Revisão sobre Busca

Um problema de busca em IA pode ser definido em termos de... Algumas definições básicas (1/2)

- Um espaço de estados possíveis, incluindo:
  - um estado inicial
    - Em (Recife)
    - Estar (pobre)
  - um ou mais estados finais => objetivo
    - Em (João Pessoa)
    - Estar (rico)
  - Espaço de Estados:
    - conjunto de todos os estados alcançáveis a partir do estado inicial por qualquer seqüência de ações.
    - pode ser representado como uma árvore onde os estados são nós e as operações são arcos.
  - Ex., dirigir de Recife a João Pessoa
    - Espaço de estados: todas as cidades da região

Um problema de busca em IA pode ser definido em termos de... Algumas definições básicas(2/2)

- Um conjunto de ações (ou operadores) que permitem passar de um estado a outro
  - Ex., dirigir de Recife a João Pessoa
    - Ações: dirigir de uma cidade a outra na região
    - E.g. Dirigir (Recife, Abreu e Lima)
  - Ficar rico
    - Jogar(megasena).

Um problema de busca em IA pode ser definido em termos de...

- Um estado inicial
- Um conjunto de ações (ou operadores)
  - que permitem passar de um estado a outro
  - os estados podem não “estar lá” concretamente.
    - No caso do problema de dirigir… as cidades estão lá
    - No caso de ficar rico… não necessariamente.
- Um teste de término
  - Verifica se um dado estado é o objetivo
  - Objetivo => um ou mais estados finais
    - propriedade abstrata (em intenção)
      - ex., condição de xeque-mate no Xadrez
    - conjunto de estados finais do problema (em extensão)
      - ex., estar em João Pessoa

Um problema de busca em IA pode ser definido em termos de...

- Custo de caminho
  - Função que associa um custo a cada caminho possível
  - Um caminho é uma sequência de estados conectados por ações possíveis.
  - Cada ação tem um custo associado
    - O custo de dirigir de Recife a Abreu e Lima, por exemplo, poderia ser a distância entre as duas cidades.

Algumas definições

- Solução
  - caminho (seqüência de ações) que leva do estado inicial a um estado final (objetivo).
  - Cuidado! A solução **não** é o estado final!

Custo da Busca

- Custo total da busca =
  - custo de busca (tempo e memória, i.e. custo computacional) -> busca da solução
  - + custo do caminho -> execução da solução
- Espaço de estados grande
  - compromisso (conflito) entre determinar
    - a melhor solução em termos de custo do caminho (é uma boa solução?) e
    - a melhor solução em termos de custo computacional (é computacionalmente barata?)

Formulação, Busca e Execução

Algoritmo de Busca

Busca em Espaço de Estados
Algoritmo de Geração e Teste

- **Fronteira** do espaço de estados
  - Lista contendo os nós (estados) a serem expandidos
  - Inicialmente, a fronteira contém apenas o estado inicial do problema
- **Algoritmo:**

1. Selecionar o primeiro nó (estado) da fronteira do espaço de estados;

   - se a fronteira está vazia, o algoritmo termina com falha.

2. Testar se o nó selecionado é um estado final (objetivo):

   - se "sim", então retornar nó - a busca termina com sucesso.

3. Gerar um novo conjunto de estados aplicando ações ao estado selecionado;

4. Inserir os nós gerados na fronteira, de acordo com a estratégia de busca usada, e voltar para o passo (1).

Busca em Espaço de Estados
Implementação do Algoritmo

- Os nós da fronteira devem guardar mais informação do que apenas o estado:

→ Na verdade nós são uma estrutura de dados com 5 componentes:

1. o estado (configuração) correspondente ao nó atual
2. o seu nó pai – ou o caminho inteiro para não precisar de operações extras
3. a ação aplicada ao pai para gerar o nó – verifica de onde veio para evitar loops
4. o custo do nó desde a raiz (*g(n)*)
5. a profundidade do nó – se guardar o caminho não precisa!

Métodos de Busca

- Busca exaustiva (cega)
  - Não sabe qual o melhor nó da fronteira a ser expandido
    - i.e., menor custo de caminho desse nó até um nó final (objetivo).
  - Estratégias de Busca (ordem de expansão dos nós):
    - caminhamento em largura
    - caminhamento em profundidade

- Busca heurística (informada)
  - Estima qual o melhor nó da fronteira a ser expandido com base em funções heurísticas => conhecimento
  - Estratégia de busca: *best-first search* (melhor

Critérios de Avaliação das Estratégias de Busca

- Completude:
  - a estratégia sempre encontra uma solução quando existe alguma?
- Qualidade ("otimalidade" - *optimality*):
  - a estratégia encontra a melhor solução quando existem diferentes soluções?
  - i.e., solução de menor custo de caminho
- Custo do tempo:
  - quanto tempo gasta para encontrar a 1ª solução?
- Custo de memória:
  - quanta memória é necessária para realizar a busca?

Busca Cega (Exaustiva)

- Estratégias para determinar a ordem de expansão dos nós

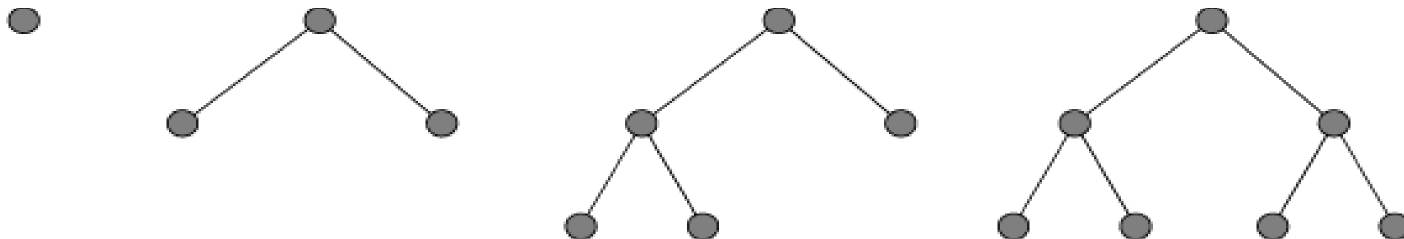
1. Busca em largura
2. Busca de custo uniforme
3. Busca em profundidade
4. Busca com aprofundamento iterativo

Busca em Largura

- Ordem de expansão dos nós:
  1. Nó raiz
  2. Todos os nós de profundidade 1
  3. Todos os nós de profundidade 2, etc…
- Algoritmo:

função **Busca-em-Largura** (***problema***)
    retorna **uma solução ou falha**

Busca-Genérica (*problema*, Insere-no-Fim)



Busca em Largura
Qualidade

- Esta estratégia é *completa*
- É *ótima* ?
  - Sempre encontra a solução mais “rasa”

    → que nem sempre é a solução de menor **custo de caminho,** caso os operadores tenham valores diferentes.
- É *ótima* se
  - $\forall n,n'$ profundidade(n’) $\geq$ profundidade(n) $\Rightarrow$

    custo de caminho(n’) $\geq$ custo de caminho (n).
    - A função **custo de caminho** é não-decrescente com a profundidade do nó.
    - Essa função acumula o custo do caminho da origem ao nó atual.
  - Geralmente, isto só ocorre quando todos os operadores têm o mesmo custo (=1)

Busca em Largura
Custo

- Fator de expansão da árvore de busca:
  - número de nós gerados a partir de cada nó (*b*)
- Custo de tempo:
  - se o fator de expansão do problema = *b*, e a primeira solução para o problema está no nível *d*,
  - então o número máximo de nós gerados até se encontrar a solução = $1 + b + b^2 + b^3 + \ldots + b^d$
  - **custo exponencial** = $O(b^d)$.
- Custo de memória:
  - a *fronteira* do espaço de estados deve permanecer na memória
  - é um problema mais crucial do que o tempo de execução da busca

Busca de Custo Uniforme

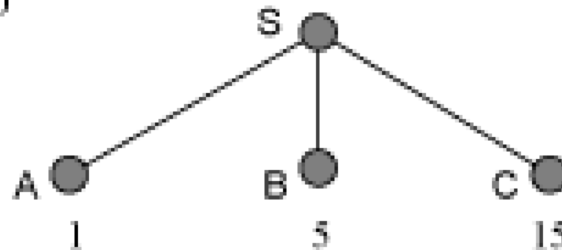
- Modifica a busca em largura:
  - expande o nó da fronteira com menor custo de caminho na fronteira do espaço de estados
  - cada operador pode ter um custo associado diferente, medido pela função *g(n)*, para o nó *n*.
    - onde g(n) dá o custo do caminho da origem ao nó n
- Na busca em largura: ***g(n)* = *profundidade* (*n*)**
- Algoritmo:

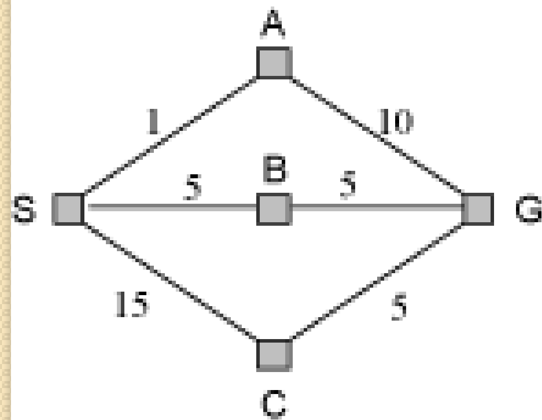
função **Busca-de-Custo-Uniforme** (***problema***)

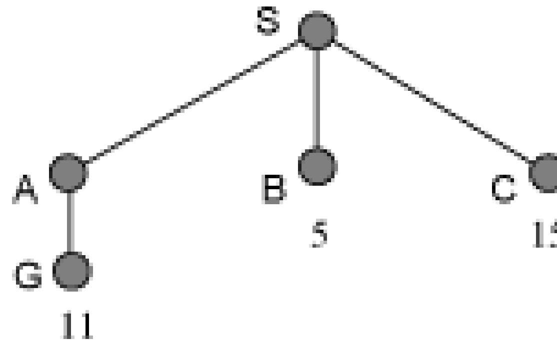
retorna **uma solução ou falha**

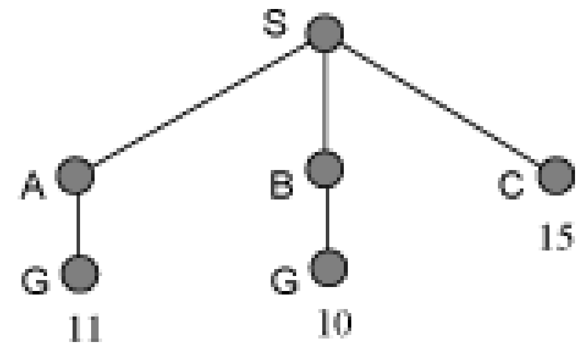
Busca-Genérica (*problema*, Insere-Ordem-Crescente)

Busca de Custo Uniforme









Busca de Custo Uniforme

Fronteira do exemplo anterior

- F = {S}
  - testa se S é o estado objetivo, expande-o e guarda seus filhos A, B e C ordenadamente na fronteira
- F = {A, B, C}
  - testa A, expande-o e guarda seu filho G_A ordenadamente
  - **obs.:** o algoritmo de geração e teste guarda na fronteira todos os nós gerados, testando se um nó é o objetivo apenas quando ele é retirado da lista!
- F= {B, G_A, C}
  - testa B, expande-o e guarda seu filho G_B ordenadamente
- F= {G_B, G_A, C}
  - testa G_B e para!

Busca de Custo Uniforme

- Esta estratégia é *completa*
- É *ótima* se
  - g (sucessor(n)) $\geq$ g (n)
    - custo de caminho **no mesmo caminho** não decresce
    - i.e., não tem operadores com **custo negativo**
  - caso contrário, teríamos que expandir todo o espaço de estados em busca da melhor solução.
    - Ex. Seria necessário expandir também o nó C do exemplo, pois o próximo operador poderia ter custo associado = -13, por exemplo, gerando um caminho mais barato do que através de B
- Custo de tempo e de memória
  - teoricamente, igual ao da Busca em Largura

Busca em Profundidade

- Ordem de expansão dos nós:
  - sempre expande o nó no *nível mais profundo* da árvore:

    1. nó raiz
    2. primeiro nó de profundidade 1
    3. primeiro nó de profundidade 2, etc.…
  - Quando um nó final não é solução, o algoritmo volta para expandir os nós que ainda estão na fronteira do espaço de estados
- Algoritmo:

função Busca-em-Profundidade (*problema*)
  retorna **uma solução ou falha**
 Busca-Genérica (*problema*, Insere-no-Começo)

Busca em Profundidade

- Esta estratégia não é *completa* nem é *ótima*.
- Custo de memória:
  - mantém na memória o caminho sendo expandido no momento, e os nós irmãos dos nós no caminho (para possibilitar o *backtracking*)
    - necessita armazenar apenas *b.m* nós para um espaço de estados com fator de expansão *b* e profundidade *m*, onde *m* pode ser maior que *d* (profundidade da 1a. solução)
- Custo de tempo: $O(b^m)$, no pior caso.
- Observações:
  - Para problemas com várias soluções, esta estratégia pode ser bem mais rápida do que busca em largura.
  - Esta estratégia deve ser evitada quando as árvores geradas são muito *profundas* ou geram *caminhos infinitos*.

Busca com Aprofundamento Iterativo

- Evita o problema de caminhos muito longos ou infinitos impondo um limite máximo *(l)* de profundidade para os caminhos gerados.
  - $l \geq d$, onde *l* é o limite de profundidade e *d* é a profundidade da primeira solução do problema

Busca com Aprofundamento Iterativo

- Esta estratégia tenta limites com valores crescentes, partindo de zero, até encontrar a primeira solução
  - fixa profundidade = i, executa busca
  - se não chegou a um objetivo, recomeça busca com profundidade = $i + n$ (n qualquer)
  - piora o tempo de busca, porém melhora o custo de memória!
- Igual à Busca em Largura para $i=1$ e $n=1$

Busca com Aprofundamento Iterativo

- Combina as vantagens de *busca em largura* com *busca em profundidade.*
- É *ótima* e *completa*
  - com n = 1 e operadores com custos iguais
- Custo de memória:
  - necessita armazenar apenas *b.d* nós para um espaço de estados com fator de expansão *b* e limite de profundidade *d*
- Custo de tempo:
  - $O(b^d)$
- Bons resultados quando o espaço de estados é *grande* e de *profundidade desconhecida*.

Busca Heurística

- 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

Busca pela Melhor Escolha
Best-First Search

- Busca pela Melhor Escolha - BME
  - 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 pela Melhor Escolha

Algoritmo geral

- Função-Insere
  - insere novos nós na fronteira ordenados com base na Função-Avaliação
    - Que está baseada na função heurística

função Busca-Melhor-Escolha **(***problema*, *Função-Avaliação***)**

retorna **uma solução**

Busca-Genérica (*problema, Função-Insere*)

BME: Busca Gulosa

- Semelhante à busca em profundidade com *backtracking*
- Tenta expandir o nó mais próximo do nó final com base na estimativa feita pela função heurística *h*
- *Função-Avaliação*
  - função heurística *h*

Busca Gulosa

- Custo de busca mínimo!
  - não expande nós fora do caminho
- Porém *não* é ótima:
  - escolhe o caminho que é mais econômico à primeira vista
    - Belém do S. Francisco, Petrolândia = 4,4 unidades
  - porém, existe um caminho mais curto de Canudos a Petrolândia
    - Jeremoabo, P. Afonso, Petrolândia = 4 unidades
- A solução via Belém do S. Francisco foi escolhida por este algoritmo porque
  - $h_{dd}(BSF) = 1{,}5$ *u.*, enquanto $h_{dd}(Jer) = 2{,}1$ u.

Busca Gulosa

- Não é completa:
  - pode entrar em *looping* se não detectar a expansão de estados repetidos
  - pode tentar desenvolver um caminho infinito
- Custo de tempo e memória: $O(b^d)$
  - guarda todos os nós expandidos na memória

BME: Algoritmo A*

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

Algoritmo A*

- Se *h* é *admissível*, então *f (n)* é admissível também
  - i.e., *f* nunca irá superestimar o custo real da melhor solução através de *n*
  - pois *g* guarda o valor exato do caminho já percorrido.

- Com A*, a rota escolhida entre *Canudos* e *Petrolândia* é de fato a mais curta, uma vez que:
  - $f\ (BSF) = 2{,}5\ u + 1{,}5\ u = 4\ u$
  - $f\ (Jeremoabo) = 1{,}5\ u + 2{,}1\ u = 3{,}6\ u$

Algoritmo A*:
Análise do comportamento

- A estratégia é completa e ótima
- Custo de tempo:
  - exponencial com o comprimento da solução, porém boas funções heurísticas diminuem significativamente esse custo
    - o fator de expansão fica próximo de 1
- Custo memória: $O(b^d)$
  - guarda todos os nós expandidos na memória, para possibilitar o *backtracking*

Algoritmo A*

Análise do comportamento

- A estratégia apresenta eficiência ótima
  - nenhum outro algoritmo ótimo garante expandir menos nós
- A* só expande nós com $f(n) \leq C^*$, onde C^* é o custo do caminho ótimo
- Para se garantir otimalidade do A*, o valor de f em um caminho particular deve ser não decrescente!!!
  - $f(sucessor(n)) \geq f(n)$
  - i.e., o custo de cada nó gerado no mesmo caminho nunca é menor do que o custo de seus antecessores

Algoritmo A*

Análise do comportamento

- $f = g + h$ deve ser não decrescente
  - g é não decrescente (para operadores não negativos)
    - custo real do caminho já percorrido
  - h deve ser não-crescente (consistente, monotônica)
    - h (n) $\geq$ h (sucessor(n))
    - i.e., quanto mais próximo do nó final, menor o valor de *h*
    - isso vale para a maioria das funções heurísticas
- Quando *h* não é consistente, para se garantir otimalidade do A*, temos:
  - quando f(suc(n)) < f (n)
  - usa-se f(suc(n)) = max (f(n), g(suc(n)) + h(suc(n)))

Busca com Limite de Memória
Memory Bounded Search

- IDA* (Iterative Deepening A*)
  - igual ao aprofundamento iterativo, porém seu limite é dado pela função de avaliação (f) , e não pela profundidade (d).
  - necessita de menos memória do que A*
- SMA* (Simplified Memory-Bounded A*)
  - O número de nós guardados em memória é fixado previamente

Algoritmos de Melhorias Iterativas

- Dois exemplos clássicos
  - Subida da encosta
  - Têmpera simulada

Subida da Encosta - *Hill-Climbing*

- O algoritmo não mantém uma árvore de busca:
  - guarda apenas o estado atual e sua avaliação
- É simplesmente um “loop” que se move
  - na direção crescente da função de avaliação
    - para maximizar
  - ou na direção decrescente da função de avaliação
    - para minimizar
    - Pode ser o caso se a função de avaliação representar o custo, por exemplo...

Subida da Encosta: algoritmo

- função **<u>Hill-Climbing</u> (*problema*)** retorna **uma solução**

  variáveis locais: *atual* (o nó atual), *próximo* (o próximo nó)

atual ← Estado-Inicial do *Problema*

loop do

  próximo ← **sucessor** do nó *atual* **de maior/menor valor**

        (i.e., expande nó *atual* e seleciona seu melhor filho)

  se Valor[*próximo*] < Valor[*atual*] (ou >, para minimizar)

      então retorna nó *atual* (o algoritmo pára)

  atual ← *próximo*

end

Subida da Encosta
Máximos locais

- Os máximos locais são picos mais baixos do que o pico mais alto no espaço de estados
  - máximo global - solução ótima
- Nestes casos, a função de avaliação leva a um valor máximo para o caminho sendo percorrido
  - a função de avaliação é menor para todos os filhos do estado atual, apesar de o objetivo estar em um ponto mais alto
    - essa função utiliza informação "local"
  - e.g., xadrez:
    - eliminar a Rainha do adversário pode levar o jogador a perder o jogo.

Subida da Encosta
Máximos locais

- O algoritmo pára no máximo local
  - só pode mover-se com taxa crescente de variação de f
    - restrição do algoritmo
  - Exemplo de taxa de variação negativa
    - Jogo dos 8 números:
      - mover uma peça para fora da sua posição correta para dar passagem a outra peça que está fora do lugar tem taxa de variação negativa!!!

Subida da Encosta
Platôs (Planícies)

- Uma região do espaço de estados onde a função de avaliação dá o mesmo resultado
  - todos os movimentos são iguais (taxa de variação zero)
    - f(n) = f(filhos(n))
- O algoritmo pára depois de algumas tentativas
  - Restrição do algoritmo
- Exemplo: jogo 8-números
  - em algumas situações, nenhum movimento possível vai influenciar no valor de *f*, pois nenhum número vai chegar ao seu objetivo.

Subida da Encosta

Encostas e Picos

- Apesar de o algoritmo estar em uma direção que leva ao pico (máximo global), não existem operadores válidos que conduzam o algoritmo nessa direção
  - Os movimentos possíveis têm taxa de variação zero ou negativa
    - restrição do problema e do algoritmo
- Exemplo: cálculo de rotas
  - quando é necessário permutar dois pontos e o caminho resultante não está conectado.

Subida da Encosta
Problemas - solução

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

Subida da Encosta: análise

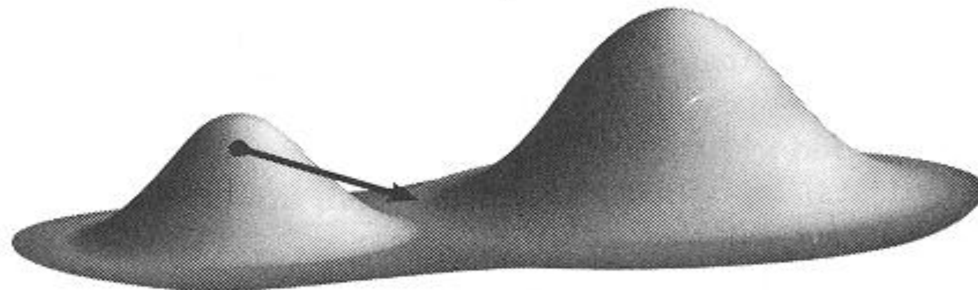
- O algoritmo é completo?
  - **SIM,** para problemas de *otimização*
    - uma vez que cada nó tratado pelo algoritmo é sempre um estado completo (uma solução)
  - **NÃO,** para problemas onde os nós não são estados completos
    - e.g., jogo dos 8-números
    - semelhante à busca em profundidade
- O algoritmo é ótimo?
  - **TALVEZ,** para problemas de *otimização*
    - quando iterações suficientes forem permitidas...
  - **NÃO,** para problemas onde os nós não são estados completos

Subida da Encosta: análise

- 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
  - caso contrário, o custo de tempo é exponencial.

Têmpera Simulada -*Simulated Annealing*

- Este algoritmo é semelhante à Subida da Encosta, porém oferece meios para escapar de máximos locais
  - quando a busca fica "presa" em um máximo local, o algoritmo não reinicia a busca aleatoriamente
  - ele retrocede para escapar desse máximo local
  - esses retrocessos são chamados de passos indiretos
- Apesar de aumentar o tempo de busca, essa estratégia consegue escapar dos máximos locais



Têmpera Simulada

- Analogia com cozimento de vidros ou metais:
  - processo de resfriar um líquido gradualmente até ele se solidificar
- O algoritmo utiliza um mapeamento de resfriamento de instantes de tempo (t) em temperaturas (T).

Têmpera Simulada

- Nas iterações iniciais, não escolhe necessariamente o “melhor” passo, e sim um movimento aleatório:
  - se a situação melhorar, esse movimento será sempre escolhido posteriormente;
  - caso contrário, associa a esse movimento uma probabilidade de escolha menor do que 1.
- Essa probabilidade depende de dois parâmetros, e decresce exponencialmente com a piora causada pelo movimento,
  - $e^{\Delta E/T}$, onde:

    ΔE = Valor[próximo-nó] - Valor[nó-atual]

    T = Temperatura

Têmpera Simulada: algoritmo

função **Anelamento-Simulado** (***problema, mapeamento***)
retorna **uma solução**
variáveis locais: *atual*, *próximo, T* (temperatura que controla a probabilidade de passos para trás)

atual ← Faz-Nó(Estado-Inicial[*problema*])

for *t* ← 1 **to** ∞ **do**
T ← mapeamento[*t*]
Se *T* = 0
então retorna *atual*
próximo ← um sucessor de *atual* escolhido aleatoriamente
ΔE ← Valor[*próximo*] - Valor[*atual*]
Se *ΔE* > 0
então *atual* ← *próximo*
senão *atual* ← *próximo com probabilidade* = $e^{-\Delta E/T}$

Têmpera Simulada

- Para valores de T próximos de zero
  - a expressão $\Delta E/T$ cresce
  - a expressão $e^{-\Delta E/T}$ tende a zero
  - a probabilidade de aceitar um valor de próximo menor que corrente tende a zero
  - o algoritmo tende a aceitar apenas valores de próximo maiores que corrente
- Conclusão
  - com o passar do tempo (diminuição da temperatura), este algoritmo passa a funcionar como Subida da Encosta

Têmpera Simulada

- Implementação (dica)
  - Gerar número aleatório entre (0,1) e comparar com o valor da probabilidade
  - Se número sorteado < probabilidade, aceitar movimento para trás
- Análise
  - O algoritmo é completo
  - O algoritmo é ótimo se o mapeamento de resfriamento tiver muitas entradas com variações suaves
    - isto é, se o mapeamento diminui *T* suficientemente devagar no tempo, o algoritmo vai encontrar um máximo global ótimo.

Exercícios do Livro Sugeridos

- Da 2ª Edição (Livro Verde)
  - Capítulo 3
    - 3.1, 3.2, 3.7, 3.8, 3.9
  - Capítulo 4
    - 4.1
- Da 3ª Edição
  - Capítulo 3
    - 3.1, 3.2, 3.6, 3.9, 3.10, 3.11, 3.14, 3.15, 3.16 (a e b)