Busca em Espaço de Estados

Jomi F. Hübner

Universidade Federal de Santa Catarina
Departamento de Automação e Sistemas
http://jomifred.github.io/ia

Introdução

Agente orientado a meta

- O projetista determina que objetivo o agente deve alcançar (e não um programa a ser executado)
- É necessário que o próprio agente construa um plano de ações que atinjam seu objetivo
(como se o próprio agente construísse seu programa)
- Exemplos: o agente aspirador de pó, um agente motorista de táxi, uma sonda espacial, ...

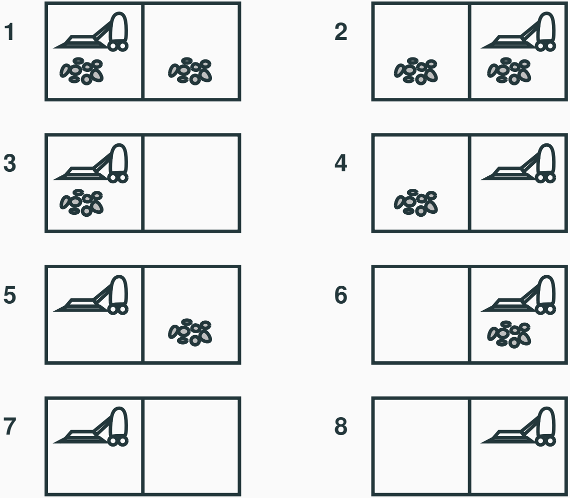
Exemplo do aspirador de pó i

- Um robô aspirador de pó deve limpar uma casa com duas posições. Operações que ele sabe executar:
    - sugar
    - ir para a posição da esquerda
    - ir para a posição da direita
- Como o aspirador pode montar um plano para limpar a casa se inicialmente ele esta na posição direita e as duas posições têm sujeira?
    - Quais os estados possíveis do mundo do aspirador e as transições?

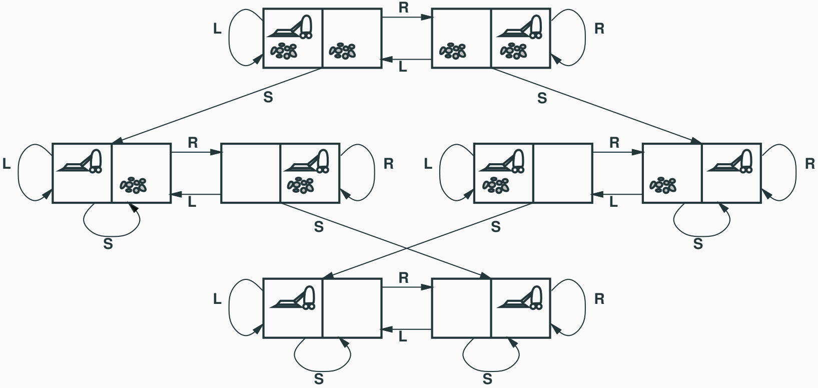
Exemplo do aspirador de pó ii

Estados possíveis:

Exemplo do aspirador de pó iii



Espaço de busca



O que é busca?

- O mundo do agente é modelado por conjunto de **estados** possíveis (muitas vezes este conjunto é infinito)
- Existem **transições** entre os estados do mundo, formando um grafo
- São utilizados **algoritmos** para encontrar um caminho neste grafo
    - partindo do estado inicial (atual)
    - até o estado objetivo

Por que estados?

- As informações do mundo real são absurdamente complexas, é praticamente impossível considerar todas
    - No exemplo do aspirador, o mundo tem várias outras informações: a cor do tapete, se é dia, de que material o aspirador é feito, quanto ele tem de energia, como é o nome do/a proprietário/a, ...
- A noção de estado **abstrai** esses detalhes e **modela** somente o que é relevante para a solução do problema
- O mesmo se dá com as operações: são abstrações das operações reais (ir para a posição da direita implica em várias outras operações)

Exemplo dos jarros

- Temos dois jarros, um com capacidade para 4 litros de água e outro com capacidade para 3 litros.
- Utilizando somente operações de encher, esvaziar e derramar a água de um jarro no outro, o agente deve encontrar uma seqüência de operações que deixa o jarro com capacidade para 3 litros com 2 litros de água
- Quais os estados e as transições?

Busca como desenvolvimento de software i

- No desenvolvimento de um software para resolver um problema, o projetista pode optar por várias paradigmas de modelagem do problema:
    - O sistema é modelado por procedimentos que alteram os dados de entrada
    - O sistema é modelado por funções
    - O sistema é modelado por predicados
    - O sistema é modelado por objetos
    - ...

Busca como desenvolvimento de software ii

- Busca é mais uma forma de modelar um problema:
  - Definir os estados
  - Definir as transições
  - Escolher um algoritmo de busca

Exercício i

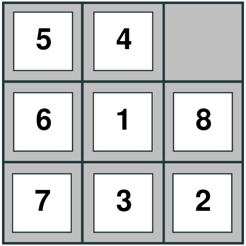
O que é

- estado
- transição
- estado meta e
- custo da solução encontrada

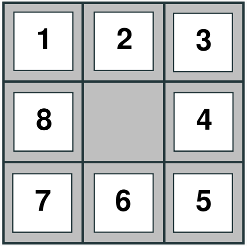
para os seguintes problemas

Exercício ii

- 8-Puzzle

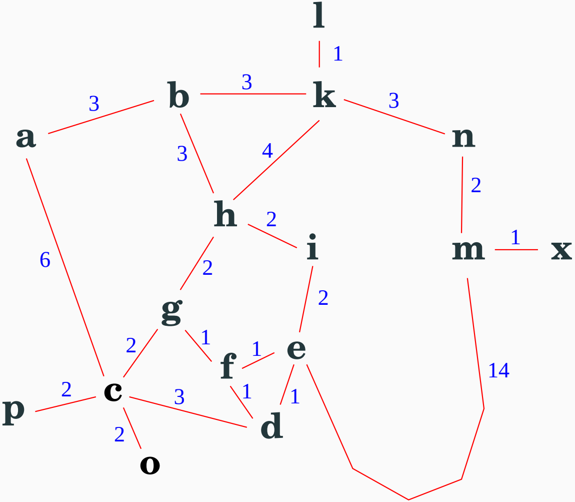


Start State



Goal State

- Encontrar um caminho da cidade "i" até "x"



Algoritmos de Busca “Cega”

Árvore de busca

- Coloca-se o estado inicial como nodo raiz
- Cada operação sobre o estado raiz gera um novo nodo (chamado de **sucessor**)
- Repete-se este processo para os novos nodos até gerar o nodo que representa o estado meta
- **Estratégia** de busca: que nodo escolher para expandir
- Exemplo: [fazer as árvores para o exemplo do aspirador e do jarro]

Estratégias de busca

- Busca em **largura**: o nodo mais **antigo** é escolhido para gerar sucessores
- Busca em **profundidade**: o nodo mais **recente** é escolhido para gerar sucessores

Nodos da árvore

- Cada nodo tem
  - o estado que representa
  - o nodo pai
  - o operador que o gerou
  - sua profundidade na árvore de busca
  - o custo de ter sido gerado (denotado por g)
  - opcionalmente, os nodos sucessores

Estratégias de poda da árvore de busca i

- Um nodo não gera um sucessor igual a seu pai
- Um nodo não gera um sucessor igual a um de seus ascendentes
- Um nodo não gera um sucessor que já exista na árvore de busca

Estratégias de poda da árvore de busca ii

- Detalhes de implementação:
  - Verificar se um estado já esta na árvore pode levar muito tempo
    - imagine uma árvore com milhares de estados do jogo de xadrez, cada novo estado deve ser comparado com outros milhares de estados!
  - Ter uma tabela hash (que tem tempo de ótimo para consulta) para saber se determinado nodo existe na árvore

Algoritmo de busca em largura

```
function BL(Estado inicial): Nodo

PriorityQueue(g) abertos {fila ordenada por g}
abertos.add(new Nodo(inicial))
while not abertos.isEmpty() do
   Nodo n ← abertos.remove()
   if n.getEstado().éMeta() then
      return n
   end if
   abertos.add(n.sucessores())
end while
return null
```

Critérios de comparação entre os algoritmos

- **Completo**: o algoritmo encontra a solução se ela existir
- **Ótimo**: o algoritmo encontra a solução de menor custo
- **Tempo**: quanto tempo o algoritmo leva para encontrar a solução no pior caso
- **Espaço**: quanto de memória o algoritmo ocupa

Análise do algoritmo BL

- Completo: sim
- Ótimo: sim
- Tempo: explorar todos os nodos da árvore
  - b = fator de ramificação
  - d = profundidade do estado meta
  - tamanho da árvore: $1 + b + b^2 + b^3 + ... + b^d$
  - complexidade: $O(b^d)$
- Espaço: $O(b^d)$

Exemplo de complexidade

Prof.	Nodos	Tempo	Memória
0	1	1ms	100 bytes
2	111	0,1 seg	11 Kbytes
4	11.111	11 seg	1 Mbyte
6	10^6	18 min	111 Mbytes
8	10^8	31 horas	11 Gbytes
12	10^{12}	35 anos	111 Tbytes
14	10^{14}	3500 anos	11.111 Tbytes

($b = 10$, 1000 nodos por segundo, 100 bytes por nodo)

Algoritmo de busca em profundidade

```
function BP(Estado inicial, int m): Nodo
Stack abertos
abertos.add(new Nodo(inicial))
while not abertos.isEmpty() do
    Nodo n ← abertos.remove()
    if n.getEstado().éMeta() then
        return n
    end if
    if n.getProfundidade() < m then
        abertos.add(n.sucessores())
    end if
end while
return null
```

Análise do algoritmo BP

- Completo: não (caso a meta esteja em profundidade maior que m)
  Se $m = \infty$, é completo se o espaço de estados é finito e existe poda para não haver loops entre as operações
- Ótimo: não
- Tempo: explorar $O(b^m)$ nodos (ruim se m é muito maior que d)
- Espaço: guardar $O(bm)$ nodos. (em profundidade 12, ocupa 12 Kbytes!)

Algoritmo de busca em profundidade iterativo

function BPI(Estado *inicial*): Nodo
int $p \leftarrow 1$
loop
  Nodo $n \leftarrow$ **BP**($inicial$, p)
  if $n \neq$ **null then**
    return n
  end if
  $p \leftarrow p + 1$
end loop

Análise do algoritmo BPI

- Completo: sim
- Ótimo: sim
  se todas as ações tem o mesmo custo
- Tempo: explorar $O(b^d)$ nodos
- Espaço: guardar $O(bd)$ nodos.

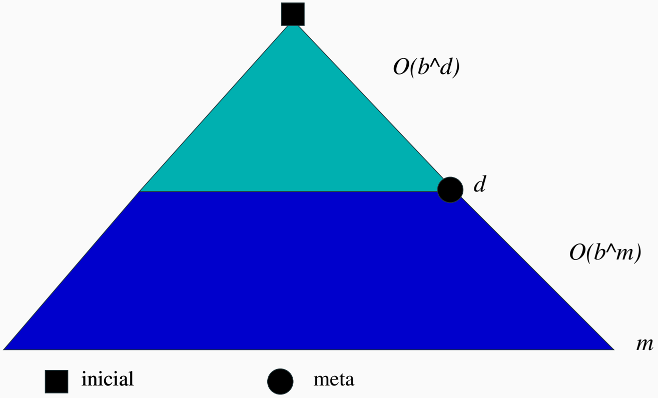
Algoritmo de busca em Bidirecional

function BBD(Estado *inicial*, *meta*): Nodo
Queue $abCima$, $abBaixo$
$abCima$.**add**(**new** Nodo($inicial$))
$abBaixo$.**add**(**new** Nodo($meta$))
while not ($abCima$.**empty**() **and** $abBaixo$.**empty**()) **do**
    Nodo $n \leftarrow abCima$.**remove**() {verifica lista de cima}
    if n.**getEstado**() $\in abBaixo$ **then return** $meta$ **end if**
    $abCima$.**add**(n.**sucessores**())
    $n \leftarrow abBaixo$.**remove**() {verifica lista de baixo}
    if n.**getEstado**() $\in abCima$ **then return** $meta$ **end if**
    $abBaixo$.**add**(n.**antecessores**())
end while
return null

Análise do algoritmo BBD

- Completo: sim
- Ótimo: sim
- Tempo: explorar $O(b^{d/2})$ nodos
- Espaço: guardar $O(b^{d/2})$ nodos
- Observação: deve ser possível gera antecessores

Resumo i

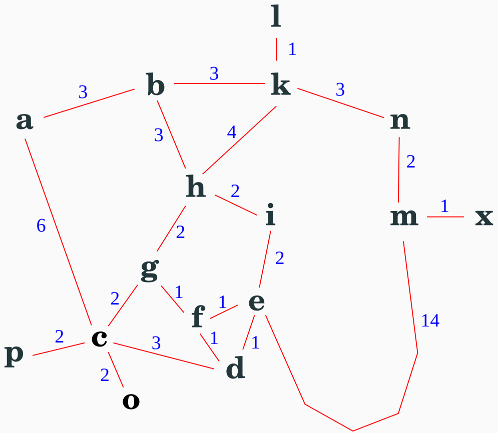


Resumo ii

	BL	BP	BPI	BBD
Completo	sim	não	sim	sim
Ótimo	sim	não	sim	sim
Tempo	$O(b^d)$	$O(b^m)$	$O(b^d)$	$O(b^{d/2})$
Espaço	$O(b^d)$	$O(bm)$	$O(bd)$	$O(b^{d/2})$

Algoritmos de Busca “Inteligente”

Exemplo: ir de “h” para “o” (com BL)



A árvore de busca gerada é “inteligente”?

Heurística

- Heurística: **Estimativa** de custo até a meta. (denotado pela função $h : Estados \rightarrow Reais$)
- No exemplo das cidades, poderia ser a distância em linha reta
- Algoritmo de **busca gananciosa**: retira de abertos sempre o nodo com menor estimativa de custo
  - Refazer a busca de um caminho entre "h" e "o". **ótimo**!
  - Refazer a busca de um caminho entre "i" e "x". **não** ótimo!

Busca A*

- **Idéia**: Evitar explorar caminhos que **já** estão muito caros e **preferir** os que têm menor expectativa de custo.
- Utilizar na escolha de um nodo da lista de abertos
  - tanto a estimativa de custo de um nodo ($h(n)$)
  - quanto o custo acumulado para chegar no nodo ($g(n)$)

  $f(n) = g(n) + h(n)$
- Refazer a busca de um caminho entre "i" e "x" utilizando f.

Algoritmo de busca A*

```
function BA*(Estado inicial): Nodo

PriorityQueue(f) abertos {fila ordenada por f}
abertos.add(new Nodo(inicial))
while not abertos.isEmpty() do
    Nodo n ← abertos.remove()
    if n.getEstado().éMeta() then
        return n
    end if
    abertos.add(n.sucessores())
end while
return null
```

Propriedades da função h

- Supondo que o valor de h, no exemplo das cidades, é dados por 10 * a distância em linha reta
- O algoritmo A* ainda é ótimo?
- $h(n)$: estimativa de custo de n até a meta
- $h^*(n)$: custo real de n até a meta
- Se $h(n) \leq h^*(n)$, então h é **admissível**.
- Se h é admissível, o algoritmo A* é ótimo!

Análise do algoritmo A*

- Completo: **sim**
- Ótimo: **sim** (se h é admissível)
- Tempo: explorar $O(b^d)$ nodos no pior caso (quando a heurística é "do contra")
- Espaço: guardar $O(b^d)$ nodos no pior caso.

Exercício i

- Determine uma heurística para o problema 8-Puzzle e verifique se é admissível.

5	4	
6	1	8
7	3	2

Start State

1	2	3
8		4
7	6	5

Goal State

Exercício ii

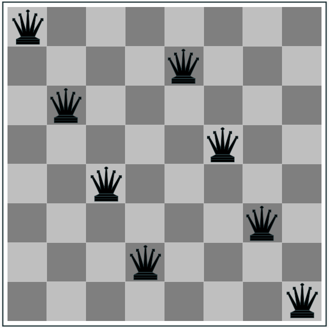
- h_1: número de peças fora do lugar
- h_2: distância de cada peça de seu lugar
- h_3: peças fora da formação de caracol
- $h_4 = h_2 + h_3$

Complexidade no problema 8-puzzle

	número de abertos			fator ramificação		
d	BPI	A*(h_1)	A*(h_2)	BPI	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
8	6384	39	25	2.80	1.33	1.24
12	364404	227	73	2.78	1.42	1.24
16	–	1301	211	–	1.45	1.25
20	–	7276	676	–	1.47	1.27
24	–	39135	1641	–	1.48	1.26

Exercício

- Determine uma heurística para o problema das 8-rainhas e verifique se é admissível.



Algoritmo Subida da Montanha-1

Idéia: escolher sempre o sucessor melhor

function BSM-1(Estado *inicial*): Estado
Estado $atual \leftarrow inicial$
loop
  $prox \leftarrow$ melhor sucessor de $atual$ (segundo h)
  if $h(prox) \geq h(atual)$ **then** {sem sucessor melhor}
    return $atual$
  end if
  $atual \leftarrow prox$
end loop

Análise do algoritmo BSM-1

- Não mantém a árvore (logo, não pode retornar o caminho que usou para chegar à meta).
- Completo: **não** (problema de **máximos locais**)
- Ótimo: não se aplica
- Tempo: **?**
- Espaço: **nada**!

Algoritmo Subida da Montanha-2

```
function BSM-2(Estado inicial): Estado
Estado atual ← inicial
loop
    prox ← melhor sucessor de atual (segundo h)
    if h(prox) ≥ h(atual) then {sem sucessor melhor}
        if atual.éMeta() then
            return atual
        else
            atual ← estado gerado aleatoriamente
        end if
    else
        atual ← prox
    end if
end loop
```

Análise do algoritmo BSM-2

- Completo: **sim** (se a geração de estados aleatórios distribuir normalmente os estados gerados)
- Ótimo: não se aplica
- Tempo: **?**
- Espaço: **nada**!

Material de consulta

- Capítulos 3 e 4 do livro do Russell & Norvig
- Implementação dos algoritmos disponível em `http://jomifred.github.io/ia`