



# Algoritmos de Busca em Grafos

Prof. Fernando Sambinelli

# Percorrer um Grafo

- Percorrer (ou caminhar em) um grafo é um problema fundamental
- Pode-se buscar um determinado vértice ou percorrer todos os vértices
- Deve-se ter uma forma sistemática para visitar as arestas e os vértices
- Há diferentes maneiras de percorrer um grafo:
  - Busca genérica
  - Busca em profundidade (*depth first search*)
  - Busca em largura (*breadth first search*)



# Percorrer um Grafo

## Definições Importantes

- INPUT:
  - Grafo  $G = (V, A)$
  - Vértice inicial  $s$
- OUTPUT: conjunto de vértices alcançáveis a partir de  $s$
- Eficiência: Não deve haver repetições de visitas desnecessárias a um vértice ou aresta
- Solução: marcar os vértices já visitados



## **BUSCA EM PROFUNDIDADE**

---

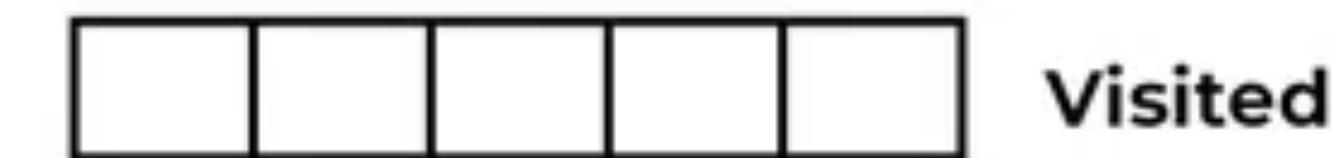
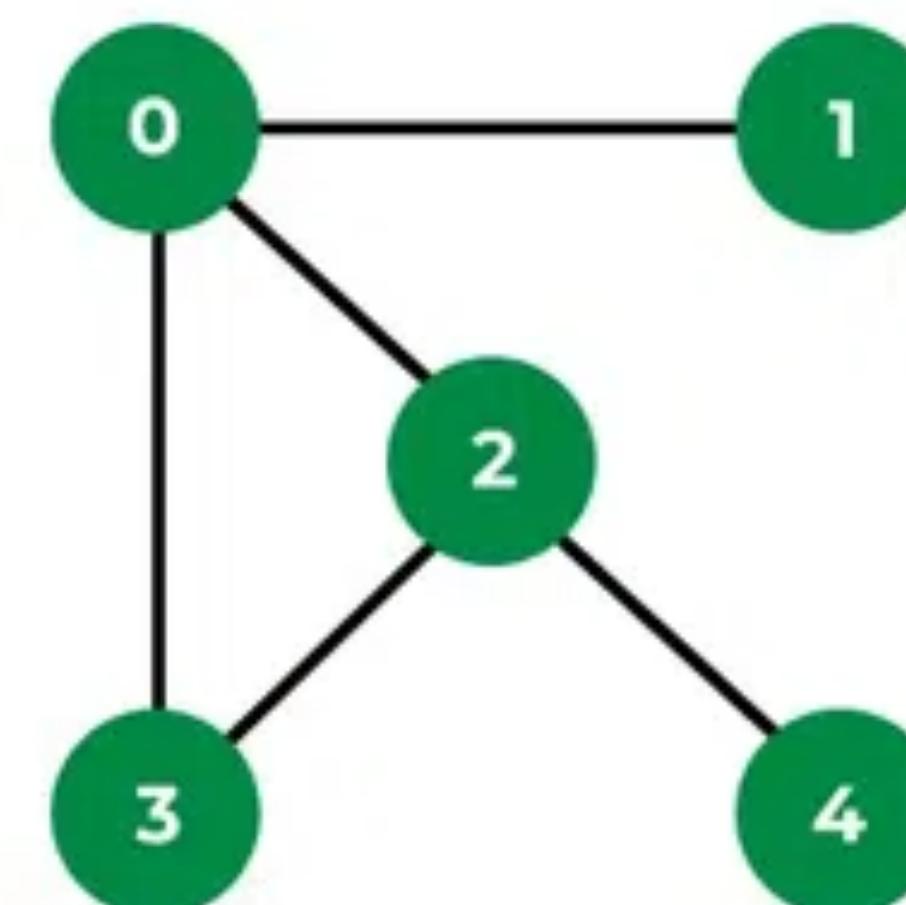
- Definição
- Características
- Exemplos

# Busca em Profundidade

- A busca em profundidade (ou DFS, do inglês *depth-first search*) é um algoritmo para percorrer ou buscar em uma estrutura de dados do tipo grafo.
- A ideia básica do algoritmo é começar a busca por um nó inicial e **explorar o máximo possível ao longo de cada ramificação antes de retroceder** e explorar outros ramos.
- Ele utiliza **uma pilha** para armazenar os nós a serem **visitados** e uma lista para manter o controle dos nós já **visitados**, a fim de evitar ciclos infinitos

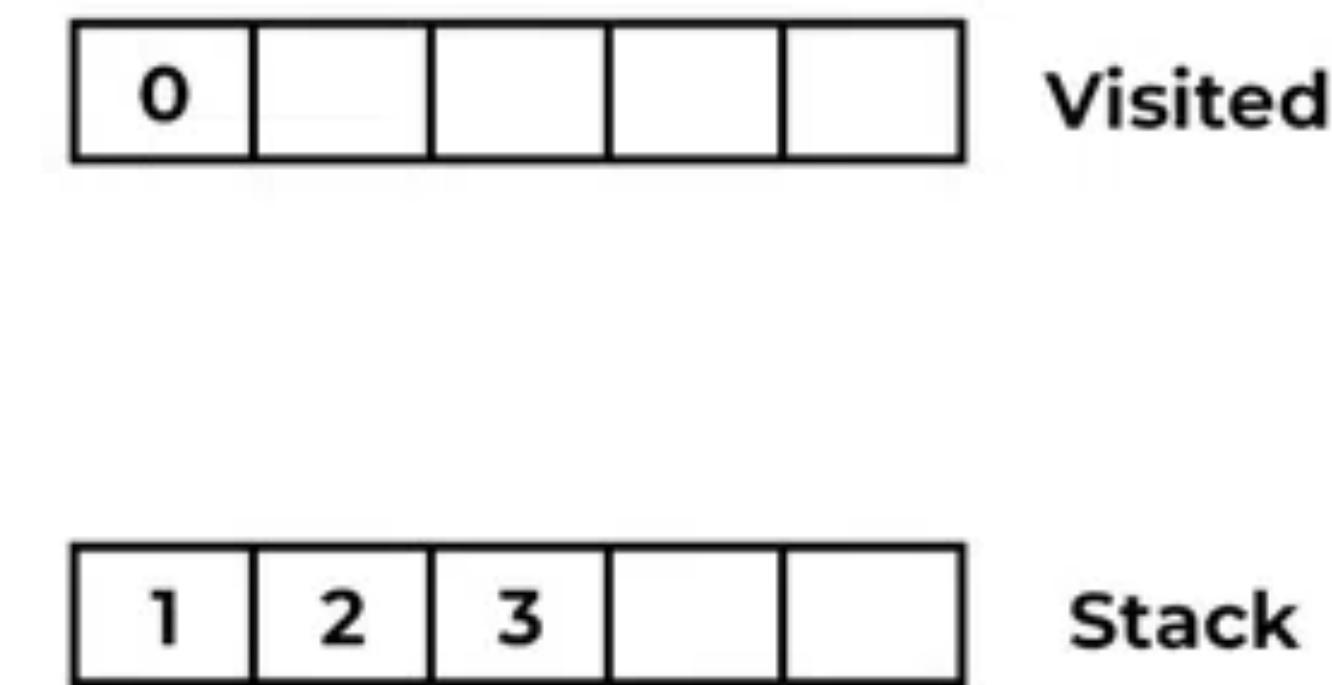
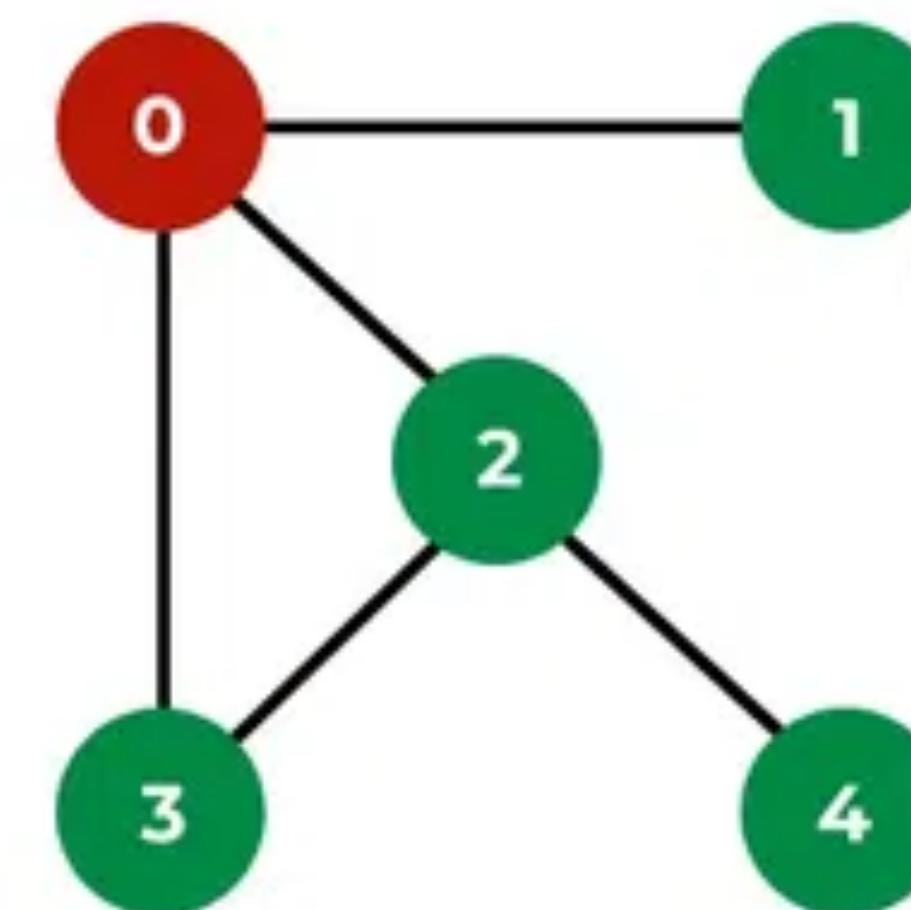
# Busca em Profundidade

PASSO 1: Inicialmente a pilha e os arrays visitados estão vazios.



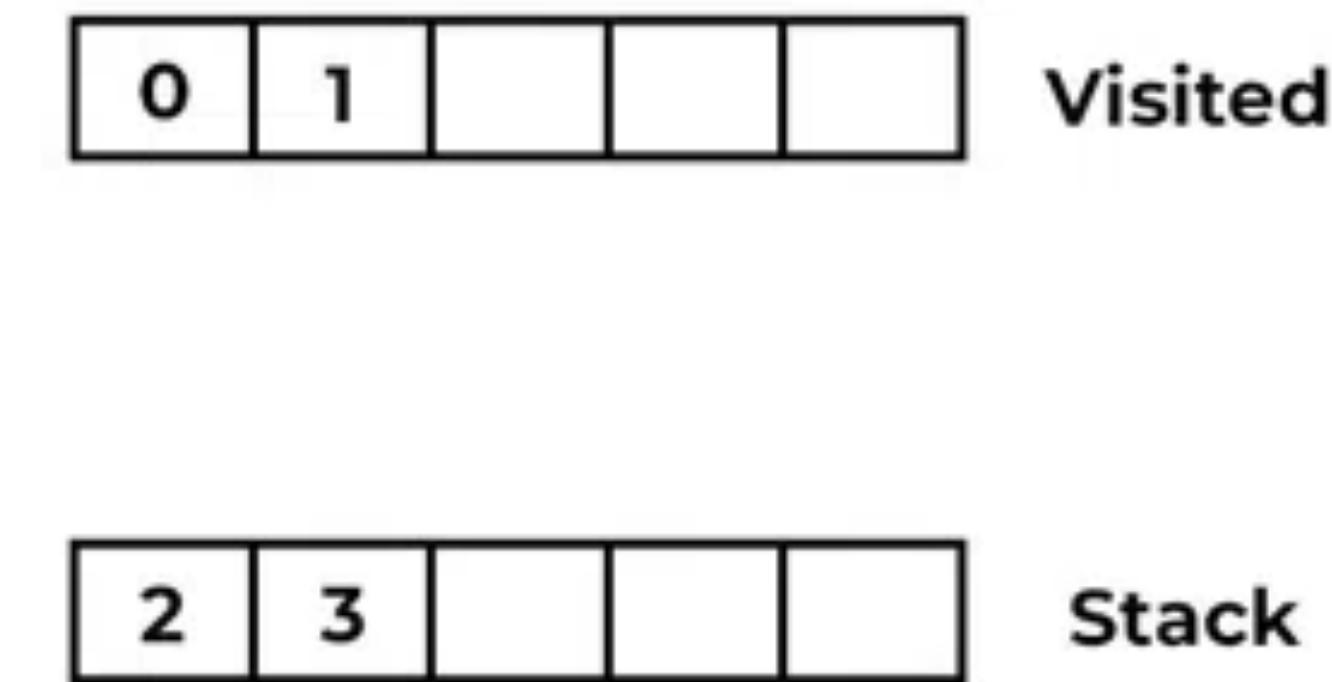
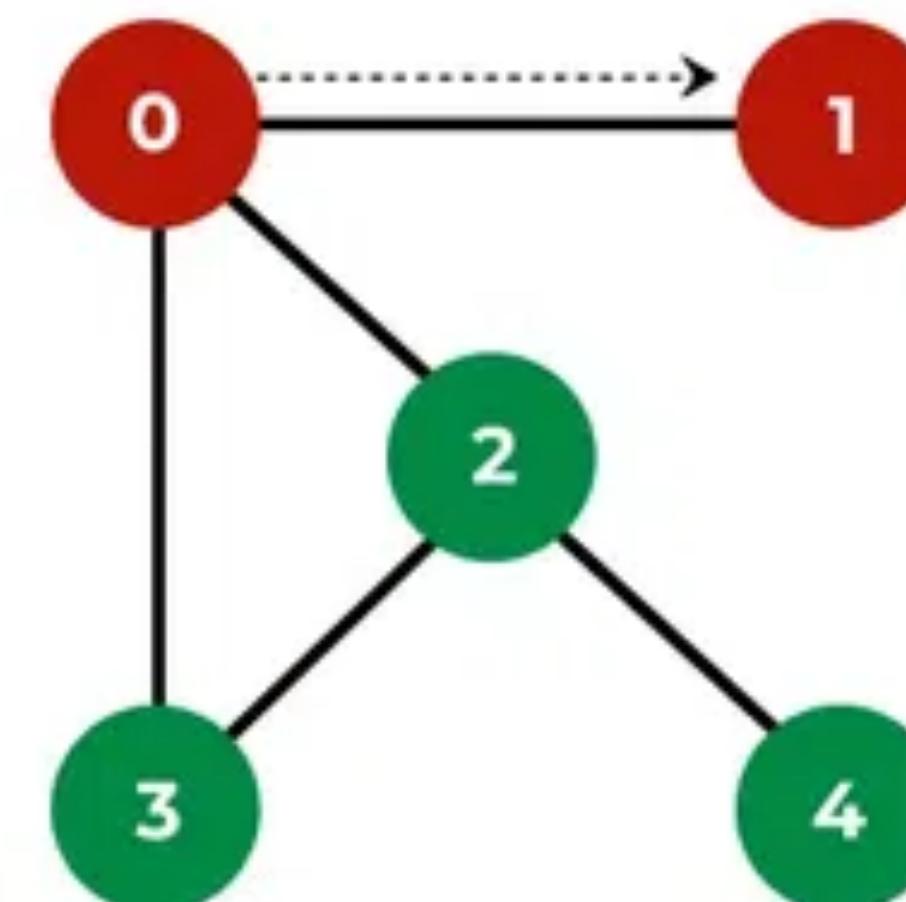
# Busca em Profundidade

**PASSO 2:** Visite 0 e coloque na pilha seus nós adjacentes que ainda não foram visitados.



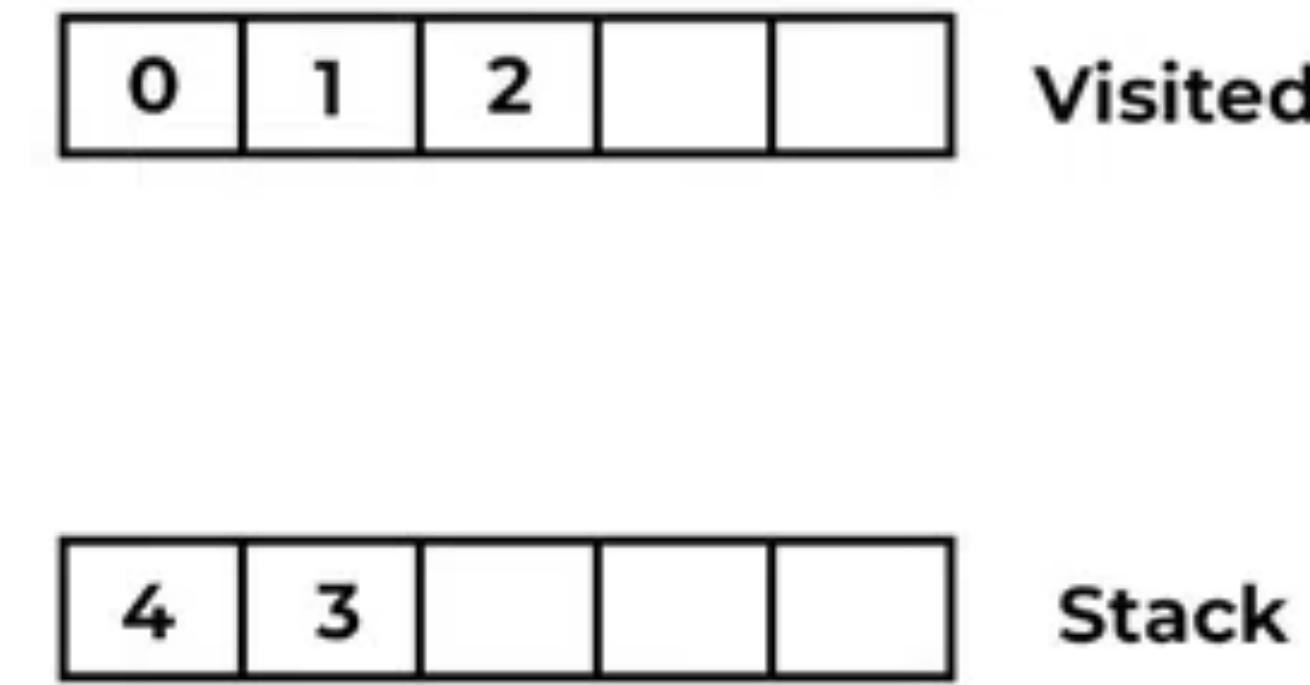
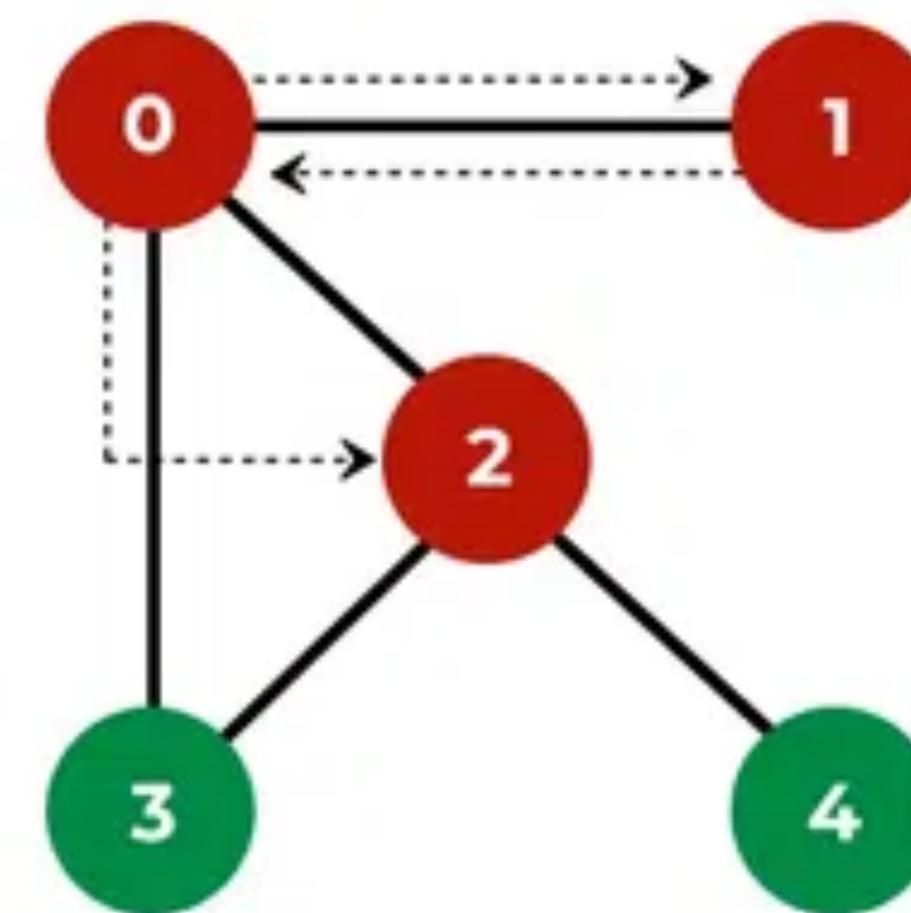
# Busca em Profundidade

PASSO 3: Agora, o nó 1 está no topo da pilha, então visite o nó 1, retire-o da pilha e coloque todos os seus nós adjacentes que não são visitados na pilha.



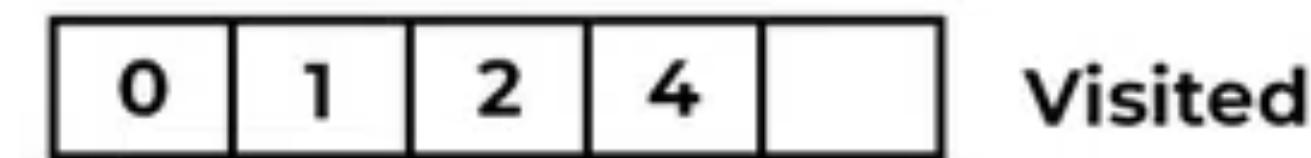
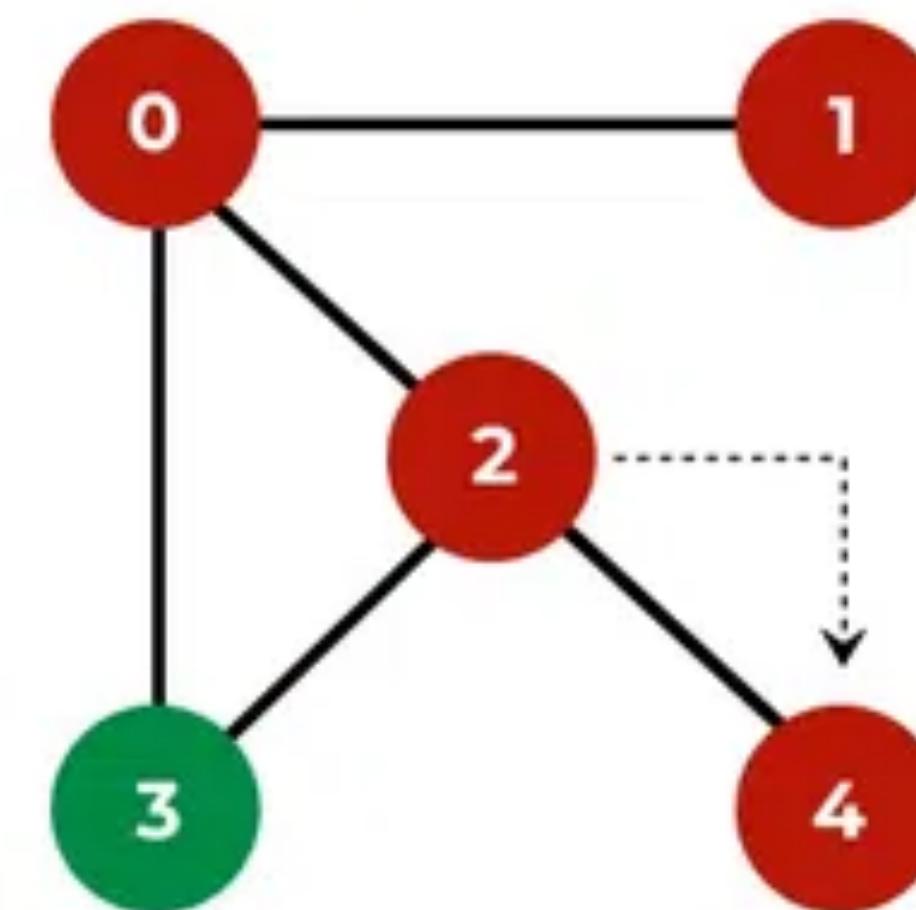
# Busca em Profundidade

PASSO 4: Agora, o nó 2 está no topo da pilha, então visite o nó 2, retire-o da pilha e coloque todos os seus nós adjacentes que não são visitados (ou seja, 3, 4) na pilha.



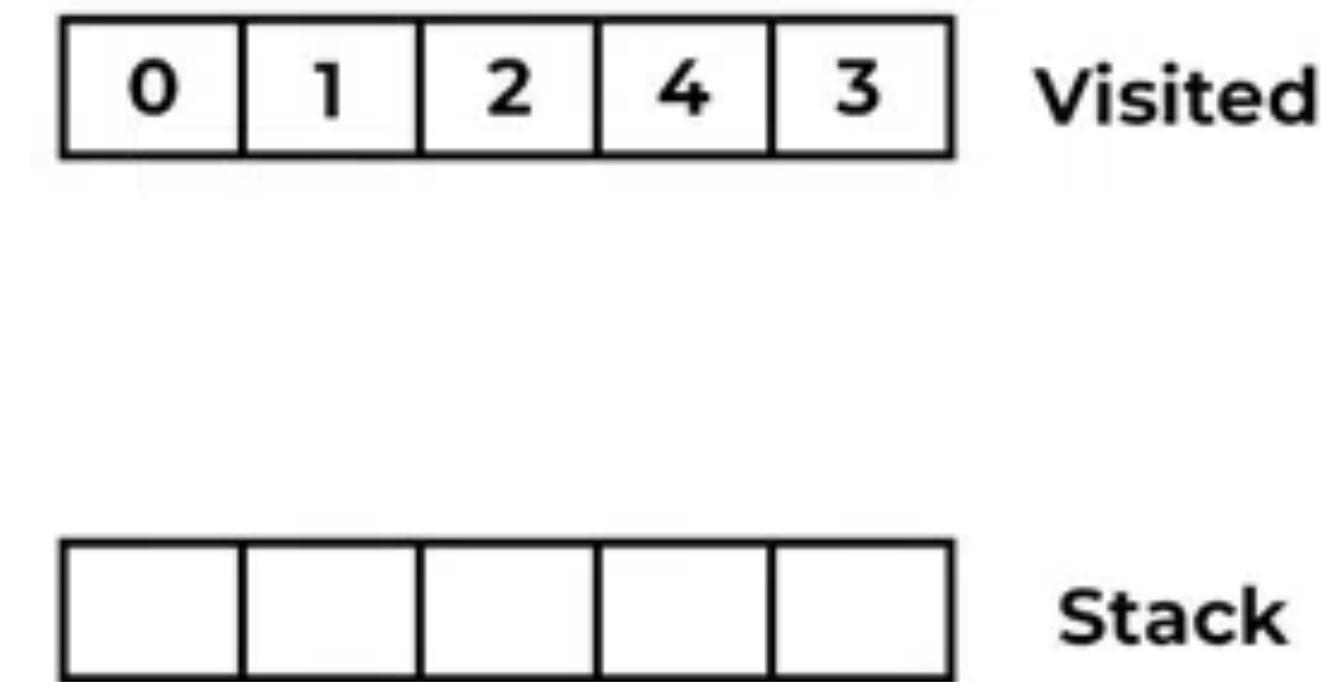
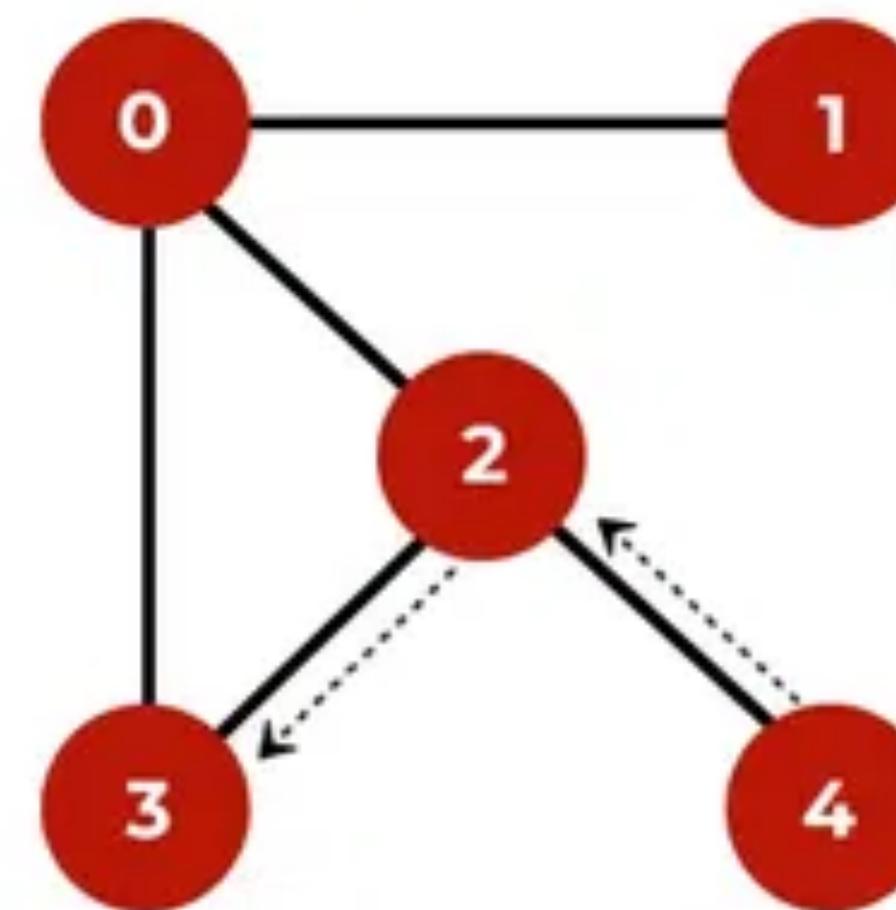
# Busca em Profundidade

PASSO 5: agora, o nó 4 está no topo da pilha, então visite o nó 4, retire-o da pilha e coloque todos os seus nós adjacentes que não são visitados na pilha.



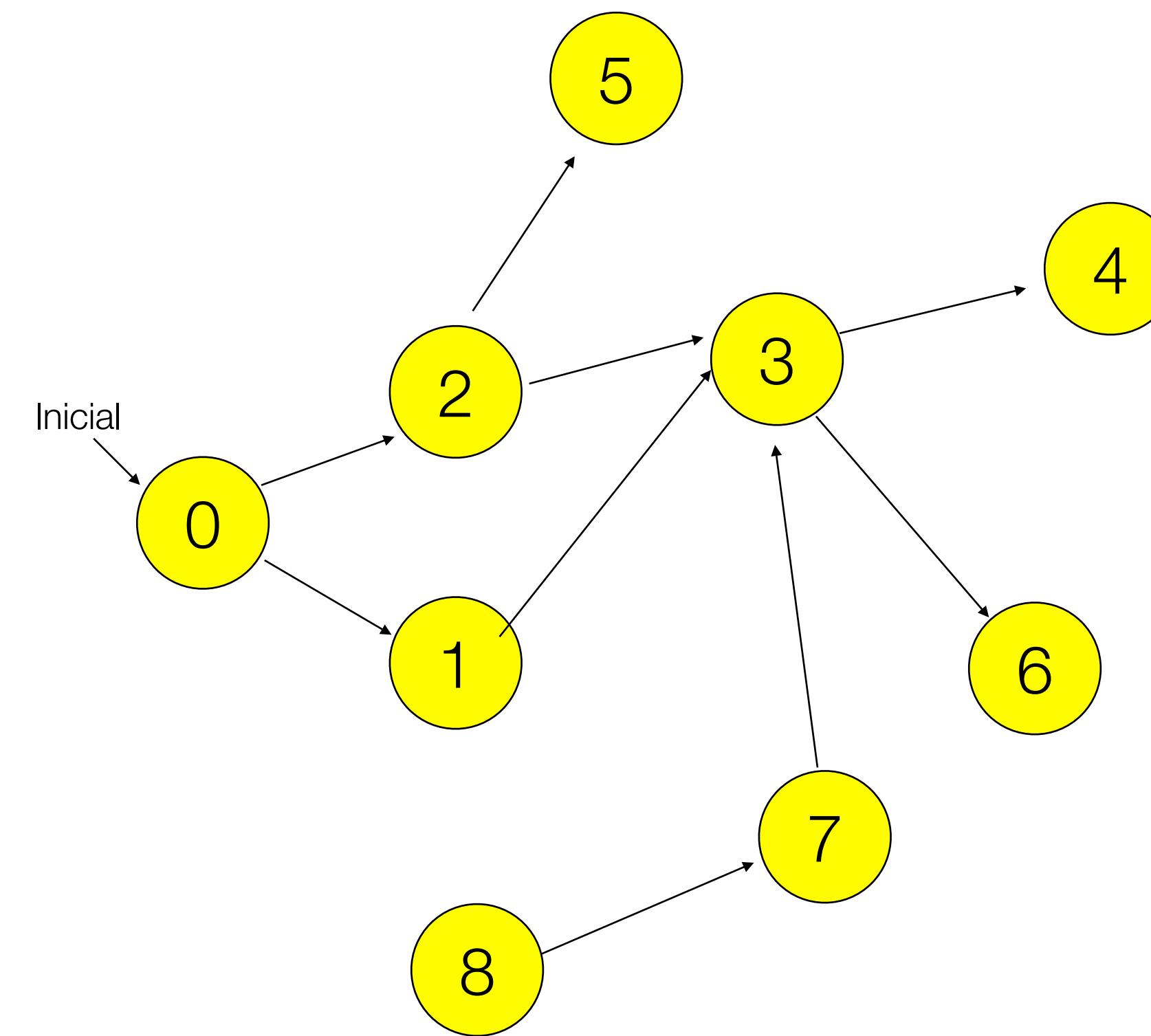
# Busca em Profundidade

PASSO 6: Agora, o nó 3 está no topo da pilha, então visite o nó 3, retire-o da pilha e coloque todos os seus nós adjacentes que não são visitados na pilha.



# Exercício

- Usando uma busca em profundidade, identifique neste grafo abaixo a sequência de visitação dos vértices



# Código em Java

```
class BuscaProfundidade {
    private int numVertices;
    private LinkedList<Integer>[] listaAdjacencia;

    public BuscaProfundidade(int numVertices) {
        this.numVertices = numVertices;
        listaAdjacencia = new LinkedList[numVertices];
        for (int i = 0; i < numVertices; i++) {
            listaAdjacencia[i] = new LinkedList<>();
        }
    }

    public void adicionarArestas(int verticeOrigem, int verticeDestino) {
        listaAdjacencia[verticeOrigem].add(verticeDestino);
    }
}
```

```
public LinkedHashSet<Integer> buscaProfundidadeGrafo(int verticeInicial) {
    LinkedHashSet<Integer> verticesVisitado = new LinkedHashSet<>();
    Stack<Integer> pilhaPercorso = new Stack<>();

    pilhaPercorso.push(verticeInicial);
    verticesVisitado.add(verticeInicial);

    while (!pilhaPercorso.isEmpty()) {
        int verticeAtual = pilhaPercorso.pop();
        verticesVisitado.add(verticeAtual);

        for (int verticeAdjacente : listaAdjacencia[verticeAtual]) {
            if (!verticesVisitado.contains(verticeAdjacente)) {
                pilhaPercorso.push(verticeAdjacente);
                verticesVisitado.add(verticeAdjacente);
            }
        }
    }
    return verticesVisitado;
}
```

## **BUSCA EM LARGURA**

---

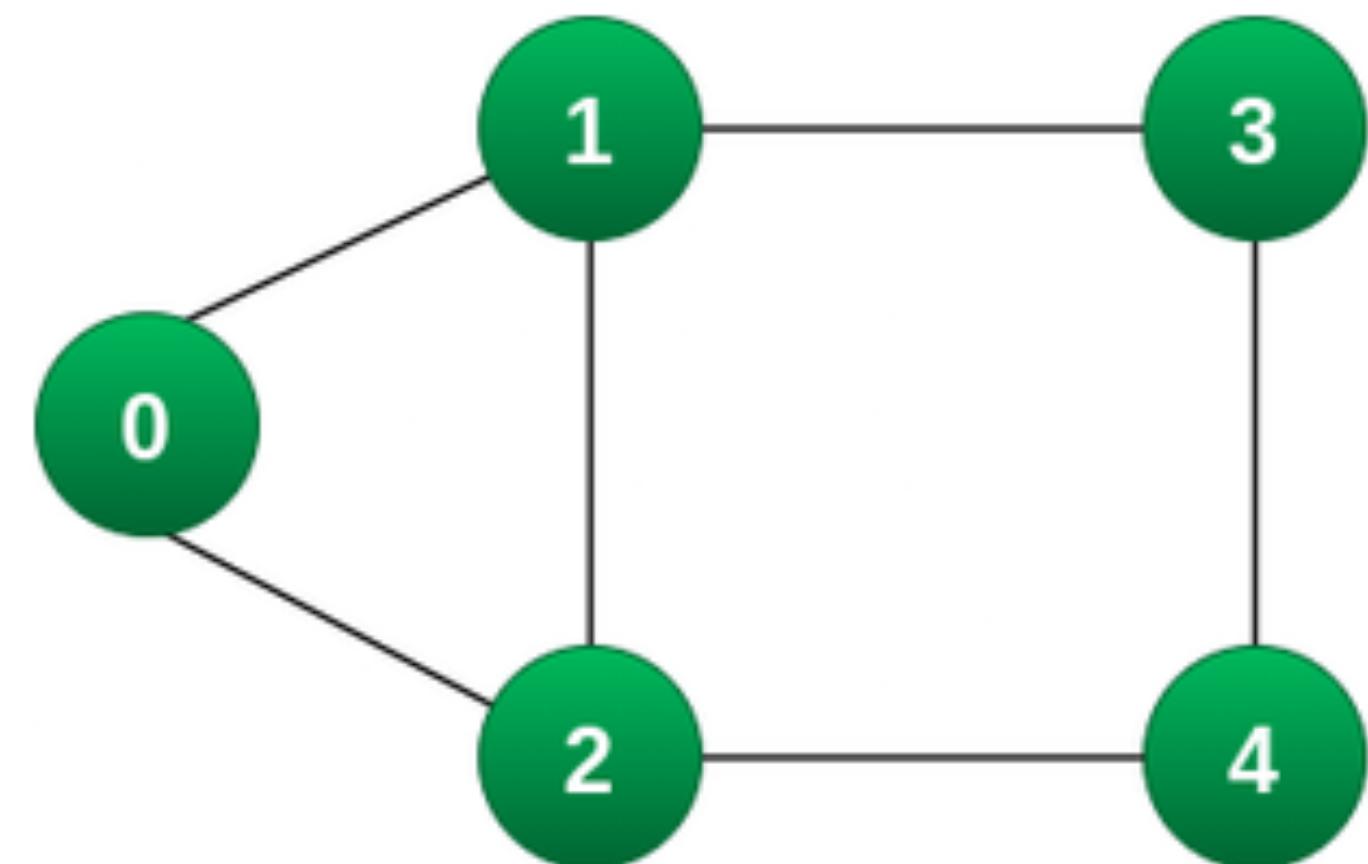
- Definição
- Características
- Exemplos

# Busca em Largura

- A busca em largura (ou BFS, do inglês *breadth-first search*) é um algoritmo para percorrer ou buscar em uma estrutura de dados do tipo grafo.
- A ideia fundamental da busca em largura é **explorar todos os vizinhos de um nó antes de avançar** para os vizinhos dos vizinhos. A busca se dá como em camadas (k)
- Ele utiliza uma **fila** para manter os nós a **serem visitados** e uma **lista** para manter o controle dos nós já **visitados**, a fim de evitar ciclos infinitos.

# Busca em Largura

PASSO 1: Inicialmente a fila e os arrays visitados estão vazios.



Visited



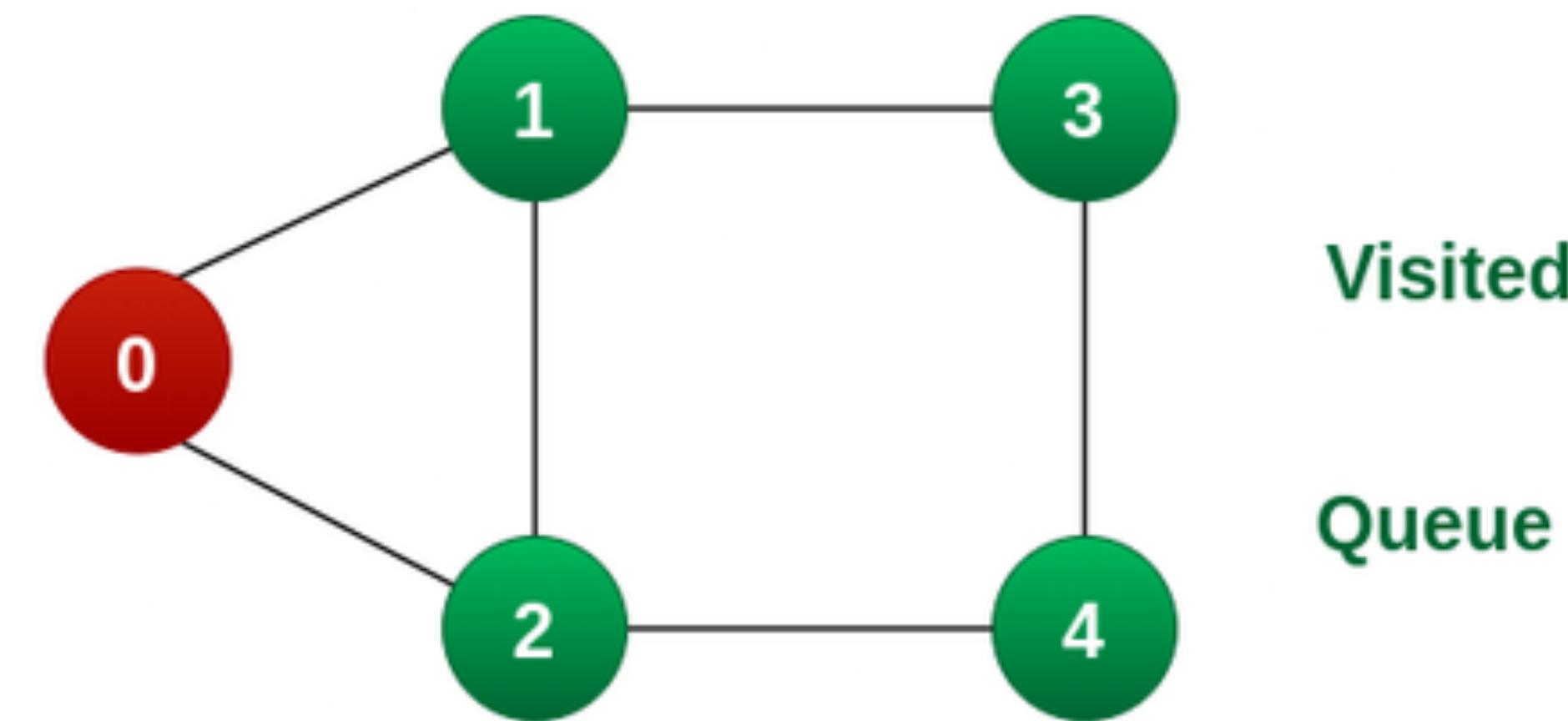
Queue



FRONT

# Busca em Largura

PASSO 2: Coloque o nó 0 na fila e marque-o como visitado.



Visited

0				
---	--	--	--	--

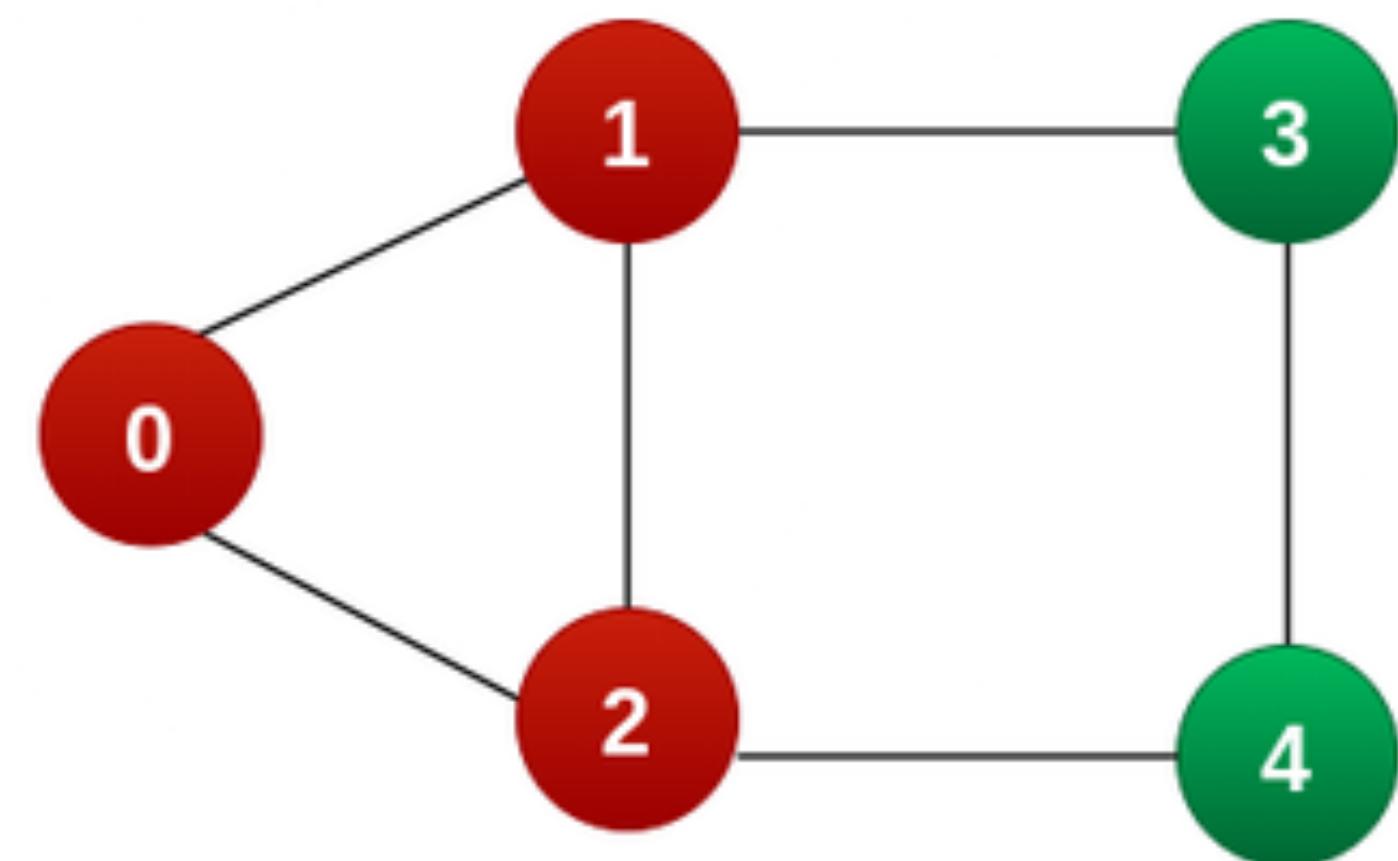
Queue

0				
---	--	--	--	--

FRONT

# Busca em Largura

PASSO 3: Remova o nó 0 do início da fila, visite os vizinhos não visitados e coloque-os na fila.



Visited

0	1	2		
---	---	---	--	--

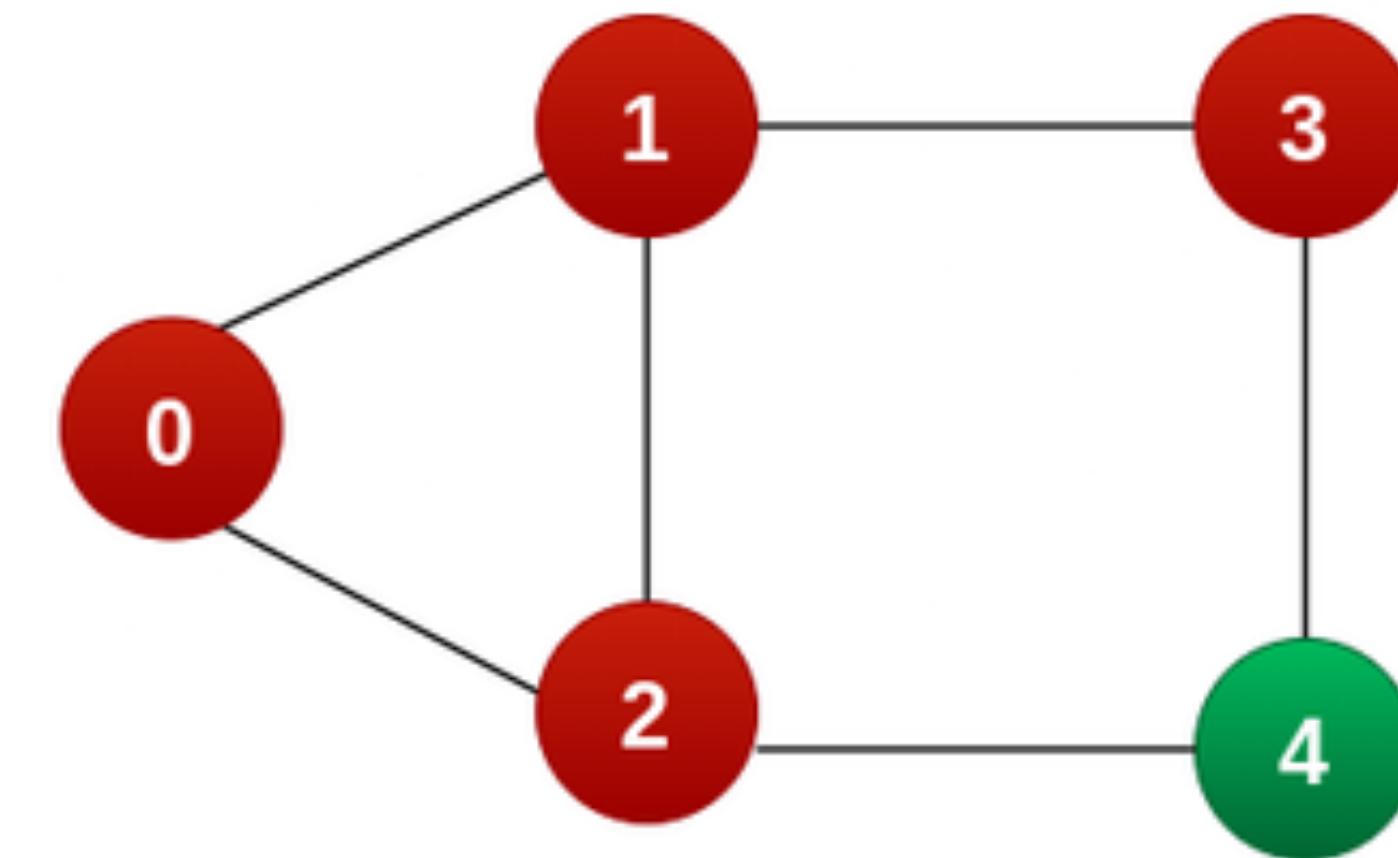
Queue

1	2			
---	---	--	--	--

FRONT

# Busca em Largura

PASSO 4: Remova o nó 1 do início da fila, visite os vizinhos não visitados e coloque-os na fila.



Visited

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

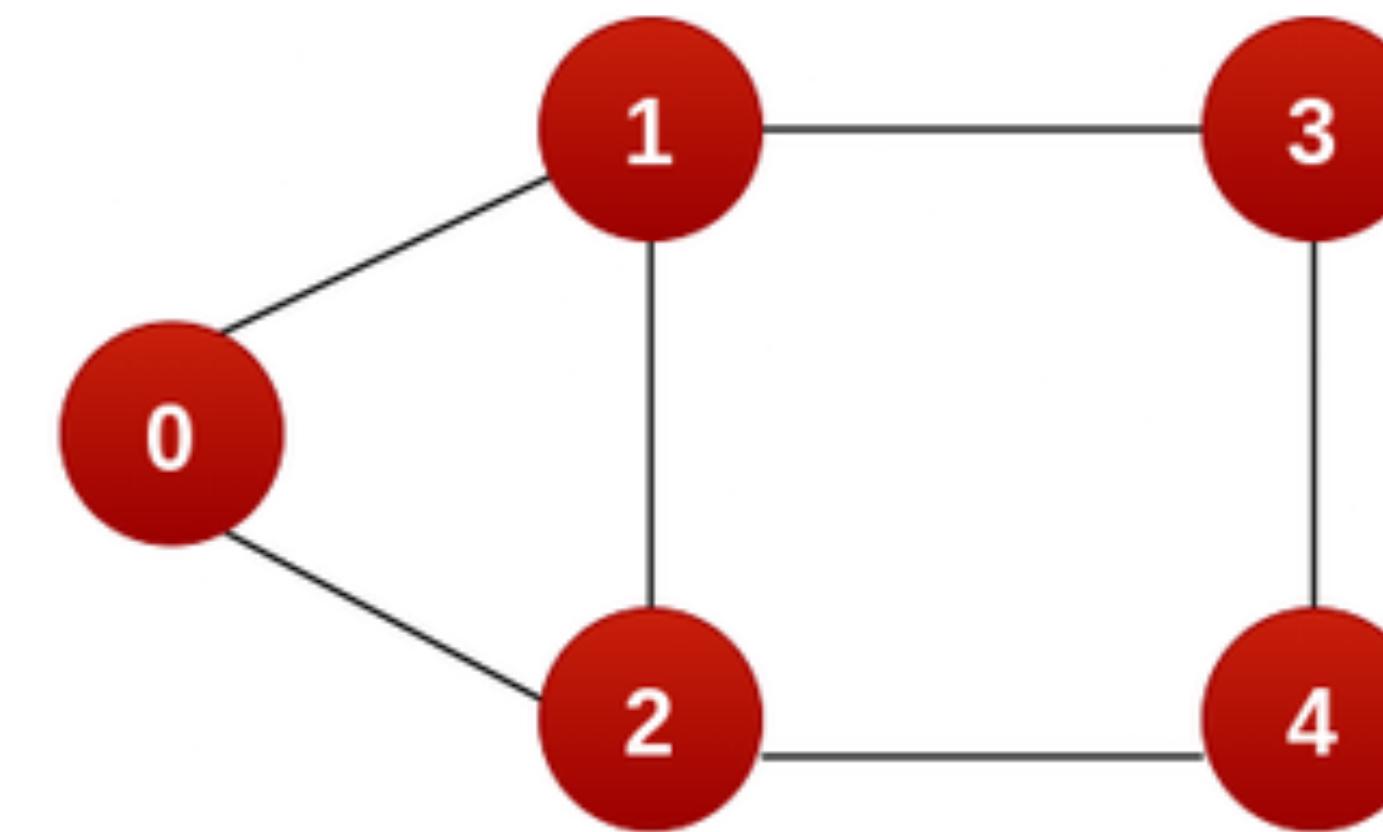
Queue

2	3			
---	---	--	--	--

FRONT

# Busca em Largura

PASSO 5: Remova o nó 2 do início da fila, visite os vizinhos não visitados e coloque-os na fila.



Visited

0	1	2	3	4
---	---	---	---	---

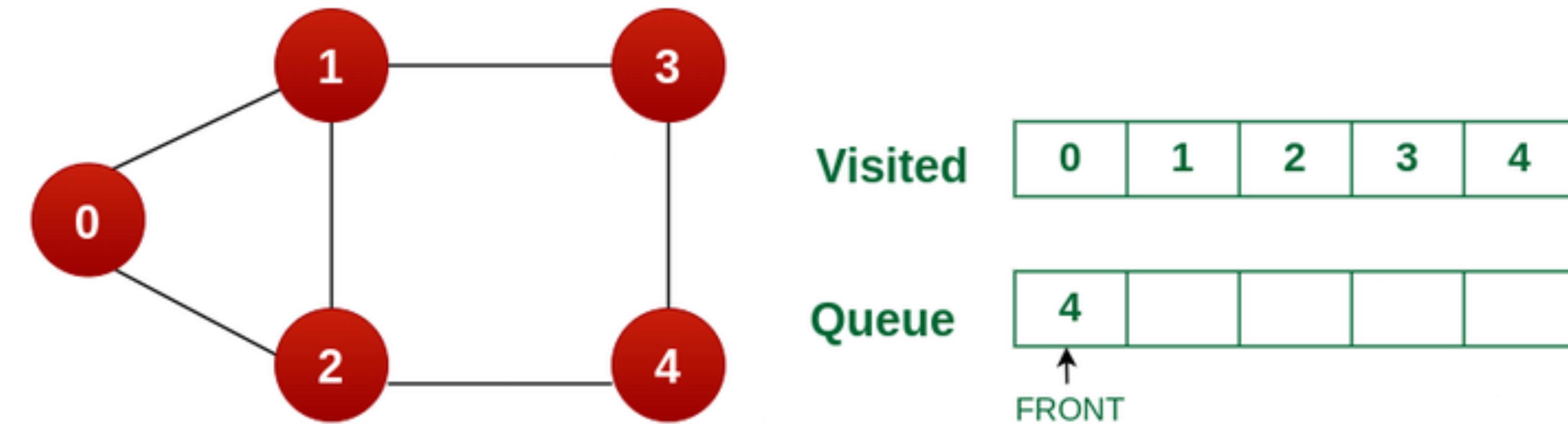
Queue

3	4			
---	---	--	--	--

FRONT

# Busca em Largura

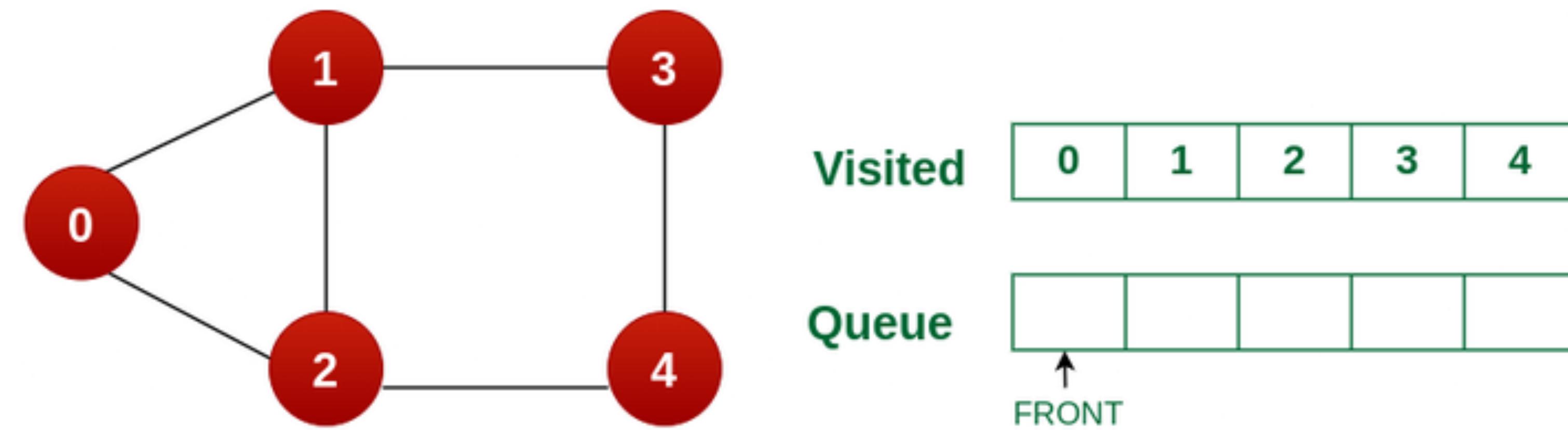
PASSO 6: Remova o nó 3 do início da fila, visite os vizinhos não visitados e coloque-os na fila.



Como podemos ver que todos os vizinhos do nó 3 são visitados, passe para o próximo nó que está na frente da fila.

# Busca em Largura

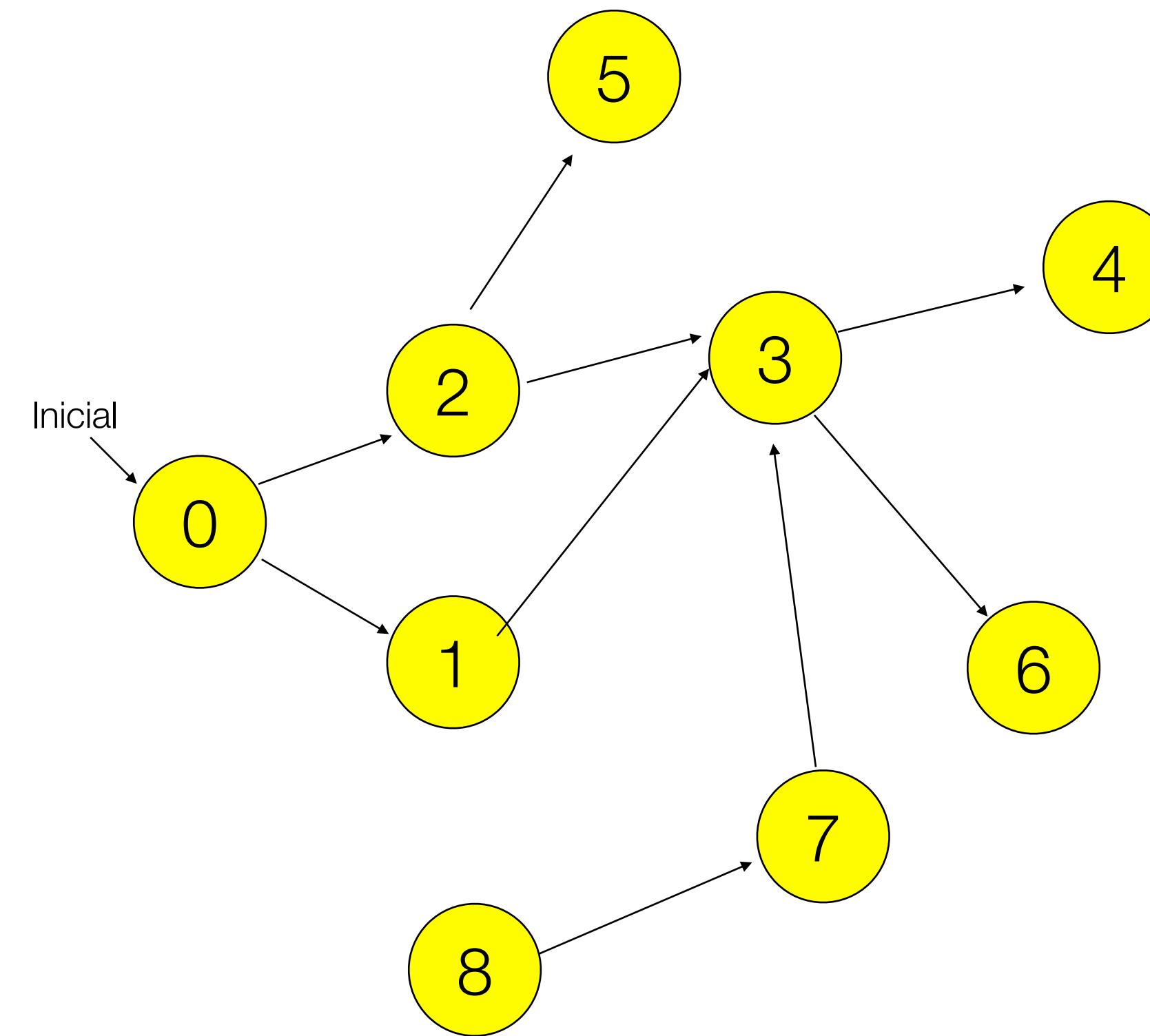
PASSO 7: Remova o nó 4 do início da fila, visite os vizinhos não visitados e coloque-os na fila.



Como podemos ver que todos os vizinhos do nó 4 são visitados, passe para o próximo nó que está na frente da fila.  
Agora, a fila fica vazia, então encerre esse processo de iteração.

# Exercício

- Usando uma busca em largura, identifique neste grafo abaixo a sequência de visitação dos vértices



# Código em Java

```
public class BuscaLargura {
    private int numVertices;
    private LinkedList<Integer>[] listaAdjacencia;

    public BuscaLargura(int numVertices) {
        this.numVertices = numVertices;
        listaAdjacencia = new LinkedList[numVertices];
        for (int i = 0; i < numVertices; i++) {
            listaAdjacencia[i] = new LinkedList<>();
        }
    }

    public void adicionarArestas(int verticeOrigem, int verticeDestino) {
        listaAdjacencia[verticeOrigem].add(verticeDestino);
    }
}
```

```
public LinkedHashSet<Integer> buscaLarguraGrafo(int verticeInicial) {
    LinkedHashSet<Integer> verticesVisitado = new LinkedHashSet<>();
    Queue<Integer> filaPercorso = new LinkedList<>();

    verticesVisitado.add(verticeInicial);
    filaPercorso.add(verticeInicial);

    while (!filaPercorso.isEmpty()) {
        int verticeAtual = filaPercorso.poll();
        verticesVisitado.add(verticeInicial);

        for (int verticeAdjacente : listaAdjacencia[verticeAtual]) {
            if (!verticesVisitado.contains(verticeAdjacente)) {
                verticesVisitado.add(verticeAdjacente);
                filaPercorso.add(verticeAdjacente);
            }
        }
    }
    return verticesVisitado;
}
```

## **APLICAÇÕES DE BUSCA EM GRAFO**

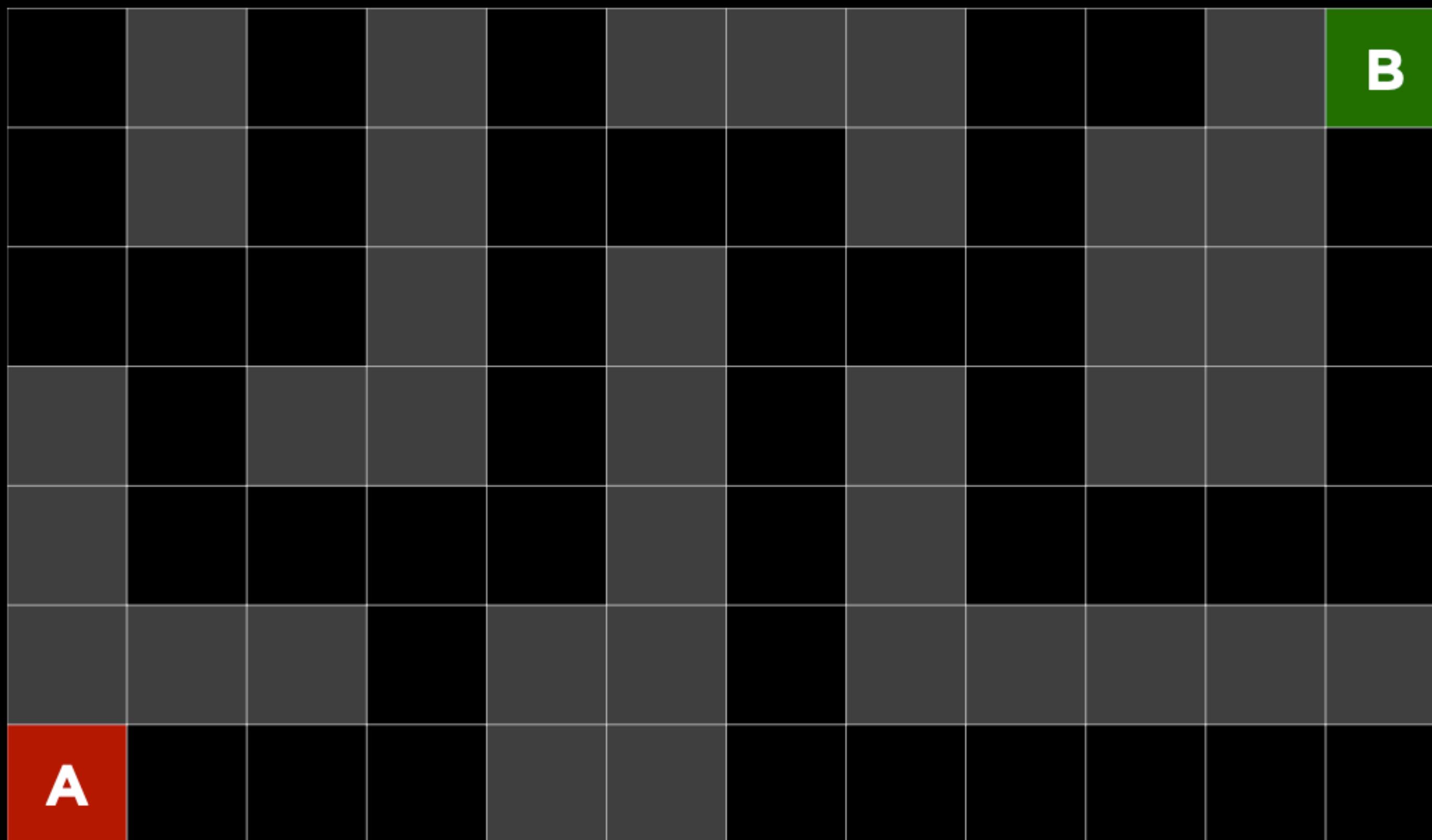
---

- Solução de Labirinto
- Busca do Menor Caminho

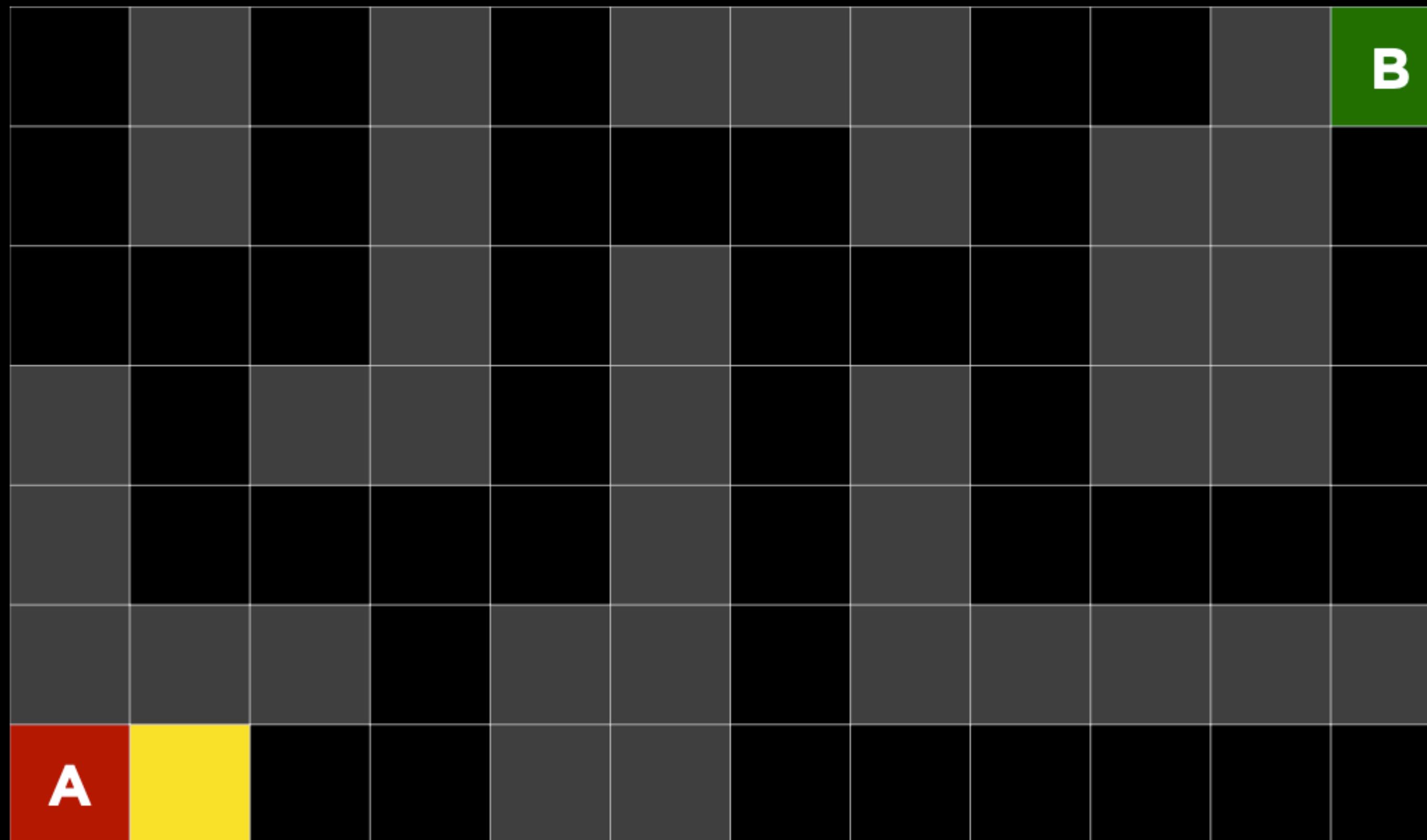
# Solução de Labirinto

# Busca em Profundidade

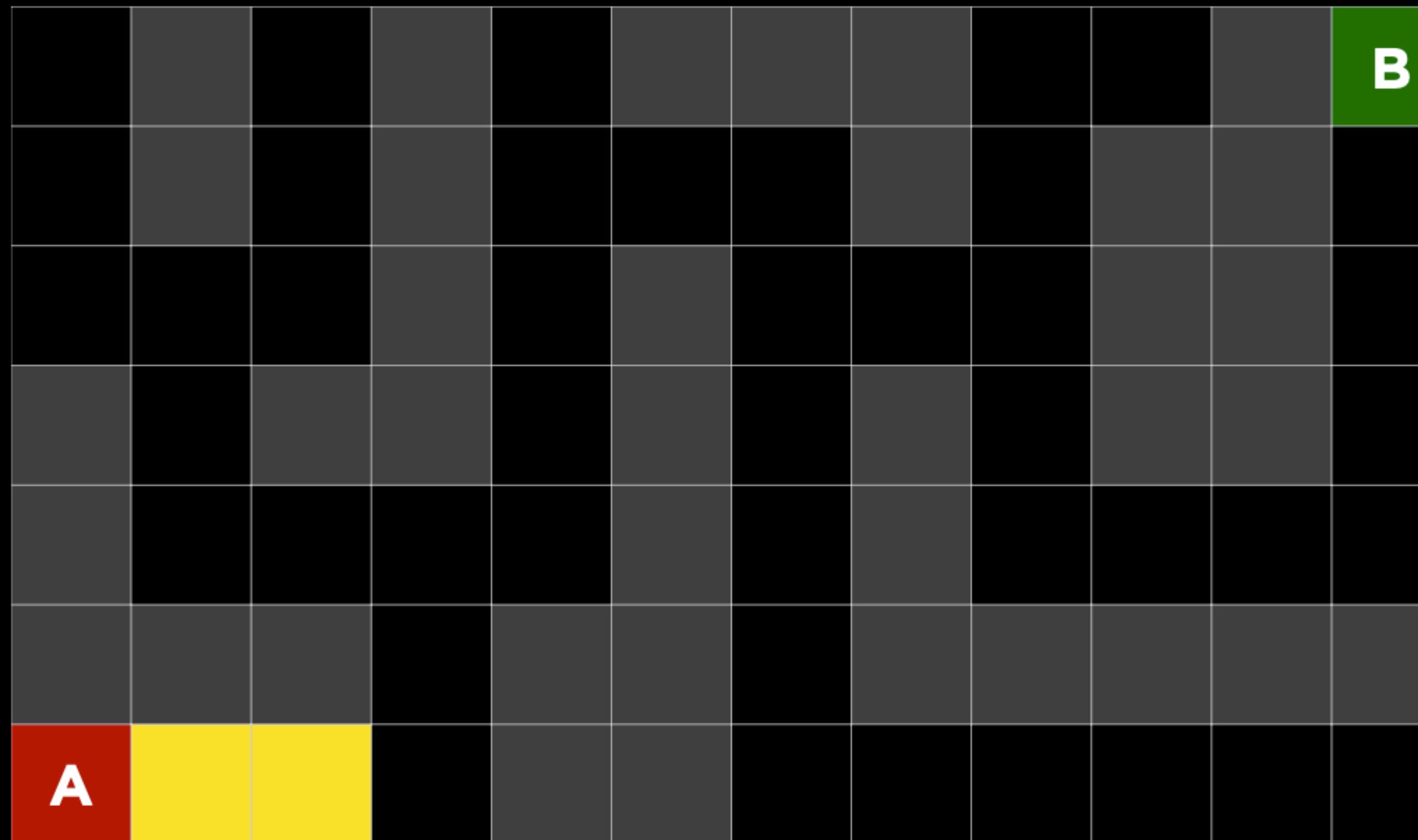
Encontre um caminho entre A e B



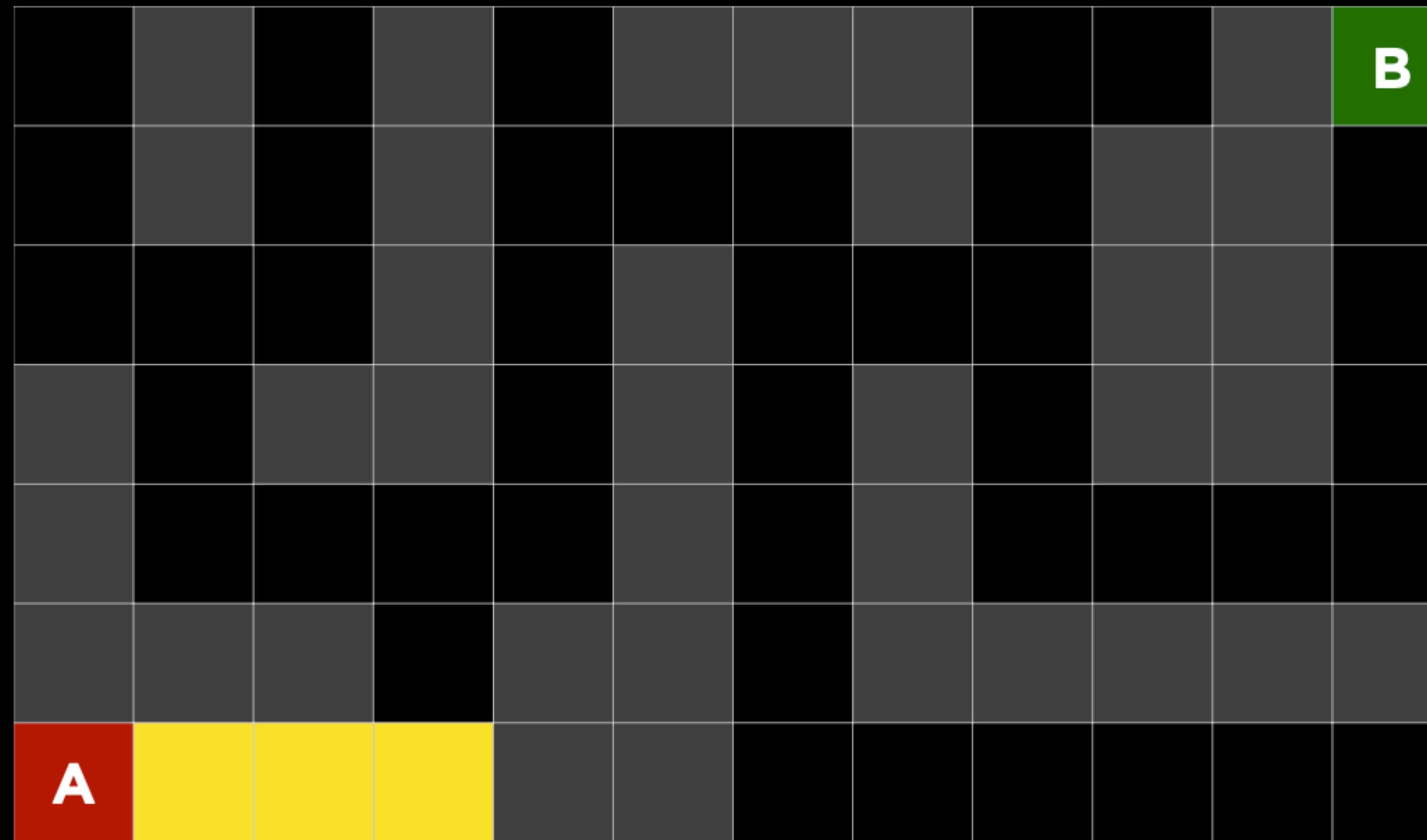
# Busca em Profundidade



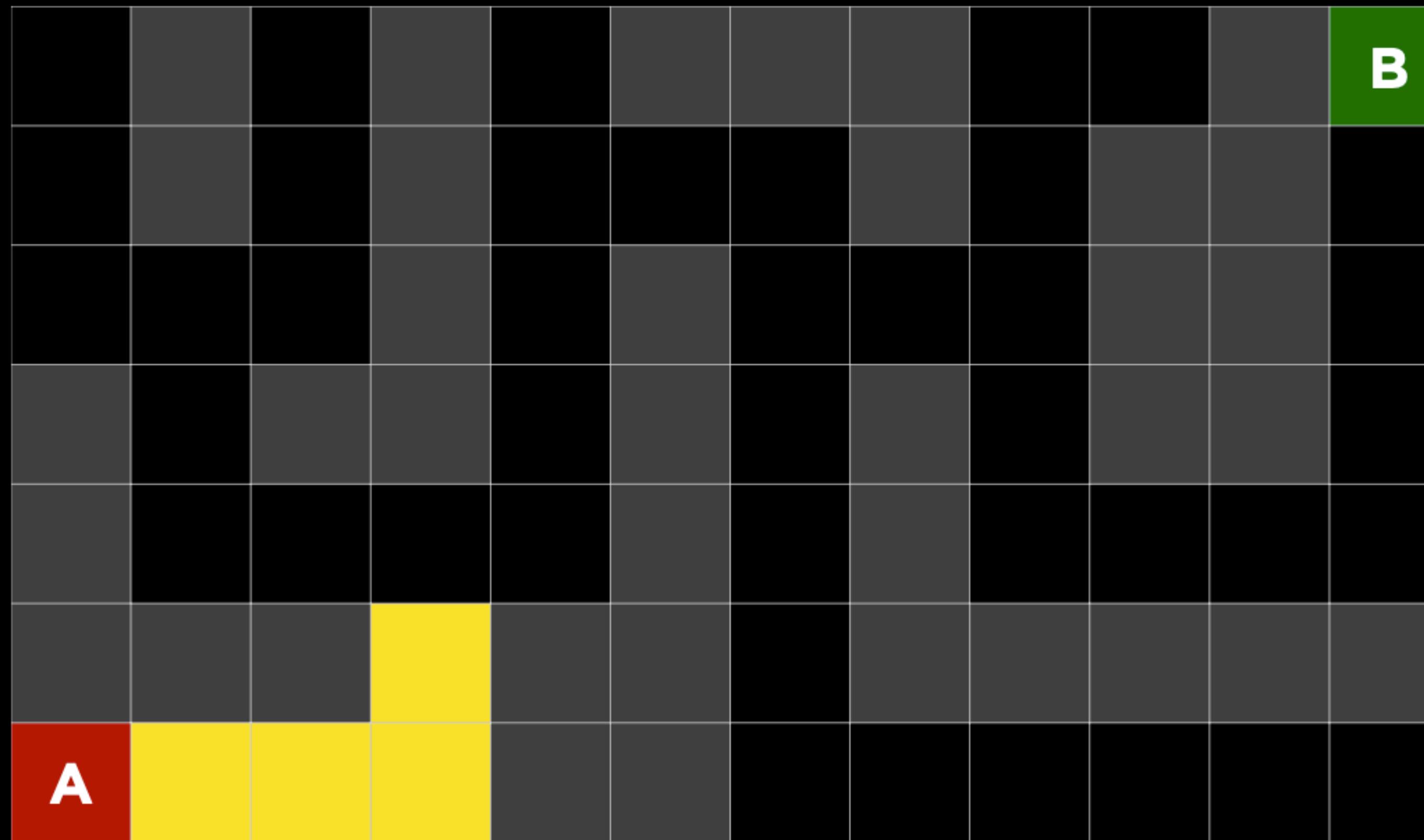
# Busca em Profundidade



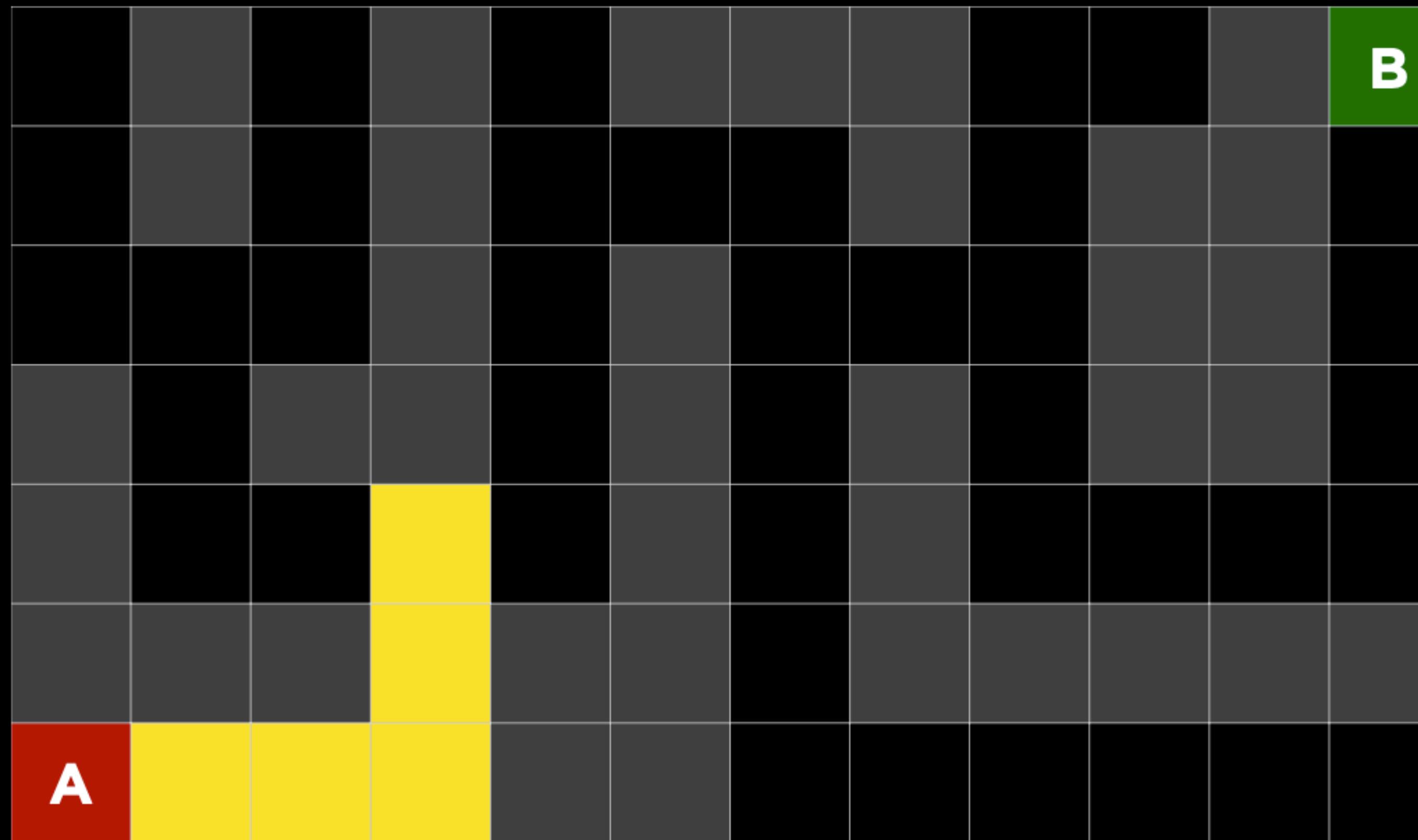
# Busca em Profundidade



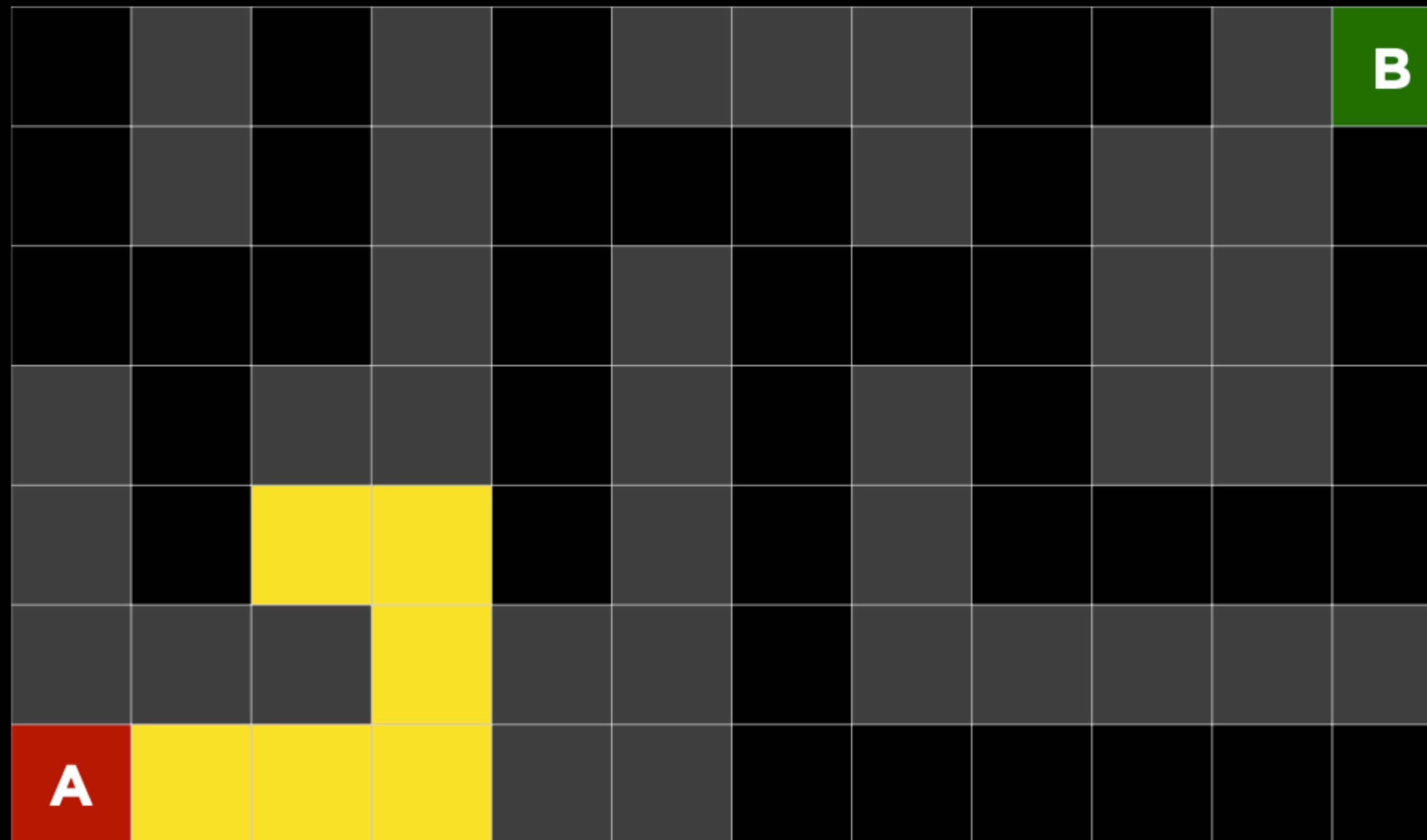
# Busca em Profundidade



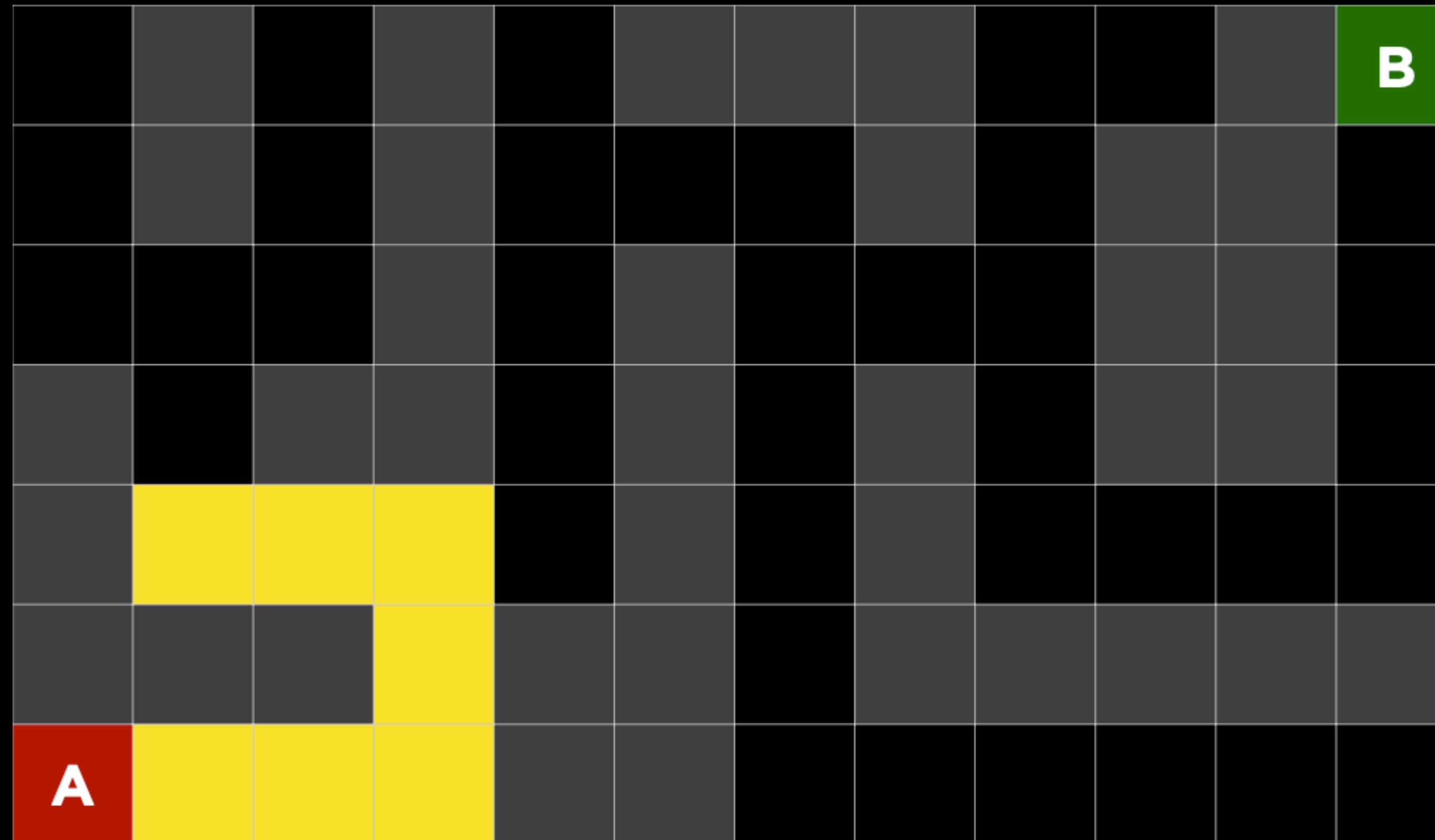
# Busca em Profundidade



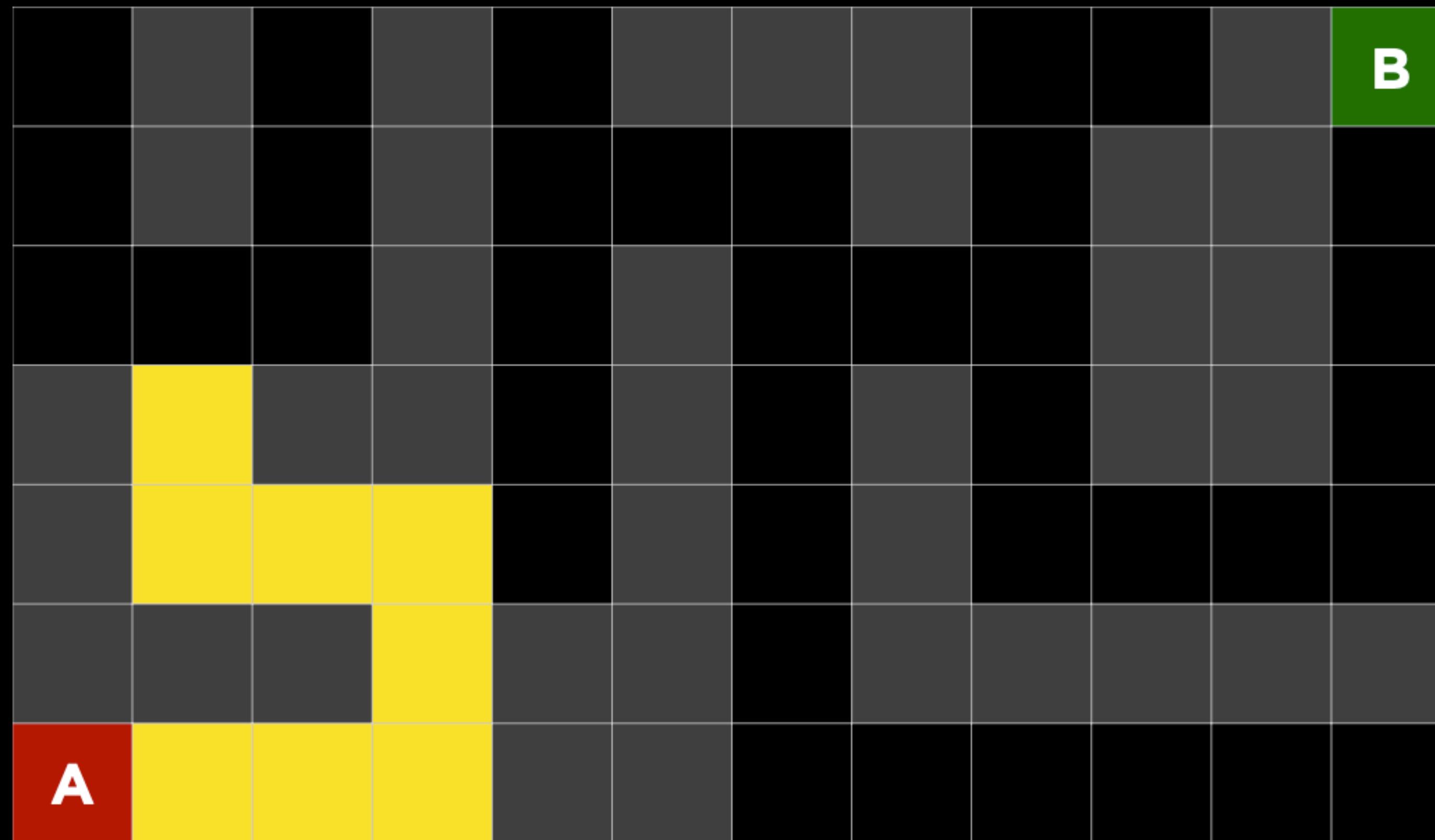
# Busca em Profundidade



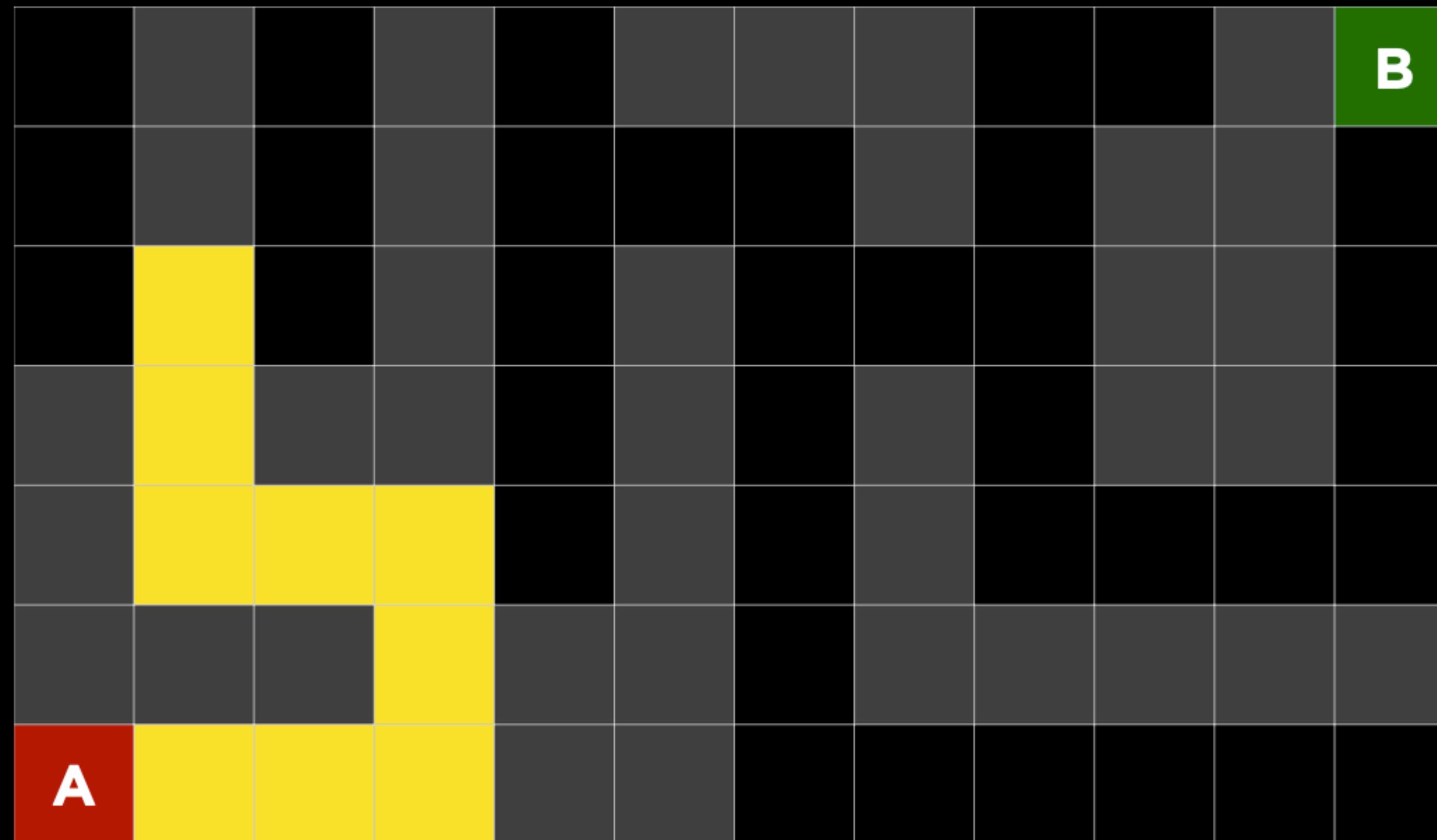
# Busca em Profundidade



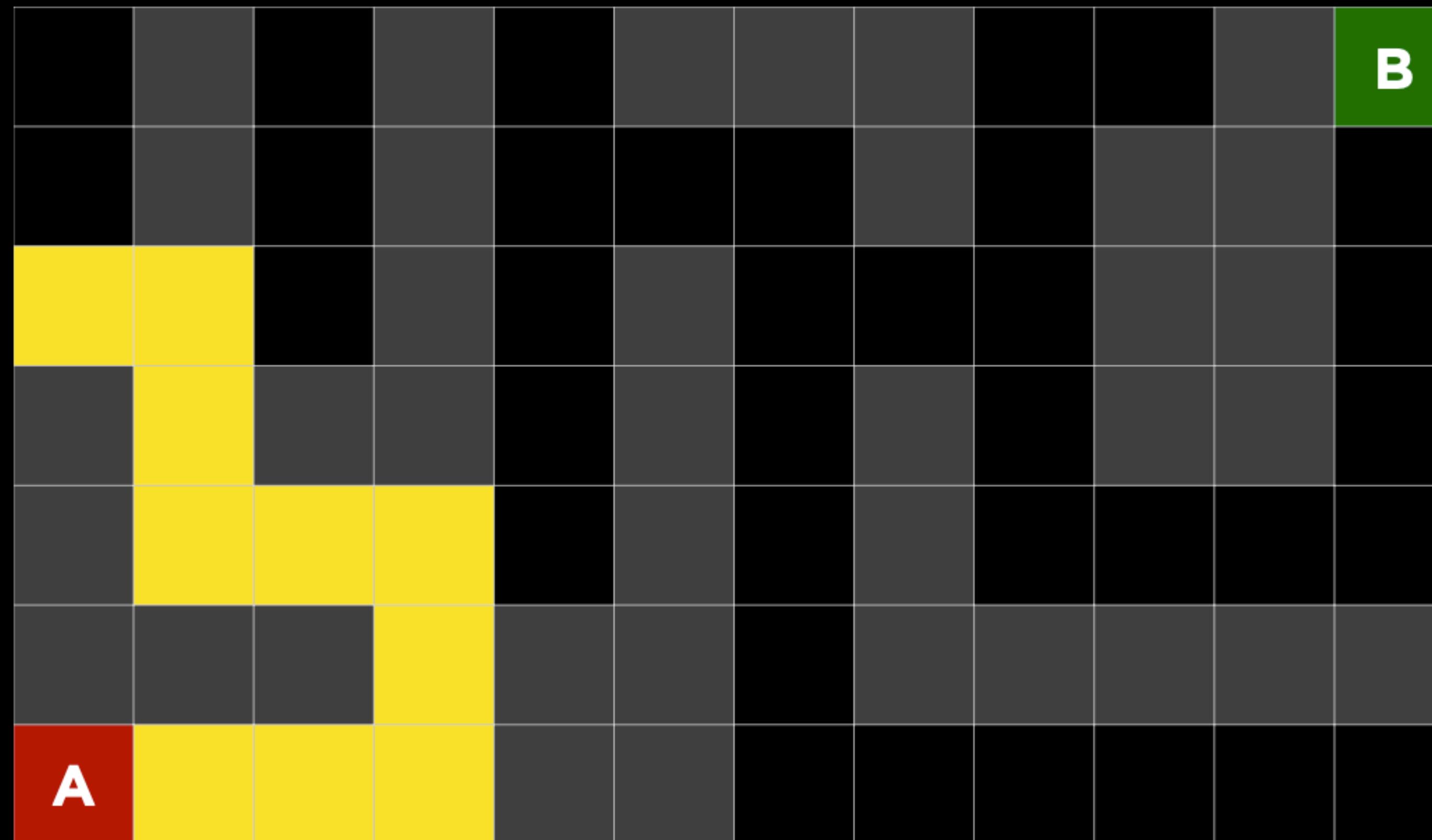
# Busca em Profundidade



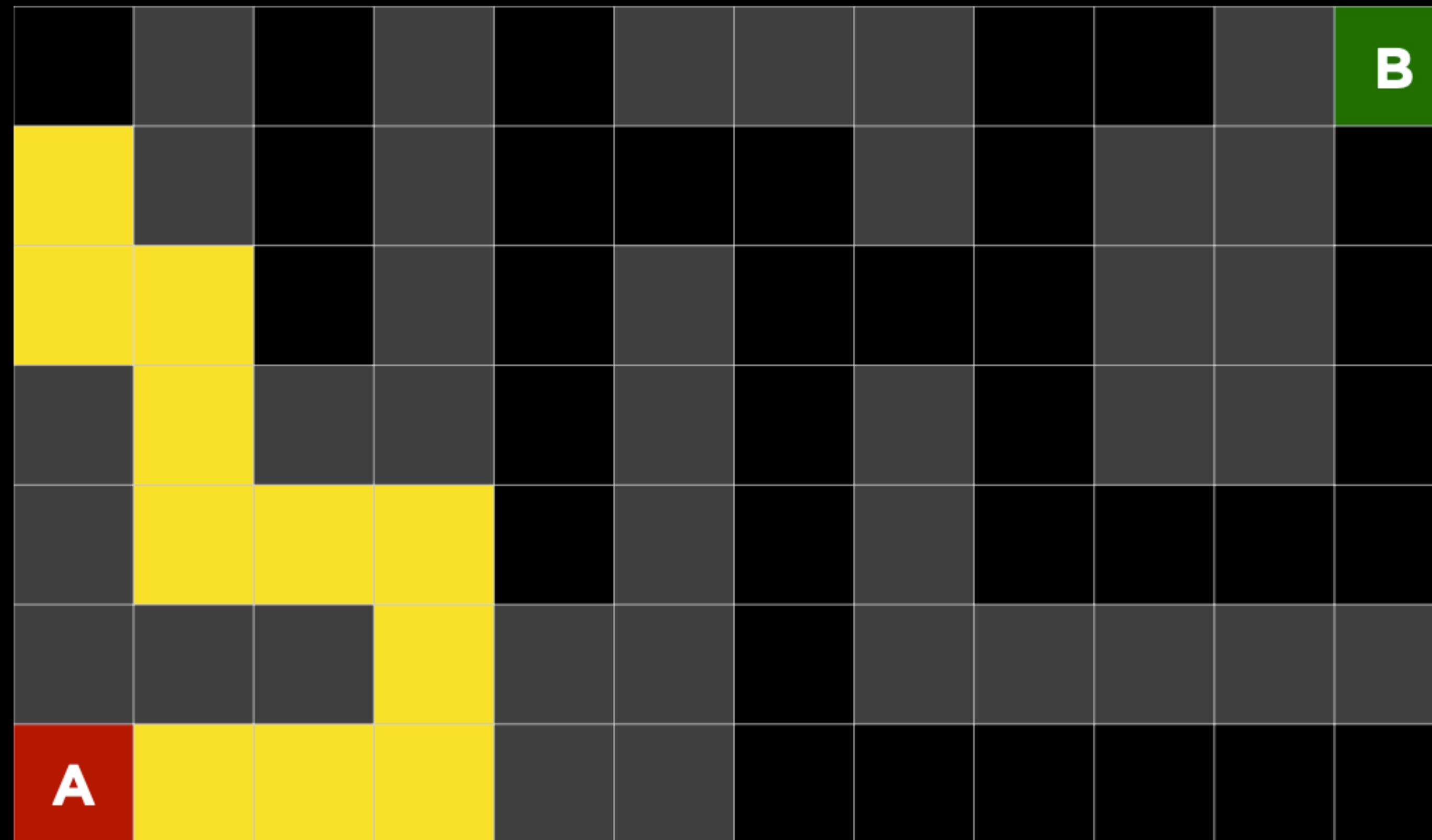
# Busca em Profundidade



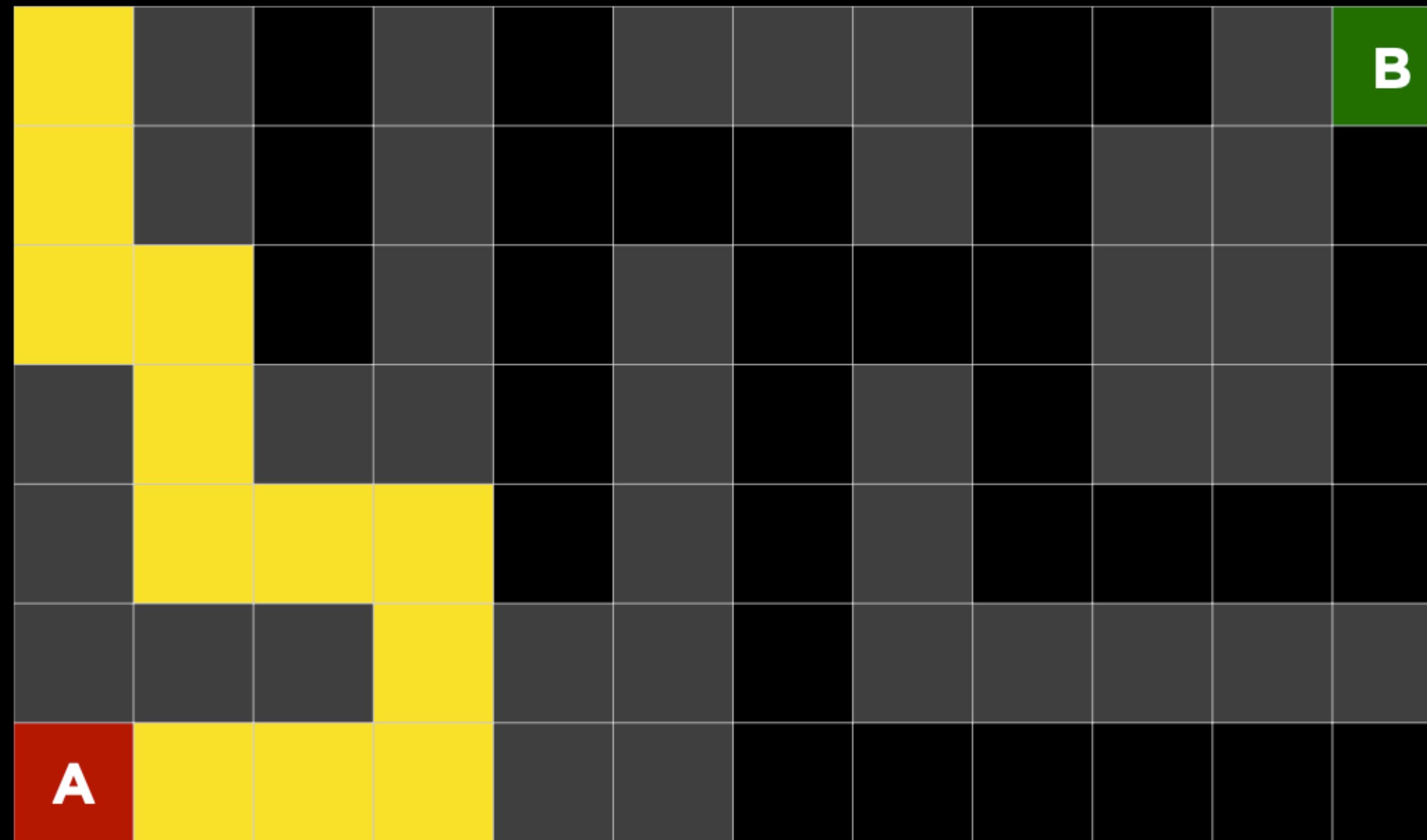
# Busca em Profundidade



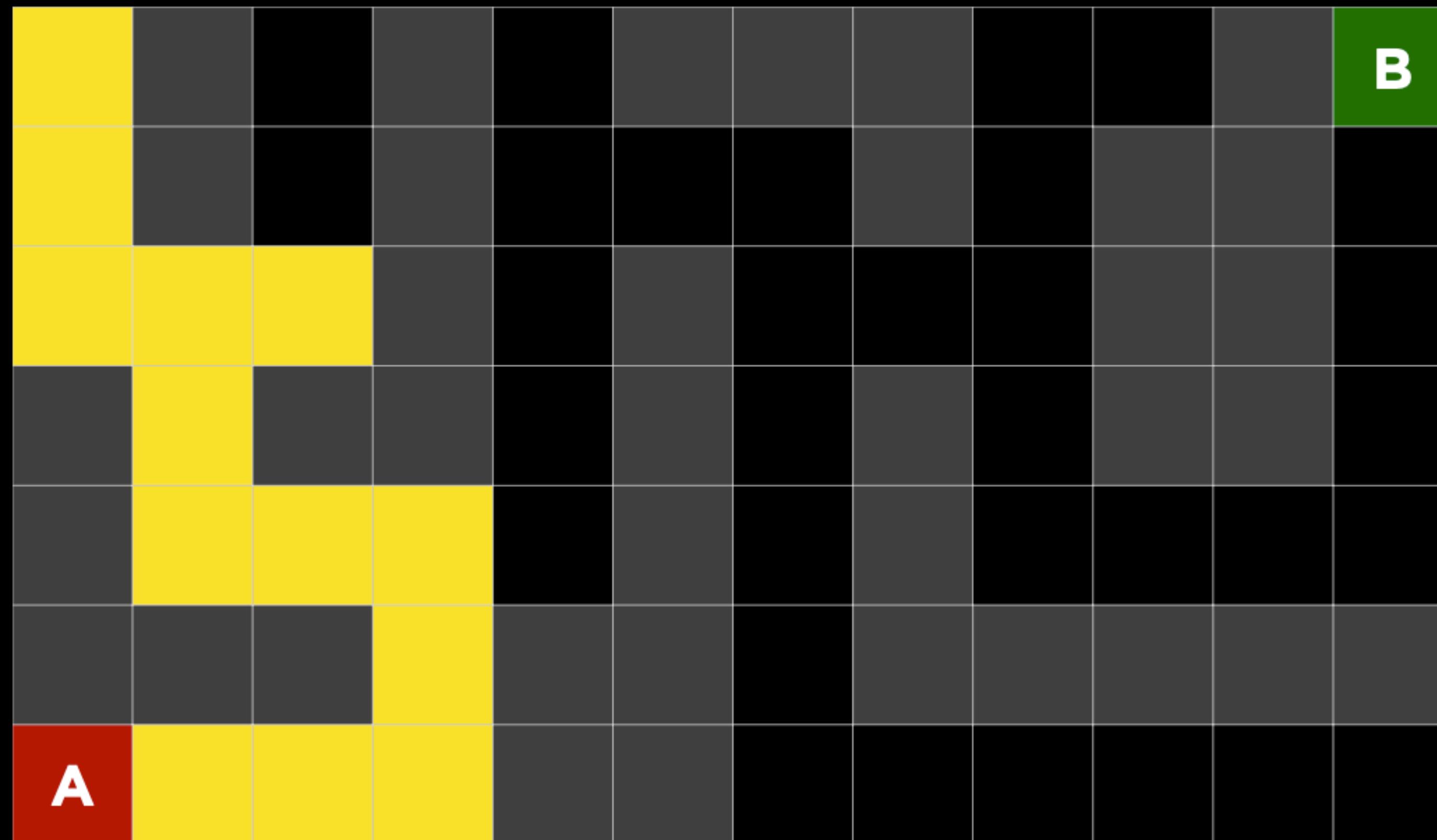
# Busca em Profundidade



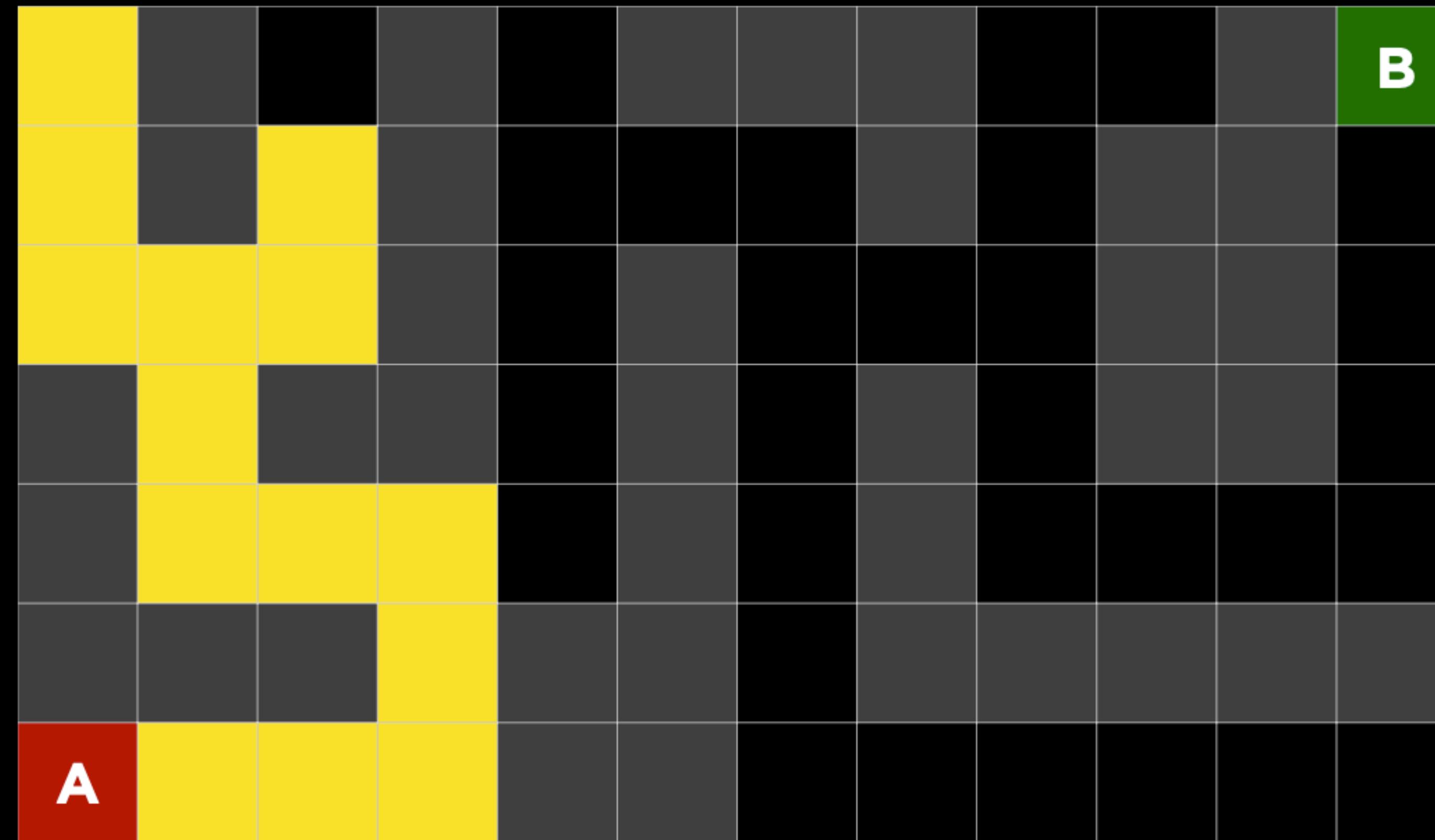
# Busca em Profundidade



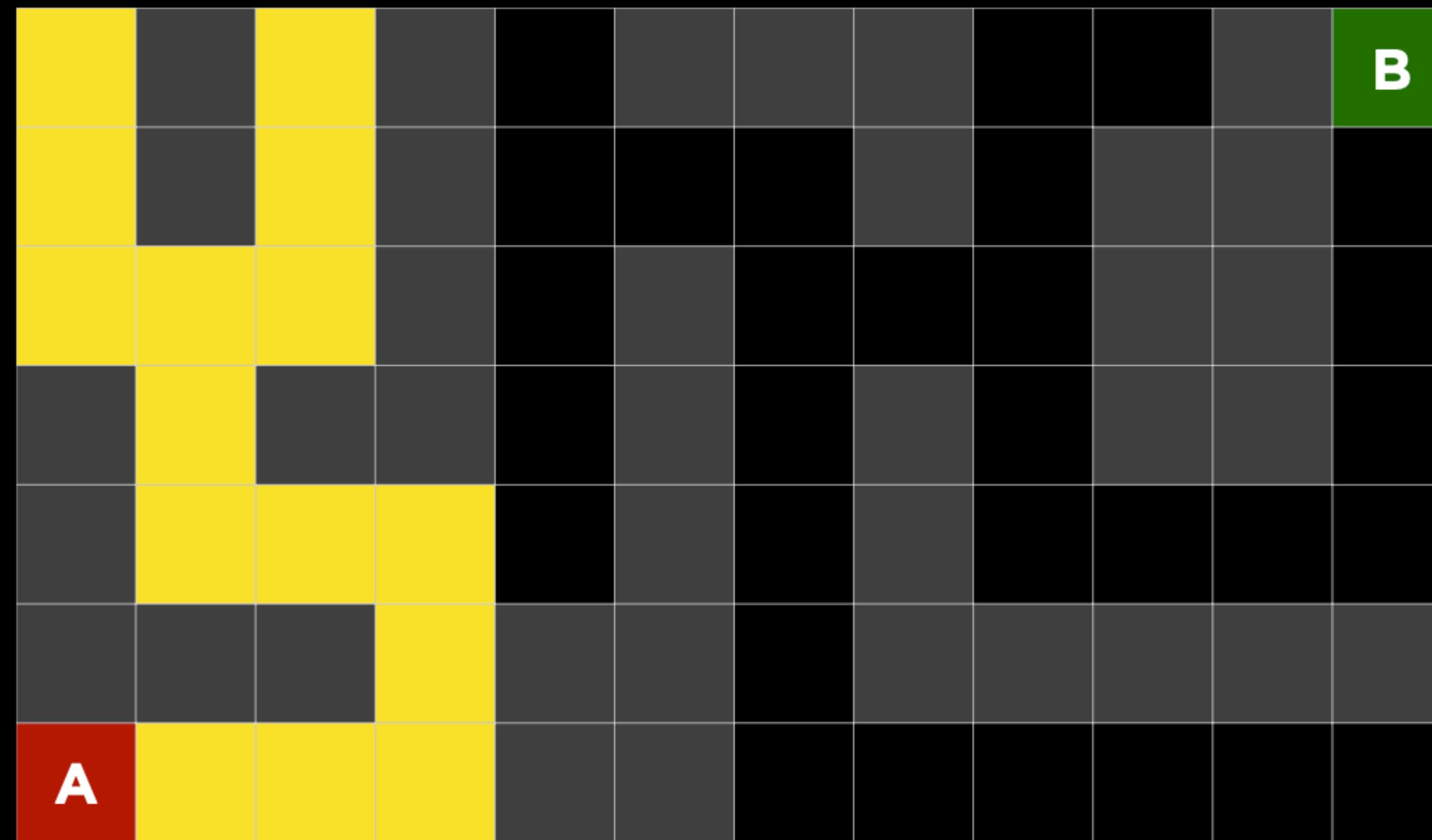
# Busca em Profundidade



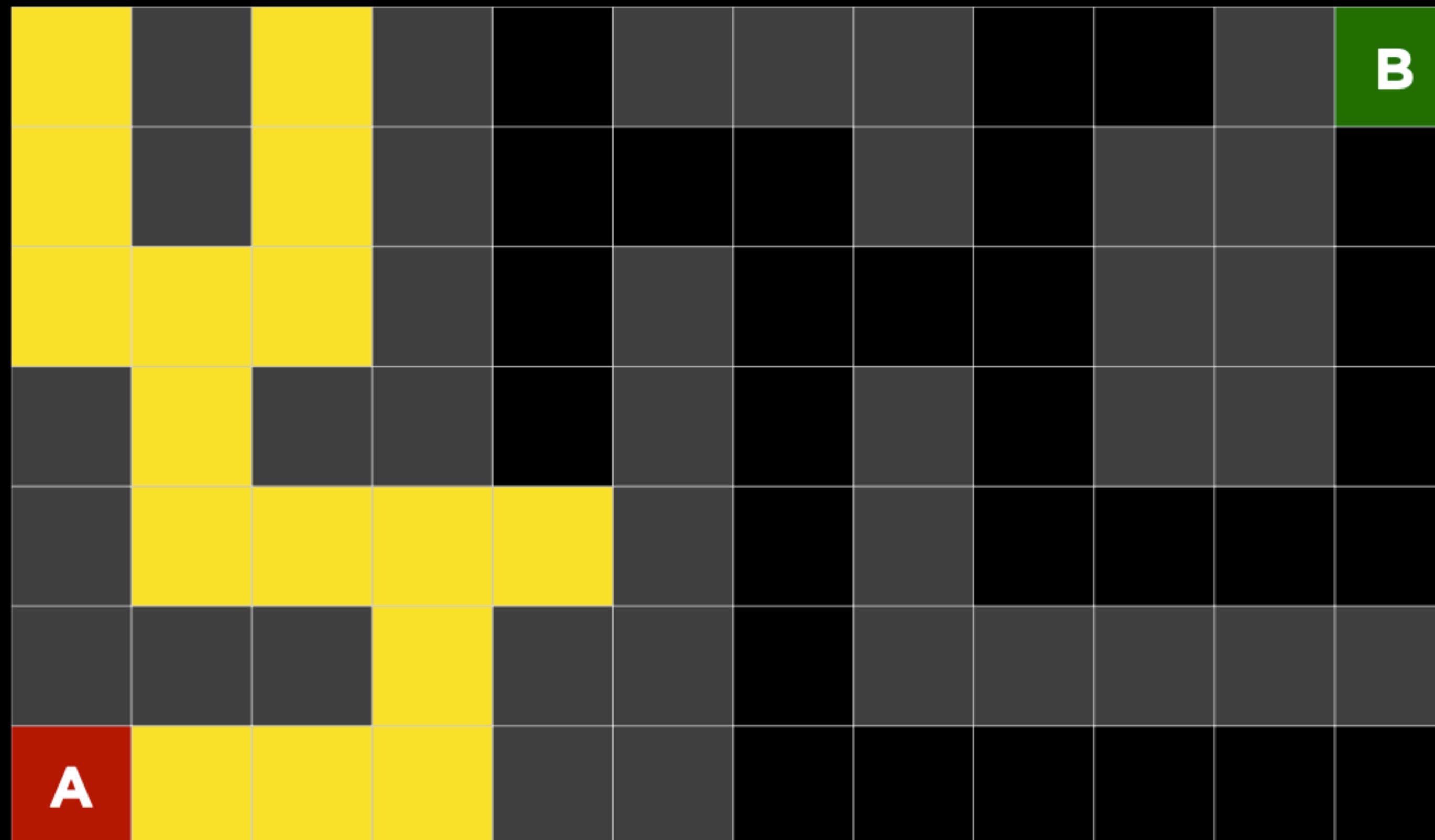
# Busca em Profundidade



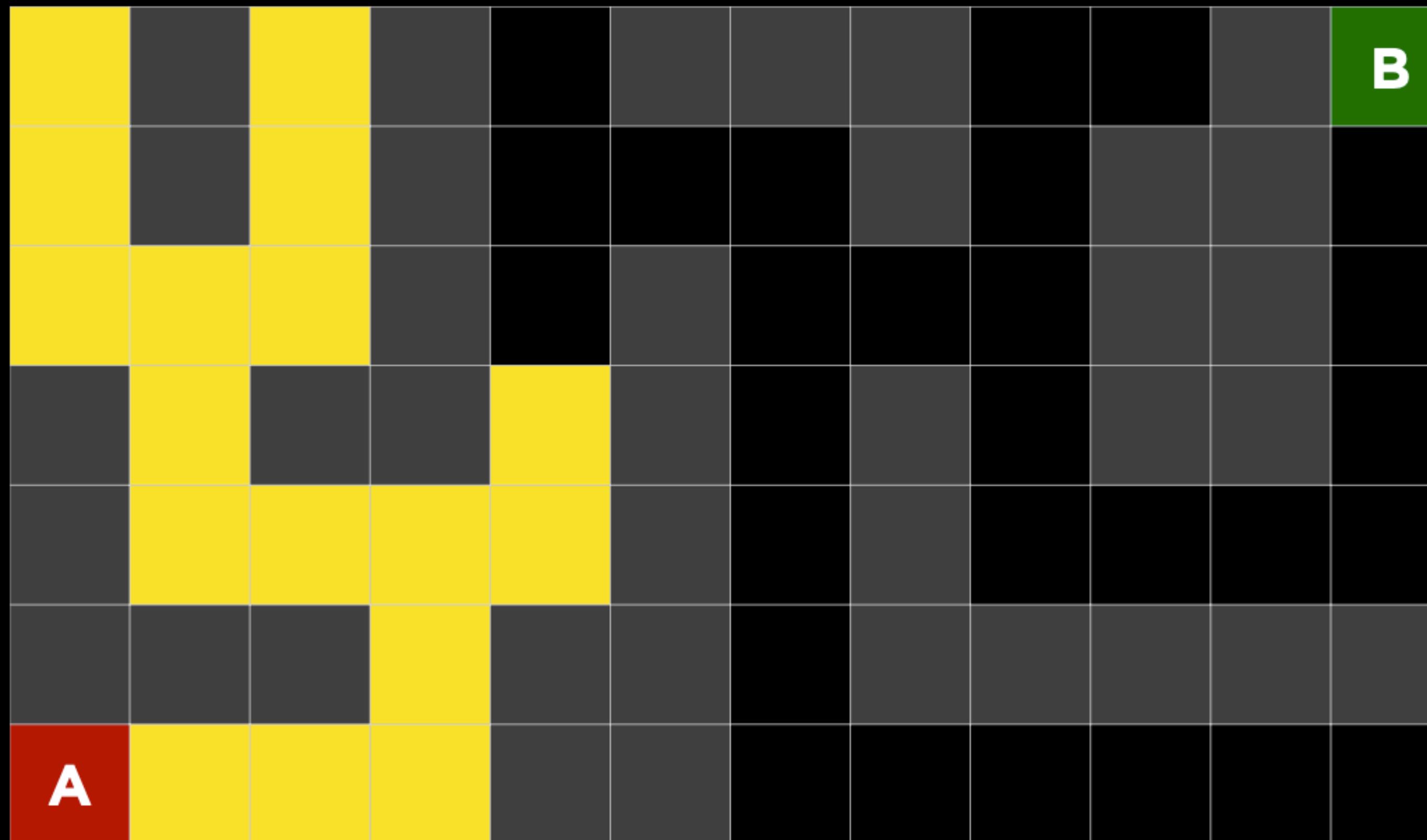
# Busca em Profundidade



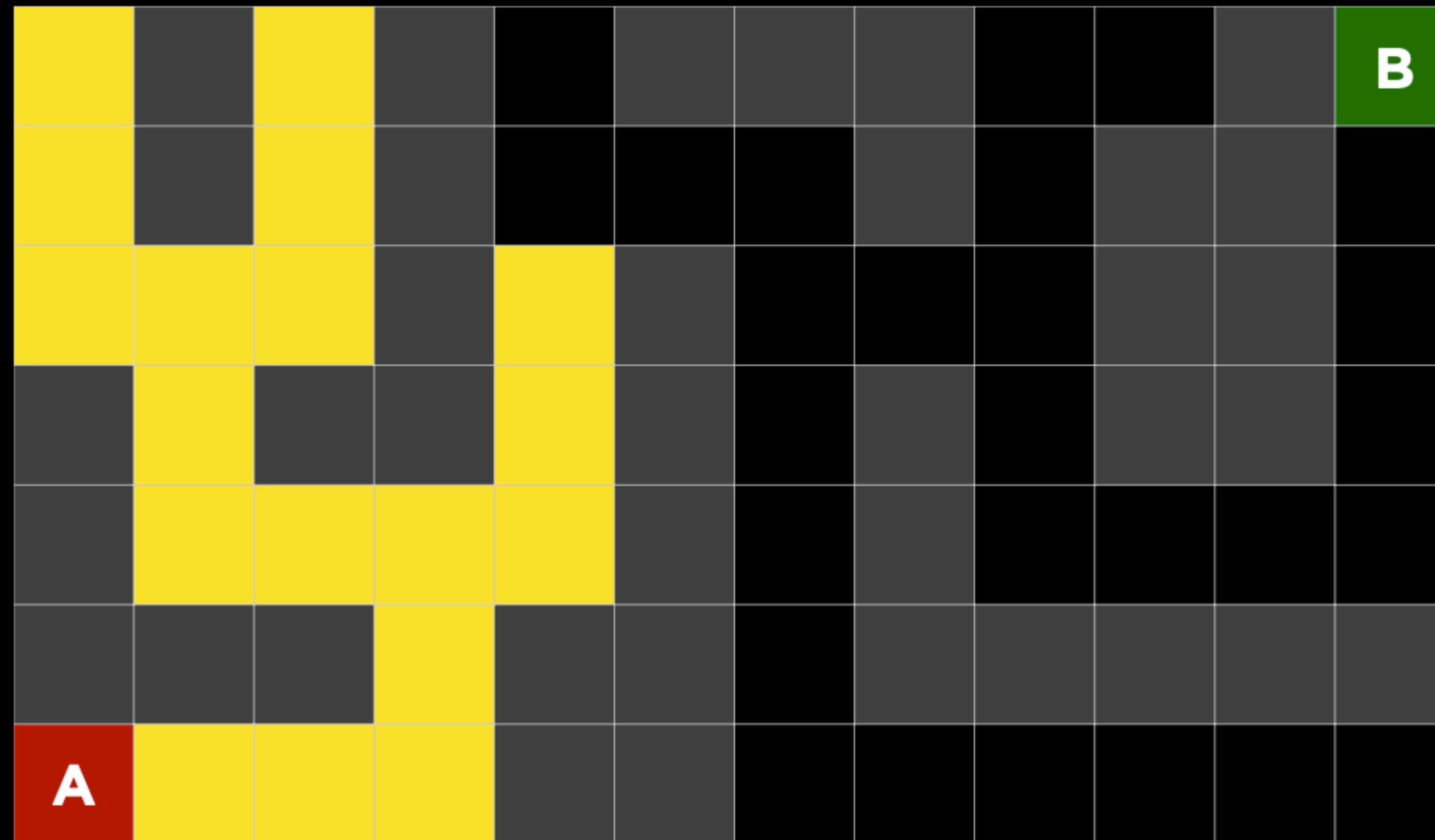
# Busca em Profundidade



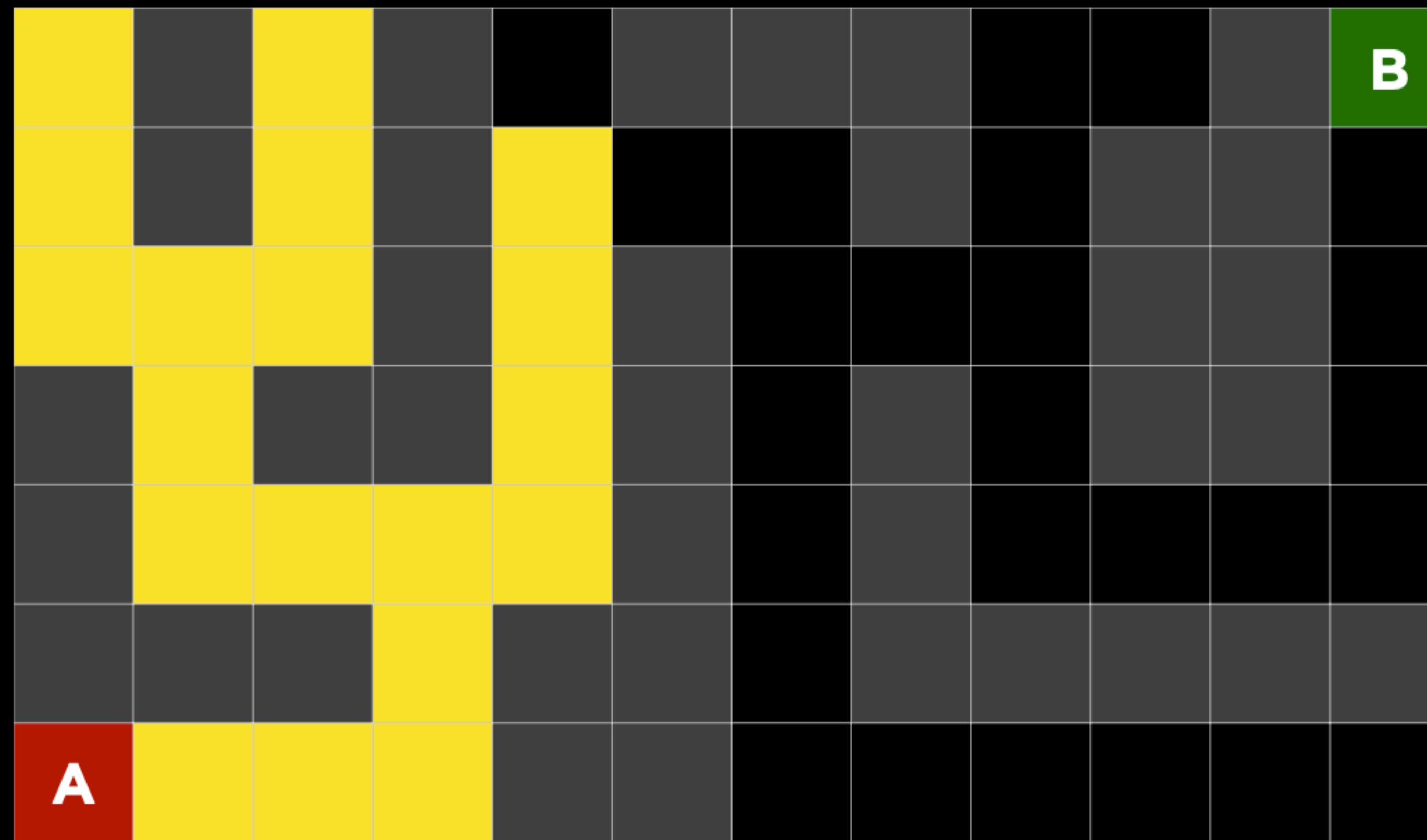
# Busca em Profundidade



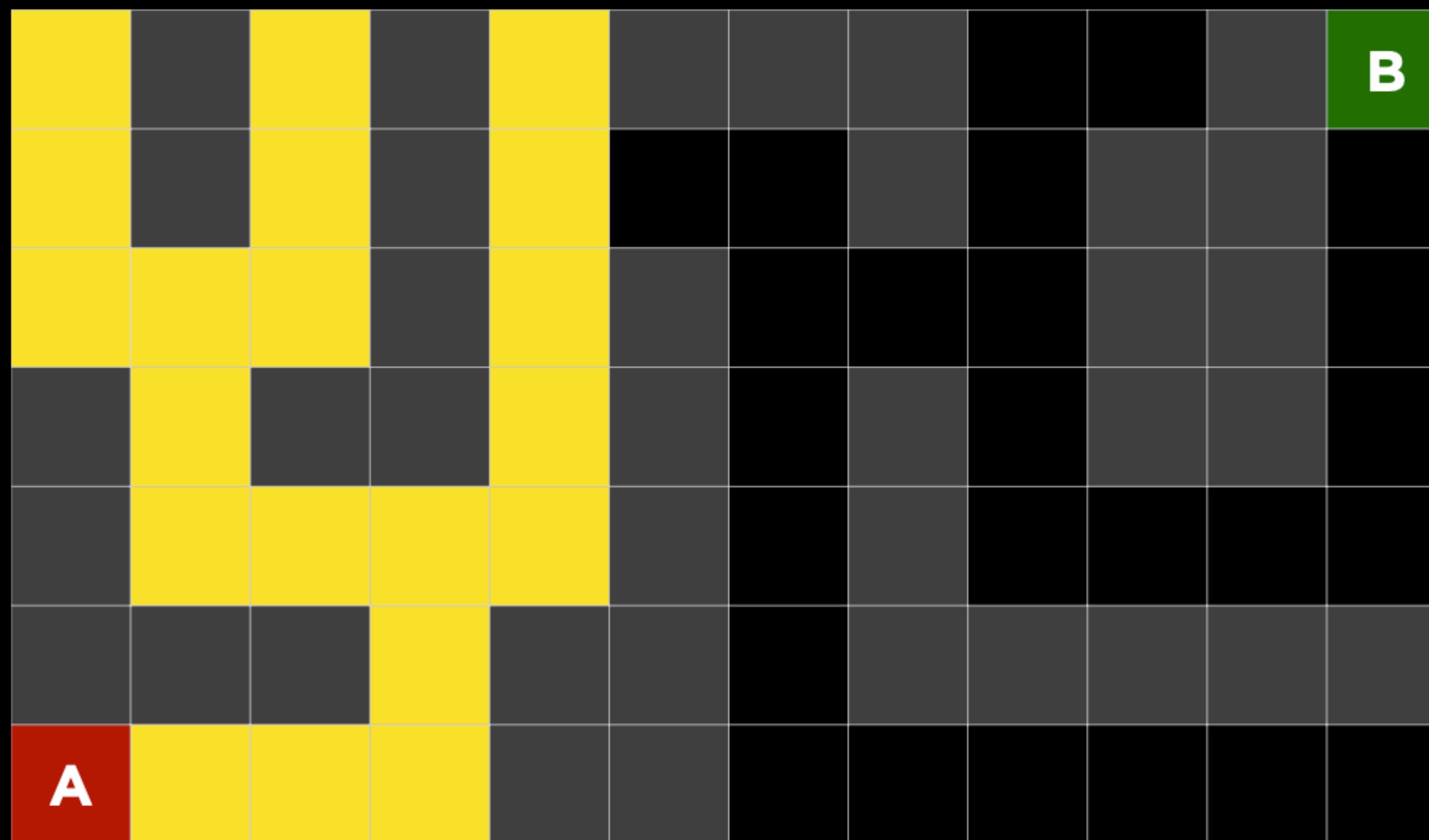
# Busca em Profundidade



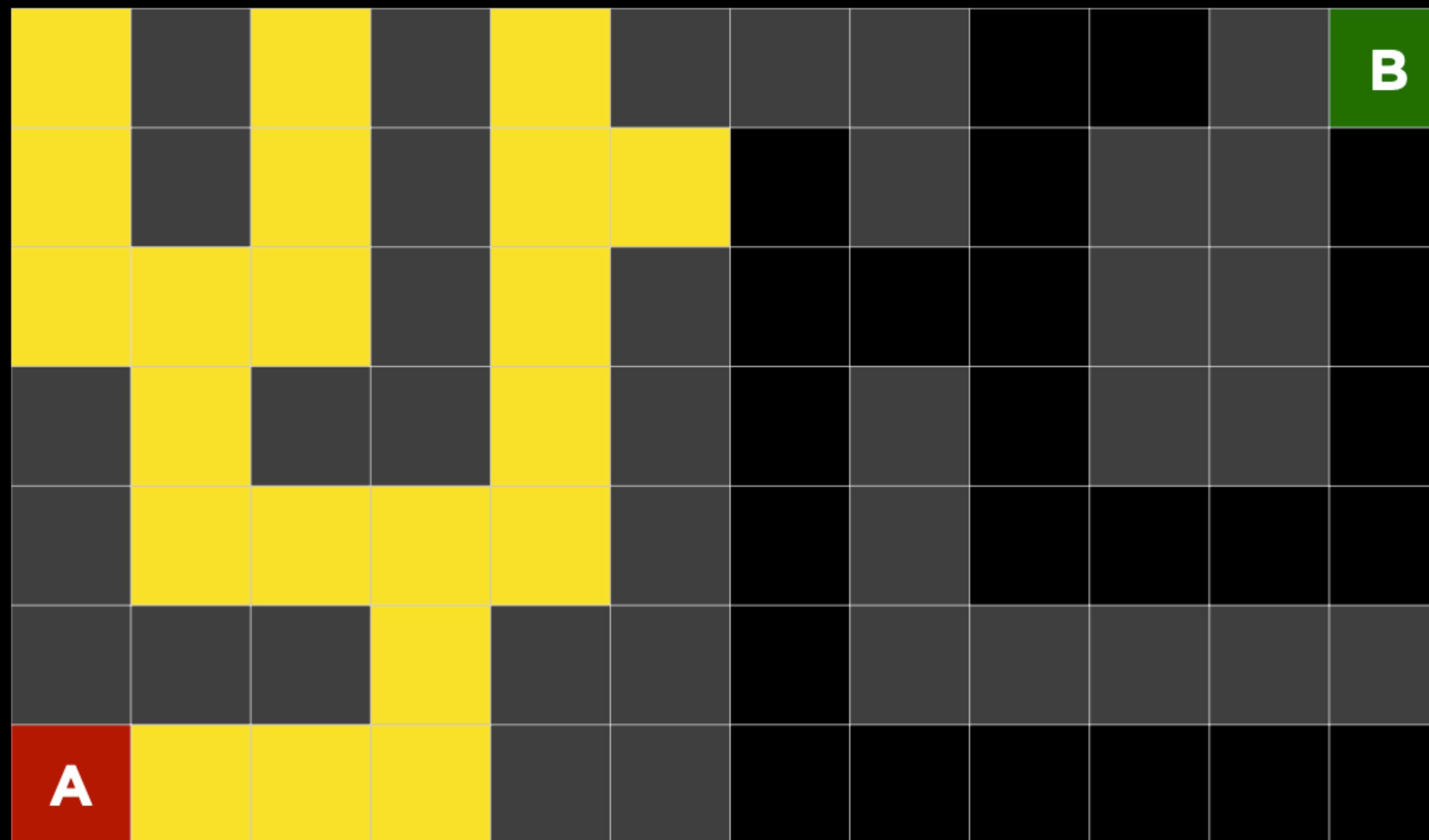
# Busca em Profundidade



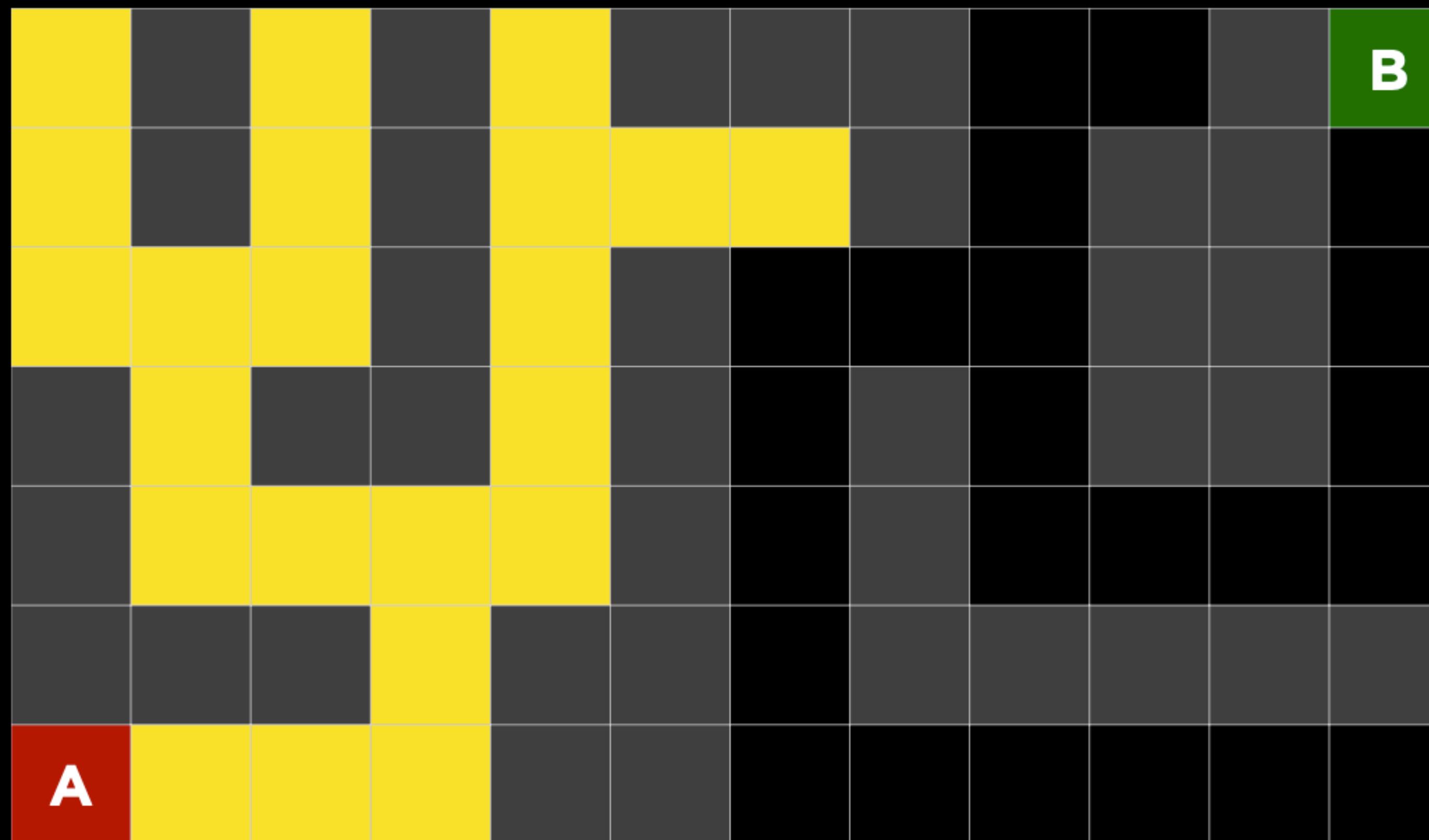
# Busca em Profundidade



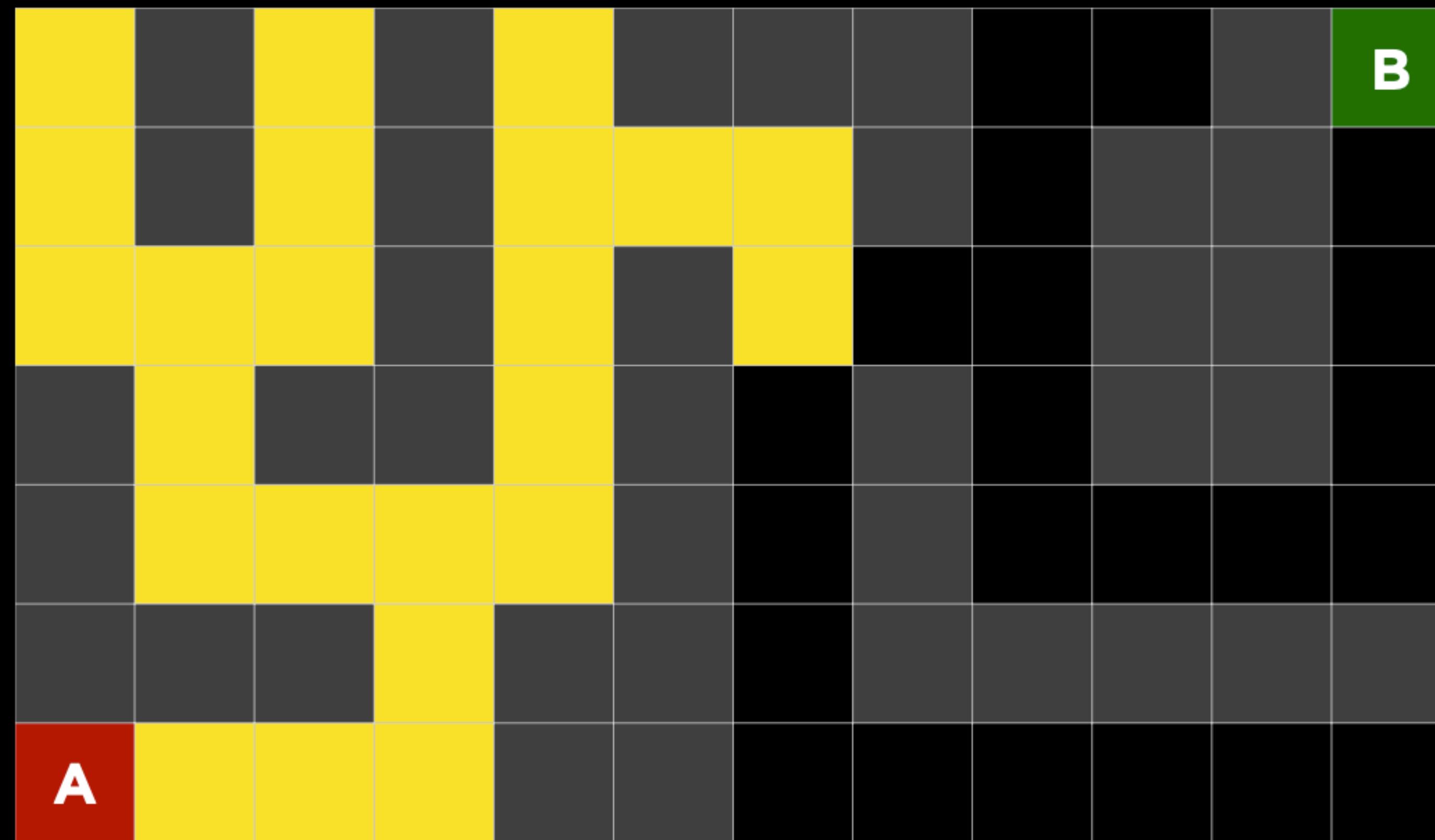
# Busca em Profundidade



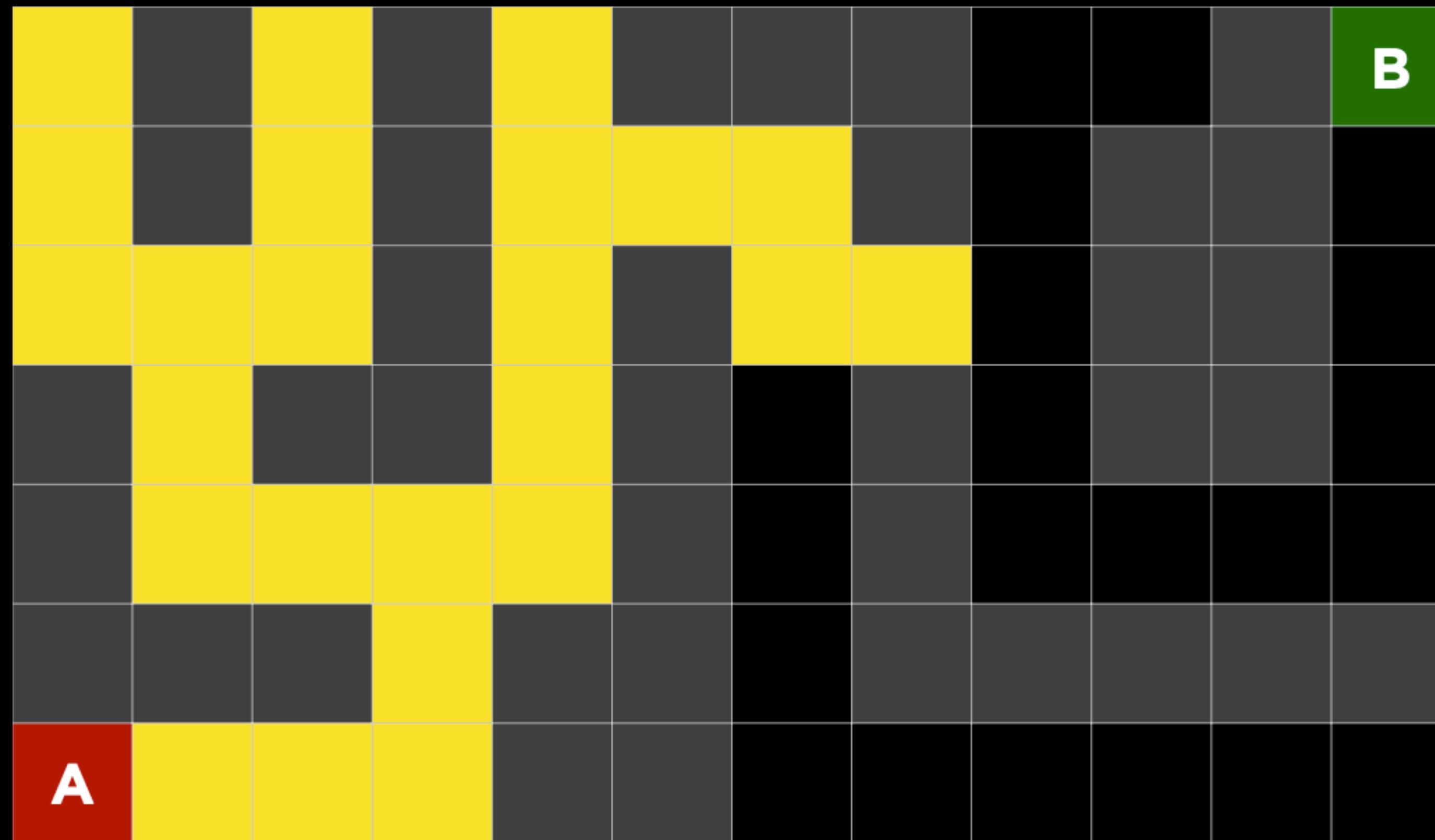
# Busca em Profundidade



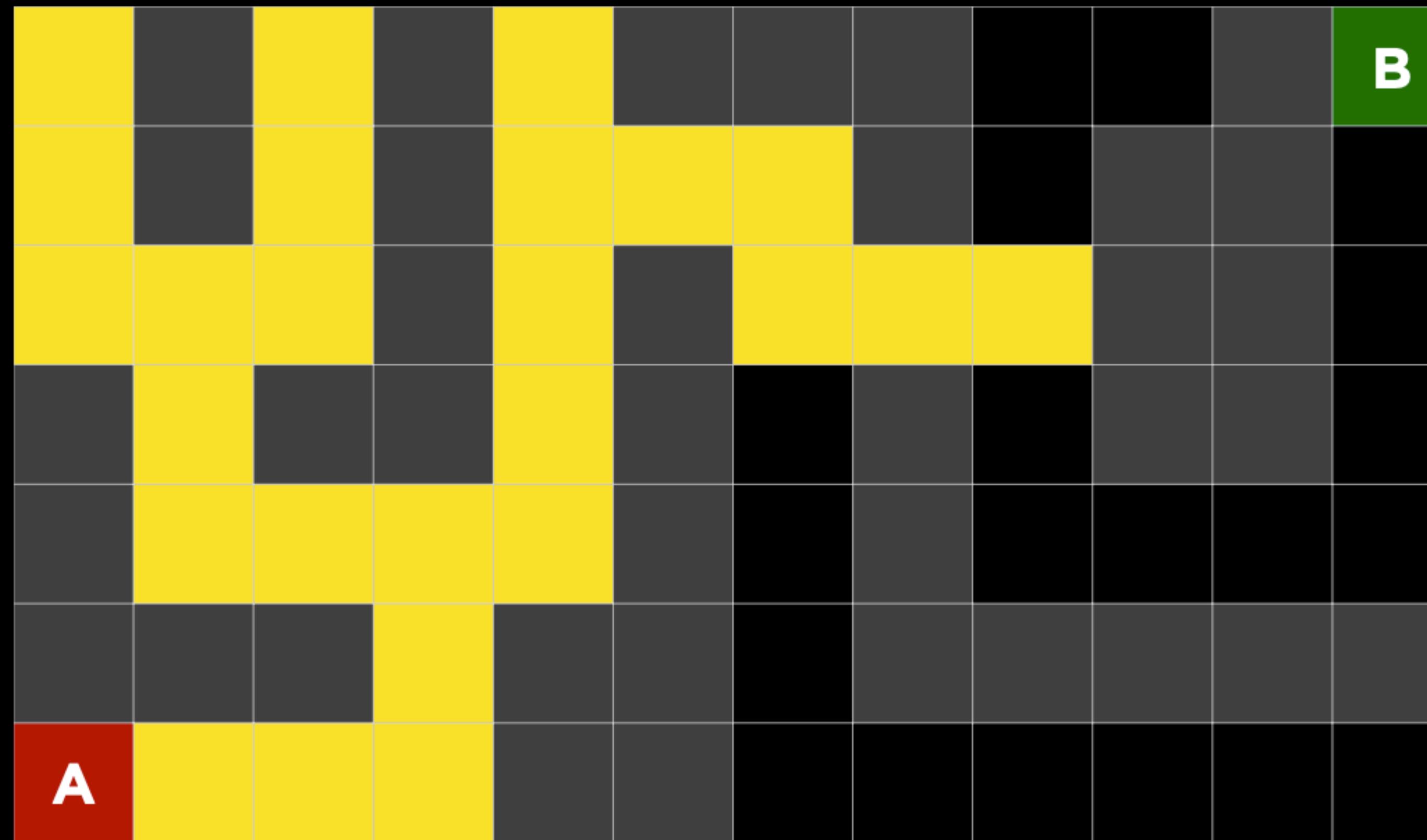
# Busca em Profundidade



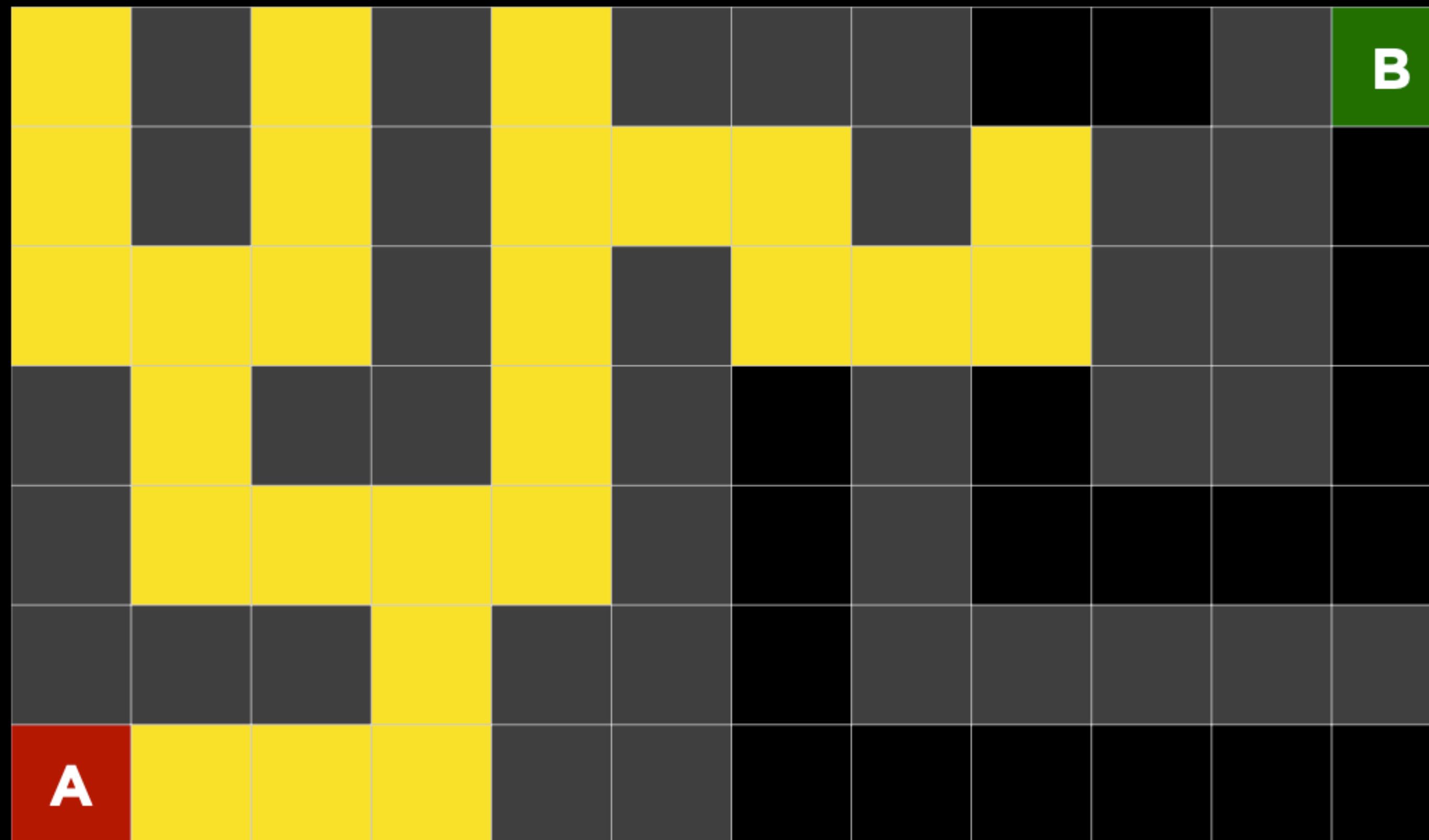
# Busca em Profundidade



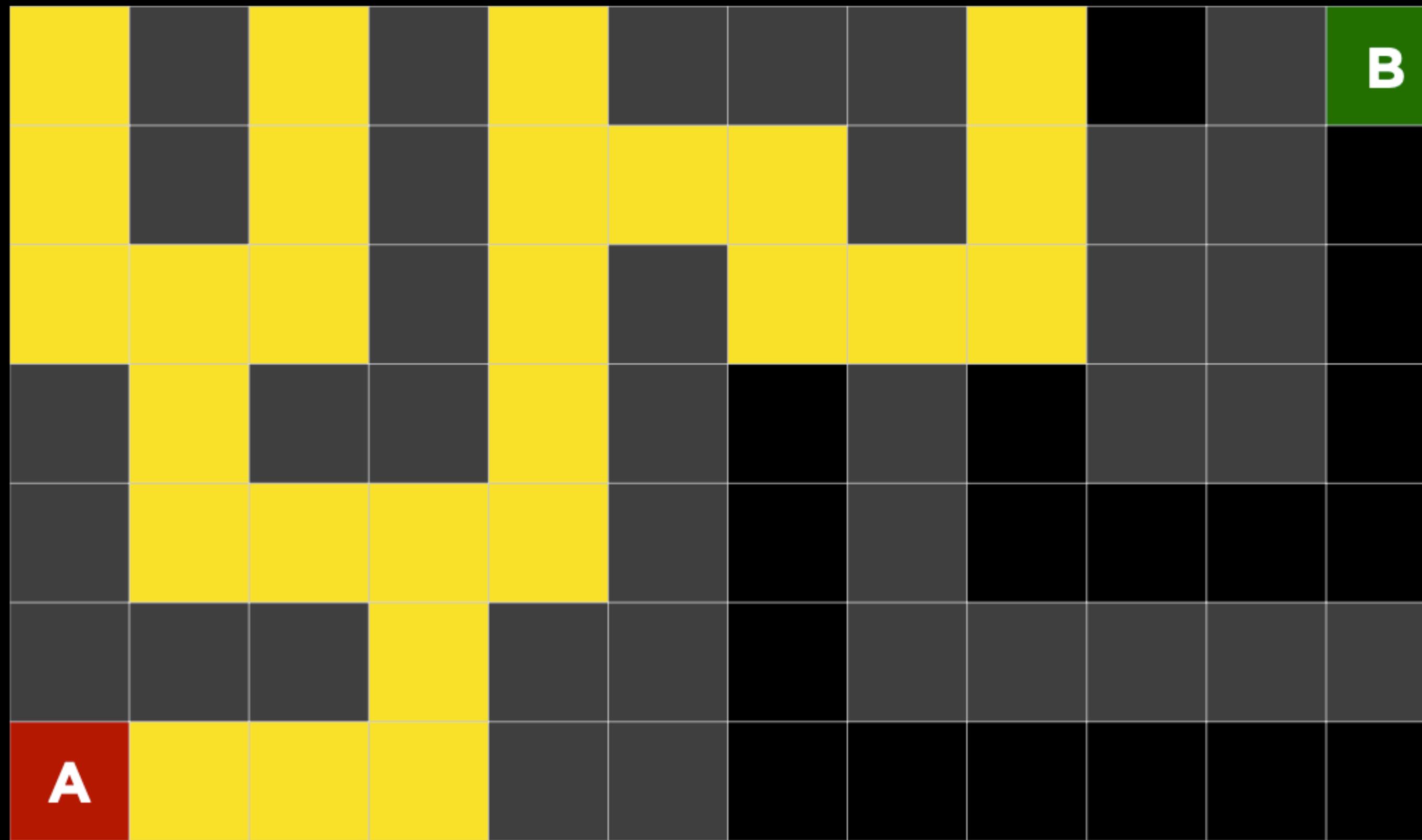
# Busca em Profundidade



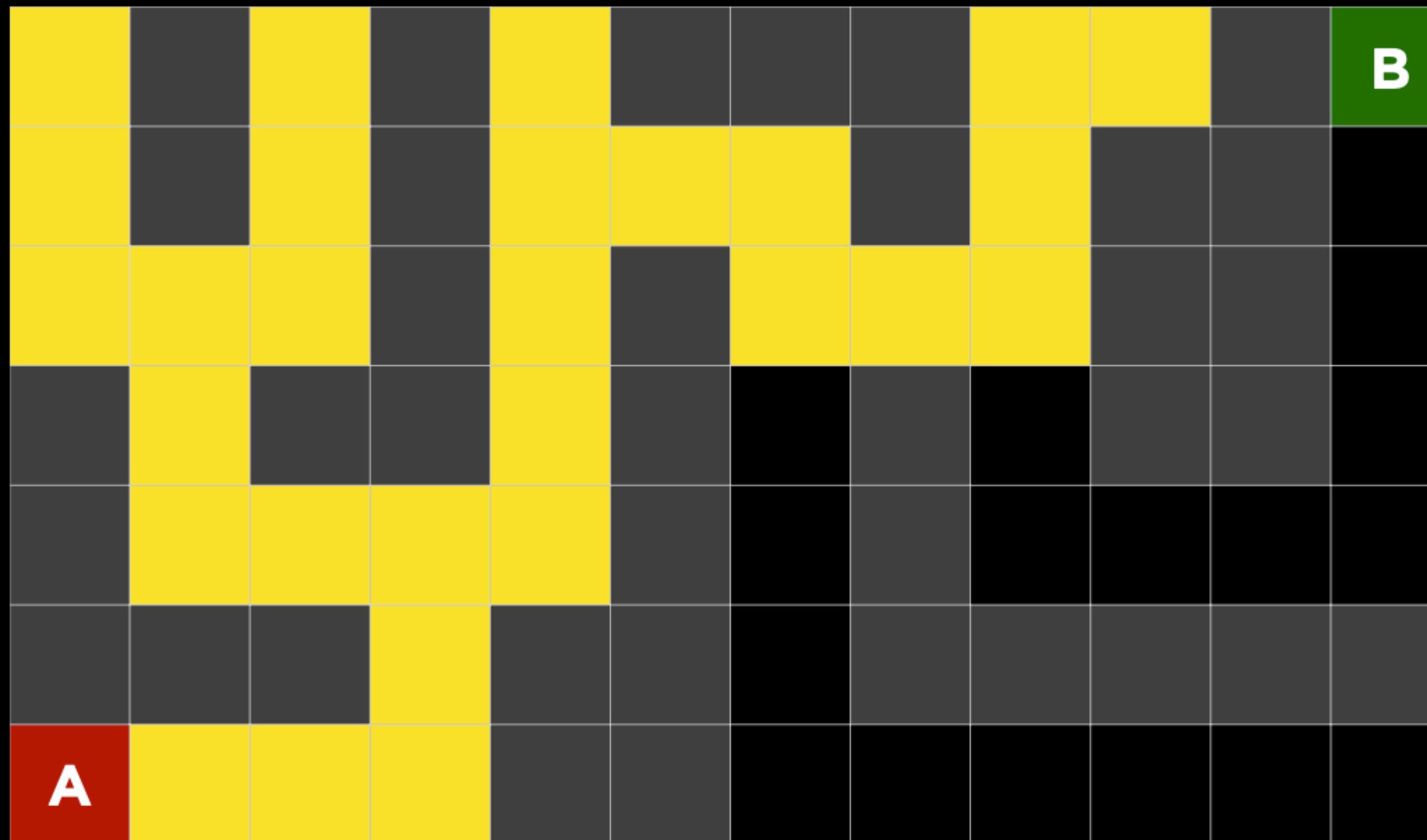
# Busca em Profundidade



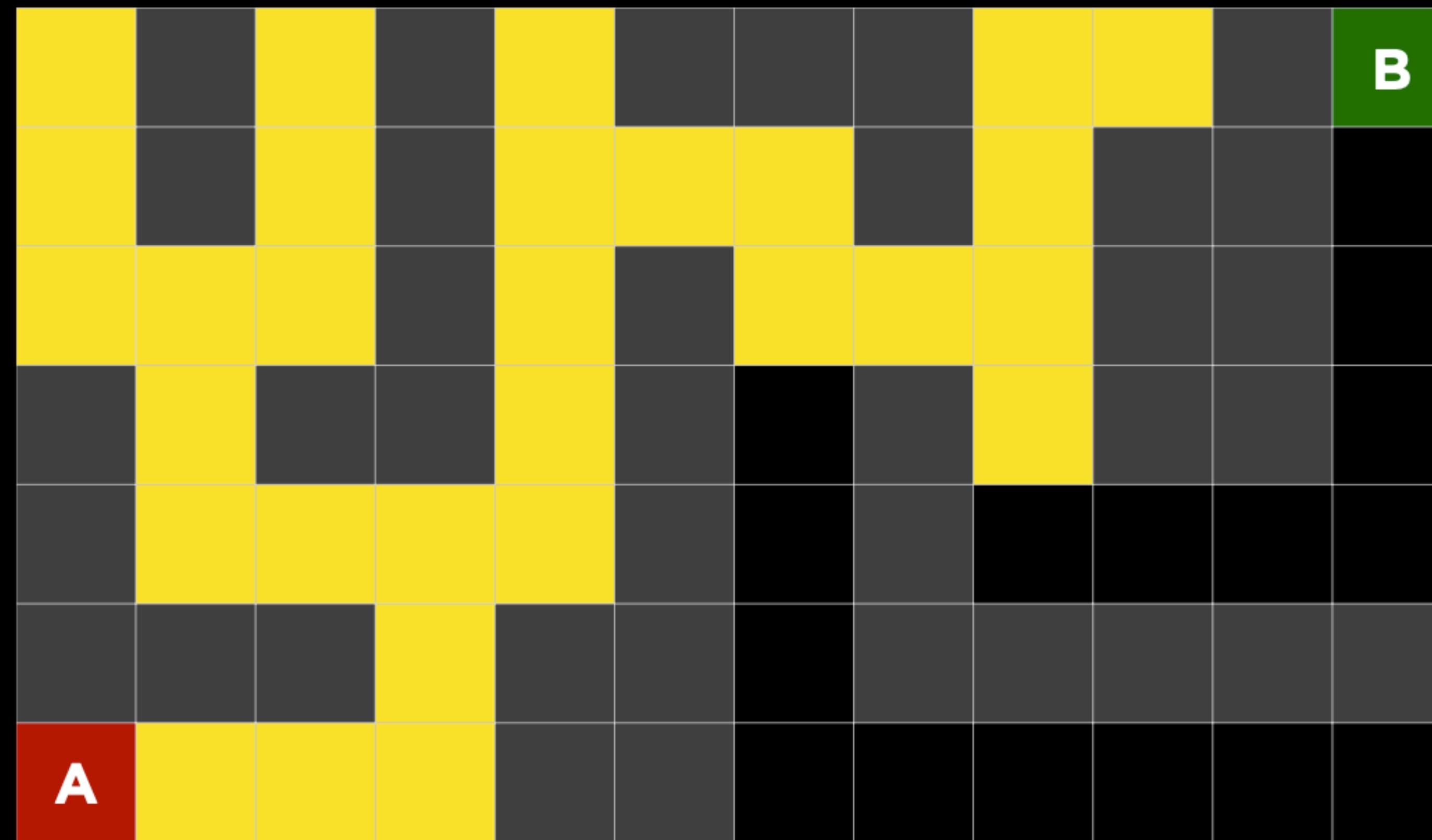
# Busca em Profundidade



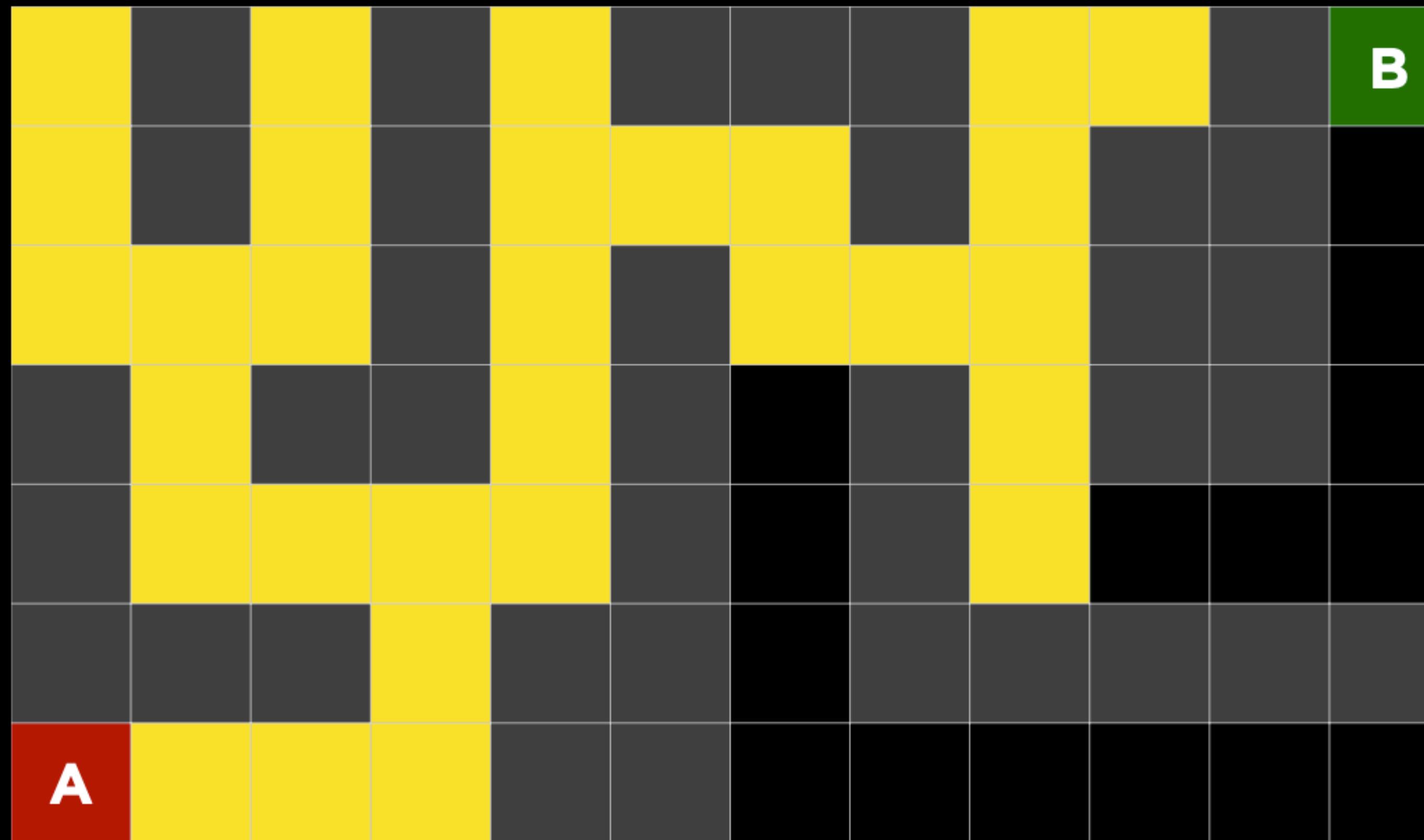
# Busca em Profundidade



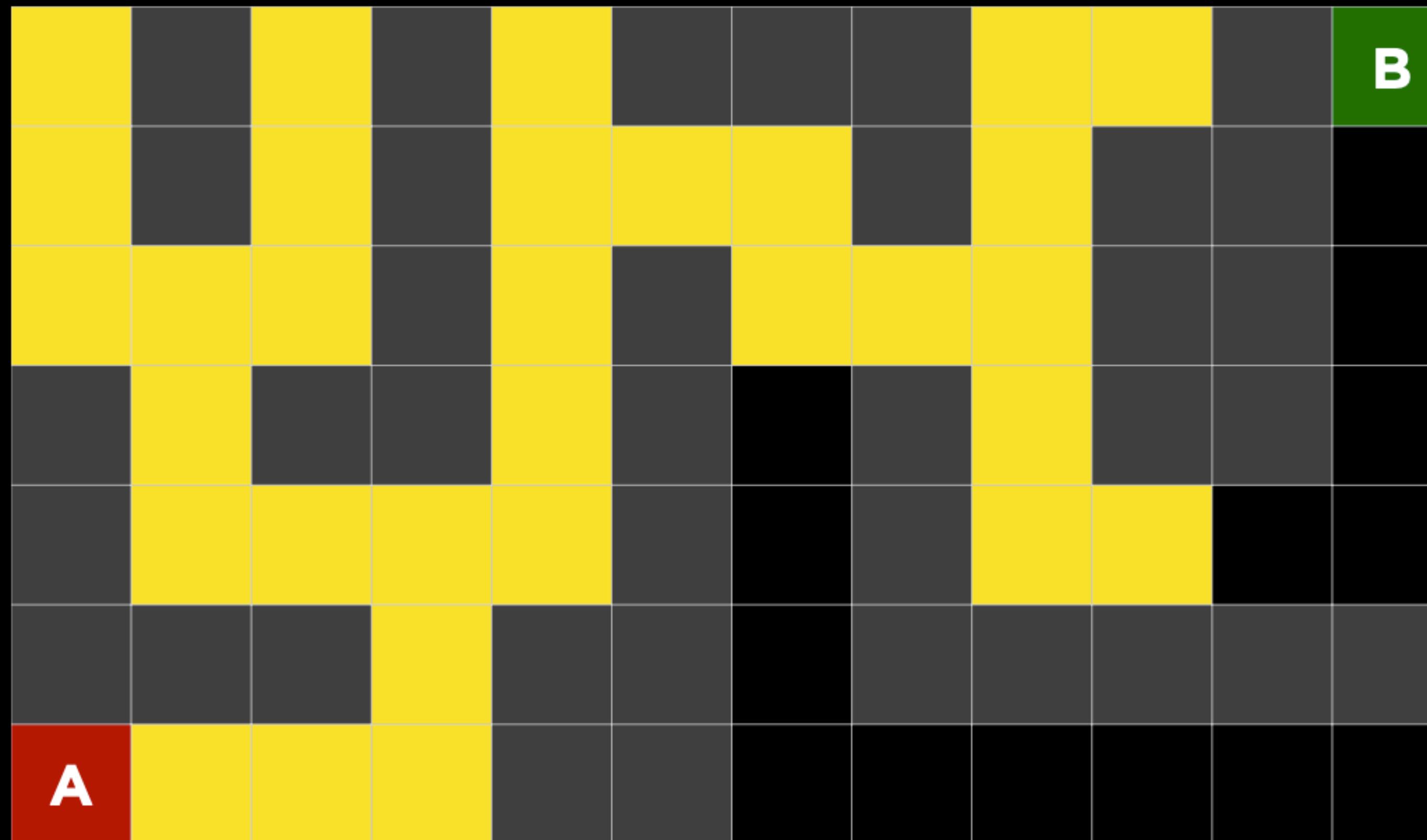
# Busca em Profundidade



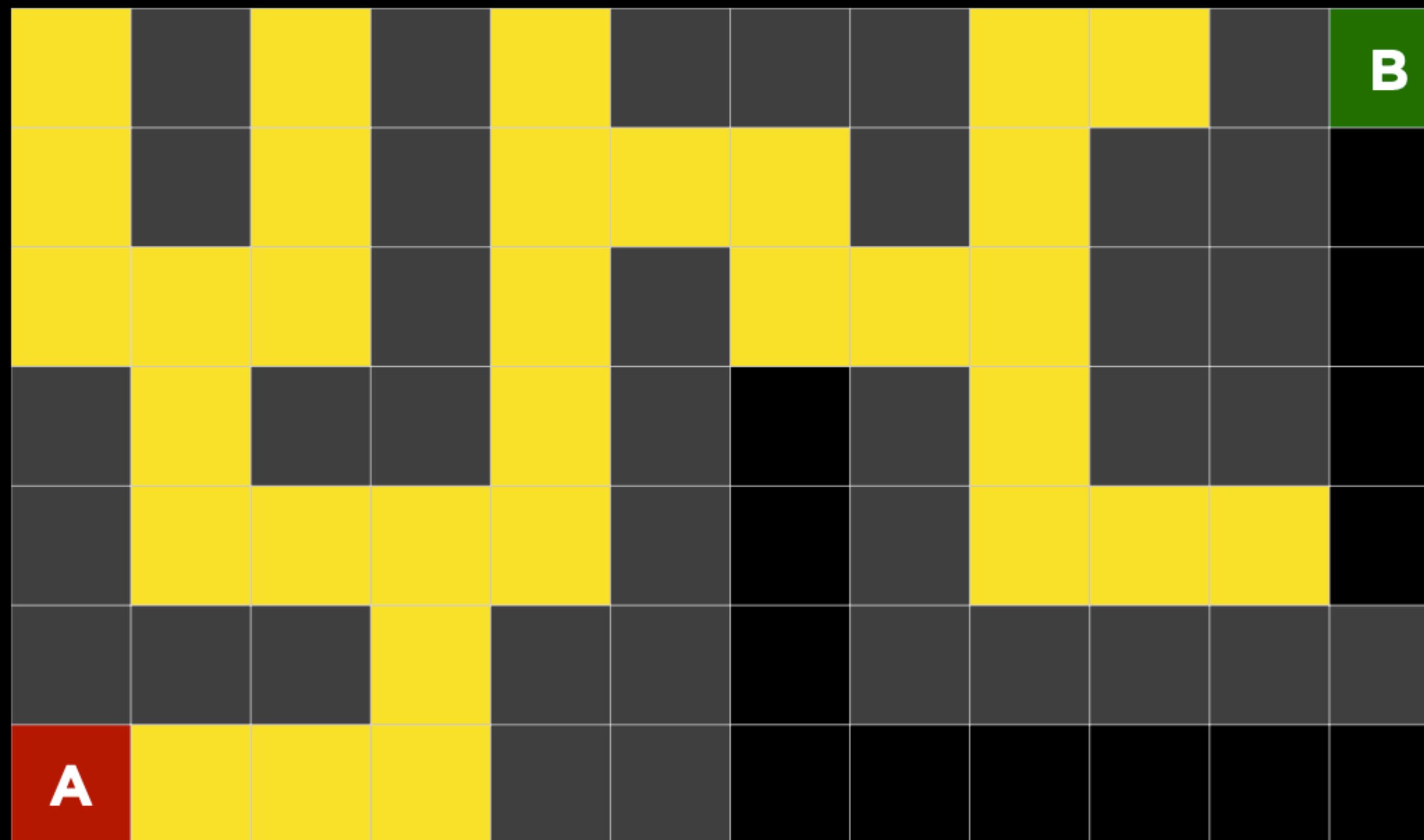
# Busca em Profundidade



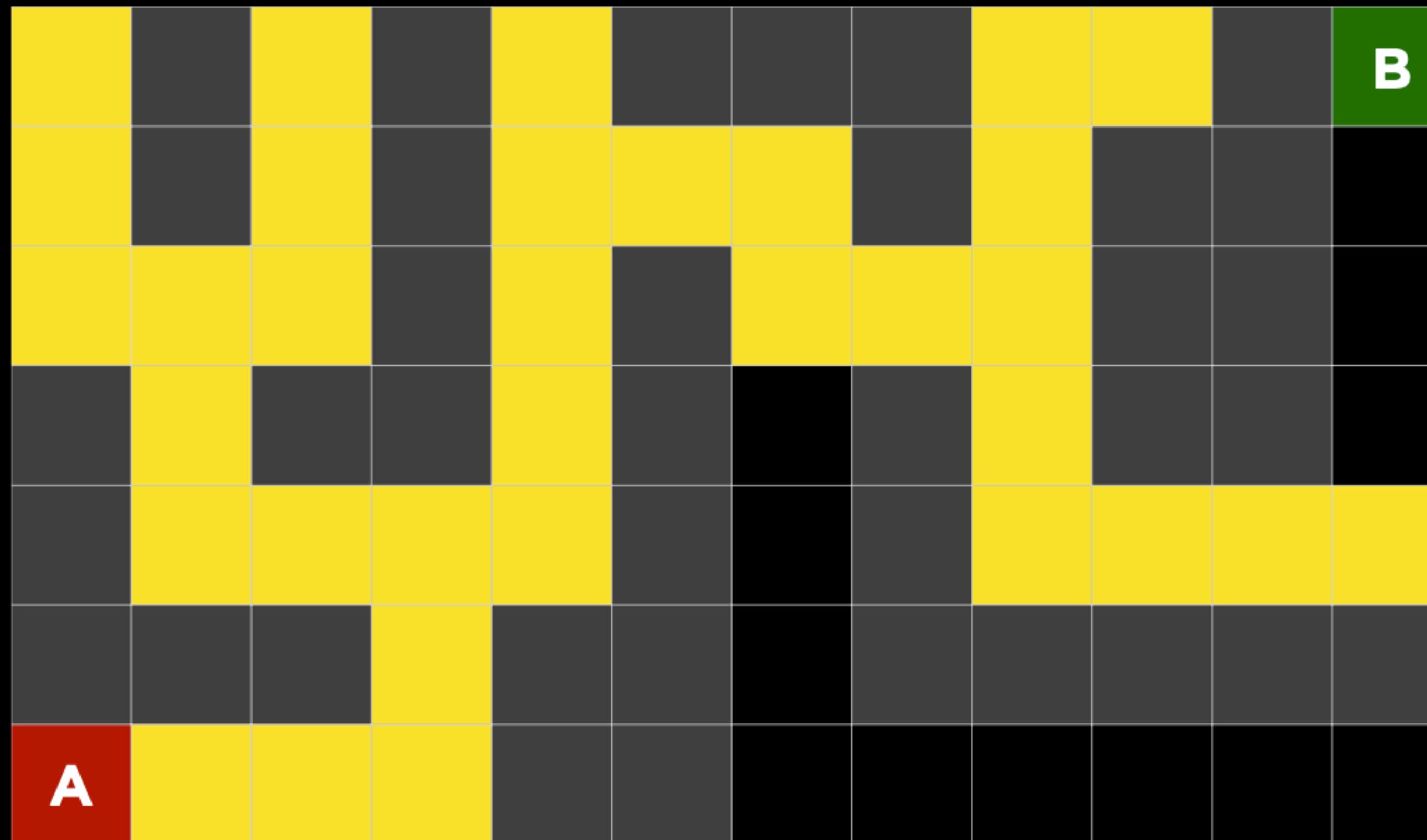
# Busca em Profundidade



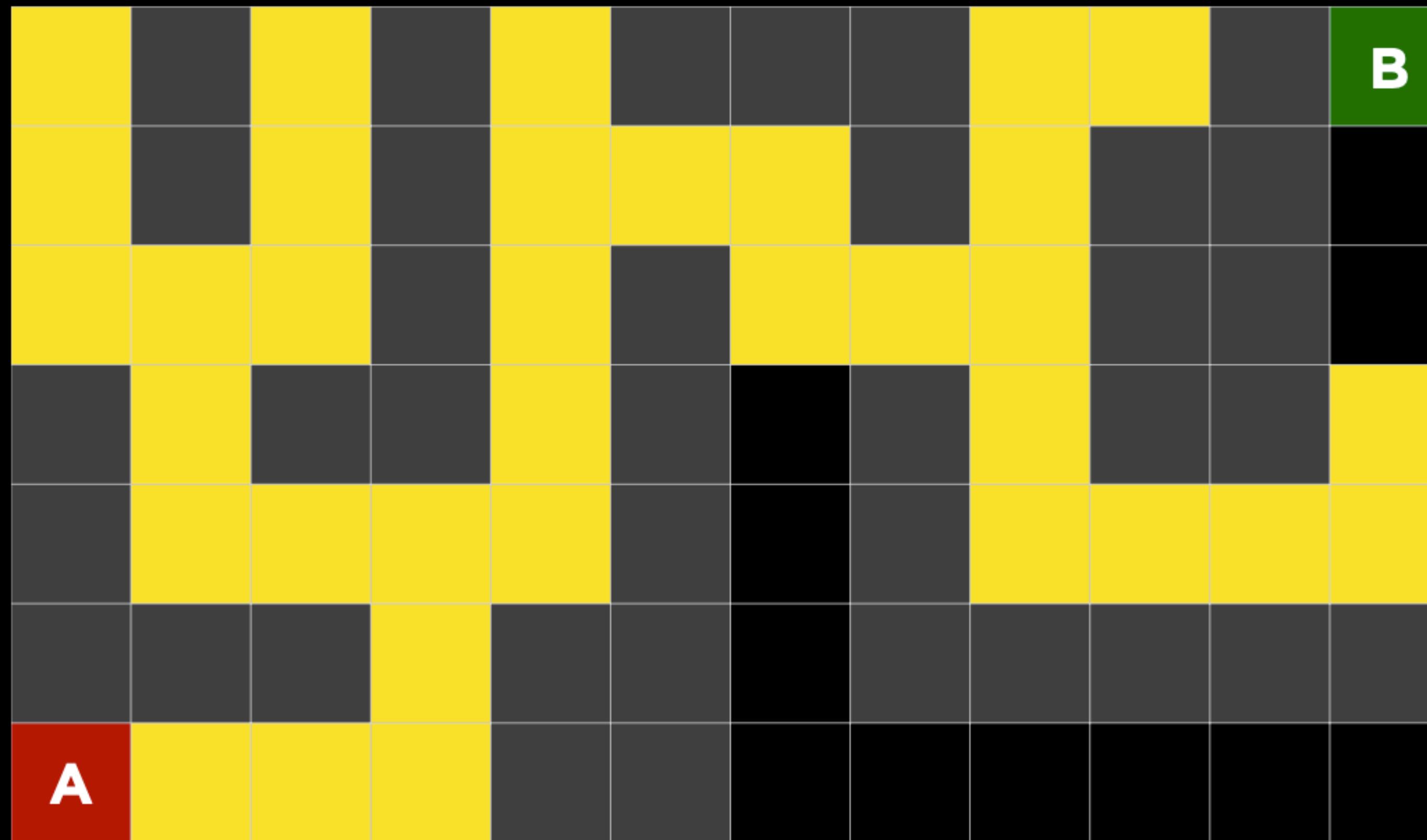
# Busca em Profundidade



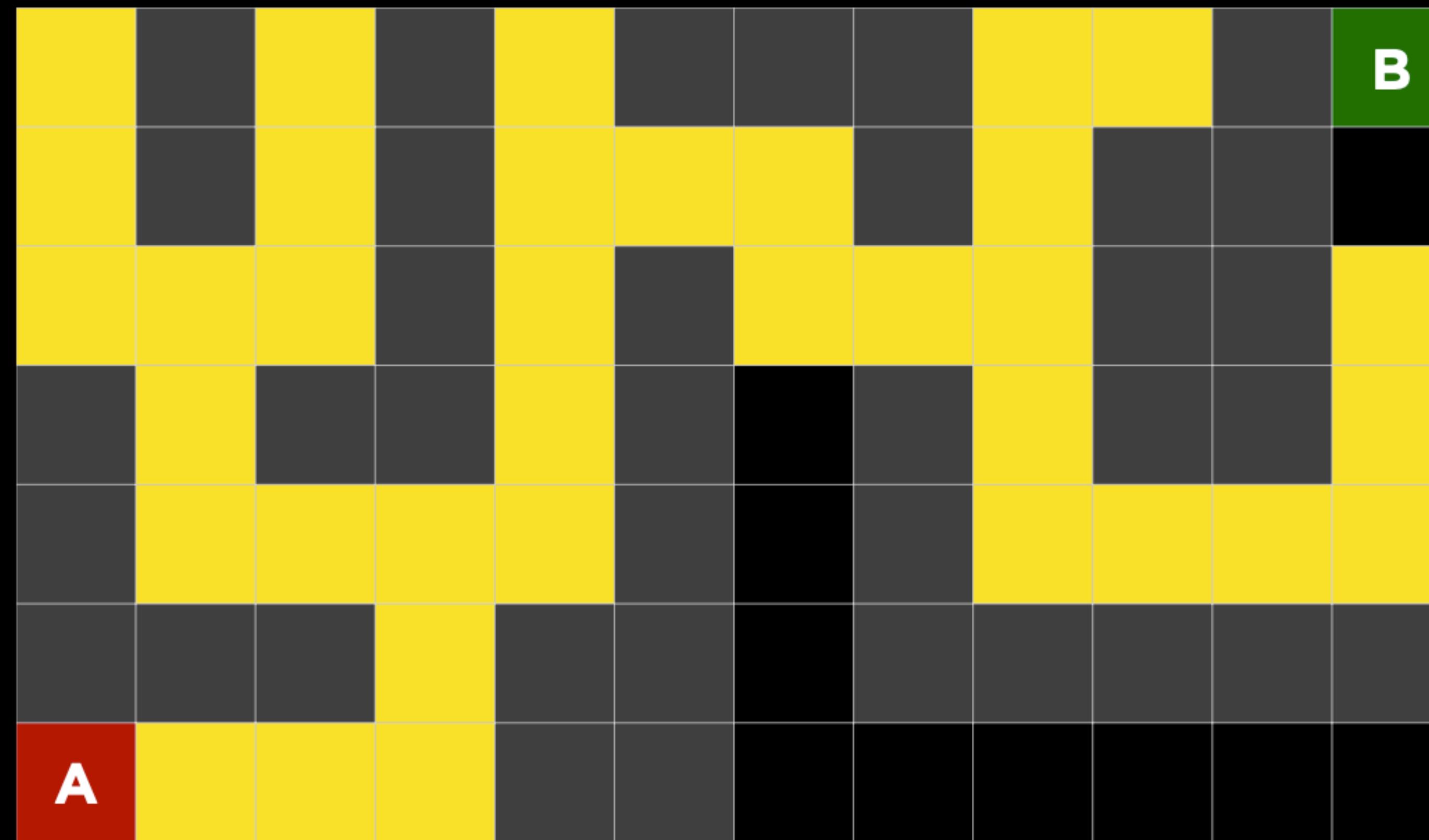
# Busca em Profundidade



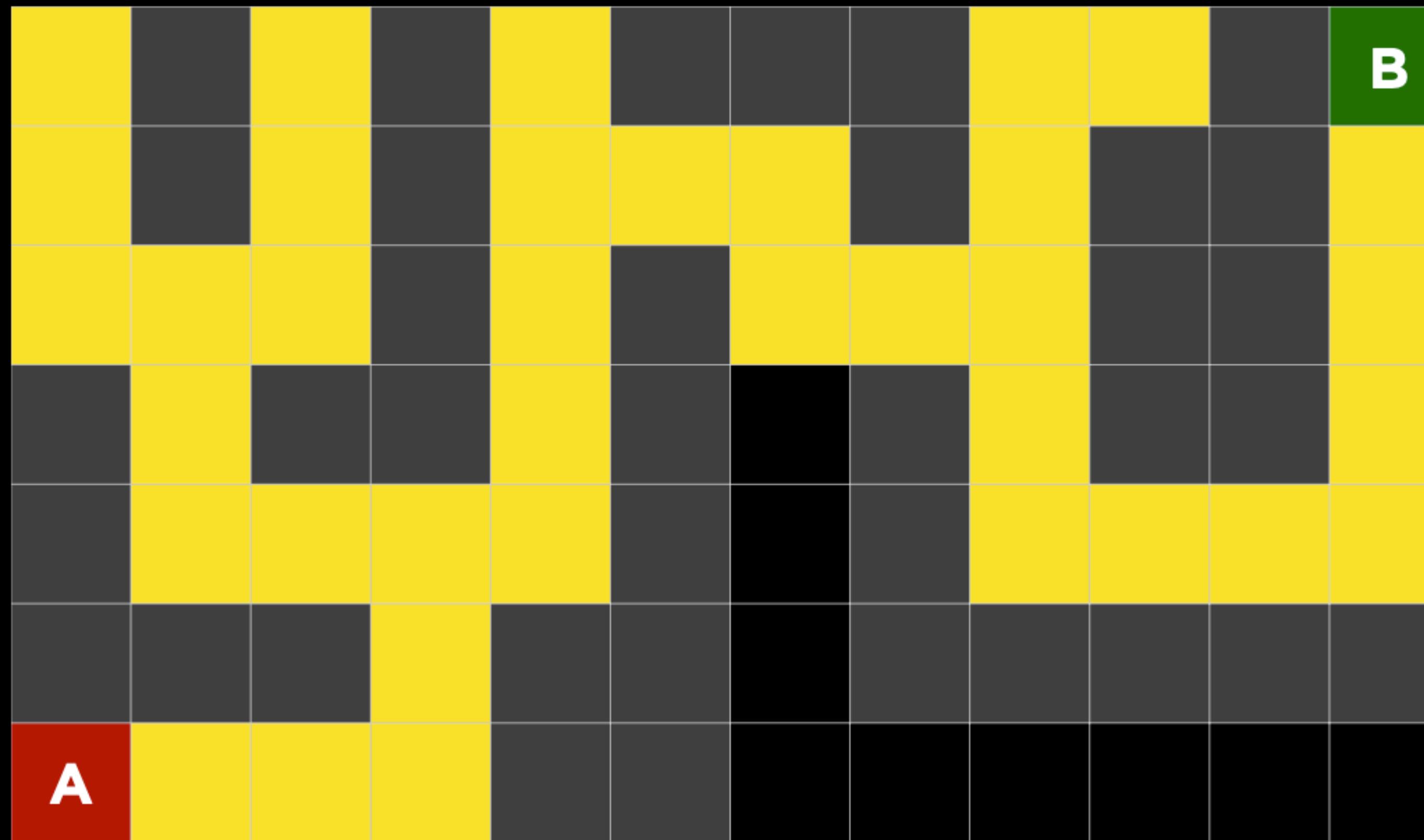
# Busca em Profundidade



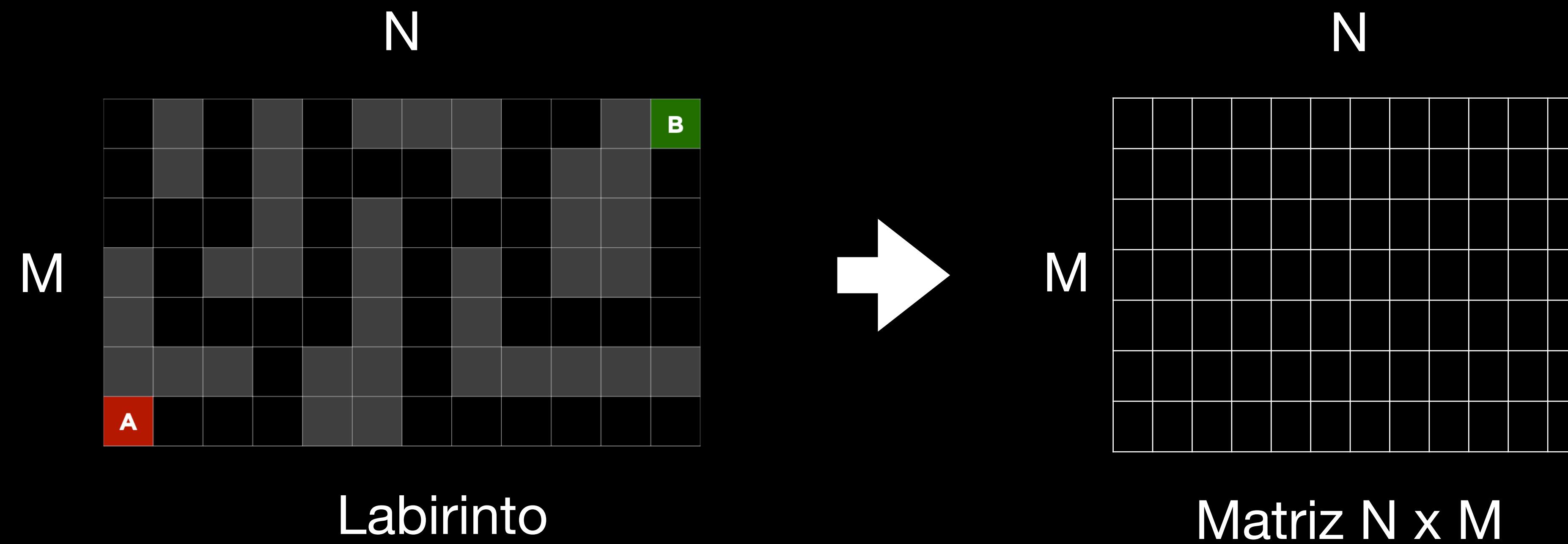
# Busca em Profundidade



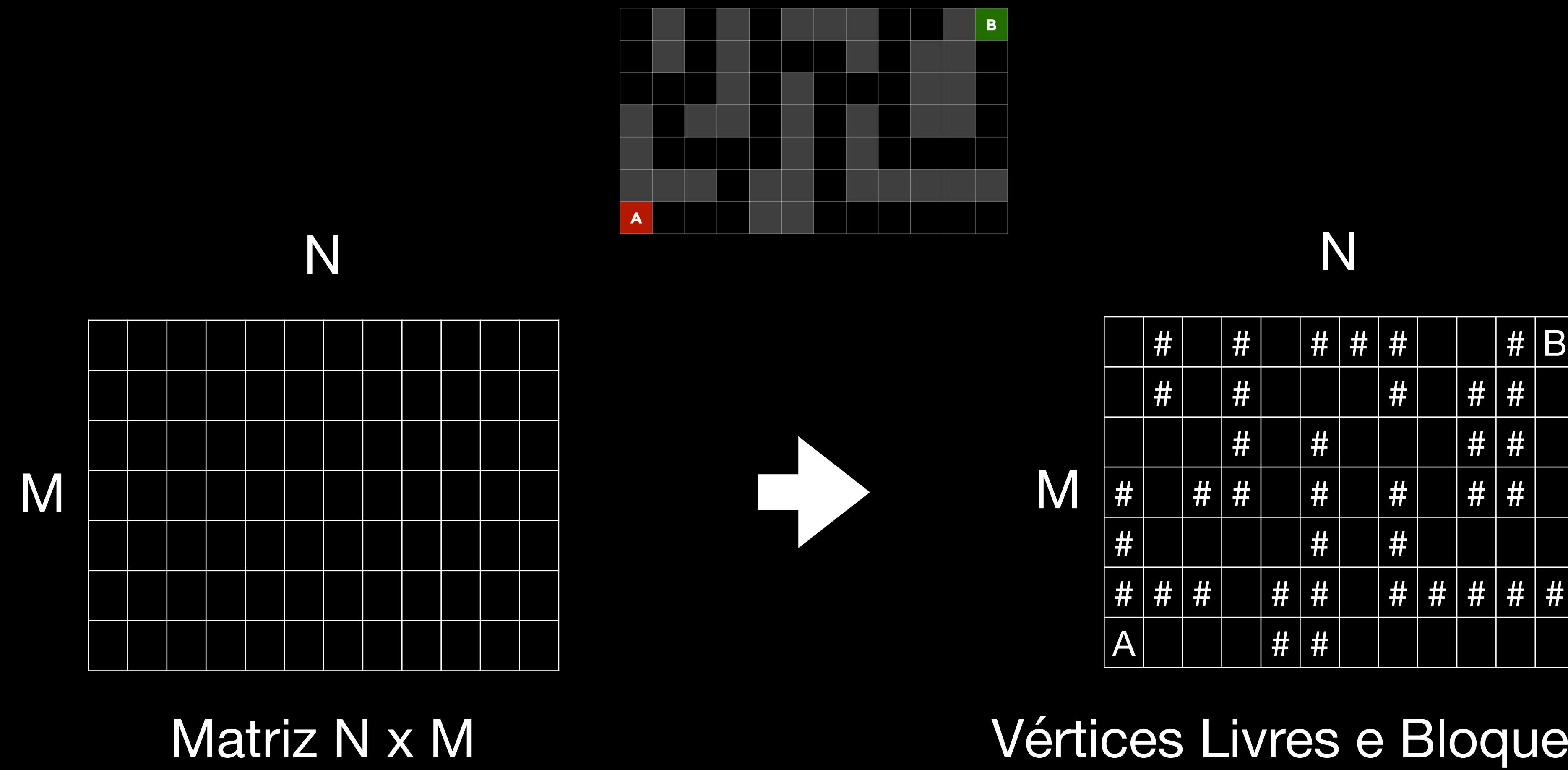
# Busca em Profundidade



# MODELAGEM DO PROBLEMA



# MODELAGEM DO PROBLEMA

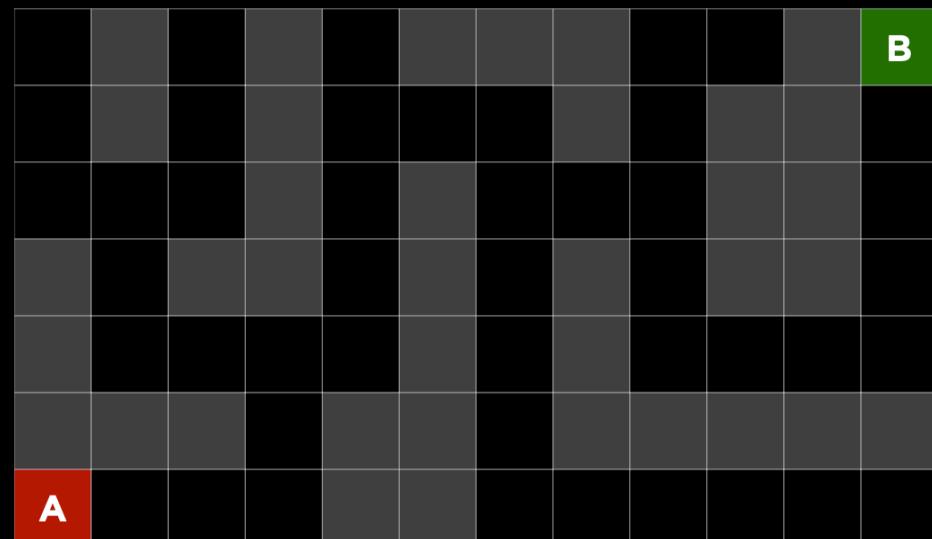


# MODELAGEM DO PROBLEMA

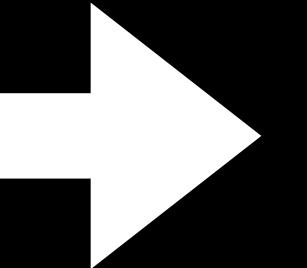
N

	#	#	#	#	#	#		#	B
	#	#		#	#	#			
		#	#			#	#		
	#	#	#	#	#	#	#	#	
	#		#	#					
	#	#	#	#	#	#	#	#	
M	A		#	#					

Vértices Livres e Bloqueados



Criar Listas Adjacências, Inicial e Objetivo



**Vértice Inicial:** [6,0] - A

**Vértice Buscado:** [0,11] - B

**Listas de Adjacências (vértices livres)**

- [0,0]=> [1,0]
- [1,0]=> [2,0]
- [2,0]=> [1,2]
- [2,1]=> [0,2], [2,2], [3,1]
- [2,2]=> [2,1], [1,2]
- ...
- ...

**Depois, rodar a Busca em Profundidade**

**A saída do Algoritmo será uma lista de vértices visitados**

# MODELAGEM DO PROBLEMA

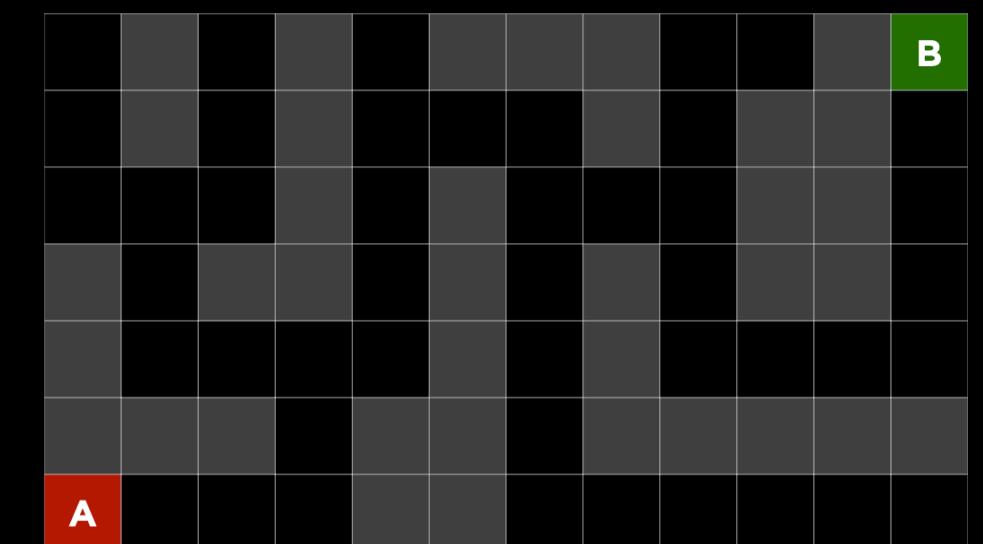
Vértice Inicial: [6,0] - A

Vértice Buscado: [0,11] - B

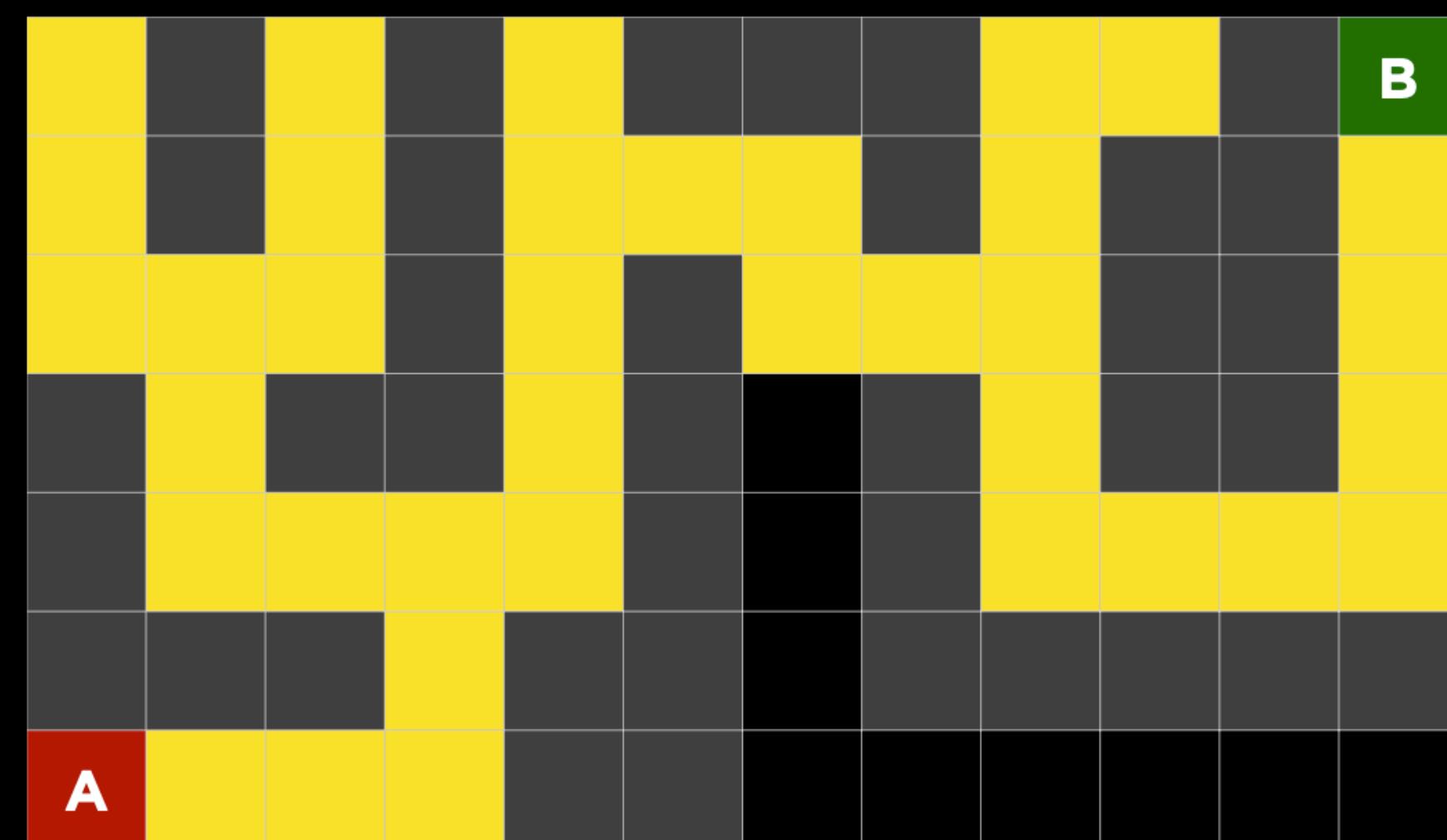
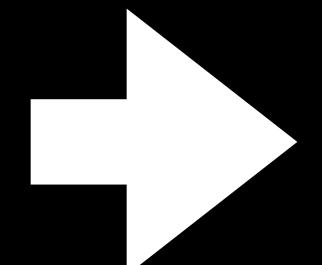
Listas de Adjacências (vértices livres)

- $[0,0] \Rightarrow [1,0]$
- $[1,0] \Rightarrow [2,0]$
- $[2,0] \Rightarrow [1,2]$
- $[2,1] \Rightarrow [0,2], [2,2], [3,1]$
- $[2,2] \Rightarrow [2,1], [1,2]$
- ...
- ...

Depois, rodar a Busca em Profundidade

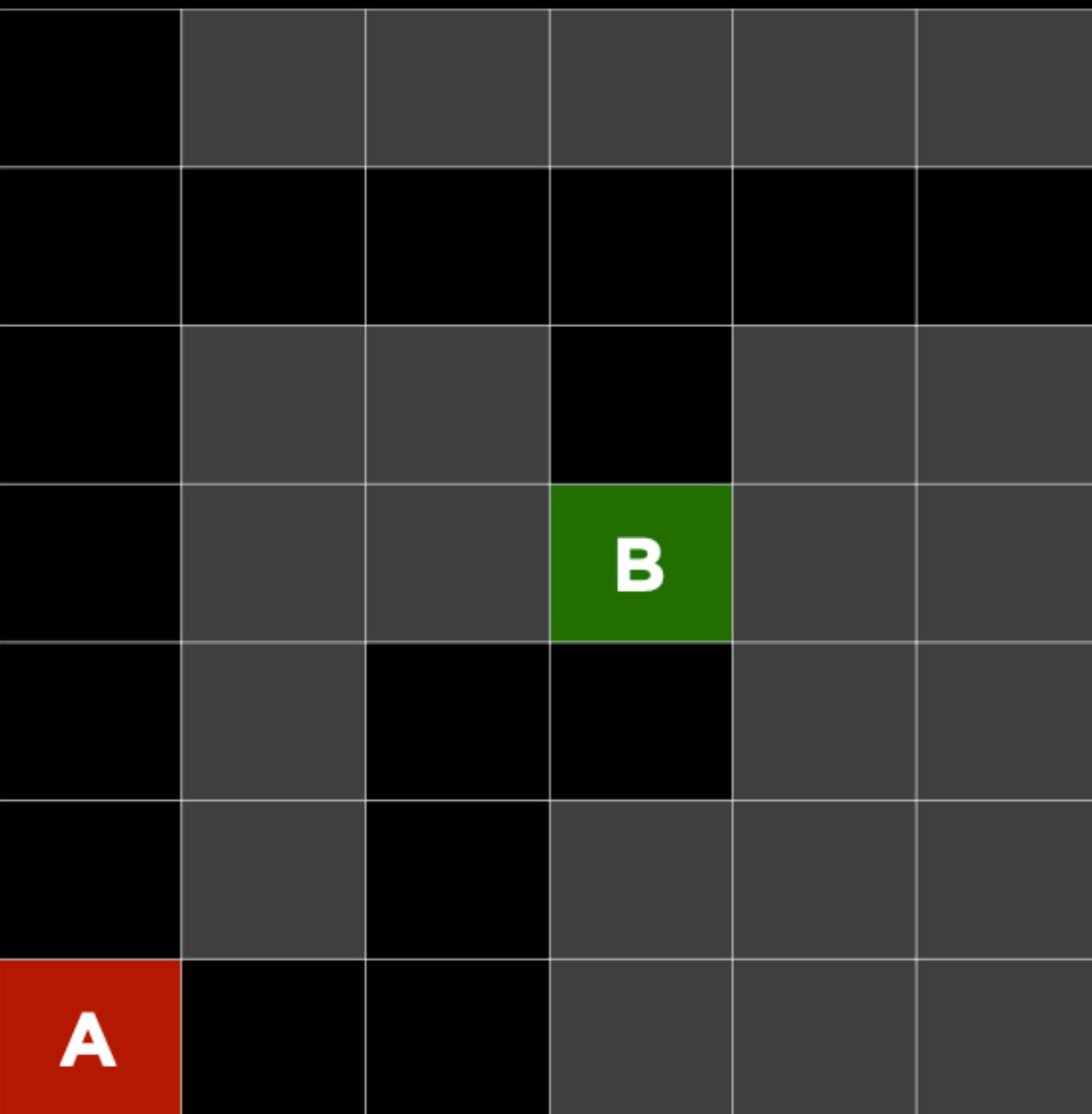


Colorir os Vértices Visitados

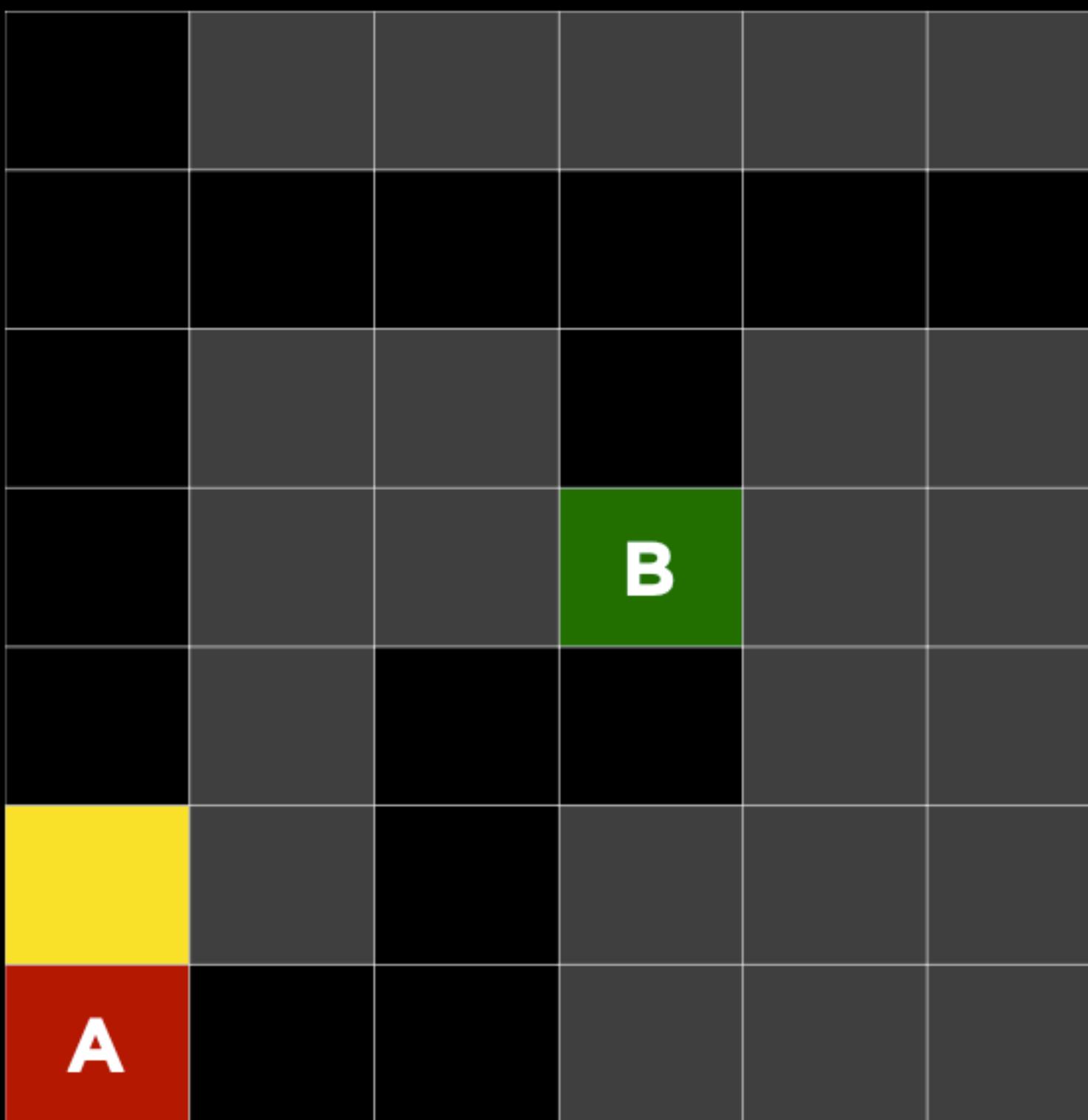


# Busca em Largura

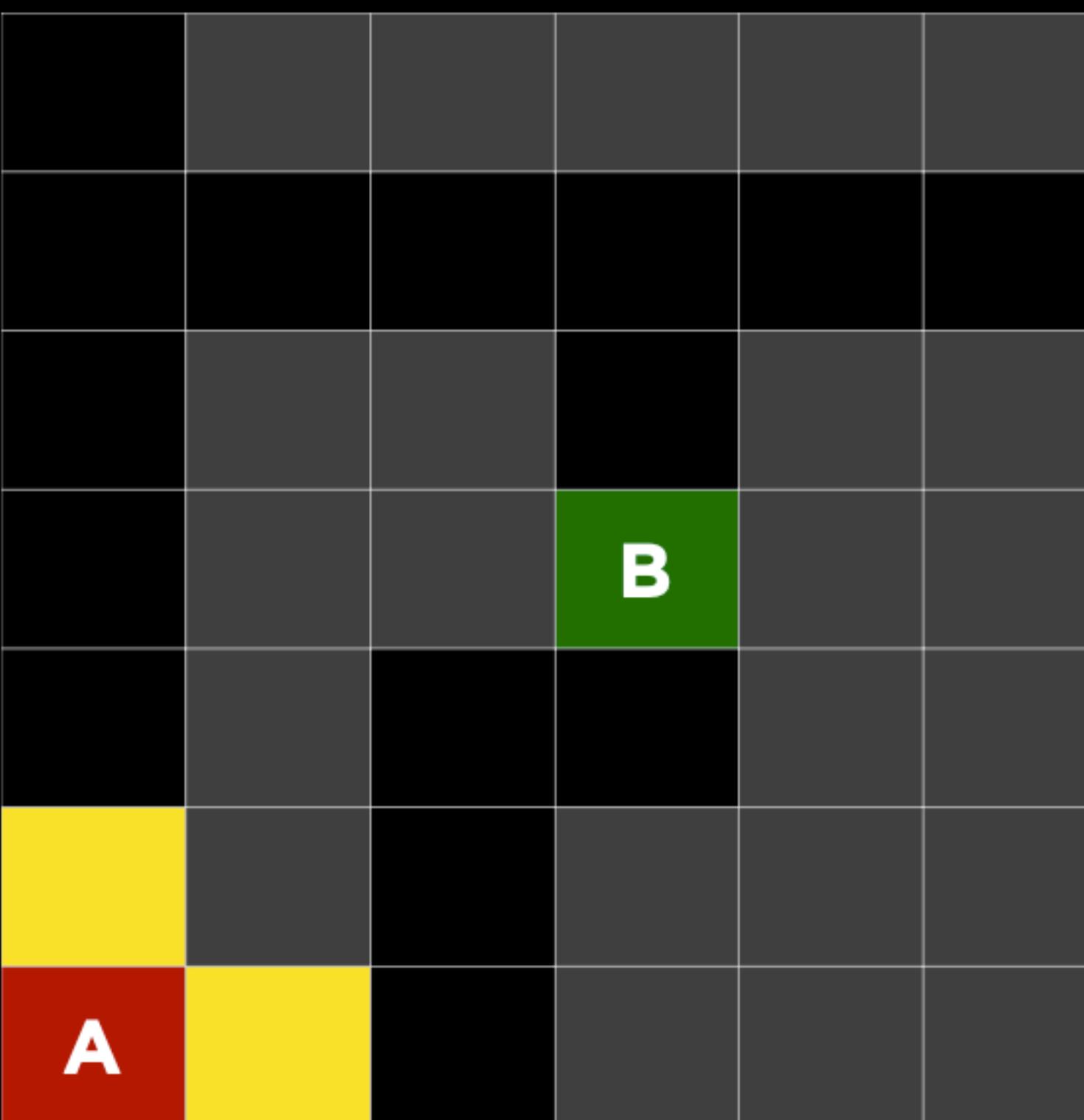
Encontre um caminho entre A e B



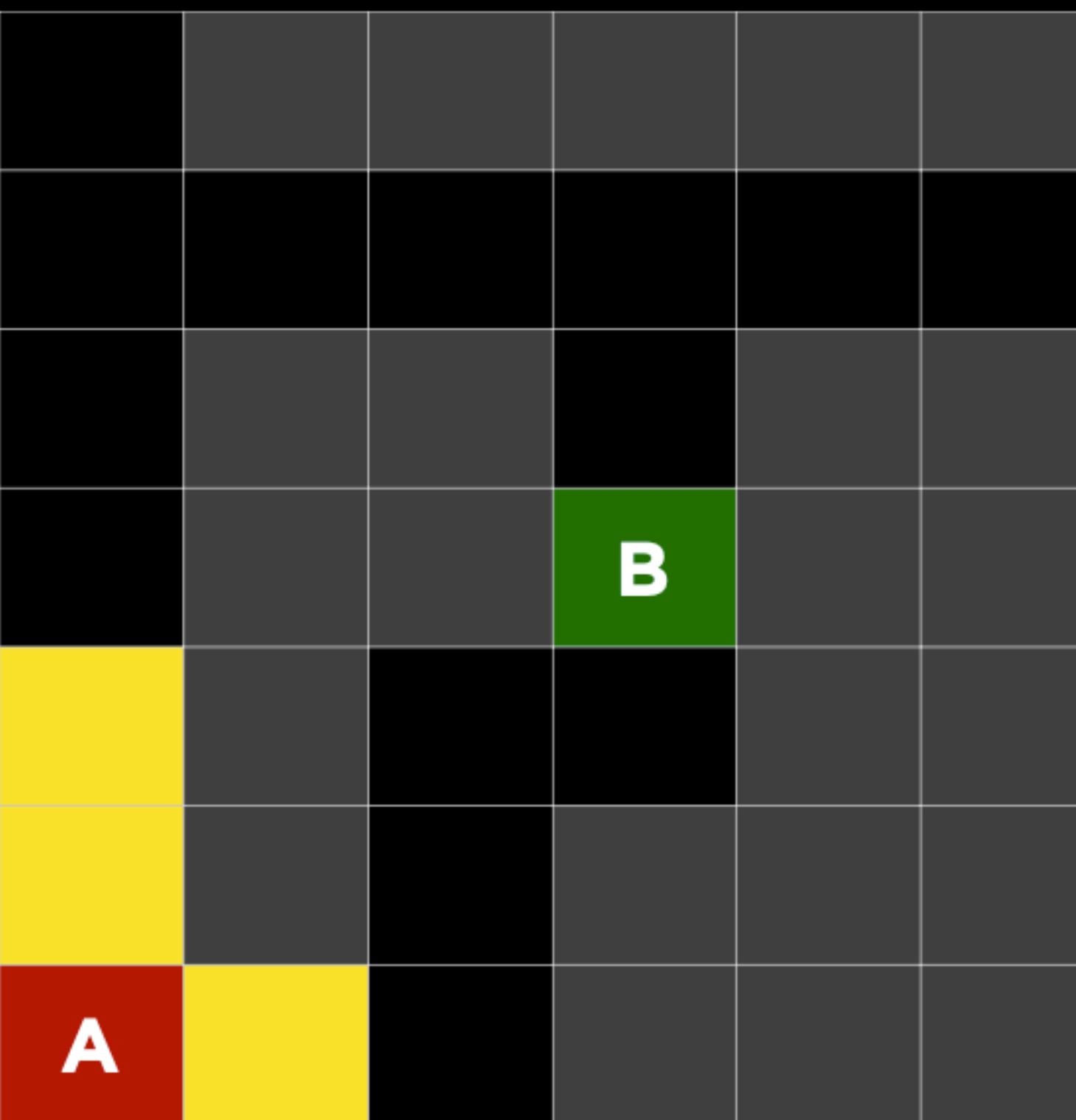
# Busca em Largura



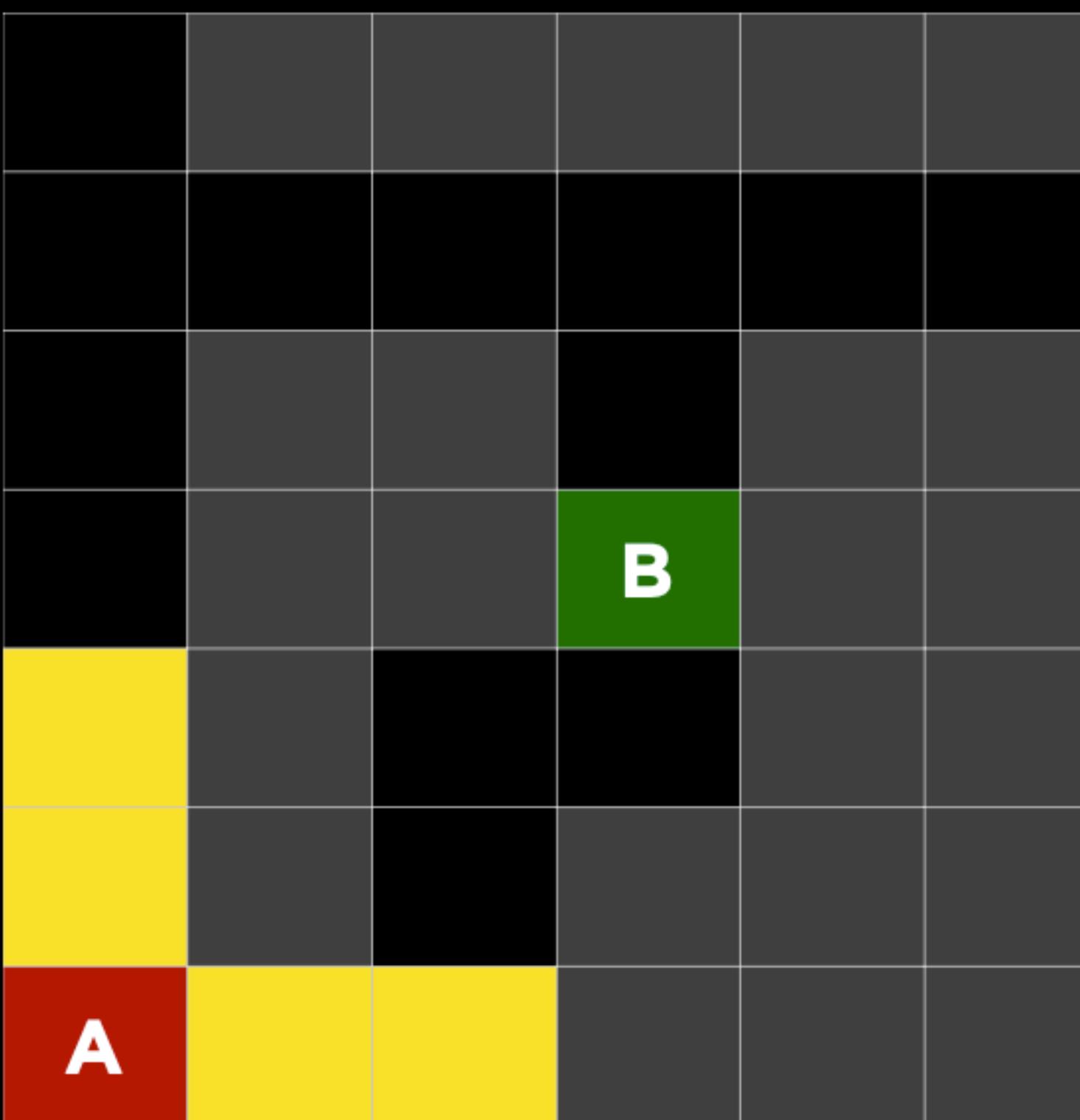
# Busca em Largura



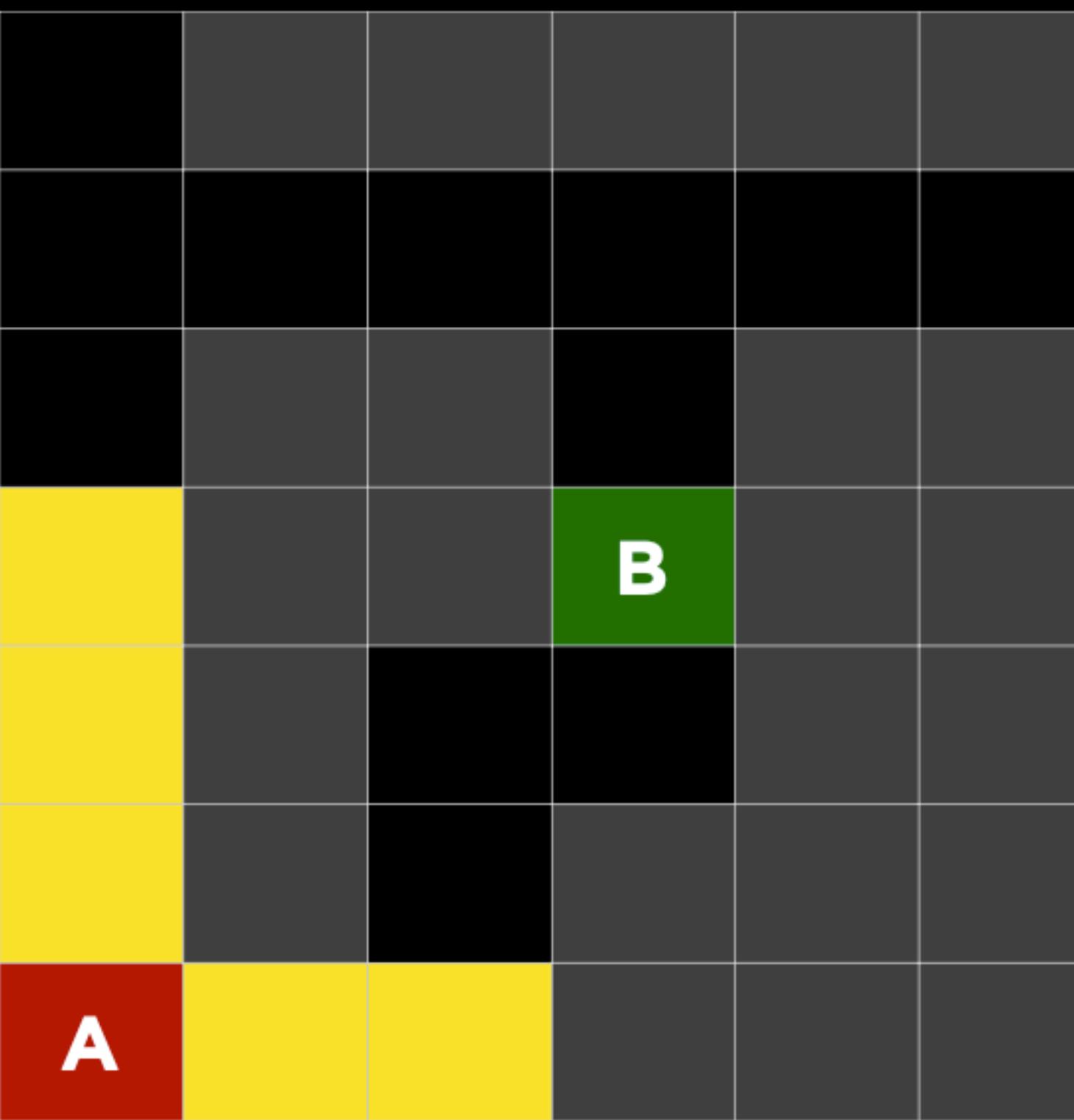
# Busca em Largura



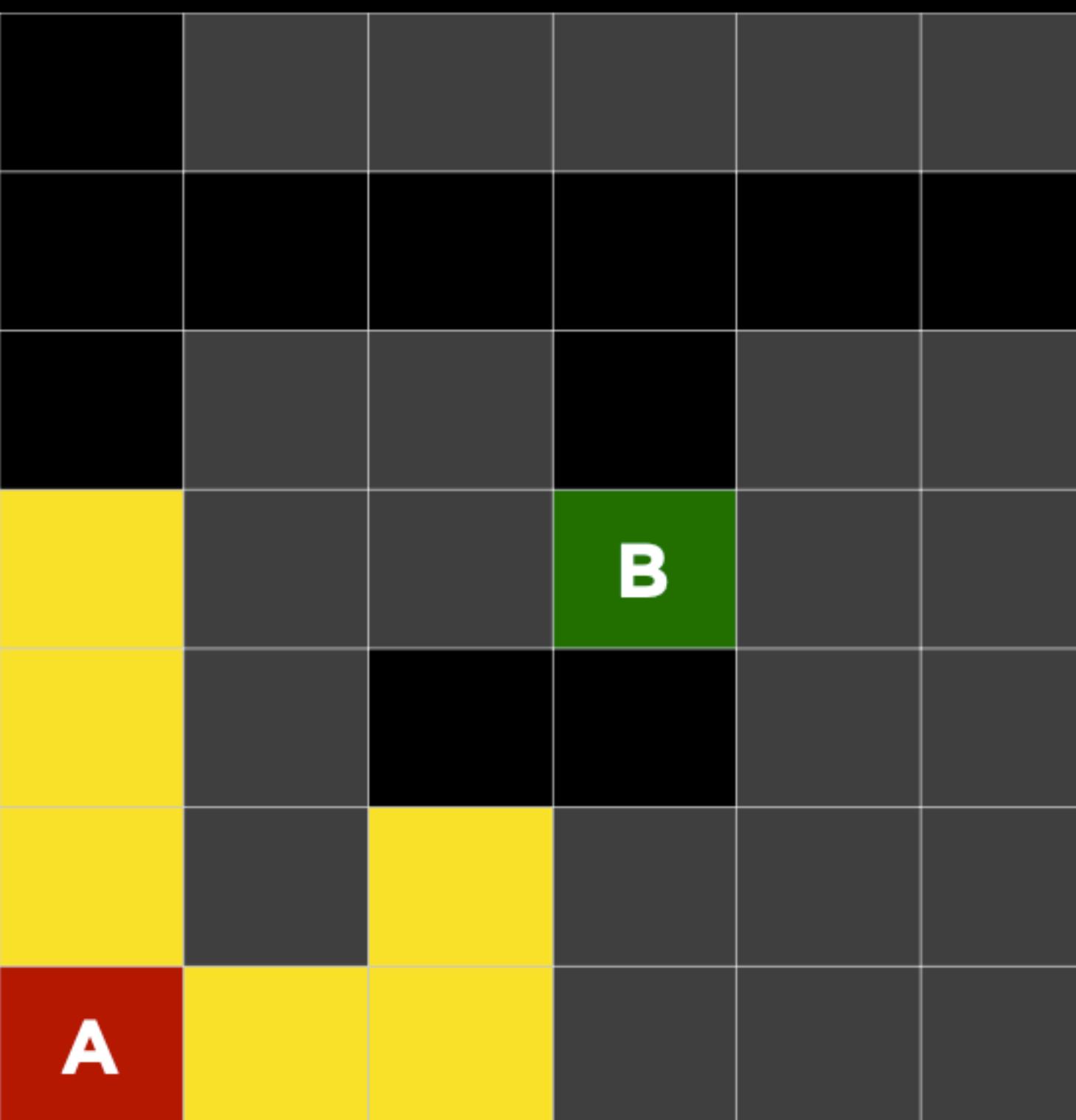
# Busca em Largura



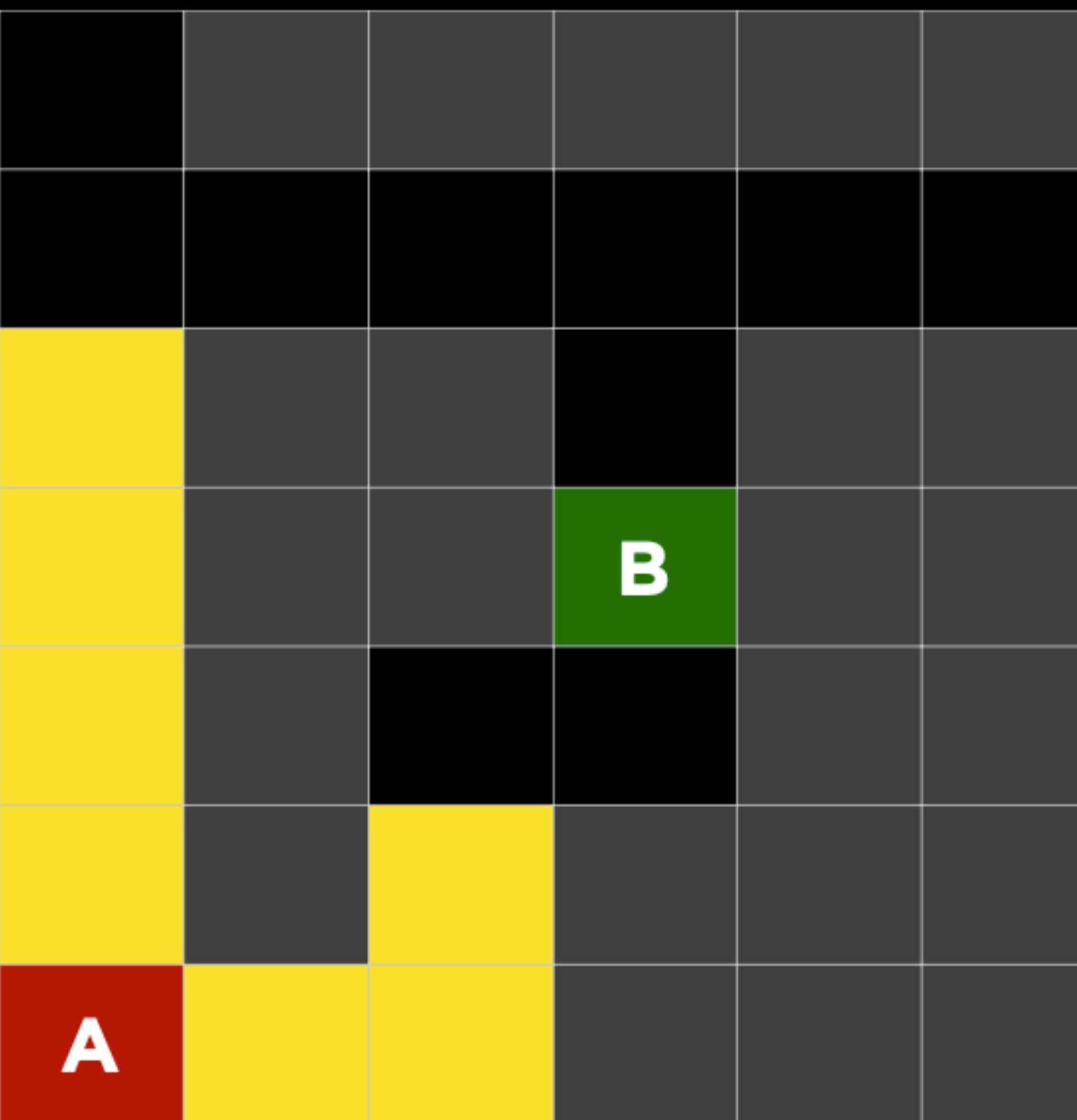
# Busca em Largura



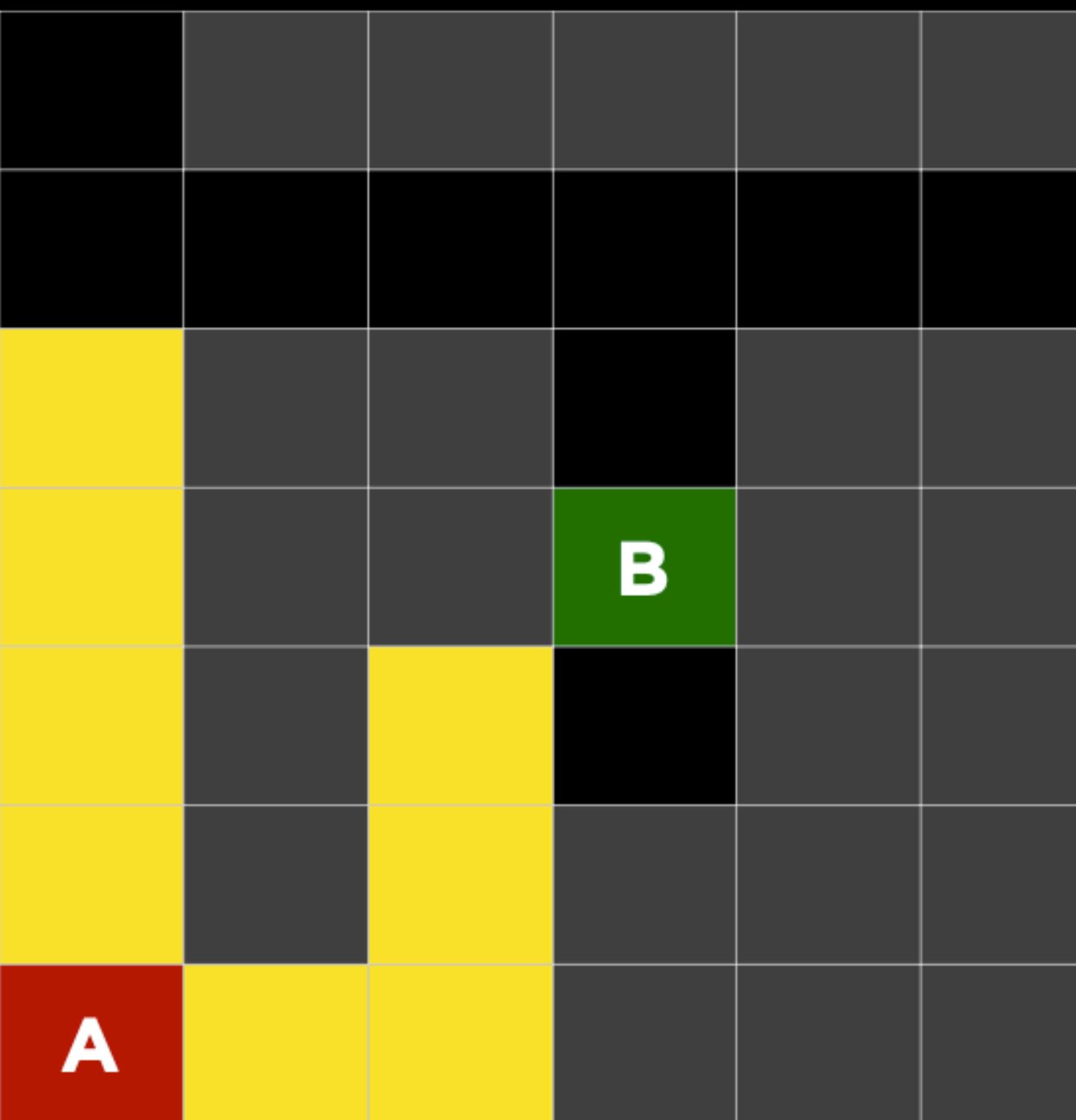
# Busca em Largura



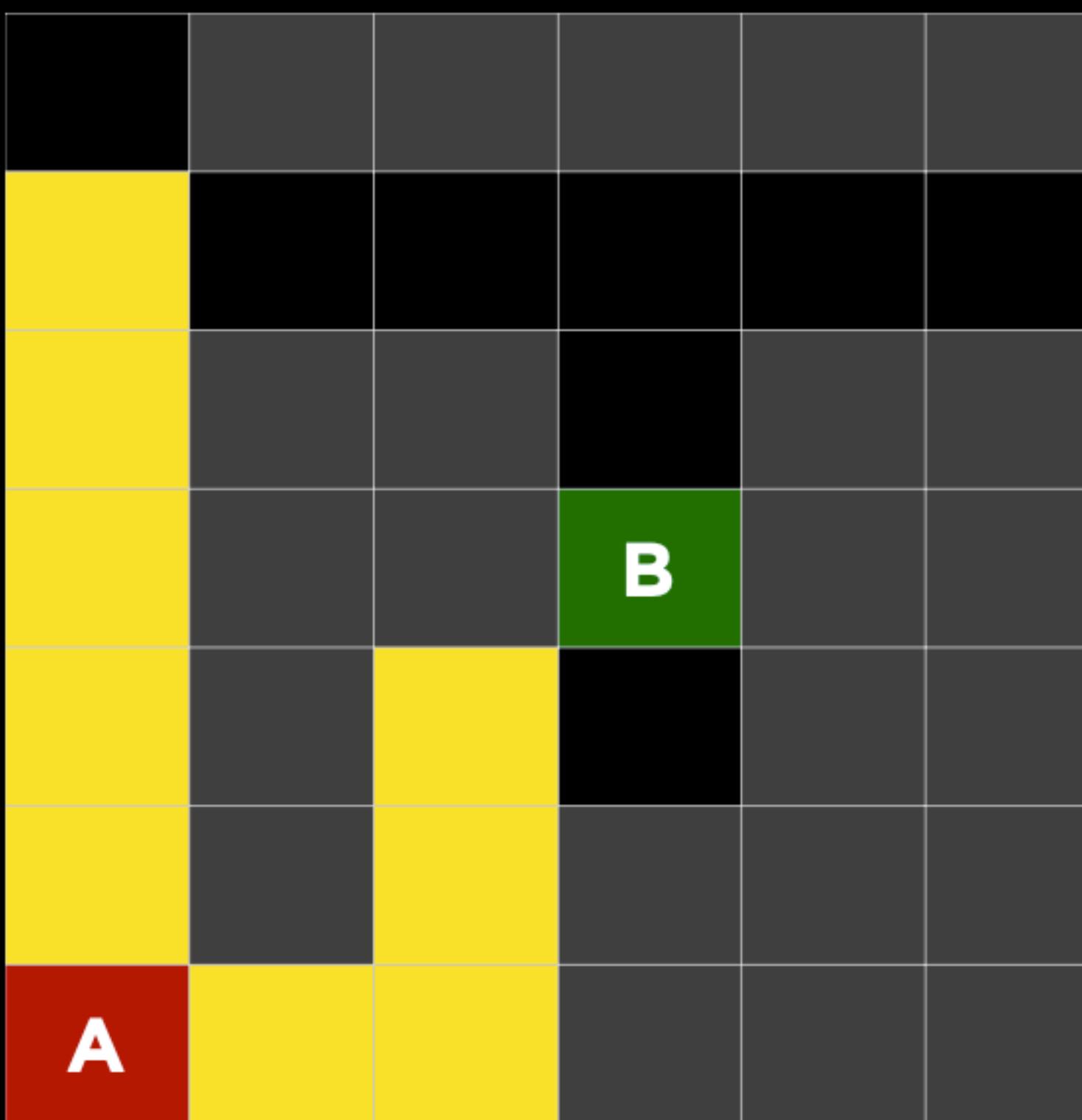
# Busca em Largura



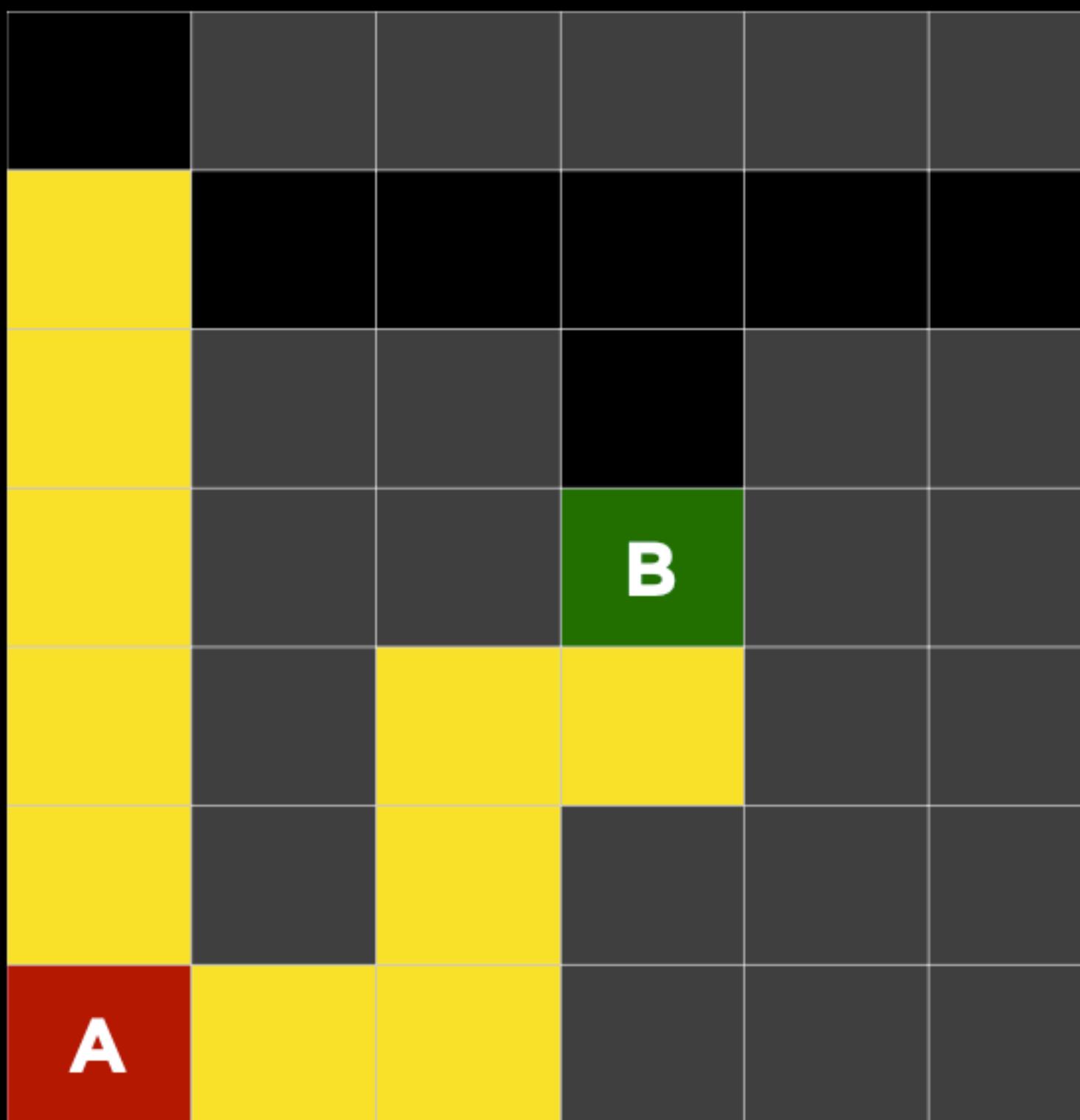
# Busca em Largura



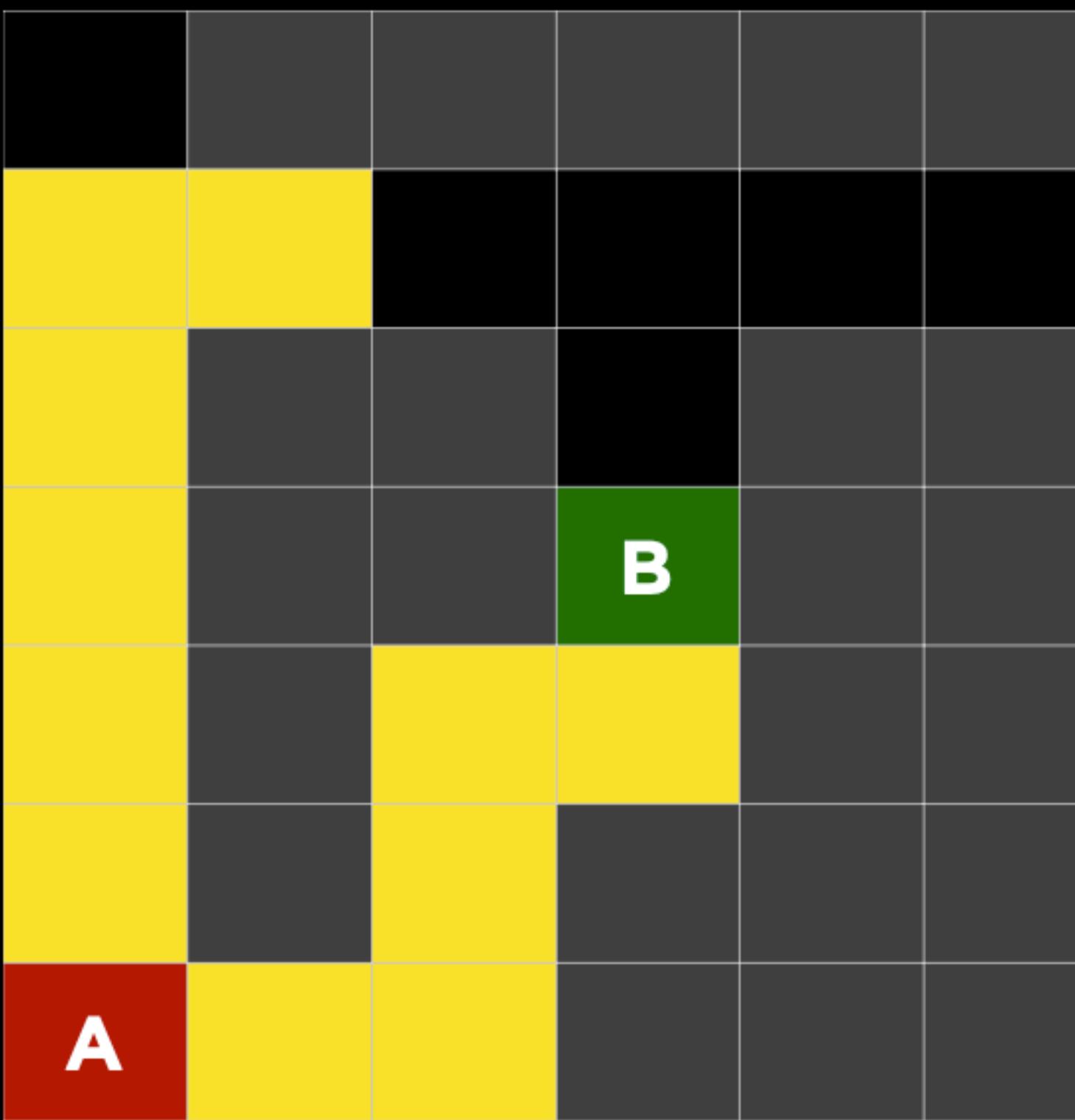
# Busca em Largura



# Busca em Largura

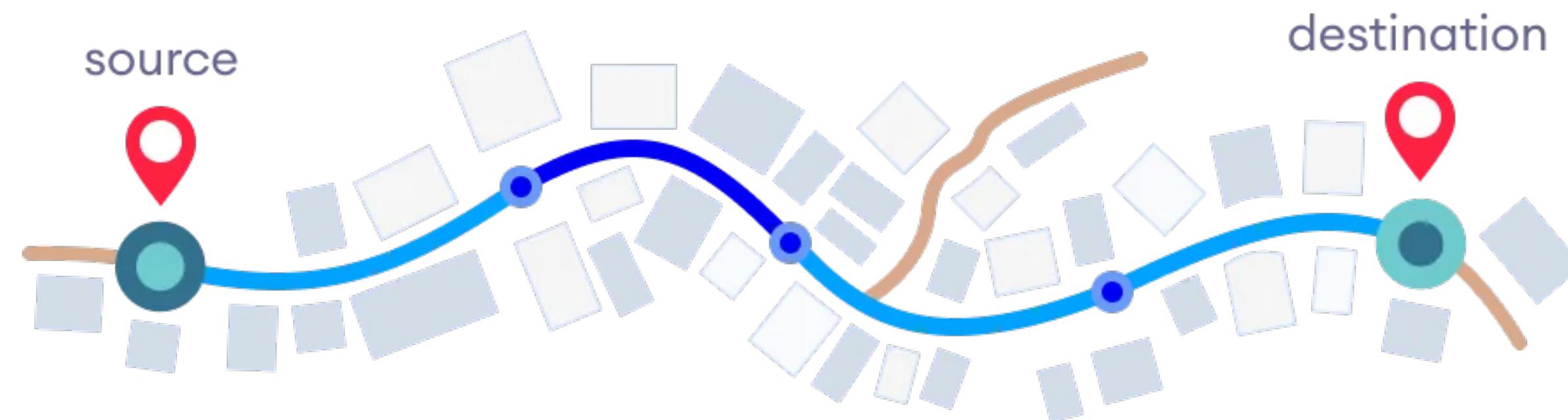


# Busca em Largura



# Busca Menor Caminho

# Algoritmo de Dijkstra

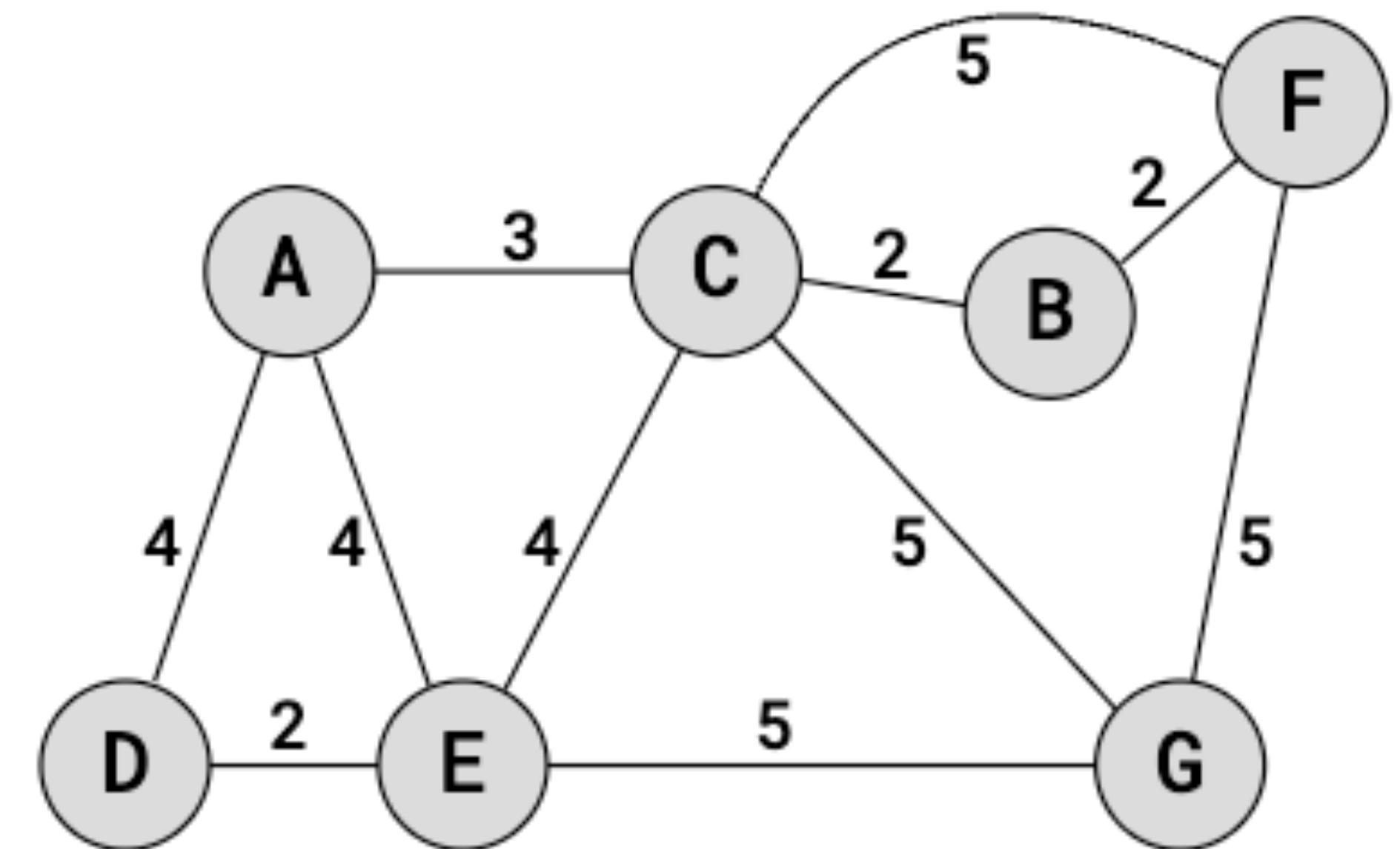


- O algoritmo de Dijkstra é frequentemente considerado o algoritmo mais direto para resolver o problema do caminho mais curto.
- Desenvolvido pelo holandês Edsger Dijkstra em 1959.
- O que o algoritmo faz? Escolhido um vértice como raiz da busca, este algoritmo calcula o custo mínimo deste vértice para todos os demais vértices do grafo
- Limitações: Não funciona para grafos com ciclos com pesos negativos

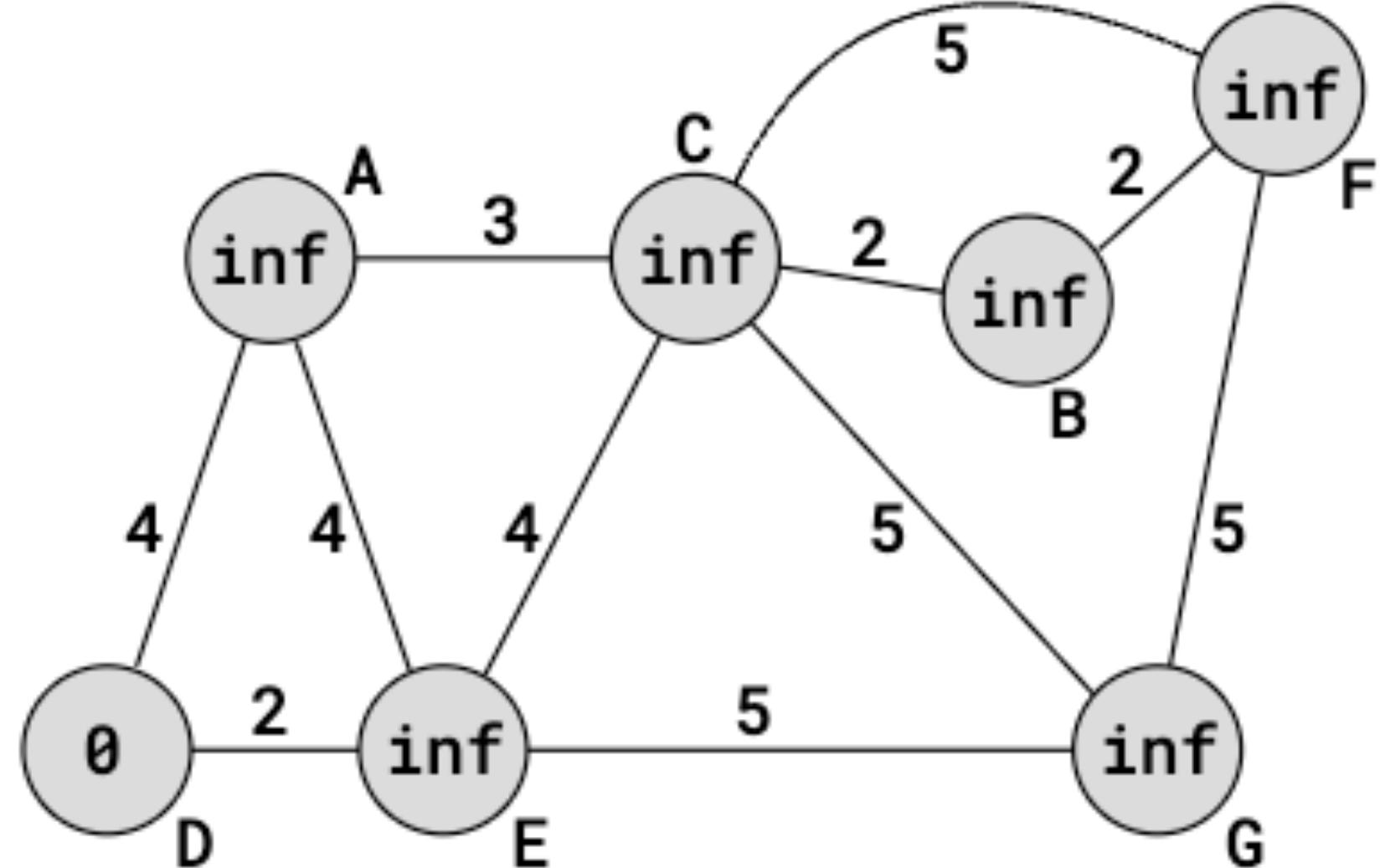
# Algoritmo de Dijkstra

Como ele funciona:

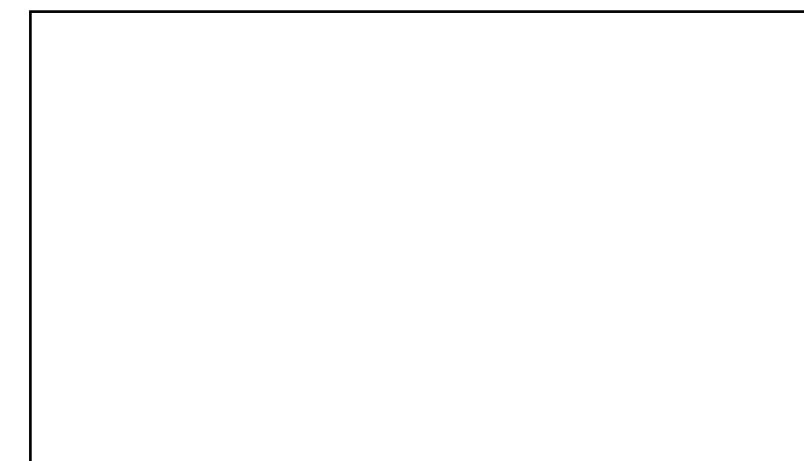
1. Defina distâncias iniciais para todos os vértices: 0 para o vértice de origem e infinito para todos os outros.
2. Escolha o vértice não visitado com a distância mais curta desde o início para ser o vértice atual. Portanto, o algoritmo sempre começará com a origem como vértice atual.
3. Para cada um dos vértices vizinhos não visitados do vértice atual, calcule a distância da origem e atualize a distância se a nova distância calculada for menor.
4. Agora terminamos com o vértice atual, então o marcamos como visitado. Um vértice visitado não é verificado novamente.
5. Volte ao passo 2 para escolher um novo vértice atual e continue repetindo essas etapas até que todos os vértices sejam visitados.
6. No final, ficamos com o caminho mais curto do vértice de origem para todos os outros vértices do gráfico.



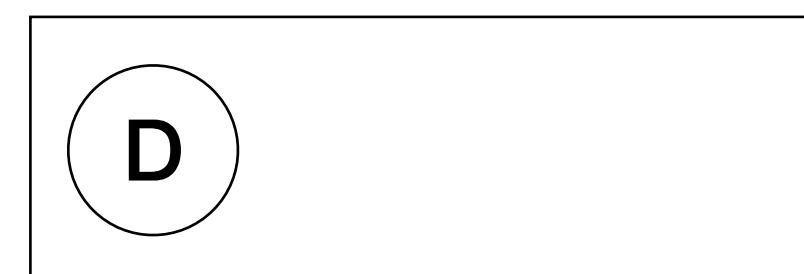
# Algoritmo de Dijkstra



VISITADOS

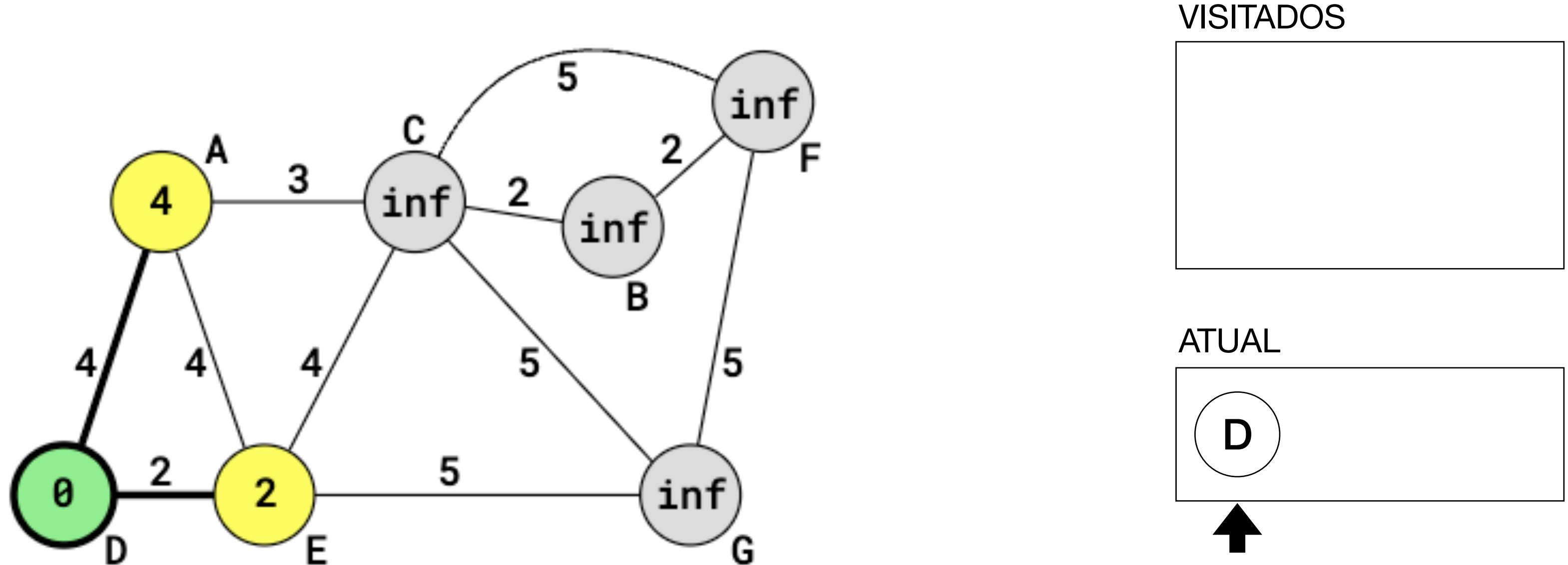


ATUAL



A imagem mostra as distâncias infinitas iniciais para outros vértices a partir do vértice inicial D. O valor da distância para o vértice D é 0 porque esse é o ponto inicial.

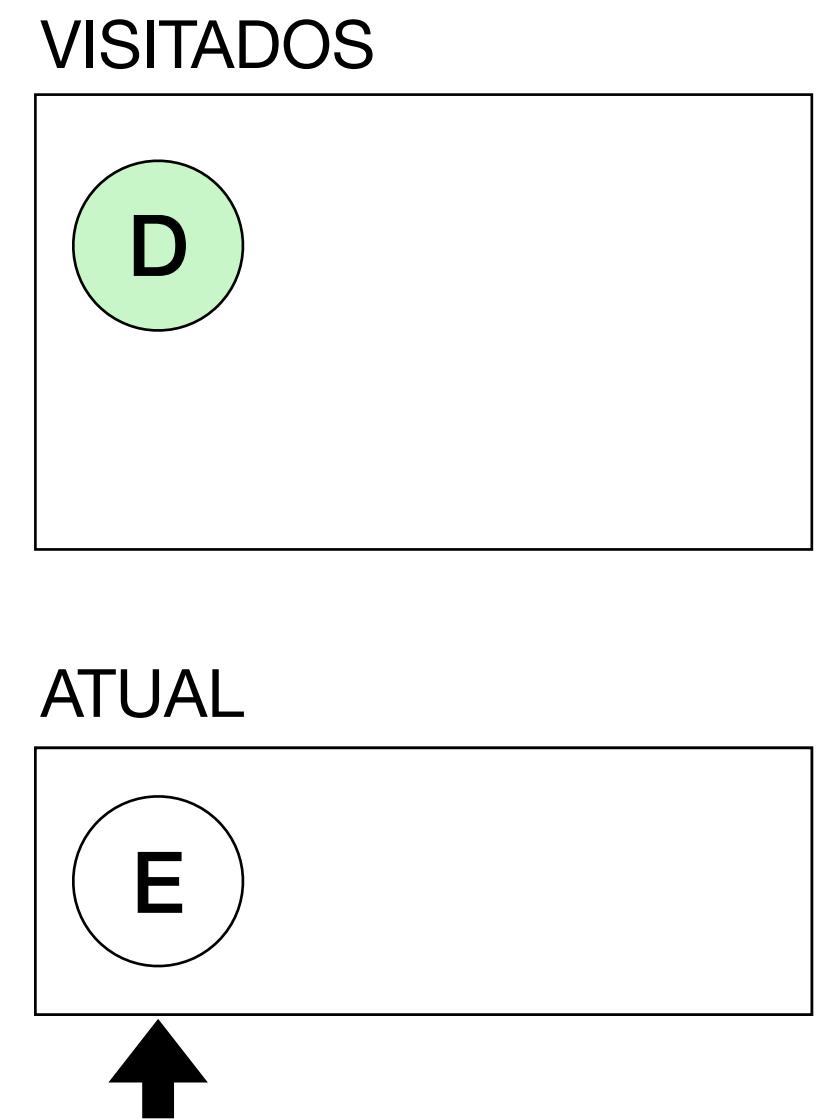
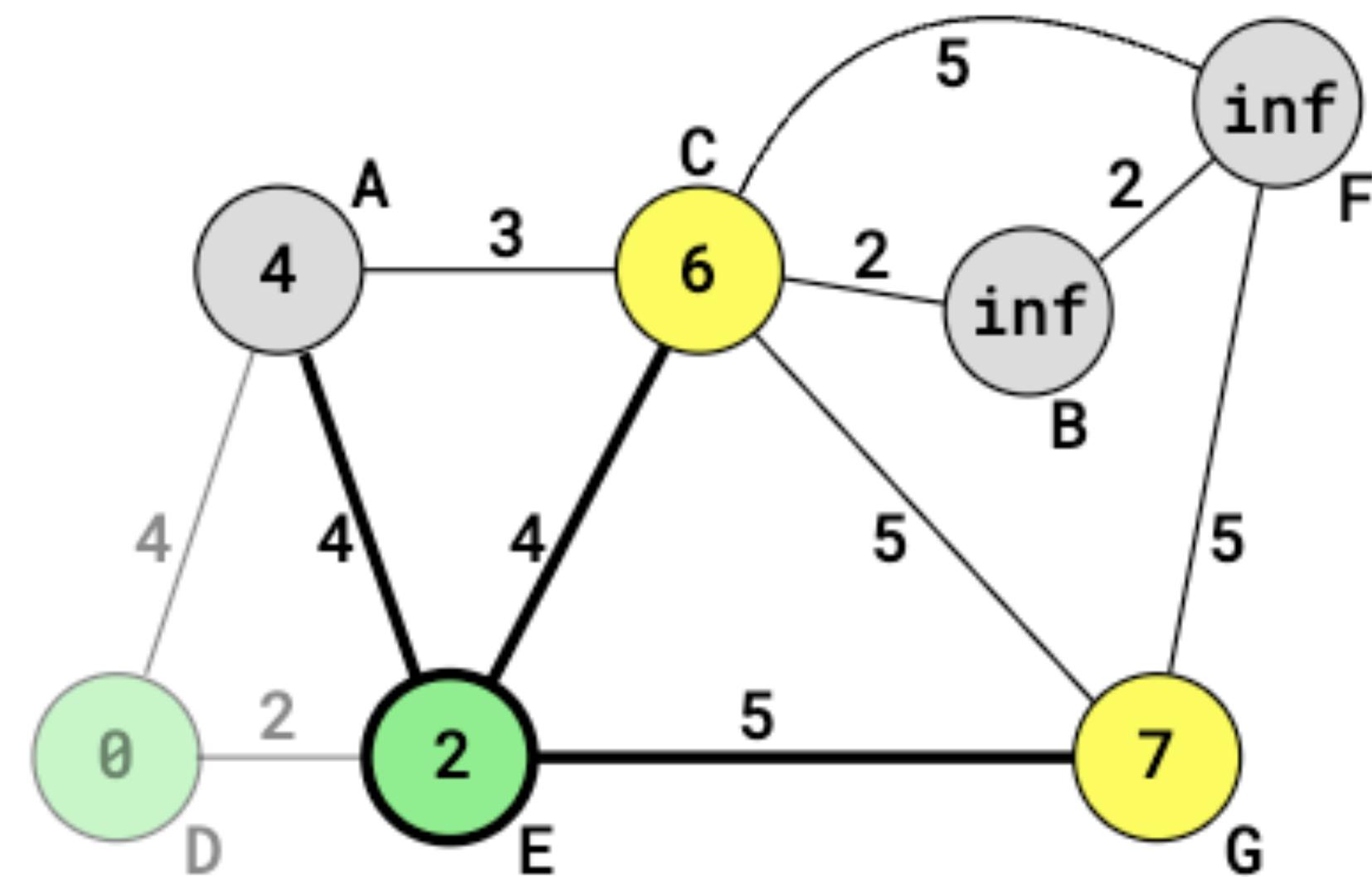
# Algoritmo de Dijkstra



O algoritmo de Dijkstra então define o vértice D como o vértice atual e observa a distância até os vértices adjacentes. Como a distância inicial aos vértices A e E é infinita, a nova distância a estes é atualizada com os pesos das arestas. Portanto, o vértice A altera a distância de inf para 4 e o vértice E altera a distância para 2. Atualizar os valores de distância dessa forma é chamado de '*relaxar*'.

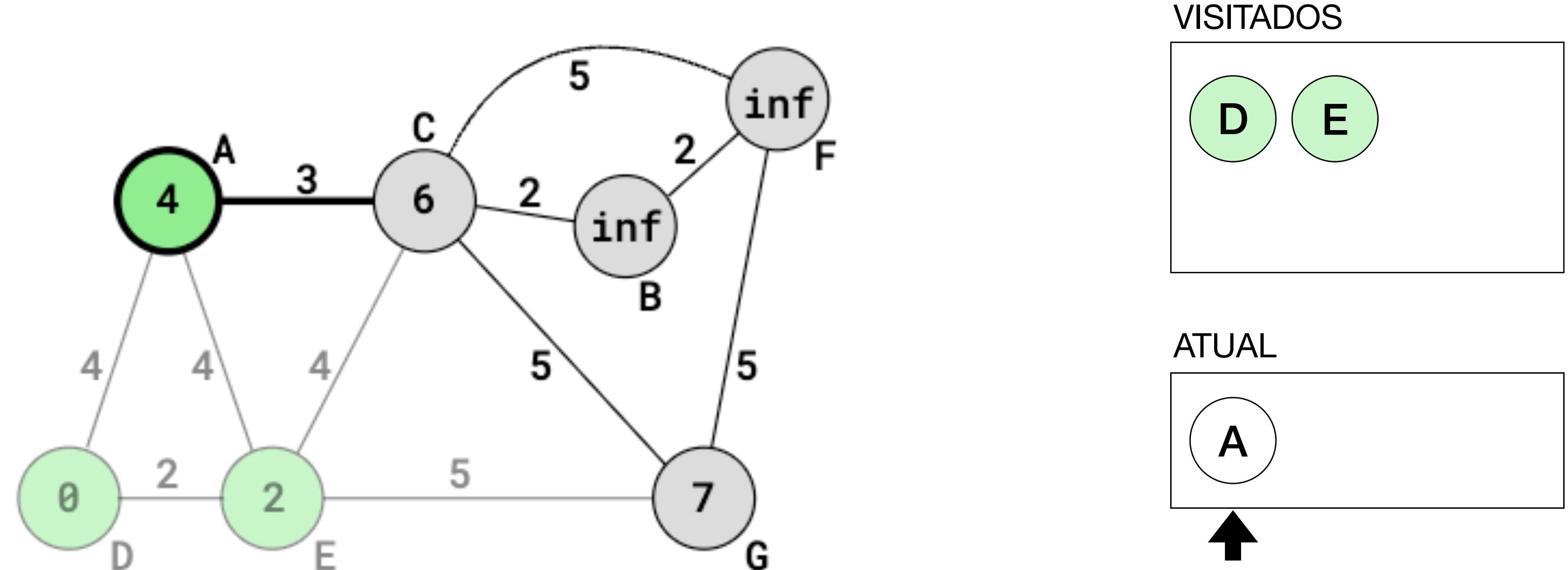
# Algoritmo de Dijkstra

Após relaxar os vértices A e E, o vértice D é considerado visitado e não será visitado novamente. O próximo vértice a ser escolhido como vértice atual deve ser o vértice com menor distância ao vértice de origem (vértice D), dentre os vértices não visitados anteriormente. O vértice E é, portanto, escolhido como o vértice atual após o vértice D.



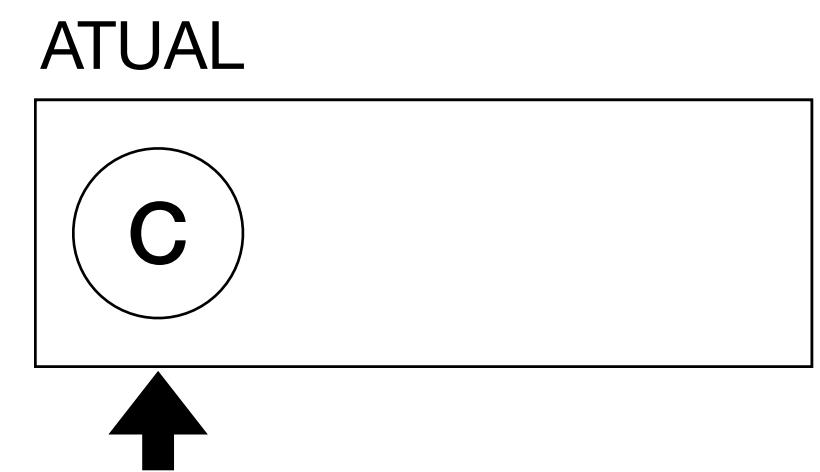
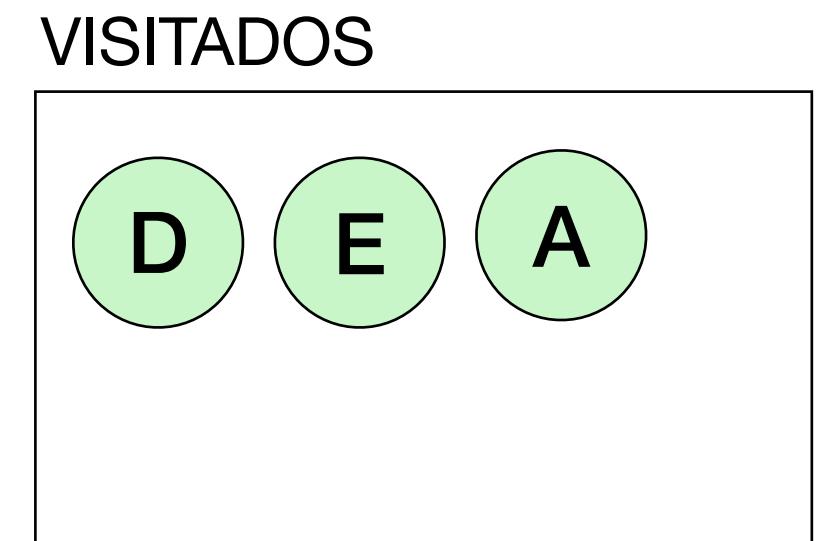
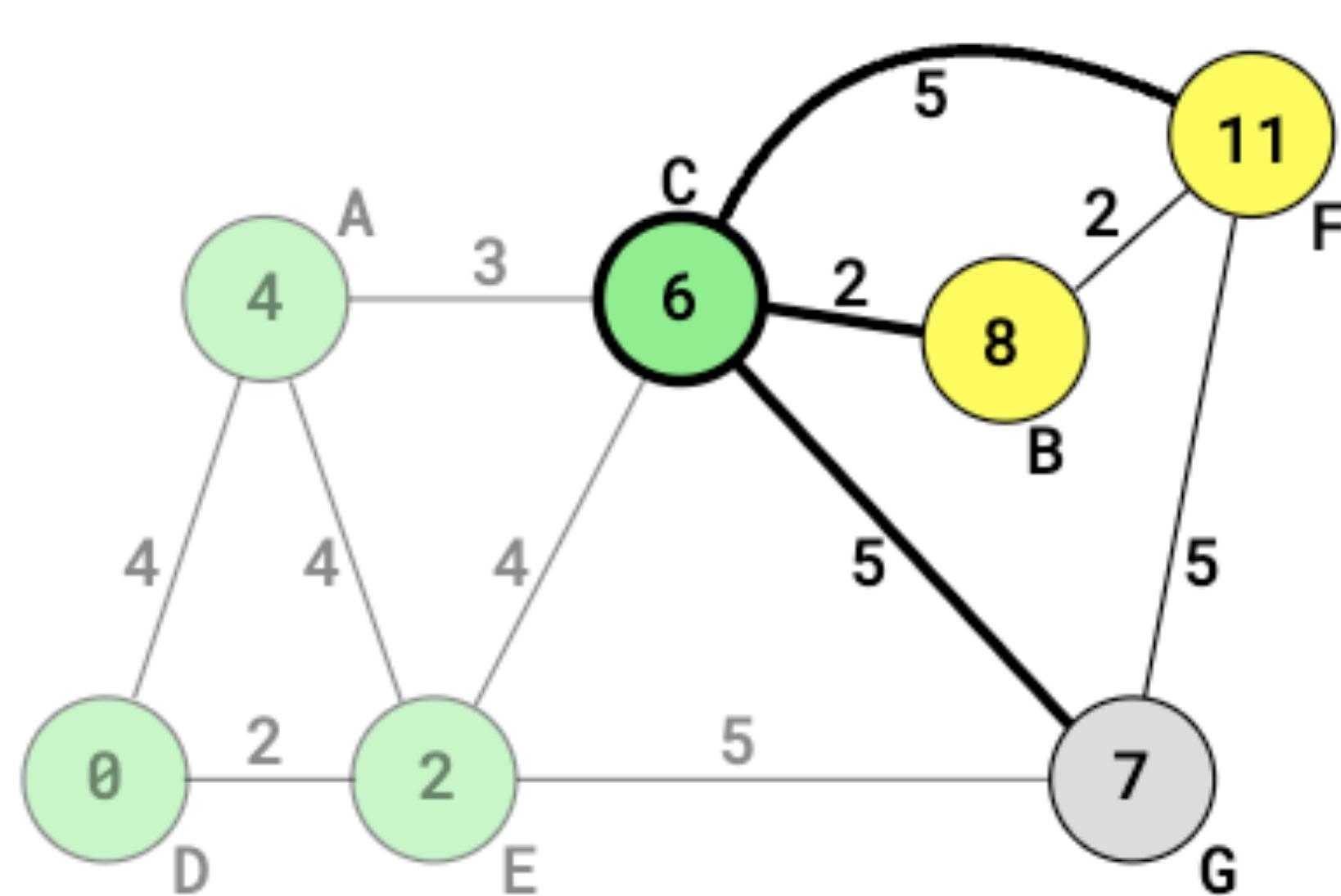
A distância do vértice E a todos os vértices adjacentes e não visitados anteriormente deve agora ser calculada e atualizada se necessário. A distância calculada de D ao vértice A, via E, é  $2+4=6$ . Mas a distância atual até o vértice A já é 4, que é menor, então a distância até o vértice A não é atualizada. A distância até o vértice C é calculada como  $2+4=6$ , que é menor que infinito, portanto, a distância até o vértice C é atualizada. Da mesma forma, a distância até o nó G é calculada e atualizada para  $2+5=7$ . O próximo vértice a ser visitado é o vértice A porque possui a menor distância de D de todos os vértices não visitados.

# Algoritmo de Dijkstra



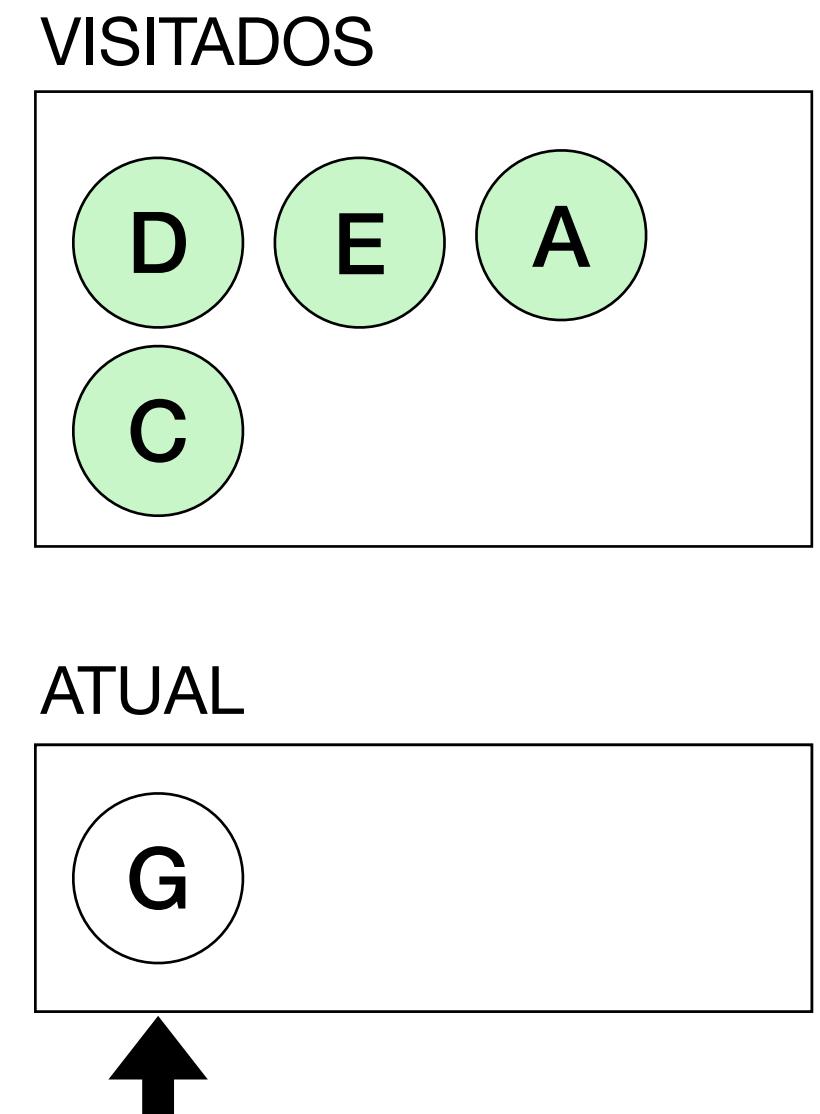
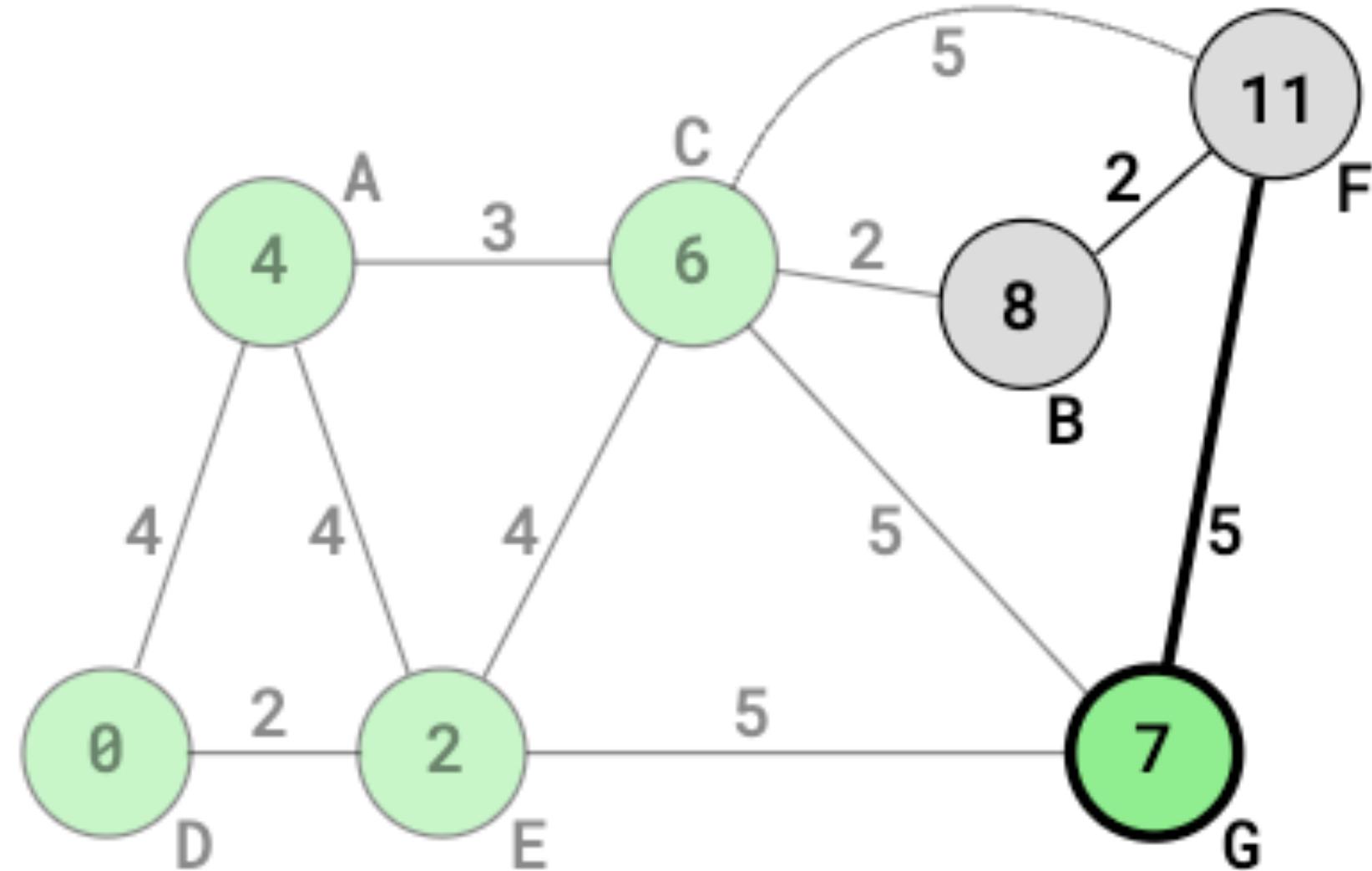
A distância calculada até o vértice C, via A, é  $4+3=7$ , que é maior que a distância já definida até o vértice C, portanto a distância até o vértice C não é atualizada. O vértice A agora está marcado como visitado e o próximo vértice atual é o vértice C porque tem a menor distância do vértice D entre os vértices não visitados restantes.

# Algoritmo de Dijkstra



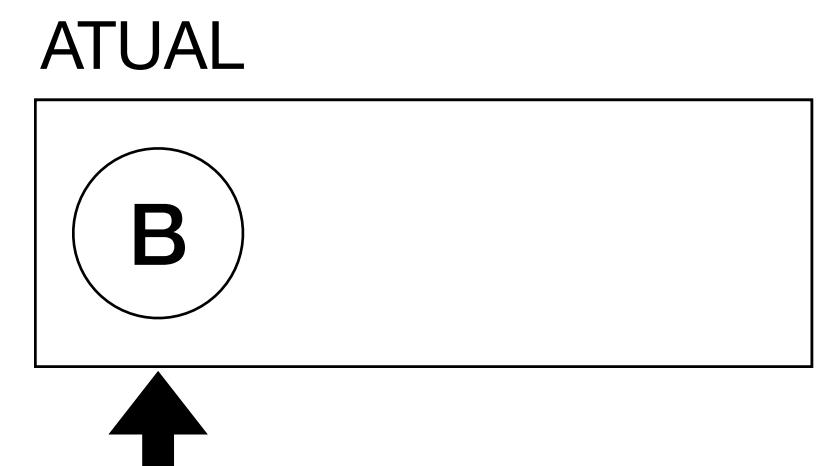
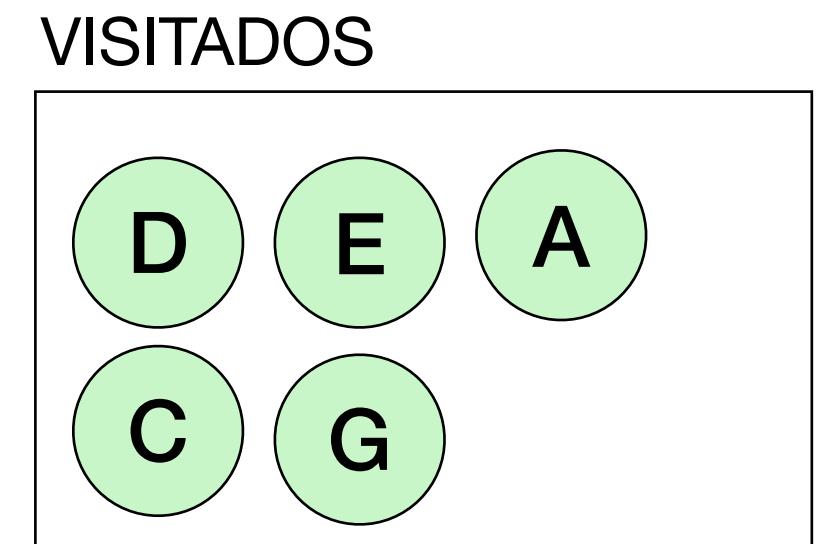
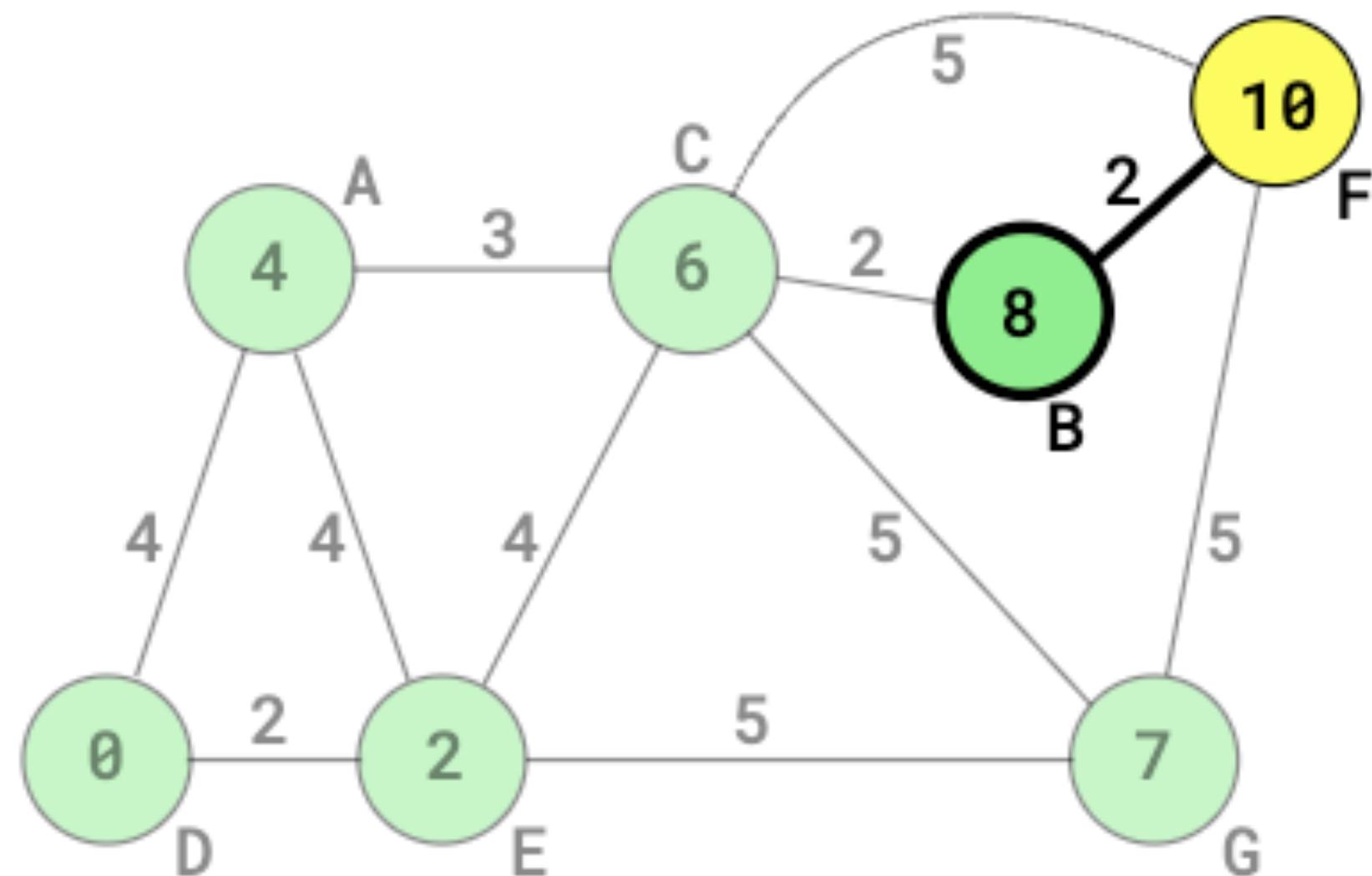
O vértice F obtém a distância atualizada  $6+5=11$  e o vértice B obtém a distância atualizada  $6+2=8$ . A distância calculada até o vértice G através do vértice C é  $6+5=11$ , que é maior que a distância já definida de 7, portanto, a distância até o vértice G não é atualizada. O vértice C é marcado como visitado e o próximo vértice a ser visitado é G porque possui a menor distância entre os vértices não visitados restantes.

# Algoritmo de Dijkstra



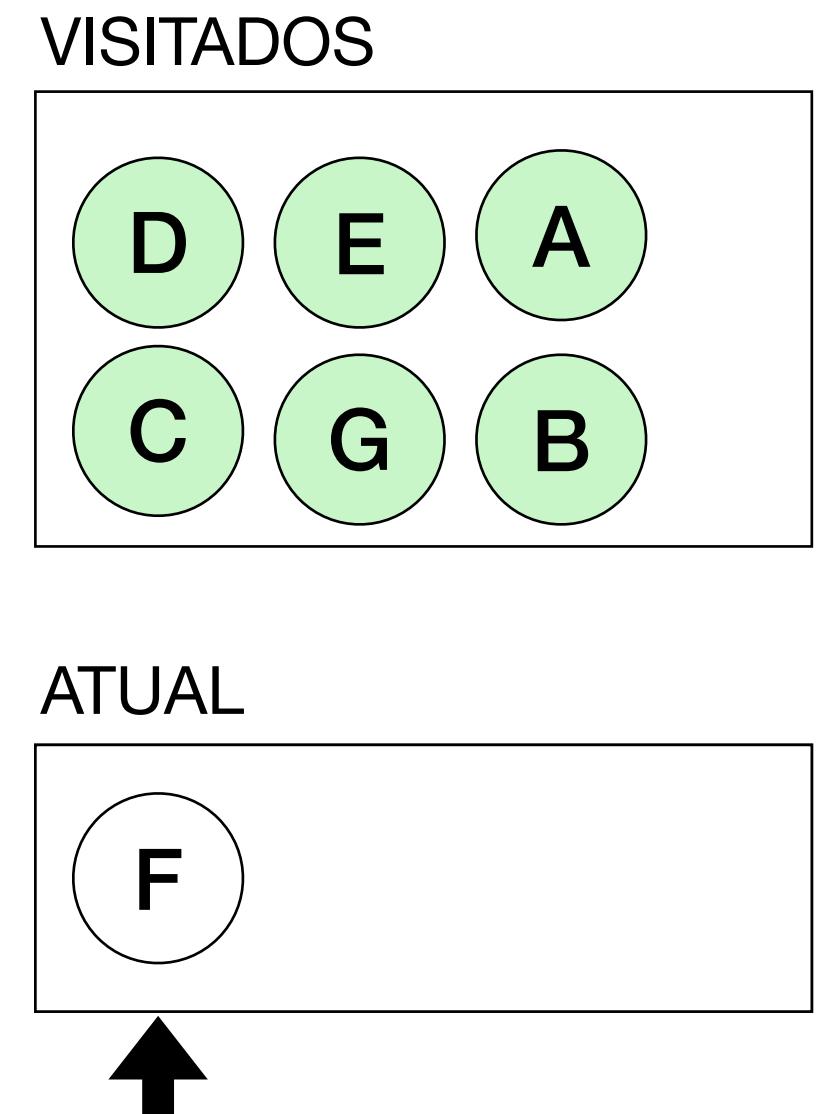
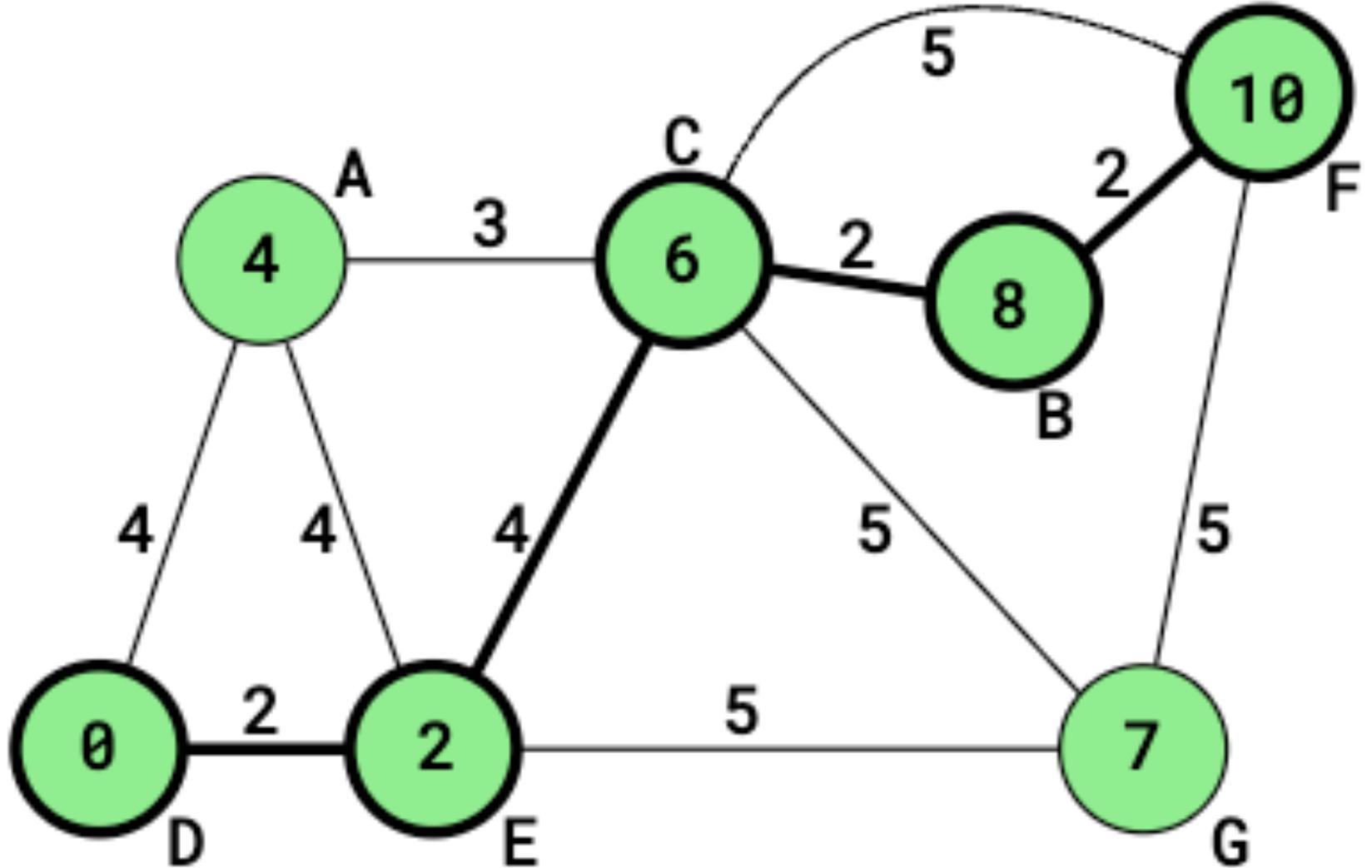
O vértice F já tem uma distância de 11. Isso é menor que a distância calculada de G, que é  $7+5=12$ , portanto a distância até o vértice F não é atualizada. O vértice G é marcado como visitado e B torna-se o vértice atual porque possui a menor distância dos vértices não visitados restantes.

# Algoritmo de Dijkstra



A nova distância para F via B é  $8+2=10$ , porque é menor que a distância existente de 11 para F. O vértice B está marcado como visitado e não há nada para verificar no último vértice F não visitado, então o algoritmo de Dijkstra está concluído.

# Algoritmo de Dijkstra

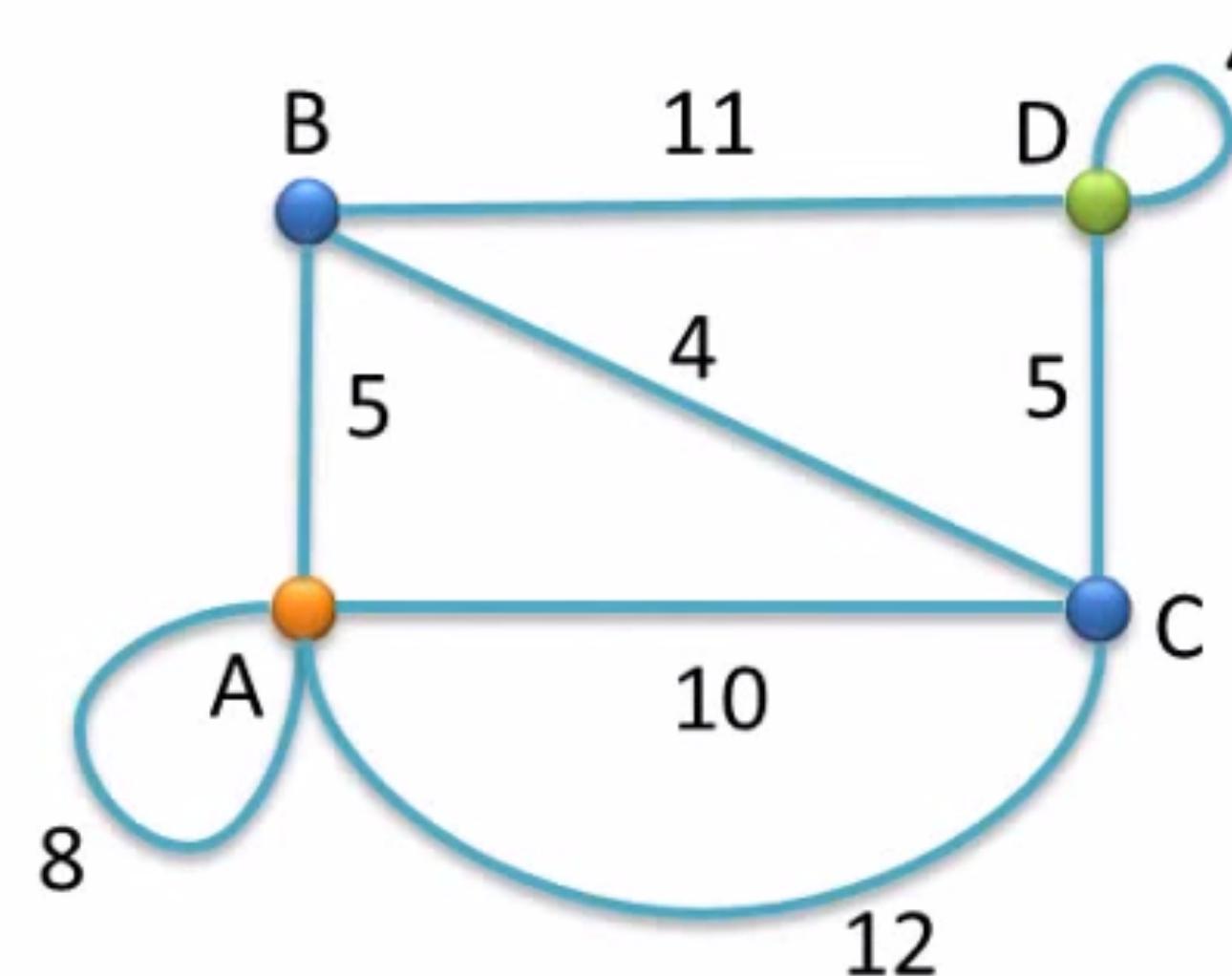


Cada vértice foi visitado apenas uma vez e o resultado é a menor distância do vértice de origem D a todos os outros vértices do gráfico. Os caminhos mais curtos reais também podem ser retornados pelo algoritmo de Dijkstra, além dos valores dos caminhos mais curtos. Assim, por exemplo, em vez de apenas retornar que o valor do caminho mais curto é 10 do vértice D ao F, o algoritmo também pode retornar que o caminho mais curto é "D->E->C->B->F".

# Exercícios

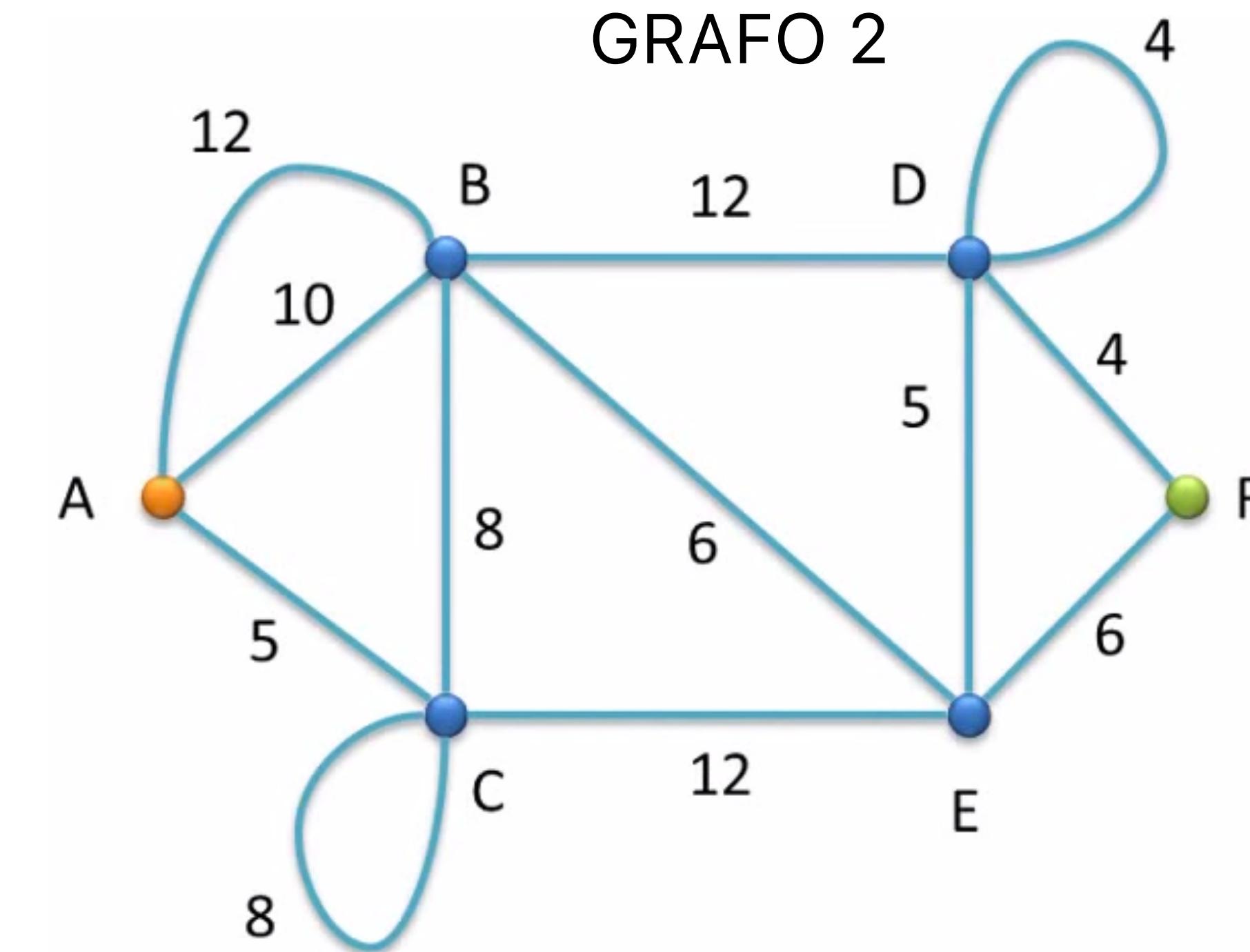
- Usando o algoritmo de Dijkstra, identifique e calcule o menor caminho entre os vértices indicados. Mostre o passo a passo do algoritmo, evidenciando os vértices visitados e o atual.

GRAFO 1



Menor caminho entre A e D

GRAFO 2



Menor caminho entre A e F



Prof. Fernando Sambinelli  
[sambinelli@ifsp.edu.br](mailto:sambinelli@ifsp.edu.br)

