

Explorador de Labirintos

Explicação da Ideia

Após um brainstorm de ideias, para decidir qual problema trataríamos, passando mapeação de objetos 3D usando voxels e calculador de colisão de laser em triângulo num espaço 3D, decidimos adotar a ideia de um **Explorador de Labirintos**.

O explorador de labirintos recebe um labirinto formatado em um texto onde os valores “1” representam paredes, os valores “2” os pontos de início e fim, e os valores “0” os pontos que compõem as partes navegáveis do labirinto.



Tópicos

01. Soluções

02. Análise

03. Conclusão

Ideia por trás das soluções

Com a divisão do grupo em duplas, desenvolvemos os algoritmos de forma separada, o que, consequentemente, levou à algumas diferenças nas soluções abordadas pelas duas duplas.

A única ideia preservada entre ambas as duplas, foi o formato em que o labirinto é lido, sendo ele uma lista das linhas do labirinto, sendo assim, uma lista de listas



Recursiva

A função **navigate** começa no ponto de partida, explorando os vizinhos de cada célula. A cada movimento, uma nova chamada de *navigate* é empurrada à pilha de memória.

(1,2)	(2,2)	(3,2)
(1,3)	(2,3)	(3,3)
(1,4)	(2,4)	(3,4)

```
path = [[0,1], [1,1], [2,1], [2,2], ...]
```

Iterativa

O algoritmo é capaz de iterar pelos pontos de intersecção do labirinto, que são obtidos através de um mapeamento prévio, de forma a montar caminhos que levem ao próximo ponto de intersecção, desde que não leve a um caminho sem saída. Portanto, o algoritmo passará pelos pontos de intersecção do labirinto até chegar ao ponto de saída do labirinto.

Ponto de Intersecção:

(1,2)	(2,2)	(3,2)
(1,3)	(2,3)	(3,3)
(1,4)	(2,4)	(3,4)

Código – Recursiva

```
def navigate(  
    m: list[list[int]],  
    root: Point,  
    destiny: Point,  
    cur: Point = None,  
    path: list[Point] = [],  
) -> list[Point] | bool:  
    if cur is None:  
        cur = root  
  
    path.append(cur)  
    h, w = len(m), len(m[0])  
  
    for p in directions(cur):  
        if not p in path and bounded(p, m, h, w):  
            if res := navigate(m, root, destiny, p, path.copy()):  
                return res  
  
            if p == destiny:  
                path.append(p)  
                return path  
  
    else:  
        return False
```

Código - Iterativo

```
1 elif not check:
2     target_index = track.index((previous[-1]))
3     track = track[:target_index+1]
4     interssections.remove(current)
5     previous.pop()
6     current = track[target_index]
7     i = 0
8     continue
9     current = interssections[i]
10    i += 1
```

```
1 def navigate(
2     m: List[List[int]],
3     start: Tuple[int,int],
4     end: Tuple[int,int],
5     interssections: List[Tuple[int,int]]
6 ) -> List[Tuple[int,int]]:
7     current = start
8     track: List[Tuple[int,int]] = []
9     track = []
10    previous: List[Tuple[int,int]] = []
11    previous = []
12    i = 0
13    while i <= len(interssections):
14        track.append(current)
15        check = False
16        if current[0] == end[0] or current[1] == end[1]:
```

```
1     if(interssections[i][0] == current[0] and interssections[i] not in previous):
2
3         check = True
4         previous.append(current)
5         if interssections[i][1] > current[1]:
6             for j in range(interssections[i][1] - current[1]):
7                 if(m[interssections[i][0]][current[1]+(j+1)]):
8                     check = False
9                     track.append((interssections[i][0],current[1]+(j+1)))
10                if current[1] > interssections[i][1]:
11                    for j in range(current[1] - interssections[i][1] - 1):
12                        if(m[interssections[i][0]][interssections[i][1]+(j+1)]):
13                            check = False
14                            track.append((interssections[i][0], interssections[i][1]+(j+1)))
```

Análise Big O



No **algoritmo recursivo**, o termo de maior complexidade é a garantia de que um ponto não é navegado mais de uma vez

```
path = [[0,1], [1,1], [2,1], [2,2], ...]  
       [[0,1], [1,1], [2,1], [2,2]]  
       [[0,1], [1,1], [2,1]]  
       [[0,1], [1,1]]  
       [[0,1]]
```

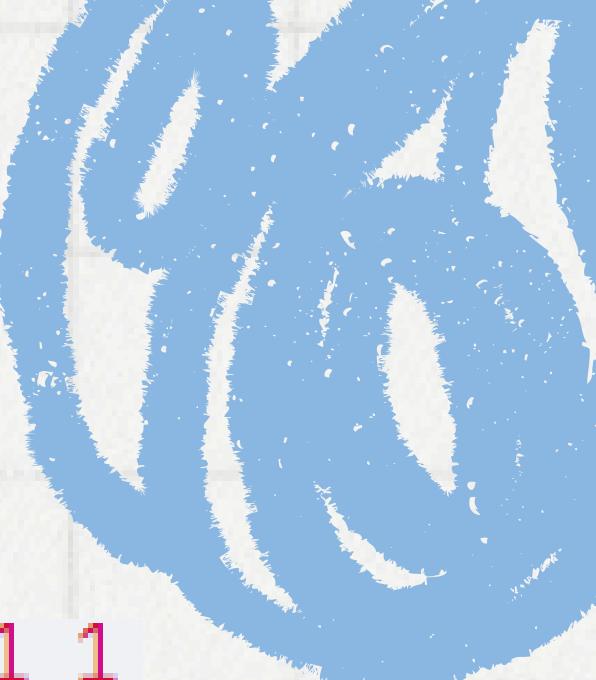
$$\text{len(path)} = Tn = (n^2 + n)/2$$

$$O(n^2)$$

1	1	1	1	1	1	1	1	1	1	1
2	0	0	0	0	0	0	0	0	0	1
1	1	1	0	1	1	1	0	1	1	
1	0	1	1	1	0	0	0	0	1	
1	0	1	0	1	0	1	1	0	1	
1	0	1	0	1	0	1	1	1	0	1
1	0	0	0	0	0	1	1	1	1	1
1	0	1	1	0	1	1	1	0	2	
1	0	0	1	0	0	0	0	0	1	
1	1	1	1	1	1	1	1	1	1	1

No pior dos casos, o algoritmo precisa navegar todas as células possíveis antes de chegar a uma conclusão. Chamamos isso de **volume** do labirinto.

Análise Big O



Já no **algoritmo iterativo**, temos que, o algoritmo depende de duas variáveis, sendo elas:

- n: Se trata do número de pontos de intersecção que existem no labirinto
 - m: Se trata do número de pontos de intersecção que levam à um caminho sem saída

Desta forma, assim que o algoritmo se depara com um caminho sem saída, ele é obrigado a reitarar por todos os pontos de intersecção, menos aquele que levou a um caminho sem saída, portanto temos:

O(n^2)



Conclusão da Análise

Recursiva: é um processo com uso intenso de memória. Dois pontos de melhoria seria evitar a duplicidade de dados de *path* entre chamadas e implementar uma busca mais eficiente para evitar pontos duplicados.

Iterativa: Também se trata de um algoritmo custoso, devido à sua necessidade de reiterar por diversos pontos. Existem diversas coisas a serem melhoradas no código, até mesmo erros que podem levar a falhas de execução.

Ambos os algoritmos não buscam necessariamente pelo melhor caminho possível, apenas o primeiro que consigam encontrar. Garantir o melhor caminho envolveria explorar todos os caminhos possíveis.

Obrigado pela atenção!

[www.github.com - iterativo](https://www.github.com/iterativo)

[www.github.com - recursivo](https://www.github.com/recursivo)