COMPUTAÇÃO EVOLUTIVA

Grupo de Pesquisas em Computação Evolutiva

Aurora Pozo
Andrea de Fatima Cavalheiro
Celso Ishida
Eduardo Spinosa
Ernesto Malta Rodrigues
Departamento de Informática
Universidade Federal do Paraná

Conteúdo

1. INTRODUÇÃO

Computação Evolucionária (CE) é um ramo de pesquisa emergente da Inteligência Artificial que propõe um novo paradigma para solução de problemas inspirado na Seleção Natural (Darwin 1859).

A Computação Evolucionária compreende um conjunto de técnicas de busca e otimização inspiradas na evolução natural das espécies. Desta forma, cria-se uma população de indivíduos que vão reproduzir e competir pela sobrevivência. Os melhores sobrevivem e transferem suas características a novas gerações. As técnicas atualmente incluem (Banzhaf 1998): Programação Evolucionária, Estratégias Evolucionárias, Algoritmos Genéticos e Programação Genética. Estes métodos estão sendo utilizados, cada vez mais, pela comunidade de inteligência artificial para obter modelos de inteligência computacional (Barreto 1997).

Algoritmos Genéticos (AG) e Programação Genética (PG) são as duas principais frentes de pesquisa em CE. Os Algoritmos Genéticos (AG) foram concebidos em 1960 por John Holland (Holland 1975), com o objetivo inicial de estudar os fenômenos relacionados à adaptação das espécies e da seleção natural que ocorre na natureza (Darwin 1859), bem como desenvolver uma maneira de incorporar estes conceitos aos computadores (Mitchell 1997).

Os AGs possuem uma larga aplicação em muitas áreas científicas, entre as quais podem ser citados problemas de otimização de soluções, aprendizado de máquina, desenvolvimento de estratégias e fórmulas matemáticas, análise de modelos econômicos, problemas de engenharia, diversas aplicações na Biologia como simulação de bactérias, sistemas imunológicos, ecossistemas, descoberta de formato e propriedades de moléculas orgânicas (Mitchell 1997).

Programação Genética (PG) é uma técnica de geração automática de programas de computador criada por John Koza (Koza 1992), inspirada na teoria de AGs de Holland. Em PG é possível criar e manipular software geneticamente, aplicando conceitos herdados da Biologia para gerar programas de computador automaticamente.

A diferença essencial entre AG e PG é que em PG as estruturas manipuladas são bastante mais complexas, assim como várias das operações realizadas pelo algoritmo. Ambas as técnicas compartilham a mesma base teórica, inspirada na competição entre indivíduos pela sobrevivência, porém não mantêm vínculos de dependência ou subordinação.

PG e AGs representam um campo novo de pesquisa dentro da Ciência da Computação. Neste campo muitos problemas continuam em aberto e a espera de novas soluções e ferramentas. Apesar disso, este paradigma vem se mostrando bastante poderoso e muitos trabalhos vêm explorando o uso de AGs e PG para solucionar diversos problemas em diferentes áreas do conhecimento desde mineração de dados e biologia molecular até o projeto de circuitos digitais e inúmeras tarefas envolvendo otimização (GECCO 2000).

2. ALGORITMOS GENÉTICOS

O desenvolvimento de simulações computacionais de sistemas genéticos teve início nos anos 50 e 60 através de muitos biólogos, mas foi John Holland que começou a desenvolver as primeiras pesquisas no tema. Em 1975, Holland publicou "*Adaptation in Natural and Artificial Systems*", ponto inicial dos Algoritmos Genéticos (AGs). David E. Goldberg, aluno de Holland, nos anos 80 obteve seu primeiro sucesso em aplicação industrial com AGs. Desde então os AGs são utilizados para solucionar problemas de otimização e aprendizado de máquinas.

Esses algoritmos simulam processos naturais de sobrevivência e reprodução das populações, essenciais em sua evolução. Na natureza, indivíduos de uma mesma população competem entre si, buscando principalmente a sobrevivência, seja através da busca de recursos como alimento, ou visando a reprodução. Os indivíduos mais aptos terão um maior número de descendentes, ao contrário dos indivíduos menos aptos. Os requisitos para a implementação de um AG são:

- Representações das possíveis soluções do problema no formato de um código genético;
- População inicial que contenha diversidade suficiente para permitir ao algoritmo combinar características e produzir novas soluções;
- Existência de um método para medir a qualidade de uma solução potencial;
- Um procedimento de combinação de soluções para gerar novos indivíduos na população;
- Um critério de escolha das soluções que permanecerão na população ou que serão retirados desta;
- Um procedimento para introduzir periodicamente alterações em algumas soluções da população. Desse modo mantém-se a diversidade da população e a possibilidade de se produzir soluções inovadoras para serem avaliadas pelo critério de seleção dos mais aptos.

A idéia básica de funcionamento dos algoritmos genéticos é a de tratar as possíveis soluções do problema como "indivíduos" de uma "população", que irá "evoluir" a cada iteração ou "geração". Para isso é necessário construir um modelo de evolução onde os indivíduos sejam soluções de um problema. A execução do algoritmo pode ser resumida nos seguintes passos:

- Inicialmente escolhe-se uma população inicial, normalmente formada por indivíduos criados aleatoriamente;

- Avalia-se toda a população de indivíduos segundo algum critério, determinado por uma função que avalia a qualidade do indivíduo (função de aptidão ou "fitness");

- Em seguida, através do operador de "seleção", escolhem-se os indivíduos de melhor valor (dado pela função de aptidão) como base para a criação de um novo conjunto de possíveis soluções, chamado de nova "geração";

- Esta nova geração é obtida aplicando-se sobre os indivíduos selecionados operações que misturem suas características (chamadas "genes"), através dos operadores de "cruzamento" ("crossover") e "mutação";

- Estes passos são repetidos até que uma solução aceitável seja encontrada, até que o número predeterminado de passos seja atingido ou até que o algoritmo não consiga mais melhorar a solução já encontrada.

Os principais componentes mostrados na figura 1 são descritos a seguir em mais detalhes.

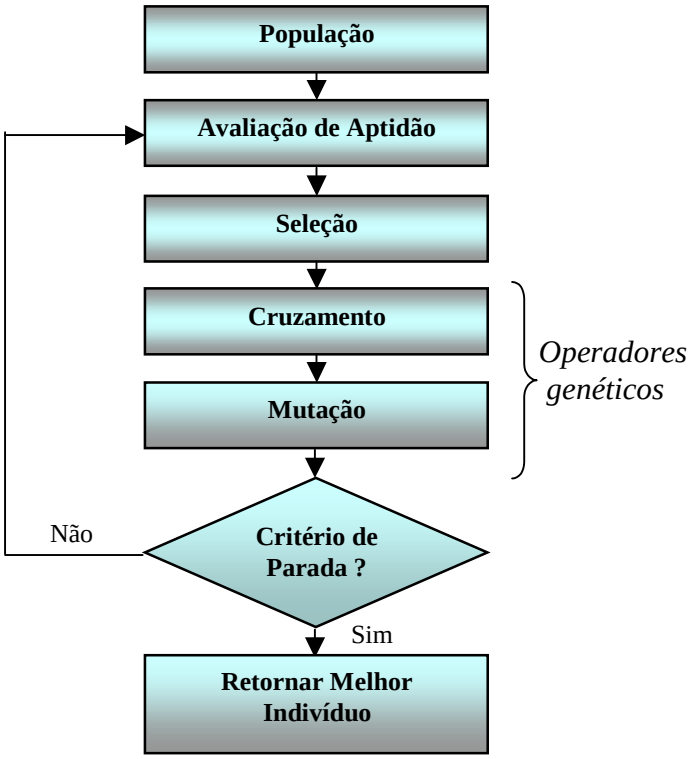

Figura 1 - Estrutura básica de um Algoritmo

2.1 População

A população de um algoritmo genético é o conjunto de indivíduos que estão sendo cogitados como solução e que serão usados para criar o novo conjunto de indivíduos para análise. O tamanho da população pode afetar o desempenho global e a eficiência dos algoritmos genéticos. Populações muito pequenas têm grandes chances de perder a diversidade necessária para convergir a uma boa solução, pois fornecem uma pequena cobertura do espaço de busca do problema. Entretanto, se a população tiver muitos indivíduos, o algoritmo poderá perder grande parte de sua eficiência pela demora em avaliar a função de aptidão de todo o conjunto a cada iteração, alem de ser necessário trabalhar com maiores recursos computacionais.

Indivíduos

O ponto de partida para a utilização de um algoritmo genético como ferramenta para solução de problemas é a representação destes problemas de maneira que os algoritmos genéticos possam trabalhar adequadamente sobre eles. Uma das principais formas é representar cada atributo como uma sequência de bits e o indivíduo como a concatenação das sequências de bits de todos os seus atributos. Outras variações de codificações binárias podem ser encontradas em (Holland 1975; Caruana 1988).

A codificação usando o próprio alfabeto do atributo que se quer representar (letras, códigos, números reais, etc.) para representar um indivíduo também é muito utilizada. Alguns exemplos podem ser encontrados em (Meyer 1992; Kitano 1994).

Diversas outras formas são possíveis, normalmente a forma mais apropriada está fortemente ligada ao tipo de problema.

2.2 Avaliação de Aptidão (Fitness)

Neste componente será calculado, através de uma determinada função, o valor de aptidão de cada indivíduo da população. Este é o componente mais importante de qualquer algoritmo genético. É através desta função que se mede quão próximo um indivíduo está da solução desejada ou quão boa é esta solução.

É essencial que esta função seja muito representativa e diferencie na proporção correta as más soluções das boas. Se houver pouca precisão na avaliação, uma ótima solução pode ser posta de lado durante a execução do algoritmo, além de gastar mais tempo explorando soluções pouco promissoras.

2.3 Seleção

Dada uma população em que a cada indivíduo foi atribuído um valor de aptidão, existe vários métodos para selecionar os indivíduos sobre os quais serão aplicados os

operadores genéticos. Há diversas formas de seleção, entre eles há o método de seleção por Roleta e o método de seleção por Torneio.

No método de seleção por Roleta (figura 2), cada indivíduo da população é representado na roleta proporcionalmente ao seu índice de aptidão. Assim, para indivíduos com alta aptidão é dada uma porção maior da roleta, enquanto aos indivíduos de aptidão mais baixa, é dada uma porção relativamente menor.

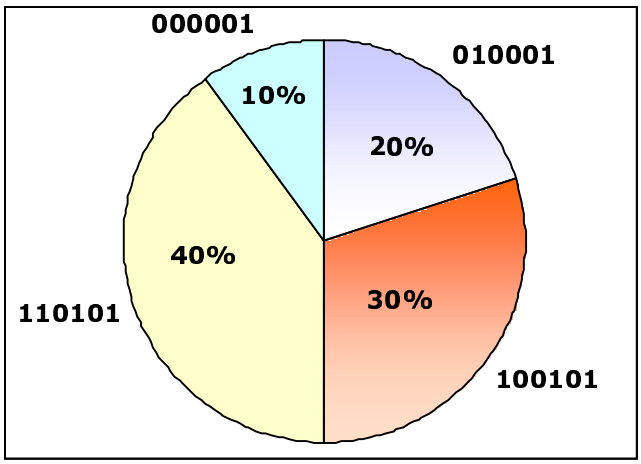

Figura 2 – Método de seleção por Roleta

Neste método, um dos problemas encontrados pode ser o tempo de processamento, já que o método exige duas passagens por todos os indivíduos da população.

Um exemplo da implementação deste método, segundo (Mitchell 1997) é mostrado a seguir na figura 3:

```
Inicio
        T = soma dos valores de aptidão de todos os indivíduos da população

        Repita N vezes para selecionar n indivíduos
                r =  valor aleatório entre 0 e T
                Percorra sequencialmente os indivíduos da população, acumulando
                em S o valor de aptidão dos indivíduos já percorridos
                Se  S  >=  r então
                        Selecione o indivíduo corrente
                Fim se

        Fim Repita
Fim
```

Figura 3 – Algoritmo básico do método de seleção por Roleta

Um outro método é a seleção por Torneio, onde um número *n* de indivíduos da população é escolhido aleatoriamente para formar uma sub-população temporária. Deste grupo, o indivíduo selecionado dependerá de uma probabilidade *k* definida previamente. Um exemplo básico da implementação deste algoritmo (Mitchell 1997) é mostrado na figura 4, onde n=2:

```
Inicio
        k = 0.75
        Repita N vezes
                Escolha 2 indivíduos da população aleatoriamente
                r =  valor aleatório entre 0 e 1
                Se  r < k
                        O melhor indivíduo é escolhido
                Senão
                        O pior indivíduo é escolhido
                Fim se
        Fim Repita
Fim
```

Figura 4 – Algoritmo básico do método de Seleção por Torneio

Este método é o mais utilizado, pois oferece a vantagem de não exigir que a comparação seja feita entre todos os indivíduos da população (Banzhaf 1998).

2.4 Operadores Genéticos

O principio básico dos operadores genéticos é transformar a população através de sucessivas gerações, estendendo a busca até chegar a um resultado satisfatório. Os operadores genéticos são necessários para que a população se diversifique e mantenha características de adaptação adquiridas pelas gerações anteriores. Os operadores de cruzamento e de mutação têm um papel fundamental em um algoritmo genético.

Cruzamento (Crossover)

Este operador é considerado o operador genético predominante. Através do cruzamento são criados novos indivíduos misturando características de dois indivíduos "pais". Esta mistura é feita tentando imitar (em um alto nível de abstração) a reprodução de genes em células. Trechos das características de um indivíduo são trocados pelo trecho equivalente do outro. O resultado desta operação é um indivíduo que potencialmente combine as melhores características dos indivíduos usados como base.

Alguns tipos de cruzamento bastante utilizados são o cruzamento em um ponto e o cruzamento em dois pontos, mostrados nas Figuras 5 e 6:

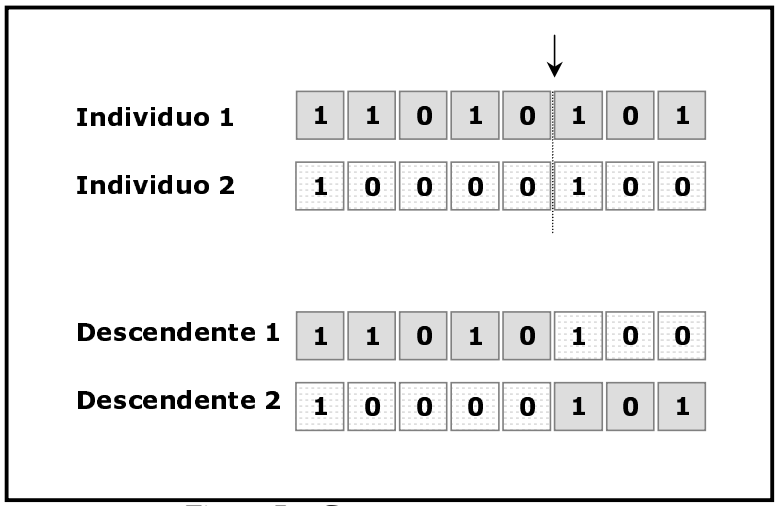

Figura 5 – Cruzamento em um ponto

Com um ponto de cruzamento, seleciona-se aleatoriamente um ponto de corte do cromossomo. Cada um dos dois descendentes recebe informação genética de cada um dos pais (Figura 5).

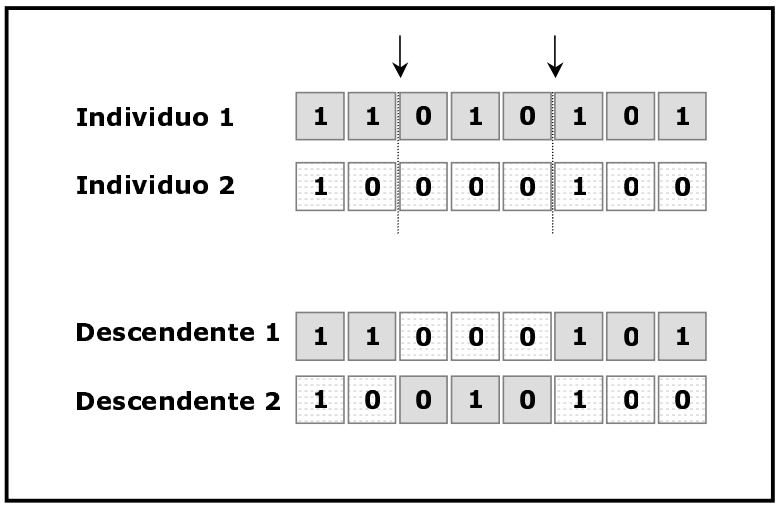

Figura 6 – Cruzamento em dois pontos

Com dois pontos de cruzamento, um dos descendentes fica com a parte central de um dos pais e as partes extremas do outro pai e vice versa (Figura 6).

Mutação

Esta operação simplesmente modifica aleatoriamente alguma característica do indivíduo sobre o qual é aplicada (ver Figura 7). Esta troca é importante, pois acaba por criar novos valores de características que não existiam ou apareciam em pequena quantidade na população em análise. O operador de mutação é necessário para a introdução e manutenção da diversidade genética da população. Desta forma, a mutação assegura que a probabilidade de se chegar a qualquer ponto do espaço de busca possivelmente não será zero. O operador de mutação é aplicado aos indivíduos através de uma taxa de mutação geralmente pequena.

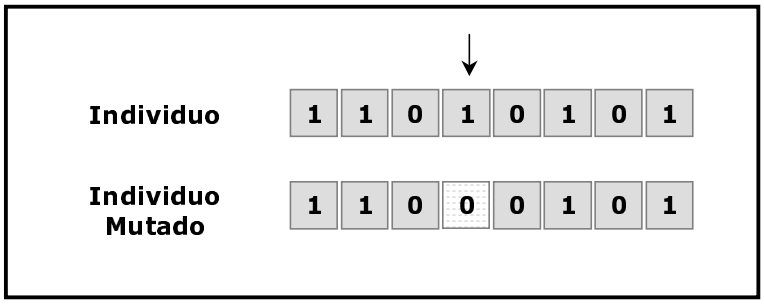

Figura 7 – Mutação Simples

2.5 Geração

A cada passo, um novo conjunto de indivíduos é gerado a partir da população anterior. A este novo conjunto dá-se o nome de "Geração". É através da criação de uma grande quantidade de gerações que é possível obter resultados dos Algoritmos Genéticos.

2.6 Considerações finais sobre AGs

Os algoritmos genéticos são apropriados para problemas complexos, mas algumas melhorias devem ser feitas no algoritmo básico. Muitas aproximações foram propostas com o objetivo comum de melhorar AGs. O primeiro grupo de estudos foca na manutenção da diversidade na população (De Jong 1989)(Eshelman 1991)(Goldberg 1989)(Goldberg 1990)(Tsutsui1993)(Tsutsui 1994) e inclui: métodos de compartilhamento de recursos que utilizam algumas funções sharing para evitar a convergência de indivíduos semelhantes, métodos crowding que obrigam a substituição de indivíduos novos, restrições de cruzamento, etc.

O segundo grupo visa melhorar o desempenho da capacidade de busca de algoritmos genéticos usando hibridização (Costa1995)(Glover 1994) (Glover 1995) (Kitano 1990)(Malek 1989)(Mantawy 1999)(Muhlenbein 1998)(Muhlenbein 1992)(Powel 1989)(Ulder 1991). Nesta abordagem algoritmos genéticos são usados com um dos seguintes paradigmas: busca tabu, redes neurais artificiais, simulated annealing, etc.

Entretanto, a maioria dos estudos na literatura têm focado na busca global através de AGs, enquanto a busca local tem sido feita por outros métodos.

O último grupo de estudos foca em problemas de funções de otimização, ou em problemas para encontrar soluções ótimas de Pareto (Cantu-Paz 1999)(Coelho 1999)(Schaffer 1985)(Srinivas 1993)(Tamaki 1996)(Fonseca 1993)(Hiroyasu 1999)(Horn 1993). Estes estudos incluem: métodos para dividir indivíduos em subgrupos, cada um representando uma função objetivo, combinação de torneio e métodos de compartilhamento de recursos, métodos para dividir soluções de Pareto em algumas áreas, entre outros.

3. TÉCNICAS PARA MANTER DIVERSIDADE POPULACIONAL EM ALGORITMOS GENÉTICOS

Um dos grandes problemas em algoritmos genéticos é o problema de convergência prematura, onde os genes de alguns indivíduos relativamente bem adaptados, contudo não ótimos, podem rapidamente dominar a população causando que o algoritmo convirja a um máximo local.

Para tentar escapar deste problema algumas técnicas podem ser utilizadas em conjunto com algoritmos genéticos, como é o caso do Compartilhamento de Recursos (Sharing), da Evolução Cooperativa e da Hibridização, descritos a seguir.

3.1 Compartilhamento de Recursos (Sharing)

A analogia da natureza é que dentro de um ambiente existem diferentes nichos que podem suportar diferentes tipos de vidas (espécies ou organismos). O número de organismos contidos dentro de um nicho é determinado pela fertilidade do nicho e pela eficiência de cada organismo para explorar essa fertilidade.

Cavicchio (1970), fez um dos primeiros estudos para tentar induzir nicho como comportamento em algoritmos genéticos. Ele introduziu um mecanismo, o qual chamou *preselection*. Nesse esquema, um descendente substitui o indivíduo pai se o valor de aptidão dele for maior que o valor de aptidão do pai. Desta maneira, a diversidade é mantida na população porque indivíduos tendem a substituir indivíduos similares a eles mesmos.

Outro mecanismo denominado *crowding*, foi proposto por De Jong (1975) para manter diversidade na população. Neste esquema, a substituição de indivíduos na população é modificada para fazer com que novos indivíduos substituam outros similares com menor valor para a função de aptidão dentro da população.

Goldberg e Richardson (Goldberg 1989) introduziram um mecanismo de compartilhamento de recursos, conhecido como Sharing. Neste mecanismo, o objetivo é reduzir o valor de aptidão de indivíduos que têm membros altamente similares dentro da população. Um esquema prático que diretamente usa sharing para induzir nicho e espécie é mostrada na figura 8. Neste esquema, uma função sharing é definida para determinar a vizinhança e o grau de compartilhamento para cada indivíduo da população.

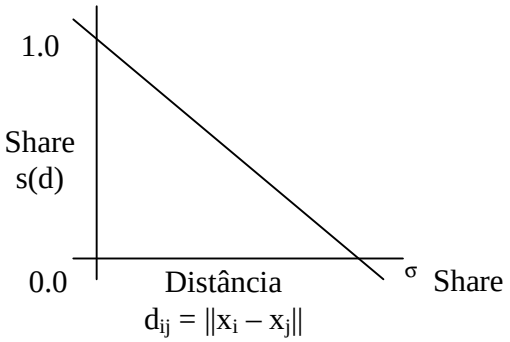

Figura 8 – Função Sharing Triangular (Goldberg 1989)

Para um determinado indivíduo o grau de compartilhamento é determinado somando os valores de função sharing contribuídos por todos os outros indivíduos na população. Indivíduos muito similares a outros indivíduos requerem um grau muito alto de compartilhamento, próximo a 1.0, e indivíduos menos similares requerem um grau muito pequeno de compartilhamento, próximo a 0.0. Se um indivíduo é idêntico a um outro indivíduo, seu grau de compartilhamento será igual a 1.0.

Depois de acumular o número total de compartilhamentos, o indivíduo que está sendo avaliado terá seu valor de aptidão reduzido, através da divisão de seu valor de aptidão pela soma acumulada do total de compartilhamentos.

$$f_s(x_i) = \frac{f(x_i)}{\sum_{j=1}^{I} s(d(x_i, x_j))}$$

Onde $f(x_i)$ é o valor de aptidão do indivíduo que está sendo avaliado, $s(d(x_i, x_j))$ é uma função de compartilhamento como a apresentada na figura 8.

Como resultado, este mecanismo limitará o crescimento descontrolado de espécies particulares dentro de uma população. A estrutura básica de um algoritmo genético com sharing pode ser vista a seguir:

```
Sharing()
{

     Iniciar a população aleatoriamente
     Avaliar o valor de aptidão dos indivíduos da população,
     reduzindo o valor de acordo com a quantidade de indivíduos
     identicos ou similares dentro da população

      Enquanto não atingir o número de gerações ou o objetivo do
      problema
      {
            Enquanto não atingir o número de indivíduos da
            população
            {
                  Selecionar indivíduos para reprodução
                  Aplicar operadores genéticos para produzir
                  descendentes
            }

            Substituir a população com os descendentes

            Avaliar o valor de aptidão dos indivíduos da nova
            população, reduzindo o valor de acordo com a quantidade
            de indivíduos idênticos ou similares dentro da
            população

      }
}
```

Figura 9 - Estrutura básica de um algoritmo genético com compartilhamento de recursos

Nesta técnica o algoritmo genético tradicional é modificado no módulo de avaliação de aptidão. Cada indivíduo tem seu valor de aptidão reduzido de acordo com a quantidade de indivíduos idênticos ou similares dentro da população. Com isto, o algoritmo tem uma menor probabilidade de que muitos indivíduos do mesmo nicho sejam selecionados, forçando uma maior diversidade populacional.

3.2 Evolução Cooperativa

Outra abordagem que lida com problemas complexos é a evolução cooperativa, proposta por Potter e De Jong (2000). Esta arquitetura modela um ecossistema consistindo de duas ou mais espécies. Nesta técnica, as espécies são geneticamente isoladas, ou seja, indivíduos somente cruzam com outros membros de sua espécie. Restrições de cruzamento são forçadas simplesmente por evoluir as espécies em populações separadas. As espécies interagem entre si dentro de um modelo de domínio compartilhado e têm um relacionamento cooperativo. O modelo básico desta abordagem é mostrado na figura 10:

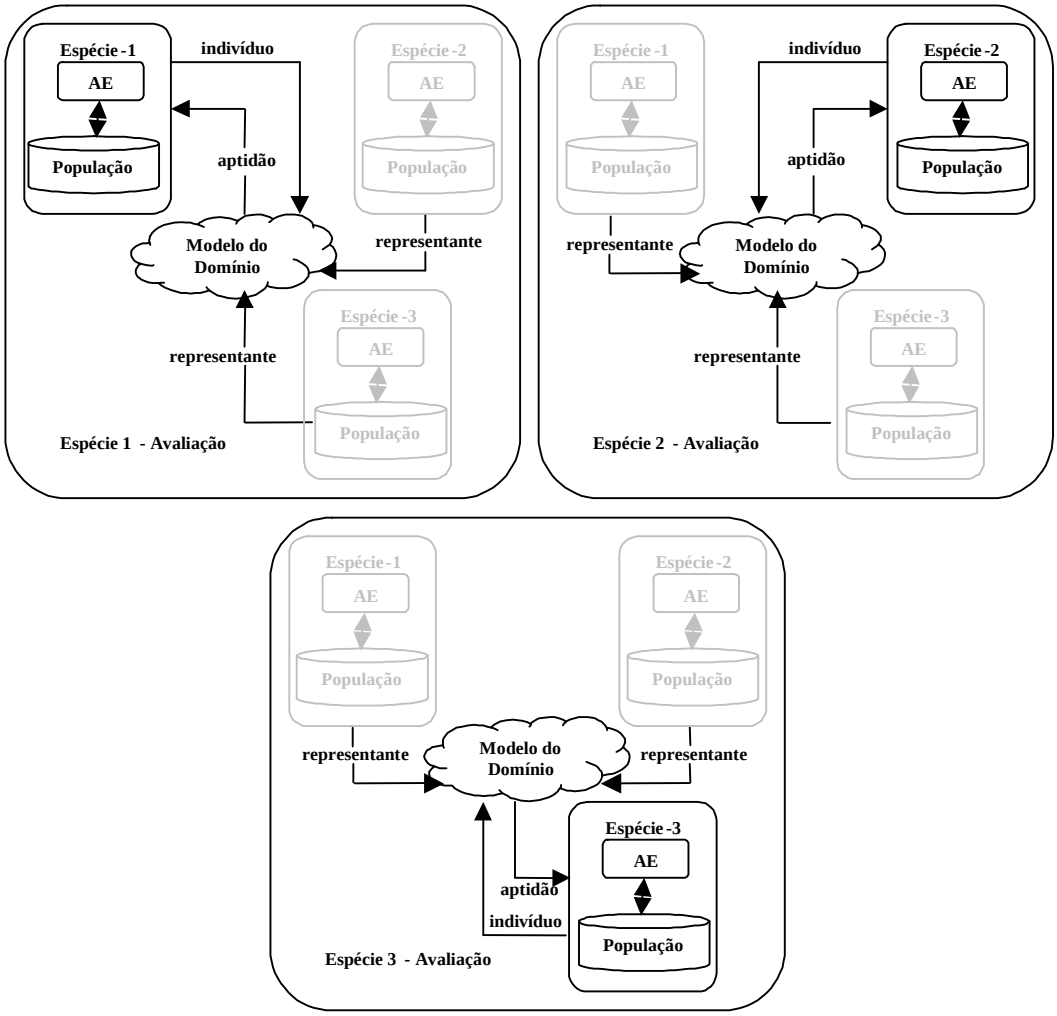

Figura 10 - Modelo de iteração de espécies (Potter 2000).

Neste modelo cada espécie é evoluída em sua própria população. A figura 10 mostra a fase de avaliação de aptidão de cada uma das três espécies. Para avaliar uma população são formadas colaborações com representantes de cada espécie.

Há muitos métodos possíveis para escolher os representantes com os quais colaborar. Em alguns casos é apropriado permitir que o melhor indivíduo corrente de cada população seja o representante. Em outros casos, estratégias alternativas são preferidas. Um algoritmo básico desta técnica pode ser visto na figura 11:

```
Inicio
{
      t = 0
      Para cada espécie S
      Inicializar Pt(S) com indivíduos aleatórios

      Para cada espécie S
      Avaliar o valor de aptidão de cada indivíduo em Pt (S)

      Repita Enquanto condição de termino é falsa
      {     Inicio
                  Para cada espécie S
                  Inicio
                        Selecionar indivíduos para reprodução de
                        Pt(S) baseado no  valor de aptidão
                        Aplicar operadores genéticos ao grupo de
                        reprodução para produzir descendentes
                        Avaliar o valor de aptidão dos descendentes
                        Substituir os membros de Pt (S) com os
                        descendentes para produzir Pt+1(S)
                  Fim
                  t = t + 1
            Fim
      }
}
Fim
```

Figura 11 - Algoritmo básico de Evolução Cooperativa (Potter 1997).

O algoritmo começa por criar um número fixo de populações. O valor de aptidão de cada membro de cada espécie é então avaliado. Se uma solução satisfatória para o problema objetivo não é encontrada inicialmente, todas as espécies são evoluídas.

Para cada espécie, o algoritmo consiste em selecionar indivíduos para reprodução baseados em seu valor de aptidão, como por exemplo através da seleção de fitness proporcional; aplicar operadores genéticos como cruzamento e mutação para criação de descendentes; avaliar o valor de aptidão dos descendentes e substituir membros da população velha com novos indivíduos.

O algoritmo de avaliação do valor de aptidão dos indivíduos em uma das espécies é mostrado a seguir:

```
Inicio
{
      Escolher representante de cada uma das outras espécies

      Para cada indivíduo i de S faça avaliação

      Inicio
            Formar colaboração entre i e representantes de outras
            espécies
            Avaliar o valor de aptidão de colaboração através do
            problema objetivo
            Atribuir o valor de aptidão de colaboração a  i
      Fim
}
Fim
```

Figura 12 - Avaliação de aptidão dos indivíduos da espécie S (Potter 1997).

Os indivíduos não são avaliados isoladamente, eles são combinados primeiro em algum domínio dependente com um representante de cada uma das outras espécies. Potter e De Jong se referem a isto como uma colaboração porque os indivíduos serão julgados no final em quão bem eles trabalham juntos para resolver o problema objetivo.

Se a evolução estagnar, pode ser que existam poucas espécies no ecossistema com o qual construir uma boa solução, então uma nova espécie será criada e sua população aleatoriamente inicializada. A estagnação pode ser descoberta monitorando a qualidade das colaborações pela aplicação da seguinte equação:

$$f(t) - f(t - K) < C,$$

Onde $f(t)$ é o valor de aptidão da melhor colaboração no tempo t , C é uma constante especificando o aumento do valor de aptidão, considerando ter uma melhoria significante, e K é uma constante especificando o tamanho de uma janela evolutiva na qual uma melhoria significante deve ser feita. Um resumo do algoritmo implementado para esta técnica pode ser visto na figura a seguir:

```
Evolução Cooperativa()
{
      Iniciar cada população com indivíduos aleatórios
      Avaliar o valor de aptidão dos indivíduos de cada população
      Enquanto não atingir o número de gerações ou o objetivo do
      problema
      {     Para cada população Repita Enquanto não atingir o
            número de indivíduos da população
            {
                  Selecionar indivíduos para reprodução
                  Aplicar operadores genéticos p/produzir
                  descendentes
            }

            Avaliar o valor de aptidão dos descendentes de cada
            população
            Substituir a população com os descendentes

            A cada geração
            {
                  Escolher representantes de cada população
                  Avaliar o valor de aptidão de colaboração para o
                  indivíduo i da população X, verificando se i está
                  colaborando para atingir o problema objetivo.
                  Atribuir o valor de aptidão de colaboração
            }
            A cada n gerações
            {
                  Se população X não está contribuindo ou está
                  convergindo para um mesmo padrão
                  {
                     Iniciar população com indivíduos aleatórios
                     Avaliar o valor de aptidão dos indivíduos
                  }
            }
      }
}
```

Figura 13 - Estrutura básica do algoritmo de Evolução Cooperativa

No algoritmo, a cada geração é feita uma cooperação entre as populações, onde é atribuído um valor de aptidão para a população que está sendo avaliada. A cada n gerações,

se a população que está sendo avaliada possui um valor de aptidão muito baixo, essa população é descartada, e então criada uma nova população, a qual irá substituí-la nas próximas gerações.

3.3 Abordagens Hibridas

Os algoritmos geneticos tradicionais, apesar de robustos, não são os algoritmos de melhor comportamento em otimização para qualquer dominio. Na hibridização algum outro método de otimização é utilizado em conjunto com AGs, por exemplo “Hill-Climbing”, Busca Tabu, etc.... Nesta seção será apresentada uma estrategia deste tipo utilizando conceitos da Busca Tabu.

Busca Tabu (*“proibido”*) é um procedimento heurístico proposto por Glover para resolver problemas de otimização combinatória. A idéia básica é evitar que a busca por soluções ótimas termine ao encontrar um mínimo local (Glover 1986). Este tipo de algoritmo faz uma busca agressiva no espaço de soluções do problema de otimização com o intuito de obter sempre as melhores alternativas que não sejam consideradas tabu. A heurística Busca Tabu algumas vezes aceita a solução considerado tabu (proibida), baseado no critério de aspiração que determina quando as restrições tabu podem ser ignoradas. Para melhor visualização, a figura 14 mostra um esquema geral do processo de busca Tabu, onde *N(x)* denota o conjunto de soluções vizinhas à *x* no espaço de busca e *T* representa a lista Tabu. Para implementação do algoritmo Busca Tabu, alguns elementos básicos devem ser especificados, tais como:

- Movimentos: operadores utilizados para transformar uma solução em outra;
- Lista Tabu: onde serão armazenadas todas as soluções anteriores durante o processo de busca do algoritmo. Essa lista é introduzida, no sentido de guardar características dos movimentos realizados, para evitar possíveis retornos a soluções já visitadas;

- Critério de Aspiração: determinará quando uma restrição Tabu deve ser ignorada, realizando o movimento independente se classificado como proibido. Um critério de aspiração comum é ignorar a restrição talvez quando isso produzir uma solução melhor do que todas as soluções geradas anteriormente;
- Término: o processo deve ser finalizado quando não existir mais nenhum movimento possível a ser realizado, ou quando atingir o número máximo de iterações definidas pelo usuário;
- Parâmetros: deve ser informado o tamanho da lista de restrições, número máximo de iterações, regras de parada, solução inicial e critério de aspiração.

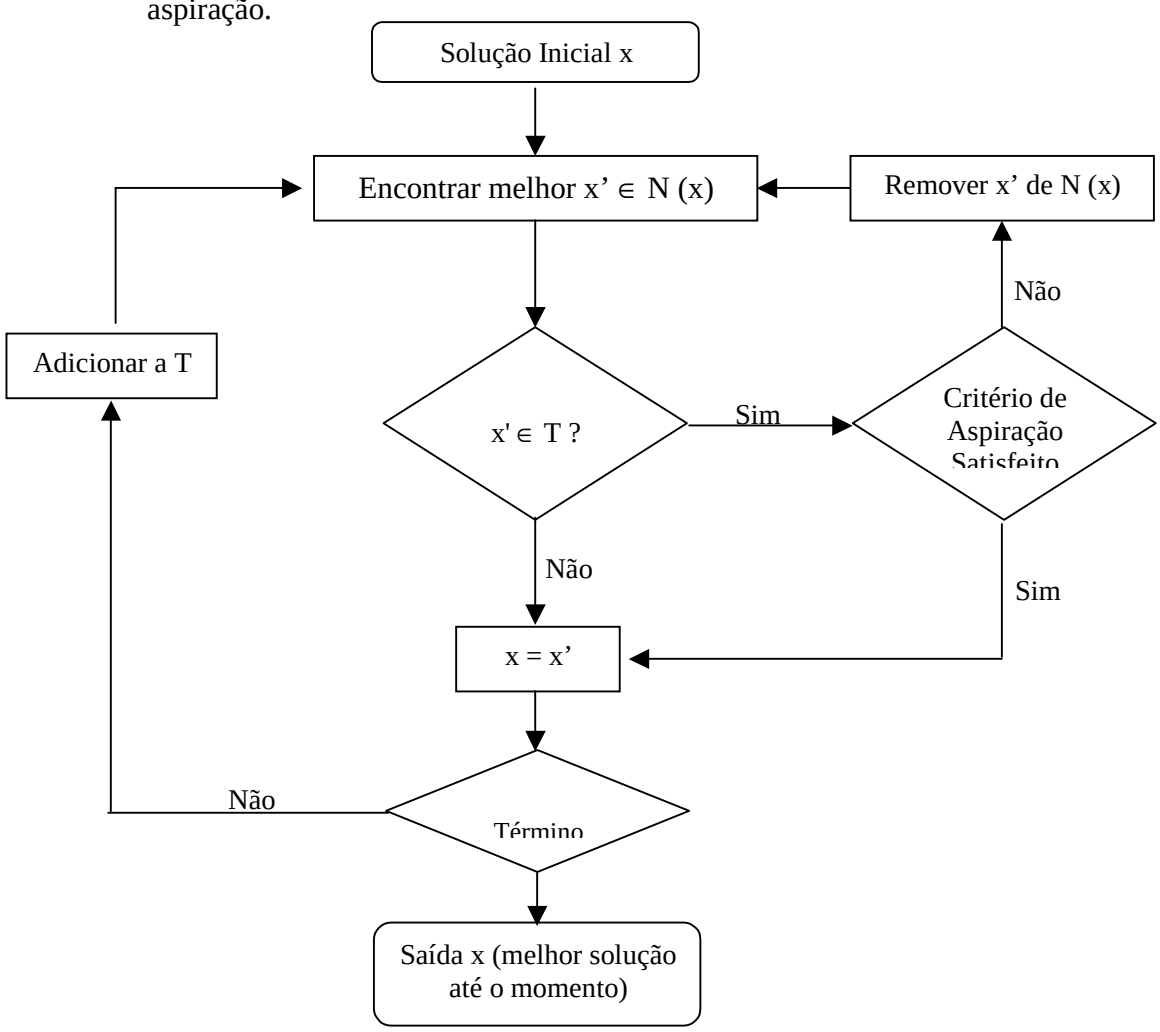

Figura 14 – Método de Busca Tabu (Krishnamachari 1999)

O processo inicia selecionando uma solução aleatória *(x)*. Uma busca local é então realizada, procurando todas as soluções vizinhas, *N(x)*. A partir dessas soluções, seleciona-se a melhor solução (*x'*), não sendo necessário que a mesma seja melhor que a solução inicial. A solução inicial é então movida para a melhor solução vizinha adicionando-se a nova solução à lista Tabu. A partir dessa nova solução, realiza-se novamente uma busca local e novamente a melhor solução vizinha é selecionada como candidata para o próximo movimento.

Para evitar que um movimento reverso seja realizado, verificam-se os movimentos das iterações anteriores armazenados na lista de restrições. Se o movimento não se encontra na lista Tabu ou se satisfeito o critério de aspiração, o movimento é aceito, senão testa-se a próxima melhor solução. Esse processo é executado até encontrar uma solução vizinha que não se encontre na lista Tabu. Enquanto não satisfeito um critério de término, o processo é repetido.

Kurahashi e Terano (2000) propõem um Algoritmo Genético utilizando-se de múltiplas listas Tabu, auxiliando o algoritmo a alcançar a solução em problemas multimodais ou com mais de uma função objetivo a otimizar.

A maioria dos métodos convencionais utiliza algoritmos genéticos para explorar candidatos globais e algoritmos adicionais para explorar pontos ótimos locais. O trabalho de Kurahashi e Terano propõe um novo algoritmo para diretamente armazenar indivíduos dentro de múltiplas listas.

A idéia geral do algoritmo é a utilização de duas listas de restrições: Lista Longa, com tamanho *m* e Lista Curta, com tamanho *n*, onde *m* e *n* são ajustados de acordo com o problema. Estas listas terão o objetivo de armazenar os melhores indivíduos das gerações anteriores, manter o elitismo, manter a diversidade populacional e evitar a convergência a um ponto ótimo local.

A dinâmica dos algoritmos baseia-se na idéia que ao final de cada geração, o melhor indivíduo será armazenado em ambas as listas. Quando se inicia a próxima geração e novos indivíduos são selecionados para a reprodução, as listas de restrições não deixam que indivíduos similares presentes nas mesmas sejam selecionados. Estas listas de restrições

podem ser aplicadas para somente um dos indivíduos escolhidos para gerar os descendentes.

Na Lista Curta serão armazenados apenas os indivíduos das iterações mais recentes. Quando a lista é preenchida completamente, para que um novo indivíduo seja adicionado à mesma, o indivíduo mais antigo deverá ser retirado. Os indivíduos pertencentes a esta lista podem ter o mesmo genótipo.

Na Lista Longa, serão armazenados indivíduos de todas as gerações anteriores. Os indivíduos presentes na lista não poderão ter genótipo idêntico ou similar. Caso surja um indivíduo com um genótipo similar a ser adicionado na Lista Longa, este somente será adicionado, se possuir um valor de aptidão superior e mediante a retirada do outro indivíduo. Este indivíduo retirado da lista sofrerá mutação e será recolocado na população a fim de participar das próximas gerações.

Assim, as soluções serão gradualmente armazenadas na Lista Longa, ou seja, até no máximo *m* soluções. Ao final do processo, o conjunto solução, será formado pelos indivíduos presentes na Lista Longa.

Testes de otimizações de funções realizados por este algoritmo foram feitos por Kurahashi e Terano (2000). Algumas de suas conclusões parciais relatam que, com a adoção desta estratégia, o algoritmo genético utilizando listas de restrições conseguiu cobrir uma área maior do espaço de busca, se comparado a um algoritmo genético puro. Para evitar que as soluções convirjam a um pico em uma função multimodal, eles definem uma medida de distancia entre um indivíduo na lista tabu e um novo candidato. São empregadas três medidas de distância, entre elas a distância de Hamming, onde é calculada a diferença de bits entre dois genótipos (conforme formula a seguir), e que será utilizada na implementação desse algoritmo para comparação com as outras técnicas.

$$d_H(a,b) = \sum_{i=1}^{n} |a_i - b_i|$$

.

```
AG Restrito()
{     Iniciar população aleatoriamente
      Avaliar o valor de aptidão dos indivíduos da população
      Enquanto não atingir o número de gerações ou o objetivo
      do problema
      {     Selecionar o melhor indivíduo da população

            Retornar indivíduo da lista longa que seja similar ao
            melhor indivíduo da população

            Se limiar de distância entre o melhor indivíduo
            da população e o indivíduo similar da lista < d
                  Inserir indivíduo na Lista Longa
            Senão
                  Verificar se indivíduo selecionado possui valor de
                  aptidão > que indivíduo da lista
                  Se sim
                              Retirar indivíduo da lista
                              Aplicar mutação
                              Colocar indivíduo retirado da lista
                              longa na nova população
                              Colocar indivíduo selecionado na Lista
                              Longa
                  Senão
                              Descartar indivíduo selecionado
            }
            Se Lista Curta não estiver completa
                  Inserir indivíduo na Lista Curta
            Senão
            {
                  Descartar o indivíduo mais antigo
                  Inserir novo indivíduo na Lista Curta
            }
            Enquanto não atingir o número de indivíduos da
            população
            {     Selecionar indivíduos para reprodução e verificar
                  se o primeiro indivíduo selecionado não possui
                  indivíduos idênticos ou similares na Lista Curta
                  ou na Lista Longa
                  Aplicar operadores genéticos para produzir
                  descendentes
            }
            Avaliar o valor de aptidão dos indivíduos da nova
            população
            Substituir a população com os descendentes
}
```

Figura 15 - Estrutura básica do algoritmo genético restrito

Quando indivíduos gerados por operações de algoritmos genéticos são selecionados via o método de seleção por torneio, apenas um deles é comparado com os indivíduos pertencentes à lista tabu (Kurahashi 2000). Se a diferença de bits entre o indivíduo selecionado e cada um dos indivíduos da lista está dentro de uma distância pré-definida, os dois indivíduos selecionados são descartados e uma nova seleção é executada

A estrutura básica deste algoritmo pode ser vista na Figura 15. No AG Restrito é possível visualizar as modificações que foram efetuadas no algoritmo genético tradicional para implementação desta técnica. Após a população ter sido criada aleatoriamente e o valor de aptidão ter sido calculado, o melhor indivíduo da população é selecionado para fazer parte das listas de restrições. Obedecidos aos critérios de inserção na lista longa e na lista curta, o próximo passo é selecionar indivíduos para reprodução, verificando se o primeiro indivíduo selecionado não possui indivíduos idênticos ou similares nas listas. Caso negativo, aplicam-se operadores genéticos, senão dois novos indivíduos são selecionados. Este processo é repetido até que o número de indivíduos na nova população seja atingido. Por fim, o valor de aptidão dos indivíduos é calculado e a população substituída pelos novos descendentes.

4. PROGRAMAÇÃO GENÉTICA

Neste capítulo é fornecida uma explicação sobre a origem, funcionamento e principais componentes da Programação Genética e algumas de suas extensões. Vários aspectos importantes são detalhados, tais como a geração da população inicial e avaliação dos programas.

O paradigma da Programação Genética foi desenvolvido por John Koza (Koza 1989; Koza 1992) com base nos trabalhos de John Holland em Algoritmos Genéticos (Holland 1975). Atualmente representa uma área muito promissora de pesquisa em Inteligência Artificial devido a sua simplicidade e robustez. Seu uso tem sido estendido a problemas de diversas áreas do conhecimento, como por exemplo: biotecnologia, engenharia elétrica, análises financeiras, processamento de imagens, reconhecimento de padrões, mineração de dados, linguagem natural, dentre muitas outras (Willis 1997)

A Programação Genética é a evolução de um conjunto de programas com o objetivo de aprendizagem por indução (Banzhaf 1998). A idéia é ensinar computadores a se programar, isto é, a partir de especificações de comportamento, o computador deve ser capaz de induzir um programa que as satisfaça (Koza 1992). A cada programa é associado um valor de mérito (*fitness*) representando o quanto ele é capaz de resolver o problema.

Basicamente, a Programação Genética mantém uma população de programas de computador, usa métodos de seleção baseados na capacidade de adaptação (*fitness*) de cada programa (escolha dos “melhores”), aplica operadores genéticos para modificá-los e convergir para uma solução. O objetivo é encontrar uma solução no espaço de todos os programas possíveis (candidatos) usando apenas um valor de *fitness* como auxílio no processo de busca (Gathercole 1998).

O mecanismo de busca da Programação Genética pode ser descrito como um ciclo “criar-testar-modificar” (Figura 16), 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 (Yu 1999).

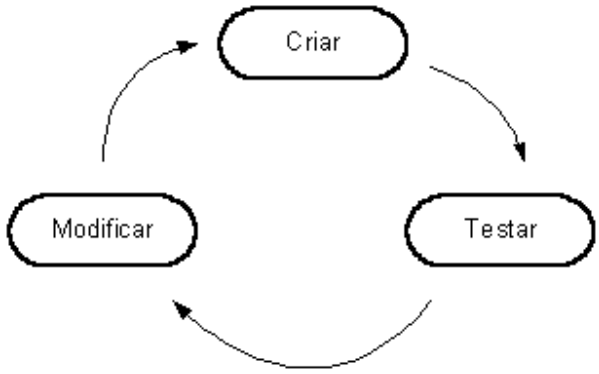

Figura 16: Ciclo “criar-testar-modificar”

A especificação de comportamento é feita normalmente através de um conjunto de valores de entrada-saída, denominados *fitness cases*, representando o conjunto de aprendizagem ou treinamento (*training set*). Com base neste conjunto, a Programação Genética procura obter um programa que: (O’Reilly 1995)

- Produza, de forma não trivial, as saídas corretas para cada entrada fornecida. Isto implica que o programa não deve mapear as entradas e saídas através de alguma forma de tabela de conversão. Portanto, o programa deverá aprender necessariamente alguma forma de algoritmo;

- Calcule as saídas de tal forma que, se as entradas forem representativamente escolhidas, o programa será capaz de produzir saídas corretas para entradas não cobertas inicialmente.

Por manipular programas diretamente, a Programação Genética lida com uma estrutura relativamente complexa e variável. Tradicionalmente, esta estrutura é uma árvore de sintaxe abstrata composta por funções em seus nós internos e por terminais em seus nós-folha. A especificação do domínio do problema é feita simplesmente pela definição dos conjuntos de funções e terminais (Koza 1992).

4.1 Visão Geral do Algoritmo de Programação Genética

O algoritmo de Programação Genética é simples e pode ser descrito resumidamente como:

- Criar aleatoriamente[1] uma população de programas;
- Executar os seguintes passos até que um Critério de Término seja satisfeito:
    - Avaliar cada programa através de uma função heurística (*fitness*), que expressa quão próximo cada programa está da solução ideal;
    - Selecionar os melhores programas de acordo com o *fitness*;
    - Aplicar a estes programas os operadores genéticos (reprodução, cruzamento e mutação)
- Retornar com o melhor programa encontrado

Cada execução deste laço representa uma nova geração de programas. Tradicionalmente, o *Critério de Término* é estabelecido como sendo encontrar uma solução satisfatória ou atingir um número máximo de gerações (Koza 1992). Porém, existem abordagens baseadas na análise do processo evolutivo, isto é, o laço permanece enquanto houver melhoria na população (Kramer 2000).

[1] A geração inicial representa uma “busca cega" pela solução.

A estrutura básica do algoritmo de Programação Genética é mostrada na Figura 17.

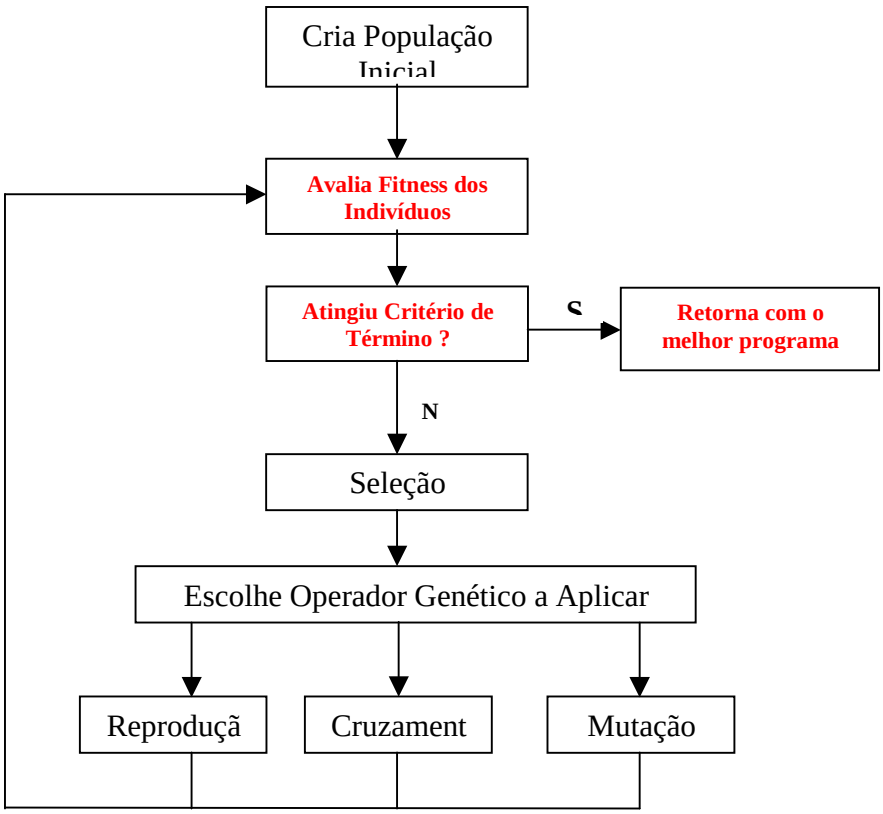

Figura 17: Estrutura Básica do Algoritmo de Programação Genética.

4.2 Representação dos Programas

A representação dos programas em Programação Genética tradicionalmente se baseia em árvore de sintaxe abstrata, isto é, os programas são formados pela livre combinação de funções e terminais adequados ao domínio do problema.

Parte-se de dois conjuntos: *F* como sendo o conjunto de funções e *T* como o conjunto de terminais. O conjunto *F* pode conter operadores aritméticos (`+`, `-`, `*` etc.), funções matemáticas (`seno`, `log` etc.), operadores lógicos (`E`, `OU` etc.) dentre outros. Cada *f*

$\in F$ tem associada uma aridade (número de argumentos) superior a zero. O conjunto T é composto pelas variáveis, constantes e funções de aridade zero (sem argumentos).

Por exemplo, considerando o conjunto dos operadores aritméticos de aridade dois (2) como sendo o conjunto de funções e a variável *x* e a constante dois (2) como terminais, isto é:

$$F = \{ +, -, *, / \}$$

$$T = \{ x, 2 \}$$

então expressões matemáticas simples tais como `x*x+2` podem ser produzidas. A representação é feita por uma árvore de sintaxe abstrata como mostrado na Figura 18.

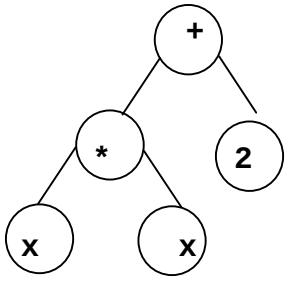

Figura 18: Árvore de Sintaxe Abstrata de `x*x+2`

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

4.3 Fechamento e Suficiência

Para garantir a viabilidade das árvores de sintaxe abstrata, John Koza definiu a propriedade de Fechamento (*closure*) (Koza 1992). Para satisfazê-la, cada função do conjunto *F* 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 1992). 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, John Koza definiu a propriedade de Suficiência (*sufficiency*) onde os conjuntos de funções *F* e o de terminais *T* devem ser capazes de representar uma solução para o problema (Koza 1992). 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.

4.4 População Inicial

Tradicionalmente, a população inicial é composta por árvores geradas aleatoriamente a partir dos conjuntos de funções *F* e de terminais *T*. Inicialmente se escolhe aleatoriamente uma função $f \in F$. Para cada um dos argumentos de *f*, escolhe-se um elemento de $\{ F \cup T \}$. 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 muitos grandes.

Porém, a “qualidade” da população inicial é um fator crítico para o sucesso do processo evolutivo (Daida 1999). A população inicial 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.

Para melhorar a qualidade dos programas gerados na população inicial, há diversos métodos, sendo os mais comuns (Luke 2001): *ramped-half-and-half* (Koza 1992), *random-branch* (Chellapilla 1997), *uniform* (Bohm 1996) e, mais recentemente, *probabilistic tree-creation* (Luke 2000).

O método *ramped-half-and-half* (Koza 1992) é uma combinação de dois métodos simples: *grow* e *full*. O método *grow* envolve a criação de árvores cuja profundidade[2] é variável. A escolha dos nós é feita aleatoriamente entre funções e terminais, respeitando-se uma profundidade máxima. O algoritmo é muito simples e está na Figura 19.

```
Dados:
        Profundidade máxima D
        Conjunto de funções F e de terminais T
Faça:
        Árvore = GROW(0)

GROW(profundidade d)
        Retorna: uma árvore de profundidade máxima ≤ D - d

        Se d = D
                Retorne aleatoriamente um terminal de T
        Senão
                Escolha aleatoriamente um elemento e ∈ { F ∪ T }
                Se e ∈ F
                        Para cada argumento a de e
                                Preencha a com GROW(d + 1)

        Retorne f com todos os argumentos preenchidos
```

Figura 19: Algoritmo Grow

[2] A profundidade de um nó *n* em uma árvore é o comprimento de caminho da raiz até *n*. A profundidade de uma árvore é o nó de maior profundidade (Terada 1991)

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* (Koza 1992). 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*.

As desvantagens deste método são (Luke 2000):

- 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[3] 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* (Chellapilla 1997) permite que se informe qual o tamanho máximo da árvore (e não a sua profundidade). O algoritmo está na Figura 20.

[3] O tamanho de uma árvore é o número de nós que a compõem (Koza 1992).

```
RANDOM-BRANCH(tamanho máximo desejado S)
Retorna: uma árvore de tamanho ≤ S

    Se um não-terminal com aridade ≤ S não existe
        Retorne aleatoriamente um terminal
Senão
        Escolha aleatoriamente um não-terminal n com aridade ≤ S
    Seja b_n a aridade de n
    Para cada argumento a de n
            Preencha a com RANDOMBRANCH(⌊a / b_n⌋)
    Retorne n  com todos os argumentos preenchidos
```

Figura 20: Algoritmo Random-Branch

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 (Luke 2001).

O método *uniform* foi desenvolvido por Bohm com o objetivo de garantir que as árvores são geradas uniformemente do conjunto de todas as árvores possíveis (Bohm 1996). O algoritmo é extremamente complexo, pois necessita calcular em várias tabelas o número de árvores possíveis de serem geradas para cada tamanho desejado. A desvantagem deste método é o seu alto custo computacional. Um exemplo de aplicação deste método é a ferramenta GPK de Helmut Horner (Horner 1996).

Os métodos *probabilistic tree-creation* (PTC) 1 e 2 (Luke 2000), 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 $t \in T$, 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}. A obtenção do valor de p é feita pela fórmula a seguir:

$$p = \frac{1 - \frac{1}{E_{tree}}}{\sum_{n \in N} q_n b_n} \quad \text{onde } b_n \text{ é a aridade do não-terminal } n \qquad (1)$$

O algoritmo do PTC1 está na Figura 21

```
Dados:
        Profundidade máxima D
        Conjunto de funções F e de terminais T
        A probabilidade p de escolher uma função
        As probabilidades q_t e q_f para cada t ∈ T e f ∈ F
Faça:
        Árvore = PTC1(0)

PTC1(profundidade d)
        Retorna: uma árvore de profundidade máxima ≤ D - d

        Se d = D
                Retorne aleatoriamente um terminal de T baseado em q_t
        Senão
                Com probabilidade p, Se uma função deve ser escolhida
                        Escolha aleatoriamente f ∈ F baseado em q_f
                        Para cada argumento a de e
                                Preencha a com PTC1(d + 1)
                        Retorne f com todos os argumentos preenchidos
                Senão
                        Retorne aleatoriamente um terminal de T baseado em q_t
```

Figura 21: Algoritmo PTC1

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, ...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.

4.5 Função de Aptidão

Na natureza os seres vivos são selecionados naturalmente com base no seu grau de adaptabilidade ao meio ambiente. Em Programação Genética, isto é expresso pela função de aptidão ou *fitness*. Os programas que melhor resolverem o problema receberão melhores valores de *fitness* e, consequentemente, terão maior chance de serem selecionados para reproduzir.

A avaliação de *fitness* depende do domínio do problema e pode ser medida de diversas formas, tanto direta quanto indiretamente. Para fins deste trabalho, apenas os domínios que permitam uma avaliação direta de *fitness* são considerados..

Usualmente, para se proceder à avaliação de *fitness*, é fornecido um conjunto de casos de treinamento, denominados *fitness cases*, contendo valores de entrada e saída a serem aprendidos. A cada programa é fornecido os valores de entrada e confronta-se a sua resposta ao valor esperado de saída. Quanto mais próxima a resposta do programa estiver do valor de saída, melhor é o programa.

Desta forma, a avaliação de *fitness* estabelece uma forma de se diferenciar os melhores dos piores, servindo como a força mestre do processo evolutivo, sendo a medida (usada durante a evolução) do quanto o programa aprendeu a predizer as saídas das entradas dentro de um domínio de aprendizagem (Banzhaf 1998).

A escolha da função de *fitness*, assim como a escolha do método de avaliação utilizado por esta função, depende do problema. Boas escolhas são essenciais para se obterem bons resultados, já que a função de *fitness* é a força-guia que direciona o algoritmo de Programação Genética na busca pela solução (Gritz 1993).

Os métodos comumente usados para avaliação de *fitness* são (Koza 1992):

1) **Aptidão nata** (*raw fitness*): representa a medida dentro do próprio domínio do problema. É a avaliação pura e simples do programa frente aos *fitness cases*. O método

mais comum de aptidão nata 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.

2) **Aptidão padronizada** (*standardized fitness)*: Devido ao fato da aptidão nata depender do domínio do problema, uma valor bom pode ser um valor pequeno (quando se avalia o erro) ou um valor grande (quando se avalia a taxa de eficiência). A avaliação da aptidão padronizada é feita através de uma função de adaptação do valor da aptidão nata de forma que quanto melhor o programa, menor deve ser a aptidão padronizada . Desta forma, o melhor programa apresentará o valor zero (0) como aptidão padronizada, independentemente do domínio do problema.

3) **Aptidão ajustada** (*adjusted fitness*): é obtida através da aptidão padronizada. Se *s(i, t)* representa a aptidão padronizada do indivíduo *i* na geração *t*, então a aptidão ajustada *a(i, t)* é calculada da seguinte forma:

$$a(i,t) = \frac{1}{1+s(i,t)} \qquad (2)$$

Percebe-se que a aptidão ajustada varia entre zero (0) e um (1), sendo que os maiores valores representam os melhores indivíduos. A aptidão ajustada tem o benefício de exagerar a importância de pequenas diferenças no valor da aptidão padronizada quando esta se aproxima de zero (Koza 1992).

4) **Aptidão normalizada** *(normalized fitness*): se *a(i, t)* é a aptidão ajustada do indivíduo *i* na geração *t*, então sua aptidão normalizada *n(i, t)* será obtida da seguinte forma:

$$n(i,t) = \frac{a(i,t)}{\sum_{k=1}^{m} a(k,t)} \qquad (3)$$

É fácil perceber que a soma de todas as aptidões normalizadas dentro de uma população vale um (1).

Para uma melhor comprensão de como pode ser feita a avaliação de *fitness*, suponha os seguintes valores de *fitness cases* mostrados na Tabela 1.

	ENTRADA	SAÍDA
FITNESS CASE 1	0	1
FITNESS CASE 2	2	5
FITNESS CASE 3	4	17
FITNESS CASE 4	6	37
FITNESS CASE 5	8	65

Tabela 1: Conjunto de Fitness Cases

Com base neste conjunto, deseja-se descobrir um programa que seja capaz de produzir as saídas para cada entrada informada. É fácil perceber que a função $f(x) = x^2+1$ é uma solução válida neste caso.

A aptidão nata (*raw fitness*) para este tipo de problema pode ser a soma das diferenças absolutas da resposta do programa pela saída correta (*Minkowski distance*). Para cada programa p pertencente a população P, associa-se um valor f_p que representa o seu *fitness* obtido na avaliação dos n *fitness cases* informados. O valor de f_p é obtido pela fórmula:

$$f_p = \sum \left| p_i - s_i \right| \qquad (4)$$

Onde p_i representa a resposta do programa ao i-ésimo *fitness case* e s_i, a saída correta. Quanto mais perto o valor de p_i estiver de s_i, menor será o valor de f_p e melhor será o programa. Neste caso, esta avaliação de *fitness* também é considerada uma aptidão padronizada (*standardized fitness*).

Em algumas aplicações, é interessante reforçar a diferença entre os diversos valores de *fitness* de uma população. Uma variação muito comum é somar o quadrado das diferenças (*squared error*), como a fórmula a seguir:

$$f_p = \sum_{i=1}^{n} (p_i - s_i)^2 \qquad (5)$$

Qual o real impacto do uso destas funções de *fitness*? Para melhor esclarecer, considere que o programa $x^2 + x$ está sendo avaliado. A Tabela 2 mostra os resultados obtidos em cada uma das funções para os valores da Tabela 1.

	ENTRADA	SAÍDA	PROGRAMA	ERRO ABSOLUTO	ERRO QUADRÁTICO
FITNESS CASE 1	0	1	0	1	1
FITNESS CASE 2	2	5	6	1	1
FITNESS CASE 3	4	17	20	3	9
FITNESS CASE 4	6	37	42	5	25
FITNESS CASE 5	8	65	72	7	49
Valor de *fitness*				17	85

Tabela 2: Dois Métodos de Cálculo de Fitness

As duas formas de avaliação de *fitness* apresentadas são adequadas quando o comportamento do programa pode ser descrito através do conjunto de *fitness cases*, isto é, uma tabela de valores de entrada e saída. Uma categoria típica destes problemas é a Regressão Simbólica.

4.6 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 preferenciar, de alguma forma, os programas que apresentem os melhores valores de *fitness*.

Os métodos atualmente usados são (Blickle 1995): Seleção Proporcional, Seleção por Torneio, Seleção por Truncamento, Seleção por Nivelamento Linear e Seleção por Nivelamento Exponencial.

1) **Seleção Proporcional** (*fitness-proportionate selection*): Apresentada por John Holland (Holland 1975) para Algoritmos Genéticos, foi o método escolhido por John Koza no seu primeiro livro (Koza 1992). Usa a aptidão normalizada disposta em uma "roleta", sendo que cada indivíduo da população ocupa uma "fatia" proporcional a sua aptidão normalizada. Em seguida é produzido um número aleatório entre zero (0) e um (1). Este número representará a posição ocupada pela "agulha" da roleta. Apesar de seu grande sucesso devido a sua simplicidade, este método é muito afetado pela escalabilidade da aptidão normalizada (Blickle 1995).

2) **Seleção por Torneio** (*tournament selection*): Apresentada por David Goldberg (Goldberg 1991) para Algoritmos Genéticos, foi utilizada em vários problemas por John Koza no seu segundo livro (Koza 1994). A seleção por torneio é feita da seguinte forma: t indivíduos são escolhidos aleatoriamente da 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 (*threshold*) T entre zero (0) e um (1), a seleção é feita aleatoriamente entre os T melhores indivíduos (Muhlenbein 1993). Por exemplo, se $T = 0.4$, então a seleção é feita entre os 40 % melhores indivíduos e os outros 60 % são descartados.

4) **Seleção por Nivelamento Linear** (*linear ranking selection*): Sugerido por Baker (Baker 1989) para eliminar as sérias desvantagens do uso de seleção proporcional. Para tal, os indivíduos são ordenados de acordo com os valores de *fitness* e o nível N é

associado ao melhor indivíduo e o nível 1, ao pior. Em seguida, a cada indivíduo *i* é associada uma probabilidade p_i de ser selecionado.

$$p_i = \frac{1}{N}\left(n^- + \left(n^+ - n^-\right)\frac{i-1}{N-1} \right) \quad \text{onde } i \in \{1,2,...N\}, \quad n^- \geq 0 \text{ e } n^+ + n^- = 2 \qquad (6)$$

O valor de $\frac{n^+}{N}$ representa a probabilidade do melhor indivíduo ser escolhido e $\frac{n^-}{N}$, a do pior ser escolhido. É interessante perceber que cada indivíduo pertence a um único nível, isto é, mesmo que dois indivíduos tenham o mesmo *fitness*, eles apresentam probabilidades diferentes de serem escolhidos.

5) **Seleção por Nivelamento Exponencial (***exponential ranking selection***):** A seleção por nivelamento exponencial se diferencia do Nivelamento Linear apenas no fato das probabilidades p_i serem exponencialmente ponderadas (Baker 1989). Um parâmetro *c* entre zero (0) e um (1) é usado como base. Quanto mais próximo de um, menor é a "exponencialidade" da seleção. Tal como no Nivelamento Linear, os indivíduos são ordenados de acordo com os valores de *fitness* e o nível *N* é associado ao melhor indivíduo e o nível 1, ao pior. Em seguida, a cada indivíduo *i* é associada uma probabilidade p_i de ser selecionado.

$$p_i = \frac{c-1}{c^{N-1}} c^{N-i} \quad \text{onde } i \in \{1,2,...N\} \qquad (7)$$

4.7 Operadores Genéticos

Uma vez que os indivíduos tenham sido selecionados, deve-se aplicar um dos operadores genéticos. Os três operadores principais são (Koza 1992):

1) **Reprodução:** um programa é selecionado e copiado para a próxima geração sem sofrer nenhuma mudança em sua estrutura.

2) **Cruzamento** *(crossover*): 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. Um exemplo de cruzamento pode ser visto na Figura 22. Neste exemplo, foram escolhidos os programas: `((2*(x+x))+1)` e `(((x+1)*x)-2)`. Foram escolhidos aleatoriamente um nó em cada árvore, identificado com um traçado mais denso na figura. As árvores são então trocadas, gerando os novos programas: `((x+1)+1)` e `(2*((x+x)*x)-2`

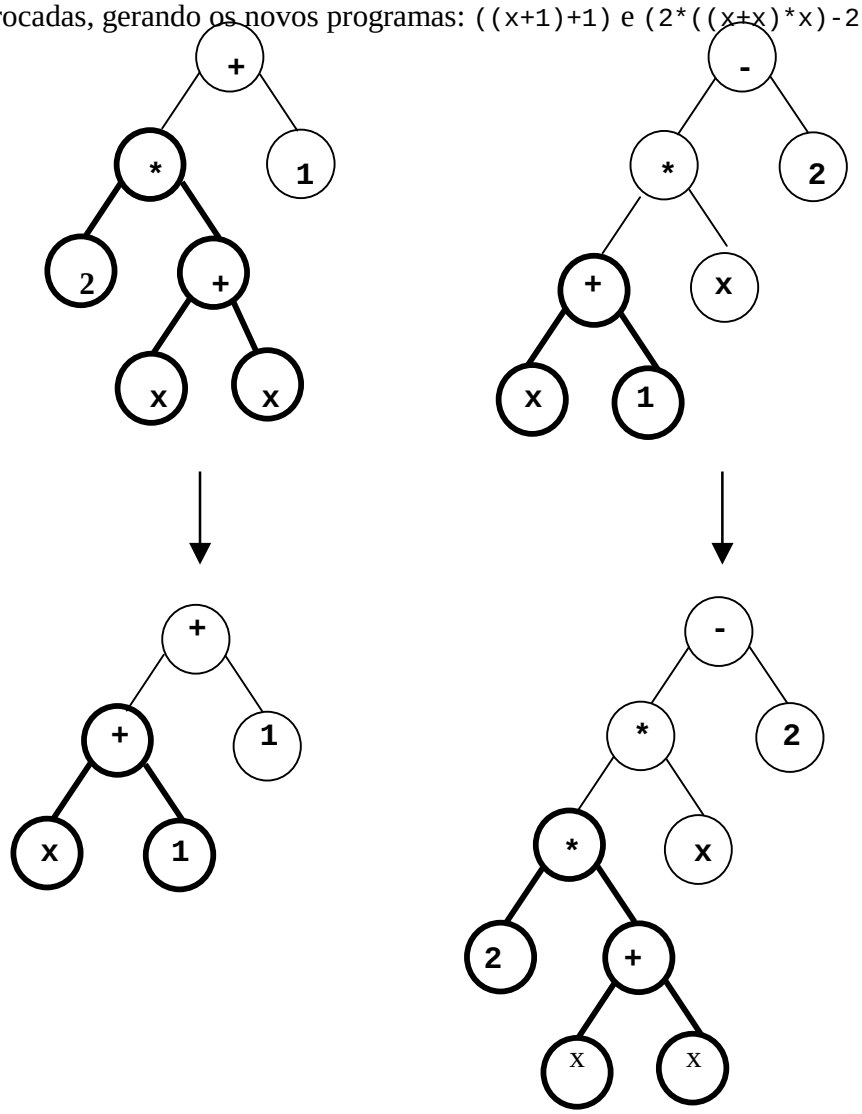

Figura 22: Exemplo de Cruzamento Entre Dois Programas

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, devem-se estabelecer critérios de restrição na escolha dos pontos de cruzamento.

3) **Mutação** (*mutation*): um programa é selecionado e um de seus nós é escolhido aleatoriamente. A árvore cuja raiz é o nó selecionado é então eliminada e substituída por uma nova árvore gerada aleatoriamente.

4.8 Critério de Término

É responsável por interromper o laço de repetição do processo evolutivo que, idealmente, não teria fim. O critério mais comum é limitar o número máximo de gerações ou até que uma solução satisfatória seja encontrada (Koza 1992), porém existem critérios baseados no próprio acompanhamento do processo evolutivo, isto é, enquanto houver melhoria na média da população, o processo evolutivo prossegue (Kramer 2000)

4.9 Limitações

A obrigatoriedade da propriedade de fechamento (*closure*) limita os domínios a serem usados, não possibilitando a aplicação ampla da Programação Genética. A necessidade do fechamento é devida ao uso irrestrito dos operadores genéticos nos programas. Para contornar estes problemas, John Koza[4] (Koza 1992) propôs alterações que devem ser feitas no algoritmo para adequá-lo a domínios que apresentem restrições sintáticas. Porém, estas alterações direcionam o algoritmo para solucionar problemas de determinado tipo, restringindo a sua aplicabilidade.

Por exemplo, para encontrar uma solução em forma de Série de Fourier, John Koza propôs a adoção de três restrições (Koza 1992):

- O nó-raiz da árvore deve ser obrigatoriamente a função especial &.

[4] Capítulo 19 – *Evolution of Constrained Syntatic Structures*, pág. 479 a 526.

- As únicas funções possíveis abaixo de uma & são as funções trigonométricas `xsin`, `xcos` e &.

- As únicas funções permitidas abaixo das trigonométricas são as funções aritméticas (`+`, `-`, `*`, `%`[5]) ou uma constante.

A função especial & faz o mesmo que a função soma (+). A função `xsin` é definida como tendo dois argumentos (`arg0` e `arg1`) e seu cálculo é `arg1*sin(arg2*x)`, sendo que o valor de `arg2` é arredondado para o inteiro mais próximo. De forma semelhante, define-se a função `xcos`.

Estas restrições tornam-se necessárias pois uma Série de Fourier tem a forma:

$$a_0 + \sum_{i=1}^{\infty} \left(a_i \cos\theta + b_i sin\theta \right)$$

Estas restrições são mantidas através da identificação do tipo de cada nó, evitando que os operadores genéticos violem o formato pré-determinado. Apesar deste método funcionar adequadamente para as Séries de Fourier, ele representa uma abordagem que exige a adequação do algoritmo para cada problema que apresente restrições em termos de formato de solução.

Para permitir a aplicabilidade da Programação Genética a qualquer problema que imponha restrições sintáticas, Frederic Gruau (Gruau 1996) propôs o uso de gramáticas. Ao invés de simplesmente informar os conjuntos *F* e *T*, as regras de formação dos programas também são fornecidas. Desta forma é possível guiar genericamente a aplicação dos operadores genéticos a fim de produzir programas sintaticamente corretos frente ao domínio. No capítulo a seguir, as alterações necessárias para permitir o uso de gramáticas em Programação Genética são apresentadas.

[5] Referindo-se a função de divisão protegida, isto é, % é idêntica a / com exceção de que uma divisão por zero resulta em um (Koza 1992)

5. PROGRAMAÇÃO GENÉTICA ORIENTADA A GRAMÁTICAS

Neste capítulo apresenta-se as alterações necessárias para adequar o algoritmo da Programação Genética a problemas descritos através de uma gramática[6]. Inicialmente, são apresentadas as vantagens do uso de gramáticas em Programação Genética. Em seguida, as mudanças na representação da população são descritas. As modificações necessárias no processo de criação da população e na atuação dos operadores genéticos são discutidas no decorrer do texto.

5.1 Motivação

Tradicionalmente, a linguagem alvo usada para Programação Genética é o LISP (Banzhaf 1998). Graças a sua sintaxe simples e ao fato de tanto dados como programas terem o mesmo formato (*S-expressions*), tornou-se a linguagem "ideal" para evoluir programas (Angeline 1994). As *S-expressions* podem ser:

- Um átomo, isto é, um símbolo (variável, função etc.) ou um não-símbolo (número, cadeia de caracteres etc.);

- Uma lista, composta por símbolos ou não-símbolos, no formato `( S-expr S-expr ...)`, por exemplo, `(+ 1 2)`, `(A (B C ))`;

A avaliação de uma *S-expression* é simples:

- Para avaliar um átomo, se ele é um não-símbolo, a sua avaliação é o próprio não-símbolo. Caso seja uma variável, a sua avaliação é o valor atual da variável;

- Para avaliar uma lista, assume-se que o primeiro elemento é uma função, sendo os elementos seguintes, seus argumentos.

[6] No Anexo A apresentamos um resumo sobre gramáticas para facilitar o entendimento deste capítulo.

Portanto é possível tratar os programas diretamente como se fossem dados, facilitando a geração e a aplicação dos operadores genéticos. Desta forma, a especificação de funções e terminais é suficiente para evoluir programas.

Porém, as diversas linguagens de programação, tais como C, Pascal ou Prolog, impõem restrições de formato e tipo, tornando muito difícil a aplicação da Programação Genética diretamente nestas linguagens. Além disto, há domínios em que restrições, quanto ao formato ou tipo, devem ser satisfeitas, tais como as Séries de Fourier apresentadas anteriormente.

Desta forma, a necessidade de se impor ou restringir o formato dos programas, seja por causa de sua linguagem ou por restrições de formato, exige mudanças no algoritmo de Programação Genética. Para cada tipo de linguagem ou restrição, há a necessidade de se adequar o algoritmo.

De forma a tornar a Programação Genética realmente genérica, isto é, aplicável a qualquer problema cuja solução possa ser em forma de um programa, Frederic Gruau (Gruau 1996) propôs o uso de gramáticas. A generalização se torna possível pelo fato de atualmente as linguagens de programação serem facilmente descritas por uma gramática.

Para o uso de gramáticas em Programação Genética, tornam-se necessárias modificações na forma de criação da população inicial e na atuação dos operadores genéticos. Estas alterações visam a preservação da consistência sintática dos programas durante o processo evolutivo e representam extensões dos originais com *S-expressions* (Whigham 1996).

As principais modificações com relação a Programação Genética tradicional são explicadas nas seções que seguem. Inicialmente, mostra-se a representação dos programas através de árvore de derivação. Posteriormente, o algoritmo de criação da população inicial é detalhado. As alterações necessárias ao comportamento dos operadores genéticos são apresentadas no final do capítulo.

5.2 Representação dos Programas

A representação dos programas criados a partir de uma gramática pode ser feita através de sua árvore de sintaxe concreta ou árvore de derivação (Whigham 1995; Ratle 2000). Por exemplo, considere a gramática $G = \{N, \Sigma, S, P\}$ apresentada a seguir:

Σ = { X, +, -, *, / }
N = { <exp>, <op>, <var> }
S = <exp>
P = { <exp> → <var> |
(<exp> <op> <exp>)
<op> → + | - | * | /
<var> → *x* }

A árvore de derivação para uma das possíveis expressões deriváveis de *G*, tal como (x+(x*x)), está na Figura 23.

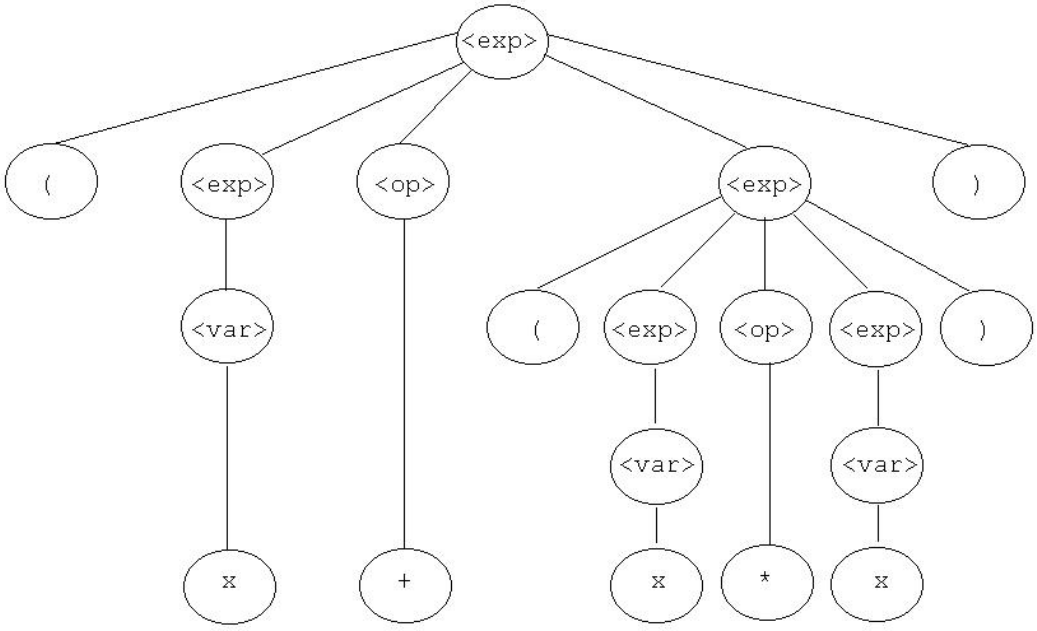

Figura 23: Árvore de Derivação para (x + (x *x))

5.3 População Inicial

O processo de criação de um programa baseado em uma gramática é simples. Partindo-se do símbolo inicial *S*, a árvore é construída através da aplicação sucessiva de produções a cada não-terminal, até que não seja mais possível aplicar nenhuma produção (Sethi 1996; Terry 1997). Se os nós-folha neste instante forem compostos somente por terminais, diz-se que a árvore está *completa*. Uma profundidade máxima é geralmente especificada para evitar árvores muito grandes. Apesar de semelhante ao método *grow* (Koza 1992), não é possível, durante a geração da árvore, prever sua profundidade máxima. Somente após a árvore completa é possível avaliar sua profundidade. Isto leva a criação de um número expressivo de árvores que serão simplesmente rejeitadas.

Para evitar a rejeição de árvores recém criadas, Peter Whigham (Whigham 1996) propôs um método que se baseia em duas fases[7]. A primeira se preocupa em calcular o número mínimo de derivações de cada produção e a segunda, gera as árvores com base na profundidade desejada.

O algoritmo da primeira fase está na Figura 24.

Inicialmente, todas as produções tem MIN_DERIV, igual a -1.
Para cada produção na forma $A \rightarrow x_1\ x_2 \ldots x_n$ onde $x_1\ x_2 \ldots x_n \in \Sigma^*$
    Adota-se MIN_DERIV($A \rightarrow x_1\ x_2 \ldots x_n$) igual a zero(0).
    Enquanto houver uma produção $A \rightarrow \alpha$ com MIN_DERIV($A \rightarrow \alpha$) = -1
        Se todos os não-terminais $t_i \in \alpha$, apresentarem MIN_DERIV(t_i) ≠ -1
            MIN_DERIV($A \rightarrow \alpha$) = MAX(MIN_DERIV(t_i)) + 1.

Figura 24: Algoritmo para o Cálculo do Número Mínimo de Derivações

[7] Este método também foi usado por Alain Ratle e Michele Sebag em (Ratle 2000)

Na segunda fase são criadas as árvores propriamente ditas através de um algoritmo semelhante ao *grow* (Koza 1992). O algoritmo está na Figura 25.

Dados:
  Profundidade máxima D
  Conjunto de produções P
  Símbolo inicial S
Faça:
  Árvore = GROW(S, D)

GROW(símbolo inicial S, profundidade máxima d)
  Retorna: uma árvore de derivação com profundidade máxima $\leq d$

  Se $d = 1$
    Retorne aleatoriamente uma produção $S \rightarrow x$ onde $x \in \Sigma^*$
  Senão
    Selecione aleatoriamente uma produção $S \rightarrow \alpha$ onde $\alpha \in \{N \cup \Sigma\}^*$ onde MIN_DERIV(S) $\leq d$
    Para cada não-terminal $A \in \alpha$
      Anexe a árvore retornada por GROW(A, d-1)

    Retorne a árvore de derivação cuja raiz é o não-terminal S

Figura 25: Algoritmo para Criação da Árvore de Derivação

5.4 Operadores Genéticos

A atuação dos operadores de cruzamento e mutação devem respeitar a gramática usada, a fim de produzir programas válidos. Para o cruzamento, é necessário que os pontos de cruzamento sejam do mesmo tipo, isto é, pertencer ao mesmo não-terminal. Um exemplo de um cruzamento entre duas árvores de derivação está na Figura 26.

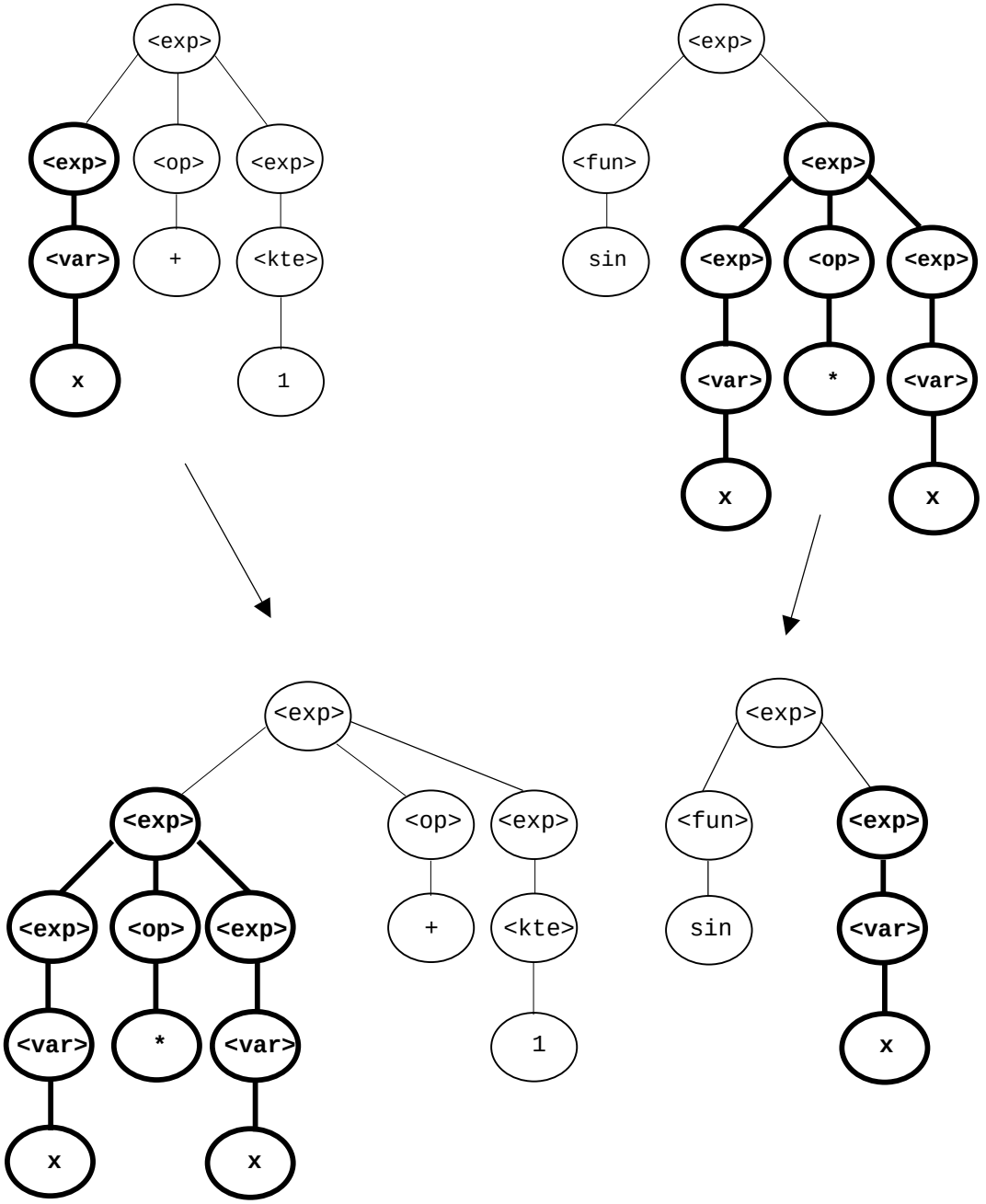

Figura 26: Exemplo de Cruzamento em Árvores de Derivação[8]

[8] Para fins de legibilidade, os parêntesis foram propositavelmente omitidos.

O algoritmo para o operador de cruzamento está na Figura 27.

```
Dados:
        Duas árvores de derivação P1 e P2
        Profundidade máxima D
Faça:
        CROSSOVER(P1, P2, D)

CROSSOVER(árvore de derivação P1, árvore de derivação P2,
                profundidade máxima d)
        Retorna: duas árvores derivação com profundidade máxima ≤ d

        Selecione aleatoriamente um nó não-terminal A ∈ P1
        Construa uma lista de nós pertencentes a P2 cuja não-terminal seja A
        Se a lista resultar vazia
                Retorne com P1 e P2
        Senão
                Selecione aleatoriamente um nó da lista
                Permute as árvores de derivação abaixo de ambos os nós,
                   gerando dois novos programas P1' e P2'
                Se a profundidade de P1' é maior que d
                        Copie P1 em P1'
                Se a profundidade de P2' é maior que d
                        Copie P2 em P2'

        Retorne com P1' e P2'
```

Figura 27: Algoritmo para Cruzamento entre Árvores de Derivação

Para o operador de mutação, o processo é semelhante. Após a escolha aleatória do ponto de mutação, substitui-se a subárvore abaixo dele por outra completamente nova, usando o mesmo algoritmo da população inicial.

REFERÊNCIAS BIBLIOGRÁFICAS

(Aho 1995) AHO, A. V.; ULLMAN, J. D. & SETHI, R.. Compiladores: princípios, técnicas e ferramentas . ISBN 8521610572. LTC, 1995.

(Andre 1994) ANDRE, D. Evolution of map making: learning, planning, and memory using genetic programming. Proceedings of the First IEEE Conference on Evolutionary Computation. Vol I. pp. 250-255. IEEE Press, 1994.

(Angeline 1993) ANGELINE, P. J. & POLLACK, J. Evolutionary module acquisition. Proceedings of the 2nd Annual Conference on Evolutionary Programming, pp. 154-163, 1993.

(Angeline 1994) ANGELINE, P. J. Genetic programming and emergent intelligence. Advances in Genetic Programming, ISBN 0262111888. pp 75-98. MIT Press, 1994.

(Baker 1989) BAKER, J. E. An analysis of the the effects of selection in genetic algorithms PhD thesis, Graduate Schol of Vanderbilt University. Nashiville, Tennessee, 1989.

(Banzhaf 1998) BANZHAF, W; NORDIN, P.; KELLER, R. E. & FRANCONE, F. D. Genetic Programming: an introduction. ISBN 155860510X. Morgan Kaufmann, 1998.

(Barreto 1997) BARRETO, J. Inteligência Artificial. No Limiar do Século XXI. ISBN 859003822X. ρρρ Edições, 1997.

(Blickle 1995) BLICKLE, T. & THIELE, L. A comparision of selection schemes used in genetic algorithms. TIK-Report nr 11 version 2, Computer Engineering and Communication Networks Lab. Swiss Federal Institute of Technology (ETH), Zurich, Switzerland, December, 1995.

(Bohm 1996) BOHM, W. & GEYER-SCHULZ, A. Exact unifrom initialization for genetic programming. Foundations of Genetic Algorithms IV (FOGA 4). ISBN 155860460X. pp 379-407. Morgan Kaufmann, 1996.

(Brameier 1998) BRAMEIER, M.; KANTSCHIK, W.; DITTRICH, P. & BANZHAF, W. SYSGP: A C++ library of different GP variants. Series Computational Intelligence, Internal Report of SFB 531, University of Dortmund, ISSN 1433-3325, Dortmund, 1998.

(Bruce 1995) BRUCE, W. S. The application of genetic programming to the automatic generation of object-oriented programs. PhD thesis. School of Computer and Information Sciences, New Southeastern University, 1995

(Cantu-Paz 1999) CANTU-PAZ, E. Topologies, migration rates, and multi-population parallel genetic algorithms, In: Proceedings GENETIC AND EVOLUTIONARY COMPUTATION CONFERENCE, 1, 1999, p.91-98.

(Caruana 1988) CARUANA, R. A.; SCHAFFER, J. D. Representation and hidden bias: Gry vs. binary coding for genetic algorithms. In: Proceedings INTERNATIONAL CONFERENCE ON MACHINE LEARNING, 5, 1988. Morgan Kaufmann, 1988.

(Cavicchio 1970) CAVICCHIO, JR., D. J. Adaptive search using simulated evolution. Doctoral dissertation, University of Michigan, Ann Arbor, MI. (University Microlms No. 25-199), 1970.

(Chellapilla 1997) CHELLAPILLA, K. Evolving computer programs without sub-tree crossover IEEE Transactions on Evolutionary Computation. Vol. 1 Issue 3. pp 209-216. IEEE Press, September, 1997.

(Coelho 1999) COELHO, A. An updated survey of evolutionary multiobjective optimization techniques: State of the art and future trends. In: Proceedings CONGRESS ON EVOLUTIONARY COMPUTATION 1999, p.1-11.

(Cohen 1995) COHEN, P. R. Empirical methods for artificial intelligence. ISBN 0262032252. MIT Press, 1995.

(Costa 1995) COSTA, D. An evolutionary tabu search algorithm and the NHL scheduling problem. Information Systems and Operational Research, 33 (3), p.161-178, 1995.

(Daida 1999) DAIDA, J. M. Challenges with verification, repeatability and meaningful comparisions in genetic programming. Proceedings of the 4th Annual Conference in Genetic Programming (GECCO'99). ISBN 1558606114. pp. 1069-1076. Morgan Kaufmann, 1999.

(De Jong 1975) DE JONG, K. A. An Analysis of the Behavior of a Class of Genetic Adaptative Systems. PhD thesis, University of Michigan, 1975.

(De Jong 1989) DE JONG, K.; SPEARS, W. Using genetic algorithms to solve NP-complete problems. In: Proceedings INTERNATIONAL CONFERENCE ON GENETIC ALGORITHMS, 3, 1989.

(Dessi 1999) DESSI, A.; GIANI, A. & STARITA, A. An analysis of automatic subroutine discovery in genetic programming. Proceedings of the 4th Annual Conference in Genetic Programming (GECCO'99). ISBN 1558606114. pp. 996-1001. Morgan Kaufmann, 1999.

(Eshelman 1991) ESHELMAN, L.; SHAFFER, J. Preventing premature convergence in genetic algorithm by preventing incest. In: Proceedings INTERNATIONAL CONFERENCE ON GENETIC ALGORITHMS, 4, p.115-122, 1991.

(Finn 1996) FINN, J. D. & ANDERSON, T. W. The new statistical analysis of data. ISBN 0387946195. Springer-Verlag, 1996.

(Fonseca 1993) FONSECA, C.; FLEMING, P. Genetic algorithms for multiobjective optimization: formulation, discussion and generalization. In: Proceedings INTERNATIONAL CONFERENCE ON GENETIC ALGORITHMS, 5, p.416-423, 1993.

(Franco 2001) FRANCO, L. & CANNAS, S. A. Generalization properties of modular networks: implementing the parity function. IEEE Transactions on Neural Networks, Vol 12, Issue 6, pp. 1306-1313. IEEE Press. November, 2001.

(Fraser 1994) FRASER, A. P. A genetic programming in C++. Technical Report 040. Cybernetics Research Intitute, University of Salford.

(Gathercole 1997) GATHERCOLE, C. & ROSS, P. Tackling the boolean even-*n*-parity problem with genetic programming and limited-error fitness. Proceedings of the Second Annual Conference in Genetic Programming (GECCO'97) ISBN 1558604838. pp. 119-127. Morgan Kaufmann, 1997.

(Gathercole 1998) GATHERCOLE, C. An investigation of supervised learning in genetic programming. PhD thesis. University of Edinburgh, 1998.

(Glover 1986) GLOVER, F. Future paths for integer programming and links to artificial intelligence. Computers and Operations Research, v. 13, p.533-549, 1986.

(Glover 1994) GLOVER, F. Tabu search for nonlinear and parametric optimization (with links to genetic algorithms). Discrete Applied Mathematics, v. 49, p.231-255, 1994.

(Glover 1995) GLOVER, F.; KELLY, J.; LAGUNA, M. Genetic algorithms and tabu search: hybrid for optimization. Computers and Operations Research, v. 22(1), p.111-134, 1995.

(Goldberg 1987) GOLDBERG, D. E.; RICHARDSON, J. Genetic algorithms with sharing for multimodal function optimization. In: Proceedings INTERNATIONAL CONFERENCE ON GENETIC ALGORITHMS, 2. 1987.

(Goldberg 1989) GOLDBERG, D. E. Genetic algorithms in search, optimization and machine learning. Alabama: Addison Wesley, 1989. 413p.

(Goldberg 1990) GOLDBERG, D. E. A note on Boltzman tournament selection for genetic algorithms and population oriented simulated annealing. Complex Systems, v. 4, p.445-460, 1990.

(Goldberg 1991) GOLDBERG, D. E. & DEB, K. A comparative analysis of selection schemes used in genetic algorithms. Foundations of Genetic Algorithms (FOGA). ISBN 1558601708. pp 69-93. Morgan Kaufmann, 1991.

(Goldberg 1991) GOLDBERG, D. E.; DEB, K. A comparative analysis of selection schemes used in genetic algorithms. In: RAWLINS, G. (Ed.), Foundations of Genetic Algorithms. San Francisco, CA: Morgan Kaufmann. 1991.

(Gritz 1993) GRITZ, L. A C++ implementation of genetic programming. Department of Electrical Engeneering and Computer Science. The George Washington University, Washington, DC.

(Gruau 1995) GRUAU, F. & WHITLEY, D. A programming language for artificial development. Proceedings of the Fourth Annual Conference on Evolutionary Programming, San Diego, CA. pp 415-434. MIT Press, 1995.

(Gruau 1996) GRUAU, F. On syntactic constraints with genetic programming. Advances in Genetic Programming II. ISBN 0262011581. pp. 377-394. MIT Press, 1996.

(Hiroyasu 1999) HIROYASU, T.; MIKI, M.; WATANABE, S. Divided range genetic algorithms in multiobjective optimization problems. In:Proceedings INTERNATIONAL WORKSHOP ON EMERGENT SYNTHESIS, p.57-66, 1999.

(Holland 1975) HOLLAND, J. H. Adaptation in natural and artificial systems. The University of Michigan Press, Ann Arbor, MI, 1975.

(Horn 1993) HORN, J.; NAFPLIOTIS, N. Multiobjective optimization using the niched pareto genetic algorithm. Technical Report IlliGAl Report 93005, University of Illinois at Urabama-Champaign, 1993

(Horner 1996) HORNER, H. A C++ class library for genetic programming. Technical Report. The Vienna University of Economics, Vienna, Austria, 1996.

(Kinnear Jr 1994) KINNEAR, K. E. Alternatives in automatic function definition: a comparision of performance. Advances in Genetic Programming, ISBN 0262111888. pp 119-141. MIT Press, 1994.

(Kitano 1990) KITANO, H. Empirical studies on the speed of convergence of neural network training using genetic algorithms, In: Proceedings NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE, 8, 1990.

(Kitano 1994) KITANO, H. Neurogenetic learning: an integrated method of designing and training neural networks using genetic algorithms. Physica D, Amsterdam, v. 75, p. 225-238, 1994.

(Koza 1989) KOZA, J. R. Hierarquical genetic algorithms operating on populations of computer programs. Proceedings of the 11th International Joint Conference on Artificial Intelligent (IJCAI-89). Detroit, MI. pp 768-774. Morgan Kaufmann, 1989.

(Koza 1992) KOZA, J. R. Genetic Programming: On the Programming of Computers by Means of Natural Selection. ISBN 0262111705. MIT Press, 1992.

(Koza 1994) KOZA, J. R. Genetic Programming II: Automatic Discovery of Reusable Programs. ISBN 0262111896. MIT Press, 1994

(Kramer 2000) KRAMER, M. D. & ZHANG, D. GAPS: a genetic programming system. The 24th Annual International Computer Software and Applications Conference 2000 (COMPSAC2000) ISBN-0769507921 pp614–619. IEEE Press, October, 2000.

(Krishnamachari 1999) KRISHNAMACHARI, B. Global optimization in the design of mobile communication systems. Ithaca, 1999. 175p. Master of Science in Electrical Engineering, Cornell University.

(Kurahashi 2000) KURAHASHI, S.; TERANO, T. A genetic algorithm with Tabu search for multimodal and multiobjective function optimization. In: Proceedings INTERNATIONAL CONFERENCE ON GENETIC ALGORITHMS, 2. 2000, p.291-298.

(Langdon 1996) LANGDON, W. B. Genetic Programming and data structures PhD thesis. Department of Computer Science. University of London, 1996.

(Lee 1997) LEE, R. C. & TEPFENHART, M. UML and C++ - a practical guide to object-oriented development. ISBN 0130290408. 2nd Edition. Prentice Hall, 1997

(Luke 2000) LUKE, S. Two fast tree-creation algorithms for genetic programming. IEEE Transactions in Evolutionary Computation, Vol 4 Issue 3 pp. 274-283 IEEE Press. September, 2000.

(Luke 2001) LUKE, S. & PAINAIT, L. A survey and comparison of tree generation algorithms. Proceedings of the 6th Annual Conference in Genetic Programming (GECCO 2001). ISBN 1558607749. Springer-Verlag, 2001.

(Malek 1989) MALEK, M.; GURUSWAMY M., et al. Serial and parallel simulated annealing and tabu search algorithms for the traveling salesman problem. Annals of Operations Research, v. 21, p.59-84, 1989.

(Mantawy 1999) MANTAWY, A.; ABDEL-MAGID, Y.; SELIM, S. A new genetic based tabu search algorithm for unit commitment problem. Eletric Power Systems Research, v. 49, p.71-78, 1999.

(Meyer 1992) MEYER, T. P.; PACKARD, N. H. Local forecasting of high-dimensional chaotic dynamics. In: CASDAGLI, N.; EUBANK, S. (Eds.), Nonlinear Modeling and Forecasting. Addison-Wesley. 1992.

(Mitchell 1997) MITCHELL, M. An introduction to genetic algorithms. Cambridge: Mit Press. 1997. 207 p.

(Montana 1994) MONTANA, D. J. Strongly typed genetic programming. BBN Technical Report #7866, Bolt Beranek and Newman, Cambridge, MA, 1994.

(Montana 1995) MONTANA, D. J. Strongly typed genetic programming. Evolutionary Computation. Vol 3 Issue 2. ISBN 10636560. pp 199-230. MIT Press, 1995.

(Muhlenbein 1992) MUHLENBEIN, H. Parallel genetic algorithm in Combinatorial Optimization. Computer Science and Operation Research, Pergamon Press.: p.441-456, 1992.

(Muhlenbein 1993) MUHLENBEIN, H. & SCHIERKAMP-VOOSEN, D. Predictive models for the breeder genetic algorithms. Evolutionary Computation Vol 1 Issue 1, ISBN 10636560. pp 25-49. MIT Press, 1993.

(Muhlenbein 1998) MUHLENBEIN, H.; GORGES-SCHELEUTER, M.; KRAMER, O. Evolution algorithms in combinatorial optimization, Parallel Computing, v. 7, p.65-85, 1998.

(Musser 2001) MUSSER, D. R.; DERGE, G. J. & SAINI, A. STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library. ISBN 0201379236. Addison-Wesley, 2001.

(O'Neil 2000) O'NEIL, M. & RYAN, C. Grammar based function definition in Grammatical Evolution. Proceedings of the 5th Annual Conference in Genetic Programming. (GECCO 2000) ISBN 1558607080. pp. 485-490. MIT Press, 2000.

(O'Reilly 1995) O'REILLY, U. An analysis of Genetic Programming. PhD thesis. Ottawa-Carleton Institute for Computer Science, Carleton University, 1995.

(O'Reilly 1996) O'REILLY, U. Investigating the generality of automatically defined functions. Proceedings of the 1st. Annual Conference in Genetic Programming. ISBN 0262611279. MIT Press, 1996

(Poli 1999) POLI, R.; PAGE, J. & LANGDON, W. B. Smooth uniform crossover, sub-machine code GP and demes: a recipe for solving high-order boolean parity problem. Proceedings of the Genetic and Evolutionary Computation Conference, Vol 2, pp. 1162-1169. Morgan Kaufmann, 1999

(Potter 1997) POTTER, M. A.; DE JONG, K. Cooperative coevolution: an architecture for evolving coadapted subcomponents. In: Proceedings EVOLUTIONARY COMPUTATION, 8(1),2000, pp.1-29.

(Potter 1997) POTTER, M.A.; DE JONG, K.- The Design and Analysis of a Computational Model of Cooperative Coevolution. PhD thesis, George Mason University, 1997.

(Powell 1989) POWELL, D.; TONG, S.; SKOLNICK, M. EnGENEous: Domain Independent machine learning for design optimization. In: Proceedings. INTERNATIONAL CONFERENCE ON GENETIC ALGORITHMS, 3, p.151-159 1989.

(Ratle 2000) RATLE, A. & SEBAG, M. Genetic programming and domain knowledge: beyond the limitations of grammar-guided machine discovery. In Proceedings of the Sixth Conference on Parallel Problems Solving from Nature, LNCS, pp. 211-220. Springer-Verlag, 2000

(Rodrigues 2001) RODRIGUES, E.; POZO, A.; VERGILIO, S. & MUSICANTE, M. Chameleon: uma ferramenta de indução de programas. SCCC 2001 – II Workshop en Inteligencia Artificial. ISBN 0769513964. Punta Arenas, Chile, November, 2001. IEEE Computer Society, 2001.

(Rosca 1996) ROSCA, J. P. & BALLARD, D. H. Discovery of subroutines in genetic programming. Advances in Genetic Programming II. ISBN 0262011581. pp 177-201. MIT Press, 1996.

(Rosca 1997) ROSCA, J. P. Hierarquical learning with procedural abstraction mechanisms. PhD thesis. University of Rochester, Rochester, NY, 1997.

(Rumelhart 1986) RUMELHART, D.; HILTON, G. & WILLIAMS, R. Learning internal representations by error propagation. Parallel Distributed Processing: Explorations in the Micro-structures of Cognition. MIT Press, 1986.

(Ryan 1998) RYAN, C.; COLLINS, J. J. & O'NEIL, M. Grammatical evolution: evolving programs for an arbitrary language. Proceedings of the First Workshop on Genetic Programming, 1998 Vol. LNCS 1391, pp. 83-96. Springer-Verlag, 1998.

(Schaffer 1985) SCHAFFER, D. Multiple objective optimization with vector evaluated genetic algorithms. In: Proceedings INTERNATIONAL CONFERENCE ON GENETIC ALGORITHMS, 1, p.93-100, 1985.

(Sethi 1996) SETHI, R. Programming languages concepts and constructs. ISBN 0201590654. Addison-Wesley, 1996.

(Spector 1995) SPECTOR, L. Evolving control structures with automatically defined macros. Working Notes of the AAAI Fall Symposium on Genetic Programming 1995. The American Association for Artificial Intelligence. pp. 99-105, 1995.

(Spector 1996) SPECTOR, L. Simultaneous evolution of programs and their control structures. Advances in Genetic Programming II.. ISBN 0262011581. pp 137-154. MIT Press, 1996.

(Srinivas 1993) SRINIVAS, N.; DEB, K. Multiobjective optimization using nondominated sorting in genetic algorithms. Technical reports, Department of Mechanical Engineering, Indian Institute of Technology, Kanput, India, 1993.

(Tamaki 1996) TAMAKI, H.; KITA, H.; KOBAYASHI, S. Multiobjective optimization by genetic algorithms. In: Proceedings INTERNATIONAL CONFERENCE ON EVOLUTIONARY COMPUTATION , p. 517-522, 1996.

(Terada 1991) TERADA, R. Desenvolvimento de algoritmos e estruturas de dados. ISBN 0074609602. McGraw-Hill, Makron, 1991.

(Terry 1997) TERRY, P. D. Compilers and Compiler Generators, an introduction with C++. ISBN 1850322988. International Thomson Computer Press, 1997.

(Tsutsui 1993) TSUTSUI, S.; FUJIMOTO, Y. Extended forking genetic algorithms for order representation. In: Proceedings CONFERENCE ON EVOLUTIONARY COMPUTATION,1994, p.170-175.

(Tsutsui 1993) TSUTSUI, S.; FUJIMOTO, Y. Forking genetic algorithms with blocking and shrinking modes. In: Proceedings INTERNATIONAL CONFERENCE ON GENETIC ALGORITHMS, 5, p.206-213, 1993.

(Ulder 1991) ULDER, N.; AARTS, E.; BANDELT, H.; LAARHOVEN, P.; PASCH, E. Genetic local search algorithms for the traveling salesman problem. Parallel Problem-solving from Nature Lecture Notes, Computer Science 496, p.109-116, 1991.

(Whigham 1995) WHIGHAM, P. A. Grammatically based genetic programming. Proceedings of ML'95 Workshop on Genetic Programming. From Theory to Real-World Applications. pp 33-41. Morgan Kauffmann, 1995.

(Whigham 1996) WHIGHAM, P. A. Grammatical Bias for Evolutionary Learning. Ph.D. thesis. School of Computer Science. University of New South Wales, Australian Defense Force Academy, 1996.

(Willis 1997) WILLIS, M. J.; HIDEN, H. G.; MARENBACH , P.; MCKAY, B. & MONTAGE, G. A. Genetic programming: an introduction and survey of applications. The Second International Conference on Genetic Algorithms in Engineering Systems:

Innovations and Applications 1997 (GALESIA 97). pp 314-319. IEEE Press, 1997

(Wong 1996) WONG, M. L. & LEUNG, K. S. Evolving recursive functions for the even-parity problem using genetic programming Advances in Genetic Programming II. ISBN 0262011581. pp. 222-240. MIT Press, 1996.

(Yu 1999) YU, T. Structure Abstraction and Genetic Programming. Proceedings of the 1999 Congress on Evolutionary Computation (CEC99). ISBN 0780355369 pp. 652-659. IEEE Press, 1999.

(Zongker 1996) ZONKGER, D.; PUNCH, B. & RAND, B. Lil-gp 1.01 User's manual Genetic Algorithms Research and Applications Group (GARAGe), Department of Computer Science, Michigan State University, 1996.