
```

To see if a percept change or move change leads to agent eliminations, I made a code that graphs the percept/move vs. length of positions. A red line indicates that the percept is unique, as a blue line indicates that the move is unique.

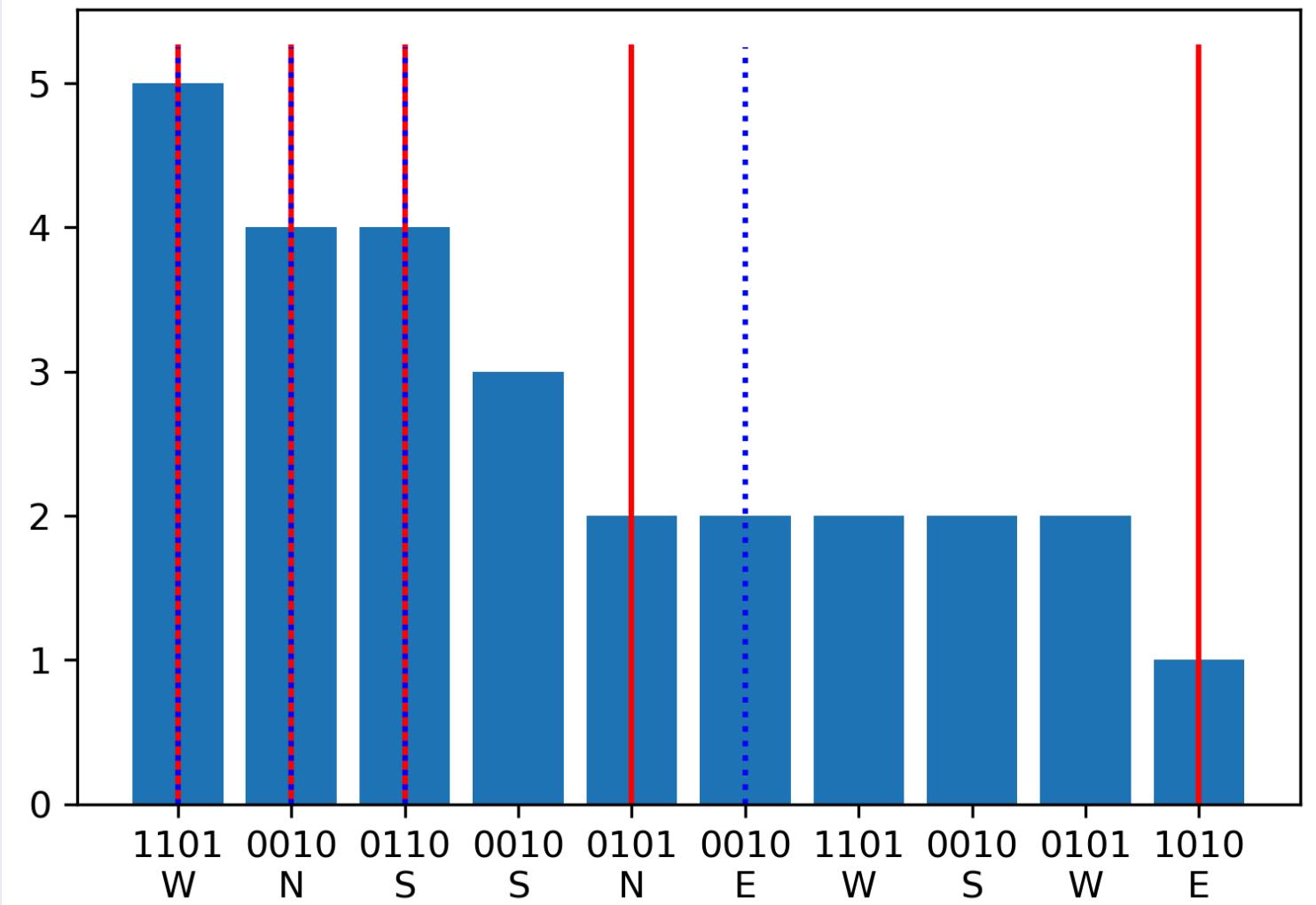
 Graphs >

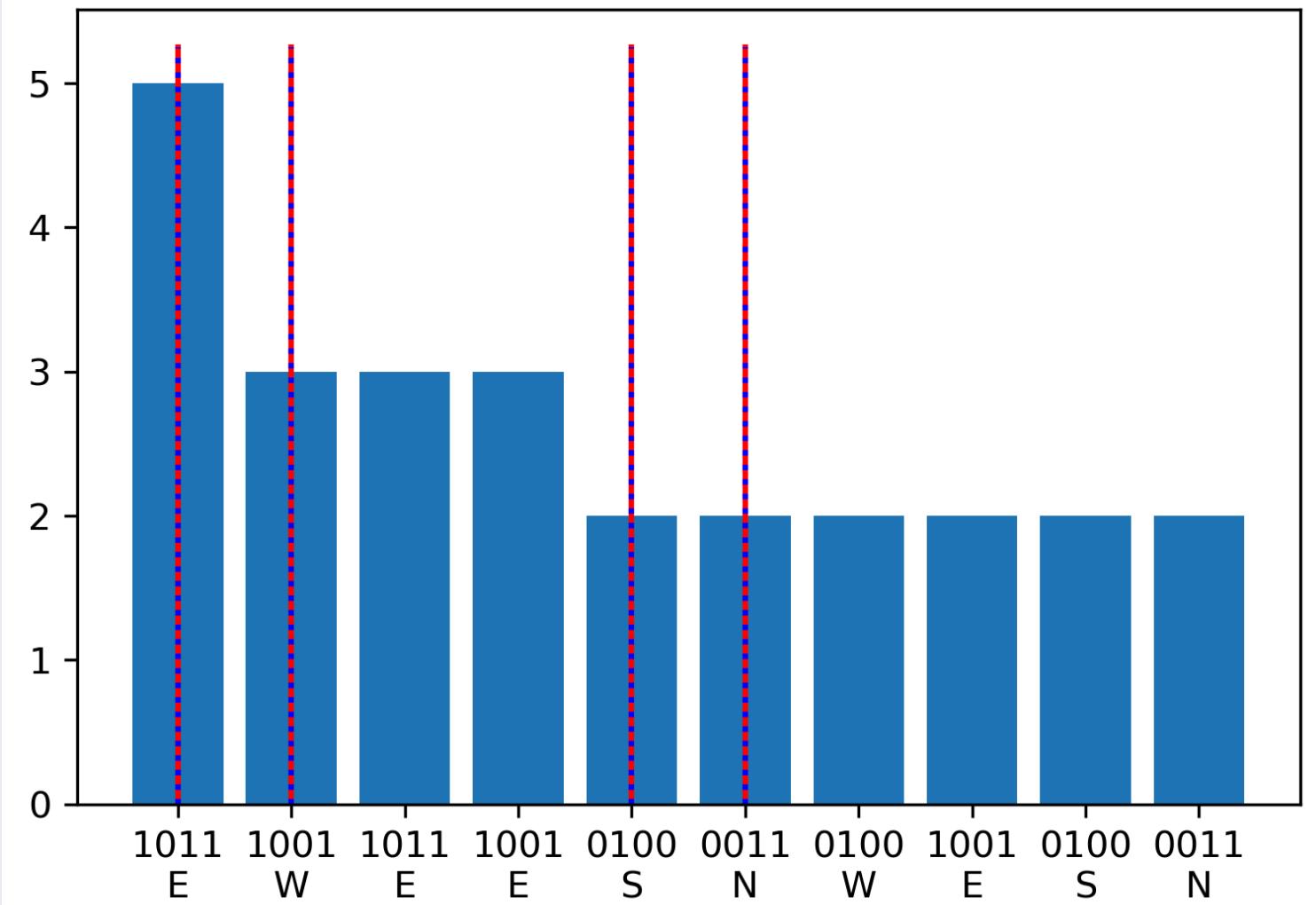
Going to count the amount of times red lead to change vs blue lead to change

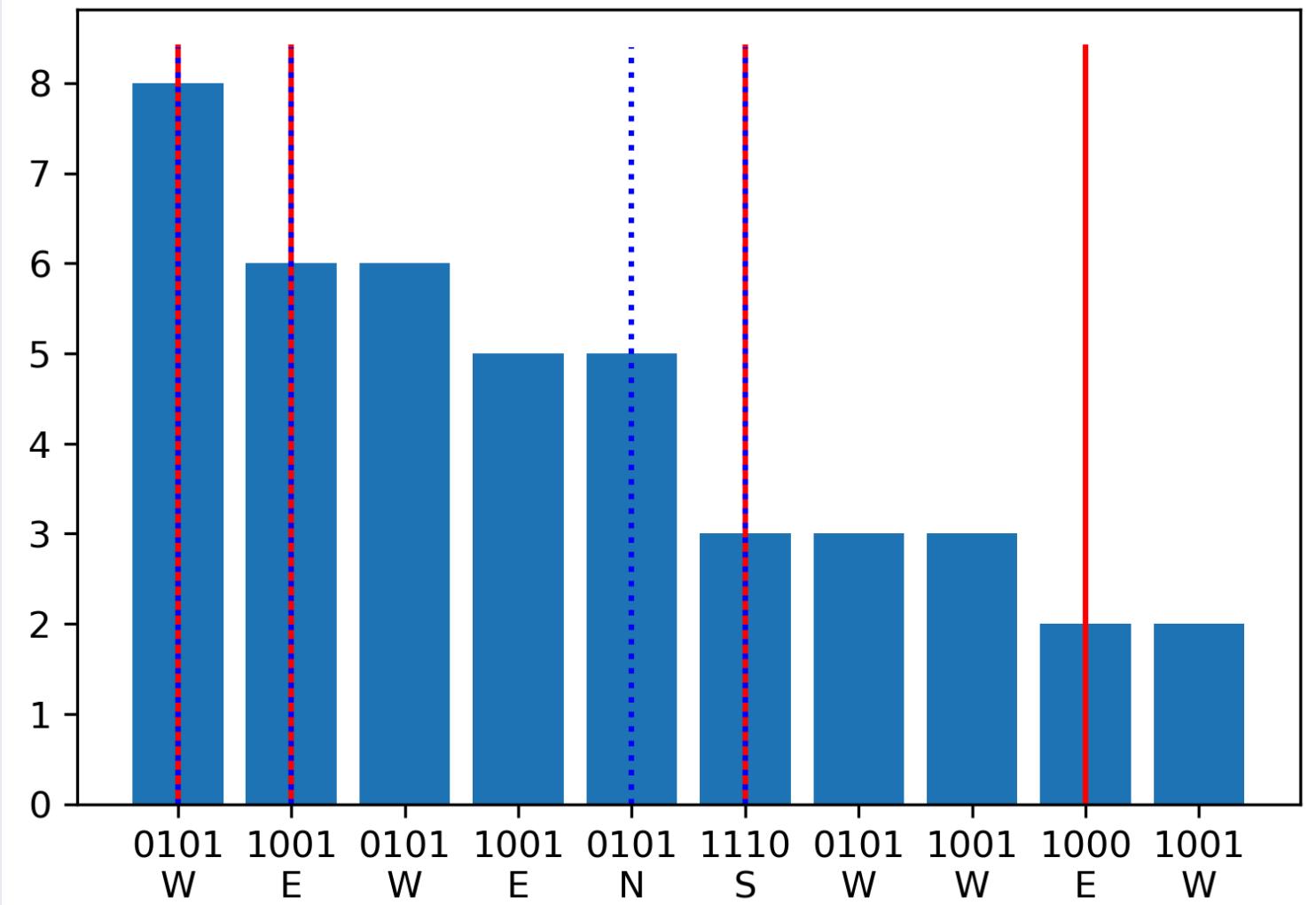


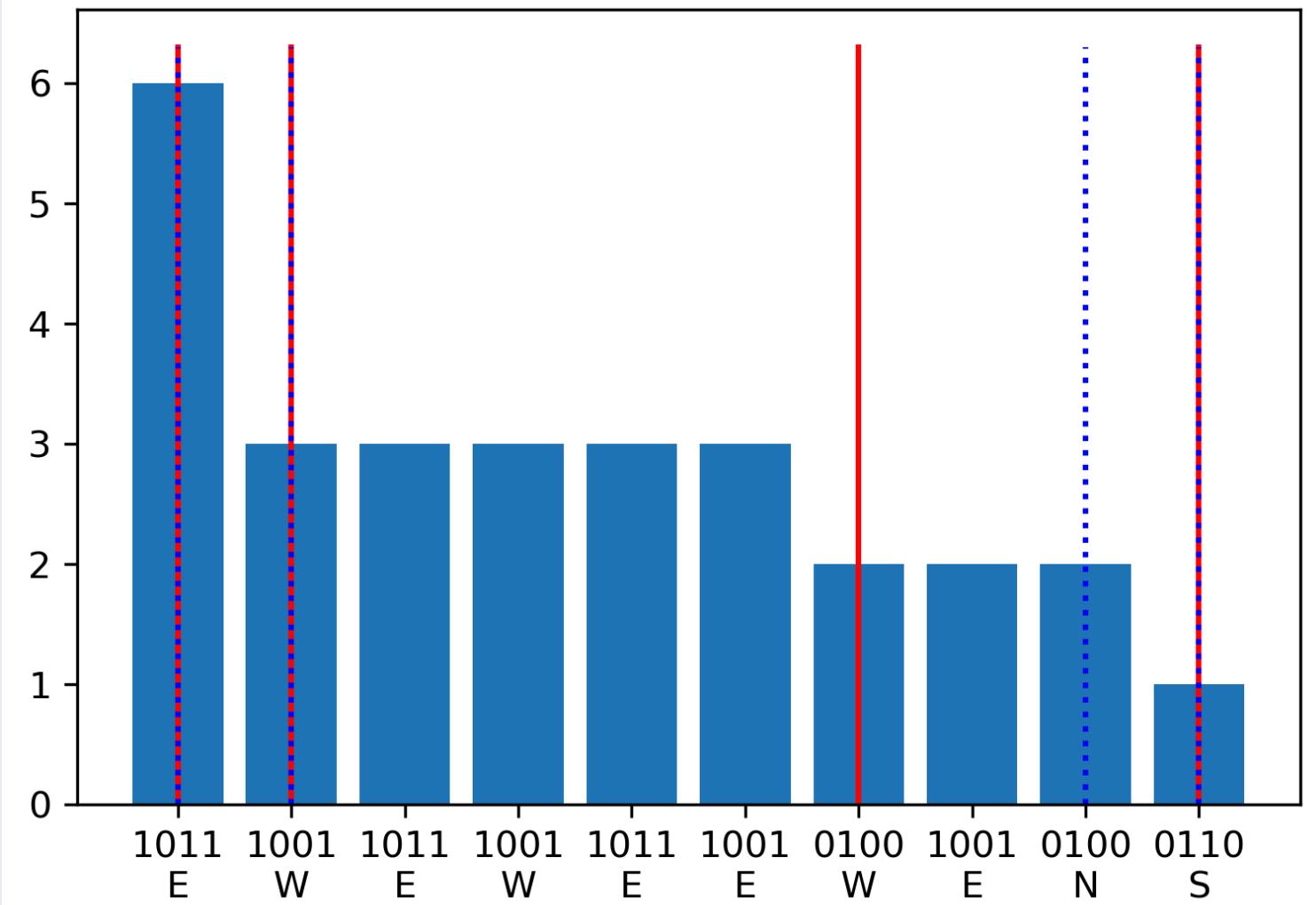


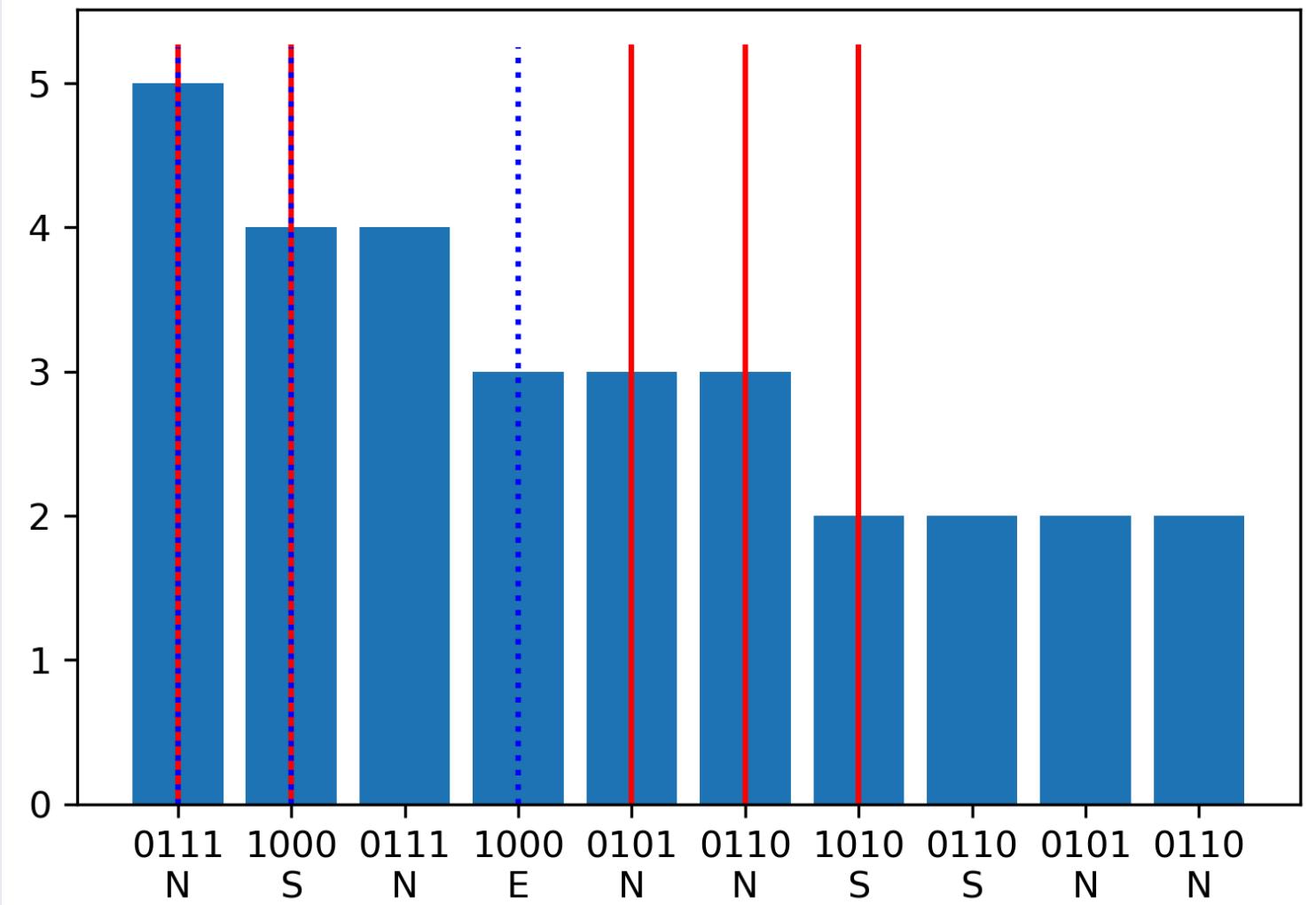
4 - 2



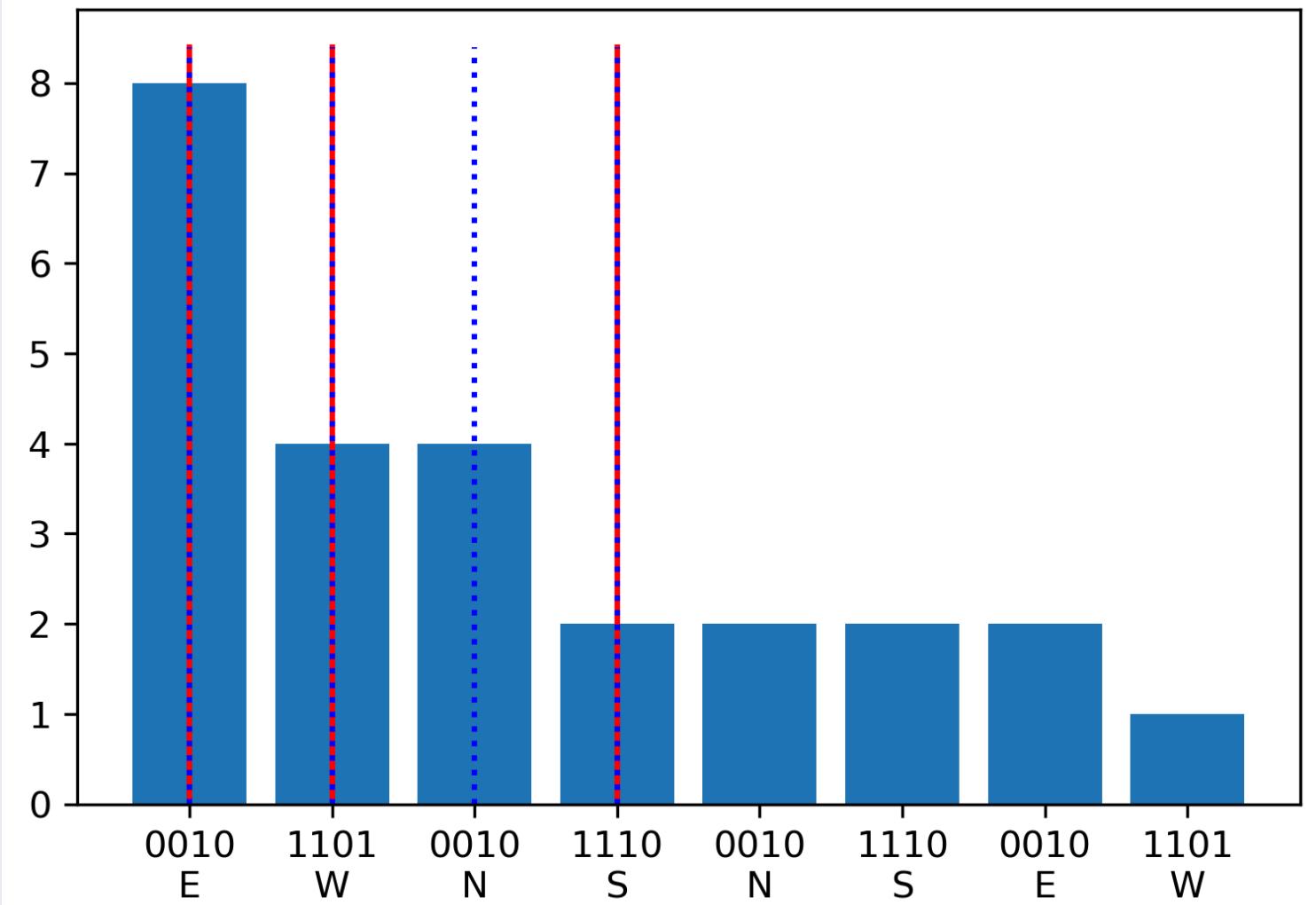




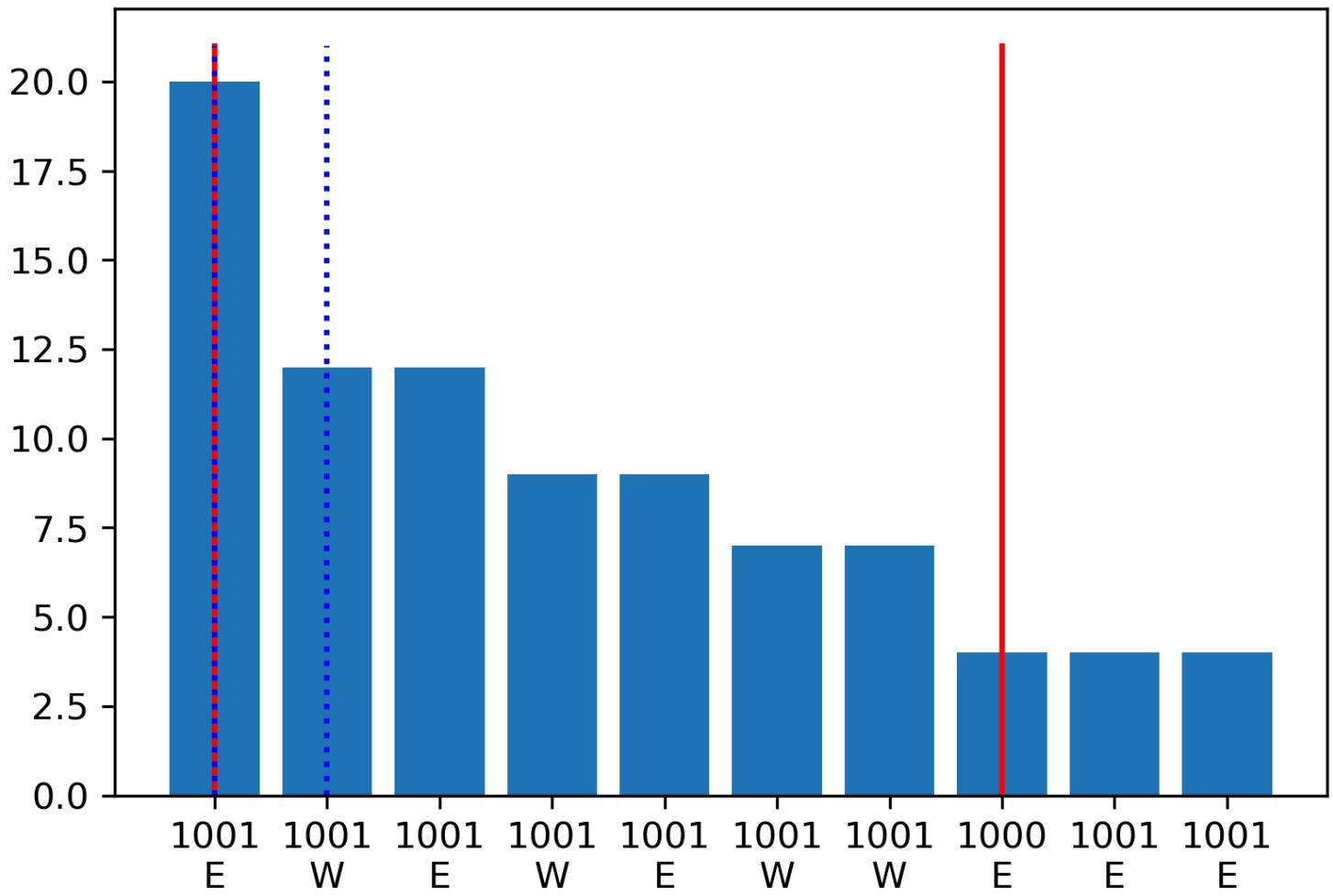




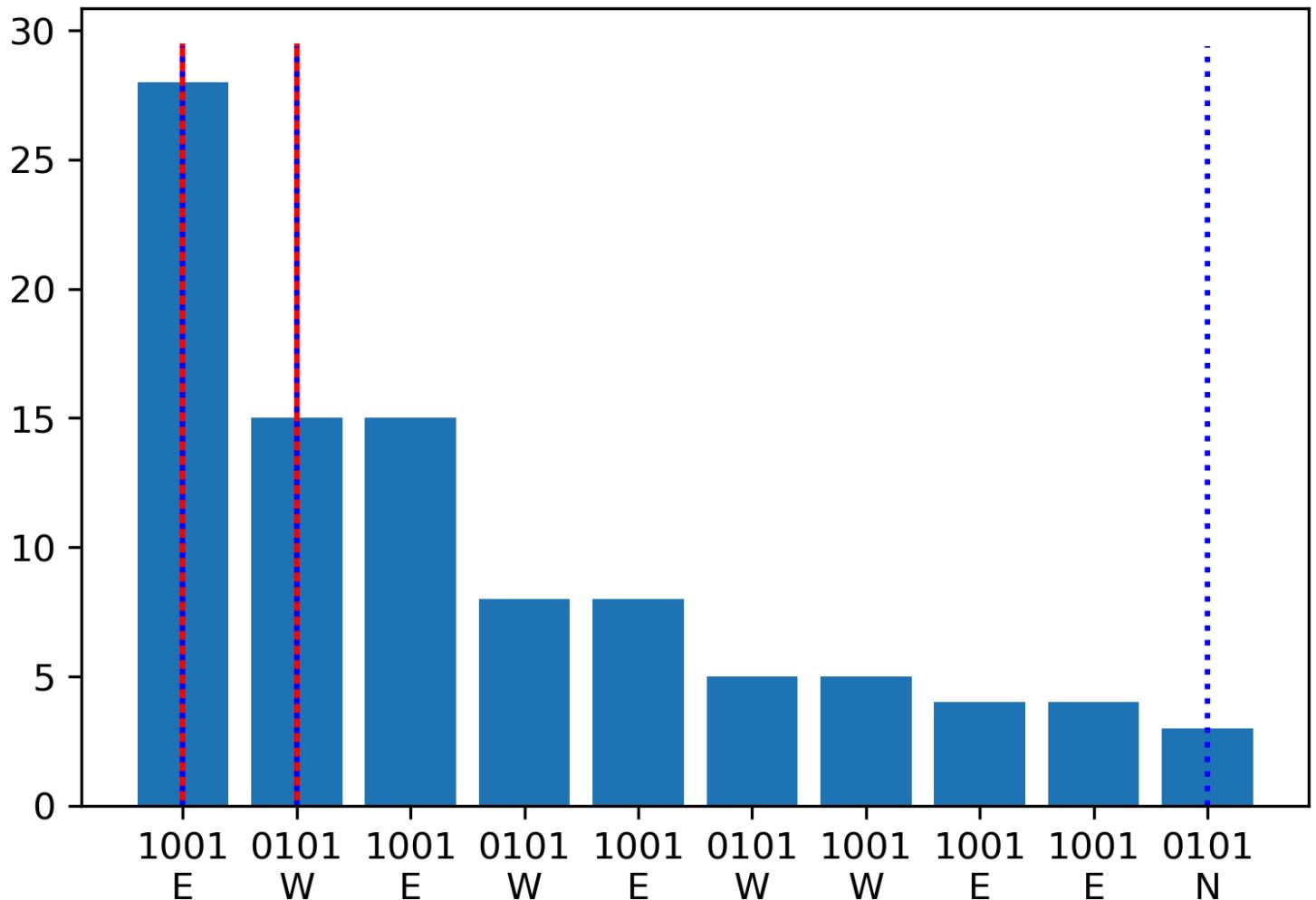
3 - 3



2 - 2



3 - 3



In total there's 33 times the positions decreased when the percept changed, and 27 times when move did.

This data could be unreliable since there are more possible percepts than moves. Moves has 4 possible values.

Either way, I originally wanted to focus on changing the percept in the `single_out_agent()` function, and the data shows this is also possibly a better idea than changing moves.

I ran another experiment but this time with less bias.

I calculated the amount of times a bar changes (positions decrease). Then, I calculated the amount of times a bar changes AND percept changes. Same for move.

Then, I ran 58 executions of the maze and collected this data:

percept_change	move_change	total_changes
5	5	5
3	3	4
4	2	4
2	2	2
4	4	4
4	4	4
4	5	5
2	2	2
3	4	4
3	3	3
3	5	5
2	4	4
5	5	5
2	1	2
4	4	4
3	2	3
2	1	2
3	3	3
3	4	4
1	1	1
4	3	4
2	2	2
1	1	2
3	5	5
3	5	5
3	3	3
2	2	2
1	4	4
3	3	3
2	2	2
5	4	5
3	1	3
2	2	2
2	4	5
1	1	1
5	5	5

1	4	5
4	4	4
1	4	4
3	3	3
3	3	3
3	3	3
2	2	2
2	2	2
3	3	3
2	2	2
2	2	2
0	1	1
5	5	5
2	5	5
4	4	4
3	4	4
3	3	3
5	5	5
3	3	3
2	2	2
3	4	4
5	5	5
165	184	197

The chance of a position decrease + percept change is

as the chance of a position decrease + move change is

So there's technically a higher chance that if the move changes agents will be eliminated. Which makes sense because the agent will have to move.

I'll think overnight if I should focus on changing the percept or move in the `single_out_agent()` function

I'm going to try to make the code easily adaptable to either. I'll first try with percept. Trying to get as much of a variety of percepts as fast as possible.

Here is the Python I used yesterday to calculate Percept vs Move

 [Percept vs Move.py](#)

 [Percept vs Move.txt](#)

The plan for the code will be:

1. Know the surrounding positions and which are possible
2. Choose the direction that leads to a unique percept in the DataFrame
3. Go there

Making a new function that finds the percept based on the direction given.

New plan because a direction is not a guaranteed percept. There are many mini agents.

1. Know which actions are available
2. For each mini-agent, collect the percept for each available action. Now we know the chance of an action's resulting percept
3. Choose action with the highest chance of a unique percept.

Another plan?

Go through each mini agent one by one. If that mini agent can move to a percept that is unique, then go that direction. This will slowly pick off mini-agents by forcing them in new positions. Not sure if there will work but I have a good feeling about it. I'll try this method first since it's simpler

First got to code `action_to_percept()`

Done that code

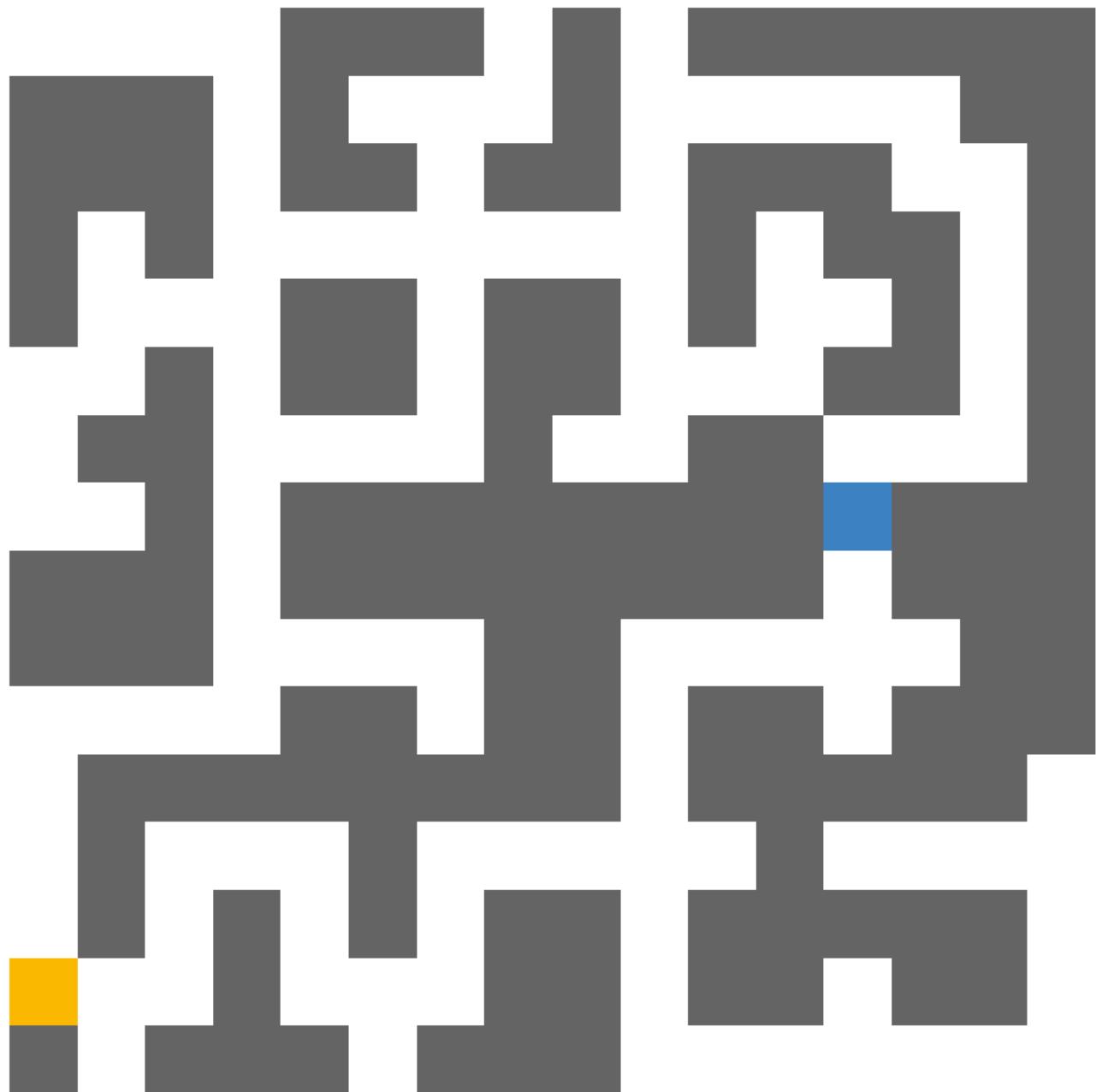
```
def action_to_percept(self, position, action):
    print(position)
    # Based on action, update position
    i=position[0]
    j=position[1]
    if action=="N":
        i+=-1
    elif action=="E":
        j+=1
    elif action=="W":
        j+=-1
    elif action=="S":
        i+=1

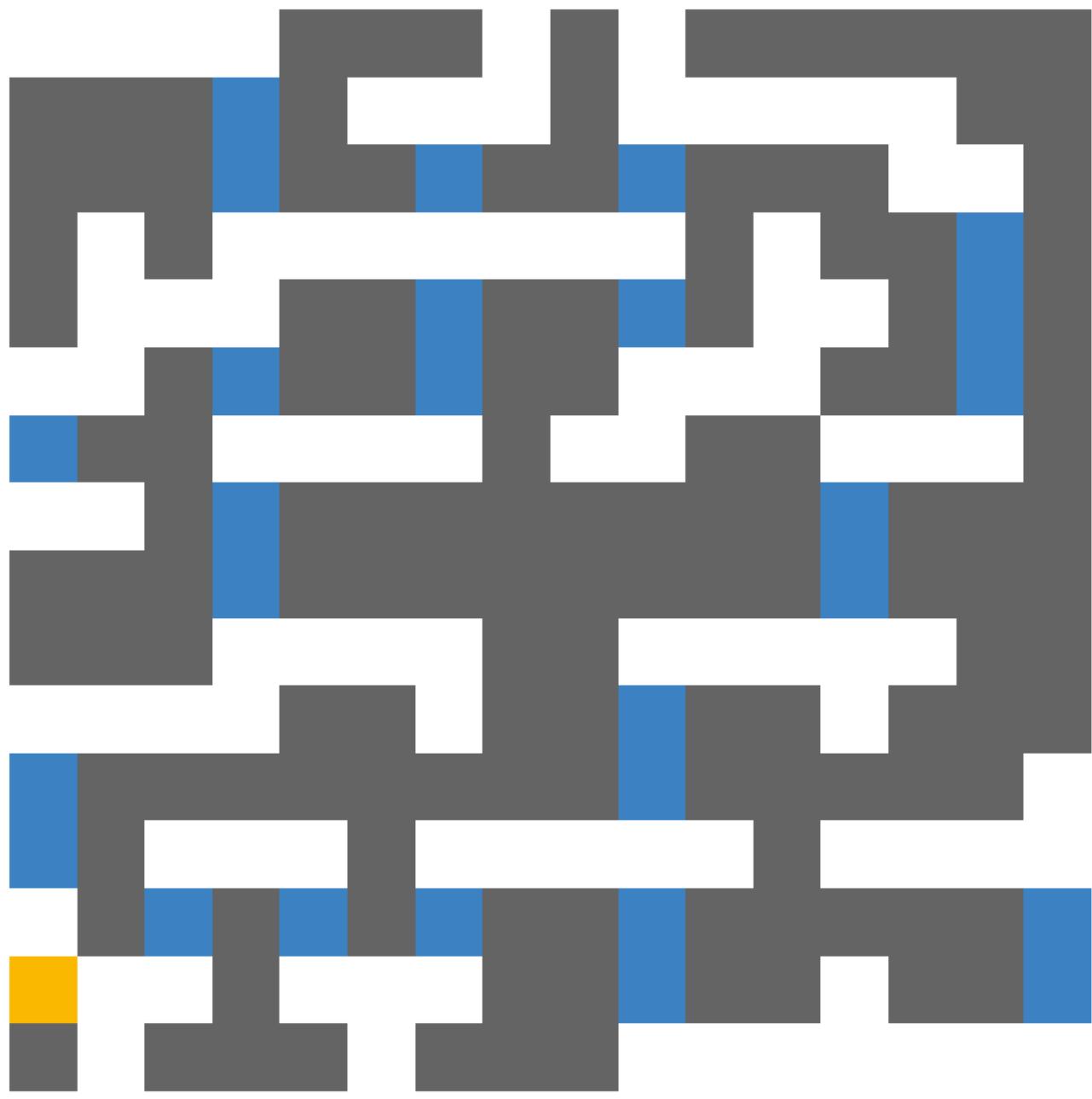
    # Find percept of new position
    return self.scan([i,j])
```

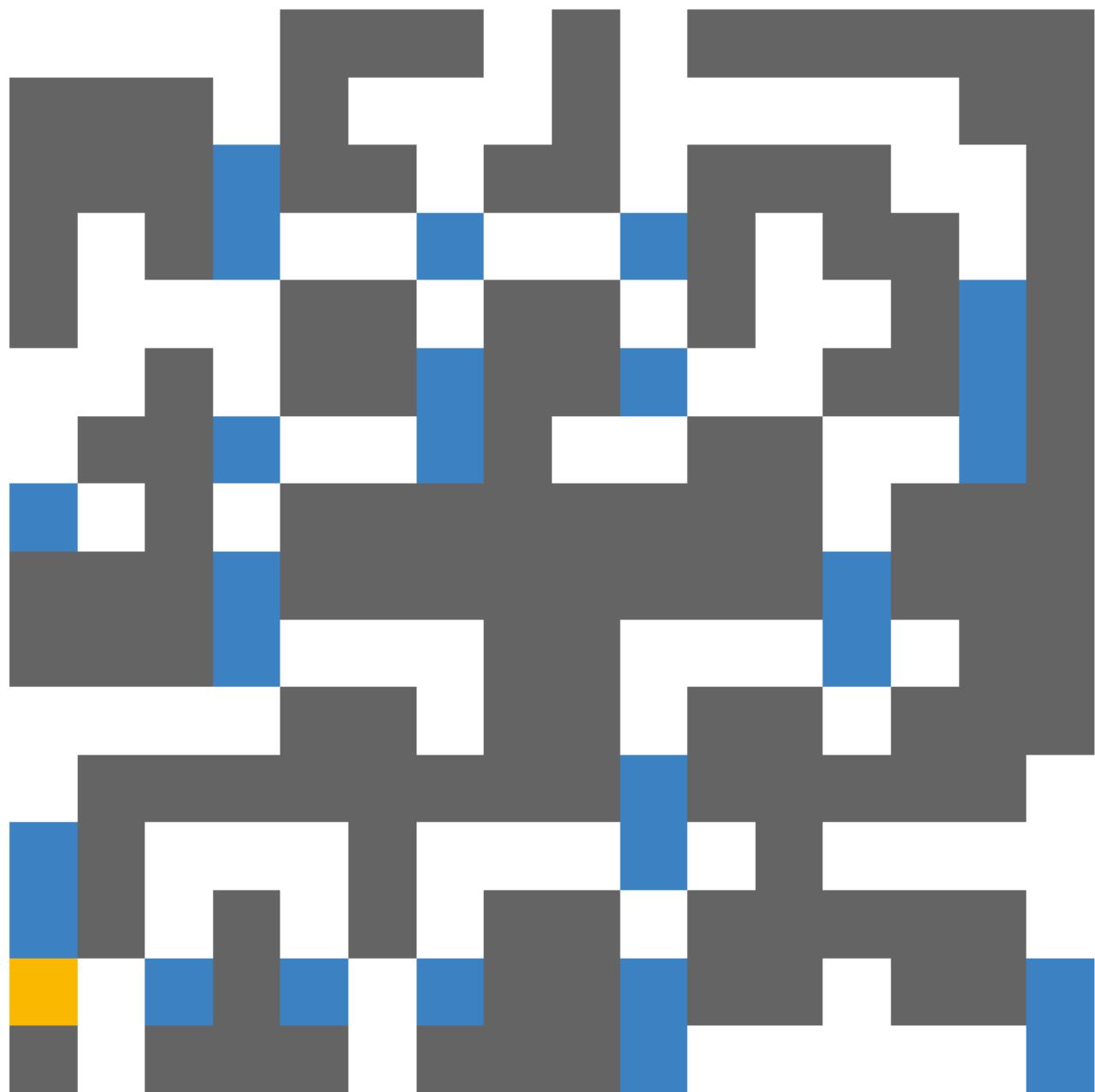
..

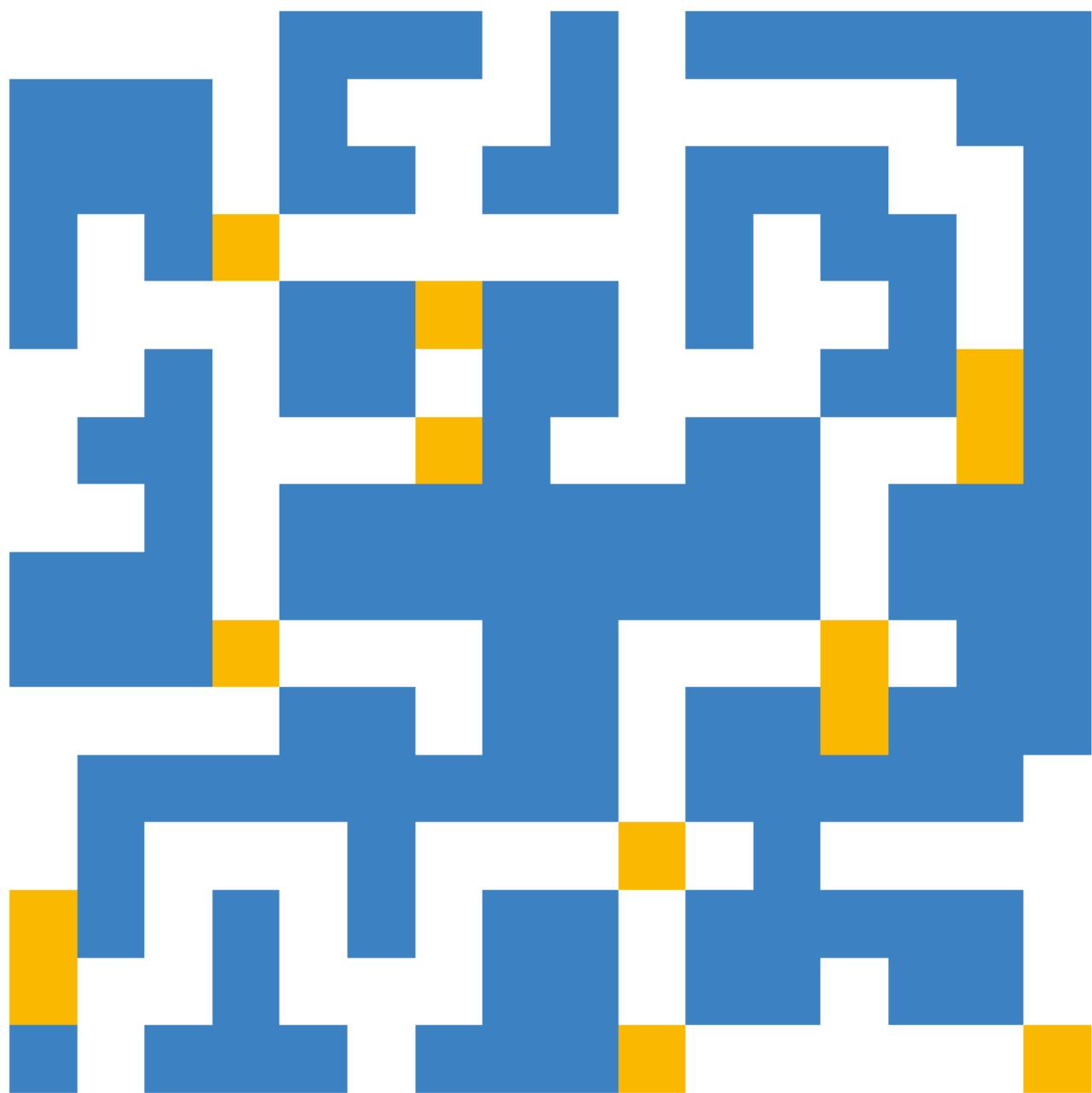
Crazy visual glitch

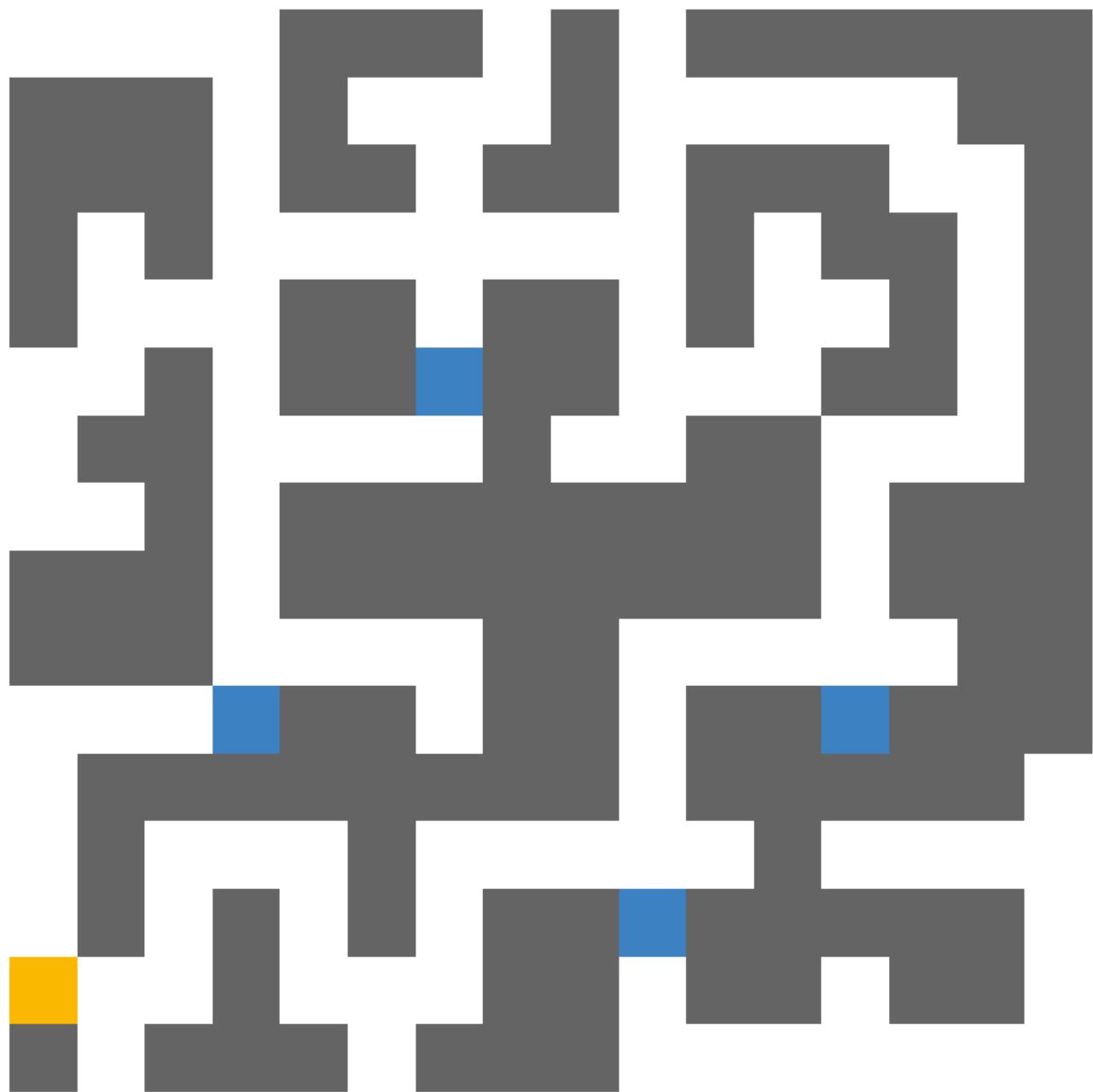
 [Glitch >](#)

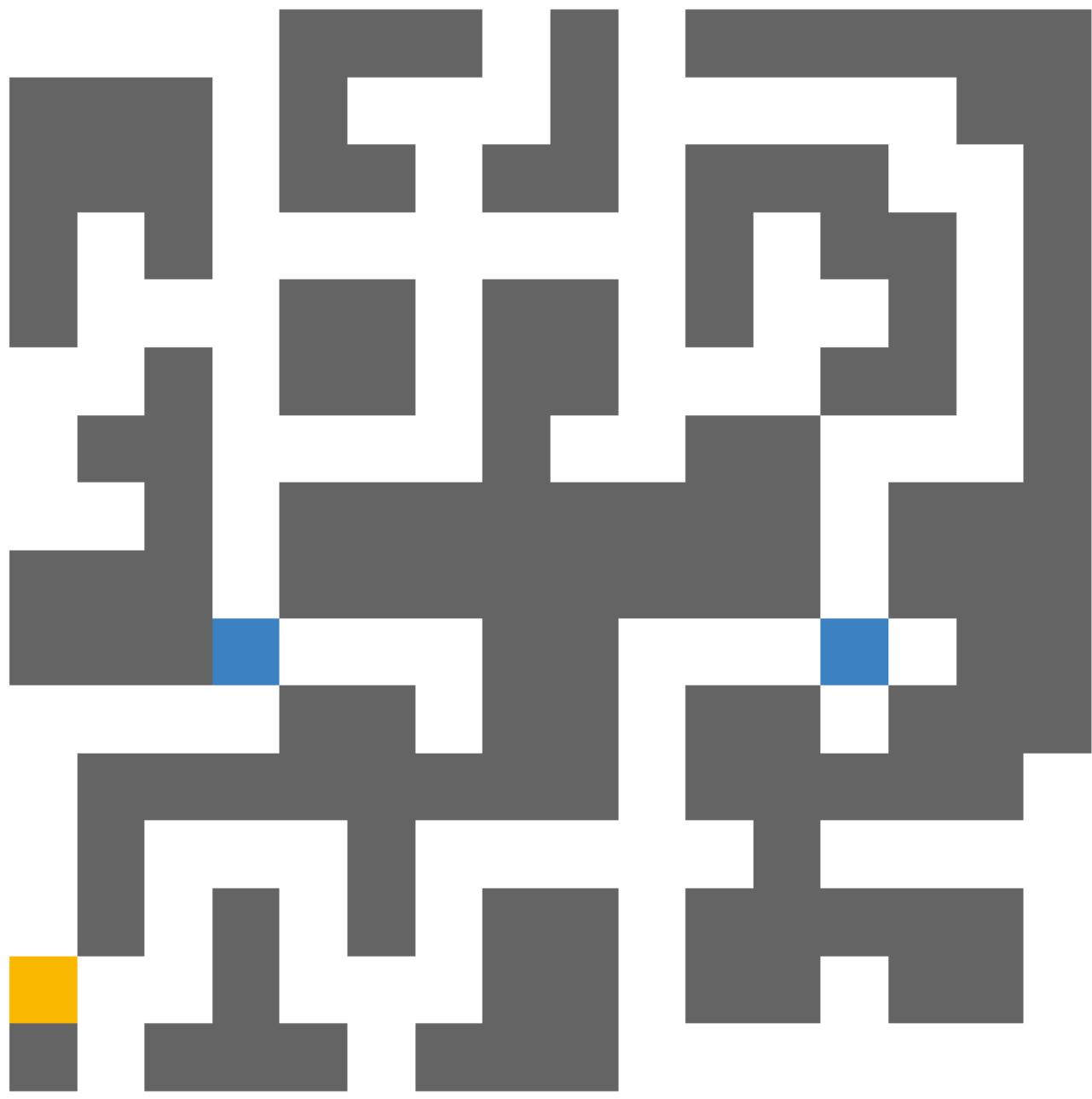


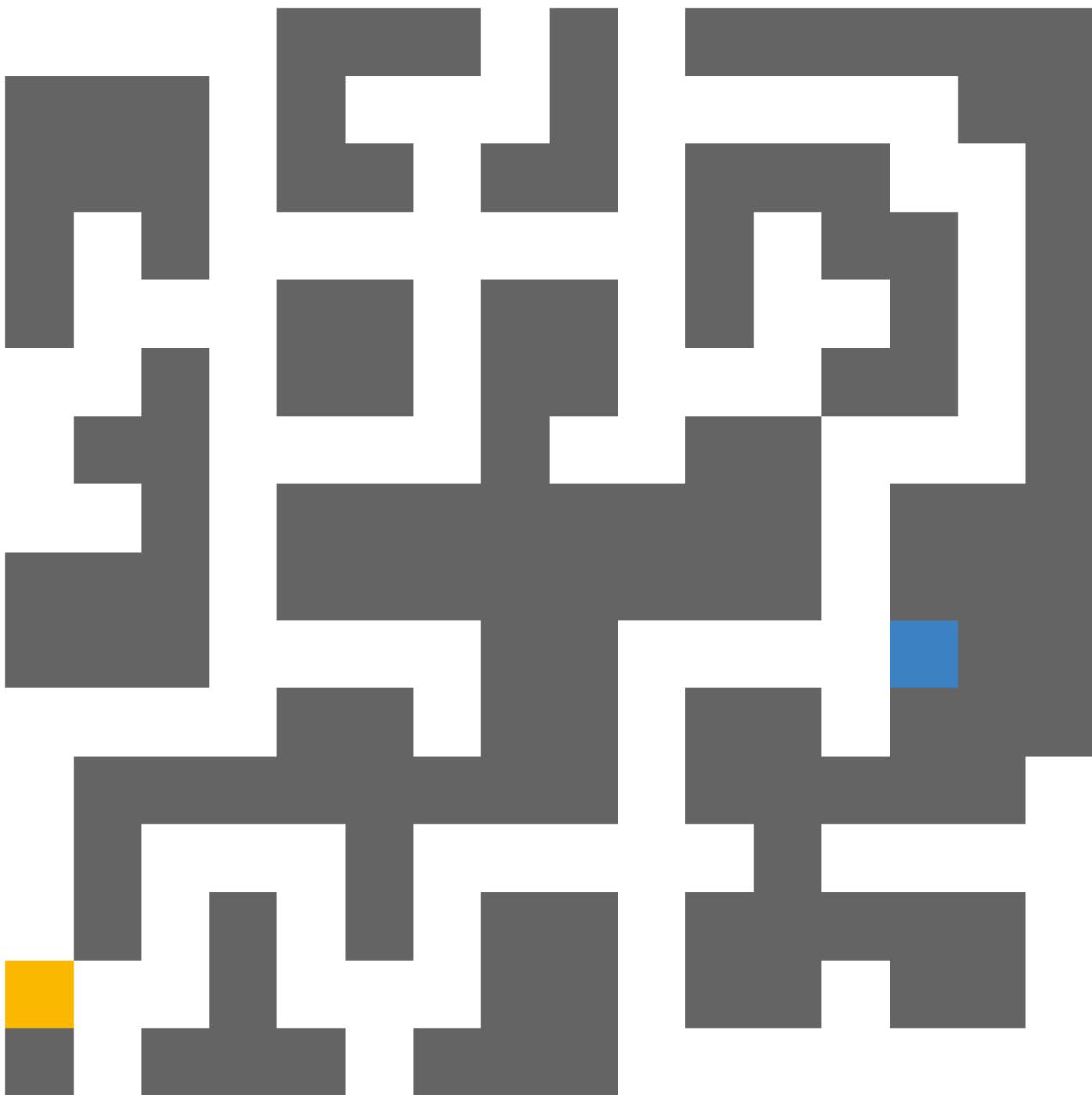












#### Problem 1:

When coding `single_out_agent()`, the real agent would keep getting deleted. This was because of these two blocks:

```
# Get percept from Environment
percept=self.enviro.percept()

# Update belief state with percept to remove old false mini-agents
self.update(percept)

# Choose a move
self.choose_action()

# Save the data
self.save_data()
```

and

```
percept=self.data["Percept"].iloc[-1]
```

```
in possible_actions()
```

The positions would be updated as `self.positions`, but the percept wouldn't because it was taking the old percept from the `self.data`. `self.data` wasn't updated yet with the new percept. So I changed the code in `possible_actions()` to

```
percept=self.enviro.percept()
```

and now the percept is the most recent.

Though after the fix, my `single_out_agent()` code does not even seem that efficient at removing the agent. Also, there's still errors where the main agent is deleted. The agent tends to go back and forth a lot which is the opposite of what I want it to do.

And for some reason, this is True which makes no sense

```
if {'S', 'W'} and {'N'}:  
    print(True)
```

This is what's allowing for the main agent to be deleted

Okay apparently

&

and

and

are two different things. The & is an Ampersand Operator.

This seems to make my code more efficient, though later I will make a new code that will check if there's an efficiency increase. Similar to what I did with Percent vs Move.

But the code is still broken since the main agent still gets deleted. So I have to fix that before moving on.

Okay there's a None in the DataFrame

This should fix it in `self.single_out_agent()`

```
# Check if the moved_percept is unique in the previous history of percept.
for i,percept in enumerate(self.data["Percept"]):
    if moved_percept not in self.data["Percept"][:i]:
        return action
    # else:
```

to

```
# Check if the moved_percept is unique in the previous history of percepts
for i,percept in enumerate(self.data["Percept"]):
    if moved_percept not in self.data["Percept"][:i]:
        return action
    else:
        pass
```

Now if it realized there isn't a move that doesn't give a new percept, it'll go onto the next position

Literally broke on the next run for the same reason and I realized `else: pass` literally changes nothing.

Added this feature and the mini-agents are still being deleted.

This is outside `for position in positions:`

```
# If it ran out of options, go a random possible direction
return random.choice(self.possible_actions(position))
```

I seemed to fix the None problem, which I thought would fix the main agent being deleted, but I guess not

Okay removing `position` should fix it

```
return random.choice(self.possible_actions())
```

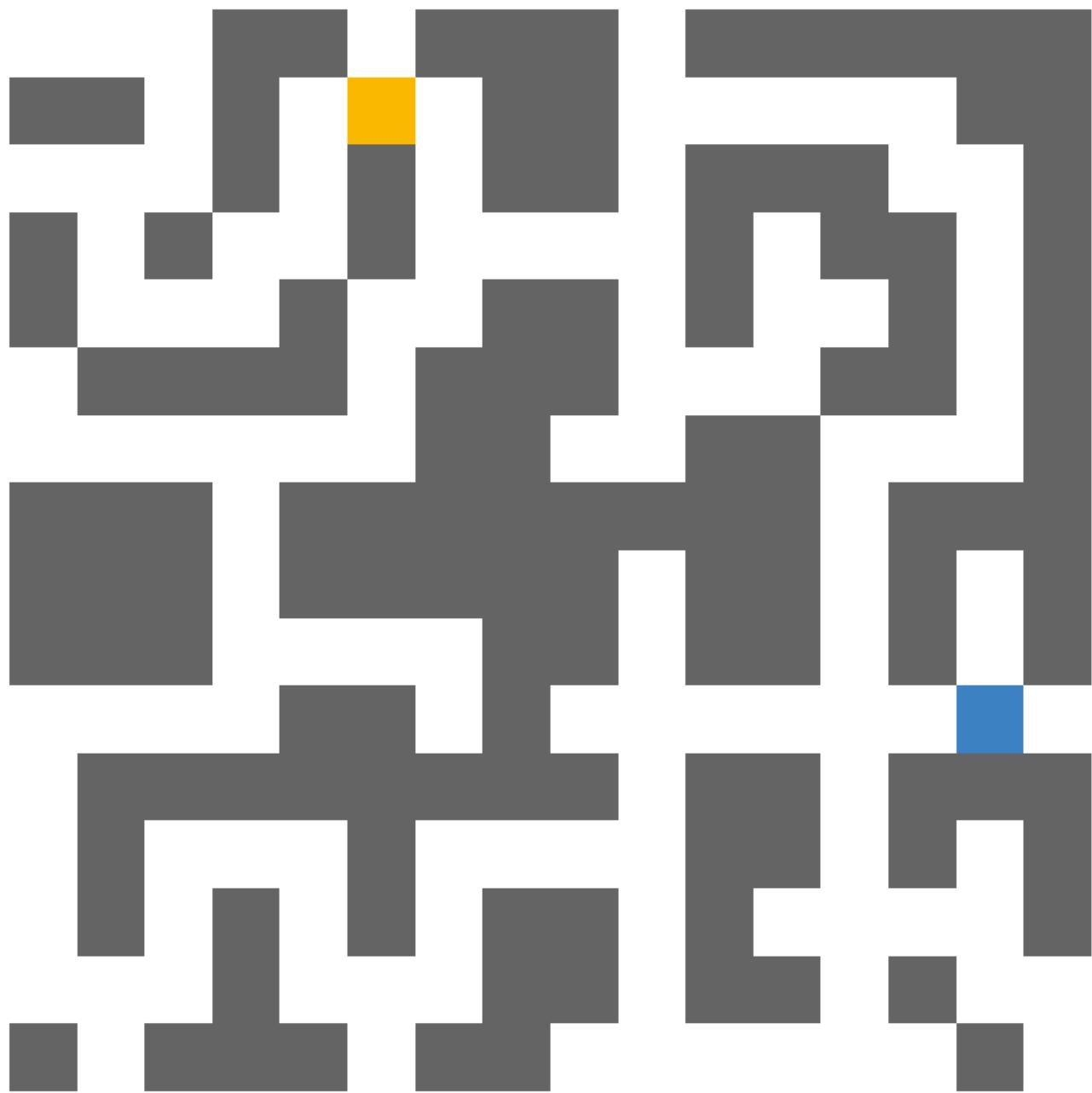
Okay I think I finally fixed the main agent being deleted. Tomorrow I would need to make a code that tests the efficiency of the code. Then I have to `self.single_out_agent()` useful by not having the agent go back and forth all the time.

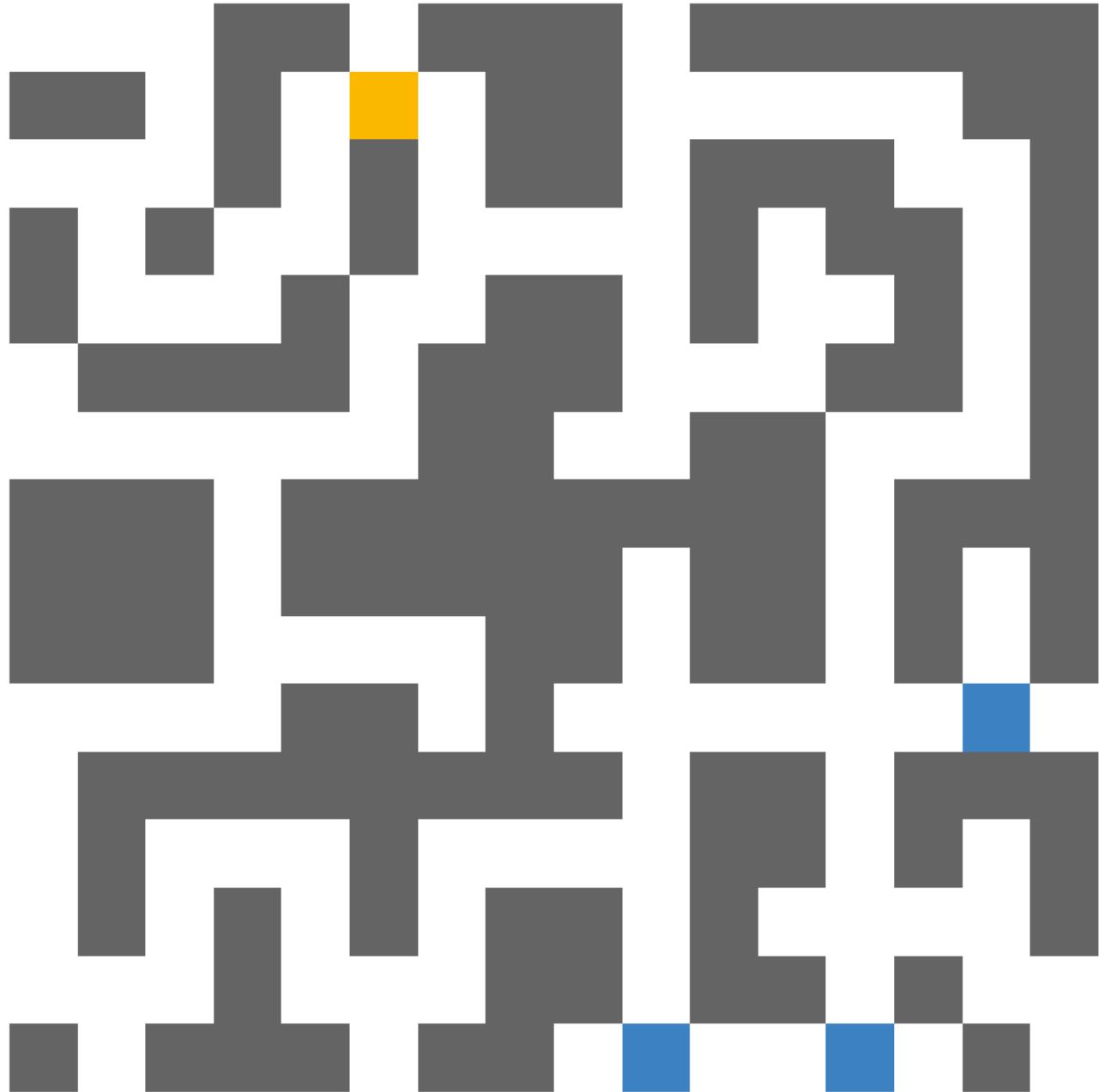
Interesting data

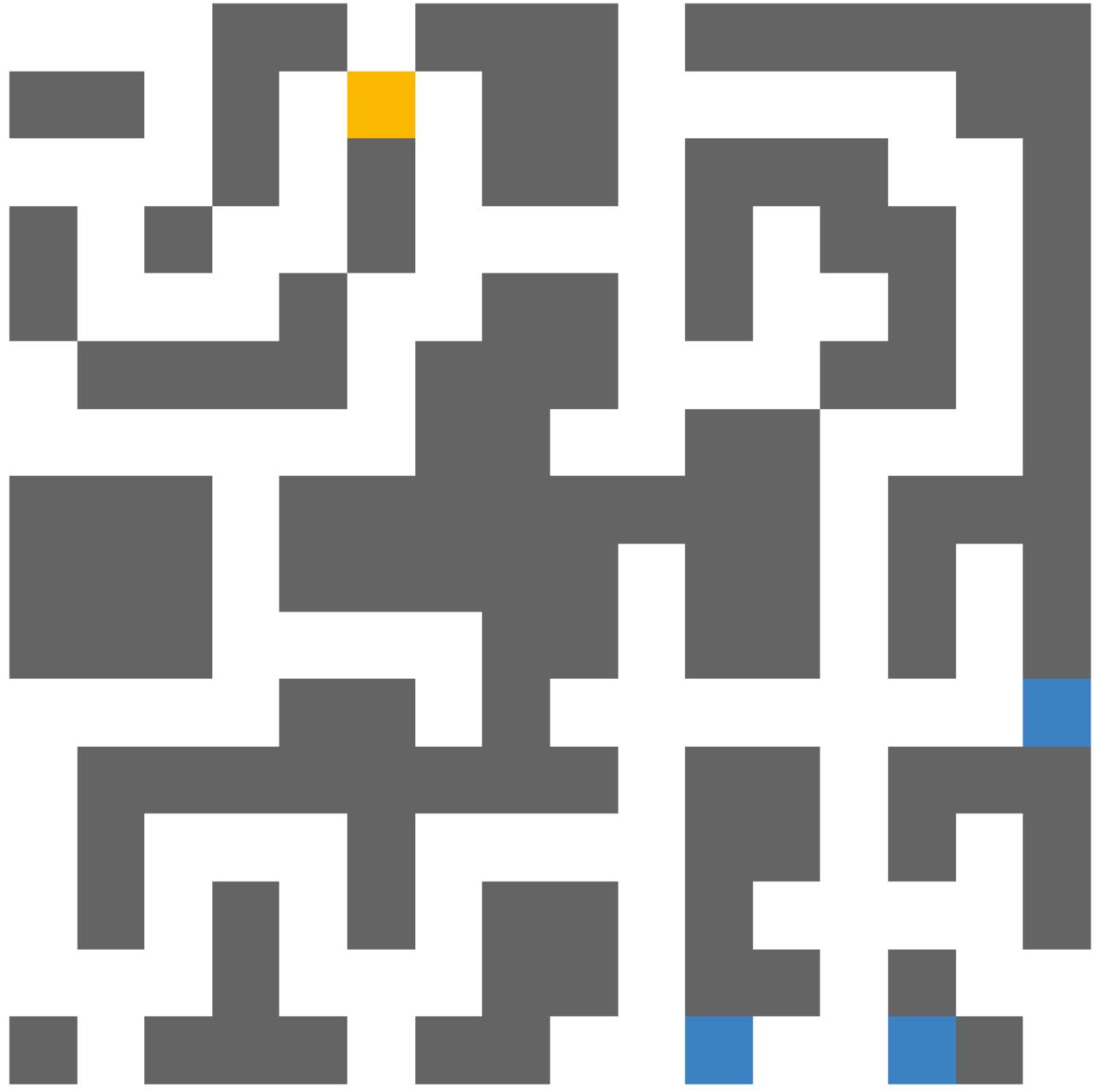
```
{'E', 'N', 'W'} {'E', 'N', 'W'}
E
{'W'} {'E', 'N', 'W'}
W
{'W'} {'E', 'N', 'W'}
W
{'E', 'N', 'W'} {'W'}
W
{'E', 'N', 'W'} {'W'}
W
{'E', 'W'} {'E', 'N', 'W'}
E
{'E', 'W'} {'E', 'N', 'W'}
E
{'E', 'N', 'W'} {'E', 'W'}
E
{'W'} {'E', 'N', 'W'}
W
{'W'} {'E', 'N', 'W'}
W
{'E', 'N', 'W'} {'W'}
W
{'E', 'W'} {'E', 'N', 'W'}
E
{'E', 'W'} {'E', 'N', 'W'}
E
{'E', 'N', 'W'} {'E', 'W'}
E
{'E', 'N', 'W'} {'E', 'W'}
```

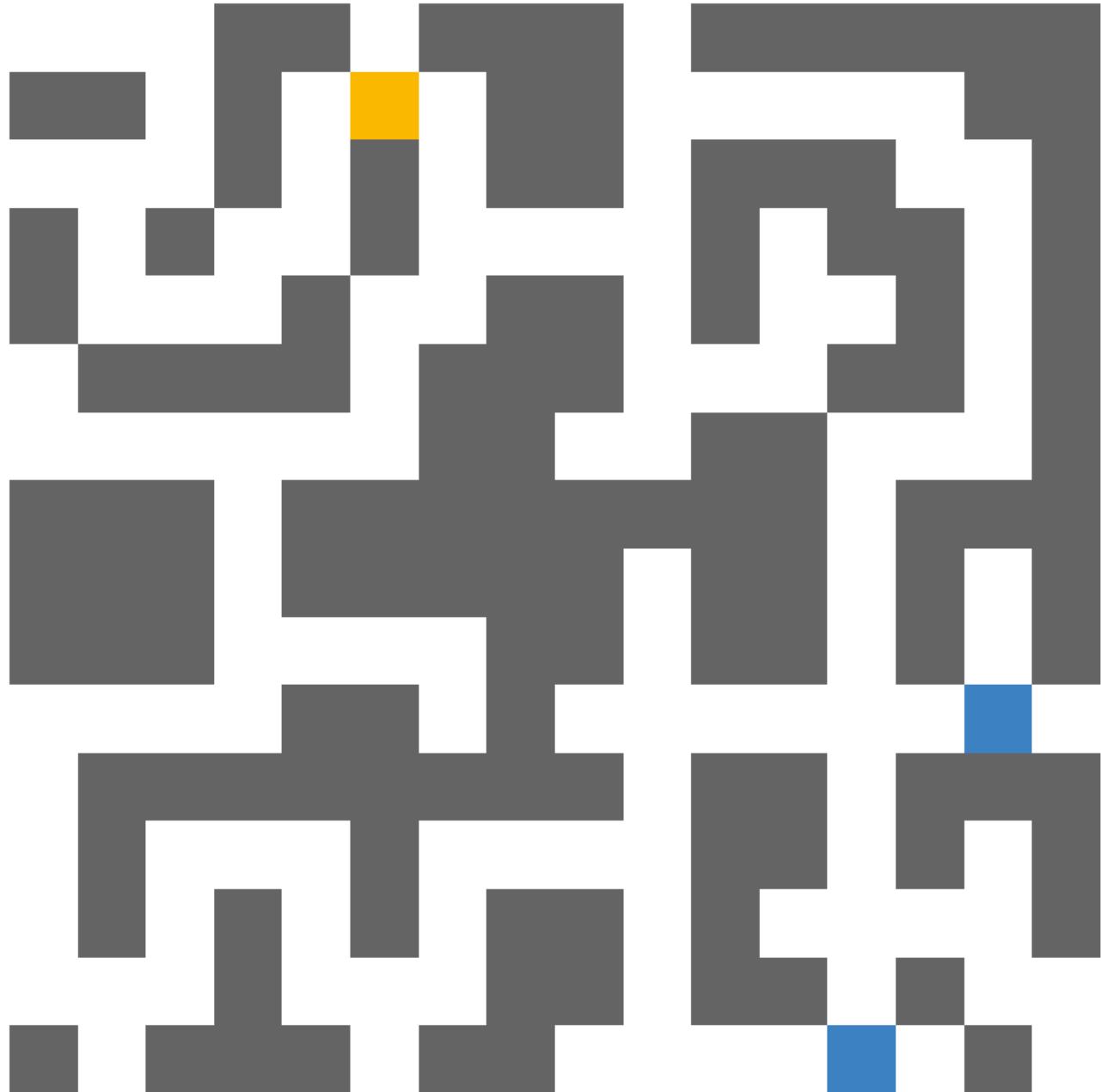
E  
{'W'} {'E', 'N', 'W'}  
W  
{'W'} {'E', 'N', 'W'}  
W

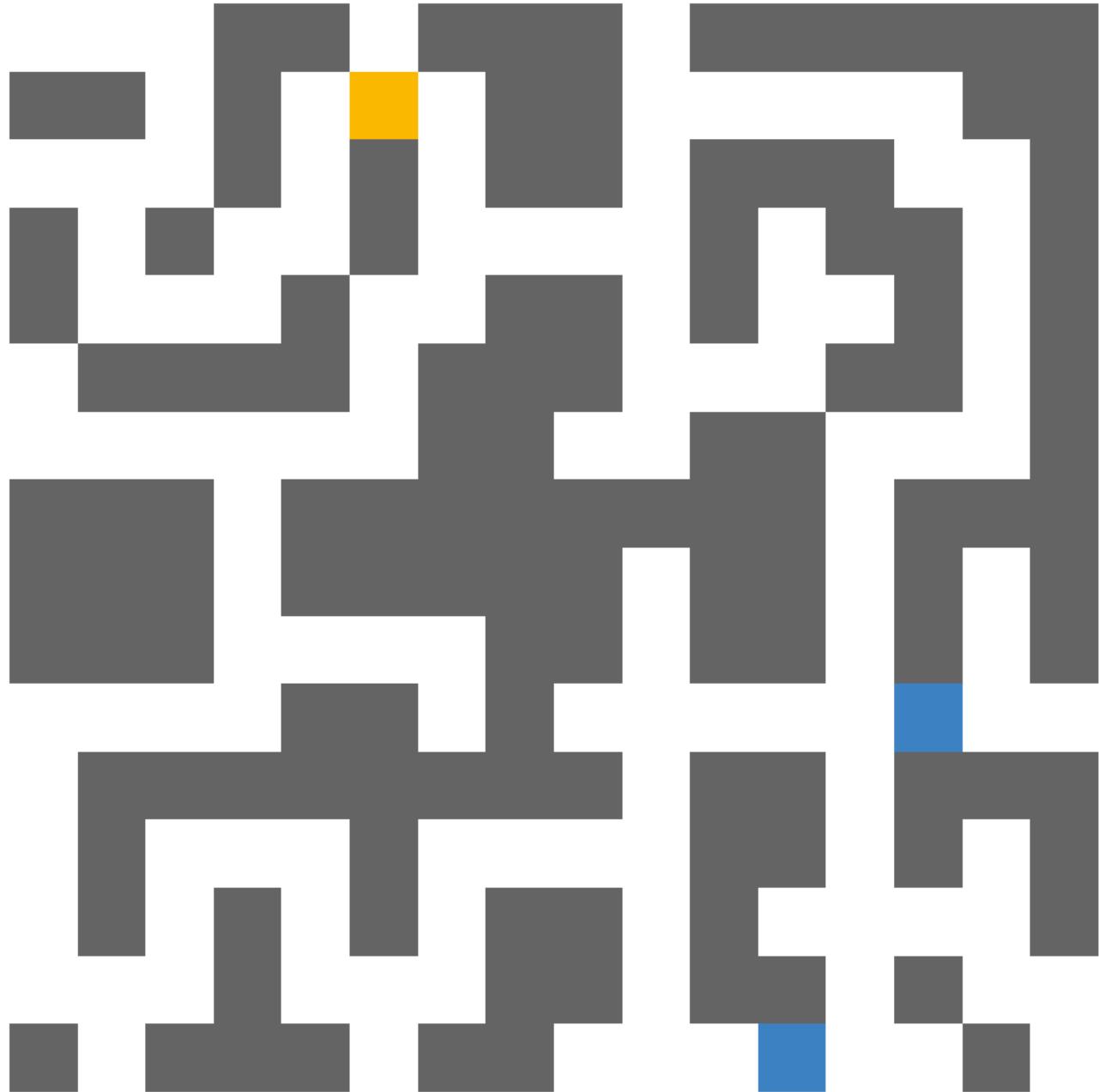
 **Looped agent** >

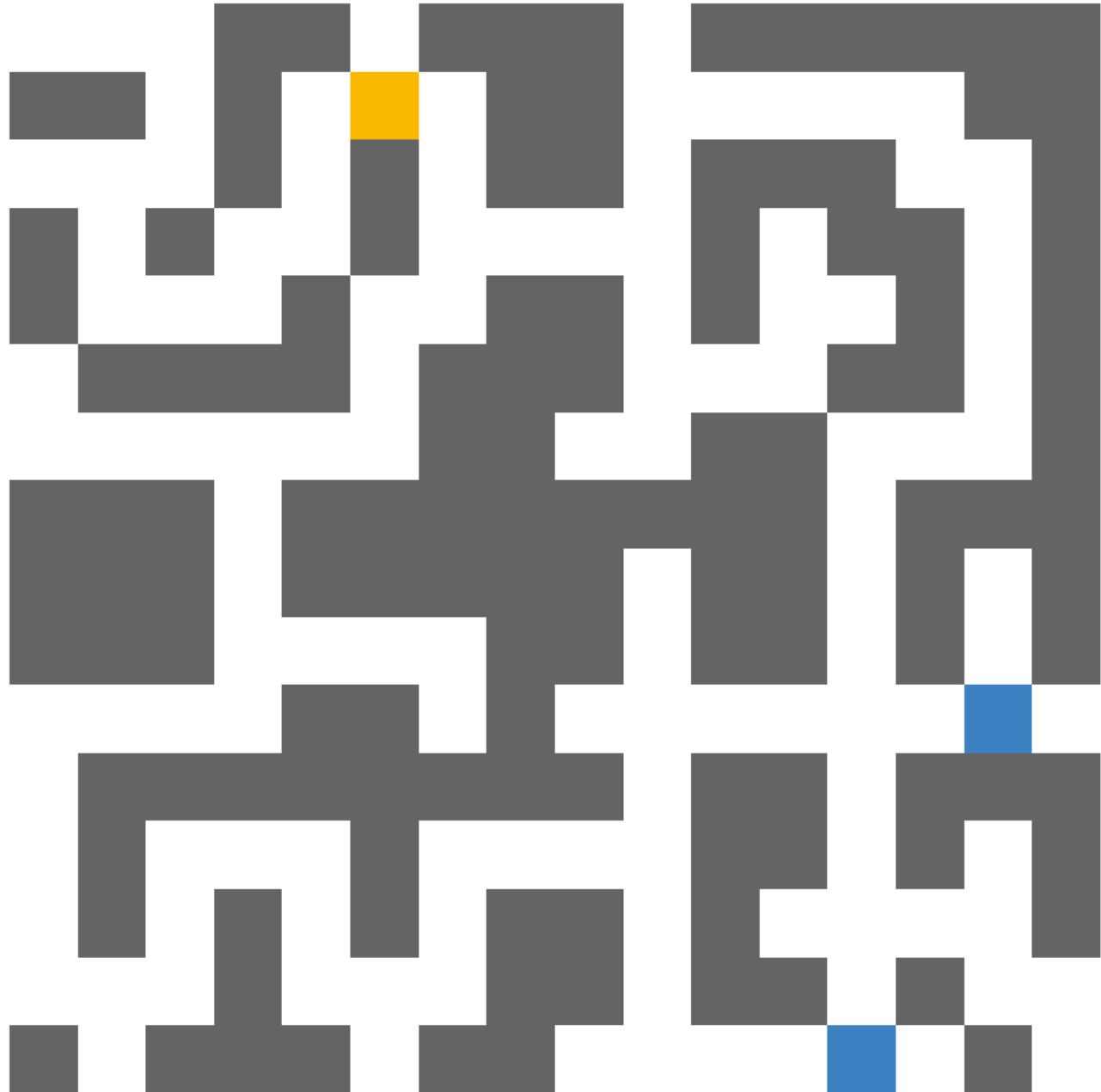


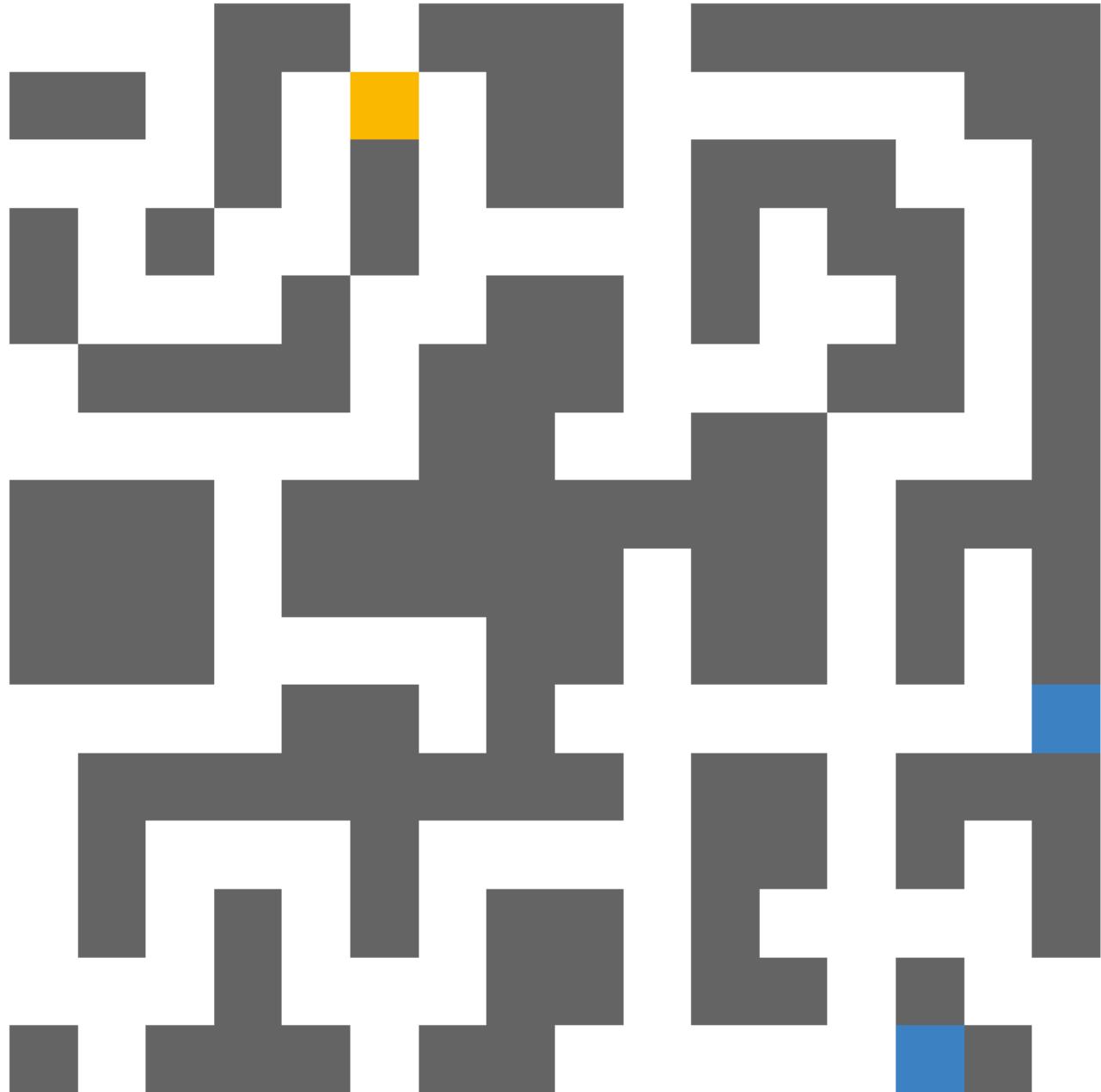


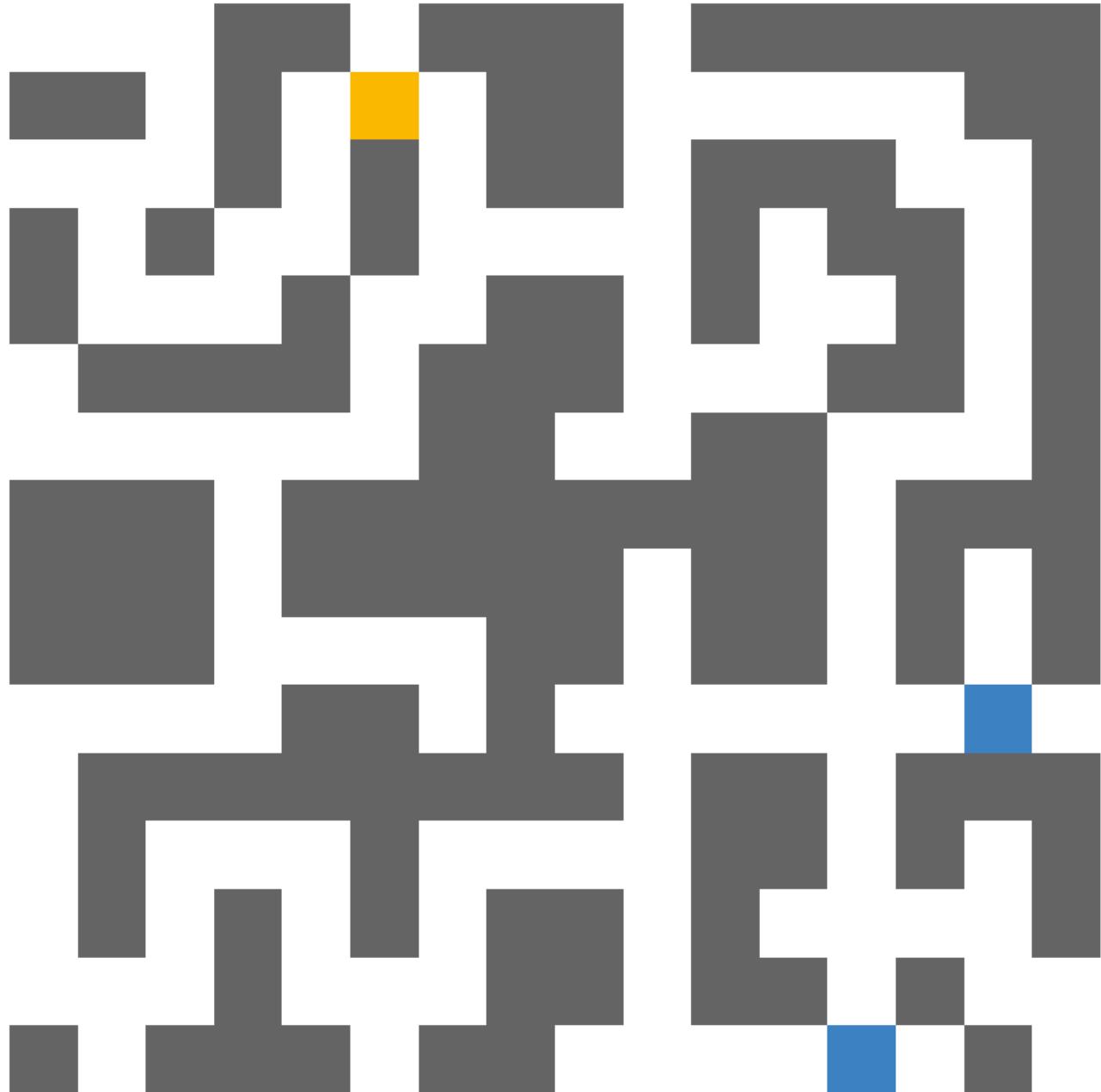




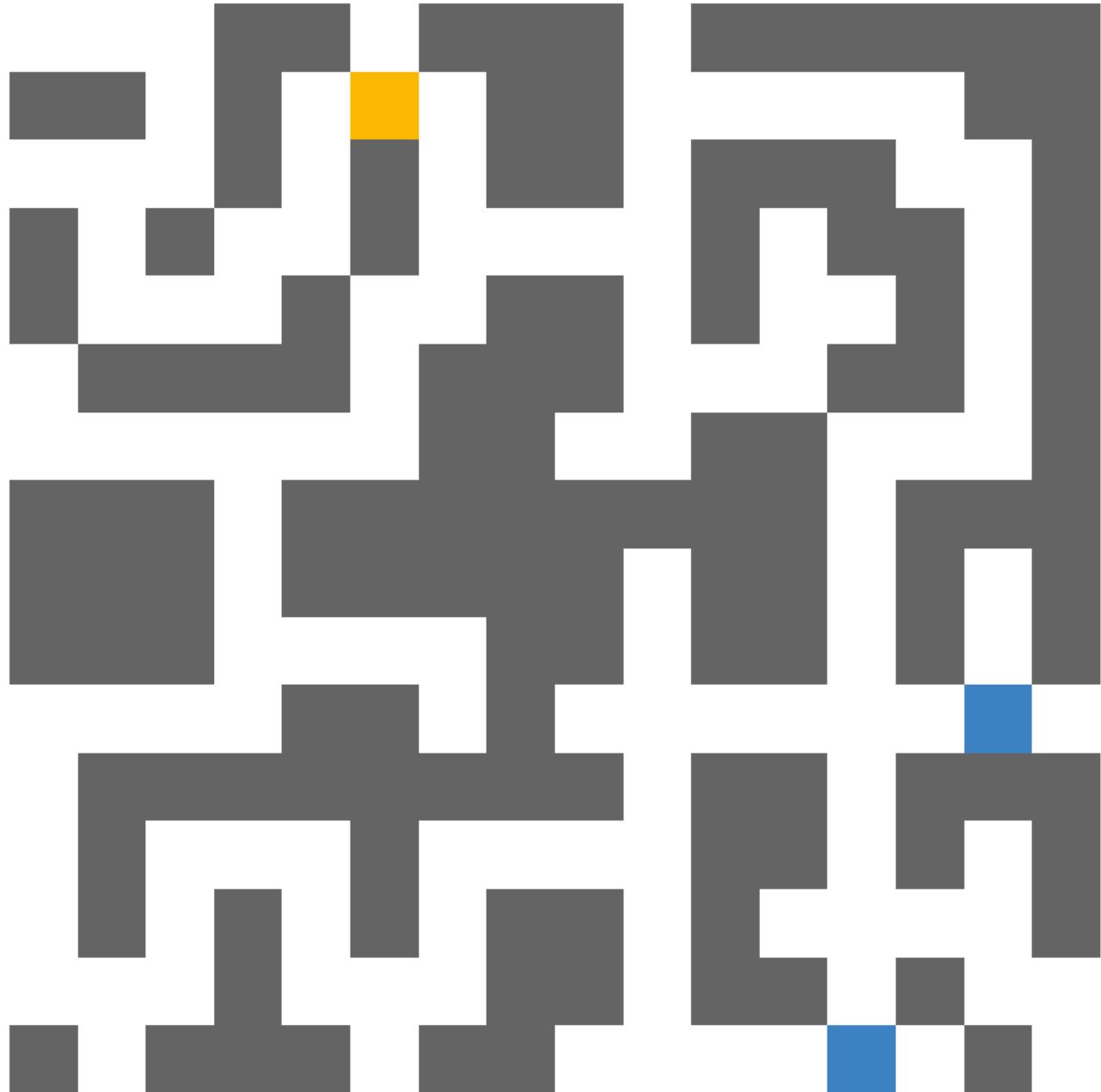


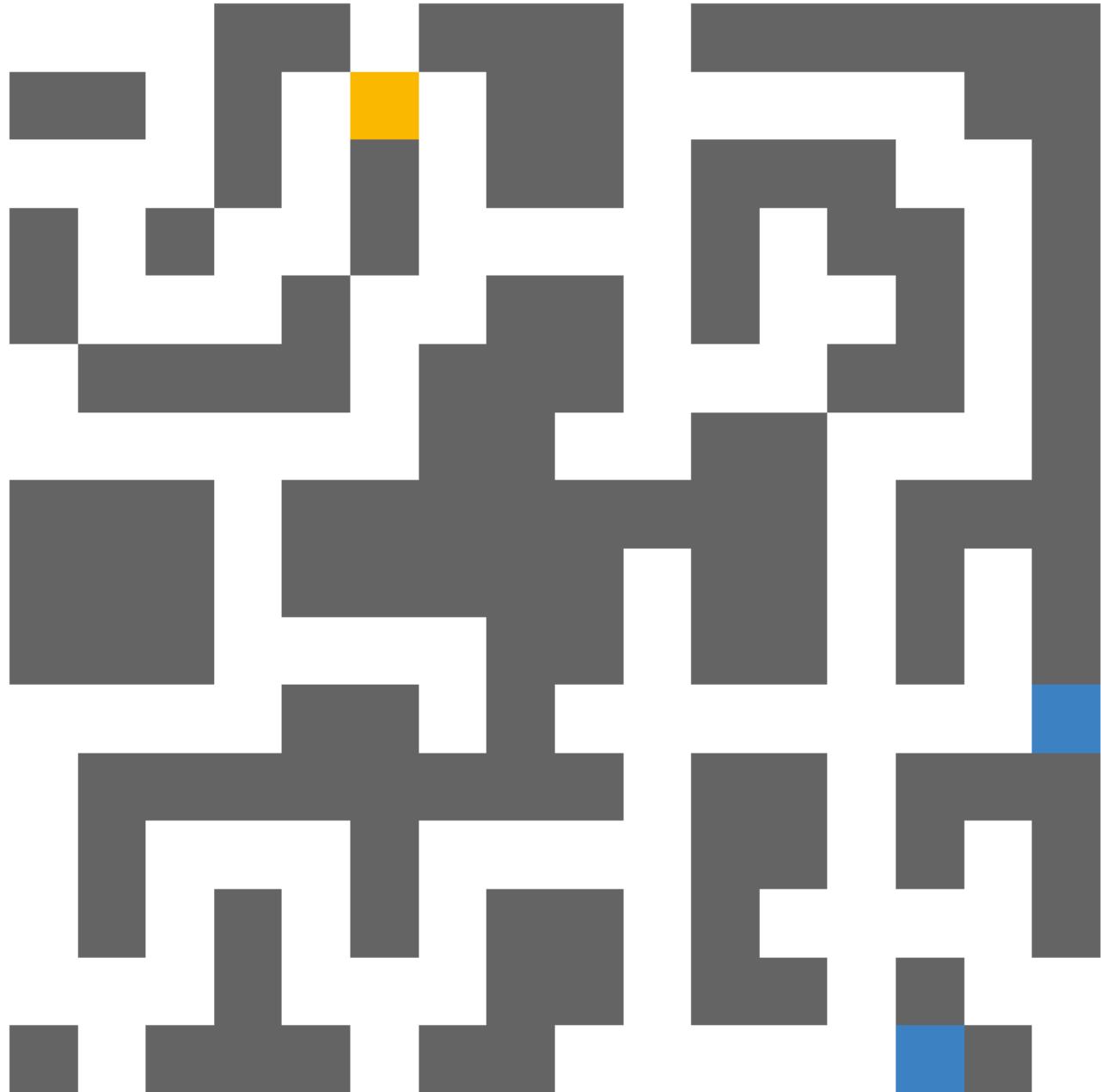


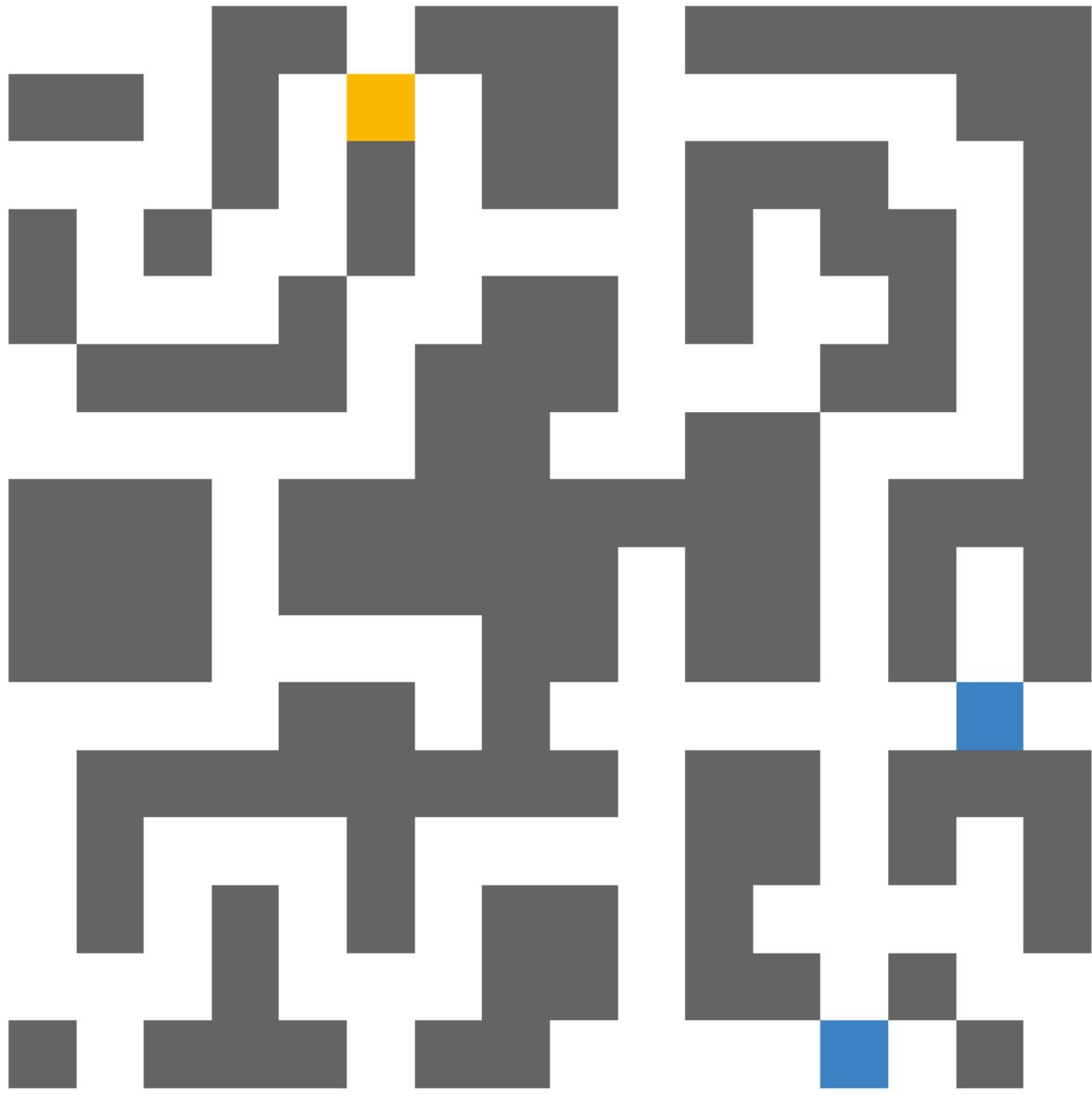












Going to make code that tests efficiency

The test efficiency test will check how many moves it took to get to a single mini-agent. That's all. Then those moves will be saved into a txt document. We then can check the average moves it takes for different algorithms

Made a new class called `Performance_Test` that's pretty simple. Inside the code it creates a new environment for the amount of iterations you give, and will save whatever data you want when the Agent is done running. Very customizable by editing the code. Not sure if I'll add parameters to the class yet

I tested the code by going random direction in the `choose_move()` section of the Agent. The average moves after 100 iterations is:  
12.84

Okay now fixing `single_out_agent()` because there's always a loop that goes for infinity for two mini-agents

Nvm it does single it out to one agent, it's just how I positioned the `sys.exit()`

Perfect now I can perform a performance test

Well it's 100% not more efficient

The average moves after 100 iterations is:

34.46

And a lot of times the reasons why the Agent stopped is because it reached its max 100 iterations of running

I think the reason why it's not more efficient is because my code is still problem and somehow the positions and percept is still not in sync. It's a tricky problem to solve

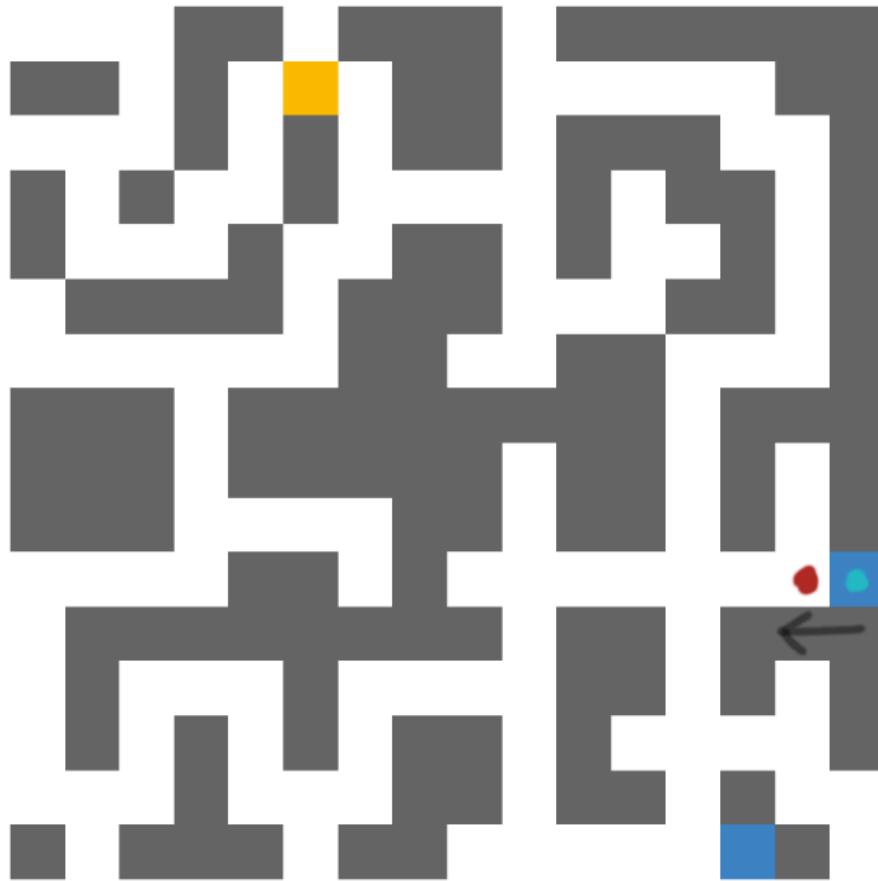
I think another part of the problem is how sets are ordered in alphabetical order. This leads to agents going back and forth. I need to make the direction it chooses random.

Here is what I suspect is the problem from this very specific example I have saved yesterday

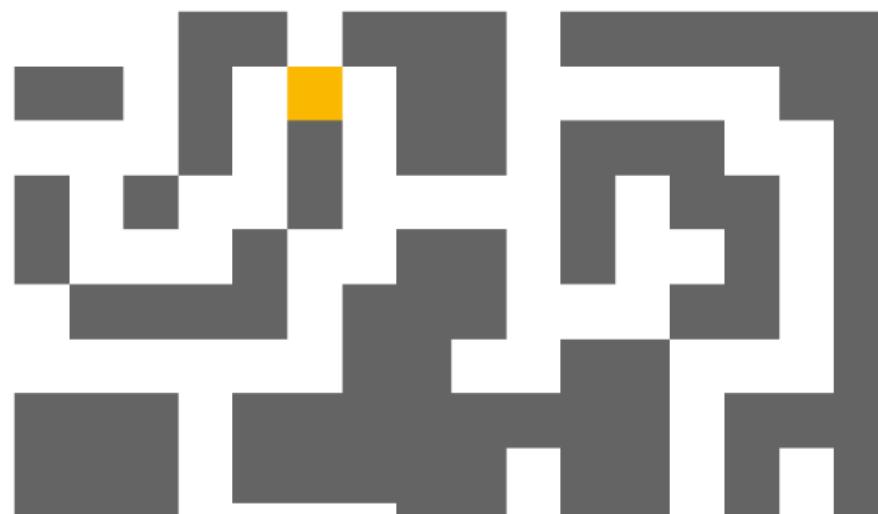
→ percept

→ Position

{'W'} {'E', 'N', 'W'}  
W



{'E', 'N', 'W'} {'W'}



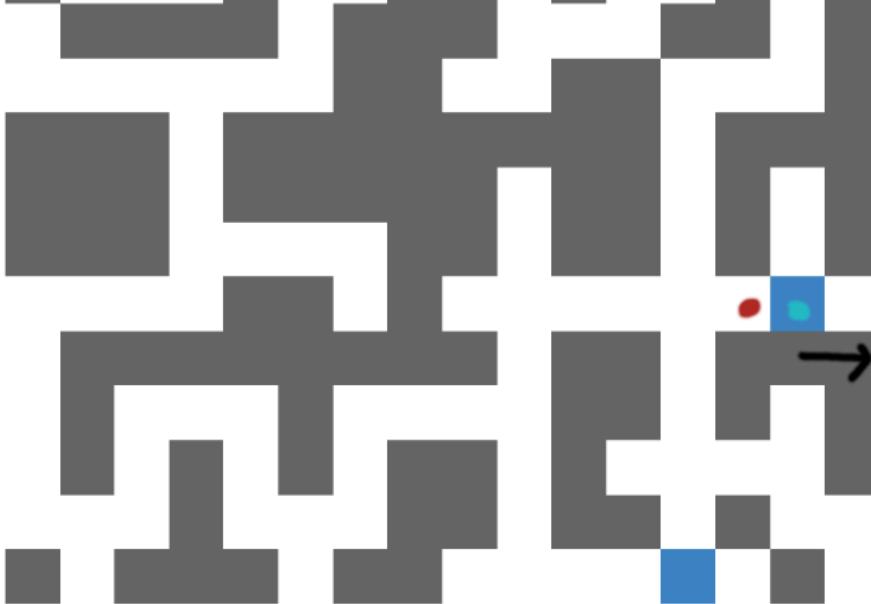


{'E', 'W'}   {'E', 'N', 'W'}  
E



{'E', 'N', 'W'}   {'E', 'W'}  
E





The sets we are seeing are the possible directions of a mini-agent (red), and the real percept (blue). By chance, this scenario has a mini-agent and real agent the same. This means that the mini-agent's position should be exactly the same as the real agent's position. But we can see the sets are different.

We can use the sets to understand how the code sees the state. We can see that the mini-agent's data is always one step behind the agent's data. The positions scanned and percept are real out of sync.

Position is one step behind percept

A reason for this error is probably how the percept is very recent being collected from the Environment class, as the positions are from the DataFrame

I'll try to get positions from `self.positions` and do a benchmark to see if anything improved

```
positions=self.data["Positions"].iloc[-1]
```

to

```
positions=self.positions
```

HOLY, that's a huge improvement!

The average moves after 100 iterations is:

16.09

Sadly, this is still slower than using random direction. It's average was 12.84

I'll call the current algorithm I'm working on to single out the real agent "Unique Percept algorithm" since it's trying to always find unique percepts. The algorithm taking random moves will be called "Random Algorithm"

So far, my UP algorithm () is less efficient than going random () directions. That's not good

I know there's still a critical error in my code since 100 moves were recorded multiple times leading to a higher average. These were the moves recorded:

```
[100, 6, 4, 7, 100, 8, 4, 8, 9, 9, 100, 8, 7, 5, 100, 2, 9, 100, 5, 3, 3, 3, 6, 6, 6, 9, 8, 6, 4, 100, 5, 8, 9, 9, 4, 8, 8, 4, 6, 3, 7, 6, 8, 4, 100, 4, 4, 7, 1, 7, 9, 7, 3, 2, 4, 3, 5, 6, 2, 5, 5, 6, 5, 2, 1, 8, 4, 7, 6, 100, 8, 4, 5, 5, 5, 8, 9, 8, 5, 8, 6, 11, 100, 4, 5, 9, 7, 100, 3, 3, 4, 2, 6, 100, 4, 11, 4, 7, 2, 9]
```

Omg I found a loop that breaks the code and it's the exact same scenario that I analyzed before



What's cool about getting the exact same scenario is that you can see the sets are now the same after syncing the position and percept

```
{'E', 'W', 'N'} {'E', 'W', 'N'}
E
{'E', 'W', 'N'} {'E', 'W', 'N'}
W
{'W'} {'W'}
W
{'W'} {'W'}
E
{'E', 'W', 'N'} {'E', 'W', 'N'}
E
{'E', 'W', 'N'} {'E', 'W', 'N'}
W
{'W'} {'W'}
W
{'W'} {'W'}
E
{'E', 'W', 'N'} {'E', 'W', 'N'}
E
{'E', 'W', 'N'} {'E', 'W', 'N'}
W
{'W'} {'W'}
W
{'W'} {'W'}
E
{'E', 'W', 'N'} {'E', 'W', 'N'}
E
{'E', 'W', 'N'} {'E', 'W', 'N'}
W
{'W'} {'W'}
W
{'W'} {'W'}
E
{'E', 'W', 'N'} {'E', 'W', 'N'}
E
{'E', 'W', 'N'} {'E', 'W', 'N'}
```

So we know that problem is fixed for sure!

And we can clearly see the problem is some sort of loop going on making the AI going back and forth. This shouldn't even happen because the AI should always want to go to a different percept. Clearly that part of the code isn't working

It seems there's a loop when the position and percept sets are the same

Okay the code seems very promising. I think I fixed the problem. The code before was this mess, where I overcomplicated with my finding unique percept

thing

```
for i,percept in enumerate(self.data["Percept"]):
    if moved_percept not in self.data["Percept"][:i]:
        print(possible_actions_percept,possible_actions_position)
        print(moved_percept,self.data["Percept"][:i])
    return action
```

and I replaced it with this where it simply checks if the `moved_percept` is not in the percept history.

```
if moved_percept not in set(self.data["Percept"]):
    # if moved_percept in percept:
    print(possible_actions_percept,possible_actions_position)
    print(moved_percept,set(self.data["Percept"]))
    print(action)
    return action
```

Another potential error could haave been how

```
if moved_percept not in self.data["Percept"]
```

always returns true. As soon as I added the `set`

```
set(self.data["Percept"])
```

it worked. Also sets are nice because you can print it and it does not show duplicates. Love the invention of set theory.

If it cannot find a future move that does not give a percept that wasn't already discovered  
(here is an example of new percept being discovered):

```
{'W', 'N'} {'W', 'N'}
1110 {'1001', '1000'}
N
```

```
if moved_percept not in set(self.data["Percept"]):
    # if moved_percept in percept:
    print(possible_actions_percept,possible_actions_position)
    print(moved_percept,set(self.data["Percept"]))
    print(action)
    return action
```

then there a random possible direction will be chosen

```
{'S'} {'S'}
No possible option ['S']
S

print(possible_actions_percept,possible_actions_position)
print("No possible option",self.possible_actions())
print(action)
return random.choice(self.possible_actions())
```

Okay, about to run the benchmark, wish me luck Alexander or Manuel whoever is reading this. Maybe aliens from 2100, calling it now.

DAMNN IT'S FAST

The average moves after 100 iterations is:

5.07

UP got an upgrade!

Let's check out efficiency compared to the random algorithm

60% faster than going randomly!! That's how you know something intelligent is choosing the moves, aka AI. That's a really awesome algorithm. I didn't expect it to work so well.

For fun let's see how much faster it was than my broken version

Okay now as much faster as I thought I think I did the math wrong.

Never mind I didn't, I'm just thinking of ratio in my head since can fit more than twice in .

So for fun let's see how many times faster the upgraded UP is

and when compared to the Random algorithm it is

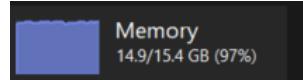
For fun I benchmarked 1000 iterations to have a precise value of how many moves it takes to get to a single agent. And it turns out it takes an average of almost exactly 5 moves! That's really good considering it knows only the percept and can find the real agent that fast

The average moves after 1000 iterations is:

5.063

Moves [5, 4, 4, 5, 8, 2, 4, 2, 3, 2, 4, 5, 6, 10, 5, 5, 9, 2, 6, 4, 6, 8, 4, 10, 5, 4, 6, 2, 8, 4, 3, 2, 2, 4, 6, 3, 8, 6, 10, 7, 7, 5, 6, 6, 7, 5, 6, 8, 6, 9, 5, 4, 5, 4, 7, 6, 3, 5, 7, 4, 7, 9, 4, 8, 4, 7, 10, 4, 4, 4, 3, 6, 5, 8, 4, 3, 6, 5, 4, 3, 4, 3, 5, 5, 5, 4, 5, 4, 6, 5, 5, 3, 4, 4, 4, 5, 3, 6, 1, 6, 6, 5, 4, 6, 4, 7, 4, 7, 5, 2, 5, 2, 8, 4, 4, 4, 3, 5, 2, 10, 4, 3, 5, 4, 7, 6, 10, 5, 6, 6, 7, 9, 6, 3, 3, 7, 5, 4, 9, 8, 3, 6, 5, 3, 5, 6, 4, 4, 5, 3, 5, 5, 4, 8, 3, 10, 5, 3, 2, 5, 3, 4, 4, 6, 4, 5, 9, 4, 8, 7, 7, 5, 4, 5, 3, 3, 7, 7, 8, 5, 5, 3, 4, 7, 3, 8, 4, 3, 4, 12, 6, 8, 8, 8, 6, 5, 3, 8, 6, 4, 3, 9, 6, 4, 5, 8, 2, 5, 6, 7, 6, 5, 3, 10, 4, 3, 8, 6, 6, 5, 3, 2, 4, 5, 4, 3, 4, 10, 3, 4, 4, 9, 4, 4, 7, 7, 5, 5, 3, 3, 12, 8, 2, 3, 4, 2, 6, 1, 5, 5, 8, 5, 5, 4, 8, 2, 5, 1, 6, 7, 6, 4, 2, 3, 21, 9, 3, 6, 5, 3, 5, 4, 6, 2, 5, 4, 2, 3, 4, 4, 3, 2, 8, 3, 4, 5, 8, 5, 5, 8, 5, 6, 7, 2, 4, 5, 3, 2, 3, 9, 5, 4, 11, 5, 6, 2, 3, 4, 10, 6, 5, 4, 7, 6, 4, 8, 4, 4, 6, 5, 3, 4, 6, 7, 5, 2, 7, 2, 5, 4, 1, 5, 7, 3, 6, 5, 4, 5, 6, 2, 6, 8, 4, 4, 2, 6, 6, 8, 5, 4, 9, 3, 10, 7, 7, 2, 4, 6, 4, 11, 4, 6, 5, 7, 5, 5, 4, 3, 4, 4, 6, 7, 6, 8, 2, 7, 3, 6, 5, 4, 6, 7, 11, 3, 7, 7, 8, 3, 5, 4, 1, 3, 5, 4, 6, 10, 3, 2, 4, 8, 10, 2, 3, 4, 5, 4, 4, 6, 6, 5, 4, 5, 2, 7, 4, 4, 6, 4, 6, 4, 7, 4, 3, 8, 6, 4, 5, 6, 2, 5, 3, 4, 4, 5, 6, 5, 4, 3, 4, 4, 10, 6, 4, 8, 7, 4, 4, 6, 4, 8, 5, 5, 5, 4, 3, 3, 4, 5, 4, 4, 10, 5, 4, 3, 9, 5, 3, 3, 6, 5, 7, 2, 3, 9, 4, 8, 4, 6, 6, 3, 5, 6, 9, 3, 4, 6, 5, 7, 5, 5, 2, 3, 3, 5, 5, 5, 8, 4, 3, 3, 5, 5, 5, 4, 5, 5, 4, 2, 7, 6, 4, 3, 3, 4, 2, 4, 6, 4, 7, 5, 4, 3, 7, 4, 3, 3, 3, 10, 6, 8, 4, 3, 6, 3, 9, 4, 6, 5, 8, 5, 5, 7, 4, 2, 2, 7, 3, 4, 8, 8, 4, 4, 6, 6, 6, 4, 7, 5, 3, 4, 5, 6, 4, 4, 4, 6, 2, 8, 7, 2, 7, 3, 3, 7, 3, 1, 4, 5, 7, 5, 6, 5, 3, 5, 4, 7, 5, 5, 6, 8, 1, 6, 4, 7, 4, 4, 6, 5, 4, 3, 5, 8, 4, 4, 4, 3, 6, 7, 3, 2, 4, 4, 5, 4, 8, 2, 5, 6, 3, 4, 7, 5, 4, 6, 3, 7, 3, 4, 3, 5, 8, 3, 2, 4, 7, 5, 4, 4, 14, 5, 5, 5, 12, 4, 4, 7, 4, 4, 7, 5, 1, 5, 1, 6, 5, 7, 2, 6, 4, 8, 3, 4, 4, 8, 2, 4, 4, 11, 3, 5, 3, 4, 6, 4, 3, 9, 2, 4, 10, 6, 7, 5, 6, 4, 4, 5, 5, 6, 3, 4, 5, 8, 4, 5, 6, 5, 8, 9, 4, 1, 6, 5, 3, 4, 10, 3, 4, 6, 6, 5, 3, 8, 6, 5, 5, 3, 7, 4, 2, 4, 4, 4, 5, 3, 4, 7, 3, 5, 10, 6, 4, 4, 3, 7, 5, 3, 5, 8, 6, 5, 2, 10, 6, 6, 5, 6, 4, 4, 4, 3, 7, 4, 5, 4, 3, 4, 5, 5, 9, 5, 4, 7, 6, 2, 6, 4, 6, 5, 5, 4, 4, 5, 8, 5, 5, 3, 5, 5, 4, 7, 10, 8, 5, 5, 9, 5, 4, 3, 7, 5, 7, 4, 2, 4, 11, 6, 5, 3, 4, 4, 2, 3, 5, 6, 5, 2, 5, 6, 4, 3, 4, 9, 5, 5, 3, 4, 4, 3, 5, 8, 2, 6, 4, 3, 5, 3, 4, 1, 8, 4, 7, 7, 4, 2, 2, 8, 7, 3, 8, 6, 5, 4, 3, 3, 2, 3, 5, 5, 5, 4, 5, 7, 7, 4, 5, 7, 3, 3, 4, 5, 4, 4, 3, 3, 6, 4, 7, 5, 4, 5, 4, 3, 3, 4, 3, 4, 8, 3, 5, 4, 4, 3, 7, 5, 7, 4, 5, 6, 3, 5, 6, 9, 3, 2, 7, 4, 5, 3, 4, 4, 2, 5, 4, 5, 4, 7, 6, 4, 3, 5, 6, 9, 4, 5, 3, 5, 7, 4, 7, 7, 8, 4, 4, 4, 5, 7, 5, 9, 8, 6, 9, 5, 6, 26, 5, 5, 4, 5, 9, 12, 6, 6, 5, 4, 5, 6, 6, 4, 8, 8, 5, 6, 4, 5, 5, 5, 5, 6, 6, 6, 8, 4, 5, 5, 3, 5, 8, 3, 6, 4, 6, 5, 8, 5]

My poor memory after the benchmark



One way to make this code faster is to make sure it can't go back the way it came from when choosing random, unless it's stuck in a corner

100% need to code that tomorrow

If you save the percepts from the other mini-agents as well, then the code will be more efficient. Not just the percepts from the DataFrame

You can use the [Manhattan distance](#) to magnet the random part of `single_out_agent()` towards the goal. This would improve the overall performance as you are slowly solving the overall problem. Idea by [Rohan](#) after showing code to him

First thing I'll code the saving off the percepts from other mini-agents, not just from the real agent's view

Okay finished adding the extra percepts to the set

```

# Initiate set to save percept history for this function. Create if DNE
if not "single_out_agent_percepts" in dir(self):
    self.single_out_agent_percepts=set(self.data["Percept"])

if moved_percept not in self.single_out_agent_percepts:
    # if moved_percept in percept:
    # Add moved_percept to percept history of function (not to main DataFrame)
    self.single_out_agent_percepts.update({moved_percept})
    self.single_out_agent_percepts.update(set(self.data["Percept"]))
    print(possible_actions_percept,possible_actions_position,position)
    print(moved_percept,self.single_out_agent_percepts)
    print(action,"\\n")
return action

```

Going to test if it's more efficient

The average moves after 100 iterations is:

5.29

Didn't seem to change much

Let me run 1000 iterations to make sure I didn't decrease performance

Not sure what happened but it's taking forever so I'll assume it's fine

Oh god there's a loop glitch again

Maybe every position can be remembered what it explored but that would be a lot to code

I think I'll remove the feature where it saves more percepts because it seems to make the code slower. Not too sure why, probably because it depends more on random. It think it's because the percepts seen from other mini-agents were not tested by `self.update()`, so it does not remove all the mini-agents, only one. So it's probably best just to keep check the percepts seen by the real-agent since it can do something about it

This kinda means I wasted 50 minutes coding that

Okay now going to code the random part to not be able to go back where it came from unless in a corner

Created the function `reverse_action()` for `single_out_agent()`'s random part where we don't want the agent to go back. We can now reverse the previous action and remove it from the list of possible action to prevent the agent from going backwards

It seems to work, let's test if it there's an efficiency improvement

Wow a very noticeable change!

The average moves after 100 iterations is:

4.81

We are officially slightly under 5 moves for UP. Now for the icing on the cake from the remaining options it can move, we will choose the direction closest to the goal

Made function to give new position based on direction

```

def action_to_position(self,action,position):
    i=position[0]
    j=position[1]

    if action=="N":

```

```

    i+=-1
elif action=="E":
    j+=1
elif action=="W":
    j+=-1
elif action=="S":
    i+=1

# Return new position
return [i,j]

```

It's tricky to code the direction the agent should take because we don't know where the real agent is. So I should find all the positions with the same possible directions, and then for each direction I sum up the Manhattan distance, then choose the direction with the lowest Manhattan distance.

Or, I could scan through all the positions and find the ones that matches the current percept. Then do the same process

Okay done coding that. Works alright. If I wanted it to work better I can try to find a ratio between how close the agent is to the goal and the h cost

```

# If it ran out of options, go a random possible direction
possible_actions=set(self.possible_actions())
# If it's not stuck in a corner, don't go back where it came from
if len(possible_actions)>1:
    # print("Don't go back",possible_actions,self.reverse_action(self.data["Move"].iloc[-1]))
    possible_actions.remove(self.reverse_action(self.data["Move"].iloc[-1]))

## Choose option closest to the goal
# Find mini-agents with same same possible moves
positions_hcost={}
for position in self.positions:
    if possible_actions & set(self.possible_actions(position)):

        # Find possible positions' h cost for each action
        for action in possible_actions:
            if action not in positions_hcost.keys():
                positions_hcost[action]=self.enviro.M_distance(self.action_to_position(action, position), self.goal)
            else:
                positions_hcost[action]+=self.enviro.M_distance(self.action_to_position(action, position), self.goal)

# Choose action with lowest cost
print("positions_hcost",positions_hcost)
print(possible_actions)
lowest_cost=min(positions_hcost.values())
for key,value in positions_hcost.items():
    if lowest_cost==value:
        action=key

# action=positions_hcost[positions_hcost.values().index(min(positions_hcost.values()))]
print(action)
return action

```

then if it's stuck in the corner (one possible option) it runs the old code

```

print(possible_actions_percept,possible_actions_position)
print("Stuck in corner",self.possible_actions())
action=random.choice(tuple(possible_actions))
print(action,"\n")
return action

```

Will do an efficiency test though I doubt anything will change. If I did an overall efficiency test once the entire AI was done then I bet there would be a change

I want to add construction cones, or "virtual" walls so I got to think of best way to do it with the code I have now.

I would rather if these construction cones stay inside the Agent class

I would know if the real agent is at a dead end, but I won't know which mini agent is the real one until `single_out_agent()` finishes its job

I'll put a `dead_end_check()` right after the action was found

```
# Choose a move
self.choose_action()

# Check for dead ends
self.dead_ends_check()
```

I really don't want to introduce a new value cause that would require a lot of recoding. So instead I'll have the agent place a `1` for a wall, and then we can compare the new state with the initial state and colour all the new walls

This is the code I got, so far it doesn't change the `self.state`

```
def dead_ends_check(self):
    # Check positions that are in a dead end
    for position in self.positions:
        if len(self.possible_actions(position))==1:
            # Create wall at dead end
            self.state[position[0]][position[1]]=1
            print("Dead end at",position)
```

Somewhere in the code the walls I'm placing is getting overwritten and I'm not sure where. I'll create a new variable `self.dead_ends` and in `move()` I'll place the new walls

It's probably this line of code in `move()` overwriting the walls since it replaces the mini agent's old spot with `0`

```
# Place new mini-agents
self.place_agents(place_positions=True)
```

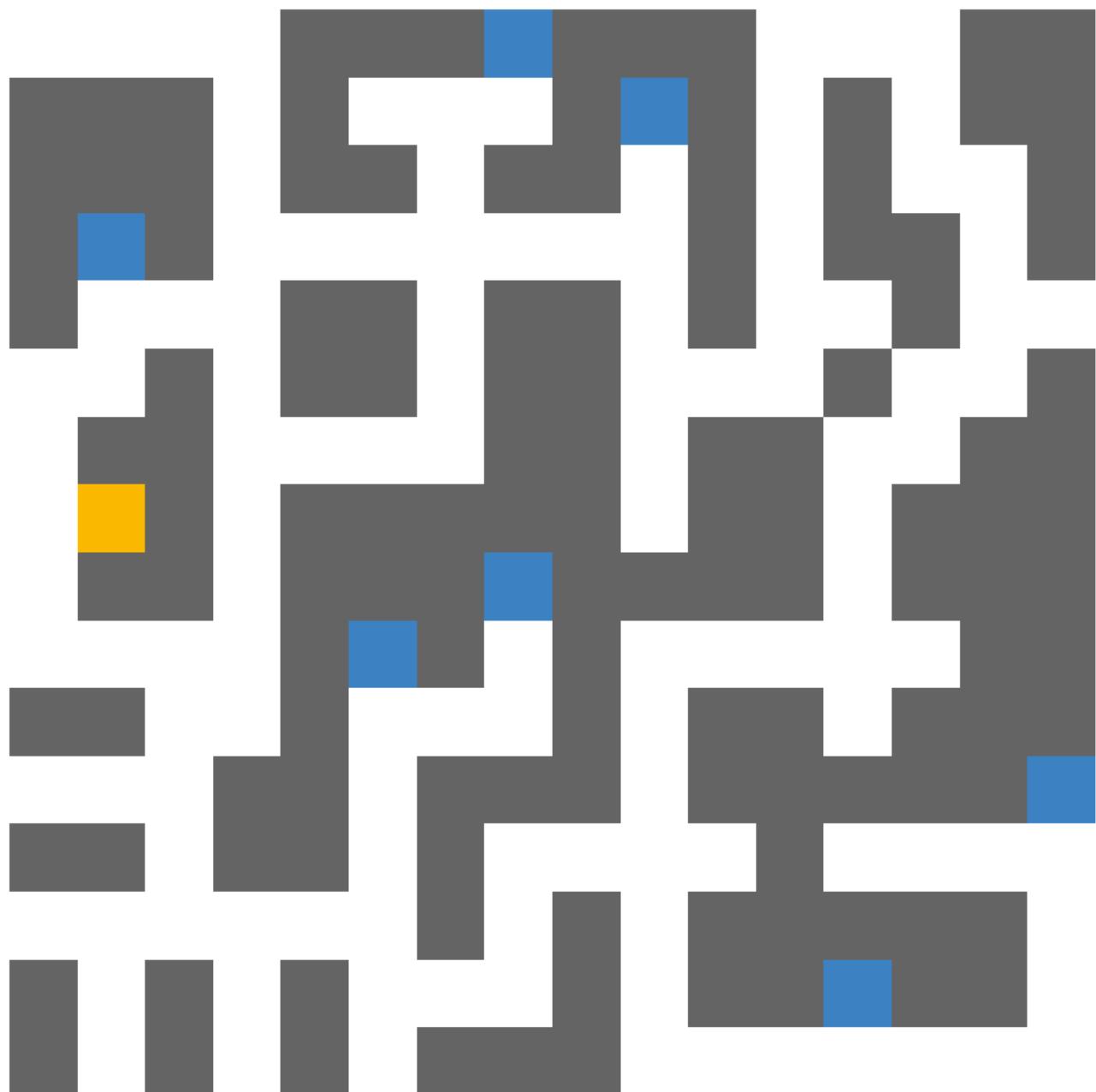
Oh never mind, the function resets the `self.state` with the initial state which is why my `1`s are being deleted

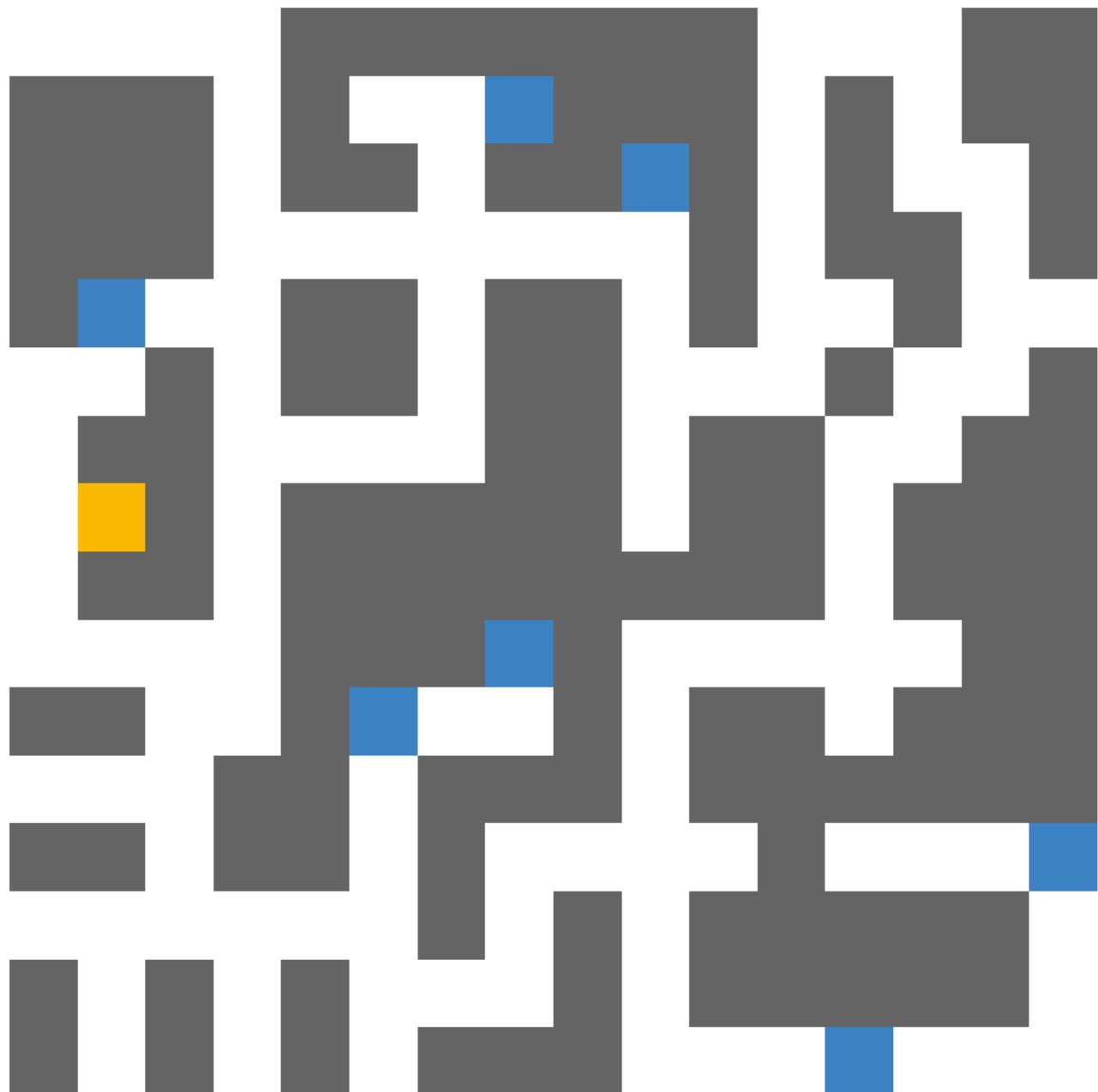
Applied the code and the real agent moved into a dead end, and then it was detected as a dead end then the real agent was deleted. I think it's because the real agent no longer matches the percept from the Environment class. So I also got to update the Environment class

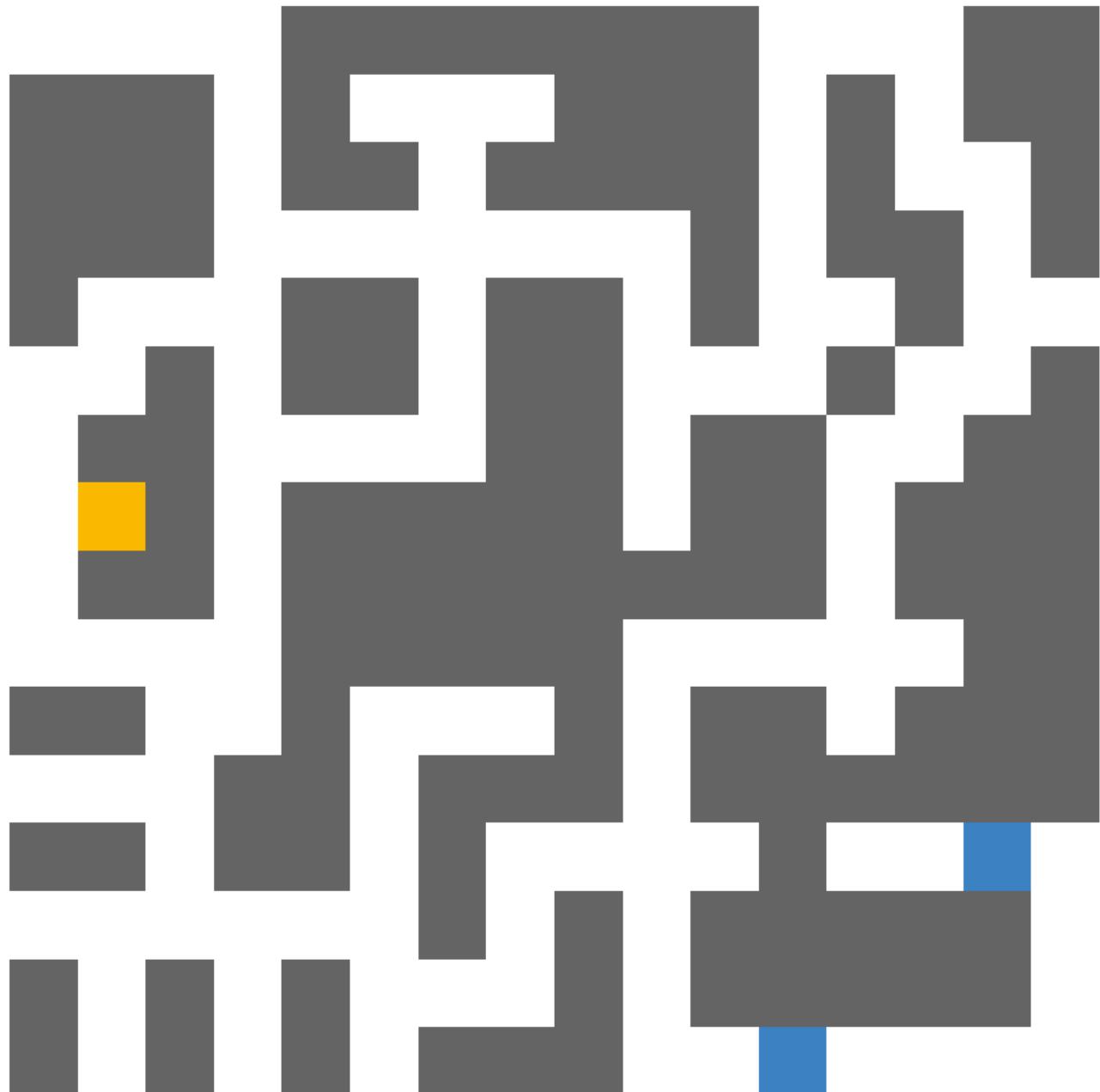
By placing a wall it also allows the real agent have a new percept to use if it goes back

 [Example >](#)

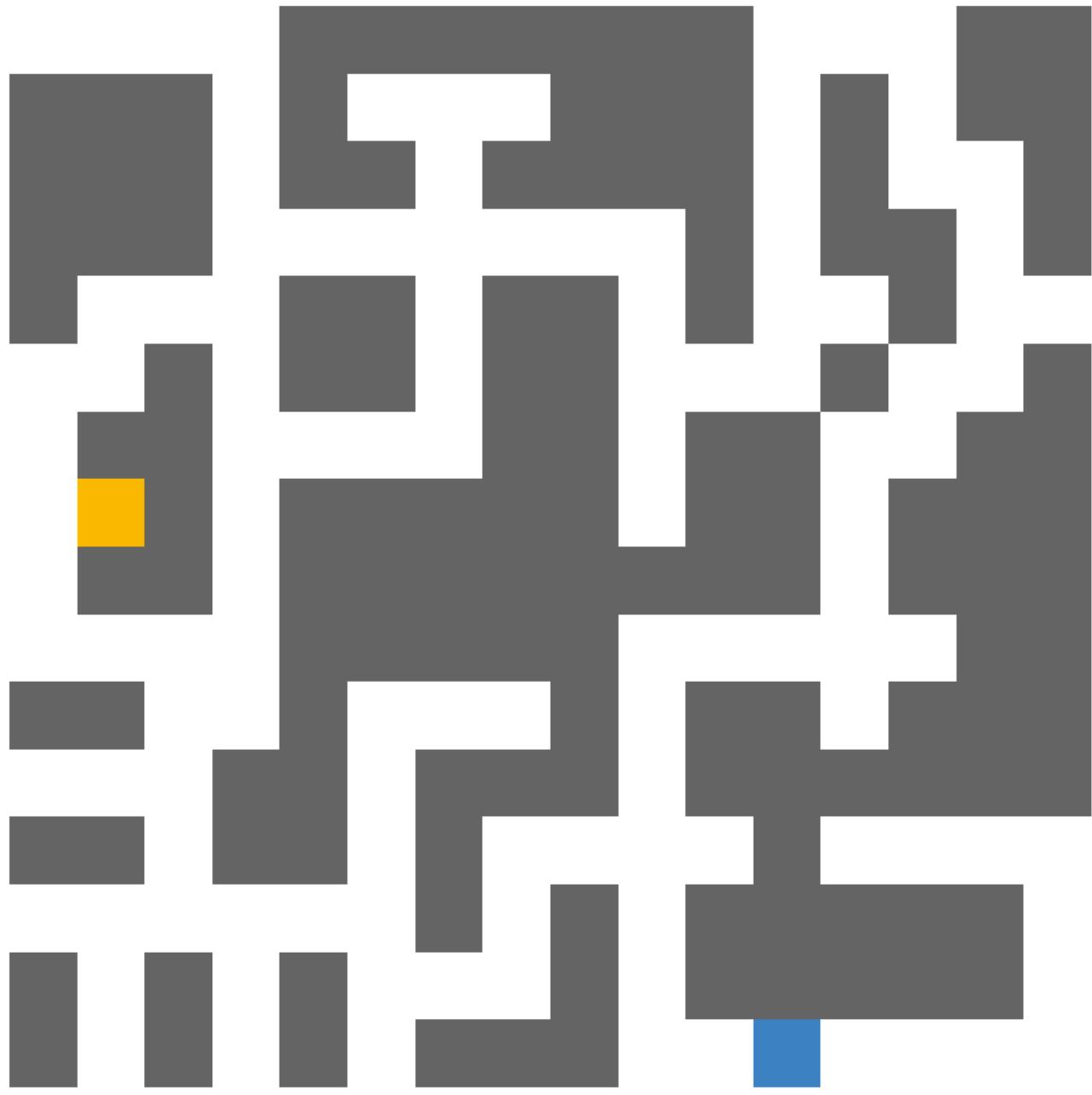












This line causes a problem probably because the new `possible_actions` will no longer have the action that brings the agent back since it's replaced with a wall

```
possible_actions.remove(self.reverse_action(self.data["Move"].iloc[-1]))
```

Okay I think I'm doing coding it this is how it works

1. In `run()`, the Agent checks where the dead ends are on the current `self.positions` before `self.move()`

```
# Choose a move  
self.choose_action()  
  
# Check for dead ends  
self.dead_ends_check()
```

2. In `move()`, before officially updating the Environment state, the walls are placed in the Agent's `self.state`. After officially updating the Environment state, the walls are placed in the Environment `self.state`

```

# Place walls at dead ends
for position in self.dead_ends:
    self.state[position[0]][position[1]]=1

# Officially update the Environment state
self.enviro.update(self.action)

# Place walls at dead ends after updating Environment state
self.enviro.place_dead_ends(self.dead_ends)

```

### 3. Repeat

What I'm scared of is that Environment's state isn't synced up with Agent's state anymore. I'll do a performance test and if anything can go wrong it will happen during a bunch of samples. But from running it myself the real agent is never deleted and everything seems to be working

Hm it seems not much has improved

```

The average moves after 100 iterations is:
5.12

```

Well, I keep the changes, now I got to have the code change color in `visualize()` when a construction cone is there

How I will visualize the orange cones

```

# For dead ends
if self.dead_ends:
    cmap=ListedColormap(["#ffffff", "#666666", "#3d85c6", "#fbbc04", "#ff6d01"])

# Replace all dead end positions with 4
for position in self.dead_ends:
    state[position[0]][position[1]]=4

```

I will create a new `cmap` if there's dead ends to be placed since I got to add a new value `4`

Turns out there's a problem with the dead end code that isn't related with `visualized()` that I have missed because there's an error at line

```
lowest_cost=min(positions_hcost.values())
```

since there were not options given which is worrying

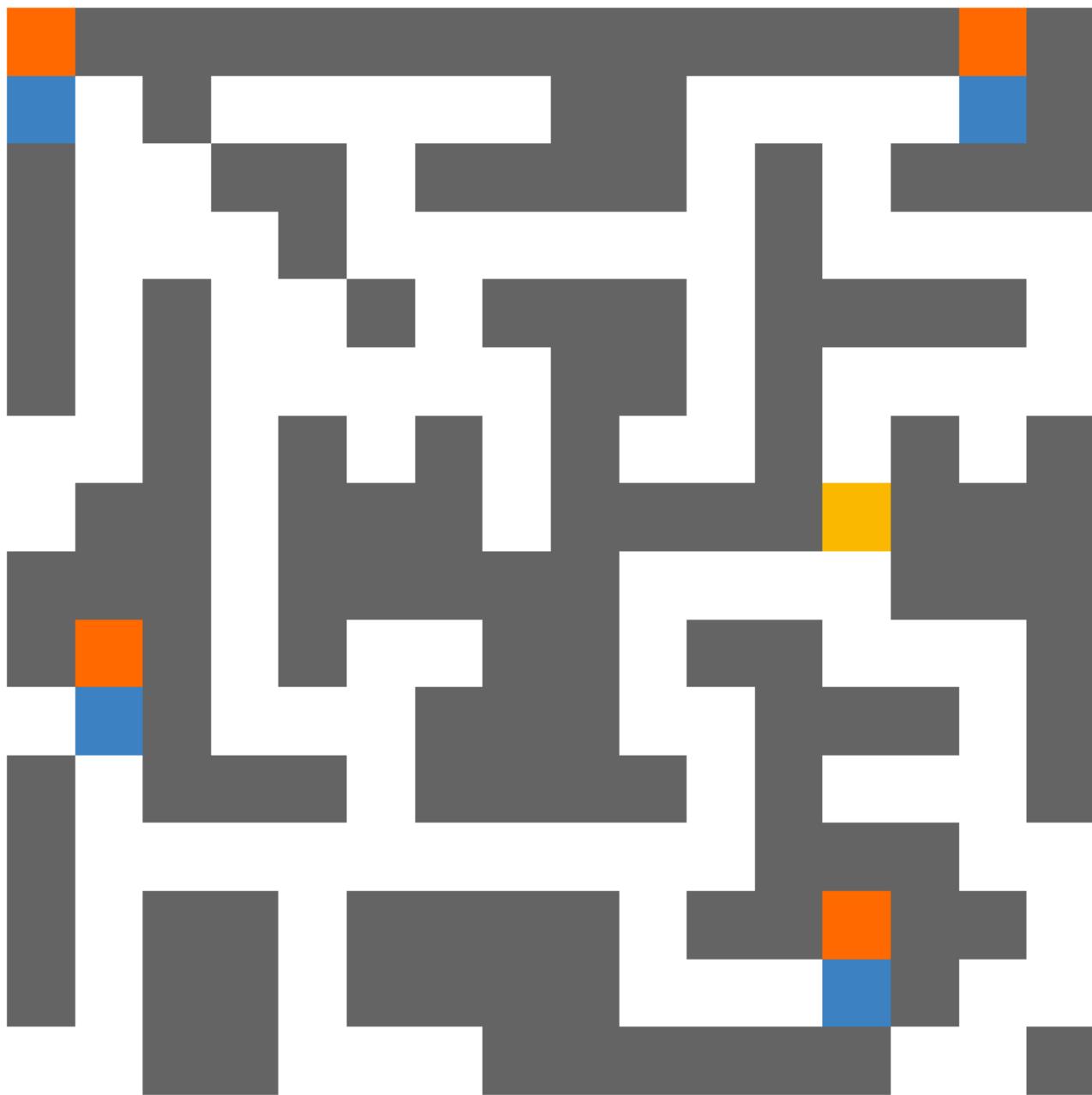
```

positions_hcost {}
{'S', 'W'}

```

Also I got to add a check in `dead_ends` to make sure the goal is not in a dead end or then the Agent will never be able to reach it

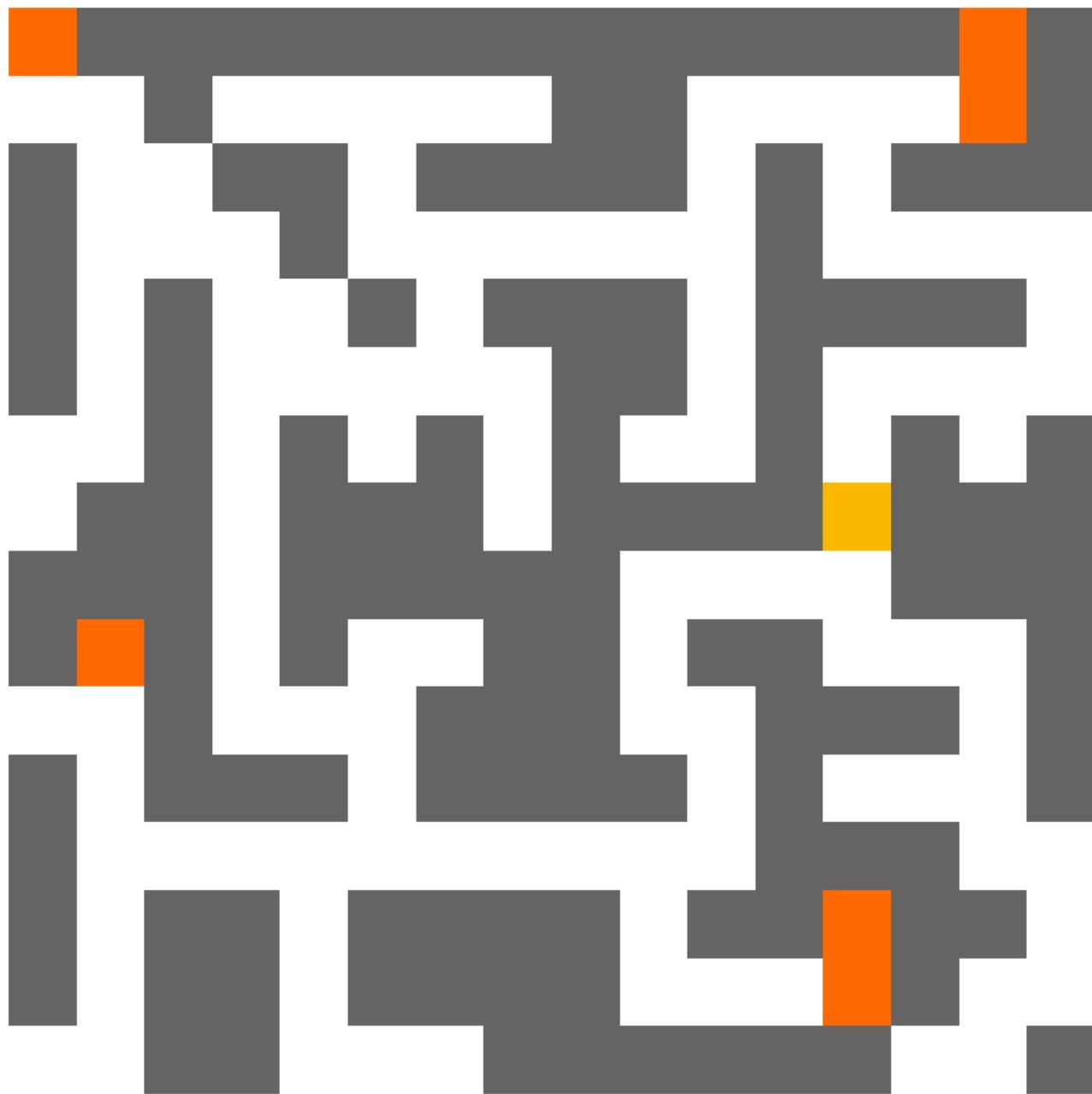
Despite it not being visualized the first agent moved E here



```
Dead end at [0, 0]
Dead end at [0, 14]
Dead end at [9, 1]
Dead end at [13, 12]
{'E'} {'W'}
Stuck in corner ['E']
E
```

and then somehow the code kept going and now it says the available options are S and W, but there should be a wall where it just came from. Also, positions\_hcost is empty so weird glitch

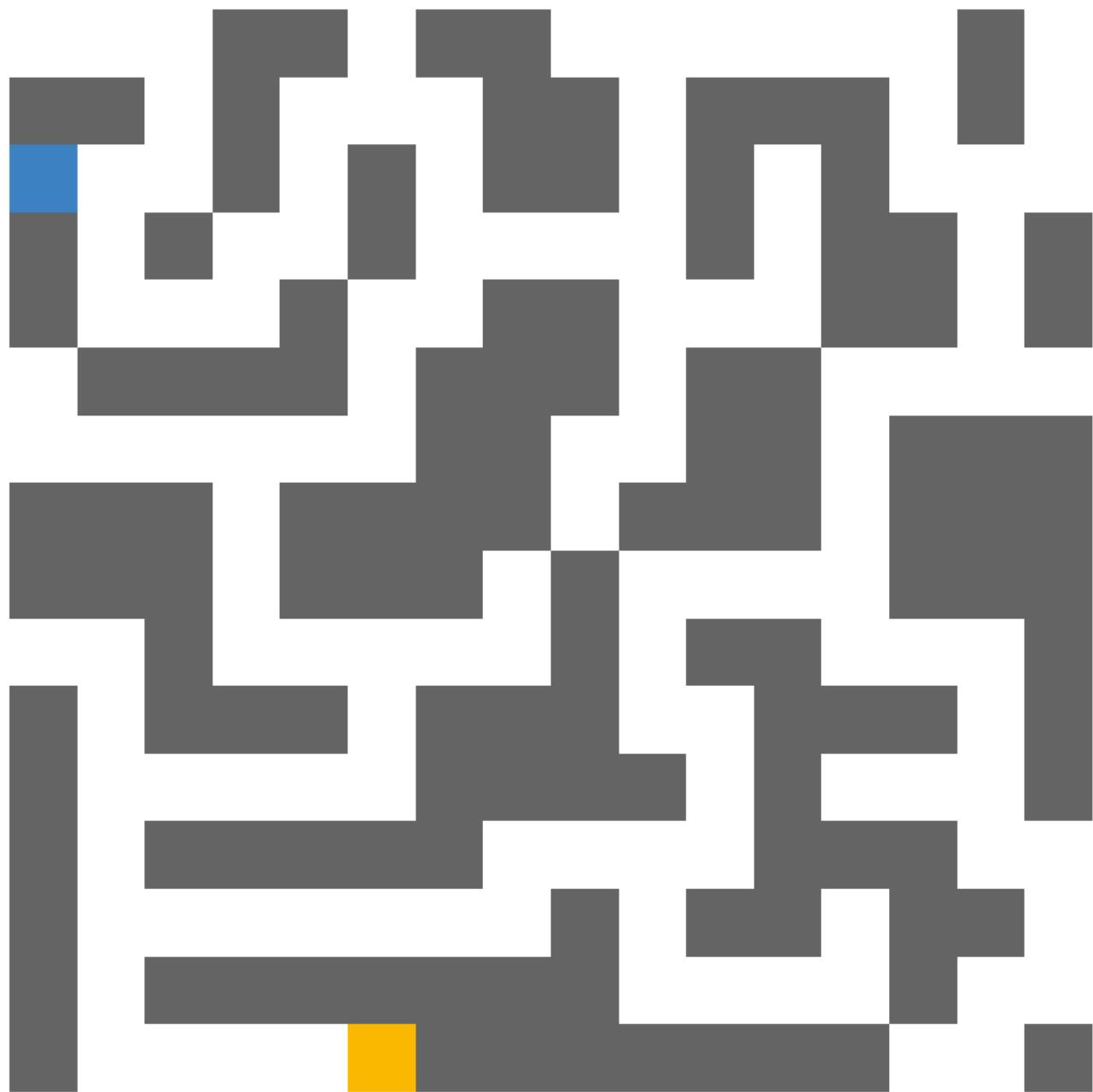
```
Dead end at [1, 14]
Dead end at [14, 12]
positions_hcost {}
{'S', 'W'}
```

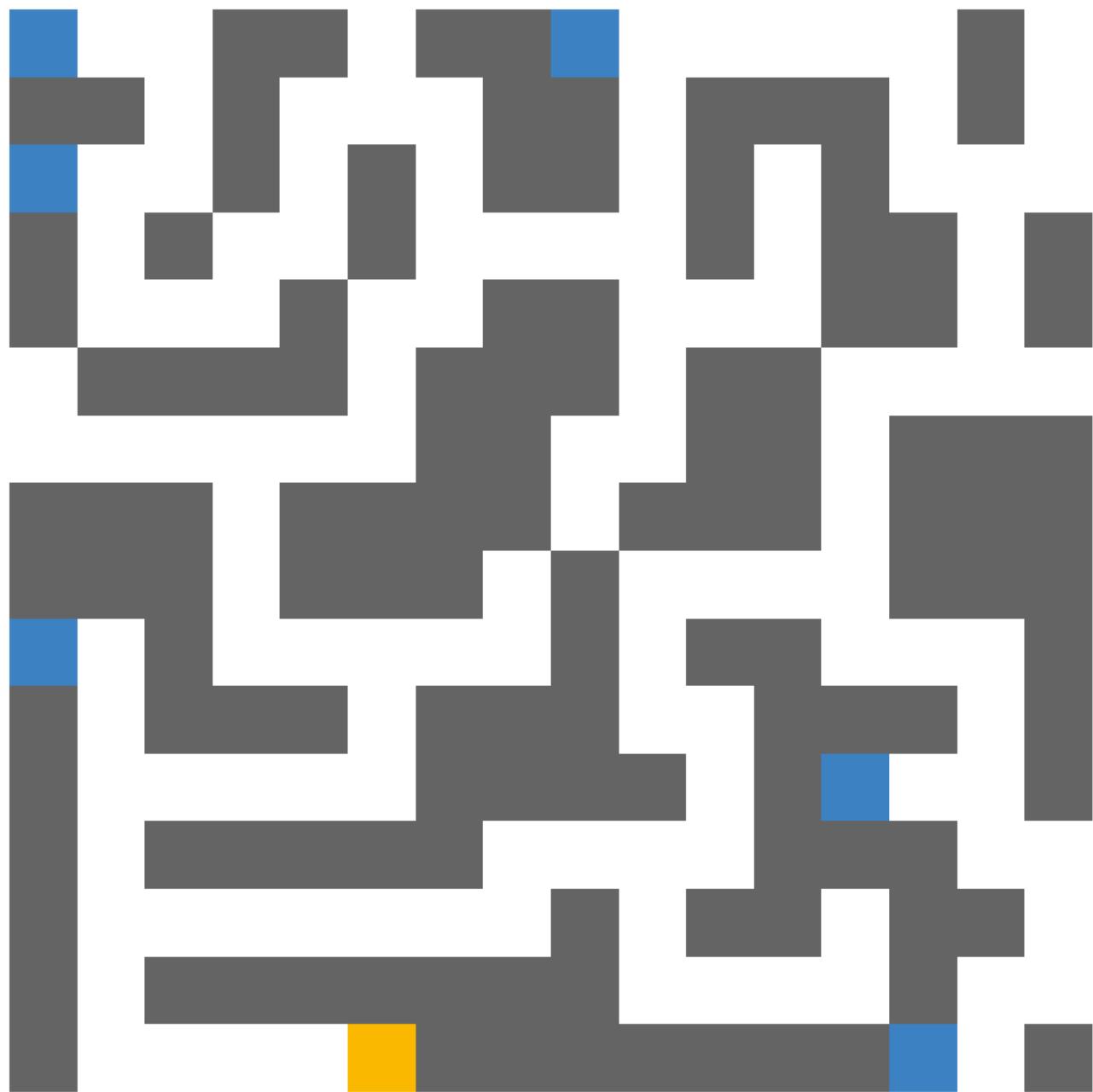


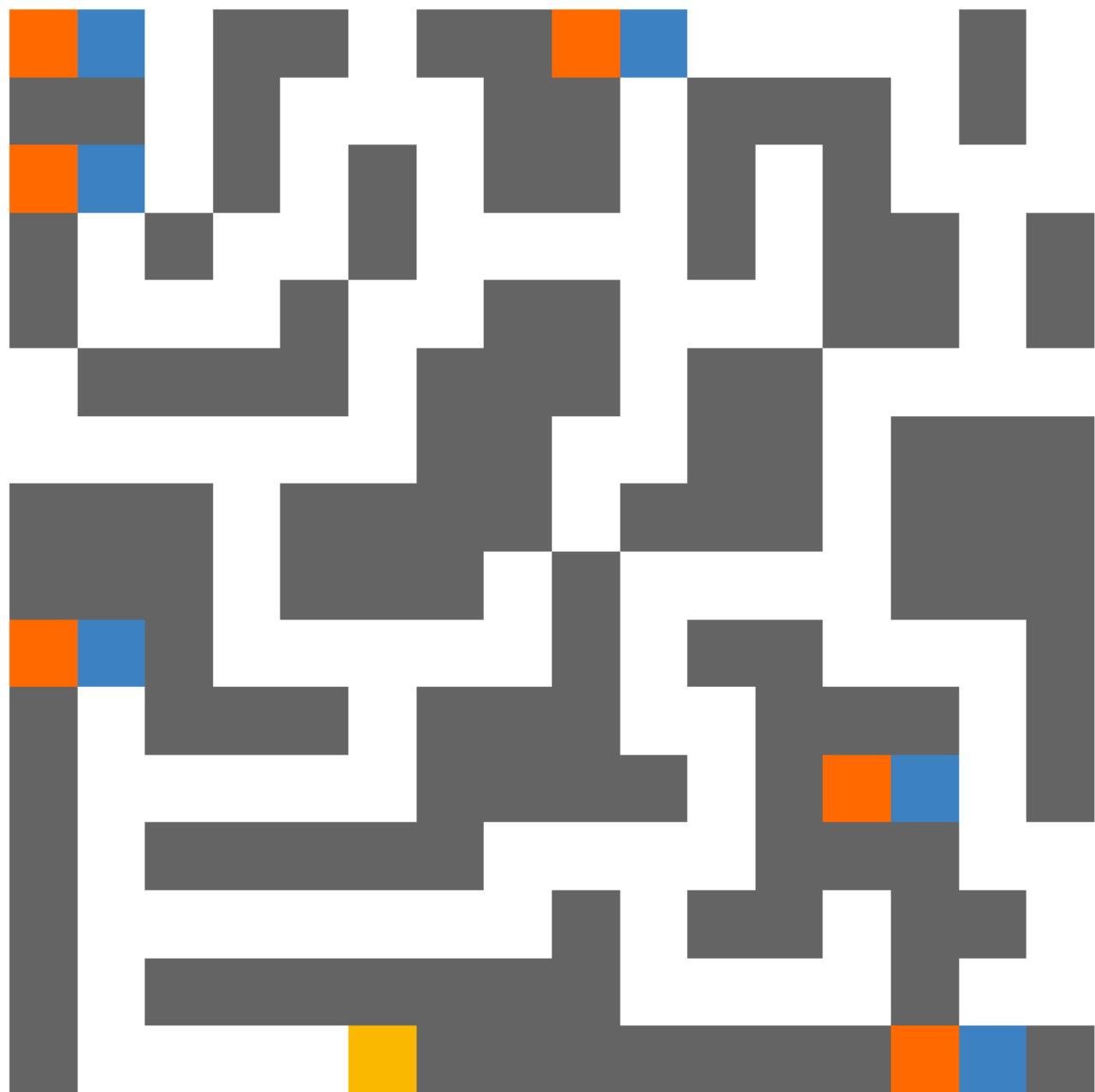
Just got a glitch where the real agent died

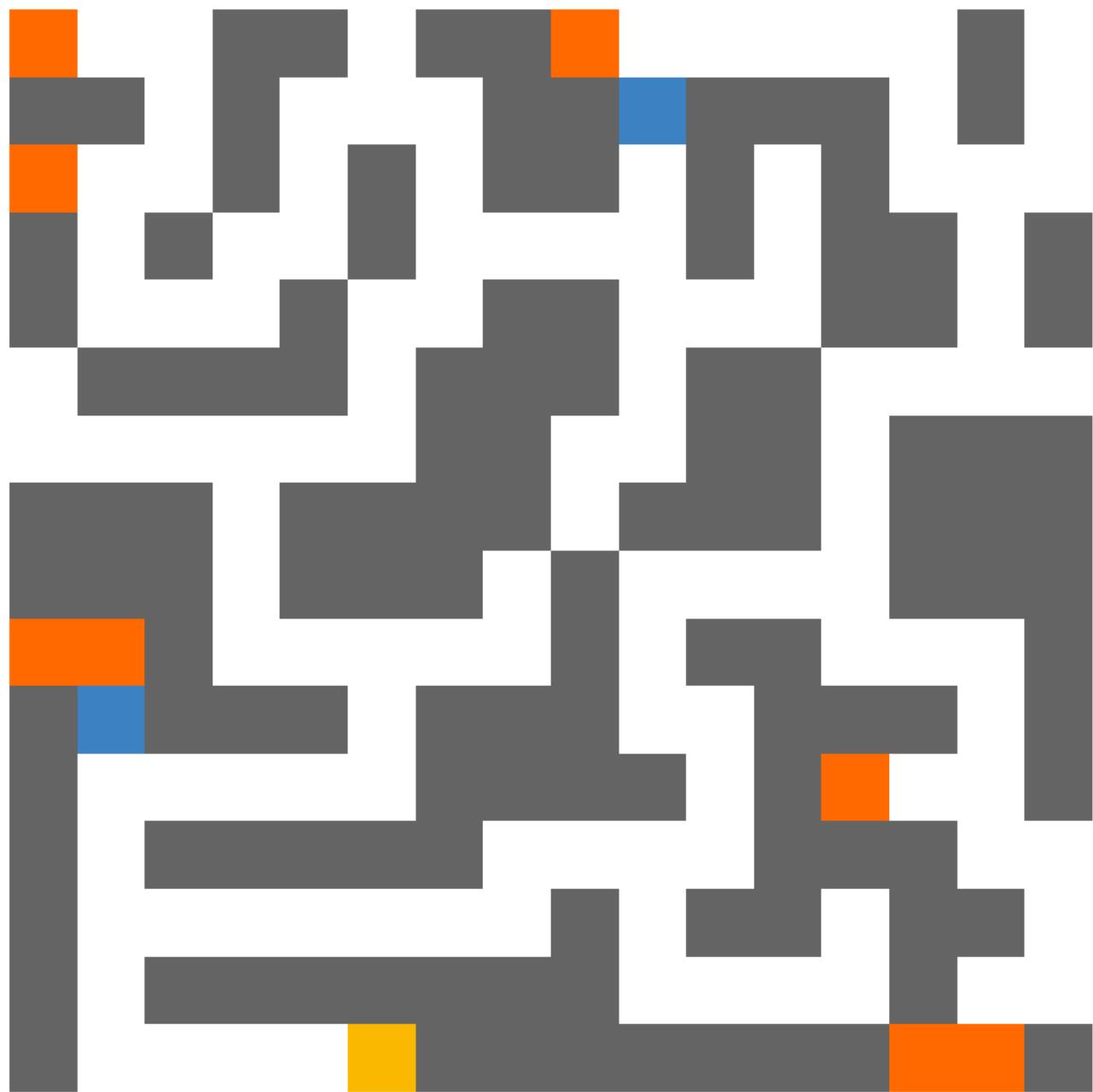
The glitch seems to happen at corners

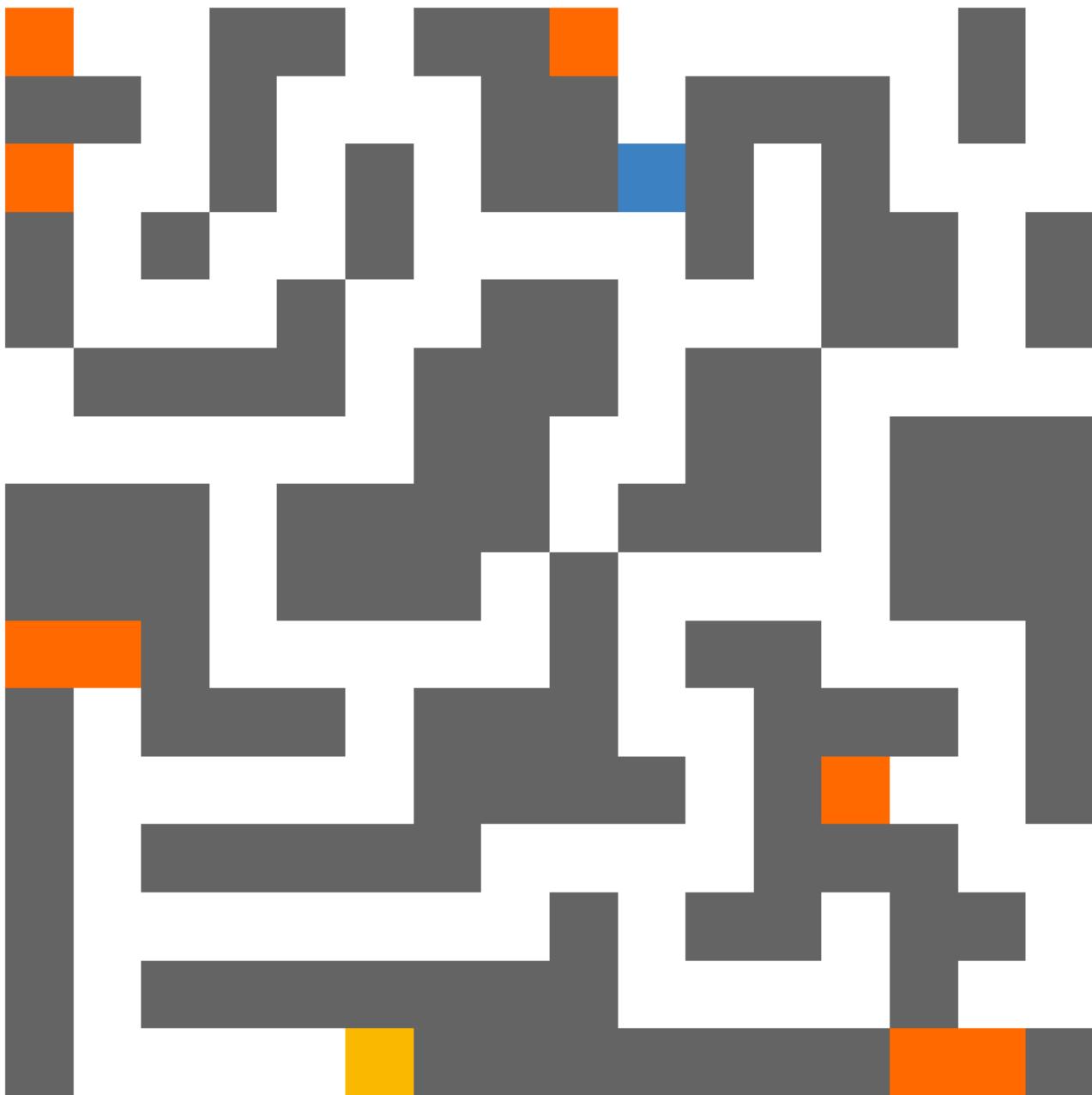
[Glitch >](#)









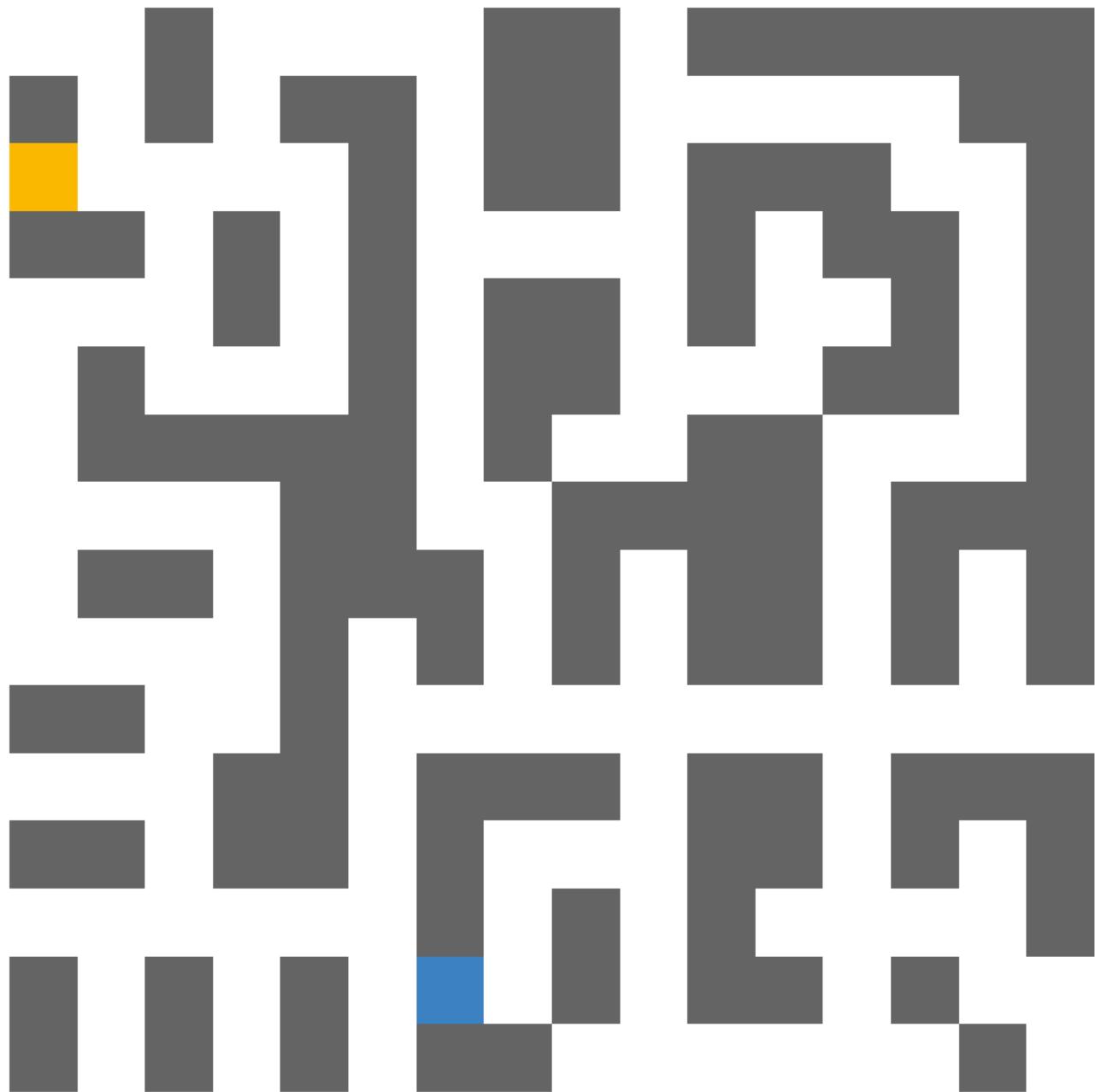


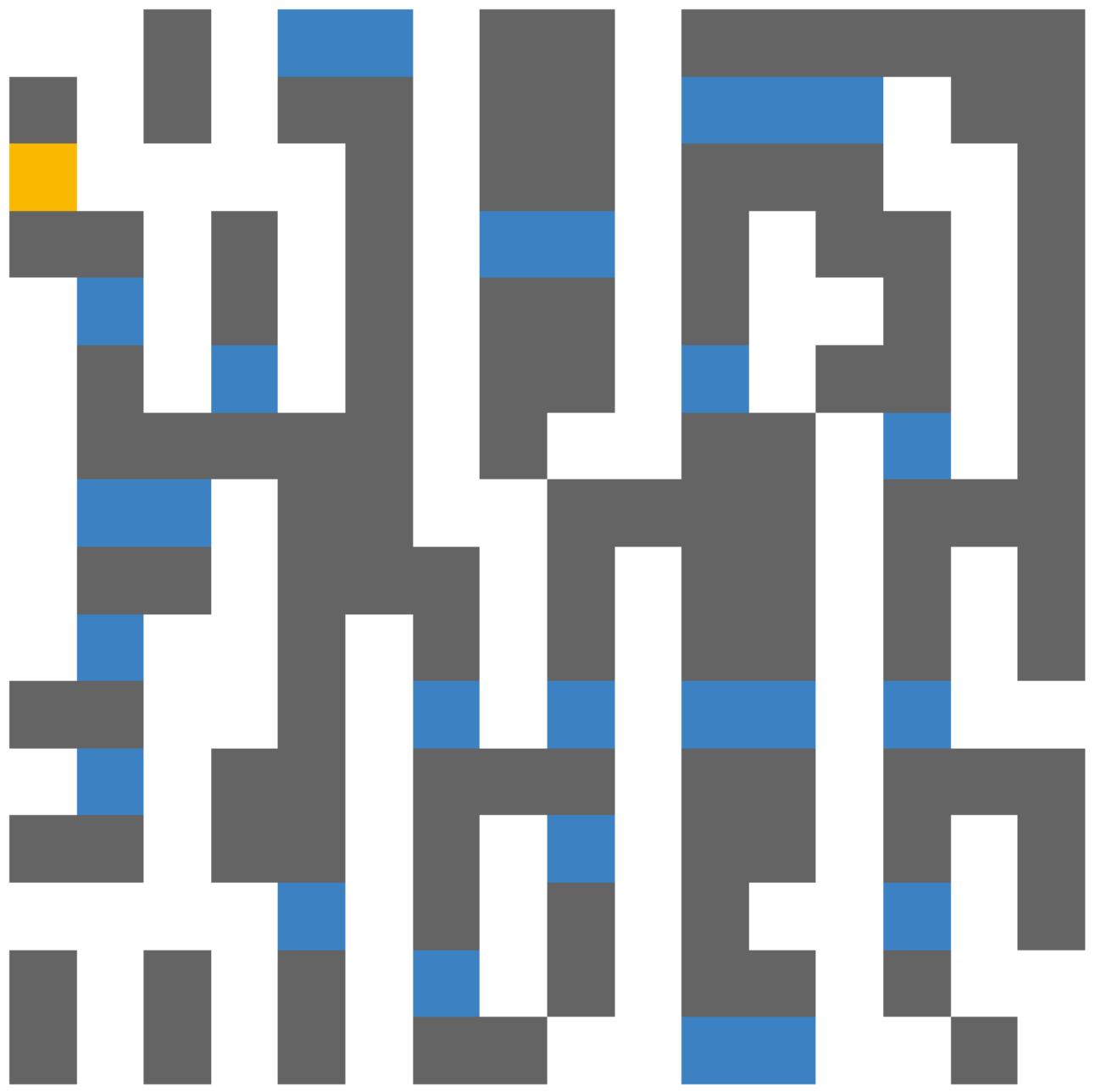
Index	Belief state	Positions	Percept	Move
0	[[2, 0, 0, 1, 1, 0, 1, 1, 2, 0, 0, 0, 0, 0, 1,... [[0, 0], [0, 8], [2, 0], [9, 0], [11, 12], [15, 13]]	[[0, 0], [0, 8], [2, 0], [9, 0], [11, 12], [15, 13]]	1011	None
1	[[2, 0, 0, 1, 1, 0, 1, 1, 2, 0, 0, 0, 0, 0, 1,... [[0, 0], [0, 8], [2, 0], [9, 0], [11, 12], [15, 13]]	[[0, 0], [0, 8], [2, 0], [9, 0], [11, 12], [15, 13]]	1011	E
2	[[4, 2, 0, 1, 1, 0, 1, 1, 4, 2, 0, 0, 0, 0, 1,... [[0, 9], [9, 1], [15, 14]]	[[0, 9], [9, 1], [15, 14]]	1010	S
3	[[4, 0, 0, 1, 1, 0, 1, 1, 4, 0, 0, 0, 0, 0, 1,... [[1, 9]]	[[1, 9]]	0110	S

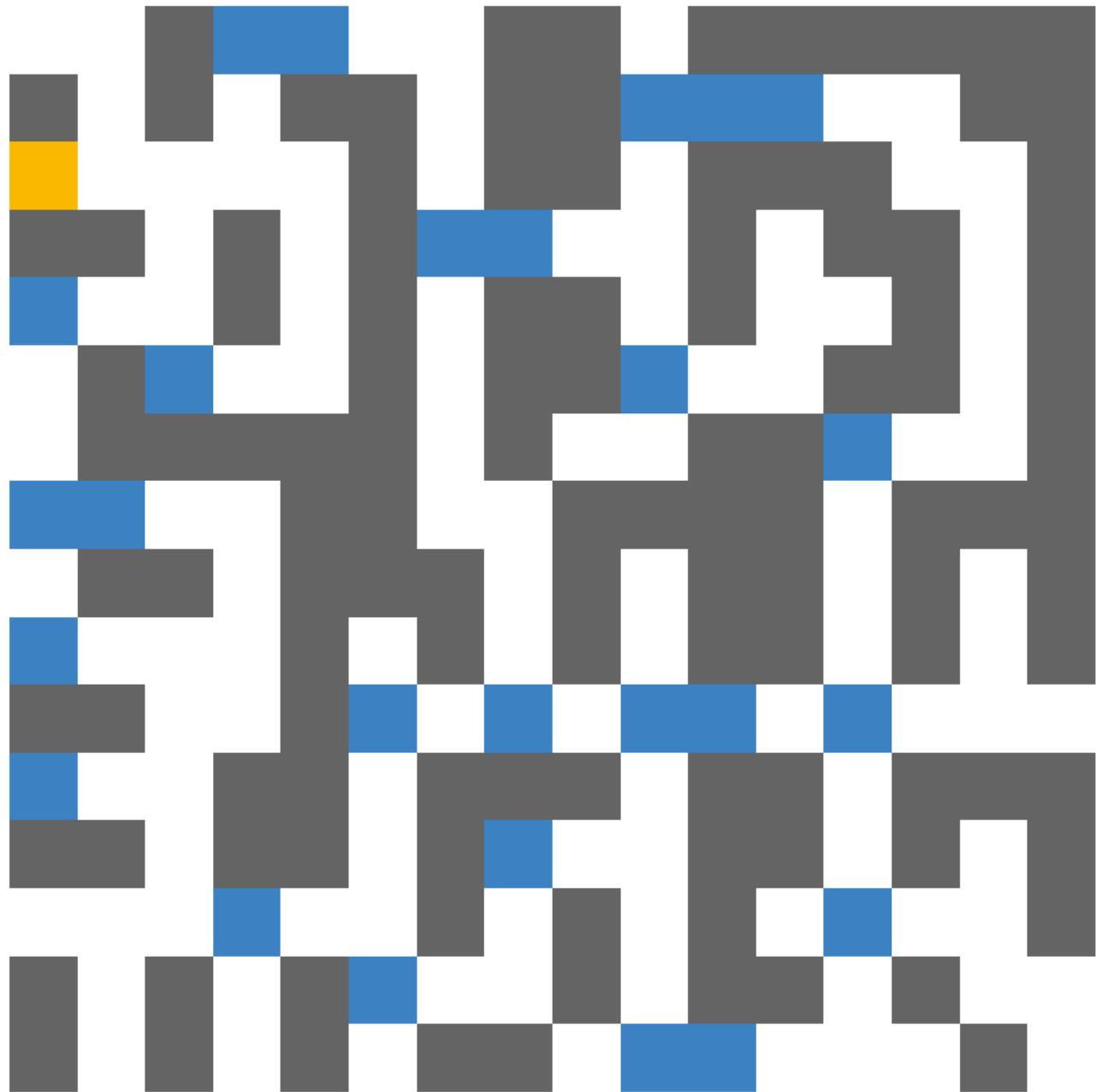
I think the percept that updated because of the orange wall is not synced with the Environment's percept

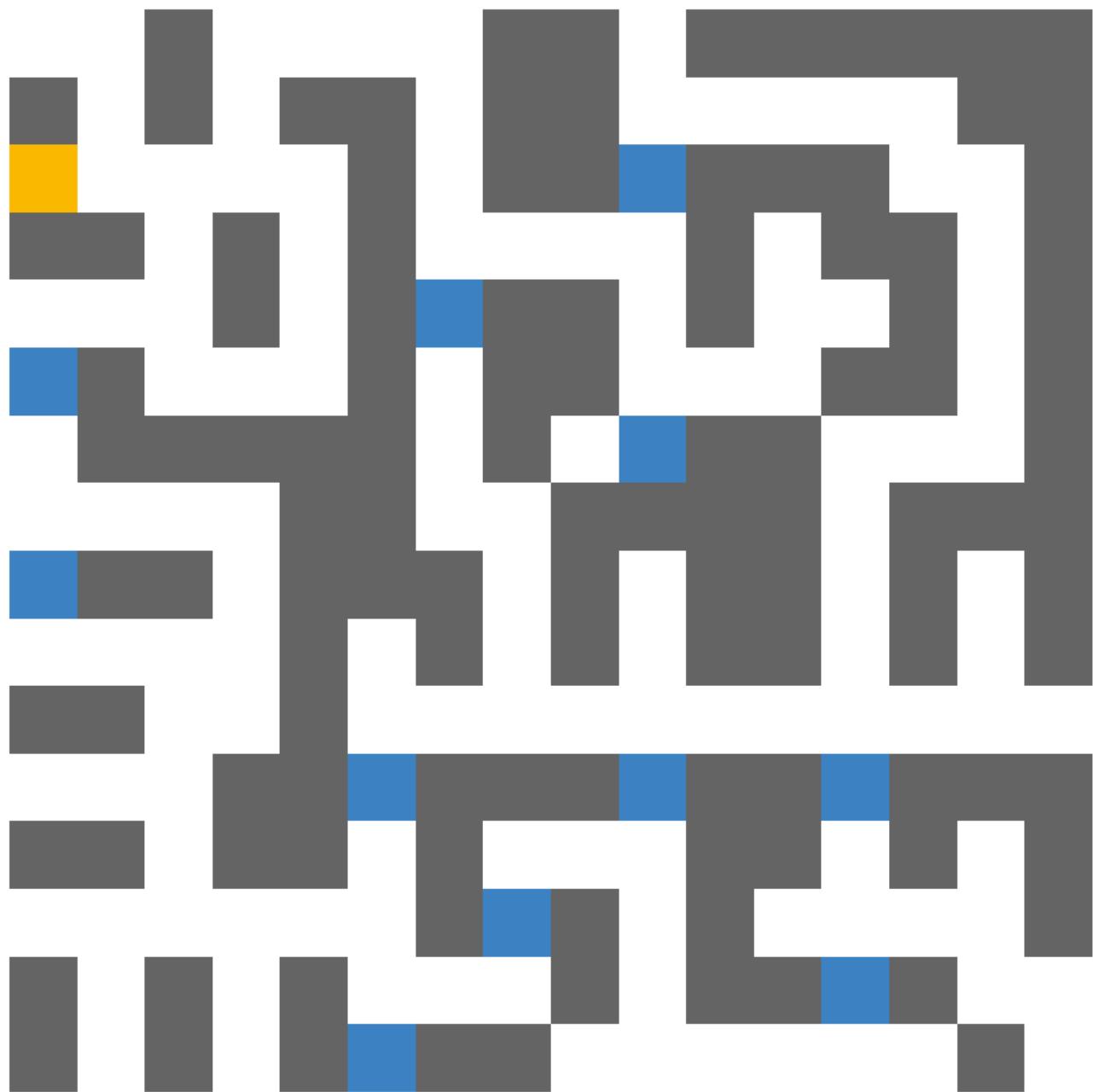
Cool example of orange wall not letting real agent back in dead end

[Example >](#)

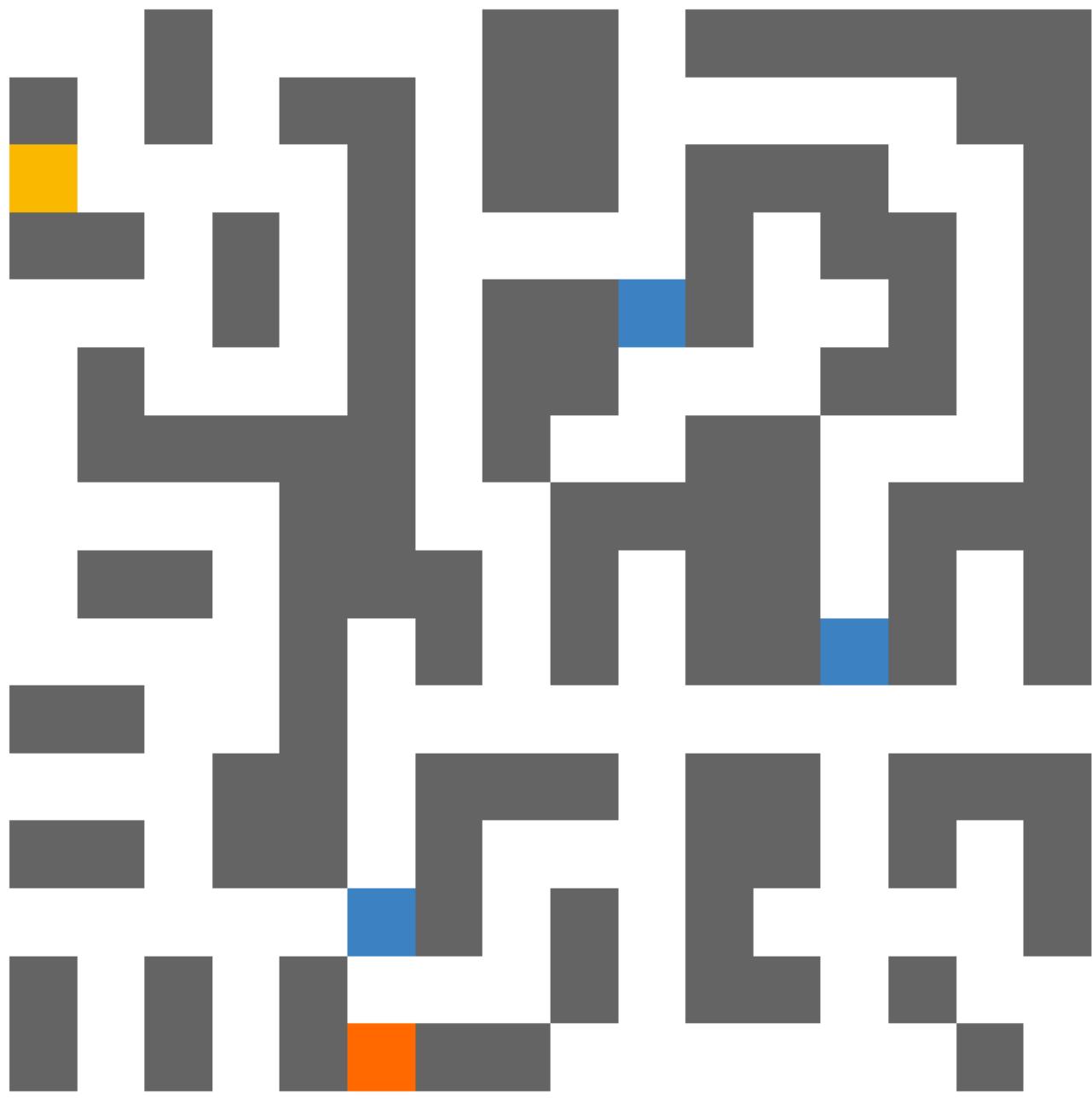


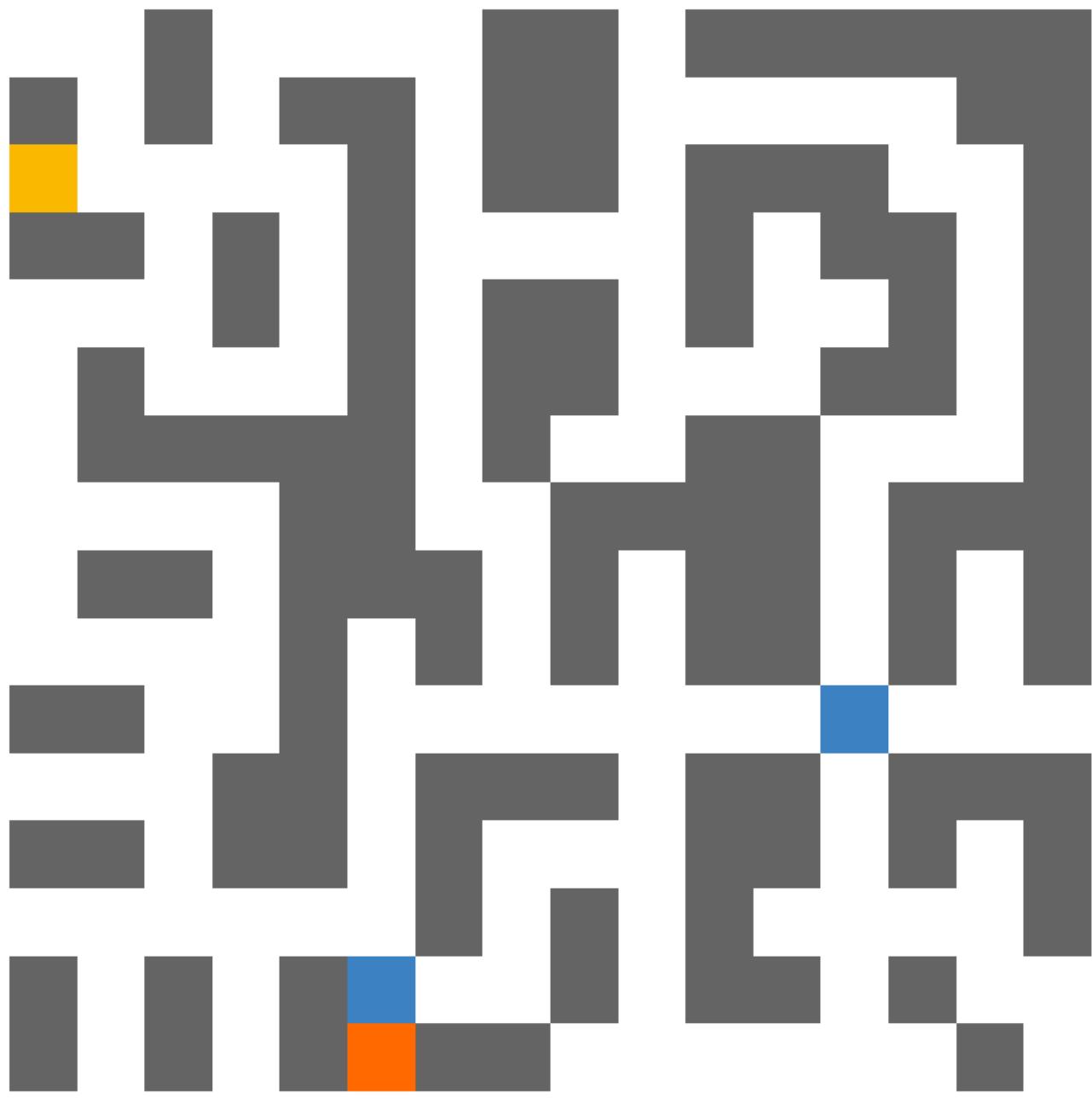














Placing `self.enviro.visualize()` after the Agent is done running is the most convenient thing ever. I get to click back one frame to see if the last agent surviving is the real agent. And I get to see the walls be filled up

Last frame:



```
self.enviro.visualize():
```



In `visualize()`, instead of

```
state=self.state
```

I replaced it with

```
state=copy.deepcopy(self.state)
```

for just in case

An infinite loop was caused by the `positions_hcost()`

```
positions_hcost {'N': 25, 'W': 25}
{'N', 'W'}
W
positions_hcost {'S': 23, 'E': 23}
{'S', 'E'}
E
positions_hcost {'N': 25, 'W': 25}
{'N', 'W'}
```

W  
positions\_hcost {'S': 23, 'E': 23}  
{'S', 'E'}  
E  
positions\_hcost {'N': 25, 'W': 25}  
{'N', 'W'}  
W  
positions\_hcost {'S': 23, 'E': 23}  
{'S', 'E'}  
E  
positions\_hcost {'N': 25, 'W': 25}  
{'N', 'W'}  
W  
positions\_hcost {'S': 23, 'E': 23}  
{'S', 'E'}  
E  
positions\_hcost {'N': 25, 'W': 25}  
{'N', 'W'}  
W  
positions\_hcost {'S': 23, 'E': 23}  
{'S', 'E'}  
E  
positions\_hcost {'N': 25, 'W': 25}  
{'N', 'W'}  
W

These two frames looped:





til



Got an error of `possible_actions` being empty

```
action=random.choice(tuple(possible_actions))
```

Got to fix `possible_actions` causing so many problems

Since the problems with `possible_actions` is rare to run into I'll ignore it until it pops up again. I got to start coding

Added an and statement to make sure the goal is not replaced with a construction cone

```
if len(self.possible_actions(position))==1 and position!=self.goal:
```

Added orange cones to Environment's `visualized()`

For some reason I cannot have Environment recognize when it has `self.Agent` in it's own class. That means I cannot have the orange cones shown using Environment visualize

```

# For dead ends (once the Agent class is created)
if hasattr(self, 'Agent'):
    if self.Agent.dead_ends:
        print("Environment visualized",self.Agent.dead_ends)
        cmap=ListedColormap(["#ffffff", "#666666", "#3d85c6", "#fbcc04", "#ff6d01"])

    # Replace all dead end positions with 4
    for position in self.Agent.dead_ends:
        state[position[0]][position[1]]=4

```

Probably because the `Agent` class is not done running it's `__init__`. So that means I should have `Agent.run()` ran inside the `Environment` class

Okay I was correct. I placed `self.visualize()` right after the `Agent` class was created in `Environment` and it worked as expected

```

self.Agent=Agent(self.agent_name,initial_state_clueless,percept,self,self.goal)
self.visualize()

```

Caused another error since `possible_actions` was empty

```

action=self.possible_actions()[0]

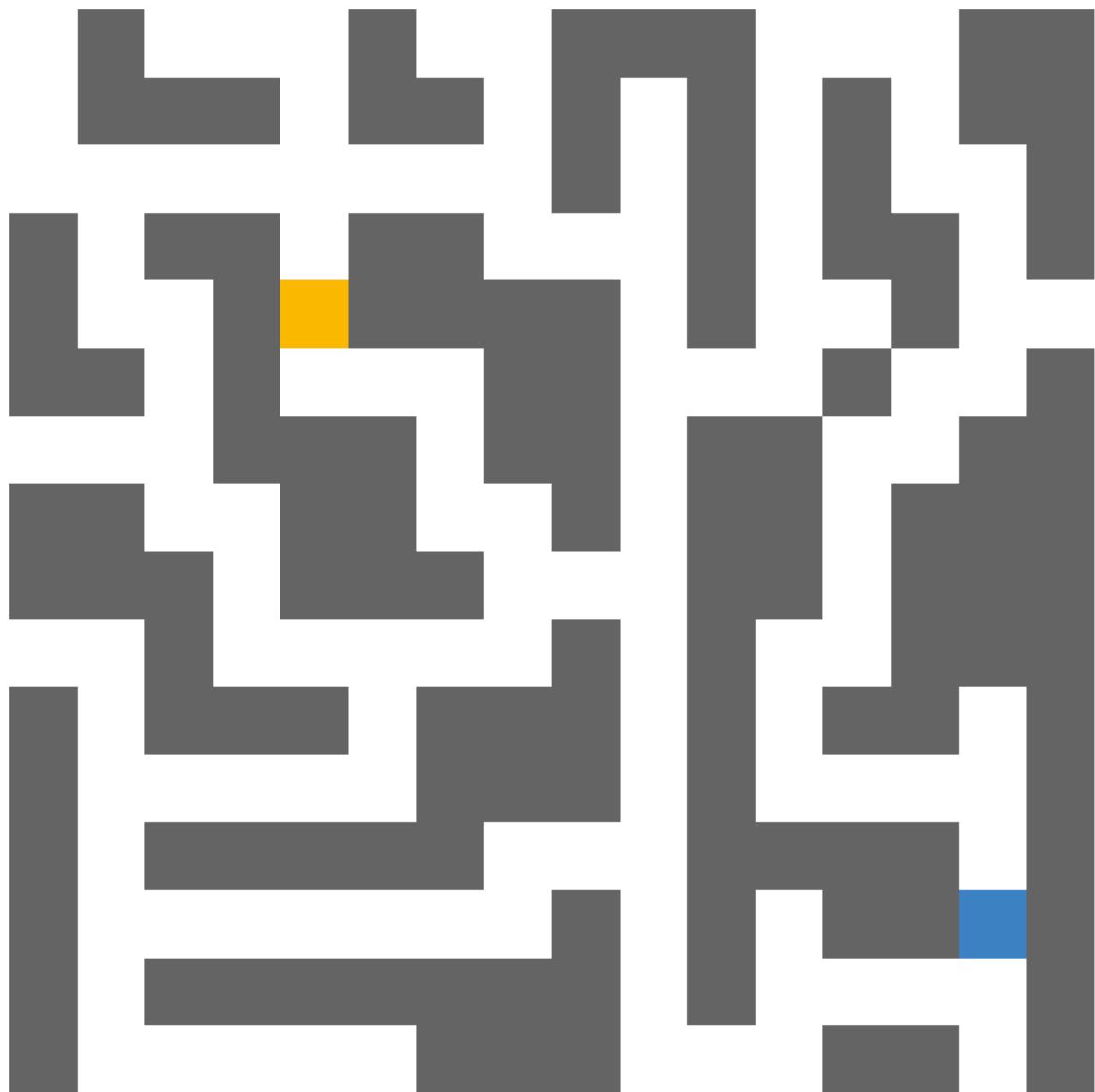
```

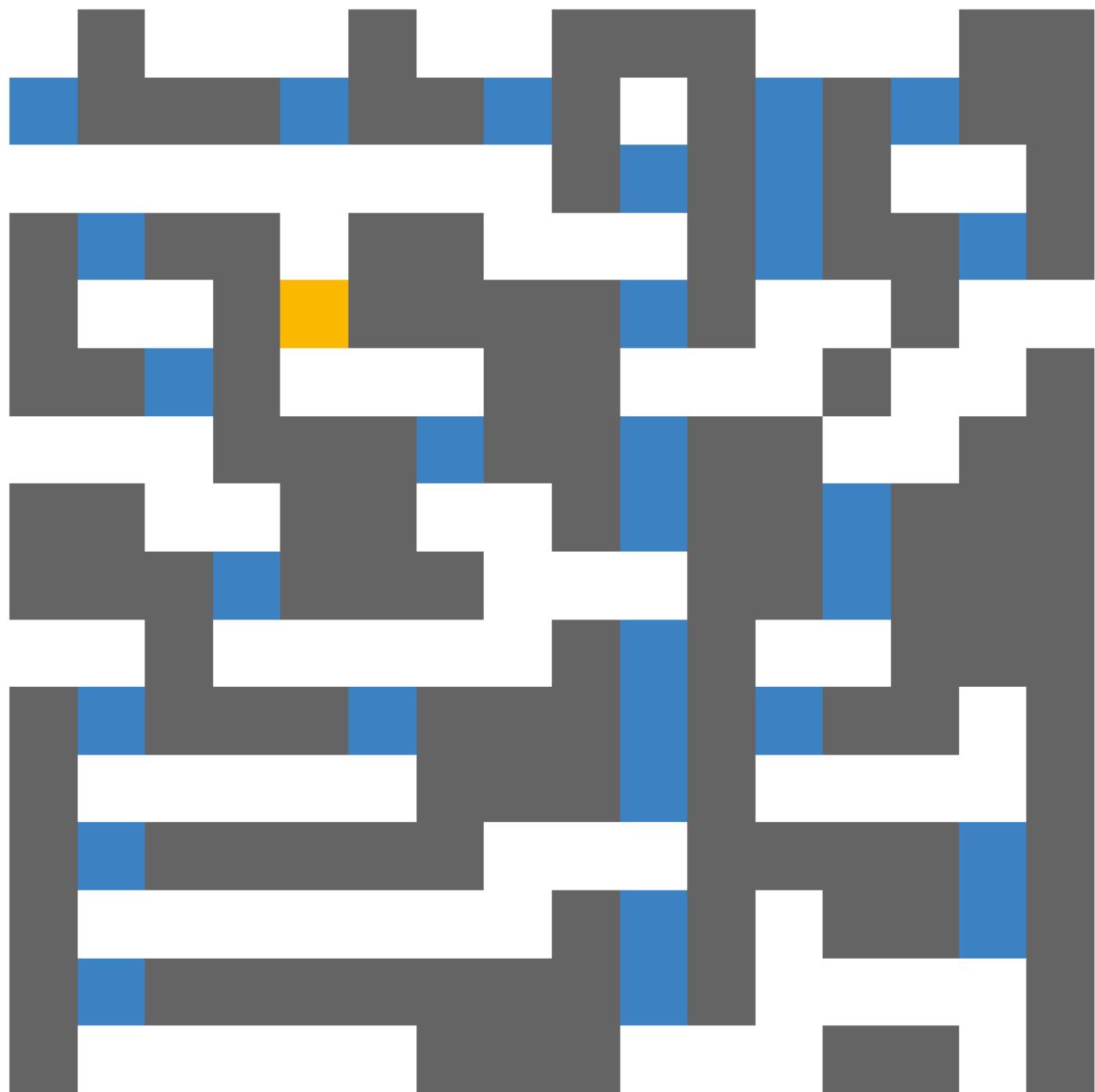
Just realized until I fixed the infinite loop error that happens sometimes with two agents and the `possible_actions` being empty then I cannot do benchmarks ahhh

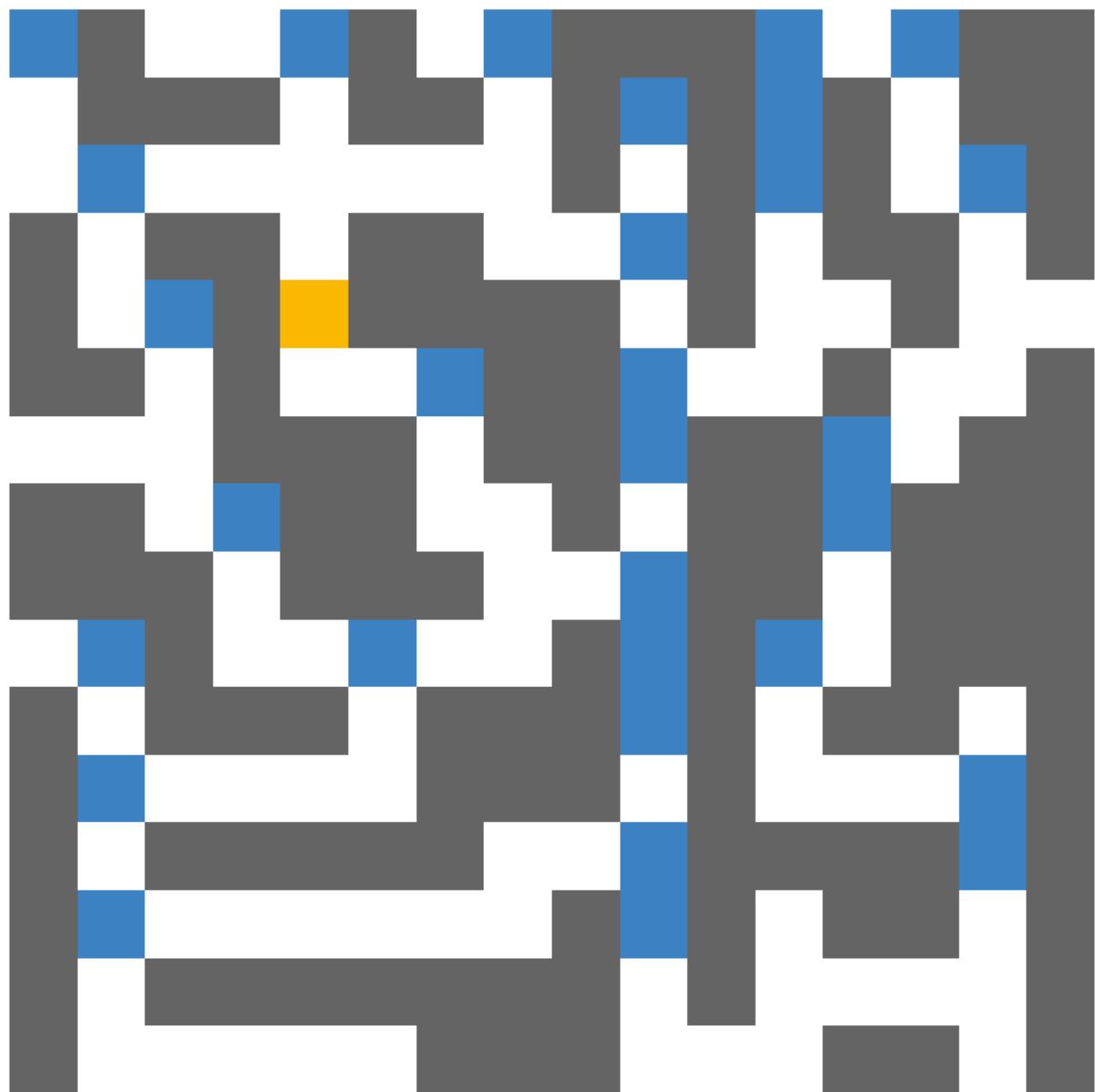
Another glitch happens where the real agent gets deleted so I got solved these three really important problems that also happens quite rarely so it's hard to verify if its fixed. Okay, final boss battle

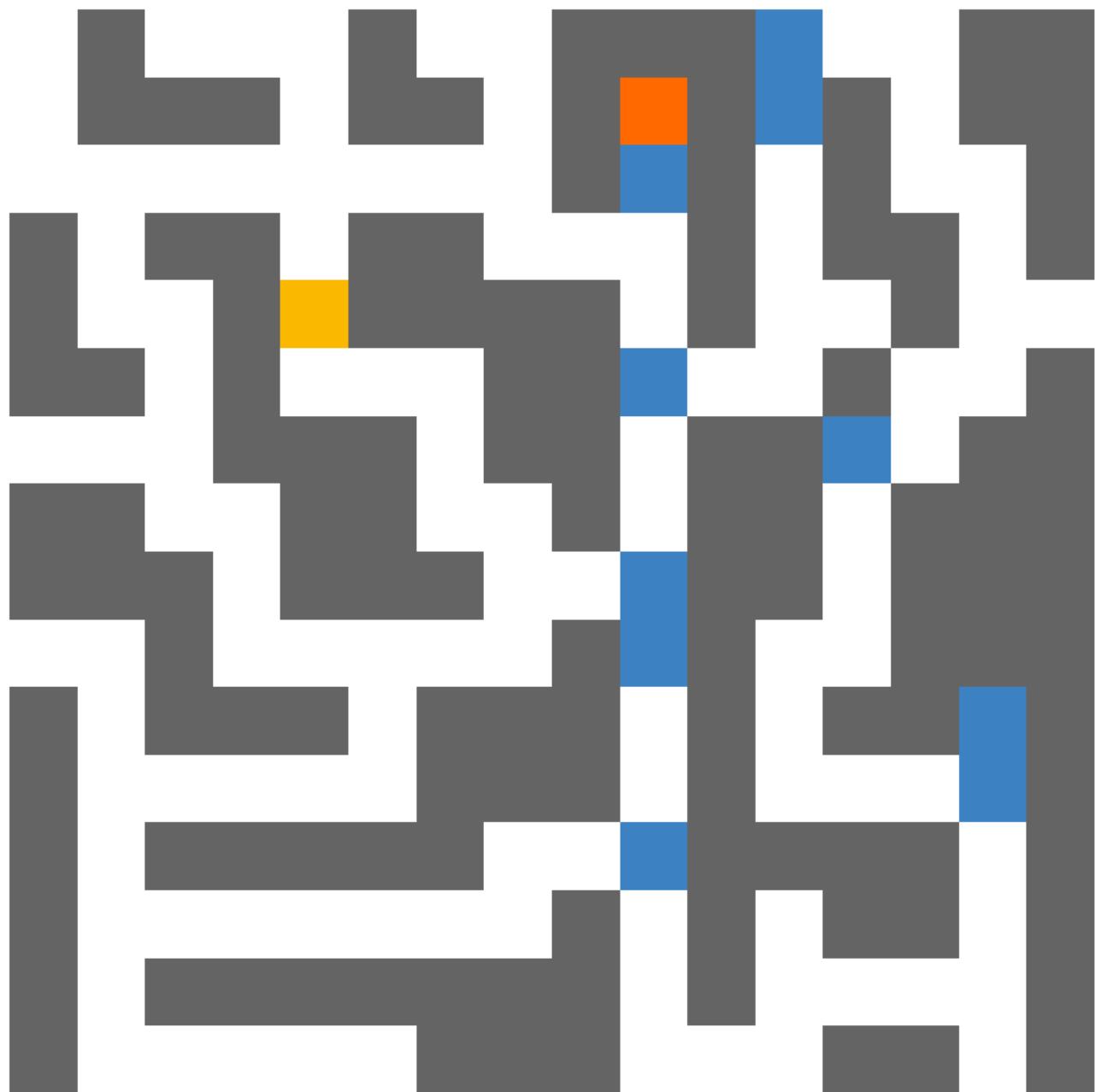
This scenario covers the real agent being deleted and also `possible_actions` being empty

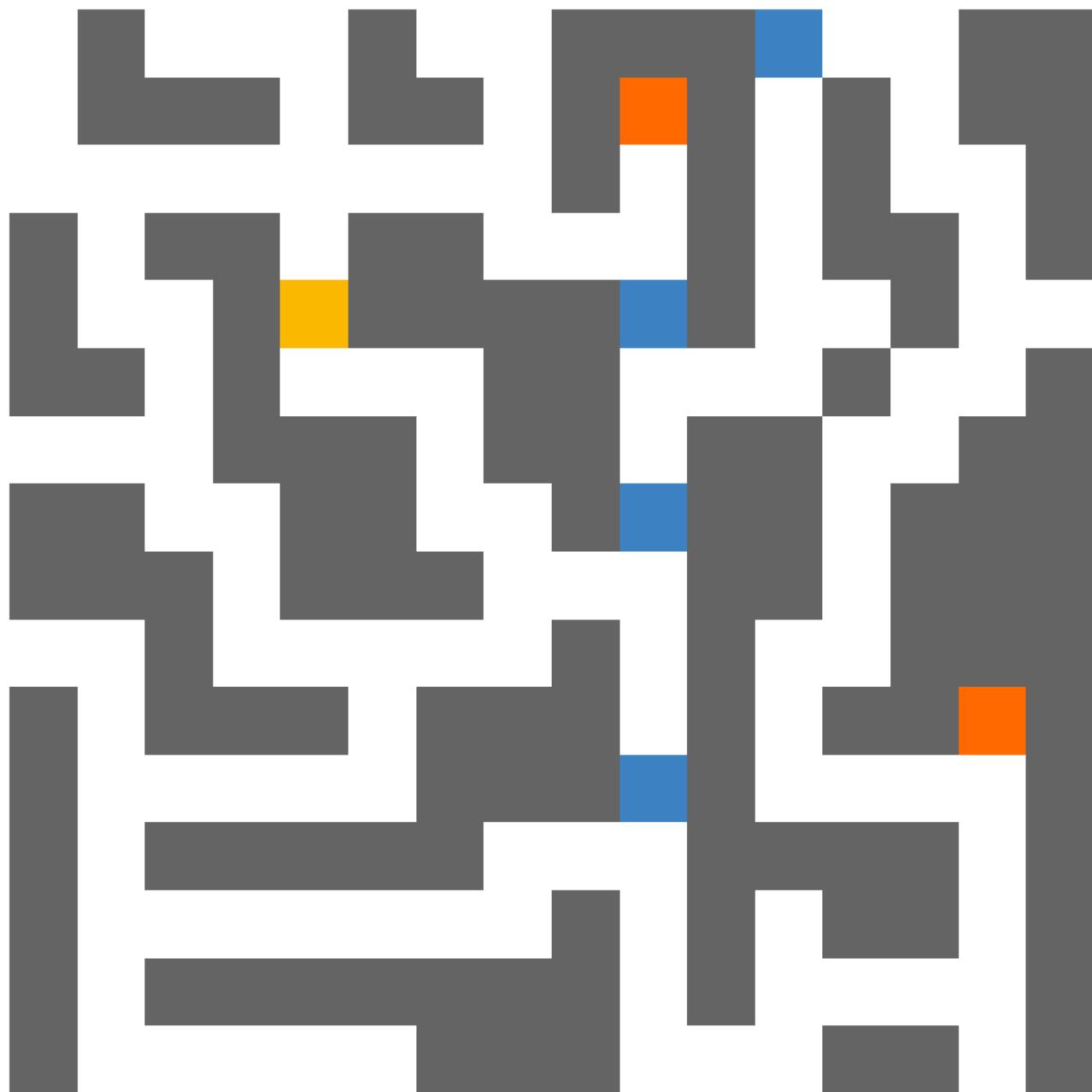
 Problem >











Is there a way where I can reuse the exact same map and place the agent in the exact same scenario?

I should probably design a seed that is always outputted and can be inputted into the Environment class where it always keep the exact same map with same agent and goal position

Okay this is worth the time in order to troubleshoot these rare problems.

This is how I'm adapting the `def choose_piece1()` in Environment's `generate()`  
Add an parameter to choose which piece in the corner

```
def choose_piece1(choose_index=None)
```

Have random number saved

```
index=random.randint(0, len(options)-1)
```

If `choose_index` is something, then set that index

```
if choose_index:  
    index=int(choose_index)
```

Save index to seed string

```
self.seed+=str(index)
```

Return

```
return options[index]
```

Going to copy and paste that for all of them

Okay now that we got the seed for the pieces I got to make sure we can use that seed to make the map

Okay got seed working by adding

```
if seed:  
    pieces={1:choose_piece1(seed[0]),2:choose_piece2(seed[1]),3:choose_piece3(seed[2]),4:choose_piece4(seed[3])}  
else:  
    pieces={1:choose_piece1(),2:choose_piece2(),3:choose_piece3(),4:choose_piece4()}
```

so now the dictionary's pieces are chosen based on the 4 first values of the `seed` string which gives the indices for the `choose_piece`'s option

Last 8 values will be the position for the agent and goal

Adding a seed parameter for

```
place_agent_and_goal(self,min_dist=17,max_dist=20,seed=None)
```

Got to add a position parameter for

```
place_agent(self,position=None)
```

and

```
place_goal(self,position=None)
```

Will add this like to `place_agent` and `place_goal`

```
if position:  
    self.state[int(position[0])][int(position[1])]=3  
    return self.state
```

Now in `place_agent_and_goal()` we have

```
if seed:  
    self.place_agent([seed[4:6],seed[6:8]])  
    agent_position=self.find_positions(True)  
else:  
    self.place_agent()  
    agent_position=self.find_positions(True)
```

the numbers will be padded like `01` and `12` in the seed string

```
Added return self.state

self.place_goal([seed[8:10],seed[10:12]])
goal_position=self.find_positions(True,True)
return self.state
```

to avoid the Manhattan distance check at the end of the function

```
if min_dist<self.M_distance(agent_position,goal_position)<max_dist:
    return self.state
```

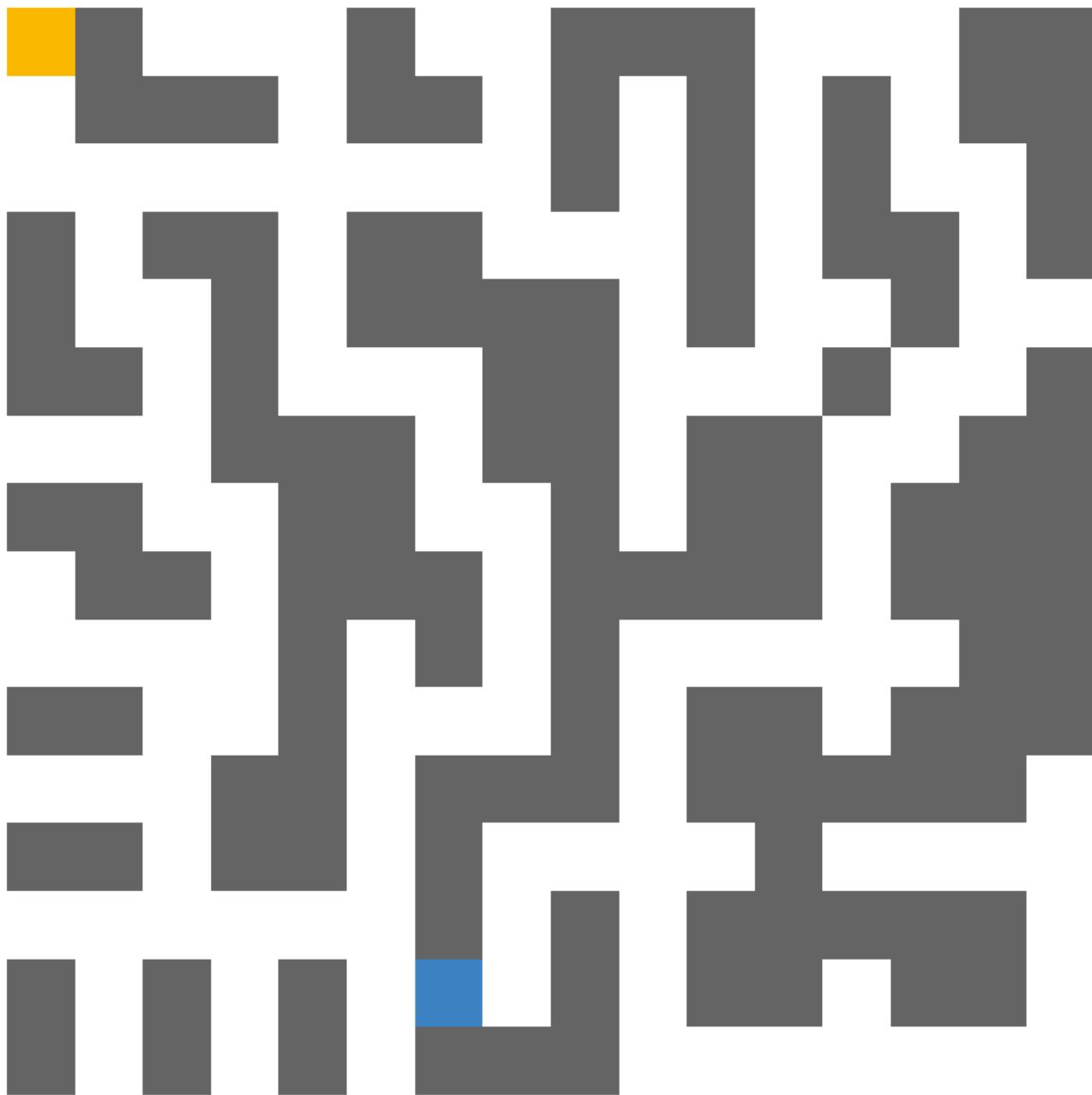
According to this [site](#), we can use `str().zfill(2)` to pad numbers. Awesome, I thought I would have to import a module

Added a section in `place_agent_and_goal()` that creates adds onto the seed first the agent position padded and then the goal

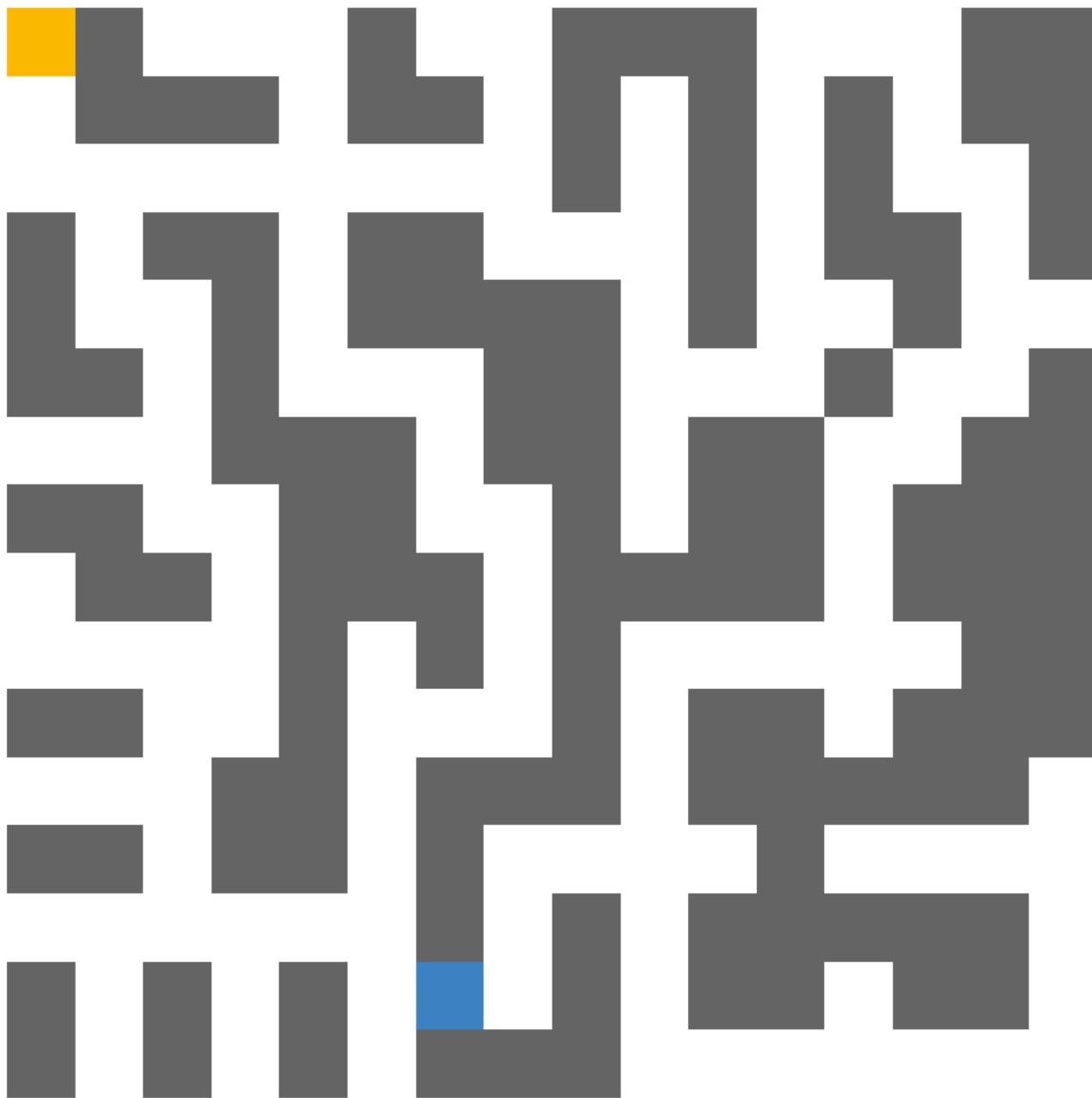
```
# Create seed for position of agent and goal
self.seed+=str(agent_position[0]).zfill(2)+str(agent_position[1]).zfill(2)
self.seed+=str(goal_position[0]).zfill(2)+str(goal_position[1]).zfill(2)
```

This is for when a seed is not given

Now the output seed given is `013212070000`. Let's see if it's the same when I input this string



It is!



And what's really cool is that this is the end of the state when `Agent` is done it's algorithm so that means it's repeating the exact same calculation. The lack of randomness being used is pretty cool

Awesome! It took around over an hour to code a seed feature so now I'll run the code on loop using my benchmark and wait for an error to pop up that I need to fix

Okay seed `003212140205` caused an error let's check it out. Got the `possible_actions` is empty error and the one that deletes the real agent

Wow it replicated the error perfectly this is awesome. Now if I can get this exact seed working I know I fixed the problem.

Oh wait before we do that I got to fix my seed error where when giving a seed it does not give the full seed

Seed `0032`

Fixed in `place_agent_and_goal` by adding `self.seed+=seed[4:12]`

```
if seed:  
    self.place_goal([seed[8:10], seed[10:12]])
```

```
goal_position=self.find_positions(True,True)

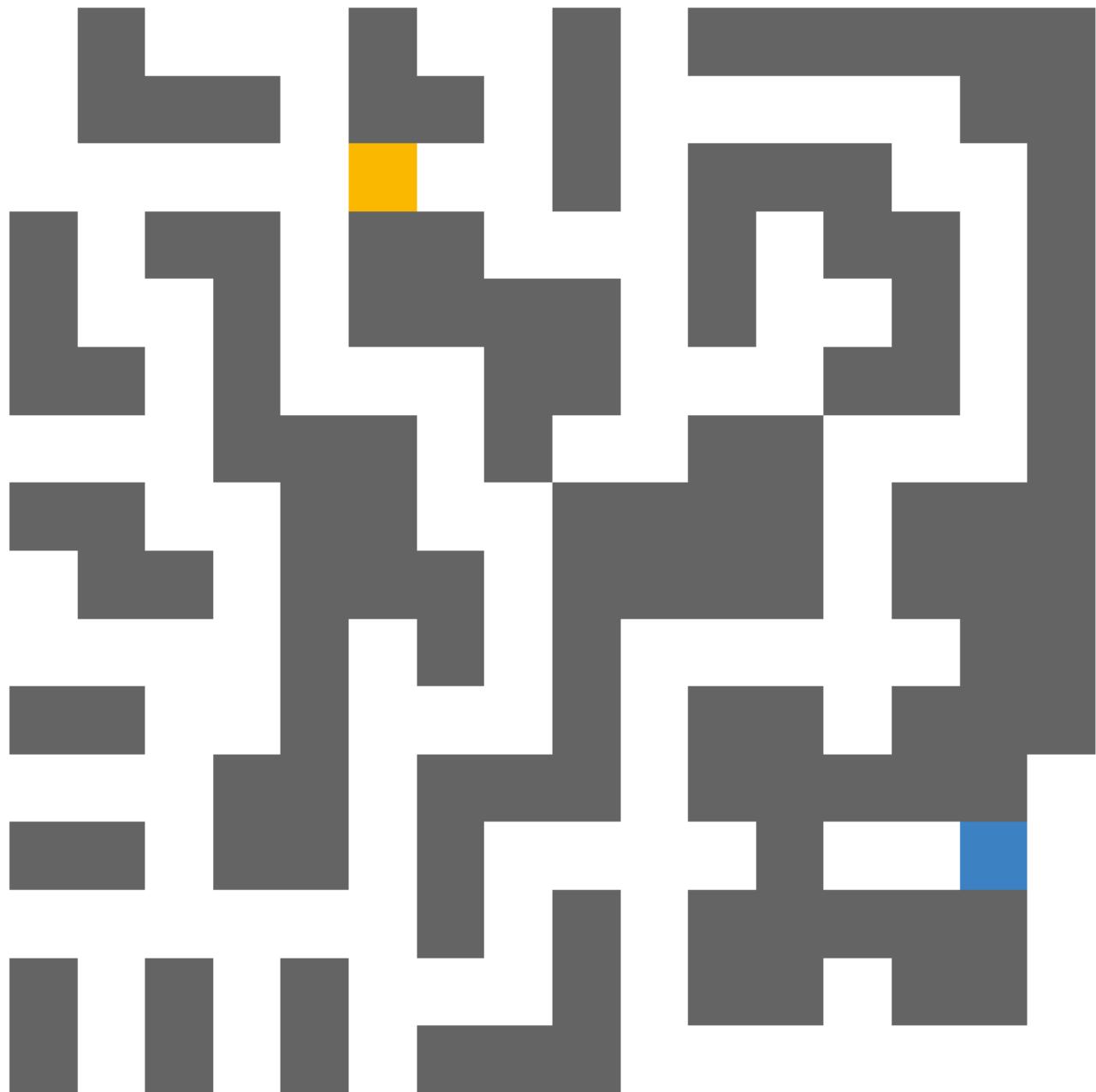
# Add positions to self.seed
self.seed+=seed[4:12]

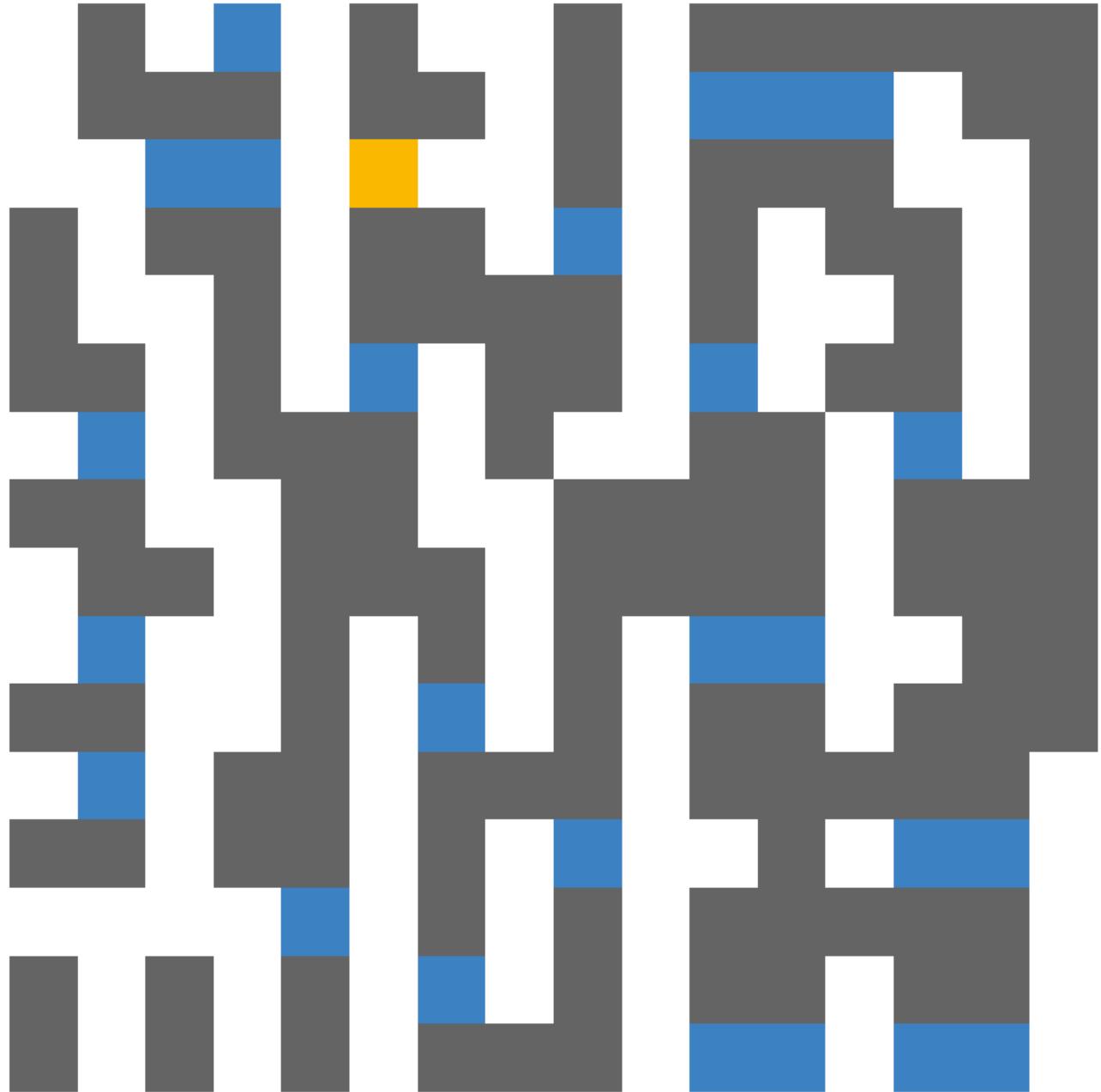
return self.state
```

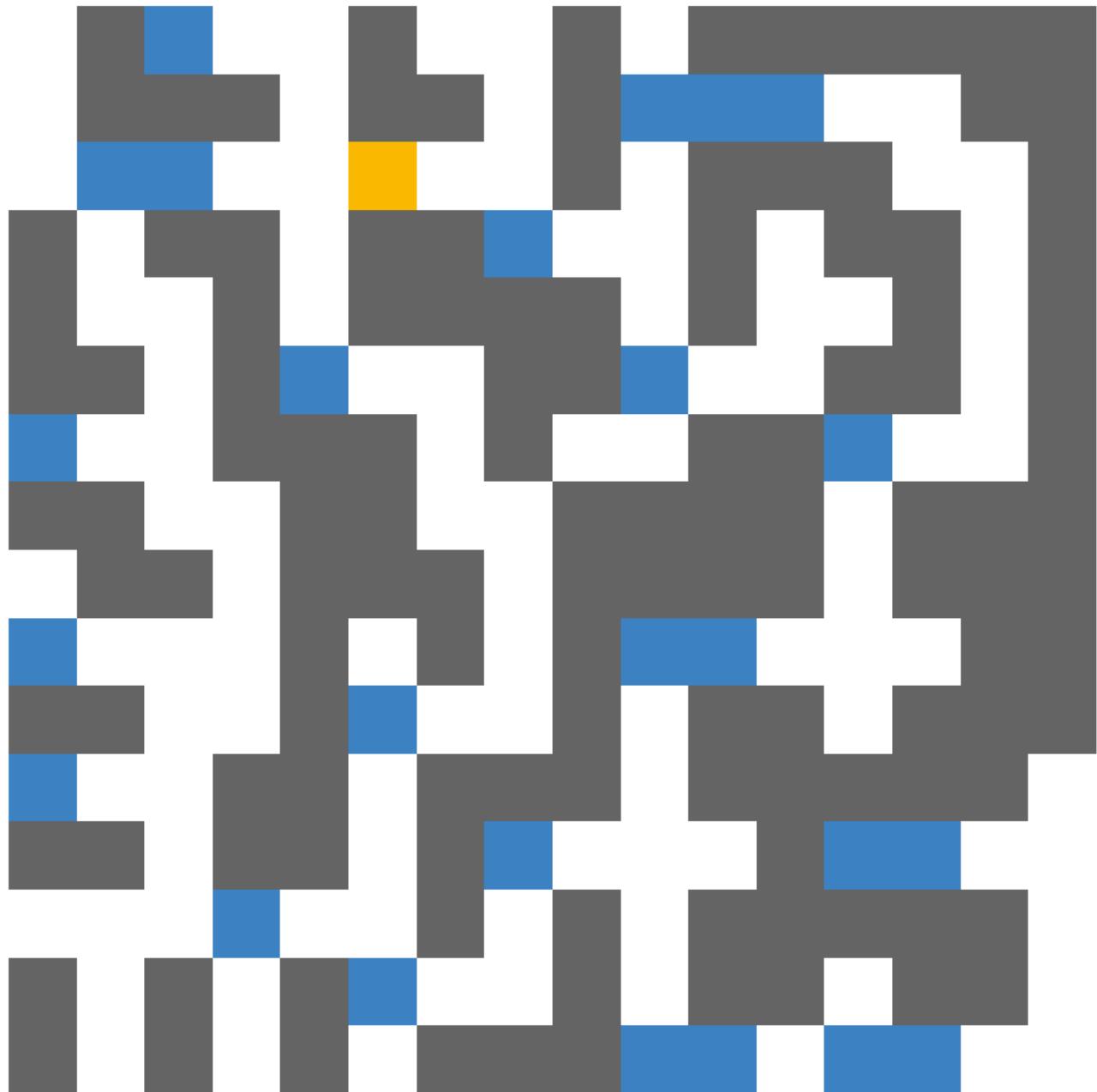
Ah it seems the `possible_actions` can't find any directions to go because the real agent's percept is `1111`

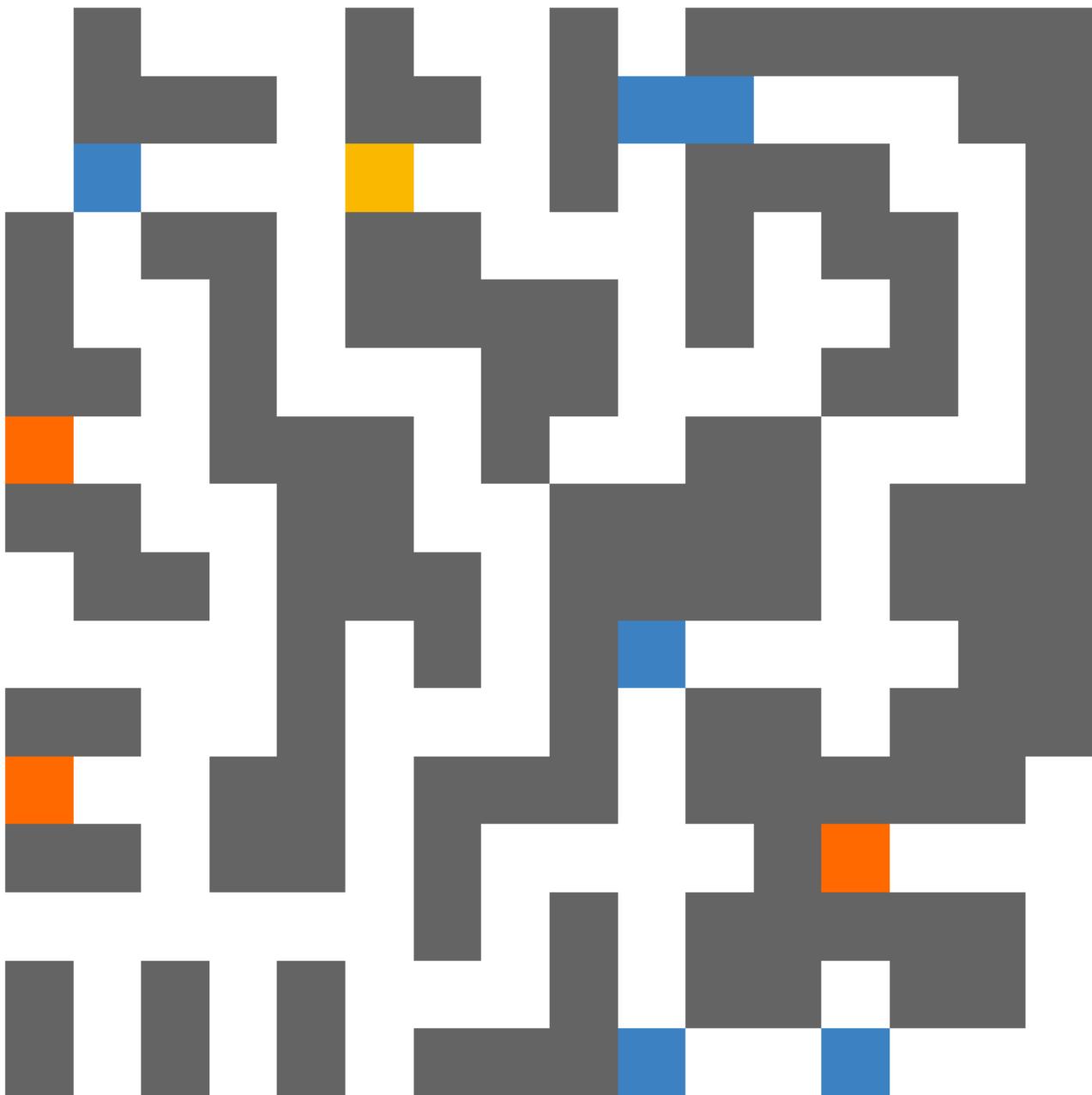
That's because it has another agent in front of it going to a dead end, and then it becomes a wall. Then something happens that makes the real agent think its surrounded by walls

 [Situation >](#)









Hm, it has been 10 minutes and I'm still not sure what causes the real agent to go in the wall. I know something is out of order somewhere. But I know it's not `self.dead_ends` placing the orange cones since it does as the last step after move. The problem happens at `choose_action()` which is early on in `run()`. And it all seems to happen at once so I don't think it will be caused after a full loop of `run()`

The real agent didn't even walk into the wall, it's just that it doesn't have any option. So it must be `scan()`

Okay so the reason why the percept is `1111` isn't because the real agent is surrounded by walls, it's because it was deleted. It no longer exists in the

Environment state



Okay so it turns out `single_out_agent()` starts off without the agent exists so it must be some order thing happening in `run()`

```
def single_out_agent(self):
    # Find first
    print("\npossible_actions", self.possible_actions())

results in possible_actions []
```

When does the real agent gets deleted?

Problem is the `run()` before the percept is `1111`, the percept is `1001` and the move is `w`. Since a orange wall will built `w` the percept should be `1011`

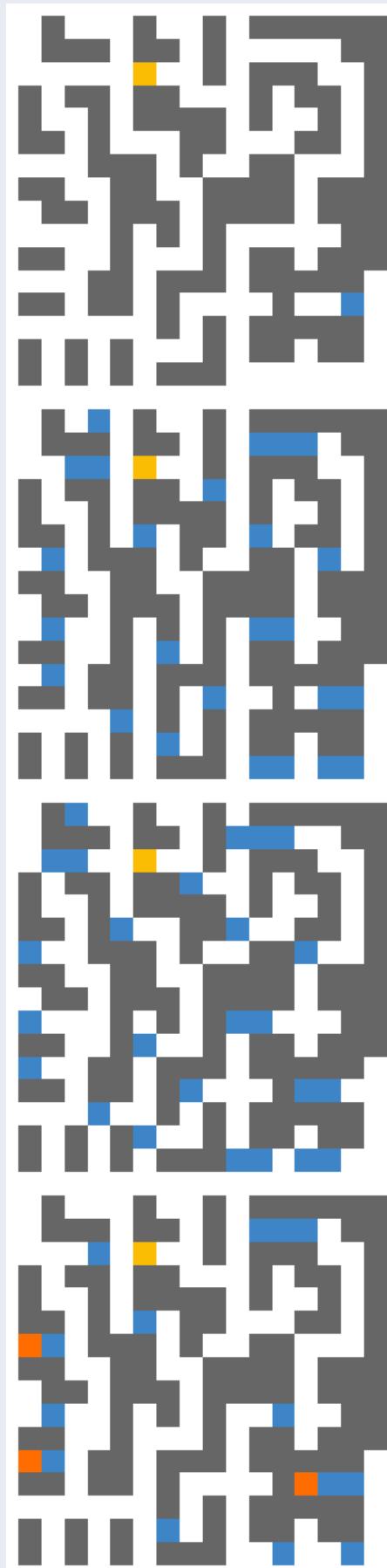
It does mean the real agent moved into a wall

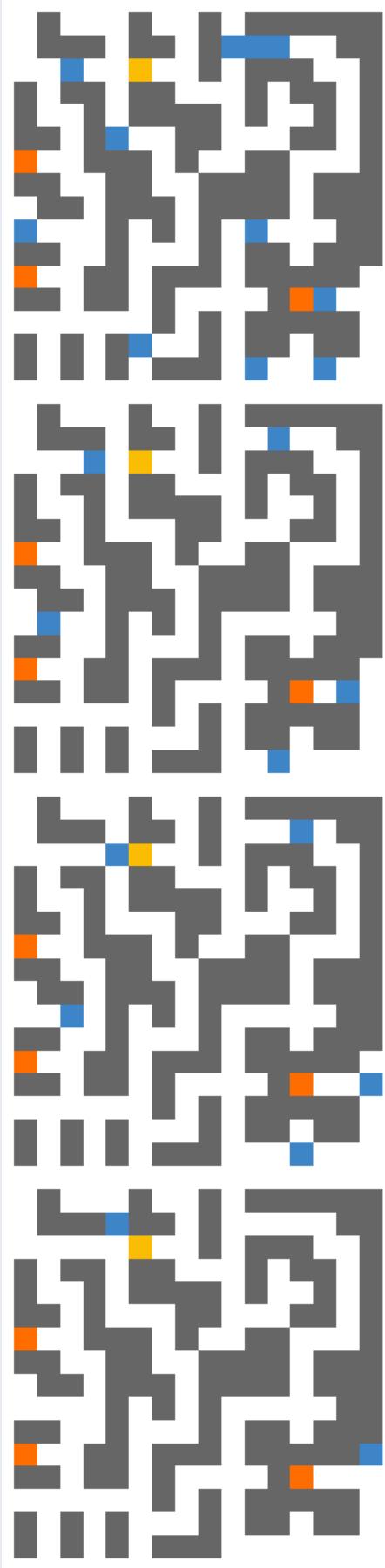
A solution would be to place the construction cones before choosing an action

Well I made the code super weird by trying to fix that, and I still get the error I'm trying to fix

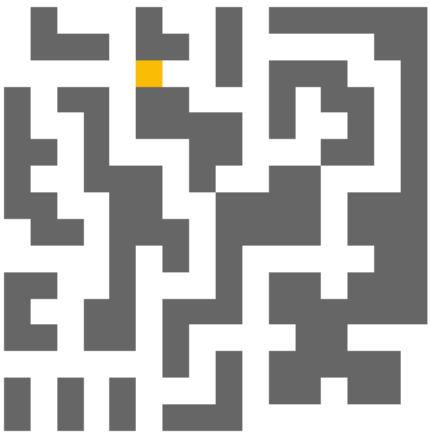
Okay I think the reason why the code is even worse is because if the real agent is in a dead end now it just gets deleted. But my solution fixed the previous problem. It's just now the real agent moved out of it's previous dead end just to die to another one. great

 Scenario >





Environment's view



And I know the real agent was replaced with a wall because in the console it says it found a dead end at the real agent's position

```
Dead end at [11, 15]
```

So far my situation is if I place the construction cones before the move is chosen, the code breaks, and as well for if I put it after the move is chosen. Hmm

What if I place during as the moves are being chosen? Is that even possible?

Okay checking and placing dead ends before the percept is given from the Environment class in `run()` seems to fix the real agent from dying. But now the satisfying wall hallway filling maneuver is broken

```
# Check for dead ends and place them
self.dead_ends_check()

# Place walls at dead ends
for position in self.dead_ends:
    self.state[position[0]][position[1]]=1

self.enviro.place_dead_ends(self.dead_ends)

# Get percept from Environment
percept=self.enviro.percept()
```

Oh and when I use it for any other seed it instantly breaks the code ahhhhh since as soon as the agent is in a dead end it is eliminated

Okay whatever I do I cannot place the dead ends before the move, but I can know where they are though

Creating a new function call `place\_dead\_ends()'

Okay having `dead_end_check()` before the percept, and `place_dead_ends()` after `move()` seems to work perfectly. Well, except for the same problem I'm trying to solve. I need to code another check for that. I cannot think of another way

Wait, since I know where the dead ends are before I'll move, I'll just have a check that if a direction leads to a dead end, then don't go there. Simple! Just got to find where to put that check

I think I got to code the check in `possible_actions()`

Wait I shouldn't because that's not generalizing the function enough. I got to put the check only where needed

I'll make a new function that checks this very specific thing and try to use it in different places

Easy code

```
def action_to_dead_end_check(self, action, position):
    if self.action_to_position(action, position) in self.dead_ends:
        return True
    else:
        return False
```

Thought by adding a dead end check for each position's move it does in `move()` it would fix it but it didn't so I need to make sure my function works

```
for position in self.positions:

    # Get position of Agent
    i=position[0]
    j=position[1]

    # If action is North, verify with transition model
    if self.action=="N" and self.transition_model("N",position) and not self.action_to_dead_end_check("N", position):

        # Place Agent one step North
        new_positions.append([i-1,j])

    # If action is East
    if self.action=="E" and self.transition_model("E",position) and not self.action_to_dead_end_check("E", position):
        new_positions.append([i,j+1])

    # If action is West
    if self.action=="W" and self.transition_model("W",position) and not self.action_to_dead_end_check("W", position):
        new_positions.append([i,j-1])

    # If action is South
    if self.action=="S" and self.transition_model("S",position) and not self.action_to_dead_end_check("S", position):
        new_positions.append([i+1,j])
```

Okay verified that action to dead end function does work.

```
print("Action to dead end check",self.action_to_dead_end_check(action, [12,13]))
```

That's the position of where the real agent is before jumping into the dead end. I have to make sure it knows a dead end if coming so it removes `W` from its action. But what's impossible about this is how from only a percept or actions, you cannot know the position. At least you know the position of the mini agent, but not the real agent. So you can protect the mini agent from going into a dead end, but not the real agent.

That's why I want to check for all positions in `move()` but somehow the function isn't working

Maybe I can check all positions with the same percept as the real one, and if one of them is going in a dead end then don't go there.

Wow okay I think I fixed it. The idea I just mentioned seems to work

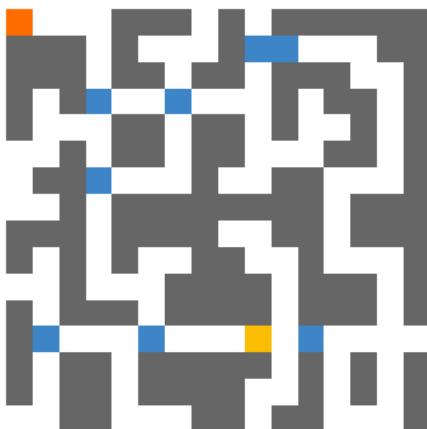
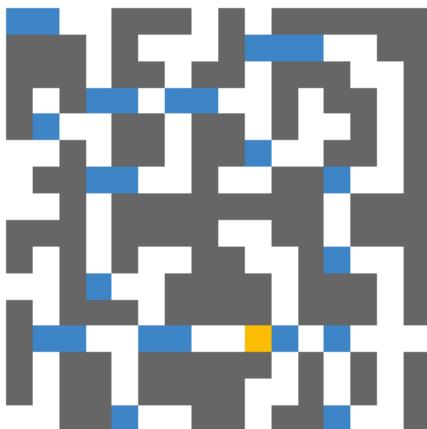
```
# Rechecking all positions to remove an action that leads to a dead end
for position in positions:
    # Positions with same percept as real agent
    if self.scan(position)==self.enviro.percept:
        # For each possible action from real agent
        for action in possible_actions_percept:
            # Find ones that lead to dead end
            if self.action_to_dead_end_check(action, position):
                # Remove it from possible actions of the real agent
                possible_actions_percept.remove(action)
```

Feels like a temporary solution though, as in, probably will cause problems later in the code. I'll test with random seeds

What I don't like about my solution is that it ruins the whole point of construction cones because now it won't move because there's a construction cone there, only because it knows it will be there. So it's like there's already construction cones all over the map. Which I guess is more efficient but defeats the point of me spending so much time on the idea construction cones

No way the exact same problem exists with seed 302400021209 . That's insanity

(top corner)



Honestly I think the solution is to do the dead end check only if the agent is next to another agent since that's the common theme between this problem. Kinda a cheap way to code it but wow this problem is haunting me

Well here is the code that solved the problem for this very specific seed. I'll test it on the seed from before

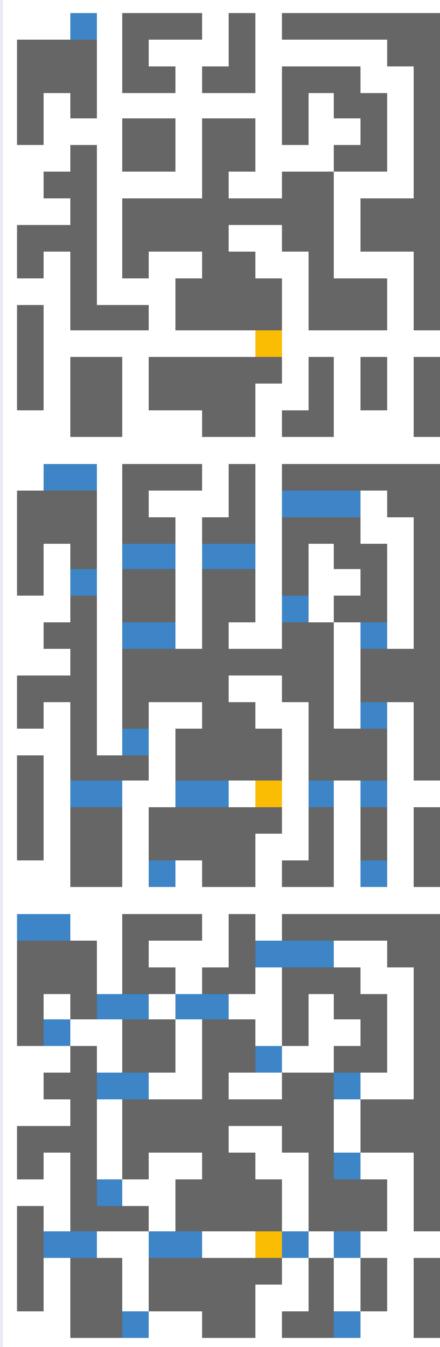
```
for position in positions:
    # If position has a chance to be the real agent
    if self.scan(position).replace("2","0") == self.enviro.percept():
        # And there's another agent next to it
        if "2" in self.scan(position):
            try:
                print(self.scan(position), self.enviro.percept())
                # Find direction of where agent is
                if self.scan(position)[0] == "2":
                    # Remove direction
                    possible_actions_percept.remove("N")
                if self.scan(position)[1] == "2":
                    possible_actions_percept.remove("E")
                if self.scan(position)[2] == "2":
                    possible_actions_percept.remove("W")
                if self.scan(position)[3] == "2":
                    possible_actions_percept.remove("S")
            except KeyError:
                pass
```

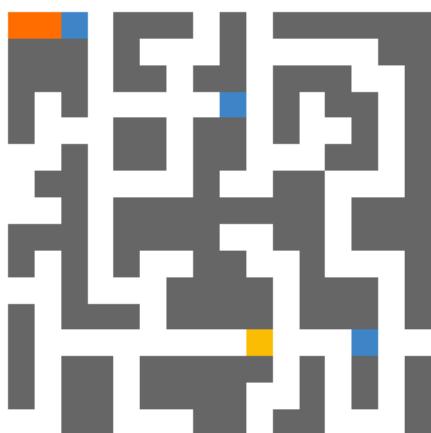
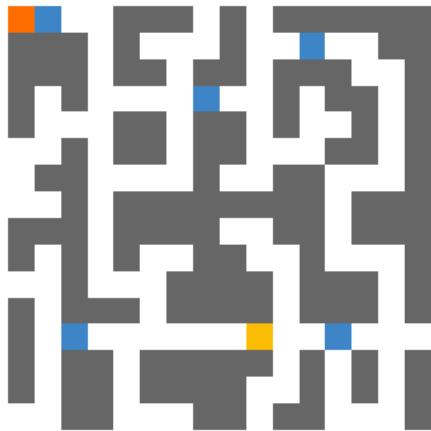
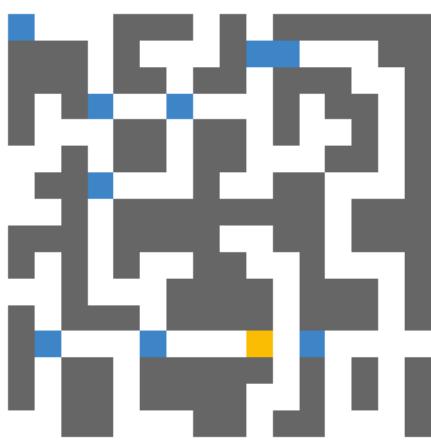
Nope it breaks the old seed 003212140205 . So cursed

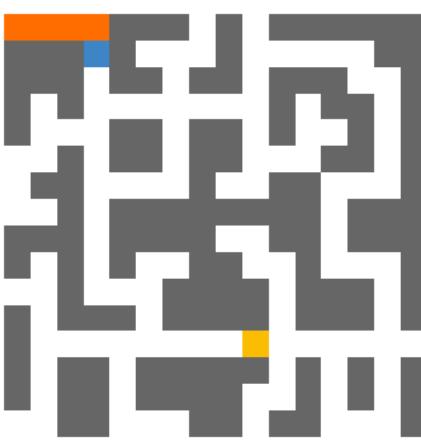
I'll give it one more last third try and then I'll probably have to remove this whole cone idea because it's using way too much time. Or maybe only start using it when there's one agent.

Holy crap I think I fixed it with a single line there's no way.

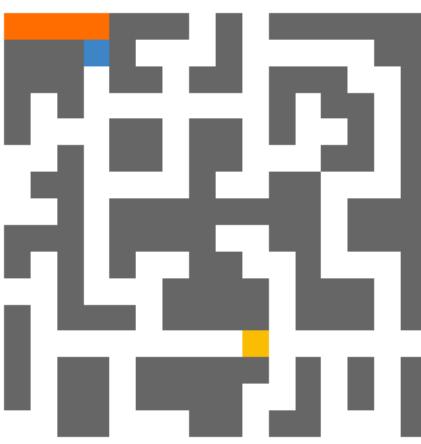
[✉ Maybe fixed!?](#) >







Environment's view



I was like hmm, what I think of the situation in the view of `dead_ends_check`. I knew the problem happens when there's another agent next to a future dead end



So I added this line where if there's a agent next to the dead end, then don't count it as a dead end

```
# Make sure there's not another agent next to it
if "2" not in self.scan(position):

in

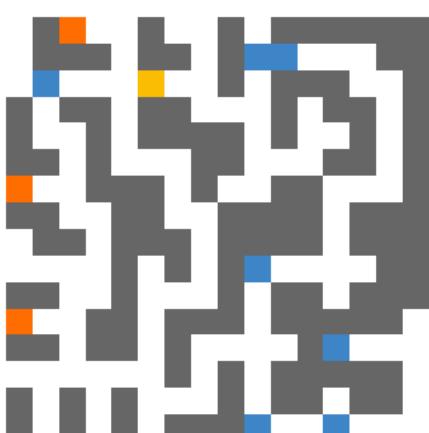
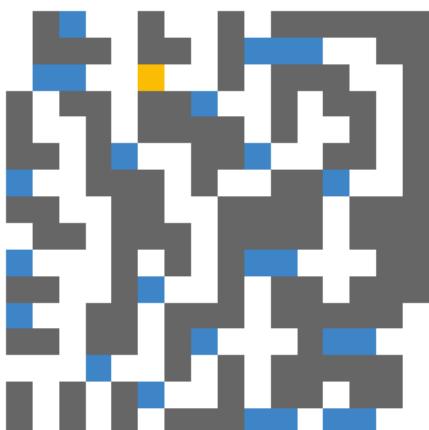
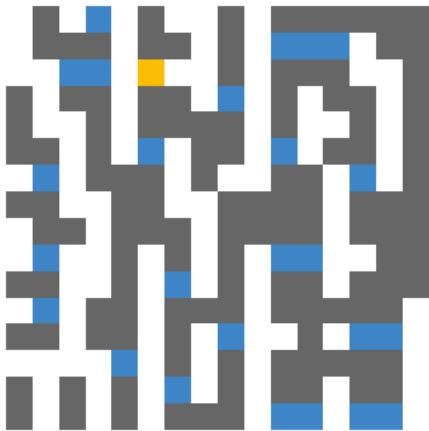
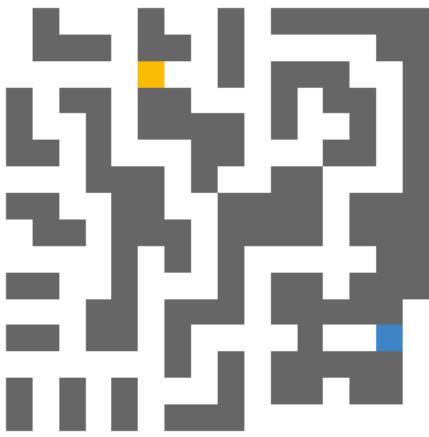
def dead_ends_check(self):
    # Check positions that are in a dead end
    for position in self.positions:
        if len(self.possible_actions(position))==1 and position!=self.goal:

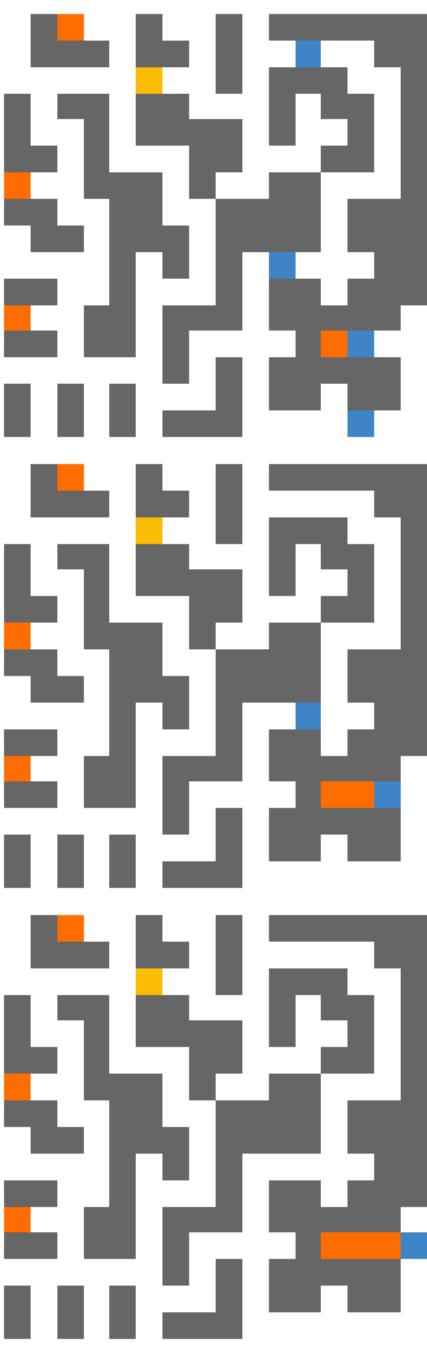
            # Make sure there's not another agent next to it
            if "2" not in self.scan(position):

                # Create wall at dead end
                print("Dead end at",position)
                self.dead_ends.append(position)
```

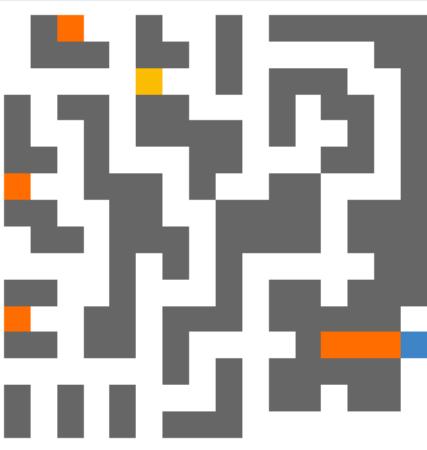
This better work with the other old seed 003212140205 so I can go to sleep happy

I'm actually the definition of never give up tonight





Environment's view (if it's the same as the last frame then it means no errors)



Wow, all of my 4 hours of troubleshooting and work to solve this specific scenario was solved by a single line of code. Code is wacky sometimes. I'm going to bed, well deserved good night

```
# My most loved and hated line of code I have written  
if "2" not in self.scan(position):
```

Okay actually before going to bed I found a scenario where there's the infinite loop. I'm sure this problem won't take too long tomorrow hahahaha 🤪  
Seed 100106130001

I literally had dreams of my brain's visualization of code. There were was no plot or objects, just instructions and code. A dream that only happens after coding for 4 hours before sleeping

Uhm the infinite loop problem from the seed I saved last night is gone. I guess it solved itself? I'll run another benchmark looking for problems

Found another seed loop with a problem, and yeah first time working code let's gooo. I decided to [challenge myself](#) by not relying on randomness, so this is what I got

```
# Check for possibility of loop
if len(positions_hcost)==2:
    # If both values are the same
    if list(positions_hcost.values())[0]==list(positions_hcost.values())[1]:
        # If past 4 moves consists of only 2 different direction
        if len(set(self.data["Move"].iloc[-4:]))==2:
            # Remove the second last move from this option
            print("Loop detected between moves {} and {}, forbidding
{}".format(self.data["Move"].iloc[-1],self.data["Move"].iloc[-2],self.data["Move"].iloc[-2]))
            del positions_hcost[self.data["Move"].iloc[-2]]
```

How the code works is if `positions_hcost` has two options, and they both contain the same cost, then check if the past 4 moves both contain only two unique actions. If so, then you are probably in a loop. Remove the action from the second last move (since if the loop pattern continues, we can predict this current `positions_hcost` is the same as the second last `positions_hcost`). Then we remove the second last move.

Going to run a benchmark test and see if there's still sessions that costed 100 moves.

It doesn't with seed 223004121202

Oh it gave a false alarm because it found a situation where the the agent kept moving `W` for 3 moves straight after moving `E`. And then after that, `N` and `E` had the same `hcost`. So it couldn't remove `W`

```
2      E
3      W
4      W
5      W

hcost dict {'N': 34, 'E': 34}
```

```
Loop detected between moves W and W, forbidding W
```

I got to make sure it's a back and forth pattern

Add this line of code I hope for the best

```
# Make sure it's a loop pattern
if self.data["Move"].iloc[-1]==self.data["Move"].iloc[-3] and self.data["Move"].iloc[-2]==self.data["Move"].iloc[-4]:
```

Oh it worked okay nice, let's see if it works for the first looped seed.

I forgot to write down the first seed I found today damn. I guess I'll run more benchmarks and if it works consistently

Got an out of range error for seed 242015140505 . I'll just add another check to make sure the DataFrame is long enough

Okay this check fixed it

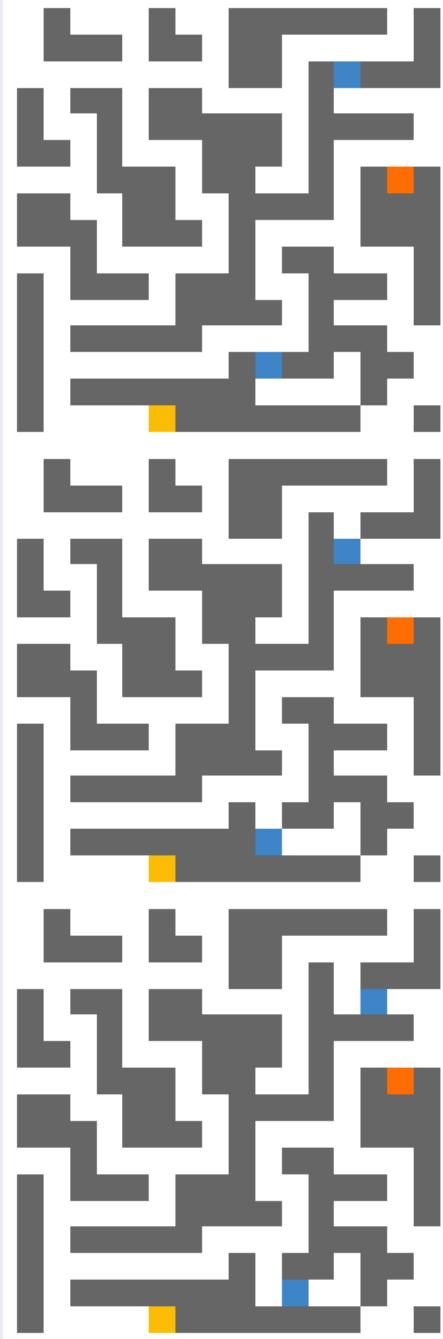
```
# Make sure dataframe is at least 4 moves long
if len(self.data)>4:
```

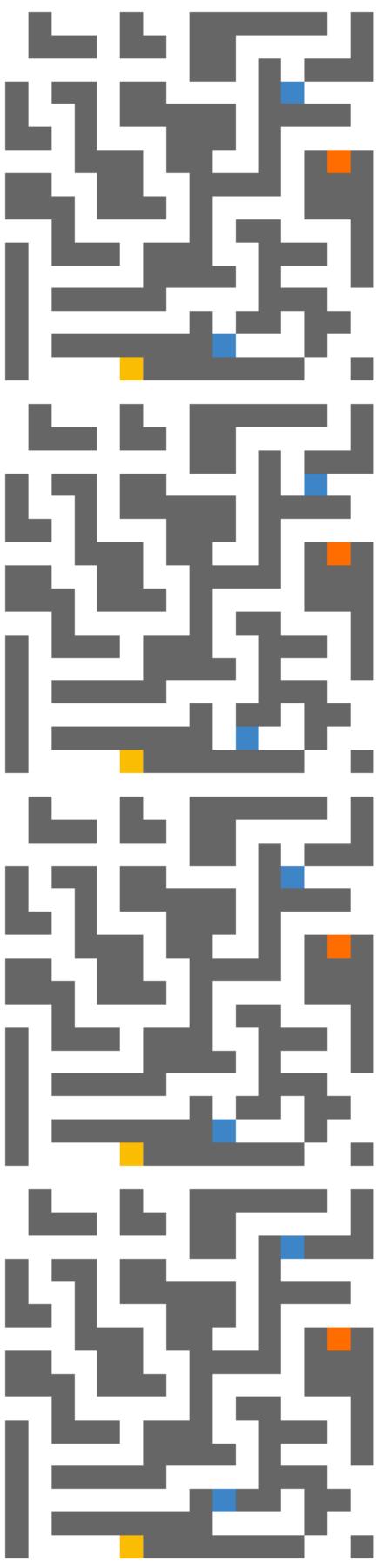
Running another benchmark

Nooo wayyy there's a loop out of 3 moves?!?! WHYYY

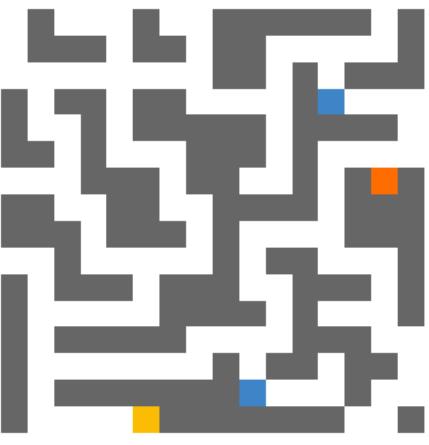
Seed 041001101505

[Three move loop >](#)





## Returns and reloops



And it's not just a simple three move loop, it has a complex pattern

What if I just check for if a list of Positions is the same as before, then remove the move that was used last time. This would work for any complex pattern quickly and efficiently. All it would need to take is at least 3 moves.

Index	Belief state	Positions	Percept	Move
4	<code>[[0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0,... [[2, 12], [13, 9]]</code>		0110	S
5	<code>[[0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0,... [[3, 12], [14, 9]]</code>		0011	E
6	<code>[[0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0,... [[3, 13], [14, 10]]</code>		1001	W
7	<code>[[0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0,... [[3, 12], [14, 9]]</code>		0011	E
8	<code>[[0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0,... [[3, 13], [14, 10]]</code>		1001	W
9	<code>[[0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0,... [[3, 12], [14, 9]]</code>		0011	N
10	<code>[[0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0,... [[2, 12], [13, 9]]</code>		0110	S
11	<code>[[0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0,... [[3, 12], [14, 9]]</code>		0011	E
12	<code>[[0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0,... [[3, 13], [14, 10]]</code>		1001	W
13	<code>[[0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0,... [[3, 12], [14, 9]]</code>		0011	E
14	<code>[[0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0,... [[3, 13], [14, 10]]</code>		1001	W
15	<code>[[0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0,... [[3, 12], [14, 9]]</code>		0011	N
16	<code>[[0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0,... [[2, 12], [13, 9]]</code>		0110	S
17	<code>[[0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0,... [[3, 12], [14, 9]]</code>		0011	E
18	<code>[[0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0,... [[3, 13], [14, 10]]</code>		1001	W
19	<code>[[0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0,... [[3, 12], [14, 9]]</code>		0011	E
20	<code>[[0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0,... [[3, 13], [14, 10]]</code>		1001	W
21	<code>[[0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0,... [[3, 12], [14, 9]]</code>		0011	N

Made a simple loop check based on my last idea

```
def loop_check(self):
    for i,positions in enumerate(list(self.data["Positions"])):
        if self.positions==positions:
            print("Loop detected, forbid move",self.data["Move"].iloc[i])
            return self.data["Move"].iloc[i]
    return None
```

Okay it seemed to work when I used it like this

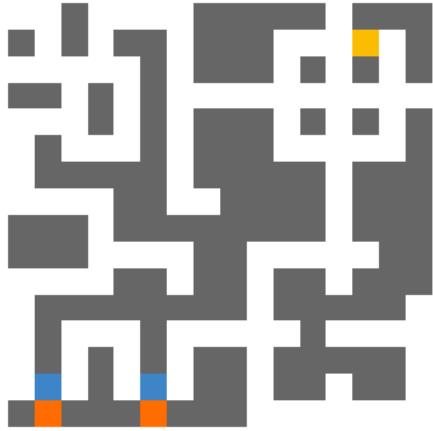
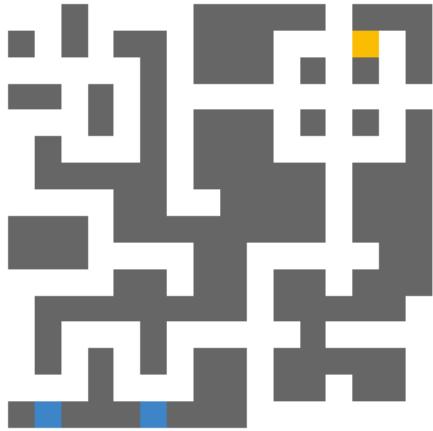
```
if self.loop_check():
    del positions_hcost[self.loop_check()]
```

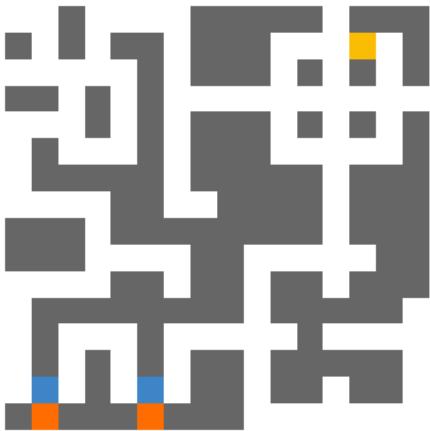
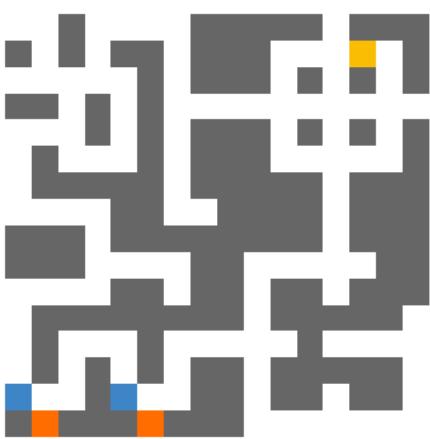
Will test with old seed that had a loop

Perfect worked with 223004121202 . Now will run benchmark 1000 times

Seed 424212060113 with an error

Ah cause it's was in a loop that would have gone S , but a construction cone blocked it





So that means I just got to remove the action if it's possible

```
if self.loop_check():
    try:
        del positions_hcost[self.loop_check()]
    except KeyError:
        pass
```

This fixed it

Another benchmark once more

 **Pro Tip: Try to generalize the problem as much as possible when problem solving**

Ayy the results are in and the three problems I had to deal with since yesterday seem to be gone! Finally can start on making the meat of the project

The average moves after 1000 iterations is:

4.68

```
Moves [4, 5, 4, 6, 6, 4, 4, 4, 8, 2, 11, 5, 4, 4, 5, 5, 5, 7, 5, 3, 5, 4, 3, 4, 6, 5, 4, 4, 4, 3, 6, 4, 3, 5, 5, 8, 3, 4, 5, 8, 5,
4, 6, 4, 4, 1, 6, 6, 6, 4, 5, 5, 4, 2, 5, 4, 8, 8, 6, 6, 3, 7, 8, 5, 5, 3, 5, 5, 3, 7, 9, 3, 7, 7, 3, 3, 5, 6, 5, 6, 2, 3, 2, 3,
6, 3, 6, 5, 6, 4, 4, 4, 4, 4, 6, 6, 6, 4, 5, 5, 4, 5, 5, 7, 10, 4, 4, 4, 5, 3, 6, 8, 7, 4, 5, 3, 4, 6, 5, 4, 3, 7, 4, 4, 5, 6, 5, 6, 5, 4,
4, 3, 5, 3, 3, 5, 5, 8, 5, 3, 6, 8, 4, 4, 4, 5, 3, 5, 5, 3, 3, 8, 3, 5, 6, 6, 5, 9, 10, 8, 3, 5, 4, 4, 8, 3, 4, 5, 4, 5, 6, 4, 3, 4,
6, 3, 5, 5, 6, 6, 5, 6, 4, 4, 5, 4, 5, 5, 4, 6, 4, 6, 3, 3, 5, 4, 7, 3, 3, 6, 5, 5, 5, 3, 8, 3, 6, 4, 5, 6, 5, 7, 7, 5, 7,
4, 5, 2, 5, 4, 5, 5, 2, 3, 2, 3, 4, 5, 3, 3, 3, 5, 2, 3, 3, 7, 4, 4, 4, 6, 5, 4, 3, 3, 4, 2, 5, 3, 3, 5, 2, 5, 6, 3, 3, 5,
4, 3, 3, 6, 6, 5, 4, 6, 6, 3, 6, 5, 5, 3, 6, 4, 3, 9, 7, 4, 5, 3, 6, 5, 9, 5, 5, 3, 5, 9, 5, 4, 4, 6, 3, 6, 2, 4, 5, 3, 5, 3, 6,
6, 5, 4, 6, 8, 2, 3, 4, 5, 7, 4, 5, 4, 6, 5, 5, 3, 5, 8, 4, 2, 8, 5, 5, 5, 2, 4, 5, 6, 3, 3, 2, 4, 2, 8, 4, 5, 5, 3, 4, 5,
2, 6, 4, 4, 5, 4, 4, 4, 6, 5, 4, 3, 2, 2, 3, 5, 3, 5, 2, 5, 6, 4, 5, 4, 5, 4, 3, 5, 7, 4, 4, 4, 6, 5, 3, 3, 7, 4, 4, 6, 6, 3, 6,
8, 4, 4, 5, 3, 4, 5, 5, 5, 3, 5, 8, 4, 4, 3, 5, 4, 4, 5, 7, 3, 2, 7, 8, 6, 6, 3, 4, 6, 3, 5, 5, 7, 2, 8, 8, 3, 3, 3, 8, 6, 5,
5, 5, 4, 5, 4, 5, 6, 4, 8, 8, 4, 5, 3, 5, 5, 3, 6, 5, 5, 5, 7, 3, 3, 4, 7, 4, 4, 6, 4, 6, 4, 4, 8, 4, 5, 5, 8, 4, 4, 5, 4, 4, 5,
5, 4, 5, 4, 6, 6, 5, 7, 3, 3, 5, 5, 6, 6, 5, 5, 4, 5, 5, 3, 2, 4, 4, 4, 2, 6, 3, 3, 3, 8, 5, 4, 5, 3, 2, 5, 8, 3, 4, 7, 5, 8, 3,
5, 4, 7, 4, 6, 5, 5, 4, 3, 4, 1, 8, 2, 3, 8, 3, 6, 5, 4, 5, 3, 6, 6, 3, 5, 5, 5, 2, 6, 6, 5, 4, 7, 4, 4, 2, 5, 4, 5, 5, 8, 5, 4,
5, 10, 4, 4, 5, 2, 4, 6, 3, 5, 5, 10, 5, 4, 5, 8, 5, 5, 5, 4, 6, 3, 6, 7, 8, 4, 5, 4, 5, 5, 5, 9, 6, 3, 3, 4, 7, 8, 5, 3,
```

5, 7, 5, 9, 3, 5, 4, 8, 4, 6, 5, 5, 6, 5, 9, 3, 7, 3, 5, 5, 3, 5, 7, 2, 2, 5, 3, 6, 2, 5, 8, 6, 5, 4, 3, 6, 3, 4, 5, 5, 3, 5, 4,  
 7, 5, 5, 4, 5, 4, 4, 8, 3, 6, 5, 5, 5, 5, 3, 3, 5, 4, 5, 4, 5, 10, 3, 3, 3, 6, 3, 5, 5, 3, 8, 4, 6, 5, 3, 2, 2, 5, 4, 4, 3, 5, 4,  
 5, 4, 5, 4, 3, 5, 6, 3, 4, 6, 4, 7, 3, 6, 5, 4, 4, 3, 6, 5, 6, 4, 5, 6, 4, 3, 5, 4, 3, 5, 6, 5, 4, 6, 4, 3, 4, 2, 4, 4, 3, 4, 4,  
 4, 5, 2, 6, 4, 4, 3, 3, 3, 4, 5, 4, 4, 4, 7, 4, 5, 5, 7, 5, 2, 7, 3, 3, 5, 5, 8, 6, 4, 4, 4, 3, 4, 4, 6, 3, 4, 5, 3, 4, 5, 4, 4,  
 4, 3, 5, 5, 4, 7, 4, 3, 4, 4, 6, 5, 5, 2, 5, 3, 6, 8, 4, 6, 5, 4, 4, 8, 3, 3, 6, 4, 5, 5, 6, 3, 3, 6, 6, 4, 4, 3, 5, 6, 3,  
 6, 6, 4, 7, 4, 4, 5, 4, 5, 4, 3, 3, 12, 3, 5, 4, 4, 5, 6, 4, 3, 3, 4, 4, 5, 8, 4, 6, 2, 6, 3, 6, 5, 4, 3, 4, 3, 6, 3, 3, 2, 4, 10,  
 3, 4, 6, 6, 6, 6, 6, 3, 6, 5, 5, 2, 6, 4, 4, 4, 4, 6, 5, 7, 5, 4, 5, 2, 3, 5, 2, 5, 4, 4, 4, 4, 6, 4, 4, 6, 5, 3, 3, 6, 5, 6, 5,  
 4, 5, 3, 3, 3, 4, 4, 4, 4, 4, 5, 7, 4, 6, 5, 7, 4, 3, 5, 4, 4, 4, 5, 3, 6, 7, 6, 5, 5, 3, 7, 5, 5, 5, 2, 6, 5, 8, 3, 5, 3, 4, 5,  
 3, 4, 3, 6, 4, 4, 5, 4, 4, 4, 5, 4, 1, 4, 8, 2, 5, 3, 6, 6, 5, 2, 4, 4, 6, 4, 7, 5, 5, 4, 5, 2, 2, 4, 5, 4, 1, 4, 5, 5, 2, 4, 4,  
 5, 4, 4, 6, 6, 6, 3, 6, 3, 6, 7, 3, 6]

It's funny cause the problems I have solved all came down to these two small pieces of code

```
if "2" not in self.scan(position):
```

and

```
def loop_check(self):
    for i,positions in enumerate(list(self.data["Positions"])):
        if self.positions==positions:
            return self.data["Move"].iloc[i]
    return None
```

And it was all done without using any randomness which I'm quite proud of.

Made the print much nicer

```
Seed  
141301121104  
-----  
  
single_agent  
1001 {'1000'}  
W
```

```
Dead end at [2, 0]
single_agent {'W', 'E'} {'W', 'E'} [1, 11]
1010 {'1000'}
W

single_agent {'E', 'S'} {'E', 'S'} [1, 10]
0110 {'1000', '1001'}
S

single_agent {'S', 'N'} {'S', 'N'} [2, 10]
0100 {'1000', '1010', '1001'}
S

positions_hcost {'W': 13, 'S': 13, 'N': 15}
S
```

-----  
Seed  
141301121104

Greedy best-first is in Chapter 3.5.1, best-first search ( $A^*$ ) in Chapter 3.5.2.

### 3.5 INFORMED (HEURISTIC) SEARCH STRATEGIES

**INFORMED SEARCH** This section shows how an **informed search** strategy—one that uses problem-specific knowledge beyond the definition of the problem itself—can find solutions more efficiently than can an uninformed strategy.

**BEST-FIRST SEARCH** The general approach we consider is called **best-first search**. Best-first search is an instance of the general TREE-SEARCH or GRAPH-SEARCH algorithm in which a node is selected for expansion based on an **evaluation function**,  $f(n)$ . The evaluation function is construed as a cost estimate, so the node with the *lowest* evaluation is expanded first. The implementation of best-first graph search is identical to that for uniform-cost search (Figure 3.14), except for the use of  $f$  instead of  $g$  to order the priority queue.

**EVALUATION FUNCTION** The choice of  $f$  determines the search strategy. (For example, as Exercise 3.22 shows, best-first tree search includes depth-first search as a special case.) Most best-first algorithms include as a component of  $f$  a **heuristic function**, denoted  $h(n)$ :

$h(n)$  = estimated cost of the cheapest path from the state at node  $n$  to a goal state.

(Notice that  $h(n)$  takes a *node* as input, but, unlike  $g(n)$ , it depends only on the *state* at that node.) For example, in Romania, one might estimate the cost of the cheapest path from Arad to Bucharest via the straight-line distance from Arad to Bucharest.

**HEURISTIC FUNCTION** Heuristic functions are the most common form in which additional knowledge of the problem is imparted to the search algorithm. We study heuristics in more depth in Section 3.6. For now, we consider them to be arbitrary, nonnegative, problem-specific functions, with one constraint: if  $n$  is a goal node, then  $h(n) = 0$ . The remainder of this section covers two ways to use heuristic information to guide search.

#### 3.5.1 Greedy best-first search

**GREEDY BEST-FIRST SEARCH** **Greedy best-first search**<sup>8</sup> tries to expand the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly. Thus, it evaluates nodes by using just the heuristic function; that is,  $f(n) = h(n)$ .

**STRAIGHT-LINE DISTANCE** Let us see how this works for route-finding problems in Romania; we use the **straight-line distance** heuristic, which we will call  $h_{SLD}$ . If the goal is Bucharest, we need to know the straight-line distances to Bucharest, which are shown in Figure 3.22. For example,  $h_{SLD}(In(Arad)) = 366$ . Notice that the values of  $h_{SLD}$  cannot be computed from the problem description itself. Moreover, it takes a certain amount of experience to know that  $h_{SLD}$  is correlated with actual road distances and is, therefore, a useful heuristic.

Figure 3.23 shows the progress of a greedy best-first search using  $h_{SLD}$  to find a path from Arad to Bucharest. The first node to be expanded from Arad will be Sibiu because it is closer to Bucharest than either Zerind or Timisoara. The next node to be expanded will be Fagaras because it is closest. Fagaras in turn generates Bucharest, which is the goal. For this particular problem, greedy best-first search using  $h_{SLD}$  finds a solution without ever

<sup>8</sup> Our first edition called this **greedy search**; other authors have called it **best-first search**. Our more general usage of the latter term follows Pearl (1984).

[Depth first search](#), and [breadth first search](#) also in Chapter 3.

Then there's the [flood fill search](#).

Holy crap without even realizing I applied virtual walls which is something I typed at the start when I created this Markdown file

| For the dead end you just went down, you can create a virtual wall, or an X to not go down that path.

It's crazy, it's as if I'm the same person or something

Okay I'll use A\*

### 3.5.1 Greedy best-first search

- For this particular problem, greedy best-first search using finds a solution without ever expanding a node that is not on the solution path; hence, its search cost is minimal.
- It is not optimal for finding the shortest path

### 3.5.2 A\* search: Minimizing the total estimated solution cost

- is the path cost from start node to current node
- estimated cost of the cheapest solution through
- A\* search is both complete and optimal

Okay now I got to code this. First I should make sure `self.cost()` works. It has to add a cost to the DataFrame every time there's a move

I'll create a variable called `self.total_cost` that will be the accumulated cost of the Agent. So every step will add 1

Add a cost for each `run()`

```
# Update belief state with percept to remove old false mini-agents
self.update(percept)

# Choose a move
self.choose_action()

# Add cost for moving
self.total_cost+=1

# Save the data
self.save_data()

# Officially move the Agent
self.move()
```

Added cost to `self.data`

```
new_data=pd.DataFrame(data={"Belief state": [self.state],
                           "Positions": [self.positions],
                           "Percept": self.enviro.percept(),
                           "Move": self.action,
                           "Cost": self.total_cost}, index=[0])
```

Index	Belief state	Positions	Percept	Move	Cost
0	[[0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0,... [[1, 14], [5, 4], [5, 15], [6, 10], [9, 7], [1...	0101	None	0	
1	[[0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0,... [[1, 14], [5, 4], [5, 15], [6, 10], [9, 7], [1...	0101	W	1	
2	[[0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0,... [[1, 13], [5, 3], [6, 9], [9, 6], [11, 4], [13,	1001	W	2	
3	[[0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0,... [[5, 2], [11, 3], [13, 5], [14, 10]]	1001	W	3	
4	[[0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0,... [[13, 4], [14, 9]]	0011	N	4	

This is what I have in `cost()` so far

```
def cost(self, position=None):
    if position:
        # Calculate g+h
        return self.cost() + self.envir.M_distance(self.positions[0], position)
```

Wait before I go any farther I'll upload to git as the version where `single_agent()` is fixed

Omg it undid all my progress!?!?

Okay what the hell, it reset my file on my PC to the last version, but luckily the new version was uploaded to git so I just copy and pasted that back on the file on my PC. Scariest 2 minutes of my life

Deleting all this extra code I had in `choose_action()`

```
print("No longer using single_out")
print("Positions",self.positions)
sys.exit()

# Initiate costs of all mini-agents
costs=[]

# For every position
for position in self.find_positions():
    # Initiate cost of specific mini-agent
    mini_agent_costs=[]

    # For every action
    for action in ["N","E","W","S"]:
        # If direction is possible, find cost of being at that new position
        if self.transition_model(action, position):
            mini_agent_costs.append(self.cost(action,[position[0],position[1]]))
        else:
            # Give high cost if it's not possible
            mini_agent_costs.append(100)

    # After append to costs
    costs.append(mini_agent_costs)

# print(costs)
# Return the direction that gives the lowest cost

# Initiate final costs to find direction from
final_costs=[]

for i in range(4):
    c=0

    # Add up the costs for each direction
    for mini_agent_costs in costs:
        c+=mini_agent_costs[i]

    final_costs.append(c)

# Translate index to direction
# print(final_costs)
# print(["N","E","W","S"][final_costs.index(min(final_costs))])
self.action=["N","E","W","S"][final_costs.index(min(final_costs))]
```

now it's just

```
def choose_action(self):
    # Eliminate mini-agents until there's one
    if len(self.positions)>1:
        self.action=self.single_out_agent()
        return self.action

    return None
```

The function `A_star` so far

```

def A_star(self):
    action_costs={}
    for action in self.possible_actions():
        action_costs[action]=self.cost(self.action_to_position(action, self.positions[0]))
    print(action_costs)

```

which will be in `choose_action()`

```

def choose_action(self):
    # Eliminate mini-agents until there's one
    if len(self.positions)>1:
        self.action=self.single_out_agent()
        return self.action

    self.action=self.A_star()
    return self.action

```

I would like to save each position's cost that was calculated in a matrix since it would be cool to colour code later and than compare with state. Also might be useful

Okay so far coded all functions being used. So in `choose_action()` we have

```
self.action=self.A_star()
```

and in `A_star()` we have

```

action_costs={}
for action in self.possible_actions():
    action_costs[action]=self.cost(self.action_to_position(action, self.positions[0]))

```

to calculate the for each possible position the agent can go to. And then we have

```

lowest_cost=min(action_costs.values())
for key,value in action_costs.items():
    if lowest_cost==value:
        print(key, "\n")
        return key

```

to calculate the direction with the lowest cost.

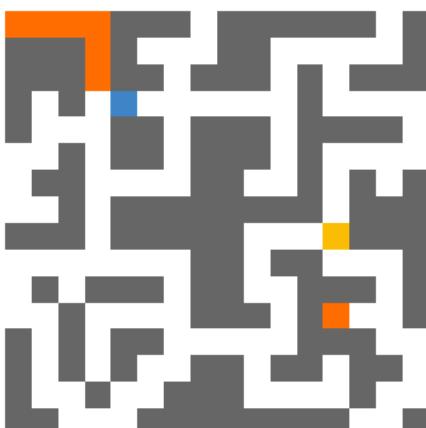
And then we have `cost()` that should be calculating just fine

```

def cost(self,position=None):
    if position:
        # Calculate g+
        return self.total_cost+self.enviro.M_distance(self.positions[0],position)

```

What's weird so far is how `E` and `W` in this situation cost the same



```
E [3, 5]
W [3, 3]
A_star {'E': 10, 'W': 10}
[3, 4]
E
```

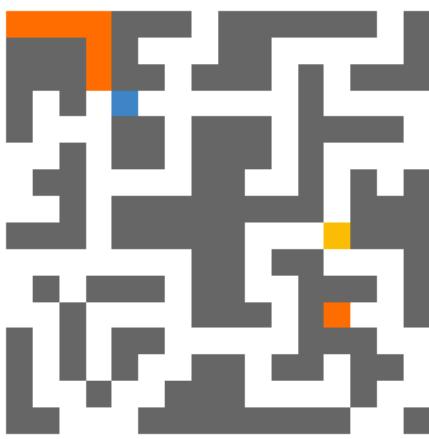
despite it's obvious that W is closer to the goal.

Omg it is because I'm calculating the [Manhattan distance](#) from the action's position to the real agent instead of the action's position to the goal ahhh

There cost() should be fixed

```
return self.total_cost+self.enviro.M_distance(position,self.goal)
```

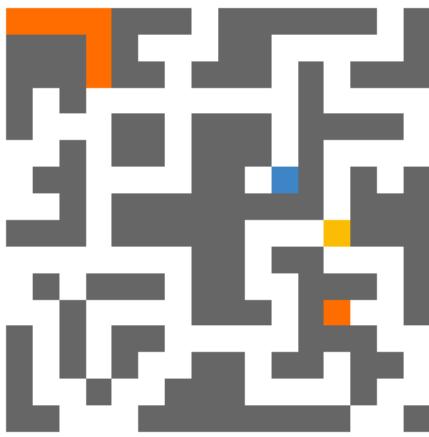
Now the values makes more sense



```
E [3, 5]
W [3, 3]
A_star {'E': 21, 'W': 23}
[3, 4]
E
```

Going to let the code run longer to see what happens

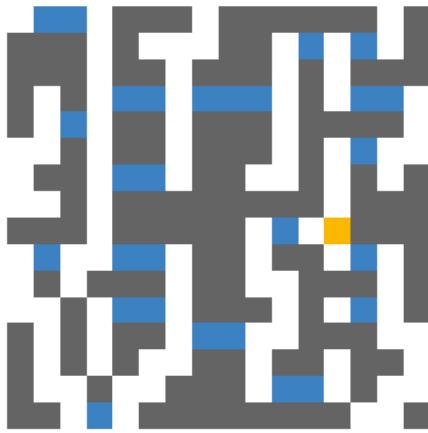
Okay the algorithm almost works, it goes so close to the goal but is caught in this corner



Intersections are something I need to deal with

I wondered if you can make gifs using Python and [you can!](#)

Ayy made a gif, now I don't have to copy and paste a bunch of images



```
def create_gif(self):
    # Make sure frames are in order
    listdir=list(os.listdir("gif"))
    listdir=sorted(listdir,key=lambda x: int(x.split('.')[0]))

    # Make images frames
    frames=[]
    for png in listdir:
        frames.append(imageio.v2.imread("gif/{}".format(png)))

    # Make the gif
    imageio.mimsave("session.gif",frames,fps=5)

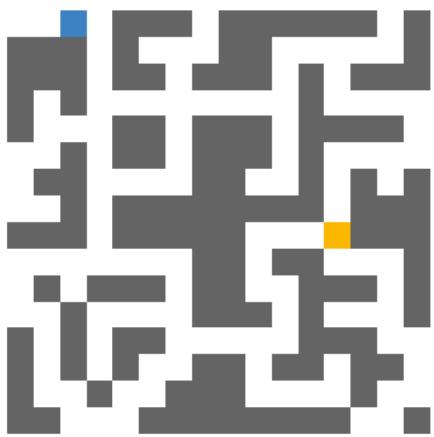
    # Delete all files in gif folder
    for png in os.listdir("gif"):
        os.remove("gif/{}".format(png))
```

In visualized()

```
# Make gif
## Create image if nothing in folder
if not os.listdir("gif"):
    plt.savefig("gif/1.png")
else:
    # Get all the numbers of the pngs
    numbers=[]
    for png in os.listdir("gif"):
        numbers.append(int(png.split('.')[0]))

    # And then create a new image one number higher than the max
    plt.savefig("gif/{}.png".format(max(numbers)+1))
```

Made the real agent stay in view for 2 frames



I guess today I got to make the code that saves intersections so if the agent reaches a dead end it can go back to it.

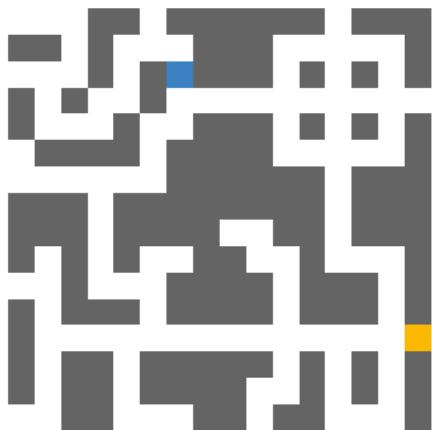
Maybe I can place a construction cone every time the agent is forced to go back the step it came from?

Actually what I need to do is have the agent keep going forward until it finds an intersection. Once it is there, it should find the direction with the best cost. If it goes into a dead end or loop, it should turn back to the intersection and try another direction.

Cause so far the agent does not follow through with the path

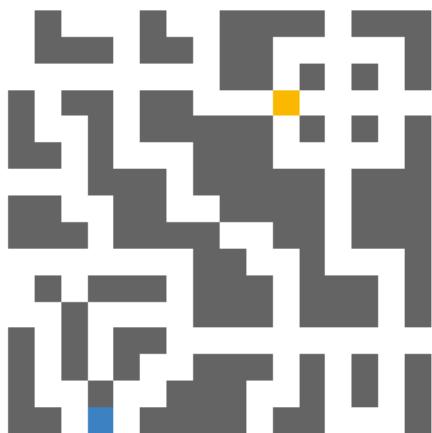


This will be a tricky situation because of how I designed the map



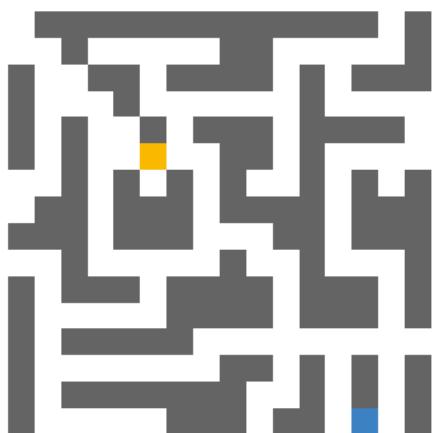
Seed 122402061215

Oh damn first time I ever saw the agent reach the goal!



Congrats Seed 020415030310

Another W



Seed 241415130505

First time seeing these messages go off

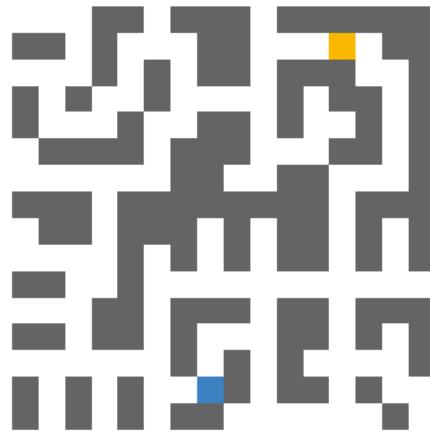
The Agent has reached the goal!

Goal was found

Goal was found

```
Goal was found  
Goal was found
```

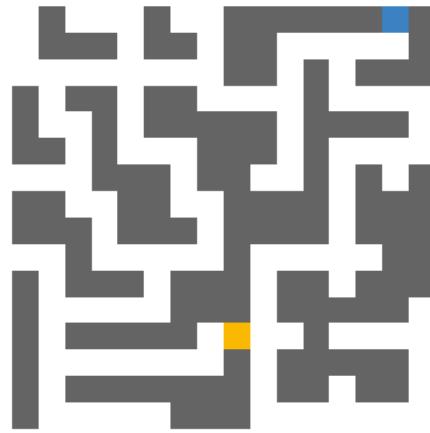
Cool



Seed 103314070112

Okay after looking through different scenarios and getting of an idea of the problem I'm solving, it's time to start coding!

Okay, first thing I need to code is the agent to follow through with a path until another intersection is found. I'll be testing the code on this seed



Seed 041200141208

I'll have `A_star()` act at intersections, and I'll make a new function that follows through with a straight path

Creating `self.follow_path()` to have agent follow through with path after choosing a direction from an intersection

Going to create a function for intersections called `if_intersection()`

Added a parameter for `scan()` so we can scan the state without the construction cones

```

def scan(self, position=None, use_initial_state=False): #percept to give to agent #Might use just to initialize the agent
    # Check if to use initial_state or current state
    if use_initial_state:
        state=copy.deepcopy(self.initial_state)
    else:
        state=copy.deepcopy(self.state)

```

Made `if_intersection()` by scanning the initial state (where's there no construction cones) and checked how many zeros are in the percept

```

def if_intersection(self, position=None):
    if position:
        pass
    else:
        # Default position is the agent
        position=self.positions[0]

    percept=self.scan(position, True)

    # If more than 2 options
    if percept.count("0")>2:
        # It is a intersection
        return True
    else:
        return False

```

Made `follow_path()` work. It finds the two possible places it can go, and then removes the opposite direction it went last turn

```

def follow_path(self):
    try:
        action=self.possible_actions()
        action.remove(self.reverse_action(self.data["Move"].iloc[-1]))
        action=action[0]
    except ValueError:
        action=self.possible_actions()[0]

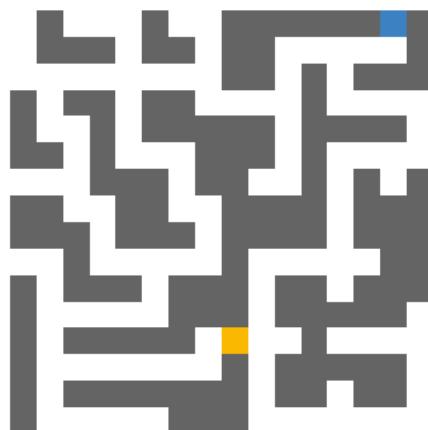
    print("follow path removed", self.reverse_action(self.data["Move"].iloc[-1]), "from", self.possible_actions())
    print(action, "\n")

    return action

```

I used `try:` in case the action to remove isn't there (most likely to a construction cone following it), and it's `except` is to use the only action left

Now I have this problem where the agent bounces between two intersections. I need to make sure the agent cannot go back where it came from unless it has to

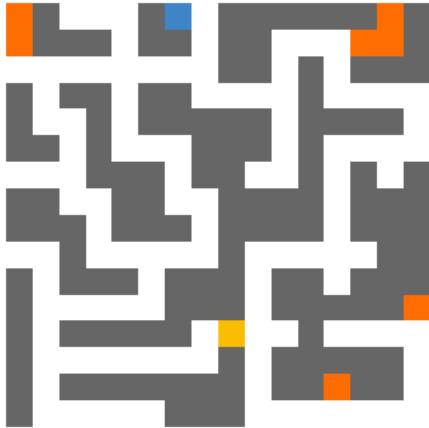


My solution will be that `A_star()` won't have the direction it came from as an option. Unless there's no other way back

Though the agent might be forced go back where it came from at the intersection because all the other paths would be filled with construction cones

From how I coded `follow_path()`, when the agent is at a dead end, it has no other options left since I removed the action that allows it to come back

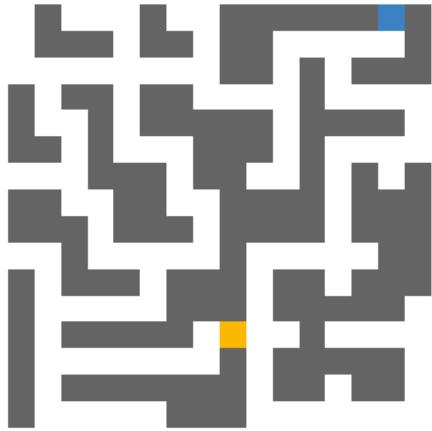
Damn can't put the gif to show the example because the code ended abruptly, but it got stuck here



Okay fixed `follow_path()`

```
# Check if agent is in dead end
if self.positions[0] in self.dead_ends:
    return self.possible_actions()[0]
```

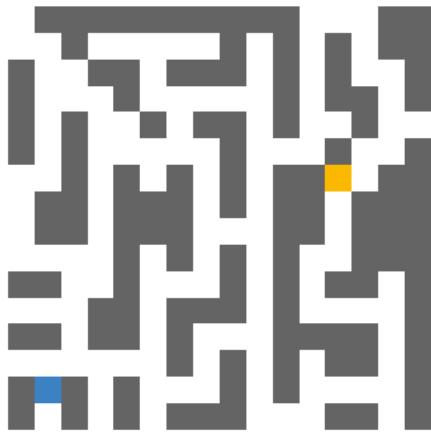
It worked!!!



Seed 041200141208

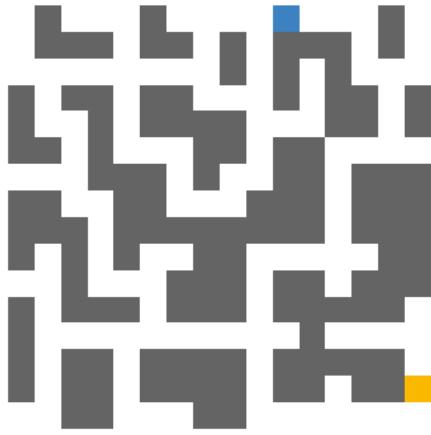
Let's see if it works for other maze cases

This one is really funny because it didn't entire loop to retry the same intersection



Seed 213114010612

Oh found my first loop, It's not over yet!

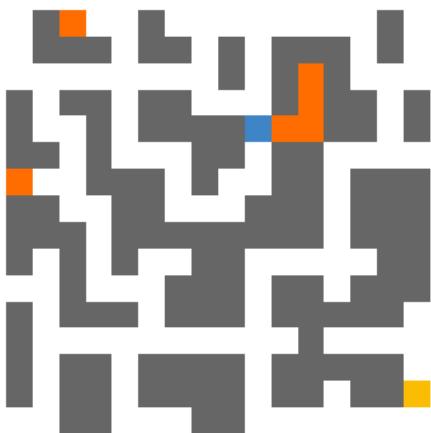


Seed 032200101415

I made `loop_check()` before where it finds if the position it's at has been at before, and then it return the move it used last time. So hopefully this works though I'm pretty sure it would cause other problems.

```
# Loop check
if self.loop_check():
    possible_actions.remove(self.loop_check())
```

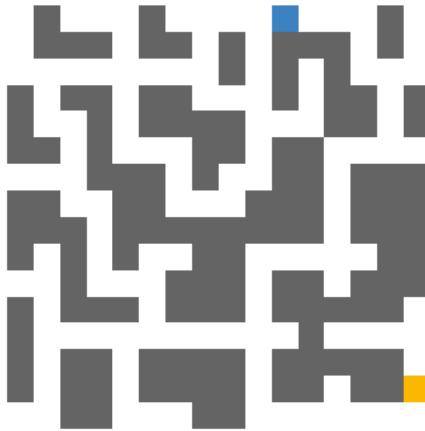
An action it did before is no longer possible because it's a dead end



So for the check I made sure it only removes the action if it's in `possible_actions`

```
# Loop check
if self.loop_check():
    # If action is in possible actions
    if self.loop_check() in possible_actions:
        possible_actions.remove(self.loop_check())
```

Omg it just found another loop



Okay my idea for a solution is to have a dictionary for each intersection the agent has passed. At each intersection it can check the directions it tried, and if possible try to not use those directions again.

Actually to only use the dictionary when needed, I can make a function that goes through all of all data, and for all positions make a dictionary their actions. I'll call it `location_history()`.

Then I can call on the location history of that position to check which actions were tried before

The `location_history()` function

```
def location_history(self, position):
    location_history = {}

    for i, location in self.data["Positions"]:
        # If there's a single agent
        if len(location) == 1:
            # Check if location in dict
            if location[0] not in location_history.keys():
                # Create location's key
                location_history[location[0]] = self.data["Move"].iloc[i]
            else:
                # Else, add move to location
                location_history[location[0]] += self.data["Move"].iloc[i]

    return location_history
```

Fixed `location_history()`

```
def location_history(self, position):
    location_history = {}

    for i, location in enumerate(self.data["Positions"]):
        # If there's a single agent
        if len(location) == 1:
            # Check if location in dict
```

```
        if str(location[0]) not in location_history.keys():
            # Create location's key
            location_history[str(location[0])] = self.data["Move"].iloc[i]
        else:
            # Else, add move to location
            location_history[str(location[0])] += self.data["Move"].iloc[i]

    return location_history[str(position)]
```

A problem with my `location_history` is that from `self.data` the move that was chosen when at the position is given one index after `data - DataFrame`

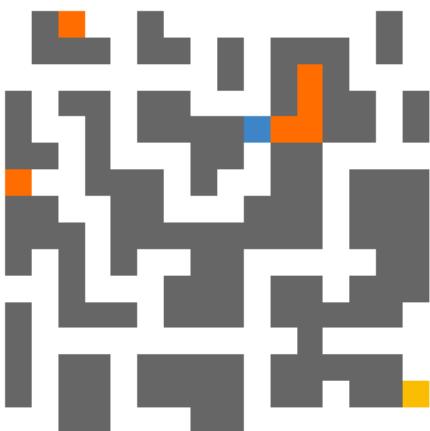
Index	Belief state	Positions	Percept	Move	Cost
	[[0, 1, 0, 2, 0, 1, 0, 0, 2, 0, 2, 2, 2, 0, 1, ..., [0, 3], [0, 8], [0, 10], [0, 11], [0, 12], [2, 1], [2, 4], [3, 1], [3, 4], [4, 1], [4, 4], [10, 9], [11, 9], [12, 9]]]	[[0, 3], [0, 8], [0, 10], [0, 11], [0, 12], [2, 1], [2, 4], [3, 1], [3, 4], [4, 1], [4, 4], [10, 9], [11, 9], [12, 9]]]	1001	None	0
	[[0, 1, 0, 2, 0, 1, 0, 0, 2, 0, 2, 2, 2, 0, 1, ..., [0, 3], [0, 8], [0, 10], [0, 11], [0, 12], [2, 1], [2, 4], [3, 1], [3, 4], [4, 1], [4, 4], [10, 9], [11, 9], [12, 9]]]	[[0, 3], [0, 8], [0, 10], [0, 11], [0, 12], [2, 1], [2, 4], [3, 1], [3, 4], [4, 1], [4, 4], [10, 9], [11, 9], [12, 9]]]	1001	W	1
	[[0, 1, 2, 0, 0, 1, 0, 2, 0, 2, 2, 2, 0, 0, 1, ..., [0, 7], [0, 9], [0, 11], [2, 1], [2, 4], [3, 1], [3, 4], [4, 1], [4, 4], [10, 9], [11, 9], [12, 9]]]	[[0, 7], [0, 9], [0, 11], [2, 1], [2, 4], [3, 1], [3, 4], [4, 1], [4, 4], [10, 9], [11, 9], [12, 9]]]	1000	S	2
	[[0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, ..., [1, 7], [1, 9], [3, 1], [3, 4], [10, 9]]]	[[1, 7], [1, 9], [3, 1], [3, 4], [10, 9]]]	0110	S	3
	[[0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, ..., [2, 9], [4, 4], [11, 9]]]	[[2, 9], [4, 4], [11, 9]]]	0110	S	4
	[[0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, ..., [3, 9], [12, 9]]]	[[3, 9], [12, 9]]]	0100	S	5
	[[0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, ..., [4, 9]]]	[[4, 9]]	0010	E	6
	[[0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, ..., [4, 10]]]	[[4, 10]]	1001	E	7
	[[0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, ..., [4, 11]]]	[[4, 11]]	0101	N	8
	[[0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, ..., [3, 11]]]	[[3, 11]]	0110	N	9
0	[[0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, ..., [2, 11]]]	[[2, 11]]	1110	S	10
1	[[0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, ..., [3, 11]]]	[[3, 11]]	1110	S	11
2	[[0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, ..., [4, 11]]]	[[4, 11]]	1101	W	12
3	[[0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, ..., [4, 10]]]	[[4, 10]]	1101	W	13

That means we need to add one `i` to the move

```
location_history[str(location[0])] = self.data["Move"].iloc[i+1]
```

But now I have the problem where I removed all the actions from `possible_actions`, so it cannot move anywhere

Never mind it must be a problem in my code because this agent never tried `N` and code says `possible_actions` is empty



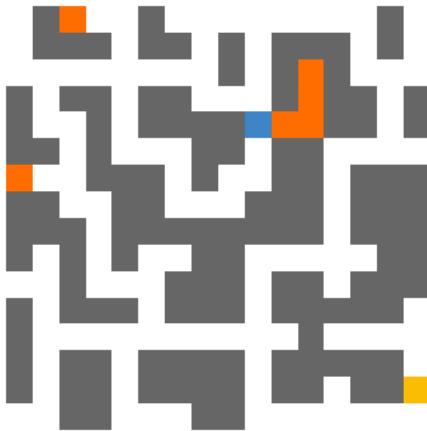
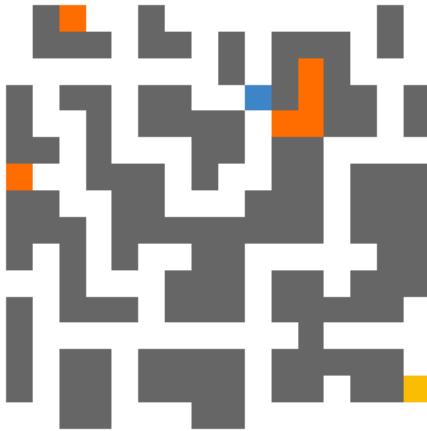
Says it removed **E** and **S** which is perfect, so it forces the agent **N**, but the options are empty

```
move E | possible actions ['N', 'S']
move S | possible actions ['N', 'S']
A_star {}
```

Okay this is the problem

```
# Prevent going back
try:
    possible_actions.remove(self.reverse_action(self.data["Move"].iloc[-1]))
except ValueError:
    pass
```

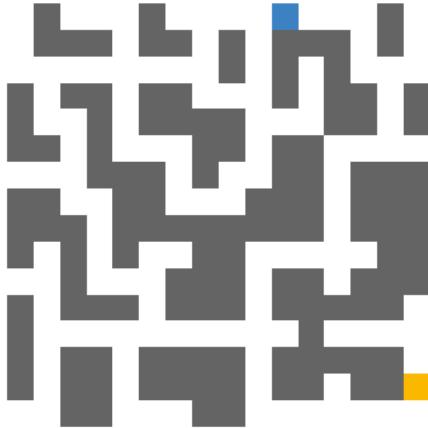
since the agent came from N



Tried to only remove the way it came from if it's not in a loop. If it's in a loop then it's okay to come back the way you came from

```
# Prevent going back
if not self.loop_check():
    try:
        possible_actions.remove(self.reverse_action(self.data["Move"].iloc[-1]))
    except ValueError:
        pass
```

but it lead to this



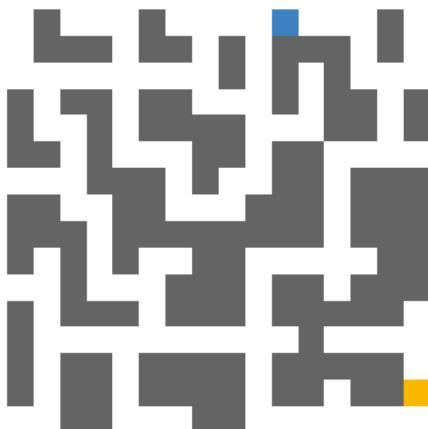
This is happening probably because there's two intersections

A loop is no longer a loop if there's a wall that breaks the circle. So if the agent is SURE it's in a loop then it should place a construction cone

Well I tried to add a wall when the agent realizes it only has one way back and it's doing this which I'm not sure is a good thing or a bad thing



Yay! It technically did it



Okay new plan. The agent could dip its toes into the loop just to make sure it's a loop. So, if the agent detects it could be in a loop, then it verifies this by seeing if the next move is predictable. If the pattern does start repeating, then place a wall. This allows the agent to take paths it has before

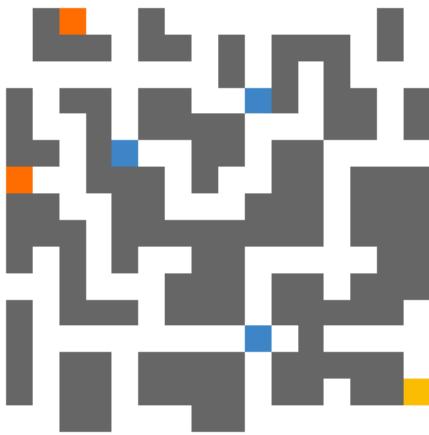
The difference between a loop and turning back is that turning back is the reverse of where you came from, as a loop is only by moving forward

So an agent always returns to a position from a certain angle in a loop. When turning back it returns to a position from not the same angle as before

If a loop is a circle, then placing a wall would never trap the agent.

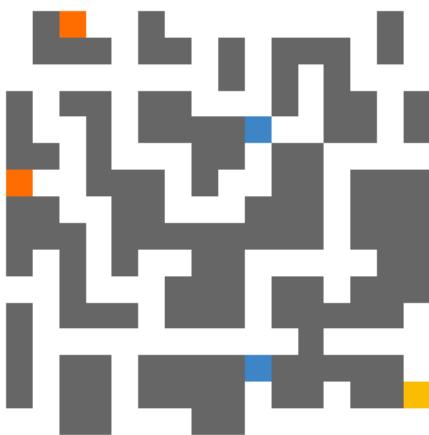
Can tell this agent came from south from this value

4	[[0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, ... [[2, 9], [4, 4], [11, 9]]]	0110	S	4
5	[[0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, ... [[3, 9], [12, 9]]]	0100	S	5
6	[[0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, ... [[4, 9]]]	0010	E	6



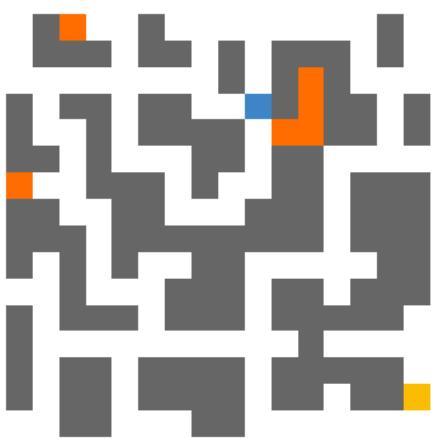
and the next agent came from south from this value

4	[[0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, ... [[2, 9], [4, 4], [11, 9]]]	0110	S	4
5	[[0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, ... [[3, 9], [12, 9]]]	0100	S	5
6	[[0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, ... [[4, 9]]]	0010	E	6



Now if a different case, we can tell this agent came from W from this value

12	[[0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, ... [[4, 11]]]	1101	W	12
13	[[0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, ... [[4, 10]]]	1101	W	13
14	[[0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, ... [[4, 9]]]	0110	S	14



and this one from

Wait never mind the moves are aligned with the positions. The position the agent is currently in because of the move in the same row

Ah never mind, the positions are aligned with the move that the turn took. If you want the move that made the position, it's the move before

Final answer

Got to have the code understand these values match

Instance 1 of the loop

16	[[0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1,... [[6, 9]]]	0101	W	16
17	[[0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1,... [[6, 8]]]	1010	S	17
18	[[0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1,... [[7, 8]]]	0101	W	18

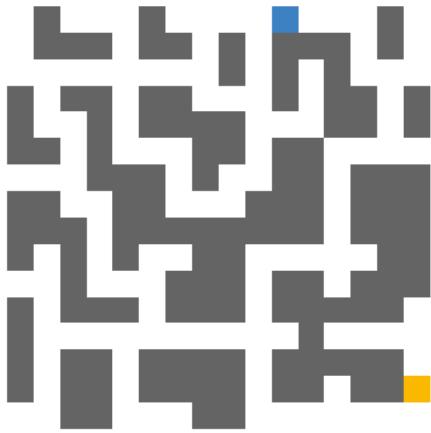
Instance 2 of the loop

36	[[0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1,... [[6, 9]]]	0101	W	36
37	[[0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1,... [[6, 8]]]	1010	S	37
38	[[0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1,... [[7, 8]]]	0101	W	38

Omg I have been trouble shooting `loop_check_v2()` for the longest time when I realize why the results are not showing is because I only placed the function in `A_star()` where intersections are handled. Never placed the functions where straight paths are handled which is where I was looking for. Omggggg

My code must have been working for the past hour but I never knew because the function wasn't being executed in the right spot. The only reason I knew it was working was when I let the code run longer until it hit an intersection and executed `loop_check_v2()`

Welp it found another loop



I'm so confused cause it had a clear chance to go left when at the bottom there but it decided to go up??

Oh it's because it's the same cost

```
{'N': 72, 'W': 72}
```

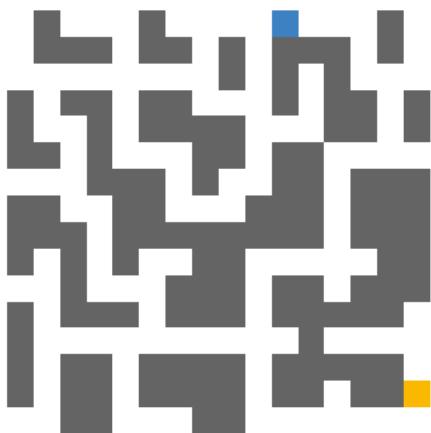
Maybe I should make it so if it's the same cost then try to not do the direction it had tried before

Added code in `A_star()` to make sure if cost are the same then remove the actions that were used before

```
# If all costs the same, don't choose a direction agent tried before
if len(set(action_costs.values()))==1 and len(action_costs)>1:
    # Find direction tried before
    if str(self.positions[0]) in self.location_history().keys():
        # Check for actions that was used before
        for action in self.location_history()[str(self.positions[0])]:
            # Make sure action is in possible actions
            if action in action_costs.keys():
                # Remove action
                del action_costs[action]
```

Never mind the last code doesn't work

Okay for now I'm leaving this loop thing alone until I think of a better solution. This is what I got so far with a cost of 161



But my loop solution works really well, it's just the agent is not taking the best turns. And because of that it puts itself in an unnecessary loop that is detected by my loop check.

And I tried my best to think of the best definition of a loop. A loop is when you repeat the exact same path from the same direction

Maybe there has to be a more mathematical way to define it

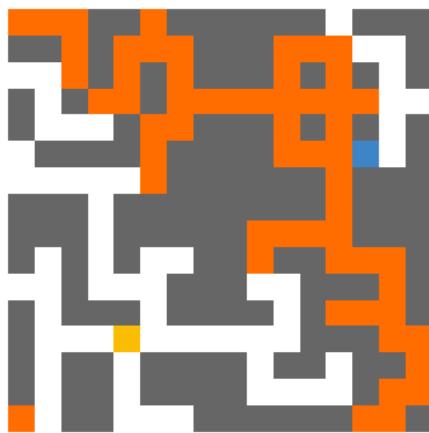
Running a benchmark for cost

An error happened during it. It's Seed 401309120101

Oh it's from my gif making thing

Okay run attempt 2

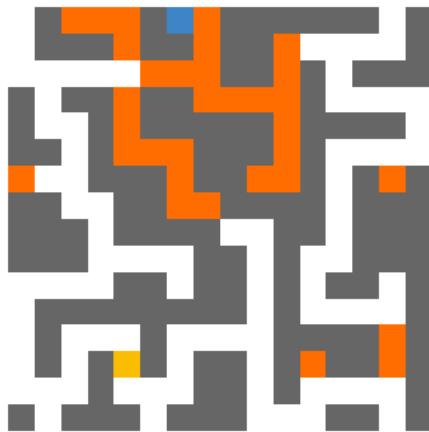
Okay after 4 hours of coding the loop prevention it doesn't work and causes the agent to go in dead ends very easily



So I cannot benchmark

Maybe I should have the loop thing only at intersections

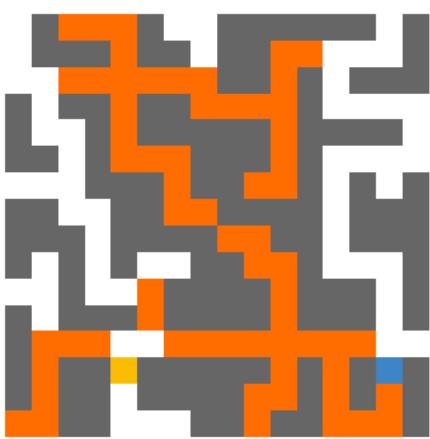
Never mind still gives dead ends with ease



Seed 401309120101

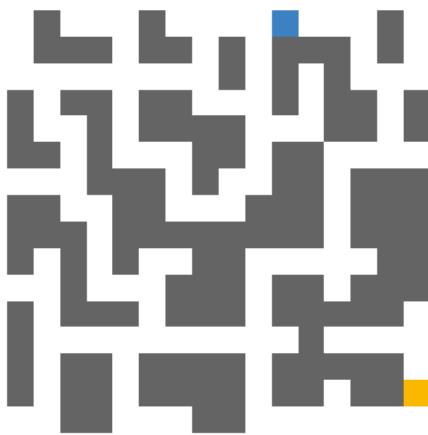
Cleaning up and removing comments from code

Massive problem with Seed 042401101304 is how goal is not a possible direction. Which is weird because the h distance should be 0



My dad's idea to fix the loop is to have a one way wall. That means if an agent enters a position from one direction, it never can go through that position again. So that allows the agent to go through it once (one-way construction cone), and if it hits a dead end, it can go through the position once more. After that, it's a full wall.

Uhhmmm for the reason isn't behaving as it did last night. It should instantly go into the loop but now it goes N ? I must have changed something and forgot about it ahh



I must have messed something up in `single_out_agent()` since it's no longer doing the same actions as before

I wished I didn't clean up the code until I was done the code for sure and also when I wasn't tired. I guess I'll leave it and just try to code my dad's idea. I know I'll end up spending all day on trying to fix that. What matters is that it still goes to the loop section though now in way more steps

New loop check

```
current_position=self.positions[0]

# Check all positions around current position
possible_positions=[]
for action in self.possible_actions():
    possible_positions.append(self.action_to_position(action))

# Check each possible position's previous entering action
forbidden_actions=[]
for position in possible_positions:
    for i,data_position in self.data["Positions"]:
        # Make sure we are checking after single_agent()
        if len(data_position)==1:
            if position==data_position[0]:
                forbidden_actions.append(self.data["Move"].iloc[i-1])
```

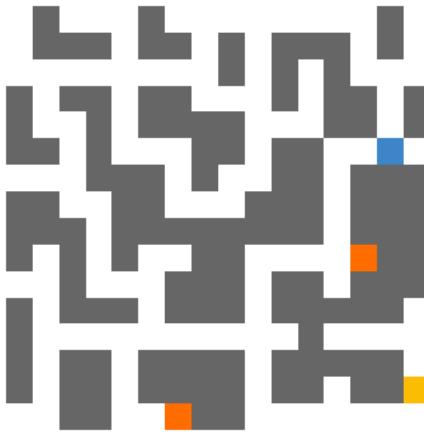
```

# Remove forbidden actions from possible actions if possible
if possible_actions:
    for action in forbidden_actions:
        # If agent can do the forbidden action
        if action in possible_actions:
            # Remove action
            possible_actions.remove(action)
        else:
            pass
    # Return new possible actions
    return possible_actions
else:
    # Return forbidden actions. These actions should not be available for the agent to do
    return forbidden_actions
# return True

```

Trying out `loop_check_v2()` only at intersections and the agent got deleted. Using original loop map Seed 032200101415

Oh because `A_star` thought it could move `S` into a wall?



```

A_star {'S': 20}
[5, 14]
S

```

Oh it's just because I forgot the put in the input `possible_actions` in

```
possible_actions=self.loop_check_v2(possible_actions)
```

The agent got in this position by going down `S` from the square to left of it. That means my loop check will not allow the agent to approach it from `S`

```

forbidden actions ['S']
possible actions ['N', 'W', 'S']

```

The thing is, removing `S` won't work because the agent is currently not in the same position it was in before. This just prevents it from going `S` in its current position.

Removing `S` would only be good if the agent was above the square on the left. If it was in the same position was when it went `S`. I got to add that check

Adding this check fixed that

```

# Only forbid action if agent is in same position as when previously entered
if current_position==self.data["Positions"].iloc[i-1]:

```

I also want to have the code to place a wall if both directions were approached.

Added the code

```
# Check if agent approached possible position from all possible angles
for position in possible_positions:
    # Find all possible ways you can enter possible position
    possible_actions_reversed=[]
    for action in self.possible_actions(position):
        possible_actions_reversed.append(self.reverse_action(action))

    # Find all previously entered action into possible position
    previous_actions=[]
    for i,data_position in enumerate(self.data["Positions"]):
        # Make sure we are checking after single_agent()
        if len(data_position)==1:
            # If possible position matches position in data
            if position==data_position[0]:
                # Append previous action
                previous_actions.append(self.data["Move"].iloc[i-1])

    # If all the available ways to enter the position has been done, place a wall
    if set(possible_actions_reversed)==set(previous_actions):
        self.dead_ends.append(position)
```

Well it didn't work for the main case I wanted to

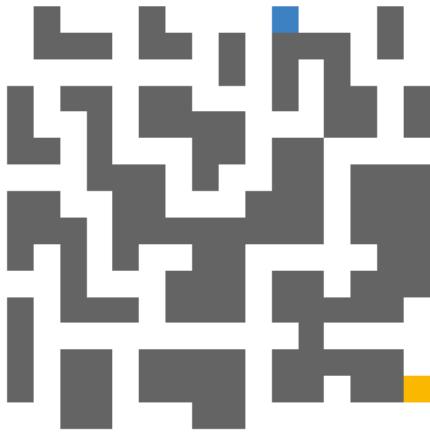


It shouldn't be allowed to re-entered the same spot

Problem was this line didn't have [0]

```
if current_position==self.data["Positions"].iloc[i-1][0]:
```

The agent gets stuck in an intersection where all options are exhausted.



Okay the main problem with this code is that the agent is just not smart. The directions it takes is all over the place and always gets itself in trouble. There was to be a way to keep this agent more organized that avoids loops and everything. I thought after coding the A\* algorithm at each intersection and blocking off dead ends the code would be proper. And maybe after a little bit of loop fixing for rare cases it would work. But no matter what solution I try to think of it will not work because the agent is flawed since the start. I need have an organized system for the agent to follow so it explores the maze properly. I'm wasting a lot of time and I don't want to finish with a project that I worked on for month to have a result that sometimes work.

How do I get the agent from bouncing everywhere to a systematic system?

Other algorithms such as [Depth first search](#), and [breadth first search](#) remembers to go back to its intersection as soon as it finds a dead end. Then once back at its intersection it tries a different path.

If it finds a new intersection, apply the same logic until all of those paths are also exhausted.

If I could code depth first into this code as well, then my code might be saved.

I got one more shot. I got around 3 hours to code this. To avoid wasting time I'll make a plan in a flowchart and strictly follow that. I'll think of everything I need to do before coding. I'm unlocking Alex v2. No more messing around. I've been bullied by this agent for way too long >:(

Wait [he](#) said

Whenever you reach a dead end or a loop,

Hmm, I hope that my dad's loop check is good enough for situations like that. It is a good way to define a loop. It's just the agent cannot repeat paths for no reason. Once again, it needs a systematic way to move through the maze.

I've noticed the mouse keeps moving until it hits a dead end, so that means even if there's an intersection, keep going. Interesting

A loop is when you end up at the same intersection you started at.

Okay this is my plan. I tested this path in Krita and it should work. All I got to do is code it. Some parts I'm not sure of but I'm sure I'll figure it along the way. For example, I'm not sure what will be the `entrance_action` if the agent spawns on an intersection

Here is the canvas [Loop fixand](#) here is my sketch in Krita



This is how the `self.interactions` will be saved into `self.data`

```
self.interaction={"actions":None,"entrance_action":None}
```

Added new column to `self.data`

```
new_data=pd.DataFrame(data={"Belief state":[self.state],
                           "Positions":[self.positions],
                           "Percept":self.enviro.percept(),
                           "Move":self.action,
                           "Cost":self.total_cost,
                           "Intersection":self.intersection},index=[0])
```

New function called `algorithm()`

```
if self.if_intersection():
    self.action=self.algorithm()
```

Actually would be better if entrance action was a position instead. More consistent

Never mind it will be an action, we just need to reverse the action so it's always the same reference point from the intersection

Using list instead of dataframe after realizing you can't use `dict` or a list with a dataframe. Just will use a single variable `self.intersection`

Progress so far

```
[[None, None, None], [[2, 14], 'EWS', 'W'], [[5, 14], 'NEW', 'N'], [[0, 9], 'EWS', 'E'], [[3, 9], 'NWS', 'N'], [[4, 9], 'NES',
'N'], [[2, 4], 'NEWS', 'S'], [[2, 7], 'NWS', 'W']]
```

A small win feels nice since for the past hour I kept failing to even start

Okay what I got so far in `algorithm()`

```
def algorithm(self):
    # Current position is an intersection
    current_position=self.positions[0]

    # Check if intersection is in data
```

```

positions_list=[]

## Add list of all previous positions
for l in self.intersection:
    positions_list.append(l[0])

## Check if current intersection is in data, if not then append intersection
if current_position in positions_list:
    pass
else:
    # Appending position, possible actions, and entrance_action

self.intersection.append([current_position,"".join(self.possible_actions()),self.reverse_action(self.data["Move"].iloc[-1])])

# In case self.intersection still has [None,None,None]:
if self.intersection[0][0]==None:
    self.intersection.pop(0)

# Get intersection data
for i,intersection in enumerate(self.intersection):
    if intersection[0]==current_position:
        interdata=self.intersection[i]
        actions=interdata[1]
        entrance_action=interdata[2]
        interi=i

# Check if intersection has possible actions
if actions:
    # Check if it's a dead end
    if len(interdata[0])==1:
        # Return only option left
        return actions[0]
    else:
        # Check if only entrance action left
        if set(actions)^set(entrance_action):
            return entrance_action
        else:
            # Loop check
            ## Check if current possible actions has an action that actions doesn't have
            if (set(actions)^set(self.possible_actions()))&set(self.possible_actions):
                loop_action=(set(actions)^set(self.possible_actions()))&set(self.possible_actions)

                ## Block path
                self.dead_ends.append(self.action_to_position(loop_action))

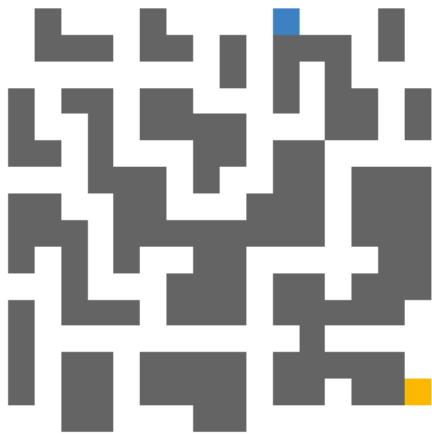
            # Passed checks to choose cheapest action
    else:
        # Choose cheapest action using A*
        result=self.A_star(actions)
        # Remove action from possible from self.intersection
        self.intersection[interi][1].strip(result)

    return self.A_star(actions)

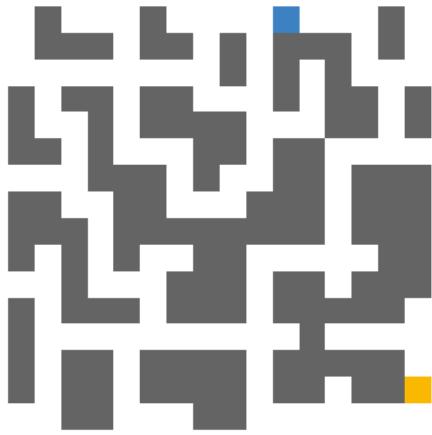
```

I couldn't really test any of this code so I was coding it very carefully. So I'll test it now

Okay the code didn't die on me and instead made an encouraging loop for me to destroy instead

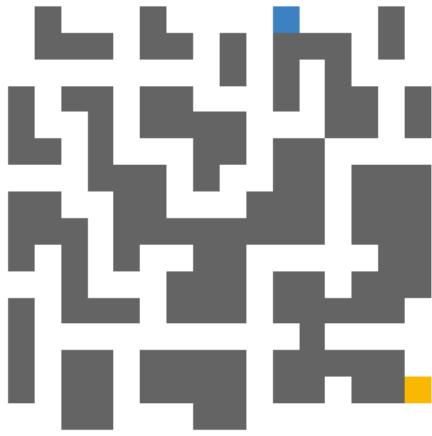


OMG I DIDN'T SEE MOVEMENT IS YEARS

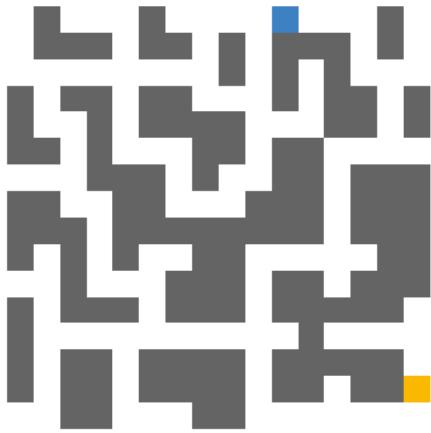


It went through my entire `algorithm()` code without errors, let's see what happens if it runs longer

Jesus christ not again



Damn



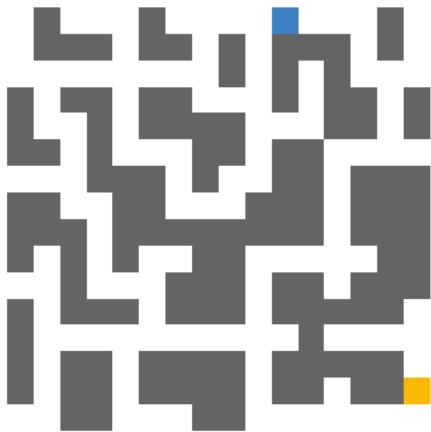
My goodness god this is what happens when you go on a 6+ hour coding streak.

```
def A_star(self,possible_actions=None):  
  
    # Possible acitons choice  
    if possible_actions:  
        possible_actions=self.possible_actions()  
    else:  
        pass
```

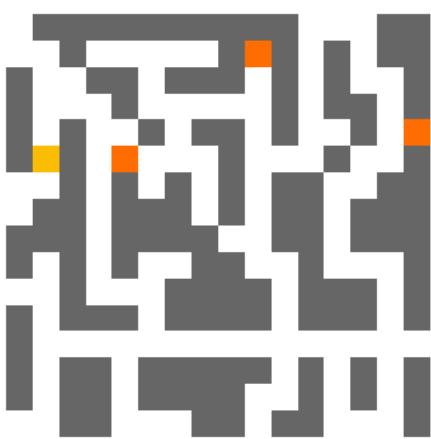
That was my problem for awhile because I knew that by having if `possible_actions` when it's `None`, it would use the default `possible_actions()`. And for some reason I relooked at the code about like 50 times and didn't notice that I had the if statement wrong. It should be

```
def A_star(self,possible_actions=None):  
  
    # Possible acitons choice  
    if not possible_actions:  
        possible_actions=self.possible_actions()  
    else:  
        pass
```

Now I get this beautiful result that I hope is not a fluke



Okay it was, but I think I know the problem. It's that the placing of the wall kills the agent



From how I coded my code I didn't even need to place a wall. In fact, placing a wall would have cause problems like so



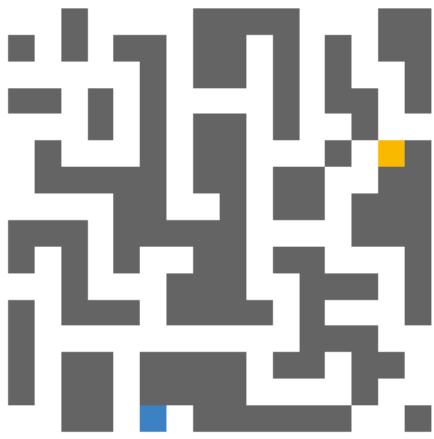
Although it got to the goal there was a very real chance that the goal wasn't in that corner of the map. So, by not putting the wall, I just instead just removed the action

This is how it looks instead without the wall placement. Much safer and also cool cause you can see how indepdent my agent grew up to be. Not needing any construction cones anymore :')



Seed 212411140501

Oh right I have this error somewhere where goals at intersections don't count



Seed 412015050514

Cheap code but I gotta get that gooalll

```
def goal_snipe(self):
    if "3" in self.scan(self.positions[0]):
        for i,number in enumerate(self.scan(self.positions[0])):
            if number=="3":
                if i==0:
                    return "N"
                if i==1:
                    return "E"
                if i==2:
                    return "W"
                if i==3:
                    return "S"
```

Okay it seems like `scan()` or `enviro.percept()` does not return the goal if it's in view? I was sure it does but this goal is not getting detected

I tried 3 different methods to force this agent to the goal but it won't do it I'm super confused. I think the goal check might be broken?

```
def goal_snipe(self):
    # percept=""
    position=self.positions[0]
    # state=copy.deepcopy(self.initial_state)

    # i=position[0]
    # j=position[1]

    # #N
    # percept+=str(state[i-1][j])
    # #E
    # percept+=str(state[i][j+1])
    # #W
    # percept+=str(state[i][j-1])
    # #S
    # percept+=str(state[i+1][j])

    # print("SNIPEEE",percept)
    # if "3" in percept:
    #     for i,number in enumerate(percept):
    #         print("NUBMER",number)
    #         if number=="3":
    #             if i==0:
    #                 return "N"
    #             if i==1:
    #                 return "E"
    #             if i==2:
```

```

#             return "W"
#         if i==3:
#             return "S"

for action in self.possible_actions():
    if self.action_to_position(action)==self.goal:
        return action

```

I'll just save the seed because I want to fix the loops that are still present  
Seed 412015050514

This seed just died instantly Seed 202413010613

Okay I think after a certain time it's fine to give. I'm going to record the video explaining where I am at. I seriously wanted to finish this code and I was sure I would yesterday afternoon. I had no idea this loop problem was so deadly. Just I gave it so much hope and time, and every time I feel like I get close, I just realize how many hours or days I am away from getting a 100% success rate. So far my success rate is about 20%, which is not something I would like couple months of work to lead up to. But I guess it was about the journey.

After 7+ hours of coding straight is when I give up, and I think that's a strength I have. It literally felt like I was coding for 2 hours since it's fun and I could keep going but I got to wake up early tomorrow.

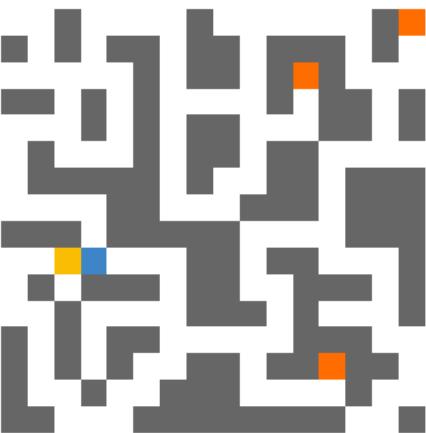
Going to try to fix the problem where the goal can't be found at intersections today

Need to fix this problem without using `goal_snipe()`

The 3 is in the `self.state`, so that's good

Going to test only using `A_star()`, and seed 430014150902 is the perfect example to use

What the agent sees in this situation



```

A star possible actions ['N', 'E', 'S']
Action_costs {'N': 27, 'E': 29, 'S': 29}
N

```

Removed `self.total_cost` from part of calculation since it's unnecessary

```

def cost(self,position=None):
    if position:
        # Calculate g+
        return self.total_cost+self.enviro.M_distance(position,self.goal)

```

to

```

def cost(self, position=None):
    if position:
        # Calculate g+h
        return self.enviro.M_distance(position, self.goal)

```

Yeah something isn't calculated right because the cost shouldn't 8

```

A star possible actions ['N', 'E', 'S']
Action_costs {'N': 6, 'E': 8, 'S': 8}
N

```

Agent is at [9,3] and these are the positions when using `action_to_position()`

```

action to position N [11, 6]
action to position E [12, 7]
action to position S [13, 6]

```

The `self.positions[0]` is wrong??

```

Position [12, 6]
A star possible actions ['N', 'E', 'S']
action to position N [11, 6]
action to position E [12, 7]
action to position S [13, 6]
Action_costs {'N': 6, 'E': 8, 'S': 8}
N

```

I believe the problem is due because `self.if_intersection()` does not see the goal as an intersection. The position I gave above is of an intersection

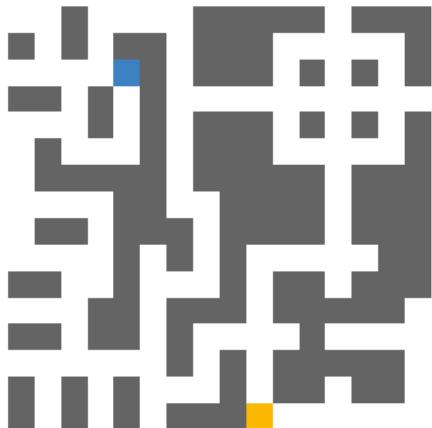
Okay the fix was to add a check in `self.if_intersection` to see if the goal is present in the percept

```

# If more than 2 options
if percept.count("0")>2 or "3" in percept:
    # It is a intersection
    return True
else:
    return False

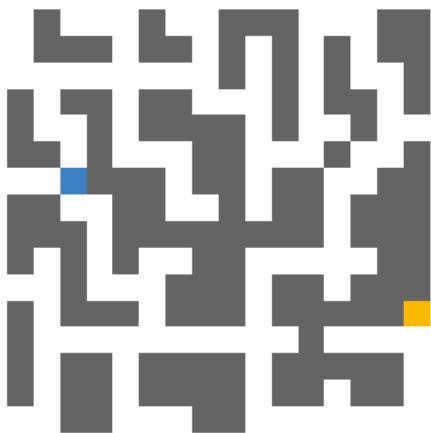
```

Loop on seed 423202041509



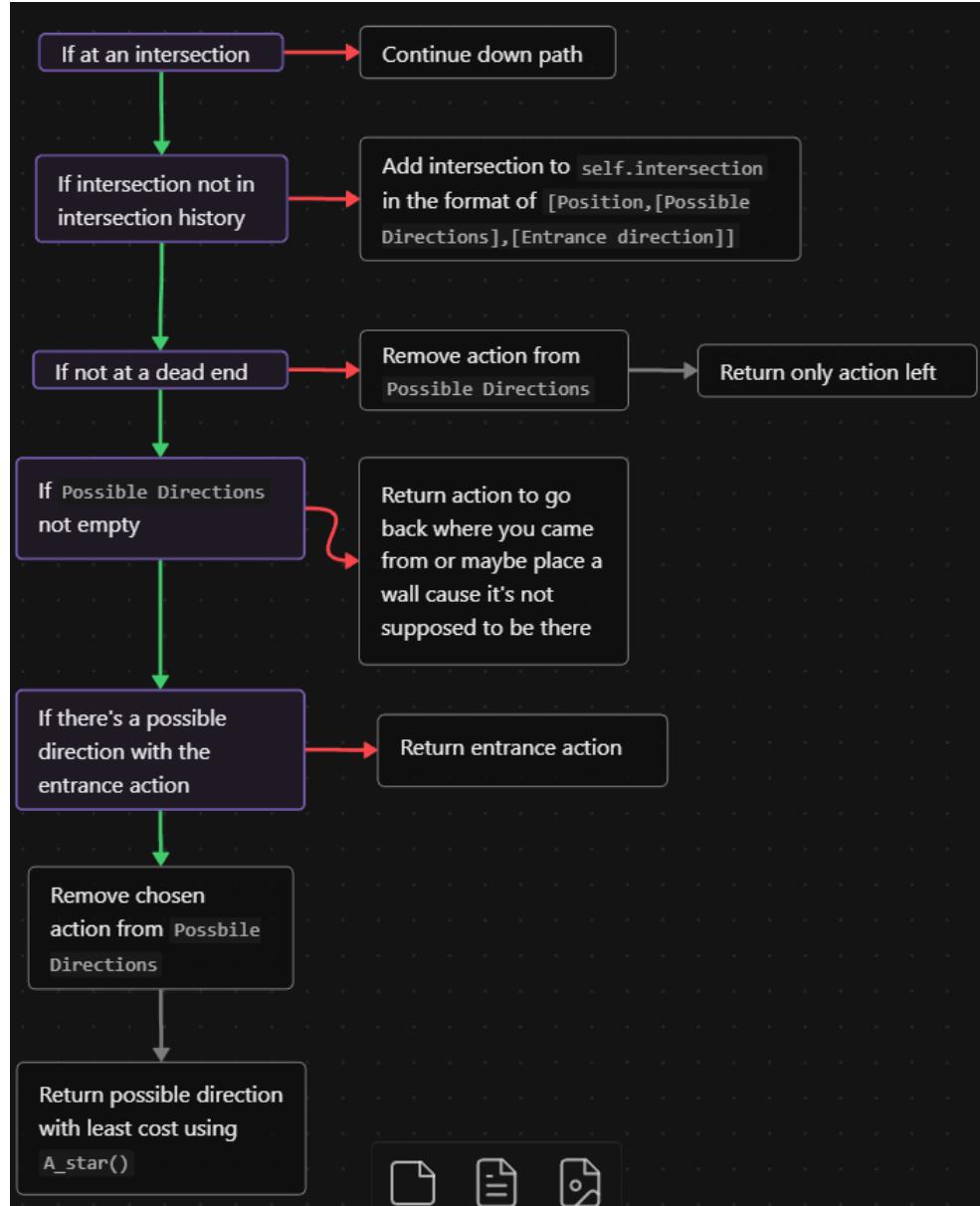
I think I'll replan my ~~Loop fix~~ flowchart

Seed with a good back and forth example



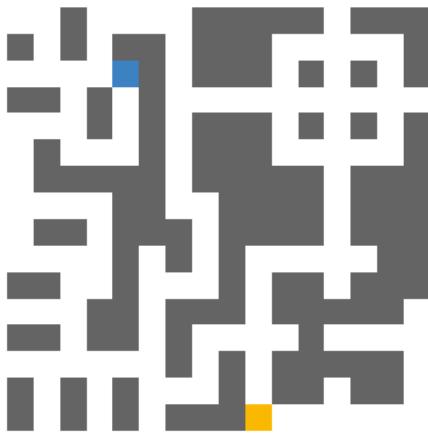
Seed 012206021115

New flowchart I got



Going to verify the flowchart before recoding it

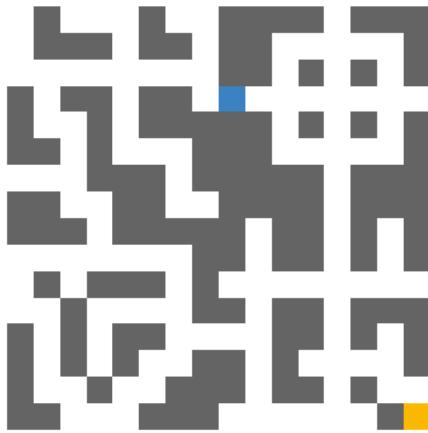
This seed fixed itself?? It was going to be the one where I fixed the loop. I have no idea why it's fixed



Seed 423202041509

I guess I'll find a new loop

Perfect



Seed 020303081515

Hm although my flowchart does not specifically look for loops, the way it works still should avoid loops by throwing the agent out of orbit. I'll code this flowchart as it's somewhat more simple, and if there's still loops then I'll rethink the plan. I just need to do some testing

Going to comment out this part of the code and try again from scratch

```
# Check if intersection has possible actions
print("interdata",interdata,"\\nactions",actions,"\\nentrance action",entrance_action,"\\ncurrent
position",current_position,"\\nentrance check",set(actions)^set(entrance_action),"\\npossible
actions",self.possible_actions(),"\\nset(self.possible_actions())",set(self.possible_actions()),"\\\nloop action",
(set(actions)^set(self.possible_actions()))&set(self.possible_actions()))
if actions: #
    # Check if it's a dead end
    if len(actions)==1: #
        # Return only option left
        return actions[0]
else:
    # Check if only entrance action left
```

```

if set(actions)==set(entrance_action): #⚠
    return entrance_action
else:

    # Loop check
    ## Check if current possible actions has an action that actions doesn't
    if (set(actions)^set(self.possible_actions())&set(self.possible_actions())):
        loop_action=(set(actions)^set(self.possible_actions())&set(self.possible_actions()))

        ## Block path
        # self.dead_ends.append(self.action_to_position(loop_action))

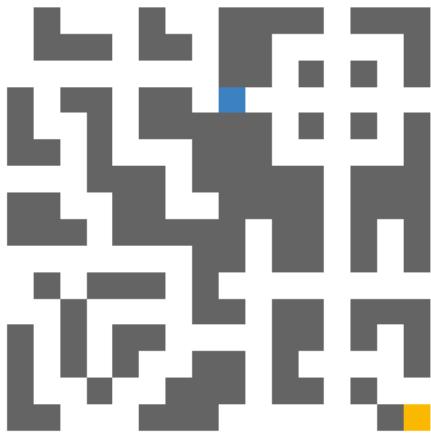
        ## Return action that isn't loop_action
        loop_feed=set(actions_no_ent)^set(loop_action)
        result=self.A_star(list(loop_feed))

        print("\nloop feed",loop_feed,"\nresult",result)
        return result

    # Passed checks to choose cheapest action
else:
    # Choose cheapest action using A*
    print("\nA star feed",list(actions_no_ent)) # Wait if there is only two options left it should choose the one that
isn't
    result=self.A_star(list(actions_no_ent)) #⚠ Shouldn't be empty since it shouldn't be only entrance action
    # Remove action from possible from self.intersection
    self.intersection[interi][1]=self.intersection[interi][1].replace(result,"") #✓
    print("\nresult",result,"\nintersection interi",self.intersection[interi])

```

Huh first try it basically worked exactly how I wanted to. Instead of detecting a loop, it encouraged the agent to try a different direction



The code I just did with inspiration from the last algorithm

```

# Main algorithm
# Check if at dead end
if len(self.possible_actions())==1:
    # Return only possible action
    result=self.possible_actions()[0]

    # Remove action from possible actions
    self.intersection[interi][1]=self.intersection[interi][1].replace(result,"")

    return result
else:
    # Check if possible actions is empty
    if not actions:
        # If empty then return action where agent came from
        return self.reverse_action(self.data["Move"].iloc[-1])
    else:
        # Check there's a possible action that is not the entrance action
        if actions_no_ent:

```

```

# Calculate lowest cost action
result=self.A_star(list(actions_no_ent))

# Remove chosen action from possible actions
self.intersection[interi][1]=self.intersection[interi][1].replace(result,"")

return result
else:
    # Else return entrance action
    result=entrance_action

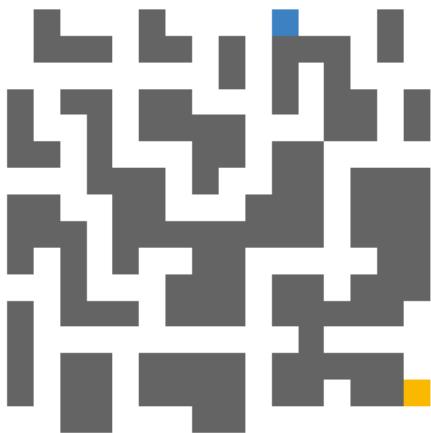
    # Remove entrance action from possible actions
    self.intersection[interi][1]=self.intersection[interi][1].replace(result,"")

return result

```

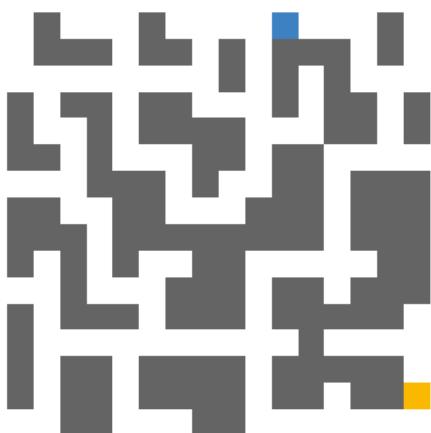
Let's try other seeds and see if there's a loop still. Maybe I can search for old seeds that had loops

Let's check the map where I found my first loop and see if it's fixed there



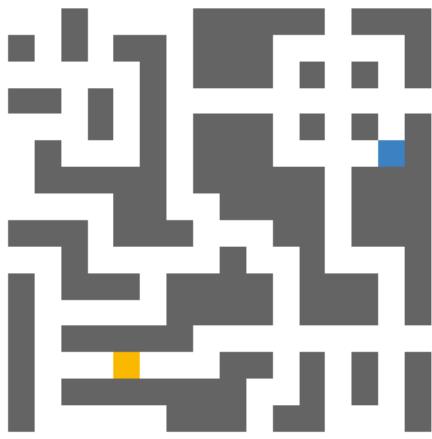
Seed 032200101415

Well this one doesn't really count cause for some reason it never touches the loop



I'll just do random seeds

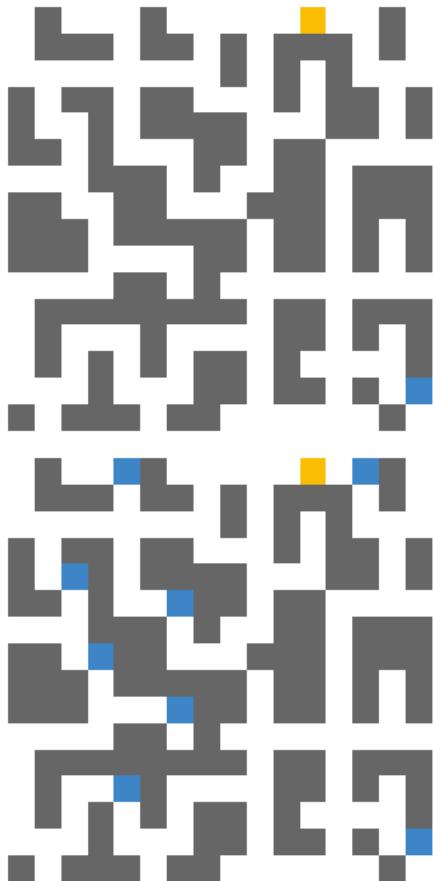
This one was about to loop but it didn't cause it wasn't allowed to repeat a direction. And noticed how when it returned to the intersection it entered the loop from, it didn't take the action that lead it back

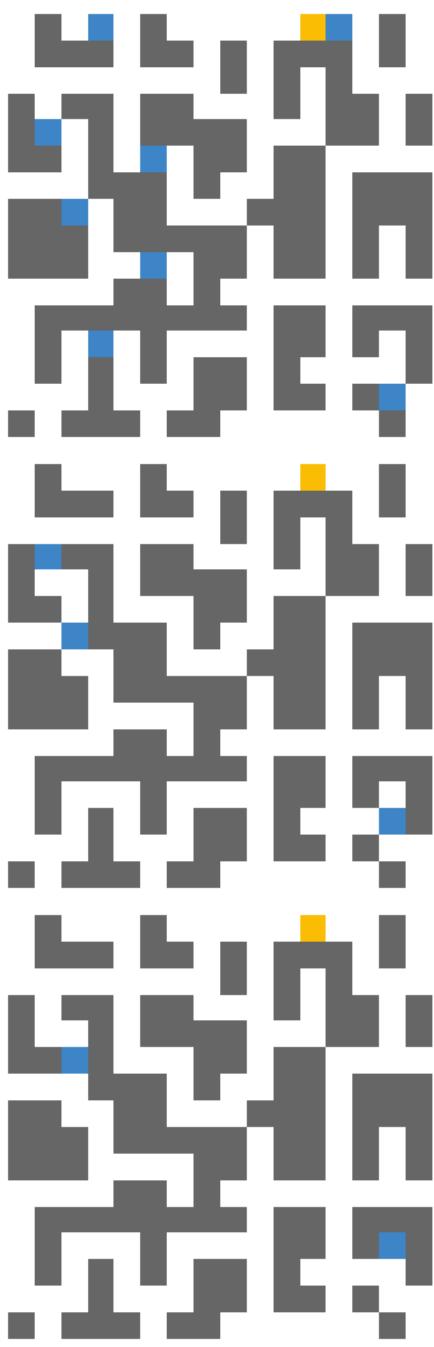


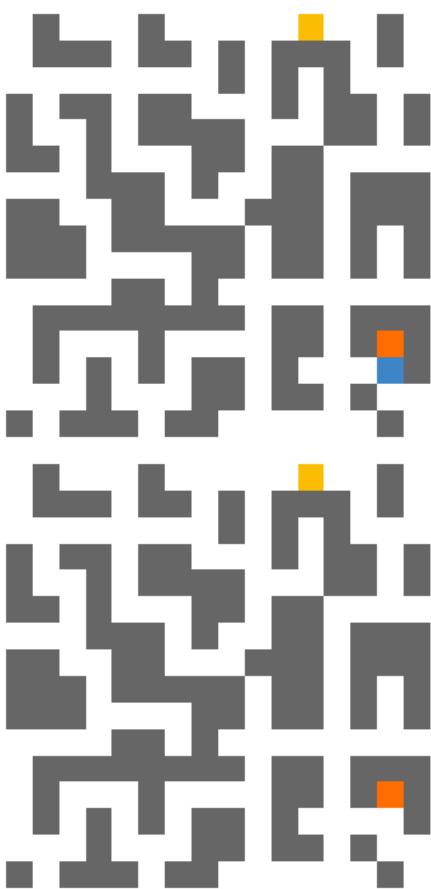
Seed 421405141304

Actually this one avoided two loops pretty well

Okay Seed 034314150011 dies since the agent wasn't stuck in a wall, but a wall was placed right before it was going to make a decision. That means I need to make a check where if it's future position will have a dead end, then remove it from possible actions and then follow through with the main algorithm







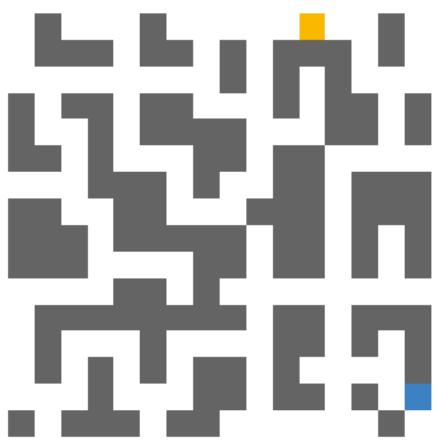
It seems the problem is how the entrance action `N` is removed instantly by the wall.

```
actions WS
entrance action N
actions_no_ent {'N', 'W', 'S'}
```

That means I have to update `actions_no_ent` if it's involves an action that is not possible

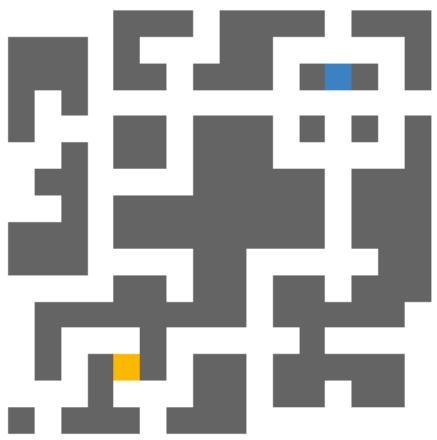
Okay this check fixed it but not sure if it will cause problems later. It shouldn't because of the specific length check

```
# Make sure actions and actions_no_ent are in sync
## If there's more options in actions_no_ent than actions
if len(actions_no_ent)>len(actions):
    # Remove extra action in action_no_ent
    for action in actions_no_ent:
        if action not in actions:
            # Only removes first action it finds
            actions_no_ent.remove(action)
            break
```



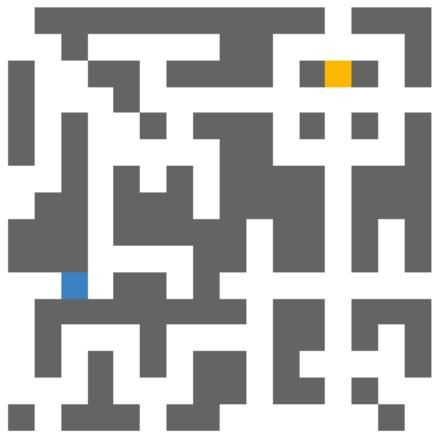
Seed 034314150011

Another loop prevented



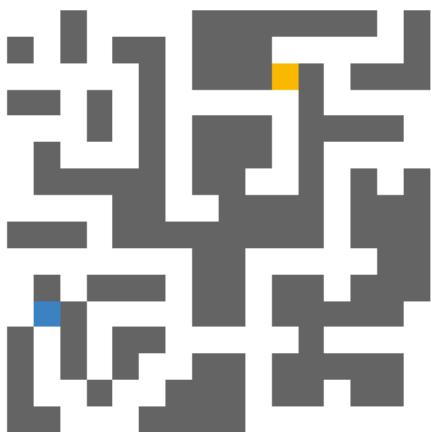
Seed 324202121304

It got out of this very spicy loop here as well



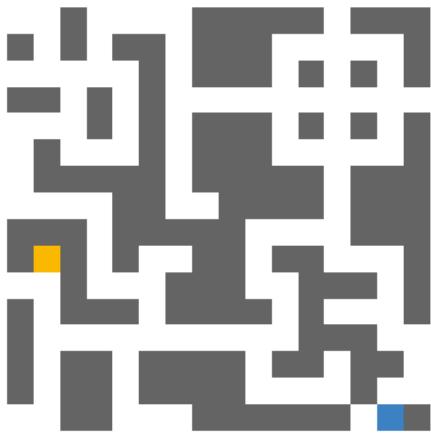
Seed 224310020212

Ate this loop up



Seed 440211010210

Holy moly, if it didn't get stuck here, then I think the loop problem is fixed. That is crazy. This took 191 moves



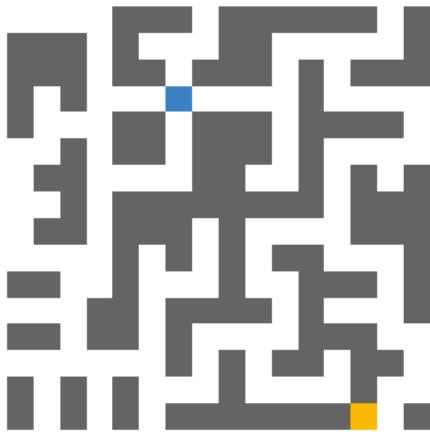
Seed 422015140901

Ran a test and at 207 iterations it has an error at seed 343003061513

It's a situation where the agent starts in an intersection

Simple fix

```
# If entrance_acion is None (agent starts at intersection)
if not entrance_action:
    return self.A_star(self.possible_actions())
```



Seed 343003061513

Running another 1000 benchmark test

Omg I'm done the code

The average cost after 1000 iterations is:

54.605

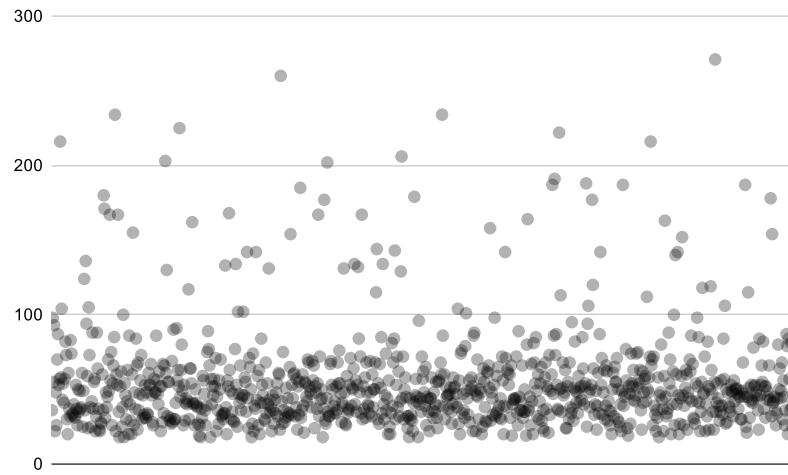
Moves [36, 98, 55, 93, 22, 48, 26, 70, 87, 53, 57, 216, 58, 104, 43, 54, 41, 57, 82, 73, 30, 20, 61, 37, 32, 83, 74, 33, 34, 40, 32, 31, 53, 37, 26, 50, 35, 49, 37, 38, 61, 25, 49, 124, 36, 136, 94, 59, 24, 105, 73, 38, 25, 42, 88, 37, 55, 57, 22, 32, 88, 64, 37, 23, 22, 28, 27, 60, 36, 180, 171, 31, 48, 38, 45, 70, 49, 167, 56, 75, 26, 54, 66, 85, 234, 22, 60, 53, 167, 35, 18, 52, 62, 25, 38, 100, 18, 54, 46, 30, 42, 63, 20, 86, 40, 29, 20, 60, 155, 26, 48, 37, 84, 66, 23, 68, 36, 47, 56, 73, 30, 46, 31, 33, 53, 31, 48, 34, 33, 61, 53, 53, 67, 46, 28, 26, 57, 48, 48, 86, 56, 47, 40, 62, 52, 22, 52, 28, 67, 34, 66, 203, 61, 130, 32, 55, 55, 29, 31, 27, 69, 28, 90, 30, 30, 41, 91, 56, 63, 52, 225, 45, 63, 80, 49, 26, 53, 29, 41, 44, 40, 42, 117, 30, 64, 64, 54, 162, 41, 40, 26, 50, 41, 39, 38, 27, 19, 26, 18, 37, 31, 59, 56, 47, 36, 26, 39, 75, 89, 77, 66, 18, 57, 40, 52, 46, 57, 52, 71, 22, 45, 36, 27, 56, 50, 70, 32, 33, 30, 34, 34, 133, 38, 20, 56, 52, 168, 63, 49, 30, 35, 47, 29, 28, 77, 134, 53, 25, 102, 47, 41, 66, 36, 44, 46, 102, 46, 58, 55, 35, 142, 44, 24, 71, 65, 50, 18, 27, 74, 22, 39, 37, 142, 43, 59, 26, 63, 20, 41, 84, 53, 46, 41, 30, 29, 68, 34, 54, 40, 131, 43, 35, 38, 22, 46, 36, 27, 43, 34, 48, 54, 32, 61, 60, 23, 260, 42, 27, 75, 36, 28, 31, 51, 35, 44, 32, 34, 65, 154, 33, 53, 44, 60, 61, 32, 38, 57, 23, 52, 33, 55, 185, 31, 36, 60, 39, 21, 55, 54, 45, 42, 70, 68, 52, 49, 47, 38, 69, 56, 66, 24, 52, 48, 37, 54, 167, 32, 72, 43, 50, 44, 18, 37, 177, 29, 27, 44, 202, 35, 32, 32, 67, 69, 53, 50, 34, 39, 42, 34, 52, 36, 68, 54, 76, 54, 47, 28, 41, 39, 131, 48, 27, 26, 59, 34, 37, 36, 53, 71, 39, 51, 50, 64, 134, 40, 56, 35, 53, 132, 84, 48, 72, 34, 167, 30, 52, 37, 58, 31, 68, 48, 40, 39, 34, 69, 48, 32, 60, 38, 50, 50, 68, 115, 144, 30, 54, 27, 67, 51, 85, 34, 134, 40, 30, 46, 74, 36, 44, 20, 56, 25, 20, 34, 81, 39, 68, 84, 143, 30, 53, 72, 38, 62, 32, 42, 129, 206, 44, 72, 30, 50, 58, 45, 32, 24, 32, 18, 44, 30, 33, 29, 22, 41, 179, 46, 57, 42, 45, 18, 96, 40, 44, 35, 72, 57, 62, 48, 39, 65, 26, 45, 43, 35, 22, 44, 44, 58, 36, 31, 48, 35, 40, 55, 31, 24, 55, 56, 39, 68, 60, 234, 86, 38, 60, 30, 52, 45, 46, 37, 47, 51, 35, 27, 40, 58, 51, 52, 41, 47, 38, 104, 27, 56, 20, 74, 76, 34, 47, 36, 31, 79, 101, 49, 44, 33, 56, 28, 50, 58, 52, 22, 86, 88, 49, 22, 46, 70, 45, 30, 50, 57, 48, 20, 25, 64, 55, 38, 29, 62, 61, 66, 51, 158, 29, 48, 44, 33, 53, 98, 32, 34, 59, 36, 65, 39, 31, 44, 30, 72, 56, 20, 39, 142, 71, 38, 63, 61, 53, 60, 31, 66, 19, 40, 43, 44, 43, 44, 72, 34, 26, 89, 45, 36, 42, 36, 66, 31, 35, 50, 35, 19, 80, 164, 36, 38, 45, 26, 67, 42, 20, 81, 68, 54, 85, 49, 32, 60, 30, 42, 54, 65, 29, 24, 36, 22, 58, 68, 37, 52, 50, 21, 73, 33, 49, 60, 187, 47, 86, 191, 73, 87, 53, 37, 36, 222, 50, 113, 35, 51, 68, 46, 48, 63, 24, 24, 52, 55, 30, 68, 38, 34, 95, 37, 29, 51, 82, 49, 41, 48, 34, 59, 51, 42, 52, 43, 85, 64, 40, 52, 54, 188, 25, 94, 106, 48, 38, 50, 53, 177, 120, 64, 33, 40, 49, 23, 39, 61, 34, 87, 142, 53, 71, 33, 48, 63, 30, 39, 66, 62, 29, 42, 36, 26, 35, 20, 50, 61, 32, 65, 55, 55, 71, 30, 50, 28, 25, 40, 22, 49, 187, 39, 45, 47, 77, 56, 24, 19, 52, 47, 27, 74, 54, 52, 63, 43, 42, 46, 74, 51, 75, 56, 54, 63, 36, 23, 62, 41, 48, 57, 59, 42, 112, 50, 48, 58, 72, 216, 66, 70, 26, 72, 39, 33, 22, 36, 31, 58, 18, 46, 48, 80, 43, 37, 22, 59, 163, 45, 32, 34, 37, 88, 54, 70, 20, 49, 40, 62, 100, 26, 140, 60, 40, 142, 46, 20, 27, 44, 59, 152, 46, 26, 39, 51, 63, 60, 23, 48, 44, 70, 62, 86, 42, 67, 31, 30, 27, 54, 42, 98, 52, 85, 41, 22, 35, 55, 118, 47, 48, 72, 22, 40, 32, 82, 52, 23, 50, 119, 63, 47, 35, 23, 55, 271, 35, 41, 54, 44, 39, 29, 35, 56, 39, 33, 84, 57, 106, 27, 50, 28, 48, 40, 38, 35, 53, 49, 48, 32, 56, 42, 50, 51, 47, 50, 37, 45, 45, 30, 30, 49, 68, 47, 45, 187, 25, 21, 45, 115, 51, 31, 42, 53, 43, 41, 78, 52, 53, 48, 46, 44, 36, 23, 84, 46, 53, 66, 55, 82, 37, 52, 31, 52, 49, 43, 45, 66, 43, 178, 32, 154, 41, 42, 45, 36, 26, 34, 44, 80, 50, 64, 44, 49, 68, 30, 56, 45, 26, 36, 87, 79, 20, 84, 57, 36, 81, 38, 25, 47, 41, 42, 142, 37, 29, 38, 30, 34, 19, 64, 42]

Facts

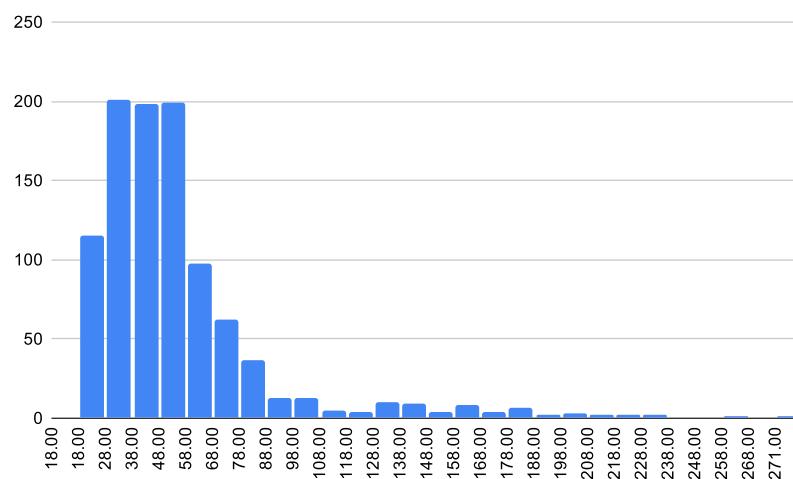
Max	Min	Total
271	18	54605

I don't know how the max was 271. I wish I knew what seed it was. I mean, how can an agent even move that much and not be in a loop and also find the goal?

The plot in a scatterplot



The plot in a histogram with a bucket size of 10



Also could be found on

Here is interactable data

[scatterplot.html](#)

[histogram.html](#)

Going to clean up code's comments and all

Turns out I never used this massive block of code

```
def loop_check_v2(self, possible_actions=None):
    # Check if previous and current position was approach from same angle before
    # previous_position=self.data["Positions"].iloc[-1][0]
    # current_position=self.positions[0]
    # previous_last_move=self.data["Move"].iloc[-2]
    # current_last_move=self.data["Move"].iloc[-1]

    # Check if current position in data
    # for i,position in reversed(list(enumerate(self.data["Positions"]))):
    #     # Make sure there's one position
    #     if len(position)==1:
    #         # And the position before that
    #         if len(self.data["Positions"].iloc[i-1])==1:
```

```

# If current position is the same as previous loop's position
if current_position==self.data["Positions"].iloc[i][0]:
    # If the previous loop's previous position is also the same as current's previous position
    if previous_position==self.data["Positions"].iloc[i-1][0]:
        # Check if current last move is same as previous loop's last move
        if current_last_move==self.data["Move"].iloc[i-1]:
            # Same for previous last move with previous loop's previous last move
            if previous_last_move==self.data["Move"].iloc[i-2]:
                # Last check is to make sure we can place wall in front of current agent based on previous
loop's future
    #
    # wall_position=self.data["Positions"].iloc[i+2][0]
    # if self.state[wall_position[0]][wall_position[1]]==0:
    #     # Add wall to interrupt the loop
    #     self.dead_ends.append(wall_position)

#
#             return True

current_position=self.positions[0]

# Check all positions around current position
possible_positions=[]
for action in self.possible_actions():
    possible_positions.append(self.action_to_position(action))

# Check if agent approached possible position from all possible angles
for position in possible_positions:
    # Find all possible ways you can enter possible position
    possible_actions_reversed=[]
    for action in self.possible_actions(position):
        possible_actions_reversed.append(self.reverse_action(action))

    # Find all previously entered action into possible position
    previous_actions=[]
    for i,data_position in enumerate(self.data["Positions"]):
        # Make sure we are checking after single_agent()
        if len(data_position)==1:
            # If possible position matches position in data
            if position==data_position[0]:
                # Append previous action
                previous_actions.append(self.data["Move"].iloc[i-1])

    # If all the available ways to enter the position has been done, place a wall
    if set(possible_actions_reversed)==set(previous_actions):
        self.dead_ends.append(position)

# Check each possible position's previous entering action
forbidden_actions=[]
for position in possible_positions:
    for i,data_position in enumerate(self.data["Positions"]):
        # Make sure we are checking after single_agent()
        if len(data_position)==1:
            # If possible position matches position in data
            if position==data_position[0]:
                # Only forbid action if agent is in same position as when previously entered
                if current_position==self.data["Positions"].iloc[i-1][0]:
                    # Add forbidden action
                    forbidden_actions.append(self.data["Move"].iloc[i-1])

# if self.positions[0]==[3,9]:
#     print("forbidden actions",forbidden_actions,"\\npossible actions",possible_actions,"\\npossible
positions",possible_positions)
# Remove fobidden actions from possible actions if possible
if possible_actions:
    for action in forbidden_actions:
        # If agent can do the forbidden action
        if action in possible_actions:
            # Remove action
            possible_actions.remove(action)

```

```

else:
    pass
# Return new possible actions
return possible_actions
else:
    # Return forbidden actions. These actions should not be available for the agent to do
    return forbidden_actions
# return True

```

Also never used `goal_snipe()`

```

def goal_snipe(self):
    # percept=""
    position=self.positions[0]
    # state=copy.deepcopy(self.initial_state)

    # i=position[0]
    # j=position[1]

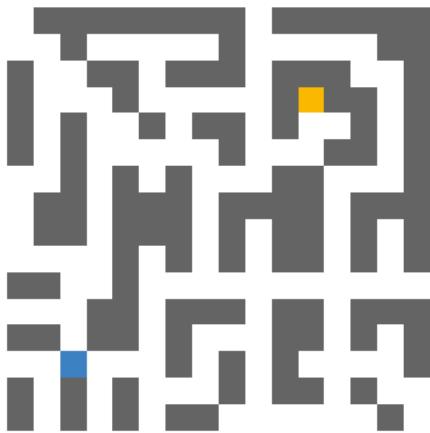
    # #N
    # percept+=str(state[i-1][j])
    # #E
    # percept+=str(state[i][j+1])
    # #W
    # percept+=str(state[i][j-1])
    # #S
    # percept+=str(state[i+1][j])

    # print("SNIPEEE",percept)
    # if "3" in percept:
    #     for i,number in enumerate(percept):
    #         print("NUBMER",number)
    #         if number=="3":
    #             if i==0:
    #                 if i==0:
    #                     return "N"
    #                 if i==1:
    #                     return "E"
    #                 if i==2:
    #                     return "W"
    #                 if i==3:
    #                     return "S"

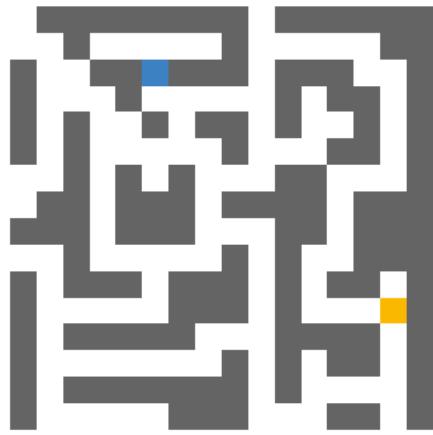
    for action in self.possible_actions():
        if self.action_to_position(action)==self.goal:
            return action

```

Very good loop stop

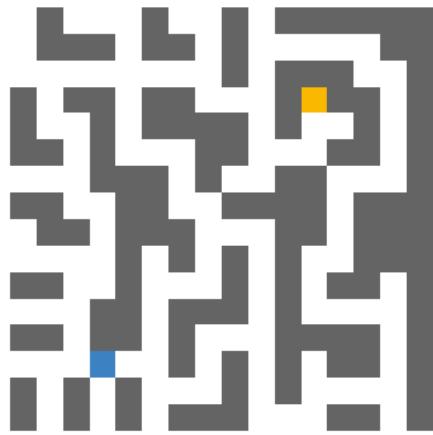


Ahhh so gooood



Seed 201102051114

Jesus



Seed 003113030311