Algoritmos Genéticos *em Java*

Existem problemas relativamente do cotidiano aparentemente simples, mas que são tão complexos que não admitem solução em um tempo viável através de técnicas tradicionais. Para resolvê-los, podemos usar os Algoritmos Genéticos, que se inspiram na evolução das espécies para encontrar soluções para estes problemas.

A Inteligência Artificial é uma área da computação que busca desenvolver sistemas que tenham comportamento similar a certos aspectos do comportamento inteligente, especialmente a capacidade de resolver problemas extremamente difíceis ainda que de maneira aproximada. Pensando assim, podemos tentar nos inspirar na biologia e na genética para criar um algoritmo que nos permita criar programas de computadores mais "espertos" do que os programas tradicionais, sendo capazes de encontrar soluções que estes últimos não encontrariam.

A palavra "encontrar" usada aqui é apropriada, porque o objetivo final de todas as técnicas da inteligência computacional é a busca, podendo ser de uma solução numérica, do significado de uma expressão linguística, de uma previsão de carga ou de qualquer outro elemento que tenha significado em uma determinada circunstância. Não importando a área de aplicação, uma busca sempre tem como objetivo encontrar a melhor solução dentre todas as soluções possíveis (o espaço de soluções).

A verdade é que, sem exageros, a busca é o problema básico da computação. Todo problema pode ser descrito como a tentativa de alcançar um determinado objetivo, isto é, de chegar a um determinado estado onde uma certa condição é satisfeita.

Um estado nada mais é do que uma especificação de certos aspectos da realidade relevantes para o problema. Por exemplo, se meu objetivo é ganhar um milhão de reais, então o estado consiste em quanto dinheiro eu ganhei até hoje. O meu peso neste momento não é relevante para o estado do sistema. Desta maneira, pode-se dizer que um estado, na realidade, corresponde a um conjunto de valores dentre todas as variáveis que são do meu interesse. Por exemplo, um possível estado para o problema de ficar rico consiste em se obter um ganho de R$10 mil. Neste problema, este estado não leva em consideração o meu peso naquele instante e qualquer mudança de peso não afeta o estado corrente, assim como qualquer variável adicional como meu estado civil, e outras que possam ser imaginadas.

Ricardo Linden | rlinden@pobox.com

é formado em engenharia de computação pela PUC-RJ e tem doutorado em engenharia elétrica pela COPPE-UFRJ. Trabalha com Java há cerca de 8 anos, desenvolvendo aplicativos para o setor elétrico no CEPEL. Atualmente é professor da disciplina de sistemas digitais na Faculdade Salesiana Maria Auxiliadora (FSMA), de Macaé-RJ e é autor do livro "Algoritmos Genéticos", atualmente em sua 3ª edição.

Existem vários problemas em nosso dia a dia que são extremamente complexos e não admitem solução exata em um tempo razoável. Entre eles encontramos um exemplo do cotidiano que é a otimização da rota de caminhões que vão distribuir produtos entre vários clientes. Para resolver estes problemas de otimização nós usamos heurísticas, que encontram soluções boas o suficiente para usarmos na prática. Uma heurística muito importante para este tipo de problema são os algoritmos genéticos, sobre os quais falaremos neste artigo.

As diferentes ações causam modificações no estado do sistema. Por exemplo, se o meu livro sobre Algoritmos Genético vender 100 mil cópias, eu chegarei mais próximo do objetivo de ganhar um milhão de reais. As várias ações que posso realizar em cada instante devem então ser avaliadas como um caminho da busca a ser efetuada, com o intuito de encontrar o estado final desejado. Assim, podemos dizer que resolver um problema consiste em buscar um estado em que uma determinada condição seja satisfeita, ou seja, todo problema, no fundo, é um problema de busca. Um algoritmo de busca é tal que recebe um problema como entrada e retorna uma solução sob a forma de uma sequência de ações recomendadas para se atingir o objetivo desejado (Russel & Norvig, 2004).

Algoritmos genéticos (GA, genetic algorithms) são uma técnica de busca extremamente eficiente no seu objetivo de varrer o espaço de soluções e encontrar soluções próximas da solução ótima, quase sem necessitar interferência humana, sendo uma das várias técnicas da inteligência computacional mais instigantes (ao menos na minha opinião!). O problema dos GA é que eles não são tão bons assim em termos de tempo de processamento. Logo, eles são mais adequados em problemas especialmente difíceis, entre os quais incluímos aqueles denominados intratáveis (sobre os quais falaremos mais no box intitulado "Os caixeiros viajantes e os problemas intratáveis").

Agora que estamos suficientemente motivados, podemos começar a ver o que efetivamente são e o que fazem os algoritmos genéticos.

O que são os algoritmos genéticos?

Primeiramente, vamos conhecer os algoritmos evolucionários (EA, Evolutionary Algorithm), que são ferramentas que usam modelos computacionais dos processos naturais de evolução como uma ferramenta para resolver problemas (para entender um pouco mais sobre como funciona a evolução natural, leia o box intitulado "Darwin, Mendel, a teoria da evolução e a genética"). 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 do "desempenho" dos indivíduos desta espécie dentro do "ambiente".

Os algoritmos evolucionários funcionam mantendo uma população de estruturas, denominadas indivíduos ou cromossomos operando sobre estas 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. Com base 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 resumido, sem maiores detalhes,

Os caixeiros viajantes, as mochilas e os problemas intratáveis

Existe uma forma de avaliar o tempo de execução de um algoritmo que não busca avaliar o tempo de forma exata, mas sim verificar como o tempo de execução cresce conforme aumentamos o volume de dados oferecido como entrada para um algoritmo. Isto nos permite fazer uma medida que é válida (mesmo que não seja precisa) não importando a CPU, o sistema operacional, o compilador e outros fatores inerentes à máquina específica na qual o programa será executado.

Com esta métrica pode-se comparar dois algoritmos que porventura resolvam o mesmo problema, usando esta aproximação do tempo de execução para escolher aquele que é mais eficiente. Para fazer esta comparação, não precisamos do cálculo de tempo exato de cada algoritmo, mas sim de uma métrica comparativa, que seja capaz de dizer qual algoritmo vai necessitar menos tempo e/ou recursos computacionais. Ou seja, nosso objetivo fundamental é encontrar uma maneira de comparar diferentes algoritmos de forma a conseguir uma métrica independente das circunstâncias (sistema operacional, máquina etc.) que nos permita dizer: "para o problema X, o melhor algoritmo é Y".

Para fazer este cálculo do tempo de execução de nossos programas, começamos com 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 atribuímos a ele valor 1. Depois, 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 tomam cerca de 90% do tempo de execução). Consequentemente, quando analisar o seu tempo de execução, preste muita atenção nestas partes, pois elas dominam a execução total. 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 é executado e consequentemente o tempo de execução do mesmo. Fazendo este tipo de cálculo, obtemos um tempo de execução do algoritmo que é uma função do número de entradas que ele recebe (n), isto é, obtemos um tempo T(n). Este tempo consiste em uma medida aproximada da taxa de crescimento que é intrínseca ao algoritmo – o que varia de ambiente para ambiente é apenas o tempo absoluto de cada execução, o qual é dependente de fatores como poder de processamento, código compilado e outros difíceis de medir. Esta função permite analisar a complexidade de um algoritmo, fornecendo uma boa estimativa sobre qual algoritmo será o mais rápido se o tamanho das entradas crescer muito.

Com base nesta informação, podemos então definir os problemas intratáveis, os quais são definidos como aqueles cujo tempo necessário para resolvê-lo é considerado inaceitável para os requerimentos do usuário da solução. Em termos práticos, um problema é tratável se o seu limite superior de complexidade é polinomial, e é intratável se o limite superior de sua complexidade for exponencial (Toscani, 2009), ou seja, se seu tempo de execução é da ordem de uma função exponencial (como 2n) ou fatorial (n!).

Um exemplo de um problema intratável muito comum é o do caixeiro viajante que tem que visitar n estradas e precisa estabelecer um trajeto que demore o menor tempo possível, para ganhar o máximo de dinheiro no mínimo de tempo. Todas as cidades são ligadas por estradas e pode-se começar por qualquer uma delas. 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 ou da busca exaustiva – 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!. O fatorial de um número cresce muito rapidamente. Por exemplo, se tivermos 100 cidades teremos 10158 opções. Se pudermos testar um bilhão de soluções por segundo, demoraríamos um tempo igual a 10140 anos para encontrar a melhor solução. Novamente, levaremos mais tempo do que a idade do universo para encontrar uma solução.

Este problema pode parecer anacrônico e irreal, afi-

nal praticamente não existem mais caixeiros viajantes, mas ele é análogo aos problemas de distribuição enfrentados por qualquer indústria. Trocando "caixeiro viajante" por "caminhão de mercadoria" e "cidades" por "bares", temos o dificílimo problema de realizar a distribuição de mercadorias por todos os clientes da forma mais eficiente possível, minimizando o tempo e o custo associados. Outra aplicação prática consiste em trocar "caixeiro viajante" por "pacote de rede" e "cidades" por "roteadores" ou "nós de distribuição". Assim, a questão do anacronismo é, para todos os efeitos, desprezível.

Outro problema extremamente interessante e importante é o problema da mochila, que é um problema de otimização combinatória que consiste na necessidade de encher uma mochila com objetos de diferentes pesos e valores, com o maior valor possível, sem ultrapassar o peso máximo que ela suporta. Mais uma vez, não se engane pela simplicidade do enunciado: este problema é similar a problemas práticos como alocação de recursos de produção, decisão sobre investimentos e até mesmo na alocação de registradores em código compilado.

A dificuldade deste problema também é grande. Para resolvê-lo por força bruta, teríamos que checar todas as combinações possíveis de objetos, primeiro cada objeto isoladamente, depois as combinações de dois objetos, depois as combinações de três, e assim por diante. Como você lembra do ensino médio, o número de combinações é proporcional à fatorial do número de elementos e assim, mais uma vez temos um problema cuja solução pelo método da força bruta pode demorar um tempo longo demais para ser realizável.

Assim, podemos ver claramente que estes problemas são intratáveis e, tendo em vista sua importância, precisamos de uma técnica capaz de resolvê-lo. Como não podemos resolvê-lo de forma exata, precisamos de uma heurística, isto é, um programa que forneça uma solução suficientemente boa, ainda que não ótima para o problema (como, por exemplo, uma solução na qual nossos caminhões entreguem todas as mercadorias a tempo para nossos clientes, mesmo em um tempo superior ao ótimo). É aí que recorremos aos algoritmos genéticos!

pela figura 1, que descreve de forma esquemática o funcionamento de um EA, usando caixinhas que representam cada uma das funções do mesmo. Quando formos implementar o nosso algoritmo genético, cada uma destas caixinhas virará um código que, espero eu, esclareça de forma completa a sua função.

Antes de partir para a implementação, vamos falar um pouquinho mais sobre os elementos teóricos que compõem os algoritmos evolucionários.

Na figura 1 podemos perceber o funcionamento básico dos algoritmos evolucionários que consiste em buscar dentro da atual população aquelas soluções com as melhores características (caixinha de seleção na figura 1) e tentar combiná-las de forma a gerar soluções ainda melhores (caixinha que contém a aplicação dos operadores genéticos na figura 1). Este processo é repetido até que tenha se passado tempo suficiente ou que tenhamos obtido uma solução satisfatória para nosso problema.

Cada uma das repetições do laço completo (controlado pelo losango que representa uma condição em um fluxograma) é denominada de uma geração do algoritmo. O conceito é similar ao conceito de geração existente na vida real, com a exceção de que muitas vezes nos algoritmos evolucionários duas gerações não convivem, como a figura deixa implícito (veja que jogamos toda a geração dos pais no lixo).

Algoritmos genéticos (GA) são um ramo dos algoritmos evolucionários e como tal também são uma técnica de busca baseada numa metáfora do processo biológico de evolução natural na qual populações de indivíduos são criados e submetidos aos operadores genéticos: seleção, recombinação (crossover) e mutação. Assim como descrevemos para os EAs, estes operadores utilizam uma caracterização da qualidade de cada indivíduo como solução do problema em questão chamada de avaliação. Sua atuação vai gerar um processo de evolução natural destes indivíduos, que eventualmente deverá gerar um indivíduo que caracterizará uma boa solução (talvez até a melhor possível) para o nosso problema.

Nós vamos ver mais a frente a implementação da função de avaliação para um problema interessante. Entretanto, para entender melhor o conceito de função de avaliação, vamos aplicá-la ao caso do caixeiro viajante. Para tanto, imagine que temos quatro cidades para visitar e os custos de trafegar de uma cidade para a outra é dada pela seguinte tabela, onde a cidade de origem está na vertical e a cidade de destino na horizontal:

	1	**2**	**3**	**4**
1	0	500	150	600
2	500	0	100	90
3	150	100	0	140
4	600	90	140	0

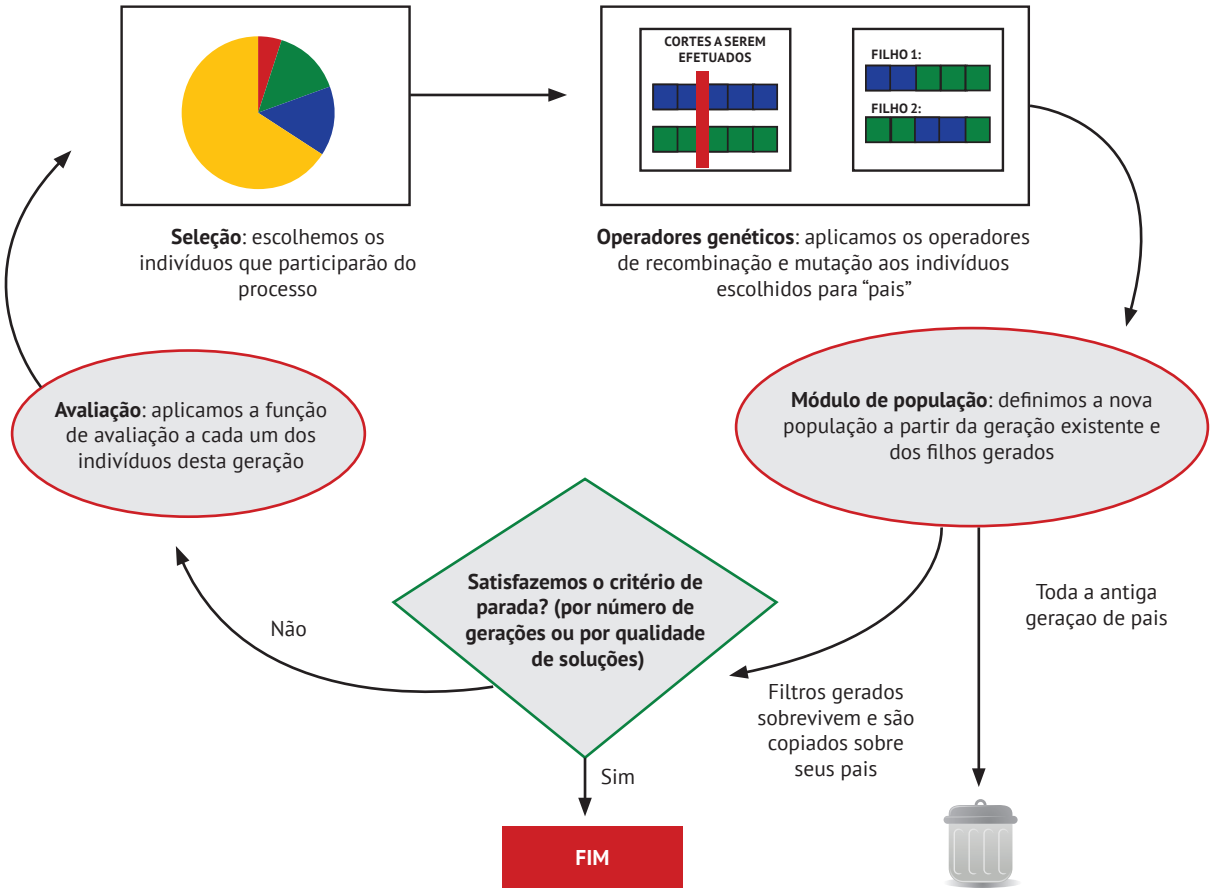

Figura 1. *Esquema de um algoritmo evolucionário (e, por conseguinte, de um algoritmo genético).*

Um custo pode ser alto por que a estrada é de má qualidade, tem pedágio alto, tem assaltos ou toma muito tempo por ter tráfego intenso de caminhões. Não colocamos unidade neste custo, pois ele pode ser uma combinação de vários fatores – quem determina os parâmetros de uma solução é o especialista no problema. Neste caso, o caixeiro viajante teria que nos informar a qualidade relativa de cada caminho de acordo com sua experiência.

Assim, vamos avaliar a solução que começa na cidade 1, vai para a 2, depois para a 3 e para a 4. O custo total é dado por 500 (custo do caminho 1→2) + 100 (custo do caminho 2→3) + 140 (custo do caminho 3→4) = 740 unidades. Já a solução que faz o caminho 1→3→2→4 tem custo igual a 150 (custo do caminho 1→3) + 100 (custo do caminho 3→2) + 90 (custo do caminho 2→4) = 340 unidades. Como estamos buscando minimizar nosso custo, então a segunda solução é claramente superior à primeira.

Lembre-se de suas aulas de matemática que para minimizar uma função f(x), é só maximizar a função 1/f(x). Assim, o problema do caixeiro viajante ainda recai na estrutura básica de funcionamento dos algoritmos genéticos.

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

A codificação da informação em cromossomos é um ponto crucial dentro do GA, e, junto com a função de avaliação, é 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. Ao fim da execução do nosso algoritmo a solução deve ser decodificada para ser utilizada na prática.

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 e recombinação gênica) se encarregará de fazer com que a população evolua. A mutação cria

Darwin, Mendel, a teoria da evolução e a genética

Até o século XIX os cientistas mais proeminentes acreditavam ou na teoria 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 outros ecossistemas, 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 Origem das espécies”.

A teoria da evolução diz que na natureza todos os indivíduos dentro de um ecossistema competem entre si por recursos limitados, tais como comida e água. Aqueles indivíduos (animais, vegetais, insetos etc.) que não obtêm êxito tendem a ter uma prole menor e esta descendência reduzida diminui a probabilidade de ter seus genes propagados ao longo de sucessivas gerações, processo este que é denominado de seleção natural.

A combinação entre as características dos indivíduos que sobrevivem pode produzir um novo indivíduo muito mais bem adaptado às características de seu meio ambiente ao mesclar características positivas de cada um dos reprodutores. Este processo implica nos descendentes de indivíduos serem variações dos seus pais. Assim, um descendente é ligeiramente diferente de seu pai, podendo esta diferença ser positiva ou negativa.

Entretanto, esta evolução natural não é um processo dirigido, com o intuito de maximizar alguma característica das espécies. Na verdade, não existe nenhuma garantia de que os descendentes de pais muito bem adaptados também o sejam. Existem muitos casos de filhos de pais fortes e saudáveis que possuem doenças terríveis ou fraquezas inerentes.

Podemos afirmar, então, que a evolução é um processo no qual os seres vivos são alterados por um conjunto de modificações que eles sofrem através dos tempos, podendo ser explicada por alguns fatores como mutação e recombinação gênica, seleção natural e isolamentos. O problema é que, na época de Darwin, ainda faltava um pedaço desta informação, que era a genética.

No início do século XX, um padre chamado Gregor Mendel compreendeu que este processo de transmissão de características positivas estava associado a uma unidade básica de informação, o gene. Na década de 1850 ele fez uma série de experimentos com ervilhas e percebeu que algumas características eram transmitidas de uma planta para seus descendentes. Através do estudo de linhagens de plantas, ele desenvolveu os princípios básicos da genética, que derivavam da existência de áreas codificadoras das características dos seres vivos, que são conhecidas com os genes.

Hoje nós sabemos que todo indivíduo, seja ele animal, vegetal ou mesmo organismos inferiores como vírus e bactérias, é formado por uma ou mais células, e dentro de cada uma delas o organismo possui uma cópia completa do conjunto de um ou mais cromossomos que descrevem o organismo, conjunto este denominado genoma. Um cromossomo consiste de genes, que são blocos de sequências de DNA. Cada gene é uma região do DNA que controla (sozinho ou em conjunto com outros genes) uma ou mais características hereditárias específicas, como cor do cabelo, altura, cor dos olhos e outras que nos tornam os indivíduos que somos.

Um conjunto específico de genes no genoma é chamado de genótipo. O genótipo é a base do fenótipo, que é a expressão das características físicas e mentais codificadas pelos genes e modificadas pelo ambiente, tais como cor dos olhos, inteligência etc. Daí, podemos concluir: nosso DNA codifica toda a informação necessária para nos descrever, mas esta informação está sob o controle de uma grande rede de regulação gênica que, associada às condições ambientais, gera as proteínas na quantidade certa, que farão de nós tudo aquilo que efetivamente somos.

Nos organismos que utilizam a reprodução sexuada, como os humanos e as moscas, cada progenitor fornece um pedaço de material genético chamado gametas. Estas gametas são resultado de um processo denominado crossing-over ou crossover, que permite que os filhos herdem características de ambos os pais sem ser exatamente iguais a estes. É importante considerar também que o processo de replicação do DNA é extremamente complexo. Pequenos erros podem ocorrer ao longo do tempo, gerando mutações dentro do código genético. Além dos erros, estas mutações podem ser causadas por fatores aleatórios, tais como presença de radiação ambiente, causando pequenas mudanças nos genes dos indivíduos. Estas mutações podem ser boas, ruins ou neutras.

Assim, temos o funcionamento básico da teoria da evolução: os genes dos bons indivíduos, combinados através do crossover e da mutação, tendem a gerar indivíduos ainda mais aptos e, assim, a evolução natural caminha, com uma tendência a gerar complexidade maior e maior adaptabilidade ao meio ambiente no qual os organismos estão inseridos.

diversidade, mudando aleatoriamente genes dentro de indivíduos e, assim como na natureza, é aplicada de forma menos frequente que a recombinação (crossover). Quando implementarmos nosso GA, veremos como estes operadores funcionam com detalhes.

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 (isto é, aqueles que têm uma função de avaliação maior) sejam selecionados mais frequentemente do que aqueles menos aptos, de forma que as boas características daqueles passem a predominar dentro da nova população de soluções. Nunca devemos descartar os indivíduos menos aptos da população reprodutora, pois 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.

Falando assim, de uma perspectiva totalmente teórica, tudo parece muito complicado. Vamos então partir para a prática, implementando um algoritmo genético em Java de forma que possamos entender exatamente os conceitos que descrevemos até aqui de forma abstrata.

Algoritmos Genéticos na prática

Primeiro nós vamos discutir como criar um único cromossomo e depois nós vamos discutir como fazer para manipular uma população inteira (isto é, o funcionamento do GA propriamente dito). Fazemos isto para ir do mais específico para o mais genérico, já que os cromossomos representam os indivíduos, que são os conceitos mais simples usados pelas populações.

Representando um cromossomo

Até agora falamos muito em termos teóricos. Vamos então implementar um GA para resolver o problema da mochila e conforme formos incrementando nossa implementação, os conceitos vistos até agora ficarão mais claros. Tudo que falarmos aqui refletirá nas listagens que colocaremos nesta seção, que são partes da classe Cromossomo. Todos os códigos usados neste artigo estão disponíveis para download no site http://www.algoritmosgeneticos.com.br.

Como colocamos na seção anterior, o nosso primeiro passo é encontrar uma representação, isto é, uma forma de codificar a informação em cromossomos que possam ser manipulados pelo nosso algoritmo genético. Para tanto, vamos analisar o nosso problema: nós temos N objetos que podem estar ou não dentro da mochila. Isto é, temos N variáveis binárias. Se considerarmos que cada variável é um bit, podemos representar uma possível solução como sendo uma string de bits na qual a posição i ser igual a 1 representa o fato de que o objeto de número i será colocado dentro da mochila. Por exemplo, o cromossomo dado por 10010001 representa o fato de que temos 8 objetos candidatos a estarem na mochila (por que a string tem comprimento igual a 8) e os objetos 1, 4 e 8 são aqueles que serão selecionados na mochila. Assim, podemos criar uma classe chamada Cromossomo que tenha dentro dela uma String e que receba como parâmetro o tamanho do cromossomo que cada objeto armazena.

Entretanto, para resolver o nosso problema da mochila, nosso cromossomo precisa de duas informações muito importantes, que são o valor e o peso de cada um dos candidatos a serem carregados. Assim, vamos colocar estes valores como parâmetros para nosso construtor. O tamanho dos vetores passados para o construtor pode, então, ser usado para inicializar o cromossomo. Temos um parâmetro adicional, que é o limite de peso que nossa mochila pode carregar – ele é muito importante, pois determina a qualidade de uma solução (uma solução de alto valor que ultrapasse o nosso limite de peso é, na realidade, uma má solução). Vamos ver como usá-lo quando implementarmos nossa função de avaliação.

Fazemos então a inicialização do cromossomo de forma aleatória – nós simplesmente escolhemos uma string de bits qualquer para cada cromossomo de nossa população e depois vamos evoluí-la. Nossa esperança é que o GA transforme este conjunto de cromossomos aleatórios em uma população que tenha ao menos uma solução excelente dentro dela. Entretanto, como faremos isto, fica para os próximos parágrafos. Neste momento, vamos inicializar nosso cromossomo simplesmente fazendo um loop que se repete N vezes (onde N é o tamanho do vetor de valores passado como parâmetro) e em cada iteração escolhe um ou zero, aleatoriamente. A implementação deste construtor pode ser visto na Listagem 1, a seguir.

Listagem 1. O construtor de nossa classe Cromossomo. A chamada ao construtor padrão, sem parâmetros, simplesmente inicializa as variáveis correntes, criando o StringBuffer que armazena o cromossomo e definindo os valores padrões das variáveis internas.

```
public Cromossomo(double pesos[], double valores[],
double limitePeso) {
   this();
   this.pesos=new double[pesos.length];
   this.valores=new double[valores.length];
   for(int i=0; i<pesos.length; i++) {
      this.pesos[i]=pesos[i];
      this.valores[i]=valores[i];
   }
   this.limitePeso=limitePeso;
   for(int i=1;i<=pesos.length;i++) {
```

```
        if (Math.random()<0.5) {
            cromossomo.append("0");
        } else {
            cromossomo.append("1");
        }
    }
}
```

Esta forma aleatória de inicializar a população é a escolha da maioria dos trabalhos feitos na área por ser a forma mais simples possível. A lei das probabilidades sugere que teremos uma distribuição que cobre praticamente todo o espaço de soluções, mas isto não pode ser garantido, pois a população tem tamanho finito (lembre-se do seu curso básico de estatística: as probabilidades só são valores exatos para um número infinito de sorteios).

Existem várias outras formas de fazer a inicialização ou de tentar garantir a diversidade da população inicial, mas como a inicialização aleatória é a forma mais simples de todas, vamos ficar com ela neste momento. Nosso objetivo é tentar entender como uma GA funciona. Quem quiser mais detalhes, pode depois consultar meu livro sobre o assunto que está mencionado na bibliografia do artigo ou então, se gostarem do artigo e quiserem uma continuação, mandem e-mails para a revista!

Eu estou o tempo todo falando em qualidade do cromossomo, pois isto é uma coisa realmente importante. Afinal, tenho que ter uma medida de como o cromossomo resolve o problema, tanto para saber se ele é uma solução aceitável quanto para poder usá-lo dentro do esquema de evolução. Vamos então criar uma função de avaliação para nosso cromossomo corrente.

A função de avaliação que usaremos é uma simples transcrição do nosso problema como um todo: somamos os valores associados aos elementos que decidimos carregar para ver o valor do nosso cromossomo. O problema é que nossa mochila também tem um limite de peso. Assim, também calculamos o peso total de nossos elementos. Se eles passarem do limite de peso, então atribuímos um valor de avaliação bem baixo. Não vamos atribuir zero, pois como veremos depois, valores de avaliação negativos ou zero não são bem tolerados por outros componentes do GA, mas isto é outra classe e nós vamos por enquanto continuar com nossa classe Cromossomo, sem nos adiantar para outras classes importantes do sistema. Assim, nossa função de avaliação fica como exposta na Listagem 2.

Listagem 2. O método que usamos para avaliar nosso cromossomo.

```
public double avaliacao() {
    double retorno=0;
    double somaPesos=0;
    String crom=this.getCromossomo().toString();
    for(int i=0;i<crom.length();i++) {
        if (crom.charAt(i)=='1') {
            retorno+=getValores()[i];
            somaPesos+=getPesos()[i];
        }
    }
    if (somaPesos>this.limitePeso) {
        retorno=1;
    }
    return(retorno);
}
```

De acordo com a teoria que colocamos nas seções anteriores (e que está exposta de forma esquematizada na figura 1), precisamos agora especificar dois operadores: o de mutação, que corresponde às alterações aleatórias na natureza e o de crossover, que corresponde à reprodução sexuada.

Vamos começar com o operador de mutação, pois ele é mais simples, operando da seguinte forma: ele tem associada uma probabilidade extremamente baixa (usualmente da ordem de 0,5%) e nós sorteamos um número entre 0 e 1. Se ele for menor que a probabilidade predeterminada então o operador atua sobre o gene em questão, alterando-lhe o valor aleatoriamente. Repete-se então o processo para todos os genes do filho sobre o qual estamos operando a mutação.

O código da Listagem 3 apresenta o método mutação da classe Cromossomo, que não recebe parâmetros e retorna um elemento da classe Cromossomo. Eu o fiz usando reflexão para permitir uma generalidade posterior – afinal, a ideia do GA é poder adaptá-lo ao seu problema específico. Para tanto, usamos o método getClass para saber qual é a classe concreta corrente e o método newInstance da classe obtida para instanciar o novo cromossomo que será o filho gerado. Uma vez criado este novo filho, tudo que fazemos é um loop em que cada operação um sorteio determina se vamos mudar o bit corrente ou mantê-lo igual.

Listagem 3. O método que realiza a mutação gerando um filho para nossa nova população. Note como usamos a reflexão para criar nosso filho, fazendo com que nosso cromossomo seja genérico e possa ser reutilizado em vários tipos distintos de problema.

```
public Cromossomo mutacao() {
    Class c=this.getClass();
    Cromossomo filho=null;
    try {
        filho = (Cromossomo) c.newInstance();
        StringBuffer resultado=new StringBuffer();
        for (int i=0;i<this.cromossomo.length();i++) {
            if (Math.random()<this.taxaMutacao) {
                if (this.cromossomo.charAt(i)=='1') {
```

```
                resultado.append('0');
            } else {
                resultado.append('1');
            }
        } else {
            resultado.append(this.cromossomo.charAt(i));
        }
    }
    filho.setCromossomo(new StringBuffer(resultado));
  } catch (Exception ex) {
    filho=null;
  }
  return(filho);
}
```

Uma vez compreendido o operador de mutação, vamos analisar o operador de crossover. Podemos dizer que este é o operador mais importante, pois ele replica a reprodução sexuada. Entenda que 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 podem-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.

Neste artigo nós vamos usar o operador de crossover mais simples, chamado de operador de crossover de um ponto. Outros operadores mais complexos (e, por consequência, mais eficientes) existem, mas não serão discutidos aqui por restrições de espaço.

O crossover de um ponto é extremamente simples. Tendo dois pais (que nós vamos discutir depois como selecionar), um ponto de corte é selecionado. Um ponto de corte constitui uma posição entre dois genes de um cromossomo. Cada indivíduo de n genes contém 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. Na figura 2 podemos ver um exemplo de pontos de corte. No caso, nosso cromossomo é composto de 5 genes e, por conseguinte, temos 4 pontos de corte possíveis.

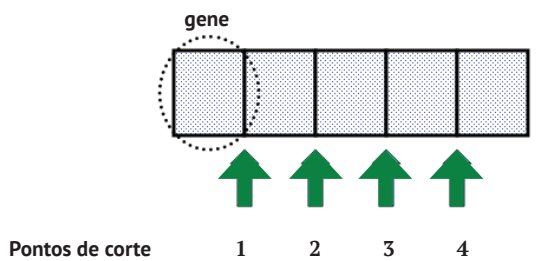

Figura 2. *Exemplo de pontos de corte.*

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

O primeiro filho é composto através da concatenação da parte do primeiro pai à esquerda do ponto de corte com a parte do segundo pai à direita do ponto de corte. O segundo filho é composto através da concatenação das partes que sobraram (a metade do segundo pai à esquerda do ponto de corte com a metade do primeiro pai à direita do ponto de corte). Um exemplo do funcionamento do crossover de um ponto pode ser visto na figura 3.

O código da Listagem 4 nos permite visualizar uma implementação deste algoritmo. Note que a essência do funcionamento do operador está marcada pelas linhas que criam as Strings stringFilho1 e stringFilho2 que fazem exatamente a concatenação de metades como descrevemos até agora. O resto do processo consiste no trabalho necessário para criar dois filhos cuja classe desconhecemos. Assim como fizemos no caso de mutação, usamos reflexão para fazê-lo. Agora, precisamos também usar a classe Array, que nos permite criar o vetor de dois filhos que serão o resultado do processo reprodutivo.

Note que ao contrário da mutação, que gerava um filho só, o operador de crossover gera dois filhos e ambos serão parte da nova população que analisaremos. Mais à frente, quando discutirmos a classe que controla a população do nosso algoritmo genético, nós veremos como controlar esta questão do número de filhos gerados.

Listagem 4. O método que realiza a reprodução dentro do algoritmo genético. Como ela fica dentro de um cromossomo e é um processo sexuado, o cromossomo this recebe como parâmetro outro cromossomo com o qual ele irá reproduzir. Note como mais uma vez usamos a reflexão para criar ambos os descendentes, fazendo com que nosso cromossomo seja genérico e possa ser reutilizado em vários tipos distintos de problema.

```
public Cromossomo[] crossover(Cromossomo outro) {
    Class c=this.getClass();
    Cromossomo[] filhos=null;
    filhos = (Cromossomo[]) Array.newInstance(c,2);
    try {
```

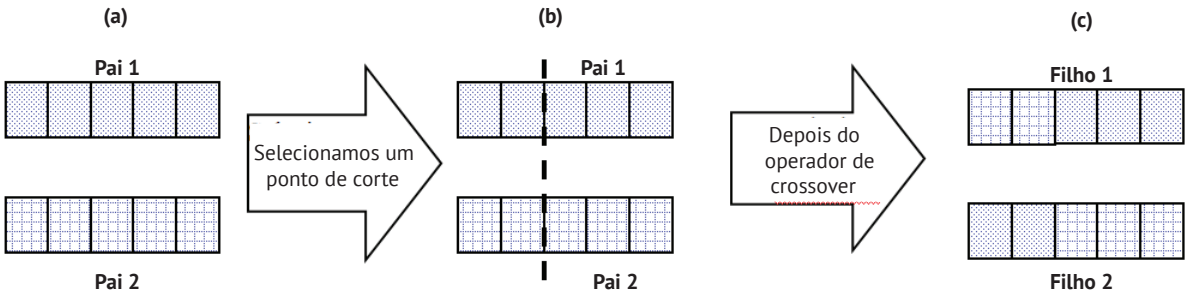

Figura 3. *Descrição da operação do operador de crossover de um ponto.*

```
        filhos[0] = (Cromossomo) c.newInstance();
        filhos[1] = (Cromossomo) c.newInstance();
        int posicaoCorte=(int) Math.round(Math.random()*
              this.cromossomo.length());
        String stringFilho1=outro.getCromossomo().
          substring(0, posicaoCorte+1) +this.
          getCromossomo().substring(posicaoCorte +1);
        String stringFilho2=this.getCromossomo().
           substring(0, posicaoCorte+1)+outro.
           getCromossomo().substring(posicaoCorte + 1);
        filhos[0].setCromossomo(new StringBuffer(
             stringFilho1));
        filhos[1].setCromossomo(new StringBuffer(
             stringFilho2));
    } catch (Exception ex) {
        filhos=null;
    }
    return(filhos);
}
```

Dentro da nossa classe Cromossomo temos mais alguns métodos, especialmente alguns utilitários que são comuns a todas as classes, como toString() e compareTo(). Não vamos comentá-los aqui por limitação de espaço, mas o código está disponível para download e análise.

A Classe controladora do GA

Até agora, tudo que nós fizemos foi criar uma classe para representar uma solução de um problema específico. Esta classe já é capaz de certas ações necessárias para um algoritmo genético, como a reprodução e a avaliação, mas para efetivamente executar um GA nós temos que controlar uma população e fazer as várias gerações serem avaliadas e reproduzidas. Para tanto, vamos criar uma classe chamada GA, que representará o arcabouço de nosso algoritmo genético.

Note que o único construtor público recebe a instância do problema da mochila que nos interessa. Afinal, um GA é uma técnica genética, mas nós temos um problema específico para resolver, não é mesmo? Temos então que embutir nosso problema dentro de nosso algoritmo genético de forma que possamos encontrar a solução que buscamos, pois este é o nosso objetivo final.

Nosso primeiro passo para executar o GA é inicializar a nossa população. Como cada cromossomo sabe inicializar-se aleatoriamente, usaremos o construtor de cada cromossomo como forma de realizar este trabalho. Assim, podemos ter um método que faz a inicialização aleatória da população, cujo código pode ser visto na Listagem 5. Este método será chamado no começo de cada execução do algoritmo genético, para que possamos reinicializar nossa tentativa de resolver o problema. Como o GA pode ser chamado várias vezes, podemos ter algo dentro da população corrente, e por isto limpamos a lista de cromossomos no começo do método.

Listagem 5. Método da classe GA que inicializa a população. Lembre-se de que nosso cromossomo, quando criado, já se inicializa de forma aleatória. Assim, usamos esta facilidade para criar nossa população.

```
public void inicializaPopulacao(int tamanhoPopulacao) {
    this.populacao.clear();
    for(int i=0;i<tamanhoPopulacao;i++) {
        this.populacao.add(new CromossomoMochila(pesos,
            valores, limitePeso));
    }
}
```

Lembre-se que uma importante característica de um GA é que ele mantém uma população de soluções a cada instante. Para tanto, nós vamos ter dentro de nossa classe um atributo que é uma List de soluções,

chamada população. Note-se que somos capazes de obter o melhor facilmente, como podemos ver na Listagem 6, que nos mostra o método getMelhorCorrente(). Usaremos este método em vários momentos, mas ele é especialmente necessário no fim da execução do GA, quando precisamos escolher uma das soluções e, obviamente, a que nos interessa mais é a melhor de todas aquelas que foram evoluídas. Note que como a classe CromossomoMochila implementa a interface Comparable, podemos nos utilizar desta facilidade para ordenar os elementos usando o método sort da classe Collections. Como este ordena em ordem ascendente, então o último é o melhor de todos (pois a comparação em CromossomoMochila se dá através da avaliação).

Listagem 6. O método que retorna a melhor solução mantida pelo GA em cada momento.

```
public CromossomoMochila getMelhorCorrente() {
    Collections.sort(populacao);
    return(populacao.get(populacao.size()-1));
}
```

Podemos agora ver o método run() da classe GA que é chamado para executarmos o nosso algoritmo genético. Como podemos ver na Listagem 7, ele simplesmente representa uma implementação dos conceitos vistos na figura 1. Temos um laço baseado na variável geracaoCorrente que determina quantas gerações foram passadas. Como a figura 1 coloca, temos que determinar um critério de término e, no caso de nosso GA mais simples, este será determinado unicamente pelo número de gerações decorridas (valor passado como parâmetro).

Dentro deste loop temos outro loop, que é controlado pela variável individuosGerados. Ele representa o fato de que temos que gerar uma nova população a cada geração, o que fazemos aplicando os operadores de crossover e mutação a pais selecionados através de chamadas ao método selecionaPai, que veremos mais à frente.

Agora que estamos iniciando nossa jornada através dos GAs, iremos trabalhar com a versão mais simples dos operadores genéticos, na qual eles atuam em conjunto, como se fossem um só. Existem outros modos de selecionar qual operador será aplicado em um determinado momento, mas agora o objetivo é entender completamente os conceitos dos operadores. Quem estiver interessado em ver outras maneiras de selecionar os operadores, pode consultar meu livro citado nas referências. Assim, para aplicarmos em conjunto, primeiro aplicamos o crossover, que nos retorna dois filhos e adicionamos à nova geração as versões destes filhos que sofreram a mutação.

A última parte do método run() consiste na parte que fica fora do loop controlado por individuosGerados. Ela corresponde ao momento em que temos uma população completa da nova geração e uma população de pais. A pergunta é: como faremos para compor a nova população?

Na área de algoritmos genéticos e na figura 1, nós chamamos esta parte de módulo de população, que é o responsável pelo controle da nossa população. Por uma questão de simplicidade, assumiremos que esta população 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, sabemos que 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 criados 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 tornar-se os pais da nova geração. É exatamente isto que o nosso código faz: limpa a lista população e depois acrescenta todos os novos indivíduos gerados a ela.

Como sempre, existem maneiras mais complexas de lidar com o módulo de população, mas por uma questão de simplicidade e de limitação de espaço, não vamos lidar com elas neste momento.

Note que nós imprimimos o indivíduo de melhor valor em cada geração, para que possamos acompanhar a evolução de nossa população (o que é ótimo em termos didáticos) e para que tenhamos a solução do problema que desejamos, o que, afinal, é nosso objetivo. Para tanto, nós usamos o nosso método getMelhorCorrente(), descrito anteriormente.

Listagem 7. Método run() da classe GA que efetivamente executa nosso algoritmo genético.

```
public void run(double taxaMutacao, int numGeracoes,
int tamanhoPopulacao) {
   CromossomoMochila melhor;
   List<CromossomoMochila> novaPopulacao=
      new ArrayList<CromossomoMochila>();
   inicializaPopulacao(tamanhoPopulacao);
   for(int geracaoCorrente=0;geracaoCorrente
   <numGeracoes;geracaoCorrente++) {
      double somaAvaliacoes=this.somaAvaliacoes();
      novaPopulacao.clear();
      for(int individuosGerados=0;individuosGerados<
      tamanhoPopulacao;individuosGerados+=2) {
         int pai1=this.selecionaPai(somaAvaliacoes);
         int pai2=this.selecionaPai(somaAvaliacoes);
         CromossomoMochila[] filhos=populacao.
            get(pai1).crossover(populacao.get(pai2));
         novaPopulacao.add(filhos[0].mutacao());
         novaPopulacao.add(filhos[1].mutacao());
      }
      populacao.clear();
      populacao.addAll(novaPopulacao);
      melhor=getMelhorCorrente();
      System.out.println("Geracao #"+geracaoCorrente+"->
         "+melhor+" com avaliação "+melhor.avaliacao());
   }
}
```

Falta apenas entender agora como os dois pais são selecionados e teremos um algoritmo genético completo em nossas mãos. O método que usaremos é chamado de método da roleta viciada e antes de partirmos para sua implementação, vamos discutir alguns conceitos importantes.

O método de seleção de pais deve simular o mecanismo de seleção natural que atua sobre as espécies biológicas, em que os pais mais capazes geram mais filhos, ao mesmo tempo em que permite que os pais menos aptos também gerem descendentes.

O conceito fundamental é que temos que privilegiar os indivíduos com função de avaliação alta, sem desprezar completamente aqueles indivíduos com função de avaliação extremamente baixa. Esta decisão é razoável, pois até indivíduos com péssima avaliação podem ter características genéticas que sejam favoráveis à criação de um indivíduo que seja a melhor solução para o problema que está sendo atacado, características estas que podem não estar presentes em nenhum outro cromossomo de nossa população.

É importante entender que se deixarmos apenas os melhores indivíduos se reproduzirem, a população tenderá a ser composta de indivíduos cada vez mais semelhantes e faltará diversidade a esta população para que a evolução possa prosseguir de forma satisfatória. A este efeito denominamos convergência genética, e selecionando de forma justa os indivíduos menos aptos podemos evitá-lo, ou pelo menos minimizá-lo. Lembre-se de que, na natureza, os indivíduos mais fracos também geram uma prole, apesar de fazê-lo com menos frequência do que os mais aptos. Logo, seria interessante reproduzir esta possibilidade dentro dos GAs. Por isto, vamos usar um método simples capaz de implementar estas características, denominado o método da roleta viciada.

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

O ato de rodar a roleta deve ser completamente aleatório, escolhendo um número entre 0 e 100 (representando a percentagem que corresponde a cada indivíduo) ou entre 0 e 360 (representando uma posição do círculo) ou ainda entre 0 e a soma total das avaliações (representando um pedaço do somatório). Quando se faz este sorteio um número suficiente de vezes, cada indivíduo é selecionado um número de vezes igual à sua fração na roleta (quem lembra de suas aulas de probabilidade no ensino médio sabe que esta igualdade só é válida, como em todos os processos probabilísticos, quando o número de sorteios é igual a infinito. Entretanto, para um número grande de sorteios, os valores obtidos são muito próximos ao valor percentual da avaliação).

Obviamente não podemos girar uma roleta dentro do computador, sendo obrigados a trabalhar com conceitos abstratos, e não roletas físicas. Logo, precisamos de uma versão computacional da roleta, que é dada por um algoritmo que pressupõe que nenhum indivíduo tenha uma avaliação nula ou negativa. O código Java que implementa estes conceitos é dado na Listagem 8.

Listagem 8. Método que implementa a seleção dos pais usando o conceito de roleta viciada.

```
private int selecionaPai(double somaAvaliacoes) {
   int retorno=-1;
   double valorSorteado=Math.random()*somaAvaliacoes;
   double soma=0;
   Iterator<CromossomoMochila> it=this.populacao.
      iterator();
   do {
      soma+=it.next().avaliacao();
      retorno++;
   } while ((it.hasNext())&&(soma<valorSorteado));
   return(retorno);
}
```

Este método primeiro escolhe um valor entre 0 e a soma das avaliações (passada como parâmetro para pelo método run). Nós usamos o método random para escolher aleatoriamente um número entre 0 e 1 que multiplicamos, nesta linha, por somaAvaliacoes para obter uma percentagem do valor armazenado nesta variável.

O loop que fazemos usando o Iterator vai somando os valores das avaliações de cada um dos indivíduos. Quando exceder o valor determinado pelo sorteio aleatório, o loop termina. Note que colocamos um teste para sair do loop se o Iterator não tiver mais elementos para nos fornecer. Isto não é necessário, mas é sempre boa prática de programação fazer o máximo para evitar que erros aconteçam em um programa. Note que se a população tiver zero indivíduos, a rotina retornaria –1 (isto é, nenhum indivíduo válido), causando uma exceção no método run(). Isto é uma situação esdrúxula, mas se você for fazer um código comercial baseado nestas listagens (que só têm fins didáticos), deve tomar alguma precaução para lidar com esta situação semi-absurda.

Obviamente, só coloquei neste artigo fragmentos de código de forma a tornar claros os conceitos de algoritmos genéticos. Mais uma vez, coloco que aqueles leitores que quiserem os códigos completos podem fazer o download do projeto para Netbeans no endereço http://www.algoritmosgeneticos.com.br onde também podem ser encontradas aulas que preparei sobre o assunto e muitos outros recursos ligados à área dos GAs.

Executando nosso algoritmo

Para testar nosso algoritmo genético nós temos uma classe Main que cria uma pequena instância do problema da mochila e executa nossa GA, como pode ser visto na Listagem 9.

Listagem 9. Método main que cria uma instância do problema da mochila e executa o GA.

public static void main(String[] args) {

```
public static void main(String[] args) {
   double pesos[]={10,20,30,20,10,90, 70, 100, 50};
   double valores[]={5,30,20,20,10, 120, 60, 10, 5};
   GA meuGA=new GA(pesos, valores, 100);
   meuGA.run(0.01, 15, 10);
}
```

A instância que criamos representa um problema particularmente difícil. Seu máximo é o valor 130, que consiste em colocarmos na mochila o 5º e o 6º itens (para nossa contagem estamos tratando o 1º como número 1, não como "zerésimo"), totalizando exatamente 100kg. Note entretanto que toda e qualquer solução que inclua o 6º item vai exceder o limite de peso da mochila, resultando em uma avaliação bem baixa. Este tipo de problema em que o máximo global está cercado de valores baixos é chamado de problema enganador e nenhum método tradicional é capaz de resolvê-lo a contento.

Note que avaliamos apenas 150 soluções (15 gerações de 10 indivíduos cada). Este é um número comparável ao número de soluções existentes que é a soma das combinações um a um (8), dois a dois (28), três a três (56), quatro a quatro (70), cinco a cinco (56), seis a seis (28), sete a sete (8) e oito a oito (1), totalizando 255 soluções. Isto só ocorre por que escolhemos um problema bem pequeno. Se tivéssemos 20 objetos ao invés de 8 o número de soluções possível chegaria aos milhões e poderíamos ainda tentar populações relativamente pequenas (uns 100) com um número baixo de gerações (uns 40), gerando uma fração pequena das soluções totais.

A figura 4 mostra o resultado de quatro execuções distintas de nosso algoritmo genético. Não alterei nenhum parâmetro entre as execuções, apenas deixei a aleatoriedade inerente dos GAs atuar de forma livre. Cada uma delas tem características interessantes que nos trazem alguns conceitos que precisamos aprender antes de usar os GA para nossos problemas reais.

Em (a) vemos uma população que começa bem ruim e que efetivamente melhora até atingir uma solução razoável. Note que esta execução mostra que não existe nenhuma garantia de desempenho nos GAs: há várias soluções superiores a esta que obtivemos. Assim, percebe-se que um GA deve ser executado várias vezes para se obter melhores resultados e a medida de seu desempenho é dada pela melhor solução obtida – um GA é uma ferramenta para solução de um problema prático e como tal deve ser tratada.

Em (b) vemos uma situação em que a população já surge com um elemento muito bom que surgiu da inicialização aleatória. Depois, a população nunca melhora. Isto é uma possibilidade real e mais uma vez implica em não podermos fazer uma única execução e também no fato de que existe uma tendência a melhora das soluções de acordo com a seleção natural, mas não uma garantia de melhora.

Em (c) vemos um caso interessante que consiste em uma população que piora durante sua execução antes de melhorar. Como falamos antes, não existe garantia de que os filhos serão melhores do que os pais – é possível que o crossover e a mutação gerem filhos piores. Existem módulos de população que preservam as melhores soluções, chamados de módulos com elitismo, mas eles só garantem a estabilidade do melhor, não a sua evolução.

Por último, em (d), vemos um GA que evoluiu para a solução ideal. Não existe garantia de que isto vai acontecer, mas mostrar que isto é uma possibilidade real é algo que provavelmente será reconfortante para você, meu caro leitor.

Considerações finais

Espero que este artigo tenha permitido que você compreenda todos os conceitos fundamentais sobre os algoritmos genéticos. Eles são uma interessante ferramenta para efetuar buscas em espaços praticamente infinitos, mas sua aplicabilidade é limitada basicamente a este tipo de problema. GAs não são uma técnica que revolucionará toda a ciência da computação. Os praticantes de uma área gostam, de uma forma geral, de promovê-la anunciando que ela seria a solução milagrosa para todos os problemas computacionais existentes, mas uma análise cuidadosa de qualquer área desmente tal hype.

É importante entender que existem várias classes de problemas com uma solução natural, que não podem ser batida por nenhuma técnica. Por exemplo, problemas de minimização de funções quadráticas devem ser atacados com o método de Newton, que é capaz de resolvê-los em uma única iteração.

Entretanto, podemos afirmar que, como ferramenta de busca, os GAs se mostram extremamente eficientes, encontrando boas soluções para problemas que talvez não fossem solúveis de outra forma, além de serem extremamente simples de implementar e modificar. O custo de pessoal para implementação de um GA é relativamente pequeno, pois GA é uma técnica que tem vários métodos que podem ser reutilizados de forma indefinida, sendo os programas quase rudimentares em sua essência. Existe um "senão" escondido nesta afirmação, que pode ser resumido em um trecho extraído de "How to solve it", de Z. Michalewicz e D. B. Fogel:

(a)

```
Melhor da geracao #0 = 000111100 com avaliação 1.0
Melhor da geracao #1 = 000110000 com avaliação 30.0
Melhor da geracao #2 = 001110000 com avaliação 50.0
Melhor da geracao #3 = 001110000 com avaliação 50.0
Melhor da geracao #4 = 001110000 com avaliação 50.0
Melhor da geracao #5 = 101110000 com avaliação 55.0
Melhor da geracao #6 = 101110000 com avaliação 55.0
Melhor da geracao #7 = 101110000 com avaliação 55.0
Melhor da geracao #8 = 000110100 com avaliação 90.0
Melhor da geracao #9 = 000110100 com avaliação 90.0
Melhor da geracao #10 = 000110100 com avaliação 90.0
Melhor da geracao #11 = 000110100 com avaliação 90.0
Melhor da geracao #12 = 000110100 com avaliação 90.0
Melhor da geracao #13 = 000110100 com avaliação 90.0
Melhor da geracao #14 = 000110100 com avaliação 90.0
```

(b)

```
Melhor da geracao #0 = 010010100 com avaliação 100.0
Melhor da geracao #1 = 010010100 com avaliação 100.0
Melhor da geracao #2 = 010010100 com avaliação 100.0
Melhor da geracao #3 = 010010100 com avaliação 100.0
Melhor da geracao #4 = 010010100 com avaliação 100.0
Melhor da geracao #5 = 010010100 com avaliação 100.0
Melhor da geracao #6 = 010010100 com avaliação 100.0
Melhor da geracao #7 = 010010100 com avaliação 100.0
Melhor da geracao #8 = 010010100 com avaliação 100.0
Melhor da geracao #9 = 010010100 com avaliação 100.0
Melhor da geracao #10 = 010010100 com avaliação 100.0
Melhor da geracao #11 = 010010100 com avaliação 100.0
Melhor da geracao #12 = 010010100 com avaliação 100.0
Melhor da geracao #13 = 010010100 com avaliação 100.0
Melhor da geracao #14 = 010010100 com avaliação 100.0
```

(c)

```
Melhor da geracao #0 = 010100001 com avaliação 55.0
Melhor da geracao #1 = 010110000 com avaliação 60.0
Melhor da geracao #2 = 010110001 com avaliação 65.0
Melhor da geracao #3 = 110110000 com avaliação 65.0
Melhor da geracao #4 = 110100001 com avaliação 60.0
Melhor da geracao #5 = 110100001 com avaliação 60.0
Melhor da geracao #6 = 110100001 com avaliação 60.0
Melhor da geracao #7 = 110100001 com avaliação 60.0
Melhor da geracao #8 = 010110001 com avaliação 65.0
Melhor da geracao #9 = 110100001 com avaliação 60.0
Melhor da geracao #10 = 010100001 com avaliação 55.0
Melhor da geracao #11 = 010110001 com avaliação 65.0
Melhor da geracao #12 = 110000001 com avaliação 40.0
Melhor da geracao #13 = 110000100 com avaliação 95.0
Melhor da geracao #14 = 110000100 com avaliação 95.0
```

(d)

```
Melhor da geracao #0 = 110010001 com avaliação 50.0
Melhor da geracao #1 = 110110000 com avaliação 65.0
Melhor da geracao #2 = 110110000 com avaliação 65.0
Melhor da geracao #3 = 110110000 com avaliação 65.0
Melhor da geracao #4 = 010010100 com avaliação 100.0
Melhor da geracao #5 = 010010100 com avaliação 100.0
Melhor da geracao #6 = 010010100 com avaliação 100.0
Melhor da geracao #7 = 000011000 com avaliação 130.0
Melhor da geracao #8 = 000011000 com avaliação 130.0
Melhor da geracao #9 = 000011000 com avaliação 130.0
Melhor da geracao #10 = 000011000 com avaliação 130.0
Melhor da geracao #11 = 000011000 com avaliação 130.0
Melhor da geracao #12 = 000011000 com avaliação 130.0
Melhor da geracao #13 = 000011000 com avaliação 130.0
Melhor da geracao #14 = 000011000 com avaliação 130.0
```

Figura 4. *Resultado de quatro execuções distintas do nosso algoritmo genético.*

"Ao invés de devotar o tempo necessário e o pensamento crítico necessário para entender um problema e para ajustar nossa representação dos seus requisitos, nós ficamos complacentes e simplesmente buscamos a sub-rotina mais conveniente, uma pílula mágica para curar nossas doenças".

Esta afirmação procura dizer que a abordagem de simplesmente pegar um GA padronizado e executá-lo até o computador cansar ou até que um bom resultado seja obtido não é a melhor possível. Muitos praticantes da área têm o hábito de simplesmente pegar um GA com representação binária, com operadores competitivos e probabilidade de 80% para o crossover, taxa de mutação de 1% e população de 100 indivíduos, para tentar aplicá-lo ao problema que eles estão enfrentando neste momento. Será que isto parece uma solução razoável? Espero que você tenha respondido "não" para esta pergunta.

A verdade é que, como Michalewicz e Fogel apontam e como o teorema da inexistência do almoço grátis categoricamente afirma, dois métodos não informados terão, em média, o mesmo desempenho ao longo de uma grande série de problemas. Você pode até ter sorte na resolução de um problema específico, mas sorte não é um bom substituto para a competência e o estudo cuidadoso.

Este resultado não deve deprimi-lo nem fazê-lo dizer que qualquer ferramenta é igual. Ao contrário, ele deve fazê-lo ficar feliz por poder optar pelo uso de algoritmos genéticos para a resolução de problemas. Afinal, os GAs permitem que informações sobre o problema sejam embutidas dentro da representação (proibindo soluções que desrespeitem restrições do problema), dentro da função de avaliação (recompensando aquelas soluções que estão mais próximas de resolver o problema) e até mesmo dentro dos operadores genéticos. Isto quer dizer que se pode esperar que os GAs tenham um desempenho superior aos algoritmos não informados em vários problemas difíceis de nossa vida cotidiana.

A palavra mágica neste caso é informação. Isto é, antes de tentar resolver um problema, você deve entendê-lo profundamente. Assim, você poderá escolher a ferramenta correta. Se porventura esta ferramenta for um algoritmo genético, a sua compreensão do problema permitirá que você faça escolhas inteligentes de representação, função de avaliação e operadores genéticos, de forma que o desempenho de seu GA seja maximizado.

Tudo isto faz com que cheguemos à conclusão de que um GA não deve ser, necessariamente, a primeira ferramenta na sua mente. Existem várias técnicas tradicionais de resolução de problemas que podem ser aplicadas em várias situações. Considere-as primeiro e, somente se elas se mostrarem incapazes de resolver o seu problema a contento, parta para um algoritmo genético.

Lembre-se: uma pessoa que tem uma grande caixa de ferramentas é capaz de resolver mais problemas, e cada um deles de forma mais eficiente, do que uma pessoa que só tem um martelo ou uma chave de fenda. Mantenha a sua caixa de ferramentas sempre cheia!

/para saber mais

> "Na MundoJ 46 eu escrevi um artigo intitulado "Começando a pensar reflexivamente" que cobre os principais aspectos da reflexão, especialmente aqueles que usamos para criar os filhos dos nossos cromossomos.

> "A área de algoritmos genéticos é muito maior do que um artigo de revista, apesar deste ser suficiente para transmitir os conceitos fundamentais para sua compreensão. Entretanto, aqueles que quiserem se aprofundar podem buscar o meu livro, "Algoritmos Genéticos", publicado pela Editora Ciência Moderna e que já está em sua terceira edição.

/referências

> LINDEN, R., 2012, "Algoritmos Genéticos", 3ª edição, Editora Ciência Moderna, Rio de Janeiro, 2012.

> MICHALEWICZ, Z., FOGEL, D. B., 2010, "How to Solve It: Modern Heuristics", 2ª Edição, Springer-Verlag, Berlim, Alemanha.

> RUSSEL, S. J.; NORVIG, P., "Inteligência Artificial", 2ª edição, Elsevier Editora, Rio de Janeiro, 2004.

> TOSCANI, L. V., VELOSO, P. A. S., "Complexidade de Algoritmos: análise, projetos e métodos", Ed. Bookman, Porto Alegre, 2009.