ESTADOS E BUSCAS I

Prof. Dr. Igor da Penha Natal

Resolução de problemas por busca

- Muitas vezes não recebemos um algoritmo para resolver um problema, mas apenas uma especificação do que é uma solução — então temos de procurar por uma solução.

- Um problema típico ocorre quando o agente está em um estado inicial e quer chegar a um estado objetivo, tendo um conjunto de ações determinísticas que ele pode realizar.

- Muitos problemas de IA podem ser abstraídos para o problema de encontrar um caminho em um grafo dirigido.

- Muitas vezes há mais de uma maneira de representar um problema como grafo.

Tarefas para a resolução de problemas por busca

- **Formulação**
  - do OBJETIVO
    - Identificar o que se quer atingir?
  - do PROBLEMA
    - Decidir quais ações e estados considerar.
- **Busca**
  - Dada várias sequências de ações, qual é a melhor?
- **Execução**

Formulação → Busca →Execução

Estrutura básica de um agente resolvedor de problemas por Busca

```
def agente_resolvedor_problemas_simples(percepção):

    entrada: percepção # uma percepção

    local: sequencia # uma sequência de ações, inicialmente vazia
           estado # uma descrição do estado atual do mundo
           objetivo # um objetivo, inicialmente nulo
           problema # uma formulação de problema


    estado ← ATUALIZAR_ESTADO(estado, percepção)

    se sequencia está vazia:
          objetivo  ← FORMULAR_OBJETIVO(estado)
          problema  ← FORMULAR_PROBLEMA(estado, objetivo)
          sequencia ← BUSCA(problema)

    ação ← PRIMEIRO(sequencia)
    sequencia ← RESTO(sequencia)

    retornar ação
```

Espaço de estados

- Um estado contém toda a informação necessária para predizer os efeitos de uma ação e determinar se ele é um estado objetivo.

- O nível de abstração, ou nível de detalhe, que modelamos o mundo interfere na resolução do problema.
  - Um nível muito fino e perderemos a visão do problemas global.
  - Um nível muito grosso e perderemos detalhes críticos para resolver o problema.

- O número de estados depende da representação e do nível de abstração escolhidos.

Espaço de estados

- O estado inicial e uma função de ação definem implicitamente o espaço de estados, o qual é o conjunto de todos os estados acessíveis a partir do estado inicial.

- O espaço de estados forma um grafo (nem sempre explícito) em que os nós são estados e os arcos são as ações

- Um caminho no espaço de estados é uma sequência de estados conectados por uma sequência de ações.

Problemas no espaço de estados

- Consistem de:
  - Um conjunto de estados;
  - Um conjunto distinto de estado chamado de estados iniciais;
  - Um conjunto de ações disponíveis para o agente em cada estado;
  - Uma função de ação que dado um estado e uma ação retorna um novo estado;
  - Um conjunto de estados objetivos, sempre especificado por uma função boolena, ***objetivo*(*s*)**, que é verdadeira quando **s** é um estado objetivo;
  - Um critério que especifica a qualidade de uma solução aceitável.

Tipo de problemas

- Problemas com um **único estado**:
  - São problemas em que é possível calcular exatamente que estados estaremos após uma sequência de ações.
- Problemas com **múltiplos estados**:
  - São problemas em que sabemos os efeitos das ações, mas temos acesso limitado ao estado no qual se encontra o ambiente.
- Problemas de **contingência**:
  - São problemas em que as ações realizadas pelo agente podem resultar em efeitos não esperados (**não determinístico**). Não há como fazer uma árvore de busca que garanta o resultado.
- Problemas de **exploração**:
  - São problemas em que o agente não tem informações sobre o efeitos de suas ações. Ele sabe somente sobre o que foi aprendido anteriormente pela exploração do problema.

Características dos tipo de problemas

- Problemas de **estado único:**
  - O agente sabe em que estado está.
  - O agente sabe o que cada uma das suas ações fazem.
  - O agente poderá calcular exatamente o que fazer.

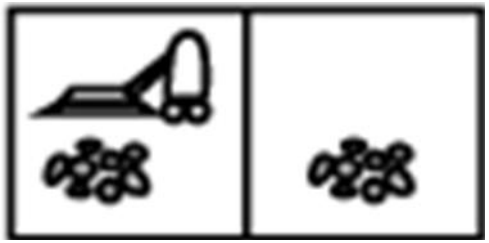
- Problemas de **múltiplos estados:**
  - O agente sabe o que cada uma das suas ações fazem.

- Problemas de **contingência:**
  - O agente deve usar os sensores enquanto age.
  - A solução é uma árvore de ações ou uma política.
  - Às vezes pode ser viável o entrelaçamento (agir antes de buscar).

- Problemas de **exploração:**
  - O espaço de estados é desconhecido.

Exemplo: Mundo do aspirador

Estado inicial possível



Estado final

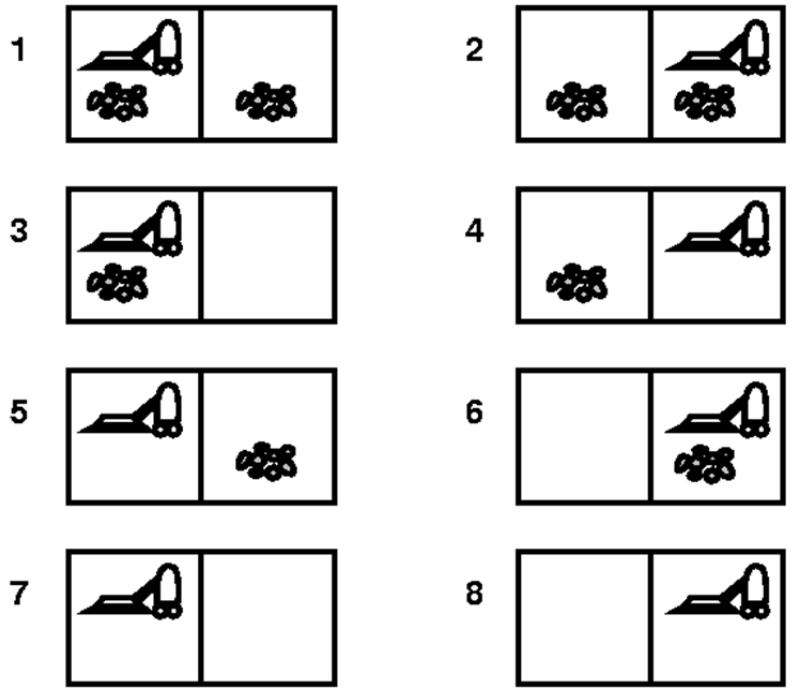
- Estado é representado por um par:

  <local do robô; sujidade do local>

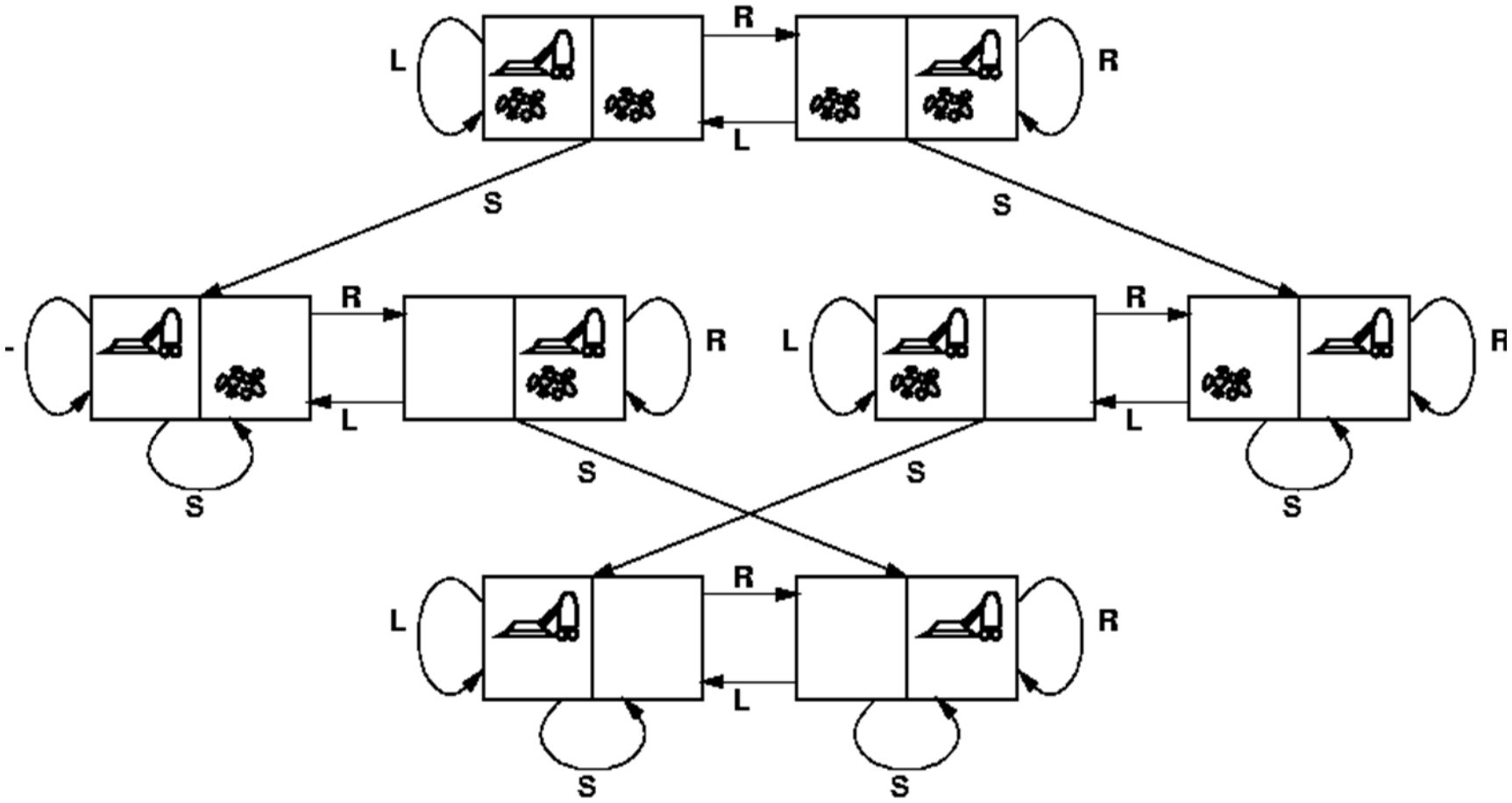
- Número de estados = 2.2^2 → k. 2^k onde k é o número de locais.
  - **Ex.: 5x5 = 5.2^{25} = 167.772.160** **10x10 = $100.2^{100} \approx 1.27x10^{32}$**

Exemplo de tipos de problemas: Mundo do aspirador

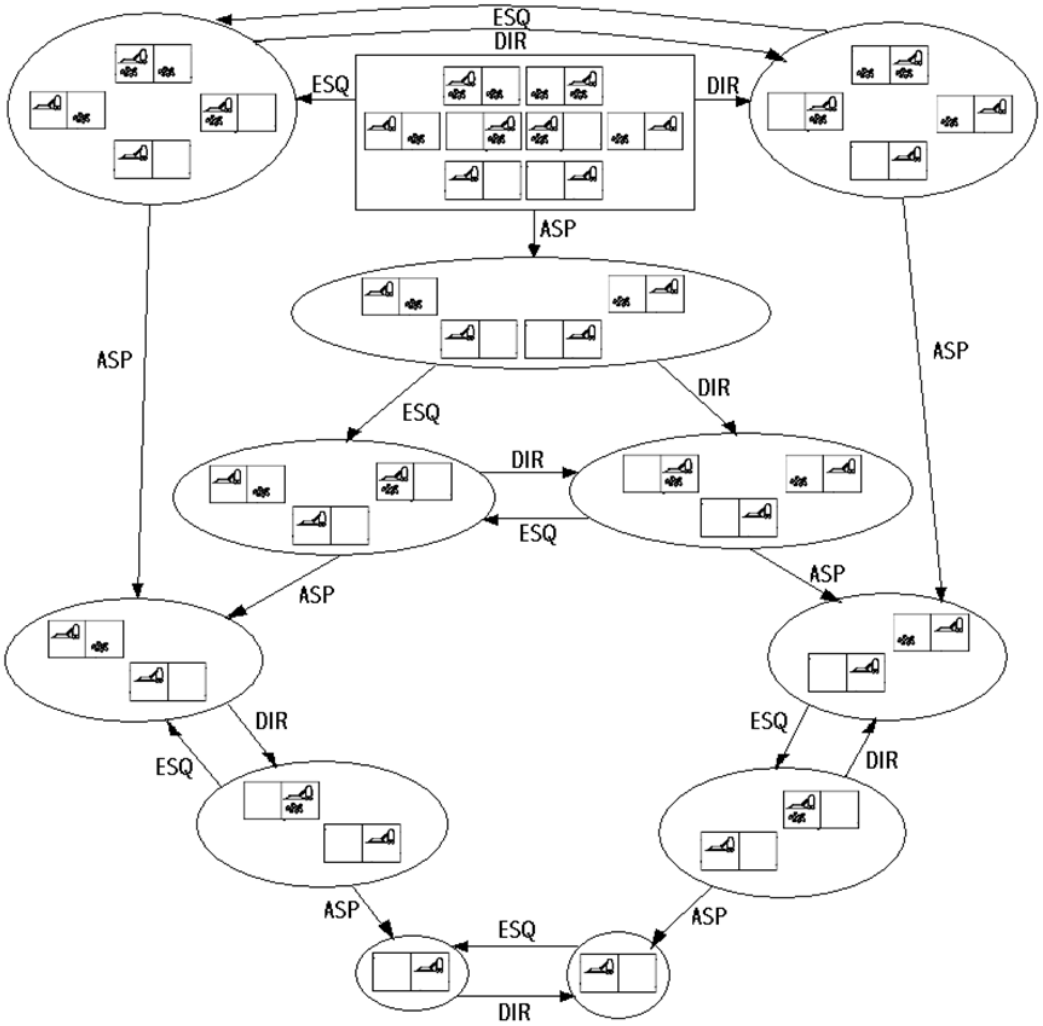
- **Estado único:**
  - Início no estado 5.
- **Múltiplos estados:**
  - Início em
    - {1, 2, 3, 4, 5, 6, 7, 8)
  - Ação → Direita
    - {2, 4, 6, 8}
- **Contingência:**
  - Ação → Aspirar
    - Pode sujar um carpete limpo.
  - Sensoriamento
    - Local
    - Sujo ou não



Espaço de estados em problemas de estado único



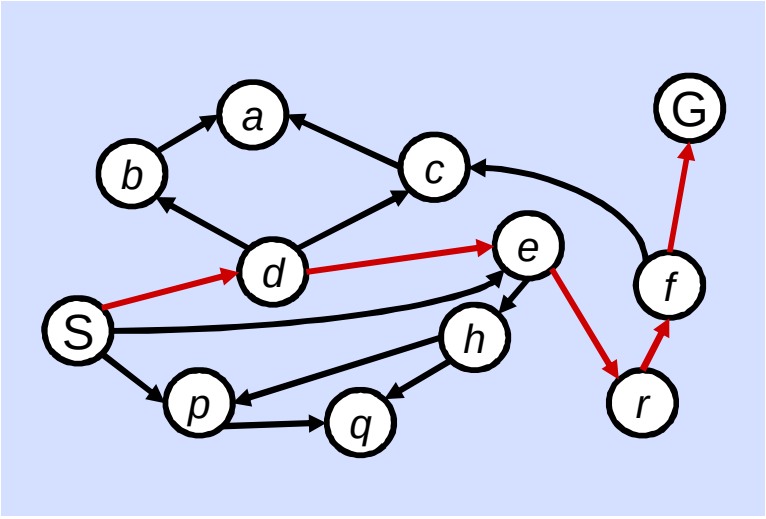
Espaço de estado em problemas de múltiplos estados



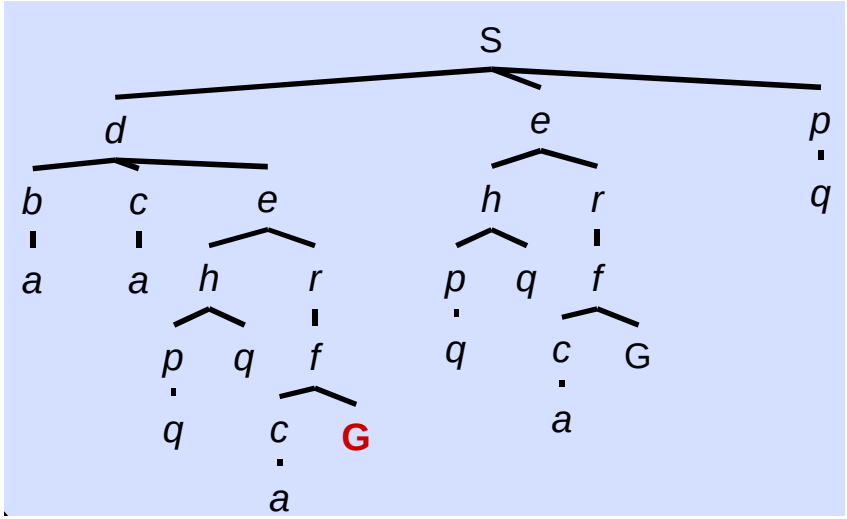
Grafos de estados vs. Árvores de busca

- Cada nó na árvore de busca é um caminho completo no grafo do problema.
- Nós construímos ambos sobre demanda – e construímos somente o necessário.

Grafo de estados

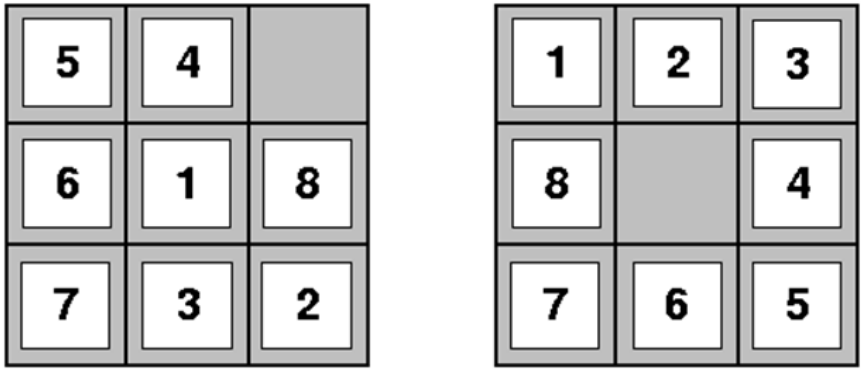


Árvore de busca



Exemplo: 8-*puzzle*

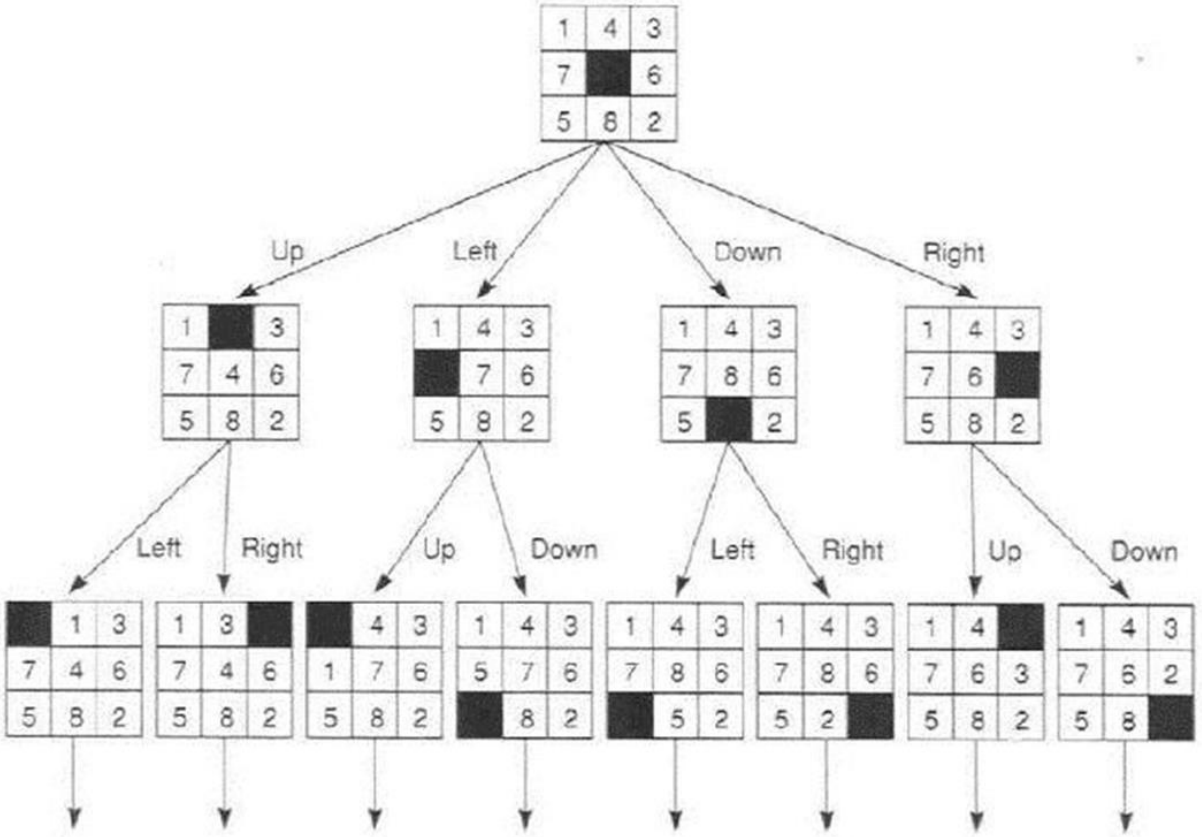
- **Estados**: matriz 3x3 com a especificação da posição de cada símbolo de 1-8 e o espaço em branco.
- **Estado Inicial**: qualquer um dos estados acima especificados.
- **Função de ação**: gera os estados válidos resultantes das quatro ações de mover o espaço vazio para *direita*, *esquerda*, *cima*, *baixo*.
- **Teste de objetivo**: verifica se o estado corresponde ao estado objetivo.



estado inicial estado final

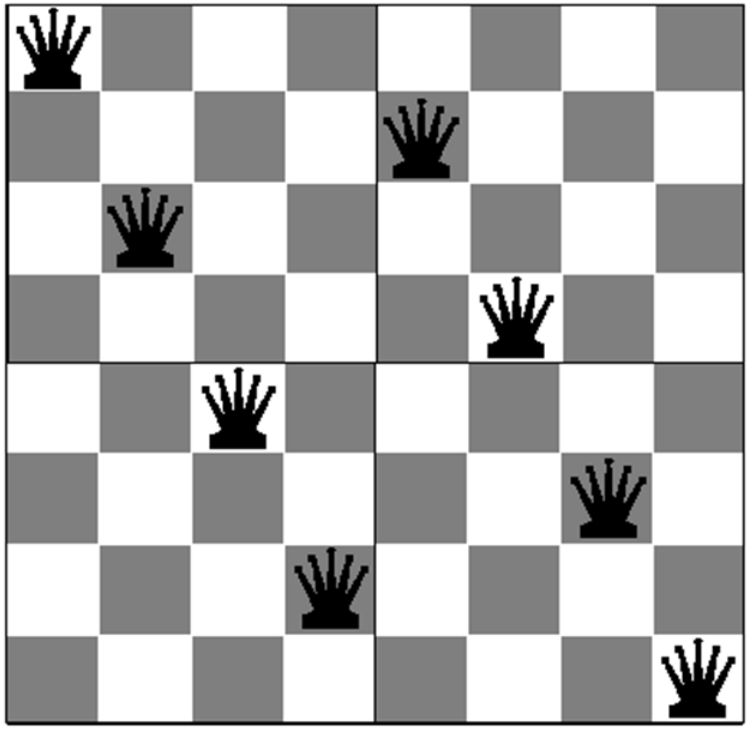
Exemplo: 8-*puzzle* – Espaço de estados

- Número de estados = 9! = 362.880



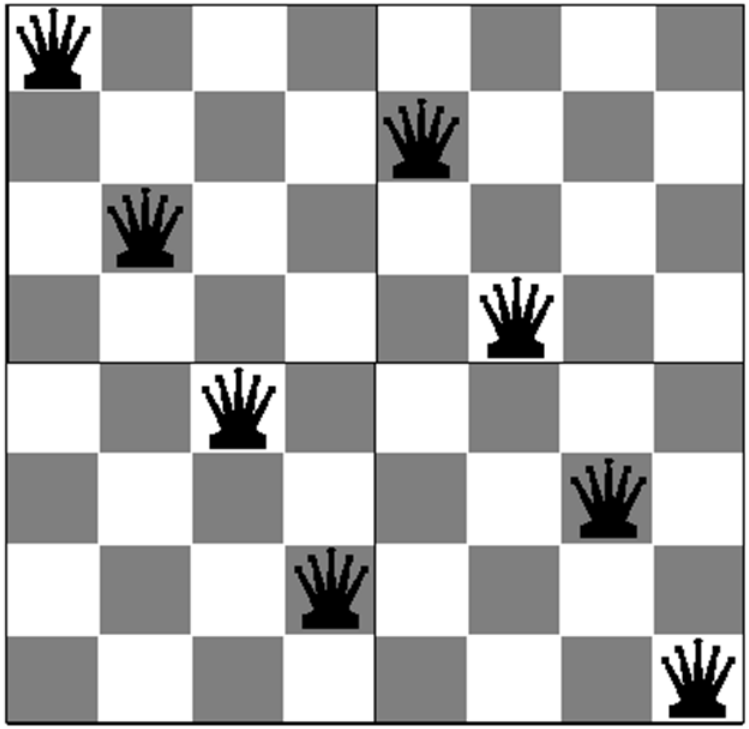
Exemplo: 8 Rainhas

- Estados: qualquer arranjo de 0 a 8 rainhas no tabuleiro.
- Estado inicial: nenhuma rainha no tabuleiro.
- Função de ação: colocar uma rainha em uma casa.
- Teste de Objetivo: 8 rainhas estão no tabuleiro e nenhuma está sendo atacada.
- $3x10^{14}$ possíveis sequências a serem investigadas.



Exemplo: 8 Rainhas – Outra formulação

- Estados: 0 a 8 rainhas no tabuleiro, uma por coluna nas n colunas mais a esquerda e sem ataques.
- Estado inicial: nenhuma rainha no tabuleiro.
- Função de ação: colocar uma rainha em uma casa cuja coluna seja a primeira à esquerda que não esteja ameaçada por outra rainha.
- 2057 possíveis sequências a serem investigadas.



A formulação do problema é **FUNDAMENTAL!!!**

Busca no espaço de estados

- A busca no espaço de estados assume que:
  - O agente tem um conhecimento perfeito do espaço de estados e pode observar em que estado ele está.
  - O agente tem um conjunto de ações que tem efeitos determinísticos conhecidos.
  - Alguns estados são estados objetivo e o agente quer atingir um destes estados, podendo reconhecer um estado objetivo.
  - Uma solução é uma sequência de ações que levam o agente do seu estado atual para um estado objetivo.

Busca sistemática em grafos

- Existem muitas formas de representar um problema como um grafo, mas duas formas merecem atenção:

- Grafo do Espaço de Estados: no qual um nó representa um estado do mundo e um arco representa uma mudança de um estado para outro no mundo.
  - Ex.: problema das 8 rainhas – mudar uma rainha de linha gera um novo estado.

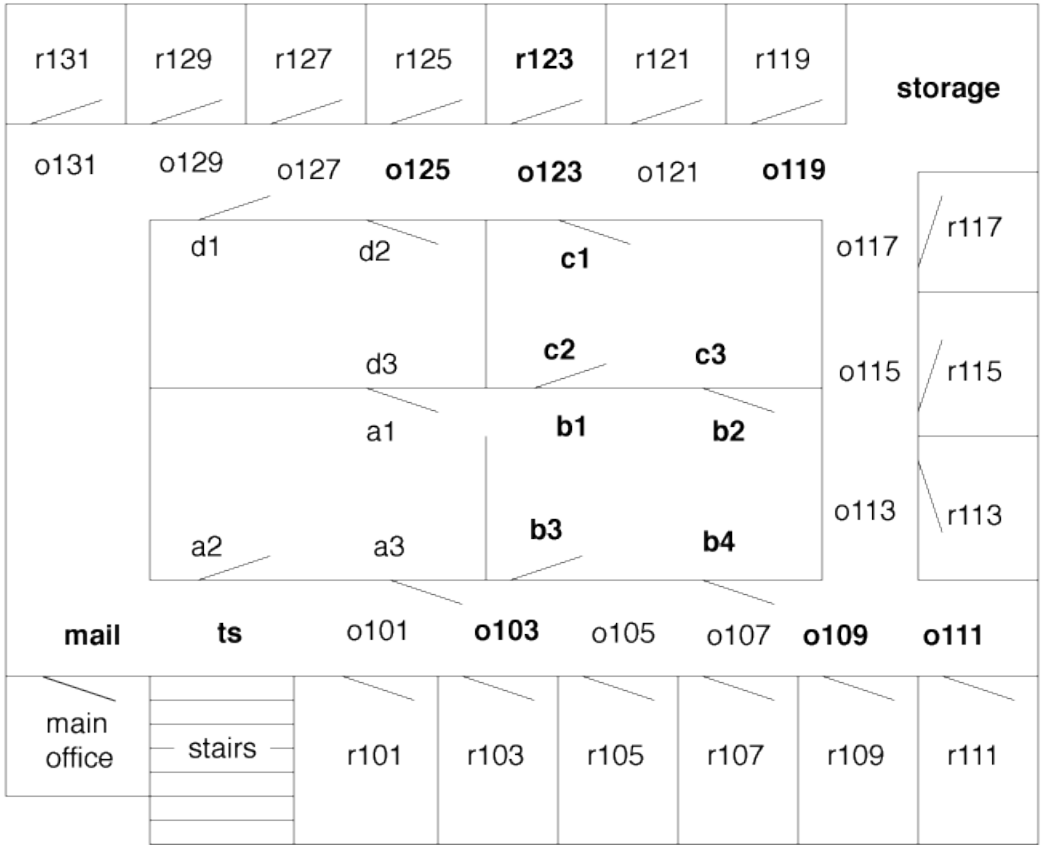
- Grafo do Espaço de Problemas: no qual um nó representa um problema a ser resolvido e um arco representa decomposições alternativas do problemas.
  - Ex.: problema das 8 rainhas – colocar uma rainha na n-ésima coluna, sem gerar conflito com as outras que já estão no tabuleiro, gera um novo estado.

Grafos dirigidos

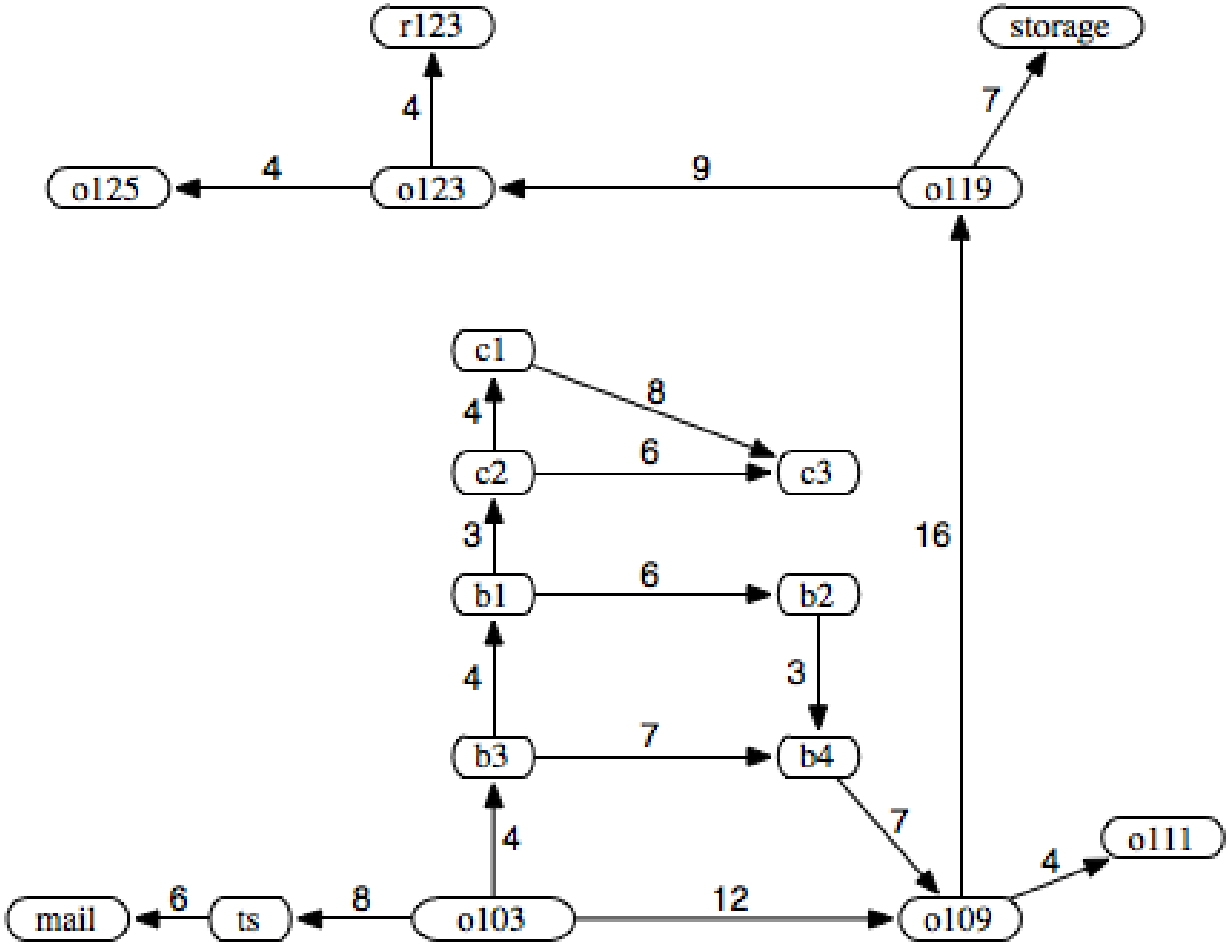
- Um grafo consiste em um conjunto **N** de nós e um conjunto **A** de pares ordenados de nós $<n_x, n_y>$, chamados arcos.
- O nó n_2 é um vizinho de n_1 se houver um arco de n_1 para n_2. Ou seja, se $\langle n_1, n_2 \rangle \in A$.
- Um caminho é uma sequência de nós $\langle n_0, n_1, ..., n_K \rangle$ que tal que $\langle n_{i-1}, n_i \rangle \in A$.
- Tendo em conta um conjunto de nós de início e de nós objetivo, uma solução é um caminho de um nó de início para um nó objetivo.
- Muitas vezes há um custo associado com os arcos e o custo de um caminho é a soma dos custos dos arcos no caminho.

Exemplo: O robô de entrega

- O robô quer ir de fora do quarto 103 para dentro do quarto 123.

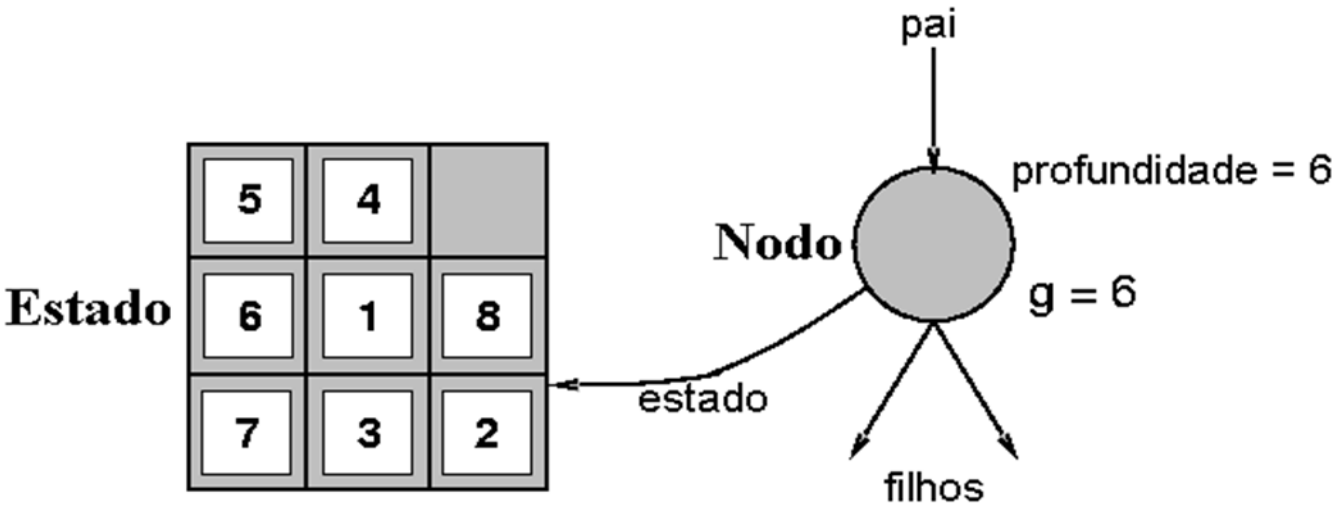


Exemplo: Grafo para o robô de entrega



Estados vs. Nós

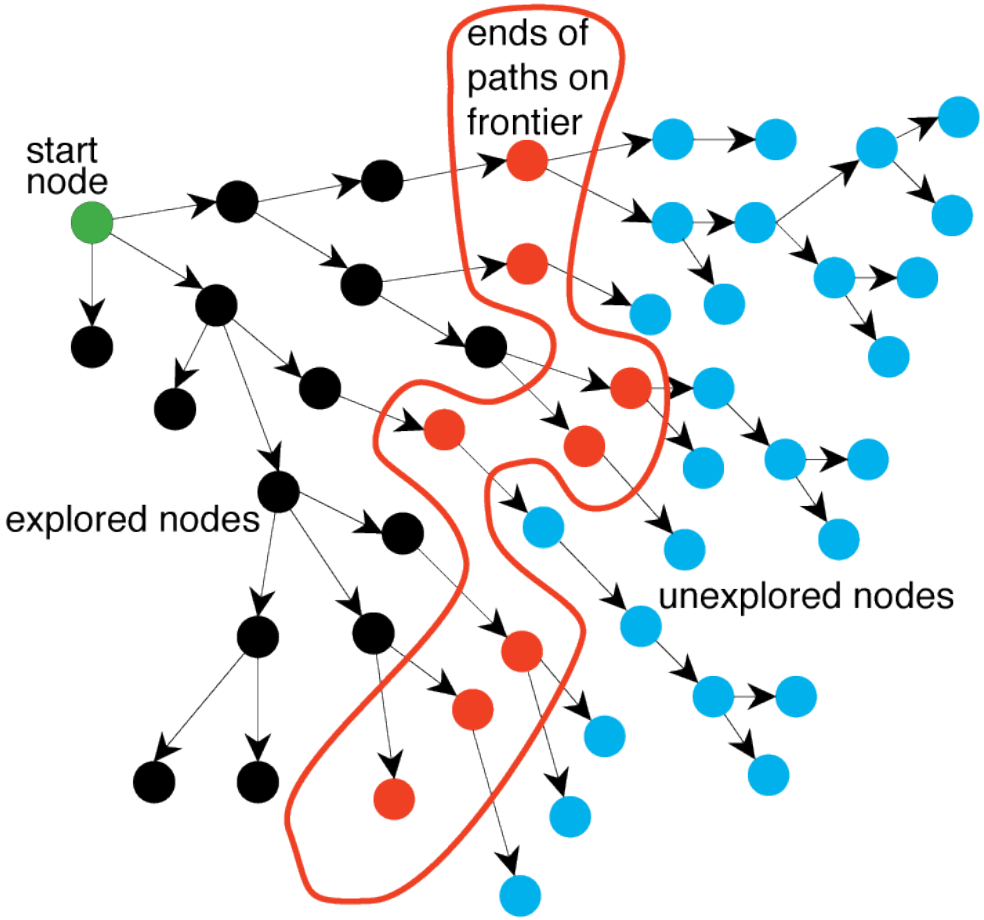
- Um estado é uma representação de uma configuração física.
- Um nó é uma estrutura de dados.
- Estados não têm pais, filhos, profundidade ou custo.



Fazendo busca sistemática em grafos

- Algoritmo genérico de busca:
  - Dado um grafo, nós iniciais e nós objetivos, explorar incrementalmente os caminhos a partir dos nós de início até um nó objetivo.
  - Manter uma **fronteira** de caminhos, a partir do nó de início, que já tenham sido explorados.
  - Com o andamento da busca, a fronteira se expande para os nós inexplorados até que seja encontrado um nó objetivo.
- A maneira na qual a fronteira é expandida define a **estratégia de busca**.

Resolvendo problemas pela busca sistemática em grafos



Direção da busca

- A função de exploração (ou *função de ação*) cria novos nós usando os operadores do problema para criar os estados correspondentes.
- A definição de *busca é simétrica*: é possível encontrar o caminho dos nós iniciais para a um nó objetivo ou de um nó objetivo para algum nó inicial.
- Direção da expansão:
  - Do estado inicial para um estado final (busca dirigida por dados ou encadeamento para frente).
  - De um estado final para o estado inicial (busca baseada em objetivo ou encadeamento para trás).
  - Busca bidirecional (é realizada nas duas direções ao mesmo tempo).

Algoritmo Geral de busca em grafo

```
def busca(G, S, goal):
  entrada: G # a grafo
           S # um conjunto de nós iniciais
           objetivo # função booleana que testa se S é um nó objetivo

  saída: um caminho de um membro de S para um nó para o qual a
         função objetivo é verdadeira, ou  se não existir solução

  local: fronteira # um conjunto de caminhos


  fronteira ← {<s>|s  S}
  while fronteira :
     selecione e remova <s0,…,sk> da fronteira

     if objetivo(sk): return <s0,…,sk>

     fronteira ← fronteira  # depende da estratégia
  return
```

Critérios de avaliação das estratégias de busca

- Completude (completeza):
  - A estratégia é completa se, sempre que existir uma solução, ela a encontra em uma quantidade de tempo finito.
- Qualidade/otimicidade:
  - A estratégia é ótima se, quando ela acha uma solução, esta é a melhor solução.
- Custo do tempo:
  - Quanto tempo a estratégia gasta para encontrar a solução no pior caso. Normalmente medida em termos do número máximo de nós expandidos.
- Custo de memória:
  - Quanta memória a estratégia usa para encontrar a solução, no pior caso. Normalmente medida pelo tamanho máximo que a lista de nós abertos (fronteira) assume durante a busca.

Critérios de avaliação das estratégias de busca

- As complexidades de tempo e espaço são medidas em termos dos seguintes fatores:
  - ***b*** : Fator de ramificação máximo da árvore de busca.
    - O *fator de ramificação* de um nó é o seu número de vizinhos.
      - Fator de ramificação para frente: número de arcos saindo de um nó.
      - Fator de ramificação para trás: número de arcos entrando um nó.
  - ***d*** : Profundidade da solução de menor custo.
  - ***m*** : Profundidade máxima do espaço de estados (pode ser ∞).

Busca Sistemática Desinformada

Estratégias comuns de busca desinformada:

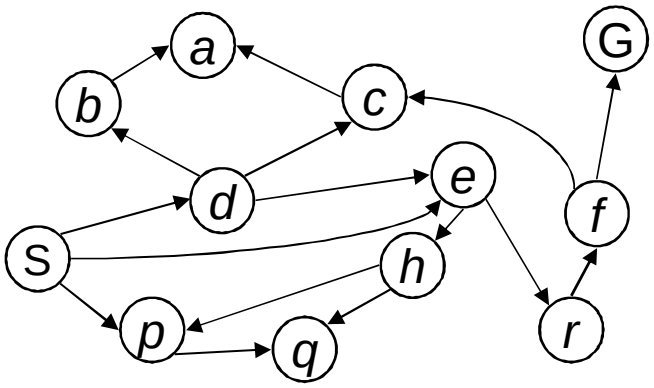
- Busca em profundidade.
- Busca em largura.
- Busca pelo custo uniforme
- Busca por aprofundamento iterativo.

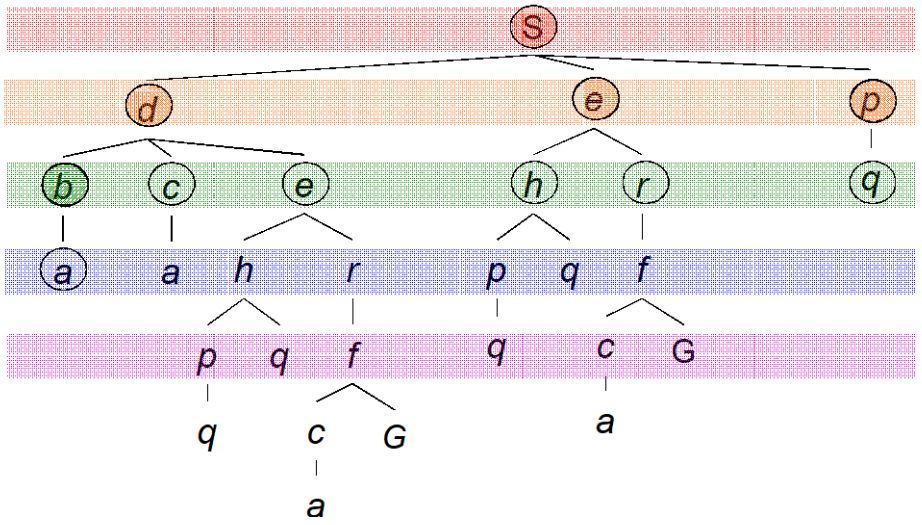
Busca em largura

- Trata a fronteira como uma ***fila***.
- Sempre seleciona um dos primeiros elementos adicionados à fronteira.
- Se a lista de caminhos na fronteira for [p_1, p_2, ... , p_r]:
  - Sendo p_1 selecionada, seus vizinhos são adicionados ao final da fila, após p_r.
  - p_2 será selecionado em seguida.

Busca em profundidade

- Estratégia: expanda o nó mais raso primeiro.





Algoritmo para busca em largura

```
def busca_em_largura(G, S, goal):
  entrada: G # a grafo
           S # um conjunto de nós iniciais
           objetivo # função booleana que testa se S é um nó objetivo

  saída: um caminho de um membro de S para um nó para o qual a
         função objetivo é verdadeira, ou ⊥ se não existir solução

  Local: fronteira # um conjunto de caminhos


  fronteira ← {<s>|s ∈ S}
  while fronteira :
     selecione e remova o 1º nó <s_0,…,s_k> da fronteira

     if objetivo(s_k): return <s_0,…,s_k>

     adicione(fim, fronteira, {<s_0,…,s_k,s>}|<s_k,s> ∈ S})
  return ⊥
```

Complexidade da busca em largura

- Se o fator de ramificação para todos os nós é finito, a busca em largura é **completa**. Também é garantido que ela encontrará o **caminho com o menor número de arcos**.
- A **complexidade do espaço** é exponencial no comprimento do caminho $O(b^n)$, onde b é fator de ramificação, n é o comprimento do caminho.
- A **complexidade de tempo** é exponencial no comprimento do caminho: $O(b^n)$, onde b é fator de ramificação, n é o comprimento do caminho.
- A busca não é restringida pelo objetivo.

Quando a busca em largura é apropriada

- Apropriada:
  - Quando espaço de memória não é problema.
  - Quando é necessário achar a solução com menos arcos.
  - Apesar de nem todas as soluções serem rasas, algumas são.
- Inapropriada:
  - Quando o espaço de memória é limitado.
  - Quando todas as soluções estão fundas na árvore de busca.
  - Quando o fator de ramificação é muito grande.

Busca em profundidade

- Trata a fronteira como uma ***pilha***.
  - i.e. sempre seleciona um dos últimos elementos adicionados à fronteira.

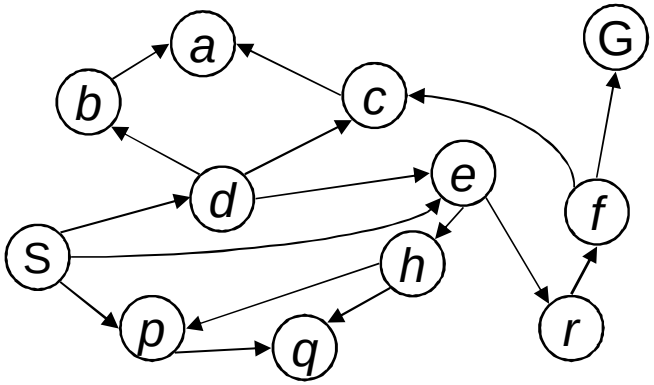
- Se a lista de caminhos na fronteira é [p_1, p_2, ...]

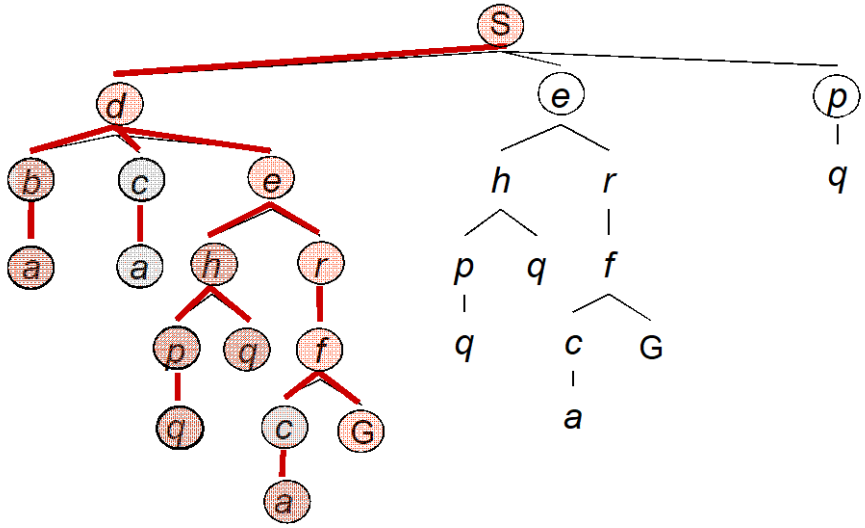
  - Se p_1 é selecionado, os caminhos que estendem p_1 são adicionados à frente da pilha (em frente a p_2).

  - p_2 é selecionado somente quando todos os caminhos de p_1 foram explorados.

Busca em profundidade

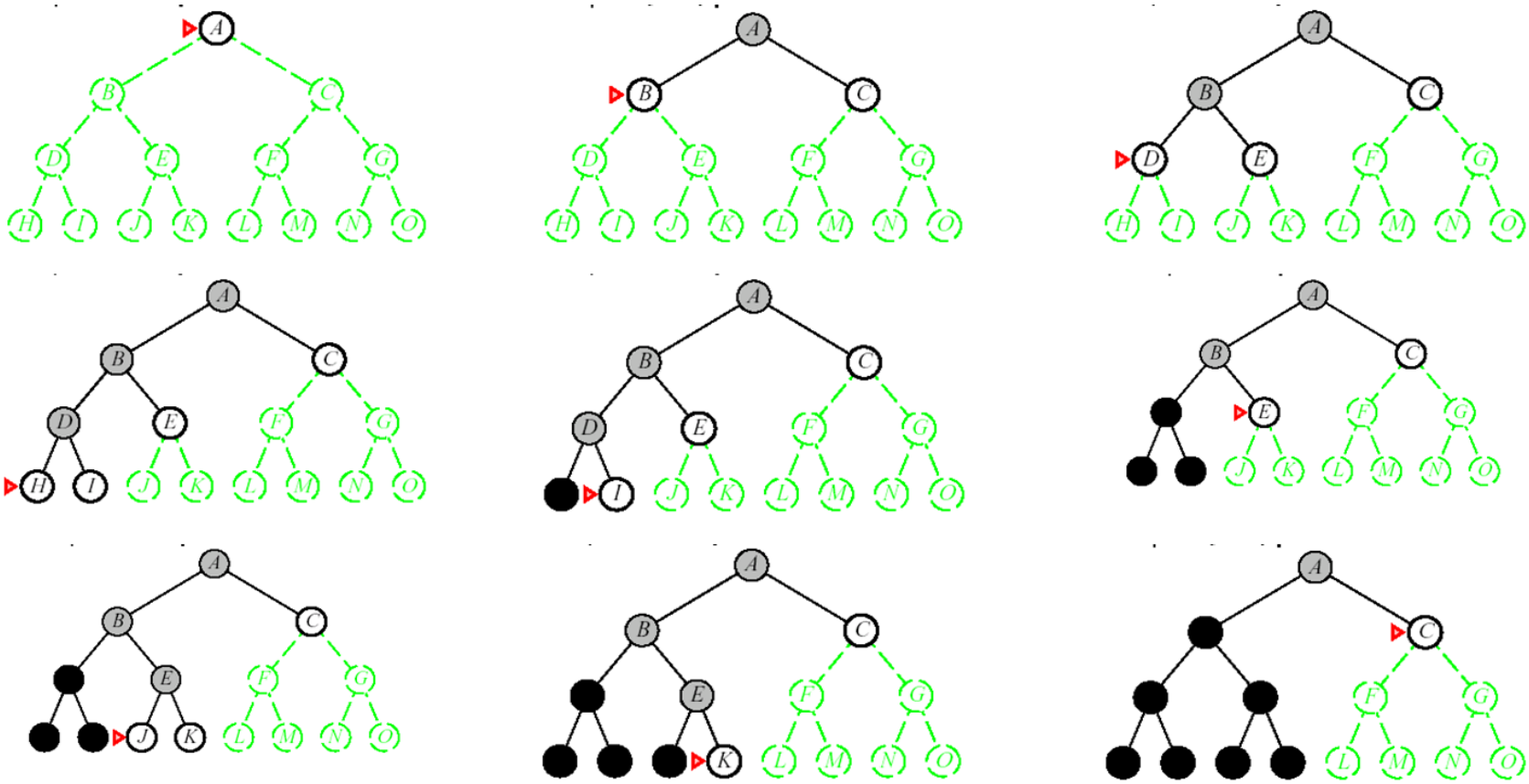
- Estratégia: expanda o nó mais profundo primeiro.





Árvore de busca da busca em profundidade

- O algoritmo de busca gera como resultado implícito uma árvore de busca:



Algoritmo para busca em profundidade

def busca_em_profundidade(G, S, *goal*):

entrada: G # a grafo

S # um conjunto de nós iniciais

objetivo # função booleana que testa se *S* é um nó objetivo

saída: um caminho de um membro de **S** para um nó para o qual a função ***objetivo*** é verdadeira, ou ⊥ se não existir solução

Local: *fronteira* # um conjunto de caminhos

fronteira ← {<s>|s ∈ S}

while *fronteira* :

selecione e **remova** o **1º** nó $\langle s_0,\ldots,s_k\rangle$ da *fronteira*

if *objetivo*(s_k): **return** $\langle s_0,\ldots,s_k\rangle$

adicione(**início**, *fronteira*, $\{\langle s_0,\ldots,s_k,s\rangle \mid \langle s_k,s\rangle \in S\}$)

return ⊥

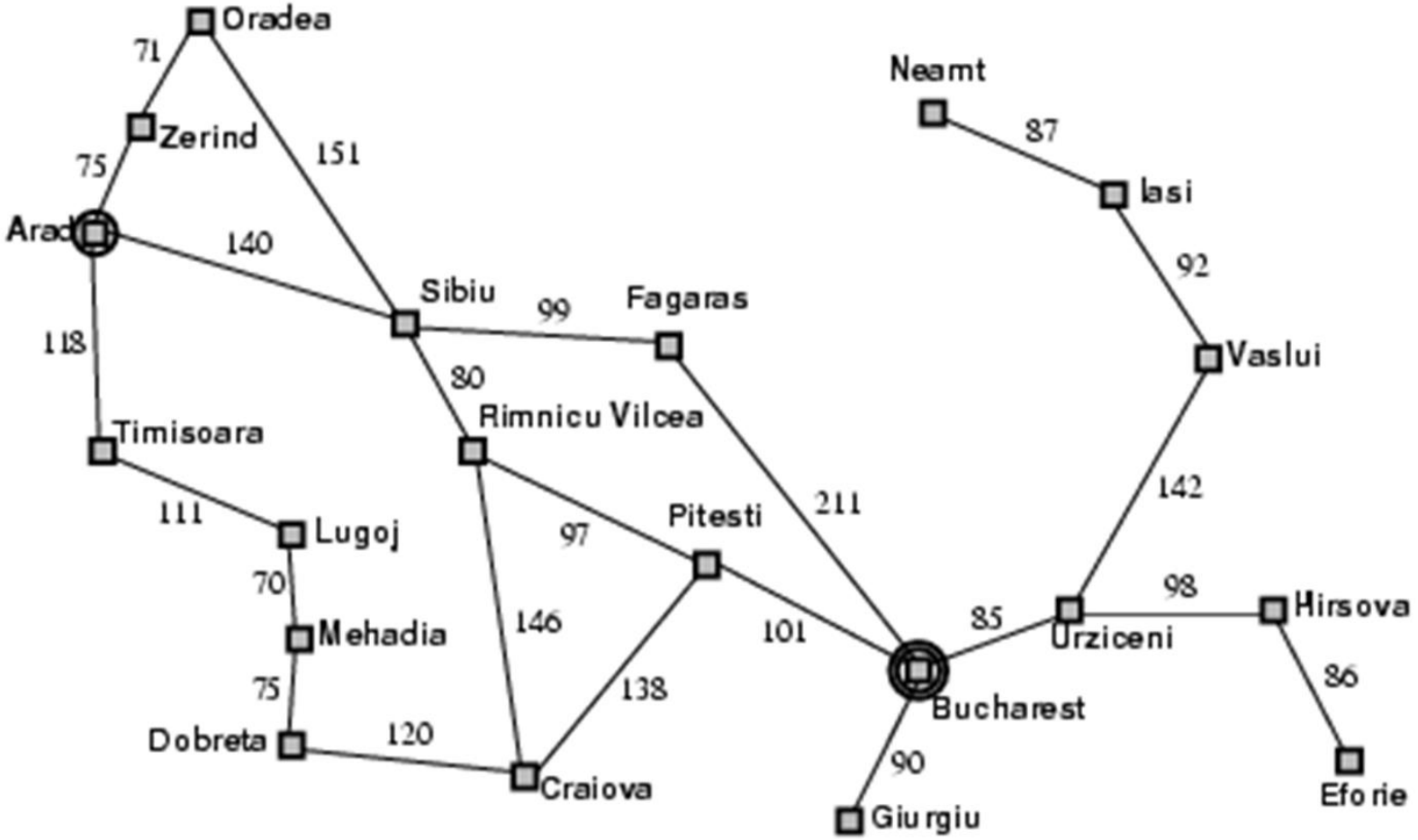
Complexidade da busca em profundidade

- A busca em profundidade **não tem garantia de terminar** em grafos infinitos ou em grafos com ciclos.
- A **complexidade do espaço** é linear no tamanho do caminho a ser explorado **O(mb)**, onde ***b*** é o fator de ramificação e ***m*** é o tamanho do caminho.
- A **complexidade do tempo** é exponencial no tamanho do caminho a ser explorado $\mathbf{O(b^m)}$, onde ***b*** é o fator de ramificação e ***m*** é o tamanho do caminho.
- A busca não é restringida pelo objetivo até que aconteça dela tropeçar no objetivo.

Quando a busca em profundidade é apropriada

- Apropriada:
  - O espaço de estados é restrito (representação complexa de estados, e.g. robótica)
  - Existe muitas soluções, talvez com caminhos longos, particularmente para o caso em que todos os caminhos levam a uma solução.
- Inapropriada:
  - Quando se tem ciclos no espaço de estados.
  - Quando existem soluções rasas.
  - Se a otimalidade é importante.

Busca desinformada com custos



Busca pelo custo uniforme

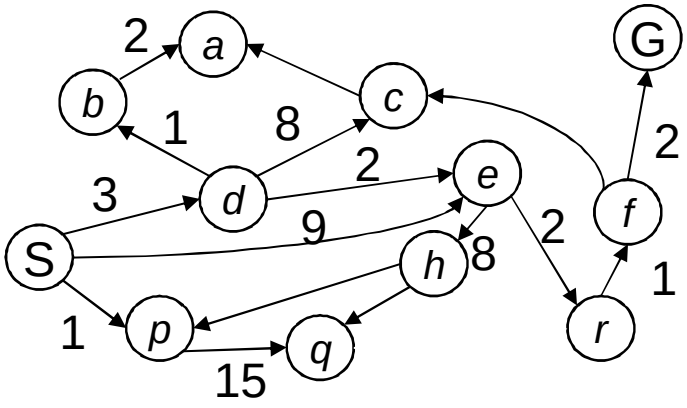
- Às vezes há **custos** associados com os arcos. O custo de um caminho é a soma dos custos de seus arcos.

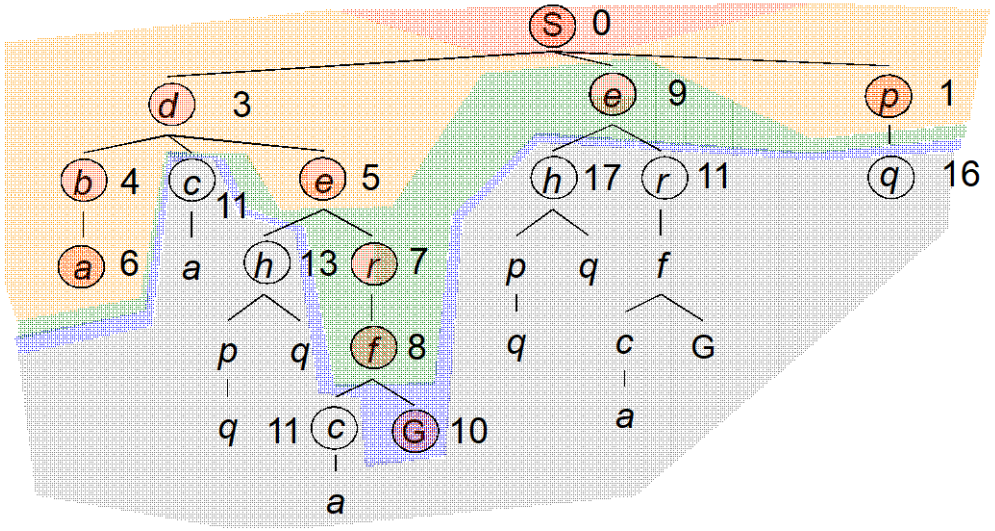
$$cost(\langle n_0, \ldots, n_k \rangle) = \sum_{i=1}^{k} |n_{i-1}, n_i|$$

- Em cada fase, a busca pelo menor-custo-primeiro seleciona um caminho na fronteira com o menor custo.
- A fronteira é uma *fila de prioridade* ordenada pelo custo do caminho.
- Encontra um caminho de menor custo para um nó de objetivo.
- Quando os custos do arco são iguais temos uma busca em largura.

Busca pelo custo uniforme

- Estratégia: expanda o nó mais barato primeiro.





Algoritmo para busca pelo custo uniforme

def busca_pelo_custo_uniforme(G, S, *goal*):

entrada: G # a grafo

S # um conjunto de nós iniciais

objetivo # função booleana que testa se *S* é um nó objetivo

saída: um caminho de um membro de **S** para um nó para o qual a função ***objetivo*** é verdadeira, ou ⊥ se não existir solução

Local: *fronteira* # um conjunto de caminhos

fronteira ← {<s>|s ∈ S}

while *fronteira* :

selecione e **remova** o **1º** nó $\langle s_0,\ldots,s_k\rangle$ da *fronteira*

if *objetivo*(s_k): **return** $\langle s_0,\ldots,s_k\rangle$

adicione(ordem crescente de custo, *fronteira*, $\{\langle s_0,\ldots,s_k,s\rangle | \langle s_k,s\rangle \in S\}$)

return ⊥

Análise da busca pelo custo uniforme

- Completude:
  - Não, pois um ciclo com custo zero ou negativo poderia ser seguido para sempre.
  - Sim, dado que os custos dos arcos sejam sempre positivos.
- Otimalidade:
  - Não, pois podem existir arcos com custo negativos.
  - Sim, sempre que os arcos tiverem custos não negativos.
- A complexidade de tempo é exponencial $O(b^m)$.
- A complexidade de espaço é exponencial $O(b^m)$, pois se tem que armazenar toda a fronteira na memória.

Busca por aprofundamento iterativo

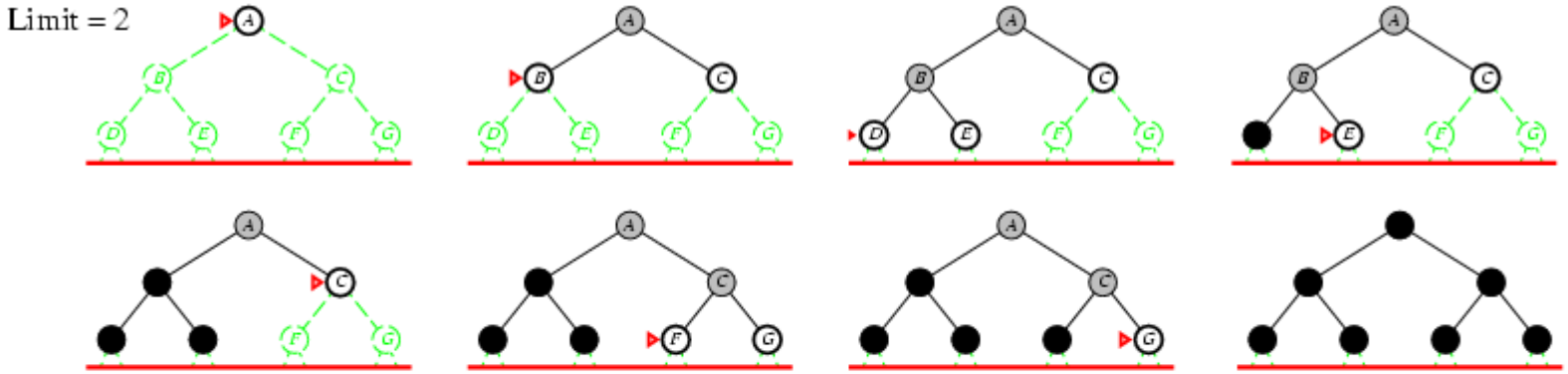
- Até agora todas as estratégias de pesquisa que são garantidas de parar usam espaço exponencial.
- Ideia:
  - Vamos recalcular os elementos de fronteira em vez de salvá-los.
  - Assim, vamos procurar caminhos da profundidade 0 e, em seguida, 1, 2, 3, etc.
  - Se um caminho não pode ser encontrado na profundidade ***B***, procure por um caminho na profundidade ***B + 1***. Aumente o limite de profundidade quando a busca falhar de forma não natural (i.e. o limite de profundidade for atingido).
- É necessário um buscador em profundidade que verifique um limite máximo para a profundidade da busca.

Exemplo: Busca por aprofundamento iterativo

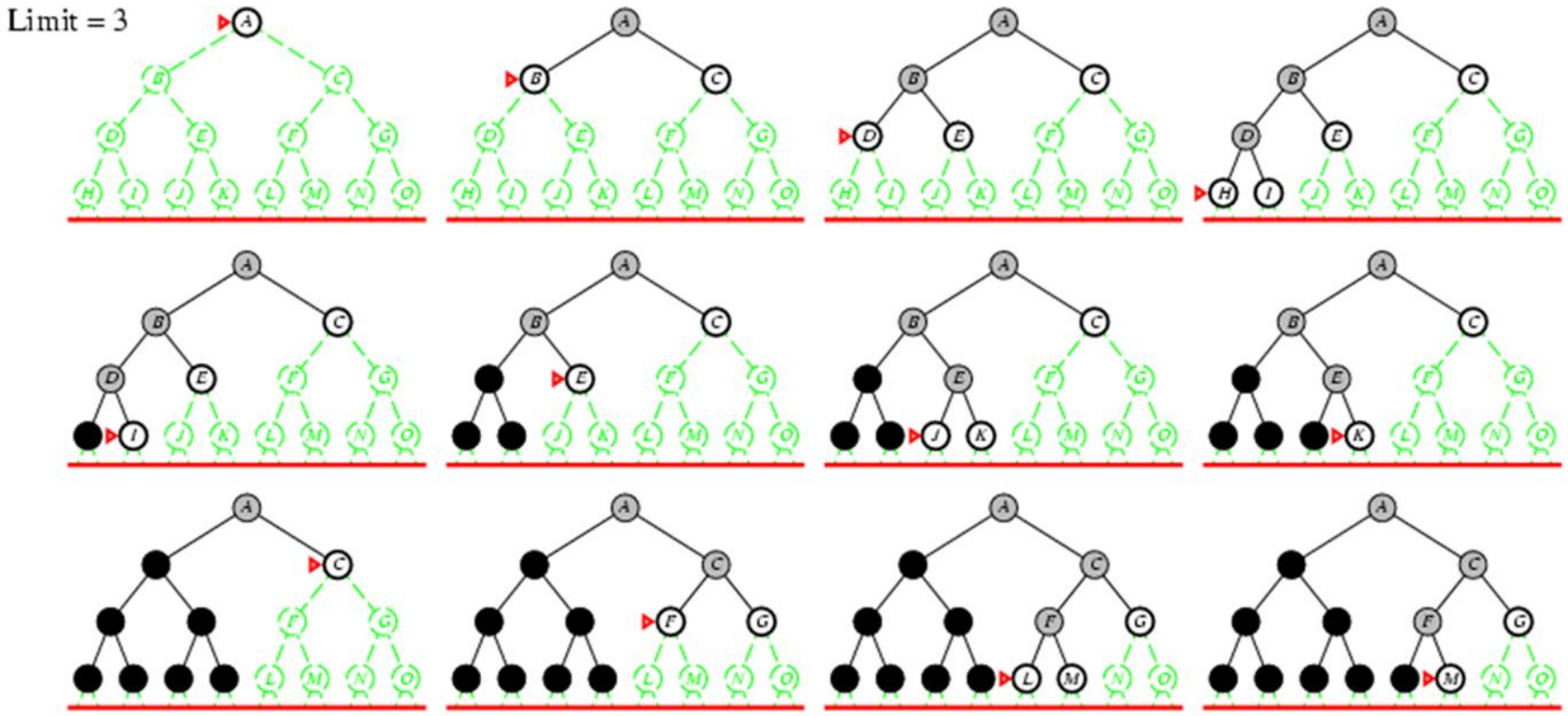
Limit = 0

Exemplo: Busca por aprofundamento iterativo

Exemplo: Busca por aprofundamento iterativo



Exemplo: Busca por aprofundamento iterativo



Busca por aprofundamento iterativo

- Apesar de parecer bem pouco eficiente…
  - Os nós que são “recomputados” várias vezes são os nós dos níveis mais altos a árvore de busca , os quais normalmente são em menor quantidade.
  - O custo de memória é linear como na busca em profundidade.
- É completa quando o fator de ramificação for finito.
- É ótima quando o custo do caminho é uma função não-decrescente da profundidade do nó.

Busca por aprofundamento iterativo

```
def busca_aprofudamento_iterativo(S, G, objetivo):
    entrada: G # a grafo
             S # um conjunto de nós iniciais
             objetivo # função booleana que testa se S é um nó objetivo
    saída: um caminho de um membro de S para um nó para o qual a função
    objetivo é verdadeira, ou ⊥ se não existir solução
    local: limite # profundidade máxima

    def busca_profundidade_limitada(<s0,…,sk>, G, limite):
        local: atingiu_limite # máxima profundidade

         atingiu_limite ← False
         if objetivo(sk): return <s0,…,sk>
         elif profundidade(sk) = limite: return interrupção
         else: for each {<s0,…,sk,s>}|<sk,s> ∈ S}:
                   resultado ← busca_profundidade_limitada(<s0,…,sk,s>, G, limite)
                   if resultado = interrupção: atingiu_limite ← True
                   elif resultado ≠ ⊥: return resultado
         if atingiu_limite: return interrupção
         else: return resultado

    for limite = 0 until ∞:
         resultado ← busca_profundidade_limitada({<s> ∈ S}, G, limite)
         if resultado ≠ ⊥ : return resultado
```

Complexidade da busca por aprofundamento iterativo

- Complexidade com uma solução na profundidade ***k*** e fator de ramificação ***b***:

level	breadth-first	iterative deepening	# nodes
1	1	k	b
2	1	$k-1$	b^2
$k-1$	1	2	b^{k-1}
k	1	1	b^k
	$\geq b^k$	$\leq b^k \left(\frac{b}{b-1}\right)^2$	

Sumário: Busca sistemática desinformada

Tipo de Busca	Seleção da fronteira	Completa	Ótima	Custo de Tempo	Custo de Espaço
Largura	Primeiro nó adicionado	S	S → custos constantes	$O(b^m)$	$O(b^m)$
Profundidade	Último nó adicionado	N S → se espaço de busca for sem ciclos e finito	N	∞ *ou* $O(b^m)$	$O(mb)$
Custo Uniforme	Nó de custo mínimo → *cost*(n)	S → custos > 0	S → custos > 0	$O(b^m)$	$O(b^m)$
Aprofundamento interativo	Último nó adicionado	N S → se espaço de busca for sem ciclos e finito	S → custos constantes	$O(b^m)$	$O(mb)$

Exercícios:

- Dado o mapa abaixo aplique os algoritmos de busca em largura, busca em profundidade e busca de custo uniforme com o estado inicial em **Ulm** e o estado final em **Frankfurt**.