Departamento de Sistemas e Computação – FURB
Curso de Ciência da Computação
Disciplina de Teoria dos Grafos

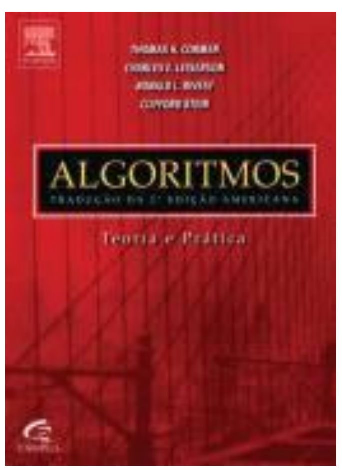
Busca em grafos

Prof. Aurélio Hoppe
aureliof@furb.br
http://www.inf.furb.br/~aurelio/

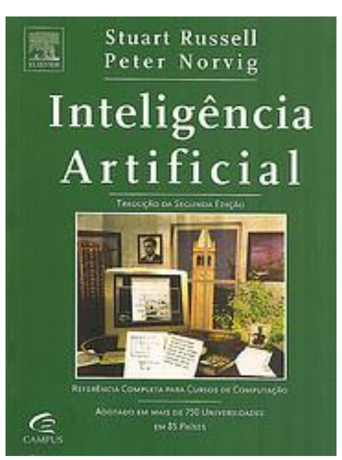
Grupo de Processamento de Imagens,
Análise de dados, Robótica e
Simulação computacional

Bibliografia

Márcia A. Rabuske. **Introdução à Teoria dos Grafos**. Editora da UFSC. 1992



Thomas Cormen et al. **Algoritmos**: teoria e prática. Ed. Campus. 2004.



Stuart Russell e Peter Norvig. **Inteligência artificial**. Rio de Janeiro : Campus, 2004, 1021p.

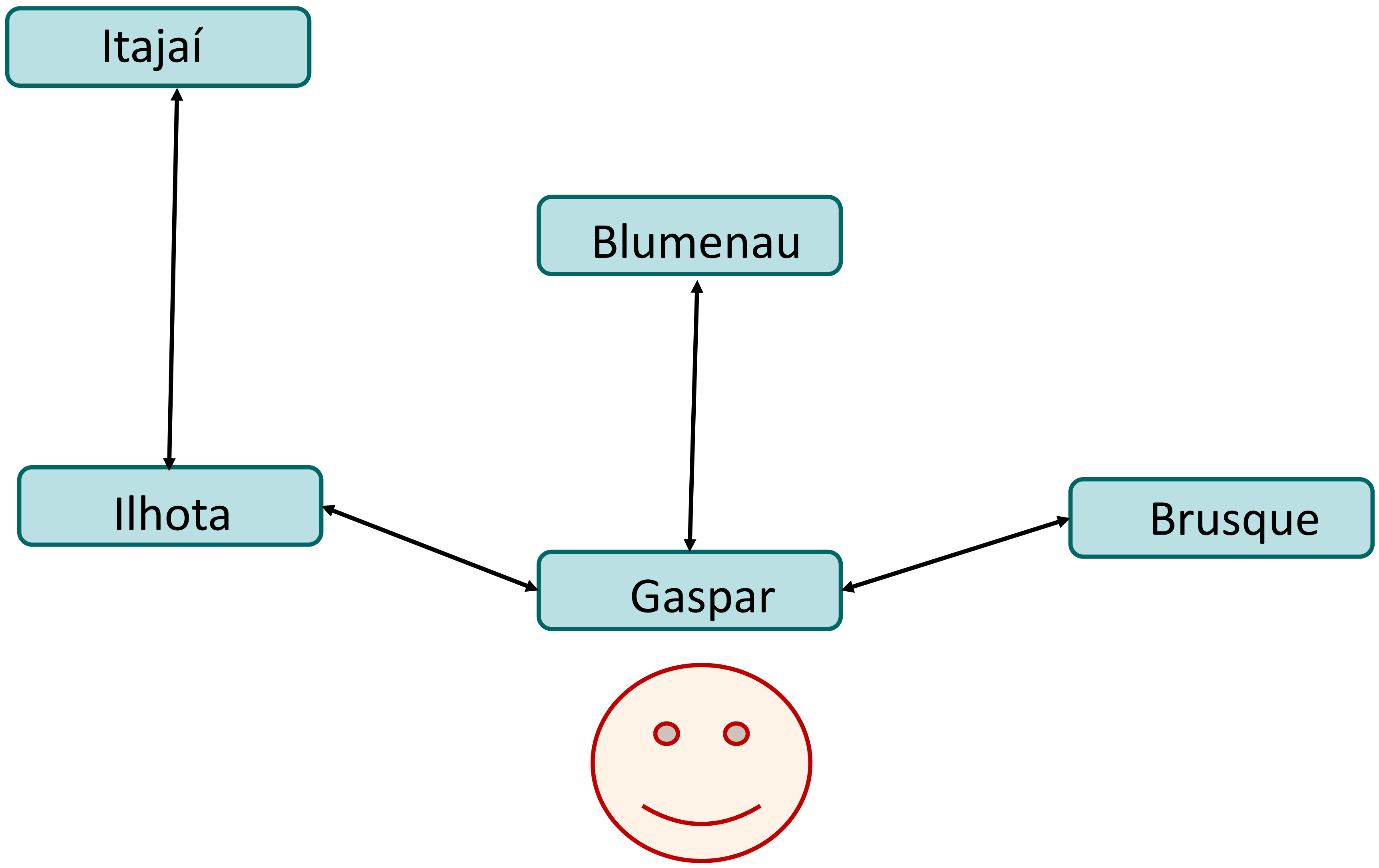
Problema de busca

- O processo de tentar encontrar uma sequência de ações que leva de um estado até um estado objetivo é chamado de busca.

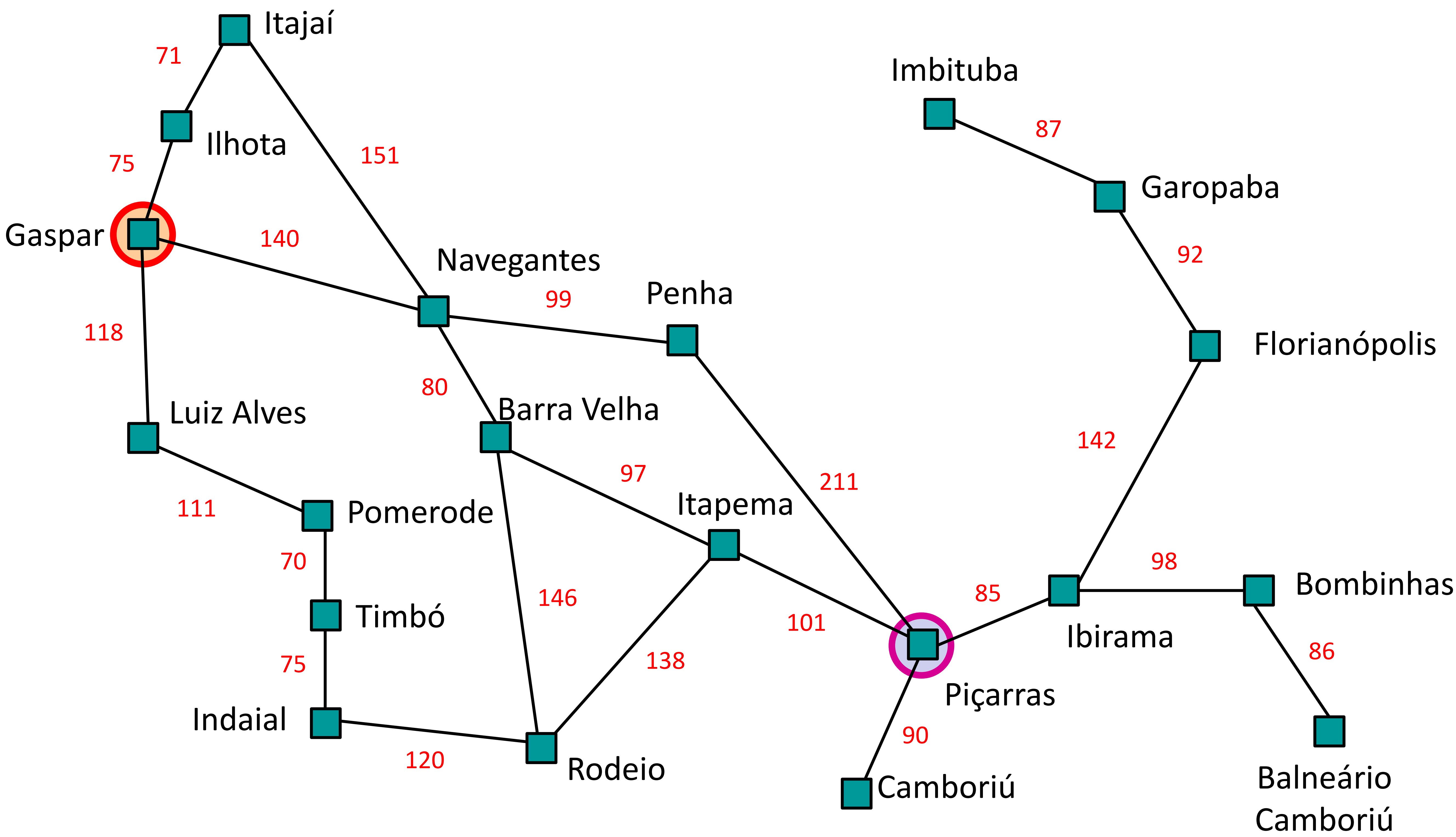
- Uma vez encontrada a solução, pode-se executar a sequência de ações para chegar no objetivo.

- Fases:
  - Formular objetivo
  - Buscar objetivo
  - Executar sequencia de ações

Problema de Busca



Problema de Busca



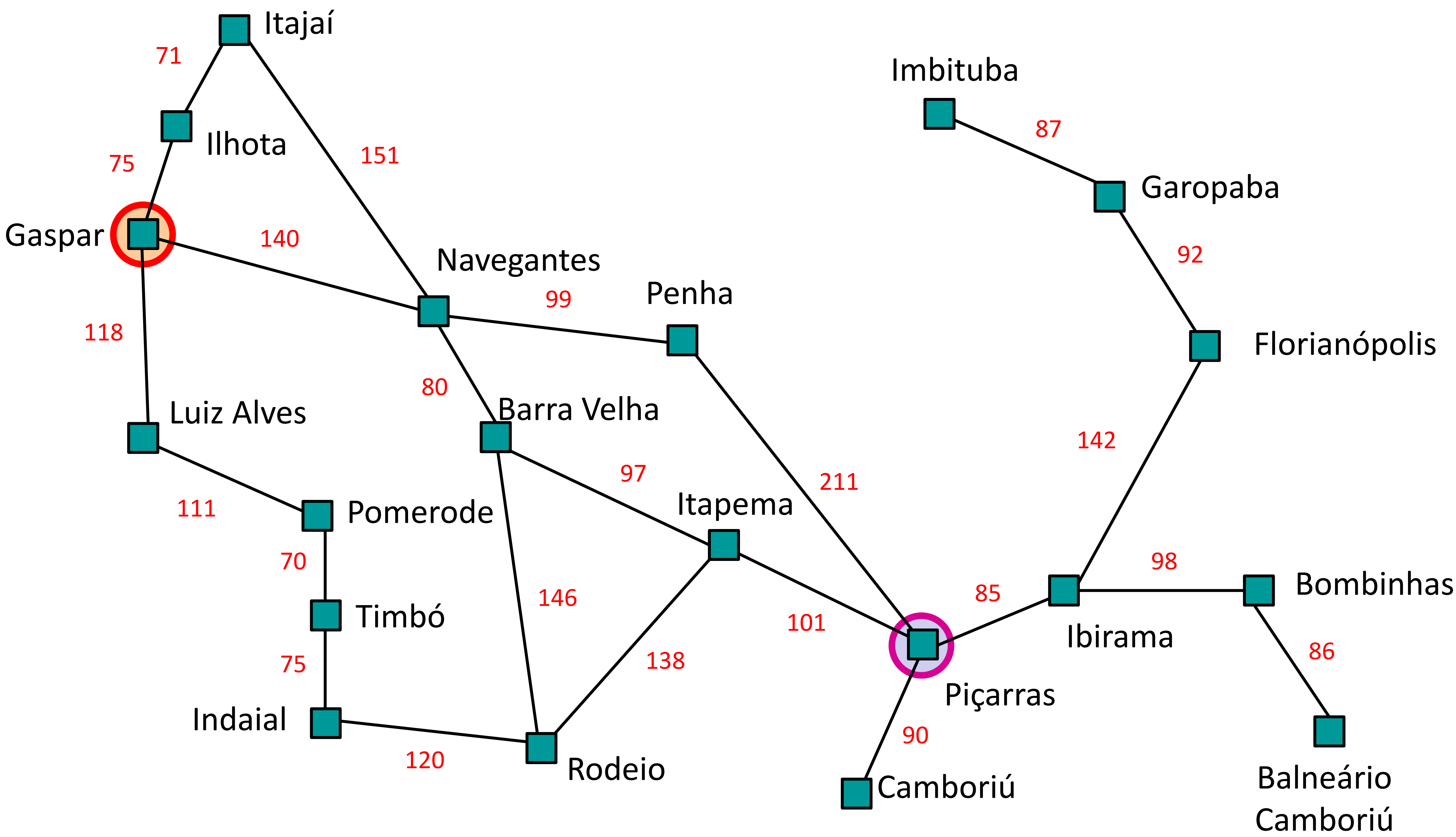
Definição do problema

- A **definição do problema** é a primeira e mais importante etapa do processo de resolução de problemas por meio de buscas.
- Consiste em analisar o **espaço de possibilidades** de resolução do problema, encontrar sequências de ações que levem a um objetivo desejado

Definição do problema

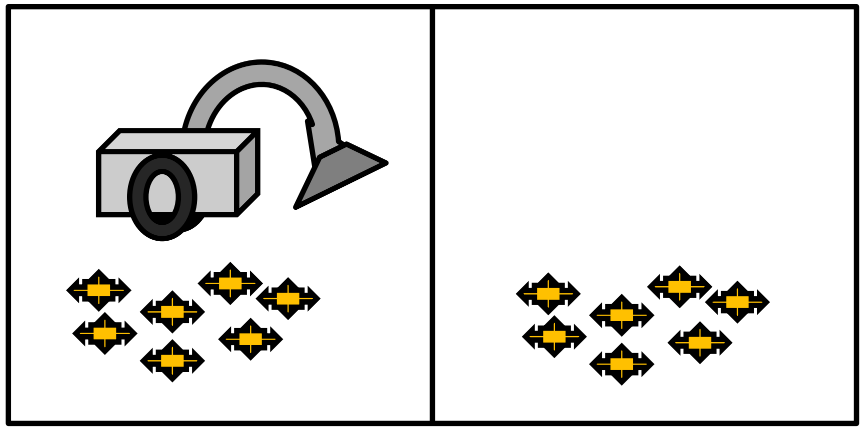
- **Estado Inicial**: Estado inicial do agente.
  - Ex: Em(Gaspar)
- **Estado Final**: Estado buscado pelo agente.
  - Ex: Em(Piçarras)
- **Ações Possíveis**: Conjunto de ações que o agente pode executar.
  - Ex: Ir(Cidade, PróximaCidade)
- **Espaço de Estados**: Conjunto de estados que podem ser atingidos a partir do estado inicial.
  - Ex: Mapa da Santa Catarina.
- **Custo**: Custo numérico de cada caminho.
  - Ex: Distância em KM entre as cidades.

Problema de Busca



Mundo do Aspirador

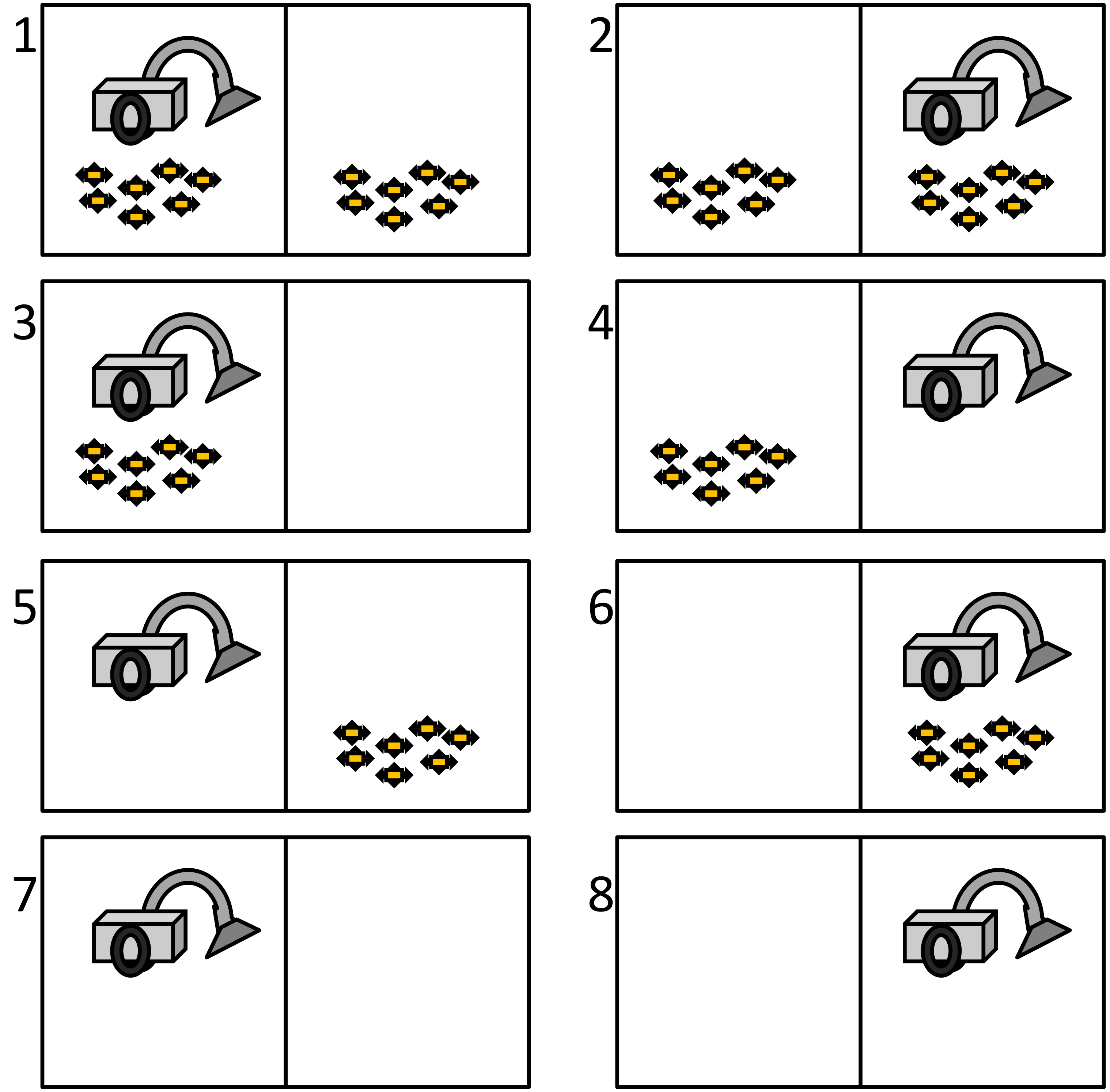
- **Estado do mundo**: localização do agente e localizações com sujeira
- **Conjunto de Estados**: o agente está em um dos 2 cômodos e cada um pode estar sujo ou limpo: 8 estados do mundo possíveis
- **Estado Inicial**: qualquer um dos 8 estados
- **Função sucessor**: gera os estados legais resultantes da execução das 3 ações: *Esquerda, Direita e Aspira*



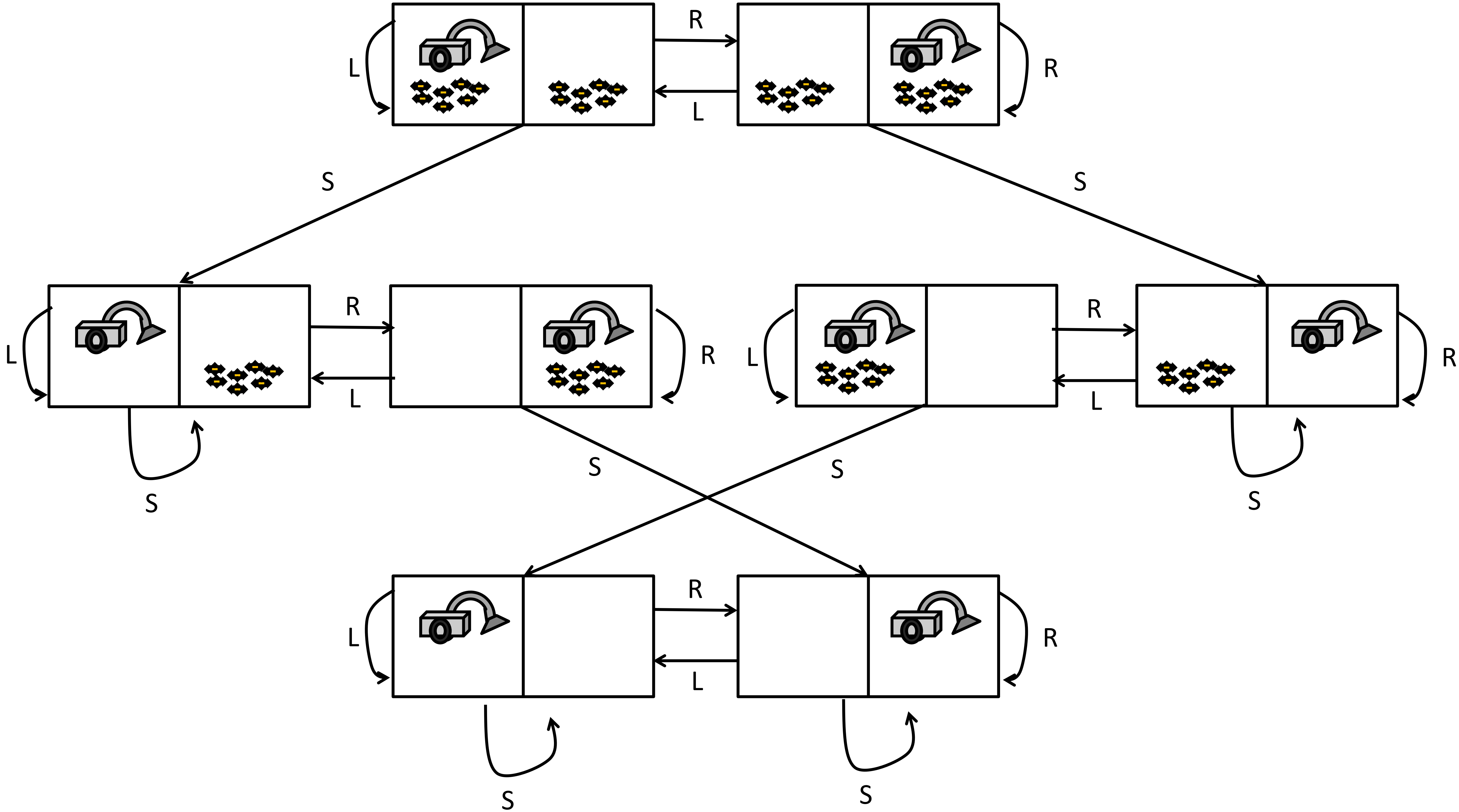
- **Teste de meta**: teste se os 2 cômodos estão limpos (sensores: sensor de posição; sensor local de sujeira)
- **Custo**: cada passo custo 1, o custo do caminho é o número de passos do caminho

Exemplo Aspirador de pó

- **Espaço de Estados:** 8 estados possíveis (figura ao lado)
- **Estado Inicial:** Qualquer estado
- **Estado Final:** Estado 7 ou 8 (ambos quadrados limpos)
- **Ações Possíveis:** Mover para direita, mover para esquerda e limpar
- **Custo:** Cada passo tem o custo 1, assim o custo do caminho é definido pelo número de passos



Exemplo Aspirador de pó



Exemplo: 8-Puzzle

- **Espaço de Estados:** 181.440 possíveis estados
- **Estado Inicial:** Qualquer estado
- **Estado Final:** Figura ao lado – Goal State
- **Ações Possíveis:** Mover o quadrado vazio para direita, para esquerda, para cima ou para baixo
- **Custo:** Cada passo tem o custo 1, assim o custo do caminho é definido pelo número de passos

- **15-puzzle (4x4) –** 1.3 trilhões estados possíveis.
- **24-puzzle (5x5) –** 10^{25} estados possíveis

7	2	4
5		6
8	3	1

Estado inicial

	1	2
3	4	5
6	7	8

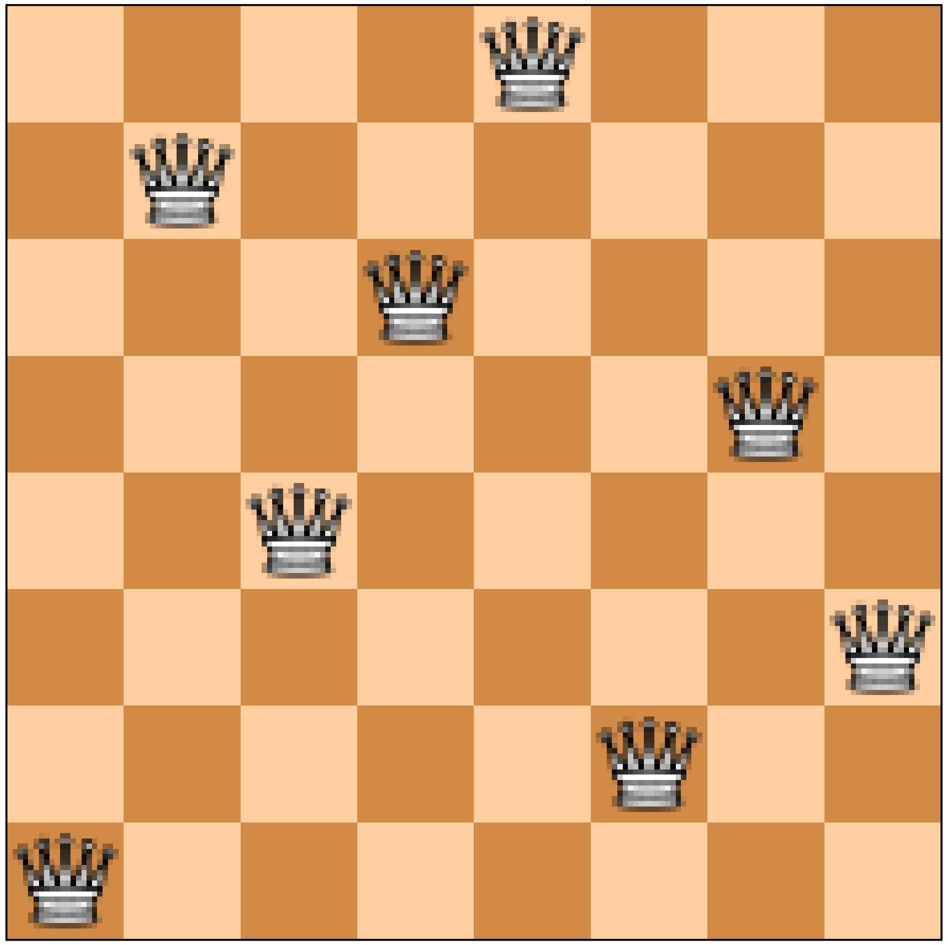
Estado final

Exemplo: Xadrez

- **Espaço de Estados:** Aproximadamente 10^{40} possíveis estados (Claude Shannon, 1950)
- **Estado Inicial:** Posição inicial de um jogo de xadrez
- **Estado Final:** Qualquer estado onde o rei adversário está sendo atacado e o adversário não possui movimentos válidos
- **Ações Possíveis:** Regras de movimentação de cada peça do xadrez
- **Custo:** Quantidade de posições examinadas

Exemplo: 8 Rainhas

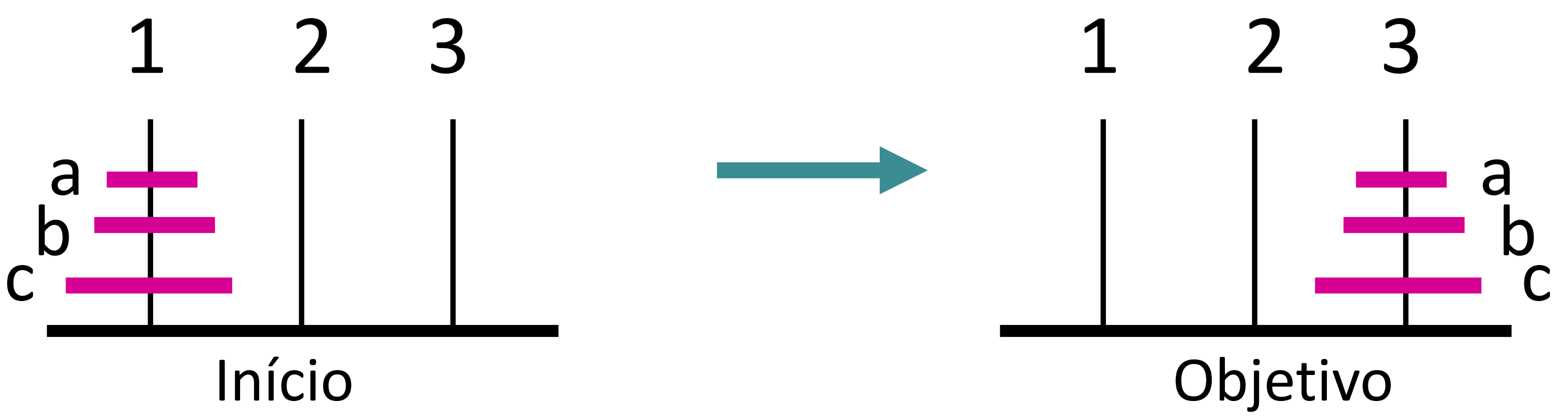
- **Espaço de Estados:** Qualquer disposição de 0 a 8 rainhas no tabuleiro (1.8×10^{14} possíveis estados);
- **Estado Inicial:** Nenhuma rainha no tabuleiro;
- **Estado Final:** Qualquer estado onde as 8 rainhas estão no tabuleiro e nenhuma esta sendo atacada;
- **Ações Possíveis:** Colocar uma rainha em um espaço vazio do tabuleiro;
- **Custo:** Não importa nesse caso;



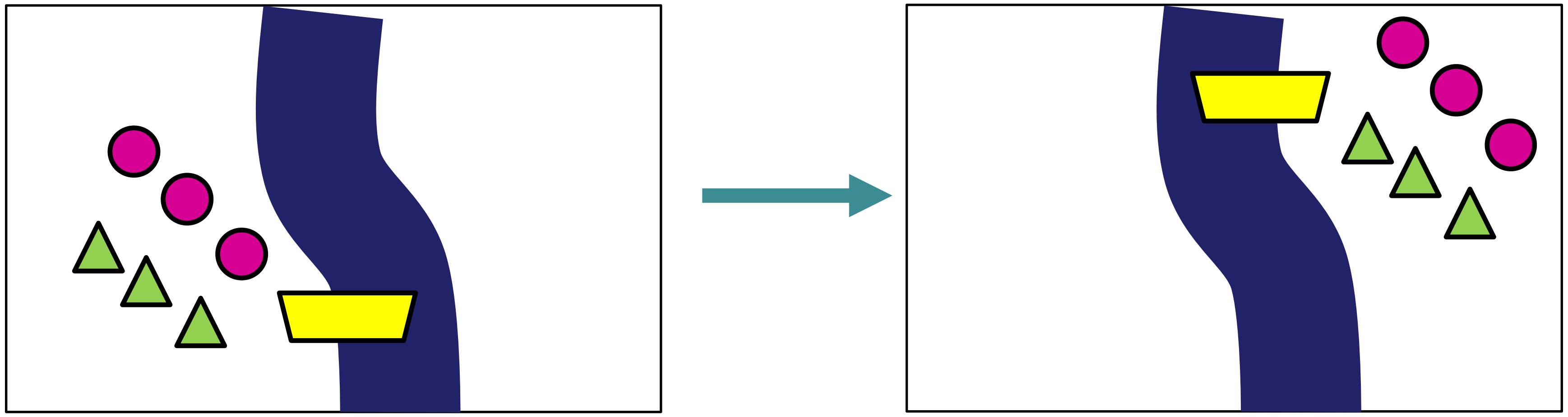
* O jogo possui apenas 92 possíveis soluções (considerando diferentes rotações e reflexões). E apenas 12 soluções únicas.

Exercícios

- Torre de Hanói?



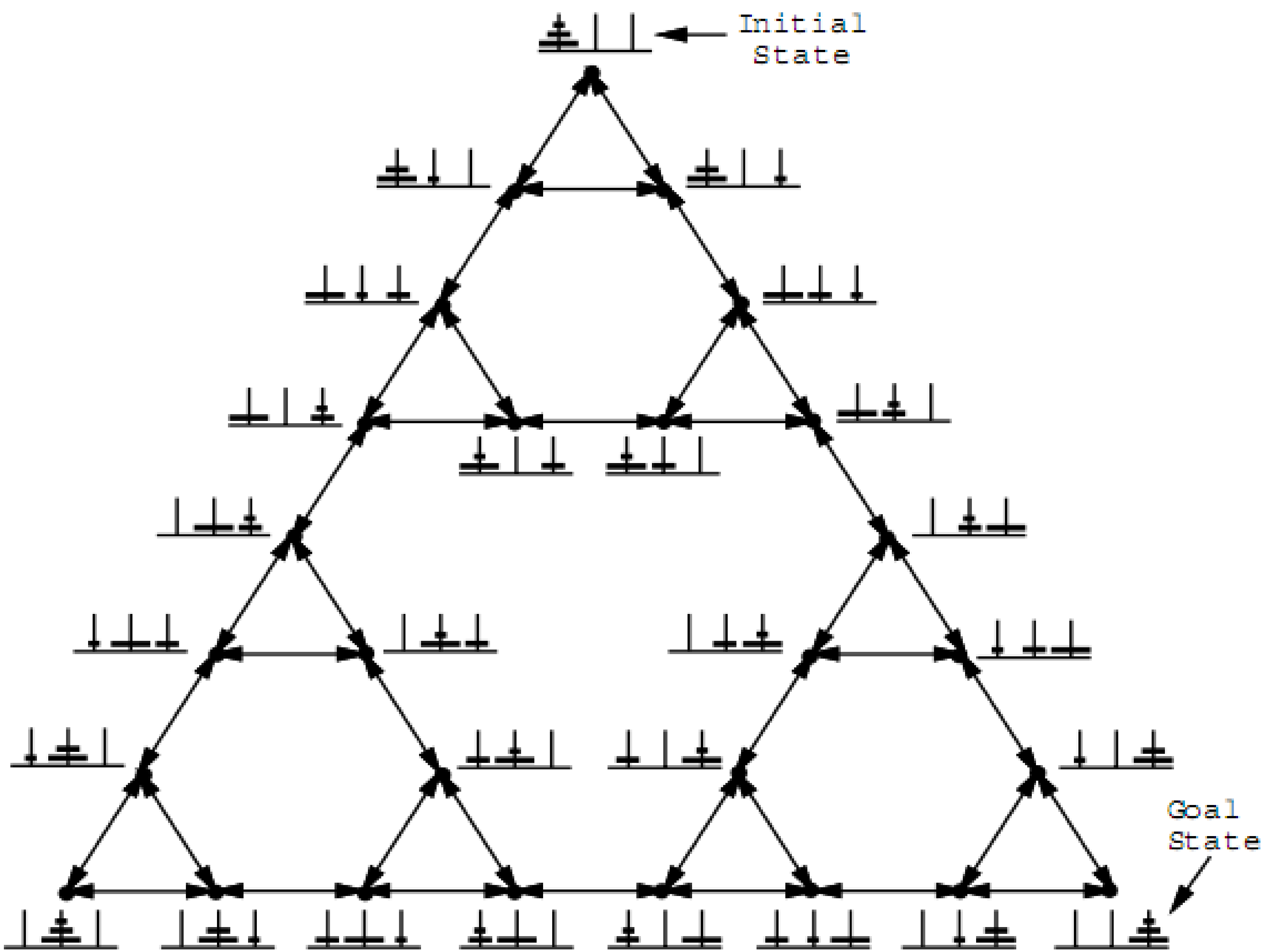
- Canibais e Missionários?



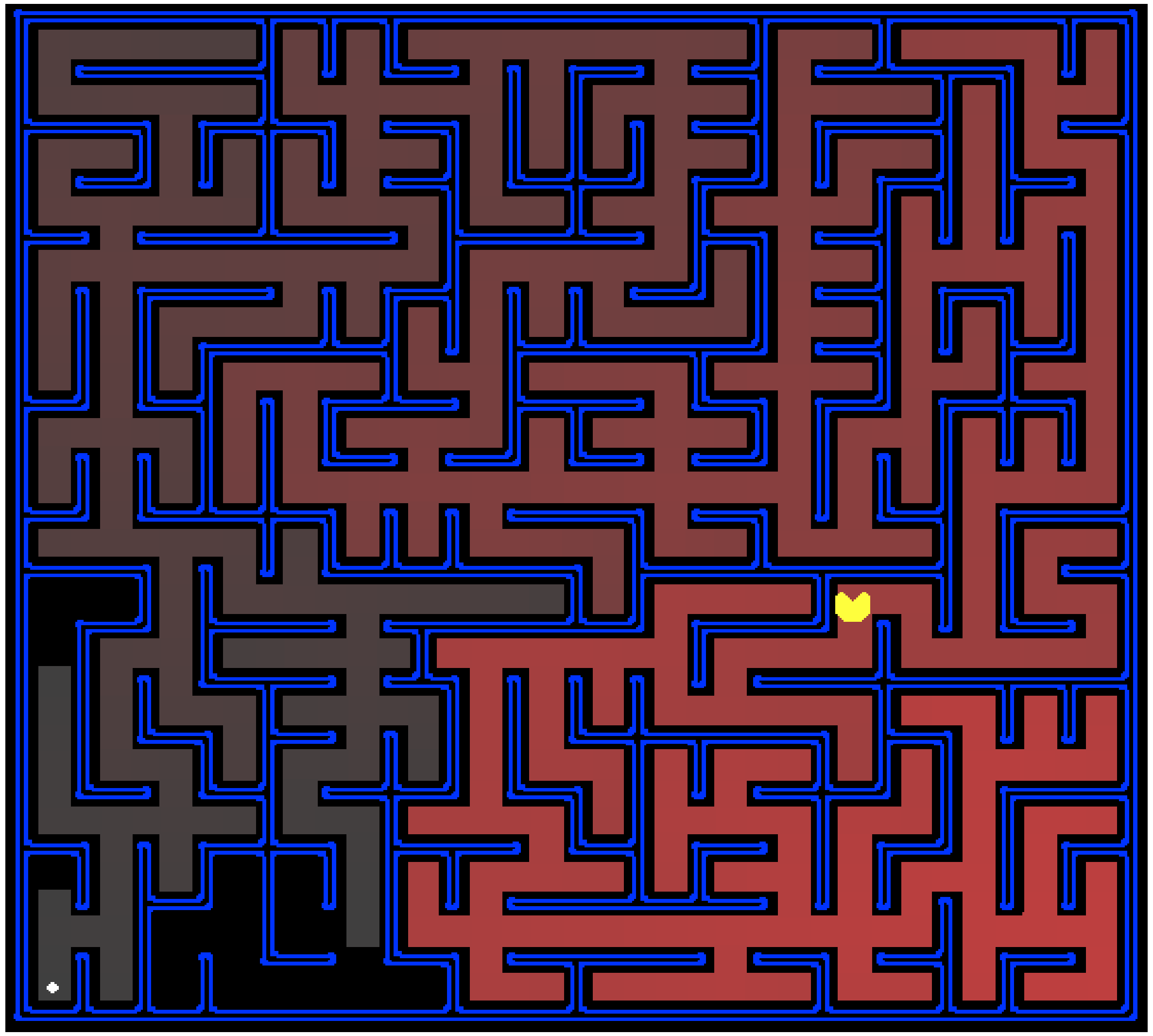
Exercícios

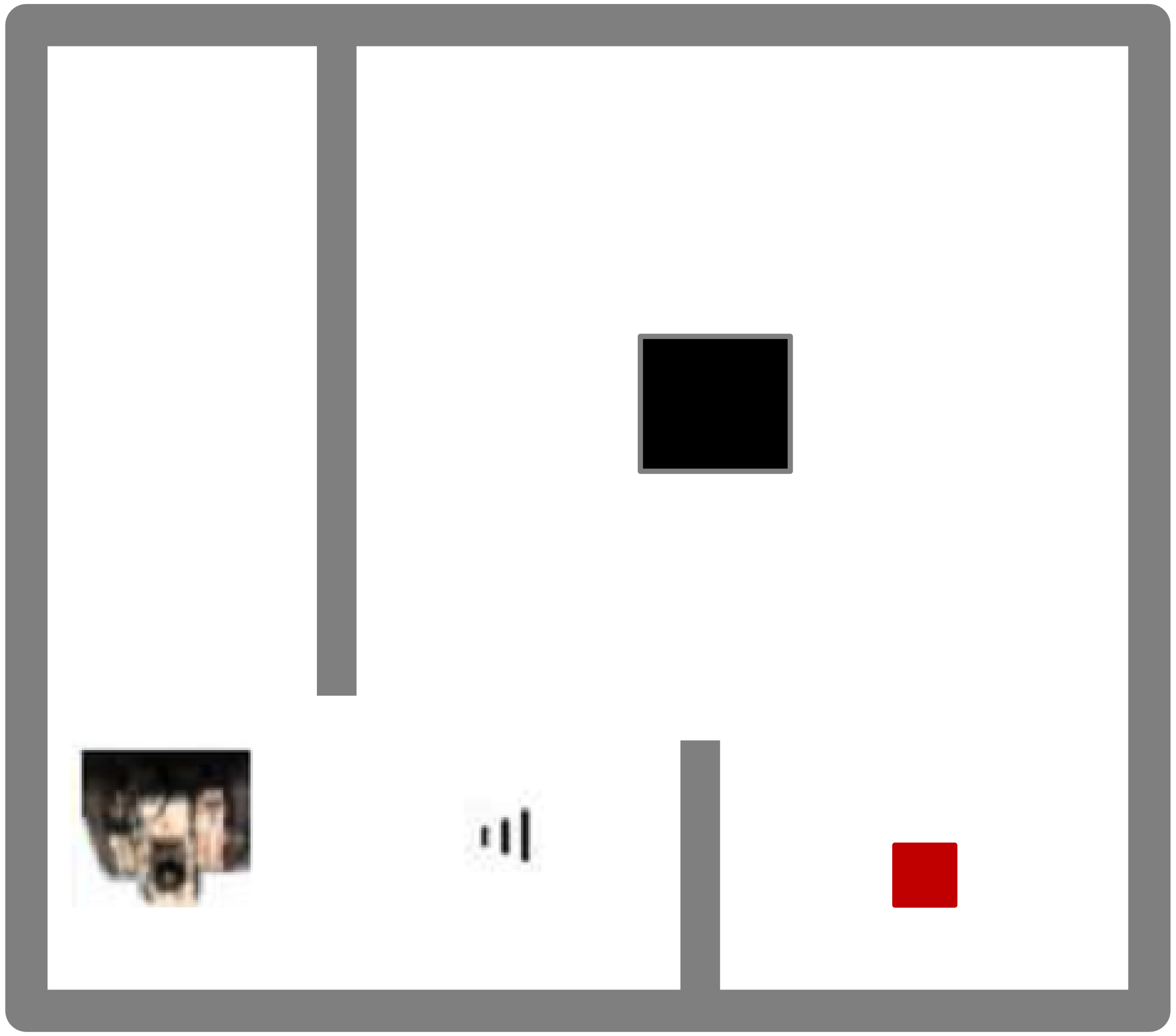
- Torre de Hanói:
  - **Espaço de Estados**: Todas as possíveis configurações de argolas em todos os pinos (27 possíveis estados).
  - **Ações Possíveis:** Mover a primeira argola de qualquer pino para o pino da direita ou da esquerda.
  - **Custo**: Cada movimento tem 1 de custo.

Exercícios



Aplicações em problemas reais





- **Robótica:**
  - Navegação e busca de rotas em ambientes reais
  - Montagem de objetos por robôs

Aplicações em problemas reais

- **Cálculo de Rotas:**
  - Planejamento de rotas de aviões
  - Sistemas de planejamento de viagens
  - Caixeiro viajante
  - Rotas em redes de computadores
  - Jogos de computadores (rotas dos personagens)
- **Alocação**
  - Salas de aula
  - Máquinas industriais
- **Circuitos Eletrônicos:**
  - Posicionamento de componentes
  - Rotas de circuitos
- **Outras aplicações:**
  - Compiladores
  - Função “localizar arquivo” no sistema operacional
  - Detecção de deadlocks
  - Dentre centenas de outras aplicações....

Como Encontrar a Solução?

- Uma vez o problema bem formulado, o estado final (objetivo) deve ser “**buscado**” no espaço de estados.
- A busca é representada em uma **árvore de busca**:
  - Raiz: corresponde ao estado inicial
  - Expande-se o estado corrente, gerando um novo conjunto de sucessores
  - Escolhe-se o próximo estado a expandir seguindo uma estratégia de busca
  - Prossegue-se até chegar ao estado final (solução) ou falhar na busca pela solução;

Medida de Desempenho

- **Desempenho do Algoritmo:**
  - (1) O algoritmo encontrou alguma solução?
  - (2) É uma boa solução?
    - Custo de caminho (qualidade da solução).
  - (3) É uma solução computacionalmente barata?
    - Custo da busca (tempo e memória).
- **Custo Total**
  - Custo do Caminho + Custo de Busca.

Métodos de Busca

- **Busca Cega ou Exaustiva:**
  - Não sabe qual o melhor nó da fronteira a ser expandido. Apenas distingue o estado objetivo dos não objetivos.
- **Busca Heurística:**
  - Estima qual o melhor nó da fronteira a ser expandido com base em funções heurísticas.

Busca Cega

- **Algoritmos de Busca Cega:**
  - Busca em largura
  - Busca de custo uniforme
  - Busca em profundidade
  - Busca com aprofundamento iterativo

Busca em grafos

Definição: a Busca em Grafos (ou Percurso em Grafos) é a examinação de vértices e arestas de um grafo

Em uma busca:

- Uma aresta ou vértice ainda não examinados são marcados como não explorados ou não visitados
- Inicialmente, todos os vértices e arestas são marcados como não explorados
- Após terem sido examinados, os mesmos são marcados como explorados ou visitados
- Ao final, todos os vértices e arestas são marcados como explorados (no caso de uma busca completa).

Busca em grafos

Alguns objetivos:

- determinar quais vértices são alcançáveis através de um vértice inicial...
- determinar se um determinado objeto está presente no grafo...
- identificar algumas características dos grafos...

Busca em grafos

Atravessando labirintos:

Como escapar?

Busca em grafos

Método do século XIX - Algoritmo de *Trémaux*

Pierre Trémaux (20 de Julho 1818 - 12 Março de 1895)

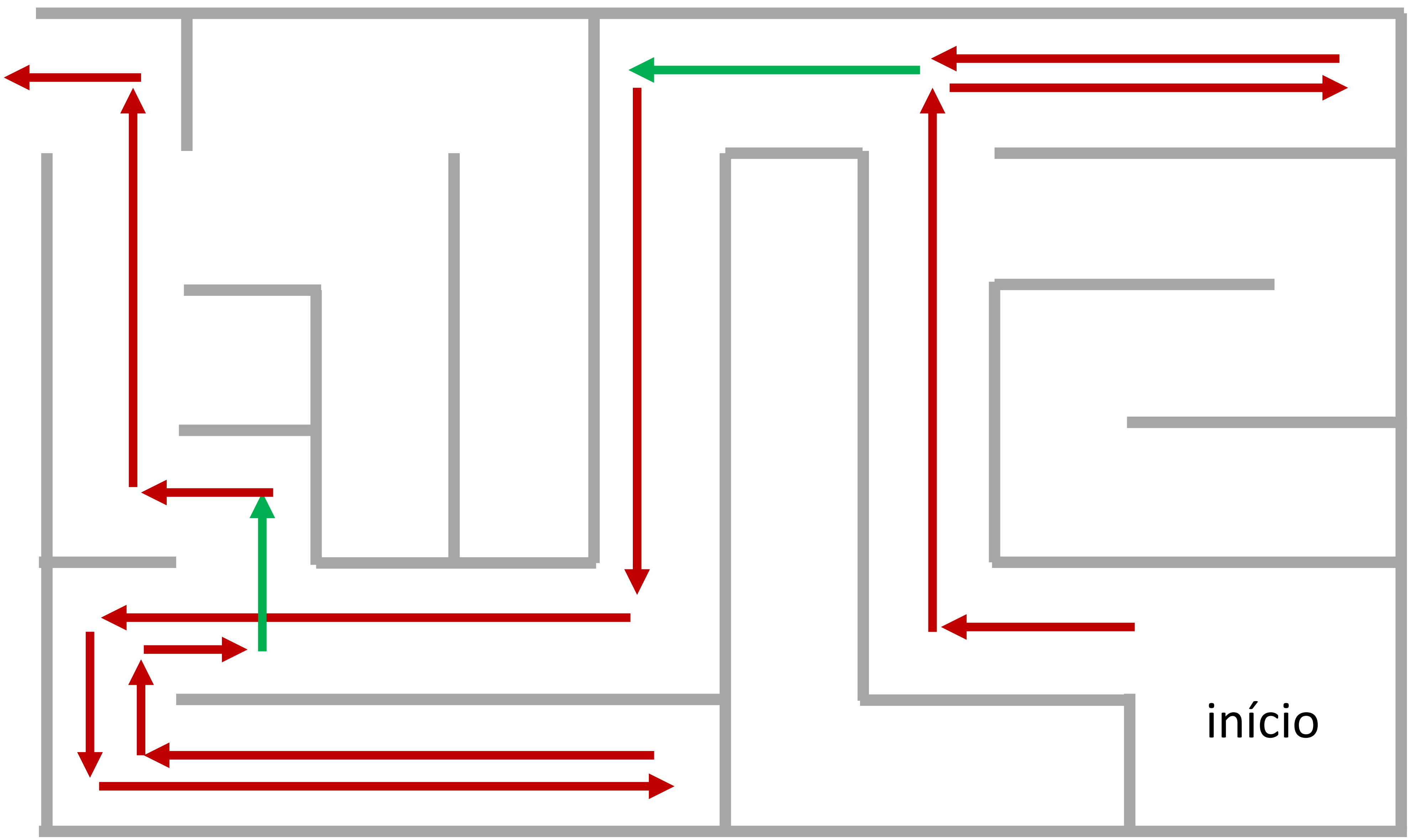
- Francês arquiteto, fotógrafo e orientalista
- Autor de várias publicações científicas e etnografias
- Segunda pessoa a ganhar o *Prix de Romea*.
  Bolsa de estudo destinada a estudantes das artes e atribuída pelo governo francês a jovens artistas que se distinguissem na pintura, escultura e arquitetura, criada em 1663 durante o reinado de Luís XIV de França

Busca em grafos

O algoritmo de *Trémaux* *(século XIX)* requer que para encontrar a saída de um labirinto seja riscada uma linha no chão para marcar os caminhos percorridos:

1. Inicialmente, uma direção aleatória é escolhida
2. Ao chegar em uma junção não visitada (ou seja, sem nenhuma linha), escolha uma direção aleatória e risque o caminho
3. Ao chegar em uma junção por um caminho já marcado, vire-se e caminhe de volta, marcando o caminho pela segunda vez
4. Se este não for o caso, escolha o caminho com menos linhas e marque-o novamente.
5. Quando finalmente chegar à saída do labirinto, os caminhos marcados com apenas uma linha indicarão o caminho direto até o ponto inicial
6. Se não houver saída, você voltará ao ponto inicial, no qual todos os caminhos possuem duas linhas.

Busca em grafos



Exemplo de execução do Algoritmo de *Trémaux*.

Busca em grafos

Dependendo do critério utilizado para escolha dos vértices e arestas a serem examinados, diferentes tipos de buscas são desenvolvidos.

Algoritmos clássicos de Busca:

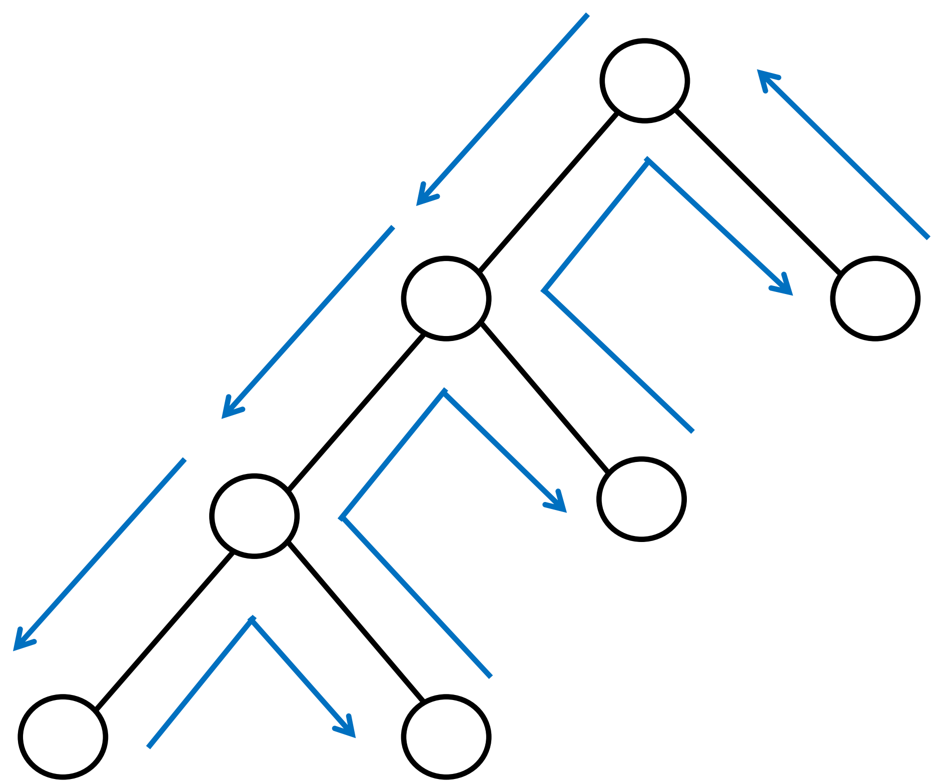
- Busca em Profundidade (ou **DFS** – *Depth-First Search*)

- Busca em Largura (ou **BFS** – *Breadth-First Search*)

Busca em profundidade

A busca em profundidade (do inglês depth-first search-DFS) é um algoritmo para caminhar no grafo.

A DFS explora todos os vértices de um grafo, usando como critério o vértice visitado mais recentemente e não marcado



Característica: utiliza uma pilha implícita ou recursividade para guiar a busca.

Busca em profundidade

<u>Ideia geral:</u>

Quando todas arestas adjacentes a v tiverem sido exploradas, a busca “anda para trás” (do inglês backtrack) para explorar vértices do qual v foi descoberto

- O processo continua até que sejam descobertos todos os vértices que são alcançáveis a partir do vértice original

- Se todos os vértices já foram descobertos, então é o fim.

Caso contrário o processo continua a partir de um novo vértice de origem ainda não descoberto (grafos desconexos).

- Este é um ponto diferenciado da busca em árvore que vocês já conhecem
- Pois ao final de uma busca simples, pode haver vértices que não foram alcançados

Busca em profundidade

Legenda para descoberta e finalização...

a	/	Vértice desconhecido
b	d/	Vértice encontrado
c	d/f	Vértice encontrado, com fecho positivo totalmente visitado

- d: marcador do instante que o vértice foi descoberto
- f: marcador do instante que o fecho transitivo do vértice foi totalmente visitado (considerado então finalizado)

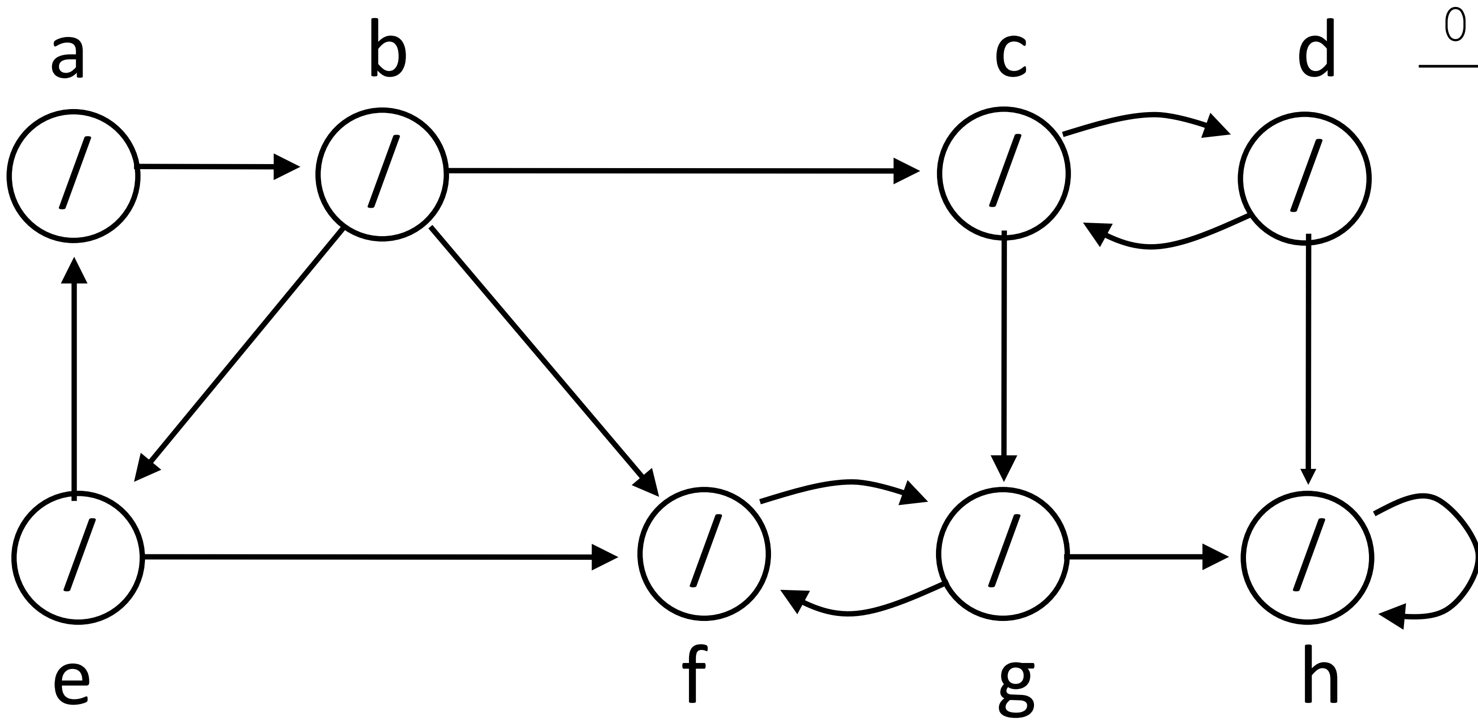
Busca em profundidade

```
DFS(G)
para cada vértice u ← V[G]
    cor[u] ← BRANCO
tempo ← 0
para cada vértice u ∈ V[G]
    se cor[u] = BRANCO
       DFS-VISIT(u)
```

```
DFS-VISIT(u)
cor[u] ← CINZA
tempo ← tempo + 1
d[u] ← tempo
para cada vértice v ∈ Adj(u)
    se cor [v] = BRANCO
       DFS-VISIT(v)
cor[u] ← PRETO
f[u] ← tempo ← (tempo + 1)
```

Busca em profundidade

```
DFS(G)
para cada vértice u ← V[G]
    cor[u] ← BRANCO
tempo ← 0
para cada vértice u ∈ V[G]
    se cor[u] = BRANCO
        DFS-VISIT(u)
```



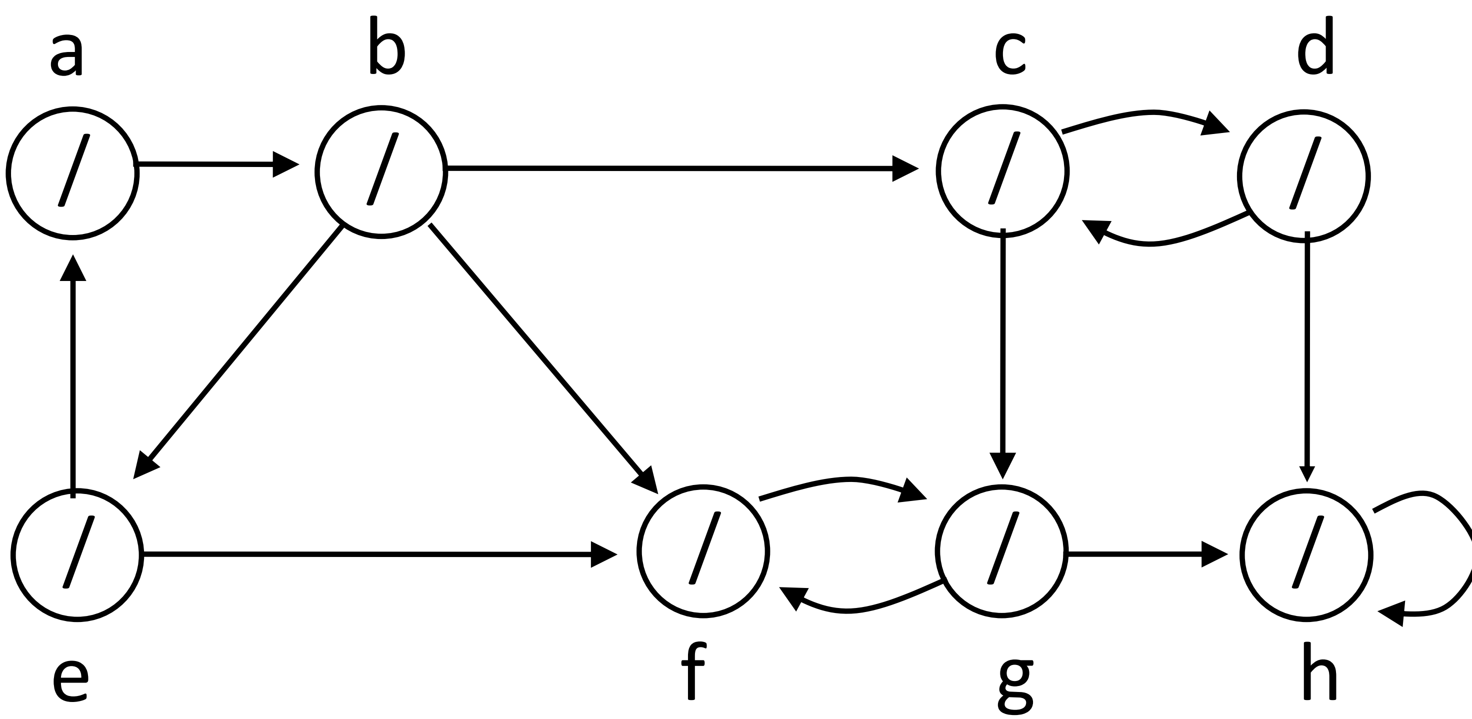
Lista [c, a, b, d, e, f, g, h]

Dado um grafo, temos uma lista de todos os vértices...

Busca em profundidade

Chamada de função DFS_Visit(c)

Vai empilhar a função DFS(G), com o CP = 4, e próximo u=a




```
DFS(G)
01. para cada vértice u ← V[G]
02.      cor[u] ← BRANCO
03. tempo ← 0
04. para cada vértice u ∈ V[G]
05.      se cor[u] = BRANCO
06.          DFS-VISIT(u)
```

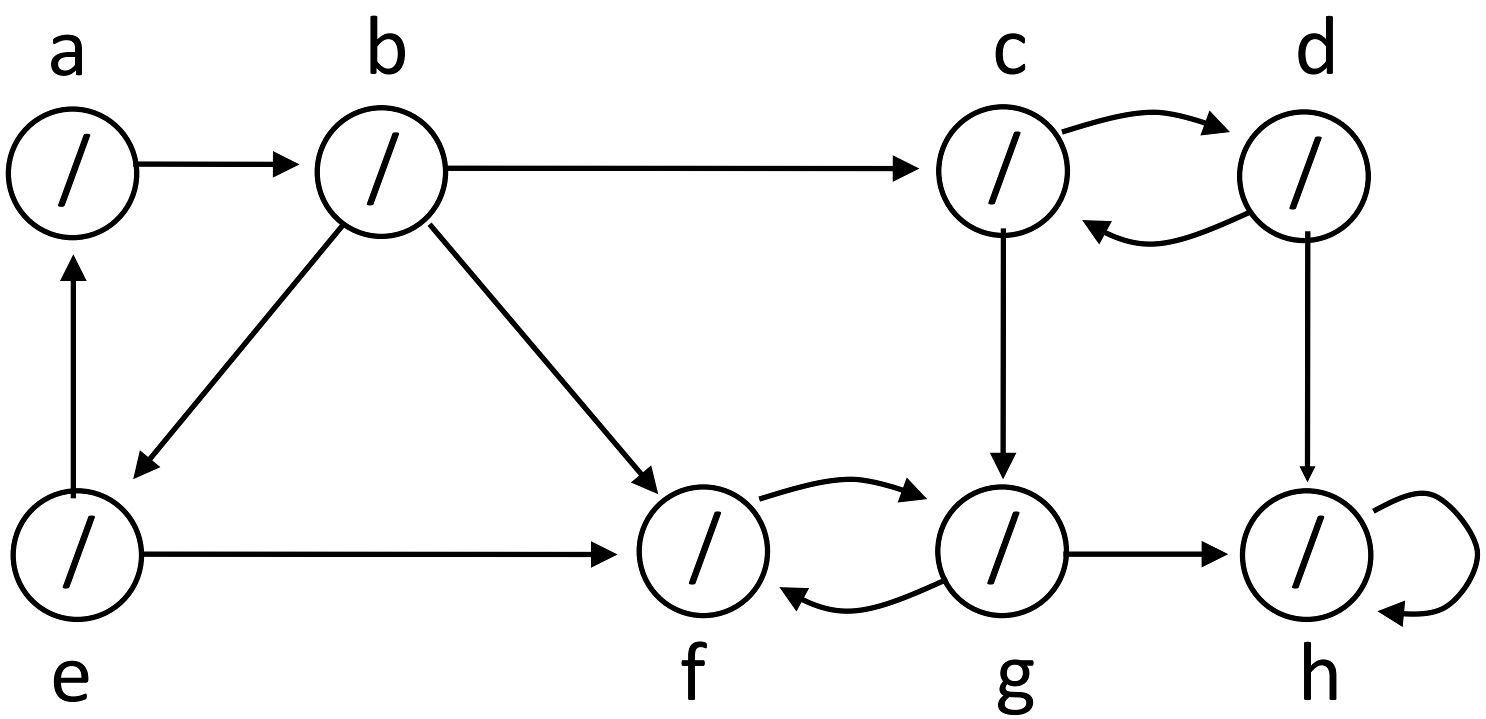
Lista [c, a, b, d, e, f, g, h]

tempo = 0

Busca em profundidade

Chamada de função DFS_Visit(c)

```
DFS-VISIT(u)
01. cor[u] ← CINZA
03. tempo ← tempo + 1
03. d[u] ← tempo
04. para cada vértice v ∈ Adj(u)
05.      se cor [v] = BRANCO
06.         DFS-VISIT(v)
07. cor[u] ← PRETO
08. f[u] ← tempo ← (tempo + 1)
```



Lista [c, a, b, d, e, f, g, h]

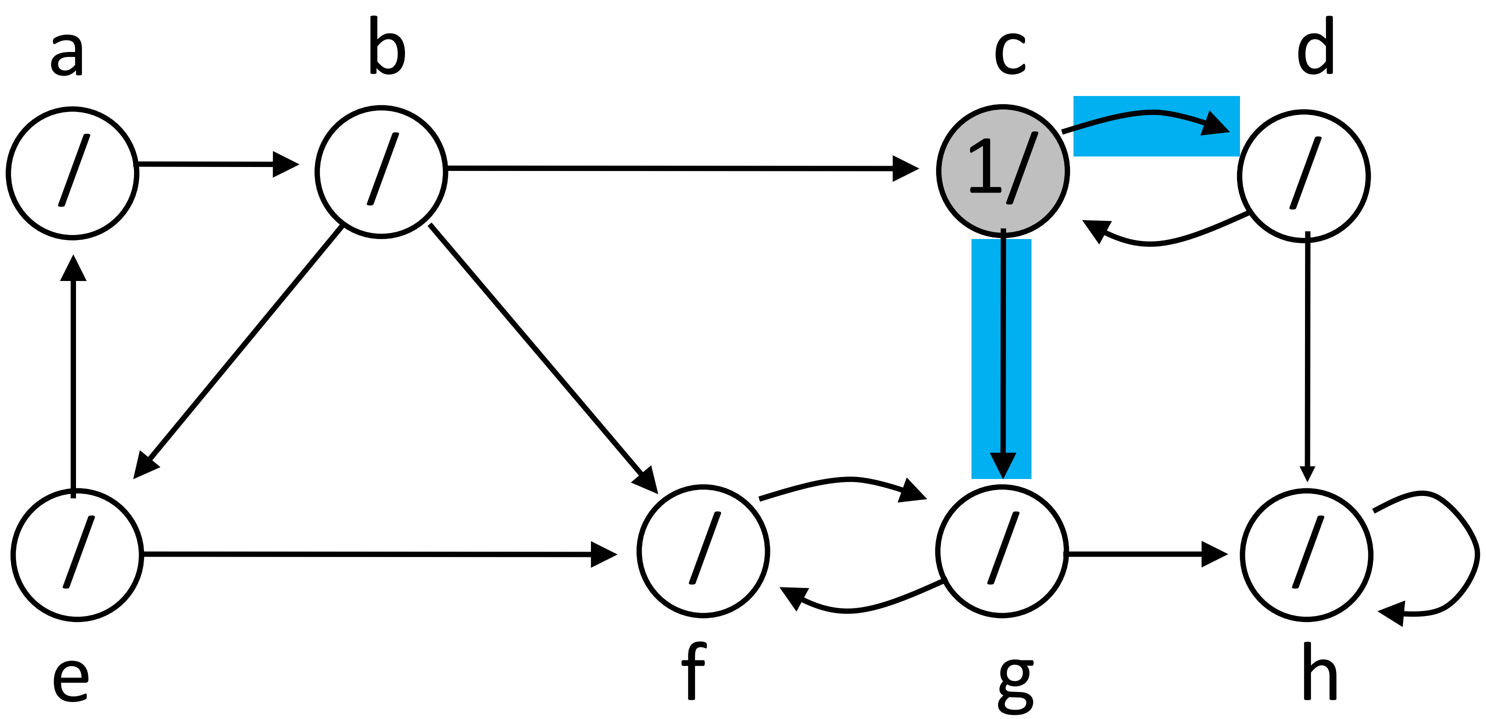
Pilha de execução:

DFS(G) – CP: linha 4 – próximo: u = a

tempo = 0

Busca em profundidade

```
DFS-VISIT(u)
01. cor[u] ← CINZA
03. tempo ← tempo + 1
03. d[u] ← tempo
04. para cada vértice v ∈ Adj(u)
05.       se cor [v] = BRANCO
06.          DFS-VISIT(v)
07. cor[u] ← PRETO
08. f[u] ← tempo ← (tempo + 1)
```



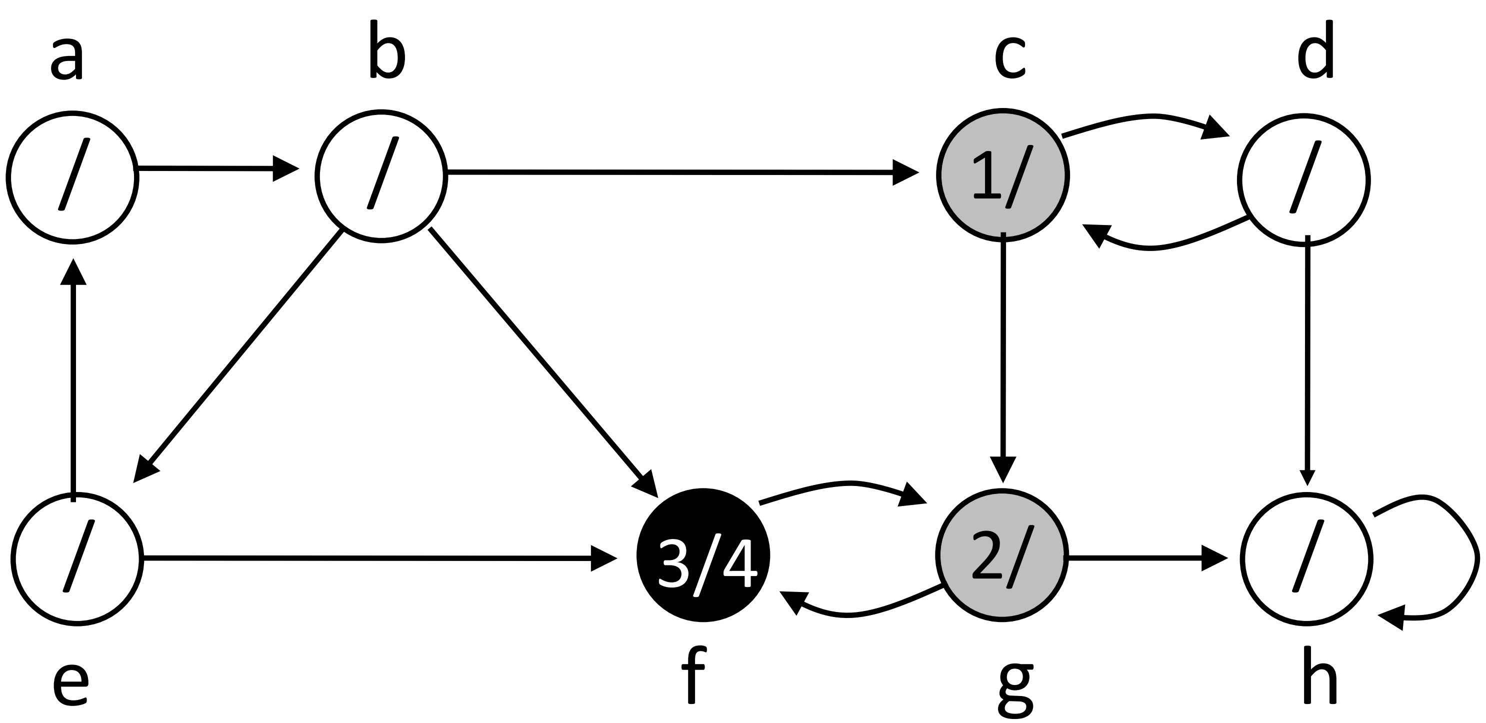
Lista [c, a, b, d, e, f, g, h]

Pilha de execução:

DFS(G) – CP: linha 4 – próximo: u = a

tempo = 1

Busca em profundidade



Lista [c, a, b, d, e, f, g, h]

```
DFS-VISIT(u)
01. cor[u] ← CINZA
03. tempo ← tempo + 1
03. d[u] ← tempo
04. para cada vértice v ∈ Adj(u)
05.       se cor [v] = BRANCO
06.          DFS-VISIT(v)
07. cor[u] ← PRETO
08. f[u] ← tempo ← (tempo + 1)
```

Pilha de execução:

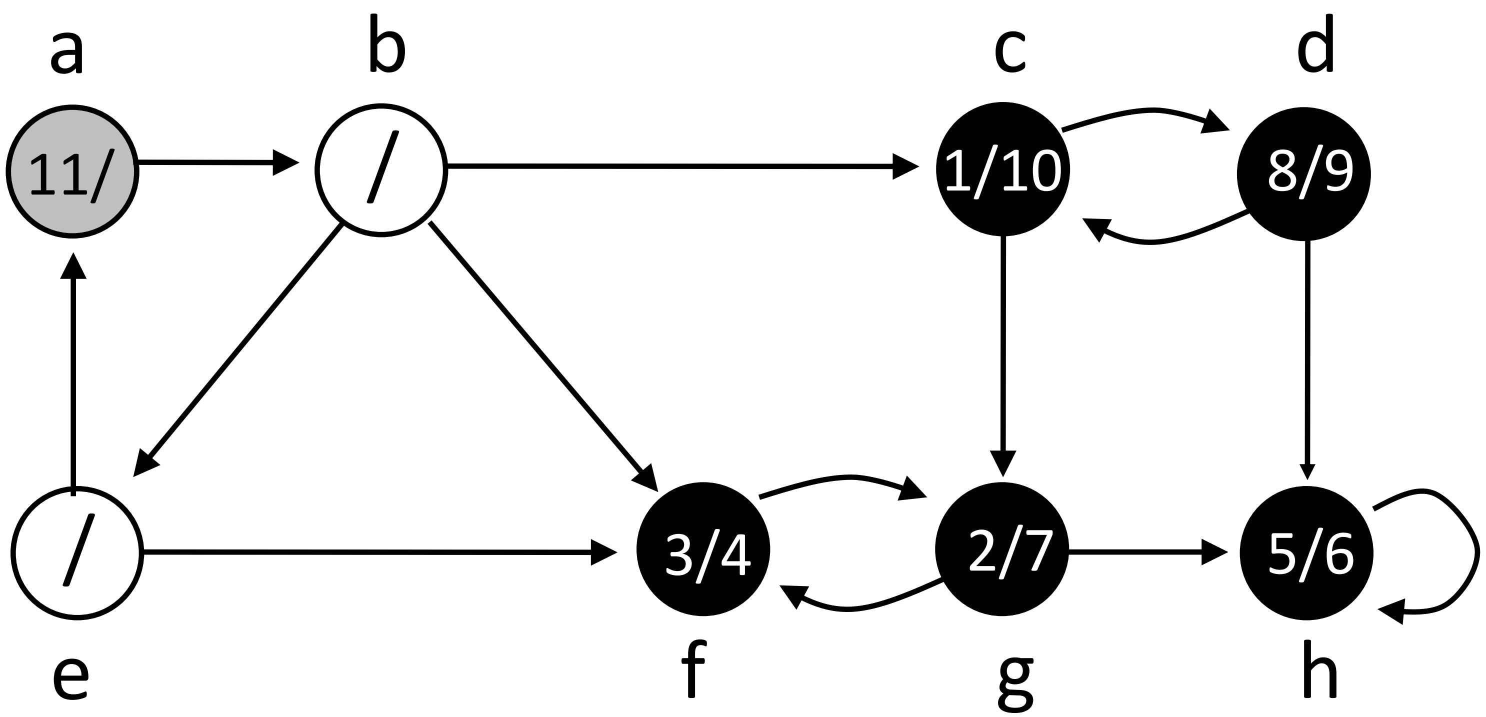
DFS_VISIT(g), CP: linha 4

DFS_VISIT(c), CP: linha 4

DFS(G) – CP: linha 4 – próximo: u = a

tempo = 3 => 4

Busca em profundidade



Lista [c, a, b, d, e, f, g, h]

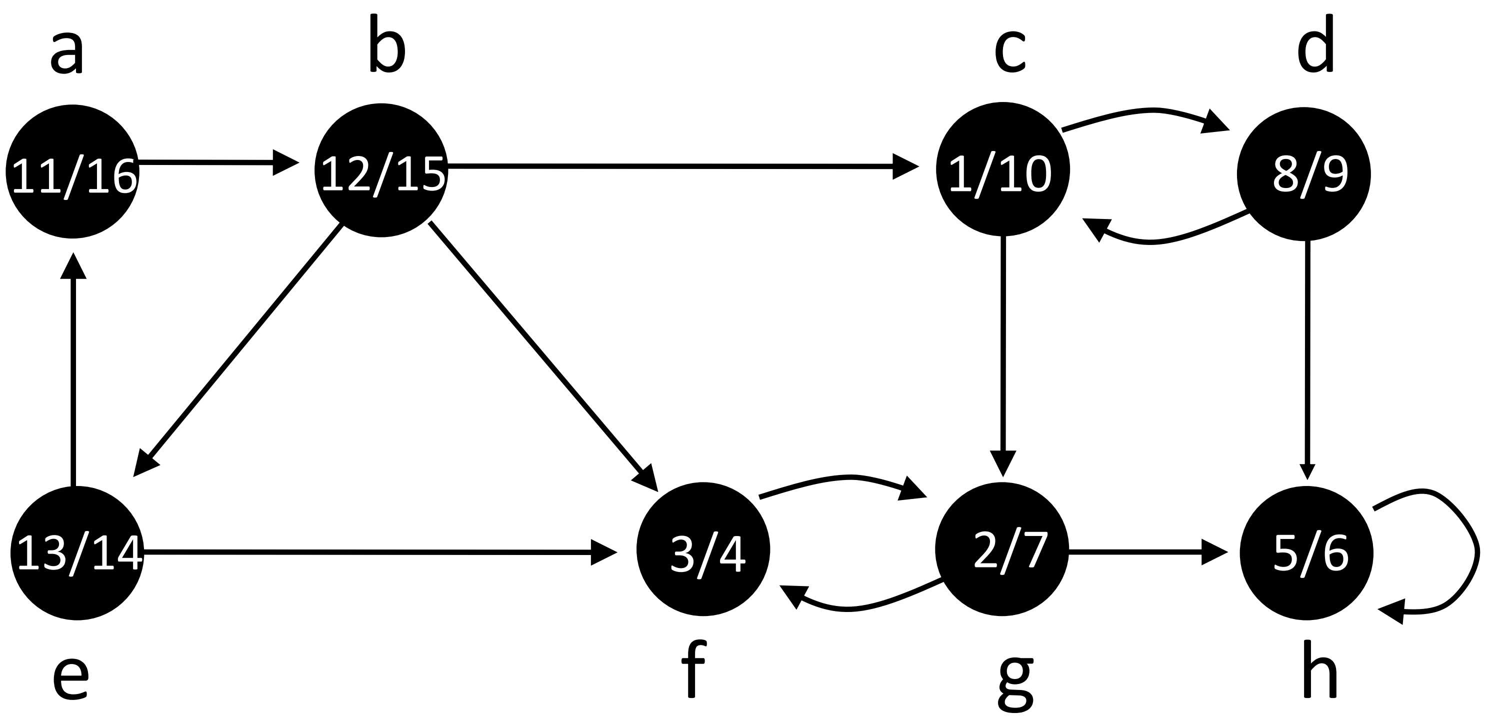
```
DFS-VISIT(u)
01. cor[u] ← CINZA
03. tempo ← tempo + 1
03. d[u] ← tempo
04. para cada vértice v ∈ Adj(u)
05.      se cor [v] = BRANCO
06.          DFS-VISIT(v)
07. cor[u] ← PRETO
08. f[u] ← tempo ← (tempo + 1)
```

Pilha de execução:

DFS(G) – CP: linha 4 – próximo: u = b

tempo = 11

Busca em profundidade



Lista [c, a, b, d, e, f, g, h]

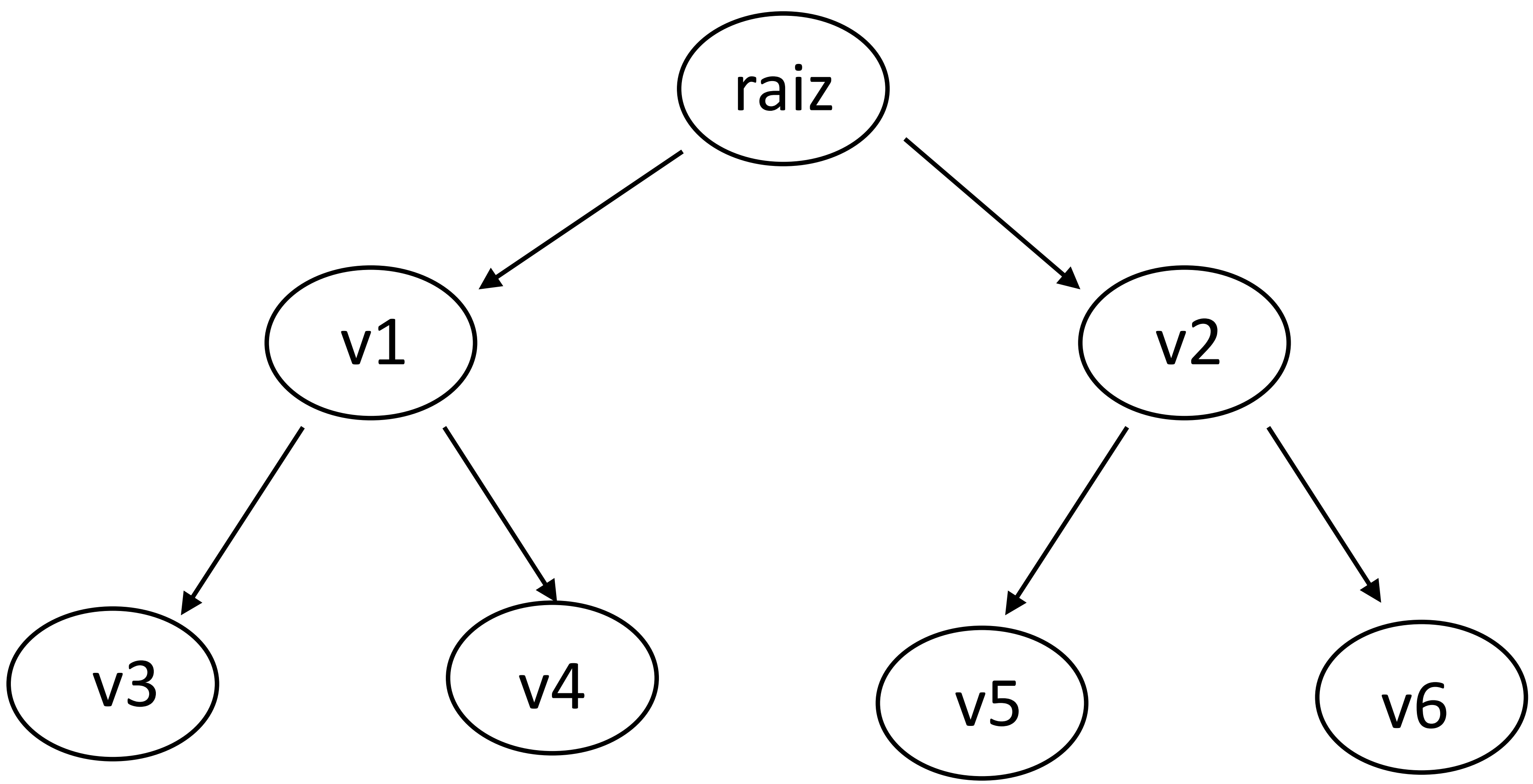
```
DFS-VISIT(u)
01. cor[u] ← CINZA
03. tempo ← tempo + 1
03. d[u] ← tempo
04. para cada vértice v ∈ Adj(u)
05.      se cor [v] = BRANCO
06.          DFS-VISIT(v)
07. cor[u] ← PRETO
08. f[u] ← tempo ← (tempo + 1)
```

Pilha de execução:
Vazia

tempo = 16

Busca em profundidade

Aplicando busca em profundidade em uma árvore



Busca em profundidade

ATENÇÃO!

A aplicação da DFS em grafos direcionados é essencialmente igual à aplicação em grafos não direcionados

No entanto, mesmo o grafo direcionado sendo conexo, a DFS pode precisar ser chamada repetidas vezes enquanto houver vértices não explorados, retornando uma floresta

Este é o mesmo caso quando a DFS é aplicada a um grafo desconexo

Complexidade

Para cada vértice do grafo, a DFS percorre todos os seus vizinhos. Cada aresta é visitada duas vezes

Se representarmos o grafo por uma lista de adjacências, a DFS tem complexidade $O(n + m)$.

Busca em largura

Um dos algoritmos mais simples da área de grafos

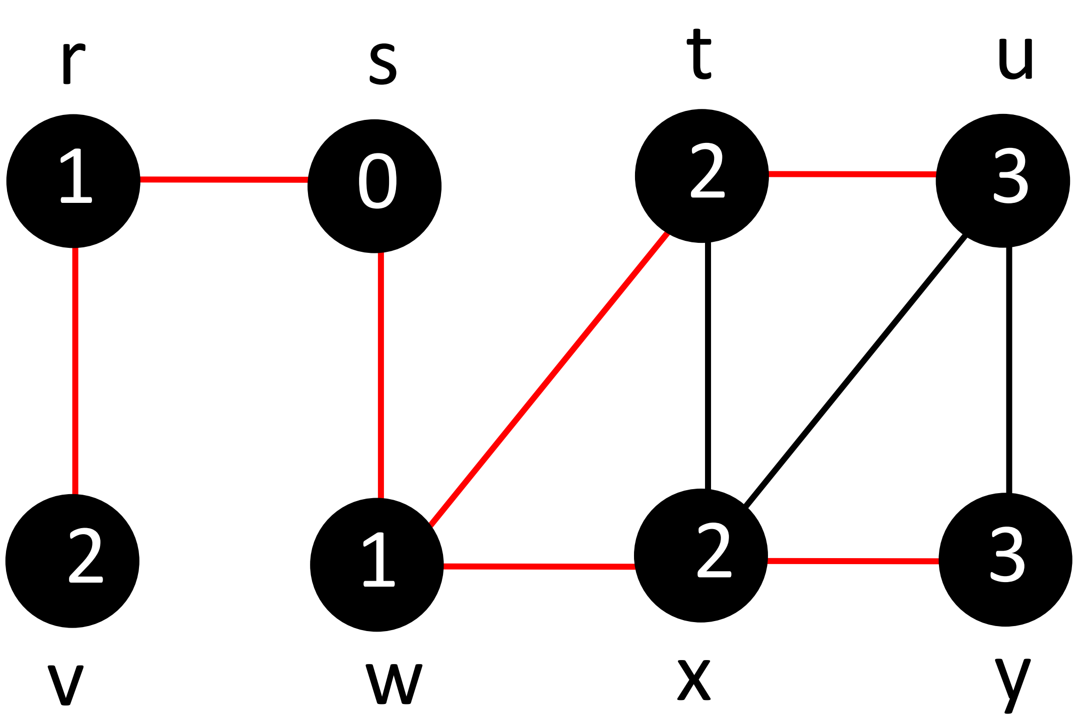
Serve de base para vários outros algoritmos:

- Base para Caminho mais curto (Dijkstra)
  - Utilizado para calcular rotas de custo mínimo em um par de localidades em um mapa, por exemplo
- Base para Árvore Geradora Mínima - AGM (Prim)
  - Utilizado para interligar localidades a um custo mínimo, por exemplo.

Busca em largura

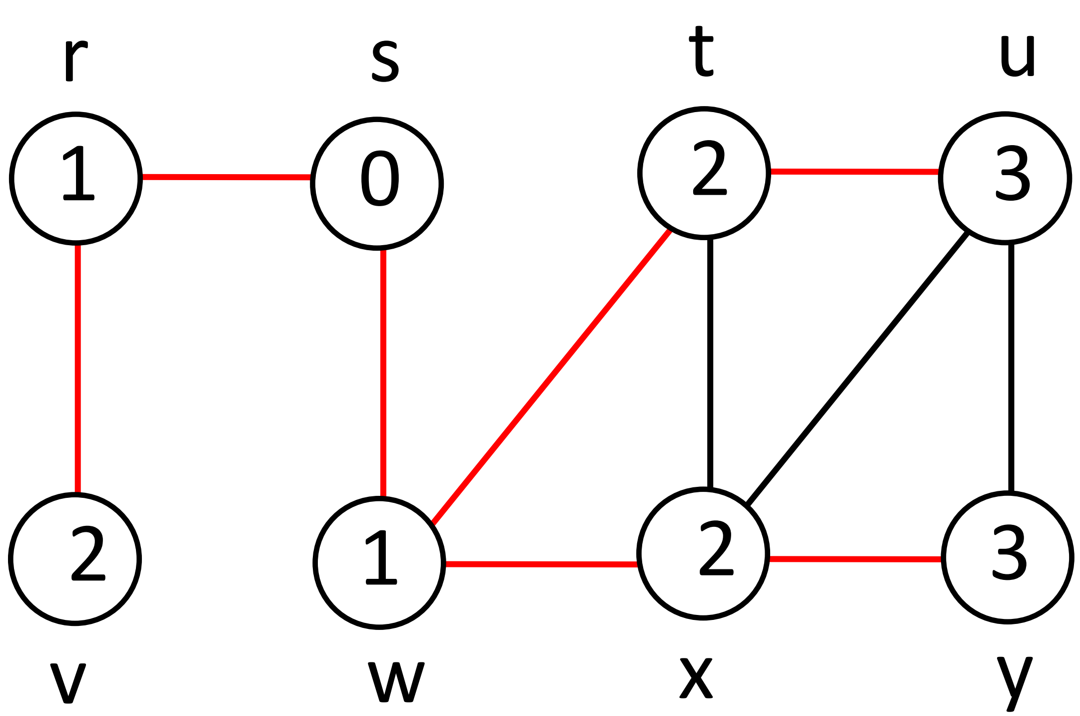
O algoritmo da Busca em Largura (do inglês *Breadth-First Search* -BFS) calcula a distância (menor número de arestas) desde o vértice s (raiz) até todos os vértices acessíveis

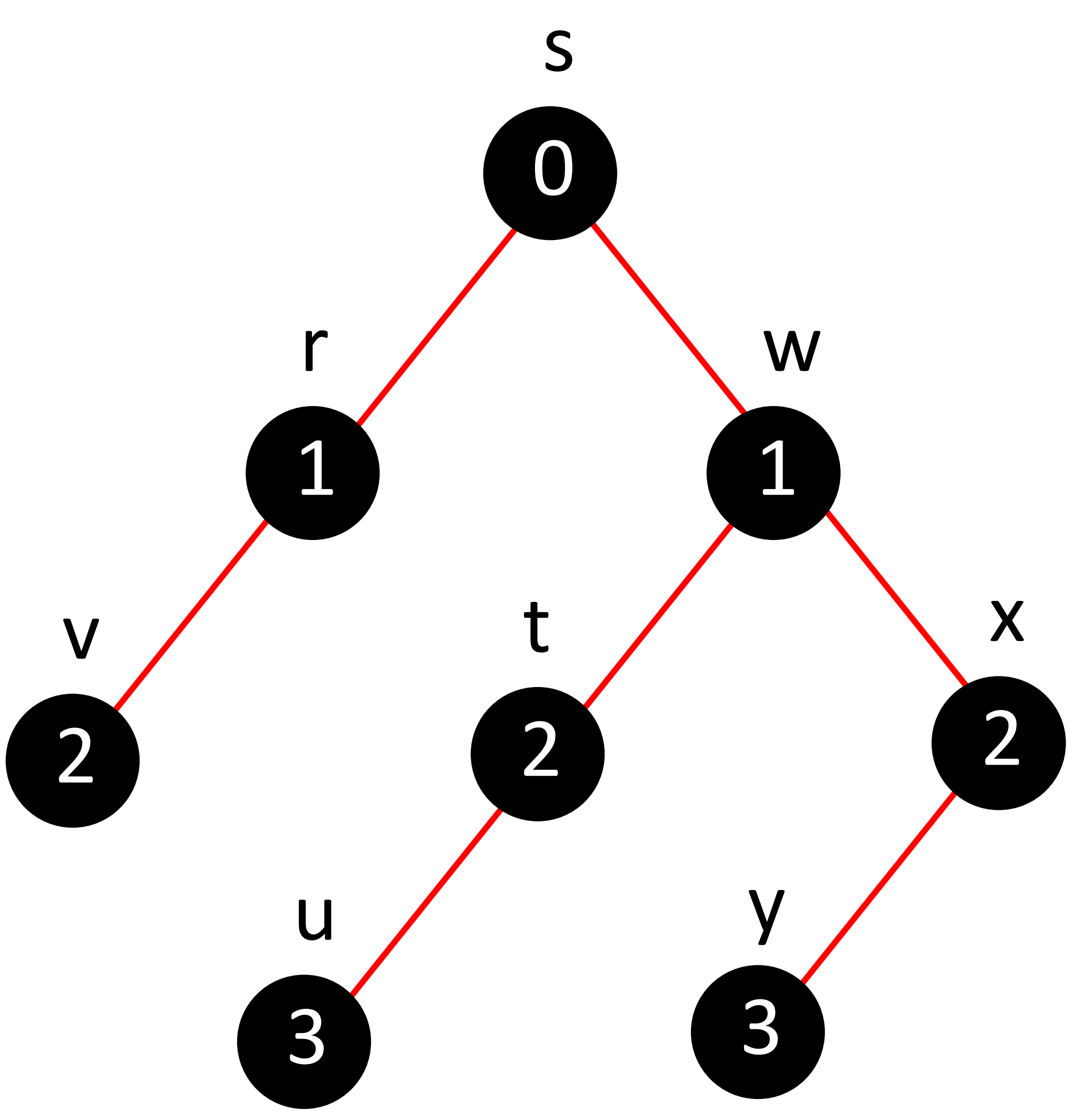
- Considera a quantidade de saltos necessários mínimos para alcançar outro vértice do grafo



Busca em largura

Ele também produz uma "Árvore Primeiro na Extensão", com raiz em no vértice de partida, que contém todos os vértices acessíveis



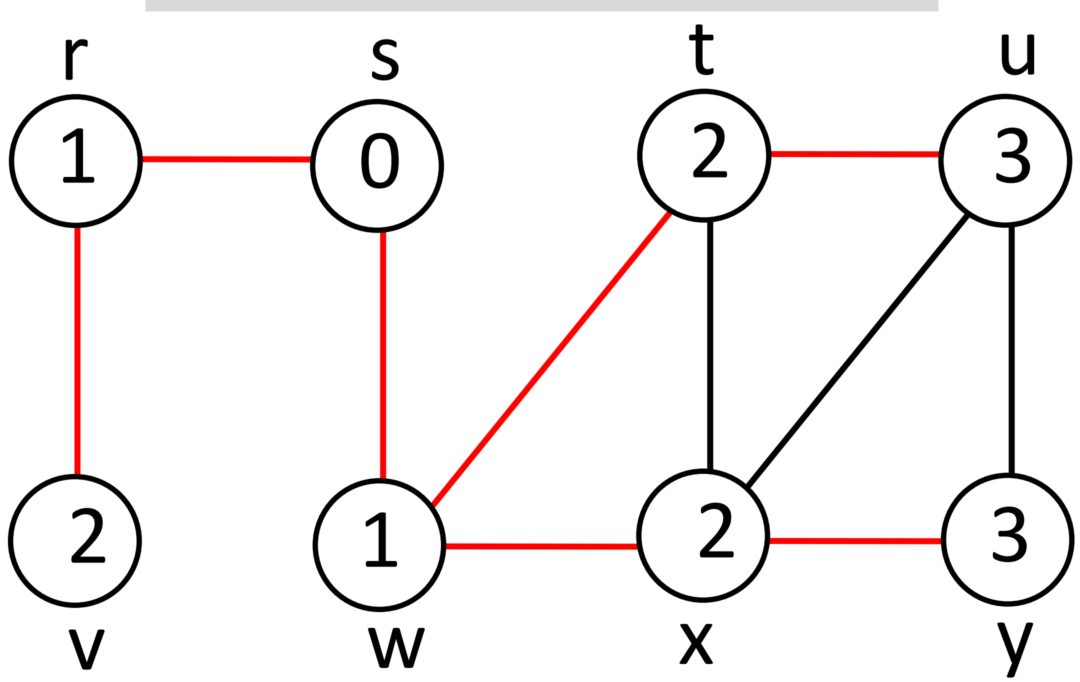


Busca em largura

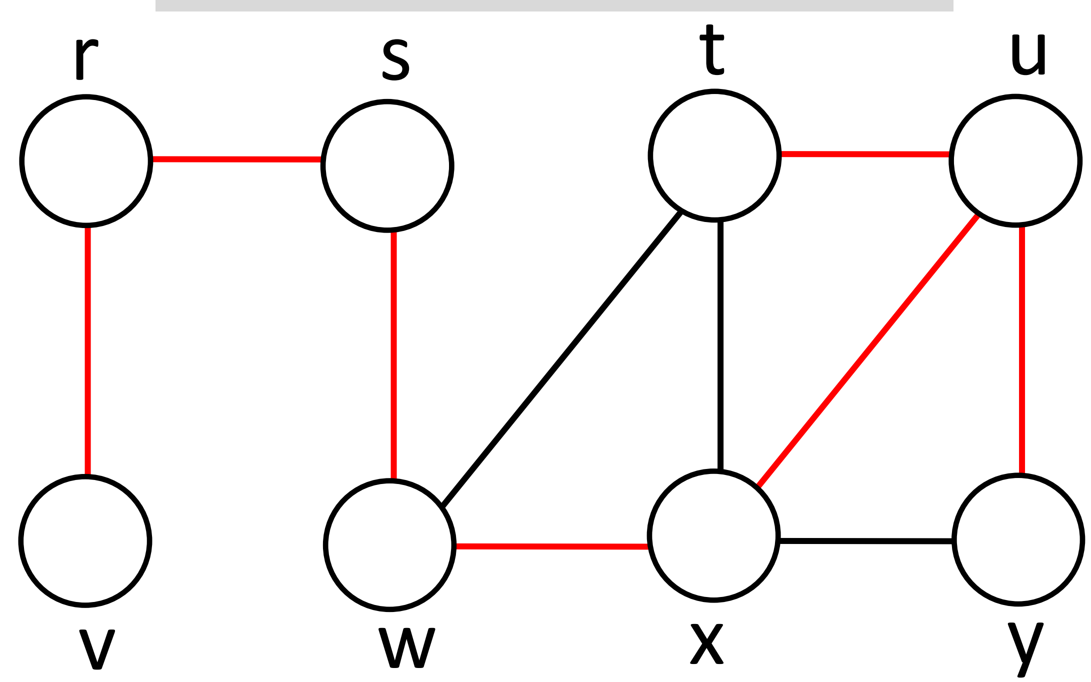
Para cada vértice v acessível a partir de s, o caminho na árvore primeiro na extensão de s até v corresponde a um "caminho mais curto" de s até v, ou seja, um caminho que contém um número mínimo de arestas

- Só é possível porque a busca é "guiada de nível em nível"

na BFS => (s,t) = 2



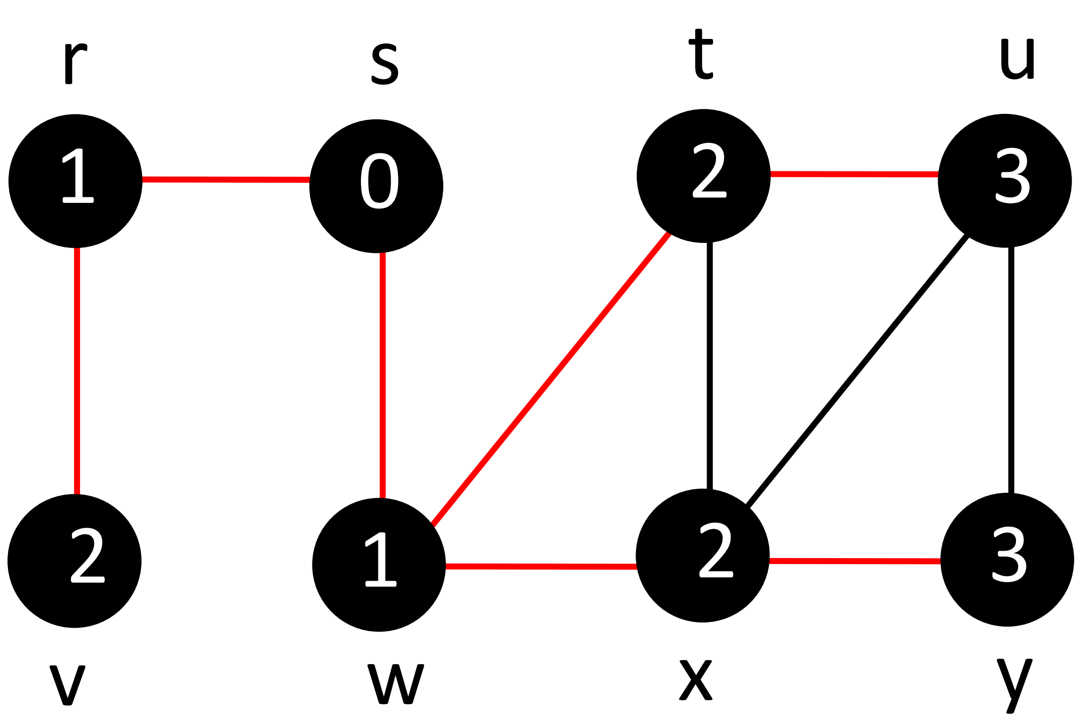
na DFS => (s,t) = 4



Busca em largura

A busca em largura recebe esse nome porque expande a fronteira entre vértices descobertos e não descobertos uniformemente ao longo da extensão da fronteira

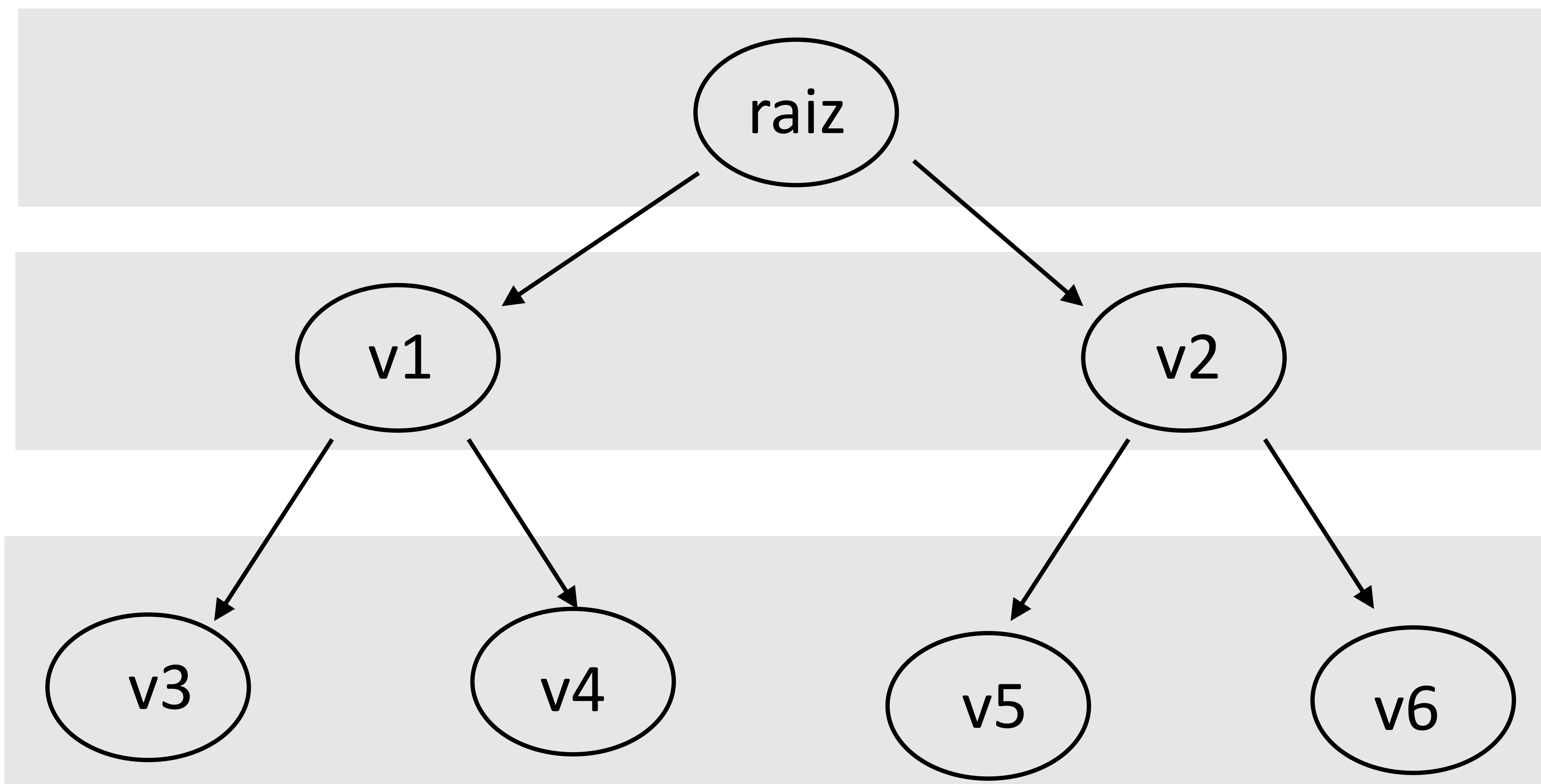
- Isto é, o algoritmo descobre todos os vértices à distância k a partir de s, antes de descobrir quaisquer vértices à distância k+1; (ponto chave)



Movimento da água

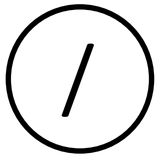
Busca em largura

Aplicando busca em largura em uma árvore

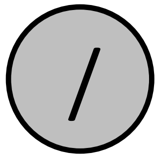


Busca em largura

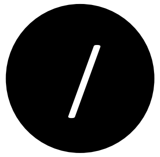
O controle do descobrimento dos nós na busca em largura é feito de forma semelhante ao controle utilizado na busca em profundidade

a 

Vértice desconhecido

b 

Vértice encontrado, seus adjacentes não foram inseridos em uma fila

c 

Vértice encontrado, todos os seus adjacentes foram inseridos na fila

Busca em largura

Ideia geral:

- Um vértice é descoberto na primeira vez em que é encontrado
- Neste momento ele se torna não branco
- Assim como na DFS, os vértices de cor cinza e preta distinguem os vértices já localizados em duas categorias
- Vértices de cor cinza podem ter alguns vértices adjacentes brancos; Eles representam a fronteira entre vértices descobertos e não descobertos
- A Busca em largura constrói uma árvore primeiro na extensão, contendo inicialmente apenas sua raiz
- Sempre que um vértice v é descoberto no curso da varredura da lista de adjacências de um vértice u já descoberto, o vértice v e a aresta (u,v) são adicionados à árvore primeiro na extensão

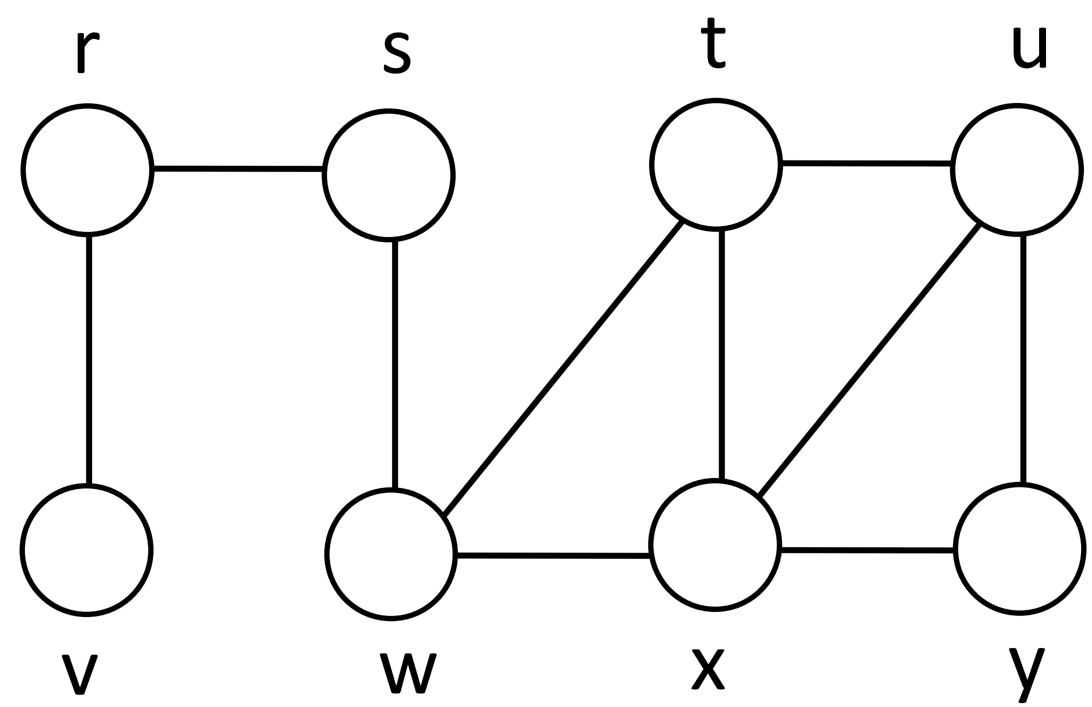
Busca em largura

Assim como na DFS, a BFS faz uso de algumas estruturas auxiliares durante a pesquisa:

- s //representa o vértice inicial
- cor[u] //indicativo de atingibilidade
- Π[u] //indica o vértice predecessor de u(pai)
- d[u] //indica a distância desde a origem d(s,u) - em arestas
- Q //indica a fila (FIFO) - ponto chave do algoritmo

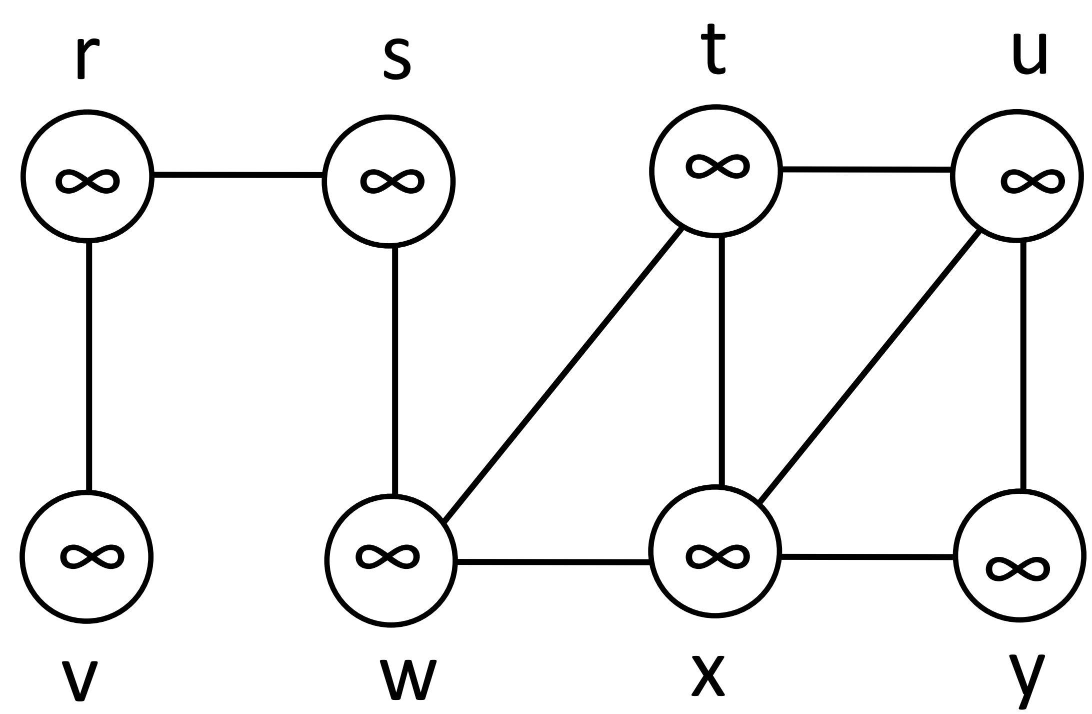
Busca em largura

```
BFS(G,s)
para cada v ∈ V faça
    cor(v) ← BRANCO;
    π(v) ← nil;
    d(v) ← ∞;
d(s) ← 0;
cor(s) ← CINZA;
Q ←∅;
INSERE(Q,s);
enquanto Q ≠ ∅ faça
    u ← REMOVE(Q);
    para cada v ∈ Adj(u) faça
       se cor(v)= BRANCO então
           INSERE(Q,v);
           cor(v) ← CINZA;
           π(v) ← u;
           d(v) ← d(u)+1;
    cor(u) ← PRETO
```



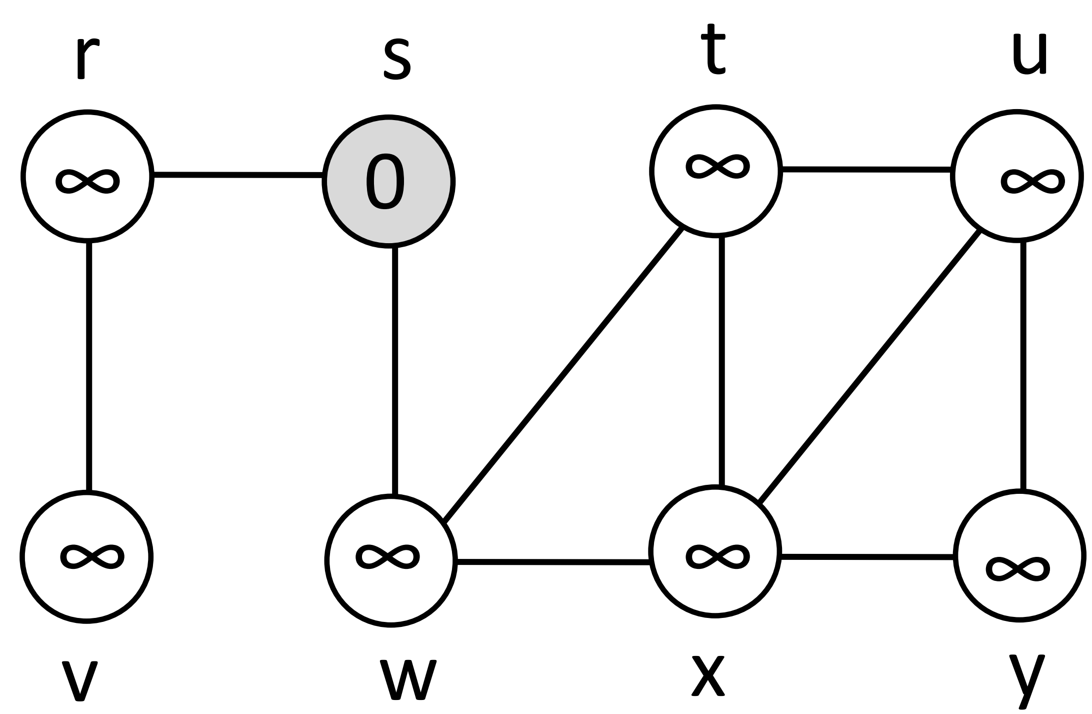
Busca em largura

```
BFS(G,s)
para cada v ∈ V faça
    cor(v) ← BRANCO;
    π(v) ← nil;
    d(v) ← ∞;
d(s) ← 0;
cor(s) ← CINZA;
Q ←Ø;
INSERE(Q,s);
enquanto Q ≠ Ø faça
    u ← REMOVE(Q);
    para cada v ∈ Adj(u) faça
        se cor(v)= BRANCO então
            INSERE(Q,v);
            cor(v) ← CINZA;
            π(v) ← u;
            d(v) ← d(u)+1;
    cor(u) ← PRETO
```



Busca em largura

```
BFS(G,s)
para cada v ∈ V faça
     cor(v) ← BRANCO;
     π(v) ← nil;
     d(v) ← ∞;
d(s) ← 0;
cor(s) ← CINZA;
Q ←Ø;
INSERE(Q,s);
enquanto Q ≠ Ø faça
     u ← REMOVE(Q);
     para cada v ∈ Adj(u) faça
         se cor(v)= BRANCO então
              INSERE(Q,v);
              cor(v) ← CINZA;
              π(v) ← u;
              d(v) ← d(u)+1;
     cor(u) ← PRETO
```

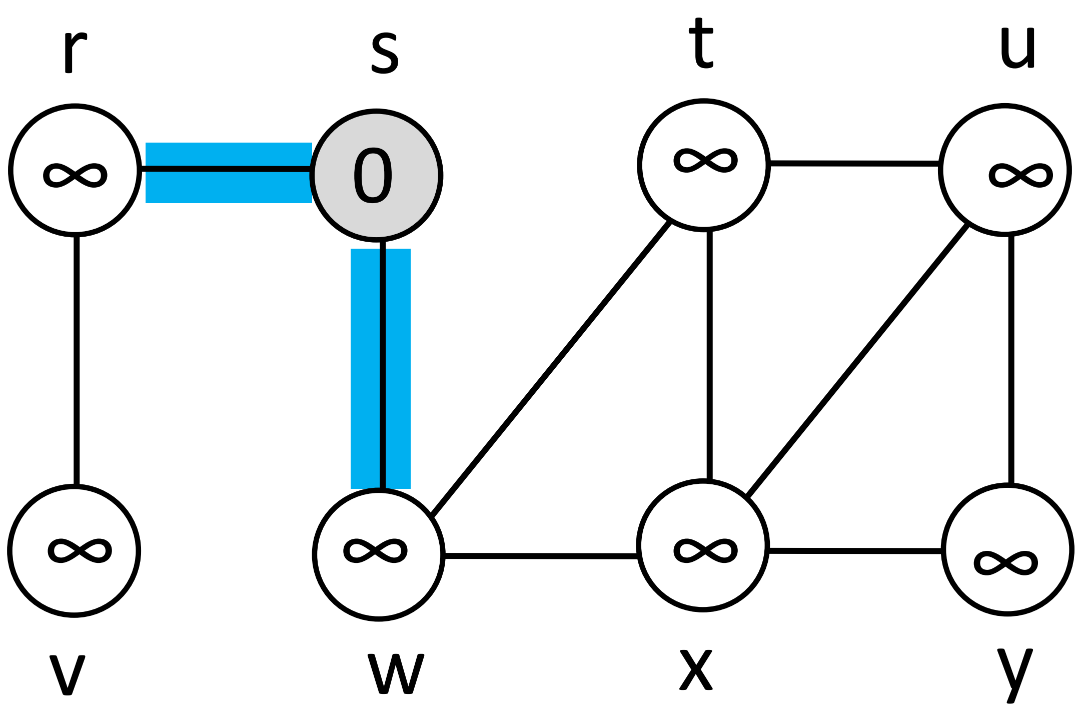


Q = S

0

Busca em largura

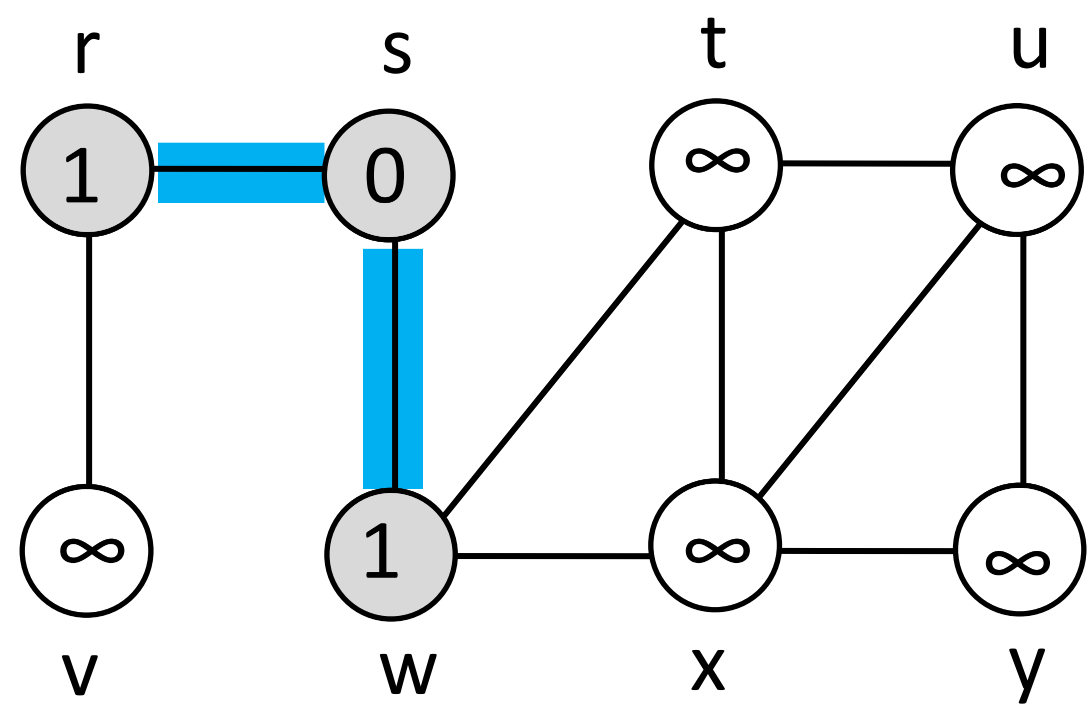
```
BFS(G,s)
para cada v ∈ V faça
     cor(v) ← BRANCO;
     π(v) ← nil;
     d(v) ← ∞;
d(s) ← 0;
cor(s) ← CINZA;
Q ←Ø;
INSERE(Q,s);
enquanto Q ≠ Ø faça
     u ← REMOVE(Q);
     para cada v ∈ Adj(u) faça
          se cor(v)= BRANCO então
               INSERE(Q,v);
               cor(v) ← CINZA;
               π(v) ← u;
               d(v) ← d(u)+1;
     cor(u) ← PRETO
```



Q =

Busca em largura

```
BFS(G,s)
para cada v ∈ V faça
    cor(v) ← BRANCO;
    π(v) ← nil;
    d(v) ← ∞;
d(s) ← 0;
cor(s) ← CINZA;
Q ←Ø;
INSERE(Q,s);
enquanto Q ≠ Ø faça
    u ← REMOVE(Q);
    para cada v ∈ Adj(u) faça
        se cor(v)= BRANCO então
            INSERE(Q,v);
            cor(v) ← CINZA;
            π(v) ← u;
            d(v) ← d(u)+1;
    cor(u) ← PRETO
```

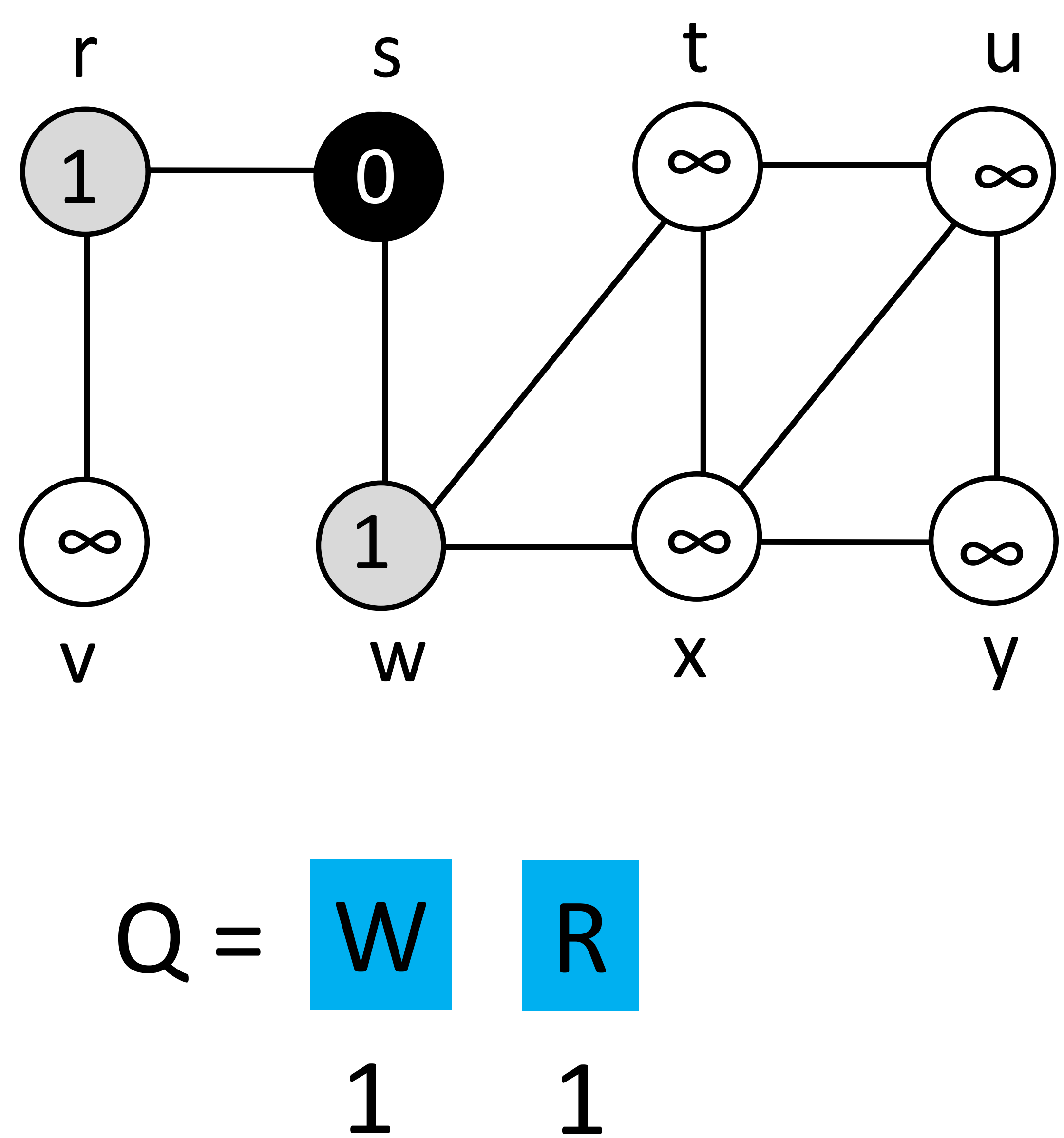


Q = W R

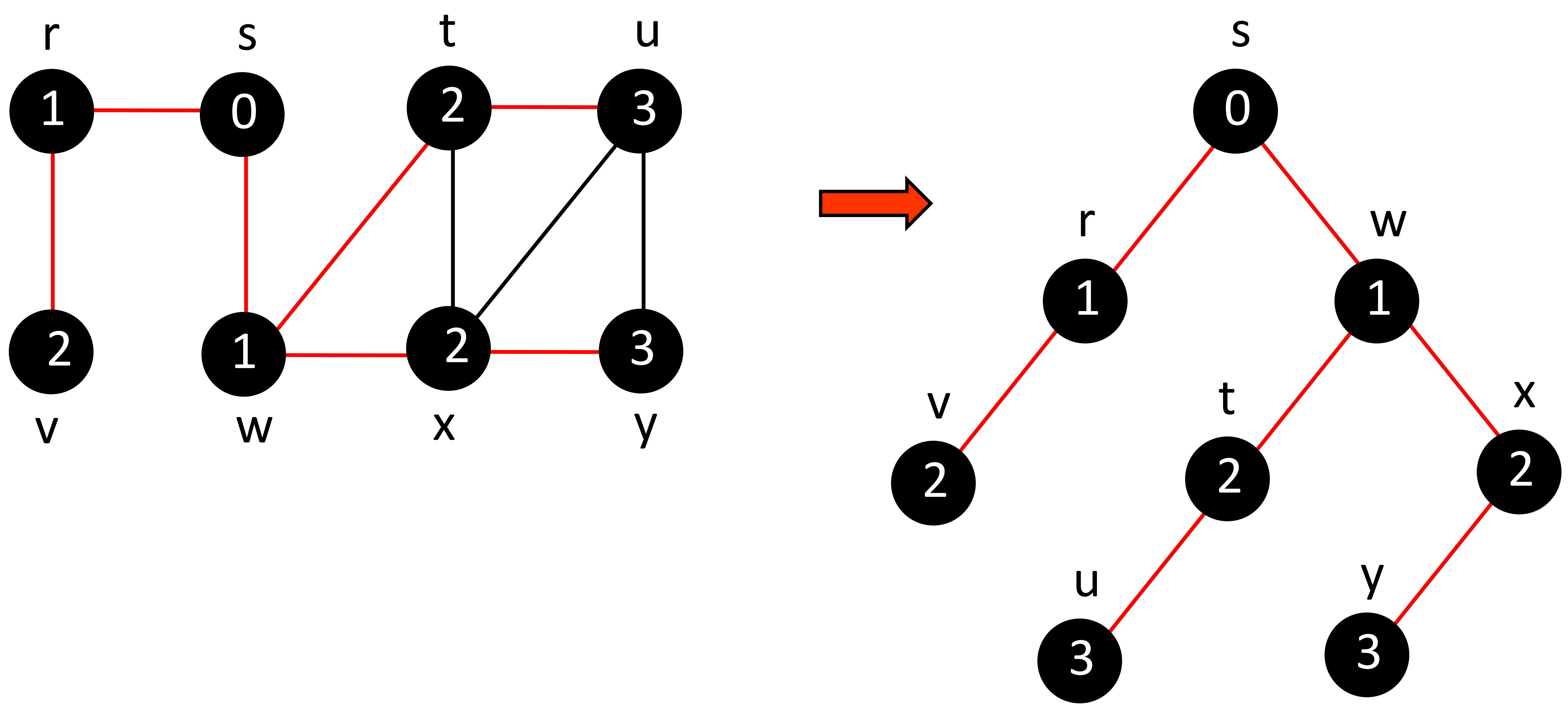
1 1

Busca em largura

```
BFS(G,s)
para cada v ∈ V faça
    cor(v) ← BRANCO;
    π(v) ← nil;
    d(v) ← ∞;
d(s) ← 0;
cor(s) ← CINZA;
Q ←Ø;
INSERE(Q,s);
enquanto Q ≠ Ø faça
    u ← REMOVE(Q);
    para cada v ∈ Adj(u) faça
       se cor(v)= BRANCO então
           INSERE(Q,v);
           cor(v) ← CINZA;
           π(v) ← u;
           d(v) ← d(u)+1;
    cor(u) ← PRETO
```



Busca em largura



Vetor Π

Vértice	S	R	W	V	T	X	U	Y
Π	nil	S	S	R	W	W	T	X

Busca em largura

Complexidade

Cada vértice só entra na fila uma vez

Inserir e remover na fila possuem complexidade constante, realizadas $|V|$ vezes cada;

A lista de adjacências de cada vértice é examinada apenas uma vez, e a soma dos comprimentos de todas as listas é $\Theta(m)$

Logo, se representarmos o grafo por uma lista de adjacências, a BFS tem complexidade $O(n + m)$

DFS x BFS

Busca em profundidade

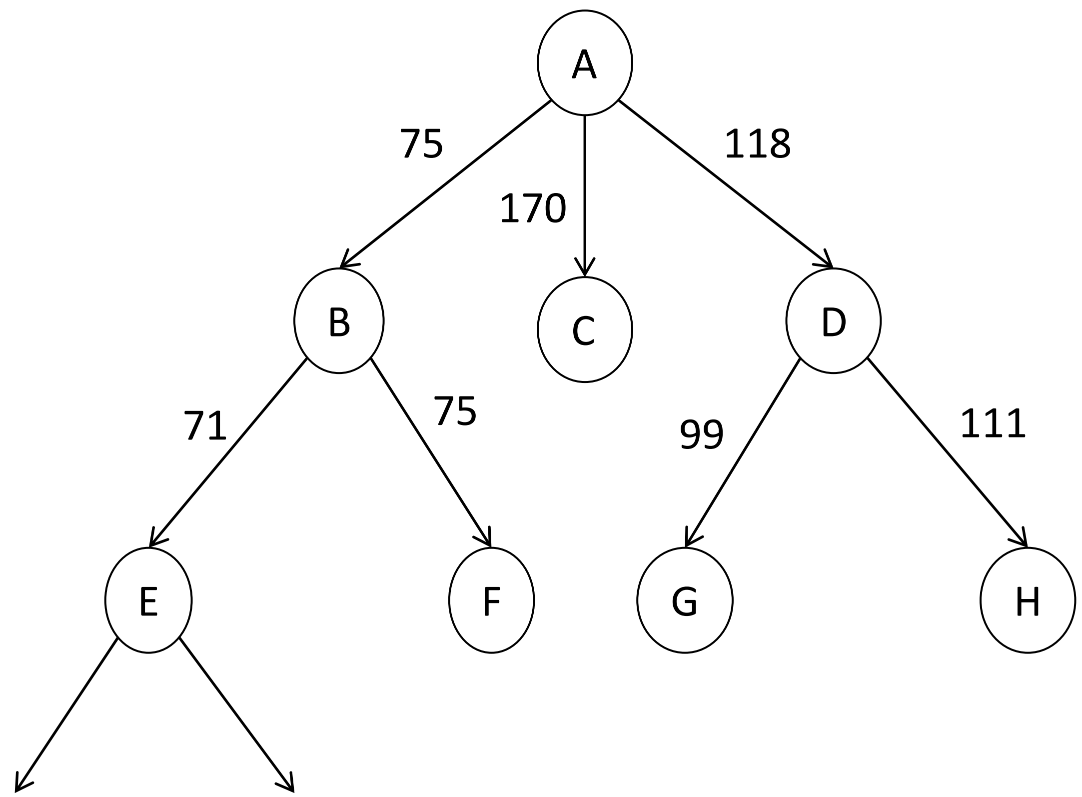
- Incursões **profundas** no grafo, voltando somente quando não existem mais vértices desconhecidos pela frente
- Com reinício: visita todos os vértices
- Marca o vértice antes de visitar toda sua vizinhança
- Uso de pilha

Busca em largura

- Busca progride em “largura”: certifica-se de que vizinhos próximos sejam visitados primeiro
- Sem reinício: interessam apenas os vértices alcançáveis a partir de *v*
- Marca o vértice depois de visitar toda sua vizinhança
- Uso de fila

Busca de Custo Uniforme

- **Estratégia:**
  - Expande sempre o nó de menor custo de caminho. Se o custo de todos os passos for o mesmo, o algoritmo acaba sendo o mesmo que a busca em largura.

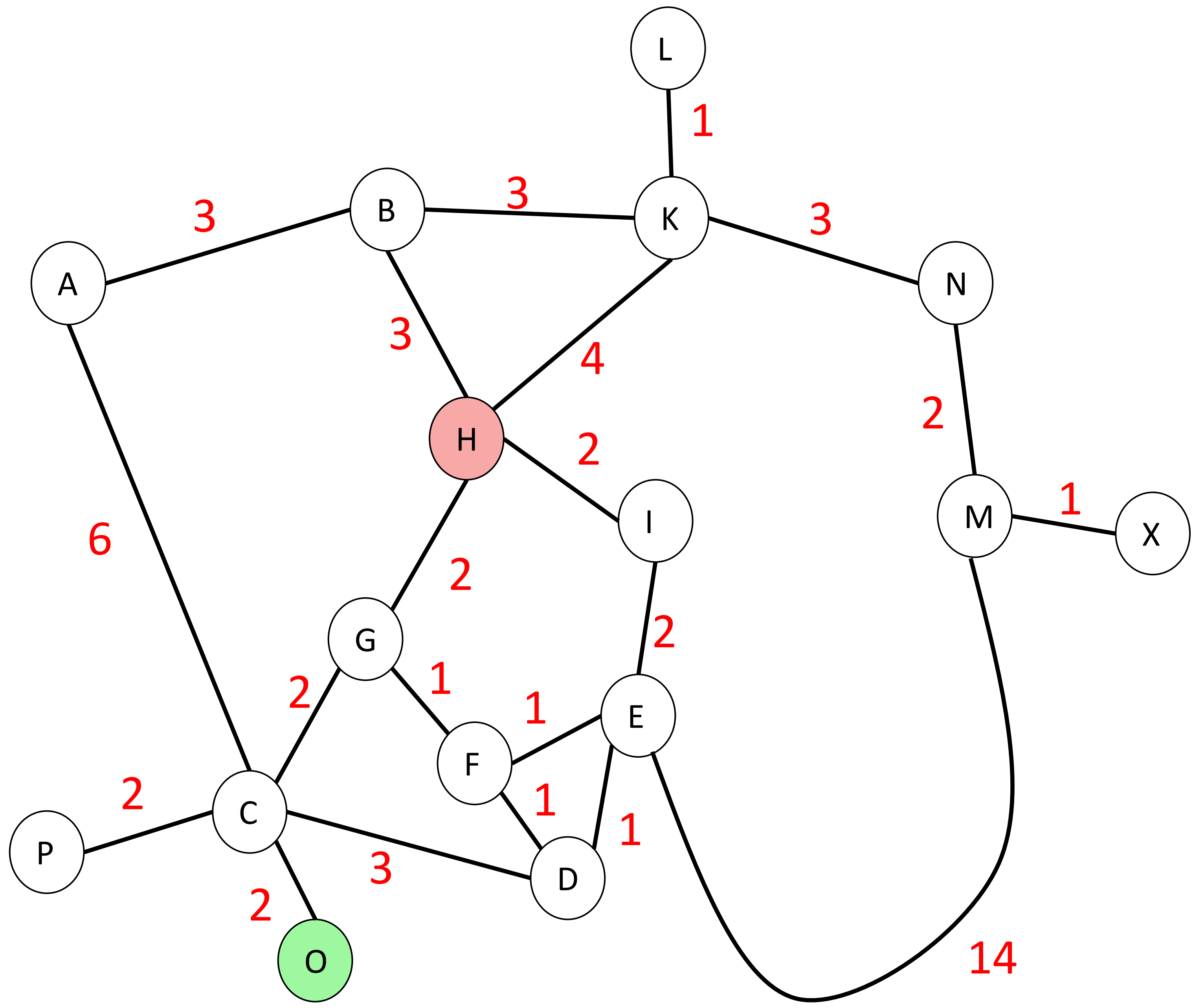


Busca de Custo Uniforme

```
Algoritmo Busca - Uniforme
1. Definir um conjunto L de nós iniciais
2. Ordene L em ordem crescente de custo
3. Se L é vazio Então
      Busca não foi bem sucedida
   Senão
      seja n o primeiro nó de L;
4. Se n é um nó objetivo Então
      Retornar caminho do nó inicial até N;
      Parar
   Senão
      Remover n de L;
      Adicionar em L todos os nós filhos de n, rotulando
           cada nó com o seu caminho até o nó inicial;
      Ordene L em ordem crescente de custo;
      Volte ao passo 3.
```

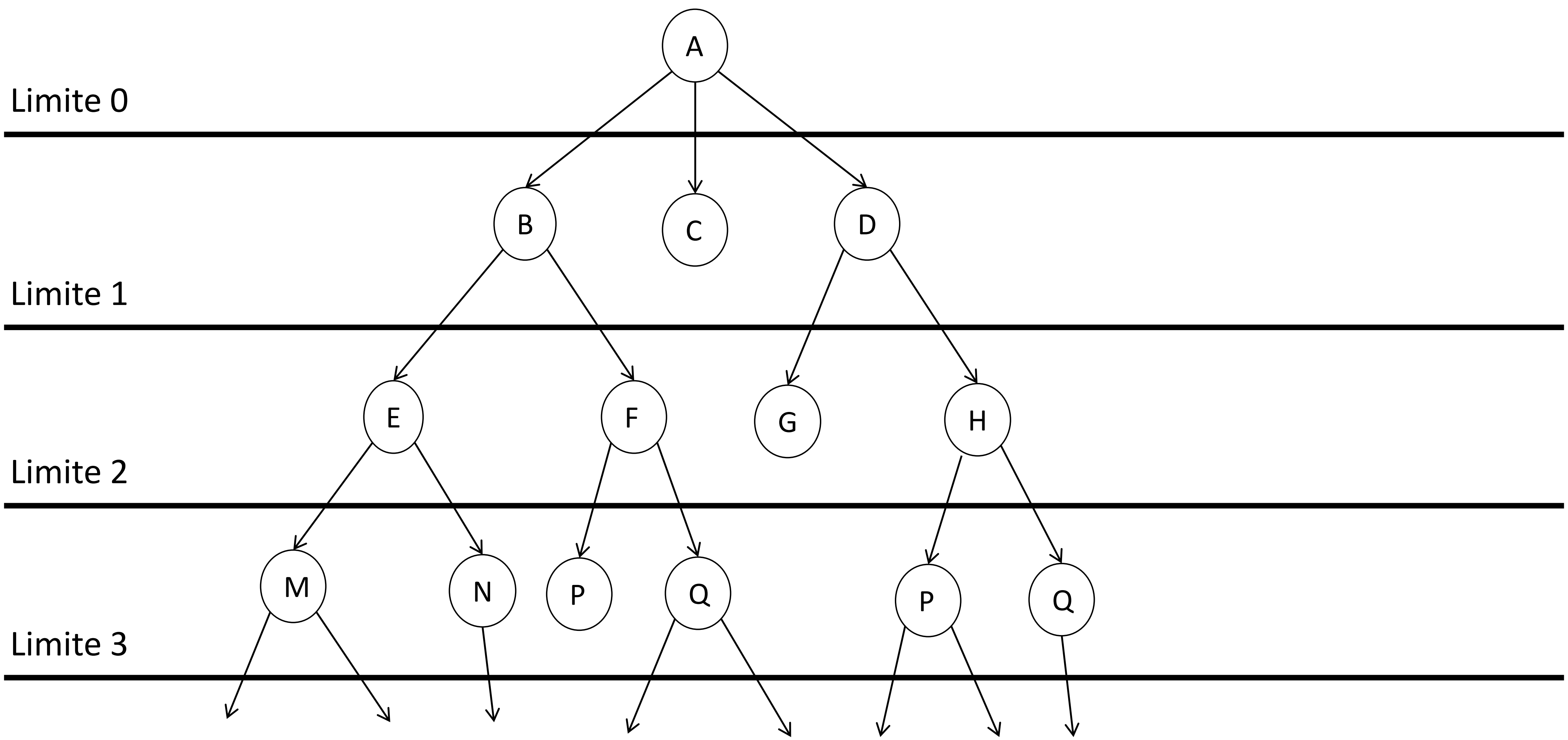
Exercício

Sair de “H” e chegar em “O”



Busca com Aprofundamento Iterativo

- **Estratégia:** Consiste em uma busca em profundidade onde o limite de profundidade é incrementado gradualmente.



Busca com Aprofundamento Iterativo

- Combina os benefícios da busca em largura com os benefícios da busca em profundidade.
- Evita o problema de caminhos muito longos ou infinitos.
- A repetição da expansão de estados não é tão ruim, pois a maior parte dos estados está nos níveis mais baixos.
- Cria menos estados que a busca em largura e consome menos memória

Busca com Aprofundamento Iterativo

- Esta estratégia tenta limites com valores crescentes, partindo de zero, até encontrar a primeira solução
- fixa profundidade = i, executa busca
- se não chegou a um objetivo, recomeça busca com profundidade = i+ n (n qualquer)
- piora o tempo de busca, porém melhora o custo de memória!
- igual à busca em Largura para i=l e n=l