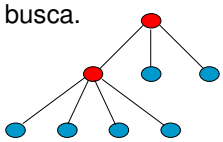
Inteligência Artificial

Estratégias de Busca Não Informada

Busca não informada

- A busca não informada (ou busca cega) não possui estimativas sobre qual sucessor é mais promissor para atingir a meta.
- Fronteira : todos os nós gerados e ainda não expandidos (ou visitados) da árvore de busca.



: Nós expandidos

: Nós da fronteira

2

Estratégias de Busca Cega

- Busca em Largura
- Busca de Custo Uniforme
- Busca em Profundidade
- Busca em Profundidade Limitada
- Busca em Profundidade com Aprofundamento Iterativo
- Busca Bidirecional
- Evitando Estados Repetidos
- Busca com Conhecimento Incompleto

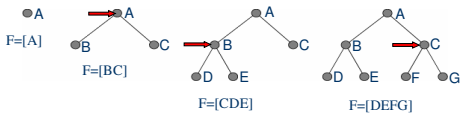
3

Busca em Largura

- Ordem de expansão dos nós:
  1. Nó raiz
  2. Todos os nós de profundidade 1
  3. Todos os nós de profundidade 2, etc…

Fronteira = FIFO (first-in-first-out)

➔ insere no fim da fila



4

Desempenho da busca em largura

- Completa?
  - Se b finito, é completa: se um nó-meta estiver a uma profundidade d, a busca em largura sempre irá encontrá-lo.
- Ótima?
  - Nem sempre – caminho mais curto ≠ melhor caminho
  - É ótima se o custo do caminho for uma função não-decrescente da profundidade do nó (ex: todas ações têm mesmo custo)

5

Desempenho da busca em largura

- Complexidade de tempo
  - Meta em d, cada nó tem b filhos. No pior caso, vem:

$$1 + b + b^2 + b^3 + \ldots + b^d + (b^{d+1} - b) = \mathcal{O}(b^{d+1})$$

- Complexidade de espaço
  - Mantém todos nós gerados (ou está na fronteira ou é ancestral (está na lista de expandidos))

$$1 + b + b^2 + b^3 + \ldots + b^d + (b^{d+1} - b) = \mathcal{O}(b^{d+1})$$

6

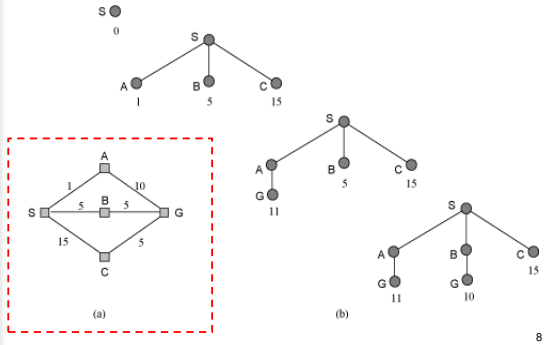
Busca de Custo Uniforme

- Modifica a busca em largura:
  - Em vez de expandir o nó gerado primeiro, expande o nó da fronteira com menor custo de caminho (da raiz ao nó)

  Fronteira ➔ insere em ordem crescente
- Não se importa com o número de passos, mas com o custo total
- g(n) dá o custo do caminho da raiz ao nó n
  - Na busca em largura: ***g(n) = profundidade (n)***

7

Busca de Custo Uniforme



8

Busca de Custo Uniforme
Fronteira do exemplo anterior

- F = {S}
  - testa se S é o estado objetivo, expande-o e guarda seus filhos A, B e C ordenadamente (segundo custo) na fronteira
- F = {A, B, C}
  - testa A, expande-o e guarda seu filho G_A ordenadamente
  - **obs.:** o algoritmo de geração guarda na fronteira todos os nós gerados, testando se um nó é o objetivo apenas quando ele é retirado da lista!

9

Busca de Custo Uniforme
Fronteira do exemplo anterior

- F= {B, G_A, C}
  - testa B, expande-o e guarda seu filho G_B ordenadamente
- F= {G_B, G_A, C}
  - testa G_B e pára!

10

Desempenho da Busca de Custo Uniforme

- Completa? Só *se custo de cada ação* $\geq \varepsilon$, $\forall n$
  - ε *é uma constante pequena positiva*
    - Loop infinito: se expande nó que tem ação de custo=0 levando de volta ao mesmo nó.
- Ótima? *Só se g (sucessor(n)) > g (n)*
  - *custo* ***no mesmo caminho*** *sempre cresce*
  - *i.e., não tem ação com* ***custo negativo ou 0***
- Complexidade de tempo e de espaço
  - *C*=custo da solução ótima, custo de cada ação* $\geq \varepsilon$

  ➔ Pior caso: $O(b^{1+\lfloor C^*/\varepsilon \rfloor})$, o que pode ser bem maior que b^d

11

Busca em Profundidade

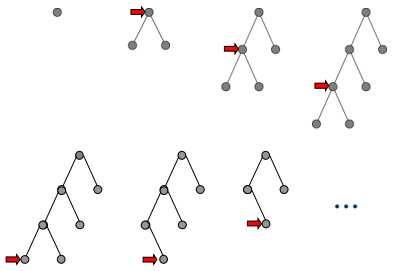
- Ordem de expansão dos nós:
  1. Nó raiz
  2. Primeiro nó de profundidade 1
  3. Primeiro nó de profundidade 2, etc…

  Fronteira = LIFO (last-in-first-out)

  ➔ insere no início da fila

12

Busca em Profundidade



Exemplo: nós com profundidade 3 não têm sucessores

13

Desempenho da Busca em Profundidade

- Esta estratégia **não é *completa*** (caminho pode ser infinito) **nem é *ótima*.**
- Complexidade espacial:
  - mantém na memória o caminho que está sendo expandido no momento, e os nós irmãos dos nós no caminho para possibilitar o retrocesso (***backtracking***)
  - Apaga subárvores já visitadas
  - Para espaço de estados com fator de ramificação b e **profundidade máxima m** (m pode ser >> d), requer **bm+1** de memória ➔ $\mathcal{O}$ (bm)

14

Desempenho da Busca em Profundidade

- Complexidade temporal: $\mathcal{O}$ (b^m), no pior caso.
  - Para problemas com várias soluções, esta estratégia pode ser bem mais rápida do que busca em largura.
  - Esta estratégia deve ser evitada quando as árvores geradas são muito *profundas* ou geram *caminhos infinitos*.

15

Variante: Busca com Retrocesso

- Parecida com BP, mas somente UM sucessor é gerado em cada iteração
  - na BP, todos os sucessores são gerados na expansão do nó pai
- Portanto, requer só $\mathcal{O}$ (m) de memória
  - BP requer $\mathcal{O}$ (bm) de memória
- Deve ser capaz de retornar ao pai e criar novo sucessor

16

Busca com Aprofundamento Limitado

- Evita o problema de árvores não limitadas impondo um limite máximo (ℓ) de profundidade para os caminhos gerados.
  - O domínio do problema estabelece a profundidade limite.
  - Problema: definir limite ℓ adequado!
- Completa? Somente se $\ell \geq$ d.
- Ótima? Não, exceto se $\ell =$ d.
- Complexidade espacial: $\mathcal{O}$ (b . ℓ)
- Complexidade temporal: $\mathcal{O}$ (b^{ℓ}) no pior caso.

17

Busca com Aprofundamento Iterativo (BAI)

- Tenta limites com valores crescentes, partindo de zero, até encontrar a primeira solução (em d).
- Combina vantagens da busca em largura (BL) com as da busca em profundidade (BP).
- Em geral, é a estratégia preferida de busca cega para quando o espaço de estados é muito grande e a profundidade da solução d é desconhecida.
- Possui uma variante, a Busca com Comprimento Iterativo (BCI) – analogia entre BAI e BL, e BCI e Custo Uniforme: usa incremento iterativo do custo do caminho em vez de incremento na profundidade (mas BCI não é eficiente!)

18

Desempenho da BAI

- Completa? Sim se b for finito (idem BL).
- Ótima? Sim se o custo do caminho for uma função crescente com a profundidade do nó (idem BL).
- Complexidade espacial:

  $\mathcal{O}$ (bd) (idem BP)

19

Desempenho da BAI

- Complexidade temporal: nós na profundidade da menor solução (d) são gerados 1 vez, em d-1 são gerados 2 vezes, ... na profundidade 1 são gerados d vezes:

$(d)b + (d-1)b^2 + (d-2)b^3 + ... + (1)b^d = \mathcal{O}(b^d)$

OBS: BL gera alguns nós em d+1 e BAI não gera.

BL: $\mathcal{O}(b^{d+1})$

Na realidade, BAI é mais rápida que BL.

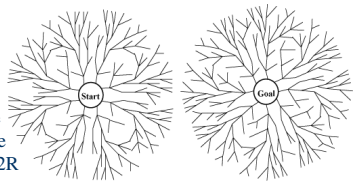
20

Busca Bidirecional (1)

- Busca em duas direções (duas buscas simultâneas):
  - para frente, a partir do nó inicial, e
  - para trás, a partir do nó final (objetivo)
- A busca pára quando o nó a ser expandido por uma busca se encontra na fronteira da outra busca.
- **Motivação:**

$b^{d/2} + b^{d/2} < b^d$.

Ou: a área de dois círculos de raio R é menor que a área de um círculo de raio 2R



Busca Bidirecional (2)

- Para BL nas duas direções: $\mathcal{O}(b^{d/2})$ – tempo e espaço. Completude e otimalidade: idem BL.
  - É possível utilizar *estratégias* diferentes em cada direção da busca (podendo sacrificar desempenho)
- Porém, encadeamento reverso (da meta para o início) só é possível se todas as ações no espaço de estado forem reversíveis.
- Outro problema: quando há vários estados-meta ou quando é muito difícil computar os estados-meta pelo teste de término (ex. estados para cheque-mate).

22

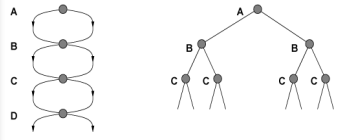
Evitando Estados Repetidos (1)

- Problema geral em Busca
  - expandir estados já previamente encontrados e expandidos
- É inevitável quando há operadores reversíveis
  - ex. encontrar rotas, canibais e missionários, 8-números, etc.
  - a árvore de busca é potencialmente infinita

23

Evitando Estados Repetidos (2)

Espaço de estados *Árvore de busca*



Exemplo:
tamanho do espaço de estados=m+1;
2^m caminhos na árvore

- Idéia:
  - **podar** (*prune*) estados repetidos, para gerar apenas a parte da árvore que corresponde ao grafo do espaço de estados (que é finito!)
  - mesmo quando esta árvore é finita...evitar estados repetidos pode reduzir exponencialmente o custo da busca

24

Evitando Estados Repetidos (3)

- **Problema**: deve armazenar todos nós gerados!
  - Além da lista de fronteira (também chamada de ***open list***), os algoritmos precisam da lista de nós visitados / expandidos (***closed list***)
  - Cada nó gerado é comparado com aqueles da *closed list*: se for repetido, descarta aquele de caminho com custo pior.
  - pode ser implementado mais eficientemente com *hash tables*
  - BP e BAI: perdem propriedade de complexidade linear no espaço.

25