Algoritmos Genéticos

Teoria e Implementação

Ricardo Linden

Prefácio do autor

A natureza é sábia e com ela podemos aprender muito. Isto pode parecer frase de algum pajé indígena ou de algum membro do Greenpeace, mas dificilmente lembra alguém envolvido com programação.

Entretanto, a computação, e especialmente o campo da inteligência computacional, tem se beneficiado muito da observação da natureza. Através desta observação surgiram, entre outros, as redes neurais (simulação do comportamento do cérebro humano), o anelamento simulado (simulação do processo de resfriamento de metais) e, finalmente, os algoritmos genéticos (simulação da evolução natural).

Só para deixar bem claro para todos que se aventurarem a ler este texto: apesar do que o termo "algoritmos genéticos" pode sugerir, não há nenhum tipo de experiência genética descrita neste livro, nem nenhum tipo de apoio sugerido a qualquer idéia de eugenia ou superioridade genética.

No decorrer deste livro procurei usar uma mesma nomenclatura e termos técnicos consistentes, preferencialmente em português. A exceção mais gritante foi o uso da sigla GA (de *genetic algorithms*) para me referir aos algoritmos genéticos. Eu a escolhi não por submissão a um suposto colonialismo cultural, mas sim por ser a mais usada entre todos os pesquisadores, mesmo aqueles que costumam usar termos em português.

Todos os códigos descritos neste livro foram compilados na versão 1.4.2 do J2SDK da Sun. Apesar de não haver um CD ou disquete associado a este livro (decisão que tomamos para diminuir os custos de produção), todos podem ser baixados diretamente do site `http://www.pobox.com/~rlinden`. O dowload é gratuito para todos os interessados, inclusive para uso didático.

Eu espero que o estilo do livro agrade a todos, apesar de eu saber de antemão que isto é virtualmente impossível. Procurei fazer um livro que seja fácil e agradável de ler e que ainda assim seja completo e preciso o suficiente para servir como material introdutório à área a apoio a classes de graduação.

Várias pessoas foram gentis a ponto de ler e revisar o meu texto, ou ceder fragmentos de código/texto para melhorar o livro. Parte da qualidade do livro é mérito delas, mas quaisquer erros que porventura tenham permanecido são culpa única e exclusiva minha.

Se você encontrar um erro, quiser fazer um comentário sobre o livro ou apenas bater um papo, ficarei feliz de receber seus e-mails pelo endereço *rlinden@pobox.com*.

Ricardo Linden

1) Introdução

1.1) Inteligência Computacional

Muitos debatem hoje em dia a diferença entre inteligência artificial e inteligência computacional e eu não pretendo entrar nesta discussão neste livro[1]. Vários livros não diferenciam entre os termos, enquanto que outros dizem que a inteligência artificial é a ciência que tenta compreender e emular a inteligência humana como um todo (tanto no comportamento quanto no processo cognitivo), enquanto que a inteligência computacional é a ciência que procura desenvolver sistemas que tenham comportamento similares a certos aspectos do comportamento inteligente. Qualquer uma das duas definições está boa para nós neste momento.

Neste livro nós vamos nos concentrar em uma técnica da inteligência computacional que procura usar alguns aspectos do mundo natural para resolver uma classe de problemas interessantes.

Pensando nisto, as melhores definições de inteligência computacional que podemos utilizar neste livro são seguintes:

- Boose disse que "inteligência artificial é um campo de estudo multidisciplinar e interdisciplinar, que se apóia no conhecimento e evolução de outras áreas de conhecimento".
- Winston disse que inteligência artificial é o estudo das idéias que permitem aos computadores sem inteligências

Apesar de ambas as definições terem sido cunhadas para a inteligência artificial, ela vale perfeitamente para o que vamos descrever neste livro. Nós vamos nos basear fortemente na biologia e na genética para criar um algoritmo[2]. Esperamos que o resultado de nossos esforços sejam programas de computadores mais "espertos" e capazes de encontrar soluções do que os programas tradicionais.

---

[1] Isto quer dizer que eu eventualmente posso escrever outro só para discutir isto. Cartas de apoio serão bem vindas! ;-)

[2] Isto não quer dizer que sem conhecer biologia você não poderá ou terá alguma dificuldade para entender o que será explicado aqui. O grau de dificuldade (ou de facilidade, para os otimistas) é igual para todos.

Afinal, o objetivo final de todas as técnicas da inteligência computacional é a busca. Uma busca de uma solução numérica, do significado de uma expressão lingüística, de uma previsão de carga ou de qualquer outro elemento que tenha significado em uma determinada circunstância.

O algoritmo genético é uma técnica de busca extremamente eficiente no seu objetivo de varrer o espaço de soluções[3] e encontrar soluções próximas da solução ótima, quase sem necessitar de interferência humana, sendo uma das várias técnicas da inteligência computacional dignas de estudo.

1.2) Tempo de execução de algoritmos

Nesta seção nós introduziremos o conceito de avaliação de performance de algoritmos. Nós não tentaremos avaliar o tempo de forma exata, mas sim verificar como o tempo de execução cresce conforme aumentamos o tamanho das entradas. Isto permite-nos fazer uma medida que é válida (mesmo que não seja precisa) não importando a CPU, o sistema operacional, o compilador e outros fatores.

Isto nos ajuda a realizar comparações entre dois algoritmos que porventura resolvam o mesmo problema. Calculando o tempo de cada um, mesmo que aproximadamente, podemos escolher aquele que seja mais eficiente em termos de tempo de execução.

Existem algumas medidas que podem ser feitas sobre um algoritmo:

1. Tempo de execução do pior caso → é o limite superior de tempo que o algoritmo leva para qualquer entrada. Em muitos algoritmos, acontece freqüentemente[4].
2. Tempo médio de execução → costuma ser tão ruim quanto o pior caso, pois as entradas são aleatórias.

---

[3] Mas não tão ótima assim em termos de tempo de processamento. Use e abuse dos algoritmos genéticos somente em problemas NP-Completos. Para maiores detalhes, veja as seções a seguir.

[4] Em outros, é uma medida digna da hiena Hardy Ha-Ha (Oh, céus, oh vida, oh azar!) :-) Entretanto, temos que ter uma idéia do que podemos enfrentar em termos de tempo de execução antes de oferecer os recursos de nossa máquina para um programa.

3. Tempo de melhor caso → só serve para fazermos apresentação em feiras ou vender nosso software para clientes desavisados!

Normalmente, ao avaliarmos a performance de um algoritmo estamos mais interessados no tempo de pior caso. Isto não decorre de uma abordagem negativista em relação à vida, mas que ao fato de que se soubermos o pior tempo possível, sabemos qual é o limite superior de tempo que nosso algoritmo necessitará em todos os casos.

Isto posto, vamos partir para o cálculo do tempo de execução de nossos programas. Primeiro, nós vamos partir de um conceito que obviamente não é preciso: determinamos que todas instruções demoram o mesmo tempo para executar. Chamaremos este tempo de unidade de execução, e para efeito de cálculos, atribuimos a ele valor 1. Veja exemplos de instruções às quais atribuímos tempo de execução igual a 1:

(a) x=2*y+1;
(b) System.out.println(“o valor de x é”+y+1);
(c) z=Math.sin(x*y)+Math.random()*2+y/3;

É óbvio que a instrução (a) demora mais que a instrução (b), que por sua vez demora menos que a instrução (c). Entretanto, nós não estamos calculando o tempo exato de execução de nossos algoritmos, mas sim uma aproximação em termos de ordem de grandeza.

Para termos o valor exato do do tempo de execução, teríamos que pegar o código compilado e checar os manuais de assembler da máquina em questão para determinar o custo computacional de cada instrução. Isto mudaria de máquina para máquina e seria extremamente trabalhoso e aborrecido.

Agora temos que determinar quantas vezes cada uma de nossas instruções de tempo 1 são repetidas, pois os loops são a parte mais importante de um programa (geralmente, 10% do código toma cerca de 90% do tempo de execução). Para tanto, procure os loops que operam sobre as estruturas de dados do programa. Se você sabe o tamanho da estrutura de dados, você sabe quantas vezes o loop executa e conseqüentemente o tempo de execução do loop.

Assim, podemos estabelecer uma receita de como determinar quantas vezes cada uma de nossas instruções de tempo 1 são repetidas. Para tanto, siga os seguintes passos:

1. Divida o algoritmo em pedaços menores e analise a complexidade de cada um deles. Por exemplo, analise o corpo de um loop primeiro e depois veja quantas vezes este loop é executado.
2. Procure os loops que operam sobre toda uma estrutura de dados. Se você sabe o tamanho da estrutura de dados, você sabe quantas vezes o loop executa e conseqüentemente o tempo de execução do loop.

Os loops são a parte mais importante de um programa. Conseqüentemente, quando analisar o seu tempo de execução, preste muita atenção nestas partes, pois elas dominam a execução total.

Para entender melhor, vamos fazer um exemplo. Seja a seguinte função:

```
1 public int arrayMax(Vector V) {
2    int currentMax = (Integer)V.get(0)).intValue;
3    for(int i=1; i<V.size();i++) {
4    if (((Integer)V.get(i)).intValue > currentMax)
5       currentMax=((Integer)V.get(i)).intValue;
6    }
7    return (currentMax);
8 }
```

As linhas 2,4,5 e 7 têm custo unitário, conforme determinamos no texto acima. A linha 5 só ocorre se a condição expressa na linha 4 for verdadeira. Como estamos tentando determinar o tempo de pior caso, vamos considerar que ela sempre ocorre. Assim, o tempo do algoritmo fica um pouco pior.

Fora do loop definido pela linha 3, nós temos a linha 2 e 7, cjuos tempos, conforme determinamos no parágrafo anterior, somam 2. Agora falta determinar o tempo de execução do loop.

O loop começa com a atribução de i=1 da linha 3, atribuição esta que tem custo igual a 1. O resto do loop consiste no teste do for (custo 1), na repetição das linhas 4 e 5 (soma dos custos igual a 2) e no incremento da variável i (custo igual a 1), o que faz com que o custo total de cada repetição seja de 4 unidades de tempo.

Quantas vezes o loop repete? A variável i começa e 1 e é incrementada de 1 em 1 até chegar no tamanho do vetor v (número de

elementos que v contém). Logo, o número de repetições é o tamanho do vetor v menos 1. Se chamarmos o tamanho de v de N, o número de repetições será de N-1 e o custo do loop é de 4*(N-1).

Consequentemente o custo total do algoritmo (T(N)) é dado pelo tempo de execução fora do loop mais o tempo dentro do loop chegando ao total de T(N)=4*(N-1)+3= 4N-1. Veja um resumo disto na figura 1.1 abaixo.

Isto posto, temos que o tempo de execução de nosso algoritmo é de O(N) e Ω(N), o que quer dizer que o tempo de execução de nosso algoritmo varia linearmente com o número de dados contidos no vetor v.

Isto não quer dizer nada sobre o tempo real de execução deste algoritmo, mas sim que ele executa bem mais rápido que um algoritmo que seja $O(N^2)$ ou mesmo $O(N^3)$, e deve ser preferido se o tempo de execução for algo crítico para o programa.

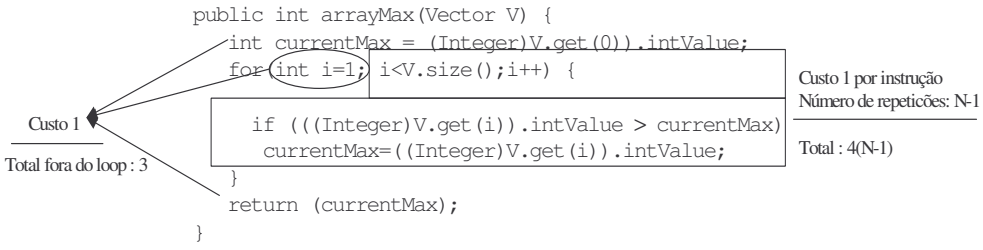


Fig 1.1 → Cálculo do tempo de execução de nosso algoritmo de exemplo.

1.3) Problemas NP-Completos

A sigla NP na denominação NP-Completos significa não-polinomial e faz referência a todos os problemas cujos tempos de execução são associados a funções exponenciais (como 2^n, por exemplo) ou fatoriais (n!).

Para perceber o desastre que é termos um tempo de execução associado a qualquer uma destas funções basta ver os tempos de execução associados a algumas da principais funções de tempos de execução, descritos na tabela a seguir:

n	n^2	n^3	2^n	n!
10	10^2	10^3	$\approx 10^3$	$\approx 10^6$
100	10^4	10^6	$\approx 10^{30}$	$\approx 10^{60}$
1000	10^6	10^9	$\approx 10^{300}$	$>> 10^{300}$
10000	10^8	10^{12}	$> 10^{3000}$	$>> 10^{3000}$

Agora levemos em consideração que uma máquina moderna pode realizar cerca de 10^9 operações em um segundo e veremos que se tivermos que realizar 10^{30} operações (o suficiente para tentarmos resolver um problema de cujo tempo seja proprocional a 2^n com n=100 elementos), levaremos um tempo da ordem de 10^{21} para terminá-las. Pensando que um dia tem pouco menos de 10^5 segundos, isto significa que levaremos um total de 10^{16} dias ou aproximadamente 10^{13} anos – mais do que a idade do universo.

Alguém poderá dizer que os problema NP-Completos são raríssimos e de pouco interesse prático. Seria bom se isto fosse verdade, mas infelizmente, não o é. Os problemas de complexidade não polinomial permeiam nossa vida e precisam ser resolvidos. Vejamos um exemplo.

Um problema NP-completo muito comum é o do caixeiro viajante que tem que visitar n estradas e tem que estabelecer um trajeto que demore o menor tempo possível, para que ele ganhe o máximo de dinheiro no mínimo de tempo. Todas as cidades são ligadas por estradas e pode-se começar por qualquer uma delas (qualquer outra restrição pode ser imposta sem alterar o problema – veremos este problema mais adiante com muito mais detalhe). Como descobrir o caminho mínimo?

A resposta óbvia: calcule todos os caminhos e escolha o de menor custo. Esta resposta usa o famoso método da força bruta – isto é, use muito poder computacional e pronto.

Vamos tentar calcular quantos caminhos temos:

- O caixeiro pode começar em qualquer uma das n cidades.
- Dali ele pode partir para qualquer uma das outras n-1 cidades.
- Da segunda, ele pode partir para qualquer uma das n-2 cidades restantes.
- E assim por diante, até chegar na última cidade.

Isto nos dá um número de opções igual a n(n-1)(n-2)...(2)(1)=n! Neste momento alguém poderá dizer que este problema é totalmente inventado e não tem nenhuma relação com a realidade.

Podemos descartar este argumento substituindo o caixeiro viajante por um caminhão de mercadorias e as cidades a serem visitadas pelos pontos de venda daquela fábrica.

Agora nosso problema consiste em realizar a distribuição de nossa mercadoria da forma mais rápida possível, minimizando o tempo e o custo associados à distribuição. Parece real o suficiente?

Agora que todos estão convencidos, vamos entender o que são os algoritmos genéticos.

1.4) Exercícios

1) Determine o tempo de execução dos seguintes algoritmos

a)
```
Public class Exemplo_Aninhados {
     public static void main (String[] args)
     {
        int n1=0,n2,lim=Integer.parseInt(args[0]);
      while(++n1<10) {
           n2=n1+1;
           while(n2<10) {
                System.out.println(n1+ " "+n2++);
           }
      }
      System.exit(0);
     }
}
```

b)
```
import javax.swing.JOptionPane;
public class Senha1 {
    public static void main (String[] args)
     {
        String senha="Senha1";
```

```
            String suaTentativa;
            int i=1;
            while ((i<=3) &&
                   (!suaTentativa.equals(senha))) {
              suaTentativa=Integer.parseInt(
                JOptionPane.showInputDialog("Senha:"));
              System.out.println("Sua palavra tem " +
                suaTentativa.length()+"letras");
              i++;
            }
            if (i>3) {
              System.out.println( "Você não descobriu a
                                   senha.");
            }
            System.exit(0);
        }
    }
```

2) Determine se as seguintes afirmações são verdadeiras:
   a) 3^{n+1} é $O(2^n)$.
   b) 2^{2n} é $O(4^n)$.
   c) N^N é W (2^N)
   d) NlogN é Q (log N)
   e) n^n é O(n!).

3) Diga qual dos seguintes problemas são NP-Completos:
   a) Ordenação de vetores
   b) Seleção de sub-conjuntos de qualquer tamanho que satisfaçam um determinado critério.
   c) Seleção de melhor caminho para distribuição de produtos para n clientes.

2) Introdução

Neste capítulo procuraremos introduzir os conceitos fundamentais associados aos algoritmos genéticos: em que eles se inspiraram e um esqueleto de como são desenvolvidos.

Não se preocupem se tudo ainda parecer muito abstrato. Teremos todos os capítulos seguintes para desenvolver melhor nossa compreensão e prática.

Entretanto, não pulem esta seção. Vários conceitos interessantes que facilitarão sua compreensão dos capítulos posteriores serão explicados. Um pouco de filosofia e teoria com certeza não lhe farão mal!!!!

2.1) Um pouco de história

Se quisermos falar a verdadeira e completa história dos algoritmos genéticos, temos que começar desde o big-bang. Entretanto, este livro é um pouco curto para descrevermos os último 15 bilhões de anos[5].

Até o século XIX os cientistas mais proeminentes acreditavam em uma dentre as teorias do criacionismo ("Deus criou o universo da forma que ele é hoje") ou da geração espontânea ("a vida surge de essências presentes no ar").

Em torno de 1850 Charles Darwin fez uma longa viagem no navio HMS Beagle. Ele visitou vários lugares e sua grande habilidade para observação permitiu que ele percebesse que animais da uma espécie eram ligeiramente diferentes que seus parentes em outos ecossistemas diferentes, sendo mais adaptados às necessidades e oportunidades oferecidas pelo seu ecossistema específico.

Estas e outras observações culminaram na teoria da evolução das espécies, que foram descritas meticulosamente em seu livro de 1859, "A

[5] Ou os últimos 6 mil se você acreditar no criacionismo. Por mim, tudo bem! ;-)

Origem das espécies". O livro foi amplamente combatido mas hoje é aceito por toda a comunidade acadêmica mundial[6].

A teoria da evolução diz que na natureza todos os indivíduos dentro de um ecosistema competem entre si por recursos limitados, tais como comida e água. Aqueles dentre os indivíduos (animais, vegetais, insetos, etc) de uma mesma espécie que não obtêm êxito tendem a ter uma prole menor e esta descendência reduzida faz com que a probabilidade de ter seus genes propagados ao longo de sucessivas gerações seja menor. A combinação entre os genes dos indivíduos que sobrevivem pode produzir um novo indivíduo muito melhor adaptado às características de seu meio ambiente ao combinar características positivas de cada um dos reprodutores.

Já em termos computacionais, a história dos algorimos genéticos começa na década de 1940 quando os cientistas começam a tentar se inspirar na natureza para criar o ramo da inteligência artificial. A pesquisa se desenvolveu mais nos ramos da pesquisa cognitiva e na compreensão dos processos de raciocínios e aprendizado até o final da década de 50, quando começou-se a buscar modelos de sistemas genéricos que pudessem gerar soluções candidatas para problemas que eram difíceis demais para resolver computacionalmente.

Mas o pai dos algoritmos genéticos mostrou-se finalmente na década de 1960, quando John Holland inventa os Algoritmos Genéticos. Holland estudou formalmente a evolução das espécies e propôs um modelo heurístico computacional que quando implementado poderia oferecer boas soluções para problemas extremamente difícies que eram insolúveis computacionalmente até aquela época. Em 1975 Hooland publica seu livro, "" e a partir daquele momento os algoritmos gnéticos começam a expandir-se por toda a comunidade científica, gerando uma série de aplicações que puderam ajudar a resolver problemas extremamente importantes que talvez não fossem abordados de outra maneira[7].

---

[6] Mas não por toda a comunidade religiosa. Em 2001, um grupo de religiosos nos Estados Unidos tentou proibir o ensino da teoria da evolução em escolas públicas. Sem sucesso, felizmente.

[7] Com certeza alguém inventaria outra maneira de resolvê-los, mas talvez não tão eficiente.

2.2) O que são algoritmos evolucionários?

Algoritmos evolucionários usam modelos computacionais dos processos naturais de evolução como uma ferramenta para resolver problemas. Apesar de haver uma grande variedade de modelos computacionais propostos, todos eles têm em comum o conceito de simulação da evolução das espécies através de seleção, mutação e reprodução, processos estes que dependem da "performance" dos indivíduos desta espécie dentro do "ambiente".

Basicamente os algoritmos evolucionários funcionam mantendo uma população de estruturas que evoluem de forma semelhante à evolução das espécies. A estas estruturas são aplicados os chamados operadores genéticos, como recombinação e mutação, entre outros. Cada indivíduo recebe uma avaliação que é uma quantificação da sua qualidade como solução do problema em questão, e baseado nesta avaliação serão aplicados os operadores genéticos de forma a simular a sobrevivência do mais apto.

Os operadores genéticos consistem em aproximações computacionais de fenômenos vistos na natureza, como a reprodução sexuada, a mutação genética e quaisquer outros que a imaginação dos programadores consiga reproduzir.

O comportamento padrão dos algoritmos evolucionários pode ser resumidos, sem maiores detalhes pelo seguinte pseudo-código.

```
t:=0 // Inicializamos o contador de tempo
Inicializa_População P(0) // Inicializamos a população randômicamente
Enquanto não terminar faça   //condição de término:por tempo, por avaliação, etc.
        Avalie_População P(t) //Avalie a população neste instante
        P':=Selecione_Pais P(t) //Selecionamos sub-população que gerará nova geração
        P'=Recombinação_e_mutação P' //Aplicamos os operadores genéticos
        Avalie_População P'     //Avalie esta nova população
        P(t+1)=Selecione_sobreviventes P(t),P'   //Selecione sobreviventes desta geração
        t:=t+1 //Incrementamos o contador de tempo
Fim enquanto
```

Neste algoritmo podemos perceber o comportamento básico dos algoritmos evolucionários que consiste em buscar dentro da atual

população aquelas soluções que possuem as melhores características e tentar combiná-las de forma a gerar soluções ainda melhores, repetindo este processo até que tenha se passado tempo suficiente ou que tenhamos obtido uma solução satisfatória para nosso problema.

Como se pode perceber claramente dentro do algoritmo, os algoritmos evolucionários são extremamente dependente de fatores estocásticos (probabilísticos), tanto na fase de inicialização da população quanto na fase de evolução (durante a seleção dos pais, principalmente).

Isto faz com que os seus resultados raramente sejam perfeitamente reprodutíveis. Ademais, claramente os algoritmos evolucionários são heurísticas[8] que não garantem a obtenção do melhor resultado possível em todas as suas execuções.

Todas estas condições demonstram um único fato básico: se você tem um algoritmo com tempo de execução razoável para solução de um problema, então não há nenhuma necessidade de se usar um algoritmo evolucionário. Sempre dê prioridade aos algoritmos exatos. Os algoritmos evolucionários entram em cena para ersolver aqueles problemas cujos algoritmos são extraordinariamente lentos (problemas NP-completos) ou incapazes de obter solução (como por exemplo, problemas de maximização de funções multi-modais, como veremos mais à frente).

Agora que usamos todos estes termos complexos, os algoritmos evolucionários parecem ser terrivelmente complicados, mas, como veremos mais adiante, eles são de simples implementação.

2.3) O que são algoritmos genéticos?

Algoritmos genéticos (GA) são um ramo dos algoritmos evolucionários e como tal podem ser definidos como uma técnica de busca baseada numa metáfora do processo biológico de evolução natural.

[8] Heurísticas são algoritmos polinomiais que não têm garantia nenhuma sobre a qualidade da solução encontrada, mas que usualmente tendem a encontrar a solução ótima ou ficar bem próximos dela.

Os algoritmos genéticos são técnicas heurísticas[9] de otimização global. A questão da otimização global opõe os GAs aos métodos como hill climbing, que seguem a derivada de uma função de forma a encontrar o máximo de uma função, ficando facilmente retidos em máximos locais, como vemos na figura 2.1.

Nos algoritmos genéticos populações de indivíduos são criados e submetidos aos operadores genéticos : seleção, crossover e mutação. Estes operadores utilizam uma caracterização da qualidade de cada indivíduo como solução do problema em questão chamada de avaliação deste indivíduo e vão gerar um processo de evolução natural destes indivíduos, que eventualmente gerará um indivíduo que caracterizará

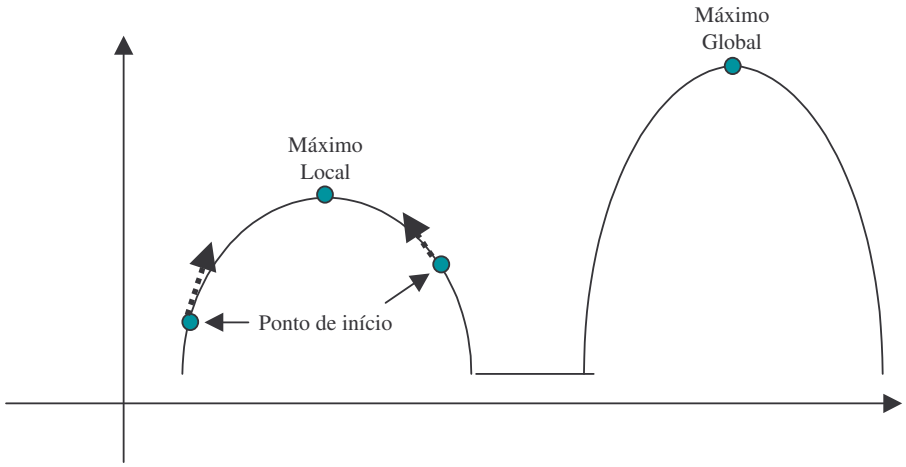


Fig. 2.1 → Função hipotética com um máximo local e outro global. Uma técnica de hill climbing que se inicie em qualquer um dos pontos de início marcados seguirá o gradiente (direção de maior crescimento) e acabará presa no ponto d emáximo local (onde a derivada é zero). Algoritmos genéticos não têm esta dependência tão forte dos valores iniciais.

uma boa solução (talvez até a melhor possível) para o nosso problema.

[9] Muitos de vocês vão se aborrecer com o uso excessivo da palavra "heurística" neste livro. Entretanto, isto não foi feito só para aborrecê-los, mas sim para deixar claro que os algoritmos genéticos, apesar do seu nome sugerir o contrário, não encontram necessariamente a solução ótima pra um problema e, quando o fazem, nem sempre conseguem repetir o feito!

Definindo de outra maneira, podemos dizer que algoritmos genéticos são algoritmos de busca baseados nos mecanismos de seleção natural e genética. Eles combinam a sobrevivência entre os melhores indivíduos com uma forma estruturada de troca de informação genética entre dois indivíduos para formar uma estrutura heurística[10] de busca.

Como dissemos anteriormente, os GAs não são métodos de "hill climbing", logo eles não ficarão estagnados simplesmente pelo fato de terem encontrado um máximo local. Neste ponto, eles se parecem com a evolução natural, que só por que encontrou um indivíduo que é instantaneamente o melhor de um certo grupo não pára de procurar outros indivíduos ainda melhores.

Na evolução natural isto também decorre de circuntâncias que mudam de um momento para outro. Uma bactéria pode ser a melhor em um ambiente livre de antibióticos, mas quando estes são usados outras que antes eram menos fortes tornam-se as únicas sobreviventes por serem as únicas adaptadas.

No caso dos algoritmos genéticos, o ambiente é um só. Entretanto, conforme as gerações vão se passando e os operadores genéticos vão atuando, faz-se uma grande busca pelo espaço de soluções, busca esta que seria realizada pela evolução natural (das bactérias ou de qualquer outro organismo) se elas ficassem permanentemente em um ambiente imutável.

Mas, ao contrário do que as pessoas costumam pensar, é importante ser ressaltado que a evolução natural não é um processo dirigido à obtenção da solução ótima. Na verdade, o processo simplesmente consiste em fazer competir uma série de indivíduos e pelo processo de sobrevivência do mais apto, os melhores indivíduos tendem a sobreviver. Um GA tem o mesmo comportamento que a evolução natural : a competição entre os indivíduos é que determina as soluções obtidas. Eventualmente, devido à sobrevivência do mais apto, os melhores indivíduos prevalecerão. É claro que pode acontecer de uma geração ser muito pior que a geração que a antecedeu, apesar de isto não ser muito comum (nem provável).

[10] Olha a palavra heurística aí de novo! Será que estou fazendo a mensagem ser bem clara?

Sendo assim, devemos salientar que GAs , apesar do seu nome implicar no contrário, não constituem um algoritmo de busca da solução ótima de um problema, mas sim uma heurística que encontra boas soluções a cada execução, mas não necessariamente a mesma todas as vezes[11] (podemos encontrar máximos - ou mínimos - locais, próximos ou não do máximo global).

A codificação da informação em cromossomos é um ponto crucial dentro do GA, e é, junto com a função de avaliação, o que liga o GA ao problema a ser resolvido. Se a codificação for feita de forma inteligente, esta já incluirá as idiossincrasias do problema (como por exemplo restrições sobre quando podemos ligar ou desligar uma máquina, etc) e permitirá que se evitem testes de viabilidade de cada uma das soluções geradas. Obviamente, ao fim da execução do nosso algoritmo a solução deve ser decodificada para ser utilizada na prática. Veremos mais adiante como codificar as informações em cromossomos de acordo com as características de nossos problemas.

Assim como na natureza, a informação deve ser codificada nos cromossomos (ou genomas) e a reprodução (que no caso dos GAs, é equivalente à reprodução sexuada[12]) se encarregará de fazer com que a população evolua. A mutação cria diversidade, mudando aleatoriamente gens dentro de indivíduos,e assim como na natureza, é aplicada de forma menos frequente que a recombinação, que é o fruto da reprodução (e que, dentro do nosso texto, será chamada de crossover). Nas

[11] Esta situação corresponde àquela de duas ilhas pertencentes a um mesmo arquipélago, descrita por Darwin em seu livro "A origem das espécies". As ilhas eram separadas por menos de cem metros de água, tinham o mesmo clima e aproximadamente os mesmos nutrientes, mas algumas espécies de animais, especialmente os terrestres, eram completamente diferentes em cada uma das ilhas. Darwin ficou pasmo com esta descoberta, mas hoje nós sabemos que isto se deve à diversidade das populações iniciais de indivíduos e à imprevisibilidade do processo de evolução natural.

[12] A escolha da reprodução sexuada como modelo para os algoritmos genéticos não é ocasional. A reprodução sexuada é utilizada por todos os animais superiores e garante a diversidade biológica, visto que combinando pedaços de genomas dos dois genitores pode-se gerar filhos mais aptos e consequentemente com o passar das gerações a população tende a evoluir. Já a reprodução assexuada não cria diversidade, visto que cada filho é idêntico a seu genitor e consequentemente tem exatamente as mesmas habilidades e aptidões.

próximas seções veremos exatamente como funciona cada um destes operadores e sua importância para o processo como um todo.

A reprodução e a mutação são aplicadas em indivíduos selecionados dentro da nossa população. A seleção deve ser feita de tal forma que os indivíduos mais aptos sejam selecionados mais freqüentemente do que aqueles menos aptos, de forma que as boas características daqueles passem a predominar dentro da nossa população de soluções. De forma alguma os indivíduos menos aptos têm que ser descartados da população reprodutora. Isto causaria uma rápida convergência genética de todas as soluções para um mesmo conjunto de características e evitaria uma busca mais ampla pelo espaço de soluções[13].

Nos capítulos a seguir veremos em detalhe cada uma das fases do nosso GA, explorando-as e aprendendo a modificá-las de acordo com nossas necessidades.

2.4) Terminologia

Antes de iniciarmos nosso tour pela área dos algoritmos genéticos, é importante que nos familiarizemos com a terminologia adotada. Obviamente, como GAs são altamente inspirados na genética e na teoria da evolução das espécies, há uma analogia muito forte entre os termos da biologia e os termos usados no campo dos GAs.

Nos sistemas naturais um ou mais cromossomos se combinam para formar as características genéticas básicas do indivíduo em questão. Na área dos GAs, os termos cromossomo e indivíduo são intercambiáveis, sendo usado de forma razoavelmente aleatória neste texto. Como a representação binária é dominante em vários dos textos básicos da área, muitas vezes pode-se escrever string (de bits) significando o mesmo que cromossomo.

---

[13] Na natureza os indivíduos menos aptos conseguem reproduzir-se, só que menos freqüentemente do que os mais fortes. Certas espécies possuem mecanismos extremamente violentos e impressionantes para que os machos mais fracos tentem passar seus gens adiante, incluindo estupro, enganação, traição e outras atitudes que seriam crimes em qualquer sociedade humana organizada.

No campo da genética os cromossomos são formados por gens, que podem ter um valor entre vários possíveis, chamados de alelos. A posição do gen é chamada de seu locus (plural : loci). Os termos biológicos são aplicáveis também à área de GA, mas podemos usar os termos características para significar gen, valores significando alelos e posição significando locus. Lembre-se que esta é uma área ligada à informática e muitas vezes os cientistas da computação esquecem o campo onde suas técnicas se inspiraram.

Outros termos importantes são genoma, genótipo e fenótipo. Genótipo é a estrutura do cromossomo, e pode ser identificada na área de GA com o termo estrutra. Fenótipo corresponde à interação do conteúdo genético com o ambiente, interação esta que se dá no nosso campo através do conjunto de parâmetros do algoritmo. Genoma é o significado do pacote genético e não possui análogo na área de GA.

Podemos então resumir tudo que dissemos nesta parte na seguinte tabela, na qual só incluimos a nomenclatura que distingue a área de GA da área da genética. Está implícito que os termos da ciência natural podem ser utilizados também no campo dos GAs, apesar dos termos listados abaixo serem mais comuns na literatura.

Linguagem natural	**GA**
cromossomo	indivíduo,string
gen	característica
alelo	valor
locus	posição
genótipo	estrutura
fenótipo	conjunto de parâmetros

Neste texto serão usados mais os termos descritos na segunda coluna, posto que este é um livro voltado para estudantes de informática. Mas de vez em quando, de forma sorrateira, um ou outro termo biológico pode penetrar em nosso meio. Façamos com que seja bem-vindo!

2.5) Características de GAs

GAs são técnicas probabilísticas, e não técnicas determinísticas. Assim sendo, iniciando um GA com a mesma população inicial e o mesmo conjunto de parâmetros podemos encontrar soluções diferentes a cada vez que rodamos o programa.

GAs são em geral programas extremamente simples que necessitam somente de informações locais ao nosso ponto (relativas à adequabilidade do ponto como solução do problema em questão), não necessitando de derivadas ou qualquer outra informação adicional. Isto faz com que GAs sejam extremamente aplicáveis a problemas do mundo real que em geral incluem descontinuidades duras

Descontinuidades duras são situações onde os dados são discretos ou não possuem derivadas. Isto é muito comum em situações do mundo real em que temos que alocar recursos. Não temos como alocar uma fração de um caminhão ou de uma sala de aula, logo estes problemas não admitem soluções reais, somente inteiras. Logo, não temos como calcular derivadas ou gradientes e não temos como usar técnicas numéricas tradicionais.

GAs trabalham com uma grande população de pontos, sendo uma heurística de busca no espaço de soluções. Um GA diferencia-se dos esquemas enumerativos pelo fato de não procurar em todos os pontos possíveis, mas sim em um (quiçá pequeno) subconjunto destes pontos.

Um exemplo muito claro é no uso de GAs para a solução do problema do caixeiro viajante (ou outro similares) em que o número de soluções possíveis é proporcional à fatorial do número de cidades. Um GA vai procurar o mesmo número de soluções que os seus parâmetros definirem, e nunca uma fração significativa das soluções possíveis (caso contrário, ficaríamos muitas vidas esperando por uma solução).

Além disto, GAs diferenciam-se de esquemas aleatórios por serem uma busca que utiliza informação pertinente ao problema e não trabalham com caminhadas aleatórias (random walks) pelo espaço de soluções.

Isto é garantido pelo fato de usarmos o valor da função de avaliação como guia na escolha dos elementos reprodutores. Apesar de

termos fatores estocásticos associados ao nosso processo, estamos sempre usando-os para tentar caminhar na direção correta[14].

É importante salientar também que GAs trabalham com uma forma codificada dos parâmetros a serem otimizados e não com os parâmetros propriamente ditos. Assim, deve ser definido um esquema de codificação e decodificação destes parâmetros. Isto equivale à representação cromossomial discutida anteriormente e que será aprofundada nos próximos capítulos.

Mas, por outro lado, não importa ao GA como codificam-se ou decodificam-se a informação dos parâmetros. Ele só se importa com a representação em si. Toda a informação relativa ao problema está contida na função de avaliação de problema, que embute os módulos de codificação e decodificação dos parâmetros. Assim, um mesmo GA pode ser utilizado para uma infinidade de problemas, necessitando-se apenas mudar a função de avaliação, o que obviamente gera uma grande economia de tempo e dinheiro para as organizações.

Em nossas implementações usamos a linguagem Java que é uma linguagem orientada a objetos. Isto lhe fornece uma característica de reusabilidade quando bem aplicada. Logo, vamos codificar nossas classes com cuidado de forma que qualquer um possa fazer uma sub-classe de nossas classes básicas e fazer o seu próprio GA rápida e facilmente.

2.6) Exercícios

1) Qual é a diferença entre um máximo local e um máximo global?
2) Qual é a diferença entre uma heurística e um algoritmo?
3) Explique, no contexto de GAs, a diferença entre cromossomo e gen.
4) Qual é a importância da função de avaliação?
5) Qual é a principal vantagem biológica da reprodução sexuada?
6) A evolução natural sempre busca a solução ótima em termos de adaptabilidade do organismo. Verdadeiro ou Falso? Explique.

[14] Como diz o ditado, às vezes é necessário dar um passinho para trás antes de dar dois para frente!

3) O GA mais básico

3.1) Esquema de um GA

Visto que GA são um ramo da computação evolucionária, seu funcionamento é extremamente similar àquele descrito no pseudo-código descrito no capítulo 1, onde cada iteração do loop acima é chamada de uma geração[15]. Podemos também resumir o funcionamento de um GA através da figura 3.1

Basicamente, o esquema acima pode ser descrito algoritmicamente da seguinte maneira :

a) Inicialize a população de cromossomos
b) Avalie cada cromossomo na população
c) Selecione os pais para gerar novos cromossomos. Aplique os operadores de recombinação e mutação a estes pais de forma a gerar os indivíduos da nova geração
d) Apague os velhos membros da população
e) Avalie todos os novos cromossomos e insira-os na população
f) Se o tempo acabou, ou o melhor cromossomo satisfaz os requerimentos e performance, retorne-o, caso contrário volte para o passo c).

Obviamente esta é somente uma visão de alto nível de nosso algoritmo. O que ela esconde é a complexidade do processo de obtenção de uma representação cromossomial que seja adequada ao problema e de uma função de avaliação que penalize soluções

[15] O conceito de geração no campo dos algoritmos genéticos é praticamente idêntico àquele utilizado no discurso coloquial. A única possível diferença é que em alguns tipos de algoritmo genético, como este mais simples que aqui descrevemos, uma geração não convive com a outra, isto é, todos os pais "morrem" imediatamente antes do "nascimento" de todos os filhos. Veremos mais adiante outras versões de GAs que fazem com que os pais vão "morrendo" gradualmente enquanto os filhos vão "nascendo". A outra grande diferença dos GAs para o dia-a-dia é que conflitos de gerações e problemas de diferença de idade e/ou sexo entre dois indivíduos reproduzindo não existem.

implausíveis para nosso problema e que avalie satisfatoriamente o grau de adequabilidade de cada indivíduo como solução do problema em questão. Veremos aos poucos como obter cada uma delas.

Entretanto, como veremos a seguir, o GA é altamente genérico. Vários de seus componentes são invariáveis de um problema para outro. Isto favorece sua implementação em uma linguagem orientada a objeto como Java, permitindo o reaproveitamento do código para solução de vários problemas diferentes.

Traduzindo: todos os techos de código que são mostrados neste livro podem ser utilizados sem grandes adaptações para a solução de

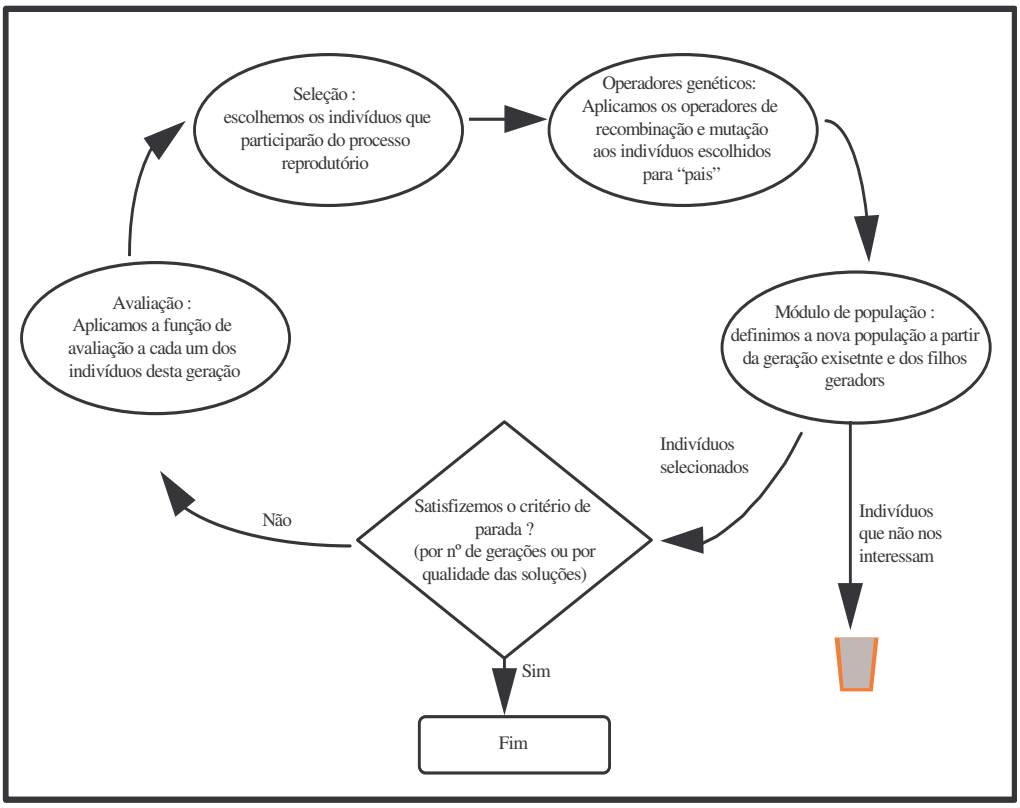


fig 3.1 ⇒ Esquema de um algoritmo genético

seus próprios problemas. Tudo que você precisará é codificar sua própria função de avaliação.

Agora que você está totalmente motivado[16], vamos partir para entender todos estes termos.

3.2) Representação cromossomial

A representação cromossomial é fundamental para o nosso algoritmo genético. Basicamente ela consiste em uma maneira de traduzir a informação do nosso problema em uma maneira viável de ser tratada pelo computador.

Cada pedaço indivisível desta representação é chamado de um gen, por analogia com os pedaços fundamentais que compõem um cromossomo biológico.

É importante notar que a representação cromossomial é completamente arbitrária, ficando sua definição de acordo com o gosto do programador. É interessante apenas que algumas regras gerais sejam seguidas :

a) A representação deve ser a mais simples possível

b) Se houver soluções proibidas ao problema, então elas não devem ter uma representação

c) Se o problema impuser condições de algum tipo, estas devem estar implícitas dentro da nossa representação.

Mais adiante veremos alguns exemplos de como estas regras podem ser seguidas.

Normalmente a representação mais usada é a binária, isto é, um cromossomo nada mais é do que uma sequência de bits e um gen é somente um bit. O que cada bit e/ou conjunto de bits representa é inerente ao problema, como podemos ver nos dois exemplos a seguir.

exemplo 1 : Seja o problema de encontrar o mínimo da seguinte função :

$$f6(x,y) = 0.5 - \frac{\left[\sin\left((x^2 + y^2)^{1/2}\right)\right]^2 - 0.5}{}$$

[16] Ou pelo menos eu assim espero! :-)

$$[\ 1.0 + 0.001(\ x^2 + y^2\ )]^2$$

Esta função é uma função multi-modal contendo vários máximos, como podemos ver na figura abaixo. Logo, uma abordagem do tipo hill climbing não seria adequada, pois a inicialização aleatória poderia nos colocar na encosta de qualquer um dos máximos sub-ótimos e fazer com que nós encontremos algum deles como solução.

Para resolver este problema podemos usar uma representação binária de 44 bits, tamanho este definido pelo grau de precisão que queremos para a solução. Obviamente a solução que procuramos consiste em dois números x e y, que queremos que estejam entre 100 e -100 (por conveniência do espaço de busca). Então convencionamos que os primeiros 22 bits representam x e os 22 bits seguintes representam y. Convertemos as duas sequências de bits em números decimais e encontraremos números entre 0 e 2^{23}-1. Para colocá-los na faixa de -100 a 100, multiplicamo-los por 0.00004768372718899898 (levando este número para a faixa de 0 a 200) e depois diminuímos 100 do resultado obtido.

Teremos dois números que podem então ser aplicados à função de avaliação do nosso problema (que é a função f6(x,y)!) e obteremos a medida de qualidade desta nova solução.

Neste caso vimos que podemos usar uma representação binária para representar dois números (obviamente poderíamos estender esta representação para *n* números, usando mais bits ou atribuindo menos bits para cada número). A representação binária é boa pois para ela os operadores genéticos são extremamente simples e o computador lida com ela de forma extremamente natural. Mas outras representações também são possíveis, como veremos em outros capítulos deste mesmo texto.

Uma função que poderia inicializar estes cromossomos seria a seguinte:

```
1  private void inicializaElemento(int tamanho) {
2        int i;
3        this.valor="";
4        for(i=0;i<tamanho;++i) {
```

```
5            if (java.lang.Math.random()<0.5) {
6               this.valor=this.valor+"0";
7            } else {
8               this.valor=this.valor+"1";
9            }
10        }
11 }
```

Como será padrão neste texto, nós numeramos as linhas de código apenas para facilitar a discussão dos programas no texto. Estes números obviamente não pertencem à listagem e devem ser descartados quando os programas forem digitados. Conforme dissemos no prefácio do autor, todos os códigos fontes de todos os programas colocados aqui podem ser encontrados no site *http://www.pobox.com/~rlinden*.

Na linha 1 do código acima vemos o cabeçalho da função `inicializaElemento`, que recebe uma parâmetro inteiro. Este parâmetro consiste nao tamanho da string de números binários a ser usada em nosso GA.

Note que a implementação discutida neste capítulo é voltada para GAs com cromossomos binários. Como dissemos anteriormente, existem outros tipos de GAs que serão discutidos em outros capítulos.

Nas linhas de 5 a 8 escolhemos aleatoriamente o elemento que ficará em uma posição do cromossomo (posição esta que é determinada pelo loop controlado pelo `for` da linha 4). Repare que usamos um gerador de números aleatórios e dividimos o intervalo de números gerados em dois pedaços exatamente iguais, de forma que seja equiprovável que cada posição tenha um número 0 ou um número 1.

A rotina que listamos acima inicializa um único elemento da população. Precisamos agora de uma rotina que inicialize todos os indivíduos da população usando a rotina que explicamos. Esta rotina tem pode ter o seguinte código fonte:

```
1  public void inicializaPopulacao(int tamanho) {
2     int i;
3     this.populacao=new Vector();
4     for(i=0;i<tamanho;++i) {
5        this.populacao.add(new ElementoGA1());
```

```
6       }
7   }
```

A rotina anterior é definida dentro da classe que define os indivíduos (ElementoGA), enquanto esta é definida dentro da classe que define o comportamento do GA como um todo (GA).

Os fragmentos mostrados neste capítulo são de várias classes distintas e foram selecionados apenas pelo seu efeito didático. As listagens completas de todas as classes criadas para este capítulo estão mostradas na seção intitulada *Listagens* neste mesmo capítulo.

Quanto ao código explicitado acima, alguns comentários:

- A linha 5 consiste em adicionar elementos da classe dos indivíduos definidos para este GA. Caso queiramos implementar outro GA reaproveitando a nossa classe `GA`, basta fazer uma nova sub-classe que herde de `GA` e sobre-escrever a função `inicializaPopulacao` com outra que inicialize os elementos de acordo com a classe de indivíduos usada. Necessariamente a classe usada deve ser uma sub-classe da classe `ElementoGA`.
- temos que saber qual é o tamanho da população. Este deve ser grande o suficiente para gerar diversidade ao mesmo tempo em que não seja grande demais a ponto de tornar o programa demasiadamente lento.

3.3) Função de avaliação

A função de avaliação é a maneira utilizada pelos GAs para determinar a qualidade de um indivíduo como solução do problema em questão. Na verdade, dada a generalidade dos GAs, a função de avaliação é a única ligação verdadeira do programa com o problema real. Isto pode não parecer óbvio à primeira vista, mas basta pensar que exatamente o mesmo programa pode ser usado para descobrir o máximo de toda e qualquer função de *n* variáveis sem nenhuma alteração das estruturas de dados e procedimentos adotados, mas alterando a função de avaliação (que neste caso específico, seria exatamente a função a ser maximizada).

É interessante observar que a função de avaliação não é necessariamente uma função real a coeficientes reais. Ela pode ser uma função discreta ou até mesmo uma função de inteiros. Veremos mais adiante, no capítulo sobre aplicações, um GA aplicado a escalonamento de horários de salas de aula em faculdades, e a função de avaliação deste GA corresponde a uma função de quantos alunos foram deixados sem sala (um número inteiro com certeza, pois 0.2 alunos nada significa, já que não podemos deixar apenas a perna de um aluno de fora de uma sala).

A função de avaliação deve portanto ser escolhida com grande cuidado. Ela deve embutir todo o conhecimento que se possui sobre o problema a ser resolvido, tanto suas restrições quanto seus objetivos de qualidade.

Um ponto muito importante a ser levado em consideração é que nenhum elemento deve ter avaliação negativa ou zero. Isto faria com que a soma das avaliações diminuisse, alterando a roleta e fazendo com que houve mais de um elemento possivelmente selecionado dada a escolha do mesmo valor de intervalo. Ademais, ficaria difícil associar um determinado espaço na roleta para um número negativo.

Não se preocupe se isto não ficou muito claro agora – na seção intitulada *Seleção de pais* nós vamos explicar novamente esta questão. O importante agora é entender que idealmente a função de avaliação deve ter um contradomínio estritamente positivo.

A maneira de fazer isto é extremamente simples. Se a menor das funções f(x) é –c, basta nós tentarmos maximizar a função f'(x)=f(x)+c', onde c'>c. Isto garante que nenhuma avaliação pode ser igual ou menor a zero.

Outro importante ponto a ser levado em consideração é o fato de que os algoritmos genéticos são técnicas de maximização. É difícil alterar o método da roleta para selecionar com mais freqüência aqueles indivíduos que possuam uma avaliação menor. A melhor maneira de fazer isto, caso desejemos encontrar um elemento que minimize uma função é invertendo a função de avaliação de interesse (f(x)) e maximizando a função g(x)=1/f(x).

Neste caso, nós obviamente temos que nos preocupar com o caso em que f(x)=0. Caso este ponto esteja presente no contradomínio

da nossa função, nós usamos a mesma técnica descrita acima: maximizar a função h(x)=1/(f(x)+c), onde c é uma constante real positiva qualquer.

No caso do nosso problema, nós queremos dscobrir o máximo de uma função que, coincidentemente[17], varia de 0 a 1, o que faz com que todos os valores da função sejam positivos. Logo, podemos usar o valor calculado para a função diretamente como sendo o valor da função de avaliação do GA.

O requerimento de que os valores da função de avaliação sejam positivos (idealmente, estritamente positivos) será explicado mais adiante, na seção intitulada *Seleção de pais*. Por enquanto, tenham fé neste quesito!

Consequentemente, uma implementação possível desta função de avaliação seria:

```
1  private float converteBooleano(int inicio,int fim) {
2        int i;
3        float aux=0;
4        String s=this.getValor();
5        for(i=inicio;i<=fim;++i) {
6           aux*=2;
7           if (s.substring(i,i+1).equals("1")) {
8              aux+=1;
9           }
10       }
11       return(aux);
12 }

13 public double calculaAvaliacao() {
14    double x=this.converteBooleano(0,21);
15    double y=this.converteBooleano(22,43);
16    x=x*0.00004768372718899898-100;
17    y=y*0.00004768372718899898-100;
18    double numerador=Math.sin(Math.sqrt(x*x+y*y))-0.5;
19    double denominador=(1.0 + 0.001*(x*x+y*y));
20    denominador*=denominador;
21    this.avaliacao=0.5-numerador/denominador;
```

[17] Amodéstia me impede, neste momento, de dizer que a função de avaliação foi brilhantemente escolhida, de forma a ter excelente valor didático, blá, blá, blá. :-)

```
22    return(this.avaliacao);
23 }
```

Assim como anteriormente, os números nada significam. Ademais, durante este capítulo, nós estamos misturando a listagem de várias classes distintas, pinçando apenas as funções que efetivamente nos interessam neste momento. Ao final do capítulo nós veremos as funções por completo, na seção intitulada *Listagens*.

3.4) Seleção de pais

O método de seleção de pais que utilizaremos deve tentar simular o mecanismo de seleção natural que atua sobre as espécies biológicas, em que os pais mais capazes geram mais filhos, mas mesmo os pais menos aptos também podem gerar descendentes.

Consequentemente, temos que privilegiar os indivíduos com função de avaliação alta, sem desprezar completamente aqueles indivíduos com função de avaliação extremamente baixa.

Isto ocorre pois até indivíduos com péssima avaliação podem ter características genéticas que sejam favoráveis à criação de um "superindivíduo", características estas que podem não estar presentes em nenhum outro cromossomo de nossa população.

Veja-se por exemplo a tabela a seguir. Obviamente a função de avaliação corresponde ao quadrado do valor numérico do indivíduo binário, e consequentemente o cromossomo '0001' tem uma péssima função de avaliação. Mas ele tem uma boa característica, que é o 1 na última posição, não presente nos dois melhores indivíduos. Se utilizássemos somente os dois melhores indivíduos para reproduzir nossa população nunca avançaria para um indivíduo melhor que 0110.

Indivíduo	*Avaliação*	*Pedaço da roleta (%)*	*Pedaço da roleta (º)*
0001	1	1.61	5.8
0011	9	14.51	52.2
0100	16	25.81	92.9

0110	36	58.07	209.1
Total	*62*	*100.00*	*360.0*

Logo, agora que ficou estabelecido que precisamos dar chance a todos, precisamos decidir como fazê-lo. A maneira que a grande maioria dos pesquisadores de GA utiliza é o método da roleta viciada.

Neste método criamos uma roleta (virtual) na qual cada cromossomo recebe um pedaço proporcional à sua avaliação (obviamente a soma dos pedaços na podem superar 100%). Depois rodamos a roleta e o selecinado será o indivíduo sobre o qual ela parar.

A roleta viciada para o grupo de indivíduos da tabela acima é mostrada na seguinte figura.

Repare que nós somamos todas as avaliações e para cada indivíduo alocamos um espaço igual a a avaliação deste indivíduo dividida pela soma das avaliações de todos os indivíduos. O que acon-

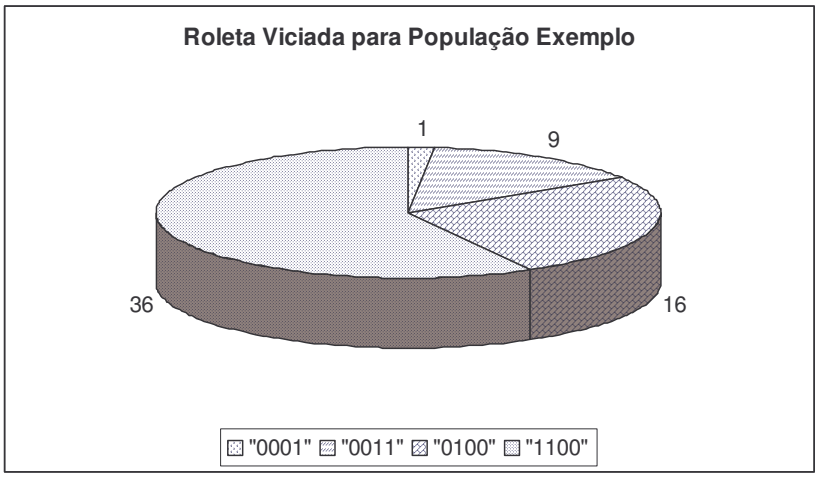


fig 3.2 ⇒ Roleta viciada para indivíduos do exemplo

teceria então se tivéssemos um ou mais indivíduo com avaliação negativa?

A resposta é: a soma dos espaços alocados para os de avaliação positiva excederia 360º (isto é, a soma dos pedaços seria maior que o

círculo) e nós ainda teríamos que lidar com o problma de alocar um espaço negativo para o indivíduo em questão.

Para deixar a questão mais clara, vamos fazer um exemplo hipotético, com uma função de avaliação f(x)=x.

Indivíduo	*Avaliação f(x)=x*	*Pedaço da roleta (%)*	*Pedaço da roleta (º)*
1	1	**6,25**	**22,5**
-5	-5	-31,25	-112,5
20	20	**125**	**450**
Total	*16*	*100.00*	*360.0*

Se olhássemos apenas para os valores totais (última linha da tabela) tudo pareceria estar OK. Afinal, a soma dos elementos em termos percentuais atinge 100% e em termos de graus atinge 360º (circunferência completa). Qual é o problema então?

Repare que a soma dos valores positivos (marcados em negrito) excede 360º (na verdade eles chegam a 472,5º, que consiste na soma de 360º com o módulo do espaço ocupado pelo valor negativo). Como podemos alocar este espaço para eles? Estaríamos dando mais de uma volta pra depois retroceder quando encontrássemos o valor negativo? Isto é impraticável em termos lógicos e de implementação.

Dado que escolhemos sempre um valor entre 0 e 16 para rodar a nossa roleta, o elemento com valores negativos nunca seria escolhido, dado que a soma dos valores de avaliação até ele são sempre menores que o valor da soma até seu antecessor. Logo, se o antecessor não foi escolhido, então não há jeito do indivíduo com avaliação negativa ser escolhido.

Vejamos um exemplo. Rodei a roleta e escolhi um valor igual a 3. A avaliação do primeiro é 1, logo ele não foi escolhido. A soma da avaliação do primeiro com o segundo é –4, que também é menor que 3, logo ele também não pode ser escolhido. Vocês podem testar com todos os números positivos no intervalor [2,16] e verão que não há jeito do indivíduo de avaliação negativa ser selecionado.

Vamos resolver isto então usando a técnica descrita na seção anterior (somar uma constante maior que zero ao valor da função de

avaliação). Logo vamos usar a função de avaliação f'(x)=x+6. O valor 6 foi escolhido porque seu módulo é maior do que o módulo do maior valor negativo da função de avaliação. Agora o quadro para roleta fica assim:

Indivíduo	*Avaliação f'(x)=x+6*	*Pedaço da roleta (%)*	*Pedaço da roleta (º)*
1	7	20,6	74,1
-5	1	2,9	10,6
20	26	76,5	275,3
Total	*34*	*100.00*	*360.0*

Agora nenhum valor é negativo, todos os pedaços são menores que a roleta como um todo e podemos aplicar sem problemas o nosso método.

Obviamente nós não podemos girar uma roleta dentro do computador. Logo, a versão computacional da roleta é dada pelo seguinte algoritmo, pressupondo que nenhum indivíduo tenha uma avaliação nula ou negativa. Se isto ocorrer, para resolver basta deslocar todas as avaliações por uma constante igual ao módulo da menor avaliação negativa, ou então usar alguma das técnicas que veremos no capítulo 6 de nosso texto.

(a) Some todas as avaliações para uma variável soma
(b) Ordene todos os indivíduos em ordem crescente de avaliação (opcional)
(c) Selecione um número s entre 0 e soma (Não incluídos)
(d) i=1
(e) aux=avaliação do indivíduo 1
(f) enquanto aux<s
(g) i = i + 1
(h) aux=aux+avaliação do indivíduo i
(i) fim enquanto

O índice do indivíduo escolhido é dado pela variável i, e o algoritmo tem parada garantida, visto que aux tende ao somatório das avaliações dos indivíduos e s é menor que esta soma.

O código Java que implementa o pseudo-código descrito acima é dado a seguir. Como veremos na seção *Listagens* mais adiante neste texto, as funções descritas a seguir assumem que a classe possui dois atributos: um Vector chamado `populacao`, que armazena todos os indivíduos da geração corrente e um atributo de objeto do tipo double chamada `somaAvaliacoes`[18], que armazena a soma total das avaliacoes da geração corrente.

```
1 Private double calculaSomaAvaliacoes() {
2     Int i;
3     this.somaAvaliacoes=0;
4     for(i=0;i<populacao.size();++i) {
5      this.somaAvaliacoes+=((ElementoGA)
6                             populacao.get(i)).getAvaliacao();
7     }
8     return(this.somaAvaliacoes);
9   }
```

Repare que a função descrita nas linhas de 1 a 9 implementa exatamente a linha (a) do pseudo-código.

```
10 public int roleta() {
11    int i;
12    double aux=0;
13    calculaSomaAvaliacoes();
14    double limite=Math.random()*this.somaAvaliacoes;
15    for(i=0;( (i<this.populacao.size())&&(aux<limite) );++i) {
16       aux+=((ElementoGA) populacao.get(i)).getAvaliacao();
17    }
      /*Como somamos antes de testar, então tiramos 1 de i pois
       o anterior ao valor final consiste no elemento escolhido*/
18    i--;
19    return(i);
20 }
```

[18] É importante ressaltar que todos os atributos foram definidos nas classes como sendo do tipo protected. Isto foi feito por ser boa prática de programação orientada a objetos encapsular todos os atributos diminuindo sua visibilidade. Existem em todas as classes métodos acessores para cada um dos atributos. Veja as listagens completas na seção intitulada *Listagens.*

A linha 14 faz o que a linha (c) do pseudo-código previu: escolher um valor entre 0 é a soma das avaliações. Nós usamos a função `random()` para escolhe raleatóriamente um número entre 0 e 1 que torna-se uma percentagem do valor da `somaAvaliacoes` quando por este multiplicado.

O loop das linhas 15 a 17 vai somando os valores das avaliações de cada um dos indivíduos. Quando ele exceder o valor determinado na linha 14, o loop termina. Note que colocamos um teste para sair se excedermos o tamanho do loop. Isto não é necessário, mas é sempre boa prática de programação fazer o máximo para evitar que erros[19] aconteçam em um programa.

Na linha 18 nós temos uma diminuição do valor de i. Ela é feita pois começamos nosso somatório em zero em vez de começar com a avaliação do elemento na primeira posição como a linha (e) do pseudo-código sugeriu.

Isto foi feito para garantir que nenhum erro ocorreria se alguém chamasse esta função com zero indivíduos definidos nela. Neste caso a rotina retornaria –1 (isto é, nenhum indivíduo válido). Isto é só uma precaução para uma situação que poderia ser considerada até absurda e pode ser eliminado caso desejado[20].

3.5) Operador de crossover e mutação

Agora que estamos iniciando nossa jornada através dos GAs, iremos trabalhar com a versão mais simples dos operadores genéticos, que é a versão em que eles estão operando em conjunto, como se fossem um só operador.

Mais à frente nós veremos outros modos de selecionar qual operador será aplicado em um determinado momento, mas neste

[19] Na verdade, em Java os erros são chamados de exceção e embutem um processamento maior do que simplesmente causar erros. Nós não vamos tratar todos os erros possíveis em nossos programas com blocos try..catch. Caso você queira usar os GAs descritos neste livro em uma implementação real, sugiro fortemente que faça estes testes.

[20] Existe um ditado entre os programadores que diz que nenhum erro é estúpido demais para ser cometido por um usuário. Minhas desculpas aos usuários, mas isto parece ser verdade...

momento o objetivo é fazer com que sejam claros os conceitos dos operadores. Concentrem-se neles agora.

3.5.a) Operador de crossover

Como estamos em um capítulo introdutório, vamos começar com o operador de crossover mais simples, chamado de operador de crossover de um ponto. Outros operadores mais complexos (e, por conseqüência, mais eficientes) serão apresentados em capítulos posteriores.

A operação deste operador é extremamente simples. Depois de selecionados dois pais pelo módulo de seleção de pais, um ponto de corte é selecionado.

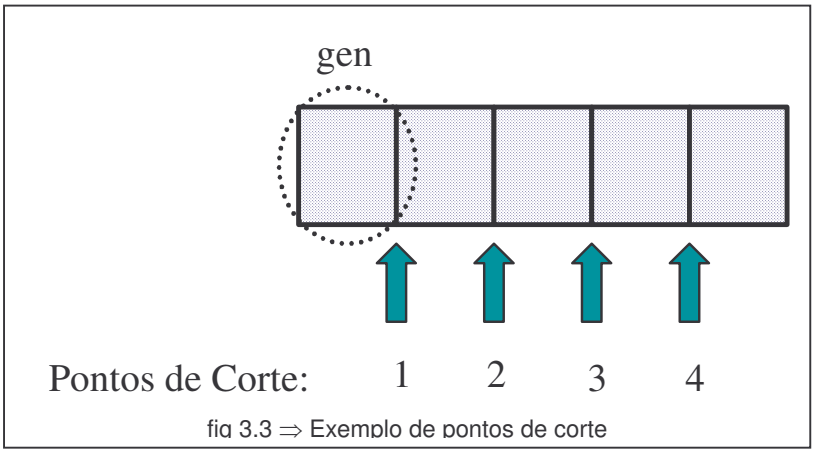


fig 3.3 ⇒ Exemplo de pontos de corte

Um ponto de corte constitui uma posição entre dois gens de um cromossomo. Cada indivíduo de n gens contem n-1 pontos de corte, e este ponto de corte é o ponto de separação entre cada um dos genes que compõem o material genético de cada pai. A metade à esquerda do ponto de corte vai para um filho e a metade à direita vai para outro.

Na figura 3.3 nós podemos ver um exemplo de pontos de corte. No caso, nosso cromossomo é composto de 5 gens e por conseguinte temos 4 pontos de corte possíveis.

Depois de sorteado o ponto de corte, nós separamos os pais em duas partes: uma à esquerda do ponto de corte e outra à direita. É importante notar que não necessariamente estas duas partes têm o mesmo tamanho. Por exemplo, se selecionarmos o ponto de corte número 4 da figura, a parte esquerda de cada pai tem 4 genes enquanto que a parte direita tem 1[21].

O primeiro filho é composto através da concatenação da parte esquerda do primeiro pai e com a parte direita do segundo pai. O segundo filho é composto através da concatenação das metades que sobraram (a metade esquerda do segundo pai com a metade à direita do primeiro pai).

Este processo é parecido com o que acontece na natureza durante a formação cromossomial de um indivíduo pertencente a uma espécie que adota a reprodução sexuada. A diferença é que a natureza não se restringe a apenas um ponto de corte[22].

O código fonte que implementa o crossover de um ponto é dado a seguir:

```
1   public ElementoGA crossoverUmPonto(ElementoGA outroPai)  {
2            String aux1;
3            ElementoGA retorno=null;
4            int pontoCorte=(new Double(java.lang.Math.random()
                        *this.valor.length())).intValue();
5            if (java.lang.Math.random()<0.5) {
6               aux1=this.valor.substring(0,pontoCorte)+
                     outroPai.getValor().substring(pontoCorte,
                     outroPai.getValor().length());
7            } else {
8               aux1=outroPai.getValor().substring(0,pontoCorte)+
                     this.valor.substring(pontoCorte,
                     this.valor.length());
9            }
10           try {
11              retorno=(ElementoGA)
                          outroPai.getClass().newInstance();
12              retorno.setValor(aux1);
```

[21] Se pensarmos um pouco vamos perceber que se o número de genes for ímpar, é imopssível que as duas partes de cada pai tenham exatamente o mesmo tamanho. Afinal, números ímpares não são divisíveis por dois!

[22] Nem os operadores mais avançados que vamos ver mais à frente. Mas não vamos colocar o carro na frente dos bois...

```
13            } catch (Exception e) {
14            }
15            return(retorno);
16  }
```

É fácil entender porque este operador é denominado crossover de um ponto. Afinal, nós sorteamos apenas um ponto de corte, o qu é feito na linha 4.

Nesta função nós estamos só gerando um filho, que pode vir com a parte esquerda do primeiro pai e direita do segundo ou vice-versa, de acordo com o teste feito na linha 5. Isto foi feito apenas para que houvesse um valor de retorno facilmente compreensível e a função fosse mais simples.

Um pouco de raciocínio analógico nos faz concluir que se selecionássemos dois pontos de corte nós teríamos um crossover de dois pontos, não é mesmo?

Isto é verdade, mas a operação do crossover de dois pontos é um pouco mais complexa[23] e nós vamos estudá-lo no próximo capítulo.

Já que existe o crossover de um ponto e o de dois pontos, então existe também o de três, o de quatro e assim por diante, não é mesmo?

Na verdade, não (ou pelo menos, não que eu saiba!). Entretanto existe um outro operador mais avançado denominado crossover uniforme que veremos mais à frente.

3.5.b) Operador de mutação

[23] Mas só um pouquinho! Não desanime, você está indo muito bem!

Depois de compostos o filho, entra em ação o operador de mutação. Este opera da seguinte forma : ele tem associada a ele uma probabilidade extremamente baixa (da ordem de 0,5%) e nós sorteamos um número entre 0 e 1. Se ele for menor que a probabilidade então o operador atua sobre o gen em questão, alterando-lhe o valor aleatoriamente. Repete-se então o processo para todos os gens componentes dos dois filhos.

O valor da probabilidade que decide se o operador de mutação será ou não aplicado é um dos parâmetros do GA que apenas a experiência pode determinar. No capítulo 10 há algumas dicas interessantes (que, obviamente, refletem a minha experiência) sobre a determinação de valores de alguns dos parâmetros mais importantes de

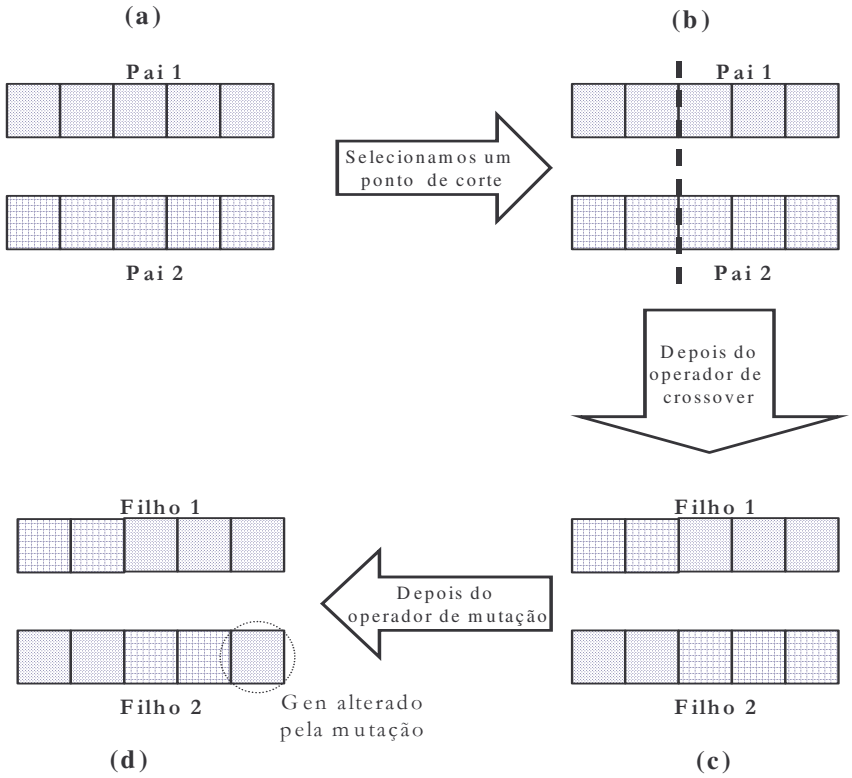


fig 3.4 ⇒ Descrição da operação do operador de crossover de um ponto e mutação

nossos GAs.

Alguns textos preferem que o operador de mutação não aja de forma aleatória, mas sim, quando selecionado, altere o valor do gen para outro valor válido do nosso alfabeto genético. É fácil perceber que este procedimento corresponde em multiplicar a probabilidade do operador de mutação por n/(n-1), onde n é a cardinalidade (número de símbolos distintos) do nosso alfabeto genético. No caso binário, a cardinalidade é dois (existem somente zero e um no nosso alfabeto), logo estamos dobrando a probabilidade do operador de mutação, caso usemos esta segunda técnica.

A operação completa da conjunção do operador de crossover com o de mutação é mostrada na figura 3.4, enquanto que o código que implementa a mutação é dada a seguir:

```
1     public void mutacao(double chance) {
2        int i;
3        int tamanho=this.valor.length();
4        String aux,inicio,fim;
5        for(i=0;i<tamanho;i++) {
6           if (java.lang.Math.random()<chance) {
7              aux=this.valor.substring(i,i+1);
8              if (aux.equals("1")) {aux="0";}
9              else {aux="1";}
10             inicio=this.valor.substring(0,i);
11             fim=this.valor.substring(i+1,tamanho);
12             this.valor=inicio+aux+fim;
13          }
14       }
15    }
```

3.6) Módulo de população

O módulo de população é responsável pelo controle da nossa população, que assumiremos por questão de simplicidade que não pode crescer. Logo, os pais têm que ser substituídos conforme os filhos vão nascendo, pois estamos agindo como se o mundo fosse um lugar pequeno demais para ambos conviverem.

Isto pode parecer estranho, visto que estamos acostumados a ver a população humana sempre crescendo. Afinal, da nossa experiência de vida, quando nasce um bebê, não é obrigatório que alguém de alguma geração anterior caia fulminado!

Entretanto, em ambientes de recursos limitados (água, ar, comida, etc) este crescimento sem controle não é permitido e os próprios organismos tendem a limitar o tamanho da população, seja tendo menos filhos, seja devorando-os ou de qualquer outra maneira que a natureza considerar adequada.

Podemos então considerar que o nosso GA opera em um ambiente de recursos limitados. Diga-se de passagem, isto é verdade, pois nosso computador tem uma quantidade limitada de memória e ciclos de processador. É claro que estamos limitando a população a um tamanho bem inferior ao da memória como um todo, mas poderíamos aumentá-la, caso necessário fosse.

O módulo de população que utilizaremos por enquanto é extremamente simples. Sabemos que a cada atuação do nosso operador genético estamos criando dois filhos. Estes vão sendo armazenados em um espaço auxiliar até que o número de filhos criado seja igual ao tamanho da nossa população.

Neste ponto o módulo de população entra em ação. Todos os pais são então descartados e os filhos copiados para cima de suas posições de memória, indo se tornar os pais da nova geração.

O código fonte que implementa esta módulod e população é dado por:

```
1   public void moduloPopulacao() {
2           populacao.removeAllElements();
3           populacao.addAll(nova_populacao);
4       }
```

Note a simplicidade da implementação. Simplesmente copiamos, na linha 3, uma estrutura auxiliar (nova_população, que no caso é um Vector) para dentro de outra (populacao, outro Vector), que havia sido previamente limpa na linha 2.

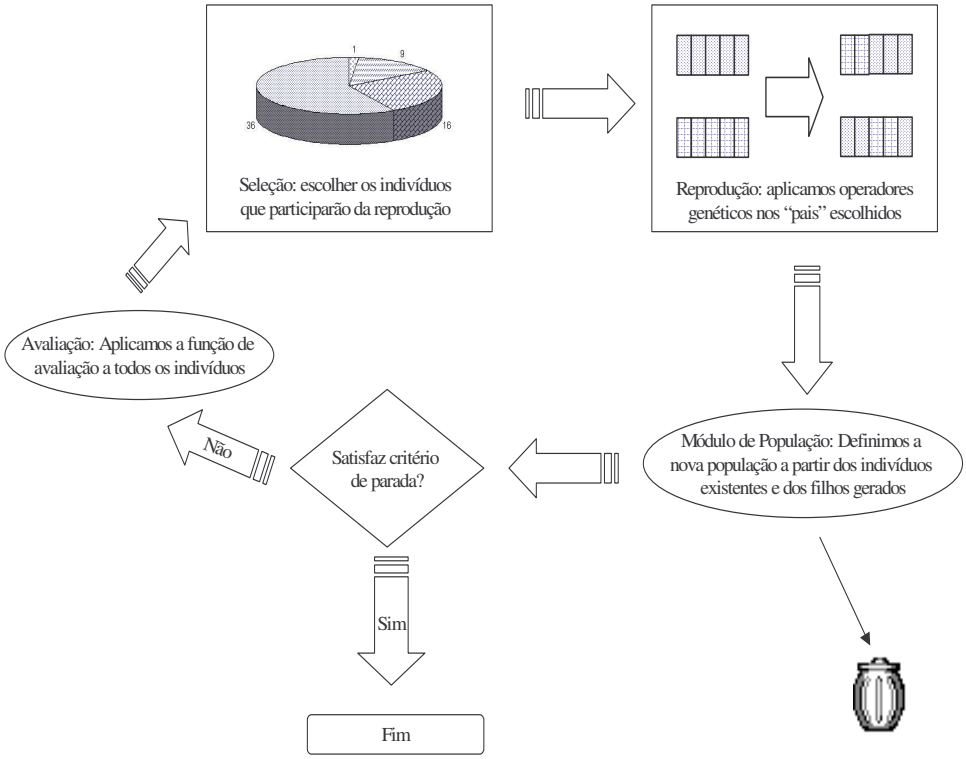


fig 3.5 ⇒ Versão final do nosso GA mais simples

Existem outras maneiras mais inteligentes e mais produtivas de realizar esta mudança de gerações. Como de hábito, nós vamos deferir o tratamento desta questão para capítulos posteriores[24].

3.7) Versão final do GA

Nosso GA mais simples está então concluído. A sua operação total está mostrada então pela figura 3.5.

O código de execução que faz o mostrado na figura é dado pela função executa da nossa classe GA, cujo código é o seguinte:

```
1   public void executa() {
2         int i;
```

[24] Estou tentando manter o suspense com um "a seguir cenas dos próximos capítulos". Se funciona nas novelas, porque não aqui?

```
3       this.inicializaPopulacao();
4       for (i=0;i<this.numero_geracoes;++i) {
5             this.avaliaTodos();
6             this.geracao();
7             this.moduloPopulacao();
8       }
9       i=this.determinaMelhor();
10      System.out.println((ElementoGA) this.populacao.get(i));
11  }
```

Na linha 3 nós inicializamos a população. Esta função é sobreescrita em cada uma das classes filhas da GA para inicializar os indivíduos com a classe apropriada (aquela que possui a função de avaliação ligada ao problema que queremos resolver).

Na linha 4 vemos que o único critério de parada que usamos atualmente é o número de gerações decorridas. Em outras versões mais avançadas, podemos usar outros critérios como qualidade da melhor solução encontrada ou número de gerações em melhora da melhor solução da população corrente.

Nas linhas 5, 6 e 7, realizamos os passos do GA: avaliar a população (função avaliaTodos), aplicar os operadores genéticos naqueles escolhidos pela roleta (função geracao) e substituir a população corrente pela nova população gerada (função moduloPopulacao).

Nas linhas 9 e 10 escolhemos o melhor elemento para mostrá-lo na tela como resultado final do programa.

Existem vários melhoramentos para este GA, que tornam-no mais eficiente em sua busca de soluções melhores para o problema que estamos tratando. Veremos algumas no decorrer deste texto.

O interessante de nosso GA é que ele é uma metáfora extremamente incompleta do processo de evolução natural. Não existe em nosso problema versões para questões como envelhecimento, atração sexual, liderança e outras coisas especificamente humanas como por exemplo o conflito de gerações, stress, etc. Mas ainda assim o nosso GA funciona de forma notável em sua busca (não dirigida, assim como a evolução natural) de boas soluções.

É claro que alguém poderia invertar similares computacionais destas características naturais. Se isto for feito, provavelmente os GAs

se tornarão não só ferramentas de busca mais eficientes como também serão uma notável analogia das sociedades naturais, podendo até ser objeto de estudos sociológicos[25].

O processo também não é de todo empírico, como pode parecer até agora. Existe uma teoria razoavelmente sólida por trás dos GAs, teoria esta que omitimos até agora por questões de conveniência, mas que veremos no capítulo a seguir.

3.8) Listagens

3.8.a) Arquivo ElementoGA.java

Consiste nas operações básicas para um GA que usa uma representação booleana.

```
import java.util.*;

public class GA {
   protected Vector populacao;
   protected double somaAvaliacoes;
   protected double chance_mutacao;
   protected Vector nova_populacao;
   protected int numero_geracoes,tamanho_populacao;

   private double calculaSomaAvaliacoes() {
      int i;
      this.somaAvaliacoes=0;
      for(i=0;i<populacao.size();++i) {
            this.somaAvaliacoes+=((ElementoGA)
populacao.get(i)).getAvaliacao();
      }
      return(this.somaAvaliacoes);
   }

   public int roleta() {
      int i;
      double aux=0;
```

[25] É claro que isto é apenas uma digressão, mas a idéia é bastante interessante para merecer uma menção aqui!

```
      calculaSomaAvaliacoes();
      double limite=Math.random()*this.somaAvaliacoes;
      for(i=0;( (i<this.populacao.size())&&(aux<limite) );++i) {
         aux+=((ElementoGA) populacao.get(i)).getAvaliacao();
      }
      /*Como somamos antes de testar, então tiramos 1 de i pois
        o anterior ao valor final consiste no elemento
escolhido*/
      i--;
      System.out.println("Escolhi o elemento de indice "+i);
      return(i);
   }

   public void inicializaPopulacao() {
   /*Esta funcao tem que ser substituida por uma que inicialize a
populacao
     com a subclasse apropriada de elementoGA*/
      int i;
      this.populacao=new Vector();
      for(i=0;i<this.tamanho_populacao;++i) {
         this.populacao.add(new ElementoGA());
      }
   }
   public void geracao() {
      nova_populacao=new Vector();
        ElementoGA pai1,pai2, filho;
      int i;
      System.out.println("Calculando nova geracao...\n");
      for(i=0;i<this.populacao.size();++i) {
            pai1 = (ElementoGA)populacao.get(this.roleta());
            pai2  =  (ElementoGA)  populacao.get(this.roleta());

              filho= pai1.crossoverUmPonto(pai2);
            filho.mutacao(chance_mutacao);
            System.out.println("Vou adicionar...");
            nova_populacao.add(filho);
      }
   }

   public void moduloPopulacao() {
        populacao.removeAllElements();
        populacao.addAll(nova_populacao);
   }

   private int determinaMelhor() {
      int i,ind_melhor=0;
```

```
      ElementoGA aux;
      this.avaliaTodos();
      double
aval_melhor=((ElementoGA)this.populacao.get(0)).getAvaliacao();
      for(i=1;i<this.populacao.size();++i) {
            aux=(ElementoGA)this.populacao.get(i);
            System.out.println("i="+i+"
aval="+aux.getAvaliacao());
            if (aux.getAvaliacao()>aval_melhor) {
               aval_melhor=aux.getAvaliacao();
               ind_melhor=i;
            }
      }
      return(ind_melhor);
   }

   private void avaliaTodos() {
      int i;
      ElementoGA aux;
      System.out.println("Avaliando todos...\n");
      for(i=0;i<this.populacao.size();++i) {
            aux=(ElementoGA)this.populacao.get(i);
            aux.calculaAvaliacao();
      }
      this.somaAvaliacoes=calculaSomaAvaliacoes();
      System.out.println("A     soma     das     avaliacoes     eh
"+this.somaAvaliacoes);
   }

   public void executa() {
      int i;
      this.inicializaPopulacao();
      for (i=0;i<this.numero_geracoes;++i) {
            System.out.println("Geracao "+i+"\n");
            this.avaliaTodos();
            this.geracao();
            this.moduloPopulacao();
      }
      i=this.determinaMelhor();
      System.out.println((ElementoGA) this.populacao.get(i));
   }

   /****************/
   /* Construtores */
   /****************/
```

```
   public GA(int num_geracoes,int tam_pop, double prob_mut) {
      this.chance_mutacao=prob_mut;
      this.numero_geracoes=num_geracoes;
      this.tamanho_populacao=tam_pop;
   }

   public GA(int tam_populacao, double prob_mut) {
      this(60,tam_populacao,prob_mut);
   }

   public GA(double prob_mut) {
      this(60,100,prob_mut);
   }

   public GA() {
      this(60,100,0.001);
   }
}
```

3.8.b) ElementoGA1

É a classe que usamos para definir os elementos binários que serão usados para solução do nosso problema específico. Note que a maioria das funções é deixada como definida na superclasse. Nós definimos apenas o que é específico para nosso problema específico, que neste caso é a função de avaliação.

```
import java.util.*;

 public class GA1 extends GA {

   public void inicializaPopulacao() {
      int i;
      this.populacao=new Vector();
      for(i=0;i<this.tamanho_populacao;++i) {
         this.populacao.add(new ElementoGA1());
      }
   }

   /****************/
   /* Construtores */
   /****************/
```

```
   public GA1(int num_geracoes,int tam_pop, double prob_mut) {
      super(num_geracoes,tam_pop,prob_mut);
   }

   public GA1(int tam_populacao, double prob_mut) {
      super(60,tam_populacao,prob_mut);
   }

   public GA1(double prob_mut) {
      super(60,100,prob_mut);
   }

   public GA1() {
      super(60,100,0.001);
   }
}
```

Note como é simples impementar sua versão de um indivíduo para nosso algoritmo genético. É só sobreescrever oa função de avaliação e pronto![26]

3.8.c) GA

É a classe que implementa a execução básica de nosso GA.

```
import java.util.*;

public class GA {
   protected Vector populacao;
   protected double somaAvaliacoes;
   private double chance_mutacao;
   private Vector nova_populacao;

   private double calculaSomaAvaliacoes() {
      int i;
      this.somaAvaliacoes=0;
      for(i=0;i<populacao.size();++i) {
            this.somaAvaliacoes+=((ElementoGA)
```

[26] Ah, como é bela a orientação a objetos, não é mesmo?

```
populacao.get(i)).getAvaliacao();
      }
      return(this.somaAvaliacoes);
   }

   public int roleta() {
      int i;
      double aux=0;
      calculaSomaAvaliacoes();
      double limite=Math.random()*this.somaAvaliacoes;
      for(i=0;( (i<this.populacao.size())&&(aux<limite) );++i) {
         aux+=((ElementoGA) populacao.get(i)).getAvaliacao();
      }
      /*Como somamos antes de testar, então tiramos 1 de i pois
        o anterior ao valor final consiste no elemento
escolhido*/
      i--;
      System.out.println("Escolhi o elemento de indice "+i);
      return(i);
   }

   public void inicializaPopulacao(int tamanho) {
   /*Esta funcao tem que ser substituida por uma que inicialize a
populacao
     com a subclasse apropriada de elementoGA*/
      int i;
      this.populacao=new Vector();
      for(i=0;i<tamanho;++i) {
         this.populacao.add(new ElementoGA());
      }
   }

   public void geracao() {
      nova_populacao=new Vector();
        ElementoGA pai1,pai2, filho;
      int i;
      System.out.println("Calculando nova geracao...\n");
      for(i=0;i<this.populacao.size();++i) {
            pai1 = (ElementoGA)populacao.get(this.roleta());
            pai2 = (ElementoGA) populacao.get(this.roleta());

              filho= pai1.crossoverUmPonto(pai2);
            filho.mutacao(chance_mutacao);
            System.out.println("Vou adicionar...");
            nova_populacao.add(filho);
      }
```

```
    }

    public void moduloPopulacao() {
         populacao.removeAllElements();
         populacao.addAll(nova_populacao);
    }

    private int determinaMelhor() {
       int i,ind_melhor=0;
       ElementoGA aux;
       this.avaliaTodos();
       double
aval_melhor=((ElementoGA)this.populacao.get(0)).getAvaliacao();
       for(i=1;i<this.populacao.size();++i) {
             aux=(ElementoGA)this.populacao.get(i);
             System.out.println("i="+i+"
aval="+aux.getAvaliacao());
             if (aux.getAvaliacao()>aval_melhor) {
                aval_melhor=aux.getAvaliacao();
                ind_melhor=i;
             }
       }
       return(ind_melhor);
    }

    private void avaliaTodos() {
       int i;
       ElementoGA aux;
       System.out.println("Avaliando todos...\n");
       for(i=0;i<this.populacao.size();++i) {
             aux=(ElementoGA)this.populacao.get(i);
             aux.calculaAvaliacao();
       }
       this.somaAvaliacoes=calculaSomaAvaliacoes();
       System.out.println("A soma das avaliacoes eh
"+this.somaAvaliacoes);
    }


    /****************/
    /* Construtores */
    /****************/

    public GA(int num_geracoes,int tamanho_populacao, double
prob_mut) {
       int i;
```

```
      this.chance_mutacao=prob_mut;
      this.inicializaPopulacao(tamanho_populacao);
      for (i=0;i<num_geracoes;++i) {
            System.out.println("Geracao "+i+"\n");
            this.avaliaTodos();
            this.geracao();
            this.moduloPopulacao();
      }
      i=this.determinaMelhor();
      System.out.println((ElementoGA) this.populacao.get(i));
   }

}
```

3.8.d) GA1

É a classe que implementa a execução de nosso GA voltado para a solução de um problema específico. Note como o único método que precisamos sobreescrever é o método de inicialização da população, que deve fazê-lo com indivíduos da classe cuja função de avaliação é adequada.

```
import java.util.*;

 public class GA1 extends GA {

   public void inicializaPopulacao(int tamanho) {
      int i;
      this.populacao=new Vector();
      for(i=0;i<tamanho;++i) {
         this.populacao.add(new ElementoGA1());
      }
   }

   public GA1(int num_geracoes,int tamanho_populacao, double
prob_mut) {
      super(num_geracoes, tamanho_populacao,prob_mut);
   }
 }
```

3.9) Exercícios

1) Realize os seguintes crossovers de um ponto
   a) 000111 e 101010 com ponto de corte=4
   b) 11011110 e 00001010 com ponto de corte=1
   c) 1010 e 0101 com ponto de corte=2

2) Simule a execução de uma geração de um GA com população de 6 elementos dados por 001100, 010101, 111000, 000111, 101011, 101000 cuja função sendo maximizada é $f(x)=x^2$.

3) Implemente as sub-classes necessárias (de ElementoGA e GA) para resolver o problema de maximização da função $f(x)=x^2*\sin(x)*e^{-|x|}$, no intervalo de -1000 a 1000. Use 30 bits para representar x.

4) Explique porque todos os organismos superiores utilizam reprodução sexuada.

5) Explique porque o módulo de população que usamos atualmente não reflete o que efetivamente acontece na natureza.

4) Teoria dos GAs

Para algoritmos determinísticos como o método de hill-climbing, o método de Newton-Raphson e outros, é extremamente comum termos algum tipo de prova formal para a convergência até os resultados ótimos após um certo número de iterações.

Entretanto, para algoritmos probabilísticos isto não é possível, pois seu comportamento ao longo das iterações não é previsível. A única coisa que podemos afirmar é seu comportamento médio/esperado ao longo do tempo.

Algoritmos genéticos são um pesadelo em termos de análise, dado que sua própria estrutura (qual operador vai operar e como) é probabilística por natureza. Logo, nós não pretendemos explicar aqui matematicamente[27] todas as suas propriedades mas sim explicar basicamente seus fundamentos e dar uma boa idéia de porque os GAs funcionam.

4.1) Conceitos básicos

Um esquema consiste em um template descrevendo um subconjunto dentre o conjunto de todos os indivíduos possíveis. O esquema descreve similaridades entre os indivíduos que pertencem a este subconjunto, ou seja, descreve quais posições dos seus genomas são idênticas.

O alfabeto de esquemas consiste no alfabeto de símbolos utilizados na nossa representação mais o símbolo *, que significa "não-importa" (don't care ou wildcard), isto é, que os indivíduos que correspondem àquele esquema diferem exatamente nas posições onde encontramos este símbolo.

Formalmente podemos definir um esquema como sendo uma string $s=\{s_1\ s_2\ ...\ s_1\}$, de comprimento n, cujas posições pertence ao conjunto Γ (alfabeto isado) + {*} (símbolo de wildcard). Cada posição da

[27] Até porque a maioria dos leitores pularia o capítulo! :-)

string dada por $s_k \neq$ '*' é chamada de especificação, enquanto que o símbolo * especifica um wildcard.

exemplo 1 : Se considerarmos as populações de strings de bits, temos o nosso alfabeto de esquemas descritos pelos símbolos {0, 1 e *}[28], onde * significa que aquela posição pode ser qualquer coisa, não pertencendo ao esquema. Assim, temos o seguinte:

Esquema	Indivíduos que representa
1*	10 , 11
1*0*1	10001, 10011, 11001, 11011
**0	000, 010, 100, 110

exemplo 2 : Se considerarmos o alfabeto romano Γ={a,b, ..., z} mais o símbolo * (não importa) como nosso alfabeto de esquemas, temos o seguinte :

Esquema	Indivíduos que representa
a*	aa, ab, ..., az
a*b	aab, abb, ..., azb
**xy	aaxy, abxy, ..., azxy, baxy, bbxy, ..., bzxy, ..., zaxy, zbxy, ..., zzxy

Podemos ver claramente nos exemplos acima que se nosso alfabeto de esquemas contém n símbolos, e nosso esquema contém p posições com *, então nosso esquema representa exatamente $(n-1)^p$ indivíduos, onde o menos um provém do fato que o * não entrará na composição dos indivíduos.

Já o número de esquemas presentes em um determinado indivíduo é dependente do comprimento da string e do número de opções presentes no alfabeto de codificação.

Podemos facilmente inferir que se nossa string tem tamanho l e nosso alfabeto de esquemas contém n símbolos, então o número de

[28] Neste caso o alfabeto usado, definido por Γ, é igual a{0,1}

esquemas existente na nossa população é exatamente n^l. Por exemplo, se nosso alfabeto de esquemas é aquele do exemplo 1, {0, 1, *} e nossa string tem tamanho 2, temos exatamente $3^2 = 8$ esquemas possíveis, que são os seguintes : 00, 01, 10, 11, 1*, 0*, *1, *0, **.

Dizemos então que uma string x satisfaz um esquema se para todo símbolo s_k pertence à string s definidora do esquema diferente do símbolo de wildcard, temos que $s_k = x_k$. Isto é, em toda a posição k (k=1,2,...,n) que não é igual ao asterisco, a string x contém o mesmo caracter que s.

A questão agora é : por que os esquemas são importantes? Eles o são pois um GA na verdade é um manipulador de esquemas. Os esquemas contém as características positivas e negativas que podem levar a uma boa ou má avaliação e o GA nada mais faz do que tentar propagar estes bons esquemas por toda a população enquanto roda.

Podemos aqui fazer uma breve analogia biológica: muitos biólogos acreditam na teoria do gen egoísta, no qual toda a natureza, tanto dos humanos quanto dos animais, é apenas uma forma que os genes encontraram de multiplicarem-se com mais eficiência. Da mesma maneira, um GA nada mais seria que uma maneira dos esquemas fazerem o mesmo!

Agora podemos entender a característica do paralelismo implícito dos GAs: na verdade o seu paralelismo está embutido no fato que para cada elemento da população um GA manipula dezenas, quiçá centenas de esquemas simultaneamente (todos aqueles presentes em cada indivíduo) e vai encontrando os melhores dentre todos eles, pois os mecanismos de seleção natural vão fazer com que os melhores esquemas acabem reproduzindo mais e permanecendo mais tempo na população.

Isto quer dizer que o importante não é o indivíduo e sim o esquema. Pode ser que o indivíduo morra, mas o esquema que o torna bom prolifera e continua na população.

Um esquema tem duas características importantes : sua ordem e seu tamanho. Ordem de um esquema, denotado por o(H), corresponde ao número de posições neste esquema diferentes de *, e tamanho do esquema, denotado por δ(H), corresponde ao número de posições entre a primeira e a última posições diferentes de * dentro do esquema.

exemplos :

Esquema	Ordem	Tamanho
*****1***	1	0
11*0	3	5
1******0	2	7
101010	6	5

4.2) Teorema dos esquemas

O teorema dos esquemas, enunciado por John Holland, diz que um esquema ocorrendo nos cromossomos com avaliação superior à média tende a ocorrer mais frequentemente (com a freqüência crescendo de forma exponencial) nas próximas gerações e aqueles esquemas ocorrendo em cromossomos com avaliações abaixo da média tendem a desaparecer.

Sendo mais detalhistas, podemos dizer que, sendo *n* o número de indivíduos contendo um certo esquema *s*, com média de avaliação igual a *r* e sendo *x* a média das avaliações de toda a população, então o número esperado de ocorrências de *s* na próxima geração é de *n*r/x*.

Este número não é exato por dois motivos. Primeiro porque normalmente ele não é inteiro e só podemos ter um número inteiro de indivíduos. Segundo porque o GA não é determinístico, e sim probabilístico,logo, o número tende a ser aquele calculado, mas muita sorte (ou muito azar) nos sorteios pode mudar este número.

exemplo 1 : Seja o problema de achar o máximo de x^2 entre 0 e 31. Usamos representação binária (5 bits) e em um dado instante poderíamos ter uma população (de 4 indivíduos) da seguinte forma :

Indivíduo	Avaliação
01101	169
11000	576

01000	64
10011	361
Média	292.5

Seja o esquema 1****. Há dois indivíduos que o contém e sua média de avaliação é 468.5. Logo esperamos que ele esteja presente em 468.5*2/292.5 ≈ 3.2 indivíduos.

Já o esquema 0**0* está presente em dois indivíduos com média de avaliação 116.5. Logo ele deve estar presente em 2*116.5/292.5 ≈ 0.8 indivíduos.

Note que estes números são apenas probabilísticos. No caso acima temos exatamente $3^5 = 243$ esquemas, cada um gerando um certo número de indivíduos. Consequentemente o número previsto de indivíduos será superior ao tamanho da população (também porque vários esquemas estão contidos uns nos outros, como por exemplo 11*** está contido em 1****).

Isto é verdade se não consideramos o efeito dos operadores de crossover e de mutação nos esquemas em questão. Quando estamos aplicando crossover, um corte no meio de um esquema destruí-lo-á para sempre (a não ser que o indivíduo que estiver reproduzindo com o pai que contém o esquema seja idêntico a este depois da posição de corte, mas por questões de simplificação de hipótese nós vamos ignorar esta suposição).

exemplos : Suponha que os seguintes esquemas estão reproduzindo e que por uma incrível concidência todos os pontos de corte (denotado por |) para estes indivíduos são iguais e entre a 4ª e a 5ª posição do indivíduo.

Esquema	Situação depois do corte
1**1 | ****	Íntegro
1*** | ****	Íntegro
1*** | ***0	Destruído
1**1| **1*	Destruído

Está claro nos exemplos acima que quanto maior for o tamanho do esquema ($\delta(H)$), maior a sua probabilidade de ser destruído. Obviamente um esquema de ordem 1 e tamanho zero nunca pode ser destruído, não importe onde o operador de crossover corte.

Logo, podemos reformular o teorema dos esquemas para afirmar que quanto maior a avaliação do esquema e menor o seu tamanho, mais cópias ele terá na próxima geração.

Agora temos que avaliar também o efeito do operador de mutação sobre os esquemas. Quanto maior a ordem do esquema, mais chances deste ser corrompido pelo operador de mutação.

Assim sendo, chegamos à forma final do teorema dos esquemas, que é a seguinte afirmação : "o GA tende a preservar com o decorrer do tempo aqueles esquemas com maior avaliação média e com menores ordem e tamanho, combinando-os como blocos de armar de forma a buscar a melhor solução".

O GA procura então criar soluções incrementalmente melhores através da aplicação dos operadores genéticos em esquemas de baixa ordem e alta função de avaliação. Podemos considerar os esquemas como as pecinhas de lego que nos leverão a construir grandes brinquedos[29].

A afirmação acima (e a teoria dos esquemas em geral) se baseia no princípio de que pais de boa avaliação sempre geram filhos com avaliações tão boas quanto ou melhores que as suas.

Na natureza isto tendo a ser verdade e nos GAs também. Podemos sempre citar uma série de contra-exemplos (quase todos baseados em casos patológicos ou codificações mal feitas) a este princípio, mas de uma forma geral, ele tende a funcionar. Via de regra, podemos dizer que se a codificação e afunção de avaliação forem bem escolhidas, o seu GA funcionará sem problemas.

Existem provas matemáticas para o cálculo de quantos indivíduos devem conter um certo esquema na geração seguinte à atual. Esta fórmula tem interesse fundamentalmente teórico e explicamo-na no

[29] ou soluções de alta avaliação, se você não gostar da metáfora.

apêndice A deste livro para aqueles que têm interesse na matemática da questão.

Entretanto, é importante para que todos percebam que GAs não são simplesmente "chutometria aplicada" e têm um fundamento matemático respeitável.

As conclusões deste capítulo podem então ser resumidas nas seguintes frases:

- Esquemas de baixa ordem com avaliação acima da média tendem a estar presentes em um número que cresce de forma exponencial.
- Apesar de um algoritmo genético só proecssar m estruturas de cada vez, ele acumula e explora informações sobre um número muito maior de esquemas em paralelo.
- Se o seu esquema de codificação e avaliação for relativamente eficiente, um algoritmo genético produz suas soluções através da justaposição de pequenos pedaços eficientes (os blocos básicos de construção).

5) Outros operadores genéticos

5.1) Introdução

Até agora nós usamos somente um operador genético, o operador de crossover mais mutação de bit. A partir de agora nós dividiremos este operador em dois operadores separados (um para crossover e outro para mutação) e veremos alternativas para ambos os operadores.

O motivo para separá-los é simples : separando-os nós aumentamos nosso controle sobre a operação de cada um deles. Podemos aumentar ou diminuir a incidência de cada um dos operadores sobre nossa população e assim comandar mais de perto o desenvolvimento dos cromossomos.

A partir de agora cada operador receberá uma avaliação e para decidir qual operador será aplicado a cada instante rodaremos uma roleta viciada, da mesma maneira que fazemos para selecionar indivíduos. A seleção de indivíduos é, obviamente, feita depois da seleção do operador genético a ser aplicado, visto que o operador de mutação requer somente um indivíduo enquanto o operador de crossover requer dois.

Normalmente o operador de crossover recebe uma fitness bem maior que o operador de mutação, visto que a reprodução é a grande característica de um GA (a mutação tem como função apenas preservar a diversidade genética de nossa população de soluções).

A partir de agora, criamos mais uma incerteza dentro do nosso GA (cada vez mais podemos chegar à conclusão que o nome da área deveria ser genetic heuristic!), e em rodadas infelizes podemos ter a supremacia do operador de mutação sobre o de crossover, mas a estatística diz que isto probabilisticamente é impossível[30].

[30] A ciência do caos (ramo da matemática que estuda sistemas altamente não lineares) parece não concordar muito com esta afirmação da estatística. Em uma das teorias sobre jogos, chamada de *Gambler's ruin*, ela prevê ao contrário do que a estatística e o senso comum dizem, que se apostarmos sobre uma moeda não viciada (escolhemos sempre cara e ganhamos $1 se acertamos e perdemos $1 se erramos) nós inevitavelmente iremos perder muito dinheiro. Pode parecer

5.2) Crossover de dois pontos

Existem dezenas de esquemas que o crossover de 1 só ponto não consegue preservar, como por exemplo 1******1. Consequentemente, se não mudarmos o nosso operador de crossover, nosso GA ficará limitado na sua capacidade de processar esquemas.

Para melhorar nossa capacidade de processar esquemas, podemos introduzir o crossover de 2 pontos. Seu funcionamento, que podemos ver na figura 5.1, é similar ao do crossover de 1 ponto, com uma pequena diferença : em vez de sortearmos um só ponto de corte, sorteamos dois. O primeiro filho será então formado pela parte do primeiro pai fora dos pontos de corte e pela parte do segundo pai entre os pontos de corte e segundo filho será formado pelas partes restantes.

Por exemplo, digamos que temos dois filhos de tamanho 10 dados respectivamente pelas strings *0101010101* e *1111000011*. Executamos o crossover de dois pontos nestes dois cromossomos e sorteamos os pontos de corte 4 e 8. O primeiro filho será dado então pela parte do primeiro pai até o ponto de corte 4 (*0101*), a parte do segundo pai entre o ponto de corte 4 e o ponto de corte 8 (*0000*) e a parte do primeiro pai localizada após o ponto de corte 8 (*01*). No final, o valor deste filho será *0101000001*.

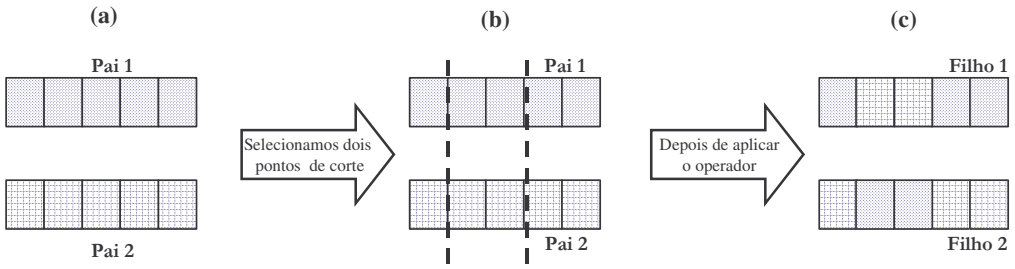


fig. 5.1 ⇒ Funcionamento do crossover de dois pontos

O código que usamos para implementar o crossover de dois pontos é o seguinte:

---

absurdo, mas o fato é que os cassinos continuam sendo um dos negócios mais lucrativos do mundo.

```
1   public ElementoGA_Avancado crossoverDoisPontos(ElementoGA
    outroPai) {
2          String aux1;
3          ElementoGA retorno=null;
4          int pontoCorte1=(new Double(java.lang.Math.random()*
                        (this.valor.length()-1))).intValue();
5          int pontoCorte2=(new Double(java.lang.Math.random()*
            (this.valor.length()-(pontoCorte1+1)))).intValue();
6          pontoCorte2+=pontoCorte1;
7          if (java.lang.Math.random()<0.5) {
8               aux1=this.valor.substring(0,pontoCorte1);
9               aux1=aux1+outroPai.getValor().substring(
                      pontoCorte1, pontoCorte2);
10              aux1=aux1+this.valor.substring(pontoCorte2,
                      this.valor.length());
11            } else {
12              aux1=outroPai.getValor().substring(0,
                     pontoCorte1);
13              aux1=aux1+this.valor.substring(pontoCorte1,
                     pontoCorte2);
14              aux1=aux1+outroPai.getValor().substring(
                     pontoCorte2,outroPai.getValor().length());
15            }
16            try {
17              retorno=(ElementoGA)
                          outroPai.getClass().newInstance();
18              retorno.setValor(aux1);
19            } catch (Exception e) {
20            }
21            return(retorno);
22        }
```

Nas linhas 4 e 5 nós sorteamos dois pontos de corte. O segundo ponto de corte (pontoCorte2) é selecionado como sendo um valor entre pontoCorte1 e o tamanho da string. Assim, na linha 6 nós adicionamos o valor de pontoCorte1 ao valor sorteado de forma a termos o valor real do segundo pontoCorte.

Um exemplo para simplificar: digamos que o tamanho da string seja 10 e que o valor sorteado para pontoCorte1 seja 4. Então pontoCorte2 receberá um valor entre 0 e 6 (ou seja, um número que representa uma quantidade de posições de pontoCorte1 até 10). Este

valor tem que ser somado ao 4 para representar o segundo ponto de corte efetivo.

Assim como no caso do crossoverUmPonto, esta função retorna apenas um filho, que pode vir com a parte esquerda do primeiro pai e direita do segundo ou vice-versa, de acordo com o teste feito no if da linha 7. Isto foi feito apenas para que houvesse um valor de retorno facilmente compreensível e a função fosse mais simples.

As linhas de 8 a 10 (e as de 12 a 14) pegam respectivamente a parte de um dos pais à esquerda do primeiro ponto de corte, a parte do outro pai entre os pontos de corte e a parte do primeiro pai à direita do segundo ponto de corte, concatenando-os para realizar o crossover como esperado.

Obviamente a operação do crossover de dois pontos é ligeiramente mais complexa do que a operação do seu equivalente de um só ponto, mas a diferença de performance conseguida em geral faz com que o custo extra (que pode ser considerado negligenciável, se considerarmos que o custo da função da avaliação tende a ser muito mais alto) seja válido.

O número de esquemas que podem ser efetivamente transferidos aos descendentes usando-se este operador aumenta de forma considerável, mas é ainda maior se usarmos o operador de crossover uniforme descrito a seguir.

5.3) Crossover uniforme

Apesar do crossover de dois pontos ser capaz de combinar vários esquemas fora da alçada do crossover de um só ponto, existem alguns esquemas que ele não é capaz de manter. Por isto, foi desenvolvido o crossover uniforme, que é capaz de combinar todo e qualquer esquema existente.

O funcionamento do crossover uniforme é o seguinte : para cade gen é sorteado um número zero ou um. Se o sorteado for um, o filho número um recebe o gen do primeiro pai e o segundo filho o gen do segundo pai, e se o sorteado for zero, o primeiro filho recebe o gen do segundo pai e o segundo filho recebe o gen do primeiro pai.

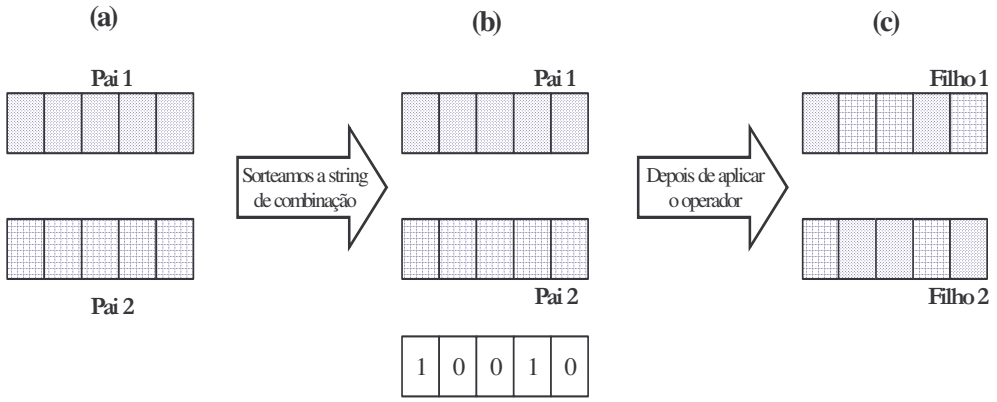


fig 5.2 ⇒ Funcionamento do crossover uniforme

Uma característica interessante do crossover uniforme é que ao contrário dos seus predecessores, que tendiam a quebrar esquemas de maior comprimento, este operador tende a conservar esquemas longos com a mesma probabilidade que preserva esquemas de menor comprimento, desde que, é claro, ambos tenham a mesma ordem.

O funcionamento do crossover uniforme é mostrado na figura 5.2 e o seu código fonte é o seguinte:

```
1   public    ElementoGA_Avancado    crossoverUniforme(ElementoGA
    outroPai)  {
2           String aux1="";
3           ElementoGA_Avancado retorno=null;
4           int i;
5           for(i=0;i<this.valor.length();i++) {
6              if (java.lang.Math.random()<0.5) {
7                 aux1=aux1+this.valor.substring(i,i+1);
8              } else {
9                     aux1=aux1+
                           outroPai.getValor().substring(i,i+1);
10             }
11          }
12          try {
13             retorno=(ElementoGA_Avancado)
                        outroPai.getClass().newInstance();
14             retorno.setValor(aux1);
15          } catch (Exception e) {
16          }
17          return(retorno);
18       }
```

O código fonte deste operador é aparentemente mais simples do que aquele de seus antecessores. Na linha 5 nós temos um loop que vai da primeira posição da string[31] até a última e para cada uma delas nós fazemos um sorteio (linha 6) para decidir se vamos pegar aquela posição de um pai (linha 7) ou de outro (linha 9).

Obviamente, este crossover tem uma grande possibilidade de estragar todo e qualquer esquema, mas em média a sua performance é extremamente superior à dos seus antecessores. Alguns pesquisadores acham que a performance do crossover de dois pontos é superior à performance do crossover uniforme, mas o que é certo é o fato que o crossover de dois pontos é mais rápido, visto que existem menos sorteios por reprodução.

5.4) Operadores com percentagens variáveis

Até agora, quando falávamos em dois operadores, nós atribuíamos a cada um deles uma percentagem fixa (tal que a soma das duas era 100%) e rodávamos uma roleta viciada de forma a escolher qual dos operadores seria aplicado sobre o indivíduo selecionado.

Mas o que acontece quando executamos um GA é que não há uma fitness que seja adequada para os dois operadores durante toda a execução do algoritmo. Na realidade, no início do GA, nós queremos executar muita reprodução e pouca mutação, visto que há muita diversidade genética e queremos explorar o máximo possível nosso espaço de soluções. Depois de um grande número de rodadas, há pouca diversidade genética na população e seria extremamente interessante que o operador de mutação fosse escolhido mais frequentemente do que o operador de crossover, para que possamos reinserir diversidade genética dentro da nossa população.

[31] Em Java as strings começam na posição 0, como no C, e não na posição 1, como no Pascal. Reclamações com a Sun, por favor!

Pensando assim, precisaríamos que a fitness do operador de crossover fosse caindo com o decorrer do algoritmo e que a fitness do operador de mutação fosse concomitantemente aumentando.

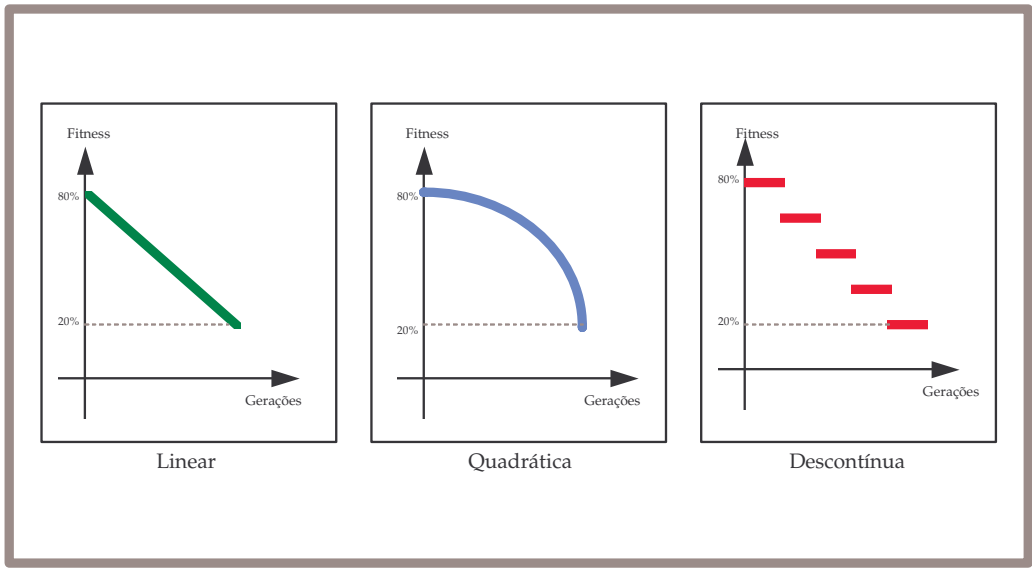


fig. 4.3 ⇒ Exemplo das técnicas de interpolação parâmetros, cada uma das quais está interpolando a fitness de um parâmetro desde 80% até o valor final de 20%.

Para tanto nós podemos interpolar as fitnesses dos dois operadores, fazendo com que o que desejemos aconteça. Um exemplo disto é fazer com que a fitness do operador de crossover começe com 80% e caia no final do algoritmo para 20%, enquanto a fitness do operador de mutação faça o caminho inverso.

Há várias técnicas de interpolação candidatas, mas entre elas podemos ressaltar as técnicas linear, quadrática e descontínua, técnicas estas demonstradas na figura 4.3.

A técnica linear simplesmente traça uma reta entre as duas fitnesses (inicial e final) e a cada geração caminha um pouquinho ao longo desta reta. A técnica quadrática faz o mesmo, só que o caminho é uma parábola. A vantagem desta técnica sobre a técnica linear é que a interpolação vai mais devagar no começo quando não precisamos que as fitnesses dos operadores mudem, mas vão crescer mais rapidamente

no final quando é extremamente importante que o operador de mutação prevaleça. A técnica descontínua simplesmente vai mudando aos saltos o valor da fitness do operador, fazendo com que a cada k gerações a fitness mude um pouco. Esta é a relativamente mais difícil de operar, visto que não implica em somente uma aplicação direta de fórmula matemática, como as outras duas faziam, mas sim consiste de um if seguido da operação.

Normalmente as três técnicas de interpolação nos dão resultados finais extremamente parecidos, no que tange à performance do GA, logo costuma-se escolher a técnica linear por ser a mais simples de todas.

5.5) Operador de mutação dirigida

A grande maioria dos pesquisadores procura resolver o problema da estagnação tornando a mutação mais provável que o cruzamento de duas soluções. Assim, há a tendência que novamente venha a surgir uma variedade genética dentro da população de soluções.

A questão é que, usando o operador de mutação tradicional, todas as partes de cada uma das soluções têm igual probabilidade de serem modificadas por este operador, sem distinção.

Mas em vários casos o problema pode estar concentrado no esquema dominante entre as melhores soluções (que tendem a ser escolhidas com mais frequência). Se este esquema não for modificado de forma agressiva, pode ser que não cheguemos a lugar algum, por mais que usemos o operador de mutação.

Uma maneira de implementar esta modificação do esquema dominane é criar um novo operador de mutação, que procurasse se concentrar neste esquema dominante, de forma a criar variedade genética dentro das áreas que nos interessam.

O operador funciona de maneira simples. Ele só começa a agir depois de um grande número de gerações. Quando ativado, ele busca as *n* melhores soluções dentro da população padrão e verifica qual é a bagagem cromossomial que elas têm em comum.

Depois de descoberto qual é o esquema em comum, o operador realiza as mutações (dentro deste esquema somente). Isso vai fazer com

que surjam novas soluções com bagagem genética radicalmente diferente daquela que domina a população naquele instante.

Obviamente, há alguns problemas na utilização deste novo operador, sem dúvida alguma.

O primeiro é a questão do ajuste dos parâmetros. Se o *n* (número de soluções pesquisadas) for pequeno ou grande demais, podemos não encontrar o esquema que realmente domina as melhores soluções. Se o número de rodadas decorridas antes de começarmos a aplicar este operador não for grande o suficiente, então este estará usado sem necessidade, visto que ainda há variedade genética na população. Já se o número de rodadas for grande demais, um grande número de rodadas sem valor terão decorrido, visto que as soluções produzidas não avançam muito em relação à geração anterior.

O segundo problema é a questão da sobrevida destas novas soluções. Aplicando mutação exatamente sobre o esquema dominante, temos a tendência de gerar soluções ruins para o problema (visto que, em média, é exatamente este esquema que faz com que as soluções sejam boas). Assim, temos que conseguir algum jeito de manter estas novas soluções "vivas" e sendo escolhidas pelo nosso módulo de seleção.

O terceiro problema é descobrir quando vale a pena continuar a aplicar este operador. Em algum momento já teremos introduzido variedade genética suficiente e podemos deixar o algoritmo voltar a rodar normalmente. A solução mais simples para este problema é limitar o número de indivíduos criados por este novo operador. Quando este limite tiver sido alcançado, o operador será novamente inibido.

6) Outros módulos de populacão

6.1) Elitismo

Elitismo é uma pequena alteração no módulo de população que quase não altera o tempo de processamento, mas que garante que a performance do GA está sempre crescendo com o decorrer das gerações.

A idéia básica por trás do elitismo é a seguinte : os n melhores indivíduos de cada geração não devem "morrer" junto com a sua geração, mas sim passar para a próxima geração para garantir que seus genomas seja preservado.

Visto que a maioria dos esquemas de avaliação de performance de um GA medem apenas a adequação da melhor solução dentre todos os indivíduos, a manutenção do melhor indivíduo da geração k na população da geração k+1 garante que o melhor indivíduo da geração k+1 é pelo menos igual que o melhor indivíduo da geração k (no pior caso, em que nenhum indivíduo melhor é criado).

O overhead de processamento é realmente muito pequeno, visto que já temos que determinar a avaliação de cada indivíduo para aplicação da roleta, logo basta que armazenemos o índice do melhor indivíduo e o módulo de população encarregar-se-á de copiá-lo para a próxima geração. Se há n indivíduos e o algoritmo executa durante m gerações então o overhead de operações é da ordem de mn. Note-se que este valor é baixo, visto que em média temos cerca de 70 indivíduos executando por não mais de 80 gerações, o que faz com que tenhamos cerca de 6000 operações como overhead, o que não chega a colaborar com uma fração de segundo sequer para o tempo final de execução do nosso GA.

6.2) Steady state

Nos GAs que vimos até agora toda uma geração nasce ao mesmo tempo enquanto que a geração anterior morre também toda de uma vez e é substituída por esta nova geração.

No "mundo real" não é isto que acontece. Indivíduos vão nascendo aos poucos e os mais velhos vão morrendo de forma lenta e há interação entre as gerações. Isto quer dizer que indivíduos da geração *k+1* podem reproduzir com indivíduos da geração *K* (sexismo e preconceitos de idade à parte).

A idéia por trás da técnica de steady state é exatamente reproduzir este tipo de característica natural das populações biológicas. Ao invés de criarmos uma população completa de uma só vez, vamos criando os "filhos" um a um (ou dois a dois, por ser mais conveniente para o operador de crossover) e substituindo os piores "pais" por estes novos indivíduos.

Algumas implementações de GA substituem pais aleatórios em vez de substituir necessariamente os piores pais. Apesar de isto parecer loucura, pois estamos mantendo indivíduos “piores” em detrimento de outros com melhores avaliações, é importante considerar que manter somente os melhores faz com que tenhamos uma tendência maior à convergência genética (situação em que todos os indivíduos são muito parecidos e a reprodução praticamente não gera indivíduos novos).

Usando o steady state o conceito de geração dentro do nosso GA fica muito difuso, quase inexistente, e pode haver reprodução entre indivíduos recém criados e indivíduos da geração anterior (até mesmo incesto!). Isto permite uma maior dominação dos melhores esquemas, e normalmente faz com que a população convirja mais rapidamente.

Utilizar steady state embute algumas questões interessantes. Por exemplo, seja a nossa população de tamanho *p*, e admita que recém criamos um indivíduo e inserimo-lo na população. Ele pode ser descartado já na próxima reprodução (supondo que ele seja o pior) ou vamos preservá-lo até que sejam criados mais *p* indivíduos? Se descartarmos o indivíduo rapidamente estaremos possivelmente diminuindo cada vez mais a diversidade genética da nossa população, mas se nós não os descartarmos, demoraremos mais para convergir.

Outro ponto interessante do steady state é que caso o indivíduo reproduza com um dos seus pais há uma chance muito grande de que

os filhos desta "relação" sejam exatamente iguais aos pais, ou que pelo menos não acrescentem nada de novo à diversidade genética da população. Isto é impossível de evitar, pois marcar quem são os pais de cada indivíduo é uma tarefa praticamente impossível, pois implicaria em manter uma tabela enorme que tem dezenas de campos alterados a cada "reprodução" de dois indivíduos.

6.3) Steady state sem duplicatas

Um dos grandes problemas do steady state é que há uma convergência muito rápida da população, com a consequente diminuição da variedade genética, que ocorre principalmente quando deixamos os filhos desaparecem logo após serem gerados.

O motivo principal para isto é que os filhos tendem a ter avaliações razoavelmente semelhantes às dos seus pais pois herdam os principais esquemas destes. Consequentemente a população tende a se reunir em "clusters" de indivíduos e os "clusters" com maiores avaliações médias tendem a predominar (o que é desejável, mas não rápido demais, pois senão, sem a variedade genética, pode-se perder o paralelismo intrínseco dos GAs e acabar ficando preso em um máximo local).

Para evitar que a convergência seja rápida demais, pode-se usar a técnica de steady state sem duplicata. Esta é praticamente idêntica à técnica de steady state, com a pequena diferença que se o indivíduo gerado for idêntico a algum já presente na população ele é descartado.

A ténica de steady state sem duplicata tende a conseguir melhores resultados que a técnica de steady state, pois há em média mais variedade genética com a qual o algoritmo pode trabalhar e a população converge mais lentamente. Mas, em contrapartida, ganhamos um overhead grande de teste para verificar se o indivíduo já está presente na população : se há *p* indivíduos com *g* genes e o algoritmo executa durante *n* gerações então nós estaremos realizando mais *png* operações do que estaríamos realizando se utilizássemos a técnica de steady state simples.

7) Outros tipos de função de avaliação

7.1) Introdução

Até agora nós usamos como medida de avaliação do indivíduo o valor exato fornecido pela função de avaliação, procedimento este que é conhecido pelo nome de *fitness is evaluation*.

Este procedimento, que é bom na maioria dos casos, pode degenerar em vários outros, fazendo com que a performance do GA fique bem pior, ou até mesmo impedindo-o de funcionar direito. A seguir, listamos alguns destes casos.

caso 1 : Superindivíduo

Algumas vezes há um ou mais indivíduos que são extremamente melhores que os outros indivíduos da população. Neste caso, este indivíduo ou este grupo será permanentemente selecionado pelo módulo de seleção, causando uma perda imediata da diversidade genética nas gerações imediatamente subsequentes.

exemplo : Seja a seguinte população, com as seguintes avaliações :

Indivíduo	Avaliação
10000	256
00100	16
00001	1
00011	9
00010	4

Obviamente, usando o método da roleta viciada para seleção dos indivíduos reprodutores, teremos uma probabilidade que o primeiro indivíduo seja selecionado aproximadamente 256/286 ≈ 90% das vezes. Isto fará com que percamos as características benéficas de vários outros indivíduos (que é o 1 em cada posição, pois como o leitor poderá perceber, a função de avaliação é x^2!).

caso 2 : Pequena diferença entre as fitness

Muitas vezes pode ocorrer que todos os indivíduos têm funções de avaliação que diferem muito pouco percentualmente. Na maioria dos casos em que isto ocorre, uma pequena diferença entre funções de avaliação significa uma grande diferença na qualidade da solução, mas o algoritmo não consegue perceber isto, dando espaços praticamente iguais para todos os indivíduos na roleta viciada.

exemplo : Seja o algoritmo para encontrar o máximo da seguinte função

$$f6(x,y) = 999.5 - \frac{\left[\sin\left((x^2 + y^2)^{1/2} \right)\right]^2 - 0.5}{\left[1.0 + 0.001(x^2 + y^2)\right]^2}$$

Neste caso, todas as avaliações dos indivíduos se concentrarão no intervalo [999,1000], mas a solução que no fornece 1000 é muito superior a todas as outras! Mas, se tivermos dois indivíduos, um com avaliação 999.001 e outro 999.999, apesar do segundo ser claramente melhor que o primeiro como solução do nosso problema, eles receberão espaços quase idênticos dentro da nossa roleta viciada.

Agora que vimos algumas das motivações para alterarmos nossa função de avaliação. Veremos agora como resolver cada um dos problemas descritos acima.

7.2) Normalização linear

Esta técnica pode ser descrita da seguinte maneira : ordene os cromossomos em ordem decrescente de valor. Crie novas funções de avaliação para cada um dos indivíduos de forma que o melhor de todos receba um valor fixo (*k*) e os outros vão recebdo valores iguais ao valor do indivíduo imediatamente anterior na lista ordenada menos um valor de decremento constante (*t*).

Ou seja, podemos resumir esta técnica na seguinte fórmula recursiva:

$$aval_0 = k$$
$$aval_i = aval_{i-1} - t$$

Esta técnica resolve o problema do superindivíduo e o problema de aglomeração das funções de avaliação, mas em contrapartida cria mais um problema : há mais dois parâmetros para otimizar.

Apesar de a primeira vista na parecer, a escolha de *k* e de *t* é crítica para o desempenho do sistema. Um valor de *t* muito pequeno faz-nos ficar em uma situação extremamente parecida àquela especificada no caso nº 2 da introdução deste capítulo, enquanto que um valor muito grande pode criar desigualdades artificias entre indivíduos que anteriormente tinham valores de avaliação extremamente próximos.

7.3) Normalização não linear

Este método consiste em passar os valores da função de avaliação por uma função não linear que possa ser aplicada para fazer com que as avaliações sejam mais discriminadoras das diferenças entre os indivíduos.

O problema neste tipo de normalização é encontrar uma função que atenda nossos propósitos, resolvendo nossos problemas sem criar novas situações difíceis de resolver pelo nosso GA.

Por exemplo, o problema do superindivíduo pode ser resolvido usando como função de normalização a função log. Se usássemo-la nos indivíduos especificados no exemplo do 1º caso da introdução teríamos a seguinte situação:

Indivíduo	Avaliação	Nova avaliação
10000	256	2.41
00100	16	1.20
00001	1	0
00011	9	0.95
00010	4	0.60

Note-se como agora não existe mais o problema do superindivíduo, mas mesmo assim a diferença de qualidade entre as várias soluções está presente em suas novas avaliações.

Diferentemente, funções de avaliação não lineares podem ser utilizadas para punir com mais rigor soluções "ruins", aumentando a pressão evolucionária em favor das "boas" soluções. O tipo de função que faz isto são aquelas do tipo x^n, onde $n>1$ para casos de funções de avaliação cujos módulos maiores que 1 e $n<1$ para casos de função de avaliação no intervalor [0,1].

7.4) Windowing

Esta técnica pode ser resumida da seguinte maneira : ache o valor mínimo dentre as funções de avaliação da nossa população. Designe para cada um dos cromossomos uma avaliação que seja igual à quantidade que excede este valor mínimo.

Isto serve para resolver o nosso segundo problema citado na introdução. No exemplo citado, por exemplo, em vez das avaliações irem de 999.001 a 999.999 elas passarão a ir de 0.001 a 0.999, voltando a haver uma grande pressão seletiva em favor do melhor indivíduo (que agora passará a ganhar um espaço 999 vezes maior que o pior indivíduo na nossa roleta viciada).

O único problema com este método é que ele designa um valor de avaliação zero para o pior indivíduo, o que garante que seus gens não serão transplantados para a próxima geração. Para evitar isto, pdemos designar um mínimo valor de avaliação para cada indivíduo, e se a avaliação de um indivíduo específico estiver abaixo deste valor então a ele será designado o valor mínimo, o que garante que todos os indivíduos terão um espaço razoável detro da roleta.

Note-se que este método não resolve o problema do superindivíduo. Logo, o desenvolvedor deve ter consciência do tipo de problemas que está enfrentando antes de optar por esta específica modificação da sua função de avaliação.

8) GA baseado em ordem

8.1) Introdução

Existe uma classe de problemas que não consiste de otimização numérica, mas sim de otimização combinatorial e que pode ser resolvida perfeitamente através de GAs.

Estes problemas são em geral NP-completos (NP vem de non-polinomyal, significando que ao invés do espaço de busca ser função de uma potência do número de incóginitas, ele é função da fatorial deste número), o que significa que seu espaço de busca é, para efeitos práticos, considerado como infinito.

Dois problemas típicos deste caso são o problema de colorir um grafo e o problema do caixeiro viajante.

O problema de colorir um grafo consiste em termos um grafo com *n* nós, cada um com um peso distinto e lhe são dadas *k* cores para colorir este grafo (*k*<*n*). O problema consiste em conseguir o maior escore possível somando os pesos dos nós coloridos, sendo que dois nós adjacentes (isto é, ligados por um link) não podem receber a mesma cor.

O problema do caixeiro viajante é extremamente semelhante. Um caixeiro tem de percorrer *n* cidades, sem passar duas vezes por nenhuma e percorrendo a menor distância possível. Qual o caminho que ele tem de percorrer?

Para podermos resolver este problema através de GAs, temos que entender como as seguintes questões devem ser resolvidas:

- Qual a representação adotada?
- Para esta representação, qual é o mecanismo dos operadores genéticos?
- Qual foi a função de avaliação utilizada?

Estas três questões são fundamentais para resolução de problemas através de algoritmos genéticos, e aparecerão neste texto toda vez que quisermos encontrar uma maneira de resolver um problema real, diferente de otimização numérica, através de GAs.

8.2) Representação e função de avaliação

No caso dos dois problemas que melhor representam este problema, nós estamos interessados na ordem em que o problema é resolvido (no caso do grafo, a ordem em que os nós serão coloridos e no caso do problema do caixeiro viajante, a ordem em que as cidades serão percorridas).

Logo, queremos uma representação que contenha todos os nós (ou cidades) colocados em uma ordem para a resolução do problema, o que nos leva a optar pela representação em lista.

Esta representação consiste em uma lista de todos os elementos presentes no problema (todas as cidades ou todos os nós do grafo) colocado em uma ordem qualquer.

exemplo :

- (1 4 6 5 2 3 7) é um cromossomo válido
- (1 6 2 5 7 4) não é um cromossomo válido, visto que o elemento 3 não está presente na nossa lista.

A função de avaliação deve, como já sabemos, deve representar a qualidade de cada um dos cromossomos, e é a única conexão profunda que temos entre o GA e o nosso problema. Logo, a maneira pela qual resolvemos o problema influencia na escolha da nossa função de avaliação, o que nos faz concluir que outras funções de avaliação distintas das que explicitaremos a seguir são possíveis.

No caso do problema do caixeiro viajante basta somar a distância entre a cidade contida no gene *i* à cidade contida no gene *i+1*, para i variando de 0 até o número de genes menos um.

No caso do problema do grafo, pegaremos os nós um a um, na ordem fornecida pelo cromossomo e designar-lhes-emos a primeira cor possível. Se houver alguma cor, somaremos seu peso e, caso seja impossível colori-lo com qualquer uma das cores que nos foram inicialmente designadas nós não somaremos seu peso à avaliação deste cromossomo. Um exemplo deste processo é mostrado na figura 7.1.

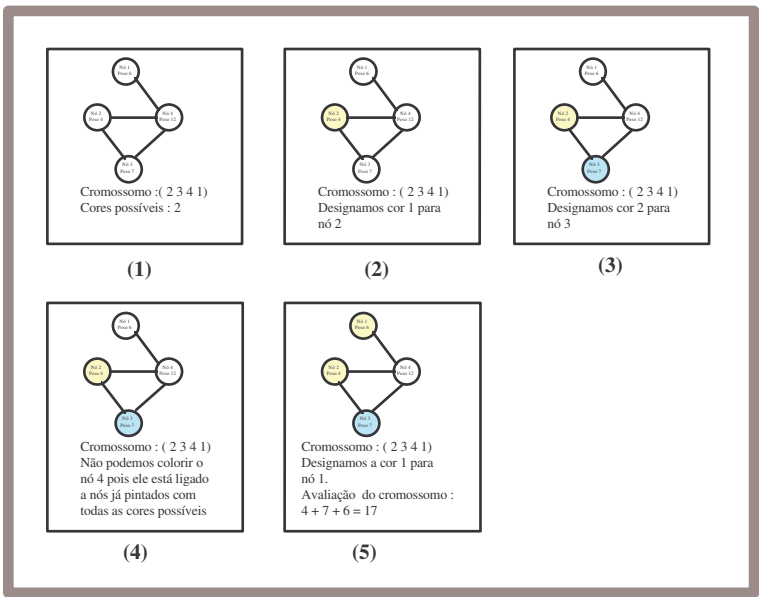


fig. 7.1 $\Rightarrow$ Demonstração do processo de avaliação de um cromossomo da representação baseada em ordem para solução do problema de colorir um grafo.

8.3) Operador de crossover baseado em ordem

O nosso operador de crossover para GAs baseados em ordem é uma versão especial do operador de crossover uniforme. As pequenas diferenças existentes decorrem das características especiais dos cromossomos da representação baseada em ordem.

Nós não podemos simplesmente copiar posições do primeiro pai quando sortearmos um 1 e copiarmos posições do segundo pai quando sortearmos um 0, pois isto poderia fazer com que gerássemos um cromossomo com elementos repetidos, como vemos no exemplo da figura 7.2.

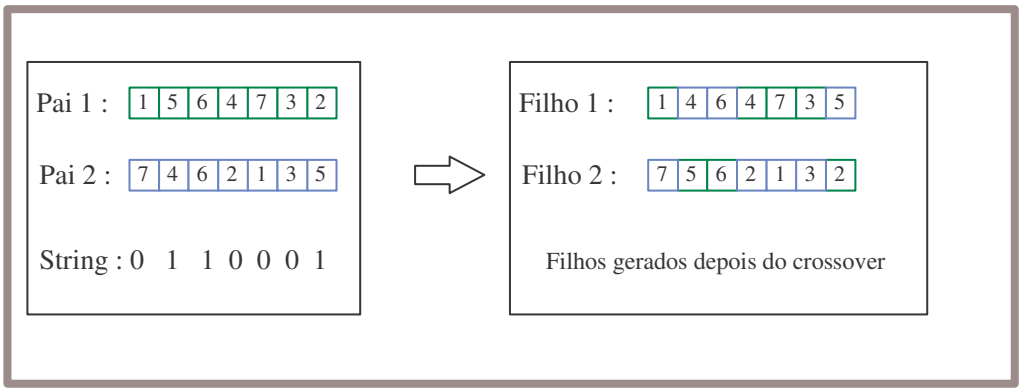


fig 7.2 ⇒ Exemplo de utilização do crossover uniforme não adaptado para representação baseada em ordem. Note-se como cada um dos filhos gerados tem um elemento repetido, o que é proibido na representação baseada em ordem.

Temos então que modificar a forma de atuação do crossover uniforme de modo que geremos sempre filhos válidos dentro deste nosso novo esquema de representação.

A partir de agora, o que nós iremos preservar não é a posição absoluta de um gen, mas sim as ordens relativas dos gens, isto é, no caso do pai 1 da figura 7.1, o importante é que o 6 vem antes do 7 e não que o 6 está na posição três e o 7 na posição cinco.

Isto pode ser entendido quando estendemos o conceito de esquema para representação baseada em ordem. Esquema agora é toda sub-lista de nosso cromossomo, sendo que os don't cares correspondem a simplesmente ignorar a posição do pai. Exemplos de esquemas para o pai 1 da figura 7.1 são (1 6 7), (1 4 3 2), (1 5), (6 4 7), etc.

A partir desta nova visão dos cromossomos, gens e esquemas, podemos chegar ao seguinte algoritmo para atuação do crossover baseado em ordem :

Passo 1	• *Gere uma string de bits aleatória do mesmo tamanho que os elementos (assim como no crossover uniforme)*
Passo 2	• *Copie para o filho 1 os elementos do pai 1 referentes àquelas posições onde a string de bits possui um 1*
Passo 3	• *Faça uma lista dos elementos do pai 1 referentes a zeros da string de bits*

Passo 4	• *Permute esta lista de forma que os elementos apareçam na mesma ordem que no pai 2*
Passo 5	• *Coloque estes elementos nos espaços do pai 1 na ordem gerada no passo anterior*
Analogia	• *Repita o processo para gerar o filho 2, substituindo o pai 1 pelo 2 e vice-versa*

Um exemplo deste processo pode ser visto na figura 7.3 a seguir.

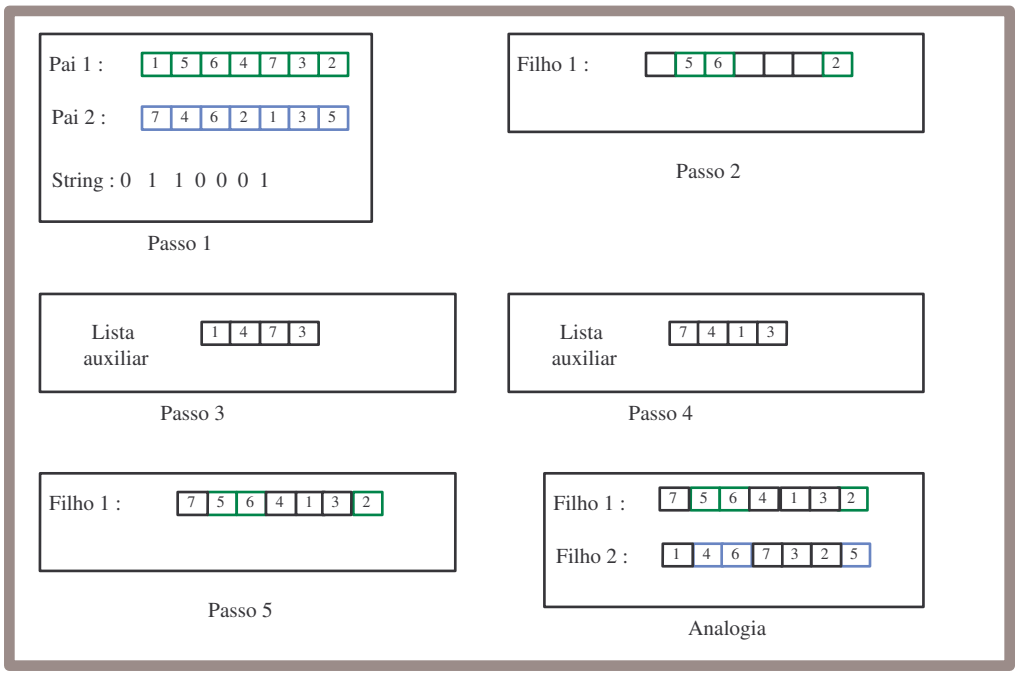


fig. 7.3 ⇒ Exemplo da atuação do operador de crossover baseado em ordem

A princípio este operador pode parecer bastante complexo, mas sua utilização é na realidade simples de impementar sobre as nossas estruturas de dados e agora conseguimos preservar os esquemas tanto do primeiro quanto do segundo pai, mantendo nossa forte analogia com os processos de reprodução naturais.

8.4) Operador de mutação baseado em ordem

A mutação realiza mudanças locais em cromossomos. No caso de representação baseada em ordem, não há bits a inverter e não podemos designar valores aleatoriamente, pois poderíamos ter repretições de alguns nós enquanto outros ficam de fora. Logo, temos operar com diversos gens de um mesmo cromossomo simultaneamente. Existem duas maneiras de fazê-lo : a permutação de elementos e a mistura de sub-listas.

A permutação de elementos é simples : escolhem-se dois elementos ao acaso dentro do nosso cromossomo e trocam-se as suas posições. O processo é mostrado na figura 7.4a.

O operador de mutação que utiliza mistura de sublistas é igualmente simples. Escolhem-se duas pontos de corte dentro do nosso cromossomo, pontos estes que delimitarão uma sub-lista. Para realizar a mutação basta-nos então fazer uma permutação aleatória dos elementos desta sub-lista. Este processo é mostrado na figura 7.4b.

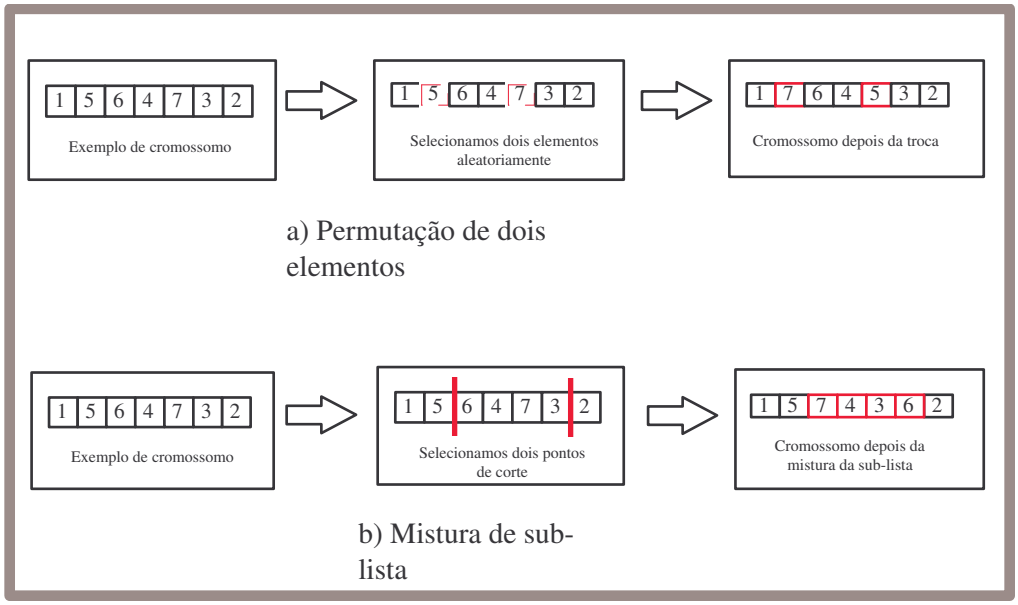


fig. 7.4 ⇒ Exemplo de operador de mutação baseado em ordem

Lawrence Davis, em seu livro clássico, afirma que o operador de mistura de sub-lista é muito mais eficiente que o operador de mutação baseado em permutação de elementos.

Obviamente tal declaração é totalmente empírica, entretanto em muitos casos ela é verdadeira. O operador de mistura de sub-lista tende a ser bastante agressivo apenas com uma fração do esquema usado, mantendo intactos outros componentes. Quando os elementos mantidos são aqueles que faz o cromossomo ter uma avaliação alta, resultados mais satisfatórios tendem a ser atingidos mais rapidamente.

9) Sistemas híbridos

9.1) Introdução

Neste capítulo veremos exemplos de como podemos utilizar GAs para melhorar a performance de outras técnicas de inteligência computacional. Seguindo a filosofia desta apostila, este capítulo não pretende ser exaustivo, pretendendo dar apenas uma visão sobre a área. Outras maneiras de hibridizar tecnologias existem e podem até ser melhor do que as expostas aqui.

A hibridização pode ser uma estratégia válida para passar por cima de certos problemas existentes em cada uma das áreas de inteligência computacional.

Os algoritmos genéticos, como já vimos anteriormente, são extremamente eficientes em varrer espaços de busca virtualmente infinitos, o que lhes confere autoridade para serem utilizados como otimizadores dos parâmetros de redes neurais e de lógica nebulosa.

Para podermos hibridizar nossas técnicas inteligentes temos antes de respoder às seguintes perguntas, e consequentemente teremos uma solução pronta para ser usada:

- Qual a representação adotada?
- Para esta representação, qual é o mecanismo dos operadores genéticos?
- Qual foi a função de avaliação utilizada?

9.2) GA + Fuzzy

9.2.a) Lógica Fuzzy

Quantas vezes nós já comentamos com alguém que Fulano é muito alto? Provavelmente muitas. Entretanto a pergunta é: qual é a altura mínima para dizer que alguém é muito alto? Alguns dirão que é

1,9m, outros que é 1,8m, mas todas as respostas são insatisfatórias, pois qualquer uma delas assume que uma pessoa que tem exatamente um centímetro a menos (uma diferença quase imperceptível) que o limite estabelecido não pertence de forma alguma à categoria dos "muito altos".

Os programadores estão muito acostumados a esta situação. Em um programa feito em uma linguagem de programação fictícia (portugol) teríamos algo como:

SE altura>1,9m ENTÃO
ESCREVA "É muito alto"
SENÃO
ESCREVA "Não é muito alto"
FIM SE

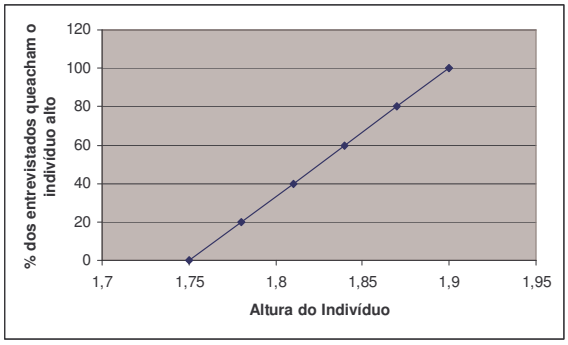
O que este código quer dizer é que se a pessoa tem mais de 1,9m então ela pertence ao conjunto dos muito altos com 100% de certeza e se ela tiver 1,9m ou menos, então ela não pertence ao conjunto dos muito altos (pertinência de 0%). Isto é compatível com o que chamamos de lógica tradicional (*crisp logic*) que é tradicionalmente aplicada em computadores, nos quais só existem o conceito de verdadeiro e sua negação, o falso.

Obviamente isto não é verdade no mundo real. Existem graus de pertinência ao conjunto dos muito altos diferentes de 0 e 100%. Nós podemos achar que um indivíduo é "mais ou menos muito alto", "quase muito alto" e assim por diante.

Por exemplo poderíamos usar a opinião das pessoas como medida, então poderíamos pesquisar um grupo de pessoas e chegar às seguintes opiniões:

- Se o indivíduo tem 1,9m ou mais, então 100% dos entrevistados acham que ele é muito alto.
- Se o indivíduo tem 1,87, então 80% acham que ele é muito alto
- Se o indivíduo tem 1,84, então 60% acham que ele é muito alto
- Se o indivíduo tem 1,81, então 40% acham que ele é muito alto
- Se o indivíduo tem 1,78, então 20% acham que ele é muito alto
- Se o indivíduo tem 1,75, então ninguém (0%) acha que ele é muito alto.

Nós poderíamos então fazer um gráfico representando a opinião de nossos entrevistados, gráfico este que ficaria mais ou menos assim:



Agora nós temos uma medida razoável do que é ser alto, que é altamente compatível com aquilo que pensamos no mundo real e queremos usá-la dentro de um programa de computador.

Para tanto, precisamos aplicar a lógica nebulosa (fuzzy logic) que é uma generalização da lógica tradicional. Usando lógica nebulosa nós podemos ter grau de pertinência diferentes de 100% e 0%, e podemos fazer os programas de computador agirem usando uma linha de raciocínio mais próxima daquela que usamos no nosso cotidiano.

Isto é, o mundo não é composto de verdades e falsidades absolutas. Alguém pode pertencer 80% ao conjunto dos altos, um carro pode pertencer 20% ao conjunto dos carros que estão andando rápido e assim por diante.

É importante apenas não confundir o grau de pertinência com probabilidade. Quando dizemos que um valor pertence com um certo grau a um conjunto não há nenhuma incerteza relacionada a isto. *A probabilidade é uma dúvida enquanto que a pertinência na lógica nebulosa é uma certeza.*

Vejamos um exemplo. Se eu disser que existe uma probabilidade de 80% de um carro na BR101 estar rápido, eu não sei qual é a sua velocidade, não sei nem mesmo se aquele carro está andando em alta velocidade (tudo que eu sei é que se eu medir a velocidade deste carro, 8 e cada 10 vezes ele deve estar trafegando rapidamente).

Em contrapartida, quando em digo que um carro está em uma velocidade que pertence 80% ao conjunto dos carros rápidos, todos nós sabemos exatamente a velocidade em que ele está trafegando. Para tanto, basta olhar para o gráfico do conjunto dos carros rápidos e ver qual das velocidades corresponde a 80% de pertinência.

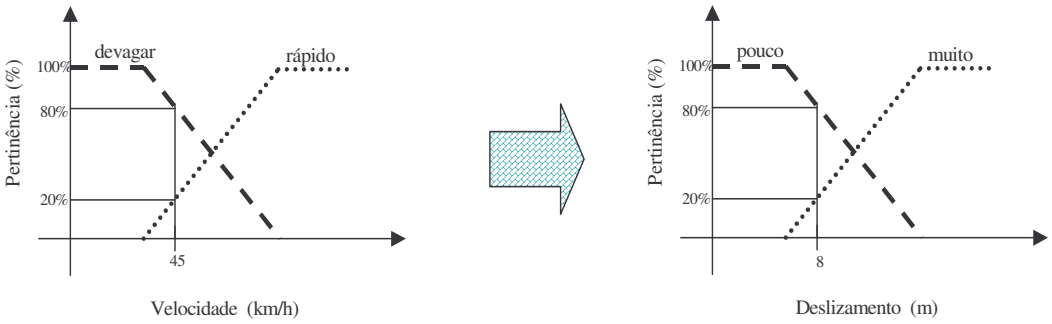
Isto posto, temos que pensar em como usar isto dentro de um programa. Para tanto, basta entender que criamos uma base de conhecimento contendo regras do tipo:

- **Regra 1:** Se o carro estiver rápido, então ele vai deslizar muito quando frear
- **Regra 2:** Se o carro estiver devagar, então ele vai deslizar pouco quando frear

Cada um dos antecendentes tem um determinado grau de verdade dado pela velocidade atual do carro e sua pertinência no conjunto explícito na regra. Conseqüentemente podemos determinar a pertinência de cada um dos conjuntos do conseqüente e, com base nesta pertinência, determinar quanto o carro vai deslizar antes de parar.

Para isto ficar mais claro, nada melhor que um exemplo. Imagine que as pertinências nos conjuntos explícitos nas regras acima são dadas pelos gráficos da figura 8.1.

Como podemos ver no gráfico da esquerda, se um carro está andando a 45 km/h, isto significa que ele pertence 20% ao conjunto dos carros que andam rápido Logo, pela regra 1, nós sabemos que o carro pertence 20% ao conjunto dos carros que deslizam muito antes de frear.



O mesmo é feito com a regra 2 e, graças ao fato de que 45 km/h corresponde a uma pertinência de 80% no conjunto dos carros que

andam devagar, nós sabemos que este é o grau de pertinência no conjunto dos carros que deslizam pouco.

Traçamos então uma reta que intercepta o gráfico do conjunto do pouco deslizamento no ponto do 80% e o conjunto do muito deslizamento no ponto de 20% e o ponto em que esta regra toca o eixo das abcissas (horizontal) é o deslizamento do carro. No caso, 8m (sem nenhuma chance de ser outro valor, como seria o caso em um modelo baseado em probabilidades).

É claro que a matemática da lógica nebulosa é mais complexa do que este modelo simples e que criar as regras é uma tarefa longa e dependente de um especialista na área sendo estudada. Entretanto, os conceitos básicos da lógica nebulosa são estes que colocamos aqui.

Hoje em dia existem vários produtos que usam a lógica nebulosa para poder controlar de forma mais precisa seu desempenho, incluindo aparelhos de ar-condicionado (termostato), máquinas fotográficas (controle de foco) e até mesmo elevadores (controle de velocidade e frenagem).

9.2.b) Usando GA em conjunto com a lógica fuzzy

Tecnologias híbridas em geral têm um desempenho melhor do que cada uma das tecnologias separadas. O objetivo ao aplicar GA à lógica fuzzy é utilizar a grande capacidade dos GAs como heurística para encontrar pontos de operação próximos do ótimo a um programa usando lógica fuzzy de forma a encontrar um excelente conjunto de regras ou os pontos de definição dos conjuntos fuzzy (substituindo um especialista humano).

Quando estamos trabalhando com as regras, estamos tentando descobrir um conjunto de regras que traduzam da melhor forma possível o conhecimento sobre a área em questão, sem consultar a um especialista humano.

Há uma forma de fazê-lo é aplicável quando todo o conhecimento possível e imáginário (tanto o prático quanto aquele sem nenhum siginificado) pode ser descrito por um número finito de regras. Neste caso cada regra pode ser receber um número de ordem e o GA trabalha

com sequências de bits que descrevem os números de ordem de *n* regras, aplicando-lhes os operadores genéticos e trabalhando com elas.

Por exemplo, seja o caso que estamos querendo escolher 5 regras de um universo de 81 possíveis. Precisaríamos de 7 bits para descrever todas as regras possíveis e um total de 35 bits em cada um dos nossos indivíduos e poderíamos ter um indivíduo igual a :

000001	1000001	1010000	0110000	0000100

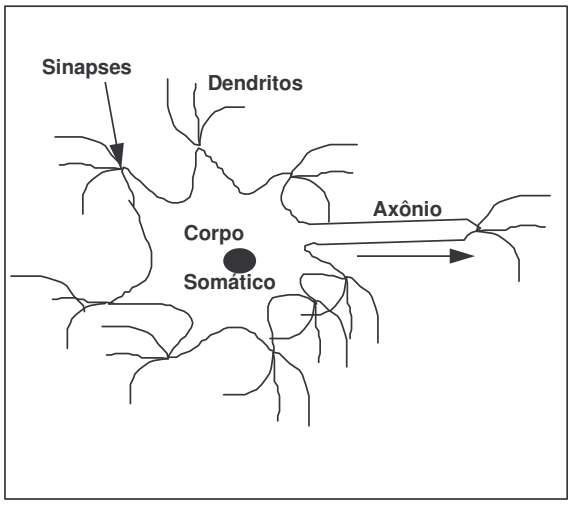
que significa que o conjunto de regras a ser utilizado é tal que contém as regras 1, 65, 80, 48 e 8.

Verificamos claramente que temos um problema de representação. Afinal temos 81 regras possíveis e nossos cromossomos representam regras de 0 a 127! Precisamos aplicar algum tipo de filtro na nossa população, seja eliminando os indivíduos *inválidos* seja punindo-os com uma avaliação sofrível, de forma que eles acabem sendo eliminados naturalmente pelo GA. Outra forma inteligente é desprezar as regras inválidas em cada indivíduo e utilizar somente as regras válidas. Este indivíduo provavelmente vai ter uma pior performance visto que usará menos regras para modelar o conhecimento e provavelmente sumirá, mas sua base genética não será desperdiçada.

9.3) GA + Redes neurais

8.3.a) Redes Neurais

As redes neurais são um tema da computação altamente inspirada na natureza que nos cerca. Durante anos e anos os homens trabalharam para fazer computadores mais rápidos e mais potentes, mas apesar do seu incrível poder computacional estes computadores falhavam em fazer tarefas que uma criança de 3 anos faria imediatamente, como reconhecer uma pessoa ou aprender algo novo só com a experiência.



Resolveu-se então buscar criar um modelo computacional que emulasse o comportamento do cérebro humano. Criaram-se neurônios artificiais extremamente similares aos humanos e interligaram-nos para formar redes que mostraram poder fazer tarefas antes restritas aos cérebros.

Além disso, os pesquisadores encontraram nas redes neurais outras características semelhantes às do cérebro : robustez e tolerância a falhas, flexibilidade, capacidade para lidar com informações ruidosas, probabilísticas ou inconsistentes, processamento paralelo, arquitetura compacta e com pouca dissipação de energia.

Encontrou-se uma arquitetura capaz não só de aprender como também generalizar. Daí podemos entender o frisson do meio científico em relação a esta área.

É importante que se entenda que as redes neurais não são a solução dos problemas computacionais da humanidade. Elas nunca superarão as arquiteturas tradicionais no campo da computação numérica, por exemplo.Mas em alguns campos elas estão se tornando ferramentas valiosas.

A capacidade de extrair características semelhantes de padrões aparentemente ruidos e incompatíveis torna as redes neurais uma ferramenta extremamente interessante para ser usada no campo da

previsão de séries temporais. As redes neurais podem encontrar recorrências e padrões onde o olho humano só vê ruído, e tudo isto sem uma única fórmula matemática.

Para entender como funcionam as redes neurais, podemos dar uma rápida olhada nos neurônios biológicos. O neurônio é a unidade fundamental constituinte do sistema nervoso. Como podemos ver na figura 8.2 a seguir, ele é constituído de um volumoso corpo central denominado pericário no qual são produzidos os impulsos nervosos e de prolongamentos finos e delgados através dos quais estes impulsos são transmitidos e recebidos. Fundamentalmente existem dois tipos de prolongamentos:

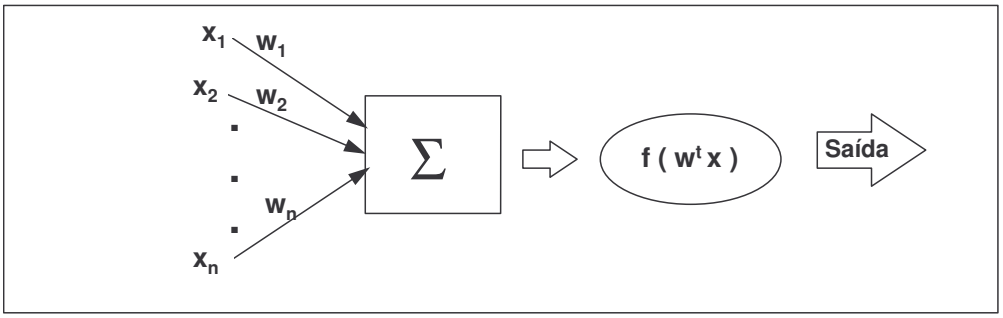
- os dendritos (ou dendrônios) : mais curtos e ramificados, através dos quais são recebidos os impulsos nervosos provenientes dos órgãos receptores e que se destinam ao corpo central.
- o axônio (ou cilindro eixo): através do qual a célula nervosa transmite os impulsos nela originados. Em geral os axônios são muito longos (alcançando às vezes o tamanho de 1m de comprimento) e são únicos para cada célula. Nele os dendritos de outros neurônios se ligarão de forma a obter o impulso correspondente ao "resultado de saída" desta célula. Este está envolvido por uma bainha de mielina, externamente à qual ainda podem haver outras bainhas

Os neurônios não trabalham de forma autônoma, mas em estreita colaboração recíproca. É exatamente dessa cooperação (que, indiretamente, põe em comunicação o conjunto das células do sistema nervoso) que deriva toda a complexidade do sistema nervoso.

Em condições normais, como já afirmamos anteriormente, duas células nervosas se associam estabelecendo contato entre o dendrito de uma e o axônio de outra; esta modalidade de associação recíproca é chamada de sinapse.

Existem basicamente dois tipos de sinapse no mundo animal : as sinapses elétricas e as sinapses químicas. Sob o ponto de vista prático, quase todas as sinapses utilizadas para a transmissão de sinais do sistema nervoso central são sinapses químicas.

Nestas, o primeiro neurônio secreta, na sinapse, uma substância química denominada neurotransmissor, e este transmissor, por sua vez,



age sobre todas as proteínas receptoras na membrana do neurônio, excitando-o, inibindo-o ou modificando sua sensibilidade de algum modo.

As sinapses elétricas por sua vez caracterizam-se por canais diretos que conduzem a eletricidade de uma célula para uma próxima (uma espécie de curto circuito).

Todo neurônio tem um pequeno potencial elétrico de repouso na sua membrana, da ordem de -65mV. A ação dos neurônios anteriores pode inibir ou excitar um neurônio pós-sináptico respectivamente diminuindo (tornando mais negativo) ou aumentando (tornando menos negativo) o valor de seu potencial.

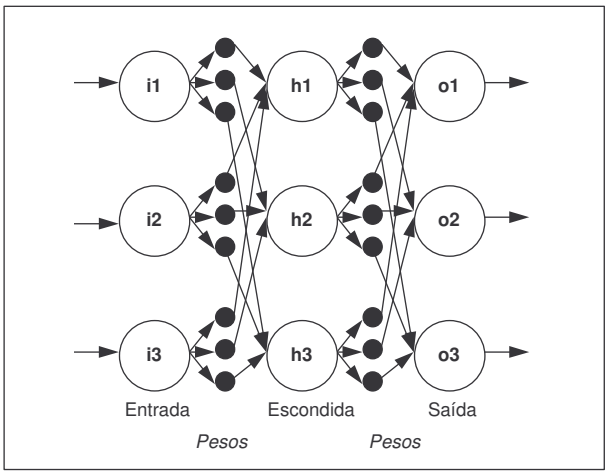
A ação dos neurônios pré-sinápticos se soma em um neurônio e altera seu potencial elétrico. Quando este atinge a marca de -45mV, o neurônio atinge o que se chama potencial de ação, como vemos na. Para atingir este estado é necessária a atuação de vários neurônio pré-sinápticos (cerca de 70 para o neurônio motor típico)

O modelo de neurônio artificial que usamos atualmente nas redes neurais é muito semelhante ao neurônio biológico que discutimos acima, conforme vemos na figura 8.3.

A operação deste neurônio é muito simples : as entradas são apresentadas ao neurônio e são multiplicadas cada um por um peso. O resultado desta operação é chamado net. A seguir é aplicada uma função não linear a net, resultando no resultado de saída do neurônio (também denominado out).

As redes neurais podem ser classificadas em redes de múltiplas camadas, ou multicamadas, caso contenham mais do que uma camada

de neurônios, como mostrado na fig. 8.4 a seguir. Exemplos de redes de múltiplas camadas são as redes que usaremos na seção 8.3b a seguir.



9.3.b) Usando GA em conjunto com redes neurais

Para resolver o problema do aprendizado de pesos de uma rede neural usando GAs, é necessário que transformemos a estrutura do problema em alguma forma compreensível para nosso GAs. Além disto, precisamos entender que alterar apenas uma sinapse tem uma influência em cascata pelo desempenho de toda a nossa rede neural. Logo, temos que desenvolver uma estrutura para nossos GAs tal que possamos restringir em blocos a influência da alteração de sinapses, tal que os operadores de crossover e de mutação possam operar e gerar resultados consistentes.

Uma estrutura bem simples para esta representação seria concentrar todos os pesos relativos a um neurônio (desde todas as camadas) em cada gen. Isto está representado na figura 8.1 a seguir.

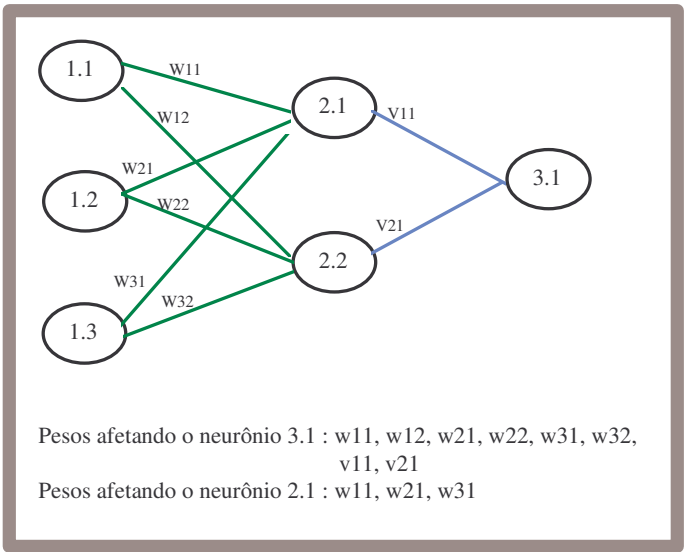


fig 8.1 ⇒ Desenho de uma rede neural e exemplificação da estrutura de um gen para treinamento genético dos pesos neurais

Podemos ver duas coisas interessantes dentro do nosso esquema de representação. Primeira, os gens têm tamanhos diferentes entre si, mas que são fixos no tempo. Logo, ao realizar o crossover, devemos respeitar as "bordas" dos gens. Segunda, há várias localidades que contêm um peso específico, como por exemplo, vemos na figura 8.1, tanto o gen para o neurônio 3.1 quanto o gen para o neurônio 2.1 contêm o peso w11. Logo deve-se copiar a estrutura de fenótipos encontrada na natureza e fazer com que o peso final seja uma função de todas as posições em que o peso está representado. A média aritmética é uma função extremamente adequada.

O operador de mutação para nosso treinamento pode ser a estrutura de treinamento em backpropagation, com pesos iniciais iguais àqueles representados no nosso indivíduo, ou então a alteração aleatória de bits em posições quaisquer do inidivíduo em questão.

A função de avaliação do nosso algoritmo híbrido é exatamente a qualidade do mapeamento executado pela rede. Podemos usar erro médio quadrático, por exemplo, como medida desta qualidade.

O critério de parada do algoritmo deve ser a performance da rede neural. Assim que o erro cometido por esta for suficientemente pequeno, pararemos de executar o nosso algoritmo genético.

10) Dicas gerais sobre GA

10.1) Parâmetros

10.1.a) Tamanho da População

A performance do algoritmo genético é extremamente sensível ao tamanho da população, logo este parâmetro deve ser definido com muito cuidado.

Caso este número seja pequeno demais, não haverá espaço para termos uma grande variedade genética dentro da nossa população, o que fará com que nosso algoritmo seja incapaz de achar boas soluções. É difícil dizer com precisão qual é o limite mínimo para o qual o GA ainda terá uma boa performance, pois este limite varia com o número de genes presentes em cada indivíduo, mas, via de regra, pode-se definir que este mínimo fica em torno de 40 indivíduos.

A capacidade de armazenamento de estados de um GA é exponencialmente proporcional ao número de indivíduos presentes. Consequentemente há um limite superior para o tamanho da população onde ainda verifica-se melhora na performance conforme aumenta-se o tamanho da população. É claro que quanto maior a população, mais tempo o GA demorará para processar cada geração e mais demoraremos para conseguir uma resposta. Consequentemente, não devemos aumentar o tamanho da população indiscriminadamente, há um limite superior para a nossa escolha, limite este que, assim como o limite inferior, é dependente do problema e da codificação adotada, mas normalmente podemos adotar o limite de 100 indivíduos como nosso limite máximo.

A conclusão é : se você não quiser ficar testando horas parâmetros do seu algoritmo, adote uma população com um número de indivíduos entre 50 e 70 para obter uma boa performance.

b) Operador de mutação

O operador de mutação é fundamental para um GA, pois é ele que garante a continuidade da existência de diversidade genética na população, apesar do operador de crossover contribuir fortemente para a igualdade entre os indivíduos. Logo, se o valor do operador de mutação for baixo demais então a população não terá diversidade depois de um certo número de gerações, estagnando bem rápido.

Mas, em contrapartida, se o operador de mutação receber um valor de probabilidade alto demais, então o nosso GA passará a ter um comportamento mais parecido com um algoritmo aleatório (random walk) e perderá suas características interessantes.

Consequentemente, precisamos determinar um valor ótimo para nosso operador de mutação. Obviamente, cada tipo de população tem o seu, mas a experiência difundida mostrou que um bom valor para atribuir-lhe é tal que sejam mudados cerca de 0.4% dos gens a cada rodada.

Isto significa que se estamos usando representação binária e o operador de mutação for do tipo que, se ativo, inverte o bit, então o valor a atribuir-lhe é de 0.004. Se o operador de mutação for tal que escolhe um novo bit aleatório para a posição em questão, então ele deve receber o valor de 0.008.

Outro ponto interessante é que para evitar estagnação podemos usar o operador de mutação dirigida. Este trabalha somente depois da população começar a estagnar e sua função é exatamente criar diversidade genética. Su valor ótimo ainda é um mistério, mas bons resultados com representação binária já foram conseguidos. O único problema deste operador é o overhead de processamento que ele gera, o que pode fazer com que não compense utilizá-lo, em face da sua limitação em melhorar os resultados. Sua utilização realmente só é recomendada quando excelentes resultados são necessários, e apenas boas soluções não servem.

10.2) Módulo de seleção

10.2.a) Escolha dos pais

A escolha dos pais que vão gerar a próxima geração de indivíduos é outro problema sensível dos GAs. Caso sejamos muito restritivos e só usemos pais com excelentes avaliações poderemos estar jogando fora bons esquemas presentes nos indivíduos "ruins", mas se permitimos muito que os indivíduos com avaliações ruins participem do processo reprodutivo, os esquemas que os tornam ruins não desaparecerão da população.

Um exemplo claro deste caso é o seguinte : imagine que estamos procurando a solução ótima para a nossa f6. Como falamos anteriormente, um valor alto para x ou para y torna o indivíduo uma péssima solução. Imagine agora que um indivíduo tem um valor muito alto para x e zero (que é o valor ótimo) para y. Sua avaliação é ruim, somente devido à presença do valor muito alto em x. Se os melhores indivíduos de nossa população forem todos soluções que possuem valores médios para x e y e nós formos muito restritivos, o valor ótimo de y nunca vai "cruzar" com outros. Mas se permitirmos que o péssimo indivíduo reproduza demais, o valor alto de x vai permear-se entre as novas gerações piorando-as sensivelmente.

11) Programação genética

Há muitos anos os cientistas estão animados com a idéia de computadores se auto-programando. Isto está presente em vários filmes de ficção científica, desde Guerra nas Estrelas até os mais recentes filmes da Matrix.

Um ramo dos algoritmos evolucionários chamado programação genética (GP) consiste em tentar evoluir programas de forma a resolver um problema em questão.

A programação genética funciona da mesma maneira que os algoritmos genéticos tradicionais: temos que encontrar uma forma de codificar nossos programas e a partir daí aplicar operadores de crossover e mutação de forma a evoluí-lo e encontrar a melhor solução para o problema em questão.

É claro que precisamos embutir dentro de nosso algoritmo conhecimento sobre a estrutura da linguagem de programação na qual os programas serão gerados, pois os programas que geramos só serão válidos se compilarem e executarem!

As áreas em que já há programas desenvolvidos usando GP que resolvem de forma satisfatória os problemas em questão incluem:

- Controle de processos (pêndulo invertido)
- Controle de robôs para problemas de empilhamento
- Estratégias de jogos (pôquer e tic-tac-toe)
- Classificação
- Computação Simbólica
  - Indução de sequências
  - Resolução de equações
  - Diferenciação e integração simbólica
  - Descoberta automática de identidades trigonométicas

Agora que você está suficientemente interessado, vamos ver um pouco da teoria!

12) GA paralelos

12.1) Introdução

Cada dia que passa a tecnologia evolui e consequentemente fica mais fácil e mais barato utilizar paralelismo em hardware e consequentemente a pesquisa sobre GAs paralelos cresceu muito com isto.

GAs são excelente candidatos para paralelização, visto que eles têm como princípio básico evoluir em paralelo uma grande população de indivíduos. Mas, apesar do seu paralelismo intrínseco, os GAs tradicionais não são paralelizáveis, devido à sua inerente estrutura sequencial e devido à necessidade de se utilizar um controle global sobre a população.

Consequentemente vários novos modelos para paralelização de algoritmos genéticos surgiram para que pudéssemos aproveitar este novo e relativamente barato hardware que está disponível no mercado. Estes novos algoritmos têm diferenças fundamentais sobre o modelo tradicional que permitem a sua paralelização. Veremos a seguir mais detalhes sobre cada um deles.

12.2) Panmitic

Este tipo de implementação de PGAs (GAs paralelos) constituem a classe mais simples de implementaçao, sendo mais indicados para máquinas com memória compartilhada.

Basicamente, este tipo de GA consiste em vários GAs simples executando cada um em um processador disitnto e operando sobre uma única população global.

Os processadores sincronizam no operador de seleção, isto é, eles vão selecionando indivíduos e quando o número desejado de indivíduos já tiver sido selecionado, cada processador "cospe" os filhos que gerou e que vão compor a nova população.

Este tipo de algoritmo não apresenta ganho de performance do algoritmo: a sua única vantagem sobre GAs não paralelos é que a versão paralela é muito mais rápida (mas também muito mais complexa, devido à necessidade de se gerenciar o paralelismo).

12.3) Island

Os biológos perceberam claramento que ambientes isolados, como por exemplo ilhas, frequentemente geram espécies animais melhor adaptadas às peculiaridades dos seus ambientes do que áreas de maior extensão.

Isto fez nascer a teoria dos nichos ecológicos e inspirou a comunidade científica que trabalha com GAs a criar novos modelos e arquiteturas. Em particular, isto levou à teoria que várias pequenas populações competindo poderiam ser mais efetivas na busca de boas soluções do que uma grande população do tamanho da soma dos tamanhos das pequenas populações.

Propôs-se então uma modelo distribuído para GAs chamado modelo Island, no qual a população de cromossomos é particionada em um certo número de subpopulações isoladas, cada uma das quais evolui de forma independente, tentando maximizar a mesma função. Uma estrutura de vizinhança é definida sobre este conjunto de subpopulações de forma que periodicamente cada subpopulação troca um número n de seus piores indivíduos pelos n melhores indivíduos de cada vizinho. Esta atividade de troca de indivíduos é chamada de migração, sendo ilustrada pela figura 10.1 mostrada a seguir.

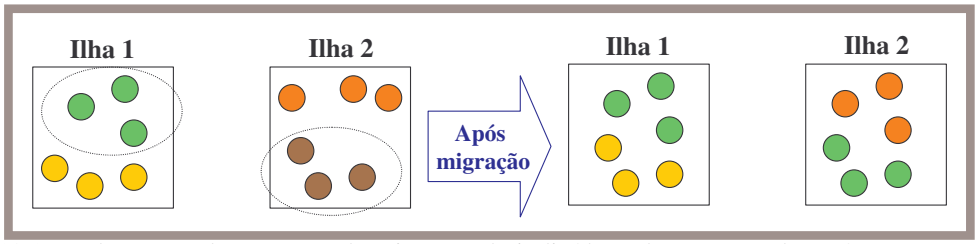


Fig. 10.1 → Ilustração do processo de migração de indivíduos de uma população à outra. As n melhores soluções da ilha 1 (em verde) migraram para a ilha 2, substituindo os n piores elementos desta ilha (em marrom). Obviamente, em uma situação real, os melhores da ilha 2

também migrariam para a ilha 1 e a percentagem de indivíduos não seria tão alta como a mostrada na figura.

O modelo pode então ser resumido pelo seguinte algoritmo :

Passo 1	• *Defina um modelo adequado para resolver o problema;* • *Gere aleatoriamente uma população de indivíduos e divida-os entre as várias subpopulações.* • *Defina uma estrutura de vizinhança entre as subpopulações*
Passo 2	• *Execute um GA comum em cada uma das subpopulações durante um certo número de gerações*
Passo 3	• *Mande os melhores n indivíduos para os vizinhos* • *Receba os melhores n de cada um deles* • *Substitua n indivíduos da sua população pelos n recebidos de acordo com alguma estratégia pré-definida*
Passo 4	• *Se o GA não tiver acabado (de acordo com padrões pré-definidos, volte para o passo 2.*

A estratégia de substituição dos n indivíduos da população pelos n recebidos fica a critério do desenvolvedor. Exemplos possíveis são a substituição dos n piores ou então a substituição dos n mais parecidos com os recebidos.

A maneira de definir se o GA terminou ou não é exatamente idêntica à maneira utilizada nos GAs sequenciais.

Agora temos que ajustar mais alguns parâmetros, que são as vizinhanças entre as várias subpopulações e o número de indivíduos que migram a cada período.

Este tipo de implementação de PGAs é utilizado em sistemas distribuídos que possuem memória local privada. Visto que não há uma

esquema central de seleção o sistema é completamente assíncrono, e os processadores sincronizam durante o período de migração.

12.4) Massively parallel

Este modelo, também chamado de modelo de vizinhança (neighbouhood) difere do modelo anterior por evoluir somente uma única população. Cada indivíduo pertencente a esta população é colocada em uma cela de uma grade (ou uma hipergrade de n dimensões, se quisermos generalizar) e os operadores genéticos só são aplicados sobre indivíduos que sejam vizinhos nesta grade (vizinhança esta que é pré-definida de acordo com a estrutura da nossa hipergrade).

Hardware que possui uma estrutura massivamente paralela é eficiente na execução de atividades simples e locais, logo, um GA voltado para este tipo de hardware deve consistir de uma série de atividades extremamente simples que possam ser executadas simultaneamente, sem congestionar os canais de memória.

O modelo que estamos trabalhando contém uma noção de espaço, pois os indivíduos na população estão designados para posições específicas da nossa grade (usalmente implementada em duas dimensões, como se fosse a superfície de um toróide, para evitar efeitos de bordas). Pode-se, é claro, usar-se noções de grades cúbicas ou mesmo hipergrades (de *n* dimensões, $n \geq 4$), mas não foi comprovado todavia aumento de performance quando se utilizam estes tipos de conformação espacial.

O importante é que, assim como no modelo island, deve se estabelecer noções de vizinhança entre dois indivíduos de nossa população. Cada indivíduo deve ter um mínimo de vizinhos (pelo menos 3), pois senão a diversidade genética pode não florescer no meio de nossa população.

O algoritmo de execução do modelo massively parallel pode ser resumido da seguinte forma :

Passo 1	• *Defina um modelo adequado para resolver o problema;* • *Gere aleatoriamente uma população de indivíduos, compute sua função de avaliação e divida-os entre as células.* • *Defina uma estrutura de vizinhança entre as subpopulações*
Passo 2	• *Escolha um indivíduo na vizinhança de cada célula, reproduza o ocupante com o escolhido e coloque um dos filhos na célula em questão*
Passo 3	• *Faça mutação em cada célula com probabilidade p_m.* • *Compute a função de avaliação de cada um dos novos indivíduos*
Passo 4	• *Se o GA não tiver acabado (de acordo com padrões pré-definidos, volte para o passo 2.*

A maneira de escolher o pai que irá reproduzir com o ocupante da célula em questão fica inteiramente a critério do desenvolvedor. Pode-se, por exemplo, utilizar o método da roleta com todos os indivíduos localizados na vizinhança de cada célula.

A maneira de definir se o GA terminou ou não é exatamente idêntica à maneira utilizada nos GAs sequenciais.

13) Aplicações

13.1) Introdução

A aplicabilidade de GAs é praticamente infinita – sempre que houver uma necessidade de otimização (incluindo soluções de equações), um AG pode ser considerado. Nosso problema consiste somente em tornar os problemas que nos afligem em problemas tratáveis por algoritmos genéticos.

Consequentemente, quando queremos entender como um problema é resolvido, temos que entender como as seguintes questões foram resolvidas:

- Qual a representação adotada?
- Para esta representação, qual é o mecanismo dos operadores genéticos?
- Qual foi a função de avaliação utilizada?

Respondidas estas três questões, o resto é muito simples, seguindo o mecanismo tradicional de "seleção natural" adotados pelos GAs nos problemas tradicionais já conhecidos por todos nós.

13.2) Escalonamento de horários

Este problema é um problema extremamente interessante e complexo. Cada semestre todos os departamentos de uma faculdade fornecem ao departamento acadêmico central a relação de turmas e horários requeridos, e pede que lhes seja designada uma sala. Mas o número de salas é limitado, e normalmente é impossível colocar todas as turmas nos horários propostos pelos departamentos. Como proceder então?

É óbvio que o espaço de busca é praticamente infinito, dado que o número de combinações turma/sala é proporcional à fatorial do número de turmas e salas.

Além das salas serem em menor número que o necessário, algumas delas não servem para atender algums turmas, por serem pequenas demais. Logo, também deve ser levado em consideração que as menores turmas devem receber as menores salas.

A maneira encontrada para resolver isto via algoritmos genéticos é extremamente engenhosa, e baseia-se numa pequena variação do GA baseado em ordem.

Os cromossomos consistem das turmas ordenadas e os operadores de mutação e crossover são exatamente aqueles explicados no capítulo 7 deste mesmo texto (GA baseado em ordem).

A função de avaliação é a seguinte : para cada indivíduo na lista é designada a menor sala que possa atendê-lo dentre a lista de salas livres para aquele horário. Depois de todas as designações terem sido efetuadas, calculamos a função de avaliação que consiste em um número inversamente proporcional ao número de alunos sem sala de aula e ao número de salas que não receberam nenhuma turma.

Obviamente o problema acabou ficando com cara de problema de maximização, que é o ponto forte dos nossos GAs, logo podemos aplicá-los sem sustos e obteremos resultados.

13.3) Alocação de capacitores

Capacitores são frequentemente usados em sistemas de distribuição de energia para compensação, para conseguir regular a voltagem fornecida e para aumentar a capacidade de liberação de energia do sistema. A extensão dos benefícios conseguidos depende fortemente de como os capacitores são colocados e controlados. Logo, o problema de alocação de capacitores consiste em escolher localização, tipo, tamanho e esquema de controle dos capacitores em um sistema de distribuição de energia, de forma que a relação benefícios/custo seja maximizada e que restrições de corrente, voltagem, etc. em cada nó e ramo sejam satisfeitas.

No nosso caso, estudaremos a seguir somente o problema de alocação de capacitores em localizações fixas, o que faz então que as únicas incógnitas deste problema sejam as magnitudes dos capacitores.

Para representar os nossos cromossomos usa-se então codificação binária, sendo que um certo número fixo de bits representa a magnitude de cada capacitor, fazendo com que possamos entender cada string de bits como mostra a figura 11.1 a seguir. A magnitude de cada capacitor está representada como um número inteiro que representará o múltiplo do tamanho mínimo de capacitor utilizado.

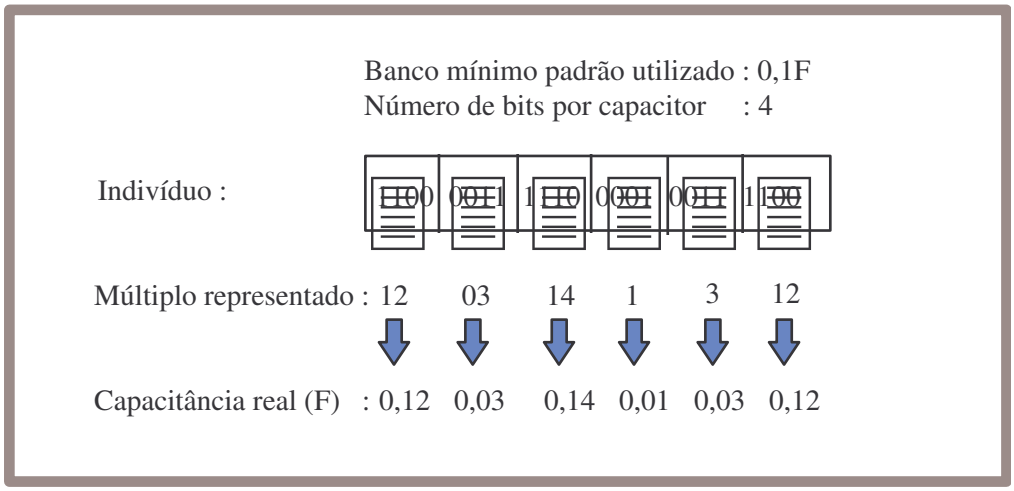


fig. 11.1 ⇒ Exemplo da representação utilizada no problema de alocação de capacitores

Dado que estamos usando a representação binária comum, os operadores genéticos são exatamente iguais àqueles usados anteriormente e que nos são bem conhecidos.

A função de avaliação utilizada é a seguinte :

$$F = \sum_{k} C_k u_k^0 + K_1 * \sum_{i,k} T_i P_{perda,i}(x^i, C_k) + K_2 * F(x^i, C_k) + K_3 * H(x^i, C_k)$$

,onde :

$C_k u_k^0$ = Custo da colocação do capacitor k

$F(x^i, C_k)$ = Restrinções de fluxo de carga

$H(x^i, C_k)$ = Restrições operacionais

$T_i P_{perda,i}(x^i, C_k)$ = Perda gerada pela alocação do capacitor k na linha i

K_1, K_2 e K_3 = constantes

Nosso problema é claramente um problema de minimização desta função. Se, ao contrário quisermos maximizá-la, basta calcular a avaliação como sendo F'= 1 / (1 + F), e então aplicamos nosso algoritmo genético tradicional.

14) Conclusões

Algoritmos genéticos são uma interessante ferramenta para efetuar buscas em espaços infinitos, mas sua aplicabilidade é limitada somente a este tipo de problema. Não é possível dizer que esta é uma técnica que revolucionará toda a ciência da computação, pois isso não é verdade.

Mas como ferramenta de busca GA se mostram extremamente eficientes, encontrando soluções quase ótimas quase sempre (lembremo-nos sempre que GAs são heurísticas, e não algoritmos), além de serem extremamente simples de implementar e modificar. O custo de pessoal para implementação de um GA é quase mínimo, pois GA é uma técnica de só necessita de tempo de processamento para obter resultados, sendo os programas quase rudimentares em sua essência.

O ponto mais importante a se enfatizar nesta conclusão é que não há motivo nenhum para pararmos de aprender com a natureza. Os GAs atuais funcionam (muito bem), mas provavelmente a inclusão de técnicas já usadas na natureza podem melhorar bastante a performance destes algoritmos.

15) Referências bibliográficas

- Bodenhofer, Ulrich – “Genetic Algorithms – Theory and applications”, Fuzzy Logic Laboratory at the Linz-Hagenberg University, Alemanha, 2003

- Boone, Gary and Chiang, Hsiao-Dong - "Optimal capactior placement in distribuition systems by genetic algorithm" in "Electrical Power and energy systems", vol 15, no. 3, 1993.

- Davis, Lawrence - "Handbook of genectic algorithms", Van Reinhold Nostrand, USA, 1991

- Darwin, Charles - "A origem das espécies"

- Dimeo, Robert M. & Lee, Kwang Y. - "Genetics based control of a mimo boiler-turbine plant"

- Dorigo, Marco & Maniezzo, Vittorio - "Parallel Genetic Algoriths : Introduction and overview of current research"

- Fernandes, Anita Maria da Rocha - “Inteligência Artificial – noções gerais”, Editora Visual Books, Florianópolis, Brasil, 2003

- Goldberg, David E. - "Genetic algorithms in search, optimization and machine learning" - Addison Wesley Publishing Co. , USA, 1989

- Heitkotter, Jorg - "The Hitch-Hiker's guide to evolutionary computation" - FAQ in comp.ai.genetic

- Karr, Chuck - "Applying genetics to fuzzy logic" - AI Expert, march, 1991

- Linden, Ricardo - "Operador de mutação dirigida" - Simpósio Brasileiro de Inteligência Artificial (SBIA'96) - Curitiba, Outubro/96

- Palagi, Patrícia M. & Carvalho, Luís Alfredo V. de - "Neural networks learning with genetic algorithms"

- Velloso, Maria Luiza F.; Pacheco, Marco Aurelio C. & Vellasco, Marley - "Alocação de salas de aulas utilizando algoritmos genéticos"

Apêndice A – Teorema dos esquemas