Inteligência Artificial

Aula 6
Profª Bianca Zadrozny
http://www.ic.uff.br/~bianca/ia

Busca com informação e exploração

Capítulo 4 – Russell & Norvig
Seção 4.2 e 4.3

Revisão da aula passada: Busca A*

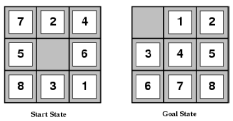
- Idéia: evitar expandir caminhos que já são caros
- Função de avaliação $f(n) = g(n) + h(n)$
  - $g(n)$ = custo até o momento para alcançar n
  - $h(n)$ = custo estimado de n até o objetivo
  - $f(n)$ = custo total estimado do caminho através de n até o objetivo.

Revisão da aula passada: Heurística Admissível

- Uma heurística $h(n)$ é admissível se para cada nó n,
  $h(n) \leq h^*(n)$, onde $h^*(n)$ é o custo verdadeiro de alcançar o estado objetivo a partir de n.
- Uma heurística admissível nunca superestima o custo de alcançar o objetivo, isto é, ela é otimista.
- Exemplo: $h_{DLR}(n)$ (distância em linha reta nunca é maior que distância pela estrada).
- Teorema: Se $h(n)$ é admissível, A* usando algoritmo `BUSCA-EM-ARVORE` é ótima.

Exemplo: Heurísticas Admissíveis

- Para o quebra-cabeça de 8 peças:
  - $h_1(n)$ = número de peças fora da posição
  - $h_2(n)$ = distância "Manhattan" total (para cada peça calcular a distância em "quadras" até a sua posição)



- $h_1(S) = ?$ 8
- $h_2(S) = ?$ 3+1+2+2+2+3+3+2 = 18

Medindo a qualidade de uma heurística

- Fator de ramificação efetiva
  - A* gera **N** nós
  - Profundidade da solução é **d**
  - Supondo uma árvore uniforme, podemos calcular o fator de ramificação efetiva b* a partir de

$$\mathbf{N+1 = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d}$$

Exemplo: Quebra-cabeça de 8 peças

	Custo da busca			Fator de ramificação efetiva		
d	BAI	$A^*(h_1)$	$A^*(h_2)$	BAI	$A^*(h_1)$	$A^*(h_2)$
2	10	6	6	2,45	1,79	1,79
4	112	13	12	2,87	1,48	1,45
6	680	20	18	2,73	1,34	1,30
8	6384	39	25	2,80	1,33	1,24
10	47127	93	39	2,79	1,38	1,22
12	3644035	227	73	2,78	1,42	1,24
14	–	539	113	–	1,44	1,23
16	–	1301	211	–	1,45	1,25
18	–	3056	363	–	1,46	1,26
20	–	7276	676	–	1,47	1,27
22	–	18094	1219	–	1,48	1,28
24	–	39135	1641	–	1,48	1,26

Dominância

- h_2 é melhor que h_1 e muito melhor que a busca por aprofundamento iterativo.
- h_2 é sempre melhor que h_1 pois

$$\forall n \; h_2(n) \geq h_1(n)$$

- h_2 domina h_1
- Como ambas heurísticas são admissíveis, menos nós serão expandidos pela heurística dominante.
  - Escolhe nós mais próximos da solução.

Como criar heurísticas admissíveis?

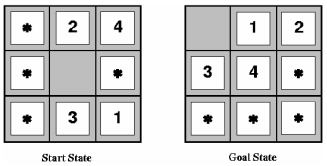
1. A solução de uma simplificação de um problema (problema relaxado) é uma heurística para o problema original.
   - **Admissível**: a solução do problema relaxado não vai superestimar a do problema original.
   - É **consistente** para o problema original se for consistente para o relaxado.

Exemplo: Quebra-cabeça de 8 peças

- h_1 daria a solução ótima para um problema "relaxado" em que as peças pudessem se deslocar para qualquer lugar.
- h_2 daria a solução ótima para um problema "relaxado" em que as peças pudessem se mover um quadrado por vez em qualquer direção.

Como criar heurísticas admissíveis?

2. Usar o custo da solução de um subproblema do problema original.



Calcular o custo da solução exata sem se preocupar com os *
Limite inferior do custo do problema completo

Como criar heurísticas admissíveis?

3. Banco de dados de padrões:
   - Armazenar o custo exato das soluções de muitos subproblemas.
   - Para um determinado estado procurar o subproblema referentes àquele estado.
   - Exemplo: todas as configurações das 4 peças na figura anterior.

Algoritmos de Busca Local

- Em muitos problemas de otimização o caminho para o objetivo é irrelevante.
  - Queremos apenas encontrar o estado objetivo, não importando a seqüência de ações.
  - Espaço de estados = conjunto de configurações completas.
    - Queremos encontrar a melhor configuração.
  - Neste caso podemos usar algoritmos de busca local.
    - Mantêm apenas o estado atual, sem a necessidade de manter a árvore de busca.

Exemplo: *n*-rainhas

- Colocar n rainhas em um tabuleiro $n \times n$, sendo que cada linha coluna ou diagonal pode ter apenas uma rainha.

Busca de Subida de Encosta

- "É como subir o Everest em meio a um nevoeiro durante uma crise de amnésia"

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
  inputs: problem, a problem
  local variables: current, a node
                   neighbor, a node
  current ← MAKE-NODE(INITIAL-STATE[problem])
  loop do
      neighbor ← a highest-valued successor of current
      if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
      current ← neighbor
```


Busca de Subida de Encosta

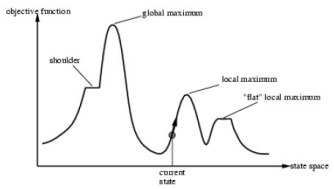
- Elevação é a função objetivo: queremos encontrar o máximo global.
- Elevação é o custo: queremos encontrar o mínimo global.
- O algoritmo consiste em uma repetição que percorre o espaço de estados no sentido do valor crescente (ou decrescente).
- Termina quando encontra um pico (ou vale) em que nenhuma vizinho tem valor mais alto.

Busca de Subida de Encosta

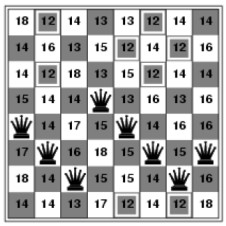
- Não mantém uma árvore, o nó atual só registra o estado atual e o valor da função objetivo.
- Não examina antecipadamente valores de estados além dos valores dos vizinhos imediatos do estado atual.

Busca de Subida de Encosta

- Problema: dependendo do estado inicial pode ficar presa em máximos (ou mínimos) locais.

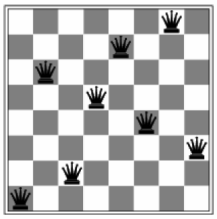


Busca de Subida de Encosta: Problema das 8-rainhas



- *h* = número de pares de rainhas que estão "se atacando", direta ou indiretamente
- *h = 17* para o estado acima
- Em cada quadrado, valor de h para cada sucessor possível obtido pela **movimentação de uma** rainha dentro de sua coluna

Busca de Subida de Encosta: Problema das 8-rainhas



- Um mínimo local com *h* = 1.

Subida de encosta: melhorias

- Movimento lateral para evitar platôs
  - Porém pode ocorrer repetição infinita, temos que impor um limite para o número de movimentos laterais.
- Subida de encosta com reinícios aleatórios.
  - Conduz várias buscas a partir de vários estados iniciais escolhidos aleatoriamente.
  - É completa, pois no pior acaso irá acabar gerando o estado objetivo como estado inicial, porém é ineficiente.

Busca de têmpera simulada (simulated annealing)

- Combina a subida de encosta com um percurso aleatório resultando em eficiência e completeza.
- Subida de encosta dando uma "chacoalhada" nos estados sucessores.
  - Estados com avaliação pior podem ser escolhidos com uma certa probabilidade.
  - Esta probabilidade diminui com o tempo.

Busca de têmpera simulada

- Escapa de máximos locais permitindo alguns passos "ruins" mas gradualmente decresce a sua freqüência.

```
function SIMULATED-ANNEALING( problem, schedule) returns a solution state
   inputs: problem, a problem
           schedule, a mapping from time to "temperature"
   local variables: current, a node
                    next, a node
                    T, a "temperature" controlling prob. of downward steps
   current ← MAKE-NODE(INITIAL-STATE[problem])
   for t ← 1 to ∞ do
       T ← schedule[t]
       if T = 0 then return current
       next ← a randomly selected successor of current
       ΔE ← VALUE[next] – VALUE[current]
       if ΔE > 0 then current ← next
       else current ← next only with probability e^(Δ E/T)
```


Propriedades da busca de têmpera simulada

- Pode-se provar que se T decresce devagar o suficiente, a busca pode achar uma solução ótima global com probabilidade tendendo a 1.
- Muito usada em projetos de circuitos integrados, layout de instalações industriais, otimização de redes de telecomunicações, etc.

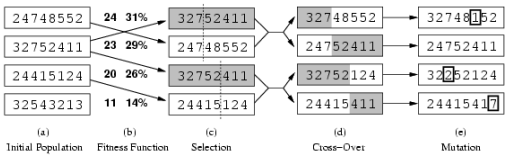
Busca em feixe local

- Manter *k* estados ao invés de um.
- Começa com *k* estados gerados aleatoriamente.
- A cada iteração, todos os sucessores dos *k* estados são gerados.
- Se qualquer um deles for o estado objetivo, a busca para; senão seleciona-se os *k* melhores estados da lista pra continuar.

Algoritmos genéticos

- Um estado sucessor é gerado através da combinação de dois estados pais.
- Começa com *k* estados gerados aleatoriamente (população).
- Um estado é representado por uma string de um alfabeto finito (normalmente strings de 0s e 1s).
- Função de avaliação (função de fitness). Valores mais altos pra estados melhores.
- Produz a próxima geração de estados por seleção, mutação e crossover.

Algoritmos genéticos



- Função de fitness: número de pares de rainhas que não estão se atacando (min = 0, max = 8 × 7/2 = 28)
- 24/(24+23+20+11) = 31%
- 23/(24+23+20+11) = 29% etc

Algoritmos genéticos