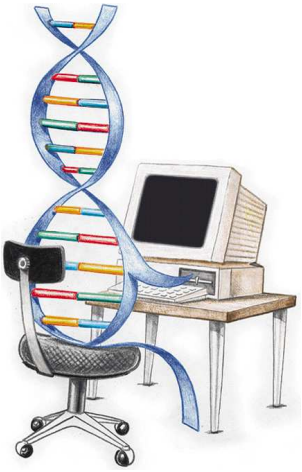
Pontifícia Universidade Católica do Paraná
Programa de Pós-Graduação em Informática Aplicada

Computação Evolutiva



Luiz Eduardo S. Oliveira, Ph.D.

2005

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A computação evolutiva é composta por um conjunto de técnicas de otimização estocástica inspiradas no processo evolutivo biológico. Tais técnicas têm recebido crescente interesse nas últimas décadas, devido principalmente a sua versatilidade para a resolução de problemas complexos de otimização. Neste capítulo apresentamos os conceitos básicos necessários para utilizar computação evolutiva na resolução de problemas reais. Cinco grandes áreas da computação evolutiva serão abordadas:

- Algoritmos genéticos.
- Programação evolutiva.
- Estratégias evolutivas.
- Programação genética.
- Inteligência Coletiva.

As técnicas tradicionais de busca e otimização geralmente utilizam regras determinísticas para se deslocarem no espaço de busca. Uma das desvantagens dessa abordagem é a alta probabilidade de ficar preso em um ótimo local. Já as técnicas de computação evolutiva, são baseadas em população de soluções, as quais são reproduzidas a cada época (geração) do algoritmo. Desta maneira, vários máximos e mínimos podem ser explorados simultaneamente, reduzindo assim a probabilidade de ficar preso em um ótimo local, e conseqüentemente encontrando o ótimo global. Essa terminologia está ilustrada na Figura 1.1

Independentemente do paradigma implementado, as ferramentas da computação evolutiva seguem um procedimento similar:

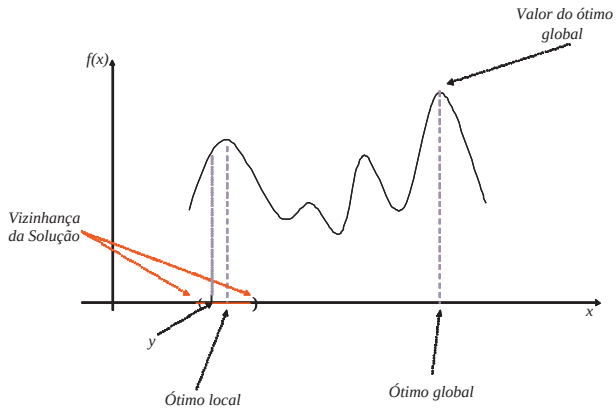


Figura 1.1: Terminologia utilizada em problemas de otimização.

- Inicializar a população.
- Calcular a fitness[1] de cada indivíduo da população.
- Reproduzir os indivíduos selecionados para produzir uma nova população.
- Submeter a população à operações genéticas, tais como cruzamento e mutação.
- Voltar ao ítem 2 até que alguma condição seja satisfeita.

Como o nome sugere, a inicialização consiste em inicializar a população com valores aleatórios. Quando os parâmetros são representados por strings binárias, essa inicialização simplesmente atribui valores zero ou um para cada bit da string (seguindo uma distribuição uniforme). Um procedimento bastante utilizado consiste em inicializar um indivíduo da população com uma solução conhecida, a qual sabe-se que está perto de uma solução ótima.

O valor da fitness é geralmente proporcional ao resultado da função que está sendo otimizada, podendo ainda ser uma combinação dos resultados de várias funções. Em muitos casos, uma pequena porção dos recursos computacionais disponíveis é usada para executar as operações genéticas do algoritmo, sendo a maior parte dos recursos usada para o cálculo da fitness.

A seleção dos indivíduos usados para a reprodução, os quais darão origem a uma nova geração e geralmente baseada no valor da fitness. Quanto maior for a fitness de um

[1] A tradução mais adequada de fitness seria aptidão. Porém nesse texto optamos pelo uso do termo em inglês.

indivíduo, maior a sua probabilidade de ser selecionado para ser um reprodutor. Entretanto, alguns paradigmas, como por exemplo, *Particle Swarm Optimization* (discutido no Capítulo 6), mantém todos os membros da população de geração em geração.

O algoritmo é finalizado quando algum indivíduo alcança uma fitness pré-determinada ou quando o número máximo de iterações é executado.

Em muitos casos, senão na maioria, existe um ótimo global em um ponto do espaço de decisão. Além disso, podem existir ruídos estocásticos bem como caóticos. Algumas vezes o ótimo global que se busca pode mudar dinamicamente em decorrência de influências externas. Em outros casos, existem ótimos locais muito bons. Por esses e outros motivos, muitas vezes não é sensato esperar que um método de otimização encontre o ótimo global (mesmo que ele exista) em um tempo finito. O melhor que pode-se esperar é que o algoritmo encontre uma solução próxima à ótima.

Isso nos leva a *Lei da Suficiência*. Se uma solução é suficientemente boa, rápida e barata, então dizemos que ela é *suficiente*. Na maioria das aplicações reais, buscamos, e estamos satisfeitos com soluções suficientes[2]. Uma pergunta feita freqüentemente é: Qual é o melhor algoritmo da computação evolutiva para problemas de otimização? A resposta pode ser encontrada no *No Free Lunch Theorem* [9], o qual diz que não existe o melhor algoritmo, mas sim que cada algoritmo é eficiente para uma determinado domínio de aplicação. Wolpert e MacReady [26] mostram que todos os algoritmos de otimização tem exatamente o mesmo desempenho quando se faz a média de todas as possíveis funções de custo. Em particular, se o algoritmo A supera o algoritmo B em alguns casos, então existirão outros casos onde o algoritmo B superará A. Isso é conhecido com *No Free Lunch Theorem*.

Nas próximas seções revisaremos as cinco áreas da computação evolutiva: algoritmos genéticos, computação evolutiva, estratégias evolutivas, programação genética e inteligência de enxame. Nosso foco principal serão os algoritmos genéticos, uma vez que os mesmos têm sido os mais citados na literatura. Entretanto, é importante salientar que estratégias híbridas combinando computação evolutiva e outras técnicas de inteligência computacional têm se tornado cada dia mais comuns.

[2]Nestes casos, entende-se por suficientes aquelas soluções que estão de acordo com as especificações do usuário.

Capítulo 2

Algoritmos Genéticos

Os primeiros trabalhos envolvendo algoritmos genéticos (AGs) datam da década de 50. A. Fraser, um pesquisador Australiano, publicou um dos primeiros trabalhos sobre AGs em 1957 [11]. Entretanto a pessoa que pode ser considerada o pai dos AGs é John H. Holland da universidade de Michigan. Seu livro, *Adaptation in Natural and Artificial Systems* [14] foi um dos mais importantes livros publicados sobre esse assunto. Outra pessoa bastante ativa ainda nos dias de hoje é David Goldberg, um ex-aluno de Holland. Seu livro *Genetic Algorithms in Search, Optimization, and Machine Learning* [12] é ainda hoje um dos livros texto sobre AGs mais citados na literatura. Outro autor de um importante livro neste campo é Lawrence Davis. Seu livro, *Handbook of Genetic Algorithms*, é dividido em duas partes: a primeira parte apresenta um compreensivo tutorial, enquanto a segunda parte apresenta vários estudos de casos.

2.1 Principais Conceitos

AGs são algoritmos de busca baseado na idéia da evolução natural. Entretanto, cientistas da computação em engenheiros geralmente ignoram os fundamentos biológicos dos AGs e como eles podem ser usados na interpretação dos resultados. AGs fornecem um mecanismo de busca bastante atrativo que pode ser usado tanto em problemas de classificação como otimização.

AGs trabalham com uma população de indivíduos (também chamados de cromossomos devido a analogia com a evolução natural), os quais representam soluções potenciais para o problema em questão.

Uma vez que números reais podem ser codificados em AGs com representação (cromossomos) binária, a dimensão do problema pode ser diferente da dimensão do cromos-

somo. Um elemento do cromossomo (gene) geralmente corresponde a um parâmetro ou dimensão do vetor numérico. Cada elemento pode ser codificado usando um ou vários bits, dependendo do tipo de representação de cada parâmetro. O número total de bits define a dimensão do espaço de busca.

O pseudo-código de um AG clássico é apresentado a seguir:

Algorithm 1 Algoritmo Genético Clássico

```
1: t ← 0
2: Inicializar População(t)
3: while condição de término não for satisfeita do
4:    t ← t + 1
5:    Seleciona População(t) da População(t − 1)
6:    Cruzamento População(t)
7:    Mutação População(t)
8:    Avaliação da População(t)
9: end while
```

2.2 Um Simples Exemplo

A implementação de um AG clássico é bastante simples. Desta maneira, o uso de um problema também simples parece ser a melhor maneira de apresentar os conceitos básicos dos AGs. Veremos que a implementação de um AG clássico envolve basicamente cópias de strings, trocas de porções de strings e mudanças de bits.

Nosso exemplo didático consiste em encontrar os valores de x que maximizam a função $f(x) = sin(\pi x/256)$, na qual $0 \leq x \leq 255$ e x podendo assumir somente valores inteiros. Essa função é ilustrada na Figura 2.1. Como pode-se observar, o valor que maximiza tal função é 128 ou $\pi/2$. Neste problema, o valor da função e o valor da fitness são idênticos.

Uma única variável está sendo considerada neste problema: x. Utilizaremos uma codificação binária uma vez que a variável x pode assumir valores inteiros somente. Sendo assim, é lógico representar os indivíduos da nossa população com uma string de oito bits. Portanto, a string 00000000 representará o inteiro 0 enquanto a string 11111111 representará o inteiro 255.

O próximo passo consiste em estipular o número de indivíduos da população. Em uma aplicação real, é comum contar com uma população entre algumas dezenas e poucas centenas de indivíduos. Discutiremos esta questão mais adiante. Neste exemplo, utilizaremos uma população de oito indivíduos.

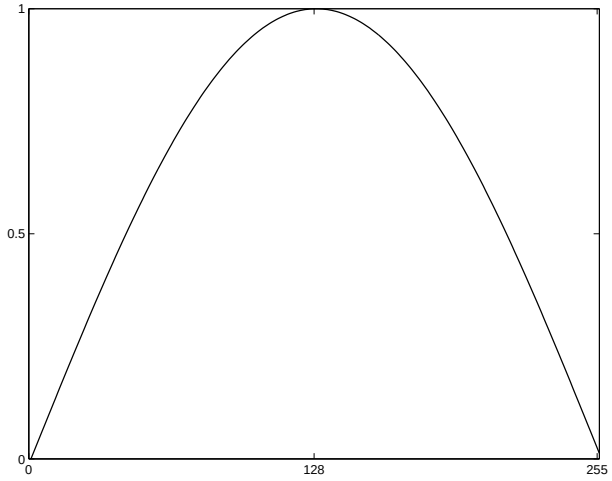


Figura 2.1: Função a ser otimizada no exemplo proposto.

Uma vez determinada o tamanho da população, deve-se inicializar a mesma. Isso é geralmente feito de maneira aleatória. De posse da primeira população, pode-se então calcular a fitness de cada indivíduo. A Tabela 2.1 mostra a população inicial inicializada, o valor de x e o valor da fitness, que neste caso é igual a $f(x)$.

Tabela 2.1: População inicial e fitness do problema exemplo.

Indivíduos	x	$f(x)$	f_{norm}	f_{acm}
10111101	189	0.733	0.144	0.144
11011000	216	0.471	0.093	0.237
01100011	99	0.937	0.184	0.421
11101100	236	0.243	0.048	0.469
10101110	174	0.845	0.166	0.635
01001010	75	0.788	0.155	0.790
00100011	35	0.416	0.082	0.872
00110101	53	0.650	0.128	1.000

Após o cálculo da fitness, o proximo passo é a reprodução, a qual consiste em gerar uma nova população com o mesmo número de indivíduos. Para isso, utiliza-se um processo estocástico que leva em consideração a fitness normalizada de cada indivíduo da população. A normalização é realizada dividindo a fitness de cada individuo pelo somatório de todas as fitness dos indivíduos da população. As fitness normalizadas são então utilizadas num processo conhecido como "Roleta Russa". A roleta contém uma porção para cada indivíduo da população, onde o tamanho da porção reflete a fitness

normalizada do indivíduo (Figura 2.2).

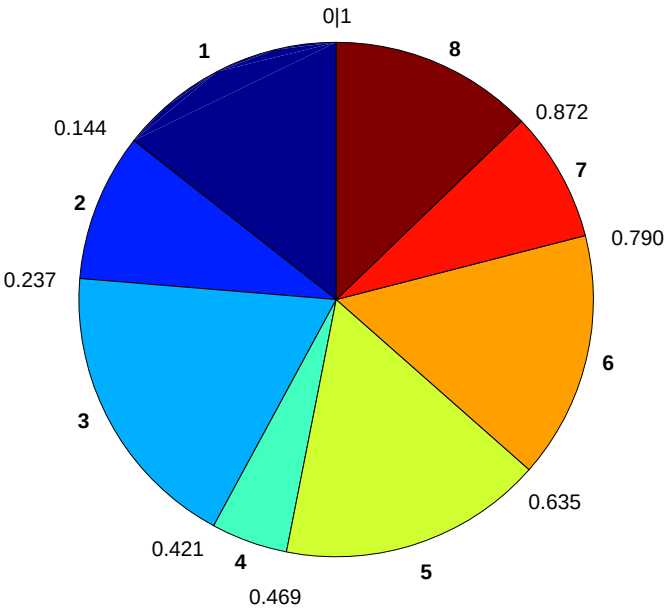


Figura 2.2: Roleta para exemplo proposto.

Rodamos a roleta gerando oito número aleatórios entre 0 e 1. Se o número cair entre 0 e 0.144, o primeiro indivíduo da população é selecionado para a próxima população. Se o número cair entre 0.144 e 0.237, o segundo indivíduo da população é selecionado para a próxima população, e assim por diante. A probabilidade de seleção de um indivíduo é então proporcional a sua fitness. É possível então, embora altamente improvável, que um indivíduo com baixa fitness seja selecionado oito vezes, compondo assim a nova geração (algoritmo estocástico). É muito mais provável que os indivíduos com mais alta fitness sejam selecionados. Os oito número aleatórios gerados no nosso exemplo são: 0.293, 0.971, 0.160, 0.469, 0.664, 0.568, 0.371 e 0.109. Desta maneira, os seguintes indivíduos foram selecionados para a compor a próxima população: 3, 8, 2, 5, 6, 5, 3 e 1 (Tabela 2.2).

A próxima operação é o cruzamento. Esta é a operação que troca porções de strings de dois indivíduos pais. Uma probabilidade (P_c) atribuída ao processo de cruzamento indica, dados dois pais, se o cruzamento ocorrerá ou não. Um valor entre 0.6 e 0.8 é geralmente utilizado como probabilidade de cruzamento. Para o nosso problema exemplo, utilizaremos 0.75.

Afim de preservar alguns bons indivíduos gerados durante a reprodução, nem todos os indivíduos são submetidos à operação de cruzamento. Como no nosso caso estamos

Tabela 2.2: População após reprodução.

Índice	Indivíduos
3	01100011
8	00110101
2	11011000
5	10101110
6	01001010
5	10101110
3	01100011
1	10111101

usando $P_c = 0.75$, o número de indivíduos que sofrerá cruzamento é igual ao total de indivíduos na população $\times P_c$ ($8 \times 0.75 = 6$). Os indivíduos não utilizados no cruzamento serão simplesmente copiados para a nova população.

Sendo assim, seis indivíduos (três pares) são selecionados aleatoriamente para a realização do cruzamento. Os dois restantes serão simplesmente copiados. Por uma questão de simplicidade, selecionamos os seis primeiros indivíduos para o cruzamento. Sobre esses indivíduos aplicamos um cruzamento em dois pontos (as outras técnicas de cruzamento serão discutidas posteriormente), o qual consiste em trocar porções dos indivíduos entre os pontos de cruzamento. Esse processo está ilustrado na Figura 2.3.

Após a operação de cruzamento, o próximo operador é a mutação. Essa operação é utilizada para garantir uma maior varredura do espaço de estados e evitar que o algoritmo genético convirja muito cedo para mínimos locais. A mutação é efetuada alterando-se o valor de um gene de um indivíduo selecionado aleatoriamente com uma determinada probabilidade, denominada probabilidade de mutação (P_m), ou seja, vários indivíduos da nova população podem ter um de seus genes alterado aleatoriamente. O valor de P_m pode variar de acordo com a aplicação, sendo que valores entre 0.001 e 0.01 são freqüentemente utilizados. Isso significa uma inserção entre 0.1 e 1% de genes aleatórios na população. Como no nosso exemplo existem 64 genes (8 indivíduos $\times$ 8 genes), é bem possível que nenhum gene sofra mutação. Desta maneira, vamos considerar a população apresentada na Figura 2.3b como população final (nova geração) após uma iteração do AG.

Note que essa população possui dois indivíduos com fitness > 0.99. A população da Figura 2.3b está pronta para uma nova iteração do AG e assim por diante até que um critério de parada seja satisfeito. No nosso caso, o critério de parada seria $f(x) = 1$,

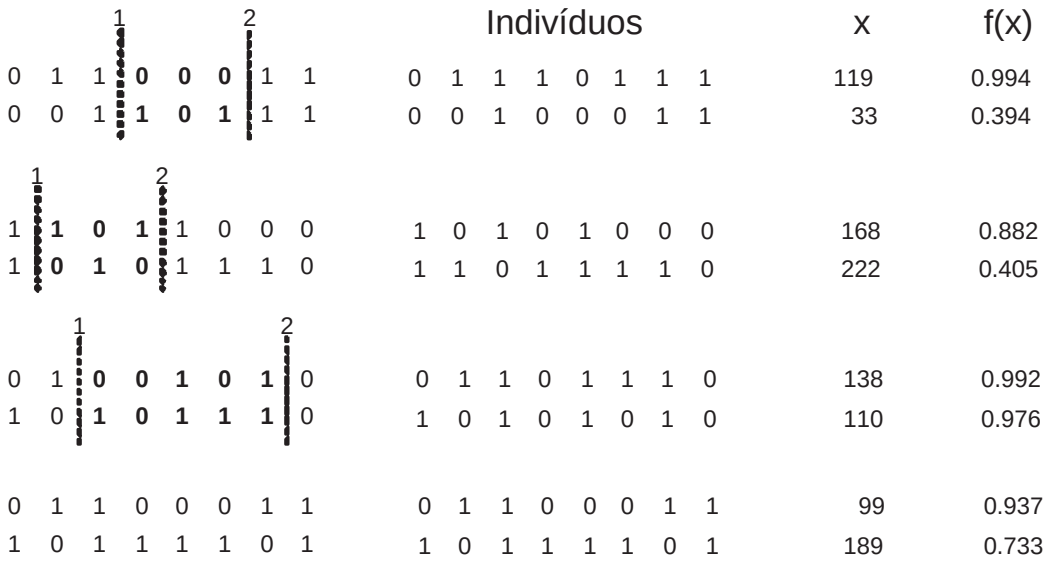


Figura 2.3: (a)População antes do cruzamento indicando os pontos de cruzamento, (b) após cruzamento, (c) valores de x e (d) valores de $f(x)$.

porém pode-se parar o algoritmo assim que ele alcance um número pré-determinado de iterações.

2.3 Entendendo o AG

Agora que uma iteração do AG para o problema exemplo foi concluída, vamos entender como e porque os AGs funcionam. Antes, algumas considerações sobre os tipos de representação, tamanho da população, operadores genéticos e seleção serão apresentados.

2.3.1 Representação das Variáveis

No exemplo apresentado anteriormente, a representação dos valores da variável x foi feita diretamente através da escolha de um intervalo dinâmico [0,255], o qual foi representado por uma string binária de tamanho 8. No algoritmo genético clássico proposto por Holland [14], as soluções candidatas são codificadas em arranjos binários de tamanho fixo. A motivação para o uso da codificação binária vem da teoria dos esquemas (*schemata theory*), a qual veremos mais adiante, utilizada com relativo sucesso para explicar por que os algoritmos genéticos funcionam. Na segunda edição de seu

livro, Holland [15] argumenta que seria benéfico para o desempenho do algoritmo maximizar o paralelismo implícito inerente ao algoritmo genético, e prova que um alfabeto binário maximiza o paralelismo implícito. No nosso exemplo, a função seno é maximizada quando $x = 128$. A representação binária de 128 é 10000000; já a representação de 127 é 01111111. Como podemos notar, para uma pequena variação no valor da fitness, é necessário que todos os bits da string sejam alterados. Ou seja, para se ter uma pequena mudança de valor no espaço real, é necessário uma grande mudança no valor binário. Esse tipo de situação não é o ideal, pois faz com que a busca se torne mais lenta.

Um outro problema com a codificação binária é a precisão. Suponha que desejamos codificar uma variável real que possa assumir qualquer valor no intervalo [2.500,6.500]. Para codificar esse valor (precisão de três casas decimais), será necessário uma string binária de tamanho 12, onde a string 000000000000 representará o valor 2.500. Agora, considere que tenhamos um problema de otimização com 100 variáveis reais. Isso nos leva a uma string binária de 1200 bits. Como se pode notar, quanto maior a precisão, maior deve ser o tamanho da string binária, o que nos leva a um tamanho de população maior, e conseqüentemente uma maior complexidade computacional [12]. Por outro lado, esse tipo de codificação possibilita ao usuário enxergar a população de indivíduos como sendo vetores de valores reais e não strings binárias, e tornando assim, a implementação do AG mais simples. Teoricamente, o alfabeto utilizado na representação do problema pode ser qualquer alfabeto finito, entretanto, o alfabeto binário tem sido o mais utilizado. Mais adiante apresentaremos alguns exemplos com diferentes tipos de alfabetos.

2.3.2 População: Tamanho e Inicialização

O tamanho da população tem relação direta com o espaço de busca, ou seja, quanto maior a população, mais completa será a busca realizada pelo algoritmo. Por outro lado, maior também ser a complexidade computacional envolvida. Geralmente se emprega populações de tamanho variando de 20 a 200 indivíduos, dependendo, como citamos anteriormente, do tamanho do indivíduo (espaço de busca). Pode-se afirmar também que o tamanho da população depende ainda da complexidade do problema em questão.

A inicialização da população é geralmente feita de maneira estocástica, embora em alguns casos seja interessante inserir um ou mais bons indivíduos conhecidos. Deste modo, o algoritmo tende a procurar em algumas regiões promissoras (onde esses indivíduos estão situados).

2.3.3 O Operador de Cruzamento

O operador de cruzamento cria novos indivíduos através da combinação de dois ou mais indivíduos. A idéia intuitiva por trás desta operação é a troca de informação entre diferentes soluções candidatas. O operador de cruzamento mais empregado é o crossover de um ponto, o qual seleciona dois indivíduos (pais) e a partir de seus cromossomos são gerados dois novos indivíduos (filhos). A idéia é a mesma vista anteriormente, porém somente um ponto de corte é gerado aleatoriamente.

Outro tipo de cruzamento é o cruzamento uniforme, onde para cada bit no primeiro filho é decidido (com alguma probabilidade fixa p) qual pai vai contribuir com seu valor para aquela posição. Como o cruzamento uniforme troca bits ao invés de segmentos de bits, ele pode combinar características independentemente da sua posição relativa no cromossomo. No entanto, não há nenhum operador de crossover que claramente apresente um desempenho superior aos demais. Uma conclusão que se pode chegar é que cada operador de cruzamento é particularmente eficiente para uma determinada classe de problemas e extremamente ineficiente para outras.

Os operadores de cruzamento descritos até aqui também podem ser utilizados em cromossomos com codificação em ponto flutuante. Entretanto existem operadores de cruzamento especialmente desenvolvidos para uso com codificação em ponto flutuante. Um exemplo é o chamado cruzamento aritmético [21]. Este operador é definido como uma combinação linear de dois cromossomos: sejam x_1 e x_2 dois indivíduos selecionados para crossover, então os dois filhos resultantes serão $x'_1 = ax_1 + (1-a)x_2$ e $x'_2 = (1-a)x_1 + ax_2$ 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 x_1 e x_2 pertencem à região factível, combinações convexas de x_1 e x_2 serão também factíveis. Desta maneira, garante-se que o cruzamento não gera indivíduos inválidos para o problema em questão.

2.3.4 O Operador de Mutação

O operador de mutação modifica aleatoriamente um ou mais genes de um cromossomo. A probabilidade de ocorrência de mutação em um gene é denominada probabilidade de mutação. Usualmente, são atribuídos valores pequenos para a taxa de mutação. A idéia intuitiva por trás do operador de mutação é criar uma diversidade extra na população, mas sem destruir o progresso já obtido com a busca. Considerando uma codificação binária, o operador de mutação padrão simplesmente troca o valor de

um gene em um cromossomo. Assim, se um gene selecionado para mutação tem valor 1, o seu valor passará a ser 0 após a aplicação da mutação, e vice-versa.

No caso de problemas com codificação em ponto flutuante, os operadores de mutação mais populares são a mutação uniforme e a mutação Gaussiana [21]. O operador para mutação uniforme seleciona aleatoriamente um componente $k \in \{1, 2, \ldots, n\}$ do cromossomo $x = [x_1, x_2, \ldots, x_n]$ e gera um indivíduo $x' = [x'_1, x'_2, \ldots, x'_n]$, onde x'_k é um número aleatório (com distribuição de probabilidade uniforme) amostrado no intervalo [LB, UB] e LB e UB são, respectivamente, os limites inferior e superior para o valor da posição (gene) x_k. Já no caso da mutação Gaussiana, todos os componentes de um cromossomo $x = [x_1, x_2, \ldots, x_n]$ são modificados na forma $x' = x + N(0, \sigma)$ onde $N(0, \sigma)$ é um vetor de variáveis aleatórias Gaussianas independentes, com média zero e desvio padrão σ. Outro operador de mutação, especialmente desenvolvido para problemas de otimização com restrições e codificação em ponto flutuante, é a chamada mutação não-uniforme, destinada a realizar a sintonia fina junto aos indivíduos da população. Este e outros exemplos de operadores de mutação para problemas de otimização numérica podem ser encontrados em [21].

2.3.5 Seleção

O método de seleção mais empregado no AG clássico é a roleta russa, o qual vimos no exemplo apresentado anteriormente. Esse método atribui a cada indivíduo de uma população uma probabilidade de passar para a próxima geração proporcional a sua fitness medida em relação à somatória da fitness de todos os indivíduos da população. Assim, quanto maior a fitness de um indivíduo, maior a probabilidade dele passar para a próxima geração. Desta maneira, a seleção de indivíduos pela roleta russa pode fazer com que o melhor indivíduo da população seja perdido, ou seja, não passe para a próxima geração. Uma alternativa é escolher como solução o melhor indivíduo encontrado em todas as gerações do algoritmo. Outra opção é simplesmente manter sempre o melhor indivíduo da geração atual na geração seguinte, estratégia essa conhecida como seleção elitista [7]. Outro exemplo de mecanismo de seleção é a seleção baseada em *ranking* [1]. Esta estratégia utiliza as posições dos indivíduos quando ordenados de acordo com a fitness para determinar a probabilidade de seleção.

2.3.6 Como o AG Funciona

O principio de funcionamento do AG descrito até então é simples, e envolve basicamente cópia, trocas de porções de strings e alterações de bits. De fato, é surpreendente que com simples operações como essas, um poderoso algoritmo de busca possa ser construído. O teorema que descreve a razão do AG ser eficiente para lidar como problemas de otimização é chamado *schema theorem*. A idéia de esquema permite referir-se de forma compacta às similaridades entre cromossomos. Um esquema (*schema*; plural: *schemata*) é uma estrutura de cromossomo com posições de conteúdo fixo e posições em aberto, tradicionalmente representadas por asteriscos. Sua finalidade é estabelecer "famílias" de cromossomos, caracterizadas originalmente pelos códigos binários imutáveis das posições fixas e pelo comprimento (número de genes) do cromossomo. Os melhores esquemas tendem a perpetuarem-se através das gerações e conseqüentemente suas contribuições para a função objetivo. Os esquemas que servem como base para a construção de futuras gerações são chamados de *building blocks* [15].

Utilizando como exemplo um cromossomo de tamanho 7, pode-se visualizar que um esquema 1*001*1 possui quatro cromossomos 1000111, 1100111, 1000101, 1100101, o que nos leva a concluir que para alfabetos de cardinalidade K e strings de comprimento δ existem $(K+1)\delta$ esquemas, bem como pode-se visualizar que para uma população de N indivíduos podem existir até $N * 2\delta$ esquemas, quando se considera que cada string também é um membro de 2δ (considerando que já se tem o cromossomo). Os fundamentos matemáticos a respeito de esquemas podem ser encontrados em [15, 12].

2.4 AG Multi-Objetivos

A dominância dos cromossomos com alto valor de aptidão nas populações iniciais, pode fazer com que o algoritmo venha a convergir muito rapidamente para um ponto de máximo local no espaço de soluções. Deste fato, surge a necessidade de modificar a função objetivo em prol da contribuição dos indivíduos menos aptos, através de vários mecanismos, dentre os quais podemos citar o procedimento de escala, *ranking*, torneio e roleta russa (os três últimos vistos anteriormente). A idéia básica do procedimento de escala é limitar a competição entre os cromossomos nas iterações iniciais e estimulá-la progressivamente, através de uma mudança (linear em geral) de escala da fitness.

Agora considere que a função a ser otimizada $F(x)$ é composta não somente por um objetivo (f), mas sim por N objetivos (f_i). Vale a pena ressaltar que é isso o que acontece na grande maioria dos problemas reais. Nesses casos, o procedimento

de escala deveria ser aplicado a cada objetivo (f_i) independentemente. Além disso, é necessário utilizar um método que combine os objetivos que estão sendo otimizamos em um único valor de fitness. Um método bastante utilizado nesses casos é a soma ponderada, na qual atribui-se um peso ω_i para cada objetivo f_i e então soma-se tudo para obter o valor da fitness (Equação 2.1)

$$F(x) = \sum_{i=1}^{N} \omega_i f_i(x) \tag{2.1}$$

onde $x \in X$, X é o espaço de objetivos, ω_i o vetor de pesos ($0 \leq \omega_i \leq 1$), e $\sum_{i=1}^{N} \omega_i = 1$. Como podemos notar, além do problema de escala, agora temos que encontrar os pesos adequados a cada função objetivo. Nesses casos vem a tona um outro problema dos AG clássicos: a convergência prematura em função dos pesos escolhidos [4].

Para superar esse tipo de problema, diferentes algoritmos evolutivos baseados no conceito de Pareto têm sido propostos. Antes de discutirmos esses algoritmos, vamos introduzir rapidamente o conceito de dominância. Em um problema multi-objetivo, as soluções podem ser expressas em termos de pontos superiores, ou pontos dominantes. Em um problema de minimização, por exemplo, o vetor $x^{(1)}$ é parcialmente menor que o vetor $x^{(2)}$ ($x^{(1)} \prec x^{(2)}$), quando nenhum valor de $x^{(2)}$ for menor que $x^{(1)}$ e pelo menos um valor de $x^{(2)}$ for maior que $x^{(1)}$. Se $x^{(1)}$ for parcialmente menor que $x^{(2)}$, dizemos que o vetor $x^{(1)}$ *domina* $x^{(2)}$. Desta maneira, qualquer vetor que não seja dominado por algum outro é dito vetor dominante ou não-dominado. As soluções ótimas para um problema de otimização multi-objetivos são as soluções não-dominadas. Elas também são conhecidas como soluções Pareto-ótimas. Por exemplo, considere um problema de minimização com dois objetivos:

$$\begin{cases} \text{Minimize} & f(x) = (f_1(x), f_2(x)) \\ \text{De maneira que} & x \in X(\text{espaço de busca}) \end{cases}$$

uma solução potencial $x^{(1)}$ domina $x^{(2)}$ se e somente se:

$$\begin{aligned} &\forall i \in \{1,2\} : f_i(x^{(1)}) \leq f_i(x^{(2)}) \quad \wedge \\ &\exists j \in \{1,2\} : f_j(x^{(1)}) < f_j(x^{(2)}) \end{aligned} \tag{2.2}$$

A idéia de utilizar o conceito de dominância em algoritmos evolutivos foi primeiramente explorada por Goldberg [12], onde a idéia era utilizar de maneira explicita esse

conceito para determinar a probabilidade de reprodução de cada indivíduo. Basicamente, a idéia consiste em atribuir *rank* 1 para os indivíduos não dominados (frente 1), removê-los da população, encontrar novos indivíduos não dominados, atribuir *rank* 2 (frente 2), e assim por diante. A Figura 2.4 ilustra esse processo.

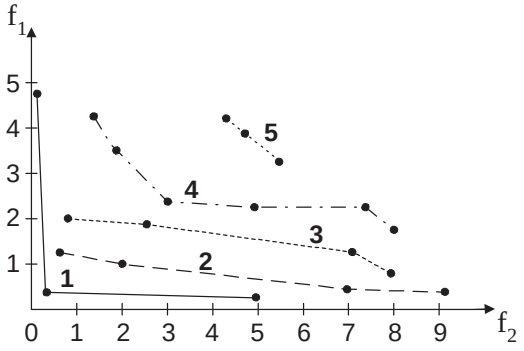


Figura 2.4: Ordenação por frentes não-dominadas

.

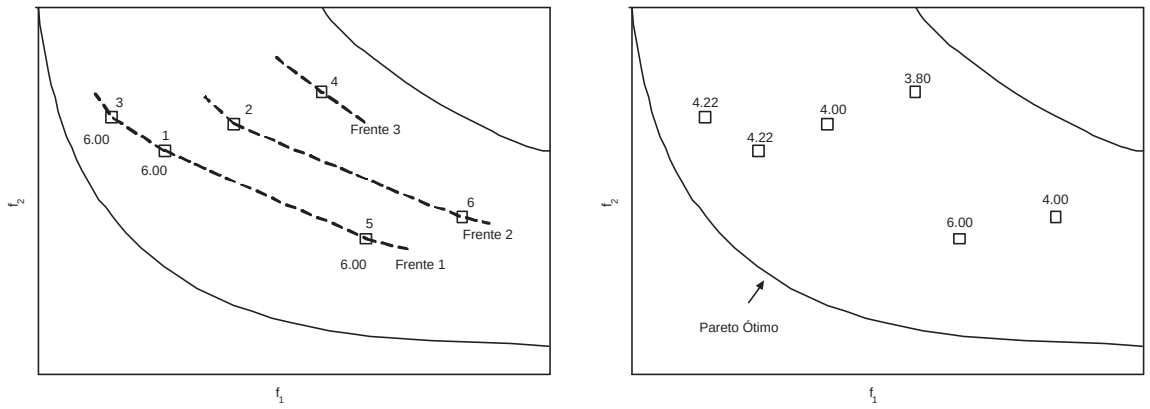
A estratégia proposta por Goldberg atribui a todos os indivíduos da mesma frente, o mesmo valor de fitness, entretanto, isso não garante que o Pareto seja uniformemente distribuído. Quando existem várias soluções ótimas, a população tende a convergir somente para uma delas. Isso se deve a erros estocásticos no processo de seleção. Esse fenômeno é conhecido como *genetic drift*. Para evitar esse tipo de problema, Goldberg e Richardson [13] propuseram o compartilhamento do valor da fitness de alguns indivíduos. A idéia é encontrar alguns nichos populosos e compartilhar a fitness dos mesmos, fazendo assim com que outros nichos menos populosos tenham igual chance de reprodução. Ou seja, a intenção é criar diversidade evitando assim a convergência prematura para algum ponto do espaço de busca.

Vários AGs multi-objetivos têm sido propostos na literatura. Um estudo comparativo entre vários algoritmos foi apresentado por Zitzler et al [27]. Afim de melhor ilustrar como um AG multi-objetivo utiliza os conceitos apresentados anteriormente, apresentaremos aqui o algoritmo proposto por Srinivas e Deb [24, 4], chamado NSGA (*Non-Dominated Sorting Genetic Algorithm*).

O NSGA utiliza o conceito de ranking apresentado na Figura 2.4. Ele difere do AG clássico somente na maneira em que os indivíduos são selecionados para a próxima geração (operador de seleção). Os operadores de cruzamento e mutação permanecem inalterados. Antes da seleção, os indivíduos são ordenados com base na não-dominância.

Todos os indivíduos da primeira frente recebem o mesmo valor de fitness (*dummy fitness*), o que garante a mesma chance de reprodução a todos os indivíduos de uma mesma frente.

Para manter uma certa diversidade na população e fazer que o Pareto seja explorado de uma maneira uniforme, os indivíduos de uma mesma frente têm seu valor de fitness compartilhado através da divisão do mesmo pela quantidade proporcional de indivíduos em torno dele. Após esse compartilhamento, os indivíduos da frente corrente são ignorados temporariamente, e então o restante da população é processado da mesma maneira para encontrar a segunda frente. Os pontos da segunda frente recebem uma fitness menor do que a menor fitness (após o compartilhamento) da primeira frente. Esse processo continua até que todos os indivíduos da população estejam classificados em frentes. A Figura 2.5 ilustra esse processo.



(a) (b)

Figura 2.5: Ordenação da população baseado no conceito de não-dominância: (a) População classificada em três frentes não dominadas e (b) Valores das fitness após compartilhamento.

Podemos observar nesta Figura que os indivíduos "1" e "3" tiveram suas fitness compartilhadas porque eles estão próximos um do outro. Nesse caso, a fitness deles foram reduzidas de 6.00 para 4.22. A fitness que é atribuída aos indivíduos da segunda frente é a menor fitness da primeira frente multiplicada por uma constante k (por exemplo $k = 0.95$ neste exemplo). Desta maneira, os indivíduos da segunda frente recebem uma fitness igual a 4.00 (4.22×0.95). Uma vez que os dois indivíduos da segunda frente não estão próximos um do outro, elas não são compartilhadas. A fitness atribuída à terceira frente é então 3.80 (4.00×0.95).

A população é então reproduzida de acordo com os valores das fitness através de qualquer método de seleção apresentado anteriormente. Levando-se em consideração

que os valores da primeira frente possuem valores de fitness maiores, os mesmos terão maiores chances de reprodução do que o resto da população. A idéia por traz disso é explorar regiões que contenham indivíduos não-dominados. A eficiência do NSGA está na maneira em como vários objetivos podem ser reduzidos a um único valor de fitness (*dummy fitness*).

O compartilhamento em cada frente é realizado através do cálculo de uma função de compartilhamento que considera dois indivíduos da mesma frente:

$$Sh(d(i,j)) = \begin{cases} 1 - \left(\frac{d(i,j)}{\sigma_{share}}\right)^2 & \text{se} \quad d(i,j) < \sigma_{share} \\ 0 & \text{caso contrário} \end{cases} \tag{2.3}$$

onde $d(i,j)$ é a distância Euclidiana entre dois indivíduos i e j da frente corrente e σ_{share} é a distância máxima permitida entre dois indivíduos para que eles se tornem membros do mesmo nicho. Esse parâmetro pode ser calculado da seguinte maneira [4]:

$$\sigma_{share} \approx \frac{0.5}{\sqrt[p]{q}} \tag{2.4}$$

onde q é o número desejado de soluções no Pareto e p é o número de variáveis de decisões do problema em questão. Embora o cálculo de σ_{share} dependa do parâmetro q, Srinivas e Deb [24] demonstraram que $q \approx 10$ funciona bem na maioria dos problemas testados.

2.5 Aplicações

Agora que vimos os conceitos básicos que envolvem os AGs, vamos voltar nossa atenção à algumas das possíveis aplicações desses algoritmos. Como vimos anteriormente, AG é uma ferramenta poderosa para resolver qualquer problema de otimização, seja ele composto de um ou vários objetivos. Portanto, todo e qualquer problema que possa ser formulado na forma de um problema de otimização (com funções a serem minimizadas ou maximizadas) pode utilizar AG como ferramenta para a sua solução. Entretanto, AG são particularmente atrativos quando o espaço de busca é enorme.

2.5.1 Seleção de características

A grande maioria dos sistema inteligentes conta com um classificador, o qual é treinado com um determinado conjunto de características. Durante a construção deste módulo, geralmente não se conhece a priori quais são as melhores características para

a resolução do problema em questão. Sendo assim, diversas características são combinadas e testadas. Porém, a medida que o número de características cresce, o número de combinações possíveis explode na proporção 2^N, onde N é o número de características.

Dentro deste contexto, os AGs têm sido largamente utilizado em problemas de seleção de características, o qual consiste em escolher o melhor subconjunto de características X a partir de um conjunto Z, e portanto pode ser formulado na forma de um problema de otimização. Para mais detalhes sobre seleção de características utilizando AG, consulte [22, 5]

2.5.2 Data Mining

Suponha que uma loja de departamentos mantenha uma base de dados de tudo que seus clientes cadastrados compram. Nesse caso, os AGs se apresentam como uma ferramenta poderosa para a mineração desses dados, ou seja, encontrar algumas regras que não são visíveis devido a grande quantidade de dados.

No contexto de data mining, os indivíduos representam regras de previsão, ou outra forma de conhecimento. A função de fitness mede a quantidade das regras ou conhecimento associado com os indivíduos. Por exemplo, um indivíduo representando uma regra para prever quando um cliente comprará um produto oferecido a ele poderia ser escrita da seguinte maneira:

```
(idade < 18) e (produto = videogame)
```

Os fatores que podem ser medidos em funções de fitness (assumindo que um indivíduo representa uma regra de previsão), podem ser:

- Taxa de acerto da regra: número de tuplas corretamente e erroneamente classificadas por uma regra.
- Generalidade: número de tuplas coberta pela regra.
- Simplicidade: sintática da regra.

Pode-se utilizar pesos para combinar tais regras, ou como visto anteriormente, um algoritmo que leve em consideração múltiplos objetivos.

Dentro deste contexto a mutação poderia ocorrer da seguinte maneira:

```
Antes do cruzamento:
(idade < 18)(produto = videogame)     |(sexo=F)
(idade < 35)(produto = liquidificador)|(sexo=M)

Após o cruzamento:
(idade < 18)(produto = videogame)     (sexo=M)
(idade < 35)(produto = liquidificador)(sexo=F)
```

Como visto até então, a mutação simplesmente muda o valor do gene:

```
Antes da mutação:
(idade < 18)(produto = liquidificador)(sexo=M)

Após a mutação:
(idade < 18)(produto = videogame)     (sexo=M)
```

A fitness pode ser calculada através do número de tuplas selecionadas por uma determinada regra. Um exemplo de fitness baixa poderia ser por exemplo:

```
(idade > 65)(produto = videogame)(sexo=F)(compra=SIM)
```

Já um exemplo de fitness alta poderia ser:

```
(idade < 18)(produto = videogame)(sexo=M)(compra=SIM)
```

Desta maneira, o indivíduo correspondente a esse último exemplo provavelmente terá mais filhos, pois deverá ser selecionado mais freqüentemente do que indivíduo correspondente ao exemplo anterior.

2.5.3 Otimização

Freqüentemente aplicações do mundo real podem ser formuladas na forma de um processo de otimização, como por exemplo a seleção de características discutida anteriormente. Um domínio bastante complexo onde os AGs têm sido utilizado com sucesso é a Formula 1. Um carro de corrida possui centenas de parâmetros que devem ser ajustados em função da pista, temperatura, etc. Para mais informações consulte [25].

Capítulo 3

Programação Evolutiva

A programação evolutiva (PE) foi proposta por Fogel, Owens e Walsh em meados da década de 60 no livro "*Artificial Intelligence Through Simulated Evolution*" [10]. Ainda que a proposta original tratasse de predição de comportamento de máquinas de estado finitos, o enfoque da PE se adapta a qualquer estrutura de problema. Na PE, cada indivíduo gera um único descendente através de mutação, e a seguir a (melhor) metade da população ascendente e a (melhor) metade da população descendente são reunidas para formar a nova geração. Diferentemente do AG, o qual simula operações genéticas, a PE enfatiza o desenvolvimento de modelos comportamentais, ou seja, modelos que capturem a interação do sistema com o seu ambiente. Afim de exemplificar o funcionamento da PE, apresentamos na próxima sessão um exemplo de PE aplicado a evolução de uma máquina de estado finito.

3.1 Evolução de uma Máquina de Estado Finito

PE é utilizada freqüentemente em problemas envolvendo predição. Uma maneira de prever uma ação é através de ações passadas. Se cada ação é representada por um símbolo, então, dado uma seqüência de símbolos devemos prever qual será o próximo símbolo. Assim como nos AGs, os símbolos devem pertencer a um alfabeto finito. Desta maneira, podemos utilizar um sistema baseado em uma máquina de estado finito para analisar uma seqüência de símbolos e gerar uma saída que otimize uma dada função de fitness, a qual envolve a previsão do próximo símbolo da seqüência. Podemos citar alguns exemplos, tais como, mercado financeiro, previsão do tempo, etc...

Uma máquina de estado finito é definida como sendo um transdutor que ao ser estimulado por um alfabeto finito de símbolos, pode responder com um outro alfabeto

finito de símbolos e possui um número finito de estados [6]. Os alfabetos de entrada e saída não são necessariamente idênticos. Devemos especificar o estado inicial da máquina e também especificar, para cada estado e símbolo de entrada, o símbolo de saída e o próximo estado. A Tabela 3.1 apresenta uma tabela de estados para uma máquina de três estados com um alfabeto de entrada de dois símbolos e um alfabeto de saída de três símbolos. Nesta tabela as linhas representam os estados, as colunas representam as entradas e as células da matriz indicam a saída e o próximo estado. A representação gráfica da máquina pode ser visualizada na Figura 3.1.

Tabela 3.1: Tabela de estados de uma máquina de estado finito com três estados.

Estados | Entradas	1	0
A	Y,A	Y,B
B	X,C	Z,B
C	Z,A	Y,B

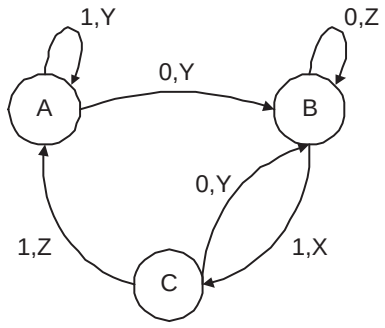


Figura 3.1: Representação da máquina de estado finito de três estados.

.

Máquinas de estado finito são basicamente um sub-conjunto das máquinas de Turing, as quais foram desenvolvidas pelo matemático e cientista da computação inglês Alan Turing (1937). Tais máquina são capazes, em principio, de resolver todos os problemas matemáticos (de uma classe geral definida) em seqüência. As máquinas de estado finito usadas na PE podem modelar ou representar um organismo ou um sistema.

Diferentemente dos AGs, onde o operador de cruzamento é um importante componente para a produção de uma nova geração, a mutação é o único operador usado na PE. Cada membro da população corrente normalmente sofre mutação e produz um

filho. Levando-se em consideração o tipo de máquina de estado finito e suas operações, cinco tipos principais de mutação podem ocorrer (desde que a máquina seja composta por mais de um estado):

1. O estado inicial pode mudar.
2. O estado iniciar pode ser eliminado.
3. Um estado pode ser adicionado.
4. Uma transição entre estados pode ser mudada.
5. O símbolo de saída para um determinado estado e símbolo de entrada pode ser mudado.

Embora o número de filhos produzido por cada pai seja um parâmetro do sistema, cada pai normalmente produz um filho. Sendo assim, a população dobra de tamanho após a operação de mutação. Após calcular a fitness de cada indivíduo, a melhor metade é mantida, o que garante uma população de tamanho constante. Em um determinado momento, algumas aplicações podem desejar fazer uma predição do próximo símbolo da seqüência. O indivíduo que possuir a maior fitness será escolhido então para gerar o próximo símbolo da seqüência.

Diferentemente de outros paradigmas evolutivos, na PE a mutação pode mudar o tamanho do indivíduo (estados podem ser adicionados ou eliminados). Este fato e possíveis mudanças nas transições entre estados podem ocasionar alguns espaços não preenchidos na tabela de especificação. Fogel [6] define essas mutações como *Mutações Neutras*. Também é possível criar situações via mutação onde uma transição não seja possível pois um estado pode ter sido eliminado durante a operação de mutação. Esses tipos de problemas tendem a ter menos efeitos quando são consideradas máquina com um grande números de estados, entretanto, eles ainda podem causar erros se não forem identificados e corrigidos.

Embora PE possa ter indivíduos de tamanhos variáveis, também é possível evoluir uma máquina de estados finitos usando PE com indivíduos de tamanho fixo. Primeiramente, o número máximo de estados deve ser definido. Para exemplificar, vamos considerar a máquina definida na Figura 3.2. O cálculo da fitness pode ser realizado com base no número de símbolos que são prognosticados corretamente. Por exemplo, para a seqüência de entrada 011101 a saída gerada pela máquina é 110111. Se compararmos

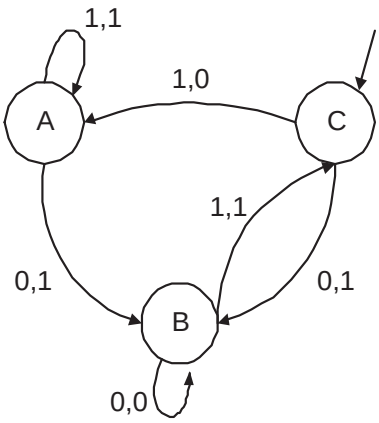


Figura 3.2: Máquina de estado finito usada para fazer predição do próximo símbolo.

as duas strings, podemos verificar que a máquina acertou somente 3 símbolos, ou seja, uma fitness de 50%.

Cada estado pode ser representado por uma string de sete bits. O primeiro bit representa o nível de ativação: 1 para estado ativo e 0 para estado inativo (se o estado não existe). O segundo e terceiro bits representam os símbolo de entrada: 0 ou 1. O terceiro e quarto bits representam os símbolos de saída 0 e 1. Note que no nosso exemplo temos apenas dois símbolos de saída. Para ser mais consistente teríamos que ter um terceiro símbolo para indicar a não existência de um estado. Os dois últimos bits representam um dos quatro estados de saída. A Figura 3.3 mostra um exemplo desta codificação.

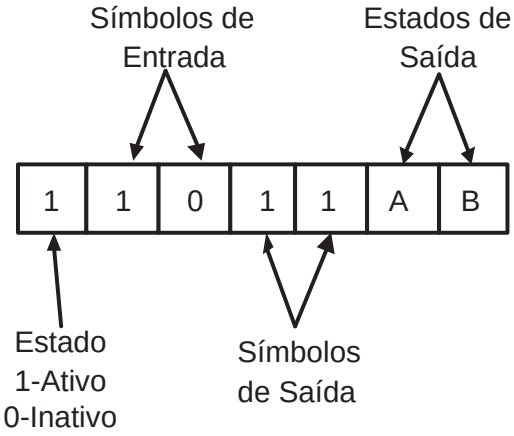


Figura 3.3: Codificação do estado A da Figura 3.2.

A população é então inicializada com indivíduos de 28 bits. Por exemplo, poderia ser uma boa idéia considerar que somente indivíduos que tenham pelo menos dois estados ativos sejam admitidos na população inicial. Considerando os cinco tipos de mutação apresentados anteriormente, um procedimento possível de mutação seria:

1. Para cada indivíduo, gere um número aleatório entre 0 e 1.
2. Se o número estiver entre 0.0 e 0.2; mude o estado inicial; se estiver entre 0.2 e 0.4, elimine o estado; e assim por diante.
3. A mutação selecionada no passo 2 e feita com uma probabilidade igual sobre todas as possibilidades. Por exemplo, se o estado inicial deve ser mudado e existirem n estados ativos, então um estado ativo será selecionado para ser o estado inicial. A probabilidade de um estado ativo ser escolhido para ser estado inicial é de $1/n$.
4. Transições impossíveis de estados são modificadas para se tornarem possíveis. Se uma transição para um estado inativo foi especificada, um dos estados ativos é selecionado para ser o objeto da transição. Como no caso anterior, cada estado ativo tem uma probabilidade de $1/n$ de ser selecionado.
5. Avaliar a fitness e manter os melhores 50%, resultando assim uma nova população do mesmo tamanho da inicial.

O cenário mostrado acima é somente um dos vários possíveis. Mas o que as máquina de estado finito têm haver com computação evolutiva. Um exemplo bastante interessante foi apresentado por Fogel [8], no qual ele evolve uma máquina de estado finito para jogar o jogo do dilema do prisioneiro (um dos jogos mais conhecidos da Teoria dos Jogos). Neste jogo, um jogador tem que tomar uma determinada decisão em face da decisão do outro. No fundo, é uma questão de decisão entre altruísmo ou egoísmo, como veremos.

O Dilema do Prisioneiro é a situação em que dois comparsas são pegos cometendo um crime. Levados à delegacia e colocados em salas separadas, lhes é colocada a seguinte situação com as respectivas opções de decisão:

- Se ambos ficarem quietos, cada um deles pode ser condenado a um mês de prisão;
- Se apenas um acusa o outro, o acusador sai livre. O outro, condenado em um ano;

- Aquele que foi traído pode trair também e, neste caso, ambos pegam seis meses.

As decisões são simultâneas e um não sabe nada sobre a decisão do outro. Considera-se também que os suspeitos irão decidir única e exclusivamente de forma racional. O dilema do prisioneiro mostra que, em cada decisão, o prisioneiro pode satisfazer o seu próprio interesse (desertar) ou atender ao interesse do grupo (cooperar). O primeiro prisioneiro pensa da seguinte forma: "Vou admitir inicialmente que meu comparsa planeja cooperar, ficando quieto. Neste caso, se eu cooperar também, ficarei um mês atrás das grades (um bom resultado); mas, ainda admitindo a cooperação do meu comparsa, se eu desertar confessando o crime, eu saio livre (o melhor resultado possível). Porém, se eu supor que meu comparsa vai desertar e eu continuar cooperando, eu ficarei um ano na cadeia (o pior resultado possível) e ele sai livre. Mas se eu desertar também, eu ficarei somente seis meses preso (um resultado intermediário). Eu concluo então que, em ambos os casos (se ele cooperar ou não), sempre será melhor desertar, e é o que eu vou fazer."

Acontece que o segundo prisioneiro pensa da mesma maneira e ambos desertam. Se ambos cooperassem, haveria um ganho maior para ambos, mas a otimização dos resultados não é o que acontece. Ao invés deles ficarem somente um mês presos, eles passarão seis meses na cadeia para evitar o risco de ficar um ano se o outro optar por desertar. Mais que isso: desertando, cada parte tem a possibilidade de sair livre se a outra parte cooperar.

A repetição do jogo, entretanto, muda radicalmente a forma de pensar do prisioneiro. Dois comparsas de longa data terão uma tendência muito maior à cooperação. Com isto, formam-se outras opções de estratégia. A teoria dos jogos é bastante utilizada na economia para descrever e prever o comportamento econômico. Muitas decisões do tipo econômico dependem das expectativas que se tem sobre o comportamento dos demais agentes econômicos.

Voltando ao nosso assunto principal, a idéia é utilizar PE para evolver o jogo de maneira que possamos utilizar o modelo para prever a jogada do oponente. Figura 3.4 mostra o diagrama de uma máquina de estado finito de sete estados para jogar o dilema do prisioneiro. O estado inicial é o estado 6 e o jogo é iniciado pela cooperação. Nesta figura, "C" e "D" significam cooperar e desertar, respectivamente. O alfabeto de entrada compreende $[(C,C),(C,D),(D,C)(D,D)]$, onde a primeira letra representa o movimento anterior da máquina e a segunda o movimento anterior do oponente. Por exemplo, o rótulo $C,D/C$ na flecha que vai de um estado X para um estado Y significa que o sistema está no estado X e no movimento anterior a máquina cooperou

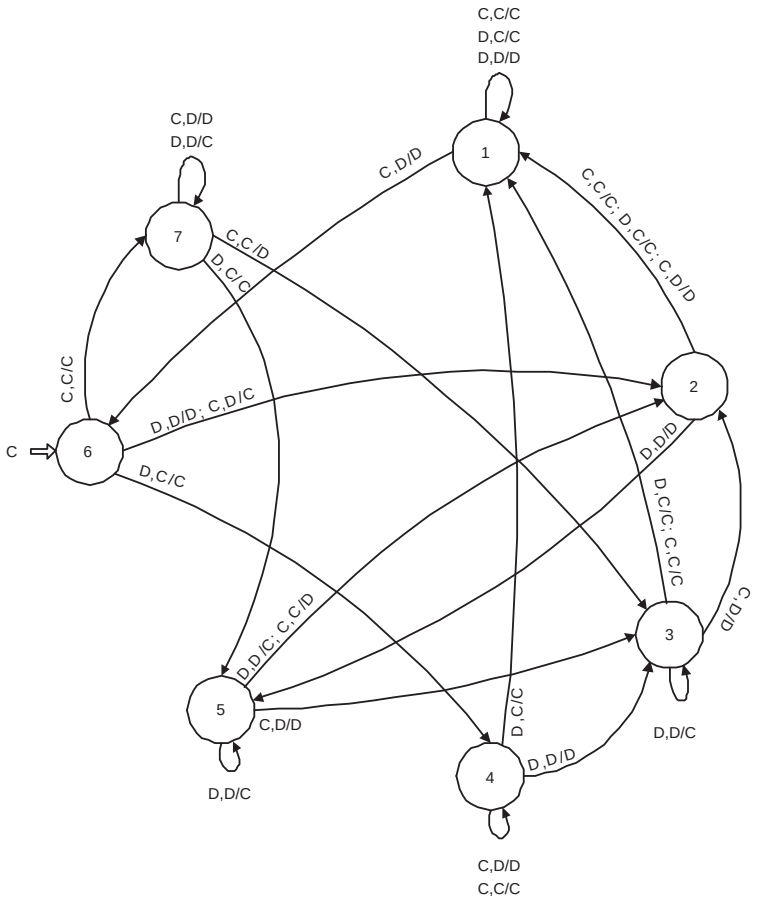


Figura 3.4: Uma máquina de estado finito para o jogo do dilema do prisioneiro (extraído de [8]).

.

e o oponente desertou, então coopere e vá para o estado Y. Algumas vezes, mais que uma situação pode resultar na mesma transição de estado. Por exemplo, na Figura 3.4, assuma que a máquina está no estado 6. Nesse caso, se a máquina e o oponente desertaram no movimento anterior, a máquina deserta (D,D/D) e vai para o estado 2. Da mesma maneira, a transição do estado 6 para o estado 2 ocorre se a máquina cooperou e o oponente desertou no movimento anterior; a máquina coopera nesse caso (C,D/C) e vai para o estado 2. Como o jogo é jogado várias vezes, a PE evolve a máquina de modo que em um determinado momento a jogada do oponente possa ser prevista.

Capítulo 4

Estratégias Evolutivas

Estratégias evolutivas (EE) são baseadas na evolução da evolução. Se levarmos em consideração que os processos biológicos foram otimizados através da evolução, e a evolução não deixa de ser um processo biológico, então a evolução também deve ter sido otimizada através dos tempos. Apesar das EE utilizarem operadores de mutação e cruzamento (geralmente chamado de recombinação na literatura de EE), eles têm uma pequena diferença dos operadores utilizados nos AGs e PE.

As EE foram desenvolvidas inicialmente na Alemanha, na década de 60. Naquele tempo, o objetivo era a resolução de problemas contínuos de otimização paramétrica [23]. Portanto, a representação usada nas EE é um vetor de números reais de tamanho fixo. Assim como nos AGs, cada posição do vetor corresponde a uma característica do problema. A seguir descrevemos os principais operadores das EE: mutação, recombinação e seleção.

4.1 Mutação

A idéia da mutação é criar uma nova geração de indivíduos. Para isso, adiciona-se números aleatórios (extraídos de uma distribuição normal) às coordenadas dos pais. Considere por exemplo um vetor pai $X = (x_1, x_2, \ldots, x_n)$. O filho o indivíduo X é dado pela seguinte equação:

$$X' = X + N(0, \sigma) \tag{4.1}$$

A Equação 4.1 nos mostra que um indivíduo é determinado por um conjunto de características e seus respectivos parâmetros, os quais geralmente são representados pelo

desvio padrão. Intuitivamente, podemos notar que se aumentarmos o desvio padrão, aumentaremos também a variabilidade dos indivíduos, ou seja, eles serão mais diferentes de seus pais. Em outras palavras, altos desvios significam um exploração em toda a região (*exploration*) de busca, enquanto baixos valores significam um exploração local (*exploitation*) em determinadas regiões do espaço de busca. A Figura 4.1 mostra um exemplo dessa mutação.

Figura 4.1: Mutação Gaussiana de um pai (a) para formar um filho (b).

.

Em EE, a mutação deve ser realizada com um desvio padrão ótimo, o qual é definida através da regra de 1/5. Ou seja, se a taxa de sucesso na mutação (entende-se por sucesso o filho que produz uma fitness melhor que o pai) for maior que 1/5, então o desvio padrão deve ser aumentado. Se a taxa de sucesso for menor que 1/5, então o desvio padrão deve ser reduzido.

A razão intuitiva por traz da regra de 1/5 é o aumento da eficiência na busca. Ou seja, se bem sucedida, a busca continua a passos maiores, caso contrário o passo deve ser reduzido.

4.2 Recombinação

Existem dois métodos de recombinação em EE. O primeiro e mais comum consiste em formar um novo indivíduo com base em dois pais selecionados aleatóriamente. Esse método é denominado *método local*. No segundo método, denominado *método global*, os valores do indivíduo resultante podem vir de vários pais e não somente dois.

Ambos os métodos, global e local, podem ser implementados de duas maneiras diferentes. A primeira, chamada de *recombinação discreta*, seleciona o valor que o indivíduo filho irá receber de um dos pais. A segunda, chamada *recombinação intermediária*, seleciona um ponto médio dos valores dos pais, o qual deverá ser atribuído ao filho. Sejam os pais A e B, e o i-ésimo parâmetro está sendo determinado, então o valor estabelecido usando a recombinação intermediária é

$$x_i^{new} = C(X_{B,i} + X_{A,i}) \tag{4.2}$$

onde C é uma constante normalmente igual a 0.5 para que possa produzir o ponto médio dos valores dos pais. Figura 4.2 exemplifica esse processo.

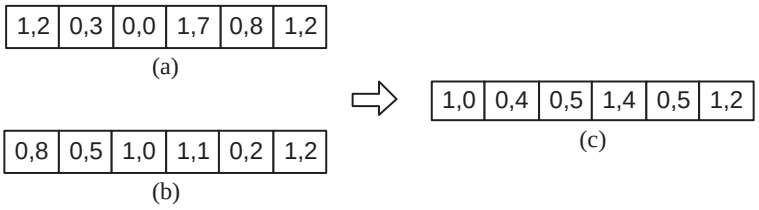


Figura 4.2: Recombinação intermediária dos pais (a) e (b) para formar o filho (c).

.

Podemos notar que EE contém um componente de representação sexual de características. Na recombinação intermediária, por exemplo, os filhos são computados através da média dos seus pais, enquanto na recombinação discreta, o indivíduo pode sair da recombinação intacto ou com características de um pai ou outro.

4.3 Seleção

Assim como nas outras técnicas de computação evolutiva, EE também determina a probabilidade de reprodução de um indivíduo através da sua fitness. Uma maneira bastante simples de fazer a seleção consiste em ordenar todos os indivíduos, selecionar aqueles com melhores fitness e descartar o resto. A natureza, entretanto, não age desta maneira. A sobrevivência de um indivíduo depende do ambiente em que ele vive. Imagine uma lebre que sofra uma mutação que torne sua pelagem preta no inverno. É obvio que esta lebre será um alvo mais fácil do que as outras. Pode acontecer, porém, que essa lebre passe o inverno inteiro sem ser alvo de caça, pois os predadores não estão presentes na região em que ela vive. Isso pode acontecer, apenas a probabilidade de que isso aconteça é muito baixa. Isso sugere que a seleção deve ser estocástica. Um método estocástico bastante utilizado para seleção é o método da roleta russa visto anteriormente.

As versões mais comuns de EE são (μ, λ) e $(\mu+\lambda)$-EE. Em ambas versões , o número de filhos gerados a partir de μ pais é $\lambda > \mu$. Normalmente a proporção é de 7 filhos para cada pai. Na versão original (1+1)-EE, um pai produz um filho e somente o que possuir a melhor fitness sobrevive. Vale a pena ressaltar que essa versão do algoritmo é raramente utilizada.

Na versão (μ, λ), os μ indivíduos com as melhores fitness são escolhidos entre os λ filhos. Note que os μ pais não são elegíveis nesse esquema de seleção, somente os filhos. Na versão $(\mu + \lambda)$-EE os melhores μ indivíduos são selecionados entre um grupo de candidatos que incluem os μ pais e os λ filhos.

O algoritmo clássico da EE pode ser resumido nos seguinte passos:

1. Inicializar a população.
2. Realizar a recombinação utilizando μ pais para formar λ filhos.
3. Realizar a mutação em todos os filhos.
4. Avaliar a fitness de μ ou $\mu + \lambda$ indivíduos (de acordo com a estratégia escolhida).
5. Selecionar μ indivíduos para compor a nova população.
6. Se o critério de parada não foi alcançado, então volte ao item 2; Caso contrario, fim.

Capítulo 5

Programação Genética

As três áreas da computação evolutiva discutidas até aqui envolveram estruturas definidas como strings. Em alguns casos, as strings continham valores binários e em outros valores reais, mas sempre strings (ou vetores). A programação genética (PG) [18] evolui programas de computadores os quais são representados através de árvores de sintaxe abstrata. A PG pode ser vista como um sub-conjunto dos AGs. As principais diferenças entre PG e AG são:

- Os membros da população são estruturas executáveis (geralmente na forma de programas de computador), e não strings de bits ou valores reais.
- A fitness de um membro da população é conseguida através da execução deste programa.

A PG é a evolução de um conjunto de programas com o objetivo de aprendizagem por indução. A objetivo é ensinar os computadores a se auto programarem, isto é, a partir de especificações de comportamento, o computador deve ser capaz de induzir um programa que as satisfaça. A cada programa é associado uma fitness representando o quanto ele é capaz de resolver o problema. O mecanismo de busca da PG pode ser descrito como um ciclo "criar-testar-modificar", muito similar a forma com que os humanos desenvolvem seus programas. Inicialmente, programas são criados baseados no conhecimento sobre o domínio do problema. Em seguida, são testados para verificar sua funcionalidade. Se os resultados não forem satisfatórios, modificações são feitas para melhorá-los. Este ciclo é repetido até que uma solução satisfatória seja encontrada ou um determinado critério seja satisfeito.

Como foi dito anteriormente, cada programa é representado por uma árvore de sintaxe abstrata, onde as funções definidas para o problema aparecem nos nós internos

da árvore e as constantes e variáveis aparecem nos nós-folhas. A implementação da PG consistem em:

- Determinar o conjunto dos terminais.
- Determinar o conjunto das funções válidas.
- Determinar a medida de fitness.
- Selecionar os parâmetros de controle.
- Determinar as condições de parada.

As funções válidas são limitadas pela linguagem de programação utilizada na construção dos programas dentro da PG. Elas podem ser, por exemplo, funções matemáticas (seno, cosseno, etc.), aritméticas (+, -, ×, etc.), operadores Booleanos (AND, NOT, etc.), operadores condicionais (if-then-else), funções iterativas e funções recursivas. A tarefa de especificar o conjunto de funções válidas consiste em selecionar o conjunto mínimo de funções necessárias para realizar a tarefa desejada. Cada função do conjunto de funções válidas requer um certo número de argumentos, conhecido como *aridade* da função.

Isso nos leva a duas propriedades desejáveis em uma aplicação de PG: fechamento e suficiência. Para garantir a viabilidade das árvores de sintaxe abstrata, Koza [18] definiu a propriedade de Fechamento (*closure*). Para satisfazê-la, cada função do conjunto de funções válidas deve aceitar, como seus argumentos, qualquer valor que possa ser retornado por qualquer função ou terminal. Esta imposição garante que qualquer árvore gerada pode ser avaliada corretamente. Um caso típico de problema de Fechamento é a operação de divisão. Matematicamente, não é possível dividir um valor por zero. Uma abordagem possível é definir uma função alternativa que permita um valor para a divisão por zero.

É o caso da função de divisão protegida (*protected division*) "%" proposta por Koza [18]. A função "%" recebe dois argumentos e retorna o valor 1 (um) caso seja feita uma divisão por zero e, caso contrário, o seu quociente. Para garantir a convergência para uma solução, a propriedade de Suficiência (*sufficiency*) foi definida. Ela diz que os conjuntos de funções válidas e terminais devem ser capazes de representar uma solução para o problema. Isto implica que deve existir uma forte evidência de que alguma composição de funções e terminais possa produzir uma solução. Dependendo

do problema, esta propriedade pode ser óbvia ou exigir algum conhecimento prévio de como deverá ser a solução.

A fitness utilizada geralmente é intensamente proporcional ao erro produzido pela saída do programa. Os dois principais parâmetros na PG são o tamanho da população e o número máximo de gerações. Outros parâmetros a serem determinados são a probabilidade de reprodução, probabilidade de cruzamento e o tamanho máximo de um indivíduo (profundidade da árvore).

5.1 Entendendo a PG

O algoritmo de PG pode ser descrito resumidamente da seguinte forma:

1. Inicializar a população de programas.
2. Determinar a fitness de cada indivíduo.
3. Realizar a reprodução de acordo com o valor da fitness e a probabilidade de reprodução.
4. Realizar o cruzamento
5. Voltar ao passo 2 até que uma condição de parada seja alcançada.

Cada execução deste laço representa uma nova geração de programas. Tradicionalmente, a condição de parada é estabelecida como sendo encontrar uma solução satisfatória ou atingir um número máximo de gerações. Existem também abordagens baseadas na análise do processo evolutivo, ou seja, o laço permanece enquanto houver melhoria na população.

5.1.1 Criando um indivíduo

A representação dos programa em PG se baseia tradicionalmente em árvores de sintaxe abstrata. Os programas são formados pela livre combinação de funções e terminais adequados ao domínio do problema. Primeiramente, deve-se definir os conjuntos de funções F e de terminais T. Cada $f \in F$ tem associada uma aridade superior a zero. O conjunto T é composto pelas variáveis, constantes e funções de aridade zero.

Considere por exemplo os seguintes conjuntos T e F: $F = \{+, -, \times, \div\}$ e $T = \{x, 3, 6\}$. Ou seja, o conjunto das funções válidas é o conjunto das operações aritméticas

de aridade dois, e o conjunto dos terminais é composto pela variável x e as constantes 3 e 6. Sendo assim, expressões matemáticas simples tais como $3 \times (x + 6)$ podem ser produzidas. A representação é feita por uma árvore de sintaxe abstrata como mostrado na Figura 5.1

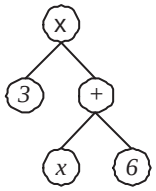


Figura 5.1: Árvore de sintaxe abstrata de $3 \times (x + 6)$

.

O espaço de busca é determinado por todas as árvores que possam ser criadas pela livre combinação de elementos dos conjuntos F e T.

5.1.2 Criando um População Aleatória

A população inicial normalmente é composta por árvores geradas aleatoriamente a partir dos conjuntos de funções F e de terminais T. Primeiramente se escolhe de maneira aleatória uma função $f \in F$. Para cada um dos argumentos de f, escolhe-se um elemento de $\{F \bigcup T\}$. O processo prossegue até que se tenha apenas terminais como nós folha da árvore. Usualmente se especifica um limite máximo para a profundidade da árvore para se evitar árvores muito grandes.

Um aspecto muito importante para o sucesso do processo evolutivo é a qualidade da população inicial. Ela deve ser uma amostra significativa do espaço de busca, apresentando uma grande variedade de composição nos programas, para que seja possível através da recombinação de seus códigos, convergir para uma solução. Afim de melhorar a qualidade da população inicial, diversos métodos têm sido propostos: *ramped-half-and-half* [18], *random-branch* [2] e *probabilistic tree-creation* [20]. O método ramped-half-and-half proposto por Koza [18] é uma combinação de dois métodos simples: *grow* e *full*. O método *grow* envolve a criação de árvores cuja profundidade é variável. A escolha dos nós é feita aleatoriamente entre funções e terminais, respeitando-se uma profundidade máxima.

Já o método *full* envolve a criação de árvores completas, isto é, todas as árvores terão a mesma profundidade. Isto é facilmente feito através da seleção de funções para os nós cuja profundidade seja inferior a desejada e a seleção de terminais para os nós

de profundidade máxima. Combinar os métodos *full* e *grow* com objetivo de gerar um número igual de árvores para cada profundidade, entre dois e a profundidade máxima, é a base do método ramped-half-and-half. Por exemplo, supondo que a profundidade máxima seja seis, então serão geradas árvores com profundidades de dois, três, quatro, cinco e seis eqüitativamente. Isto significa que 20% terão profundidade dois, 20% terão profundidade três e assim sucessivamente. Para cada profundidade, 50% são geradas pelo método *full* e 50% pelo método *grow*.

Segundo Luke [20] este método apresenta algumas desvantagens:

- Impõe uma faixa fixa de profundidades (normalmente entre 2 e 6), independentemente do tamanho da árvore. Dependendo do número de argumentos (aridade) de cada função, mesmo com a mesma profundidade, podem ser geradas árvores de tamanhos muito diferentes.
- A escolha da profundidade máxima, antes de se gerar a árvore, não é aleatória e sim de forma proporcional.
- Se o conjunto de funções for maior que o de terminais (como na maioria dos problemas), a tendência é gerar a maior árvore possível ao aplicar *grow*.

O método *random-branch* [2] permite que se informe qual o tamanho máximo da árvore (e não a sua profundidade). Porém, devido ao fato de *random-branch* dividir igualmente S dentre as árvores de um nó-pai não-terminal, existem muitas árvores que não são possíveis de serem produzidas. Isto torna o método muito restritivo apesar de ter complexidade linear.

Os métodos *probabilistic tree-creation* (PTC) 1 e 2 [20], ao contrário dos outros métodos, não procuram gerar estruturas de árvores completamente uniformes. Ao invés disso, permite definir as probabilidades de ocorrência das funções na árvore. O PTC1 é uma variante do *grow* onde para cada terminal $f \in F$, associa-se uma probabilidade q_t dele ser escolhido quando houver necessidade de um terminal. O mesmo se faz com cada $f \in F$, associando-se uma probabilidade q_f. Antes de gerar qualquer árvore, o algoritmo calcula p, a probabilidade de escolher um não-terminal ao invés de um terminal, de forma a produzir uma árvore de tamanho esperado E_{tree}. O valor de p é calculado utilizando a seguinte formula:

$$p = \frac{1 - \frac{1}{E_{tree}}}{\sum_{n \in N} q_n b_n} \tag{5.1}$$

PTC1 garante que as árvores serão geradas dentro de um tamanho esperado. Uma variante deste método (PTC2) usa um tamanho máximo S e uma distribuição de probabilidades $w_1, w_2, \ldots, w_s$ para cada árvore de tamanho 1 a S. Além do controle sobre o tamanho esperado da árvore, tem-se um controle sobre a distribuição destes tamanhos.

5.1.3 Fitness

Assim como nos AGs, após a criação da população inicial o próximo passo consiste na avaliação da fitness de cada indivíduo. Alguns métodos para avaliação de fitness são descritos abaixo.

- *Raw fitness*: representa a medida dentro do próprio domínio do problema. É a avaliação pura e simples do programa. O método mais comum de *raw fitness* é a avaliação do erro cometido, isto é, a soma de todas as diferenças absolutas entre o resultado obtido pelo programa e o seu valor correto.

- *Standardized fitness*: Devido ao fato da *raw fitness* depender do domínio do problema, um valor bom pode ser um valor baixo (quando se avalia o erro) ou um valor alto (quando se avalia o desempenho). A avaliação desta fitness é feita através de uma função de adaptação do valor da *raw fitness* de forma que quanto melhor o programa, menor deve ser a *standardized fitness*. Desta forma, o melhor programa apresentará o valor zero (0) como *standardized fitness*, independentemente do domínio do problema.

- *Adjusted fitness*: é obtida através da *standardized fitness*. Se $s(i,t)$ representa a *standardized fitness* do indivíduo i na geração t, então a *adjusted fitness* $a(i,t)$ é calculada da seguinte maneira:

$$a(i,t) = \frac{1}{1 + s(i,t)} \tag{5.2}$$

  Percebe-se que a *adjusted fitness* varia entre zero (0) e um (1), sendo que os maiores valores representam os melhores indivíduos. A *adjusted fitness* tem o benefício de exagerar a importância de pequenas diferenças no valor da *standardized fitness* quando esta se aproxima de zero.

- *Normalized fitness*: se $a(i,t)$ é a *adjusted fitness* do indivíduo i na geração t, então sua *normalized fitness* $n(i,t)$ será obtida da seguinte forma:

$$n(i,t) = \frac{a(i,t)}{\sum_{k=1}^{m} a(k,t)} \tag{5.3}$$

5.1.4 Operadores Genéticos

O processo evolutivo começa após o calculo da fitness. Os indivíduos de uma nova população são formados através da aplicação de três operadores: reprodução, cruzamento e mutação. Uma vez formada a nova população (do mesmo tamanho da anterior), a população antiga é destruída.

Reprodução

Um programa é selecionado e copiado para a próxima geração sem sofrer nenhuma mudança de estrutura. Koza [18] permite que 10% da população seja reproduzida. A seleção dos indivíduos que serão reproduzidos é responsabilidade dos métodos de seleção, os quais serão discutidos na próxima seção.

Cruzamento

Dois programas são selecionados e são recombinados para gerar outros dois programas. Um ponto aleatório de cruzamento é escolhido em cada programa pai e as árvores abaixo destes pontos são trocadas. A Figura 5.2 ilustra esse processo.

Neste exemplo, foram escolhidos os programas: $(2 \times 8) + (4 \div 1)$ e $3 \times (x + 6)$. Foram escolhidas aleatoriamente uma sub-á rvore em cada árvore, identificada por um retângulo (Figura 5.2). As árvores são então trocadas, gerando os novos programas: $(5 + 6) + (4 \div 1)$ e $3 \times (2 \times 8)$

Para que o cruzamento seja sempre possível, o conjunto de funções deve apresentar a propriedade de Fechamento (*closure*), isto é, as funções devem suportar como argumento qualquer outra função ou terminal. Se não for possível, deve-se estabelecer critérios de restrição na escolha dos pontos de cruzamento.

Mutação

Um programa é selecionado e um de seus nós é escolhido aleatóriamente. A árvore cuja raiz é o nó selecionado deve ser eliminada e substituída por uma nova árvore gerada aleatóriamente.

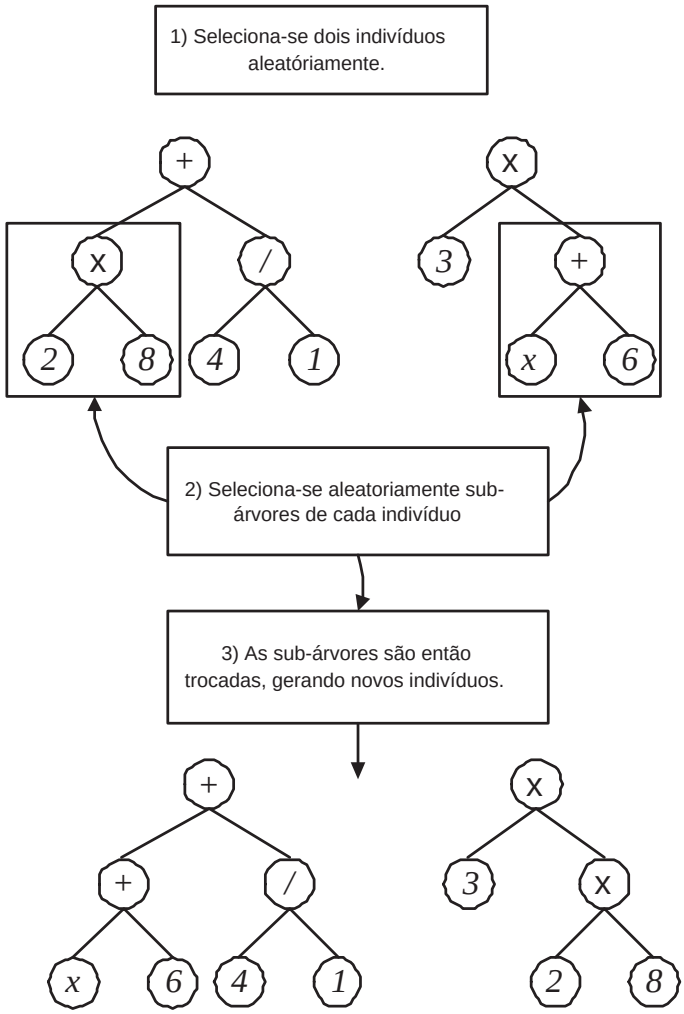


Figura 5.2: Exemplo de cruzamento entre dois programas.

5.1.5 Métodos de Seleção

O método de seleção tem por objetivo escolher quais programas deverão sofrer a ação dos operadores genéticos e compor uma nova geração. Dado que a "qualidade" de um programa é dada pelo seu valor de fitness, a seleção deve priorizar, de alguma forma, os programas que apresentem os melhores valores de fitness. Alguns métodos usados são: Seleção Proporcional, Seleção por Torneio e Seleção por Truncamento.

1. Seleção Proporcional (*fitness-proportionate selection*): Apresentada por John Holland [14] para AGs, foi o método escolhido por Koza [18]. Ele usa a *normalized fitness* disposta em uma "roleta", sendo que cada indivíduo da população ocupa uma "fatia" proporcional a sua aptidão normalizada (veja Figura 2.2, Seção 2.2).

Em seguida é produzido um número aleatório no intervalo [0,1]. Este número representará a posição ocupada pela "agulha" da roleta.

2. Seleção por Torneio (*tournament selection*): Apresentada por David Goldberg [12] para AGs, foi utilizada em vários problemas por John Koza [19]. A seleção por torneio é feita da seguinte forma: t indivíduos são selecionados aleatóriamente na população e o melhor deles é o escolhido. Este processo é repetido até que se tenha uma nova população. O valor de t é conhecido como o tamanho do torneio.

3. Seleção por Truncamento (*truncation selection*): Com base em um valor de limiar T entre [0,1], a seleção é feita aleatóriamente entre os T melhores indivíduos. Por exemplo, se $T = 0.4$, então a seleção é feita entre os 40% melhores indivíduos e os outros 60% são descartados.

Capítulo 6

Inteligência Coletiva

No início dos anos 90, alguns pesquisadores começaram a fazer analogias entre o comportamento dos enxames de criaturas e os problemas de otimização. Colorni et al [3] desenvolveram uma estratégia distribuída de otimização baseada em colônias de formigas (ACO - Ant Colony Optmization), a qual é baseada no comportamento social das formigas. Em 1995, Kennedy e Eberhart propuseram uma técnica de otimização baseada em revoadas (enxame) de pássaros, chamada *Particle Swarm Intelligence* (PSO) [16]. Essas técnicas são classificadas como "Inteligência de Enxame" ou "Inteligência Coletiva" . Neste capítulo discutiremos com mais detalhes PSO.

6.1 Particle Swarm Intelligence

PSO tem várias similaridades com as técnicas evolutivas discutidas anteriormente. O sistema é inicializado com uma população de soluções aleatórias e busca uma solução ótima através das gerações. Diferentemente dos AGs, PSO não conta com os operadores evolutivos, tais como cruzamento e mutação. Em PSO, as soluções potenciais, chamadas *Partículas*, voam no espaço de busca seguindo as partículas ótimas.

Comparado aos AGs, PSO tem a vantagem de ser fácil e simples de implementar e exigir menos parâmetros ajustáveis. Diferentemente dos AGs que foram concebidos inicialmente para lidar com representações binárias, PSO foi desenvolvido para representações contínuas. Entretanto, recentemente Kennedy e Eberhart [17] apresentaram algumas alterações no algoritmo original para representações binárias.

6.1.1 O Algoritmo

Como mencionado anteriormente, PSO simula o comportamento de revoadas de pássaros. Considere o seguinte cenário: um grupo de pássaros procurando comida em uma determinada área, a qual tem um único pedaço de comida. Os pássaros não sabem onde está a comida, mas sabem o quão distante está a comida a cada iteração. Então qual seria a melhor estratégia para procurar comida? Uma estratégia que parece bastante eficaz é seguir o pássaro que está mais próximo da comida.

PSO aprende a partir do cenário e usa esse aprendizado para resolver problemas de otimização. Deste modo, cada solução é um pássaro (partícula) no espaço de busca. Todas as partículas têm um valor de fitness, a qual depende da função sendo otimizada, e velocidades que indicam a maneira de voar das mesmas.

Por uma questão de simplicidade, vamos considerar um espaço bi-dimensional. A posição de cada partícula é representada por (x, y), a velocidade no eixo x é representada por vx e a velocidade no eixo y é representada por vy. A modificação da partícula é realizada com base nas variáveis de velocidade. PSO é inicializado com um grupo de indivíduos (partículas) que são atualizados através das gerações. Em cada iteração, cada partícula é atualizada baseada em dois valores. O primeiro é a melhor posição (fitness) que ela mesma encontrou até então. Esse valor é conhecido como *pbest*. O segundo valor e a melhor posição (fitness) encontrada dentre todas as partículas da população, ou seja, o melhor de todos os *pbest* . Esse valor é conhecido com *gbest*. Essa informação indica como as outras partículas estão se saindo.

Cada agente tenta modificar a sua posição levando em consideração as seguintes informações:

- Sua posição corrente (x, y).
- As velocidades correntes (vx e vy).
- A distância entre a posição corrente e *pbest*.
- A distância entre a posição corrente e *gbest*.

Essa alteração de posição se dá através da seguinte equação:

$$v_i^{k+1} = wv_i^k + c_1 rand() \times (pbest_i - s_i^k) + c_2 rand() \times (gbest - s_i^k) \tag{6.1}$$

onde v_i^k é a velocidade da partícula i na iteração k, w é uma função de peso, c_j são os fatores de aprendizagem (normalmente $c_1 = c_2 = 2$), $rand()$ é um número aleatório

entre 0 e 1, s_i^k é a posição da partícula i na iteração k, $pbest_i$, é a melhor posição da partícula i e *gbest* é a melhor posição do grupo.

A seguinte função de peso é geralmente utilizada:

$$w = w_{max} - \frac{w_{max} - w_{min}}{\text{iter}_{max}} \times \text{iter} \tag{6.2}$$

onde w_{max} é o peso inicial, w_{min} é o peso final, iter_{max} é o número máximo de iterações e iter é a iteração corrente. Quanto maior for o valor do peso, mais diversa tende ser a busca (*exploration*). A medida que o peso diminui, a exploração se torna local (*exploitation*). Uma técnica normalmente aplicada consiste em inicializar o algoritmo com um peso alto (proximo de 1) e diminuir seu valor a medida que o processo evolui.

A posição corrente pode ser modificada utilizando a seguinte equação:

$$s_i^{k+1} = s_i^k + v_i^{k+1} \tag{6.3}$$

A Figura 6.1a mostra a representação gráfica da modificação de um ponto no espaço de busca.

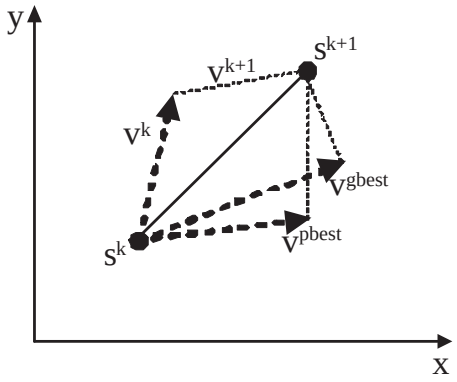


Figura 6.1: Representação gráfica da modificação de um ponto no espaço de busca.

O pseudo-código do PSO é descrito abaixo

```
Para cada partícula
    Inicialize a partícula
Fim

Faça
    Para cada partícula
```

```
        Calcule of valor de fitness
        Se o valor de fitness >= pbest
            pbest = valor da fitness
    Fim

    gbest = melhor partícula

    Para cada partícula
        Calcule a velocidade da partícula
        Atualize a posição da partícula
    Fim

Enquanto não atingir o critério de parada
ou número máximo de iterações.
```

As velocidades de cada dimensão das partículas são restringidas à uma velocidade V_{max}. Se a soma das acelerações (dimensões) ultrapassar V_{max} (parâmetro definido pelo usuário), a velocidade naquela dimensão será limitada a V_{max}.

Com base nas equações 6.1, 6.2 e 6.3 podemos observar que o PSO clássico foi desenvolvido para resolver problemas de otimização com representação contínua. A equação 6.1 responsável pela atualização da velocidade das partículas pode ser dividida em três termos. O primeiro termo é a velocidade antiga da partícula. O segundo e terceiro termos são utilizados para mudar a velocidade da partícula. Sem eles, ela ficaria voando na mesma direção até bater na fronteira do espaço de busca.

Como pode-se verificar, PSO tem vários pontos em comum com AGs. Ambos começam com uma população aleatória e avaliam as fitness de cada indivíduo. Além disso, ambos buscam um ponto ótimo de maneira estocástica. Ambas estratégias não garantem sucesso. Comparando PSO e AG, a maneira pela qual se compartilha informações entre os indivíduos é bem diferente. Nos AGs, os cromossomos compartilham informações uns com os outros, e portanto, toda a população se move como um grupo em direção do ponto ótimo. Em PSO, somente *gbest* fornece essa informação para os outros indivíduos. Experimentos têm mostrado que PSO converge mais rapidamente que os AGs.

6.1.2 Controlando os Parâmetros

Como citado anteriormente, a lista de parâmetros a serem controlados não é tão extensa. Nesta seção descrevemos os parâmetros típicos em PSO.

- Número de partículas: Geralmente entre 20 e 40. Em muitos problemas 10 é suficiente.
- Dimensão das partículas: Assim como em outras técnicas de computação evolutiva, depende do problema.
- Domínio das partículas: Valores máximo e mínimo que uma partícula pode assumir. Também é dependente do problema.
- V_{max}: Determina a velocidade máxima de uma partícula em uma determinada iteração. Por exemplo, considera a partícula (x_1, x_2, x_3), onde x_i pode assumir valores entre [-10,10]. Nesse caso, $V_{max} = 20$.
- Fatores de aprendizagem (c_1, c_2): Geralmente igual a 2. Entretanto outros valores entre [0,4] podem ser utilizados.

6.1.3 PSO Discreto

Como discutido anteriormente, PSO foi inicialmente concebido para problemas de otimização com representação contínua. Entretanto, vários problemas são encontrados na forma discreta. Para suprir essa carência, Kennedy e Eberhart desenvolveram uma versão binária do PSO [17]. No modelo binário, a probabilidade de uma partícula decidir sim ou não, é uma função que leva em consideração fatores pessoais e sociais como segue:

$$P(s_i^{k+1} = 1) = f(s_i^k, v_i^k, pbest_i, gbest) \tag{6.4}$$

O parâmetro v, que representa a predisposição da partícula fazer uma ou outra escolha, determina um limiar de probabilidade. Se o valor de v for elevado, provavelmente a escolha será 1, caso contrário 0. Para determinar isso, recorre-se a uma função sigmoid:

$$sigmoid(v_i^k) = \frac{1}{1 + \exp(-v_i^k)} \tag{6.5}$$

A formula que ajusta a velocidade das partículas no modelo discreto é:

$$v_i^{k+1} = v_i^k + rand() \times (pbest_i - s_i^k) + rand() \times (gbest - s_i^k) \tag{6.6}$$

$$If\ \rho_i^{k+1} < \text{sigmoid}(v_i^{k+1}) \text{então} s_i^{k+1} = 1 \text{senão} = 0 \tag{6.7}$$

onde ρ_i^{k+1} é um vetor de números aleatórios [0.0, 1.0].

Referências Bibliográficas

[1] T. Back. *Evolutionary Algorithms in Theory and Practice.* Oxford University Press, 1996.

[2] K. Chellapilla. Evolving programs without sub-tree crossover. *IEEE Trans. on Evolutionary Computation*, 1(3):209–216, 1997.

[3] A. Colorni, M. Dorigo, and V. Maniezzo. Distributed optimization by ant colonies. In *Procs. of the 1st European Conference on Artifical Life*, pages 134–142, 1991.

[4] K. Deb. *Multi-Objetive Optimization using Evolutionary Algorithms.* Wiley, 2 edition, 2001.

[5] C. Emmanouilidis, A. Hunter, and J. MacIntyre. A multiobjective evolutionary setting for feature selection and a commonality-based crossover operator. In *Procs. of the Congress on Evolutionary Computation*, volume 1, pages 309–316, 2000.

[6] D. B. Fogel. *System identification through simulated evolution: Machine Learning Approach to Modeling.* Ginn Press, 1991.

[7] D. B. Fogel. An introduction to simulated evolutionary computation. *IEEE Transactions on Neural Networks*, 5:3–14, 1994.

[8] D. B. Fogel. *Evolutionary Computation: Toward a new philosophy of machine intelligence.* IEEE Press, 1995.

[9] D. B. Fogel. *Evolutionary Computing.* IEEE Press, 2002.

[10] L. J. Fogel, A. J. Owens, and M. J. Walsh. *Artificial Intelligence Through Simulated Evolution.* John Wiley & Sons, 1966.

[11] A. S. Fraser. Simulation of genetic systems by automatic digital computers. *Australian Journal of Biological Science*, 10:484–499, 1957.

[12] D. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, 1989.

[13] D. E. Goldberg and J. Richardson. Genetic algorithms with sharing for multimodal function optimisation. In *Procs. of the 2nd International Conference on Genetic Algorithms and Their Applications*, pages 41–49, 1987.

[14] J. H. Holland. *Adaptation in Natural and Artificial Systems.* MIT Press, 1975.

[15] J. H. Holland. *Adaptation in Natural and Artificial Systems.* MIT Press, 2 edition, 1992.

[16] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Procs. of the International Conference on Neural Networks*, pages 1942–1948, 1995.

[17] J. Kennedy and R. C. Eberhart. *Swarm Intelligence.* Morgan Kaufmann, 2001.

[18] J. R. Koza. *Genetic Programming: On the programming of computers by means of natural selection.* MIT Press, 1992.

[19] J. R. Koza. *Genetic Programming II: Automatic discovery of reusable programs.* MIT Press, 1994.

[20] S. Luke. Two fast tree-creation algorithms for genetic programming. *IEEE Trans. on Evolutionary Computation*, 4(3):274–283, 2000.

[21] Z. Michalewicz and M. Schoenauer. Evolutionary algorithms for constrained parameter optimization problems. *Evolutionary Computation*, 4:1–32, 1996.

[22] L. S. Oliveira, R. Sabourin, F. Bortolozzi, and C. Y. Suen. A methodology for feature selection using multi-objective genetic algorithms for handwritten digit string recognition. *International Journal of Pattern Recognition and Artificial Intelligence*, 17(6):903–930, 2003.

[23] I. Rechenberg. Evolution strategy. In J. Zurada et al, editor, *Computational Intelligence-Imitating Life*, pages 147–159. IEEE Press, 1994.

[24] N. Srinivas and K. Deb. Multiobjective optimization using nondominated sorting in genetic algorithms. *Evolutionary Computation*, 2(3):221–248, 1995.

[25] K. Wloch and P. J. Bentley. Optimising the performace of a formula one car using a genetic algorithm. In *Procs of the 8th International Conference on Parallel Problem Solving From Nature*, 2004.

[26] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1:67–82, 1997.

[27] E. Zitzler, K. Deb, and L. Thiele. Comparison of multiobjective evolutionary algorithms: Empirical results. *Evolutionary Computation*, 8(2):173–195, 2000.