Ministério da Ciência e Tecnologia – MCT

Instituto Nacional de Pesquisas Espaciais – INPE

ALGORITMOS EVOLUTIVOS PARA PROBLEMAS DE OTIMIZAÇÃO NUMÉRICA COM VARIÁVEIS REAIS

Alexandre César Muniz de Oliveira

Monografia apresentada para o Exame de Qualificação do Curso de Computação Aplicada – CAP

São José dos Campos - SP
Setembro-2001

CAPÍTULO I

1. Introdução

Otimização é a busca da melhor solução para um dado problema dentro de um conjunto finito ou infinito de possíveis soluções. O processo de busca pode partir de uma solução inicial ou de um conjunto delas, realizando melhoramentos progressivos até chegar a um outro conjunto que contenha uma ou todas as melhores soluções possíveis dentro do espaço de busca.

Problemas de otimização podem ser formulados genericamente como:

Otimize f(x),	$x=(x_1, x_2, x_3,.., x_m)^T \in \Re^m$
Sujeito ou não	a restrições de igualdade (na forma $c_j(x) = 0$)
	e restrições de desigualdade (na forma $c_j(x) \geq 0$)

Otimizar significa minimizar custos ou maximizar ganhos, ou seja, encontrar pontos de mínimo ou máximo valor para f(x). Uma vez sujeito a condições de restrição, c(x), do espaço de busca, o problema é chamado de restrito.

Devido a natureza da função objetivo, o problema pode ser linear, quadrático ou não-linear. As variáveis de controle, por sua vez podem assumir valores reais, inteiros ou ambos.

Quando se tem um universo enumerável de possíveis combinações/permutações de elementos que contabilizam custos ou ganhos que se pretende minimizar ou maximizar, tem-se uma classe chamada de otimização combinatória.

Uma solução x* para um problema de minimização é chamada mínima global quando não existir outra x, pertencente ao espaço de busca, cujo o valor da função objetivo $f(x) < f(x^*)$. Em problemas de maximização, o chamado máximo global x* atende a $f(x^*) > f(x)$ para todo x pertencente ao espaço de busca.

Se a solução x possui f(x) mínimo apenas dentro de uma certa região em torno de x, chamada de vizinhança de x, diz-se que o mínimo é local. Os mínimos locais podem ser

boas soluções, mas não são as melhores. Para certos métodos de busca esses pontos são indesejáveis, pois interrompem a busca por soluções melhores.

A função objetivo pode ter um ou muitos pontos de mínimos, o que define se ela é uno ou multimodal. Uma função unimodal apresenta apenas um ponto de mínimo ou máximo. Uma função multimodal, por sua vez, possui várias inflexões de sua superfície, o que caracterizam múltiplos pontos de mínimo ou máximo.

Problemas irrestritos possuem mínimos globais ou locais em pontos chamados de estacionários (sem declive), quando o gradiente de x nesse ponto é zero. Em problemas restritos, isso pode não acontecer. Nesse caso, um ponto mínimo ou máximo pode ser uma fronteira gerada por uma restrição. A Figuras 1.1 mostra exemplos de pontos de mínimos em uma função multimodal.

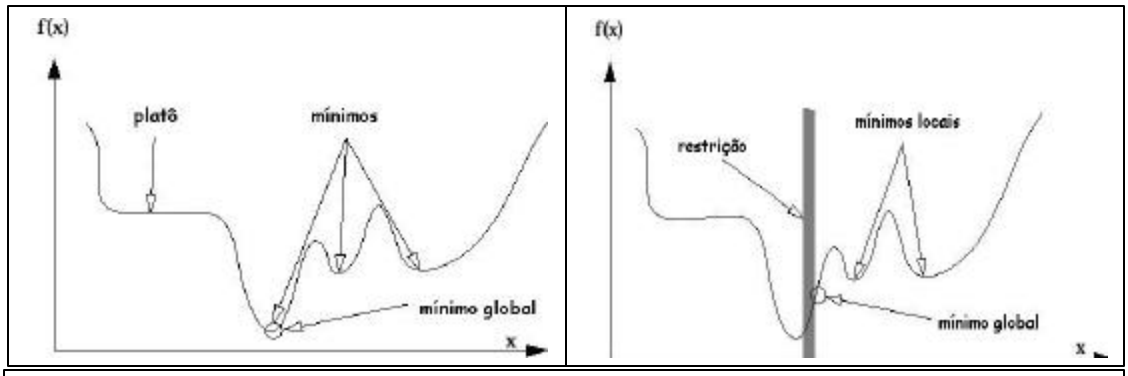


Figura 1.1 - Exemplo de pontos de mínimo: (a) sem restrição os mínimos são regiões estacionárias; (b) com restrição, os mínimos podem ocorrer em regiões

1.1 Métodos para Solução de Problemas de Otimização

O cálculo diferencial fornece técnicas para determinar os pontos de mínimo/máximo de uma função. Se x é um ponto de inflexão, então necessariamente a primeira derivada nesse ponto é zero. Dessa forma, igualando-se a primeira derivada a zero, tem-se uma equação, cuja as raízes são os pontos de inflexão da função. Para saber se esses pontos são de máximo ou de mínimo, recorre-se a segunda derivada. Se esta for maior que zero, tem-se um mínimo nesse ponto.

As funções multimodais possuem vários pontos de inflexão e, por isso, esses métodos analíticos não são eficazes para se encontrar o ponto ótimo (máximo ou mínimo) da função. Por isso, recorre-se aos mais variados tipos de algoritmos com suas diferentes estratégias e aplicabilidades.

A solução para problemas de otimização, em geral, é obtida a partir de uma configuração inicial I_0 que contém uma solução X_0 (ou um conjunto P_0 delas) e controles A_0 específicos do algoritmo a ser empregado.

Seguidas iterações são necessárias para se melhorar a qualidade dessa solução (ou soluções) até que se chegue a uma condição de término que pode ser a obtenção da solução ótima. Um determinado algoritmo é sempre mais apropriado a determinada classe de problema. As pesquisas investem sempre mais naqueles métodos que têm maior aplicabilidade. A Figura 1.2 ilustra esse processo.

```
BuscaGenérica
        Escolher X0 e A0 ;
        t = 0;
        Xt = X0;
        At = A0
        Enquanto (Não CondiçãoDeTérmino)
                t = t+1;
                Xt = MétodoDeMelhoramento(Xt-1,At-1);
                At = AtualizaConfiguração(At-1);
        FimEnquanto;
Fim;
```

Figura 1.2 - Algoritmo de busca genérico

Algoritmos com característica de conseguir obter uma solução ótima a partir de um ponto qualquer do espaço de busca é considerado um algoritmo global. Por sua vez, algoritmos locais estão mais dependentes de configurações iniciais ou pontos de partida, uma vez que tendem a seguir superfícies de funções e portando atingirem pontos estacionários a partir dos quais não conseguem mais melhorar a solução. Portanto, algoritmos locais são menos robustos que os algoritmos globais. A Figura 1.3 mostra o comportamento de algoritmos locais em relação aos pontos iniciais x_0.

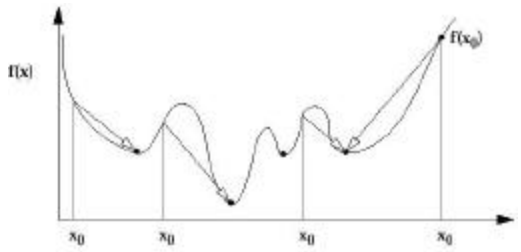


Figura 1.3 - Algoritmo de busca local

Uma outra característica relevante é o comportamento do algoritmo de otimização quanto a sua previsibilidade para uma certa entrada. Por exemplo, dada uma configuração inicial (entrada I_0), um algoritmo determinístico sempre chega na mesma configuração final I_t. Por outro lado, existem os chamados algoritmos estocásticos (não determinísticos) cujos os passos não podem ser previstos a partir de uma configuração inicial. A Figura 1.4 mostra o comportamento dos algoritmos determinísticos e estocásticos.

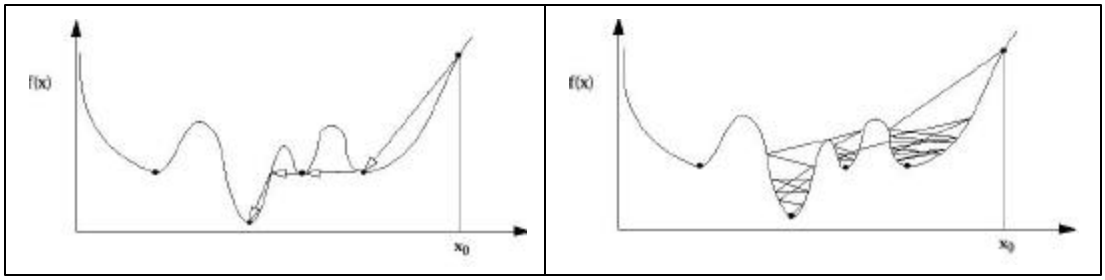


Figura 1.4 - Algoritmos de busca global: (a) determinístico; (b) estocástico

Estão dentro da classe de algoritmos estocásticos, os chamados busca tabu (*tabu search*) algoritmo evolutivo (*evolutionary algorithm*) e recozimento simulado (*simulated annealing*), cujo exemplo pode ser observado na Figura 1.5.

```
Simulated Annealing
        Escolher um ponto arbitrário X0 ;
        Escolher um fator de decaimento a;
        Inicializar a temperatura T suficientemente alta;
        Xt= X0;
        Enquanto (T > 0)
                Enquanto (tentativas)
                        Xt = Vizinhança (Xt-1);
                        ÇE = E(Xt) - E(Xt-1);
                        Se (ÇE < 0)
                                Aceita  Xt incondicionalmente;
                        Se não
                                Aceita  Xt com probabilidade exp(-ÇE/T);
                Fim
                T = a.T;
        FimEnquanto;
Fim;
```

Figura 1.5 - *Simulated annealing*

A função de energia *E*, em problemas de otimização, pode ser modelada como a função objetivo. Observe que o método de melhoramento do *simulated annealing* reside no acaso de encontrar uma solução melhor dentro de uma certa vizinhança, em geral produzida por

uma perturbação aleatória em X_i . Ele é considerado global por que permite movimentos laterais e até piores, principalmente no início do processo, quando a temperatura T ainda está alta, possibilitando fugir de mínimos locais. No decorrer do esfriamento tais movimentos vão tendo menor probabilidade de ocorrer. Espera-se que o sistema "esfrie" em um ponto de mínimo global. Por trabalhar a partir de uma única solução inicial, o recozimento simulado é um método menos robusto que o algoritmo evolutivo que trabalha com população de soluções.

1.2 O Enfoque Evolutivo

Os mecanismos naturais que promovem a evolução dos seres vivos podem ser considerados processos inteligentes. A maioria das pesquisas sobre evolução estuda o comportamento inteligente de seres vivos em coletividade buscando se auto-organizar para atingir um dado objetivo. Nestes termos, são incluídos na modelagem, mecanismos de auto-organização, adaptação, evolução, competição e cooperação, dentre outros.

Os algoritmos evolutivos possuem características que os tornam aplicáveis a uma vasta quantidade de aplicações. Algumas características desejáveis são:

a) são métodos de otimização global, robustos e podem encontrar várias soluções;
b) podem otimizar um grande número de parâmetros discretos, contínuos ou combinações deles;
c) realizam buscas simultâneas em várias regiões do espaço de busca (paralelismo inerente);
d) utilizam informações de custo ou ganho e não necessitam obrigatoriamente de conhecimento matemático do problema (tais como derivadas);
e) podem ser eficientemente combinados com heurísticas e outras técnicas de busca local;
f) são modulares e facilmente adaptáveis a qualquer tipo de problema.

Naturalmente, também possuem algumas desvantagens:

g) como trabalham com população de soluções, podem ser mais lentos que outras alternativas;
h) possuem parâmetros que devem ser bem ajustados para obter eficácia

1.3 Objetivo do Trabalho

Este trabalho focaliza os algoritmos evolutivos para problemas de otimização numérica com variáveis reais. Inicialmente, são apresentados alguns fundamentos destes algoritmos, principalmente os relacionados com a codificação de soluções e operadores evolutivos básicos para variáveis reais (Capítulo 2). Posteriormente, são apresentados algumas propostas para solução de problemas de otimização numérica sem restrição e com restrição (Capítulos 3 e 4). Por último, são apresentados alguns experimentos publicados, bem como os resultados mais significativos produzidos por eles (Capítulo 5).

CAPÍTULO II

2. Algoritmos Evolutivos para Variáveis Reais

A teoria da computação evolutiva está fundamentada em alicerces oriundos da observação de processos biológicos, que interpreta a evolução natural como uma forma inteligente de adaptação, auto-organização e otimização. A designação "evolutivo", que muitas vezes se confunde com "genético", é utilizado neste trabalho sempre que estiver em discussão aspectos gerais dessa classe de algoritmos. Entretanto, procurou-se usar um termo ou outro respeitando a auto-designação dos próprios autores com relação a suas propostas.

A idéia básica dos algoritmos evolutivos é a de manter uma população de indivíduos (ou cromossomos), representando soluções candidatas para problemas concretos, que evolui ao longo do tempo (ou gerações) através de um processo de competição, onde os mais aptos (melhores *fitness*) têm maiores chances de sobreviver e se reproduzir. A reprodução se baseia em um processo de seleção de indivíduos e modificação das soluções candidatas que eles representam, através de operadores como cruzamento (ou *crossover*) e mutação. A Figura 2.1 mostra um exemplo de algoritmo evolutivo.

```
AlgoritmoEvolutivo
        t=0;
        Gere uma população P0 ;
        Defina valores para os parâmetros evolutivos;
        Avalie a aptidão de cada X Î P0;
        Enquanto (condição)
                t =t+1;
                Selecione Pt de Pt-1;
                Recombine Pt;
                Avalie a aptidão de cada X Î Pt;
                X* =Melhor(Pt);
        FimEnquanto;
Fim;
```

Figura 2.1 - Algoritmo evolutivo genérico

Pensando em termos de otimização numérica, o processo de busca da solução está camuflado na evolução da população ao longo das gerações. Espera-se que o mais evoluído indivíduo represente a solução ótima. Por isso, a avaliação de aptidão do indivíduo deve refletir todas as caraterísticas desejáveis para a solução do problema. Em geral, a aptidão do indivíduo é a própria função objetivo do problema.

O processo de geração de uma população inicial pode ser direcionado para subespaços de busca mais promissores do espaço de busca ou, se não, deve promover um espalhamento uniforme neste, objetivando ganho de diversidade que é a capacidade de uma população manter o máximo de informação diferenciada sobre determinado problema.

No decorrer das gerações, a população de soluções candidatas funciona como uma base de conhecimento a cerca do problema em questão, onde bons fragmentos de informação estão diluídos nos indivíduo bem avaliados e nos mal avaliados também. Ambos são continuamente selecionados e recombinados, gerando mais informação.

Os algoritmos evolutivos possuem alguns parâmetros cujos os valores precisam ser atribuídos adequadamente para ganho de desempenho. Os mais comuns são a pressão de seleção, probabilidade de cruzamento e de mutação.

Este capítulo apresenta aspectos gerais dos algoritmos evolutivos, tais como codificação e operadores evolutivos (basicamente cruzamento e mutação) específicos para problemas com variáveis reais, encontrados na literatura. O operador de seleção, que não manipula diretamente os cromossomos, não é discutido neste trabalho.

2.1 Codificação Binária X Codificação Real

Os algoritmos evolutivos são aplicados nas mais diferentes áreas de conhecimento e, por isso, representações de soluções em indivíduos vão desde pontos de um hiperplano, centros de conjuntos *fuzzy*, até instruções de programa.

A codificação (ou representação) de soluções de problemas em termos de indivíduos é uma problemática geral a todas as abordagens de algoritmos evolutivos. Problemas de otimização podem ser modelados de várias formas, onde as variáveis de controle podem assumir valores numéricos reais, inteiros ou até, especificamente, binários (considerando 1/0 como não inteiros).

A forma como um problema é modelado influencia fortemente a forma de representação dessas variáveis nos algoritmos evolutivos. É típico utilizar cadeias de números inteiros para representar uma permutação como solução para o problema do caixeiro viajante [1] ou minimização de leiaute de portas em circuitos VLSI [2]. Assim como pode ser natural representar soluções de problemas de p-medianas como uma cadeia de 1´s e 0´s [3].

Problemas de otimização numérica de funções com varáveis reais, por sua vez, podem ter soluções representadas por uma cadeia binária *y* de comprimento *c*, segundo a equação:

$$x = a + int(y) * fator, \qquad (2.1)$$

$$fator = (b-a)/2^{c}-1, \qquad (2.2)$$

Onde (a,b) é o intervalo real desejado. A tabela I mostra 4 exemplos de conversão de binários (com comprimento 8) para reais, em um intervalo entre (0, 1.024):

Binário	Inteiro	Fator	Real
0000 0001	1	0.004	0.004
0000 0010	2	0.004	0.008
1111 0000	240	0.004	0.960
1111 1111	255	0.004	1.024

Tabela I - Exemplo de codificação binário-real com fator de precisão 0.004

Cada cadeia binária significaria uma única variável de controle num problema de otimização numérica. Ou seja, em problemas multivariáveis, são necessárias múltiplas cadeias iguais a estas para representar todas as variáveis a serem otimizadas.

A primeira vista, parece intuitivamente mais natural pensar diretamente em termos de números reais e projetar novos operadores específicos para os problemas em questão, quando se tem uma maior proximidade entre a codificação e próprio domínio do problema. Apesar disso, decisões relativo à conveniência de codificação binária ou real são ainda objetos de discussão entre alguns pesquisadores.

Por um lado, os aspectos teóricos da codificação binária já estão bem desenvolvidos desde os primeiros algoritmos genéticos de Goldberg e de Jong [4,5]. Enquanto que, ainda são recentes os avanços teóricos da codificação real [6, 7].

Por outro lado, a alta representatividade de esquemas binários reduz significantemente o tamanho do espaço de busca e, em certos casos, essa redução do espaço de busca tem um preço pago na precisão e na qualidade das soluções encontradas. Ou seja, para se aumentar necessariamente a qualidade numérica das soluções, tem-se que aumentar em muito o espaço de otimização na codificação binária.

Uma importante desvantagem da codificação binária, com implicações sérias na velocidade do algoritmo, está na contínua necessidade de conversão de um número real para uma cadeia binária e vice-versa.

Um outro fator pró-codificação real reside no que tange a distância entre soluções vizinhas. Por exemplo, seja um valor real x, sua codificação y, um ruído δ e uma relação de vizinhança para um valor z (binário ou real) definida por $N(z,\delta)$; tem-se que:

- qualquer δ aplicado a x gera um número dentro da vizinhança de x, ou seja:
  $\forall\delta\ (x+\delta) \in N(x,\delta)$; Enquanto que
- nem todo δ aplicado a y gera um número dentro da vizinhança de y, ou seja:
  $\exists\delta\ (y+\delta) \notin N(y,\delta)$.

Essa característica é significativa para o desempenho de aplicações codificadas em binário, pois cria uma distância não natural entre soluções teoricamente vizinhas com sérias conseqüências na busca fina por soluções de melhor qualidade. Uma pequena variação nas variáveis de controle pode causar uma grande variação na função objetivo. Ou seja, a distância entre *genótipo* (codificação) e *fenótipo* (espaço de busca) é bem maior na codificação binária.

Defensores da codificação binária, entretanto, colocam essa característica como desejável para os mecanismos de exploração do espaço de busca, permitindo que qualquer ponto seja mais facilmente visitado a partir da aplicação de qualquer operação evolutiva básica (cruzamento, mutação ou inversão).

De qualquer forma, existe uma variação para codificação binária, chamada codificação *gray*, que normaliza as distâncias de *hamming* (dH) para valores numéricos vizinhos. Observe a cadeia de 3 bits *gray*, comparada a cadeia binária comum:

Decimal	Binário	dH	Gray	dH
0	000	-	000	-
1	001	1	001	1
2	010	2	011	1
3	011	1	010	1
4	100	3	110	1
5	101	1	111	1
6	110	2	101	1
7	111	1	100	1

Tabela II - Comparação das codificações gray e binária para 3 bits. A distância de hamming dH se refere aos valores vizinhos e é constante apenas na codificação gray.

Michalewicz realizou vários experimentos com diferentes funções-teste, utilizando vários operadores genéticos voltados tanto para a representação real, quanto para a representação binária. Ficou evidenciado um significante ganho de desempenho (velocidade/qualidade) para a versão codificada em real[8].

Nos últimos anos, tem havido um crescente incremento em aplicações evolutivas utilizando a codificação real. Os operadores evolutivos mais comuns, que operam com este tipo de codificação, são apresentados na próxima seção.

2.2 Operadores Evolutivos

Um algoritmo de otimização global deve ser capaz de explorar pontos inteiramente novos dentro do espaço de busca, bem como intensificar a busca em determinadas regiões consideradas promissoras. Esse mecanismo de diversificação/intensificação (exploration/explotation) é obtido nos algoritmos evolutivos pela correta aplicação dos operadores evolutivos.

Goldberg formalizou as principais características dos operadores clássicos de cruzamento (*crossover*), mutação (*mutation*) e inversão (*inversion*), principalmente no que tange a codificação binária [5]. Entretanto quando se fala de codificação real, as várias implementações desses operadores encontradas na literatura alteram significantemente seus objetivos.

Há consenso, pelo menos, que o cruzamento utiliza a informação contida em dois (ou mais) indivíduos (pais) para gerar um (ou mais) novo indivíduo (filho). Esse processo tende a não acrescentar novas informações à população, por explorar apenas a região próxima aos indivíduos pais.

A mutação, por sua vez, pode ser entendida tanto como um *diversificador* ou como um *intensificador* de busca. Em algumas abordagens, como estratégias de evolução, a mutação é a única responsável pela evolução e o que determina se o movimento é de exploração ou intensificação são parâmetros adaptáveis ao longo das gerações [9].

A mutação diversifica quando introduz uma informação inteiramente nova no indivíduo e, consequentemente à população. Por outro lado, quando apenas aplica um ruído à solução

contida no indivíduo, a mutação é um mecanismo *intensificador* de busca na vizinhança dessa solução.

A seguir são apresentados alguns operadores de cruzamento e mutação encontrados na literatura, com suas principais características e aplicação.

2.2.1 Cruzamentos

Considere os indivíduos $x^1 = (x^1_1, x^1_2,..., x^1_m)$ e $x^2 = (x^2_1, x^2_2,..., x^2_m)$, chamados de pais, que representam soluções para *f(x)* e foram previamente selecionados para uma operação de cruzamento (*crossover*) qualquer. Considere também k (k=1,2,...,) indivíduos x^{fk} de mesmas dimensões, resultado desta operação, por isso chamado de descendentes. Pode-se dizer que x^1, x^2 e x^{kf} são vetores em $\Re^m$.

Cruzamentos convencionais adaptados

Os cruzamentos *n-ponto* e *uniforme*, utilizados na codificação binária, podem ser adaptados para a codificação real. A versão chamada de *cruzamento simples* apenas troca partes das informações contidas nos pais para seus descendentes, sem gerar nenhuma nova informação. O exemplo a seguir ilustra o cruzamento *n-ponto* para n=1 (um ponto de cruzamento x_p):

$x^1=$	$(x^1_1, x^1_2, \dots, x^1_p,$	$x^1_{p+1}, \dots, x^1_m)$
$x^2=$	$(x^2_1, x^2_2, \dots, x^2_p,$	$x^2_{p+1}, \dots, x^2_m)$
$x^{f1}=$	$(x^1_1, x^1_2, \dots, x^1_p,$	$x^2_{p+1}, \dots, x^2_m)$
$x^{f2}=$	$(x^2_1, x^2_2, \dots, x^2_p,$	$x^1_{p+1}, \dots, x^1_m)$

Figura 2.2 - Exemplo de um cruzamento 1-ponto

O cruzamento *n-ponto* tende a disseminar os mesmos valores entre todas as variáveis de controle de um problema de otimização. Figura 2.3 ilustra o cruzamento *n-ponto* para variáveis reais no espaço 2-D:

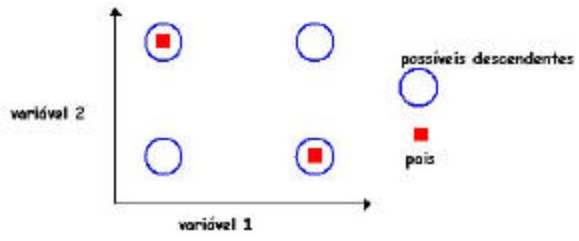


Figura 2.3 - cruzamento n-ponto em 2 dimensões

O cruzamento *flat* [10] melhora este processo por gerar um único descendente $x^f=(x^f_1, x^f_2,..., x^f_m)$, onde cada valor de variável x^f_i é aleatoriamente escolhido do intervalo (x^1_i, x^2_i).

Cruzamentos Aritméticos

Existem vários tipos de cruzamentos que geram descendentes através de simples operações aritméticas sobre os pais. Pode-se gerar descendentes deterministicamente baseando-se em médias. Os cruzamentos *média aritmética* (2.3) e *média geométrica* (2.4) são exemplos simples, mostrados a seguir:

$$x^f = \frac{(x^1 + x^2)}{2} \qquad (2.3)$$

$$x^f = \sqrt{(x^1 \cdot x^2)} \qquad (2.4)$$

Um dos mais populares cruzamentos aritméticos é o proposto por Michalewicz, que gera dois descendentes, resultado da combinação linear dos pais [7].

$$x^{1f} = a\ x^1 + (1\text{-}a)\ x^2 \qquad (2.5)$$

$$x^{2f} = (1\text{-}a)\ x^1 + a\ x^2 \qquad (2.6)$$

onde $0< a <1$. Algumas versões deste cruzamento existem em função de variações no parâmetro $\boldsymbol{a}$:

- $\boldsymbol{a}$ = 1/2 (média garantida) [11]: gera apenas um descendente e eqüivale ao cruzamento média aritmética (2.3);
- $\boldsymbol{a}$ = valor aleatório pertencente ao intervalo (0,1) [7];
- $\boldsymbol{a}$ = varia em função do número de gerações (cruzamento não uniforme) [12].

Algoritmos que utilizam o cruzamento aritmético não uniforme, iniciam com $\boldsymbol{a}$ próximos a 0.5 e decrementam-no linearmente ao longo das gerações até próximo a 0. Isto permite explorar combinações lineares inicialmente mais longe dos pais e posteriormente bem mais próximas, refinando cada vez mais o processo de busca. A Figura 2.4 ilustra esse comportamento. Observe que a maior distância dos pais *Dmax* é obtida para $a=0.5$ que eqüivale ao cruzamento *média aritmética* e produz apenas um único descendente exatamente na metade da distância euclidiana entre os dois pais.

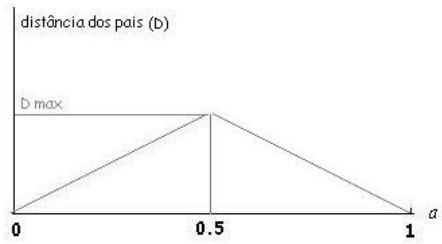


Figura 2.4 - Comportamento do cruzamento aritmético para diferentes a

Como será visto nos próximos capítulos, os problemas de otimização podem estar sujeitos pelo menos a restrições de intervalo, tais como: $L_i \leq x_i \leq U_i$, onde L_i e U_i são constantes que formam um vetor de limites inferiores e superiores (limites de viabilidade) respectivamente para cada uma das variáveis de controle x_i.

A simplicidade dos cruzamentos vistos até agora reside também no fato destes não embutirem preocupação extra com limites das variáveis de controle, pois geram descendentes em uma linha que passa pelos pais, mas não extrapola os limites definidos por eles. A tendência natural da população, dessa forma, é convergir para uma espécie de média aritmética de todos os indivíduos. Uma outra conseqüência disso, é que pais, cujas as variáveis de controle estejam com valores dentro dos limites de viabilidade, sempre geram descendentes dentro desses limites.

O cruzamento *Blend-*α (BLX-α), ao contrário, consegue gerar combinações lineares fora dos limites impostos pelos pais e, por isso já requerem cuidados adicionais com a viabilidade dos descendentes [13]. A idéia básica do BLX-α é aumentar um pouco a área entre os dois pais, possibilitando a geração de descendentes fora dela. A partir de uma das equações (2.5) ou (2.6) pode-se chegar a:

$$x^f = x^1 + \alpha\,(x^2 - x^1) \qquad (2.7)$$

onde α é escolhido aleatoriamente do intervalo $(-\alpha, 1+\alpha)$. A Figura 2.5 mostra onde seriam gerados os possíveis descendentes para este tipo de cruzamento.

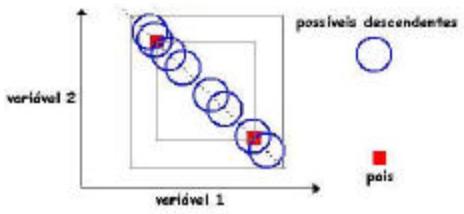


Figura 2.5 - BLX-α para 2 dimensões

O BLX-α pode explorar algo mais que a linha que conecta os dois pais. Para isso, basta que *a* seja um vetor de valores aleatórios entre $(-\alpha,1+\alpha)$, o que permitiria, no exemplo da Figura 2.5, gerar descendentes em qualquer ponto dentro do retângulo maior.

A idéia de explorar mais pontos dentro do hiperplano gerado pelos pais, tem motivado projetos de cruzamentos mais elaborados que operam com mais indivíduos e não se limitam a distribuições uniformes entre eles.

O cruzamento *Unimodal Normal Distribution Crossover* - (UNDX) trabalha com 3 pais cujas distâncias são usadas para definir os desvios-padrão que geram uma área com distribuição normal, onde podem ser gerados os descendentes [14]. A Figura 2.6 ilustra este processo.

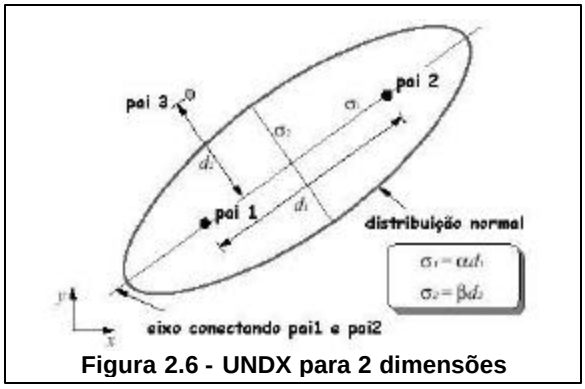


Figura 2.6 - UNDX para 2 dimensões

Conforme mostra a Figura 2.6, os desvios-padrão, $\hat{a}_1$ e $\hat{a}_2$, são proporcionais às distâncias d_1(entre *pai1* e *pai2*) e d_2(entre *pai3* e o eixo *pai1-pai2*), respectivamente. Os dois descendentes x^{1f} e x^{2f} podem ser gerados em qualquer ponto dentro da circunferência com distribuição normal, cuja a média está centrada no eixo *pai1-pai2*.

O UNDX, assim como o *Simulated binary crossover(SBX)* [15], utilizam distribuições não uniformes para geração de área de cruzamentos. A Figura 2.8 mostra a comparação das duas distribuições utilizadas por esses métodos. Ambos têm parâmetros de ajuste de área que podem tornar a geração de descendentes mais ou menos distante dos pais. O UNDX tende a gerar descendentes próximos ao ponto médio no eixo *pai1 -pai2*, enquanto que com o SBX, por sua vez, ocorre exatamente o contrário.

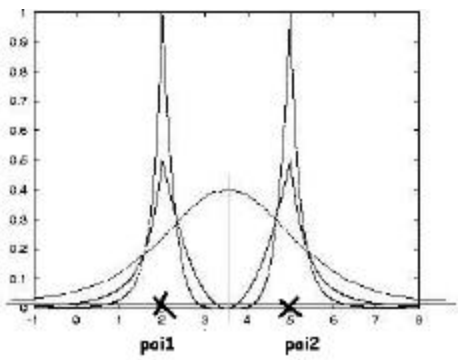


Figura 2.7 - Distribuição do UNDX e do SBX

O cruzamento *Simplex* (SPX) trabalha com *m+1* pais que formam um simplex, a partir do qual, são gerados *m+1* descendentes [16]. Em $\Re^2$, por exemplo, são selecionados 3 pais que formam um simplex a ser expandido, baseado em um parâmetro ε (similar ao parâmetro α do BLX). A Figura 2.7 ilustra este exemplo.

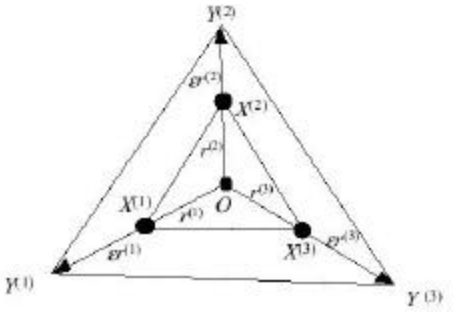


Figura 2.8 - SPX para 2 dimensões

Para o caso geral, cada um dos *k* descendentes é gerado da seguinte forma:

$$x^{kf} = (1+\varepsilon)\,(x^{k} - O) \qquad (2.8)$$

onde *O* é o centro de massa do simplex, dado por:

$$O = \frac{1}{k}\sum_{J=1}^{K} x^{J} \qquad (2.9)$$

Os descendentes resultados de cruzamentos que extrapolam as áreas determinadas por seus pais, podem também estar extrapolando a área de viabilidade (ou os limites de viabilidade de uma de suas variáveis de controle). Em codificação binária o problema de fronteira de viabilidade não ocorre, uma vez que as cadeias binárias têm , por definição, comprimentos

pré-fixados. No caso de codificação real, é necessário o uso de técnicas auxiliares que mantém os indivíduos dentro desses limites.

Cruzamentos Heurísticos

A eficiência dos cruzamentos aritméticos se baseia na força do processo de seleção: bons pais geram bons filhos. Eles não se utilizam de nenhuma informação do problema para gerar descendentes melhores.

Existem cruzamentos, entretanto, que avaliam direções e geram descendentes nas mais promissoras. Por isso são chamados de cruzamentos *heurísticos* ou *direcionais.* O mais simples deles é o proposto em [6] que gera um único descendente, a partir de dois pais, de forma similar equação (2.7):

$$x^f = x^2 + a\ (x^2 - x^1) \qquad (2.10)$$

onde $a \in (0,1)$ e avaliação $f(x^2)$ é "melhor" que $f(x^1)$ (maximização ou minimização).

Se não for gerado dependentes dentro dos limites de viabilidade, há duas possibilidades: (a) ser repetido o processo até um certo número de tentativas, ao final das quais, o processo desiste e não gera nenhum descendentes; (b) o descendente é reparado através de outra combinação linear em direção a x^2 que é viável. Essas soluções para geração de descendentes inviáveis pode ser implementada nos cruzamentos aritméticos que possuem a mesma propriedade.

2.2.2 Mutação

Considere o indivíduo (ou vetor solução em $\Re^m$) $x = (x_1, x_2, x_3,., x_m)$, previamente selecionado para uma operação de mutação qualquer. Considere também o *i-ésimo* gene de x, escolhido para ter seu valor modificado, gerando o gene x'_i.

A mutação mais elementar, pode ser entendida como uma adaptação da mutação binária. Nesta, o bit eleito assume um outro valor qualquer válido, no caso o complemento de x'_i. A mutação *aleatória* (ou *uniforme*) [7] faz o mesmo para uma gama maior de possibilidades, pois x_i pode assumir qualquer valor, com distribuição de probabilidade uniforme, no intervalo (L_i, U_i). Ela permite um relativo ganho de diversidade, pois introduz, no indivíduo, informação genética nova sem qualquer relação ou afinidade com a informação substituída.

Para equipar a mutação *aleatória* também com o poder de intensificar a busca em uma dada região, Michalewicz propôs a mutação *não-uniforme* [7]. A idéia é sintonizar melhor os processos de diversificação e intensificação com a situação do sistema evolutivo. A mutação *não-uniforme* usa o parâmetro tempo para ir alterando linearmente o padrão de geração de novos valores para x_i.

Com igual probabilidade, x_i pode ser ou incrementado ou decrementado, conforme as equações (2.11 e 2.12):

$$x_i^{'} = \begin{cases} x_i + q(t, U_i - x_i) \\ x_i - q(t, L_i - x_i) \end{cases} \qquad (2.11)$$

$$q(t, y) = y \cdot r \cdot (1 - \frac{t}{T})^b \qquad (2.12)$$

onde r é um número aleatório entre (0,1), T é o número máximo de gerações e b é o grau de não uniformidade ou de dependência sobre o número de gerações. Para t crescente, observa-se que, nas primeiras gerações, $\theta(t,y)$ tende a gerar deslocamentos aleatórios maiores para x_i, diversificando mais. Quando t se aproxima do limite máximo T, as variações aleatórias vão se tornando menores, intensificando a busca próximo a x_i.

Outros tipos de mutação como as mutações *gaussinana* [9], *modal discreta* e *modal contínua* [17] e a mutação de *Mühlenbein* [18] (mostrada a seguir) são baseadas em distribuição de probabilidade:

$$x_i^{'} = xi \pm R_i \cdot g \qquad (2.13)$$

onde

$$g = \sum_{k=0}^{15} a_k 2^{-k} \qquad (2.14)$$

e $R_i = (U_i - L_i)$, $a_k \in \{0,1\}$ com probabilidades:

- $p(a_k=1) = 1/16$,
- $p(a_k=0) = 15/16$ e
- $p(sinal = \pm) = 0.5$.

Esta mutação tende a explorar mais a vizinhança de x_i, com mínima proximidade de $R_i \cdot \frac{1}{2^{15}}$

2.3 Experimentos com Operadores Evolutivos

Existem vários estudos com relação ao desempenho de operadores evolutivos para codificação real. A maioria deles propõe novas alternativas e as compara com as já existentes. Uma importante contribuição, para análise de desempenho dos diversos operadores evolutivos encontrados na literatura, foi dada por Herrera[12]. Ele montou uma bancada de experimentos onde agrupou cruzamentos e mutações com diferentes parâmetros perfazendo um total de 21 versões diferentes de algoritmos evolutivos, alguns dos quais, propostos por ele, com características de auto-adaptação usando lógica *fuzzy* (que não fazem parte do escopo deste trabalho). Cada uma dessas versões foi executada 5 vezes para 3 diferentes funções-teste (f1,f2 e f3). Foram elas:

$$f_1(\vec{x}) = \sum_{i=1}^{n} x_i^2$$
$$f_2(\vec{x}) = \sum_{i=1}^{n-1} (100 \cdot (x_{i+1} - x_i^2)^2 + (x_i - 1)^2)$$
$$f_3(\vec{x}) = \frac{1}{4000} \sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$$

Figura 29 - Funções-teste usadas por Herrera[12]

Os parâmetros que não influenciavam o desempenho dos operadores de mutação e cruzamento utilizados (por exemplo, tamanho da população, probabilidade de cruzamento e mutação) foram mantidos constantes para todos os experimentos. Em cada teste com uma função, foi computada a melhor solução obtida para 100, 500, 1000 e 5000 gerações; e calculada a média nos 5 testes.

As Figuras 2.11 e 2.12 mostram, respetivamente, as versões de algoritmos evolutivos (com diferentes operadores e seus parâmetros associados) e os resultados de cada um deles para o teste com a terceira função (função de *Griewangk* [19]).

VERSÃO	MUTAÇÃO	CRUZAMENTO
RCGA1	Random	Simple
RCGA2	Non-uniform ($b = 5$)	Simple
RCGA3	Random	Arithmetical ($\lambda = 0.5$)
RCGA4	Non-uniform ($b = 5$)	Arithmetical ($\lambda = 0.5$)
RCGA5-α	Non-uniform ($b = 5$)	BLX-α ($\alpha = 0, .15, .3, .5$)
RCGA6	Non-uniform ($b = 5$)	Linear
RCGA7	Mühlenbein	Discrete
RCGA8	Mühlenbein	Extended line
RCGA9	Mühlenbein	Extended intermediate
RCGA10	Modal Discrete ($B_m = 2$, $rang_{min} = 1.0e-05$)	Extended intermediate
RCGA11	Modal Continuous ($B_m = 2$, $rang_{min} = 1.0e-05$)	Extended intermediate
RCGA12	Non-uniform ($b = 5$)	Wright's heuristic

Figura 2.10 - Versões dos algoritmos evolutivos testados por Herrera [12]

Pode ser observado na Figura 2.12 que aparece uma versão binária, o BCGA (binary coded genetic algorithm), que também nestes experimentos não figurou entre aqueles com melhores resultados. Para a função f3 fica evidente que os melhores resultados, ao longo das 4 amostragens, foram obtidos pela versão RCGA-5-0.5 (inclusive o melhor, 3.9e-03).

Versão-parâmetro	100	500	1000	5000
BCGA	3.3e+01	1.7e+00	7.5e+02	7.8e-02
RCGA1	1.2e+01	1.4e+00	1.1e+00	3.7e-01
RCGA2	2.1e+01	1.2e+00	1.0e+00	1.7e-02
RCGA3	6.1e+00	1.3e+00	1.0e+00	1.1e-01
RCGA4	5.3e+00	1.1e+00	9.4e-01	8.4e-03
RCGA5-0.0	4.0e+00	1.0e+00	2.6e-01	7.4e-03
RCGA5-0.15	2.9e+00	8.5e-01	1.5e-01	3.2e-02
RCGA5-0.3	1.5e+00	7.8e-01	1.4e-01	4.7e-02
RCGA5-0.5	1.2e+00	2.1e-02	2.0e-02	3.9e-03
RCGA6	3.8e+00	9.3e-01	3.3e-01	3.4e-02
RCGA7	1.3e+01	2.7e-01	3.9e-02	3.9e-02
RCGA8	1.1e+01	8.7e-01	2.3e-02	2.0e-02
RCGA9	2.8e+00	7.4e-02	2.5e-02	2.5e-02
RCGA10	1.7e+00	4.8e-02	2.2e-02	2.2e-02
RCGA11	2.0e+00	5.3e-02	1.4e-02	1.4e-02
RCGA12	5.3e+01	1.2e+00	3.3e-01	2.1e-02

Figura 2.11 - Média da melhor solução para f3, obtida em 100, 500, 1000 e 5000 gerações.

O RCGA-5-0.5, conforme consta na Figura 2.11, está equipado como a mutação *não-uniforme* (com grau de *não-uniformidade* b=5) e cruzamento *BLX-0.5*. Em geral, considerando também o desempenho com as outras funções, *f1* e *f2*, esta versão se saiu bem, principalmente em relação ao resultado final. O BLX-α com α´s pequenos (0.0, 0.15 e 0.3) não tiveram o mesmo bom desempenho. Os cruzamentos de *linha estendida* (*extended line* e *extended intermediate*) são equivalentes ao BLX-0.25 e também não foram melhores que o BLX-0.5 [18]. As conclusões a que chegou Herrera, com relação aos operadores discutidos neste trabalho, citam, em geral, o problema da convergência prematura :

- A mutação de *Mühlenbein* (RCGA7, RCGA8 e RCGA9) apresentou bons resultados até 1000 gerações, quando ocorria inevitavelmente uma convergência prematura [5];
- Os cruzamentos *discreto e simples* (RCGA2 e RCGA7) também conseguiram bons resultados no início, mas perderam rendimento nas últimas gerações;
- O cruzamento *heurístico* (*Wright´s crossover*) não obteve bons resultados, segundo Herrera, talvez em função da super exploração de espaços promissores com conseqüente convergência prematura.

CAPÍTULO III

3. Otimização de Funções sem Restrição

Problemas de otimização de funções podem estar sujeitos ou não a restrições na forma $c(\vec{x})$. Nos chamados problemas irrestritos, costuma-se delimitar o espaço de busca Ò como:

$$Ò \subseteq \Re^n \text{ , onde } Ò = \prod <L_i,U_i>, \ i = 1,..., n \qquad (3.1)$$

onde os pares L_i e U_i são constantes em $\Re$ e delimitam o espaço convexo Ò.

Os algoritmos baseados em população de soluções podem convergir para pontos isolados do espaço de busca que representem mínimos globais de funções multimodais. Entretanto, a política de renovação da população tende a excluir indivíduos com baixa aptidão, perdendo-se informação genética importante para a busca de outros mínimos globais.

A maioria dos algoritmos evolutivos para otimização de funções multimodais não são projetados para encontrar múltiplas soluções ótimas, convergindo para uma delas somente. De um jeito ou de outro, heurísticas de busca local são convidadas a participar do processo, com comprovado ganho de desempenho.

Neste capítulo, são apresentados alguns métodos de busca global para funções multivariáveis sem restrições, que envolvam algoritmos evolutivos. Foram escolhidas, para ilustrar algoritmos evolutivos voltados para problemas de otimização numérica sem restrições, quatro abordagens que representam bem as tendências atuais de pesquisa nessa área. Os resultados obtidos e a descrição dos experimentos desenvolvidos por eles são objetos de discussão no capítulo 5.

3.1 Algoritmos Evolutivos Híbridos

Uma abordagem que tem sido bastante explorada é a que agrega algoritmos evolutivos com método de busca local. A motivação por trás dessas abordagens está em dosar a robustez dos algoritmos evolutivos com a rapidez e precisão dos métodos de otimização local.

Em geral, um operador evolutivo unário, tal como a mutação, é usado para encapsular um método de busca local. Um exemplo é o Algoritmo Genético Simplex Híbrido (*Simplex Genetic Algorithm Hybrid SGAH* que utiliza uma arquitetura híbrida baseada em *elitismo*, onde o operador de busca local é aplicado somente aos melhores indivíduos da população [20,21].

A busca local usada no SGHA é uma variação probabilística do *Simplex* de *Nelder & Mead* (*Downhill simplex*)[22], onde os passos de reflexão e contração, respectivamente, γ e $\beta \in (0,1)$, com distribuição triangular de pico em 0.5. A Figura 3.1 ilustra um movimento básico de *reflexão* do simplex na direção oposta ao pior ponto. O procedimento também admite outros dois movimentos, expansão (se o ponto refletido for promissor) e contração (se o ponto refletido não for promissor).

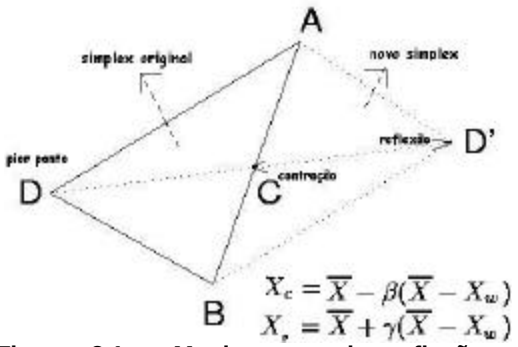


Figura 3.1 - Movimentos de reflexão e contração do Simplex de Nelder e Mead

No SGAH, toda a população é submetida aos operadores evolutivos tradicionais, mas apenas uma parcela S dela é submetida a operador de busca local *simplex probabilístico*. A nova população é composta também pelos N melhores indivíduos. A arquitetura elitista do SGHA é mostrada na Figura 3.2.

Algoritmos evolutivos com busca local podem ter problemas com espaços de busca descontínuos se a busca opera com derivadas. Além de sua simplicidade o *Simplex de Nelder & Mead* é bastante utilizado também por trabalhar apenas com avaliações de função objetivo.

Um outro método de busca local igualmente simples e bem mais eficiente, o gradiente conjugado tem a desvantagem de trabalhar com derivadas da função objetivo, sendo, portanto, vulnerável a possíveis descontinuidades do espaço de busca.

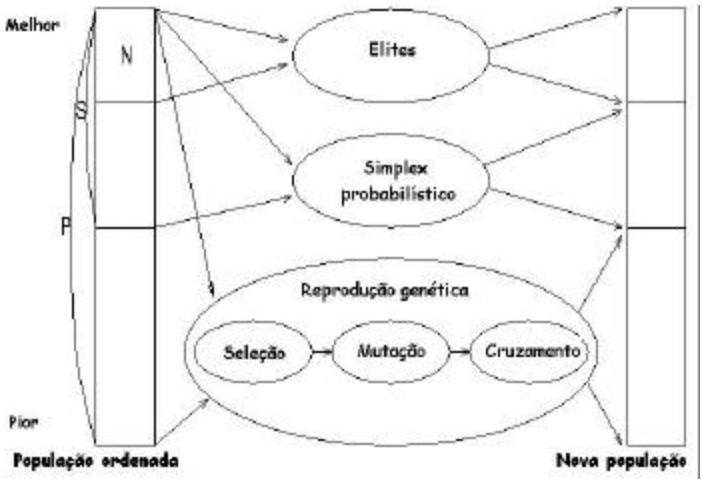


Figura 3.2 - Arquitetura elitista do SGAH: busca local apenas nos S melhores indivíduos

Birru, H. K. et al. propôs um algoritmo baseado em programação evolutiva chamado de Programação Evolutiva Rápida (F*ast Evolutionary Programming - FEP*), que incorpora uma mutação com distribuição de *Cauchy* (ao invés de mutação Gaussiana) e o método de *gradiente conjugado* para busca local [23].

O FEP é composto basicamente dos seguintes passos:

1. Inicialização: uma população inicial aleatória de indivíduos, representados como um vetor de números reais (x_i,σ_i), i = 1, 2, ...,m; onde x_i representa as variáveis de controle e l_i representa os desvios padrão associados aos parâmetros de estratégia de cada uma delas.

2. Mutação: cada pai (x_i, σ_i) produz um único descendente intermediário (x'_i, σ'_i) usando:

$$s'_i(j) = s_i(j)\exp\left(t'N(0,1) + tN_j(0,1)\right) \qquad (3.2)$$

$$x'_i(j) = x_i(j) + \sigma'_i(j)\cdot\delta_j(0,1) \qquad (3.3)$$

onde N(0,1) é um número aleatório normalmente distribuído e $\delta_j(0,1)$ é um número aleatório com distribuição de Cauchy .

3. Busca local: o operador de busca local é aplicado ao componente x'_i de cada descendente intermediário (x'_i ,σ'_i), com certa probabilidade, para gerar um descendente final (x''_i, σ''_i).

4. Torneio: cada indivíduo na população é comparado com q oponentes aleatoriamente selecionados, contabilizando-se o número de vitórias.

5. Seleção: a metade dos indivíduos, com maior número de vitórias, são selecionados para a próxima geração.

O incremento no desempenho dos algoritmos híbridos, nem sempre justifica o aumento de complexidade causado pelo método de busca local. Quando computados a quantidade de avaliações da função objetivo, esse número, dependendo do método local, chega a ser dez vezes maior que com o mesmo algoritmo sem a busca local. Birru concluiu que, com dez vezes mais gerações, o algoritmo não-híbrido consegue encontrar soluções tão boas quanto o algoritmo híbrido [23].

3.2 Algoritmos Evolutivos com Controle Adaptativo de Parâmetros

Seja um sistema evolutivo formado por uma população de indivíduos, um conjunto de parâmetros associados aos operadores evolutivos e um algoritmo que define as regras de evolução, pode-se definir controle de parâmetros como ajustes necessários para que tais parâmetros induzam os operadores a um comportamento mais coerente com o estado do sistema evolutivo em um dado instante. O controle de parâmetros pode ser classificado como: controle determinístico, controle adaptativo, e controle auto-adaptativo [24].

O controle determinístico altera o parâmetro evolutivo através de regras determinísticas, baseando-se, por exemplo, no tempo (ou geração), sem considerar nenhuma informação do sistema evolutivo em si. A mutação *não-uniforme* [7] é um bom exemplo de operador controlado dessa forma.

No controle auto-adaptativo, os parâmetros a serem controlados são codificados nos próprios indivíduos (ou cromossomos), e são ajustados de acordo com o estado específico de cada um deles. O FEP, apresentado anteriormente, é um bom exemplo deste tipo de controle.

O controle adaptativo, por sua vez, altera o parâmetro evolutivo considerando informações capturadas do próprio sistema evolutivo, como taxa de convergência, tamanho da população e grau de diversidade [1].

Herrera e Lozano propuseram o TRAMSS (*Two-Loop Real-Coded Genetic Algorithms with Adaptive Control of Mutation Step Sizes*) que implementa o controle adaptativo baseando-se na média da população para detectar se há ou não progresso na evolução (bom desempenho) [24]. A Figura 3.3, mostra basicamente o algoritmo usado por Herrera.

```
TRAMSS
    Inicialização {população P. passo de mutação S e reinicialização R;
    Enquanto (Não CondiçãoDeTérmino)
        Enquanto (S ³ S_min)
            Enquanto (G gerações)
                Seleção, Cruzamento e Mutação(S);
                Avaliação;
            Fim;
            Se a média de P melhorou
                Sucessos := Sucesso +1;
                Falhas := 0;
                S := S . 2^Sucessos;
            Se não
                Sucessos := 0;
                Falhas := Falha + 1;
                S := S / 2^Falhas;
            Fim;
            G := Proporcional a (S);
        Fim;
        Se encontrou um indivíduo melhor  R:=R/2;
        Se não                            R=R.2;
        Reinicia P com Mutação(R);
    Fim;
Fim;
```

Figura 3.3 - Controle adaptativo usado no TRAMSS

A estratégia no laço mais interno é "se houve progresso nas gerações anteriores, aumente o passo de mutação". Assim a quantidade de gerações G, bem como o passo de mutação S em cada uma dessas gerações é incrementado quando há melhoramento na média da população. Entretanto, quando é detectado uma estagnação do sistema (mau desempenho), a população é reinicializada através de uma mutação geral, também controlada por um parâmetro R. A estratégia de controle, neste caso, é oposta à anterior: "se houve progresso nas gerações anteriores, diminua o passo de mutação". Dessa forma, o processo é recomeçado aproveitando-se mais ou menos informação da população anterior, dependendo se houve mais ou menos progresso.

Em geral, implementações de controle de parâmetros com baixa complexidade são um bom mecanismo para incremento de desempenho.

3.3 Busca baseada em Tabu

Um outro enfoque evolutivo, bem recente, que tem mostrado bons resultados para otimização de funções multimodais é o *Scatter Search* proposto por Glover [25,26] e é descendente direto das estratégias adaptativas de *Busca Tabu* também proposta por ele para programação inteira[27].

O *Scatter Search* (SS) possui mecanismo explícito de manutenção de diversidade e, ao mesmo tempo, permite o emprego de buscas locais agressivas, trabalhando com populações pequenas de indivíduos.

Inicialmente, é gerado um conjunto de vetores solução através de um sistema que garanta um espalhamento mais ou menos uniforme no espaço de busca. Depois, ao longo das gerações, essas soluções são melhoradas através de alguma heurística específica e recombinadas linearmente para produzir pontos tanto dentro quanto fora das regiões limitadas por estas soluções. A Figura 3.4 mostra o processo de recombinação linear.

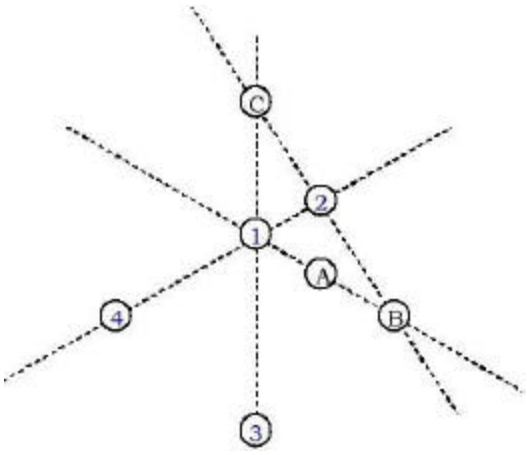


Figura 3.4 - Esquema de recombinação linear de soluções usado pelo SS

Os pontos A, B e C formam o conjunto inicial de soluções. O ponto 1 é obtido linearmente a partir de A e B. O ponto 2, por sua fez é obtido a partir de B e C. O ponto 3 é gerado a partir da combinação linear de C e 1 e assim sucessivamente.

O Scatter Search pode ser detalhado através de 5 métodos:

a) Geração de Diversificação: gerar uma coleção de soluções dentro do espaço de busca, seguindo uma distribuição uniforme entre os limites L_i e U_i de cada variável de controle x_i. São definidos contadores de freqüência para evitar que certas regiões do espaço de busca se torne mais populosas que outras.

b) Melhoramento: em geral heurísticas de busca local "leves" podem ser usadas com o propósito de melhorar cada solução. Para problemas de otimização numérica tem sido utilizado o método *Simplex de Nelder e Mead* [22].

c) Atualização dos Conjuntos de Referência: criar e manter dois subconjuntos pequenos de indivíduos (20% da população, cada), chamados de conjuntos de referência *RefSet1* e *RefSet2*. No primeiro são incluídos as soluções com melhor *f(x)* e no outro são incluídos os que possuam a maior distância euclidiana mínima em relação às soluções de *RefSet1*. Isto indicaria tratar-se de uma solução com boa diversidade *d(x)*, mas não significa necessariamente indivíduos ruins em termos de função objetivo.

d) Geração de Subconjuntos: gerar a partir de *RefSet1* e *RefSet2* os subconjuntos que serão combinados no Método de Combinação, a seguir. É sugerido gerar todos os subconjuntos de soluções tomados 2 a 2, depois 3 a 3, e assim por diante até que se tenha um único subconjunto com todas as soluções dos conjuntos de referência.

e) Combinação de Soluções: transformar um dado subconjunto produzido no método anterior em uma ou mais soluções combinadas. O método de combinação é similar a um cruzamento aritmético e é aplicado várias vezes gerando vários descendentes por cruzamento. Em geral, são sugeridos basicamente 3 modos de combinação *C1, C2 e C3* e regras para aplicação de desses modos dependendo e para atualização dos conjuntos de referência [26].

C_1: x = x' - r (x''-x')/2

C_2: x = x' + r (x''-x')/2

C_3: x = x'' + r (x''-x')/2, onde r é um número aleatório no intervalo (0,1).

R_1: Se x' e x''$\in$ RefSet1 aplicar uma C_1, C_3 e duas C_2, gerando 4 soluções;

R_2: Se x' ou x''$\in$ RefSet1 aplicar uma vez apenas C_1, C_2, C_3, gerando 3 soluções;

R_3: Se x' e x''$\notin$ RefSet1 aplicar uma vez C_2 e C_1 ou C_3, gerando 2 soluções.

As novas soluções geradas, após melhoramento, são atualizadas em RefSet1 ou RefSet2 dependendo de duas regras:

U_1: Em RefSet1, se f(x) é melhor que a pior solução em RefSet1;
U_2: Em RefSet2, se d(x) é melhor que a pior solução em RefSet2;

Percebe-se que o SS possui uma sistemática de exploração do espaço de busca combinado com a intensificação por busca local e mantém, não só os melhores indivíduos, mas também aqueles mais afastados das regiões mais promissora, evitando a convergência para um único ponto ótimo. Detalhes sobre os experimentos com o SS são apresentados no Capítulo 5.

3.4 Métodos de Nicho

Os chamados métodos de nichos (*Niching metholds*) [28,29,30] estendem os algoritmos evolutivos para domínios que requerem a localização e manutenção de soluções múltiplas. Tais domínios incluem aprendizado de máquina, classificação, otimização de funções multimodais e multiobjetivo[5].

Em analogia com natureza, um ecossistema possui diferentes subsistemas (nichos) que contém muitas espécies (subpopulações). Tais nichos podem evoluir em paralelo, convergindo para diferentes pontos ótimos no espaço de busca. Os critérios que definem se um indivíduo pertence a um ou de outro nicho é parte da estratégia específica de cada algoritmo.

Soluções diferentes com mesmo valor de função objetivo podem significar o mesmo custo/ganho. Entretanto, abrem a possibilidade de escolha entre várias alternativas. A manutenção da diversidade populacional é imprescindível para que se encontre mais de uma solução global. Existe uma tendência de indivíduos de um nicho dominarem as recombinações e os demais indivíduos de outros nichos serem descartados ao longo das gerações.

Uma forma simples de ganhar diversidade e evitar a convergência da população para uma única solução ótima, é minimizar substituições errôneas de soluções únicas e promissoras. Para alcançar esse objetivo, Grüninger & Wallace propuseram o *Struggle GA* que utiliza um conceito de similaridade entre indivíduos e sugere algumas formas para calculá-la.

O pseudo código a seguir ilustra o processo de reprodução dos indivíduos. No caso de codificação real, é sugerido a distância euclidiana para o cálculo da similaridade [30].

```
Repita até condição de parada
        Selecione pais P1 e P2 uniform emente aleatório;
        Recombine P1 e P2, gerando um descendente C;
        Aplique mutação gerando C';
        Encontre na população o indivíduo R mais similar a C' ;
        Se f(C') melhor f(R)  substitua R com C';
Fim;
```

Figura 3.5 - Reprodução em métodos de nicho

O conceito de nicho, no *Strugg le GA* está implícito. Foi verificado que formam-se nichos em torno dos indivíduos mais aptos da população inicial P^0. Assim, o espalhamento inicial da população no espaço de busca determina o número de regiões a serem exploradas e consequentemente o número de nichos. Eventualmente alguns nichos são englobados por outros, quando a similaridade entre seus indivíduos aumenta. O processo de mutação, por outro lado, pode criar novos nichos ao longo das gerações [30].

É comum aos métodos de nicho o conceito de recursos, através do qual se estabelece limites para a quantidade de indivíduos por nicho. Dessa forma, cada nicho pode suportar um número de indivíduos diretamente proporcional a sua "fertilidade" que é medida pela aptidão total deste pico em relação aos outros picos do domínio. Outro conceito, a aptidão compartilhada (*sharing fitness*), foi introduzida por Goldberg e Richardson, consistindo na redução da aptidão de um indivíduo proporcional ao número de indivíduos próximos (mesmo nicho), afetando o mecanismo de seleção desse método. O f_i de aptidão compartilhada do *i-ésimo* o indivíduo é determinado por [31]:

$$f'_i = \frac{f_i}{m_i} \qquad (3.4)$$

onde f_i é a aptidão original (função objetivo, por exemplo) e m_i é contabiliza o número de indivíduos no mesmo nicho ($d_j < \delta$):

$$\sum_{j=1}^{N} \begin{cases} 1-(\frac{d_{ij}}{d})^{a} & se \quad d < d \\ 0 & cc \end{cases} \qquad (3.5)$$

CAPÍTULO IV

4. Otimização sujeita a condições de restrição

A programação não linear trabalha com formulações do tipo:

Otimize $f(x)$, $x=(x_1, x_2, x_3,., x_n)^T \in \Re^n$
Sujeito a $p \geq 0$ equações:

$c_i(x) = 0$, $i=0,..,p$

E $m-p \geq 0$ inequações:

$c_i(x) \leq 0$, $i=p+1, ..., m$

Tem-se pesquisado muito por métodos eficientes para determinar o ótimo global para um problema geral de programação não linear (NLP). Alguns métodos, que foram desenvolvidos inicialmente para problemas sem restrição (p e m =0), servem de base também para as técnicas para problemas com restrição.

Há dois enfoques não evolutivos básicos para operar com as restrições: o direto e o indireto. Os métodos indiretos tentam extrair um ou mais problemas não lineares do problema original, enquanto o outro tenta resolver diretamente o problema original. Isto, em geral, é conseguido transformando o problema original em um problema sem restrições e , a partir daí, são aplicados os métodos de busca local conhecidos (Gradiente Conjugado, Newton Truncado, etc) [32].

Dentre as maiores dificuldades para a eficiência desses métodos é que eles são locais, dependentes de derivadas, pouco robustos em virtude de espaços de busca descontínuos, multimodais e sujeito a ruídos.

Como foi discutido anteriormente os algoritmos evolutivos possuem várias características desejáveis para otimizar funções complexas. Dentre eles ressalta-se:

- podem operar com restrições arbitrárias em espaços de busca complexos;
- podem explorar simultaneamente várias regiões do espaço de busca;
- não necessitam de conhecimento matemático do problema (tais como derivadas);
- adaptáveis a várias classes de problemas.

O principal problema para construção de algoritmos evolutivos eficientes para problemas de otimização com restrições reside na forma de se tratar (manter e avaliar) as soluções inviáveis. A seguir são apresentados as principais abordagens quem têm sido discutidas recentemente.

4.1 Abordagens para manipulação de soluções inviáveis

Quando se trabalha apenas com restrições de intervalo constantes, tem-se efetivamente um espaço de busca convexo. Entretanto, restrições não-lineares podem formar espaços de busca formado por dois subespaços disjuntos: viável V e inviável I.

Dessa forma $\Omega = V \cup I$, onde $V \cap I = \phi$. Além disso V e I podem ser não convexos e não conectados, como mostra a Figura 4.1:

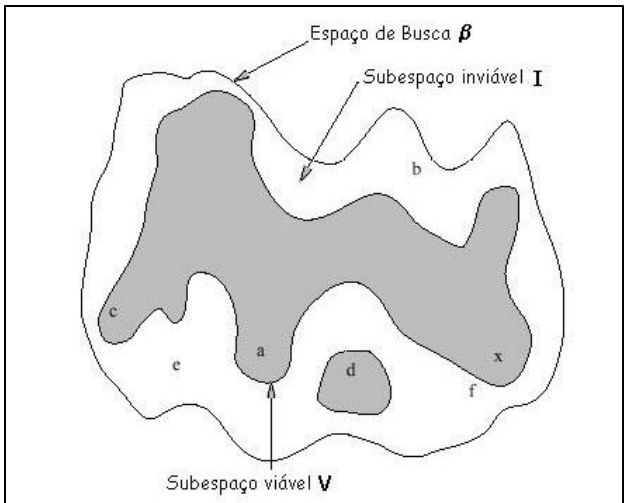


Figura 4.1 - Espaços de busca viável e inviável

Há duas abordagens básicas para manipulação de soluções inviáveis: (a) modificar o conjunto de operadores evolutivos para não haver violação de restrição; (b) penalizar soluções que violem alguma restrição.

Penalizar restrições não satisfeitas reduz a aptidão do indivíduo e sua probabilidade de participar do processo de evolução. É questionável se a penalidade deve impor que todo indivíduo inviável seja pior que qualquer indivíduo viável. Por exemplo, na Figura 4.1, suponha que o ponto *x* seja o ótimo. O ponto *f* é o mais próximo do ponto *x*, mas é inviável e, por uma política de penalizar em demasia os pontos inviáveis, tem baixa probabilidade de ser selecionado para recombinações e mutações.

Existe uma outra abordagem, também encontrada na literatura, que se enquadra nesta: o descarte de soluções inviáveis. Isto eqüivale a impor uma penalidade extrema ao indivíduo inviável, retirando-o completamente do processo de evolução.

Por outro lado, manter as soluções sempre viáveis pode ser obtido pela especialização dos operadores evolutivos para preservação de viabilidade, como também, pela correção das soluções inviáveis, que seria uma outra sub-abordagem dentro desta.

Há vantagens e desvantagens associadas a cada um desses grupos. Alguns métodos híbridos aplicam mais que um desses enfoques para equilibrar seus pontos fracos/fortes. A seguir são discutidos com mais detalhes as diferentes estratégias dentro de cada uma dessas abordagens.

4.1.1 Descarte de soluções de inviáveis

Descartar soluções inviáveis pode ser considerado como impor uma "penalidade da morte". Trata-se de um método popular usado por muitas técnicas como estratégias de evolução [9] e tende a funcionar razoavelmente bem, quando o espaço viável de busca V é convexo e se constitui em uma razoável parte do espaço de busca total Ò.

Em espaços de viabilidade não convexos, restringir o acesso a regiões inviáveis que poderiam funcionar como "corredores" para regiões viáveis, não produz bons resultados. Da mesma forma, é mais eficiente melhorar um conjunto de soluções inviáveis que rejeitá-las, quando a razão |V|/ |Ò| é pequena [33].

Sabe-se que os algoritmos evolutivos otimizam por combinar informação parcial de toda a população. Dessa forma uma solução inviável também pode prover informação essencial para o processo de evolução e não deve ser simplesmente descartada [34] .

4.1.2 Reparação de soluções inviáveis

Esta estratégia depende da existênc ia de um rápido e determinístico procedimento para conversão de uma solução inviável em uma viável. Em geral, esses procedimentos são bem específicos para cada tipo de restrição violada e demandam um razoável esforço computacional. Alguns métodos utilizam um conjunto viável de soluções para reparar outras através de combinação linear [7].

Uma vez resolvido esse problema inicial, tem-se ainda duas possibilidades: substituir definitivamente a solução inviável pela solução reparada ou utilizar a solução reparada apenas para avaliação da função objetivo, mas não substituir a original inviável. Essas duas estratégias tendem a focalizar a busca mais no interior ou nas bordas das regiões dos espaços viáveis de busca. Por isso, pode-se estipular uma probabilidade de substituição das soluções originais pelas reparadas (em torno de 5%) [35].

4.1.3 Penalidade para restrições não satisfeitas

Nesta categoria são usadas funções de penalidade para transformar um problema de otimização restrito em um irrestrito. Muitos algoritmos que utilizam esta abordagem, diferem em detalhes importantes de como as penalidades são definidas e aplicadas às soluções inviáveis.

Cada restrição entra na função objetivo associada a um peso. Quanto mais restrições violadas, pior será o resultado da avaliação do indivíduo. Os pesos, nesse caso, servem para ponderar as penalidades, inclusive, definir as mais importantes para o processo de otimização. Esses pesos devem ser corretamente sintonizados para dar um comportamento suave a cada uma das componentes (restrições) da função objetivo modificada.

Neste caso também existem duas estratégias associadas: penalidades uniformes ou variáveis. Uniforme significa pesos mantidos fixos ao longo do processo e variáveis, por sua vez, significa negligenciar os pesos no início e aumentar a pressão destes gradualmente no decorrer do processo. A justificativa deste último reside em permitir o espalhamento de indivíduos pelo espaço de busca como um todo no início do processo e, a medida que a população evolui, ir penalizando progressivamente os indivíduos inviáveis. Entretanto, essa estratégia pode afastar muito os indivíduos das regiões de viabilidade de forma irreversível, pois não há garantias que o aumento da penalidade force os indivíduos a retornarem às regiões viáveis que estejam distantes.

O método de Homaifar et al trabalha com uma matriz R_{ij} de violações i por restrições j, onde para cada restrição, tem-se vários níveis de violação [36].

$$aval(x) = f(x) + \sum_{j=1}^{m} R_{ij}\, p_j^{\,2}(x) \qquad (4.1)$$

onde *f(x)* é a função objetivo $p_j(x)$ é um conjunto de j funções associadas a cada uma das restrições. A matriz penalidade é fixa e se constitui em um número significativo parâmetros a serem ajustados para tornar o método eficiente [33].

Joines & Houck propuseram penalidade variável, reduzindo também a quantidade de parâmetros a serem ajustados, propondo[37]:

$$eval(x) = f(x) + (C.t)^{\alpha} \sum_{j=1}^{m} p_j^{\beta}(x) \qquad (4.2)$$

Onde C, α e β recebem valores pequenos e são fundamentais na eficácia do método. Por exemplo, C=0.5, $\alpha=\beta=2$ resultaram em soluções distante do ótimo global ou muitas vezes inviáveis [33].

Outra forma de impor penalidades é criar um ranking onde as avaliações de soluções viáveis sejam sempre mapeadas para valores melhores que as soluções inviáveis. Ou seja, a filosofia é o pior solução viável é melhor que a melhor solução inviável ($x \notin V$). Basicamente:

$$aval(x) = \begin{cases} f(x),\ x \in V \\ f(x) + r.p_j(x,t),\ x \in b - V \end{cases} \qquad (4.3)$$

Onde r é uma constante.

Para o caso de uma minimização, [38] propuseram uma penalidade em função da melhor solução inviável e da pior solução viável que variam dinamicamente:

$$p(x,t) = \max\{0, \max\{aval_{x \in V}(x)\} - \min_{x \notin V}\{aval(x)\}\} \qquad (4.4)$$

4.1.5 Memória de Comportamento (*Behavioral Memory*)

Schoenauer & Xanthakis propuseram a memória de comportamentos que difere um pouco das outras abordagens. A principal motivação para esta nova proposta é que a imposição de penalidades não é eficaz para muitos tipos de problemas. O fundamento principal desta técnica é o de tratar as restrições em uma ordem particular [39].

Basicamente, há dois passos:

(a) Evoluir uma população aleatória inicial (através de algum algoritmo evolutivo padrão), usando uma avaliação de aptidão que relacione a função objetivo com a primeira restrição, até que um certo percentual da população (*flip threshold*) seja viável para esta restrição. Repetir esse processo para cada uma das restrições restantes, sempre usando a população final da evolução anterior como ponto de partida e eliminando os indivíduos que violem a restrição anterior.

(b) Evoluir a partir da população final, eliminando os indivíduos inviáveis e usando desta vez, como aptidão, somente a função objetivo.

A ordem na qual as restrições são submetidas influencia os resultados, sendo portanto importante se estabelecer adequadamente tal ordem [33]. Em particular, este enfoque requer, além da ordem das restrições, mais dois parâmetros: o percentual de indivíduos viáveis para cada restrição e o fator de compartilhamento de aptidão (veja seção 3.4).

4.2 GENOCOP

Genocop (*GEnetic algorithm for Numerical Optimization for COnstraits Problems*) é um sistema baseado em algoritmo genético para otimização de função não-linear com ou sem restrições desenvolvido por Michalewicz[8].

Existem três versões disponíveis que possuem filosofias de trabalho diferentes. O GENOCOP (posteriormente re-batizado de GENOCOP I) pode minimizar ou maximizar funções não lineares com restrições de igualdade ou desigualdade lineares.

O GENOCOP II é uma versão híbrida desenvolvida para trabalhar também restrições não-lineares, utilizando o GENOCOP I sobre a função objetivo modificada pela inclusão das restrições não-lineares sujeitas a uma penalidade dinâmica.

O GENOCOP III é a última e, segundo Michalewicz, "a mais promissora" versão para operar com restrições não-lineares. Também incorpora o GENOCOP I, mas acrescenta duas populações separadas, onde o desenvolvimento dos indivíduos de uma afeta a avaliação dos indivíduos da outra. A primeira população P_s consiste dos chamados pontos de busca ~~S~~ que satisfazem, a princípio, as restrições lineares (obtido via GENOCOP I). A segunda população P_r consiste dos chamados pontos de referência ~~R~~ que satisfazem todas as restrições (inclusive as não-lineares). Os pontos de ~~R~~ são avaliados diretamente pela

função objetivo, mas os pontos de ~~S~~ são reparados para efeito de avaliação da seguinte forma:

```
Para cada ~~S~~ ∈ P_s a ser reparado
        ~~R~~ = Seleciona por Ranking (P_r);
        Faça
                a = aleatório no intervalo (0,1);
                Z = a S + ( 1 - a ) R ;
        Até ~~Z~~ ser viável
        Se ~~Z~~ é viável então
                Aval (~~S~~) = f (~~Z~~);
                Se f (~~Z~~) é melhor que Aval(~~R~~) então
                        ~~R~~ ← ~~Z~~;
                Fim
        Fim
Fim
```

Figura 4.2 - Algoritmo de reparação do GENOCOP III

Em outras palavras, o GENOCOP III cria linhas de busca entre regiões viáveis e inviáveis, com intensidade de busca maior na vizinhança dos melhores pontos de referência (garantido pela seleção por ranking). Em um segundo estágio alguns pontos de referência são movidos para P_s e o processo se repete. Vários resultados produzidos pelo GENOCOP III já foram publicados [8].

Como já foi dito anteriormente O GENOCOP II também utiliza o GENOCOP num processo híbrido que agrega inclusão de penalidades à função objetivo que são ajustadas através de funções de temperatura (similares ao *Simulated Annealing*). Apesar do GENOCOP I trabalhar efetivamente como um algoritmo evolutivo, com população de soluções, do ponto de vista do GENOCOP II , existe somente uma solução x^* sendo otimizada a cada iteração. Inicialmente, x^* é gerada aleatoriamente e pode ser uma solução inviável A cada iteração, x^* é atualizada com a melhor solução encontrada pelo GENOCOP I que otimiza a função objetivo modificada F(x) e sujeito somente às restrições lineares L. O método para avaliação de soluções inviáveis foi proposto por Michalewicz e Attia. O algoritmo a seguir ilustra o GENOCOP II [8]:

```
Procedimento GENOCOP II
    t ← 0;
    Divida o conjunto de restrições C em 03 subconjuntos
        C → L ∪ Le ∪ Li    (lineares, igualdades não lineares e desigualdades não lineares);
    Seleciona um ponto x_s;
    Gera o conjunto de restrições ativas A
        A ← Le ∪ V         (onde V ⊆ Li , é conjunto de restrições de L_i violadas por x_s);
    τ ← τ_0                (penalidade inicial);
    Enquanto (não fim)
        t ← t+1;
        Execute GENOCOP I para
            F(x, τ) = f(x) + 1/(2τ) A^T A
            Sujeito a L    (restrições lineares)
                           (Incluindo na população inicial a solução x_s);
        Salve a melhor x* encontrada;
        x_s ← x*;
        Atualize A
            A ← A - S ∪ V;     (onde S é o subconjunto de L_i satisfeitas por x*);
        τ ← g(τ,t)             (em geral τ decai geometricamente a razão de 10^-1);
    Fim
Fim
```

Figura 4.3 - Algoritmo básico do GENOCOP II

A implementação do GENOCOP II mostrou-se lenta e de pouca eficácia, o que impulsionou novas pesquisas em direção ao GENOCOP III.

O GENOCOP I é serve de base aos dois sistemas anteriores e tem servido como modelo-teste para pesquisadores no mundo inteiro. Devido a natureza dos operadores evolutivos que utiliza, ele requer o domínio $Ò \subseteq \Re^n$ e $Ò$ forme um conjunto convexo [8]. Isto garante que se dois pontos de Ò, x^1 e x^2, a combinação linear $ax^2 + (1-a)x^1$, com $a \in (0,1)$ também é um ponto de Ò. E garante também que para cada ponto $(x_1, x_2, x_3,., x_n) \in Ò$ existe um intervalo viável $<l_k,u_k>$ de uma variável x_k $(1 \le k \le n)$ onde as outras variáveis xi (i=1,...,k-1,k+1,...n) permanecem fixas. Em outras palavras, tem-se uma faixa de possíveis valores que cada variável pode assumir com as demais variáveis permanecendo fixas. Por exemplo:

$<L_1,U_1>$: $-3 \le x_1 \le 3$,

$<L_2,U_2>$: $0 \le x_2 \le 8$, e

C: $x_1^2 \le x_2 \le x_1 + 4$,

Fixando $x_1 = 2$, por exemplo, tem-se

$x_2 \ge (2)^2 \therefore x_2 \ge 4$ e $x_2 \ge 6$.

Assim <4,6> são os novos $<l_2,u_2>$ para x_2 quando x_1 for 2. Os problemas com domínio convexo podem ser formulados como se segue:

$l \leq x \leq u$, onde $l=< l_1, x_2,..., l_n>$, $u=< u_1, u_2,..., u_n>$

O total de restrições (*p* igualdades, *q* desigualdades) é m.

Igualdades:

$Ax = b$, onde $A = (a_{ij})$, $b=< b_1, b_2,..., b_p>$,

para $1 \leq i \leq p$ (número de igualdades)

e $1 \leq j \leq n$ (número de variáveis)

Desigualdades:

$Cx \leq d$, onde $C = (c_{ij})$, $d=< d_1, d_2,..., d_q>$,

para $1 \leq i \leq q$ (número de desigualdades)

e $1 \leq j \leq n$ (número de variáveis)

Em programação linear, as desigualdades são transformadas em igualdades. O primeiro passo do GENOCOP I é fazer o inverso. Ele remove as igualdades, eliminando a mesma quantidade de variáveis e, ao mesmo tempo, reduzindo o espaço de busca:

Sejam as igualdades $Ax = b$,

pode-se desmembrar A verticalmente em A_1 e A_2 de modo que

$A_1x_1 + A_2x_2 = b \therefore x_1 = (A_1)^{-1} b - (A_1)^{-1} A_2x_2$.

x_1 representa aqui variáveis que participam das igualdades (uma variável para cada equação de igualdade) e podem ser escritas em função de outras variáveis. A partir daí, tem-se que todas as desigualdades:

$Cx \leq d$ ou seja $C_1x_1 + C_2x_2 \leq d$, podem ser rescritas em função apenas de x_2:

$C_1((A_1)^{-1} b - (A_1)^{-1} A_2x_2) + C_2x_2 \leq d$. Assim como os limites de x_1:

$l_1 \leq x_1 \leq u_1 \therefore l_1 \leq (A_1)^{-1} b - (A_1)^{-1} A_2x_2 \leq u_1$ (4.5)

Como já existe um limite para x_2,

$l_2 \leq x_2 \leq u_2$ (4.6)

(4.5) e (4.6) devem ser condensados em um único limite para x_2.

Após a eliminação de igualdades, as restrições restantes, na forma de desigualdades lineares, formam um conjunto convexo, que garante que combinações lineares de soluções geram soluções sem a necessidade de verificar as restrições. As desigualdades também são usadas para gerar fronteiras dinâmicas de uma dada variável de forma eficiente.

Após esse passo inicial, o GENOCOP I tenta achar um conjunto inicial de soluções (população inicial), amostrando o espaço de busca convexo. Caso tenha dificuldade, ele solicita ao usuário pelo menos um ponto e replica esse(s) ponto(s), formando uma população inicial de n soluções iguais. O GENOCOP I provê diversos operadores evolutivos para promover a evolução da população e conseqüente achar um ponto ótimo. Os operadores usados são: mutação uniforme, mutação limitada, mutação não-uniforme, cruzamento aritmético, cruzamento heurístico e cruzamento comum.

CAPÍTULO V

5. Experimentos Computacionais

As diferentes abordagens apresentadas neste trabalho tiveram seus desempenhos avaliados através de experimentos usando funções-teste com características bem conhecidas e níveis de complexidade variáveis.

Em 1975, De Jong utilizou algoritmos genéticos para otimização de função e publicou seus experimentos em sua tese [4]. O ambiente de teste construído por ele é conhecido como as funções de De Jong. Trata-se de cinco funções com as mais variadas características: contínuas e descontínuas; convexas e não convexas; unimodais e multimodais; quadráticas e não quadráticas; alta e baixa dimensionalidades; determinísticas e estocásticas. Em duas dimensões, a função 1 e 2 são visualizadas nas Figuras 5.1 e 5.2, respectivamente.

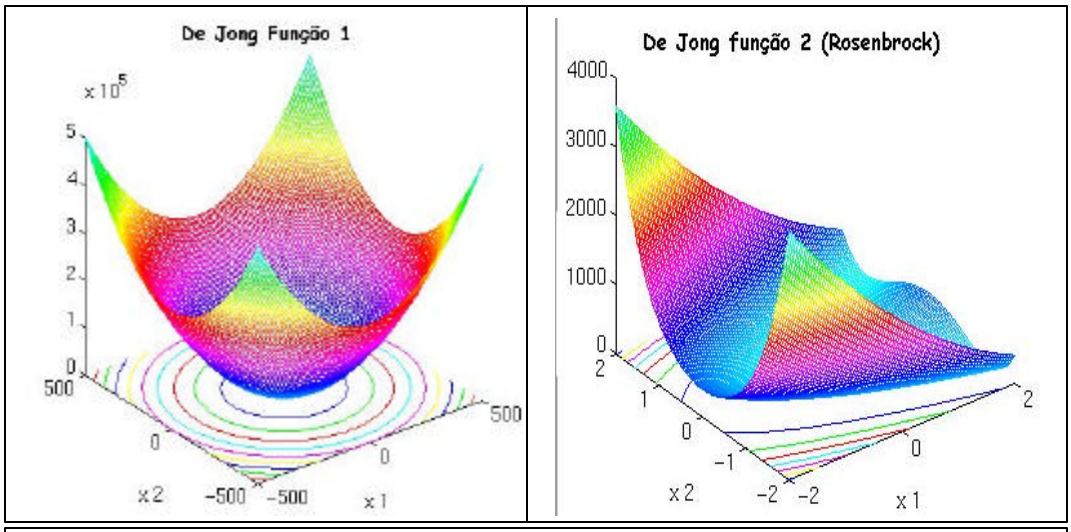


Figura 5.1 - Funções de De Jong: (a) Função 1 (esfera); (b) Função 2 (rosenbrock)

A função 1 (esfera), a mais simples das funções de De Jong, é contínua, convexa, unimodal, lisa, simétrica e possui mínimo f(0)=0. A função 2 (Vale de Rosenbrock) é um problema clássico de otimização numérica, cujo mínimo global encontra-se no interior de um vale parabólico longo e estreito. Essas duas funções são geradas pelas duas primeiras equações mostradas na Figura 2.9. Os desenhos das funções 4 e 5 são mostradas na Figura 5.2.

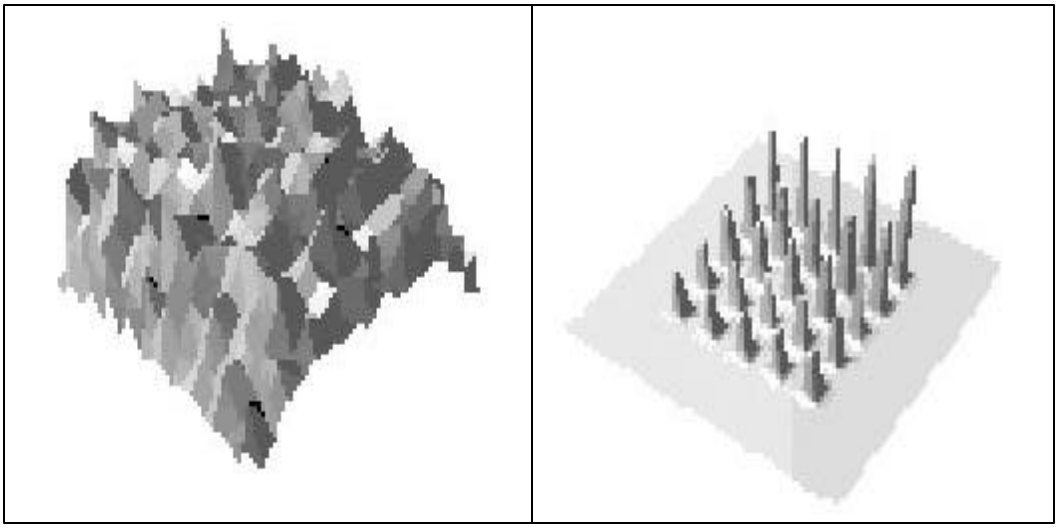


Figura 5.2 - Funções de De Jong: (a) Função 4 (quartzo) ; (b) Função 5 (buracos de raposa)

A função 4 (quartzo) é unimodal acrescida de um ruído *gaussiano* que garante que um ponto (indivíduo) nunca receberá a mesma avaliação duas vezes consecutivas. A função 5, por sua vez, apresenta múltiplos mínimos/máximos locais . Nno exemplo da figura são 24 ao todo.

Para aferir a eficácia de diferentes algoritmos genéticos, De Jong definiu duas medidas de desempenho: *on-line* e *off-line*.

O desempenho *on-line* $X_e(s)$ de uma estratégia s em um ambiente e é calculado por:

$$x_e(s) = \frac{1}{T}\sum_{1}^{T} f_e(t) \qquad (5.1)$$

onde $f_e(t)$ é a função objetivo na iteração t. Em outras palavras, $X_e(s)$ mede o desempenho em curso através da média de todas as avaliações da função objetivo até o momento.

Por outro lado, o desempenho *off-line* $X^*_e(s)$ é calculado por:

$$x^*_e(s) = \frac{1}{T}\sum_{1}^{T} f^*_e(t) \qquad (5.2)$$

onde $f^*_e(t)$ = *melhor* $\{f_e(1), f_e(2),\ldots, f_e(t)\}$. Ou seja, $X^*_e(s)$ mede a convergência do algoritmo, através da média das melhores avaliações da função objetivo a cada iteração *t*. Portanto, $X^*_e(s) \geq X_e(s)$, para problemas de maximização.

Os desempenhos *on-line* e *off-line* podem ser analisados conjuntamente, possibilitando outras interpretações. Por exemplo, quando $X^*_e(s)$ e $X_e(s)$ estão com valores muito próximos, pode-se presumir que há uma perda de diversidade populacional.

As funções de De Jong, bem como suas medidas de desempenho têm sido bastante usadas como ambientes de teste de desempenho para algoritmos evolutivos [1,24,40].

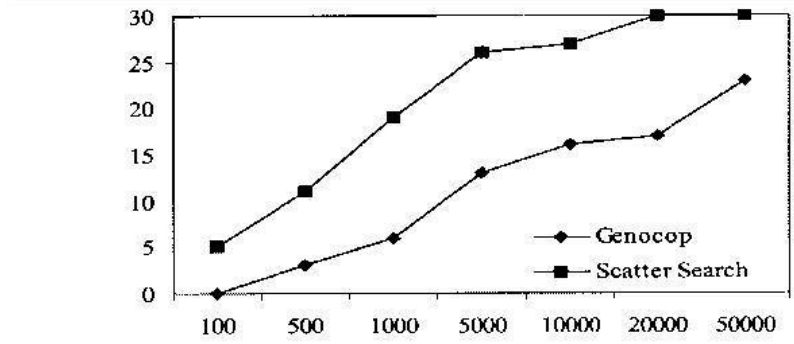
De modo geral, experimentos com algoritmos estocásticos devem ser repetidos várias vezes com diferentes sementes aleatórias, população inicial e calculando-se a média e o desvio padrão das soluções encontradas para cada problema. O número de avaliações da função objetivo também é considerado para a avaliação de desempenho, pois a grande maioria das funções objetivos são multivariáveis e podem ser altamente complexas e lentas para terem seus valores determinados em cada ponto.

O comportamento do algoritmo também pode ser observado em *checkpoints* (pontos de verificação). Dessa forma, a cada *n* gerações (iterações) a melhor solução é guardada. Depois o experimento é repetido e tem-se uma média de todas as melhores soluções a cada *n* gerações. Com esse procedimento, pode-se fazer uma análise do comportamento do algoritmo ao longo de todos os *ckeckpoints.*

Em problemas de otimização numérica, existe o chamado *critério de optimalidade* ε. Sabendo-se o valor da função objetivo no ponto ótimo f^*, pode-se considerá-lo alcançado sempre que a avaliação da função objetivo *f* estiver a um ε do valor. Ou seja, o *GAP* entre *f* e f^* seja suficientemente pequeno ($\varepsilon \cong 0$).

$$GAP = \left| f(x) - f(x^*) \right|$$
$$GAP \leq \begin{cases} \varepsilon & f(x^*) = 0 \\ \varepsilon * \left| f(x^*) \right| & f(x^*) \neq 0 \end{cases} \qquad (5.3)$$

Os diferentes algoritmos podem ser comparados entre versões canônicas de algoritmos evolutivos, ou comparados com outros algoritmos disponibilizados por seus autores para esse fim. Por exemplo, o Scatter Search (descrito na seção 3.3) foi comparado ao GENOCOP III (descrito na seção 4.2) através de uma bancada de teste composta de 40 funções. Os resultados foram em termos de GAP´s encontrados e são mostrados na Figura 5.3 [26].



Avaliações	100	500	1000	5000	10000	20000	50000
Genocop	1335.45	611.30	379.03	335.81	328.66	324.72	321.20
ScatterSearch	134.45	26.34	14.66	4.96	3.60	3.52	3.46

Figura 5.3 - Comparação entre Scatter Search e GENOCOP III: (a) gráfico que mostra a quantidade de soluções ótimas encontradas em relação ao número de avaliações de função objetivo; (b) tabela que mostra a média de GAP´s em cada faixa de número de avaliações da função objetivo

A comparação entre *Scatter Search* e GENOCOP III foi divida em faixas de número de avaliações da função objetivo. Pela Figura 5.3, o *Scatter Search* é superior ao GENOCOP III tanto no número de soluções encontradas por faixa, quanto na média de GAP´s em cada uma delas. Observa-se que após 50.000 avaliações, o GENOCOP III não encontrou GAP inferior ao *Scatter Search* em apenas 100 avaliações.

Muitos pesquisadores propõem seus algoritmos e os comparam com formas canônicas (básicas) de algoritmos evolutivos, com operadores pouco especializados e de amplo conhecimento da comunidade científica.

O Algoritmo TRAMSS (descrito na seção 3.2) foi comparado com uma série desses algoritmos com diferentes pares de mutação e cruzamento. A Tabela III mostra parte dos experimentos realizados pelos autores com seis funções-teste: Esfera (Sph)[4] , Rosenbrock (Ros)[4], Schwefel (Sch) [42], Rastrigin (Ras) [41], Griewangk (Gri) [19] e Expansão f10 (ef10)[43].

Algorithms	f_{Sph} A	f_{Sph} SD	f_{Ros} A	f_{Ros} SD	f_{Sch} A	f_{Sch} SD
R-RAN-BLX	7.7e-24 (+)	2.7e-23	2.0e1 (+)	1.5e1	5.1e-5 (+)	6.3e-5
R-BGA-BLX	6.6e-23 (+)	3.5e-22	1.7e1 (~)	9.7e0	1.7e-6 (+)	1.5e-6
R-NU-BLX	2.0e-31 (+)	5.8e-31	1.8e1 (+)	1.4e1	3.6e-7 (+)	4.2e-7
R-SELF-BLX	1.4e-26 (+)	4.7e-26	1.6e1 (~)	1.0e1	7.4e-8 (+)	5.7e-8
TRA-BLX	1.4e-200	0.0e0	1.2e1	4.8e0	8.8e-12	1.5e-11
Algorithms	f_{Ras} **A**	f_{Ras} **SD**	f_{Gri} **A**	f_{Gri} **SD**	ef_{10} **A**	ef_{10} **SD**
R-RAN-BLX	0.0e0 (-)	0.0e0	8.1e-3 (+)	1.5e-2	2.2e-5 (~)	1.6e-5
R-BGA-BLX	5.4e-1 (-)	1.4e0	4.3e-3 (+)	6.9e-3	6.5e-7 (~)	3.5e-6
R-NU-BLX	2.3e-1 (-)	9.5e-1	4.7e-3 (+)	8.4e-3	1.7e-7 (~)	9.9e-8
R-SELF-BLX	1.8e1 (~)	4.8e0	2.9e-2 (~)	1.1e-1	9.5e-1 (+)	2.2e0
TRA-BLX	3.3e0	2.2e0	0.0e0	0.0e0	1.8e-4	9.8e-4

Tabela III - Comparação entre TRAMSS e uma série de algoritmos evolutivos com diferentes operadores

A Tabela III mostra os nomes dos algoritmos testados, formados por composições de siglas dos operadores utilizados, e mostra ainda a média (A) e o desvio padrão (SD) das soluções encontradas. A Tabela IV mostra o significado de cada uma das siglas. Pode ser observado que poucos algoritmos conseguiram média 0.0e0 (solução ótima). Alguns só conseguiram 1.4e-200 (TRA-BLX) de média de GAP. Outra observação é que nem todas as versões canônicas tiveram desempenho inferior ao TRAMSS em todas as funções-teste (sinal -) e muitas tiveram desempenho, segundo o autor, equivalentes (sinal ~). Mas o TRAMSS foi superior em um número significativo de experimentos (sinal +) [12].

Sigla	Significado
R	Codificado em real
RAN	Mutação aleatória (ou uniforme)
BGA	Mutação de Mühlenbein
NU	Mutação não uniforme
BLX	Cruzamento *Blend*
SELF	Mutação auto-adptativa
TRA	Algoritmo TRAMSS (Herrera)

Tabela IV- Significado das siglas usadas em Herrera[12]

É igualmente comum contrapor os resultados do mesmo algoritmo com e sem alguma característica diferenciada, a qual se deseja mostrar a eficácia. Um exemplo desse tipo de comparação é apresentada no trabalho que propôs o FEP (descrito na seção 3.1). Foi acrescido ao FEP um operador de busca local baseado no gradiente conjugado com diferentes parâmetros de execução.

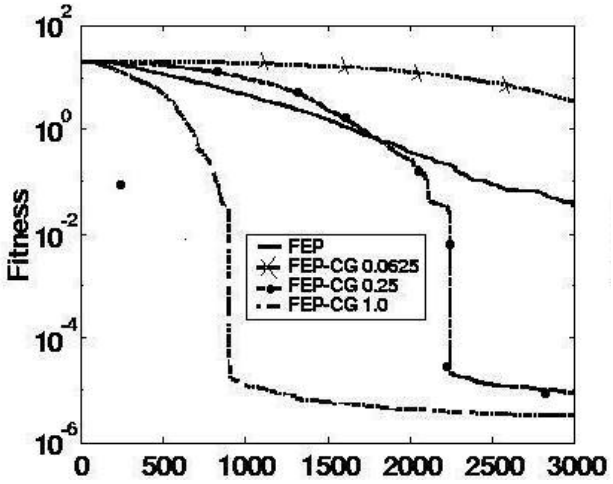


Figura 5.4 - Comparação entre FEP's com diferentes probabilidades de execução de busca local

O experimento mostrou o desempenho do FEP em relação ao número de avaliações de função objetivo para cada uma das versões do próprio FEP: FEP puro, FEP com busca local a 6.25%, FEP com busca local a 25% e FEP com busca local a 100% A Figura 5.5 mostra o gráfico resultado desses experimentos.

A Figura 5.5 mostra ,além de um significante ganho de desempenho para a versão com gradiente conjugado a 100%, que a versão pura do FEP teve desempenho superior à versão com apenas 6,25% de gradiente conjugado.

CAPÍTULO VI

6. Conclusão

Os mecanismos naturais que promovem a evolução dos seres vivos podem ser considerados processos inteligentes. Os algoritmos evolutivos, baseados nas leis da evolução natural, são algoritmos de otimização global, estocásticos que possuem características que os tornam aplicáveis a uma vasta quantidade de aplicações.

Este trabalho focalizou os algoritmos evolutivos para problemas de otimização numérica com variáveis reais. Inicialmente, foram apresentados alguns fundamentos destes algoritmos, bem como aspectos relacionados com a codificação de soluções e operadores evolutivos. Posteriormente, foram apresentados algumas propostas para solução de problemas de otimização numérica sem e com condições de restrição. Por último, foram apresentados alguns experimentos publicados, bem como algumas funções-teste e métricas para avaliação de desempenho de algoritmos evolutivos.

Dentre todas as propostas apresentadas neste trabalho, tendo em vistas os resultados encontrados na literatura a cerca de seus desempenhos, pode-se concluir que são contribuições efetivas para construção de novos algoritmos evolutivos de desempenho competitivo:

- O cruzamento BLX-α [12,13];
- O controle adaptativo de parâmetros de operadores evolutivos [24];
- A sistemática de busca usada pelo *Scatter Search* [25,26];
- O tratamento de restrições usado pelo GENOCOP III [8];
- O emprego de buscas locais não derivativas, como o *Downhill Simplex* [20,21,22]

O estudo desenvolvido neste trabalho abre perspectivas concretas para a construção de um *framework* para otimização numérica competitivo. O enfoque genético construtivo pode somar idéias e servir de alicerce para esse novo empreendimento [3].

As contribuições, que tal *framework* pode dar, são inúmeras. Especificamente, pode-se aplicá-lo em problemas inversos, treinamento de redes neurais, otimização de controladores *fuzzy*. Este trabalho pode ser considerado um primeiro passo.

REFERÊNCIAS

1 Oliveira, A. & Nascimento, E. *Uma Contribuição ao Estudo de Algoritmos Evolutivos Fuzzy.* In Proc VI Iberamia. pp. 175-186. Lisboa. 1998.

2 Oliveira, A. & Lorena, L. A. *A Constructive Evolutionary Approach to Linear Gate Assignment Problems.* In Proc Encontro Nacional de Inteligência Artificial (ENIA). Fortaleza, CE. 2001.

3 Lorena, L. A. N. and Furtado, J. C. *Constructive Genetic Algorithm for Clustering Problems*. Evolutionary Computation 9(3):309-327, 2001.

4 De Jong, K A, *An analysis of the behaviour of a class of genetic adaptive systems.* Ph.D. Dissertation, Univ. of Michigan, Ann Arbor.1975.

5 Goldberg D. E., *Genetic Algorithms in Search, Optimisation, and Machine Learning*. Addison-Wesley Publishing Company, Inc. 1989.

6 Wright, A. *Genetic algorithms for real parameter optimization*, in Foundations of Genetic Algorithms 1, G.J.E. Rawlin (Ed.), Morgan Kaufmann: San Mateo, 1991, pp. 205-218.

7 Michalewicz, Z. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, New York. 1992.

8 ___. *Genetic Algorithms + Data Structures = Evolution Programs*. 3rd Ed. Springer-Verlag, New York. 1996.

9 Bäck, T. Hoffmeister, F. and Schwefel, H.P. *A survey of evolution strategies.* In Lashon B. Belew, Richard K.; Booker, editor, Proceedings of the 4th International Conference on Genetic Algorithms, pages 2--9, San Diego, CA, July 1991.

10 Radcliffe, N. J. *Equivalence class analysis of genetic algorithms*. Complex Systems, 5:183– 20, 1991

11 Davis, L. *Adapting operator probabilities in genetic algorithms*. Proceedings of the Third International Conference on Genetic Algorithms, 60--69. 1989.

12 Herrera, F. Lozano, M. and Verdegay, J.L. *Tackling real-coded genetic algorithms: Operators and tools for behavioural analysis,* Artificial Intelligence Review, vol. 12, pp. 265--319, 1998.

13 Eshelman, L.J. Schawer, J.D. *Real-coded genetic algorithms and interval- schemata*, in Foundation of Genetic Algorithms-2, L. Darrell Whitley (Eds.), Morgan Kaufmann Publishers: San Mateo, pp. 187-202,1993.

14 Kita, H. Ono, I. and Kobayashi, K. *Theoretical Analysis of the Unimodal Normal Distribution Crossover for Real-coded Genetic Algorithms.* Proceedings of Intr. Conference on Evolutionary Computation :529-534. 1998.

15 Deb, K. and Beyer, H.G. *Self-Adaptive Genetic Algorithms with Simulated Binary Crossover*. Evolutionary Computation Journal, 9 (2), 197--221. 2001

16 Tsutsui, S. and Yamamura, M. *Multi-parent Recombination with Simplex Crossover in Real Coded Genetic Algorithms*, Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-99), pp. 657-664 .1999.

17 Voigt H. M., and Anheyer T. *Modal Mutations in Evolutionary Algorithms* Proc. IEEE Int. Conf. on Evolutionary Computation, vol. I, pp.88-92, IEEE 1994

18 Mühlenbein, H. and Schlierkamp-Voosen, D. *Predictive models for the breeder genetic algorithm I. continuous parameter optimization,* Evolutionary Computation, 1, pp. 25-49, 1993.

19 A.O. Griewangk, *Generalized descent of global optimization*, Journal of Optimization Theory and Applications, vol. 34, pp. 11-39, 1981.

20 Yen, J. and Lee, B. *A Simplex Genetic Algorithm Hybrid*, in Proceedings of the IEEE Conference on Evolutionary Computation (ICEC'97), Indianapolis, Indiana, April 13-16, 1997.

21 Yen, J. et al. *A hybrid approach to modeling metabolic systems using a genetic algorithm and simplex method*. IEEE Transactions on Systems, Man, and Cybernetics - Part B: Cybernetics, 28(2):173--191, 1998.

22 Nelder, J. A. and Mead, R. *A simplex method for function minimization.* Computer Journal, vol. 7, pp. 308–313, 1965.

23 Birru, H.K., Chellapilla, K. and Rao, S.S. *Local search operators in fast evolutionary programming*. In Proceedings of the 1999 Congress on Evolutionary Computation, volume 2, pages 1506--1513, IEEE Press, Piscataway, NJ, 1999.

24 Herrera, F. and Lozano, M. *Two-Loop Real-Coded Genetic Algorithms with Adaptive Control of Mutation Step Sizes.* Technical Report #DECSAI-97-01-28, Dept. of Computer Science and Artificial Intelligence, University of Granada, Spain, 1997. (Available at the URL address: http://decsai.ugr.es/~lozano/public.html)

25 Glover F. *A Template for Scatter Search and path Relinking*. Artificial Evolution, Lecture Notes in Computer Sciense, Springer-Verlag, pp 13-54. 1998

26 Laguna, M. and Martí, R. *Experimental Testing of Advanced Scatter Search Designs for Global Optimization of Multimodal Functions.* August 2000.

27 Glover, F. *Future paths for integer programming and links to artificial intelligence*. Computers and Operations Research, 13. p. 533- 549. 1986.

28 Mahfoud, S W. *Niching Methods for Genetic Algorithms*. PhD thesis, University of Illinois at Urbana-Champaign, IlliGAL Report No. 95001. 1995.

29 Sacco, W. F. et al. *A Genetic Algorithm with Sharing Scheme using Fuzzy Adaptive Clustering in Multimodal Function Optimization*. In Proc Encontro Nacional de Inteligência Artificial (ENIA). Fortaleza, CE. 2001.

30 Grüninger, T. and Wallace, D. *Multi-modal optimization using genetic algorithms*. Technical report . NSF CIPD Technical report. 1996. On-line: http://cadlab.mit.edu/publications/96-senin-tr9602-ga/abstract.html

31 Goldberg, D. E. and Richardson, J. J. Genetic algorithms with sharing for multimodal function optimization, Gen. Algs. and their Apps.: Proc. 2nd Intl. Conf. Gen. Algs., Lawrence Erlbaum Eds. Cambridge, MA, July 28–31, 1987.

32 Taha, H.A., *Operations Research: An Introduction.* 4th ed., Collier Macmillan, London, 1987.

33 Michalewicz, Z. and Schoenauer, M. *Evolutionary algorithms for constrained parameter optimization problems*. Evolutionary Computation, 1996.

34 Richardson, J.T., M.R. Palmer, G. Liepins and M. Hilliard. *Some Guidelines for Genetic Algorithms with Penalty Functions*. In Proceedings of the Third International Conference on Genetic Algorithms, Los Altos, CA, Morgan Kaufmann Publishers, 191--197. 1989.

35 Orvosh, D. and L. Davis. *Shall We Repair? Genetic Algorithms, Combinatorial Optimization, and Feasibility Constraints.* In Proceedings of the Fifth International Conference on Genetic Algorithms, Los Altos, CA, Morgan Kaufmann Publishers, 650. 1993.

36 Homaifar, A., S. H.-Y. Lai and X. Qi. *Constrained Optimization via Genetic Algorithms*. Simulation, 62: 242--254. 1994.

37 Joines, J.A. and C.R. Houck. *On the Use of Non-Stationary Penalty Functions to Solve Nonlinear Constrained Optimization Problems With GAs*. In Proc. of the Evolutionary Computation Conference---Poster Sessions, part of the IEEE World Congress on Computational Intelligence, Orlando, June 1994.

38 Powell, D. and M.M. Skolnick. *Using Genetic Algorithms in Engineering Design Optimization with Non-linear Constraints.* In Proceedings of the Fifth International Conference on Genetic Algorithms, 424--430. Los Altos, CA, Morgan Kaufmann Publishers, 1993.

39 Schoenauer, M., and S. Xanthakis. *Constrained GA Optimization.* In Proceedings of the Fifth International Conference on Genetic Algorithms, Los Altos, CA, Morgan Kaufmann Publishers, 1993, 573–580.

40 Lee, M. A. & Takagi, H. Dynamic control of genetic algorithms using fuzzy logic techniques. In Proc. Fifth International Conference on Genetic Algorithms (ICGA93), pp. 76-83. San Mateo. 1993.

41 Bäck, T. *Self- Adaptation in genetic algorithms*, in Proc. of the First European Conference on Articial Life, F.J. Varela, P. Bourgine (Eds.), The MIT Press: Cambridge, MA, 1992, pp. 263-271.

42 Schwefel, H-P. *Numerical Optimization of Computer Models*, Wiley: Chichester, 1981.

43 Whitley, D. Beveridge, R. Graves, C. and Mathias, K. *Test driving three 1995 genetic algorithms: new test functions and geometric matching*, Journal of Heuristics, 1, pp. 77-104, 1995.