

PROCURA LOCAL

CAP 4 (4.1 E 4.2)

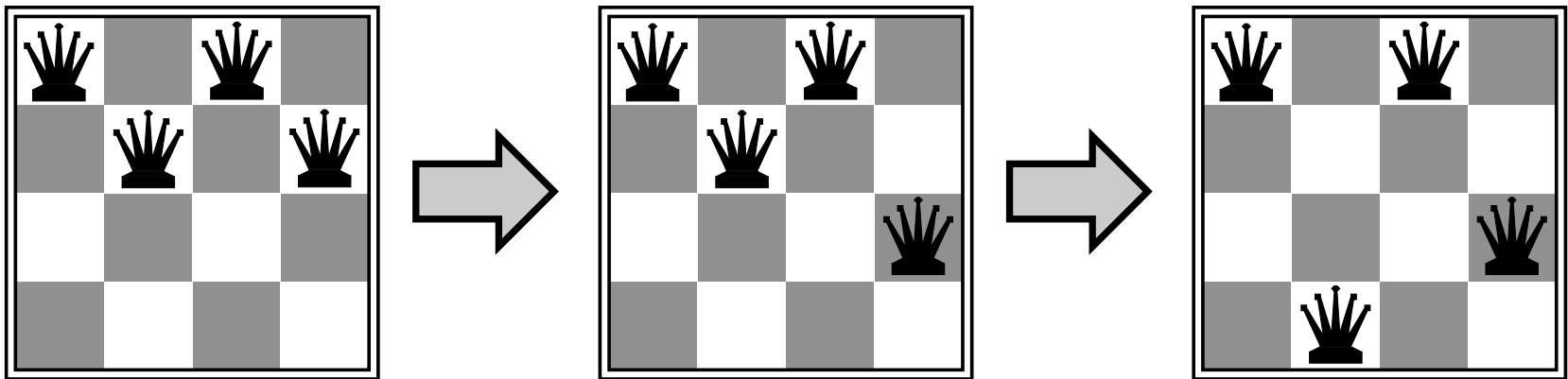
Parcialmente adaptado de
<http://aima.eecs.berkeley.edu>

Algoritmos de procura local

- Em muitos problemas de optimização o **caminho** para a solução irrelevante. O estado objectivo é a própria solução.
- O espaço de estado = conjunto de configurações “completas” (soluções candidatas)
 - encontrar a configuração óptima, e.g., TSP ou,
 - encontrar configuração que obedece a certas restrições, e.g., horário
- Nestas situações, podem-se usar algoritmos de **procura local**: mantém-se um único estado “corrente”, e tenta-se alterá-lo para melhorar a sua qualidade
- Espaço constante, apropriados para procura online e offline em espaços discretos e **contínuos**, assim como para resolução de problemas de optimização.

Problema das n -rainhas

- Colocar n rainhas num tabuleiro $n \times n$ sem que quaisquer duas rainhas se ataquem mutuamente
- Deslocar uma rainha para reduzir o número de conflitos



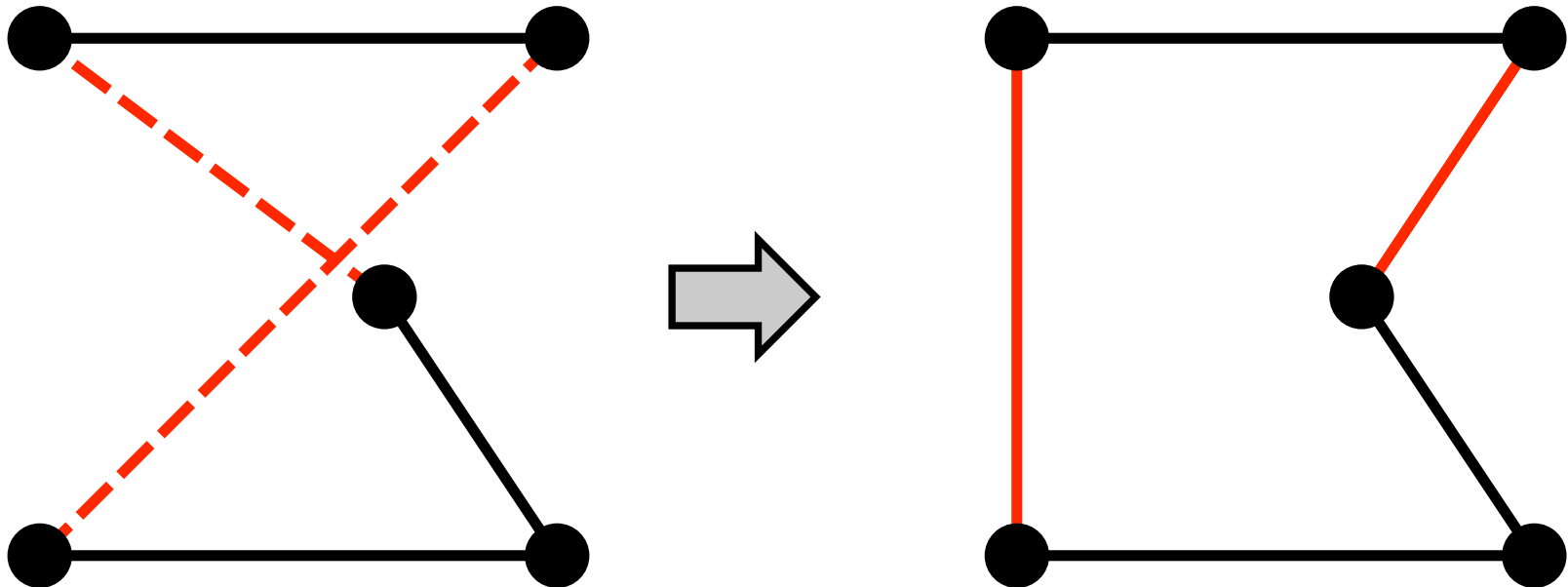
- **Nota:** este problema pode ser resolvido em tempo linear!

Problema da satisfatibilidade booleana

- Dado um conjunto de cláusulas proposicionais indicar se esse conjunto de cláusulas é satisfazível
 - Existe uma atribuição de valores lógicos a variáveis proposicionais que torna todas as cláusulas verdadeiras?
 - Método: atribuir valores aleatoriamente às variáveis e ir trocando o seu valor.
- Considere-se o conjunto de cláusulas:
 - $\neg A \vee B$
 - $A \vee \neg B$
 - $\neg A \vee \neg B \vee \neg C$
 - $A \vee B$
 - $\neg D \vee C$
 - $\neg E \vee C$
- Solução?
 - $A=B=\text{True}, C=D=E=\text{False}$
- **Nota:** este problema é NP-completo!

Problema do Caixeiro Viajante

- Começar com um circuito completo arbitrário, efetuar trocas aos pares



- **Nota:** minimizar custo é problema NP-difícil!

Problemas combinatórios

- Problemas que tipicamente envolvem encontrar grupos, ordenações ou atribuições de um número finito de objetos discretos que satisfazem um conjunto de condições ou restrições.
- Habitualmente, o espaço de soluções candidatas para uma instância particular é pelo menos exponencial no tamanho dessa instância.

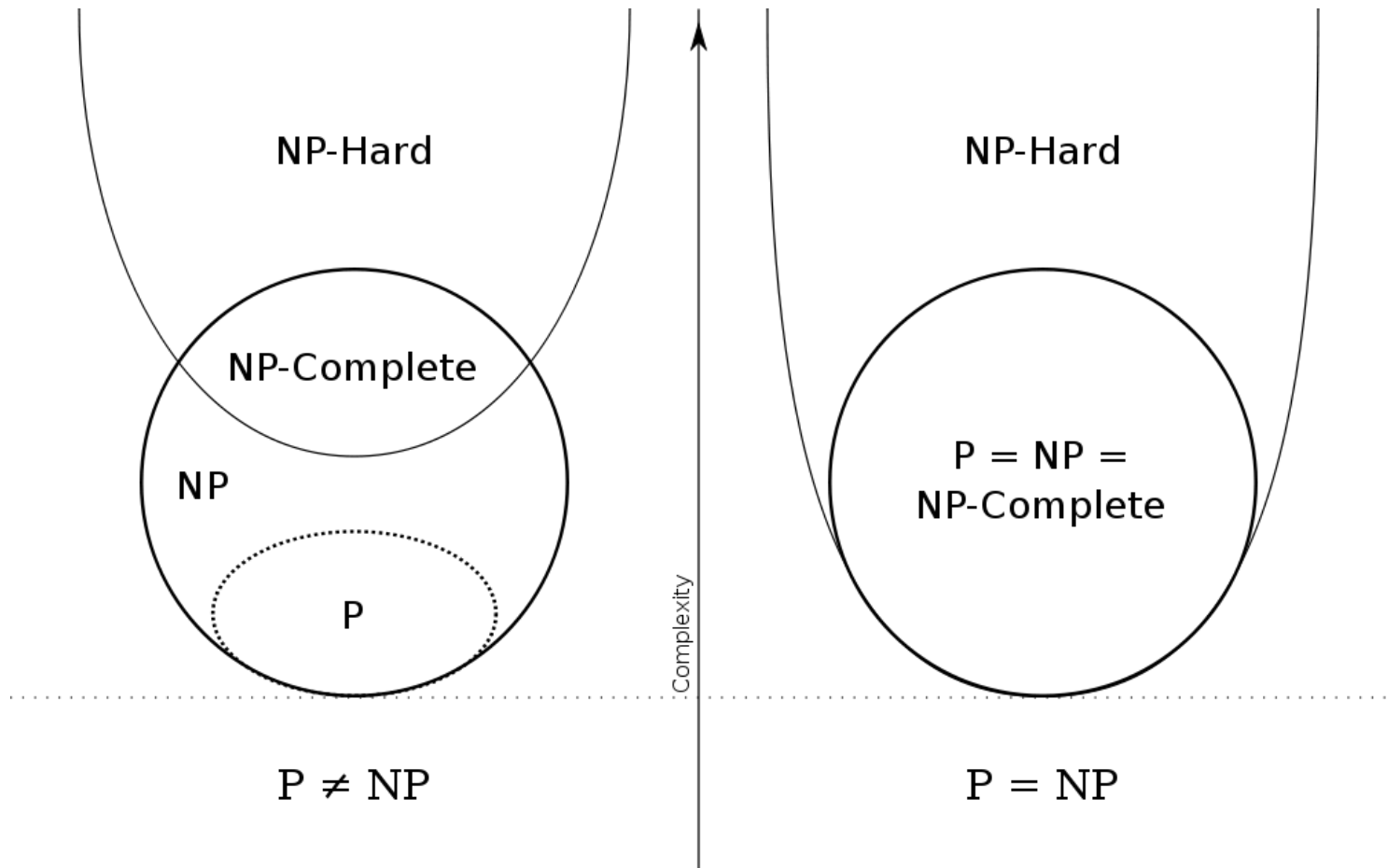
Tipos de problemas combinatórios

- Problemas de Decisão (têm resposta sim/não)
 - Variante de decisão: saber se existe ou não solução para o problema
 - Variante de procura: encontrar a solução, caso exista.
- Problemas de Optimização (encontrar solução que optimiza valor de **função objectivo**)
 - Variante de procura: encontrar a solução óptima
 - Variante de avaliação: indicar qual o valor mínimo/máximo da função objectivo
- Qualquer problema de optimização origina um problema de decisão associado, a partir de um limite **b** fornecido.
 - Se o problema é de minimização, saber se existe uma solução com valor inferior ou igual ao **b** dado.
 - Se o problema é de maximização, saber se existe uma solução com valor superior ou igual ao **b** dado.

Classes de Complexidade

- Habitualmente definidas a partir de problemas de decisão e para o pior caso.
- Classe de problemas **P**
 - Problemas de decisão que podem ser resolvidos por uma máquina **determinista** em tempo polinomial no tamanho da instância
- Classe de problemas **NP**
 - Problemas de decisão que podem ser resolvidos por uma máquina **não determinista** em tempo polinomial no tamanho da instância
 - Equivalentemente, se “adivinhar” a solução consigo verificá-la em tempo polinomial. Logo P contido em NP.
- Classe de problemas **NP-difíceis**
 - Aqueles que são mais difíceis do que todos os problemas em NP
 - Se o problema pertencer a NP, então o problema diz-se **NP-completo**.
- Os problemas NP-completos são os mais difíceis da classe NP!

P = NP ?

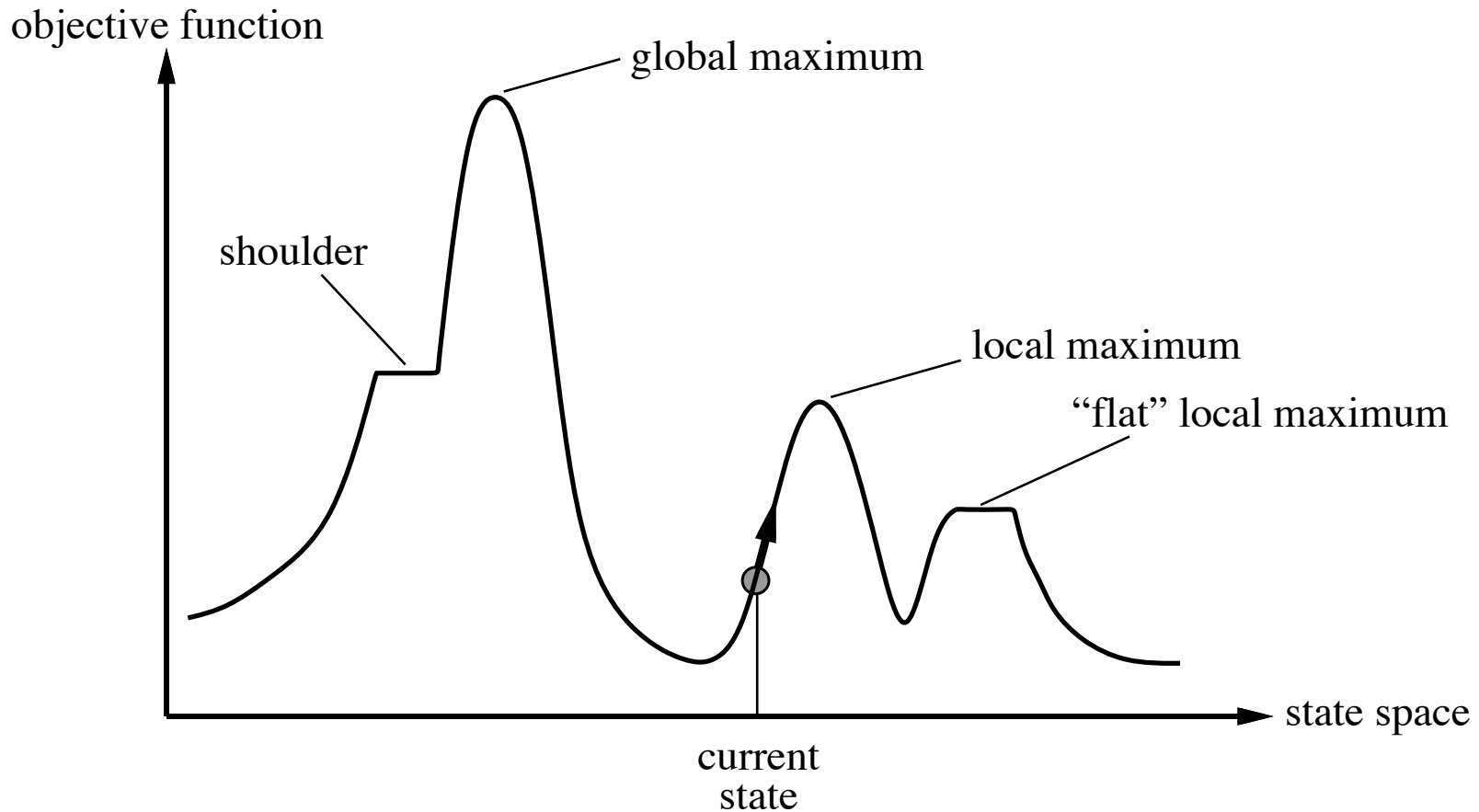


<http://en.wikipedia.org/wiki/NP-complete>

Como atacar problemas difíceis?

- Encontrar subclasses do problema que sejam interessantes e de resolução eficiente
- Usar algoritmos de aproximação eficientes
- Recorrer a aproximações estocásticas

A paisagem do espaço de estados



- Tanto podemos definir o problema como maximizar a função objectivo (proveito) ou minimizar o custo (heurística)

Trepa-colinas (Hill climbing)

- Escalar o Evereste com nevoeiro denso e 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
```

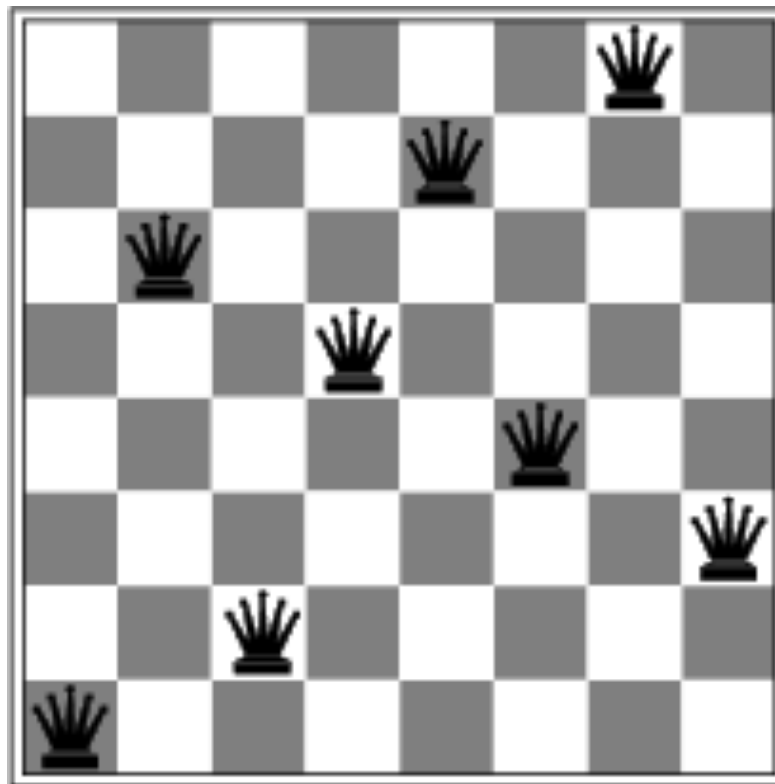
- A escolha é normalmente aleatória entre os sucessores com o mesmo valor para a função objectivo.

Trepa-colinas: problema das 8-rainhas

18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	♚	13	16	13	16
♚	14	17	15	♚	14	16	16
17	♚	16	18	15	♚	15	♚
18	14	♚	15	15	14	♚	16
14	14	13	17	12	14	12	18

- h = número de pares de rainhas que se atacam mutuamente
- $h = 17$ para o estado apresentado

Trepa-colinas: problema das 8-rainhas



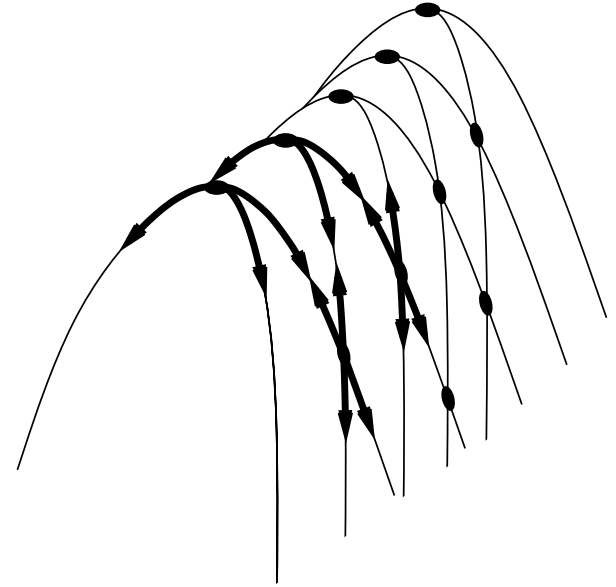
- Um mínimo local com $h = 1$

Trepa-colinas com recomeços aleatórios

- O algoritmo Trepa-colinas tem uma taxa de sucesso de 14% no problema das 8-rainhas, sendo bastante rápido:
 - 4 passos em média no caso de sucesso e 3 no caso de insucesso: - nada mau para um espaço de estados de 8^8 (cerca de 17 milhões)
- Pode-se utilizar um número limitado de movimentos laterais para se tentar sair de planaltos (substituir teste de $\leq$ por $<$ e contar sua utilização), mas não funciona para planaltos que correspondem a máximos locais.
 - Para o problema das 8-rainhas a utilização de movimentos laterais limitada a 100 aumenta a probabilidade de sucesso de 14% para 94% (21 passos no caso de sucesso e 64 no caso de insucesso).
- Para evitar ficarmos presos em máximos locais, tenta-se novamente com um novo estado inicial gerado aleatoriamente e guarda-se o melhor deles ao fim de um número determinado de iterações.
 - Permite resolver problemas das 3000000-rainhas em menos de 1 minuto.
- O sucesso deste algoritmo depende muito da paisagem do espaço de estados: poucos máximos locais e planaltos é a situação ideal.

Trepa-colinas

- Problemas:
 - pode ficar preso facilmente em máximos locais
 - travessia difícil em cristas de máximos locais
 - passeios aleatórios em planaltos
- Em espaços contínuos, problemática na escolha do tamanho do passo, convergência lenta.



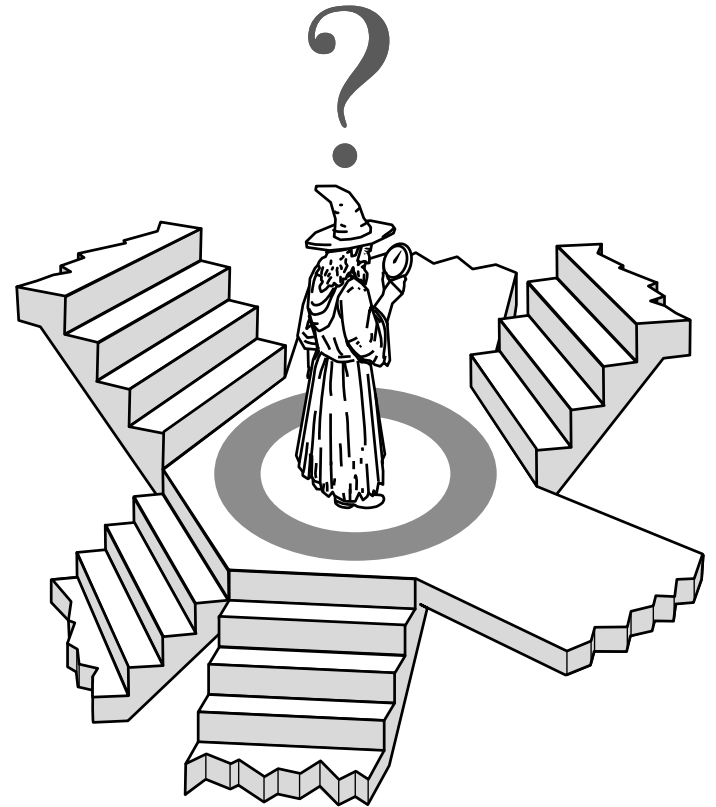
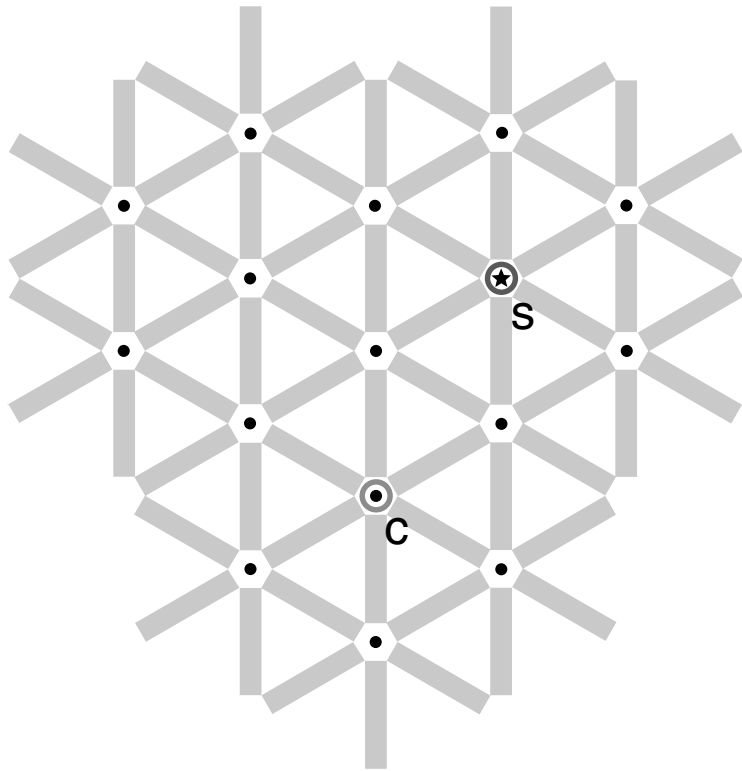
Variantes do Trepa-colinas

- Trepa-colinas estocástico
 - Seleciona o próximo sucessor aleatoriamente, entre os sucessores que representam uma melhoria (e.g. com probabilidade dependendo da inclinação - melhoria). Mais lento mas pode devolver melhores soluções para alguns problemas.
- Trepa-colinas de primeira escolha
 - Vai gerando os sucessores aleatoriamente e escolhe o primeiro que melhore a função objectivo. Adequado em situações cuja vizinhança é muito grande (e.g. infinita em espaços contínuos)

Procura Local Estocástica

- O trepa-colinas é um exemplo elementar de um algoritmo de **procura local estocástica**:
 - A procura é efectuada no espaço de soluções candidatas
 - A procura inicia-se a partir de uma solução candidata
 - O processo continua movendo-se iterativamente de uma solução candidata para outra na sua vizinhança
 - A decisão em cada passo é tomada tendo em conta apenas informação local limitada.
 - A inicialização e a decisão podem ser aleatórias (probabilísticas)
 - Pode-se utilizar memória adicional, por exemplo para guardar um número limitado de soluções candidatas visitadas recentemente.

Procura Local Estocástica



SLS-Decision(π)

```
procedure SLS-Decision( $\pi$ )  
  input: problem instance  $\pi$  in  $\Pi$   
  output: feasible solution in  $S'(\Pi)$  or  $\emptyset$   
  ( $s, m$ ) := init( $\pi$ );  
  while not terminate( $\pi, s, m$ ) do  
    ( $s, m$ ) := step( $\pi, s, m$ );  
  end  
  if  $s$  in  $S'(\Pi)$  then  
    return  $s$   
  else  
    return  $\emptyset$   
  end
```

SLS-Maximisation(π)

```
procedure SLS-Maximisation( $\pi$ )  
  input: problem instance  $\pi$  in  $\Pi$   
  output: feasible solution in  $S'(\Pi)$  or  $\emptyset$   
  ( $s, m$ ) := init( $\pi$ );  
  incumbent :=  $s$ ;  
  while not terminate( $\pi, s, m$ ) do  
    ( $s, m$ ) := step( $\pi, s, m$ );  
    if  $f(\pi, s) > f(\pi, \text{incumbent})$  then  
      incumbent :=  $s$ ;  
    end  
  end  
  if incumbent in  $S'(\Pi)$  then  
    return incumbent  
  else  
    return  $\emptyset$   
  end
```

Problema central

- Como evitar ficar preso em mínimos/máximos locais?
- Exige um equilíbrio entre estratégias de
 - **Intensificação**: tentativa de melhoramento da solução na vizinhança da solução candidata corrente.
 - **Diversificação**: tenta evitar a estagnação em zonas do espaço de procura que não contêm soluções de alta qualidade
- As duas estratégias definem os inúmeros algoritmos existentes.
- Maiores vizinhanças contêm mais e melhores soluções candidatas mas requerem mais tempo para as encontrar (normalmente o tempo cresce exponencialmente...)

Outros exemplos

- Uninformed Random Picking
 - Inicia-se com uma solução candidata obtida aleatoriamente
 - Transita aleatoriamente para qualquer solução candidata com distribuição uniforme
 - Só diversificação
- Uninformed Random Walking
 - Inicia-se com uma solução candidata obtida aleatoriamente
 - Transita aleatoriamente para qualquer solução candidata na sua vizinhança
 - Só diversificação
- Randomised Iterative Improvement
 - Inicia-se com uma solução candidata obtida aleatoriamente
 - De acordo com um parâmetro fixado wp , opta entre melhorar a solução ou mover-se aleatoriamente para uma solução candidata na vizinhança.
 - Intensificação e diversificação controlados por wp (**walk probability**).

Recristalização simulada

- Fugir de máximos locais permitindo alguns movimentos “piores” mas reduzindo gradualmente o seu tamanho e frequência

```
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
     $\Delta E \leftarrow \text{VALUE}[\textit{next}] - \text{VALUE}[\textit{current}]$ 
    if  $\Delta E > 0$  then current ← next
    else current ← next only with probability  $e^{\Delta E/T}$ 
```

Recristalização Simulada

- Seleciona-se um movimento aleatoriamente em cada iteração
- Transita-se para esse estado se melhor do que o estado atual
- A probabilidade de se movimentar para um estado pior decresce exponencialmente com o valor da mudança, ou seja

$$e^{-\frac{\Delta E}{T}}$$

- A temperatura T altera-se de acordo com o escalonamento definido
 - Temperatura inicial T_0
(pode depender das propriedades da instância)
 - Atualização da temperatura
(e.g. arrefecimento geométrico $T := \alpha * T$)
 - Número de passos a cada temperatura (normalmente múltiplo da dimensão da vizinhança)
- Critério de terminação normalmente baseado no rácio entre estados propostos versus aceites.

Propriedades da recristalização simulada

- Se a temperatura for diminuída suficientemente devagar atinge-se o melhor estado, com probabilidade que se aproxima de 1.
- Desenvolvido por Metropolis et al., 1953, para modelação de processos físicos
- Muito utilizado no desenho de VLSI, escalonamento de viagens aéreas, etc.

Procura local em feixe

- Mantêm-se k soluções candidatas em vez de apenas 1
- Começa-se com k soluções geradas aleatoriamente
- Em cada iteração, as vizinhanças das k soluções candidatas são geradas
- Se alguma é o objectivo, parar; senão seleccionam-se as k melhores e repete-se.

Procura local em espaços contínuos

- Muitos problemas podem ser representados como maximização/minimização de funções multidimensionais em $\mathbb{R}^n$.
- Existem inúmeras técnicas mas iremos abordar brevemente aquelas baseadas no gradiente de uma função $f(x_1, \dots, x_n)$ que se supõe diferenciável em $\mathbb{R}^n$.
- O gradiente da função f , denotado por ∇f define-se por:

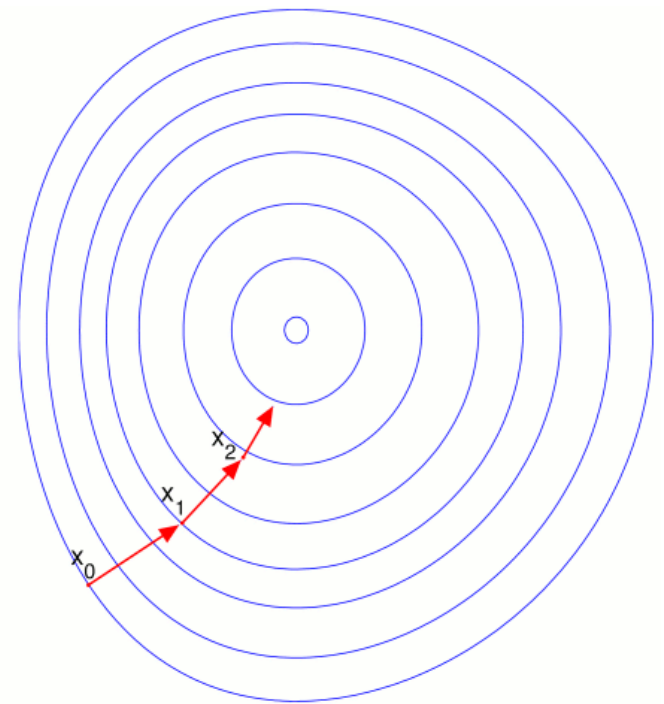
$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_n} \right)$$

Subida pelo gradiente

- Partindo de uma solução inicial x_0
- Executa-se iterativamente o algoritmo

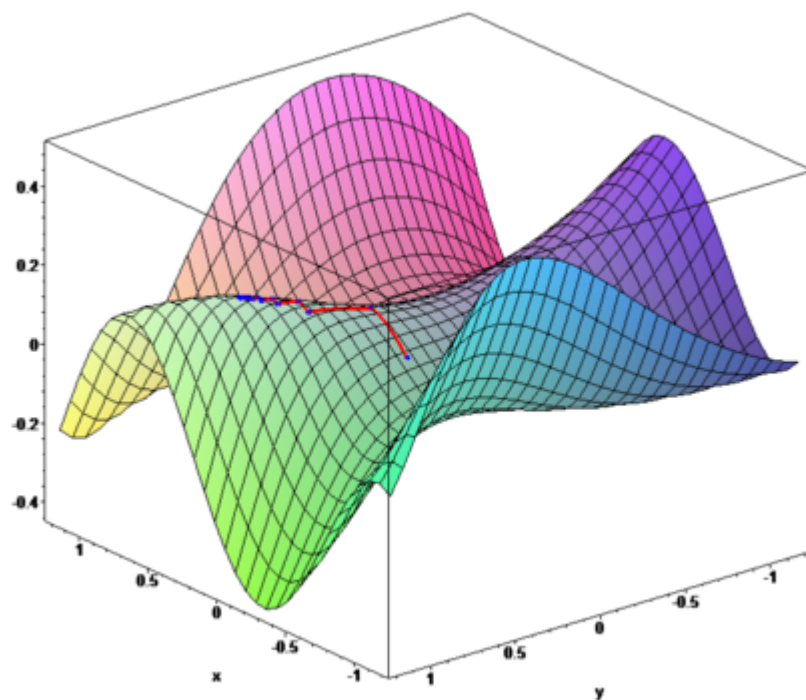
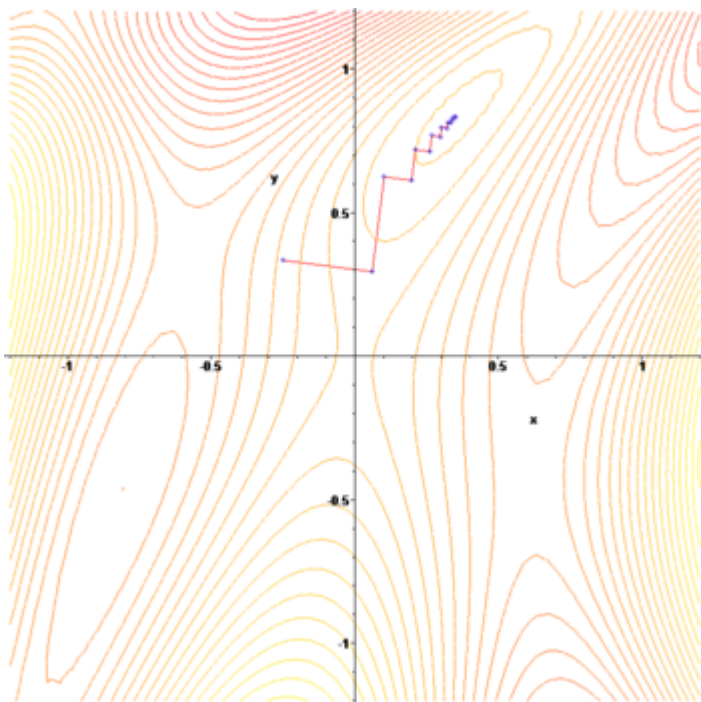
$$\mathbf{x}_{n+1} = \mathbf{x}_n + \gamma_n \nabla F(\mathbf{x}_n), \quad n \geq 0.$$

- Até que a variação entre $|\mathbf{x}_{n+1} - \mathbf{x}_n|$ atinja um erro predeterminado.
- Podemos variar em cada passo o parâmetro γ_n , que deve ser suficientemente pequeno



Exemplo: maximização de função

$$F(x, y) = \sin\left(\frac{1}{2}x^2 - \frac{1}{4}y^2 + 3\right) \cos(2x + 1 - e^y)$$

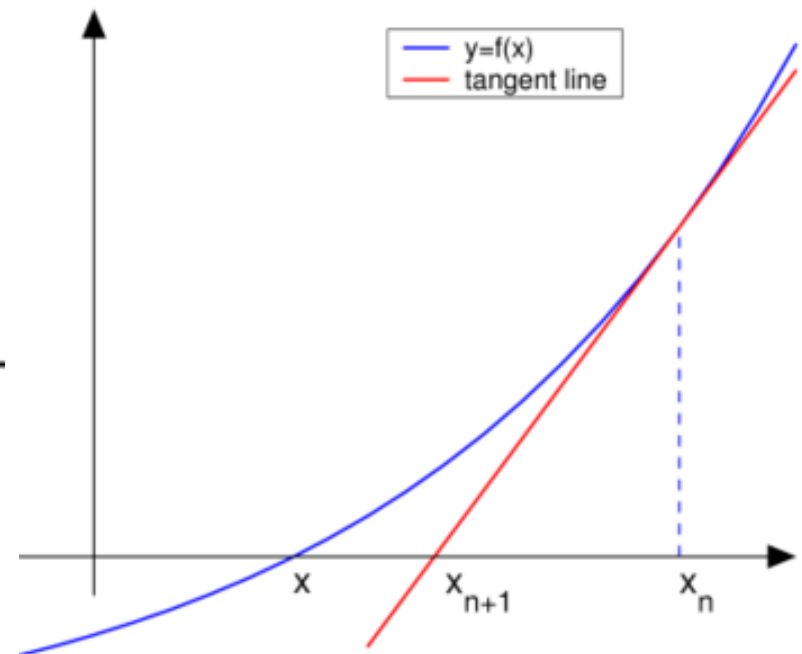


Método de Newton-Rampson

- O método de Newton-Raphson é utilizado para obter as raízes de $f(x)=0$.
- O método para funções reais corresponde a iterar

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

- Para encontrar os extremos (máximos ou mínimos) tem de se resolver a equação $f'(x) = 0$



Optimização com método de Newton-Raphson

- Para o caso de funções reais, iterar

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}, \quad n \geq 0$$

- Para o caso de funções em $\mathbb{R}^n$, generaliza-se a fórmula de aproximação para

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \gamma [Hf(\mathbf{x}_n)]^{-1} \nabla f(\mathbf{x}_n).$$

- A matriz Hessiana H_f contém as derivadas parciais de segunda ordem de f , isto é
 $H_{ij} = \partial^2 f / \partial x_i \partial x_j$

Bibliografia

- Capítulos 4.3 e 4.4 do AIMA (2ª edição) ou 4.1 e 4.2 (3ª edição)
- Capítulos 1 e 2 do livro Stochastic Local Search.
- Wikipedia
 - http://en.wikipedia.org/wiki/Gradient_descent
 - http://en.wikipedia.org/wiki/Newton%27s_method_in_optimization
- *This material is partially based on slides provided with the book 'Stochastic Local Search: Foundations and Applications' by Holger H. Hoos and Thomas Stützle (Morgan Kaufmann, 2004) - see www.sls-book.net for further information.*