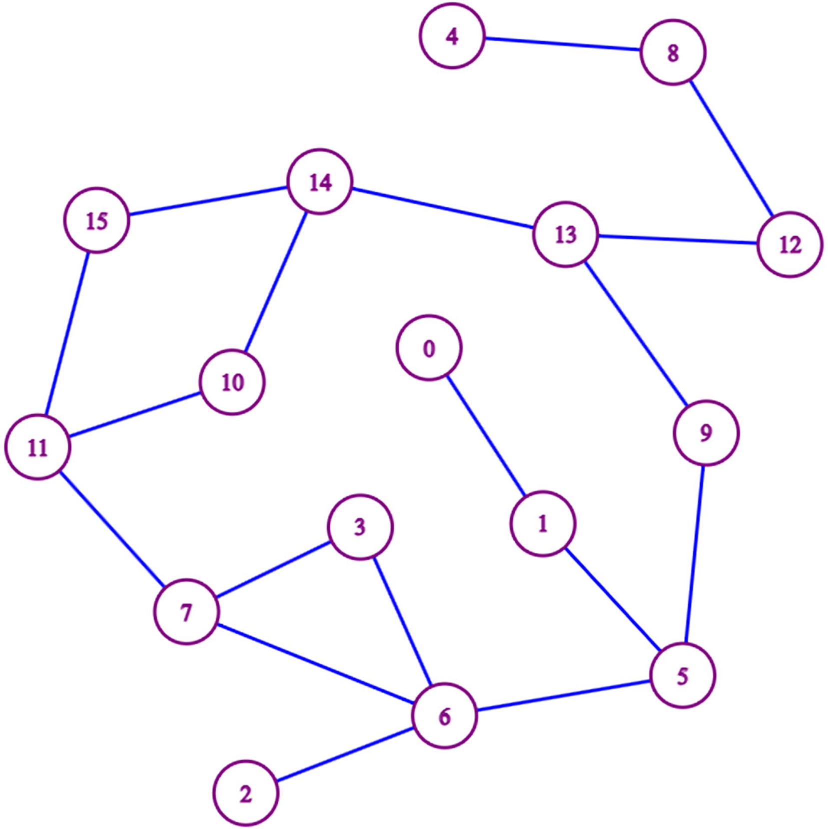
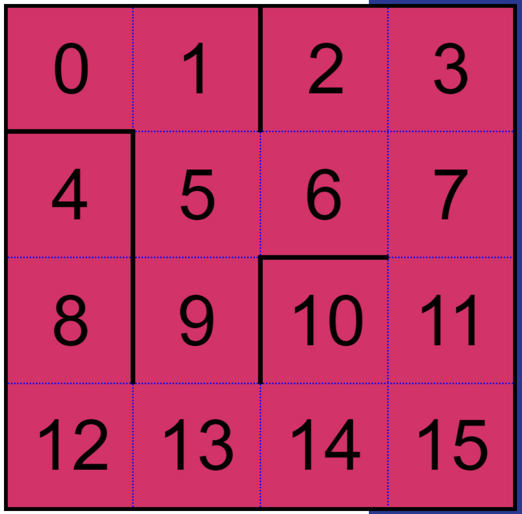
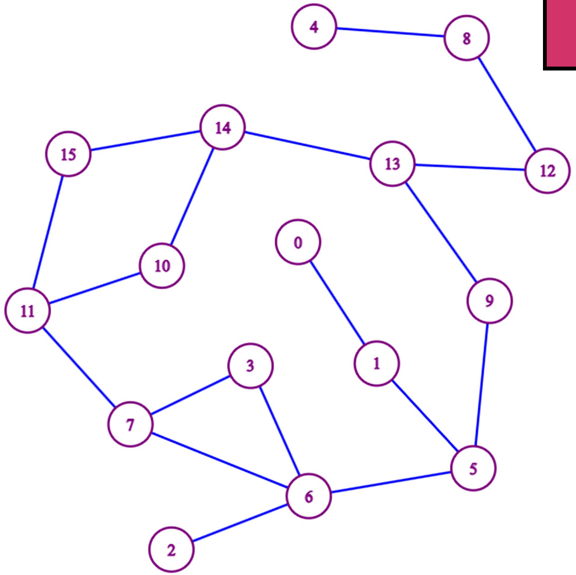
Algoritmos de grafos



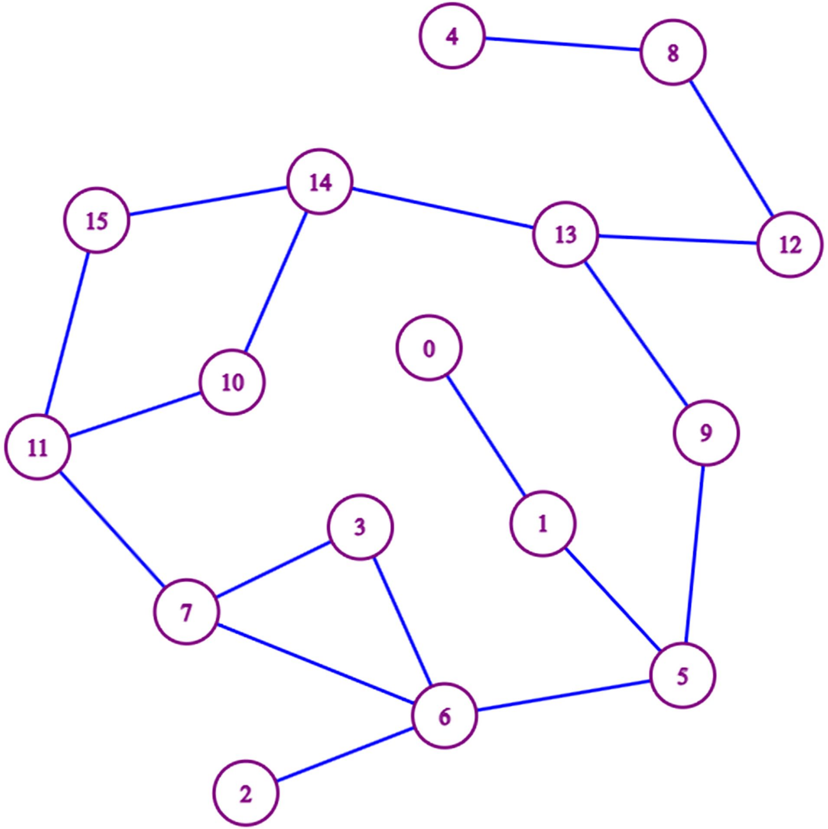
Grafos são compostos por vértices/nós e arestas





Podem representar muitas coisas:

- Ruas e esquinas
- Amizades
- Tiles de um labirinto

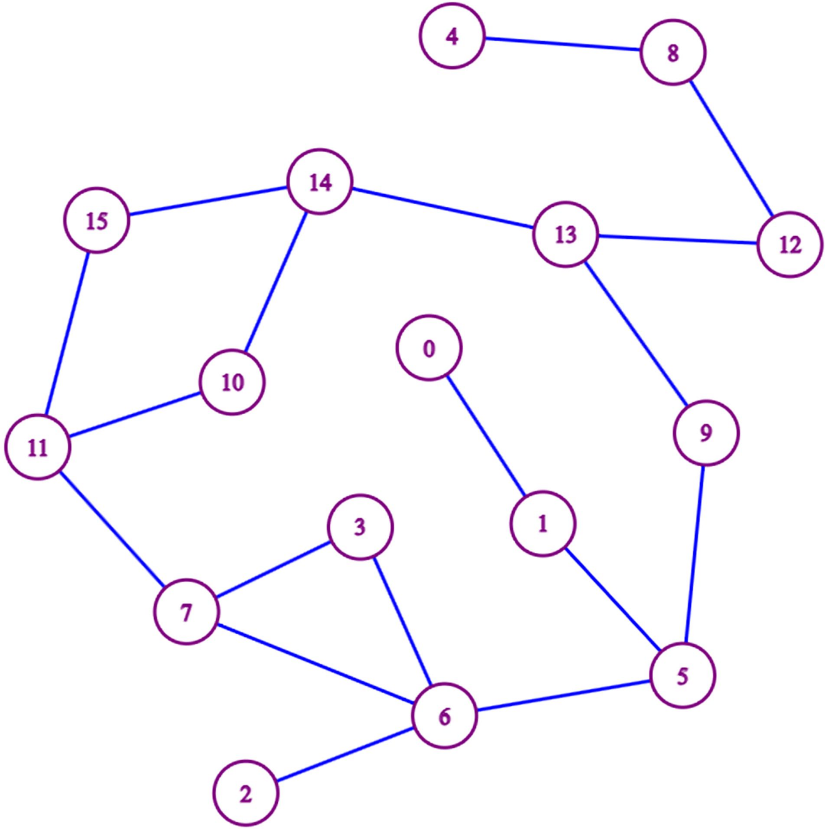


Problema fundamental:

Saber andar por um grafo!

Pra resolver isso, precisamos botar o gráfico dentro do robô…

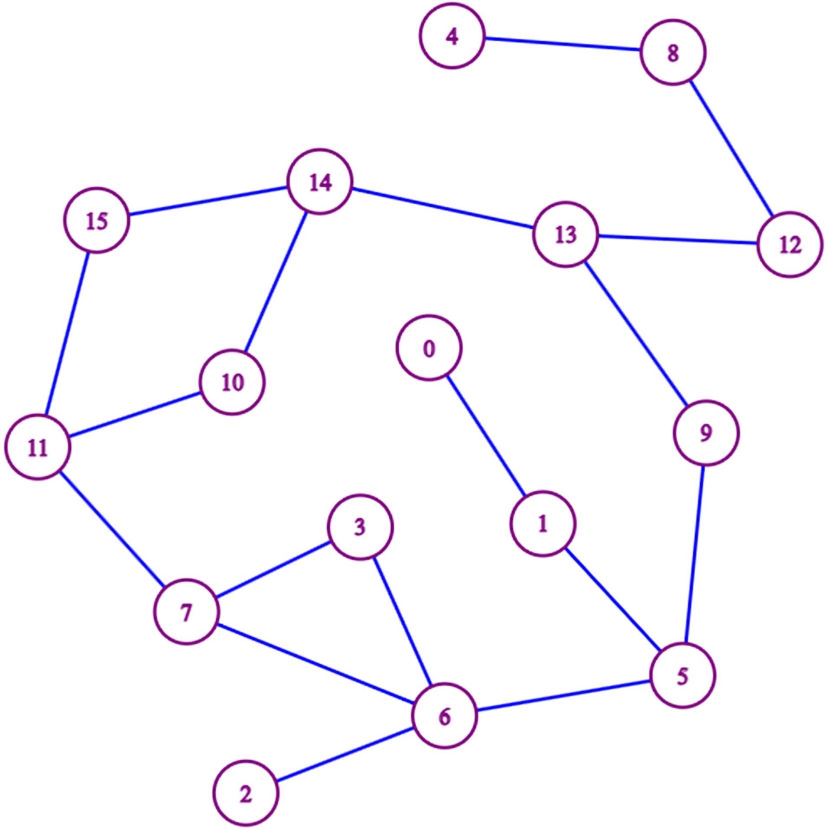
Como construir um grafo?



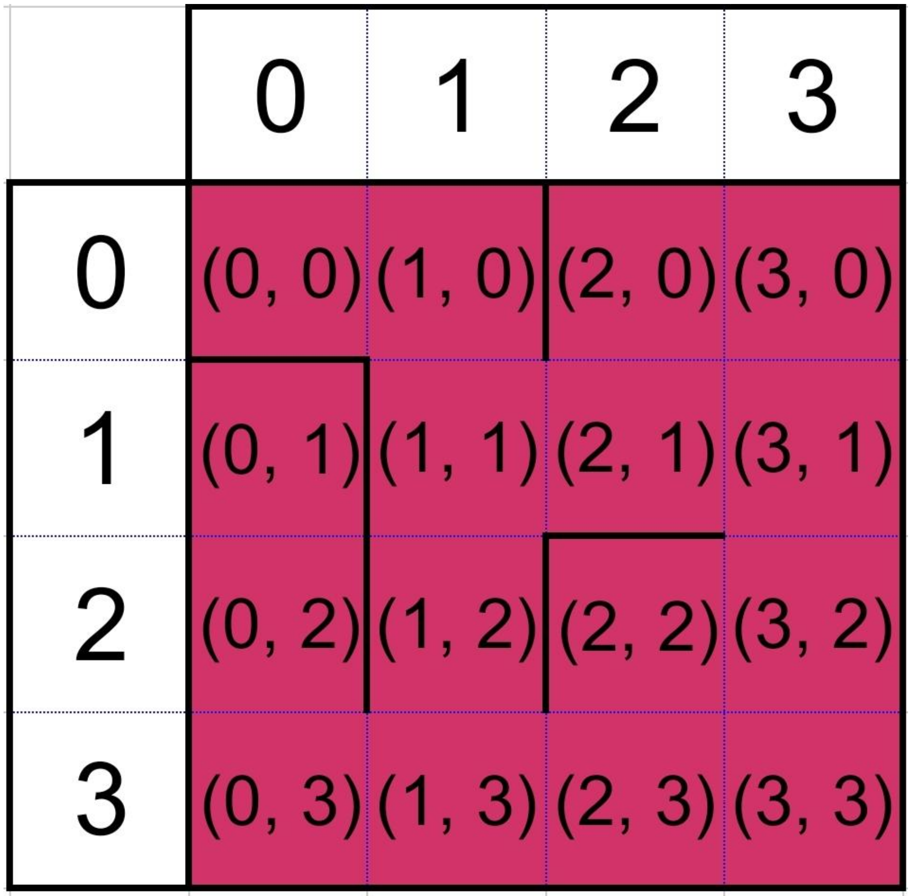
Armazenamos as arestas numa lista de listas (matriz), `vizinhos`:

0	1		
1	0	5	
2	6		
3	6	7	
4	8		
5	1	6	9

Por exemplo: `vizinhos[7]` dá a lista de vizinhos do 7



Essa representação basta!



Para o caso em que o grafo na verdade representa um labirinto, representamos os nós como coordenadas, `(i, j)`.

NB! a gente não precisa armazenar o nome do próprio tile! Basta saber que, no mapa (matriz), ele é acessado na coordenada `(i, j)`.

Ah! não importa se a primeira coordenada é horizontal ou vertical, desde que sejamos consistentes.

di = 1, dj = 0

+i	0	1	2	3
0	1	0	1	0
1	0	1	1	0
2	0	0	1	0
3	1	1	1	0

di = 0, dj = -1

-j	0	1	2	3
0	0	0	0	0
1	0	1	1	1
2	1	1	0	1
3	1	1	1	1

-i	0	1	2	3
0	0	1	0	1
1	0	0	1	1
2	0	0	0	1
3	0	1	1	1

di = -1, dj = 0

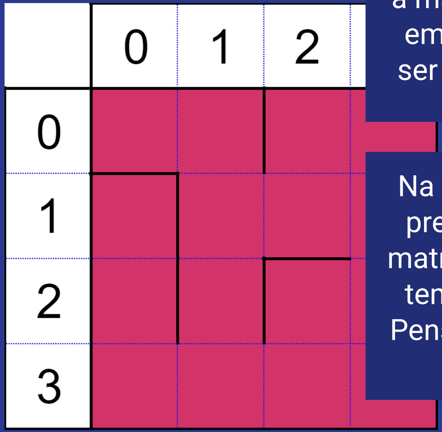
+j	0	1	2	3
0	0	1	1	1
1	1	1	0	1
2	1	1	1	1
3	0	0	0	0

di = 0, dj = 1

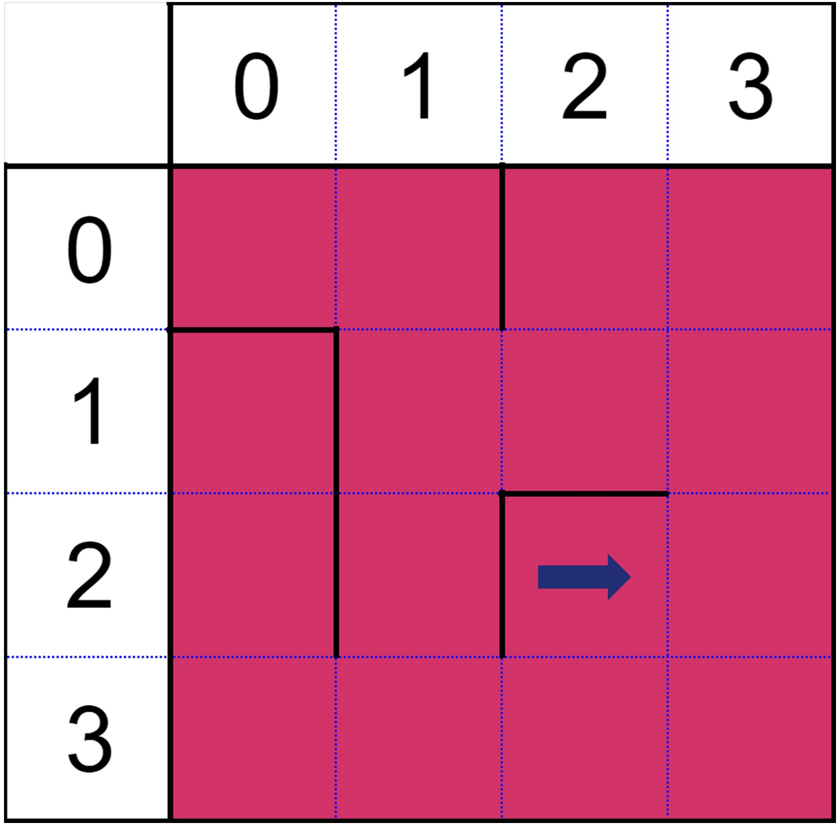
Para armazenar as conexões entre os tiles, podemos usar várias matrizes empilhadas!

Declaramos um `grafo[i][j][di][dj]`, que registra se o nó `(i, j)` está ligado ao que está em `(i + di, j + dj)`.

Sim, vamos usar índices negativos! Para isso, declaramos a matriz com 3 níveis em di e dj (o [-1] vai ser equivalente a [2], nesse caso)



Na prática, você não precisa visualizar a matriz (até porque ela tem 4 dimensões!). Pense como se fosse uma função!

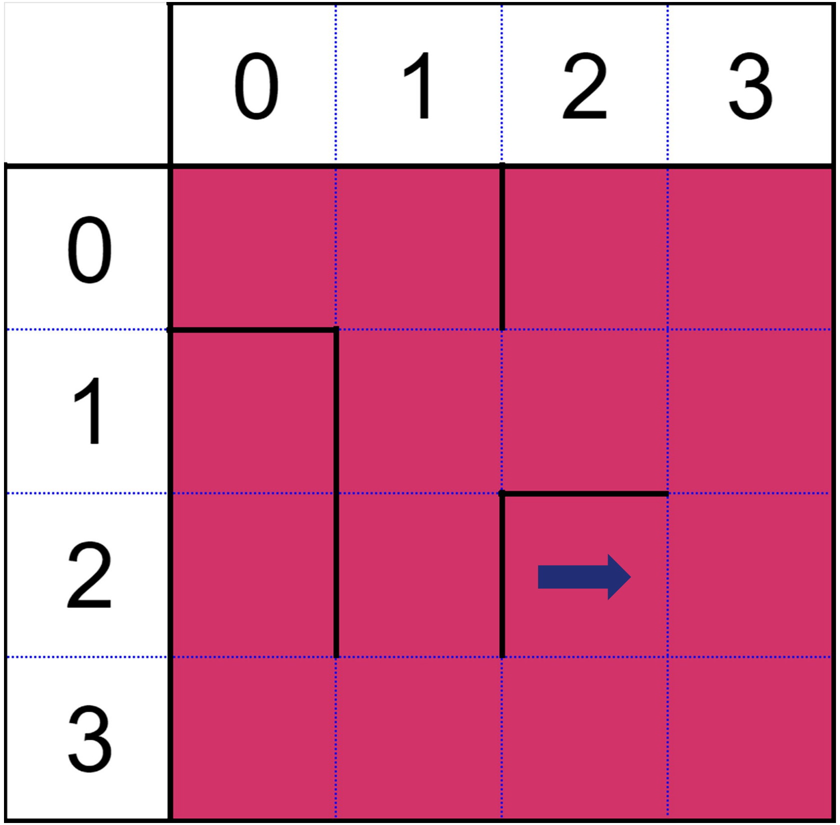


Já sabemos que estrutura de dados montar! Agora precisamos coletar os dados pra ela.

No labirinto, o robô começa em um lugar arbitrário, numa posição arbitrária. Ele é essa setinha desenhada.

Ah! Um treco que vai auxiliar nossas contas são as seguintes listas:

```
dx = [1, 0, -1, 0]
dy = [0, -1, 0, 1]
```




```
dx = [1, 0, -1, 0]
dy = [0, -1, 0, 1]
```

Sempre que quisermos guardar o que o robô viu em sua frente, vamos usar

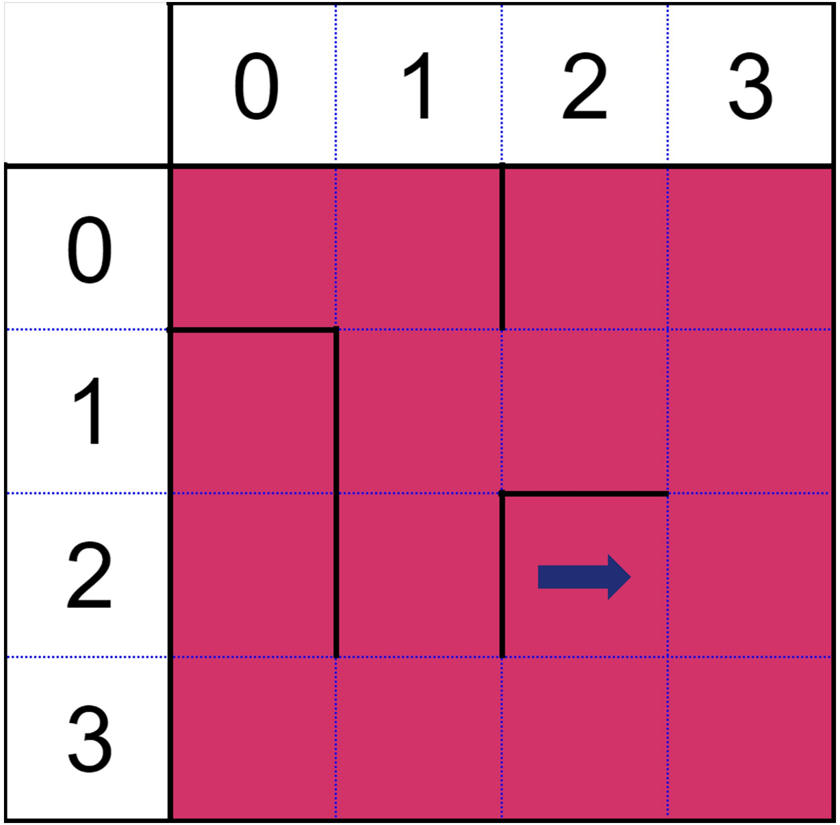
`di = dx[0] e dj = dy[0].`

Se quisermos guardar o que ele viu na esquerda,

`di = dx[1] e dj = dy[1].`

Atrás, o 2, e na direita o 3.

Isso garante que quando fizermos i+di e j+dj, vai ficar nas posições certinhas.




```
dx = [1, 0, -1, 0]
dy = [0, -1, 0, 1]
```

Exemplo: Na posição ilustrada, o robô lê:

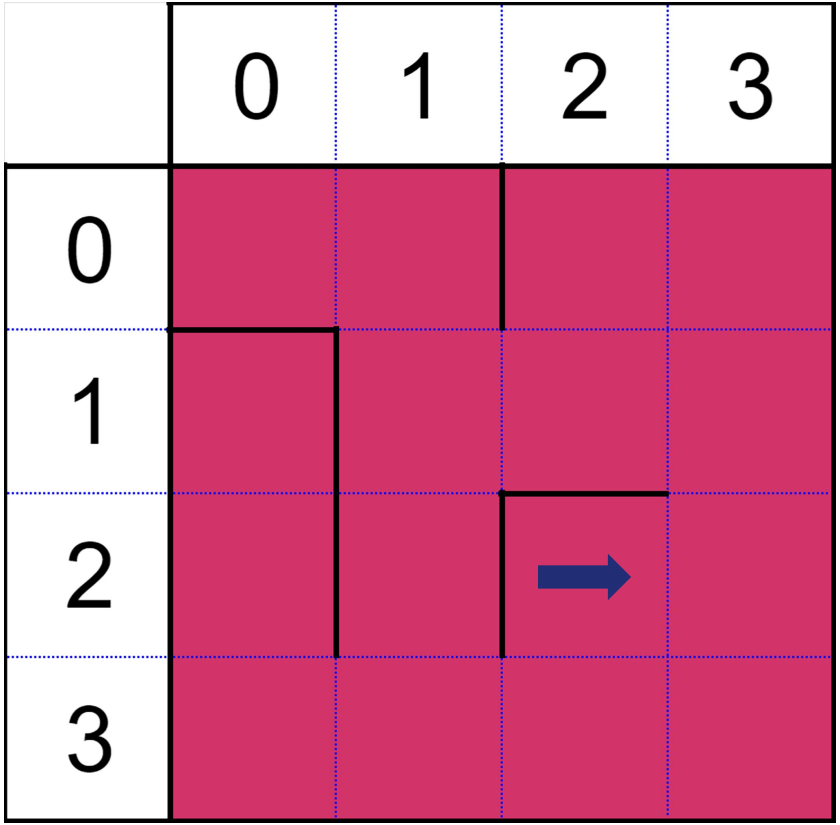
Para frente:

`grafo[2][2][dx[0]][dy[0]]`
`== grafo[2][2][1][0] = 1`

Para esquerda:

`grafo[2][2][dx[1]][dy[1]]`
`== grafo[2][2][0][-1] = 0`

Mas, se sabemos que (2, 2) não conecta com (2, 1), podemos atualizar `grafo[2][1]` (as marcações do vizinho) também!




```
dx = [1, 0, -1, 0]
dy = [0, -1, 0, 1]
```

Então, escrevemos:

```
grafo[2][1][0][1] = 0
```

Em geral, a atualização do vizinho na direção k fica assim:

Usamos k+2 para ir pra direção oposta ao tile atual, e %4 pra ciclar o índice de volta pro começo se o oposto estiver antes

```
vi = i + dx[k]
vj = j + dx[k]
vdi = dx[(k + 2)%4]
vdj = dy[(k + 2)%4]
grafo[vi][vj][vdi][vdj] = 0
```
ou `1`, dependendo se tiver uma parede ou não.

di = 1, dj = 0

+i	0	1	2	3
0				
1				
2		0	1	
3				

di = 0, dj = -1

-j	0	1	2	3
0				
1				
2			0	
3			1	

-i	0	1	2	3
0				
1				
2			0	1
3				

di = -1, dj = 0

+j	0	1	2	3
0				
1			0	
2			1	
3				

di = 0, dj = 1

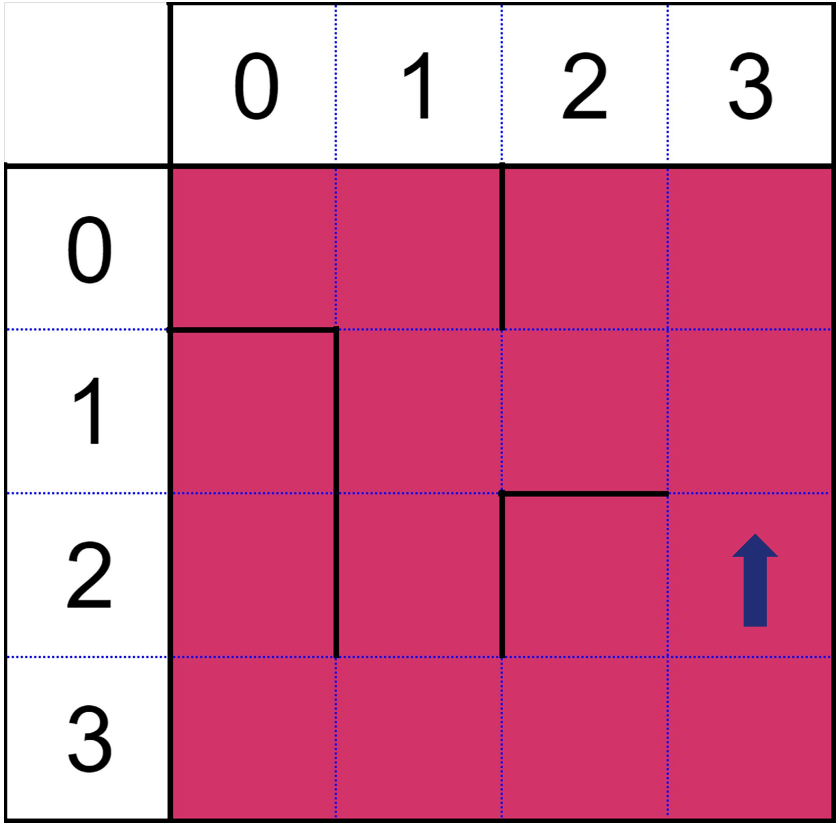
```
dx = [1, 0, -1, 0]
dy = [0, -1, 0, 1]
```

Finalmente, após olhar para todas as 4 direções, as matrizes atualizadas ficam como está à esquerda.

Na prática, pode ser que seu robô olhe só para 3 direções. Tá tudo bem, uma hora ele vai completar o mapinha.

Visualizar tudo isso acontecendo é muito difícil, mas fica mais fácil se você abstrair todos os desenhinhos e pensar só nas coordenadas, sem se preocupar com a orientação delas (isto é, focar no código).

No fim das contas, o importante é ser consistente nas suas convenções.




```
dx = [1, 0, -1, 0]
dy = [0, -1, 0, 1]
```

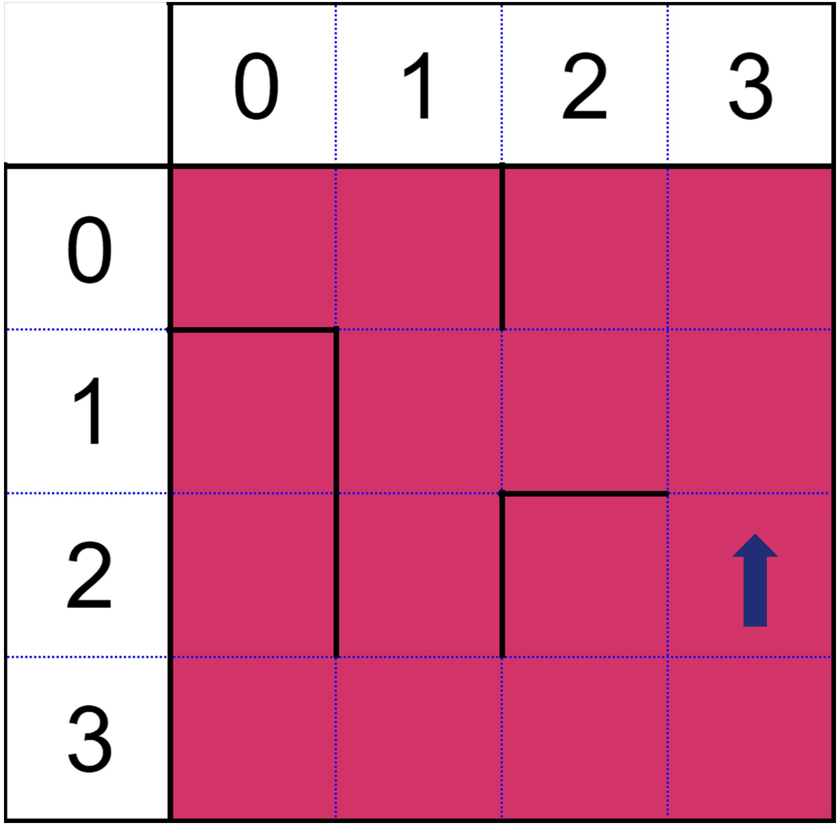
Ah, não, o robô girou! Agora as convenções de dx[0], dy[0] → frente, etc., foram por água abaixo! Por exemplo, dx[0] e dy[0] agora correspondem à direita do robô!

Na verdade é bem simples de resolver:

A gente rotaciona o dx e o dy também!

dx: `[1, 0, -1, 0]` → `[0, -1, 0, 1]`
dy: `[0, -1, 0, 1]` → `[-1, 0, 1, 0]`

Fazendo isso, atualizações feitas com os sensores vão cair nas posições certas da matriz e a convenção se mantém.




```
dx = [0, -1, 0, 1]
dy = [-1, 0, 1, 0]
```

Exemplo: Na posição ilustrada, o robô lê:

Para frente:

```
grafo[3][2][dx[0]][dy[0]]
== grafo[3][2][0][-1] = 1
```

Para esquerda:

```
grafo[3][2][dx[1]][dy[1]]
== grafo[3][2][-1][0] = 1
```

Também atualizamos os vizinhos da mesma forma de antes, como se nada tivesse acontecido.

di = 1, dj = 0

+i	0	1	2	3
0				
1				
2		0	1	0
3				

di = 0, dj = -1

-j	0	1	2	3
0				
1				
2			0	1
3			1	1

-i	0	1	2	3
0				
1				
2			0	1
3				

di = -1, dj = 0

+j	0	1	2	3
0				
1			0	1
2			1	1
3				

di = 0, dj = 1

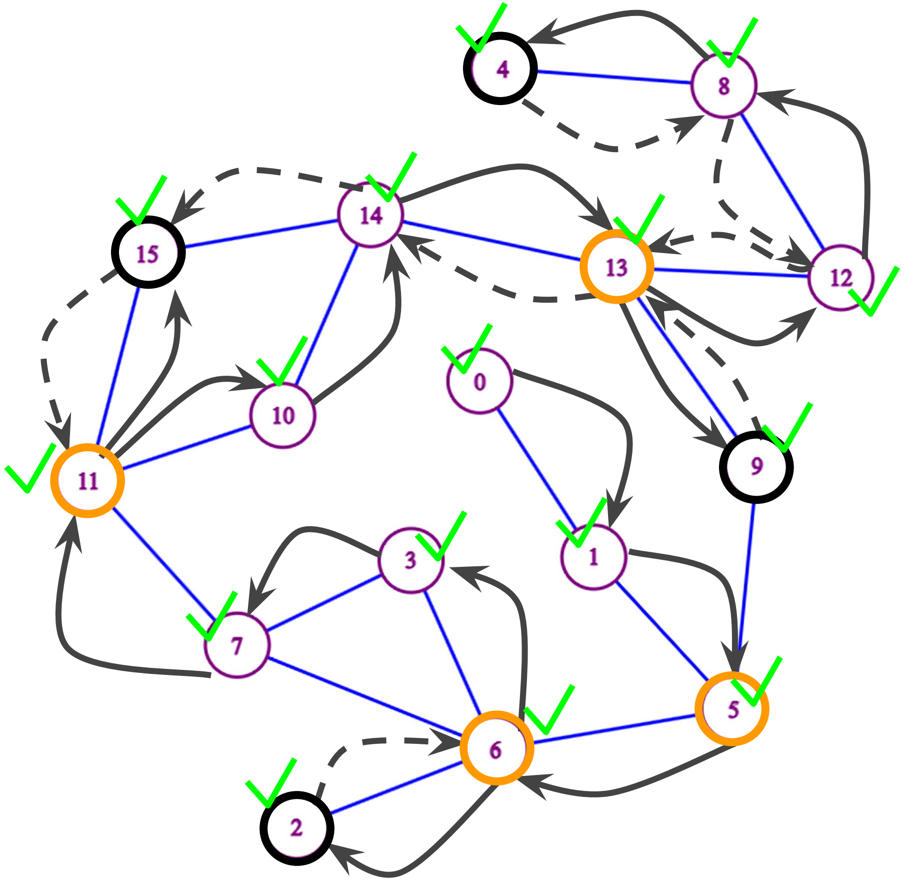
```
dx = [0, -1, 0, 1]
dy = [-1, 0, 1, 0]
```

Após olhar para todas as 4 direções, as matrizes atualizadas ficam como está à esquerda.

Note que existe um pouco de sobreposição (p.ex. ele confirmou que não existia uma parede com o (2, 2)), mas isso não tem problema.

E, assim, o robô consegue ir registrando o labirinto que ele enxerga. Agora, a gente precisa ver algoritmos pra ele andar pelo labirinto usando os registros que ele já fez!

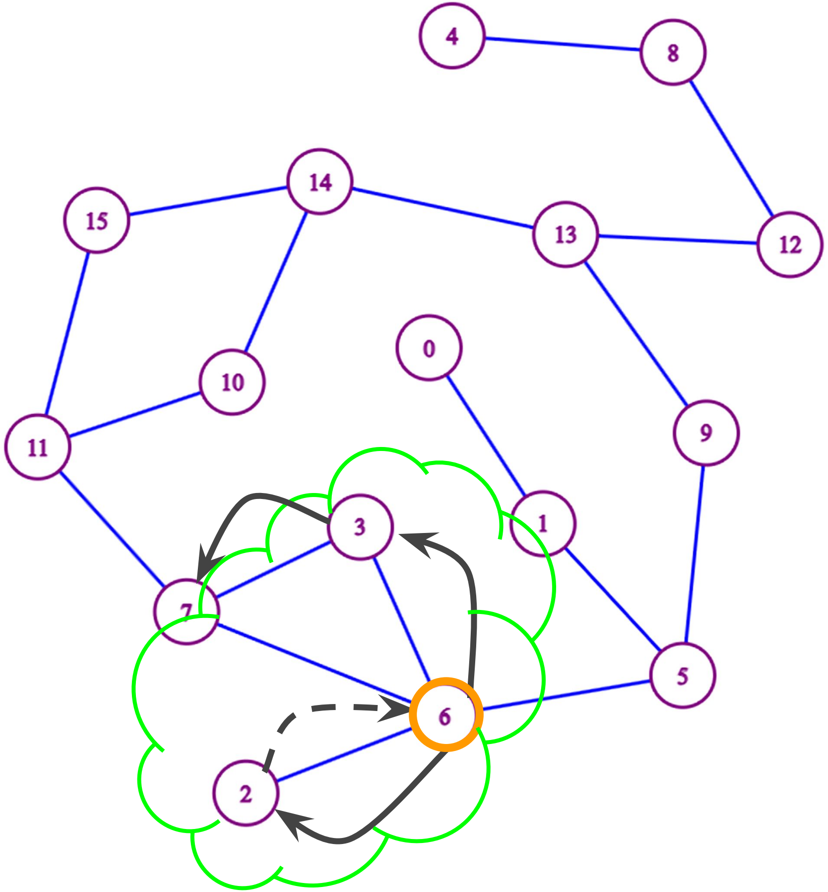
Depth-First Search (DFS)



A DFS é uma busca em profundidade: ela toma o primeiro caminho que acha e vai até o fundo.

Quando ela tá numa bifurcação, toma qualquer caminho e segue.

Quando chega numa ponta, volta pra bifurcação mais recente.

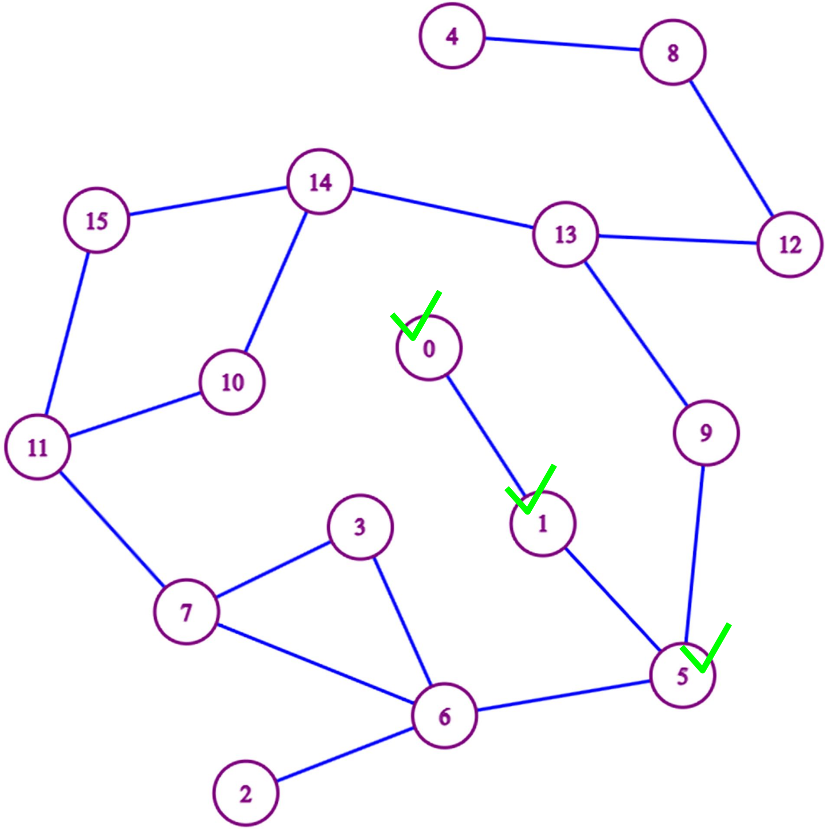


Essencialmente, a gente repete o mesmo código em cada nó!

1. Marca que já visitou
2. Passa pelos vizinhos e entra neles se não foram visitados
3. O vizinho em que entramos vira o nó atual, e repetimos o processo!

É uma recursão!

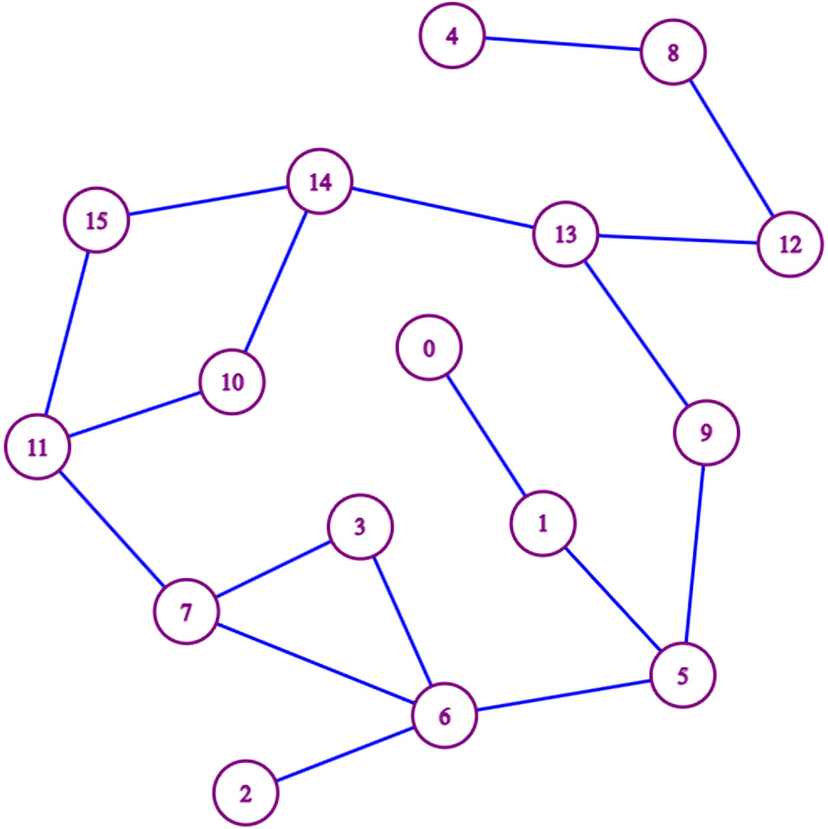
O código



Armazenamos os nós visitados em uma lista, `marc`:

`marc`

0	1	5



O código, finalmente:

```
def dfs(atual):

    marc.append(atual)

    for vizinho in vizinhos[atual]:

        if vizinho not in marc:
            dfs(vizinho)
```