Representação, Operadores Evolutivos e Busca Local

1 Introdução

- Este tópico tem por objetivo revisar as principais estruturas de dados utilizadas como representação computacional em algoritmos evolutivos, os diversos tipos de operadores genéticos e mecanismos de seleção empregados em computação evolutiva, assim como alguns operadores e técnicas de busca local.

2 Representações

- Cada método de busca manipula soluções candidatas que representam uma instância do problema a ser resolvido.
- Entretanto, a estrutura de uma solução candidata poderá depender diretamente do problema abordado. Cada método de busca também possui características que ajudam na especificação do tipo de representação a ser empregada.
- Sendo assim, a eficiência e a complexidade do método de busca irão depender da representação e, consequentemente, dos operadores de busca envolvidos.

- Fica claro, portanto, que a definição de uma representação é uma das etapas mais críticas na implementação de um algoritmo evolutivo. A definição inadequada da representação pode levar a superfícies de *fitness* extremamente "acidentadas".
- Em problemas de otimização com restrições, a codificação adotada pode fazer com que indivíduos modificados por crossover/mutação sejam infactíveis. Nestes casos, cuidados especiais devem ser tomados na definição da codificação e/ou dos operadores (BÄCK *et al.*, 2000b).

2.1 Codificação Binária

- Na maioria das aplicações de algoritmos genéticos (AGs) as estruturas de dados utilizadas como representação das soluções candidatas são cadeias binárias, mesmo quando as variáveis do problema são inteiras ou reais.
- Uma variável real $x \in (a,b)$ pode ser codificada utilizando-se cadeias binárias de comprimento l, o que resultará em uma precisão numérica de $(a-b)/(2^l-1)$.

- A motivação para o uso de codificação binária vem da teoria dos esquemas (*schemata theory*). HOLLAND (1975; 1992) argumenta que seria benéfico para o desempenho do algoritmo maximizar o paralelismo implícito inerente ao AG, e prova que um alfabeto binário maximiza o paralelismo implícito.
- Entretanto, em diversas aplicações práticas a utilização de codificação binária leva a um desempenho insatisfatório. Em problemas de otimização numérica com parâmetros reais, AGs com representação em ponto flutuante frequentemente apresentam desempenho superior à codificação binária.
- MICHALEWICZ (1996) argumenta que a representação binária apresenta desempenho pobre quando aplicada a problemas numéricos com alta dimensionalidade e em que alta precisão é requerida. Suponha, por exemplo, que temos um problema com 100 variáveis com domínio no intervalo [−500, 500] e que precisamos de 6 dígitos de precisão após a casa decimal. Neste caso precisaríamos de um cromossomo de comprimento 3000, e teríamos um espaço de

busca de dimensão aproximadamente 10^{1000}. Neste tipo de problema, o algoritmo genético clássico apresenta desempenho pobre.

- Uma das características da representação binária é que dois pontos vizinhos na codificação não são necessariamente vizinhos no espaço de busca definido pela representação do problema.
- Uma forma de amenizar este problema é utilizar uma representação do tipo código de Gray, onde a distância de Hamming entre duas cadeias consecutivas quaisquer é sempre 1.

Inteiro	Binário	Gray
0	0 0 0	0 0 0
1	0 0 1	0 0 1
2	0 1 0	0 1 1
3	0 1 1	0 1 0
4	1 0 0	1 1 0
5	1 0 1	1 1 1
6	1 1 0	1 0 1
7	1 1 1	1 0 0

- Mesmo assim, a mudança de um único bit em codificação binária ou Gray pode resultar em grandes "saltos" no valor decodificado.
- Procedimentos para transformar um código binário em Gray e vice-versa. Sejam $\boldsymbol{b} = \langle b_1, ..., b_l \rangle$ a cadeia binária e $\boldsymbol{g} = \langle g_1, ..., g_l \rangle$ a cadeia em código Gray.

```
procedimento Binário_para_Gray
  g1 ← b1
  para k = 2 até l faça,
    gk ← gk-1 XOR bk
  fim para
fim procedimento

procedimento Gray_para_Binário
  valor ← g1
  b1 ← valor
  para k = 2 até l faça,
      se gk == 1
          então valor ← NOT valor
      fim se
    bk ← valor
  fim para
fim procedimento
```


- Embora o código de Gray garanta que números inteiros vizinhos serão vizinhos no hipercubo, mesmo assim vão ocorrer vizinhanças indesejáveis. Assim, o código de Gray apenas ameniza o problema de acidentalidade da superfície de fitness.

2.2 Codificação binária e grade uniforme

- Mesmo não sendo recomendado, é relevante apresentar o formalismo que permite a representação em grade uniforme de espaços de busca contínuos. Existem também problemas que são diretamente representados por espaços de busca na forma de grades uniformes, admitindo assim uma codificação binária apropriada.
- Mas aqui será considerada a grade uniforme como uma forma de quantização para a representação de espaços contínuos. O resultado da busca na grade uniforme certamente irá produzir apenas uma aproximação da solução desejada.
- Considerando que uma certa solução candidata é representada por $x \in \Re^n$, tal que $x = [x_1 \quad x_2 \quad \cdots \quad x_n]^T$, é necessário restringir o espaço de busca a uma região

compacta, de modo que $x_i \in [u_i, v_i]$, i=1,...,n e u_i e v_i correspondem aos limites inferior e superior de x_i, respectivamente.

- Supondo que se aloquem l_x bits a cada elemento do vetor $x \in \Re^n$, então o número total de bits de cada solução candidata será dado por um vetor binário $a \in \{0,1\}^l$ de dimensão $l = n * l_x$ bits.
- A Figura 1 a seguir mostra a associação entre os bits do vetor a e os elementos do vetor x. Seja a_k o k-ésimo elemento do vetor binário $a \in \{0,1\}^l$, então x_i vai corresponder à sequência binária $[a_{(i-1)*l_x+1} \quad \cdots \quad a_{i*l_x}]$.
- Logo, existem 2^{l_x} valores possíveis para cada $x_i \in [u_i, v_i]$, i=1,...,n. A Figura 2 apresenta um exemplo de grade uniforme para n = 2, $l_x = 3$, $x_1 \in [1, 7]$ e $x_2 \in [0, 4]$.

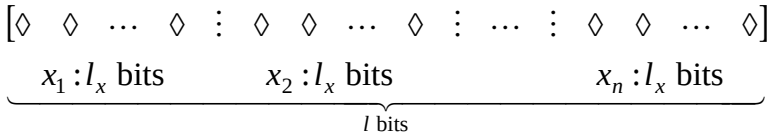

Figura 1 – Representação binária para uma região compacta do $\Re^n$

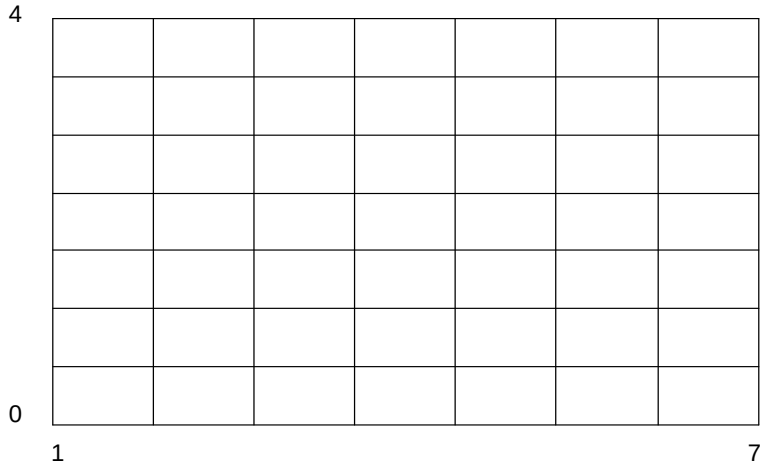

Figura 2 – Exemplo de grade uniforme para quando *n* = 2, $l_x = 3$, $x_1 \in [1, 7]$ e $x_2 \in [0, 4]$

- Para se chegar aos valores quantizados de $x_i \in [u_i, v_i]$, i=1,...,n, deve-se empregar o mapeamento $\{0,1\}^{l_x} \rightarrow [u_i, v_i]$ dado por:

$$x_i = u_i + \frac{v_i - u_i}{2^{l_x} - 1}\left(\sum_{j=0}^{l_x-1} a_{i*l_x-j} * 2^j\right)$$

2.3 Codificação em ponto flutuante

- Os computadores digitais, por empregarem apenas código binário na representação de informação, são capazes de representar apenas uma quantidade finita (embora ampla) de números racionais. Logo, muitos números racionais e nenhum número irracional é representado exatamente em computador digital.
- Esta é a razão pela qual não se deve empregar a denominação de codificação em números reais aqui, ficando mais adequada a denominação de codificação em ponto flutuante.

- Feita esta observação, a codificação em ponto flutuante é direta, havendo tantos elementos no vetor de cada solução candidata quanto a dimensão do espaço de busca.

2.4 Permutações

- Uma permutação de um conjunto finito é um arranjo ordenado de seus elementos (KNUTH, 1973). Dados *n* objetos distintos, existem *n*! permutações destes objetos.
- Existem diversos problemas que são naturalmente representados por permutações. Exemplos: TSP e sequenciamento de tarefas.
- Neste tipo de codificação, os operadores genéticos a serem empregados devem ser distintos dos utilizados com representação binária ou real.
- A codificação também é direta aqui, bastando associar um inteiro a cada objeto. A ordem em que os inteiros aparecem num vetor indica a permutação proposta.
- Permutações também podem ser associadas a vetores em ponto flutuante, pois ao ordenar os valores se chega numa permutação também.

- Não é elementar o cálculo de distância entre permutações, sendo que a distância deve envolver o número mínimo de operações que permite converter uma permutação em outra.

2.5 Máquinas de Estado Finito

- Esta representação é apropriada para evoluir sistemas de tomada de decisão a partir de eventos discretos e também na predição de séries temporais discretas.
- Uma máquina de estado finito *M* é essencialmente um programa computacional: ela representa uma sequência de instruções a serem executadas, cada qual dependendo de um estado atual da máquina e do estímulo atual.
- Formalmente uma máquina de estado finito pode ser descrita por uma quíntupla $M = \langle Q,\tau,\rho,s,o\rangle$, onde:
  - ✓ Q é um conjunto finito de *estados*;
  - ✓ τ é um conjunto finito de *símbolos de entrada* (estímulos),
  - ✓ ρ é um conjunto finito de *símbolos de saída*;

  - ✓ $s : Q \times \tau \rightarrow Q$ é a *função que mapeia para o próximo estado*; e
  - ✓ $o : Q \times \tau \rightarrow \rho$ é a *função que mapeia para a próxima saída*.
- Dado um símbolo de entrada x e estando a máquina no estado q, ela fornecerá como saída o símbolo $o(q,x)$ e mudará para o estado $s(q,x)$.
- A informação contida no estado atual da máquina é suficiente para descrever seu comportamento em relação a qualquer entrada admissível.
- O conjunto de estados serve como uma espécie de memória da máquina. Sendo assim, uma máquina de estado finito é um *conversor de símbolos*, pois pode ser estimulada por um alfabeto finito de símbolos de entrada, responde através de um alfabeto finito de símbolos de saída e possui uma quantidade finita de estados internos. Em outras palavras, uma MEF é um grafo de transição de estados, em que a transição de qualquer estado para um próximo estado depende de eventos discretos e finitos.

- Os correspondentes pares de símbolos de entrada-saída e transições para o próximo estado de cada símbolo de entrada, tomados sobre cada estado, especificam o comportamento de cada máquina de estado finito, dada uma condição inicial.
- Como um exemplo, considere a máquina *M* de três estados apresentada na Figura 3.

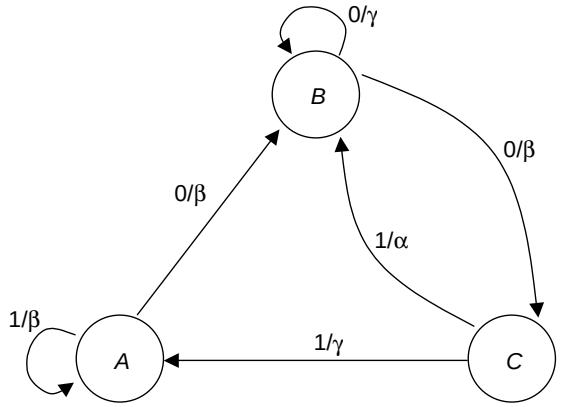

Figura 3 – Máquina de estado finito de três estados. Os símbolos de entrada são mostrados à esquerda da barra "/" (FOGEL et al., 1966).

- O alfabeto de símbolos de entrada é composto por {0,1}, o alfabeto de símbolos de saída é composto por {α,β,γ} (FOGEL, 2000).
- A máquina de estado finito transforma uma sequência de símbolos de entrada em uma sequência de símbolos de saída. A tabela abaixo indica a resposta da máquina a uma dada sequência de símbolos, considerando que a máquina encontra-se no estado *C*.

Estado atual	*C*	*B*	*C*	*A*	*A*	*B*
Símbolo de entrada	0	1	1	1	0	1
Próximo estado	*B*	*C*	*A*	*A*	*B*	*C*
Símbolo de saída	β	α	γ	β	β	α

- É suposto que a máquina atua quando um símbolo de entrada é percebido e responde antes que o próximo símbolo de entrada seja apresentado.
- As codificações mais comuns indicam, para cada estado, se ele é inicial, final ou nenhum dos dois, e quais são as funções $s(q,x)$ e $o(q,x)$ para todo q e x.

- Como uma variação, tem-se o caso em que não se consideram símbolos de saída, de modo que os símbolos de entrada afetam apenas a transição de estados. Podem ser definidos também um estado inicial e um estado final, conforme apresentado na Figura 4.

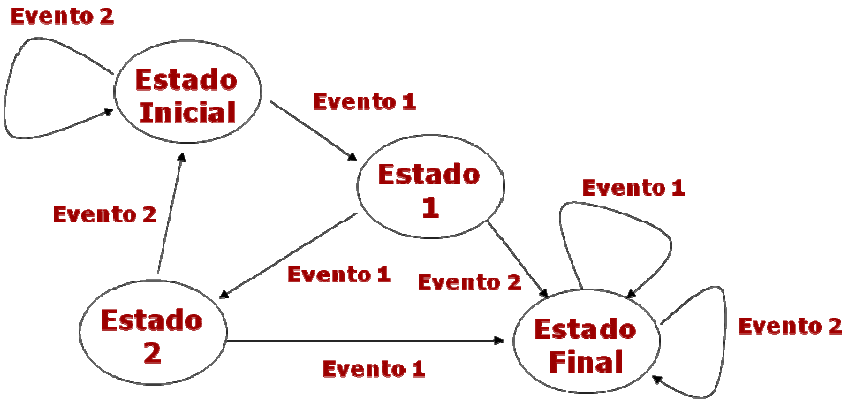

Figura 4 – Máquina de estado finito com estado inicial e final. São considerados dois símbolos de entrada possíveis (Evento 1 e Evento 2).

2.6 Árvores

- Uma árvore é um grafo que apresenta um caminho único entre quaisquer pares de seus nós. As árvores com raiz representam estruturas de dados que possuem uma relação de hierarquia entre seus nós. São muito utilizadas em inteligência artificial, particularmente para definir estratégias ótimas de solução de problemas, partindo da raiz até um nó-folha. A Figura 5 a seguir destaca esta característica de uma representação em árvore, além de indicar a notação geralmente adotada.
- Em computação evolutiva, as representações em árvore são geralmente empregadas para evoluir programas computacionais. Além de representarem estruturas mais complexas que listas, as árvores podem variar significativamente de tamanho ao longo do processo evolutivo. Todas as aplicações práticas descritas em KOZA (1992) utilizam mecanismos para limitar o tamanho final do programa evoluído. Duas técnicas são usualmente empregadas: limitação de profundidade ou de número de nós (ANGELINE & KINNEAR JR., 1996).

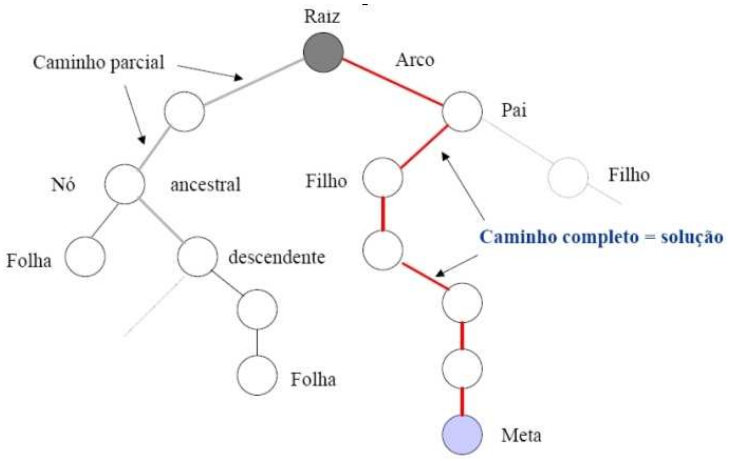

Figura 5 – Exemplo de árvore de decisão e notação correspondente

- O processo de compilação de programas computacional geralmente constroi uma árvore sintática para validar sintaticamente o programa. Sendo assim, a evolução de árvores sintáticas (parse trees) vai sempre produzir programas sintaticamente corretos, sendo um aspecto muito relevante para a representação de soluções candidatas em programação genética (BÄCK et al., 2000a).

- Na Figura 6 a seguir, encontra-se um exemplo de um programa computacional que implementa uma expressão algébrica que envolve constantes e variáveis.

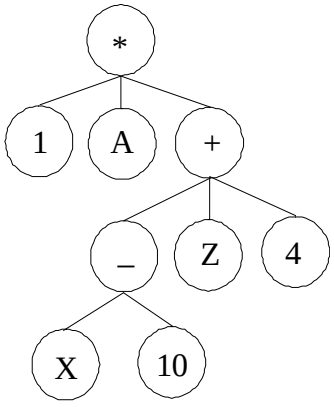

Figura 6 – Exemplo de programa computacional em uma representação em árvore

- Para a codificação de árvores como as da Figura 6, geralmente se empregam expressões simbólicas adotadas pela linguagem LISP. Elas envolvem o uso de parênteses aninhados, representando a hierarquia. Para a árvore da Figura 6, a codificação assume a forma: **`(* 1 A (+ (– X 10) Z 4 ) )`**

- Variáveis e constantes aparecem sempre nas folhas da árvore (também denominados de nós terminais), enquanto que os nós internos são responsáveis pelas operações. O aninhamento de parênteses está diretamente associado à topologia da árvore. Os argumentos que estão vinculados a cada operador presente nos nós internos devem ser consistentes com o que é requerido pelo operador.
- Existem outras codificações possíveis para árvores, como a matriz com elementos binários apresentada na Figura 7 a seguir, voltada para a codificação de árvores binárias, ou seja, árvores que apresentam sempre dois filhos para cada nó-pai. A árvore resultante da codificação da Figura 7 é apresentada na Figura 8.
- Nesta codificação alternativa, somente os nós internos aparecem nas linhas da matriz. Um bit 1 em alguma coluna indica quem é filho daquele nó-pai daquela linha. Sendo assim, esta codificação não está restrita a árvores binárias, podendo ser prontamente estendida a árvores que apresentam um número arbitrário de filhos para cada nó-pai.

Pais	E 1 (1)	E 2 (2)	E 3 (3)	E 4 (4)	E 5 (5)	A 1 (6)	A 2 (7)	A 3 (8)	A 4 (9)
A 1 (6)	0	0	0	0	0	0	1	1	0
A 2 (7)	1	1	0	0	0	0	0	0	0
A 3 (8)	0	0	1	0	0	0	0	0	1
A 4 (9)	0	0	0	1	1	0	0	0	0

Figura 7 – Exemplo de codificação em matriz binária para árvore binária com raiz

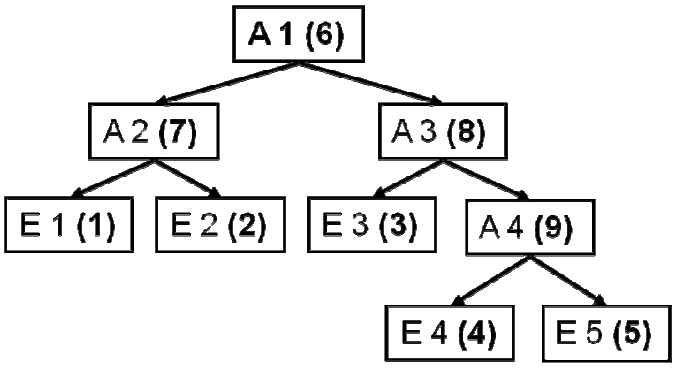

Figura 8 – Árvore resultante da codificação da Figura 7

3 Operadores de Busca

- Sabemos que a evolução é resultado da *reprodução*, *variação genética* e *seleção natural* aplicados a uma população de indivíduos.
- Até o momento já foram discutidos os principais tipos de representação dos algoritmos evolutivos. Cabe agora identificar os principais tipos de operadores genéticos a serem empregados.
- Existem *transformações unárias*, do tipo mutação, que criam um novo indivíduo (descendente) partindo de um único progenitor, e também *transformações de ordem mais elevada*, do tipo recombinação, que geram novos indivíduos através da recombinação de características de dois ou mais indivíduos progenitores.
- Como já mencionado, existe uma interdependência entre a representação empregada (que geralmente depende do problema) e os operadores genéticos escolhidos. Sendo assim, os operadores genéticos serão apresentados considerando-se a representação adotada.

4 Operadores de Mutação

- Geração de um ou mais novos indivíduos (descendentes) partindo de um único progenitor.

4.1 Codificação Binária

- A mutação inicialmente proposta para representação com cadeias binárias é denominada de *mutação pontual*. Cada posição da cadeia possui uma probabilidade p_m de sofrer mutação.
- Estudos empíricos e teóricos sugerem que:
  - Um valor inicial grande para a mutação deve ser adotado e decrescido geometricamente ao longo das gerações (FOGARTY, 1989);
  - Um limite inferior $p_m = 1/dim\_espaço\_de\_busca$ para a taxa ótima de mutação pode ser empregado (BREMERMANN et al., 1966, MÜHLENBEIN, 1992, BÄCK, 1993).

4.2 Codificação em ponto flutuante

- A mutação refere-se ao processo de gerar novos indivíduos partindo de um único progenitor. Sendo assim, dado um indivíduo $\boldsymbol{x} \in \Re$, seu correspondente valor mutado pode ser expresso por: $\boldsymbol{x}' = m(\boldsymbol{x})$, onde $m(\cdot)$ é a função de mutação. Exemplos:
  - $\boldsymbol{x}' = \boldsymbol{x} + \boldsymbol{M}$, onde $\boldsymbol{M}$ é uma variável aleatória.
  - Geralmente $\boldsymbol{M}$ possui média zero tal que $E(\boldsymbol{x}') = \boldsymbol{x}$, ou seja, a esperança matemática da diferença entre $\boldsymbol{x}$ e $\boldsymbol{x}'$ é zero.
  - *Mutação uniforme*: $\boldsymbol{M}$ pode assumir diversas formas, como, por exemplo, uma distribuição aleatória uniforme $U(a,b)^l$, onde a e b são os limites inferiores e superiores da variável. Geralmente $a = -b$.
  - *Mutação gaussiana*: Outra alternativa para $\boldsymbol{M}$ é utilizar uma distribuição normal ou gaussiana $N(mean,\sigma)^l$ de média *mean* (em geral *mean* = 0) e

desvio-padrão σ. Para uma distribuição multivariada, a média torna-se um vetor e o desvio-padrão deve ser substituído pela matriz de covariâncias.

- Outro operador de mutação, especialmente desenvolvido para problemas de otimização com restrição e codificação em ponto flutuante, é a chamada *mutação não-uniforme* (MICHALEWICZ, 1996; MICHALEWICZ & SCHOENAUER, 1996). A mutação não-uniforme é um operador dinâmico destinado a melhorar a sintonia fina do processo de busca. Podemos defini-lo da seguinte forma: seja $\mathbf{x}^t = [x_1 \ldots x_n]$ um cromossomo e suponha que o elemento x_k foi selecionado para mutação; o cromossomo resultante será $\mathbf{x}^{t+1} = [x_1 \ldots x_k' \ldots x_n]$, onde

$$x_k' = \begin{cases} x_k + \Delta(t, a - x_k), & \text{com } 50\% \text{ de probabilidade} \\ x_k - \Delta(t, x_k - b), & \text{com } 50\% \text{ de probabilidade} \end{cases}$$

onde a e b são os limites inferiores e superiores da variável x_k. A função $\Delta(t, y)$ retorna um valor no intervalo $[0, y]$ tal que a probabilidade de $\Delta(t, y)$ ser próximo de zero aumenta à medida que t aumenta.

- Esta propriedade faz com que este operador inicialmente explore o espaço de busca de forma mais ampla (quando t é pequeno) e localmente em gerações avançadas (quando t é grande);

- MICHALEWICZ (1996) propõe a seguinte função:

$$\Delta(t, y) = y \cdot \left(1 - r^{(1-t/T)^p}\right)$$

onde r é um número aleatório no intervalo $[0, 1]$, T é o número máximo de gerações e p é um parâmetro que determina o grau de dependência do número de iterações (valor proposto por MICHALEWICZ (1996): $p = 5$).

- A Figura 9 a seguir ilustra o comportamento desta função para valores específicos de seus parâmetros.

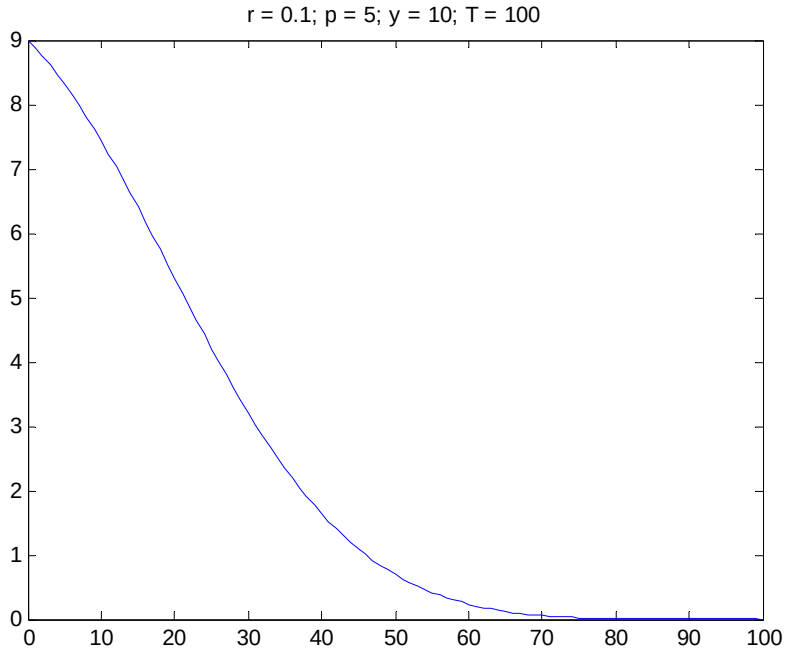

Figura 9 – Comportamento da função proposta por Michalewicz (1996)

4.3 Permutações

- Qualquer mutação a ser aplicada a uma permutação deve gerar uma estrutura de dados que também é uma permutação.
- A mutação mais comum para permutações é aquela que seleciona dois pontos da cadeia e reverte o segmento entre os pontos. Exemplo:
  - A | B C D E | F $\rightarrow$ A | E D C B | F
- O operador de mutação também pode ser estendido para *k* pontos, de modo que as múltiplas sub-sequências são revertidas. Exemplo, *k* = 3:
  - A | B C D | E F | G H | I J $\rightarrow$ A | D C B | F E | H G | I J
- Um caso particular da reversão de sequências é aquele em que duas posições (alelos) são selecionadas e seus genes trocados. Esta mutação é denominada de *mutação baseada em ordem* (SYSWERDA, 1991).

- Outro operador de mutação, denominado de *mistura* (*scramble*) seleciona uma sublista e reordena seus elementos aleatoriamente (SYSWERDA, 1991).

4.4 Máquinas de Estado Finito

- Existem diversas formas de mutar uma máquina de estado finito (MEF).
  - Mudar um símbolo de saída;
  - Mudar uma transição de estado;
  - Adicionar um estado;
  - Deletar um estado;
  - Mudar o estado inicial.

4.5 Árvores

- Dada a codificação empregando linguagem LISP, o operador de mutação pode envolver a troca de símbolos terminais, a troca de símbolos não-terminais, a inclusão e a exclusão de subramos. Maiores detalhes serão apresentados junto ao tópico de Programação Genética.

- Para a codificação em matriz binária, as Figuras 10, 12 e 14 a seguir apresentam os três tipos de trocas que podem ser realizadas, implicando que qualquer tipo de árvore binária pode ser obtida a partir de aplicações sucessivas de operações de mutação.
- Árvores binárias, além de estarem associadas a programas computacionais, podem também representar relações filogenéticas entre espécies, dentre outras aplicações (PRADO, 2001).

	E 1 (1)	E 2 (2)	E 3 (3)	E 4 (4)	E 5 (5)	A 1 (6)	A 2 (7)	A 3 (8)	A 4 (9)
A 1 (6)	0	0	0	0	0	0	1	1	0
A 2 (7)	1	0	0	1	0	0	0	0	0
A 3 (8)	0	0	1	0	0	0	0	0	1
A 4 (9)	0	1	0	0	1	0	0	0	0

Figura 10 – Mutação que envolve a troca entre duas colunas à esquerda da raiz

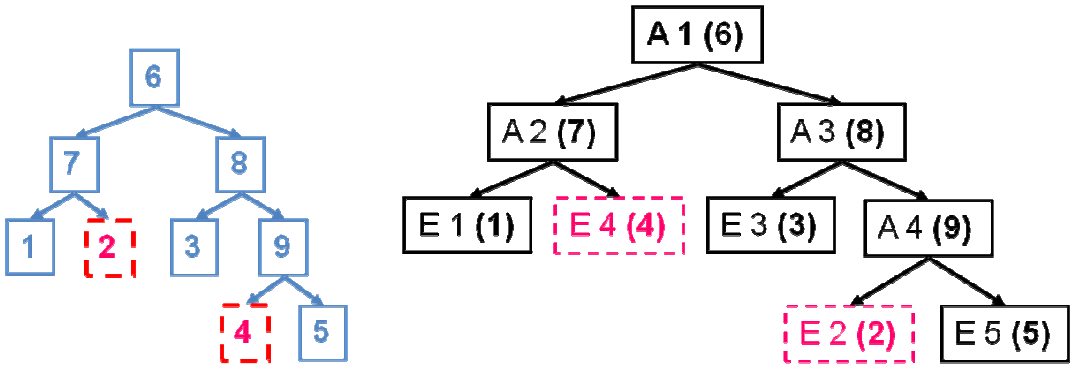

Figura 11 – Árvore antes e depois da mutação da Figura 10

	E 1 (1)	E 2 (2)	E 3 (3)	E 4 (4)	E 5 (5)	A 1 (6)	A 2 (7)	A 3 (8)	A 4 (9)
A 1 (6)	0	0	0	0	0	0	0	1	1
A 2 (7)	1	1	0	0	0	0	0	0	0
A 3 (8)	0	0	1	0	0	0	1	0	0
A 4 (9)	0	0	0	1	1	0	0	0	0

Figura 12 – Mutação que envolve a troca entre duas colunas à direita da raiz

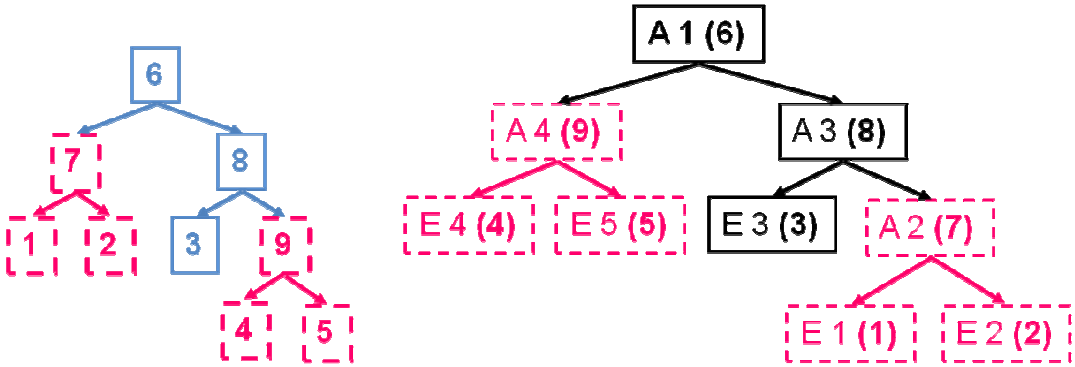

Figura 13 – Árvore antes e depois da mutação da Figura 12

	E 1 (1)	E 2 (2)	E 3 (3)	E 4 (4)	E 5 (5)	A 1 (6)	A 2 (7)	A 3 (8)	A 4 (9)
A 1 (6)	0	0	0	0	0	0	1	1	0
A 2 (7)	1	0	0	0	0	0	0	0	1
A 3 (8)	0	1	1	0	0	0	0	0	0
A 4 (9)	0	0	0	1	1	0	0	0	0

Figura 14 – Mutação que envolve a troca entre uma coluna à esquerda e outra à direita da raiz

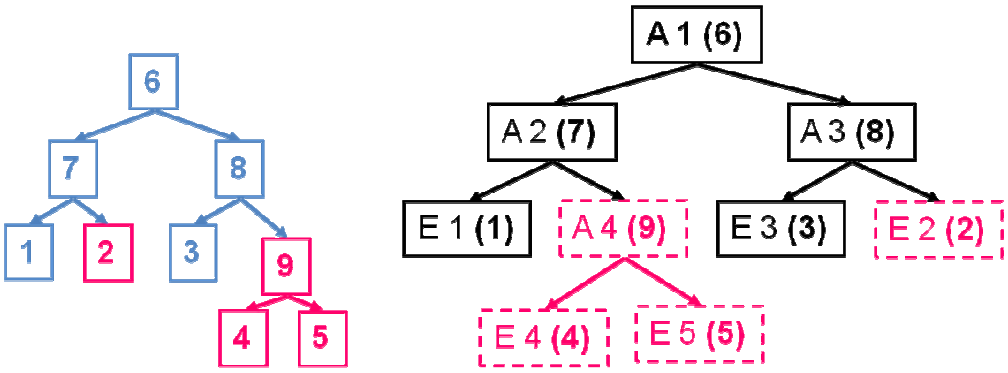

Figura 15 – Árvore antes e depois da mutação da Figura 14

5 Operadores de Recombinação

- Os operadores de recombinação trocam partes das estruturas de dados de dois ou mais progenitores com o objetivo de produzir um ou mais descendentes.

5.1 Codificação Binária

- Crossover de 1 ou *n* pontos entre indivíduos aleatórios (Roulette Wheel), ou então entre indivíduo aleatório (Roulette Wheel) e melhor indivíduo.
- Crossover *uniforme* (ACKLEY, 1987; SYSWERDA, 1989) ou baseado em *máscara* entre indivíduos aleatórios (Roulette Wheel), ou então entre indivíduo aleatório (Roulette Wheel) e melhor indivíduo.
- SCHAFFER & MORISHIMA (1987) propuseram o *crossover puntuado* (*punctuated crossover*) onde uma cadeia binária representando *marcas puntuadas* m_i, $i = 1,...,l$, indicam os pontos de crossover para o crossover de múltiplos pontos. Esta cadeia binária é adicionada ao final da cadeia representante da estrutura de dados e estará sujeita ao processo evolutivo: $\boldsymbol{x} = (x_1, x_2,..., x_l, m_1,..., m_l)$

5.2 Codificação em ponto flutuante

- A maioria dos operadores de crossover utilizados com codificação em ponto flutuante são derivados das estratégias evolutivas.
- Entretanto, crossover de um ou mais pontos também podem ser empregados para representação em ponto flutuante.
- Diversos operadores de recombinação podem ser definidos. Seja x_i o atributo *i* do vetor *x* a sofrer mutação, e *a*,*b* os índices dos pais *a* ou *b*, respectivamente:
  - *Recombinação discreta* (*local*): $x_i = x_{a,i}$ ou $x_{b,i}$ (crossover *uniforme*)
  - *Recombinação intermediária* (*local*): $x_i = ½(x_{a,i} + x_{b,i})$
  - *Recombinação discreta global*: $x_i = x_{a,i}$ ou $x_{bi,i}$
  - *Recombinação intermediária global*: $x_i = ½(x_{a,i} + x_{bi,i})$

  onde *bi* é um pai qualquer escolhido da população atual.
- Outro operador para representação em ponto flutuante é o *crossover aritmético* definido como uma combinação linear de dois vetores (cromossomos): sejam $\mathbf{x}_1$ e

$\mathbf{x}_2$ dois indivíduos selecionados para crossover, então os dois descendentes resultantes serão $\mathbf{x}_1' = a\mathbf{x}_1 + (1-a)\mathbf{x}_2$ e $\mathbf{x}_2' = (1-a)\mathbf{x}_1 + a\mathbf{x}_2$, onde *a* é um número aleatório pertencente ao intervalo [0,1]. Este operador é particularmente apropriado para problemas de otimização numérica com restrições, onde a região factível é convexa. Isto porque, se $\mathbf{x}_1$ e $\mathbf{x}_2$ pertencem à região factível, combinações convexas de $\mathbf{x}_1$ e $\mathbf{x}_2$ serão também factíveis. Assim, garante-se que o crossover não gera indivíduos infactíveis para o problema em questão. Uma variação que permite extrapolar a região de ocorrência dos descendentes é tomar *a* pertencente ao intervalo [-0,1;1,1].

- Outros exemplos de crossover desenvolvidos para utilização em problemas de otimização numérica restritos e representação em ponto flutuante são:
  - *Crossover heurístico* (WRIGHT, 1994): $\boldsymbol{x}' = u(\boldsymbol{x}_1 - \boldsymbol{x}_2) + \boldsymbol{x}_2$, onde $u \in [0,1]$ e $\boldsymbol{x}_1$ e $\boldsymbol{x}_2$ são dois progenitores tais que $f(\boldsymbol{x}_2) \geq f(\boldsymbol{x}_1)$.

  - *Crossover geométrico*, *crossover esférico*, e *crossover simplex* descritos em MICHALEWICZ & SCHOENAUER (1996). Veja também BÄCK *et al.* (2000a).

5.3 Permutações

- A representação por permutações possui a característica de que os operadores de crossover mais simples falham na geração de um indivíduo que também seja uma permutação.
- Sendo assim, operadores específicos para permutações devem ser propostos. Exemplos:
  - *Crossover OX*:

P1: A B | C D E F | G H I
P2: f h | d a b c | i g e
D1: a b | C D E F | i g h
D2: H I | d a b c | E F G

- *Partially Mapped Crossover PMX* (GOLDBERG & LINGLE, 1985):

P1: A B | C D E | F G
P2: c f | e b a | d g
D1: a f | C D E | b g
D2: C D | e b a | F G

- *Crossover de ordem 2* (SYSWERDA, 1991):

P1: A B C D **E** F **G**
P2: c **f** g **b a** d e
D1: f b C D E a G
D2: c f E b a d G

- *Crossover de posição* (SYSWERDA, 1991):

P1: A B **C D E** F **G**
P2: c f **g b a** d **e**
D1: f b C D E a G
D2: C D g b a F e

- *Crossover de máxima preservação MPX* (MÜHLENBEIN, 1991):

P1: A | B C D | E F G
P2: c | f g b | a d e
D1: e B C D f g a
D2: E f g b A C D

5.4 Máquinas de Estado Finito

- Embora na versão inicialmente proposta para a PE a recombinação entre duas MEFs não fosse empregada, alguns autores propuseram mecanismos para recombinar partes de máquinas de estado finito. Exemplos:
  - Troca de estados entre MEFs, ou seja, para um dado estado, troca-se o símbolo de saída e a transição de próximo estado para cada entrada.

5.5 Árvores

- Árvores sintáticas, da forma como empregadas na PG, requerem operadores de recombinação que garantam a geração de descendentes válidos. Ou seja, a árvore deve possuir apenas símbolos terminais em suas folhas e apenas símbolos não-terminais em seus nós internos (nós de grau maior do que 1). Além disso, cada nó interno da árvore deve possuir o número correto de sub-árvores, uma para cada argumento da função ali executada.
- CRAMER (1985) definiu o operador que hoje é padrão para a recombinação de árvores sintáticas: o crossover de sub-árvores. Operações de encapsulamento impedem que certas subárvores sejam quebradas para a realização de recombinação.

5.6 Recombinações complementares

- Buscando promover habilidades distintas e complementares na exploração do espaço de busca, pode-se conceber operadores que expressam comportamentos alternativos, como exemplificados a seguir para o caso de codificação binária:
  - ✓ Conciliador: R(A [0 **0 1** 0 1], B [1 **0 1** 1 0]) → [? **0 1** ? ?] → yc
  - ✓ Rebelde: R(A [0 0 1 0 1], B [**1** 0 1 **1 0**]) → [**1** ? ? **1 0**] → yr
  - ✓ Bajulador: R(A [**0** 0 1 **0 1**], B [1 0 1 1 0]) → [**0** ? ? **0 1**] → yb
- As posições marcadas com "?" podem assumir quaisquer valores binários.
- Repare que algoritmos evolutivos que empregam recombinação geralmente utilizam operadores com aspecto predominantemente conciliador, de modo que $dist(yc, A) \leq dist(A, B)$, para todo yc.

- Repare também que, para yr = [**1** 1 0 **1** **0**], resulta *dist*(yr, A) = 5. Então, em geral, os comportamentos "rebelde" e "bajulador" não satisfazem a relação de distância verificada para o operador conciliador.
- Logo, o emprego apenas de operadores conciliadores pode implicar em perda de diversidade da população (MOSCATO & COTTA, 2001).

6 A Função de Fitness

- O processo de avaliação de indivíduos em um algoritmo evolutivo começa com a definição de uma função objetivo: $f : A_x \rightarrow \Re$, onde A_x é o espaço de atributos das variáveis.
- A função de fitness $\phi : A_x \rightarrow \Re_+$ faz um mapeamento dos valores nominais da função objetivo em um intervalo não-negativo.
- A função de fitness é geralmente uma composição da função objetivo com uma função de escalonamento: $\phi(a_i(t)) = g(f(a_i(t)))$, onde $a_i(t) \in A_x$.

- O mapeamento acima faz-se necessário caso se deseje <u>minimizar</u> a função objetivo, uma vez que valores maiores de fitness corresponderão a valores menores da função objetivo. Exemplos de funções objetivo escalonadas:
  - $\phi(a_i(t)) = f_{max} - f(a_i(t))$, caso o ótimo global seja conhecido
  - $\phi(a_i(t)) = f_{max}(t) - f(a_i(t))$, onde $f_{max}(t)$ é o máximo até a iteração t
  - $\phi(a_i(t)) = 1 / [1 + f(a_i(t)) - f_{min}(t)]$, onde $f_{min}(t)$ é o mínimo até a iteração t
  - $\phi(a_i(t)) = 1 / [1 + f_{max}(t) - f(a_i(t))]$, onde $f_{max}(t)$ é o máximo até a iteração t
- As duas últimas funções acima retornam o valor do fitness normalizado entre (0,1].

6.1 Fitness Ajustado

- O fitness ajustado é calculado a partir do fitness escalonado:
  - $\phi'(a_i(t)) = 1 / [1 + \phi(a_i(t))]$, onde $\phi(a_i(t))$ é o fitness escalonado de $a_i(t)$.
- $\phi'(a_i(t)) \in [0,1]$.

- O fitness ajustado apresenta o benefício de salientar pequenas diferenças no valor escalonado do fitness.

6.2 Fitness Normalizado

- Se o método de seleção a ser empregado é do tipo proporcional ao fitness, então o fitness a ser empregado deve ser normalizado:
  - $\phi_n(a_i(t)) = \phi'(a_i(t)) / \sum_j \phi'(a_j(t))$, onde $\phi'(a_i(t))$ é o fitness ajustado de $a_i(t)$.
- O fitness normalizado apresenta as seguintes características:
  - $\phi_n(a_i(t)) \in [0,1]$;
  - é maior para melhores indivíduos; e
  - $\sum_i \phi_n(a_i(t)) = 1$.

6.3 Escalonamento do Fitness (Fitness Scaling)

- Devido à pressão seletiva, a população tende a ser dominada por aqueles indivíduos de maior fitness.

- Nestes casos, as funções de fitness descritas acima tendem a definir valores similares de fitness a todos os membros da população.
- Para solucionar este problema, métodos de escalonamento de fitness foram propostos com o objetivo de acentuar pequenas diferenças nos valores de fitness, mantendo assim os efeitos da pressão seletiva.
- GREFENSTETTE (1986) propôs uma função de fitness como sendo uma transformação linear variante no tempo da função objetivo:
  - $\phi(a_i(t)) = \alpha.f(a_i(t)) - \beta(t)$, onde $\alpha = 1$ para problemas de maximização e $\alpha = -1$ para problemas de minimização, e $\beta(t)$ representa o pior valor encontrado nas últimas gerações.
- GOLDBERG (1989) propôs o *sigma scaling* baseado na distribuição dos valores objetivo da população atual:
  - $\phi(a_i(t)) = f(a_i(t)) - (f_{av}(t) - c\sigma_f(t))$, caso $f(a_i(t)) > (f_{av}(t) - c\sigma_f(t))$, e
  - $\phi(a_i(t)) = 0$, caso $f(a_i(t)) \leq (f_{av}(t) - c\sigma_f(t))$.

- Onde $f_{av}(t)$ é o valor médio da função objetivo da população atual, $\sigma_f(t)$ é o desvio-padrão dos valores da função objetivo da população atual, e c é uma constante.

6.4 Fitness para permutações

- Cabe um comentário referente ao cálculo do valor de fitness de soluções candidatas em permutações, como no caso do caixeiro viajante.
- É possível trabalhar com as coordenadas geográficas das cidades e, a partir disso, obter a matriz de distâncias par-a-par entre as cidades, ou então partir direto de uma matriz de distâncias par-a-par.
- Assim, a existência de uma matriz de distância não requer que tenha havido um cálculo explícito de distância a partir de valores absolutos de coordenadas.
- Quando as coordenadas estão disponíveis, cabe apontar que elas não precisam estar restritas ao espaço de duas dimensões.

7 Mecanismos de Seleção

- A seleção é um dos principais operadores dos algoritmos evolutivos cujo objetivo é selecionar os "melhores" indivíduos da população em detrimento dos "piores". A seleção pode ocorrer por várias razões, sendo a mais comum a definição de indivíduos que irão se reproduzir, gerando descendentes.
- A identificação da qualidade de um indivíduo é baseada em seu valor de fitness.
- A ideia básica é privilegiar (aumentar a probabilidade de seleção de) indivíduos com valores relativos mais elevados de fitness.
- Os operadores de seleção podem ser *determinísticos* ou *probabilísticos*.

7.1 Teoria da Pressão Seletiva

- Os operadores de seleção são caracterizados por um parâmetro conhecido como *pressão seletiva*, que relaciona o tempo de *dominância* (*takeover*) do operador.

- O tempo de dominância é definido como sendo a velocidade para que a melhor solução da população inicial domine toda a população através da aplicação isolada do operador de seleção (BÄCK, 1994; GOLDBERG & DEB, 1991).
- Se o tempo de dominância de um operador é grande, então a pressão seletiva é fraca, e vice-versa.
- Portanto, a pressão seletiva oferece uma medida de quão "ganancioso" é o operador de seleção no que se refere à dominância de um indivíduo da população.
  - Se o operador de seleção apresenta uma forte pressão seletiva, então a população perde diversidade rapidamente.
- Sendo assim, para evitar uma rápida convergência para pontos sub-ótimos, é necessário empregar populações com dimensões elevadas e/ou operadores genéticos capazes de introduzir e/ou manter a diversidade populacional.

  - Por outro lado, operadores genéticos com estas características tornam a convergência do algoritmo lenta nos casos em que a pressão seletiva é fraca. Isto permite uma maior exploração do espaço de busca.
- A discussão acima sugere que o sucesso na aplicação de um algoritmo evolutivo depende do mecanismo de seleção empregado, que, por sua vez, irá depender dos operadores genéticos e outros parâmetros escolhidos para o algoritmo.

7.2 Seleção Proporcional ao Fitness

- O GA clássico utiliza a *seleção proporcional ao fitness*, geralmente implementado com o algoritmo da roleta (*roulette wheel*).
- O *roulette wheel* atribui a cada indivíduo de uma população uma probabilidade de passar para a próxima geração proporcional a seu fitness normalizado, ou seja, o fitness medido em relação à somatória do fitness de todos os indivíduos da população.

- Assim, quanto maior o *fitness* de um indivíduo, maior a probabilidade dele passar para a próxima geração.
- Observe que a seleção de indivíduos por *roulette wheel* permite a perda do melhor indivíduo da população.

7.3 Seleção Por Torneio

- É um dos mais refinados processos de seleção, por permitir ajustar a pressão seletiva.
- Para se selecionar N indivíduos, realizam-se N torneios envolvendo q indivíduos em cada torneio, escolhidos sem levar em conta o fitness e com reposição (um indivíduo pode aparecer mais de uma vez num mesmo torneio). Vence cada torneio aquele que apresentar o maior fitness (comparado ao de seu(s) oponente(s) no torneio).
- Para propósitos práticos, $q \geq 10$ conduz a uma forte pressão seletiva, enquanto valores de q entre 3 e 5 levam a uma fraca pressão seletiva.

  - Para $q = 1$, nenhuma seleção está sendo feita.
  - Para $q = 2$, tem-se o chamado torneio binário.

7.4 Seleção Baseada em Rank

- Este mecanismo utiliza apenas as posições dos indivíduos quando ordenados de acordo com o *fitness* para determinar a probabilidade de seleção.
- A seleção por rank simplifica o processo de mapeamento do objetivo para a função de fitness: $\phi(a_i(t)) = \alpha f(a_i(t))$, onde $\alpha = +1$ para problemas de maximização e $\alpha = -1$ para problemas de minimização.
- O rank também elimina a necessidade de escalonamento do fitness, uma vez que a pressão seletiva é mantida mesmo que os valores de fitness dos indivíduos sejam muito próximos um do outro, o que normalmente ocorre após muitas gerações.
- Podem ser usados mapeamentos lineares ou não-lineares para determinar a probabilidade de seleção.

- *Ranking linear*: a probabilidade de seleção de um indivíduo é proporcional ao seu rank, onde o indivíduo menos apto (com menor fitness) possui rank 0 e o rank do melhor indivíduo é $N - 1$.
  - Sejam β_{rank} e α_{rank} a quantidade esperada de descendentes do melhor e do pior indivíduo da população a cada geração. A probabilidade de seleção de um indivíduo é dada por:
  - $p(i) = [\alpha_{rank}+[\text{rank}(i)/(N-1)].(\beta_{rank}-\alpha_{rank})]/N$,
- *Ranking não-linear*: a probabilidade de seleção de um indivíduo é uma função não-linear de seu rank. Exemplos:
  - $p(i) = \alpha(1-\alpha)^{N-1-\text{rank}(i)}$, onde $\alpha \in (0,1)$.
  - $p(i) = (1-\exp(-\text{rank}(i)))/c$, onde c é um fator de normalização.

7.5 Seleção de Boltzmann

- Os mecanismos de seleção de Boltzmann controlam termodinamicamente a pressão seletiva utilizando princípios de simulated annealing (SA).
- Os algoritmos evolutivos com seleção de Boltzmann podem ser vistos como extensões paralelas do algoritmo de SA (MAHFOUD & GOLDBERG, 1995).
- A ideia básica da seleção de Boltzmann é utilizar uma distribuição de Boltzmann-Gibbs como mecanismo de competição entre indivíduos (ver Figura 16):

$$P(x') = [1+\exp(f(x)- f(x'))/T]^{-1},$$

onde T é a temperatura do sistema, x e x' são os dois indivíduos que estão competindo para serem selecionados, e $f(\cdot)$ é o valor da função de fitness.

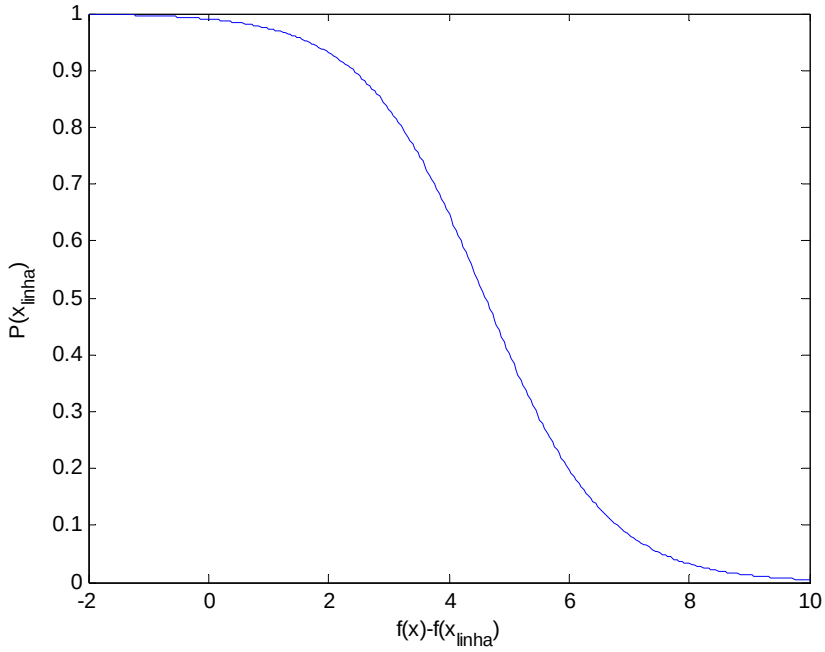

Figura 16 – Probabilidade de escolha de *x'*, segundo uma distribuição de Boltzmann-Gibbs e dado *x*.

7.6 Seleções Bi-Classista e Elitista

- Na seleção bi-classista, são escolhidos os *b*% melhores indivíduos e os w% piores indivíduos da população. O restante (100–(*b*+*w*))% é selecionado aleatoriamente, com ou sem reposição.
- O elitismo consiste em um caso particular de seleção bi-classista na qual um ou mais dos melhores indivíduos da população é sempre mantido e nenhum dos piores indivíduos é selecionado. Ou seja, $b \neq 0$ e $w = 0$.

8 Abordagens baseadas em população

- **Abordagem Michigan** – a população como um todo é a solução para o problema.
- **Abordagem Pittsburgh** – cada elemento da população corresponde a uma solução do problema.

9 Definição da População Inicial

- O método mais comum utilizado na criação da população é a inicialização aleatória dos indivíduos. Se algum conhecimento inicial a respeito do problema estiver disponível, ele pode ser utilizado na inicialização da população.
- Por exemplo, se é sabido que a solução final (supondo codificação binária) vai apresentar mais 0's do que 1's, então esta informação pode ser utilizada, mesmo que não se saiba exatamente a proporção.
- Em problemas com restrição, deve-se tomar cuidado para não gerar indivíduos inválidos já na etapa de inicialização.

10 População estruturada

- Em suas versões tradicionais, os algoritmos evolutivos são concebidos com população não-estruturada, onde todos os indivíduos pertencem a um mesmo e

único nível hierárquico, permitindo que todos os indivíduos tenham as mesmas condições de reprodução e aplicação de operadores, guiadas apenas pelo fitness.

- Por outro lado, já existem várias formas de estruturação da população, incluindo divisão em ilhas, inclusão de vizinhanças restritas (como vizinhança em redes complexas) e inserção de níveis hierárquicos. Será tratada aqui uma proposta para este último caso.

10.1 Hierarquia

- Em FRANÇA *et al.* (2001), foi proposta a organização da população em uma árvore ternária. O objetivo é permitir que a busca evolutiva se dê com base em um número bem reduzido de indivíduos, quando comparado com o número sugerido para populações não-estruturadas, conforme explorado em GONZALÉZ (2011), dentre outras aplicações bem-sucedidas.

- A Figura 17 a seguir mostra uma população organizada numa árvore ternária de 3 níveis, contendo 13 indivíduos. Pode-se admitir que a população está organizada em subgrupos de 4 indivíduos, um líder e três seguidores ou subordinados.

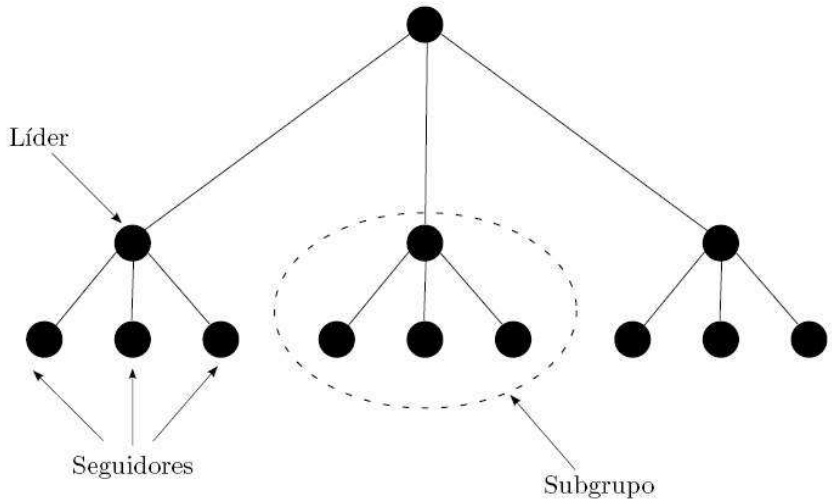

Figura 17 – População estruturada em 3 níveis hierárquicos (extraída com autorização de GONZALÉZ (2011))

- Em cada subgrupo, o líder é sempre o indivíduo com melhor fitness. Indivíduos seguidores que, porventura, superem o fitness do líder são transferidos para a liderança do subgrupo. É evidente que a árvore não precisa ser ternária, embora esta seja a configuração mais adotada em aplicações práticas.
- O número de indivíduos na população corresponde ao número de nós na árvore completa. Para a árvore ternária com 3 níveis da Figura 17, são 13 indivíduos. Aumentando um nível na árvore, resultam 40 indivíduos.
- A hierarquia faz com que indivíduos dos níveis superiores sejam mais bem adaptados que os que ocupam os níveis inferiores.
- O cruzamento de indivíduos neste tipo de população só acontece entre o líder e um dos seguidores. Para conservar a hierarquia dentro da população, faz-se necessário um reordenamento a cada iteração, caso algum seguidor supere o fitness do líder correspondente, como já mencionado. No caso da Figura 17, a população é formada por um conjunto de 4 subgrupos, com 3 níveis e 13 indivíduos no total.

- Este tipo de estruturação da população permite trabalhar com um número reduzido de indivíduos, enquanto introduz algumas características multipopulacionais. Por exemplo, a migração de características genéticas só acontece através de subgrupos do nível superior da população e nunca através de indivíduos do mesmo nível hierárquico. Além disso, dada a existência de múltiplos operadores genéticos, um subconjunto de operadores pode ser alocado a cada subgrupo, enfatizando ainda mais a natureza multipopulacional da abordagem.
- A versão apresentada na Figura 17 é uma simplificação da versão original de França *et al.* (2001), onde em cada nó da árvore existem dois indivíduos, um chamado solução *pocket* e outro chamado solução corrente.

11 Evolução e cardinalidade do espaço de busca

- No Tópico 5 deste curso, será demonstrado que o número de árvores binárias com raiz é dado por $\frac{(2N-3)!}{2^{N-2}(N-2)!}$, onde N é o número de folhas.

- Em uma aplicação envolvendo árvores filogenéticas, com a codificação em matriz binária apresentada neste tópico do curso, e empregando os operadores de mutação descritos também neste tópico do curso, é possível apresentar uma ideia do quanto cresce o custo computacional do processo evolutivo com o aumento da cardinalidade do espaço de busca.
- Esses resultados foram obtidos há 10 anos atrás em PRADO (2001), empregando um hardware bastante modesto: Pentium II – 300 Mhz – 128 Mbytes de RAM.
- É evidente que o comportamento apresentado nas Figuras 18 e 19 a seguir está vinculado ao tipo de problema, à conformação da superfície de fitness e aos operadores empregados.
- Logo, ele deve ser interpretado apenas como um indicativo do comportamento da evolução com o aumento da cardinalidade do espaço de busca.

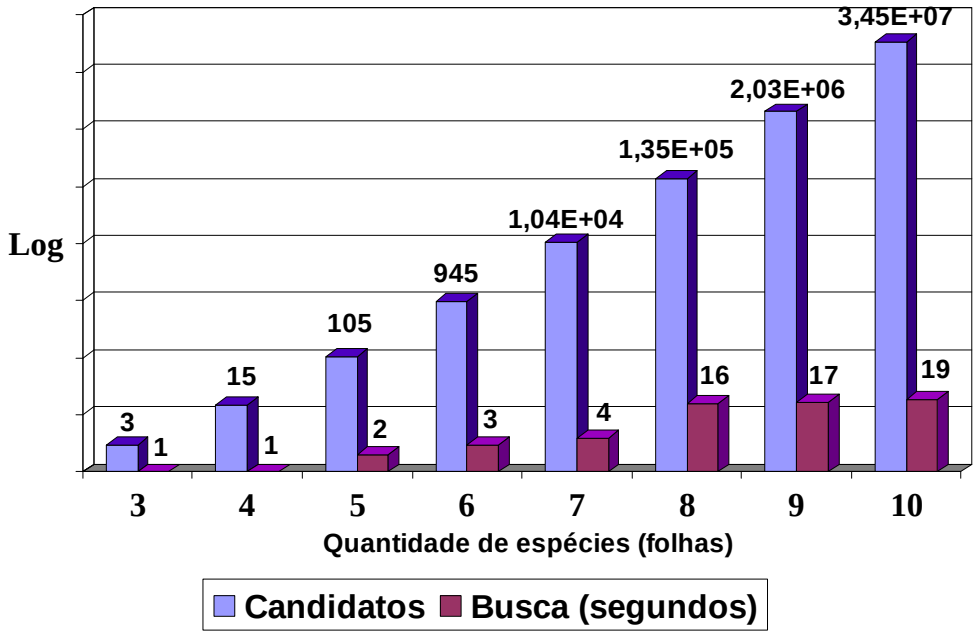

Figura 18 – Custo do processo evolutivo e cardinalidade do espaço de busca (I)

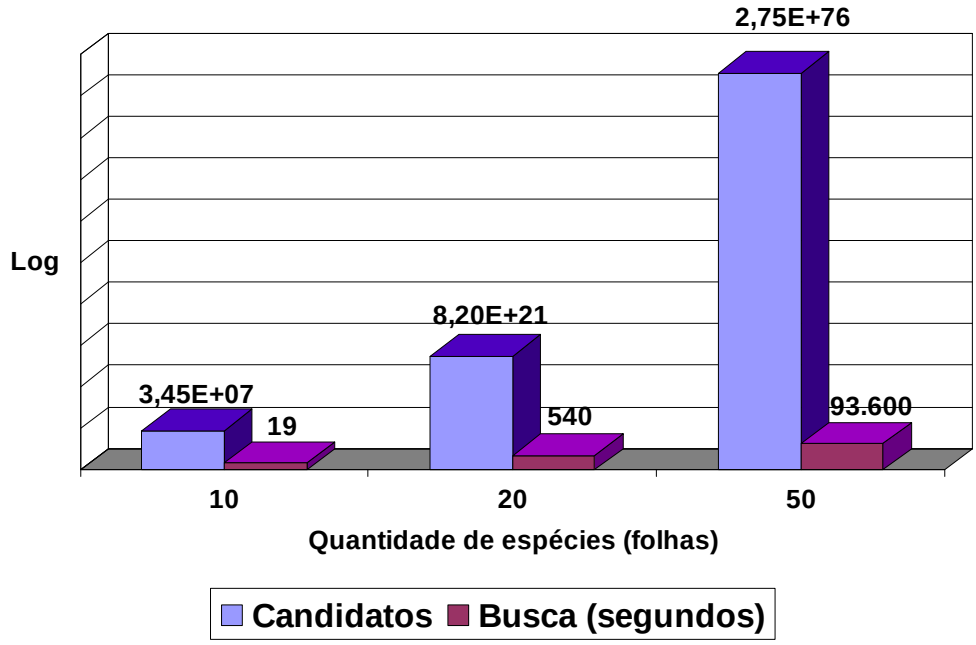

Figura 19 – Custo do processo evolutivo e cardinalidade do espaço de busca (II)

12 Busca local

- As buscas locais geralmente são específicas de cada problema e podem não estar disponíveis em muitas aplicações de computação evolutiva.
- Em algumas circunstâncias, a busca local é tão eficaz que permite chegar muito próximo da solução ótima, mesmo sem considerar estratégias de busca global.
- Geralmente requer uma vizinhança em que a busca vai se dar.
- Pode estar associada a outros algoritmos de otimização não-populacionais, como busca tabu e simulated annealing.
- Se forem custosas, geralmente são aplicadas a apenas alguns indivíduos da população e depois de transcorrido um certo número de gerações.
- Existem três tipos principais:
    1. *Best improvement;*
    2. *First improvement;*
    3. *Best improvement with limited resources.*

12.1 Buscas locais para o problema do caixeiro viajante

- Para efeito ilustrativo, serão consideradas aqui buscas locais para o problema do caixeiro viajante. As subseções a seguir estão baseadas em (DE FRANÇA, 2005).

12.1.1 Algoritmo *2-opt*

- Este é o método de busca mais simples para o problema do caixeiro viajante, sendo a vizinhança $N(S)$ da solução S definida pelo conjunto de soluções que podem ser alcançadas através da permutação de duas arestas não-adjacentes em S. Este movimento é chamado de 2-*interchange* e é descrito na Figura 20.
- Note que, dependendo do tipo de implementação de busca local, todas as trocas possíveis são efetuadas antes de ser tomada a decisão, ou pelo menos até que se encontre a primeira troca que melhore a solução, se essa troca existir.

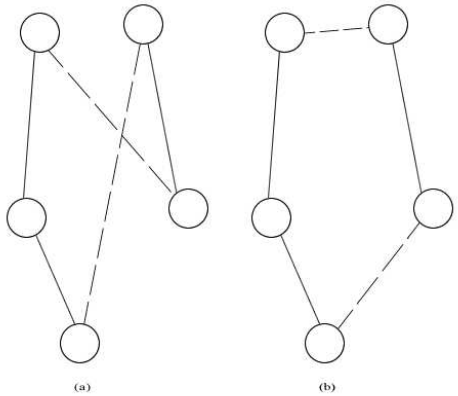

Figura 20 – Exemplo de uma operação 2-*interchange*. (a) Antes da troca. (b) Depois da troca

12.1.2 Algoritmo *k-opt*

- Este método nada mais é do que a generalização do anterior, ou seja, a vizinhança $N(S)$ da solução S é definida pelo conjunto de soluções obtidas através da permutação de até k arestas.

- O número escolhido para k definirá o tamanho da vizinhança e, consequentemente, o desempenho do algoritmo. Um valor baixo de k definirá uma vizinhança pequena, mas também uma menor chance de encontrar uma solução melhor. Por outro lado, um valor muito alto de k resulta em grandes chances de encontrar a solução ótima global, mas também resulta em uma vizinhança muito grande, tornando o algoritmo computacionalmente intratável. Geralmente, um valor de $k \leq 3$ é utilizado.

12.1.3 Algoritmo *Lin-Kernighan*

- Desenvolvido por LIN & KERNIGHAN (1973), é atualmente a mais conhecida e utilizada busca local para o problema do caixeiro viajante. Este método se baseia no *k-opt*, mas com uma diferença: o valor de k passa a ser variável. Ou seja, além de determinar qual a melhor troca, este algoritmo também determina qual o melhor k para aquela iteração.

- O algoritmo busca a sequência de arestas $\{x_1, x_2, ..., x_k\}$ que, quando trocadas pelas arestas $\{y_1, y_2, ..., y_k\}$, retornam um caminho factível e de menor custo. Uma descrição mais detalhada é apresentada no Algoritmo 1 e na discussão a seguir.

```
[caminho] = Função lin-kernighan(),
     tour = caminho_aleatório();
     G* = 0;
     i = 1;
     n2i-1 = sorteie_nó();
     n2i = escolha_nó(n2i-1);
     x[i] = (n2i-1,n2i);
     n2i+1 = escolha_nó_troca(n2i);
     y[i] = (n2i,n2i+1);
      g[i] = gain(x_aresta[i], y_aresta[i]);
      G = g[i];
     Repita
           i = i + 1;
           n2i = escolha_nó(n2i-1);
           x[i] = (n2i-1,n2i);
           n2i+1 = escolha_nó_troca(n2i);
           y[i] = (n2i,n2i+1);
      g[i] = gain(x_aresta[i], y_aresta[i]);
           G = G + g[i];
           Se não existem mais trocas de arestas que melhorem,
                 Se G > G*,
                       G* = G;
                       caminho = troca(caminho,x_aresta,y_aresta);
                 Fim
           Fim
     Até que G ≤ G*;
Fim
```

Algoritmo 1 – Algoritmo de Lin-Kernighan para o caixeiro viajante

- O algoritmo inicia construindo um caminho aleatório e sorteando um nó n_1 por onde irão começar as trocas. Em seguida, é escolhida uma aresta contendo o nó inicial $x_1 = (n_1,n_2)$ e uma aresta de troca que parta de n_2 e gere um ganho positivo $y_1 = (n_2,n_3)$. Escolhida a primeira aresta de troca, começa então o processo iterativo com $i = 2$, onde a cada passo uma aresta é escolhida contendo o último nó escolhido na iteração anterior $x_i = (n_{2i-1},n_{2i})$, e escolhe-se uma aresta partindo de n_{2i} tal que algumas condições sejam satisfeitas:

  1. Se a aresta (n_1,n_{2i}) é criada, então um caminho completo é formado;
  2. A aresta y_i é uma aresta não utilizada contendo o nó n_{2i};
  3. Para garantir a disjunção entre x_i e y_i, x_i não pode ser uma aresta já escolhida para o conjunto y, e y_i não pode ser uma aresta já escolhida pelo conjunto x;
  4. O ganho deve ser positivo;
  5. A aresta y_i deve permitir a quebra de x_{i+1} para possibilitar que na iteração seguinte existam trocas factíveis;
  6. Antes da decisão final da escolha de y_i é verificado se, ao fechar n_{2i} com n_1, gera-se um caminho com custo menor do que o original.

- Terminada a construção do conjunto de arestas originais (x) e o conjunto de arestas de troca (y), o novo caminho é construído e os passos são executados novamente até que não ocorra mais nenhum ganho. Este processo é ilustrado passo a passo nas Figuras 21 a 25.

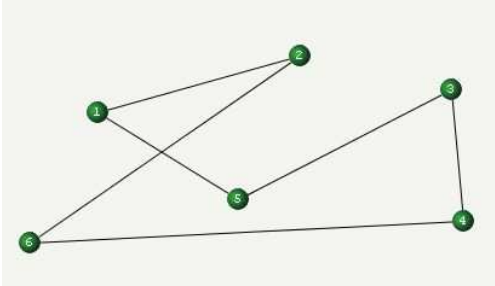

Figura 21 – Rota inicial não ótima. O primeiro vértice escolhido aleatoriamente é o "5".

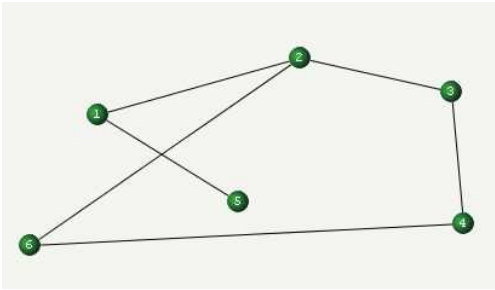

Figura 22 – Após a escolha do vértice inicial a aresta (5,3) é incluída na lista *x* de vértices a serem removidos e a aresta (3,2) é incluída na lista *y* de vértices a serem incluídos.

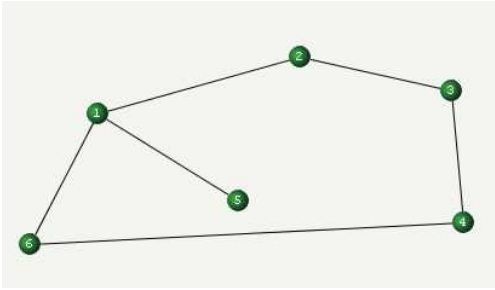

Figura 23 – Em seguida, partindo do vértice "2" é incluído em *x* a aresta (2,6) e em *y* a aresta (6,1).

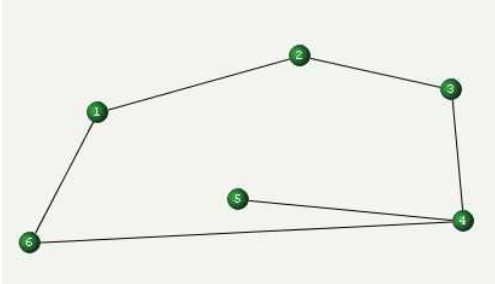

Figura 24 – A lista *x* recebe a aresta (1,5) e a lista *y* a aresta (5,4).

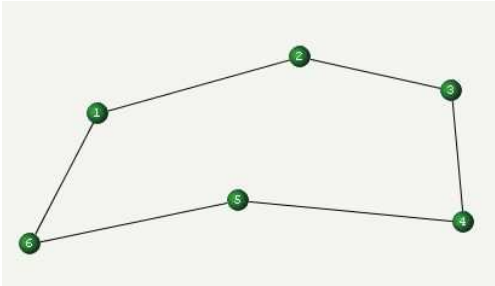

Figura 25 – Por último, a aresta escolhida para remoção é a (4,6) e para inserção a (6,5), terminando assim as escolhas possíveis de arestas e chegando na solução ótima. As listas finais de remoção e inserção de arestas ficam: *x* = {(5,3), (2,6), (1,5), (4,6)} e *y* = {(3,2), (6,1), (5,4), (6,5)}, gerando um movimento 4-opt.

- Este é o melhor algoritmo de busca local conhecido para o problema do caixeiro viajante e gera soluções cujas qualidades diferem de aproximadamente 2% do ótimo, além de ter um custo computacional relativamente baixo (MICHALEWICZ & FOGEL, 2000).

13 Decisões Críticas

- Tipo de codificação?
- Representar toda ou parte da solução?
- Só crossover, só mutação, ou ambos?
- Populações de tamanho fixo ou variável?
- Cromossomos de tamanho fixo ou variável?
- Cromossomos haploides ou diploides?
- Qual critério de parada?
- Que operadores de seleção adotar?

- Usar controle de diversidade?
- Deve-se escalar a função de fitness?
- Usar busca local?
- Usar busca multimodal?
- Usar co-evolução?
- Abordagem Michigan ou Pittsburgh?
- Usar população estruturada?
- Algoritmos genéticos, programação genética, estratégias evolutivas, programação evolutiva?
- Etc.

14 Referências bibliográficas

ACKLEY, D. H. (1987), *A Connectionist Machine for Genetic Hillclimbing*, Kluwer.

ANGELINE, P.J., KINNEAR JR., K.E. (eds.) "Advances in Genetic Programming". volume II, MIT Press, 1996.

BÄCK, T. (1994), "Selective Pressure in Evolutionary Algorithms: A Characterization of Selection Mechanisms", *Proc. Of the 1st IEEE Conf. on Evolutionary Computation*, Orlando, FL, pp. 57-62.

BÄCK, T. (1993), "Optimal Mutation Rates in Genetic Search", *Proc. of the 5th Int. Conf. on Gen. Algorithms*, S. Forrest (ed.), pp. 2-8.

BÄCK, T., FOGEL, D.B. & MICHALEWICZ, Z. (eds.) "Evolutionary Computation 1: Basic Algorithms and Operators", Institute of Physics Publishing, 2000a.

BÄCK, T., FOGEL, D.B. & MICHALEWICZ, Z. (eds.) "Evolutionary Computation 2: Advanced Algorithms and Operators", Institute of Physics Publishing, 2000b.

BERTONI, A. & DORIGO, M. "Implicit Parallelism in Genetic Algorithms", *Artificial Intelligence*, 61(2): 307-314, 1993.

BREMERMANN ET AL. (1966), "Global Properties of Evolution Processes", *Natual Automata and Useful Simulations*, H. H. Pattec et al. (eds.), pp. 3-41.

CRAMER, N.L. "A representation for the adaptive generation of simple sequential programs". *in* J.J. Grefenstette (ed.) *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, 1985.

DE FRANÇA, F. O. "Algoritmos bio-inspirados aplicados à otimização dinâmica", Tese de Mestrado, Faculdade de Engenharia Elétrica e de Computação (FEEC/Unicamp), 2005.

FOGARTY, T. C. (1989), "Varying the Probability of Mutation in the Genetic Algorithm", *Proc. of the 3rd Int. Conf. on Gen. Algorithms*, pp. 104-109.

FOGEL, D. B. "An Introduction to Simulated Evolutionary Computation", *IEEE Transactions on Neural Networks*, 5(1): 3-14, 1994.

FRANÇA, P. M., MENDES, A. S. & MOSCATO, P. A. "A memetic algorithm for the total tardiness single machine scheduling problem", European Journal of Operational Research, vol. 132, pp. 224-242, 2001.

GOLDBERG, D. E. "Messy Genetic Algorithms: Motivation, Analysis, and First Results", *Complex Systems*, 3: 493-530, 1989.

GOLDBERG, D. E. & DEB, K. (1991), "A Comparison of Selection Schemes Used in Genetic Algorithms", *Foundations of Genetic Algorithms*, G. J. E. Rawlins (ed.), Morgan Kaufmann, pp. 69-93.

GOLDBERG, D. E. & LINGLE, R. JR. (1985), "Alleles, Loci, and the Travelling Salesman Problem", *Proc. of the 1st ICGA*, S. Forrest (ed.), pp. 536-542.

GONZÁLEZ, J. F. V. "Estratégias para Redução de Perdas Técnicas e Melhoria das Condições de Operação de Redes de Distribuição de Energia Elétrica", Tese de Doutorado, Faculdade de Engenharia Elétrica e de Computação (FEEC/Unicamp), 2011.

GREFENSTETTE, J. "Optimization of Control Parameters for Genetic Algorithms", *IEEE Trans. on Syst., Man, and Cybernetics*, vol. 16, pp. 122-128, 1986.

HOLLAND, J.H. "Adaptation in Natural and Artificial Systems", University of Michigan Press, 1975.

HOLLAND, J.H. "Adaptation in Natural and Artificial Systems", 2nd edition, The MIT Press, 1992.

KNUTH, R. M. (1973), *The Art of Computer Programming Volume 3: Sorting and Searching*, Addison-Wesley.

KOZA, J.R. "Genetic Programming: On the Programming of Computers by means of Natural Selection", MIT Press, 1992.

LIN, S. & KERNIGHAN, B. (1973), "An Efficient Heuristic Procedure for the Travelling Salesman Problem", *Oper. Res.*, **21**, pp. 498-516.

MAHFOUD, S. & GOLDBERG, D. "Parallel Recombinative Simulated Annealing: A Genetic Algorithm", *Parallel Comput.*, vol. 21, pp. 1-28, 1995.

MICHALEWICZ, Z. "Genetic Algorithms + Data Structures = Evolution Programs", 3rd edition, Springer, 1996.

MICHALEWICZ, Z. & FOGEL, D. B. "How to solve it : Modern heuristics", Springer, 2000.

MICHALEWICZ, Z. & SCHOENAUER, M. "Evolutionary Algorithms for Constrained Parameter Optimization Problems", *Evolutionary Computation*, 4(1): 1-32, 1996.

MOSCATO, P.A. & COTTA, C. "A Gentle Introduction to Memetic Algorithms". *in* Glover, F. & Kochenberger, G. (Ed.). Handbook of Metaheuristics, Chapter 5, Kluwer Academic Publishers, 2001.

MÜHLENBEIN, H. (1992), "How Genetic Algorithms Really Work: I Mutation and Hillclimbing", *PPSN*, **2** pp. 15-25.

MÜHLENBEIN, H. (1991), "Evolution in Time and Space – the Parallel Genetic Algorithm", *Foundations of Genetic Algorithms*, G. J. E. Rawlins (ed.), Morgan-Kaufmann.

PRADO, O. G. "Computação Evolutiva Empregada na Reconstrução de Árvores Filogenéticas", Tese de Mestrado, Faculdade de Engenharia Elétrica e de Computação (FEEC/Unicamp), 2001.

SCHAFFER, J. D., & MORISHIMA, A. (1987), "An Adaptive Crossover Distribution Mechanism for Genetic Algorithms", *Proc. of the 2nd ICGA*, J. J. Grefensttete (ed.), pp. 36-40.

SYSWERDA, G. (1991), "Schedule Optimization Using Genetic Algorithms", *Handbook of Genetic Algorithms*, L. Davis (ed.), pp. 332-349.

SYSWERDA, G. "Uniform Crossover in Genetic Algorithms", *in* Schaffer, J.D. (ed.), *Proceedings of the Third International Conference on Genetic Algorithms*, Morgan Kaufmann Publishers, pp. 2-9, 1989.

WRIGHT, A. H. 1994, "Genetic Algorithms for Real Parameter Optimization", *Foundations of Genetic Algorithms*, G. Rawlins (ed.), Morgan kaufmann, pp. 205-218.

<u>Observação</u>: Parte dessas notas de aula foi baseada em material gerado no ano de 2002 por Fernando J. Von Zuben e Leandro Nunes de Castro, quando ambos ministraram a disciplina de IA707 na FEEC/Unicamp.