Inteligência Artificial

Prof. Rafael Stubs Parpinelli
DCC / UDESC-Joinville
rafael.parpinelli@udesc.br

Busca Heurística ou Busca Informada

- Estratégias de Busca Exaustiva
  - Encontram soluções para problemas pela geração *sistemática* de novos estados, que são comparados ao objetivo
  - São *ineficientes* na maioria dos problemas do mundo real
    - são capazes de calcular *apenas* o *custo de caminho* do nó atual ao nó inicial (função *g(n)*) para decidir qual o próximo nó da fronteira a ser expandido
    - essa medida não necessariamente conduz a busca na direção do objetivo. Olha só para o passado!

Busca Heurística

- Estratégias de Busca Heurística
  - utilizam *conhecimento específico* do problema na escolha do próximo nó a ser expandido
- Aplicação de uma *função de avaliação* a cada nó na fronteira do espaço de estados
  - essa função estima o *custo de caminho* do nó atual ao objetivo mais próximo utilizando uma ***função heurística***.
  - Qual dos nós supostamente é o mais próximo do objetivo?
- Classes de algoritmos para busca heurística:

1. Busca pela melhor escolha (*Best-First Search*)
2. Busca com limite de memória
3. Busca com melhora iterativa

Funções Heurísticas

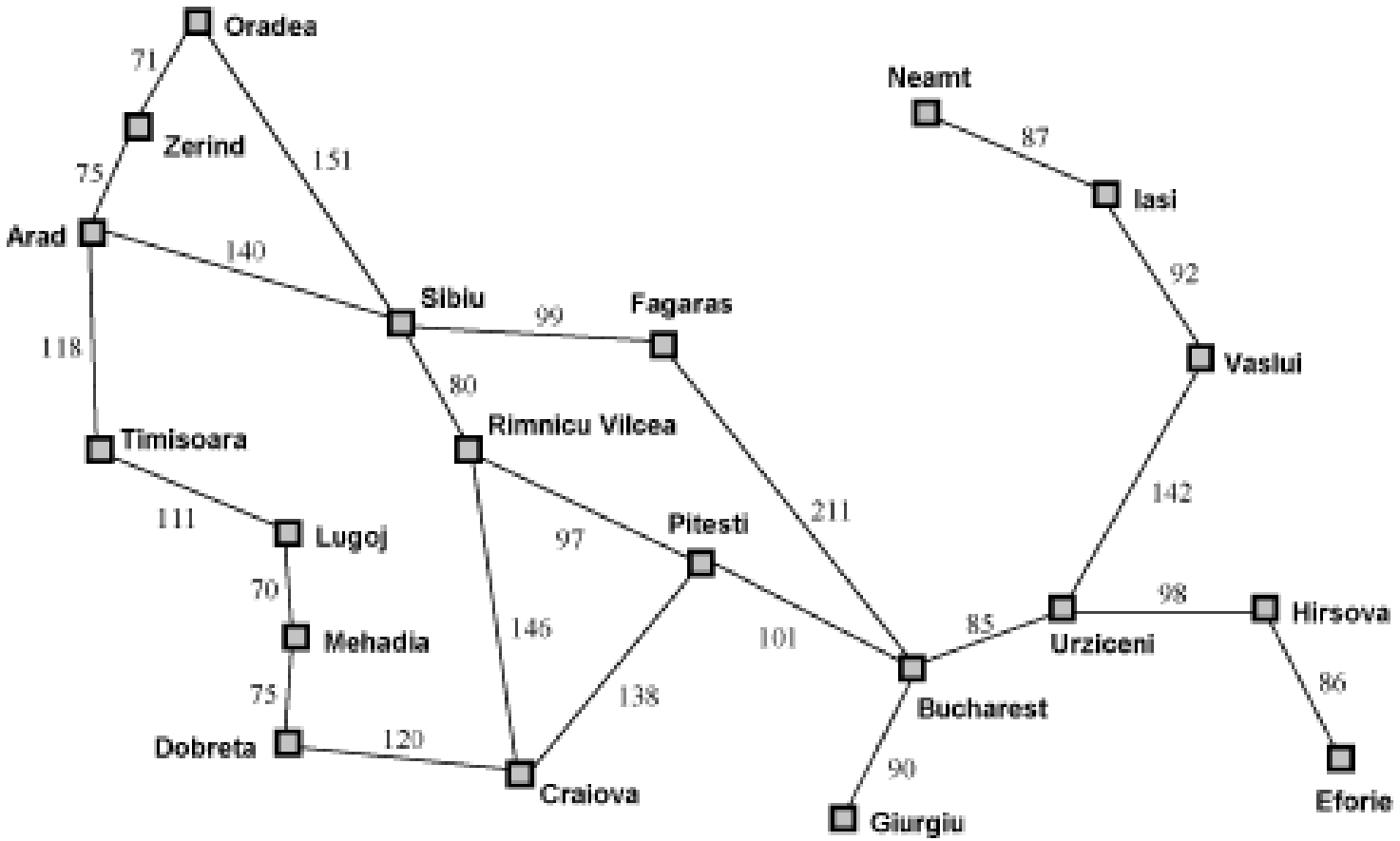
- Função heurística *h(n)*
  - **estima** o custo do caminho entre o nó ***n*** e o objetivo
  - depende o problema
- Exemplo
  - encontrar a rota de caminho mínimo entre duas localidades
  - *hdd(n)* = distância direta (em linha reta) entre o nó ***n*** e o nó final
- Como escolher uma boa função heurística?
  - ela deve ser admissível i.e., nunca *superestimar* o custo real da solução
  - Distância direta (*hdd*) é *admissível* porque o caminho mais curto entre dois pontos é sempre uma linha reta

Busca pela Melhor Escolha (*Best-First Search*)

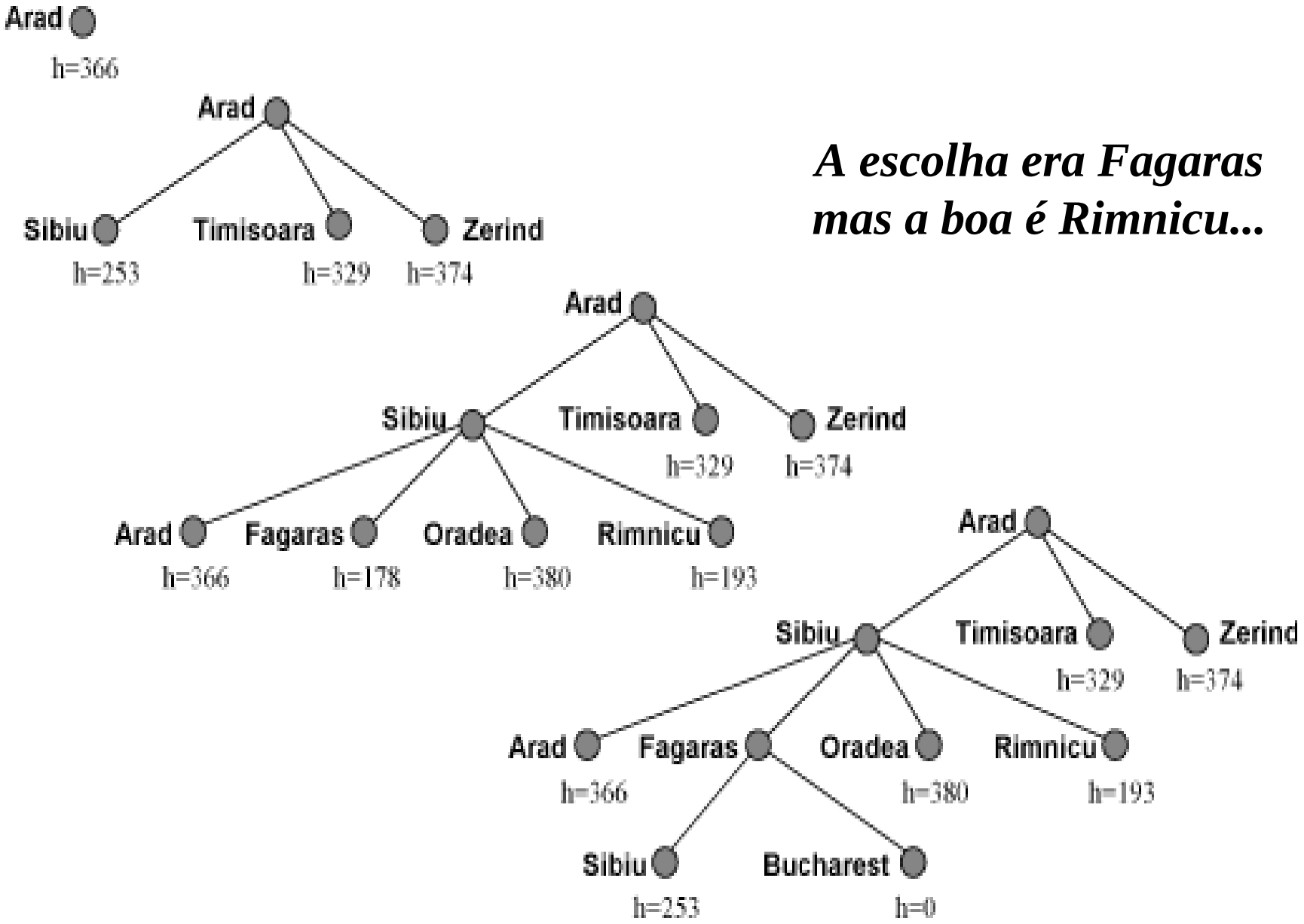
- Busca genérica onde o **nó de menor custo "aparente"** na fronteira do espaço de estados é expandido primeiro
- Duas abordagens básicas:
  - 1. Busca Gulosa (*Greedy Search*)
  - 2. Algoritmo A*
- Busca gulosa
  - Semelhante à **busca em profundidade** com *backtracking*
  - Tenta expandir o nó mais próximo ao nó final com base na estimativa feita pela função heurística ***h***.

Busca Gulosa

- Exemplo:
  - encontrar a rota mais curta de Arad a Bucharest
  - hdd(n) = distância direta entre o nó ***n*** e o nó final
  - hdd é admissível!



Straight line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



A escolha era Fagaras mas a boa é Rimnicu...

Busca Gulosa: conclusões

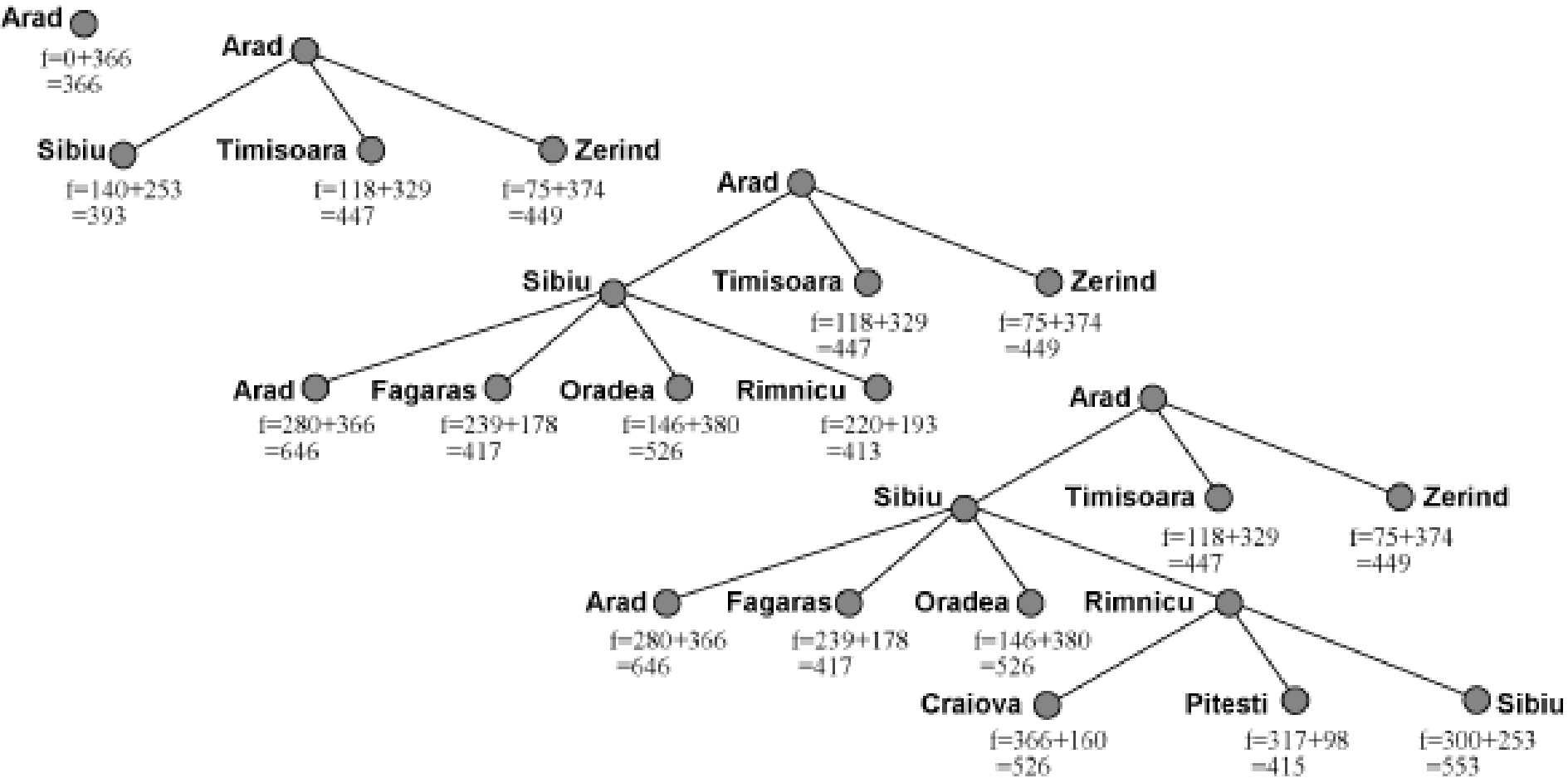
- Não é ótima... **(semelhante à busca em profundidade)**
  - porque só olha para o futuro!
- Não é completa:
  - pode entrar em "*loop*" se não detectar a expansão de estados repetidos
  - pode tentar desenvolver um caminho infinito
- Custo de busca é minimizado
  - não expande nós fora do caminho
- Porém... escolhe o caminho que é mais econômico à primeira vista
- E no entanto, podem existir caminhos mais curtos...

Algoritmo A*

- Tenta minimizar o custo total da solução combinando:
  - Busca Gulosa
    - econômica, porém não é completa nem ótima
  - Busca de Custo Uniforme
    - ineficiente, porém completa e ótima
- Função de avaliação:
  - f(n) = g(n) + h(n)
  - g(n) = distância de ***n*** ao nó inicial
  - h(n) = distância estimada de ***n*** ao nó final
- A* expande o nó de menor valor de ***f*** na fronteira do espaço de estados.
  - Olha o futuro sem esquecer do passado!

Algoritmo A*

- Qual seria a rota desenvolvida pelo A* entre as cidades de Arad e Bucharest?



Algoritmo A*

- O algoritmo é monotônico
  - o custo de cada nó gerado no mesmo caminho nunca diminui
  - mas pode haver problemas: ***f(n') < f (n)***, onde ***n*** é o pai de ***n'***
    - f(n) = g(n) + h(n) = 3 + 4 = 7 e f(n') = g(n') + h(n') = 4 + 2 = 6
- Para se garantir a monotonicidade de ***f***
  - quando usa-se ***f(n') = max(f(n), g(n') + h(n'))***
  - uma vez que todo caminho que passa por ***n'*** passa também por ***n*** (seu pai)
- Semelhante à busca em largura
  - mas ao invés de geração (profundidade) da árvore o que conta é o contorno ***f=g+h***

Algoritmo A*

- A* completa e ótima
  - como a busca em largura...
- A* é otimamente eficiente:
  - não existem algoritmos expandindo menos nós com a mesma *f*

Busca Limitada pela Memória (*Memory Bounded Search*)

- IDA* (*Iterative Deepening A**)
  - igual ao aprofundamento iterativo, porém com limite na função de avaliação (***f***) no lugar da profundidade (***d***).
  - necessita menos memória do que A*
- SMA* (*Simplified Memory-Bounded A**)
  - O número de nós guardados em memória é fixado previamente

Definindo Funções Heurísticas

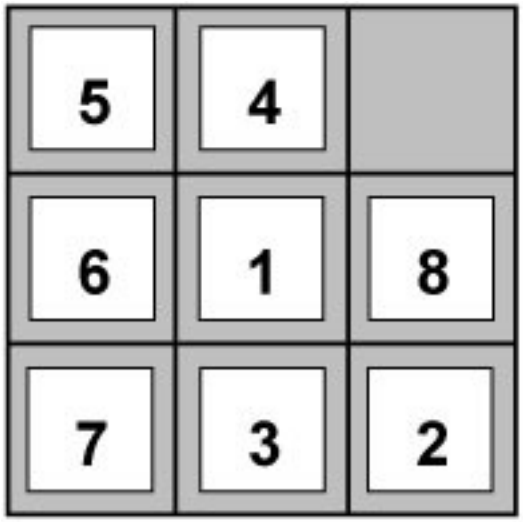
- Como escolher uma boa função heurística ***h***?
  - ***h*** depende de cada problema particular
  - ***h*** deve ser *admissível*
    - Uma heurística é admissível se para cada nodo, o valor retornado por esta heurística nunca ultrapassa o custo real do melhor caminho desse nodo até o objetivo. Não superestima o custo real da solução.
- Existem estratégias genéricas para definir ***h***
  1) Relaxar restrições do problema
  2) Usar informação estatística

(1) Relaxando o problema

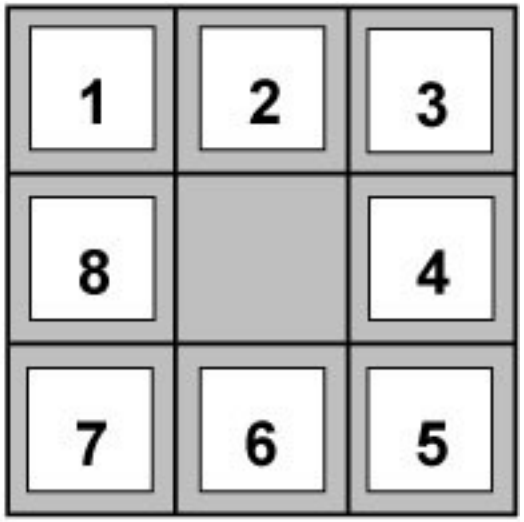
- Problema Relaxado
  - versão simplificada do problema original, onde os operadores são menos restritivos
- Exemplo: jogo dos 8 números - operador original
  - um número pode mover-se de A para B se A é adjacente a B e B está vazio
- Operadores relaxados:
  1. um número pode mover-se de A para B se A é adjacente a B
  2. um número pode mover-se de A para B se B está vazio
  3. um número pode mover-se de A para B

4	5	8
	1	6
7	2	3

Heurísticas para jogo 8 números



Start State



Goal State

- Heurísticas possíveis
  - h1 = no. de elementos fora do lugar (h1=7)
  - h2 = soma das distâncias de cada número à posição final (distância Manhatan) (h2=2+3+3+2+4+2+0+2=18)

(2) Usando informação estatística

- Funções heurísticas podem ser “melhoradas” com informação estatística:
  - executar a busca com um conjunto de treinamento (e.g., 100 configurações diferentes do jogo), e computar os resultados.
  - se, em 90% dos casos, quando $h(n) = 14$, a distância real da solução é 18,
  - então, quando o algoritmo encontrar 14 para o resultado da função, vai substituir esse valor por 18.

Escolhendo Funções Heurísticas

- É sempre melhor usar uma função heurística com valores mais altos, contanto que ela seja *admissível.*
  - Isto traz mais informação na escolha do operador (ex. h_2 melhor que h_1)
- h_i <u>domina</u> $h_k \Rightarrow h_i(n) \geq h_k(n) \ \forall \ n$ no espaço de estados
  - h_2 domina h_1 no exemplo anterior
- Caso existam muitas funções heurísticas para o mesmo problema, e nenhuma delas domine as outras, usa-se uma *heurística composta*:
  - $h(n) = max\ (h_1(n), h_2(n),\ldots,h_m(n))$
- Assim definida, *h* é *admissível* e *domina* cada função h_i individualmente

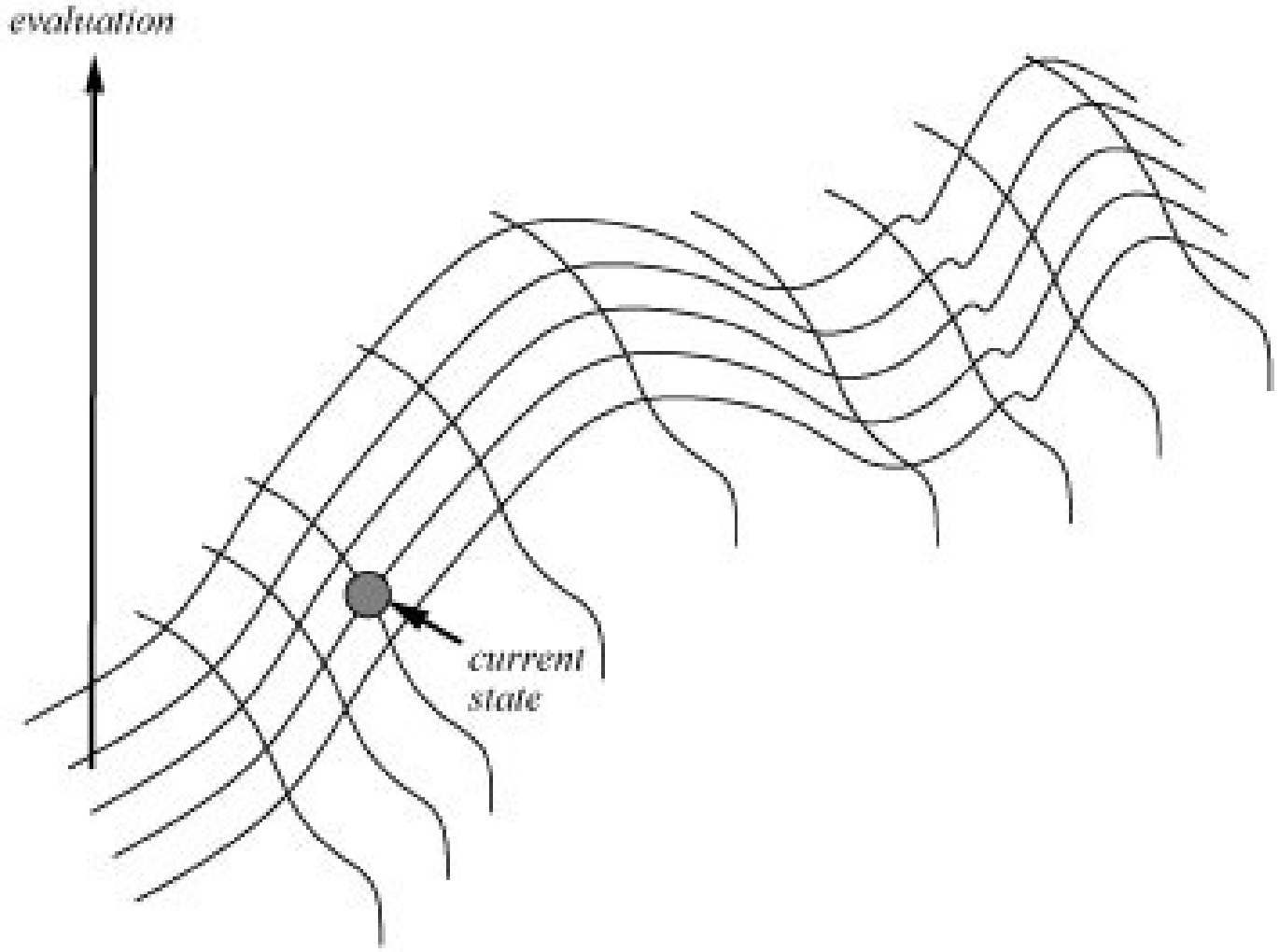
Qualidade da função heurística

- Qualidade da função heurística: **medida através do fator de expansão efetivo (*b*).***

$$N = 1 + b_1^* + (b_2^*)^2 + \ldots + (b_d^*)^d \text{ , onde}$$

  - N = total de nós expandidos para uma instância de problema
  - b_d^* = quantidade de nós expandidos na profundidade *d*
  - d = profundidade da solução
- Mede-se empiricamente a qualidade de *h* a partir do conjunto de valores experimentais de *N* e *d*.
- Se o custo de execução da função heurística for maior do que expandir nós, então ela *não* deve ser usada.
  - uma boa função heurística deve ser *eficiente* e *econômica*.

Algoritmos de Melhorias Iterativas



Algoritmos de Melhorias Iterativas *(Iterative Improvement Algorithms*)

- A idéia é começar com o *estado inicial* (= configuração completa, solução aceitável), e melhorá-lo iterativamente.
- Os estados estão representados sobre uma superfície
  - a altura de qualquer ponto na superfície corresponde à *função de avaliação* do estado naquele ponto
- O algoritmo se “move” pela superfície em busca de pontos mais altos/baixos (objetivos)
  - o ponto mais alto (máximo global) corresponde à solução ótima
    - nó onde a função de avaliação atinge seu valor máximo
- Aplicações: problemas de otimização
  - por exemplo, linha de montagem

Algoritmos de Melhorias Iterativas

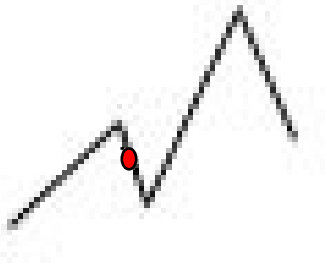
- Esses algoritmos guardam apenas o estado atual, e não vêem além dos vizinhos imediatos do estado.
- Contudo, muitas vezes são os melhores métodos para tratar problemas reais muito complexos.
- Duas classes de algoritmos:
  - ***Hill-Climbing****: Gradiente Ascendente ou Subida da Encosta*
    - só faz modificações que melhoram o estado atual.
  - ***Simulated Annealing****: Anelamento Simulado ou Recozimento Simulado ou Têmpera Simulada*
    - pode fazer modificações que pioram o estado temporariamente para possivelmente melhorá-lo no futuro.

Subida de Encosta

- O algoritmo *não* mantém uma árvore de busca:
  - Guarda apenas o estado atual e sua avaliação
  - É simplesmente um "*loop*" que fica se movendo na direção que está crescendo.
- Quando existe mais de um "melhor" sucessor para o nó atual, o algoritmo faz uma escolha aleatória

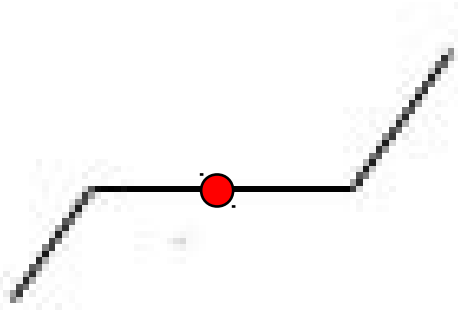
- Isso pode acarretar em 3 problemas, que podem levar ao desvio do caminho à solução:
  - 1. Máximos locais
  - 2. Planícies
  - 3. Encostas

Máximos locais

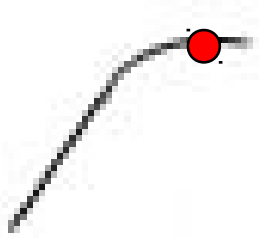


- Máximos Locais:
  - em contraste com *máximos globais*, são picos mais baixos do que o pico mais alto no espaço de estados (solução ótima).
  - a função de avaliação leva a um valor máximo para o caminho sendo percorrido: essa função utiliza informação “local”.
  - porém, o nó final está em outro ponto mais “alto”.
  - isto é uma conseqüência das decisões irrevogáveis do método
    - e.g., xadrez: eliminar a Rainha do adversário pode levar o jogador a perder o jogo.

Platôs e Picos



- Platôs:
  - uma região do espaço de estados onde a função de avaliação dá o mesmo resultado.



- Encostas:
  - encostas podem ter lados muito íngremes e o algoritmo chega ao topo com facilidade,
  - mas, o topo pode se mover ao pico vagarosamente
  - na ausência de operadores que se movam pelo topo, o algoritmo oscila entre dois pontos sem fazer grande progresso.

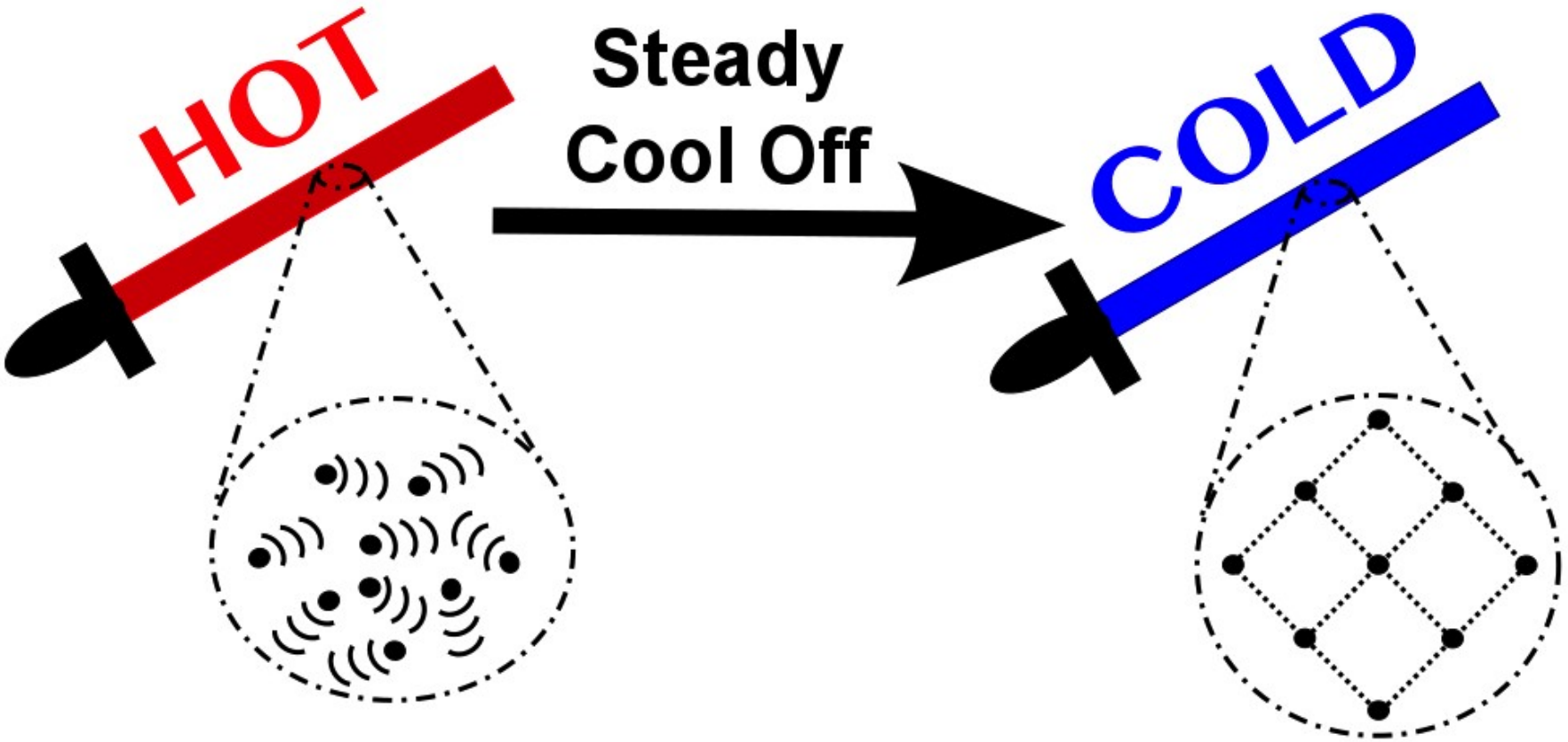
Subida de Encosta

- Nos casos acima, o algoritmo chega a um ponto onde não fez progresso.
- *Solução: random restart hill-clibing*
  - O algoritmo realiza uma série de buscas a partir de estados iniciais gerados aleatoriamente.
- Cada busca é executada
  - até que um número máximo de iterações estipulado seja atingido, ou
  - até que os resultados encontrados não apresentem melhora significativa.
- O algoritmo escolhe o melhor resultado obtido com as diferentes buscas.

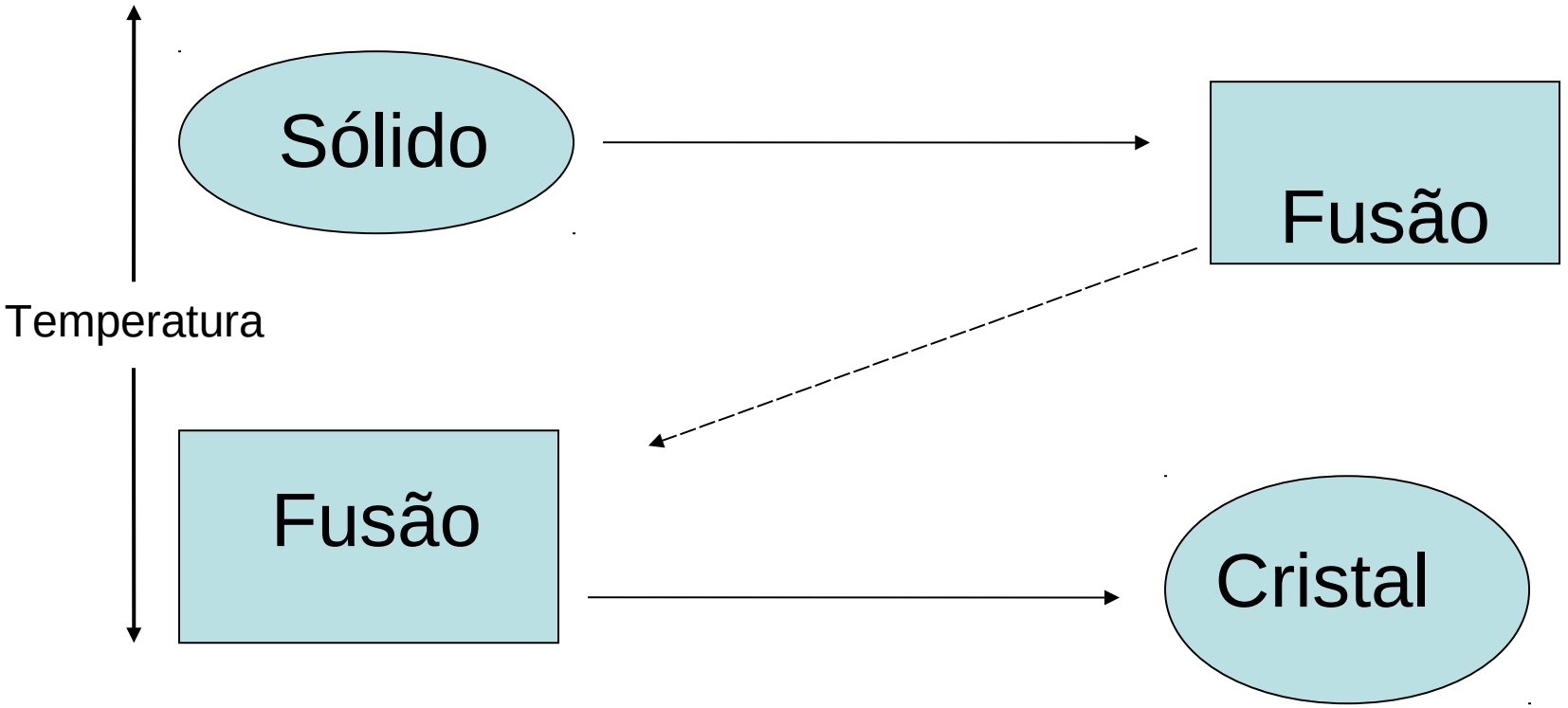
Subida de Encosta

- Pode chegar a uma solução ótima quando iterações suficientes forem permitidas.
- O sucesso deste método depende muito do formato da superfície do espaço de estados:
  - se há poucos máximos locais, o reinício aleatório encontra uma boa solução rapidamente
  - porém, para problemas NP-completos, o custo de tempo é exponencial

Algoritmo *Simulated Annealing*



Algoritmo *Simulated Annealing*



Fundamentação

- *Annealing* Físico:
  - Sólido aquecido além do seu ponto de fusão e é resfriado lentamente
  - Se o resfriamento e suficientemente lento obtêm-se uma estrutura cristalina livre de imperfeições (estado de baixa energia)

Fundamentação

- *Annealing* Simulado:
  - Algoritmo de **Metropolis** (Gibbs, 1953) empregado numa sequência de temperaturas decrescentes para gerar soluções de um problema de otimização
  - O processo começa com um valor **T** elevado e a cada **T** geram-se soluções até que o equilíbrio àquela temperatura seja alcançado
  - A temperatura é então **rebaixada** e o processo prossegue até o **congelamento**
  - A **sequência de temperaturas** empregada, juntamente com o **número de iterações** a cada temperatura, constitui uma prescrição de *annealing* que deve ser definida empiricamente

Fundamentação

- Analogia com um problema de otimização:
  - Os **estados possíveis** de um metal correspondem a **soluções do espaço** de busca
  - A **energia** em cada estado corresponde ao valor da **função objetivo**
  - A **energia mínima** (se o problema for de minimização ou máxima, se de maximização) corresponde ao valor de uma **solução ótima** do problema

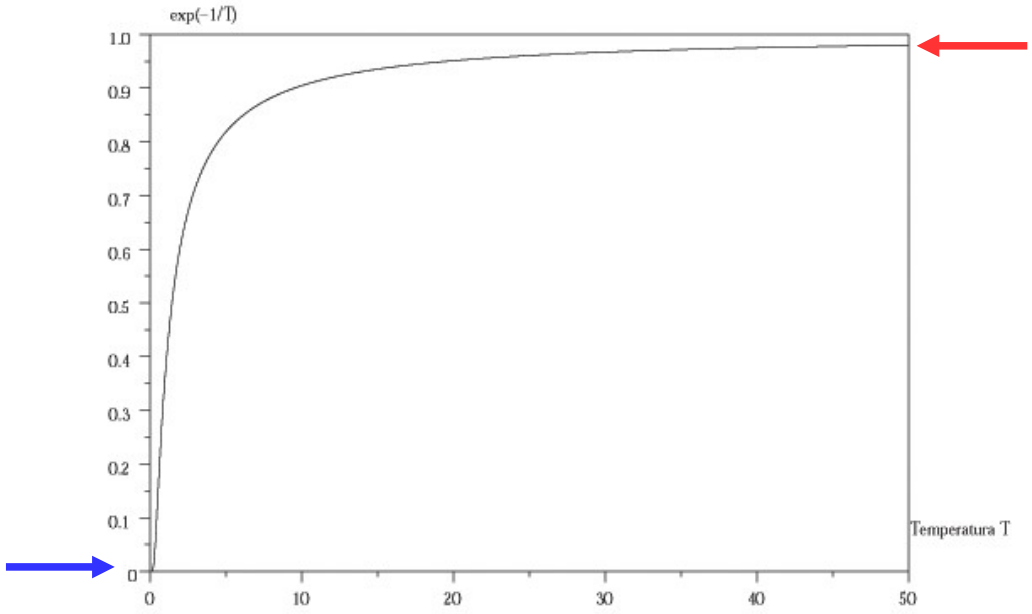
Fundamentação
(problema de minimização)

- A cada iteração do método, um **novo estado** é gerado a partir do estado corrente por uma **modificação aleatória** neste;
- Se o novo estado é de **energia menor** que o estado corrente, esse novo estado passa a ser o **estado corrente**;
- Se o novo estado tem uma **energia maior** que o estado corrente em Δ unidades, a **probabilidade** de se mudar do estado corrente para o novo estado é:
    - $e^{-\Delta/(kT)}$, onde k = constante de Boltzmann que relaciona temperatura e energia de moléculas
    - T = temperatura atual
- Este procedimento é repetido até se atingir o equilíbrio térmico (algoritmo de Metropolis)

Probabilidade de aceitação de um movimento de piora

- Baseada na fórmula: P(aceitação) = $e^{-\Delta/kT}$

Δ = *Valor[próximo-nó]* - *Valor[nó-atual]*; T = temperatura; k = 1



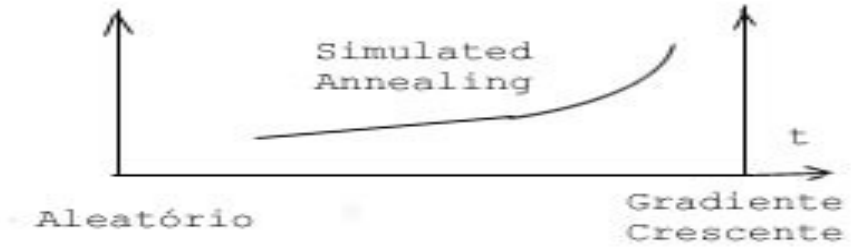
- **Temperatura ↑** ⇒ Probabilidade de aceitação ↑
- **Temperatura ↓** ⇒ Probabilidade de aceitação ↓

Fundamentação

- A **altas temperaturas**, cada estado tem (praticamente) a mesma chance de ser o estado corrente
- A **baixas temperaturas**, somente estados com baixa energia têm alta probabilidade de se tornar o estado corrente
- Atingido o **equilíbrio térmico** em uma dada temperatura, esta é diminuída e aplica-se novamente o passo de Metropolis
- O **método termina** quando a temperatura se aproxima de zero

Fundamentação

- No **início do processo**, a **temperatura é elevada** e a **probabilidade** de se aceitar soluções de piora é **maior**
- As **soluções de piora** são aceitas para **escapar de ótimos locais**
- A probabilidade de se aceitar uma solução de piora depende de um parâmetro, chamado temperatura
- Quanto menor a temperatura, menor a probabilidade de se aceitar soluções de piora
- Atingido o equilíbrio térmico, a temperatura é diminuída
- No **final do processo,** **temperatura próxima a zero**, praticamente não se aceita movimentos de piora e o método se comporta como o método da descida/subida



Pseudo-código SA

```
procedimento SA (f(.), N(.), α, SAmax, T0, s)
  s* ← s        {Melhor solução obtida até então}
  IterT ← 0     {Número de iterações na temperatura T}
  T ← T0        {temperatura corrente}
  enquanto (T > 0.0001)
    enquanto (IterT < SAmax) faça
      IterT ← IterT + 1
      Gerar um vizinho (s') aleatoriamente na vizinhança N (s)
      Δ = f(s') – f(s)
      se (Δ < 0) então
        s ← s'
        se (f(s') < f(s*)) então s* ← s'
      senão
        Tome x ∈ [0,1]
        se (x < e^-Δ/T) então
          s = s'
      fim-se
    fim-enquanto
    T = T x α
    IterT = 0
  fim-enquanto
  retorne s*
fim-procedimento
```

- *f(.)*: função objetivo
- *N(.)*: rotina para gerar vizinhança
- $0 < \alpha < 1$: taxa decaimento temperatura
- *SAmax*: iterações para equilíbrio térmico
- T_0: temperatura inicial
- *s*: solução inicial

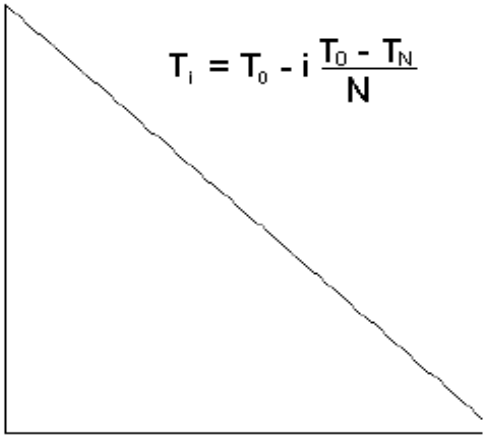
Ajuste da Temperatura Inicial

- Pela **média dos custos** das soluções vizinhas:
  - Gerar uma solução inicial qualquer
  - Gerar um certo número de vizinhos
  - Para cada vizinho, calcular o respectivo custo
  - Retornar como temperatura inicial o maior custo das soluções vizinhas
- Por **simulação**:
  - Gerar uma solução inicial qualquer
  - Partir de uma temperatura inicial baixa
  - Contar quantos vizinhos são aceitos em *SAmax* iterações nessa temperatura
  - Se o número de vizinhos aceitos for **alto** (por exemplo, 95%) retornar a temperatura corrente como a temperatura inicial do SA
  - Caso contrário, **aumentar a temperatura** (por exemplo, em 10%) e repetir o processo

Esquemas de Resfriamento

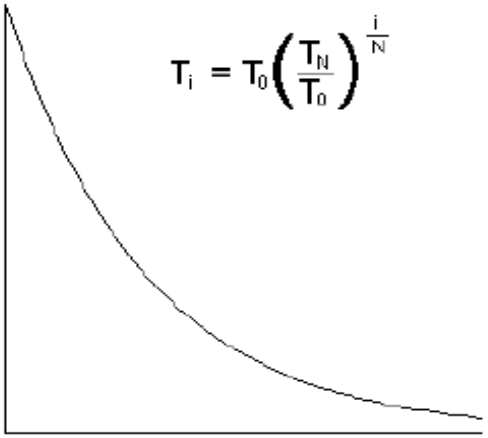
Cooling Schedule 0

$$T_i = T_0 - i\,\frac{T_0 - T_N}{N}$$



Cooling Schedule 1

$$T_i = T_0\left(\frac{T_N}{T_0}\right)^{\frac{i}{N}}$$

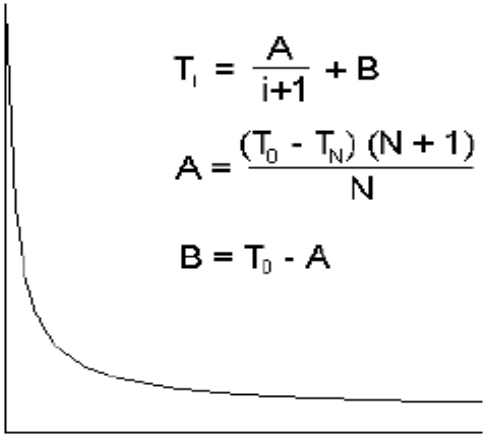


Cooling Schedule 2

$$T_i = \frac{A}{i+1} + B$$

$$A = \frac{(T_0 - T_N)\,(N+1)}{N}$$

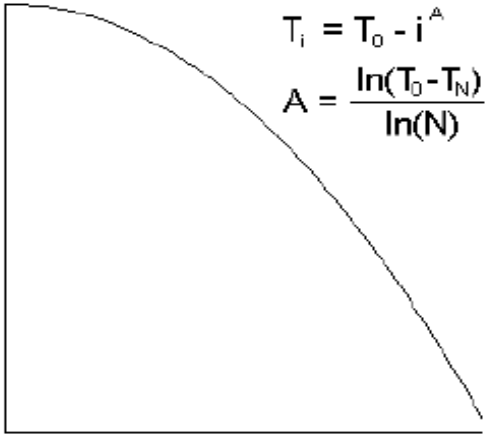
$$B = T_0 - A$$



Cooling Schedule 3

$$T_i = T_0 - i^A$$

$$A = \frac{\ln(T_0 - T_N)}{\ln(N)}$$

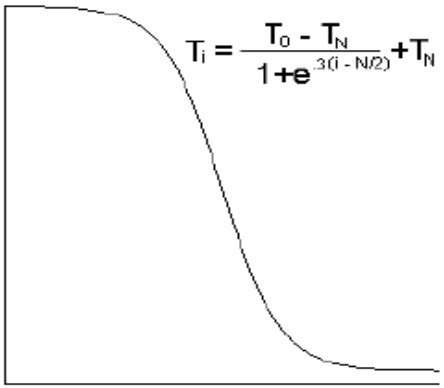


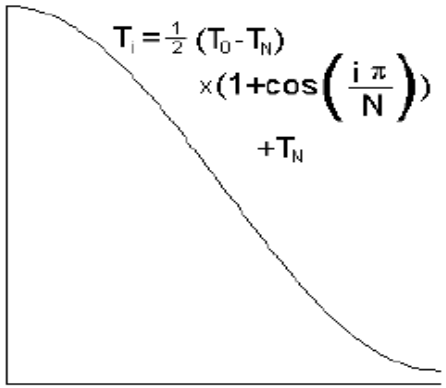
Cooling Schedule 4(sigmoid)

Cooling Schedule 5

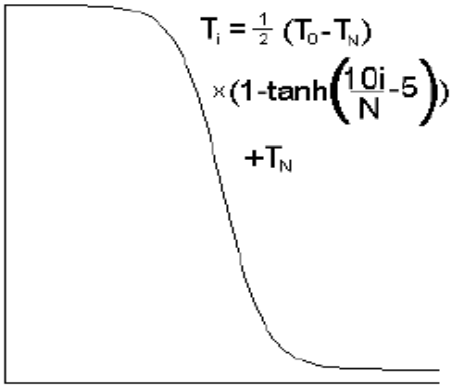
. **Ti** é a temperatura da iteração **i**, aumentando de **0** até **N**.

. As temperaturas inicial e final, $\mathbf{T_0}$ and $\mathbf{T_N}$, são determinadas pelo usuário assim como a variável **N**.

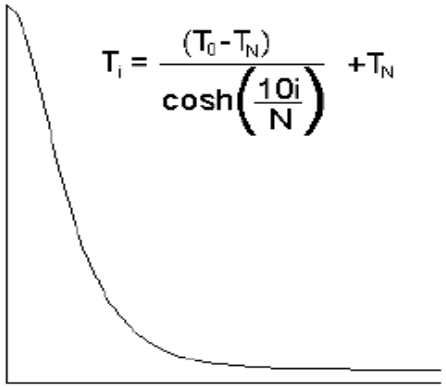




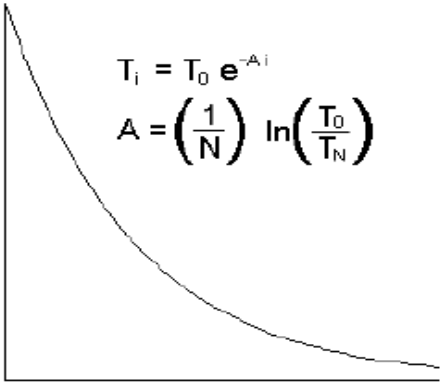
Cooling Schedule 6



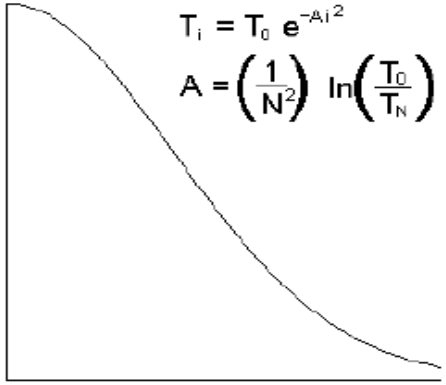
Cooling Schedule 7



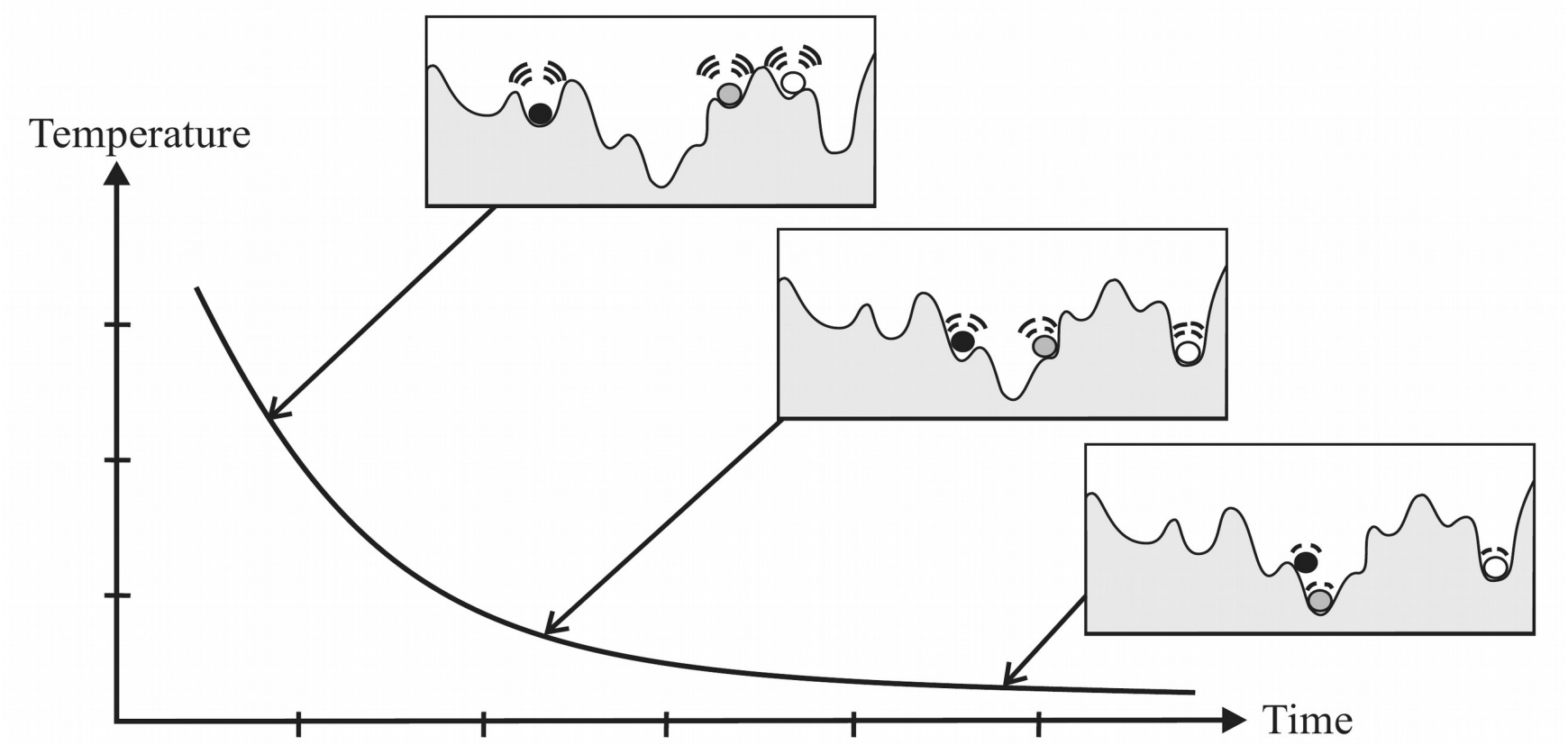
Cooling Schedule 8



Cooling Schedule 9



Ilustrativo - SA



Algoritmo *Simulated Annealing*

- Apesar de poder convergir para a solução ótima, a velocidade de redução de temperatura exigida implica em visitar um número exponencial de soluções.
- A princípio é necessário um processo lento de redução da temperatura e isso resulta em tempos de processamento elevados.
- Pouco “inteligente”, pois utiliza como informação do problema somente a variação do valor da função objetivo.

Algoritmo *Simulated Annealing*

- Muitos parâmetros para “calibrar”.
- Pela simplicidade de implementação, pode ser utilizado em conjunto com alguma outra heurística ou outra meta-heurística.
- Existem implementações, onde apenas a idéia de *simulated annealing* é utilizado para melhorar o desempenho de outra heurística/meta-heurística, como por exemplo, embutir a estratégia de aceitar soluções ruins com certa probabilidade.

Comentários Gerais

- *Solução de problemas* usando técnicas de *busca heurística:*
  - dificuldades em definir e usar a *função de avaliação*
  - não consideram conhecimento genérico do mundo (ou "senso comum")
- Função de avaliação: compromisso entre
  - tempo gasto na seleção de um nó
  - redução do espaço de busca
- Achar o melhor nó a ser expandido a cada passo pode ser tão difícil quanto o problema da busca em geral.