Sistemas Inteligentes

Busca com informação e exploração

Capítulo 4 – Russell & Norvig

Seção 4.1, 4.2 e 4.3

Busca com informação (ou heurística)

- Utiliza conhecimento específico sobre o problema para encontrar soluções de forma mais eficiente do que a busca cega.
  - Conhecimento específico além da definição do problema.
- Abordagem geral: busca pela melhor escolha.
  - Utiliza uma função de avaliação para cada nó.
  - Expande o nó que tem a função de avaliação mais baixa.
  - Dependendo da função de avaliação, a estratégia de busca muda.

Busca pela melhor escolha

- Idéia: usar uma função de avaliação $f(n)$ para cada nó.
  - estimativa do quanto aquele nó é desejável
  - → Expandir nó mais desejável que ainda não foi expandido

- Implementação:

  Ordenar nós na borda em ordem decrescente de acordo com a função de avaliação

- Casos especiais:
  - Busca gulosa pela melhor escolha
  - Busca A*

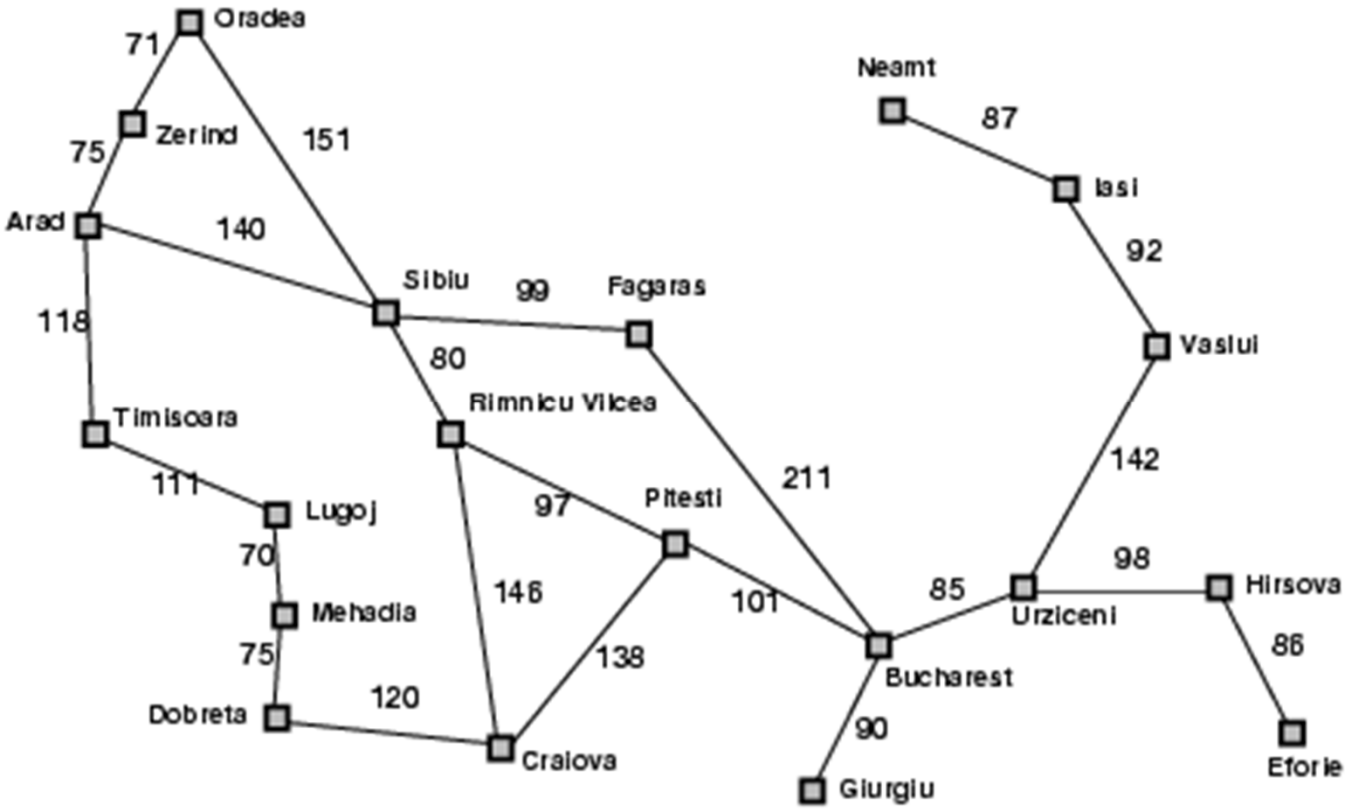
Busca gulosa pela melhor escolha

- Função de avaliação $f(n) = h(n)$ (heurística)

  = estimativa do custo de n até o objetivo

  ex., $h_{DLR}(n)$ = distância em linha reta de n até Bucareste.
- Busca gulosa pela melhor escolha expande o nó que parece mais próximo ao objetivo de acordo com a função heurística.

Romênia com custos em km



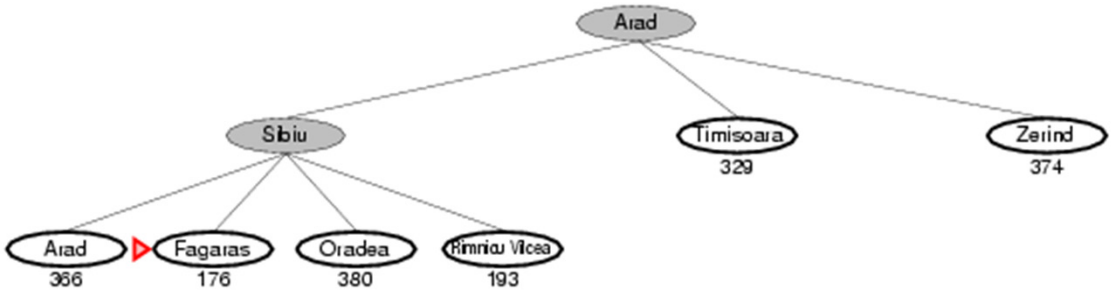
Distância em linha reta para Bucareste

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

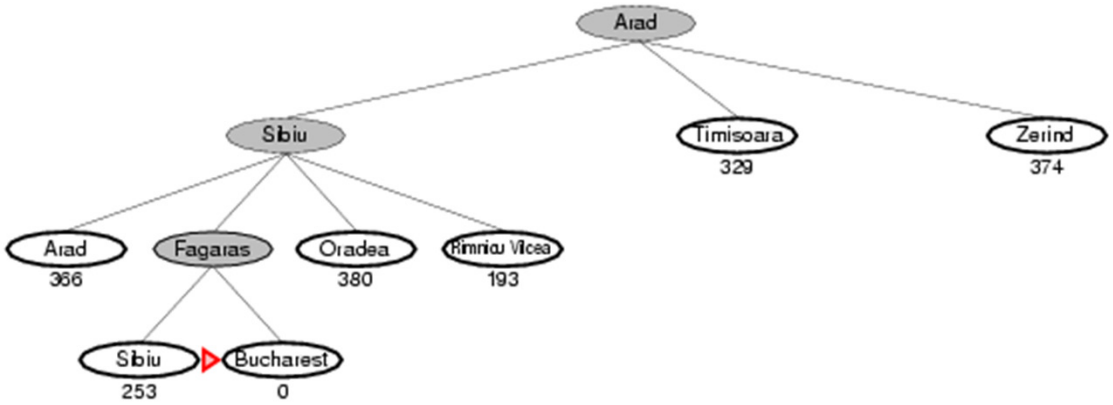
Exemplo de busca gulosa pela melhor escolha

Exemplo de busca gulosa pela melhor escolha

Exemplo de busca gulosa pela melhor escolha



Exemplo de busca gulosa pela melhor escolha



Busca gulosa pela melhor escolha

- Não é ótima, pois segue o melhor passo **considerando somente o estado atual.**
  - Pode haver um caminho melhor seguindo algumas opções piores em alguns pontos da árvore de busca.
- Minimizar $h(n)$ é suscetível a falsos inícios.
  - Ex. Ir de Iasi a Fagaras
    - Heurística sugerirá ir a Neamt, que é um beco sem saída.
    - Se repetições não forem detectadas a busca entrará em loop.

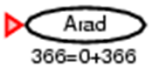
Propriedades da busca gulosa pela melhor escolha

- Completa? Não – pode ficar presa em loops, ex., Iasi → Neamt → Iasi → Neamt
- Tempo? O(b^m) no pior caso, mas uma boa função heurística pode levar a uma redução substancial
- Espaço? O(b^m) mantém todos os nós na memória
- Ótima? Não

Busca A*

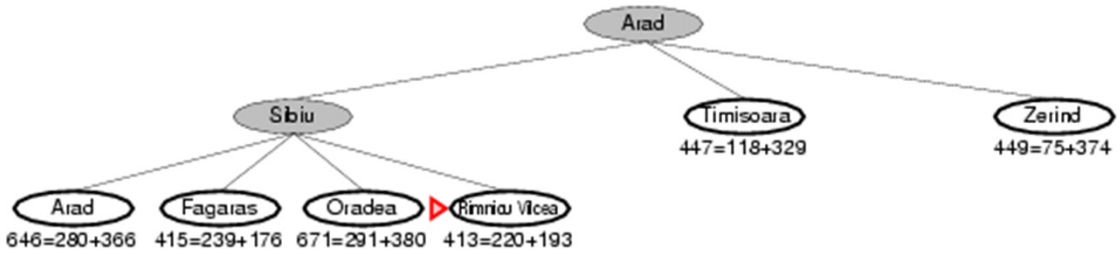
- “A estrela”
- Ideia: evitar expandir caminhos que já são caros
- Função de avaliação $f(n) = g(n) + h(n)$
  - $g(n)$ = custo até o momento para alcançar n
  - $h(n)$ = custo estimado de n até o objetivo
  - $f(n)$ = custo total estimado do caminho através de n até o objetivo.

Exemplo de busca A*

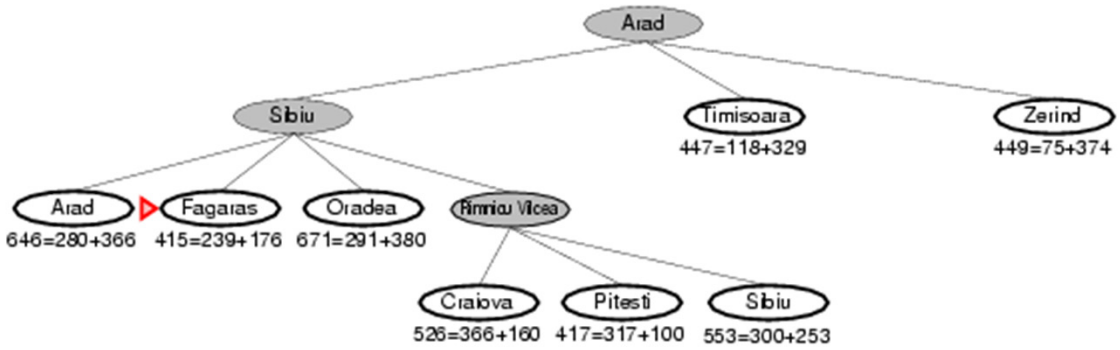


Exemplo de busca A*

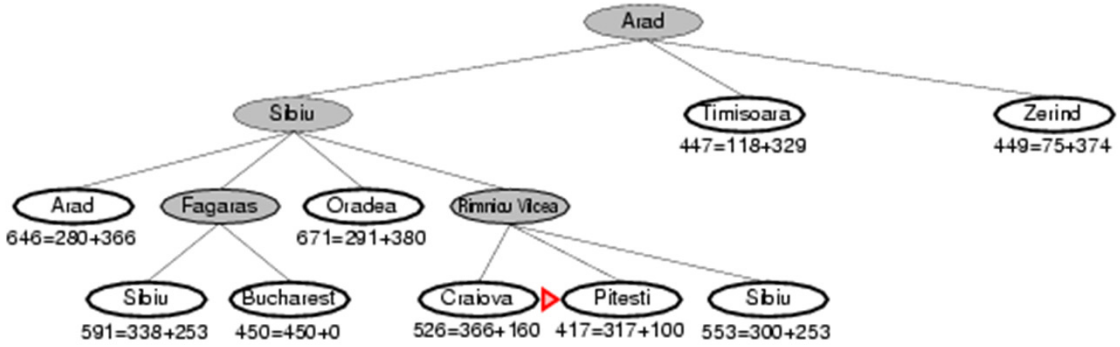
Exemplo de busca A*



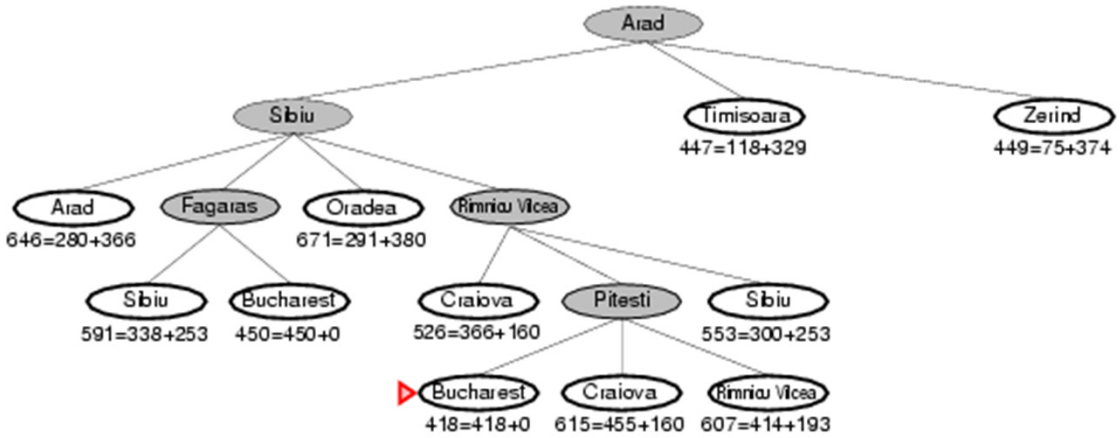
Exemplo de busca A*



Exemplo de busca A*



Exemplo de busca A*



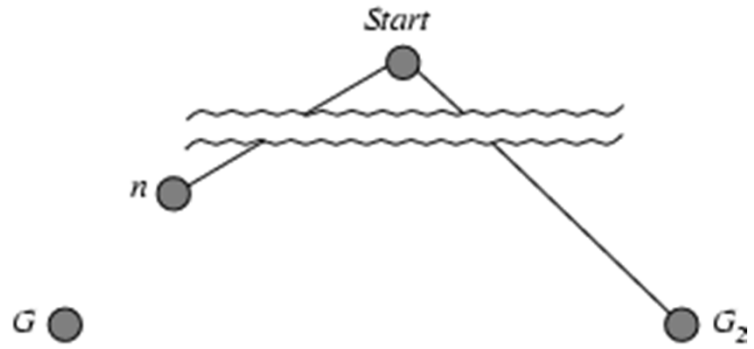
Heurística Admissível

- 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.

Prova que A* é ótima com heurística admissível

- Assuma um **nó objetivo não-ótimo** G_2, e seja C^* o custo da solução ótima. Então, como G_2 não é ótimo e $h(G_2) = 0$, sabemos que:

$$f(G_2) = g(G_2) + h(G_2)$$
$$= g(G_2) > C^*$$



- Considere qualquer nó de borda **n** que esteja num caminho de solução ótimo. Se $h(n)$ ***não superestimar o custo de completar o caminho de solução***, então: $f(n) = g(n) + h(n) \leq C^*$.

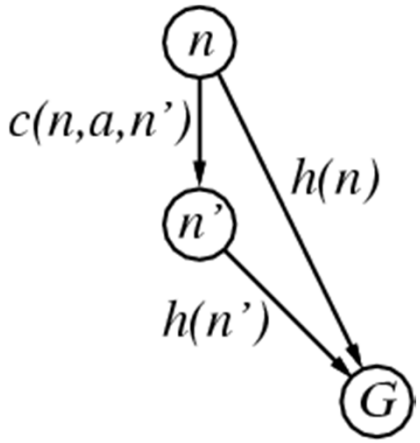
Prova que A* é ótima com heurística admissível (cont.)

- Logo, se $f(n) \leq C^* < f(G_2)$, G_2 não será expandido e A* deve retornar uma solução ótima.
- Isso vale para busca em árvore, para outras estruturas de busca pode não valer.
- Na busca em grafos temos que assegurar que o caminho ótimo para qualquer estado repetido seja o primeiro a ser seguido.
  - Requisito extra para $h(n)$: consistência

Consistência (ou monotonicidade)

- Uma heurística é consistente (ou monotônica) se para cada nó *n*, cada sucessor *n'* de *n* gerado por qualquer ação *a*,

  $h(n) \leq c(n,a,n') + h(n')$ (desigualdade triangular)



- Se *h* é consistente, temos

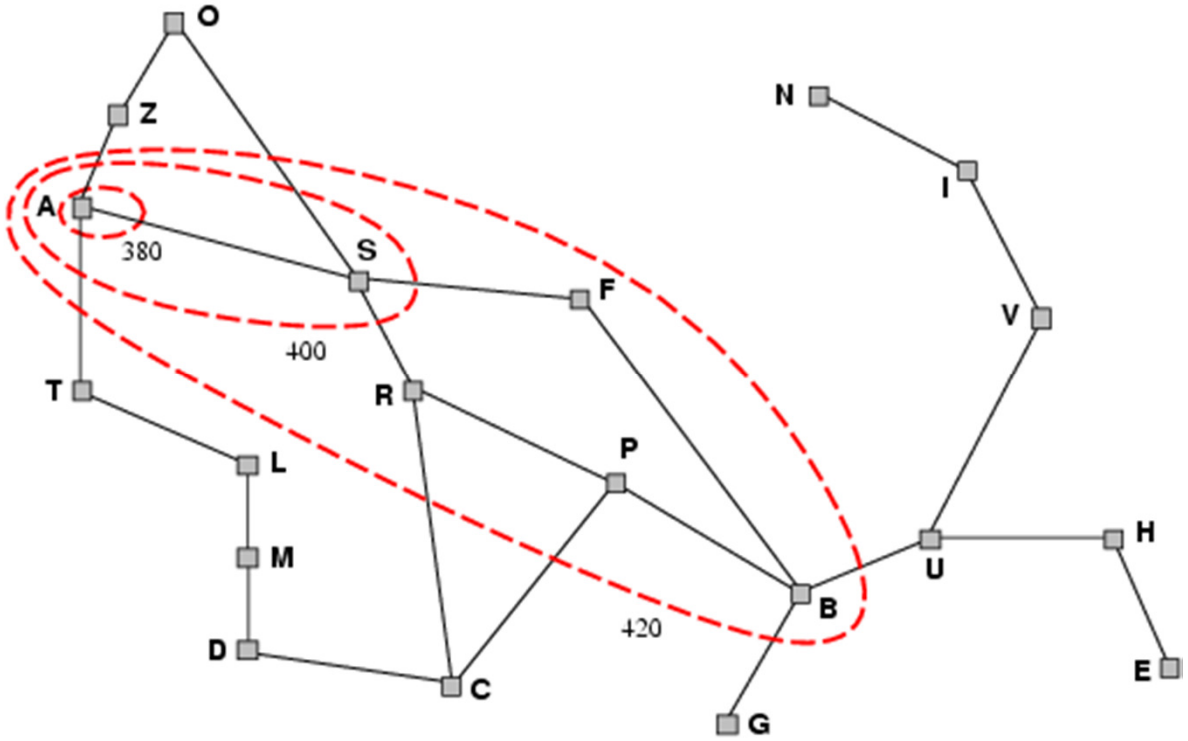
  $f(n') = g(n') + h(n')$
  $= g(n) + c(n,a,n') + h(n')$
  $\geq g(n) + h(n)$
  $= f(n)$

- Isto é, $f(n)$ é não-decrescente ao longo de qualquer caminho.

- Teorema: Se $h(n)$ é consistente, A* usando `BUSCA-EM-GRAFOS` é ótima.

A* é ótima com heurística consistente

- A* expande nós em ordem crescente de valores de *f*.
- Gradualmente adiciona "*contornos*" de nós.
- Contorno *i* tem todos os nós com $f=f_i$, onde $f_i < f_{i+1}$



Se $h(n)=0$ temos uma busca de custo uniforme ⇒ círculos concêntricos.

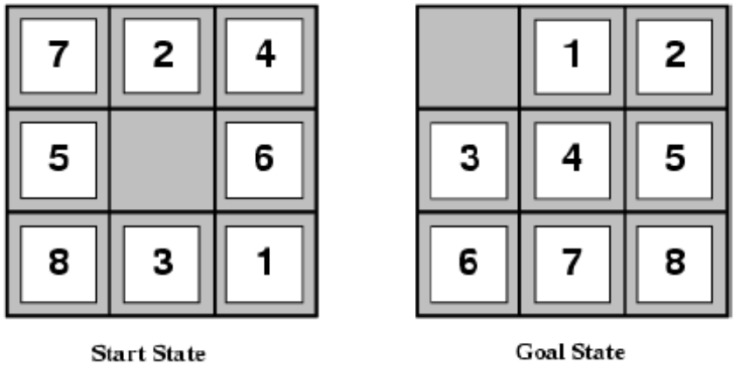
Quanto melhor a heurística mais direcionados ao objetivo serão os círculos

Propriedades da Busca A*

- Completa? Sim (a não ser que exista uma quantidade infinita de nós com $f \leq f(G)$)
- Tempo? Exponencial no pior caso
- Espaço? Mantém todos os nós na memória
- Ótima? Sim
- Otimamente eficiente
  - Nenhum outro algoritmo de busca ótimo tem garantia de expandir um número de nós menor que A*. Isso porque qualquer algoritmo que não expande todos os nós com $f(n) < C^*$ corre o risco de omitir uma solução 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**
  - O fator de ramificação efetiva **b*** a partir de

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

  - Exemplo: se A* encontra uma solução à profundidade 5 utilizando 52 nós, então o **b***=1,92
  - Heurística bem projetada teria **b*** próximo de 1

Exemplo:
Quebra-cabeça de 8 peças

	Custo da Busca (nós gerados)			Fator de Ramificação Efetivo		
d	IDS	A*(h_1)	A*(h_2)	IDS	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

IDS → busca por aprofundamento iterativo

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?

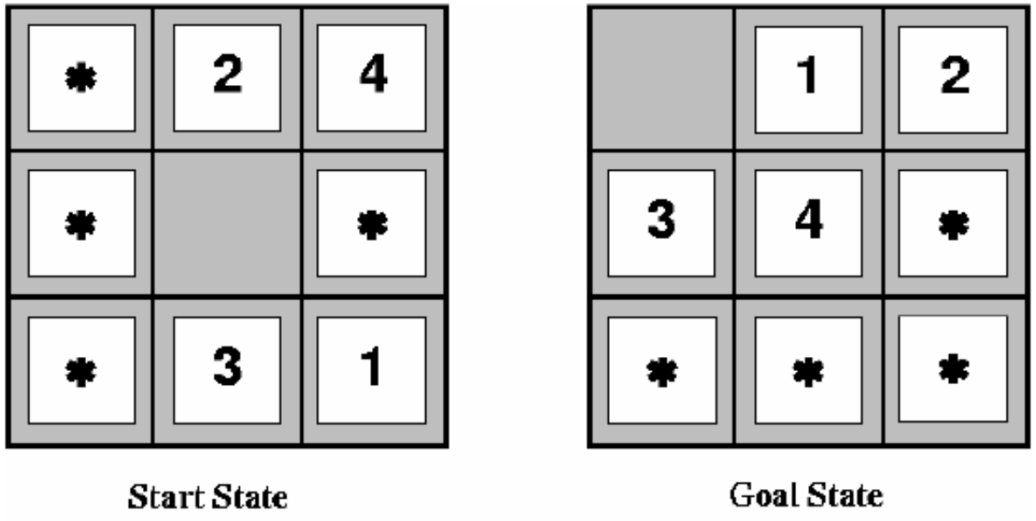
1. A solução de uma simplificação de um problema (problema relaxado) é uma heurística para o problema original.
   - **Admissível**: a solução do problema relaxado não vai superestimar a do problema original.
   - É **consistente** para o problema original se for consistente para o relaxado.

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.

Busca de Subida de Encosta

- Valor crescente até um pico (nenhum vizinho mais alto)

- A busca de subida de encosta não examina valores de estados além dos vizinhos imediatos
  - "É como subir o Everest em meio a um nevoeiro durante uma crise de amnésia"

```
função SUBIDA-DE-ENCOSTA(problema) retorna um estado que é um máximo local
  corrente ← CRIAR-NÓ(ESTADO-INICIAL[problema])
  repita
    vizinho ← um sucessor de corrente com valor mais alto
    se VALOR[vizinho] ≤ VALOR[corrente] então retornar ESTADO[corrente]
    corrente ← vizinho
```

Busca de Subida de Encosta

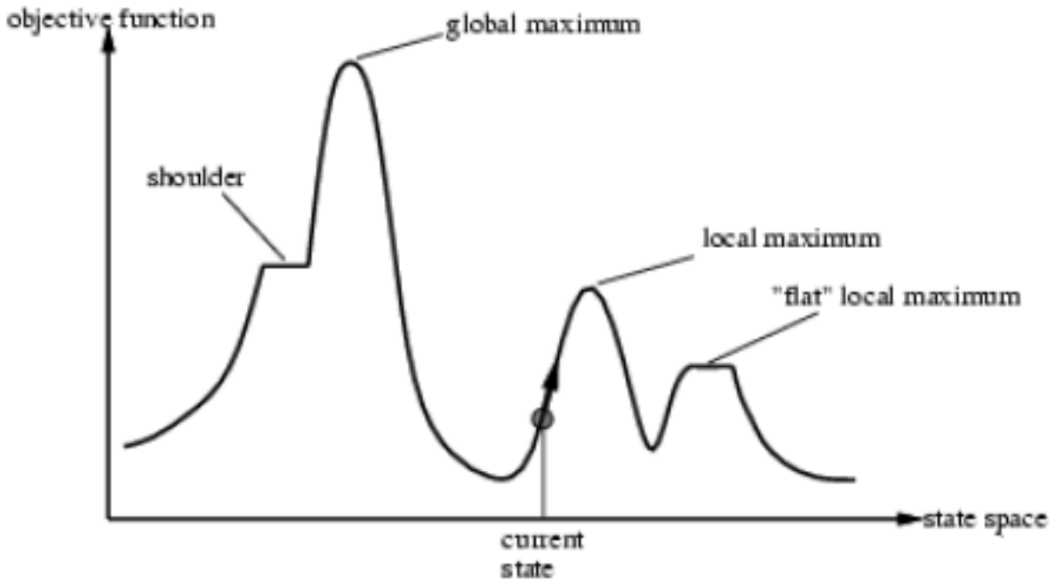
- Elevação é a função objetivo: queremos encontrar o máximo global.
- Elevação é o custo: queremos encontrar o mínimo global.
- O algoritmo consiste em uma repetição que percorre o espaço de estados no sentido do valor crescente (ou decrescente).
- Termina quando encontra um pico (ou vale) em que nenhuma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

Busca de Subida de Encosta

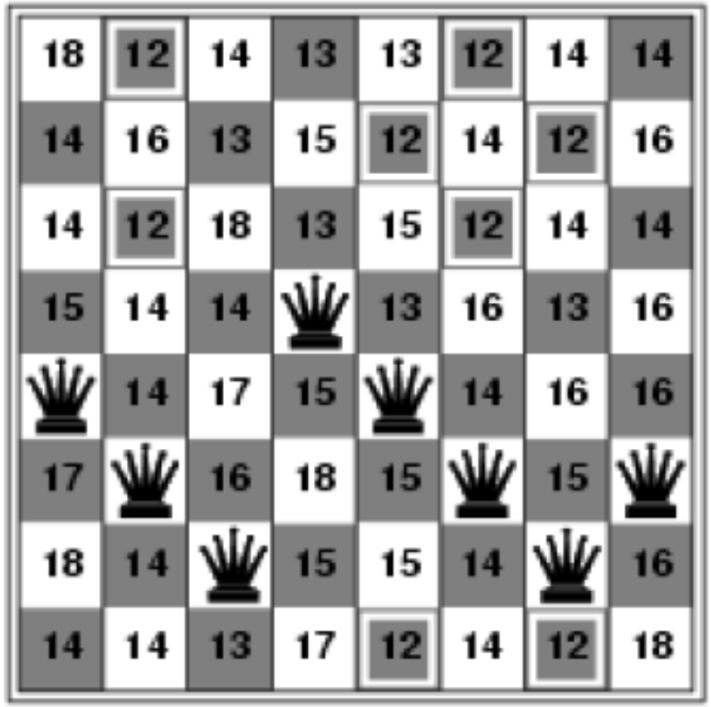
- Não mantém uma árvore, o nó atual só registra o estado atual e o valor da função objetivo.
- Não examina antecipadamente valores de estados além dos valores dos vizinhos imediatos do estado atual.

Busca de Subida de Encosta

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

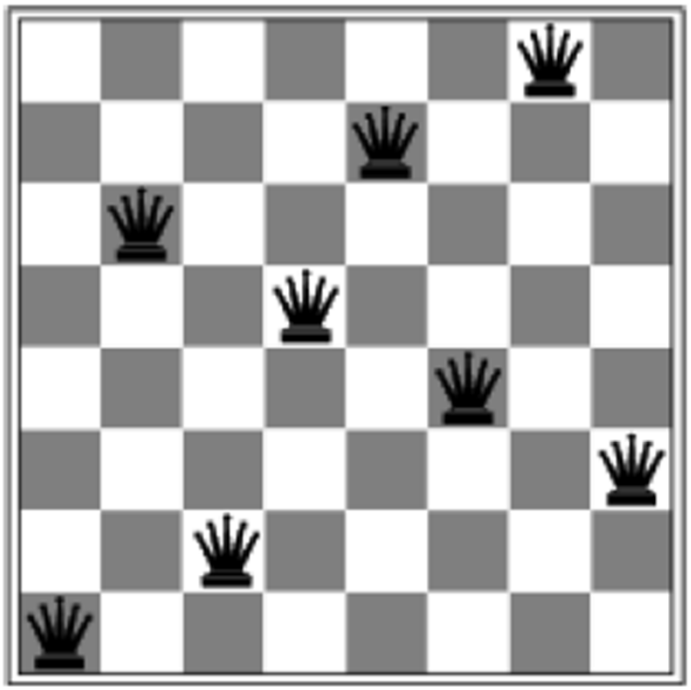


Busca de Subida de Encosta: Problema das 8-rainhas



- *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.

função TÊMPERA-SIMULADA(*problema, escalonamento*) **retorna** um estado solução
  entradas: *problema*, um problema
escalonamento, um mapeamento de tempo para “temperatura”
  atual ← CRIAR-NÓ(*problema*.ESTADO-INICIAL)
  para $t = 1$ **até** ∞ **faça**
    $T \leftarrow$ *escalonamento*$[t]$
    se $T = 0$ **então retornar** *corrente*
    próximo ← um sucessor de *atual* selecionado aleatoriamente
    $\Delta E \leftarrow$ *próximo*.VALOR – *atual*.VALOR
    se $\Delta E > 0$ **então** *atual* ← *próximo*
    senão *atual* ← *próximo* somente com probabilidade $e^{\Delta E/T}$

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 em vez 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; se não seleciona-se os *k* melhores estados da lista pra continuar.

Algoritmos genéticos

- Um estado sucessor é gerado por meio da combinação de dois estados pais.
- Começa com *k* estados gerados aleatoriamente (população).
- Um estado é representado por um 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