1. INTRODUÇÃO

1. INTRODUÇÃO

1.1. Métodos de Otimização

A otimização matemática é uma área da ciência computacional que busca responder à pergunta "O que é melhor?" para problemas em que a qualidade de uma resposta pode ser medida por um número. Estes problemas aparecem em praticamente todas as áreas do conhecimento: negócios, ciências físicas, químicas e biológicas, engenharia, arquitetura, economia e administração. A quantidade de ferramentas disponíveis para auxiliar nesta tarefa é quase tão grande quanto o número de aplicações.

Para resolver um problema, é preciso primeiro formulá-lo. A *Função Objetivo* é uma função que associa cada ponto no espaço de soluções a um número real. Este número permite medir a qualidade de uma resposta: no problema de minimização, quanto menor este valor, melhor a resposta. No problema de maximização, o inverso ocorre. O tratamento matemático de problemas de maximização e minimização é análogo, já que há várias maneiras de converter um problema no outro. Apenas por questão de uniformidade, toda a discussão ao longo deste curso se baseará no problema de minimização.

Um método de otimização é chamado de *Determinístico* se for possível prever todos os seus passos conhecendo seu ponto de partida. Em outras palavras, um método determinístico sempre leva à mesma resposta se partir do mesmo ponto inicial. Em oposição a estes métodos, existem os chamados métodos *Estocásticos* ou *Aleatórios*, onde o caráter aleatório de vários processos é simulado. Nestes métodos, várias escolhas são feitas com base em números aleatórios, sorteados no momento de execução do código. Como a cada execução do código os números sorteados serão diferentes, um método aleatório não executará a mesma seqüência de operações em duas execuções sucessivas. Partindo de um mesmo ponto inicial, cada execução do código seguirá o seu próprio caminho, e possivelmente levará a uma resposta final diferente.

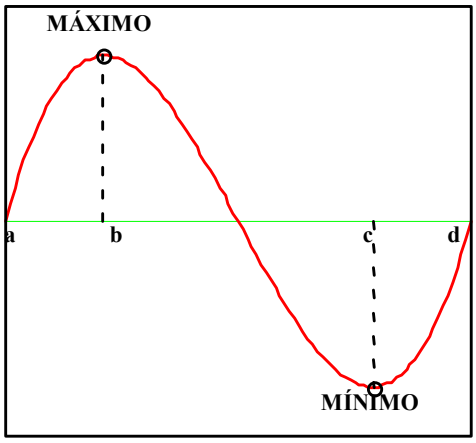
voltar para INTRODUÇÃO

1.2. Métodos Tipo Gradiente

Métodos de otimização bastante familiares para os engenheiros químicos são os que empregam a derivada de uma função para encontrar seu ótimo. Estes métodos são determinísticos, ou seja, sempre chegarão à mesma resposta se saírem do mesmo ponto inicial.

Métodos de Subida (ou Descida) Mais Íngreme

A forma mais simples do método tipo gradiente é o método chamado de *subida mais íngreme* (em inglês: *steepest ascent*), quando se deseja determinar o máximo de uma função escalar, ou o de *descida mais íngreme* (em inglês: *steepest descent*), no caso de busca do mínimo. Para entender a concepção básica deste método a figura abaixo é apresentada:



Na faixa entre a e b e entre c e d a derivada (ou o gradiente) da função é positiva e na faixa entre b e c a derivada (ou o gradiente) da função é positiva. Observe que x = b é um ponto de máximo e que x = c é um ponto de mínimo.

Um procedimento recursivo que levaria ao valor de x em que a função assume seu valor máximo (ou seja, x=b) seria aquele que partindo de condições iniciais entre a e b *caminharia* para a direita (na direção de b) e que partindo de condições iniciais entre b e c *caminharia* para a esquerda (na direção de b), portanto o procedimento recursivo *caminha* sempre na mesma direção do gradiente da função. O correspondente método iterativo pode ser representado por:

$X^{(k+1)} = X^{(k)} + \lambda \cdot f'\left[X^{(k)}\right]$, onde λ é um escalar positivo.

Por outro lado, um procedimento recursivo que levaria ao valor de x em que a função assume seu valor mínimo (ou seja, x=c) seria aquele que partindo de condições iniciais entre b e c *caminharia* para a direita (na direção de c) e que partindo de condições iniciais entre c e d *caminharia* para a esquerda (na direção de c), portanto o procedimento recursivo *caminha*

sempre na direção contrária à do gradiente da função. O correspondente método iterativo pode ser representado por:

$X^{(k+1)} = X^{(k)} - \lambda \cdot f'\left[X^{(k)}\right]$, onde λ é um escalar positivo.

Para ilustrar este procedimento a função: $f(x) = x \cdot (x^2 - 100)$ é considerada. A derivada desta função é : $f'(x) = g(x) = 3 \cdot x^2 - 100$ que se anula em : $x = \pm 10/\sqrt{3} \cong \pm 5.774$. O ponto $x = -5.774$, em vista de $f''(-5.774) = -6 \cdot 5.774 < 0$, é um ponto de **máximo** e o ponto $x = +5.774$, em vista de $f''(+5.774) = +6 \cdot 5.774 > 0$, é um ponto de **mínimo**.

O método do gradiente de *subida mais íngreme* é neste exemplo descrito pelo procedimento recursivo: $X^{(k+1)} = X^{(k)} + \lambda \cdot \left\{3 \cdot \left[X^{(k)}\right]^2 - 100\right\}$ com $\lambda > 0$ e para k= 0, 1, 2,... com $X^{(0)} \rightarrow$ *chute inicial*. Adotando este algoritmo com $X^{(0)} = -10$ e $\lambda = 0.1$ resulta:

k	0	1	2	3	4
$X^{(k)}$	-10	+10	+30	+290	+25510

não convergindo portanto ao valor desejado [$x = -5.774$]. Adotando o mesmo *chute inicial*: $X^{(0)} = -10$ e $\lambda = 0.04$ resulta:

k	0	1	2	3	4	5	6
$X^{(k)}$	-10,000	-4,000	-5,560	-5,560	-5,778	-5,773	-5,774

convergindo ao valor desejado após 6 iterações.

O método do gradiente de *descida mais íngreme* é neste exemplo descrito pelo procedimento recursivo: $X^{(k+1)} = X^{(k)} - \lambda \cdot \left\{3 \cdot \left[X^{(k)}\right]^2 - 100\right\}$ com $\lambda > 0$ e para k= 0, 1, 2,... com $X^{(0)} \rightarrow$ *chute inicial*. Adotando este algoritmo com $X^{(0)} = +10$ e $\lambda = 0.1$ resulta:

k	0	1	2	3	4
$X^{(k)}$	+10	-10	-30	-290	-25510

não convergindo portanto ao valor desejado [$x = +5.774$]. Adotando o mesmo *chute inicial*: $X^{(0)} = +10$ e $\lambda = 0.04$ resulta:

k	0	1	2	3	4	5	6
$X^{(k)}$	10.000	4.000	5,560	5,560	5,778	5,773	5,774

convergindo ao valor desejado após 6 iterações.

O exemplo acima demonstra que a escolha de um valor adequado de λ é primordial para a convergência do procedimento. Por inspeção, verifica-se que, quando se utiliza o *chute inicial* igual – 10 (ou + 10 na busca do ponto de mínimo), o procedimento só converge com valores de λ menores do que 0.06, além disto verifica-se que este valor mínimo de λ pode depender do *chute inicial* adotado.

Métodos de Newton

A aplicação do método de *Newton-Raphson* à busca do zero da função gradiente, dá origem ao *Método de Newton* de busca do extremo de funções contínuas com as duas primeiras derivadas contínuas. Este algoritmo é traduzido pelo método recursivo:

$$X^{(k+1)} = X^{(k)} - \frac{f'\left[X^{(k)}\right]}{f''\left[X^{(k)}\right]}.$$

Na realidade este procedimento *atualiza* o valor de λ do método do gradiente em cada iteração k segundo: $\lambda^{(k)} = -\frac{1}{f''\left[X^{(k)}\right]}$, procedimento válido tanto para a *subida mais íngreme* quanto para a *descida mais íngreme*.

No exemplo ilustrativo anterior, tem-se: $f''(x) = 6 \cdot x$, assim:

$$X^{(k+1)} = X^{(k)} - \frac{3 \cdot \left[X^{(k)}\right]^2 - 100}{6 \cdot X^{(k)}} = \frac{3 \cdot \left[X^{(k)}\right]^2 + 100}{6 \cdot X^{(k)}}$$

Adotando este algoritmo com $X^{(0)} = -10$ resulta:

k	0	1	2	3	4
$X^{(k)}$	-10,000	-6,667	-5,833	-5,774	-5,774

com $X^{(0)} = +10$ resulta:

k	0	1	2	3	4
$X^{(k)}$	10,000	6,667	5,833	5,774	5,774

Convergindo a ambos os pontos em 4 iterações, sem a necessidade de pesquisar o valor de λ.

Generalização dos Métodos para o Problema Multivariável

O método do gradiente pode ser generalizado para uma função escalar de n variáveis de acordo com os algoritmos recursivos:

Método da *subida mais íngreme*	$\mathbf{X}^{(k+1)} = \mathbf{X}^{(k)} + \lambda \cdot \nabla f\left[\mathbf{X}^{(k)}\right]$
Método da *descida mais íngreme*	$\mathbf{X}^{(k+1)} = \mathbf{X}^{(k)} - \lambda \cdot \nabla f\left[\mathbf{X}^{(k)}\right]$

Onde : $\nabla f[\mathbf{x}] = \begin{pmatrix} \frac{\partial f(\mathbf{x})}{\partial x_1} \\ \frac{\partial f(\mathbf{x})}{\partial x_2} \\ \vdots \\ \frac{\partial f(\mathbf{x})}{\partial x_n} \end{pmatrix}$ é o *Vetor Gradiente* da função f(**x**).

Para ilustrar este procedimento a função: $f(\mathbf{x}) = (x_1 - 3)^2 + 9 \cdot (x_2 - 5)^2$ é considerada, esta função apresenta um ponto de mínimo em $\mathbf{x} = \begin{pmatrix} 3 \\ 5 \end{pmatrix}$ e o *Vetor Gradiente* desta função é : $\nabla f[\mathbf{x}] = \begin{pmatrix} 2 \cdot (x_1 - 3) \\ 18 \cdot (x_2 - 5) \end{pmatrix}$. O Método da *descida mais íngreme* aplicado a esta função é traduzido pelo procedimento recursivo:

$$\begin{pmatrix} X_1^{(k+1)} \\ X_2^{(k+1)} \end{pmatrix} = \begin{pmatrix} X_1^{(k)} \\ X_2^{(k)} \end{pmatrix} - \lambda \cdot \begin{pmatrix} 2 \cdot \left(X_1^{(k)} - 3 \right) \\ 18 \cdot \left(X_2^{(k)} - 5 \right) \end{pmatrix} = \begin{pmatrix} 6 \cdot \lambda + (1 - 2 \cdot \lambda) \cdot X_1^{(k)} \\ 90 \cdot \lambda + (1 - 18 \cdot \lambda) \cdot X_1^{(k)} \end{pmatrix}$$

considerando o *chute inicial* : $\mathbf{X}^{(0)} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$ e λ = 0.1 os seguintes resultados numéricos são obtidos:

k	$X_1^{(k)}$	$X_2^{(k)}$
0	1	1
1	1.4000	8.2000
2	1.7200	2.4400
⋮⋮⋮	⋮⋮⋮	⋮⋮⋮
51	3.0000	5.0000

Neste procedimento o valor de λ é mantido constante. Uma modificação do método pode ser feita no sentido de buscar, em cada iteração, o *valor ótimo* de λ [esta modificação faz com que

o *método do gradiente* seja conhecido por *método do gradiente com busca do tamanho do passo* - (em inglês – *line search*)]. Esta modificação é aplicada ao exemplo ilustrativo anterior onde:

$$\begin{pmatrix} X_1^{(k+1)} \\ X_2^{(k+1)} \end{pmatrix} = \begin{pmatrix} X_1^{(k)} \\ X_2^{(k)} \end{pmatrix} - \lambda \cdot \begin{pmatrix} 2 \cdot \left(X_1^{(k)} - 3 \right) \\ 18 \cdot \left(X_2^{(k)} - 5 \right) \end{pmatrix} = \begin{pmatrix} 6 \cdot \lambda + (1 - 2 \cdot \lambda) \cdot X_1^{(k)} \\ 90 \cdot \lambda + (1 - 18 \cdot \lambda) \cdot X_1^{(k)} \end{pmatrix}$$

assim o valor de f(**x**) no final do passo <u>k</u> seria:

$$f[\mathbf{X}^{(k)}, \lambda] = (1 - 2 \cdot \lambda)^2 \cdot \left[X_1^{(k)} - 3 \right]^2 + 9 \cdot (1 - 18 \cdot \lambda)^2 \cdot \left[X_1^{(k)} - 5 \right]^2$$

considerando $f[\mathbf{X}^{(k)}, \lambda]$ uma função apenas de λ, resulta:

$$\frac{\partial f[\mathbf{X}^{(k)}, \lambda]}{\partial \lambda} = -4 \cdot (1 - 2 \cdot \lambda) \cdot \left[X^{(k)} - 3 \right]^2 - 324 \cdot (1 - 18 \cdot \lambda) \cdot \left[X_1^{(k)} - 5 \right]^2$$

o valor *ótimo* de λ é o que minimiza $f[\mathbf{X}^{(k)}, \lambda]$ ou seja : $\left. \frac{\partial f[\mathbf{X}^{(k)}, \lambda]}{\partial \lambda} \right|_{\lambda_{otimo}} = 0$ resultando em:

$$\lambda_{otimo} = \frac{[X_1^{(k)} - 3]^2 + 81 \cdot [X_2^{(k)} - 5]^2}{2 \cdot [X_1^{(k)} - 3]^2 + 1458 \cdot [X_2^{(k)} - 5]^2}.$$

Considerando o mesmo *chute inicial* : $\mathbf{X}^{(0)} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$ e λ = 0.1 os seguintes resultados numéricos são obtidos:

k	$X_1^{(k)}$	$X_2^{(k)}$	$\lambda_{ótimo}$
0	1	1	0.0557
1	1.2228	5.011	0.4880
2	2.9573	4.9146	0.0557
3	2.9991	5.0002	0.4880
4	2.9992	4.9982	0.0557
5	3.0000	5.0000	0.4880

O procedimento recursivo neste caso converge em um número bem menor de iterações, entretanto cada iteração é mais *dispendiosa*(sob o ponto de vista computacional) podendo até resultar em um maior tempo computacional!

Para encerrar a apresentação de algoritmos tipo *gradiente* o *Método de Newton* é apresentado. Este procedimento quando aplicado a uma função escalar de n variáveis é traduzido pelo procedimento recursivo:

$$\mathbf{X}^{(k+1)} = \mathbf{X}^{(k)} - \left\{\mathbf{H}\left[\mathbf{X}^{(k)}\right]\right\}^{-1} \cdot \nabla f\left[\mathbf{X}^{(k)}\right]$$

onde: $\mathbf{H}(\mathbf{x})$ é a *Matriz Hessiana* de $f(\mathbf{x})$ que armazena os valores das derivadas segundas de $f(\mathbf{x})$ segundo: $H_{ij} = \frac{\partial^2 f(\mathbf{x})}{\partial x_i \partial x_j} = \frac{\partial^2 f(\mathbf{x})}{\partial x_j \partial x_i} = H_{ji}$ para i, j = 1, 2, ..., n.

No exemplo ilustrativo anterior a matriz Hessiana é uma matriz constante: $\mathbf{H}(\mathbf{x}) = \begin{pmatrix} 2 & 0 \\ 0 & 18 \end{pmatrix}$, isto ocorre devido ao fato de $f(\mathbf{x})$ ser uma função quadrática. Neste caso, e apenas no caso da função ser quadrática, o *Método de Newton* converge em apenas uma iteração independente do valor inicial adotado! . O custo computacional da implementação do *Método de Newton* é geralmente muito elevado pois envolve, em cada iteração, o cômputo das derivadas primeiras e segundas da função.

Se o seu problema puder ser resolvido por um método tipo gradiente, esta provavelmente será sua a melhor alternativa. No entanto, nem todos os problemas podem ser tratados por um método tipo gradiente. Seu uso é inviabilizado sempre que:

- houver descontinuidades na função objetivo f(x), ou houver mais de um mínimo (discutiremos esta característica mais adiante);
- não se puder escrever uma função objetivo f(x) diferenciável;
- calcular derivadas for inviável.

Nestes casos, precisaremos de métodos alternativos para resolver nossos problemas. Estes métodos freqüentemente serão métodos heurísticos de otimização.

voltar para INTRODUÇÃO

1.3. Métodos Heurísticos

Uma "heurística" é uma regra prática derivada da experiência. Não existe uma prova conclusiva de sua validade, e espera-se que a técnica heurística funcione muitas vezes mas não sempre. Uma heurística nos ajudará a encontrar soluções boas, mas não necessariamente ótimas.

A maior parte do que fazemos no nosso dia-a-dia tanto profissional quanto pessoal envolve a resolução heurística de problemas. Um exemplo: todos os dias, ao sair de casa para ir para o trabalho, você precisa escolher que caminho fazer. Você quer fazer o *melhor* caminho, mas a sua definição de melhor envolve vários fatores: tempo de percurso, aspectos de segurança, qualidade da pavimentação, probabilidade de ficar preso em um monstruoso engarrafamento,

entre outros. Para fazer a otimização do seu trajeto antes de sair de casa, você precisaria se conectar à Internet e buscar várias informações: verificar as condições de tráfego de sua cidade naquele momento, buscar por notícias relatando acidentes, verificar se há obras nos seus percursos, etc. Com estas informações, você pesaria prós e contras de cada rota e atribuiria uma nota (um número real) a cada um dos possíveis percursos. O percurso com a melhor nota naquela manhã seria o escolhido.

Certamente você não faz isto, ou não teria tempo para o desjejum. Você conhece, pela sua experiência, que em alguns horários algumas estradas devem ser evitadas, e outras favorecidas. No máximo, você toma seu desjejum ouvindo a uma rádio que dê boletins periódicos sobre o trânsito. Depois de algumas semanas, nem isto: você simplesmente entra no seu carro e vai para o trabalho, usando a rota que, pela sua experiência acumulada, lhe parece a melhor.

Suas heurísticas vão provavelmente funcionar na maioria das vezes, mas nem sempre: algum dia você pode ter problemas na rota escolhida. Mas você não se incomoda: se ficar preso em um engarrafamento, aproveita para a oportunidade para ligar para sua mãe, que vive cobrando mais atenção.

Um método heurístico de otimização pode ser determinístico ou estocástico, a depender se empregará ou não números sorteados aleatoriamente para executar seu algoritmo.

Os anos 80 foram marcados pelo ressurgimento de métodos heurísticos de otimização como ferramentas adicionais para tentar superar as limitações das heurísticas convencionais. Embora com filosofias distintas, estas metaheurísticas possuem em comum características que as distinguem das heurísticas convencionais, como por exemplo, incluir ferramentas para tentar escapar das armadilhas dos ótimos locais e a facilidade para trabalhar em ambientes paralelos. São exatamente estes métodos heurísticos – estocásticos e inspirados em fenômenos da natureza – o objeto de estudo deste curso.

voltar para INTRODUÇÃO

1.4. Quando usar Otimização Heurística

Há duas áreas que dependem fortemente de técnicas de otimização heurística: a Otimização Global e a Otimização Combinatória. Estes dois tipos de problemas são freqüentes na Engenharia Química.

a) Otimização Global

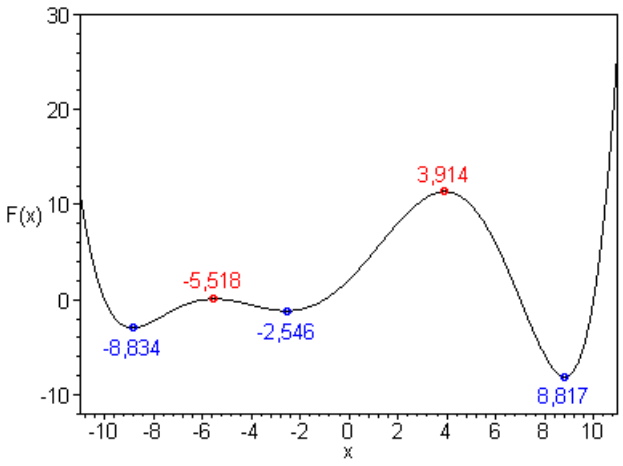
No problema de *Otimização Local*, busca-se um minimizador local da função real $F(\mathbf{x})$ onde $\mathbf{x}$ é um vetor de variáveis reais. Em outras palavras, busca-se um vetor $\mathbf{x}^*$ tal que $F(\mathbf{x}^*) \leq F(\mathbf{x})$ para todo $\mathbf{x}$ próximo a $\mathbf{x}^*$. O problema de *Otimização Global* consiste em encontrar um $\mathbf{x}^*$ que minimiza $\mathbf{F}(\mathbf{x})$ para todos os possíveis valores de $\mathbf{x}$. Este é um problema muito mais difícil, e

para a maioria das aplicações mínimos locais já são bons o suficiente, particularmente quando o usuário pode usar sua experiência para delimitar a região de busca e/ou fornecer um bom ponto de partida para o algoritmo.

Há critérios para determinar se um ponto é ou não um ótimo local. Por exemplo, para funções reais o critério de primeira derivada nula indica a existência de um extremo, e o sinal da segunda derivada indica se este extremo é um máximo ou um mínimo. No entanto, não existe qualquer critério que permita identificar um mínimo global. Para entender o porquê, considere o problema de escalar uma montanha com uma neblina muito densa. Mesmo com pouca visibilidade, é fácil dizer que você está no topo de uma montanha: ao tentar dar um passo adicional em qualquer direção, você verá que a tendência é de descida. Você certamente parará de caminhar e fincará sua bandeira. No entanto, como distinguir se você está no topo de uma montanha muito alta ou apenas no topo de um pequeno morro, como o preguiçoso abaixo? O morro é um exemplo de máximo local, e um algoritmo que apenas faça buscas locais buscando por soluções melhores não conseguirá escapar destas "armadilhas" para métodos de otimização global.

Os métodos que empregam derivadas são sempre otimizadores locais da função $F(\mathbf{x})$. A figura abaixo ilustra estes conceitos. Vamos observar a função polinomial F(x) abaixo.

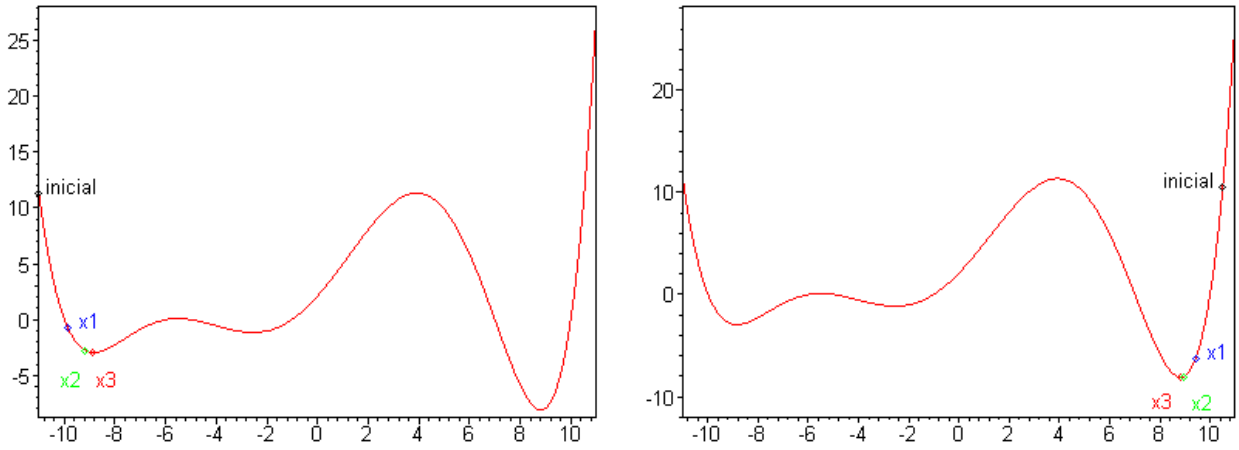
$$F(x) = \frac{1}{10000}(x+10)(x+6)(x+5)(x+1)(x-7)(x-10)$$



Observe que existem mínimos locais em x = -8,834, x = -2,546 e x = 8,817, sendo este último o mínimo global. A função apresenta máximos locais em x= -5,518 e x = 3,914. Um

método tipo gradiente não convergiria para o mínimo global desejado para qualquer condição inicial, como ilustrado nas figuras abaixo.

- se arbitrado x < -5,518, o método convergiria para x = -8,834;
- se arbitrado -5,518 < x < 3,914, o método convergiria para x = -2,546;
- se arbitrado x > 3,914, o método convergiria para x = 8,817.

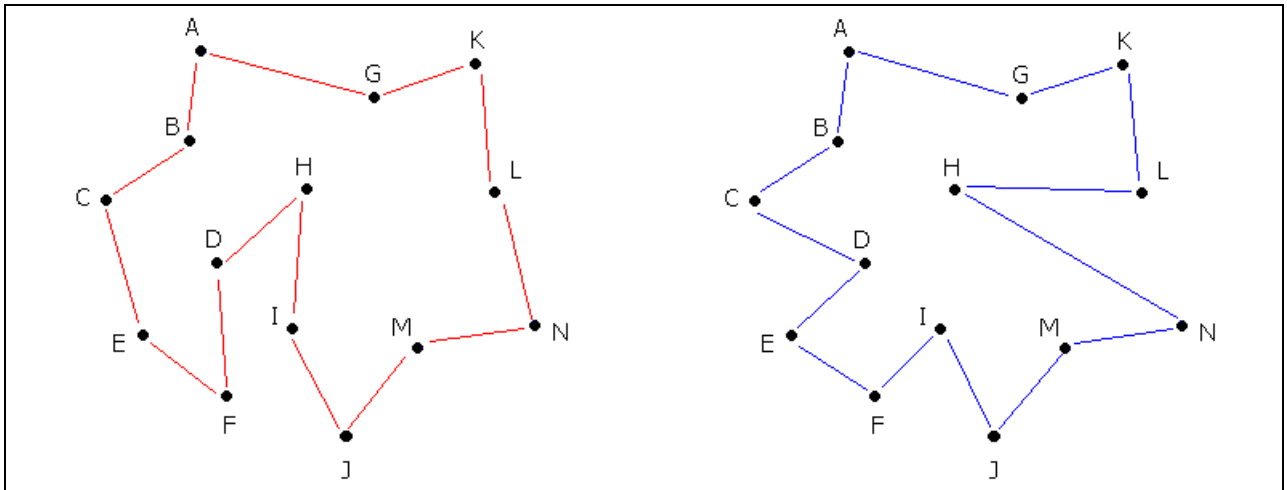


Para problemas difíceis como a Otimização Global, os métodos heurísticos de busca têm sido largamente estudados e utilizados. Não pode ser provado que estes métodos encontrarão uma solução ótima ou mesmo uma solução boa. No entanto, muitas vezes eles encontram a melhor solução conhecida, o que pode ser mais do que o suficiente para a aplicação em questão.

b) Otimização Combinatória

Uma outra área de aplicação de métodos heurísticos é a área de Otimização Combinatorial, ou Otimização Combinatória. Combinatorial geralmente significa que o espaço de estados é discreto (e.g., um espaço de tipos de moléculas, ou características de um produto). Este espaço pode ser finito ou apenas enumerável. Qualquer problema discreto pode ser visto como combinatorial.

Talvez o mais conhecido dos problemas combinatoriais seja o *Problema do Caixeiro Viajante*. Neste problema, imagina-se que um vendedor precise visitar um certo número de cidades e então voltar para casa. A tarefa do algoritmo é determinar a seqüência ótima de cidades a percorrer, de forma que a distância percorrida seja mínima. Observe na figura abaixo que alguns caminhos são melhores que outros, e determinar o menor caminho é um problema muito difícil. A função objetivo a minimizar no PCV clássico é a distância percorrida, mas pode ser adicionado um "peso" ao caminho. Assim, é possível que um trecho da estrada que esteja em boas condições seja favorecido em detrimento de um trecho perigoso ou com asfalto ruim, ou que setores onde se cobre pedágio sejam evitados.



Dois possíveis trajetos para ligar 14 cidades: qual o melhor?

Vários problemas reais e importantes podem ser interpretados como um PCV: minimização do tempo que um robô industrial gasta para soldar a carcaça de um automóvel; minimização do custo (em tempo ou combustível) da rota da distribuição diária de um jornal; minimização do tempo de abastecimento de várias bases militares envolvidas numa guerra, entre muitos outros. Tamanha a importância deste problema, a Companhia Proctor & Gamble fez um concurso em 1962 para determinar a menor rota que passasse por 33 cidades dos Estados Unidos. Houve empate entre muitos candidatos, que puderam encontrar a solução ótima.

O problema do caixeiro é um clássico exemplo de *problema de otimização combinatória*. A primeira coisa que podemos pensar para resolver esse tipo de problema é reduzi-lo a um *problema de enumeração*: achamos todas as rotas possíveis e, usando um computador, calculamos o comprimento de cada uma delas e então vemos qual a menor. (É claro que se acharmos todas as rotas estaremos contando-as, daí podermos dizer que estamos reduzindo o problema de otimização a um de enumeração).

Para acharmos o número R(n) de rotas para o caso de n cidades, basta fazer um raciocínio combinatório simples e clássico. Por exemplo, no caso de n = 4 cidades, a primeira e última posição são fixas, de modo que elas não afetam o cálculo; na segunda posição podemos colocar

qualquer uma das 3 cidades restantes, e uma vez escolhida uma delas, podemos colocar qualquer uma das 2 restantes na terceira posição; na quarta posição não teríamos nenhuma escolha, pois sobrou apenas uma cidade; consequentemente, o número de rotas é 3 x 2 x 1 = 6. De modo semelhante, para o caso de n cidades, como a primeira é fixa, o número total de escolhas que podemos fazer é (n-1) x (n-2) x ... x 3 x 2 x 1. De modo que, usando a notação de fatorial, R(n) = (n-1)!. Nossa estratégia consiste então em: gerar cada uma dessas R(n) rotas, calcular o comprimento total das viagens de cada rota e ver qual delas tem o menor comprimento total. Trabalho fácil para o computador, diria alguém. Bem, talvez não. Vejamos o porquê.

Suponha que esteja disponível um computador muito rápido, capaz de fazer 1 bilhão de adições por segundo (1 Gflops) . Isso parece uma velocidade imensa, capaz de tudo. No caso de 20 cidades, o computador precisa apenas de 19 adições para dizer qual o comprimento de uma rota, sendo capaz então de calcular 10^9 / 19 = 53 milhões de rotas por segundo. Contudo, essa imensa velocidade é um nada frente à imensidão do número 19! de rotas que precisará examinar. Acredite se puder, o valor de 19! é 121 645 100 408 832 000 (ou , aproximadamente, 1,2 10^{17} em notação científica). Consequentemente, ele precisará de 1,2 10^{17} / (53 milhões) = 2,3 10^9 segundos para completar sua tarefa, o que eqüivale a cerca de 73 anos.

A grande dificuldade dos problemas combinatoriais é que a quantidade de possíveis soluções cresce com uma velocidade fatorial, e rapidamente o computador torna-se incapaz de enumerar todas as possíveis soluções do problema. Na tabela abaixo, estes cálculos foram executados para alguns outros tamanhos de problema.

número de cidades	Capacidade [rotas/s]	rotas a avaliar [rotas]	tempo computacional
5	2,50 10^8	2,40 10^1	Insignificante
10	1,11 10^8	3,63 10^5	~ 0,003 s
15	7,14 10^7	8,72 10^{10}	~ 3,3 s
20	5,26 10^7	1,22 10^{17}	73 anos
25	4,17 10^7	6,20 10^{23}	470 milhões de anos
n	$10^9/(n-1)$	(n-1)!	(n-1)! / [$10^9/(n-1)$]

Observe que o aumento no valor do n provoca uma ligeira redução na capacidade do computador em avaliar rotas (ela diminui apenas de um sexto ao aumentar n de 5 para 25), mas provoca um aumento gigantesco no tempo total de cálculo (o número de rotas a avaliar muda da ordem de 10^1 para 10^{23}). Em outras palavras: a inviabilidade computacional é devida à presença da função fatorial na medida do esforço computacional. Com efeito, se essa complexidade fosse expressa em termos de um polinômio em n o nosso computador seria perfeitamente capaz de suportar o aumento de n. Confira isso na seguinte tabela, que corresponde a um esforço computacional polinomial R(n) = n^5:

número de cidades	Capacidade [rotas/s]	rotas a avaliar , n^3 [rotas]	tempo computacional
5	$2{,}50\ 10^8$	$1{,}25\ 10^2$	$5{,}00\ 10^{-7}$ s
10	$1{,}11\ 10^8$	$1{,}00\ 10^3$	$9{,}00\ 10^{-6}$ s
15	$7{,}14\ 10^7$	$3{,}38\ 10^3$	$4{,}73\ 10^{-5}$ s
20	$5{,}26\ 10^7$	$8{,}00\ 10^3$	$1{,}52\ 10^{-4}$ s
25	$4{,}17\ 10^7$	$1{,}56\ 10^4$	$3{,}75\ 10^{-4}$ s
n	$10^9/(n-1)$	n^3	$n^3 / [10^9/(n-1)]$

Então o método reducionista não é prático (a não ser para poucas cidades), mas será que não pode-se inventar algum método prático (por exemplo, envolvendo esforço polinomial na variável n) para resolver o problema do caixeiro? Apesar de inúmeros esforços, ainda não foi encontrado este método. Na verdade, a existência ou não de um método polinomial para resolver o problema do caixeiro viajante é um dos grandes problemas em aberto da Matemática na medida em que S. A. COOK (1971) e R. M. KARP (1972) mostraram que uma grande quantidade de problemas importantes podem ser reduzidos, em tempo polinomial, ao problema do caixeiro.

Consequentemente: se descobrirmos como resolver o problema do caixeiro em tempo polinomial ficaremos sendo capazes de resolver, também em tempo polinomial, uma grande quantidade de outros problemas matemáticos importantes; por outro lado, se um dia alguém provar que é impossível resolver o problema do caixeiro em tempo polinomial no número de cidades, também se terá estabelecido que uma grande quantidade de problemas importantes não tem solução prática.

Não se conhece um algoritmo polinomial para encontrar a solução ótima de um PCV, mas é possível construir um algoritmo polinomial para testar se um conjunto de soluções propostas é a tentativas são soluções do problema. E o PCV é representante de uma classe enorme de problemas, que podem ser reduzidos a ele. Costuma-se resumir essas propriedades do problema do caixeiro dizendo que ele pertence à categoria dos problemas não-determinísticos polinomiais completos, ou **NP - completos**.

Para problemas de otimização combinatória, não há algoritmos que levem à solução ótima em um tempo viável. Assim, usam-se sempre métodos heurísticos – determinísticos ou aleatórios - para chegar a soluções que se possam por em prática, ainda que não sejam ótimas (ou não tenhamos provas de que o são). O algoritmo que se utiliza para a obtenção de soluções aproximadas tem como finalidade que as soluções sejam encontradas num tempo razoável para os fins práticos a que se destina, e que a sua qualidade seja o melhor possível tendo em conta as limitações de tempo.

O problema do caixeiro viajante serve atualmente de problema padrão (benchmark) para testar algoritmos de otimização combinatória.

voltar para INTRODUÇÃO

1.5. Métodos de Otimização Natural: analogias com a natureza

A partir da década de 1950, foram criados vários algoritmos heurísticos na tentativa de simular fenômenos biológicos. Estes algoritmos, que são na verdade algoritmos de otimização, têm alguns aspectos em comum. O mais marcante é seu caráter aleatório, na tentativa de simular o acaso que parece governar processos distintos na natureza, desde a evolução das espécies até o comportamento social dos animais.

Na década de 1980, com a explosão da computação, se tornou viável empregar estes algoritmos para a otimização de funções e processos quando métodos mais tradicionais não tinham sucesso: problemas de otimização combinatória (e.g. problema do caixeiro viajante e problemas de coloração de mapas), problemas onde a função objetivo não pode ser expressa matematicamente (e.g. identificação de suspeitos) ou problemas com vários mínimos locais.

Vários métodos heurísticos de otimização surgiram motivados por fenômenos da natureza, como o Recozimento Simulado (*Simulated Annealing*), os Algoritmos Genéticos, as Técnicas de Enxame (*Swarm Algorithms*) e as técnicas de Otimização por Colônia de Formigas (*Ant Colony Optimization*).

a) Recozimento Simulado (Simulated Annealing)

O método de *Recozimento Simulado* (S*imulated Annealing*) é um tipo de método de busca local de implementação extremamente simples, que se origina do processo de recozimento de metais (METROPOLIS *et al.*, 1953; EGLESE, 1990). Neste processo, os metais são aquecidos acima de seu ponto de fusão e resfriados lentamente, de forma a se organizarem em estruturas cristalinas de mínima energia. Em métodos de busca local clássicos, a partir da solução atual é gerada uma nova solução-tentativa, e esta substitui a anterior se o valor da função objetivo (energia) associada for menor. Na técnica de recozimento simulado, transições para soluções de maior energia são permitidas segundo uma certa probabilidade, evitando assim que o método fique preso na bacia de atração de um mínimo local. Esta probabilidade está ligada à temperatura do sistema físico original, e diminui com o passar das iterações. A estratégia de "resfriamento" do método é um dos aspectos mais críticos em sua implementação. Como o método trabalha com uma solução apenas, é sugerida por vários autores a execução em paralelo de várias cópias do algoritmo partindo de condições iniciais distintas.

b) Algoritmos Genéticos (AG)

Algoritmos Genéticos (AG) são métodos relativamente recentes que não usam qualquer informação de derivada e, por isto, apresentam boas chances de não serem aprisionados em ótimos locais. Provas completas de sua convergência não foram demonstradas ainda. No entanto, sua aplicação em problemas práticos geralmente leva para o ótimo global ou pelo menos para soluções mais satisfatórias que as fornecidas por outros métodos. Este método se baseia no processo de seleção natural e evolução da espécies, principalmente no paradigma de sobrevivência dos mais aptos. Os indivíduos que apresentam melhor adequação (menor valor da função objetivo) apresentam maior probabilidade de se reproduzirem e passarem para a próxima geração seus genes bem sucedidos. No entanto, todos os indivíduos (aptos ou não) estão sujeitos a mutações aleatórias de tempos em tempos.

Em sua implementação, os Algoritmos Genéticos requerem a definição de vários parâmetros que afetam o desempenho do algoritmo de várias formas. O tamanho da população (o número de indivíduos formando a população) deve ser suficientemente grande para garantir diversidade suficiente e assim cobrir bem o espaço de soluções. Outros parâmetros como a probabilidade de cruzamento, taxa de mutação e mecanismos de mutação e cruzamento afetam o AG de forma menos significativa. Não há valores de parâmetros reconhecidamente ótimos, apenas faixas sugeridas de trabalho (MICHALEWICZ, 1996). Mesmo sendo muito mais rápido que métodos tipo busca exaustiva, ainda são métodos muito lentos se comparados com métodos do tipo gradiente, já que não empregam qualquer informação referente à derivada da função objetivo (GOLDBERG, 1989).

c) Técnicas de Enxame (Particle Swarm Optimization, PSO)

As técnicas de enxame (KENNEDY e EBERHART, 1995) exploram a analogia com o comportamento social de animais, como enxames de abelhas, cardumes de peixes ou bandos de pássaros. Nestes, cada indivíduo do grupo toma suas próprias decisões, mas sempre de alguma forma baseado na experiência do líder do grupo. Matematicamente, cada indivíduo do bando é considerado um ponto do espaço n-dimensional e a velocidade deste indivíduo, a direção de busca a ser usada nesta candidata a solução. A direção de busca em uma iteração é determinada através da ponderação entre a experiência daquela solução e da melhor solução já encontrada pelo grupo (metaforicamente, a solução líder). Ressalta-se que o peso de inércia w é empregado para controlar o impacto da história prévia de velocidade na velocidade atual. Um maior valor de w favorece a exploração global, enquanto um peso de inércia menor tende a facilitar exploração local. Seleção satisfatória do peso de inércia w fornece, então, um equilíbrio entre capacidade de exploração global e local. Os principais parâmetros para o método são as ponderações entre as experiências individual e coletiva (c_1 e c_2) e o fator de inércia w.

d) Ant Colony(AC)

As técnicas de colônia de formigas (DORIGO, MANIEZZO e COLORNI, 1996) baseiam-se no comportamento das formigas que apresentam a formidável capacidade de, malgrado serem insetos praticamente cegos, conseguirem estabelecer o menor caminho entre o formigueiro e a fonte de alimento e retornar da mesma forma. Etologistas (segundo o Houaiss – 2001- aquele que se dedica ao estudo do comportamento social e individual de animais!) verificaram que é através de uma substância química chamada *feromônio* que as formigas *trocam* informações relativas aos caminhos. Uma formiga em movimento secreta uma certa quantidade de feromônio no solo, criando assim uma trilha desta substância biologicamente muito ativa. Enquanto uma formiga isolada se move de uma forma absolutamente aleatória, uma formiga encontrando uma trilha já percorrida por outra(s) formiga(s) detecta seu rastro e decide com alta probabilidade seguir o mesmo caminho, reforçando desta forma a trilha escolhida com seu próprio feromônio. Este comportamento coletivo cooperativo é classificado como *autocatalítico,* pois quanto maior o número de formigas percorrendo um caminho mais atrativo torna-se o mesmo. Este processo é então caracterizado por uma retroalimentação (*feedback*) positiva, uma vez que a probabilidade de uma formiga escolher um determinado caminho é tanto maior quanto maior for o número de formigas que já o tenha percorrido.

A tabela seguinte mostra a popularização destes métodos junto à comunidade científica. Vê-se, pelo número de artigos publicados em periódicos indexados internacionais, que os três primeiros são mais tradicionais. Os recentes PSO e AC ainda estão sendo mais discutidos em conferências e congressos.

	até 1986	1987	1988	1989	1990	1991	1992	1993	1994	1995	1996	1997	1998	1999	2000	2001	2002	total
SA	0	**2**	12	11	25	17	19	26	36	76	96	93	95	83	88	84	83	**846**
Tabu	0	0	0	0	0	**1**	1	0	4	13	27	33	49	36	22	26	33	**245**
GA	0	0	0	0	**3**	2	7	26	22	145	206	278	269	274	296	363	259	**2150**
AC	0	0	0	0	0	0	0	0	0	0	0	**1**	0	2	5	4	3	**15**
PSO	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**2**	2	**4**

voltar para INTRODUÇÃO

1.6. Métodos Híbridos

Métodos puramente heurísticos incorporam pouca informação do sistema de equações, e conseqüentemente não são os mais eficientes do ponto de vista computacional (especialmente nas cercanias da solução onde métodos de busca local são capazes de resolver o problema). Desta forma, métodos híbridos de algoritmos de busca local e heurísticos de otimização parecem mais apropriados. É freqüentemente reportada na literatura a capacidade destes métodos híbridos de escapar de bacias de atração de mínimos locais ainda retendo boa eficiência computacional (GOLDBERG, 1989).

O tipo de método híbrido mais freqüentemente citado é aquele em que um método heurístico é utilizado para gerar um certo número de boas soluções, candidatas à solução ótima. Então, um método de busca local é utilizado para transformar estas soluções em um conjunto de mínimos locais. Esta população melhorada é realimentada no algoritmo heurístico até que o critério de terminação do algoritmo seja satisfeito. Várias variações deste método podem ser criadas, modificando o número de candidatas otimizadas e a freqüência desta otimização.

Tais híbridos, ao invés de apresentarem operadores ou representações padrão, incorporam o conhecimento do problema a resolver por parte do usuário, de forma a produzir soluções melhores e produzi-las ainda mais rapidamente. Mesmo sendo menos genéricos e de implementação um pouco mais sofisticada, sua utilização em problemas reais, ou de dimensões maiores, vêm se mostrando bem mais eficiente, mais do que compensando suas possíveis desvantagens.

Um exemplo de aplicação bem sucedida de um algoritmo genético híbrido foi apresentado por KRAGELUND (1997). O problema foi definir escalas de trabalho de médicos durante as férias de verão obedecendo a várias restrições, classificadas em leves (apenas preferências, não deveriam mas poderiam ser violadas) e severas (não poderiam ser violadas de forma alguma). Para resolver o problema, foi utilizado um híbrido de AG com um método de busca local. Sua implementação foi feita através de um operador que atua em 3 etapas:

- localizar todas as restrições violadas por este quadro de horários (candidato a solução)
- selecionar randomicamente uma das restrições violadas
- usar um método de otimização local simples, que varia com a restrição, e reduzir sua violação.

Segundo o autor, método simples de busca local já são suficientes para resultados satisfatórios serem obtidos. Os algoritmos híbridos mostraram-se superiores aos simples Para exemplificar, foram compilados alguns resultados apresentados pelo autor na tabela a seguir, onde os números representam violações às restrições leves e severas, perfazendo 397 e 25 restrições respectivamente. O critério de parada correspondeu a 120 minutos de computação.

Compilação dos resultados apresentados por KRAGELUND (1997)

Método Empregado	**melhor**		**média**		**pior**		
	severas	**leves**	**severas**	**leves**	**severas**	**leves**	**sucesso**
Busca Randômica	106	23	114,6	20,9	121	21	0 / 20
Steepest Descent	20	16	26,7	17,6	32	19	0 / 20
AG + Busca Local, comum	0	4	1,4	4,3	3	4	1 / 20
AG + Busca Local, subpop.	0	2	0,6	3,5	2	2	10 / 20

voltar para INTRODUÇÃO

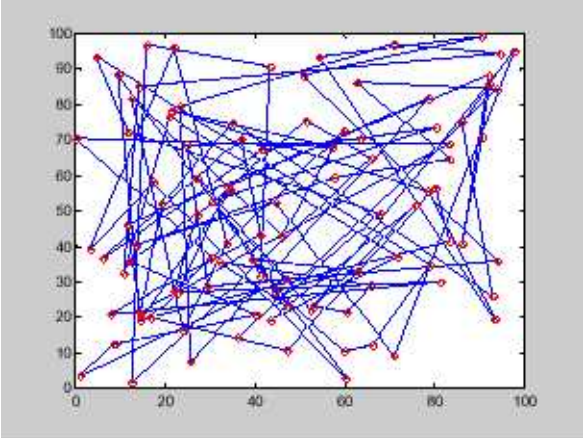
1.7. Quando não usar Otimização Heurística

Se for possível empregar um método tipo gradiente para resolver o seu problema, ele sem dúvida será o algoritmo mais eficiente que há, já que ele incorpora muito mais informação sobre o espaço de busca.

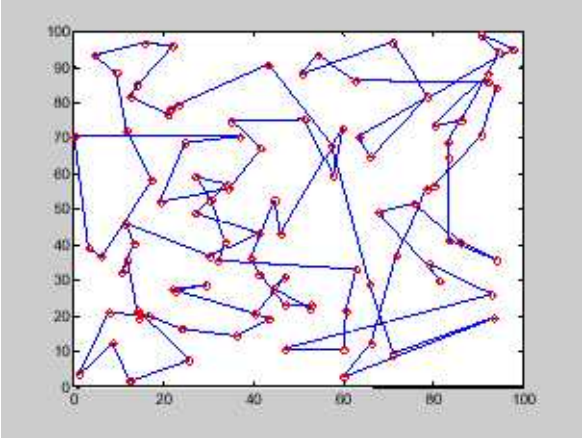
Os método heurísticos são freqüentemente chamados de “methods of last resort”, algo a ser empregado quando tudo o mais não funciona. Isto porque estes métodos são via de regra lentos, ou pelo menos muito mais lentos para resolver o mesmo problema que um método que empregue derivadas. No entanto, muitas vezes estes métodos são os únicos de que se dispõem, e consequentemente são os melhores!

Com o avanço da capacidade dos computadores e das linguagens de programação, o custo computacional associado a estes métodos se tornou muito menos assustador, e qualquer PC consegue resolver um problema razoável em alguns minutos. Por exemplo, considere o seguinte exemplo apresentado por ZUBEN: resolver o problema do caixeiro viajante para 100 cidades. O autor empregou um Algoritmo Genético, e foram testados 400.000 soluções dentre as possíveis $9{,}33\ 10^{155}$. O tempo de simulação em um Pentium III 450 MHz foi de 287 segundos, pouco menos de 5 minutos. As figuras abaixo, reportadas pelo autor, mostram a evolução da resposta obtida com o número de iterações do algoritmo.

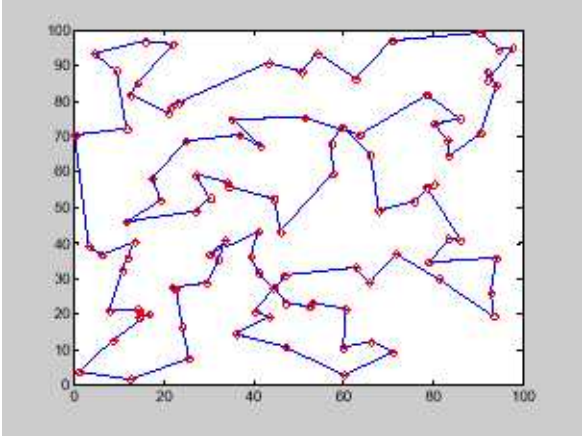
Melhor indivíduo na população inicial



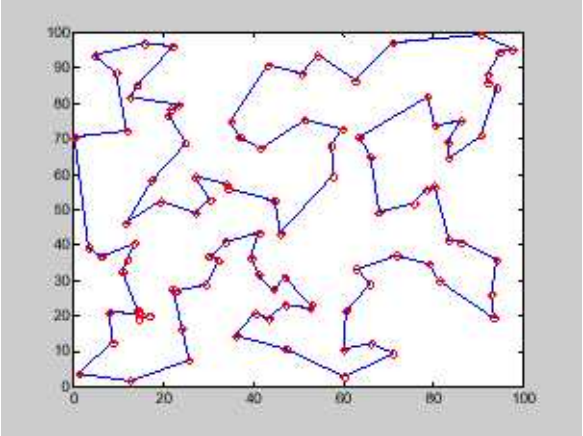
Melhor indivíduo após 500 gerações



Melhor indivíduo após 2000 gerações



Melhor indivíduo após 4000 gerações



voltar para INTRODUÇÃO

1.8. Cômputo de números pseudo-aleatórios

Usar um computador – que apenas segue instruções deterministicamente pré-programadas – para sortear números verdadeiramente aleatórios soa como uma contradição. Por isto, os números gerados por computador são mais apropriadamente denominados de números pseudo-aleatórios, ou pseudo-randômicos.

Como destacado anteriormente, as metaheurísticas propostas inspiradas na natureza têm em comum seu caráter estocástico. Isto significa que várias decisões do algoritmo se baseiam em números sorteados aleatoriamente segundo alguma distribuição de probabilidades. Há várias distribuições contínuas de probabilidade, e a função que as caracteriza é a função *Densidade de Probabilidade* (*DP*). A probabilidade de que um número verdadeiramente aleatório encontre-se no intervalo fechado [x1,x2] é dada pela integral abaixo:

$$P(x_1 \le x \le x_2) = \int_{x_2}^{x_1} DP(x)\ dx$$

que corresponde à área sob a curva DP(x) entre x_1 e x_2.

A distribuição mais usada pelos métodos de otimização estocásticos é a distribuição *uniforme*, segundo a qual qualquer número dentro do domínio [a,b] tem a mesma chance de ser sorteado. Matematicamente, a função densidade de probabilidade da distribuição uniforme é dada por:

$$DP(x) = \begin{cases} \dfrac{1}{(b-a)} & a \le x \le b \\ 0 & otherwise \end{cases}$$

A probabilidade de sortear um número dentro de um intervalo [c,d] é proporcional ao tamanho deste intervalo, e é igual à fração do domínio que este intervalo representa. Em outras palavras,

$$P(c \le x \le d) = \int_c^d \frac{1}{b-a}\ dx = \frac{d-c}{b-a}$$

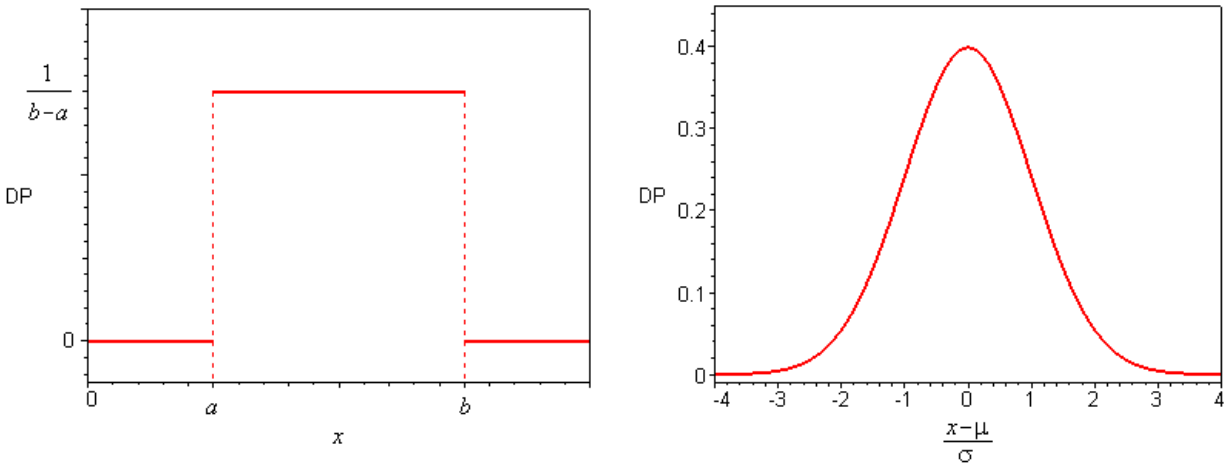
Algumas aplicações também usam a distribuição *normal*, segundo a qual um valor *x* é tão mais provável de ser sorteado quanto mais próximo for da média μ , e desvios para mais ou para menos são igualmente prováveis. A função densidade de probabilidade possui um forma de sino e é representada pela função:

$$DP(x) = \frac{1}{\sqrt{2\pi\sigma}} \exp\left[-\frac{1}{2}\left(\frac{x-\mu}{\sigma} \right)^2 \right]$$

A probabilidade de sortear um número dentro de um intervalo [c,d] é <u>não é</u> proporcional ao tamanho do intervalo, e é calculada por:

$$P(c \le x \le d) = \int_c^d DP(x)\,dx = \frac{1}{2}\sqrt{\sigma}\; erf\left(\frac{x-\mu}{\sigma\sqrt{2}}\right)$$

onde *erf*(x) é a função erro. Há várias tabelas para calcular esta integral, e todos os *softwares* matemáticos possuem a função erro embutida. As figuras abaixo mostram a forma das funções densidade de probabilidade para estas duas distribuições.



Qualquer que seja a distribuição de probabilidades do seu interesse, apenas números inteiros uniformemente distribuídos são gerados nos computadores. Algoritmos específicos (também chamados de filtros) são usados para converter os números de uma distribuição em outra. (Veja os exercícios 4 e 5).

Um gerador de números pseudo-aleatórios em um computador digital é uma função determinística chamada *função de transição* que produz uma seqüência de números, os quais emulam uma variável aleatória uniformemente distribuída. Como esta função leva de um número para outro e há uma quantidade finita de números que se pode representar no computador, esta função é periódica. O que se deseja é que ela tenha o maior período possível, idealmente da ordem da quantidade de números "representáveis".

Esta função é implementada como $rand_{k+1} = f(rand_k)$, e é necessário fornecer um número ($rand_0$) para dar partida no algoritmo. Este número é conhecido como *semente*, e sua definição é muito importante. Se for usada a mesma semente em duas execuções sucessivas de um código, exatamente a mesma seqüência de números aleatórios será gerada. Esta característica é usada quando se quer comparar o efeito de algum parâmetro do método. Na maioria das vezes, porém, esta repetição deve ser evitada. Muitos trabalhos sugerem construir o número semente a partir da data e hora da execução do programa, de forma que cada execução do seu algoritmo tenha números aleatórios diferentes.

Não é uma tarefa simples desenvolver um gerador de números pseudo-aleatórios, e há alguns algoritmos reportados na literatura. É importante que se use um bom gerador de números aleatórios. Ser bom, do ponto de vista do usuário, significa que: (i) os números gerados seguem a distribuição de probabilidades indicada; (ii) os números gerados não são correlacionados; (iii) o

gerador possui repetitividade, ou seja, gerará a mesma seqüência de números aleatórios sempre que receber a mesma semente; (iv) seu custo computacional é baixo.

Qualquer *software* matemático possui um ou mais geradores de números randômicos embutidos, e uns certamente são melhores que outros. Para programadores em linguagens Fortran e C, há subrotinas descritas e/ou implementadas nos pacotes como LINPACK e LAPACK e nos manuais tipo *Numerical Recipes*. Abaixo encontra-se a implementação em Fortran de um destes algoritmos.

```
c###################################################################
c  SUBROUTINE RAN3
c  Returns a uniform random deviate between 0.0 and 1.0.  Set idum to
c  any negative value to initialize or reinitialize the sequence.
c  This function is taken from W.H. Press', "Numerical Recipes" p. 199.
c###################################################################
      subroutine ran3(idum,rand)
      implicit double precision (a-h,m,o-z)
      parameter (mbig=4000000.,mseed=1618033.,mz=0.,fac=1./mbig)

c     According to Knuth, any large mbig, and any smaller (but
c     still large) mseed can be substituted for the above values.

      Dimension ma(55)
      Data iff /0/
      if (idum.lt.0 .or. iff.eq.0) then
         iff=1
         mj=mseed-dble(iabs(idum))
         mj=dmod(mj,mbig)
         ma(55)=mj
         mk=1
         do i=1,54
            ii=mod(21*i,55)
            ma(ii)=mk
            mk=mj-mk
            if(mk.lt.mz) mk=mk+mbig
            mj=ma(ii)
         enddo
         do k=1,4
            do i=1,55
               ma(i)=ma(i)-ma(1+mod(i+30,55))
               if(ma(i).lt.mz) ma(i)=ma(i)+mbig
            enddo
         enddo
         inext=0
         inextp=31
         idum=1
      endif
      inext=inext+1
      if(inext.eq.56) inext=1
      inextp=inextp+1
      if(inextp.eq.56) inextp=1
      mj=ma(inext)-ma(inextp)
      if(mj.lt.mz) mj=mj+mbig
      ma(inext)=mj
      rand=mj*fac
      return
C-----end of subroutine ran3----------------------------------------
      end
```

voltar para INTRODUÇÃO