UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

Análise Formal da Complexidade de Algoritmos Genéticos

por

MARILTON SANCHOTENE DE AGUIAR

Dissertação submetida a avaliação,
como requisito parcial para a obtenção do grau de
Mestre em Ciência da Computação

Profa. Dra. Laira Vieira Toscani
Orientador

Porto Alegre, abril de 1998

CIP — CATALOGAÇÃO NA PUBLICAÇÃO

Aguiar, Marilton Sanchotene de

Análise Formal da Complexidade de Algoritmos Genéticos / por Marilton Sanchotene de Aguiar. — Porto Alegre: PPGC da UFRGS, 1998.

75 f.: il.

Dissertação (mestrado) — Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 1998. Orientador: Toscani, Laira Vieira.

1. Algorithms Development. 2. Complexity. 3. Genetic Algorithms. I. Toscani, Laira Vieira. II. Título.

Dedico este trabalho aos meus familiares,
amigos e em memória de minha mãe.

Agradecimentos

Agradeço:

- à querida amiga e orientadora professora Dra. Laira Vieira Toscani, pela amizade, carinho e acolhida desde o início do curso, pela confiança, dedicação e ensinamentos indispensáveis que me fizeram perceber a importância deste trabalho;

- às professoras, amigas e mães, Graçaliz Pereira Dimuro e Renata Hax Sander Reiser, pelos anos de pesquisa juntos, pelo investimento em minha vida acadêmica que agora estamos colhendo juntos, pela presença nos diversos momentos que passei, fáceis e difíceis, " nas longas estradas da vida";

- aos amigos que me acompanharam em Porto Alegre, Ana Paula e Luis Fernando, ouviram pacientemente as dúvidas e indecisões que tive, e retribuíram carinho, alegria e segurança;

- aos amigos que conviveram comigo em Pelotas, Rafael, Fábio e Renata, pelos momentos alegres, felizes e intensos;

- aos colegas de curso, pelas discussões, sugestões e apoios recebidos, em especial, das amigas Sílvia Dias da Costa Lemos e Fabiana Zamora Wilke;

- ao CNPq, pelo auxílio em forma de bolsa; e,

- à minha família, especialmente ao meu pai Pedro e meus irmãos Maurício e Marlise que sempre me incentivaram e compreenderam as minhas preocupações e ansiedades durante todo o tempo da realização de mais esta tarefa.

Sumário

Lista de Figuras

Lista de Tabelas

Resumo

O objetivo do trabalho é estudar a viabilidade de tratar problemas de otimização, considerados intratáveis, através de Algoritmos Genéticos, desenvolvendo critérios para a avaliação qualitativa de um Algoritmo Genético. Dentro deste tema, abordam-se estudos sobre complexidade, classes de problemas, análise e desenvolvimento de algoritmos e Algoritmos Genéticos, este último sendo objeto central do estudo. Como produto do estudo deste tema, é proposto um método de desenvolvimento de Algoritmos Genéticos, utilizando todo o estudo formal de tipos de problemas, desenvolvimento de algoritmos aproximativos e análise da complexidade.

O fato de um problema ser teoricamente resolvível por um computador não é suficiente para o problema ser na prática resolvível. Um problema é denominado tratável se no pior caso possui um algoritmo razoavelmente eficiente. E um algoritmo é dito razoavelmente eficiente quando existe um polinômio p tal que para qualquer entrada de tamanho n o algoritmo termina com no máximo p(n) passos [SZW 84]. Já que um polinômio pode ser de ordem bem alta, então um algoritmo de complexidade polinomial pode ser muito ineficiente. A premissa dos Algoritmos Genéticos é que se pode encontrar soluções aproximadas de problemas de grande complexidade computacional mediante um processo de *evolução simulada*[LAG 96].

Como produto do estudo deste tema, é proposto um método de desenvolvimento de Algoritmos Genéticos com a consciência de qualidade, utilizando todo o estudo formal de tipos de problemas, desenvolvimento de algoritmos aproximativos e análise da complexidade. Uma axiomatização tem o propósito de dar a semântica do algoritmo, ou seja, ela define, formalmente, o funcionamento do algoritmo, mais especificamente das funções e procedimentos do algoritmo. E isto, possibilita ao projetista de algoritmos uma maior segurança no desenvolvimento, porque para provar a correção de um Algoritmo Genético que satisfaça esse modelo só é necessário provar que os procedimentos satisfazem os axiomas.

Para ter-se consciência da qualidade de um algoritmo aproximativo, dois fatores são relevantes: a exatidão e a complexidade. Este trabalho levanta os pontos importantes para o estudo da complexidade de um Algoritmo Genético. Infelizmente, são fatores conflitantes, pois quanto maior a exatidão, pior (mais alta) é a complexidade, e vice-versa. Assim, um estudo da qualidade de um Algoritmo Genético, considerado um algoritmo aproximativo, só estaria completa com a consideração destes dois fatores. Mas, este trabalho proporciona um grande passo em direção do estudo da viabilidade do tratamento de problemas de otimização via Algoritmos Genéticos.

Palavras-chave: Algorithms Development, Complexity, Genetic Algorithms.

TITLE: "FORMAL ANALYSIS OF GENETIC ALGORITHMS COMPLEXITY"

Abstract

The objective of the work is to study the viability of treating optimization problems, considered intractable, through Genetic Algorithms, developing approaches for the qualitative evaluation of a Genetic Algorithm. Inside this theme, approached areas: complexity, classes of problems, analysis and development of algorithms and Genetic Algorithms, this last one being central object of the study. As product of the study of this theme, a development method of Genetic Algorithms is proposed, using the whole formal study of types of problems, development of approximate algorithms and complexity analysis.

The fact that a problem theoretically solvable isn't enough to mean that it is solvable in pratice. A problem is denominated easy if in the worst case it possesses an algorithm reasonably efficient. And an algorithm is said reasonably efficient when a polynomial p exists such that for any entrance size n the algorithm terminates at maximum of p(n) steps [SZW 84]. Since a polynomial can be of very high order, then an algorithm of polynomial complexity can be very inefficient. The premise of the Genetic Algorithms is that one can find approximate solutions of problems of great computational complexity by means of a process of *simulated evolution* [LAG 96].

As product of the study of this theme, a method of development of Genetic Algorithms with the quality conscience is proposed, using the whole formal study of types of problems, development of approximate algorithms and complexity analysis. The axiom set has the purpose of giving the semantics of the algorithm, in other words, it defines formally the operation of the algorithm, more specifically of the functions and procedures of the algorithm. And this, facilitates the planner of algorithms a larger safety in the development, because in order to prove the correction of a Genetic Algorithm that satisfies that model it is only necessary to prove that the procedures satisfy the axioms.

To have conscience of the quality of an approximate algorithm, two factors are important: the accuracy and the complexity. This work lifts the important points for the study of the complexity of a Genetic Algorithm. Unhappily, they are conflicting factors, because as larger the accuracy, worse (higher) it is the complexity, and vice-versa. Thus, a study of the quality of a Genetic Algorithm, considered an approximate algorithm, would be only complete with the consideration of these two factors. But, this work provides a great step in direction of the study of the viability of the treatment of optimization problems through Genetic Algorithms.

Keywords: Algorithms Development, Complexity, Genetic Algorithms.

1 Introdução

1.1 Tema

O objetivo da dissertação é estudar a viabilidade de tratar problemas de otimização, considerados intratáveis, através de Algoritmos Genéticos, desenvolvendo critérios para a avaliação qualitativa de um Algoritmo Genético.

Dentro deste tema, abordam-se estudos sobre computabilidade e complexidade, classes de problemas, análise e desenvolvimento de algoritmos e Algoritmos Genéticos, este último sendo objeto central do estudo e formalização.

Como produto do estudo deste tema, é proposto um método de desenvolvimento de Algoritmos Genéticos, utilizando todo o estudo formal de tipos de problemas, desenvolvimento de algoritmos aproximativos e análise da complexidade. Este método é direcionado para o auxílio ao desenvolvimento de Algoritmos Genéticos com a consciência de qualidade, com uso da análise da complexidade, e permitir (estimular) o desenvolvimento deste tipo de algoritmo com aspectos de outras metodologias de desenvolvimento facilitado pela especificação formal do método.

1.2 Motivação

Os Algoritmos Genéticos representam o resultado da mimetização tecnológica que, embora muito aquém do modelo original, dá sinais de um excelente potencial para simular a vida [RAM 94]. A premissa dos Algoritmos Genéticos é que se pode encontrar soluções aproximadas de problemas de grande complexidade computacional mediante um processo de *evolução simulada*[LAG 96].

O fato de um problema ser teoricamente resolvível por um computador não é suficiente para o problema ser na prática resolvível. Um problema é denominado tratável se no pior caso possui um algoritmo razoavelmente eficiente. E um algoritmo é dito razoavelmente eficiente quando existe um polinômio p tal que para qualquer entrada de tamanho n o algoritmo termina com no máximo p(n) passos [SZW 84]. Já que um polinômio pode ser de ordem bem alta, então um algoritmo de complexidade polinomial pode ser muito ineficiente. Entretanto, se um algoritmo não possui complexidade polinomial, certamente é ineficiente e intratável na prática.

Muitos problemas de aplicação prática, são algoritmicamente resolvíveis, mas o espaço ou o tempo necessário à execução do mesmo é muito grande para que o programa tenha utilidade prática [TOS 86].

Outras motivações encontram-se nos seguintes questionamentos:

Dado um problema de otimização:

- O problema é NP-Completo?
- Sendo NP-Completo, é tolerável uma solução aproximada para o problema?
- Dado um algoritmo aproximativo, qual sua exatidão e complexidade?
- Dado um Algoritmo Genético, como medir a qualidade de uma solução obtida?
- Como acelerar o Algoritmo Genético?

1.3 Objetivos

O objetivo geral deste trabalho é estudar a viabilidade de tratar problemas de otimização, considerados intratáveis, através de Algoritmos Genéticos. Para isso, desenvolveu-se um método formal de desenvolvimento de Algoritmos Genéticos, que possibilita avaliar, tendo um conhecimento *a priori*, a qualidade do processamento do algoritmo em problemas de otimização.

Como objetivos específicos, pode-se salientar:

- analisar os procedimentos e etapas que compõem um Algoritmo Genético;
- desenvolver uma equação de complexidade para a análise de um Algoritmo Genético;
- permitir a viabilização do uso de Algoritmos Genéticos através da análise de sua complexidade.

1.4 Contribuição do trabalho

Com base nos trabalhos desenvolvidos, nos artigos publicados e nos objetivos propostos, pode-se estabelecer as seguintes contribuições do trabalho:

- estudo abrangente sobre os elementos da teoria da computabilidade e da teoria da complexidade computacional, conhecendo as definições formais de tipos de problemas, classes de complexidade, medida de qualidade de métodos de desevolvimento de algoritmos;
- estudo detalhado das variantes dos Algoritmos Genéticos quanto à estrutura dos procedimentos e funções, e quanto à aplicabilidade[AGU 96];
- especificação formal do Algoritmo Genético básico atendendo aos cuidados das demonstrações requeridas pelos axiomas que definiram a semântica do algoritmo, pela definição dos domínios envolvidos no processo, e pela sintaxe dos predicados e funções. Isto resultou em um método de desenvolvimento deste tipo de algoritmo;
- assistência à avaliação da qualidade de um Algoritmo Genético através do estudo formal e da análise da complexidade determinada a partir dos programas abstratos;
- a análise da complexidade de Algoritmos Genéticos como subsídio para decisão da viabilidade da aplicação deste tipo de algoritmo na resolução de problemas.

1.5 Estrutura do texto

A dissertação está organizada em 8 capítulos. Cada capítulo aborda os passos necessários para a formalização do método de desenvolvimento de Algoritmos Genéticos e o método propriamente dito.

O capítulo 2 descreve os tipos de problemas, com a definição formal de problema dado por [VEL 84] introduzindo as operações de redução e decomposição de

problemas e a definição de problema no ponto de vista computacional como problemas de otimização e decisão.

No capítulo 3 apresenta-se: o tipo de complexidade adotada no trabalho; as principais medidas de complexidade, inclusive para algoritmos randômicos; e introduz-se as classes de complexidade P, NP, NP-Completo, NPI, NP-Difícil e as relações entre as principais classes.

O capítulo 4 mostra: a definição de algoritmo aproximativo, introduzindo as medidas de qualidade para algoritmos aproximativos; as categorias de algoritmos aproximativos e as classes de problemas de otimização; e, o conceito de redutibilidade.

O capítulo 5 constitui-se num estudo abrangente da história, da fundamentação, do funcionamento, da constituição dos módulos (composição), das opções de otimização, e da aplicabilidade dos Algoritmos Genéticos. Este capítulo destaca-se pela sua importância no auxílio da formalização e na composição do método de desenvolvimento de Algoritmos Genéticos. Além disso, caracteriza o tipo de Algoritmo Genético estudado e formalizado.

No capítulo 6 é proposto o método de desenvolvimento propriamente dito, baseado na formalização de um MDA (Método de Desenvolvimento de Algoritmos) [TOS 88], através de um programa abstrato e um conjunto de axiomas, que dão a semântica ao algoritmo cujas funções e procedimentos estão definidos sintaticamente pelo diagrama sintático.

O capítulo 7 mostra o desenvolvimento de uma equação de complexidade para o Algoritmo Genético embasado no estudo formal efetuado, permitindo a análise da complexidade do algoritmo abstrato.

Por fim, o capítulo 8 apresenta as conclusões obtidas como resultado da realização deste trabalho.

2 Teoria dos Problemas

A capacidade de especificar uma seqüência finita de ações que conduzem a um resultado é chamada de computabilidade. Bovet [BOV 94] mostra que uma ação só pode ser entendida depois de ser definido um modelo de computação, isto é, um sistema formal, que permita expressar sem ambiguidades as ações a serem executadas.

Com a introdução do conceito de modelo de computação na Teoria da Computabilidade permitiu-se que as noções de algoritmo e problema algoritmicamente solúvel fossem formalizadas matematicamente, devido à Tese de Church, que estabelece que todo problema solúvel pode ser resolvido usando qualquer modelo de computação conhecido.

2.1 Tipos de problemas

Primeiramente, um algoritmo pode ser entendido como uma estratégia (um método) para resolver um determinado problema, mostrando de forma estruturada (passo-a-passo) uma série de procedimentos e atividades. Segundo Papadimitriou [PAP 94], problemas não são somente tarefas a resolver, mas também objetos matemáticos.

2.1.1 Problema abstrato

Um problema é definido por Veloso [VEL 84] como uma terna p = (D, R, q), onde D é o domínio dos dados, R o domínio dos resultados e q uma relação de D em R, que define o problema. A solução de um problema p = (D, R, q) é uma função α : D $\rightarrow$ R tal que para todo d $\in$ D, tem-se que o par (d, α(d)) $\in$ q.

Não há um método de desenvolvimento algorítmico adequado para qualquer problema. Então há a necessidade de modificar o problema, reduzindo-o a outros problemas com ele relacionados ou decompondo-o em partes menores. Estas duas técnicas de resolução de um problema são apresentadas em Veloso [VEL 84].

2.1.2 Problema no ponto de vista computacional

Em Garey e Johnson [GAR 79] descreve-se, informalmente, um problema como:

1. uma descrição genérica de todos os seus parâmetros, e
2. uma declaração de quais propriedades a solução deve satisfazer.

Uma instância de um problema é obtida através da especificação de valores particulares para todos os parâmetros do problema. Fornecida como entrada, a descrição da instância de um problema pode ser vista como uma simples cadeia finita de símbolos escolhidos a partir de um alfabeto finito. Embora existam maneiras diferentes de descrever as entradas de um problema, considera-se uma em particular para cada problema, com um esquema de codificação fixo, o qual associa-se cada instância do problema à cadeia que a descreve.

O tamanho da entrada (comprimento) para uma instância I de um problema p é dada pelo número de símbolos na descrição de I obtida a partir do esquema de codificação do problema, sendo usado como uma medida formal do tamanho da instância. é em relação a este parâmetro que são definidas as medidas de complexidade.

Conforme [SZW 84], existem classes gerais de problemas algorítmicos: os *problemas de Decisão*, os *problemas de Localização* e os *problemas de Otimização*. Num problema de decisão, o objetivo é decidir a resposta sim ou não à uma questão. Em um problema de localização, procura-se localizar uma certa estrutura que satisfaça um conjunto de propriedades dadas. Se as propriedades envolverem critérios de otimização, então o problema é dito de otimização. Pode-se associar problemas de decisão, otimização e localização, de tal forma que um problema possa ser derivado do outro [SZW 84]. *Problemas de decisão Por convenção a teoria da complexidade restringe-se a problemas de decisão, já que o estudo de problemas NP-completos é aplicado somente para este tipo de problema [LEA 97]. Um problema de decisão Π consiste de um conjunto de instâncias D_Π e um subconjunto $Y_\Pi \subseteq D_\Pi$ de instâncias com resposta 'sim'.

Segundo [GAR 79], esta restrição a problemas de decisão é devida a uma linguagem definida como segue.

Para qualquer conjunto finito Σ de símbolos, denota-se por Σ^* o conjunto de todas as palavras finitas de símbolos de Σ, inclusive a palavra vazia. Um subconjunto L de Σ^* é chamada uma linguagem sobre o alfabeto Σ.

Assim, um problema pode ser definido como uma relação total sobre palavras do alfabeto $\Sigma = \{0, 1\}$, isto é, sobre seqüências finitas de 0's e 1's pertencentes ao conjunto Σ^* de todas as palavras sobre o alfabeto Σ.

Qualquer problema que se pretenda resolver computacionalmente pode ser descrito utilizando este formalismo [LEA 97], pois uma vez que a memória de um computador é entendida como uma série de palavras do alfabeto Σ, naturalmente pode-se representar instâncias e soluções do problema como palavras deste mesmo alfabeto. A correspondência entre problemas de decisão e linguagens é devido ao esquema de codificação usado para especificar as instâncias de um problema. Portanto o problema Π e seu respectivo esquema de codificação particiona Σ^* em três classes de palavras: aquelas que não são codificações de instâncias do problema Π; aquelas que codificam instâncias de Π para as quais a resposta é 'não' e, aquelas que codificam instâncias para as quais a resposta é 'sim'. Esta terceira classe de palavras é a linguagem que é associada ao problema de decisão Π.

A linguagem L associada ao problema de decisão Π é representada pelo conjunto:

$$L(\Pi) = \{x \in \Sigma^* \mid \Sigma \text{ é o alfabeto de codificação e x é a codificação de uma instância } I \in Y_\Pi\}$$

A representação de problemas de decisão como linguagens, possibilita a identificação de 'resolver' um problema de decisão como 'reconhecer' a linguagem correspondente. *Problemas de otimização Um problema de otimização combinatorial Π é definido em [GAR 79] como uma terna $\Pi = (D_\Pi, S_\Pi, m_\Pi)$, onde: (i) D_Π é o conjunto das instâncias; (ii) S_Π é a função que associa a cada instância $I \in D_\Pi$, um conjunto finito $S_\Pi(I)$ de candidatas a solução para I; e (iii) m_Π é uma função que atribui a cada instância $I \in D_\Pi$ e cada candidata $\Sigma \in S_\Pi(I)$, um número racional positivo $m_\Pi(I, \Sigma)$ chamado valor de solução de Σ.

Se Π é um problema de minimização (ou maximização), uma solução ótima para uma instância $I \in D_\Pi$ é uma candidata a solução $\Sigma^* \in S_\Pi(I)$ tal que, para todo $\Sigma \in S_\Pi(I)$, $m_\Pi(I, \Sigma^*) \leq m_\Pi(I, \Sigma)[m_\Pi(I, \Sigma^*) \geq m_\Pi(I, \Sigma)]$. O valor $m_\Pi(I, \Sigma^*)$ de uma solução ótima para I é denotado por $OPT_\Pi(I)$.

Os elementos básicos de um problema de otimização são os mesmos de um problema de decisão, acrescidos da função m cujo valor mede quão boa é uma solução admissível. O problema consiste em encontrar uma solução com uma medida máxima (no caso de um problema de maximização).

2.2 Conclusão

Neste capítulo foi mostrada a definição formal de problema abstrato segundo [VEL 84], e ainda foram apresentados os principais tipos de problemas encontrados na Teoria dos Problemas sob o ponto de vista computacional, que são os problemas de decisão e otimização.

Geralmente, os problemas de otimização são os mais complicados de serem resolvidos, visto que seus elementos constituintes são os mesmos dos problemas decisão acrescidos de uma função que mede a qualidade de uma solução admissível.

A importância deste estudo deve-se à necessidade do reconhecimento do tipo de problema que será resolvido pelo algoritmo, visto que precisa-se ter informações à respeito do problema para desenvolver o algoritmo. Em particular, um Algoritmo Genético possui uma função de avaliação (melhor explicada no capítulo 5) que deve ser escolhida ou modelada a partir deste conhecimento prévio do problema.

3 Teoria da Complexidade

Devido aos limites físicos dos recursos disponíveis do computador, não basta identificar um problema e garantir que exista pelo menos uma solução para este. é necessário, também, saber se pode ser resolvido efetivamente em tempo e espaço de memória finitos.

O termo *complexidade* refere-se, em geral, aos requerimentos de recursos necessários para que um algoritmo possa resolver um problema sob o ponto de vista computacional. Segundo [COO 83], a medida de complexidade de um algoritmo mais importante é medida de tempo, devido às pesquisas serem direcionadas para projetar e analisar algoritmos quanto à eficiência, fornecendo a solução de um problema com a rapidez desejada. O algoritmo mais rápido é o algoritmo mais eficiente. [GAR 79]

Requerimentos de tempo são freqüentemente considerados como fatores dominantes para determinar se um particular algoritmo é eficiente o bastante para ser útil na prática. Por isso, usa-se somente o termo complexidade.

3.1 Complexidade de tempo

Os requerimentos de tempo de um algoritmo são expressados em termos do tamanho de uma instância do problema. Conforme [GAR 79], para que a complexidade de tempo de um algoritmo seja definida de um modo preciso, o primeiro passo é definir o tamanho de uma instância levando em conta um esquema de codificação fixo que associe instâncias do problema com os *strings* que os descrevem.

O tamanho da instância I de um problema Π é o comprimento da entrada dessa instância, dado pelo número de símbolos na descrição de I, obtida de um esquema de codificação do problema Π. A complexidade de tempo, ou somente complexidade, para um algoritmo é uma função definida sobre os tamanhos das instâncias do problema que expressa o requerimento de tempo exigido por cada tamanho de entrada possível, dado pela maior quantidade de tempo necessária para o algoritmo resolver um problema desse tamanho.

3.2 Medidas de complexidade

Conforme Aho [AHO 74], a complexidade é chamada *complexidade no pior caso*, se para um dado tamanho de instância a complexidade tomada é máxima para qualquer entrada desse mesmo tamanho. Entretanto, em muitos casos, a complexidade de um algoritmo não depende só do tamanho da entrada, mas de certas propriedades de entrada, como o grau de ordem (ou desordem). Neste caso, convém considerar a probabilidade da ocorrência de cada entrada possí vel, determinando a *complexidade do caso médio*. Embora a complexidade no caso médio seja mais significativa do que a complexidade no pior caso, a sua determinação é mais difícil, pois precisa-se fazer algumas hipóteses sobre a distribuição das entradas, e hipóteses realísticas muitas vezes não são matematicamente tratáveis [LEA 97, ROS 97]. As complexidades no pior caso e no caso médio são consideradas as principais medidas de complexidade de tempo de um algoritmo.

A Teoria da Complexidade pode ser considerada como um refinamento da

Teoria da Computabilidade [BOV 94]. Como na Teoria da Computabilidade, onde o conceito de modelo de computação levou aos conceitos de algoritmo e problema algoritmicamente resolvível, de modo similar, na Complexidade Computacional o conceito de recurso usado por uma computação levou aos conceitos de algoritmo eficiente e de problema computacionalmente admissível, conforme [LEA 97].

3.2.1 Medidas de complexidade para algoritmos randômicos

A Randomização foi formalmente introduzida por Rabin [RAB 76] e independentemente por Solovay e Strassen [SOL 77] como uma ferramenta para melhorar a eficiência de certos algoritmos. Um algoritmo randômico utiliza sorteios (em inglês: *coin-flips*) para tomar decisões em diferentes passos do algoritmo. Portanto, um algoritmo randômico é, atualmente, uma família de algoritmos onde cada membro desta família corresponde a uma seqüência de resultados provenientes dos sorteios. Duas das formas mais comuns de randomização na literatura são os algoritmos de *Las Vegas* e *Monte Carlo*.

O método *Las Vegas* garante que a saída (resultado) do algoritmo é sempre correta. Entretanto, somente parte, não mais que a metade, da família dos algoritmos terminam em um certo limite de tempo. Por outro lado, o procedimento de *Monte Carlo* sempre termina em um período pré-determinado; entretanto, o resultado final é correto com uma certa probabilidade (tipicamente mais que a metade).

Segundo Reif [REI 89], para um algoritmo geral que produz mais que somente uma saída *sim-não*, o significado preciso de uma saída incorreta torna-se subjetiva; por exemplo, como saber o quão perto se está de uma saída correta para decidir se o resultado é aceitável. Embora, sendo esta afirmativa, uma das razões para usar algoritmos do tipo *Las Vegas*, o uso destes algoritmos depende de uma aplicação particular.

Inicialmente, é necessário enfatizar as principais diferenças entre um algoritmo randômico e um algoritmo probabilístico. Algoritmos probabilísticos são aqueles algoritmo cujo desempenho depende da distribuição da entrada. Para estes algoritmos, interessa, freqüentemente, a média dos recursos usados por todas as entradas (assumindo uma entrada com uma probabilidade de distribuição fixa). Um algoritmo randômico não necessariamente depende da distribuição da entrada. Este algoritmo usa uma certa quantia dos recursos para entrada do pior caso com a probabilidade 1 - ε ($0 < \varepsilon < 1$), isto é, o limite vale para qualquer entrada (este limite é mais rígido do que os limites médios). Isto pode ser melhor entendido com o exemplo de Hoare, o algoritmo *Quicksort*. No seu formato original, ele é um algoritmo probabilístico que executa muito bem para certas entradas e claramente deteriorado para algumas outras entradas. Assumindo que todas as entradas são igualmente prováveis (conhecida como hipótese da entrada aleatória), o algoritmo executa muito bem na média. Com a introdução da randomização no algoritmo, foi mostrado [REI 89] que o algoritmo executa muito bem em *todas* as entradas com alta probabilidade. é claro, o ônus de uma execução bem sucedida de um algoritmo é, agora, responsabilidade de um resultado de um sorteio. Isto depende de certas propriedades do acaso (em inglês: *randomness properties*) de um gerador de números aleatórios.

Até agora, caracterizou-se os algoritmos randômicos com uma probabilidade de sucesso de 1 - ε sem especificar as possíveis formas de ε. Ele pode ser uma constante ou uma função (que assume valores entre 0 e 1). Deve-se ter certeza que ε seja minimizado (algoritmos determinísticos tem $\varepsilon = 0$). Intuitivamente pode-

se esperar uma relação entre ε e a quantidade de recursos utilizados. Em outras palavras, a probabilidade de falha ε deve decrescer com o aumento dos recursos. Considere o seguinte exemplo. Suponha $T_A(n)$ o tempo *esperado* de execução de um algoritmo randômico A para uma entrada de tamanho n. O que se pode dizer sobre ε? Se não houver alguma expectativa de limite pode-se usar a *Desigualdade de Markov*. Por esta desigualdade, a probabilidade que o tempo de execução exceda $k \cdot T_A(n)$ é menor que $\frac{1}{k}$. Por exemplo, se k = 2, ou seja, a probabilidade do tempo de execução do algoritmo A para uma entrada de tamanho n excede o dobro da média, então $\varepsilon = \frac{1}{2}$. Ao comparar, para o mesmo problema, este algoritmo com outro algoritmo B cuja execução exceda $k \cdot \alpha \cdot T_B(n)$, com probabilidade menor que $\frac{1}{n^\alpha}$, e supondo que para um certo α, k é uma constante independente de n, verifica-se facilmente que estas situações implicam na rápida diminuição da probabilidade de falha, à medida que n cresce.

Caracterizou-se a probabilidade de falha ε como uma função decrescente do tamanho do problema (isto é, n) e dos recursos utilizados pelo algoritmo. Reconhece-se que quanto mais rápido diminui-se ε, utilizando os parâmentros citados, melhor é o algoritmo. Isto faz que o algoritmo B seja melhor que A se $T_A(n)$ e $T_B(n)$ representam a mesma função, isto é, se ambos são algoritmos para o mesmo tipo de problema, que cuja probabilidade de sucesso é dada por ε's iguais. Em termos de probabilidades, pode-se afimar que os algoritmos possuem uma mesma distribuição. A idéia básica é que dependendo da aplicação, dado um valor para n, o usuário escolhe um certo valor para ε e respectivamente escolhe k com o objetivo de minimizar k. Não existem motivos para saber o tipo da função ε, exceto que a probabilidade de falha da segunda forma (do algoritmo B) tem sido muito utilizada na literatura e em algoritmos que têm alta probabilidade de sucesso. Este tipo de função de probabilidade de falha (do inglês: *failure probability function*) é bastante robusta com respeito ao número polinomial de procedimentos, isto é, a união de um número polinomial de eventos, cada um com alta probabilidade de sucesso, têm êxito com alta probabilidade. Isto pode ser uma tarefa não trivial de transformar um algoritmo como A em um algoritmo como B. Presume-se que algoritmos randômicos como B, que tenham alta probabilidade de sucesso, sejam competitivos em relação à algoritmos determinísticos para o mesmo problema. Conforme Adleman e Manders [ADL 77], um algoritmo randômico com probabilidade de sucesso maior que $1 - 2^k$ (para um k fixo e muito grande) tem menor probabilidade de falha do que o *hardware* no qual esté sendo executado.

3.3 Classes de complexidade

Nesta seção são definidas as principais classes de complexidade para problemas, relativamente ao desempenho de seus algoritmos com respeito a complexidade de tempo e, a introdução de questões relacionadas à tratabilidade de problemas. Todas as definições são de Garey & Johnson [GAR 79].

3.3.1 Classe P

A classe P é definida como o conjunto de todos os problemas resolvíveis por um algoritmo determinístico em tempo polinomial. Ela contém todos os problemas *simples*, aqueles que são computacionalmente tratáveis. No entanto, um problema

tratável não é necessariamente *eficiente*. Devido a classificação de um algoritmo em relação à complexidade de tempo apresentada anteriormente não ser mais do que uma definição de um limite superior para a ordem de complexidade, é possível que hajam exceções, no sentido de que existam alguns algoritmos classificados como exponenciais que para alguma instância apresentam esse tipo de complexidade (mas, no caso médio têm uma complexidade polinomial com pequeno grau), enquanto que outros, classificados como polinomiais, tenham um grau tão alto que se tornam ineficientes. Mas, muito embora nem todo o problema pertencente a classe P possa ser considerado eficiente, o fato de não estar em P, caracteriza o problema certamente como intratável, com raríssimas exceções [LEA 97]. Outra razão que justifica a ampla aceitação da definição da classe P em termos de complexidade polinomial, é a equivalência entre modelos de computação realísticos com respeito ao tempo polinomial. Sempre é possível transformar um algoritmo de tempo polinomial obtido em determinado modelo computacional num outro algoritmo também de tempo polinomial aplicável num modelo de computação básico [LEA 97].

Problemas de decisão são formalmente representados como linguagens, de tal forma que a solução de um problema de decisão equivale ao reconhecimento da linguagem correspondente por uma máquina de Turing, estudada em [GAR 79, LEA 97]. Em geral, diz-se que um programa MTD M com um alfabeto de entrada Σ aceita $x \in \Sigma^*$ se, e somente se, M pára quando aplicado à entrada x. A linguagem L_M reconhecida pelo programa M é dada por

$$L_M = \{ x \in \Sigma^* \mid \text{M aceita x}\}$$

A classe P é também definida formalmente através de uma máquina de Turing determinística (MTD). A definição formal de um algoritmo de tempo polinomial é um programa MTD em tempo polinomial, mostrado a seguir.

Seja M um programa MTD que pára, para todas as entradas $x \in \Sigma^*$ e $T_M : \mathbb{N} \rightarrow \mathbb{N}$, sua complexidade de tempo, definida por:

$$T_M(n) = \max \{ m \in \mathbb{N} \mid \exists\, x \in \Sigma^*, \text{ com } |x| = n \text{ e o número de transições de M sobre x é m}\}$$

O programa M é chamado de *programa MTD de tempo polinomial* se existe um polinômial p tal que, $\forall\, n \in \mathbb{N}$,

$$T_M(n) \leq p(n).$$

A classe P é então uma classe de linguagens definida formalmente por:

$$P = \{ L \mid \text{existe um programa MTD M de tempo polinomial para o qual } L = L_M\}$$

Se existe um método com base num modelo determinístico para resolver um dado problema de decisão Π com gasto polinomial de tempo, ou seja, uma máquina determinística reconhece a linguagem $L_\Pi = \{ x \in \Sigma^* \mid x \in Y_\Pi\}$ correspondente a resposta 'sim' para o problema em tempo polinomial, então também o problema complementar que corresponde a resposta 'não', pode ser reconhecida com o mesmo limite polinomial de tempo. De fato, se a máquina determinística pára, para qualquer que seja a entrada, pode-se fazer o reconhecimento simplesmente trocando os estados finais correspondendo às respostas 'sim' e 'não'. Se a classe de problemas complementares for representada por Co-P, verifica-se facilmente que P = Co-P.

3.3.2 Classe NP

A classe NP é definida como o conjunto de todos os problemas de decisão resolvíveis por um algoritmo não-determinístico em tempo polinomial. Um algoritmo não-determinístico é dito de tempo polinomial quando é possível checar se uma dada entrada satisfaz o problema em tempo polinomial. O não-determinismo permite várias computações sobre uma dada entrada, uma para cada conjetura. A classe NP pode ser vista, informalmente, como a classe dos problemas de decisão para os quais a verificação de uma solução estimada para uma dada entrada satisfaz todos os requerimentos do problema.

A definição formal da classe NP é dada em termos de uma máquina de Turing não-determinística (MTND). Seja M um programa MTND. O tempo requerido por M para aceitar a cadeia x $\in$ L $_M$ é definido como o tempo mínimo, sobre todas as computações de M que aceitam x, do número de transições que ocorre nos estágios de conjetura e checagem, até que seja atingido o estado de parada q.

A função de complexidade de tempo $T_M : \mathbb{N} \rightarrow \mathbb{N}$ é definida por:

$$T_M(\ n) = \max\ (\ \{\ 1\} \bigcup \{\ m \mid \quad \exists\ x \in L_M, \text{ com } |x| = n \text{ tal que}$$

$$M \text{ aceita x no tempo m}\})$$

Esta função depende somente do número de transições que ocorre nas computações aceitas. Por convenção, define-se $T_M(\ n)$ igual a 1 sempre que nenhuma entrada de comprimento n é aceita por M. O programa M é chamado de *programa MTND de tempo polinomial* se existe um polinômio p tal que $T_M(\ n) \leq p(\ n)$, para $n \geq 1$.

Finalmente, a classe NP é formalmente definida por:

$$NP = \{\ L \mid \text{existe um programa MTND de tempo polinomial M tal que } L_M = L\ \}$$

Considerando os problemas complementares à classe NP como aqueles que exigem uma justificativa à resposta 'não' com estágio de reconhecimento correspondendo a um algoritmo polinomial no tamanho da entrada do problema, define-se a classe Co-NP. é interessante observar que, contrariamente ao que ocorre com a classe P que coincide com a sua classe complementar Co-P, a classe Co-NP contém problemas que se desconhece a pertinência ou não à classe NP devido às questões (P = NP? NP = Co-NP? P = NP $\bigcap$ Co-NP?) estudadas em [GAR 79, LEA 97].

3.3.3 Relação entre P e NP

Primeiramente, P $\subseteq$ NP, o que significa que todo o problema de decisão resolvível por um algoritmo determinístico de tempo polinomial é também resolvível por um algoritmo não-determinístico de tempo polinomial. De fato, se $\Pi \in$ P e α é qualquer algoritmo determinístico de tempo polinomial para Π, pode-se obter um algoritmo não-determinístico em tempo polinomial para Π, simplesmente usando α como um algoritmo de reconhecimento para uma justificativa à resposta 'sim' de Π. Portanto, $\Pi \in$ P $\Rightarrow \Pi \in$ NP. Entretanto, a questão se P $\subseteq$ NP, ou se P = NP, continua em aberto.

Conforme [GAR 79], considera-se que algoritmos não-determinísticos de tempo polinomial parecem ser mais potentes que aqueles determinísticos de tempo polinomial, mas não se conhece nenhum método geral para converter não-determinísticos em determinísticos. Segundo [PAP 94], a redutibilidade dá uma noção precisa do

que significa um problema ser pelo menos tão difícil quanto outro. Daí porque a noção de redutibilidade entre problemas tornou-se tão importante.

Um tipo de redução, introduzida por [GAR 79], chamada de *transformação polinomial* é definida formalmente como segue.

Uma transformação polinomial de uma linguagem $L_1 \subseteq \Sigma^*$ para uma linguagem $L_2 \subseteq \Sigma^*$, é uma função $f : \Sigma^* \rightarrow \Sigma^*$ que satisfaz as duas condições seguintes:

- existe um programa para MTD de complexidade polinomial que computa f;
- para todo $x \in \Sigma^*$, $x \in L_1$ se e somente se $f(x) \in L_2$.

Se existe uma transformação polinomial de L_1 para L_2, escreve-se simbolicamente

$$L_1 \propto L_2,$$

e lê-se, L_1 é polinomialmente trasnformável em L_2.

Pode-se também estender a definição polinomial a nível de problemas de decisão: se Π_1 e Π_2 são problemas de decisão, escreve-se

$$\Pi_1 \propto \Pi_2,$$

sempre que existir uma transformação polinomial de $L(\Pi_1)$ para $L(\Pi_2)$. Uma transformação polinomial satisfaz as seguintes propriedades:

- a classe P é fechada em relação à transformação polinomial: se $L_1 \propto L_2$, então $L_2 \in P$ implica que $L_1 \in P$;
- a relação $\propto$ é transitiva: se $L_1 \propto L_2$ e $L_2 \propto L_3$, então $L_1 \propto L_3$;
- duas linguagens L_1 e L_2 são identificadas como polinomialmente equivalentes, sempre que $L_1 \propto L_2$ e $L_2 \propto L_1$.

As duas últimas propriedades caracterizam a relação $\propto$ como uma relação de equivalência. Além disso, a relação $\propto$ impõe uma ordenação parcial sobre essas classes de equivalência. Observa-se que a classe P é identificada como a menor classe de equivalência sob esta ordenação parcial e, portanto, pode ser vista como a classe que contém as linguagens (problemas de decisão) consideradas mais *fáceis* sob o ponto de vista computacional. Uma linguagem L pertencente a uma classe de complexidade C é dita C-completo se, qualquer linguagem L' em C pode ser reduzida a L.

3.3.4 Classe NP-Completo

Formalmente, uma linguagem L é definida como NP-Completa se são válidas as condições:

- $L \in NP$; e
- para toda outra linguagem $L' \in NP$ tem-se que $L' \propto L$.

Informalmente, um problema de decisão Π é NP-Completo se $\Pi \in$ NP e, para todo problema de decisão $\Pi' \in$ NP, tem-se que $\Pi' \propto \Pi$. Portanto, os problemas NP-Completos são identificados como os problemas mais difíceis em NP. Assim, se qualquer problema NP-Completo pode ser resolvido em tempo polinomial, então todos os problemas em NP podem ser resolvidos da mesma forma. A partir disso, pode-ser concluir que se um problema de decisão Π é NP-Completo então, sob a conjetura P $\neq$ NP, tem-se que $\Pi \in$ NP - P.

O primeiro problema a ser identificado como NP-Completo foi o problema da Satisfabilidade, estudado em [GAR 79]. Este resultado, conhecido como o Teorema de Cook, foi demonstrado em 1971 e se constitui num dos resultados fundamentais dentro da Teoria da Complexidade Computacional. Em 72, Karp provou que outros 21 problemas também eram NP-Completo, e, a partir daí estabeleceram-se muito mais problemas, por exemplo, cerca de trezentos problemas são apresentados por [GAR 79].

Problema da Satisfabilidade - SAT

Instância: Conjunto de literais U = { u_1, $\overline{u}_1$, ..., u_n, $\overline{u}_n$ } e conjunto de cláusulas C = { c_1, ..., c_m}, onde cada c_i, $1 \leq i \leq m$, é constituído por literais de U, na forma normal conjuntiva.

Resposta: 'Sim', se e somente se existe uma atribuição de valores lógicos a cada um dos literais em U tal que todas as cláusulas de C são satisfeitas (são verdadeiras); 'Não', caso contrário.

A demonstração do Teorema de Cook é apresentada integralmente em [GAR 79]. Em linha gerais, segue o seguinte raciocínio. Existem 2^n atribuições de valores lógicos aos literais, e, até o momento, não foi possível encontrar um algoritmo que consiga determinar em tempo não exponencial se, pelo menos uma atribuição satisfaz todas as cláusulas. Em posse de uma atribuição pode-se verificar em tempo polinomial se essa atribuição satisfaz todas as cláusulas, sendo portanto SAT pertencente à classe NP. Resta provar que SAT é completo para NP para transformações polinomiais, ou seja, para qualquer linguagem L em NP, tem-se L $\propto L_{SAT}$. Uma linguagem L que pertence a NP pode ser descrita através de uma MTND polinomial no tempo que a reconhece, então mostra-se como se pode construir uma MTD a partir da MTND em tempo polinomial.

3.3.5 Classe intermediário

A figura 3.1 mostra um simples diagrama da classe NP. Esta classe contém a subclasse P e a subclasse (aqui abreviada NPC) composta das linguagens NP-Completo. Assumindo que P $\neq$ NP, estas duas subclasses não se interceptam.

Além disso, se P $\neq$ NP, então NP não pode ser o mesmo que P $\bigcup$ NPC. Denota-se a classe NP - (P $\bigcup$ NPC) por NPI, que consiste das linguagens que possuem dificuldade *intermediária* (entre P e NPC). O fato que NPI não é vazio se P $\neq$ NP decorre como uma conseqüência de um resultado mais geral, provado em [LAD 75]. Para o propósito de expor este resultado, faz-se necessário saber que uma linguagem recursiva é qualquer linguagem que pode ser reconhecida (não

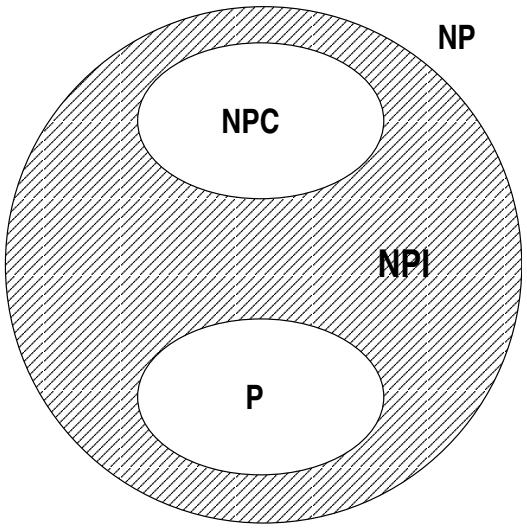

Figura 3.1 – O mundo dos NP (assumindo P $\neq$ NP)

necessariamente em tempo polinomial) por um programa DTM (determinístico em uma Máquina de Turing) que termina para todas as entradas.

Teorema. Seja B uma linguagem recursiva tal que B $\notin$ P. Então existe uma linguagem reconhecível em tempo polinomial D $\in$ P tal que a linguagem A=D$\bigcap$B não pertence a P, A $\propto$ B, e ainda não é o caso que B $\propto$ A.

Aplica-se este teorema como segue: seja B qualquer linguagem NP-Completo. Se P $\neq$ NP, então B $\notin$ P, portanto a hipótese do teorema é satisfeita. Sejam D e A = D $\bigcap$ B dados pelo teorema. A linguagem resultante A pertence a NP porque D $\in$ P e B $\in$ NP. O teorema então implica que não se cumpre B $\propto$ A, portanto A não pode ser NP-Completo, mas como A $\notin$ P, segue que A $\in$ NPI.

Ainda, pode-se concluir pelo teorema que NPI não pode ser vazio se P $\neq$ NP. O teorema também diz sobre a estrutura desta classe intermediária, pois, assumindo P $\neq$ NP, a classe NPI é constituída de uma coleção infinita de classes de linguagens equivalentes distintas. Se B é um problema NPI, o teorema permitirá construir um novo problema que é *mais fácil* que B mas ainda não em P. [LAD 75] também mostra que, se P $\neq$ NP, então NPI deve conter pares de linguagens C e D tal que, nem C $\propto$ D e nem D $\propto$ C. Portanto, na ordem parcial com respeito a *dificuldade* imposta pela relação (em NP), deverão existir elementos incomparáveis se P $\neq$ NP.

De acordo com esta apresentação teórica, é razoável perguntar se existem problemas considerados *naturais* candidatos a serem membros da classe NPI. Sobre o pressuposto de que P $\neq$ NP, o teorema pode ser usado para exibir membros desta classe, mas estas linguagens são altamente *não-naturais*, devido às complexas técnicas de diagonalização usadas para construí-las. Claro, algum problema em *aberto* em NP, isto é, um problema que ainda não foi provado pertencer a P ou a NP-Completo, pode ser visto como candidato para NPI. Entretanto, certos problemas em aberto são geralmente vistos como os melhores candidatos, tendo resistido ao *teste do tempo* e tendo certos atributos que parecem distinguí-los dos tipos de problemas que já se sabe pertencerem a classe dos NP-Completo.

3.3.6 Classe NP-Difícil

A classe NP-Completo é definida como a classe que contém os problemas mais difíceis em NP. As técnicas conhecidas para provar a NP-Completude podem também ser usadas para provar a dificuldade de problemas não restritos à classe NP. Qualquer problema de decisão Π, pertencente ou não à classe NP, que seja redutível a um problema NP-Completo, terá a propriedade de não ser resolvido em tempo polinomial, a menos que P = NP. Assim, tal problema Π é chamado NP-Difícil, já que ele apresenta dificuldade não menor que qualquer problema NP-Completo, definindo uma nova classe de problemas chamada classe NP-Difícil.

Esta noção de NP-Dificuldade pode ser generalizada, de maneira a abranger outros problemas, não somente para problemas de decisão. Isto é possível a partir da generalização da noção de uma transformação polinomial, em termos de máquina de Turing.

Um problema de busca $\Pi = (D_\Pi, S_\Pi)$ consiste de um conjunto D_Π de objetos finitos chamados instâncias e, para cada instância $I \in D_\Pi$, um conjunto $S_\Pi(I)$ de objetos finitos chamados de solução para I.

Um algoritmo resolve um problema de busca Π se, dada como entrada qualquer instância de $I \in D_\Pi$, o algoritmo retorna a resposta 'não' sempre que $S_\Pi(I)$ é vazio e, caso contrário, retorna alguma solução $s \in S_\Pi(I)$. Observa-se que qualquer problema de decisão Π pode ser formulado como um problema de busca. Basta definir que:

$$S_\Pi(I) = \begin{cases} \text{'sim'} & \text{se } I \in Y_\Pi, \text{ e} \\ \text{'não'} & \text{se } I \notin Y_\Pi \end{cases}$$

Desta maneira, um problema de decisão pode ser considerado simplesmente como um tipo especial de problema de busca.

3.4 Conclusão

Neste capítulo foi apresentado um estudo abrangente da Teoria da Complexidade, definindo-se o termo complexidade como os requerimentos de recursos (de tempo, em específico) para que um algoritmo possa resolver computacionalmente um problema.

Ainda, foram mostradas as principais medidas de complexidade, como: complexidade no *pior caso*, no *melhor caso* e no *caso médio*; e, medidas de complexidade para algoritmos randômicos.

Além disso, foram descritas as principais classes de complexidade, mostradas em [GAR 79], e suas relações ao desempenho de seus algoritmos com respeito a complexidade de tempo. O estudo se deteve à classe NP e co-relacionadas (NP-Completo, NPI e NP-Difícil) principalmente por introduzirem a questão de intratabilidade de problemas.

Segundo Cook, para um problema ser denominado intratável basta ser NP-Completo. Foi apresentada a definição de problemas NP-Completos, porque Algoritmos Genéticos têm se mostrado uma maneira eficaz de tratar este tipo de problema [LAG 96] [AGU 96]. O conceito de complexidade média também é necessário porque os Algoritmos Genéticos, tendo seu desempenho dependente de elementos probabilísticos (taxas de reprodução), têm a complexidade média bem mais significativa que outras medidas.

4 Algoritmos Aproximativos

Exitem muitos problemas NP-Completos nas áreas de Otimização, Teoria dos Grafos e de Pesquisa Operacional. Os algoritmos conhecidos que os resolvem são portanto de complexidade não polinomial, o que torna-os impraticáveis para instâncias muito grandes [TOS 86, AGU 97a]. Baseados no fato de que muitas aplicações que requerem soluções para esses problemas não exigem uma solução exata, desenvolve-se algoritmos e métodos para solução destes tipos de problemas de maneira aproximada. Esse algoritmos são denominados Algoritmos Aproximativos.

Segundo [WEI 77], as duas principais classes de algoritmos aproximativos são: uma que garante sempre uma solução próxima da solução procurada e outra que produz uma solução ótima quase sempre. Essa segunda é a classe dos algoritmos probabilísticos, de grande utilidade em cripto-análise [RAB 76].

Segundo [LEA 97], existem duas categorias adotadas como alternativas para o tratamento de problemas NP-Completos. A primeira categoria consiste daquelas abordagens que buscam aperfeiçoar os algoritmos com complexidade de tempo exponencial, reduzindo o esforço da busca exaustiva. Entre as mais usadas estão aquelas baseadas em técnicas de *branch-and-bound*, estudadas em [AGU 96], e enumeração implícita. Estas técnicas geram 'soluções parciais' dentro de um formato de busca estruturada em árvore, utilizando poderosos métodos direcionais para reconhecer soluções parciais que não podem, possivelmente, ser estendidos para soluções reais e, deste modo, eliminar ramos inteiros de busca numa simples etapa. Estas abordagens são incrementadas com a programação dinâmica, que tenta organizar a busca em conjunto com *branch-and-bound*. A segunda categoria consiste daquelas abordagens para problemas NP-Completos que são pertinentes unicamente a problemas de otimização. Assim, para produzir um algoritmo de complexidade aceitável para um problema NP-Difícil, relaxa-se o significado de 'resolver', de tal modo que uma solução viável com valor próximo da solução ótima seja aceitável. Algoritmos que produzem soluções aproximadas para problemas de otimização são chamados de *algoritmos aproximativos*.

Sendo heurísticos por natureza, os algoritmos aproximativos representam uma forte dependência em relação ao particular problema a ser resolvido.

Uma técnica muito utilizada para a obtenção de algoritmos aproximativos, é a *busca de vizinhança*, no qual um conjunto pré-selecionado de operações locais é usado para iterativamente aperfeiçoar uma solução inicial, até que nenhum progresso local possa ser feito e uma solução local ótima tenha sido alcançada.

Segundo [LEA 97], apesar do sucesso comprovado na prática, para que o algoritmo apresente um desempenho satisfatório é necessário uma quantidade considerável de ajustes. Por esse motivo, raramente é possível fazer uma análise formal *a priori* do desempenho do algoritmo. Ao invés disso, esses algoritmos são avaliados e comparados através de uma combinação de estudos empíricos e argumentos de senso-comum. Entretanto, em alguns casos pode-se provar que as soluções encontradas pelos algoritmos aproximativos nunca diferem da solução ótima mais que algum valor estipulado, providenciando desta maneira, uma qualidade garantida para o algoritmo, *a priori*.

4.1 Definições

As definições apresentadas a seguir são baseadas nos trabalhos de Horowitz & Sahni [HOR 78] e Garey & Johnson [GAR 79].

4.1.1 Algoritmos aproximativos

Seja $\Pi = (\ D_\Pi,\ S_\Pi,\ m_\Pi)$ um problema de otimização e $I \in D_\Pi$ uma instância de Π. Considere $OPT(\ I)$ o valor de uma solução ótima para I. Um algoritmo a é um algoritmo aproximativo para Π se, dada qualquer instância $I \in D_\Pi$, a encontra uma candidata a solução $\Sigma \in S_\Pi$. O valor $m_\Pi(\ I)$ da solução candidata encontrada pelo algoritmo a será denotada por $a(\ I)$.

4.1.2 Medidas de qualidade de algoritmos aproximativos

Para quantificar a proximidade entre as soluções produzidas por algoritmos aproximativos e a solução exata são definidas medidas de qualidade do algoritmo. *Razão de qualidade A razão de qualidade de um algoritmo a para uma entrada I de um problema de minimização é definida por:

$$R_a(\ I) = \frac{a(\ I)}{OPT(\ I)}$$

No caso da minimização, tem-se que $1 \leq R_a(\ I) \leq \infty$, indicando melhor qualidade quanto mais próximo de 1. *Qualidade absoluta A qualidade absoluta do algoritmo a é definida por:

$$R_a = \inf \{\ r \geq 1\ |R_a(\ I) \leq r \text{ para toda instância } I \in D_\Pi\}$$

*Qualidade assintótica A qualidade assintótica do algoritmo a é dada por:

$$R_a^\infty = \inf \{\ r \geq 1\ |(\ \exists\ n \in Z^+)(\ \forall\ I \in D_\Pi \text{ satisfazendo } OPT_\Pi(\ I) \geq n)\ R_a(\ I) \leq r\}$$

Nota-se que as medidas de qualidade para problemas de maximização são definidas pelas recíprocas das medidas definidas acima.

Categorias de algoritmos aproximativos

São definidas dependendo das condições estabelecidas *a priori*. Um algoritmo aproximativo a para um problema de otimização Π é dito: *Algoritmo de aproximação absoluta Se para cada instância I de Π, tem-se que:

$$|OPT_\Pi(\ I) - a(\ I)\ | \leq k, \text{ para alguma constante k.}$$

*Algoritmo f(n)-aproximativo Se para cada instância I de tamanho n, tem-se que:

$$\frac{|OPT_\Pi(\ I) - a(\ I)|}{OPT_\Pi(\ I)} \leq f(\ n), \text{ supondo-se } OPT_\Pi(\ I) > 0.$$

*Algoritmo ε-aproximativo Se é um algoritmo f(n)-aproximativo para o qual se tem

$$f(\ n) \leq \varepsilon, \text{ para alguma constante } \varepsilon.$$

Observa-se que num problema de maximização

$$\frac{|\mathrm{OPT}_{\Pi}(\ \mathrm{I}) - a(\ \mathrm{I})|}{\mathrm{OPT}_{\Pi}(\ \mathrm{I})} \leq 1, \text{ para cada solução possível para I.}$$

Portanto, para problemas de maximização deve-se requerer que $\varepsilon < 1$ para que o algoritmo seja ε-aproximativo.

Um problema de otimização é dito *aproximável*, se ele admite um algoritmo ε-aproximativo para um dado ε. O tipo de algoritmo aproximativo ideal é aquele de aproximação absoluta, conforme [LEA 97]. Mas, para a maioria dos problemas NP-Difíceis, pode-se mostrar que algoritmos rápidos deste tipo existem somente se P = NP. Esta constatação é verdadeira para certos problemas NP-Difíceis em relação à existência de algoritmos f(n)-aproximativos. Embora algoritmos ε-aproximativos sejam conhecidos para muitos problemas de escalonamento, por exemplo, existem problemas ε-aproximativos que são também NP-Difíceis.

4.1.3 Esquemas aproximativos

Muitas aplicações exigem uma qualidade mínima para aceitação de um algoritmo aproximativo, como requerimento de exatidão dado por um número racional positivo ε. O comportamento aproximativo exibido por tais algoritmos é caracterizado através de um esquema aproximativo. *Esquema aproximativo Um esquema aproximativo para um problema de otimização Π é um algoritmo A tal que, dado um requerimento de exatidão ε, produz um algoritmo ε-aproximativo A_{ε}.

Isto significa que, fixado ε, o algoritmo A_{ε} gera uma solução admissível $\mathrm{A}_{\varepsilon}(\ \mathrm{I})$, satisfazendo

$$\frac{|\mathrm{OPT}(\ \mathrm{I}) - \mathrm{A}_{\varepsilon}(\ \mathrm{I})|}{\mathrm{OPT}(\ \mathrm{I})} \leq \varepsilon, \text{ para cada instância I.}$$

O termo esquema é usado para um algoritmo A porque A gera uma série de algoritmos aproximativos para Π, um para cada $\varepsilon \geq 0$. *Esquema aproximativo polinomial Um esquema aproximativo é um esquema aproximativo polinomial se e somente se para cada $\varepsilon > 0$ fixo, seu tempo de computação é polinomial no tamanho do problema. *Esquema aproximativo totalmente polinomial é um esquema polinomial cujo tempo de computação é polinomial tanto no tamanho do problema como também no inverso de ε.

Observa-se que a definição de um esquema aproximativo polinomial não limita a complexidade de tempo como polinomial com respeito a $\frac{1}{\varepsilon}$. Portanto, computações com valores de ε muito pequenos podem ser praticamente impossíveis. Isto significa que esquemas aproximativos totalmente polinomiais são ideais para resolver problemas de otimização NP-Difíceis.

4.2 Classes de problemas de otimização

Bovet, [BOV 94], identifica cinco classes de problemas de otimização, mantendo uma analogia com as classes de problemas de decisão. E estão relacionadas conforme a figura 4.1 a seguir.

Classe NPO

Consiste de todos os problemas de otimização cujas linguagens correspondentes estão na classe NP.

Classe APX

é formada por todos os problemas em NPO que admitem esquema aproximativo, isto é, aqueles problemas de otimização que possuem algoritmos aproximativos satisfazendo um requerimento de exatidão ε fixado.

Classe PAS

Consiste de todos os problemas em NPO que admitem esquema aproximativo polinomial. Isto significa que os problemas desta classe podem ser aproximados por algoritmos tendo como entrada uma instância I e um requerimento de exatidão ε, com complexidade de tempo polinomial no tamanho da instância, para cada ε fixado.

Classe FPAS

Estão todos os problemas em NPO que admitem esquema aproximativo totalmente polinomial. Portanto, a classe FPAS contém todos os problemas de otimização que podem ser aproximados por algoritmos tendo como entrada uma instância I e um requerimento de exatidão ε e cuja complexidade de tempo é polinomial tanto no tamanho de I quanto em $\frac{1}{\varepsilon}$.

Classe PO

Consiste de todos os problemas em NPO que são resolvíveis em tempo polinomial.

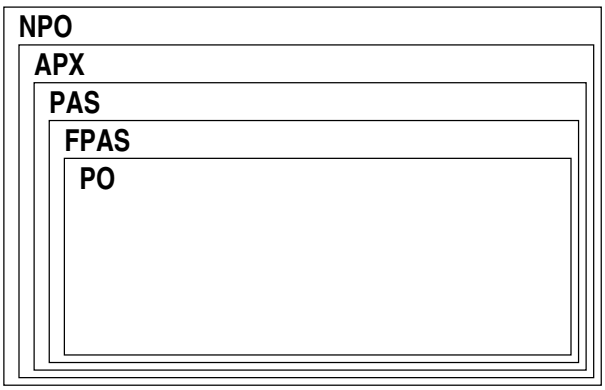

Figura 4.1 – Classes de Problemas de Otimização

4.3 Redutibilidade

Embora todos os problemas NP-Completos conhecidos sejam redutíveis a outro problema, suas correspondentes versões de otimização apresentam propriedades diversificadas em relação à aproximabilidade [LEA 97]. Dados dois problemas NP-Completos Π_1 e Π_2, sabe-se que Π_1 pode ser transformado polinomialmente em Π_2 segundo figura 4.2 a seguir e, se A_2 é um algoritmo que resolve Π_2 também resolve Π_1 [TOS 86]. Assim, se Π_2 tem algoritmo polinomial que o resolva, Π_1 também

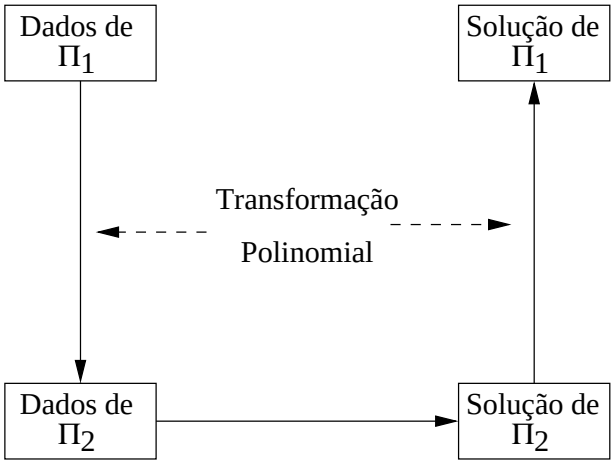

Figura 4.2 – Transformação Polinomial

tem. Logo, é de esperar que, se Π_2 tem um algoritmo aproximativo A_2 de tempo polinomial, o mesmo método gere um algoritmo polinomial A_1, aproximativo, para o problema Π_1. Mas conforme mostrado em [GAR 79], isto não acontece, uma transformação polinomial que preserva a otimalidade de uma solução, não necessariamente preserva a qualidade de uma solução aproximada. Bovet [BOV 94] mostra um tipo de redutibilidade cuja aproximação é preservada, de maneira que se um problema Π_1 se reduz a um problema Π_2 e Π_2 pertence à classe APX, então Π_1 também pertence a esta classe. Para definir uma redução apropriada entre problemas de otimização, não é suficiente mapear instâncias de um problema nas instâncias do outro problema. Necessita-se um modo eficiente de reconstruir uma solução admissível do primeiro problema a partir de uma solução admissível do segundo, além de garantir que a qualidade da solução reconstruída seja comparável à da solução inicial. Todas as considerações derivadas e as definições necessárias foram estudadas por [LEA 97].

4.4 Conclusão

Este capítulo apresentou o estudo dos algoritmos aproximativos, através da definição formal de algoritmos aproximativos e da apresentação das principais medidas de qualidade para este tipo de algoritmo.

Ainda, foram introduzidos os conceitos de: *esquema aproximativo*, importante, relativo ao requerimento de exatidão e de qualidade de um algoritmo; *redutibilidade*, o mais importante da Teoria da Complexidade, que é a habilidade de preservar a otimalidade de uma solução e reconstruir um solução admissível de um primeiro problema a partir de uma solução admissível de um segundo; e, *classes de problemas de otimização*, que é uma analogia à classe de problemas de decisão.

é importante observar que este estudo, de medidas de qualidade de algoritmos aproximativos, contribui significativamente para a decisão da viabilidade de um algoritmo para um determinado problema. Conhecida a exatidão do algoritmo aproximativo pode-se ter a medida da qualidade de sua resposta em relação à solução ótima.

5 Caracterização do Algoritmo Genético

5.1 Introdução

Entre os anos 50 e 60, vários cientistas da computação estudaram sistemas evolucionários com a idéia de que a evolução poderia ser usada como uma ferramenta de otimização para problemas na engenharia. Os sistemas desenvolvidos pretendiam gerar uma população de candidatos à solução para um dado problema. Box (1957), Friedman (1959) e Baricelli (1967) desenvolveram algoritmos inspirados na *evolução natural* para problemas de otimização e aprendizagem de máquina. Entretanto, seus trabalhos não possuíam qualquer tipo de atenção às estratégias de evolução, programação evolucionária e Algoritmos Genéticos atuais.

Os Algoritmos Genéticos foram inventados por John Holland nos anos 60 e desenvolvidos por seus alunos na Universidade de Michigan em meados de 1970. O principal objetivo de Holland não foi desenvolver algoritmos para solucionar problemas específicos, mas dedicar-se ao estudo formal do fenômeno de evolução, como ocorre na natureza, e desenvolver maneiras de importá-lo aos sistemas de computação.

Segundo [GOL 94b], existem muitas maneiras de encarar os Algoritmos Genéticos, e talvez a maioria dos usuários vem aos Algoritmos Genéticos procurando por um solucionador de problemas [JON 93], mas esta é uma visão restritiva e muito superficial. Algoritmos Genéticos pode ser visto como um grupo de diferentes funções:

- como solucionador de problemas [GOL 94a];
- como embasamento competente para aprendizagem de máquina [GOL 89];
- como modelo computacional da inovação e criatividade;
- como modelo computacional para sistemas de inovação [SAR 93];
- como jogos técnicos de descobrimento [GOL 93], e
- como guia filosófico [CSI 90].

John Holland, em meados dos anos 70, acreditou que as peculiaridades da Evolução Natural poderiam ser implementadas de forma algorítmica a fim de alcançar uma versão computacional dos processos de evolução. Este algoritmo poderia solucionar qualquer problema que apresentasse as mesmas características da evolução.

O sistema trabalhava com uma população (um conjunto) de algumas cadeias de bits (0's e 1's) denominados indivíduos (por convenção um indivíduo é constituído por um cromossomo). Semelhante à natureza, o sistema evoluía até o melhor cromossomo para atender um problema específico, mesmo sem saber que tipo de problema estava sendo solucionado. A solução era encontrada de um modo automático e não-supervisionado, e as únicas informações dadas ao sistema eram os ajustes de cada cromossomo produzido por ele.

A habilidade de uma população de cromossomos explorar o espaço de busca e combinar o melhor resultado encontrado mediante qualquer mecanismo de reprodução é intrínseca à evolução natural e é explorada pelos Algoritmos Genéticos.

Os Algoritmos Genéticos representam o resultado da mimetização tecnológica que, embora, muito aquém do modelo original, dá sinais de um excelente potencial para simular a vida [RAM 94]. A capacidade de reprodução é o principal fator de evolução dos seres vivos.

Por evolução entende-se a adaptação de um indivíduo ao meio ambiente. A adaptação por reprodução incorre-se no fato dos seres vivos modificarem-se a medida que vão se reproduzindo (Darwinismo). Esta adaptação ocorre intencionalmente, ou seja, os indivíduos se modificam de acordo com as necessidades e essa modificação passa diretamente para os descendentes (Lamarkismo).

O princípio adotado pelos Algoritmos Genéticos é o processo de evolução por seleção natural, ou seja, os indivíduos mais adaptados ao meio ambiente tem mais chances de sobreviver. As características da evolução natural podem ser enumeradas como segue [LAG 96]:

1. a evolução é um processo que modifica as estruturas que compõem um indivíduo, podendo incidir indiretamente no comportamento dele;

2. os processos de seleção natural proporcionam que determinadas estruturas (componentes dos cromossomos), se selecionadas, sejam reproduzidas mais freqüentemente do que as não selecionadas;

3. as mutações possibilitam que os cromossomos gerados tenham informações (características) diferentes dos cromossomos antecessores que os geraram, permitindo aos processos de recombinação introduzirem cromossomos diferentes na população, através da combinação de material genético dos pais (geradores);

4. evolução biológica não possui memória. Todavia, ela sabe como produzir estruturas que se adaptam melhor ao seu meio ambiente. Este conhecimento está armazenado no grupo genético do indivíduo, responsáveis pelas suas características.

A premissa dos Algoritmos Genéticos é encontrar soluções aproximadas para problemas de grande complexidade computacional [AGU 97b, GOL 89, HOL 75, LAG 96] mediante o processo de *evolução simulada* que possui uma manipulação *cega* dos cromossomos, ou seja, o processamento não possui nenhuma informação a respeito do problema que está tratando de resolver, exceto o valor da função objetivo.

No conceito original dos Algoritmos Genéticos, a função objetivo, também conhecida por função de avaliação, é a única informação que avalia o cromossomo.

Devido à seleção natural, cada população ganha uma certa quantidade de *conhecimento* [AGU 96], o qual é codificado e incorporado à nova formação de seus cromossomos. Esta formação é modificada por mecanismos de reprodução. Os mais utilizados mecanismos de reprodução são: as mutações, as inversões de partes de cromossomos, e o cruzamento de cromossomos, chamado de *crossover*.

As mutações provêem certa variação e *ocasionalmente* introduzem alterações benéficas aos cromossomos. A inversão é um mecanismo de alteração pela inversão do código do cromossomo. O cruzamento é o responsável pelo intecâmbio de material genético proveniente dos cromossomos geradores. Evidentemente, é o *crossover* que influi na eficiência dos Algoritmos Genéticos e o destaca nitidamente de outras meta-heurísticas. Usando o cruzamento, as chances das características ideais

se perpetuarem durante o processamento aumentam devido os pais com graus de adaptações maiores se reproduzirem com maior freqüência.

5.2 Funcionamento

Um Algoritmo Genético básico funciona da seguinte forma [AGU 96]: (i) uma população de cromossomos se mantém ao longo de todo o processo; (ii) a cada um dos cromossomos associa-se um valor de adaptação que está diretamente relacionado com o valor da função objetivo a otimizar; (iii) cada cromossomo codifica um ponto no espaço de busca do problema; (iv) dois cromossomos são selecionados de acordo com seus valores de adaptação para serem os geradores de duas novas configurações mediante um processo de reprodução; (v) estas novas configurações ocupam, reservam, seu espaço na nova geração. Este processo é repetido tantas vezes quantas forem necessárias.

Independente da sofisticação, o formato de um Algoritmo Genético pode ser descrito através da seguintes componentes [DAV 91]:

- Uma representação, em termos de cromossomos, das configurações assumidas no problema.
- Parâmetros de entrada do Algoritmo Genético. (Tamanho da população, número de gerações, taxas relativas aos operadores genéticos, ...)
- Uma maneira de criar e/ou inicializar as configurações assumindo a idéia de população inicial.
- Uma função de avaliação que permita ordenar, classificar, valorar, os cromossomos de acordo com o objetivo do algoritmo.
- Operadores Genéticos que gerem, produzam e/ou alterem a composição dos cromossomos durante a reprodução.

Ressalta-se que: as soluções de qualquer problema precisam estar codificadas nos cromossomos; a função de avaliação, considerada como o elo entre o Algoritmo Genético e o problema a ser resolvido, deve testar cada cromossomo, verificando se ele soluciona o problema. Um Algoritmo Genético, em forma de pseudo-código, é mostrado na figura 5.1.

Com o estudo mais detalhado da filosofia dos Algoritmos Genéticos, das características gerais do algoritmo e do pseudo-código do Algoritmo Genético básico encontrado na literatura, percebe-se certas *entidades* que se sobressaem no processamento das rotinas que mimetizam a evolução simulada. Nesta seção são relatados os módulos identificados.

5.2.1 Módulo de avaliação

O módulo de avaliação é considerado o mais importante, onde se encontra a ligação entre o algoritmo e o problema, e possui as seguintes características:

- possui uma função de avaliação. Esta função serve para valorar os indivíduos de acordo com o grau de adaptação, e é específica para cada tipo de problema;

```
procedimento AG
início
  t: = 0
  inicializa população( t);
  avalia indivíduos da população( t);
  enquanto condição de término não satisfeita faça
    t: = t + 1;
    seleciona população( t) da população( t-1);
    recombina indivíduos na população( t);
    avalia indivíduos na população( t);
  fim-enquanto
fim.
```

Figura 5.1 – Pseudo-código do Algoritmo Genético

- a configuração assumida por um cromossomo é conhecida como genótipo;
- a característica atribuída ao indivíduo é conhecida como fenótipo; e,
- genótipos diferentes podem resultar em fenótipos satisfatórios para sobrevivência do indivíduo.

5.2.2 Módulo de estruturação

O módulo de estruturação é reponsável pela população e contém um conjunto de técnicas para a seleção dos indivíduos mais aptos juntamente com suas posteriores reproduções. Também é responsável pelas seguintes tarefas no processamento: *Representação A representação adotada na maioria das implementações de Algoritmos Genéticos é a representação binária. Esta representação facilita os processos de seleção e reprodução dos indivíduos. *Inicialização A inicialização da primeira população, denominada primeira geração, é aleatória. *Seleção de pais A *Roullete Whell Parent Selection*, ou técnica da roleta para a seleção dos pais, é a técnica mais utilizada de seleção nos Algoritmos Genéticos. A técnica resume-se nos seguintes passos:

1. faz-se o somatório dos graus de adaptação de cada cromossomo (chamado de ajuste acumulado);
2. gera-se números aleatórios entre 1 e o maior ajuste acumulado; e,
3. sleciona-se o cromossomo que possui grau de adaptação maior ou igual ao número aleatório.

Como numa roleta, os cromossomos possuem partes proporcionais de acordo com os seus ajustes. Naturalmente um mau cromossomo, com menor grau de adaptação, pode ser selecionado, mas isto apenas contribui para uma maior diversidade genética no modelo. Importante observar que o número de pais a serem selecionados é, geralmente, o mesmo número de indivíduos a serem substituíduos. Isto faz-se necessário para manter constante o número de indivíduos no decorrer das gerações. Um exemplo pode ser observado na tabela 5.1.

Tabela 5.1 – Técnica da Roleta

Cromossomos	1	2	3	4	5	6	7	8	9	10
Ajustes	10	8	6	7	9	5	3	2	4	1
Ajuste Acumulado	10	18	24	31	40	45	48	50	54	55
Número Aleatório	23	49	16	13	1	27	47	1	33	36
Selecionados	3	8	2	2	1	4	7	1	5	5

Então, seja uma população com 10 cromossomos. Suponha os seguintes valores de ajustes: cromossomo 1, ajuste 10; cromossomo 2, ajuste 8; e assim por diante. Faz-se o somatório dos ajustes para cada cromossomo, logo: o cromossomo 1 possui ajuste acumulado 10; o cromossomo 2 possui ajuste acumulado 18 (ajuste acumulado para o cromossomo 1 + ajuste do cromossomo 2); o cromossomo 3 possui ajuste acumulado 24 (ajuste acumulado para o cromossomo 2 + ajuste do cromossomo 3); e assim por diante. Agora, suponha os seguintes números sorteados aleatoriamente: primeiro número sorteado, 23; segundo número sorteado, 49; terceiro número sorteado, 16; e assim por diante. Portanto, os cromossomos selecionados são os seguintes: cromossomo 3, pois seu ajuste acumulado é o seguinte maior igual ao número aleatório; depois, seleciona-se o cromossomo 8; cromossomo 2; e assim por diante conforme a tabela.

5.2.3 Módulo de reprodução

é utilizado nos pais selecionados para efetivar a reprodução, garantindo a próxima geração. O módulo é composto por quatro seções, a saber: *Técnica de reprodução A técnica mais utilizada é a da reprodução por gerações, em que a nova geração substitui inteiramente o lugar da primeira. Na reprodução este processo não se observa. Outros processos, ditos de otimização, também são utilizados, como: elitismo e a reprodução em estado constante. *Técnica de seleção de operadores Maneira como o algoritmo selecionará o operador para a reprodução. As situações mais freqüentes são: usar o primeiro, ou usar a técnica da roleta. *Lista de operadores Apresenta os operadores que serão utilizados no decorrer do algoritmo. Geralmente são utilizados os operadores de cruzamento em um ponto e mutação, devido às facilidades de implementação e da alta confiabilidade.

Mutação

A mutação incide sobre cada bit da cadeia, podendo trocar o seu valor. A função de mutação está associada a taxa de mutação, que é um dos parâmetros do programa. O desempenho do Algoritmo Genético está diretamente associado à taxa de mutação. A função dispara um número aleatório entre 0 e 1. Se o número for menor ou igual à taxa de mutação estipulada, o bit muda de valor. Um exemplo pode ser observado na tabela 5.2, neste caso a taxa é de 0.008, ou seja, o bit tem 8 em 1000 chances de trocar de valor e os cromossomos têm 4 bits. Os números aleatórios destacados são menores que a taxa de mutação, portanto, os bits correspondentes foram modificados.

Tabela 5.2 – Operador de Mutação

Cromossomo Antes	Bits Sorteados	Cromossomo Depois
1001	0401 0100 2066 0323	1001
1110	0890 (0007) (0005) 2840	1000

Cruzamento em um ponto

O cruzamento em um ponto corresponde a uma divisão do material genético dos pais em um ponto aleatório para misturá-los e gerar o novo cromossomo que pertencerá ao filho. A seleção natural responde pelo ajuste dos cromossomos e a seleção dos pais, enquanto que a mutação e o cruzamento respondem pela diversidade genética dos filhos. Um exemplo pode ser observado na tabela 5.3, utilizando cromossomos de 6 bits e a partir de um número aleatório (no exemplo, 5, escolhido entre 1 e comprimento da cadeia) divide-se o cromossomo. Uma observação impor-

Tabela 5.3 – Operador de Cruzamento em um Ponto

Pais	Divisão	Cruzamento	Filhos
1 1 1 1 1 1	1 1 1 1 — 1 1	1 1 1 1 — 0 0	1 1 1 1 0 0
0 0 0 0 0 0	0 0 0 0 — 0 0	0 0 0 0 — 1 1	0 0 0 0 1 1

tante a respeito do cruzamento é que se pode gerar filhos completamente diferentes dos pais e mesmo assim contendo diversas características em comum. Outra questão é que o cruzamento não modifica um bit na posição em que os pais tem o mesmo valor, considerada uma característica cada vez mais importante com o passar das gerações. *Técnica de deleção Está implícita na rotina de cruzamento e mutação. Quando o vetor de população recebe os filhos resultantes da reprodução dos pais escolhidos, a população anterior é automaticamente destruída.

5.3 Rotinas de otimização

Este tipo de rotina é utilizado para mudar o funcionamento do algoritmo, incrementando-o com rotinas consideradas mais inteligentes, ou que requeiram estruturas que tornem o algoritmo mais inteligente. Algumas estruturas, não-convencionais, são citadas a seguir.

5.3.1 Operador inversor

Utilizados em algumas, poucas e específicas, implementações. Este operador manipula apenas 1 (um) cromossomo por vez, invertendo a ordem dos elementos entre dois pontos escolhidos aleatoriamente. Um exemplo é observado na tabela 5.4.

Tabela 5.4 – Operador Inversor

Cromossomo	1 0 1 0 0 1 0 0 0 1
Ponto 1:	3
Ponto 2:	7
Trecho	1 0 0 1 0
Inversão	0 1 0 0 1
Cromossomo	1 0 0 1 0 0 1 0 0 1

5.3.2 Elitismo

Na natureza, o indivíduo mais apto não apenas pode reproduzir mais freqüentemente, como também possuir maior longevidade, sobrevivendo muitas vezes de uma geração para outra reproduzindo. Em Algoritmos Genéticos com este tipo de rotina pega-se os melhores cromossomos de uma população e repassa-os para a geração seguinte.

5.3.3 Reprodução em estado constante

A reprodução em estado constante age como a técnica do elitismo, protegendo os melhores indivíduos do cruzamento e da mutação, indesejáveis quando o cromossomo está com um ajuste muito alto. Ao invés de destruir a antiga população, transporta-se os filhos para a antiga população e elimina-se os cromossomos de ajuste mais baixo, se muitos estiverem com o mesmo grau de ajuste, os mais velhos são destruídos.

5.3.4 Cruzamento em dois pontos

O cruzamento em dois pontos difere do cruzamento, mostrado anteriormente, apenas no número de pontos em que se dividem os cromossomos. Um exemplo é mostrado nas tabelas 5.5, 5.6 e 5.7. Na tabela 5.5, os bits destacados representam

Tabela 5.5 – Características de Alto Desempenho

Cromossomo 1:	(1)	1	(0)	1	1	0	0	1	0	1	1	0	1	(1)
Cromossomo 2:	0	0	0	1	(0)	1	1	0	(1)	1	1	1	0	0

características de alto desempenho para os cromossomos. Em um cruzamento de um só ponto estas características se perderiam na reprodução. Com o cruzamento de dois pontos, pode-se unir os esquemas dos pais e conseguir filhos com desempenho melhor ainda. Os pontos de divisão, mostrados na tabela 5.6, não são aleatórios, são escolhidos de acordo com os padrões encontrados.

Tabela 5.6 – Pontos de Divisão para o Cruzamento em Dois Pontos

Ponto 1:	4
Ponto 2:	10
Cromossomo 1:	1 1 0 1 — 1 0 0 1 0 1 — 1 0 1 1
Cromossomo 2:	0 0 0 1 — 0 1 1 0 1 1 — 1 1 0 0

Tabela 5.7 – Resultado do Cruzamento em Dois Pontos

Filho 1:	(1)	1	(0)	1	(0)	1	1	0	(1)	1	1	0	1	(1)
Filho 2:	0	0	0	1	1	0	0	1	0	1	1	1	0	0

5.4 Aplicações dos algoritmos genéticos

As características dos Algoritmos Genéticos citadas acima são muito simples, mas variações do algoritmo básico são usados em um grande número de problemas e modelos científicos e da engenharia. Alguns exemplos são citados abaixo [MIT 96, DAV 91, AGU 97b]:

Otimização: usados em uma grande variedade de problemas de otimização, incluindo otimização de problemas numéricos e combinatoriais como o escalonamento de trabalho e o desenvolvimento de *layouts* para circuitos [AND 95, BUR 95, JAN 93, SCH 95, SCH 96].

Programação Automática: usados para desenvolver programas de computador para tarefas específicas e outras estruturas computacionais, tais como, autômatos celulares e redes de ordenação [MIT 94a, MIT 94b].

Aprendizagem de Máquina: usados para muitas aplicações em aprendizagem, incluindo classificação e predição de tarefas, tais como: previsão do tempo ou estrutura de proteínas. Usados também para desenvolver aspectos particulares de sistemas de aprendizagem, tais como: pesos para redes neurais, regras sistemas de aprendizagem classificadores ou sistemas de produção simbólicos, e sensores para robôs [JAN 93, SER 96, SYE 97].

Economia: usados para modelar processos de inovação, o desenvolvimento de estratégias de lances e na predição de mercados econômicos emergentes [KIN 95, MUH 93].

Imuno-Sistemas: usados para modelar vários aspectos dos sistemas imunológicos naturais, inclusive a mutação somática durante a vida do indivíduo e o descobrimento de famílias de multi-gens durante o tempo evolucionário.

Ecologia: usados para modelar fenômenos ecológicos como competição entre nichos biológicos, co-evolução parasita-hospedeiro, simbiose e fluxo de recursos [HAY 94, SMI 92].

Genética: usados para estudar questões da genética de populações [SMI 92], tais como: em quais condições um gene é viável, em termos evolucionários, para ser recombinado?

Evolução e Aprendizagem: usados para estudar como indivíduos aprendem e a evolução das espécies (efeitos) [JAN 93].

Sistemas Sociais: usados para estudar aspectos evolucionários de sistemas sociais, tais como a evolução do comportamento social em colônias de insetos e, mais genericamente, a evolução da cooperação e comunicação em sistemas multi-agentes.

Outros Temas: no desenvolvimento de aeronaves; antecipação dinâmica da rota em redes de telecomunicações; geração de trajetórias para robôs; modelagem de sistemas dinâmicos não-lineares aplicados em segurança; sistemas de aquisição de estratégias; diagnóstico de múltiplas falhas; análise de DNA; processamento de informações de sonares; otimização econômica no gerenciamento de rios; problema do caixeiro viajante e ao escalonamento de seqüências; etc. [AGU 96, GAG 94, GRA 97, HAY 94, JAC 95]

5.5 Conclusão

Este capítulo apresentou um estudo abrangente sobre os fundamentos dos Algoritmos Genéticos englobando: a sua história, desde os objetivos iniciais de Box, Friedman e Baricelli, sem nenhuma preocupação com estratégias de evolução até os estudos de John Holland e seus alunos; a fundamentação do algoritmo em cima das teorias oriundas da biologia, como o Darwinismo e Lamarkismo; o funcionamento, através da exposição de um conjunto de requisitos empíricos para o enquadramento de um dado algoritmo como sendo um Algoritmo Genético; constituição dos módulos, detalhamento das estruturas responsáveis pelo processamento; as opções de otimização e a aplicabilidade dos Algoritmos Genéticos.

Este estudo destaca-se pela importância no auxílio da formalização e na composição do método de desenvolvimento de Algoritmos Genéticos (mostrado no capítulo 6).

6 Método de Desenvolvimento de Algoritmos Genéticos

6.1 Definição formal

Dado um problema p = < D, R, q> e uma solução α para p, um *algoritmo* é, segundo [KNU 83], um método abstrato que computa α. As partes que compõem uma especificação formal de um **método de desenvolvimento de algoritmos** (MDA) são definidas a partir de um programa abstrato e um conjunto de axiomas. Assim tem-se:

- Um **método de desenvolvimento de algoritmos** (MDA), segundo [TOS 88], é um par (*Prog*, *Axio*), onde *Prog* é um programa abstrato definido a partir de funções sintaticamente bem definidas e *Axio* é um conjunto de axiomas que definem a semântica das funções.

- Se m = (*Prog*, *Axio*) é um **método de desenvolvimento de algoritmos**, uma instância de *Prog* que satisfaz *Axio* é um **algoritmo desenvolvido por m**. Se p é um problema e α é solução de p, então i(m, p, α) é o conjunto de todos os algoritmos desenvolvidos por m que computam α. O *domínio de m*, D(m), é o conjunto de todos os pares (p, α) tal que p é um problema, α é a solução de p e i(m, p, α) $\neq \emptyset$.

- Se m_1 e m_2 são dois MDA's, diz-se que m_2 é pelo menos tão geral quanto m_1, $m_1 \sqsubseteq m_2$ sss (se (p, α) $\in \mathsf{D}(m_1)$, então $\exists\ \alpha$' t.q. (p, α') $\in \mathsf{D}(m_2)$).

A especificação formal de um MDA consiste de um diagrama sintático que define o domínio das funções e predicados, um programa abstrato que dá a estrutura algorítmica do método e uma axiomatização que define o interrelacionamento das funções e predicados que compõem o método, além de uma verificação da correção do programa abstrato. Um algoritmo gerado pelo MDA é uma instância do programa abstrato obtido pela instanciação de funções e predicados que satisfazem os axiomas da especificação.

Nem sempre, dado um problema (definido no capítulo 2), é fácil encontrar um **método de desenvolvimento de algoritmos** adequado, muitas vezes é necessário modificar o problema, usando o conceito de redução mostrado no mesmo capítulo.

6.2 Especificação formal

A descrição do tipo de Algoritmo Genético estudado já foi feita, anteriormente (capítulo 5), então nesta seção a atenção volta-se aos aspectos formais deste tipo de algoritmo. A descrição do funcionamento do algoritmo torna-se mais precisa ao ser definida através de tipos abstratos de dados, formulada como um programa abstrato operando sobre estes tipos abstratos, cuja especificação garante a corretude do programa. Para aplicar esta estratégia a um dado problema é suficiente escrever um módulo de implementação definindo as operações em termos do problema [AGU 97c, TOS 88].

6.2.1 Definição dos domínios

Considere os seguintes domínios:

P	domínio das instâncias de problemas
I	conjunto dos indivíduos
OP	conjunto dos operadores genéticos
$\mathbb{N}$	conjunto dos números naturais
POP	POP $\subseteq$ (I x $\mathbb{N}$)$^+$
	POP = { $<<$ i_1, $v_1>$, ..., $<$ i_n, $v_n>>$ / i_1, i_2, ..., $i_n \in$ I
	$\wedge$ v_1, v_2, ..., $v_n \in \mathbb{N}$}

Seja um problema p $\in$ P, então p é uma estrutura do tipo (f_{obj}, n_{ger}, n_{ind}, $lista_{taxa}$), onde os elementos são identificados na figura 6.1.

f_{obj}	é a função a ser maximizada, ou minimizada, conforme o caso.
n_{ger}	é o número de iterações ou gerações.
n_{ind}	é o tamanho da população
$lista_{taxa}$	são as taxas, isto é, probabilidades dos operadores genéticos
	($taxa_{subs}$, $taxa_{op_1}$, $taxa_{op_2}$, ..., $taxa_{op_k}$)
	Onde: $taxa_{subs}$ determina a população resultante da reprodução.
Por exemplo:	op_1 = operador de cruzamento (op_{cross}, *crossover*).
	op_2 = operador de mutação (op_{mut}).

Figura 6.1 – Definição dos Domínios

6.2.2 Diagrama sintático

O Diagrama Sintático (figura 6.2) representa, graficamente, a sintaxe e a aridade das funções e predicados do programa abstrato. Na figura observa-se a seguinte notação: as funções são representadas por linhas com setas e os predicados com somente linhas. A sintaxe das funções e predicados encontrados no diagrama são listadas abaixo.

condição -término: POP x P $\rightarrow$ { T, F}, predicado definido pelos axiomas 1, 2, 3.

completa: POP x P $\rightarrow$ { T, F}, predicado definido pelo axioma 12.

mais-adaptado: POP x POP x P $\rightarrow$ { T, F}, predicado definido pelo axioma 13.

melhor: I x POP x $\mathbb{N}$ x P $\rightarrow$ { T, F}, predicado definido pelo axioma 14.

recupera: POP x P $\rightarrow$ I, função que retorna um indivíduo da população, que no algoritmo é usado para selecionar um elemento da última configuração assumida pela população (no caso de convergência ao elitismo coincidindo com o melhor indivíduo da população). Observa-se que cabe ao procedimento escolher o melhor, ou um dos melhores.

acrescenta: POP x POP x P $\rightarrow$ POP, função que efetua o acréscimo de uma lista, que pode ser unitária, de novos indivíduos à população.

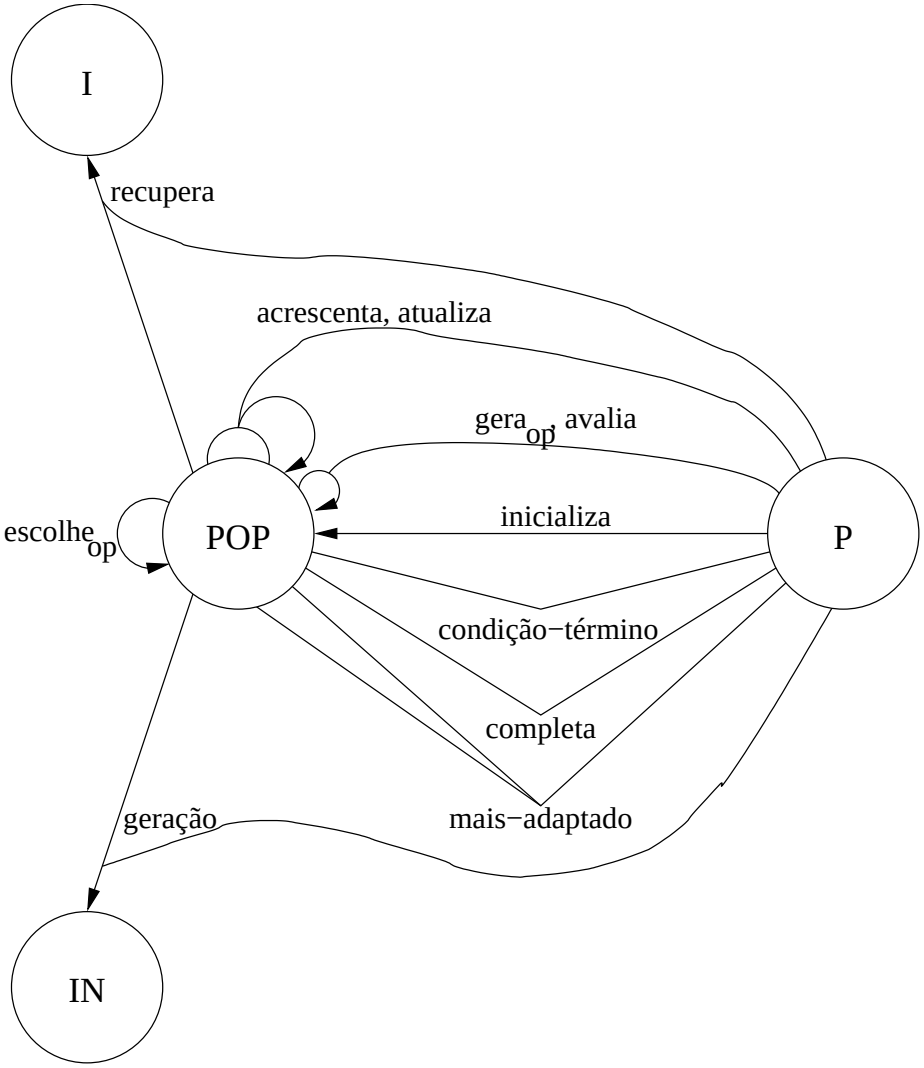

Figura 6.2 – Diagrama Sintático

atualiza: POP x POP x P → POP, função que efetua a junção de duas populações, utilizada no final do procedimento reprodução quando unem-se a nova geração e a geração anterior. Observa-se que se houver necessidade de substituir somente parte da população, apenas acrescenta-se aos novos elementos os melhores da geração anterior, caso contrário, repassa-se todos os indivíduos gerados para a nova população.

gera$_{op}$: POP x P → POP, função que retorna uma lista de novos indivíduos gerados de modo específico pelo mecanismo de reprodução (por exemplo: cruzamento – op$_{cross}$ e mutação – op$_{mut}$).

avalia: POP x P → POP, função que retorna o grau de adaptação de uma lista de indivíduos, ou seja, apenas associa valores a cada indivíduo da lista passada à função, neste caso, avalia($< i_1, i_2, \ldots, i_n >$, p) = $< f_{obj}(i_1), f_{obj}(i_2), \ldots, f_{obj}(i_n) >$, onde f_{obj} é um elemento da estrutura p.

inicializa: P → POP, função que gera a população que inicia o processamento do

algoritmo, de acordo com os parâmetros do problema.

geração: POP x P $\rightarrow \mathbb{N}$, função que retorna o índice utilizado para a verificação da condição de término do algoritmo.

escolhe$_{op}$: POP $\rightarrow$ POP, função que retorna a lista dos melhores indivíduos da população, escolhidos de modo específico pelo o mecanismo de reprodução.

escolhe-operador: OP $\rightarrow$ OP, função que retorna o operador genético de modo citado no capítulo 5 e, escolhe-operador(OP) $\in$ OP.

6.2.3 Programa abstrato

*Procedimento AG Na figura 6.3, pode ser observado o algoritmo abstrato para o pseudo-código encontrado no capítulo 5. Observa-se alguns pontos importantes

---

φ(p) = (($n_{ger} > 0$) $\wedge$ ($n_{ind} > 0$) $\wedge$ ($0 < taxa_{cross} \leq 1$) $\wedge$
($0 < taxa_{mut} \leq 1$) $\wedge$ ($0 < taxa_{subs} \leq 1$) $\wedge$ ($0 < taxa_{op_3} \leq 1$) $\wedge$
($0 < taxa_{op_4} \leq 1$) $\wedge \ldots \wedge$ ($0 < taxa_{op_k} \leq 1$))

procedimento AG(p)
l_1. pop $\leftarrow$ inicializa(p)
l_2. enquanto condição-término(pop, p)
Invariante: completa(pop, p)
l_3. pop $\leftarrow$ reprodução(pop, p)
l_4. fim-enquanto
l_5. s $\leftarrow$ recupera(pop, p)

ψ(s, p) = (($\exists$ n_{ger})($\exists < pop_1, pop_2, \ldots, pop_{n_{ger}} > \in$ POP)
($\forall$ i)(($1 \leq i \leq n_{ger} - 1$) $\Rightarrow$ (completa(pop_{i+1}, p) $\wedge$
(pop_{i+1} = reprodução(pop_i, p)) $\wedge$
mais-adaptado(pop_{i+1}, pop_i, p) $\wedge$ s $\in$ $pop_{n_{ger}}$)))

Figura 6.3 – Algoritmo Abstrato - AG

no programa abstrato, como:

- uma asserção de entrada (indicada por φ(p)), cuja finalidade é restringir o funcionamento do algoritmo para entradas que satisfazem φ(p), ou seja, uma condição necessária para o algoritmo entrar em funcionamento.
- uma invariante (*Invariante: completa(pop, p)*), que mostra um estado constante, naquela posição do algoritmo, isto é, *completa(pop, p)* deve ser verdadeira toda vez que a execução atingir este ponto.
- uma asserção de saída (indicada por ψ(s, p)), cuja finalidade é análoga à asserção de entrada, sendo uma condição necessária no término do processamento do algoritmo.

φ(pop, p) = completa(pop, p)

procedimento reprodução(pop, p)
l_1. pop' ← nil
l_2. j ← 0
l_3. m ← n_{ind} . $taxa_{subs}$
l_4. enquanto m > 0
l_5. op ← escolhe-operador(op_{cross}, op_{mut}, op_3, ..., op_k)
l_6. j ← j + 1
l_7. caso op
op_x:
l^x_1. <$lista_{ind}$>← $escolhe_{op_x}$(pop)
l^x_2. <$lista_{filho}$>← ($gera_{op_x}$, avalia)(<$lista_{ind}$>, p)
l^x_3. pop' ← acrescenta(<$lista_{filho}$>, pop', p)
l^x_4. m ← m - |$lista_{filho}$|
⋮
l_8. fim-caso

Invariante: mais-adaptado(pop', $\biguplus_{i=1}^{j} escolhe_{\overline{op}_i}$(pop), p)

l_9. fim-enquanto
l_{10}. pop_{final} ← pop'
l_{11}. população ← atualiza(pop, pop_{final}, p)
l_{12}. fim-com-saída(população)

ψ(população, pop, pop_{final}, p) = mais-adaptado(população, pop, p) ∧ completa(população, p) ∧ pop_{final} ⊆ população

Figura 6.4 – Algoritmo Abstrato - Reprodução

*Procedimento Reprodução Na figura 6.4, pode ser observado o algoritmo abstrato referente ao procedimento da reprodução (linha 3 do procedimento AG, figura 6.3).

Observa-se dois aspectos importantes na figura 6.4: na invariante, o aparecimento do $\overline{op}_i$, cuja intenção é indicar o operador utilizado na iteração i, e não o operador i; na asserção de saída, pop_{final} ⊆ população, a incumbência do controle do número de elementos que serão gerados em uma geração deve ser atribuída ao momento da implementação, neste caso a intenção é garantir que pelo menos m elementos de pop_{final} estarão em população no final do processamento.

Duas operações de reprodução que merecem um tratamento especial são o cruzamento (*crossover*) e a mutação, porque são os mais utilizados. Um exemplo para a representação, no algoritmo abstrato, dos operadores genéticos cruzamento e mutação pode ser visto na figura 6.5.

6.2.4 Axiomatização

A axiomatização é efetuada a partir do programa abstrato e obedece as seguintes condições:

	op_{cross}:
l_1^{cross}.	<< i_1, v_1>, < i_2, v_2>>← escolhe$_{cross}$(pop)
l_2^{cross}.	<< < $filho_1$, v_1'>, < $filho_2$, v_2'>>>←
	(gera$_{cross}$, avalia)(< < i_1, v_1>, < i_2, v_2>>, p)
l_3^{cross}.	pop' ← acrescenta(< < $filho_1$, v_1'>, < $filho_2$, v_2'>>, pop', p)
l_4^{cross}.	m ← m - 2
	op_{mut}:
l_1^{mut}.	<< i, v>>← escolhe$_{mut}$(pop)
l_2^{mut}.	<< filho, v'>>← (gera$_{mut}$, avalia)(<< i, v>>, p)
l_3^{mut}.	pop' ← acrescenta(<< filho, v'>>, pop', p)
l_4^{mut}.	m ← m - 1

Figura 6.5 – Uma Representação para os Operadores Genéticos

- representação: o indivíduo é representado por um cromossomo, < i, v> onde: i é o genótipo do indivíduo, criado pelo procedimento inicializa e modificado pelos módulos de reprodução, e v é o fenótipo do indivíduo, valor dado pela função avalia sempre depois do genótipo ser modificado. Não foi particularizado o tipo de representação adotada, representação binária, decimal, ou outra específica. Mas, o indivíduo deve ser capaz de codificar a solução, ou seja, o elemento que servirá como solução admissível para problema, como pode ser visto no axioma 4.

- parâmetros: foram levados em conta os parâmetros comuns de um Algoritmo Genético. (n_{ind}, n_{ger}, lista$_{taxa}$, ...)

- inicialização: representado pela função inicializa, e presente nos axiomas 0 e 1, onde parte-se do conceito de uma população inicial completa e de ser executada uma única vez, no início do processamento. Não foi evidenciado o modo como a população inicial é gerada, se aleatória ou dirigida.

- função de avaliação: está representada intrinsecamente no programa abstrato e no axioma 6 pela função avalia que modifica o fenótipo do indivíduo toda vez que é manipulado pelo módulo de reprodução.

- operadores genéticos: estão representados genericamente pelo axioma 6, que é formado pela composição das funções gera$_{op}$ e avalia aplicadas aos indivíduos escolhidos.

- tamanho constante da população: definido pelo axioma 0 e garantido pela prova da invariante do procedimento AG.

- técnica de reprodução: a técnica representada é a reprodução por gerações (axioma 2).

- término: o axioma 3 define a condição de término como número de gerações.

- técnica de deleção: é definida pelo axioma 10, quando a população recebe os indivíduos resultantes da reprodução que podem substituir totalmente a população ou não, conforme o caso.

As técnicas de seleção de pais e de seleção de operadores não foram modeladas pois são rotinas dependentes de implementação e a lista de operadores foi modelada genericamente pelo programa abstrato com o objetivo de utilizar a modularidade da representação para a utilização de outras rotinas de busca de soluções em substituição aos operadores genéticos convencionais. Além disso, nenhuma rotina de otimização foi modelada por serem rotinas não convencionais.

Os seguintes axiomas dão a semântica dos predicados e funções do programa abstrato seguindo a convenção: p ∈ P; pop, pop', pop" e pop"' ∈ POP; e, op ∈ OP. *Axioma 0 (∀p)(completa(inicializa(p), p))

A população gerada pelo procedimento inicializa deve satisfazer o predicado completa, isto é, deve ter o número de elementos determinado por n_{ind} conforme a definição do predicado completa (axioma 12). *Axioma 1 (∀p)(geração(inicializa(p), p) = 1)

A população gerada pelo procedimento inicializa corresponde à primeira geração do procedimento. *Axioma 2 (∀p)(∀pop)(geração(reprodução(pop, p), p) = geração(pop, p) + 1)

Cada execução do procedimento reprodução resulta uma próxima geração. *Axioma 3 (∀p)(∀pop)(condição-término(pop, p) ⇔ geração(pop, p) ≥ n_{ger})

O processamento deve terminar quando o número de gerações chegar ao valor estipulado pelo parâmetro n_{ger}. *Axioma 4 (∀p)(∀pop)(recupera(pop, p) ∈ pop)

O procedimento recupera retorna um elemento da população corrente. *Axioma 5 (∀p)(∀pop)(mais-adaptado(reprodução(pop, p), pop, p))

A população resultante do procedimento reprodução deve ser mais adaptada que a anterior. *Axioma 6 (∀p)(∀pop)(∀op)(mais-adaptado(($gera_{op}$, avalia)($escolhe_{op}$(pop), p), $escolhe_{op}$(pop), p))

A população resultante da aplicação das funções de geração e avaliação, nos escolhidos da população anterior, deve ser mais adaptada do que estes. *Axioma 7 (∀p)(∀pop)(∀pop')((acrescenta(pop, pop', p) ⊆ pop ⊎ pop') ∧ (pop ⊎ pop' ⊆ acrescenta(pop, pop', p)))

Este axioma define a função acrescenta como a união de duas populações. *Axioma 8 (∀pop)(∀op)(< i, v> ∈ $escolhe_{op}$(pop) ⇒ < i, v> ∈ pop)

Um indivíduo, elemento, escolhido de uma população, deve pertencer a esta população. *Axioma 9 (∀p)(∀pop)(∀pop')(completa(pop, p) ⇒ completa(atualiza(pop, pop', p), p))

Dada uma população que satisfaz o predicado completa, a população resultante da aplicação da função atualiza deve satisfazer, da mesma forma, o predicado completa. *Axioma 10

(∀p)(∀pop)(∀pop')(pop' ⊆ atualiza(pop, pop', p)) ∧ (< i, v> ∈ atualiza(pop, pop', p)) ⇒ (< i, v> ∈ pop' ∨ (< i, v> ∈ pop ∧ melhor(< i, v>, pop, |pop| - |pop'|, p)))

Todo elemento de pop' (gerado pela reprodução) deve passar para a população resultante. Ainda, um elemento que pertence à população resultante da função atualiza, ou estava em pop' (foi gerado pelo procedimento reprodução), ou pertencia à pop (geração anterior) e, além disso, era um dos melhores elementos desta população. *Axioma 11 (∀p)(∀pop)(∀pop')(∀pop")(∀pop"')((mais-adaptado(pop, pop', p) ∧ mais-adaptado(pop", pop"', p)) ⇒ (mais-adaptado(pop ⊎ pop",

pop' $\uplus$ pop"', p)))

A união de duas populações consideradas mais adaptadas que outras duas populações, é mais adaptadada que a união destas outras duas populações. *Axioma 12 ($\forall$p)($\forall$pop)(completa(pop, p) $\Leftrightarrow$ |pop| = n_{ind})

Uma população está completa quando foram gerados indivíduos suficientes para passar à nova geração. *Axioma 13 ($\forall$p)($\forall$pop)($\forall$pop') (mais-adaptado(pop, pop', p) $\Leftrightarrow$ ($\forall$< i, v> $\in$ pop)
($\exists$< i', v'> $\in$ pop')(v$\geq_p$ v'))

Para pop ser mais adaptada que pop' todo elemento de pop deve ser melhor que pelo menos um indivíduo de pop'.

*Axioma 14 ($\forall$p)($\forall$pop)($\forall$n)(melhor(< i, v>, pop, n, p) $\Leftrightarrow$ ($\forall$m $\geq$n)($\forall$< < i_1, v_1>, ..., < i_m, v_m>> $\in$pop)($\exists$j (1 <j $\leq$m))(v_j <$_p$ v))

Um indivíduo (< i, v>) está entre os n elementos de pop mais adaptados, isto é, existe no máximo n-1 elementos de pop, < i', v'> tal que v' $\geq_p$ v.

Observa-se que os Axiomas 3, 2 e 1 são os responsáveis pela definição da condição de término. Estes são imprescindíveis para a verificação de terminação do programa. Para ser modificada a condição de término basta modificar o Axioma 3, acrescentando outra alternativa para o encerramento, por exemplo, condição-término(pop, p) $\Leftrightarrow$ geração(pop, p) $\geq$ n_{ger} $\vee$ .... Percebe-se que, da mesma maneira, fica garantida a terminação do programa pela inserção da nova condição com o conetivo "$\vee$", podem ser acrescentados novos axiomas que definam semanticamente a condição modificada.

6.2.5 Verificação da correção do programa

Nesta etapa, faz-se a verificação da correção do programa através da demonstração da veracidade das asserções (de entrada e de saída), das invariantes e do término do programa, utilizando os axiomas que foram determinados na seção anterior.

*Procedimento reprodução Tem-se que demonstrar que a asserção de saída é verdadeira: ψ(população, pop, pop_{final}, p) = mais-adaptado(população, pop, p) $\wedge$ completa(população, p) $\wedge$ pop_{final} $\subseteq$ população.

Antes, se demonstra a invariante:

$$\text{mais-adaptado(pop', } \biguplus_{i=1}^{j} \text{escolhe}_{\overline{op}_i}\text{(pop), p).}$$

Prova da invariante

BI: Seja $\overline{op}_1$ = escolhe-operador(OP), na 1ª iteração. (l_5)
Seja x_1 = escolhe$_{\overline{op}_1}$(pop), na 1ª iteração. (l_1^x)
<lista$_{filho}$>= (gera$_{\overline{op}_1}$, avalia)(x_1, p). (l_2^x)
pop'$_1$ = acrescenta((gera$_{\overline{op}_1}$, avalia)(x_1, p), $\emptyset$, p). (l_3^x)
Pelo axioma 6,
mais-adaptado(pop'$_1$, x_1, p), isto é,
mais-adaptado(pop'$_1$, escolhe$_{\overline{op}_1}$(pop), p)

HI: Chame $\overline{op}_j$ = escolhe-operador(OP), na j^{ésima} iteração. (l_5)
Chame x_j = escolhe$_{\overline{op}_j}$(pop), na j^{ésima} iteração. (l_1^x)
Chame pop'$_j$ o valor da variável pop' no final do caso, isto é,

<lista$_{filho}$>= (gera$_{\overline{op}_j}$, avalia)(x$_j$, p). (l_2^x)
pop'$_j$ = acrescenta((gera$_{\overline{op}_j}$, avalia)(x$_j$, p), pop'$_{j-1}$, p). (l_3^x)
Suponha, mais-adaptado(pop'$_j$, $\biguplus_{i=1}^{j}$x$_i$, p).

PI: Seja $\overline{op}_{j+1}$ = escolhe-operador(OP), na (j + 1)$^{\underline{ésima}}$ iteração. (l_5)
Seja x$_{j+1}$ = escolhe$_{\overline{op}_{j+1}}$(pop), na (j + 1)$^{\underline{ésima}}$ iteração. (l_1^x)
mais-adaptado(pop'$_j$, $\biguplus_{i=1}^{j}$x$_i$, p) pela hipótese da indução. (*)
Pela execução do programa,
<lista$_{filho}$>= (gera$_{\overline{op}_{j+1}}$, avalia)(x$_{j+1}$, p). (l_2^x)
pop'$_{j+1}$ = acrescenta((gera$_{\overline{op}_{j+1}}$, avalia)(x$_{j+1}$, p), pop'$_j$, p). (l_3^x)
Pelo axioma 6,
mais-adaptado(<lista$_{filho}$>, x$_{j+1}$, p). (**)
Por (*), (**) e pelo axioma 11, tem-se:
mais-adaptado(pop'$_j$ $\biguplus$ <lista$_{filho}$>, ($\biguplus_{i=1}^{j}$x$_i$) $\biguplus$ x$_{j+1}$)
Mas, pelo axioma 7,
acrescenta(<lista$_{filho}$>, pop'$_j$, p) = <lista$_{filho}$> $\biguplus$ pop'$_j$
Sendo pop'$_{j+1}$ = acrescenta(<lista$_{filho}$>, pop'$_j$, p), isto é:
mais-adaptado(pop'$_{j+1}$, $\biguplus_{i=1}^{j+1}$x$_i$, p).

Prova da correção do programa
Pela invariante tem-se:
mais-adaptado(pop$_{final}$, $\biguplus_{i=1}^{j*}$escolhe$_{\overline{op}_i}$(pop), p), onde j* é o último valor de j no programa abstrato.

Suponha φ(pop, p) = completa(pop, p).

Quer se provar ψ, isto é, ψ(população, pop, pop$_{final}$, p) = mais-adaptado(população, pop, p) $\wedge$ completa(população, p) $\wedge$ pop$_{final}$ $\subseteq$ população.

Pelo axioma 13,
mais-adaptado(população, pop, p) $\Rightarrow$ ($\forall$< i, v> $\in$ população)($\exists$< i', v'> $\in$ pop)(v$\geq_p$ v')
população = atualiza(pop, pop$_{final}$, p). (l_{11})

Seja < i, v> $\in$ população $\Rightarrow$ < i, v> $\in$ atualiza(pop, pop$_{final}$, p).
Pelo axioma 10 tem-se:
< i, v>$\in$pop$_{final}$$\vee$(< i, v>$\in$pop$\wedge$melhor(< i, v>,pop,|pop|-|pop$_{final}$|,p)).

(i) Suponha < i, v> $\in$ pop$_{final}$.

Pela invariante tem-se:
mais-adaptado(pop$_{final}$, $\biguplus_{i=1}^{j*}$ escolhe$_{\overline{op}_i}$(pop), p), isto é, $\forall$< i, v> $\in$ pop$_{final}$ $\exists$

$< i', v' > \in \biguplus_{i=1}^{j*}$ escolhe$_{\overline{op}_i}$(pop) $\wedge$ v $\geq_p$ v'.

Seja $< i', v' >$ tal par. Pelo axioma 8, $< i', v' > \in$ pop (*).

(ii) Suponha agora o outro caso:
$< i, v > \in$ pop $\wedge$ melhor($< i, v >$, pop, |pop| - |pop$_{final}$|, p).

Pelo axioma 14, $< i, v >$ está entre os |pop| - |pop$_{final}$| elementos de pop mais adaptados, o que significa que existem |pop$_{final}$| elementos de pop para os quais $< i, v >$ é mais adaptado.
Seja $< i', v' >$ um deles, então:
$< i', v' > \in$ pop e v$\geq_p$v' (**).
Por (*), (**) e pelo axioma 13,
mais-adaptado(pop$_{final}$, pop, p) (***)

Por φ(pop, p) e pelos axiomas 9 e 12 (o axioma 12 define que o tamanho pop não é alterado durante o procedimento), como população = atualiza(pop, pop$_{final}$, p), então:
completa(atualiza(pop, pop$_{final}$, p), p) (****).

Pelo axioma 10, pop$_{final} \subseteq$ população. (*****)

Por (***), (****) e (*****) segue que:
ψ(população, pop, pop$_{final}$, p), isto é, mais-adaptado(população, pop, p) $\wedge$ completa(população, p) $\wedge$ pop$_{final} \subseteq$ população.
*Procedimento AG Os candidatos à seqüência são:
pop$_1$ = incializa(p) (l$_2$)
pop$_{i+1}$ = reprodução(pop$_i$, p) (l$_3$)

Seja i $\in$ { 1, ..., n$_{ger}$}
pop$_{i+1}$ = reprodução(pop$_i$, p). (l$_3$)

Prova da invariante
Pelo axioma 0 a invariante é verdade na 1ª vez que o ponto é alcançado. Pelo axioma 9 e pela asserção de saída do procedimento de reprodução a invariante, *completa(pop, p)*, é sempre verdadeira. (*)

Prova da correção do programa
Pelos axiomas 1 e 2, tem-se:
geração(pop$_1$) = 1
geração(pop$_{i+1}$) = geração(pop$_i$) + 1,
isto é, geração(pop$_i$) = i, para i = 1, ..., n$_{ger}$.
Logo, geração(pop$_{n_{ger}}$) = n$_{ger}$.
Pelo axioma 3,
condição-término(pop$_{n_{ger}}$, p) = *true*, e
condição-término(pop$_i$, p) = *false*, para i < n$_{ger}$.
Logo, na saída da iteração (linha 5), pop = pop$_{n_{ger}}$ e,

pelo axioma 4, s $\in$ $pop_{n_{ger}}$. (**)

Pelo axioma 5, mais-adaptado(pop_{i+1}, pop_i, p). (***)

Por (*), (**) e (***) tem-se:

(($\exists$ n_{ger})($\exists$ < pop_1, pop_2, ..., $pop_{n_{ger}}$ >$\in$ POP)($\forall$ i)((1 $\leq$ i $\leq$ n_{ger} - 1) $\Rightarrow$ (completa(pop_{i+1}, p) $\wedge$ (pop_{i+1} = reprodução(pop_i, p)) $\wedge$ mais-adaptado(pop_{i+1}, pop_i, p) $\wedge$ s $\in$ $pop_{n_{ger}}$)))

Logo, pela construção da seqüência e pela condição de término o programa termina satisfazendo a pós-condição.

6.3 Exemplo da verificação dos axiomas

A seguir observa-se um exemplo da verificação dos axiomas para o problema proposto por Kosko, em [KOS 91], e implementado por [RAM 94]. O programa visa automatizar a trajetória de um veículo e é listado no Anexo A.

Para conduzir o veículo, o Algoritmo Genético manipula o ângulo de conversão das rodas para a esquerda (-30 a 0 graus) ou para a direita (0 a +30 graus). Estes movimentos farão o veículo girar para a esquerda ou direita, aproximando-se da garagem.

6.3.1 Identificação da estrutura do algoritmo

De acordo com o programa exemplo, observa-se a seguinte estrutura:

- Módulo de Avaliação
  - Função de Avaliação: função distância, determinada na linha 142.
- Módulo de Estruturação
  - Técnica de Representação: binária e o comprimento da cadeia é de 31 bits, determinado nas linhas 33 e 34.
  - Tamanho da População: no máximo 200 elementos, determinada nas linhas 33 e 34 e definida na linha 302. (opção do projetista)
  - Iterações desejadas: o número de gerações é defindo na linha 303.
  - Técnica de Inicialização: aleatória e é representada pelo procedimento `inicializa()` nas linhas 106 a 114.
  - Técnica de Ajuste: o ajuste é acumulado e é representado pelo procedimento `ajuste()` nas linhas 115 a 162.
  - Ténica de Seleção de Pais: técnica da roleta e é representada pelo procedimento `roleta()` nas linhas 163 a 186.
- Módulo de Reprodução
  - Técnica de Reprodução: reprodução por gerações.
  - Operadores: cruzamento em um ponto (linhas 200 a 209) e mutação (linhas 210 a 220).

Pode-se observar claramente no algoritmo algumas áreas (linhas de código) específicas da implementação. Ou seja, áreas de definições das variáveis utilizadas no programa (da linha 16 até 43) e rotinas de inicialização gráfica, cujo resultado do algoritmo é mostrar na tela o movimento do veículo (da linha 44 até 98 e da linha 223 até 266).

6.3.2 Identificação dos parâmetros do problema

Segundo o que foi definido no capítulo 6, um problema p$\in$ P é uma estrutura do tipo (f_{obj}, n_{ger}, n_{ind}, $lista_{taxa}$), onde os elementos são identificados no programa exemplo da seguinte maneira:

f_{obj}: é determinada pela função distância na linha 142.

n_{ger}: é determinado pela variável `numint` na linha 303.

n_{ind}: é determinado pela variável `numpop` na linha 302.

$taxa_{mut}$: é determinada pela variável `tm` na linha 305.

$taxa_{cross}$ e $taxa_{subst}$: não são defindas na entrada, pois o projetista optou por implementá-las arbitrariamente, ou seja, ambas são aplicadas com taxas de 100%. Com isso, o *crossover* é aplicado a todos os indivíduos e todos os elementos da população são substituídos.

Como entrada do algoritmo observou-se os seguintes parâmetros:

Coordenadas do veículo: o par (`td, to`) nas linhas 307 e 309.

Ângulo da direção do veículo (Fi): a variável `tf` na linha 311.

Ângulo das rodas do veículo (Teta): a variável `tt` na linha 313.

Como saída do algoritmo tem-se:

Cromossomo com melhor ajuste: o cromossomo `jaux` com o seu respectivo valor de ajuste `vetaju[jaux]` identificado na linha 263.

6.3.3 Identificação das funções e predicados

Agora, localiza-se no programa exemplo os predicados e funções definidos e representados pelo algoritmo abstrato encontrado no capítulo 6. As seguintes funções e predicados são encontrados no programa:

condição -término: é observada na linha 271, inserida no procedimento `controlador()`.

completa: é observada nos procedimentos, `inicializa()` (linha 111), `ajuste()` (linha 120), `roleta()` (linha 173) e `reprodução()` (linha 195).

recupera: é observada na linha 263, no procedimento `resultado()`, quando é mostrado (na tela) o cromossomo `jaux` com o melhor ajuste, `vetaju[jaux]`. (calculados nas linhas 150 a 155)

acrescenta e atualiza: são observados nas linhas 195 a 221, quando a população (`vetcro[ ][ ]`) é atualizada através das atribuições nas linhas 202, 203, 207, 208, 216 e 218.

gera$_{op}$: são observados nas linhas 200 a 209 (*crossover*) e 210 a 220 (mutação).

avalia: é representada pelo procedimento `ajuste()` nas linhas 115 a 162. O valor do ajuste é inversamente proporcional à distância do veículo à garagem. Então, calcula-se o ajuste de cada indivíduo subtraindo de 50000 (maior distância possível, segundo [RAM 94]) a distância encontrada. (linha 142)

inicializa: é representada pelo procedimento `inicializa()` nas linhas 106 a 114.

escolha$_{op}$: é representado pelo procedimento `roleta()` nas linhas 163 a 186 e é definido pelo trecho 173 a 185.

escolhe-operador: no algoritmo existem dois operadores genéticos e o projetista optou pela aplicação de ambos sem escolha, nas linhas 200 a 209 e 210 a 220.

6.3.4 Verificação dos axiomas

A seguir, o conjunto dos axiomas, definidos no Capítulo, é verificado de acordo com a identificação das suas cláusulas (componentes dos axiomas) no programa exemplo.

Axioma 0: Segundo o axioma 12 e a linha 111 do programa, o axioma 0 é verificado.

A função geração, geração(pop, p), determina o número de vezes que pop é atualizada.

Axioma 1: Segundo a observação acima, a 1^{a} atualização de pop é a inicialização. Portanto, o axioma 1 é satisfeito.

Axioma 2: A atualização de pop após a inicialização se dá somente pela reprodução nas linhas 202, 203, 207, 208, 216 e 218 do programa. E além disso, a reprodução sempre atualiza e, portanto, o axioma 2 é verificado.

Axioma 3: Na linha 271 aparece condição-término e verifica-se o axioma 3.

Axioma 4: Na linha 263 está a função recupera e o axioma 4 é verificado.

Axioma 7: As funções acrescenta e gera$_{op}$ estão definidas e entrelaçadas sem códigos separados. Neste caso pop é vazio, pois acrescenta substitui totalmente a população e é representado pelas atribuições nas linhas 202, 203, 207, 208, 216 e 218.

Axioma 8: A escolha propriamente dita é feita nas linhas 178 a 184 e o axioma é verificado na atribuição (na linha 182) quando a população nova (`vetnov[ ][ ]`) é selecionada a partir da antiga (`vetcro[ ][ ]`).

O algoritmo não possui as linhas l_{10} e l_{11} do programa abstrato, e a função atualiza é o resultado final da função acrescenta. O programa exemplificado é um caso particular do programa abstrato definido.

Axioma 9: Pela observação acima o axioma 9 é satisfeito nas linhas 202, 203, 207, 208, 216 e 218.

Axioma 10: Este axioma é trivialmente verificado pela técnica do algoritmo que gera exatamente n_{ind} elementos que substitui todos os elementos anteriores.

Os axiomas 12, 13, 14 e 15 são axiomas auxiliares, eles não são verificáveis pela implementação, pois não envolvem procedimentos implementados. São definições de funções e predicados auxiliares à definição dos demais axiomas.

O exemplo não satisfaz os axiomas 5, 6 e 11, isto significa que não é uma instância do AG. Realmente, esses axiomas não são essenciais, isto é, um Algoritmo Genético pode não satisfazer estes axiomas. A satisfação destes axiomas garante à cada geração uma população mais adaptada, isso significa que à cada geração a solução é melhorada. Ainda, não garante que a solução esteja próxima à solução ótima, pois podem estar convergindo a um ótimo local. Note que a terminação não depende destes axiomas.

6.4 Conclusão

Este capítulo mostrou a definição formal do algoritmo, propondo um método de desenvolvimento de Algoritmo Genético, segundo os estudos de [TOS 88]. Foi efetuada a definição dos domínios (conjuntos) envolvidos no processamento; a construção do diagrama sintático, que retrata graficamente a sintaxe das operações do algoritmo; e a descrição do algoritmo abstrato para o procedimento AG (mais geral) e para o procedimento reprodução.

Além disso, foi efetuada uma axiomatização com o propósito de dar a semântica do algoritmo, ou seja, ela define, formalmente, o funcionamento do algoritmo, mais especificamente das funções e procedimentos do algoritmo. Ao se verificar a correção do programa abstrato, faz-se necessário supor a asserção de entrada, provar as invariantes e usá-las para provar a asserção de saída. E isto possibilita o projetista de algoritmos uma maior segurança no desenvolvimento, porque para provar a correção de um Algoritmo Genético que satisfaça esse modelo só é necessário provar que os procedimentos satisfazem os axiomas. A prova da correção foi realizada para o caso geral, não necessita ser repetida. Os axiomas são as premissas que uma vez satisfeitos conferem correção à implementação. A exigência de uma solução a cada geração mais adaptada é uma condição de qualidade, mesmo que incompleta, no sentido que não evita ótimos locais. é talvez a condição mais simples de ser verificada. Na conclusão é apresentada como trabalho futuro uma sugestão de relaxamento na condição de mais adaptado.

No exemplo pode-se observar: a identificação da estrutura do algoritmo, a identificação e a relação dos parâmetros do problema, os parâmetros de entrada do algoritmo, a identificação das funções e predicados definidos no programa abstrato e a verificação dos axiomas em relação aos procedimentos do programa.

7 Análise da Complexidade do Algoritmo Genético

A complexidade de um algoritmo é uma função do tamanho do problema. Neste caso, há dois enfoques possíveis. Ou considera-se, para fins do cálculo da complexidade, o algoritmo como um esquema, isto é, o programa abstrato, ou uma instância dele. Em qualquer caso o esforço computacional depende de n_{ger}, n_{ind} e $lista_{taxa}$, o que muda é como esses parâmetros são considerados. Optou-se pelo segundo caso e então esses parâmetros serão considerados como função de n. Pela sua importância, entretanto, eles permanecerão como parâmetros da complexidade do AG (C(AG)).

Agora, vai-se definir a complexidade do programa abstrato AG em função das complexidades dos procedimentos que o compõem.

Um Algoritmo Genético tem como parâmetros, além da entrada do problema o qual pretende-se resolver, n_{ger} (número de gerações), n_{ind}(número de indivíduos), que são determinantes na complexidade do algoritmo resultante. Além disso, um Algoritmo Genético tem como parâmetro $lista_{taxa}$ que determina o número médio de substituições e o número médio de escolhas de cada operador, o que corresponde a um número de elementos gerados nessa reprodução. O funcionamento do Algoritmo Genético tem uma componente aleatória regida pelas taxas de aplicação dos operadores genéticos ($lista_{taxa}$) e, em vista disso, refere-se, de modo natural, a complexidade deste algoritmo como complexidade média.

7.1 Complexidade total

A complexidade média de um Algoritmo Genético não depende somente do tamanho da entrada n (por exemplo: no problema do caixeiro viajante, n é o número de cidades que devem ser percorridas, ver [AGU 96]), mas também dos parâmetros n_{ger}, n_{ind} e $lista_{taxa}$ que pertencem à instância do algoritmo (ver definição dos domínios, capítulo 6). A complexidade do procedimento AG, C(AG), tem portanto esses 4 parâmetros, e analisando este procedimento conclui-se facilmente que:

Supondo que as linhas 2 a 4, do procedimento AG, são executadas n_{ger} vezes (pela condição de término), tem-se a seguinte equação de complexidade:

$$\begin{aligned} C(\text{ AG})(\text{ n, } n_{ger}, n_{ind}, lista_{taxa}) = \quad & C(\text{ incializa})(\text{ n, } n_{ind}) + \\ & \{\ n_{ger}.\ (\ C(\text{ condição-término})(\text{ n}) + \\ & C(\text{ reprodução})(\text{ n, } n_{ind}, lista_{taxa}))\} + \\ & C(\text{ recupera})(\text{ n, } n_{ind}) \end{aligned} \tag{7.1}$$

Faz-se necessário, agora, calcular C(inicializa), C(condição-término), C(reprodução) e C(recupera). Os procedimentos inicializa, condição-término e recupera são procedimentos simples, cuja complexidade será determinada pela implementação destes. O procedimento reprodução é o elemento mas importante do Algoritmo Genético, e foi definido também como um programa abstrato. Assim, C(reprodução) será calculada em função da complexidade das funções (procedimentos) que o compõem.

7.2 Complexidade da reprodução

A complexidade do procedimento reprodução é posta somente em função dos parâmetros n, n_{ind} e $lista_{taxa}$, visto que ela não depende do parâmetro n_{ger}, pois o caso a ser avaliado é de executar *reprodução* uma única vez. Examinando-se o algoritmo verifica-se facilmente que:

$$C(\text{reprodução})(n,n_{ind},lista_{taxa})=L \cdot \sum_{i \in OP} \frac{1}{k}(C(escolhe_i)(n_{ind})+C(mod_i)(n)) \quad (7.2)$$

onde:

L que é calculado a seguir, é o número de vezes que o enquanto é executado; e,

k é o número de operadores genéticos existentes e/ou utilizados no problema(constante do algoritmo, ver definição dos domínios, capítulo 6).

A complexidade de $escolhe_i$ será determinada pela sua implementação. Sendo mod_i um procedimento dependente de operação, naturalmente corresponderá uma complexidade dependente de operação.

Exemplifica-se as complexidades dos módulos (C(mod_i)(n)) mais utilizados, cruzamento e mutação, como segue:

C(mod_{cross})(n) = C($gera_{cross}$)(n) + 2 · C(avalia)(n) + C(acrescenta)(2)
C(mod_{mut})(n) = C($gera_{mut}$)(n) + C(avalia)(n) + C(acrescenta)

Generalizando, tem-se:

$$C(\ mod_i)(\ n) = C(\ gera_i)(\ n) + a_i \cdot C(\ avalia)(\ n) + C(\ acrescenta)(\ a_i) \quad (7.3)$$

onde: a_i é o nº de elementos gerados por vez pela $gera_i$, também chamado de grau de participação do operador no processo de reprodução.

Examinando a equação 7.2, verifica-se que L é a peça fundamental, faz-se necessário, então, calcular L.

7.3 Cálculo do número médio de execuções do enquanto

Pode-se calcular o inteiro L a partir do número de vezes que cada operador genético será executado. Assim, tem-se:

$$L = \sum_{i \in OP} g_i$$

onde, g_i é o nº de vezes que $gera_i$ é executada em todo o processamento (uma execução de *reprodução*).

Por sua vez, chega-se a g_i sabendo-se que:

$$g_i = \left\lceil \frac{n_i}{a_i} \right\rceil$$

onde:

n_i calculado a seguir, é o nº total médio de elementos gerados pelo mod_i em toda execução do enquanto e,

a_i relembrando, é o nº de elementos gerados em cada execução de $gera_i$.

Mas, como determinar o nº total de elementos gerados por um dado módulo em todo o processo, se cada operador possui a sua probabilidade de ocorrência, que podem não ser complementares, ou seja, somando-as não necessariamente valem 1? E, ainda, cada operador tem um grau de participação diferente no montante da população ($a_1 \neq a_2 \neq \ldots \neq a_K$).

Para isso, é necessário chegar a uma expressão de normalização de probabilidades, levando em conta os graus de participação de cada operador. As taxas de cada operador genético são usualmente definidas como um número racional, assim serão consideradas da seguinte maneira:

$\forall$ i $\in$ OP, $taxa_i = \frac{x_{i1}}{x_{i2}}$, onde i = 1, ..., K.

Reduzindo as taxas ao mesmo denominador obtém-se o seguinte:

$$\frac{x_{11} \cdot x_{22} \cdot \ldots \cdot x_{K2}}{x_{12} \cdot x_{22} \cdot \ldots \cdot x_{K2}}, \frac{x_{21} \cdot x_{12} \cdot \ldots \cdot x_{K2}}{x_{12} \cdot x_{22} \cdot \ldots \cdot x_{K2}}, \ldots, \frac{x_{K1} \cdot x_{12} \cdot \ldots \cdot x_{K2}}{x_{12} \cdot x_{22} \cdot \ldots \cdot x_{K2}}.$$

Levando em consideração os valores a_i (grau de participação) dos operadores, tem-se:

$$a_1 \cdot \frac{x_{11} \cdot x_{22} \cdot \ldots \cdot x_{K2}}{x_{12} \cdot x_{22} \cdot \ldots \cdot x_{K2}}, a_2 \cdot \frac{x_{21} \cdot x_{12} \cdot \ldots \cdot x_{K2}}{x_{12} \cdot x_{22} \cdot \ldots \cdot x_{K2}}, \ldots, a_K \cdot \frac{x_{K1} \cdot x_{12} \cdot \ldots \cdot x_{K2}}{x_{12} \cdot x_{22} \cdot \ldots \cdot x_{K2}}.$$

Então a soma, $a_1 \cdot x_{11} \cdot x_{22} \cdot \ldots \cdot x_{K2} + a_2 \cdot x_{21} \cdot x_{12} \cdot \ldots \cdot x_{K2} + \ldots + a_K \cdot x_{K1} \cdot x_{12} \cdot \ldots \cdot x_{K2}$, representa o valor relativo às probabilidades normalizadas do total de indivíduos gerados pelos operadores, ou seja, o operador 1, participa com $\frac{a_1 \cdot x_{11} \cdot x_{22} \cdot \ldots \cdot x_{K2}}{a_1 \cdot x_{11} \cdot x_{22} \cdot \ldots \cdot x_{K2} + a_2 \cdot x_{21} \cdot x_{12} \cdot \ldots \cdot x_{K2} + \ldots + a_K \cdot x_{K1} \cdot x_{12} \cdot \ldots \cdot x_{K2}}$ dos indivíduos gerados. Assim, obtem-se:

$$\overline{taxa_i} = \frac{a_i . x_{i1} . \prod_{j \neq i}^{k} x_{j2}}{\sum_{i=1}^{k} (a_i . x_{i1} . \prod_{j \neq i}^{k} x_{j2})}$$

onde, $\overline{taxa_i}$ é a taxa normalizada para o operador i.

Para o cálculo do inteiro n_i tem-se:

$$n_i = \lceil m \cdot \overline{taxa_i} \rceil$$

onde, m é o nº total de substituições efetuadas, e é calculado da seguinte forma:

$$m = n_{ind} \ . \ taxa_{subs}$$

Então, o cálculo de n_i é dado da seguinte forma:

$$n_1 = \lceil m \cdot \overline{taxa_1} \rceil, \text{ com}$$

$$\overline{taxa_1} = \frac{a_1 \cdot x_{11} \cdot x_{22} \cdot \ldots \cdot x_{K2}}{a_1 \cdot x_{11} \cdot x_{22} \cdot \ldots \cdot x_{K2} + a_2 \cdot x_{21} \cdot x_{12} \cdot \ldots \cdot x_{K2} + \ldots + a_K \cdot x_{K1} \cdot x_{12} \cdot \ldots \cdot x_{K2}}$$

$$n_2 = \lceil m \cdot \overline{taxa_2} \rceil, \text{ com}$$

$$\overline{taxa_2} = \frac{a_2 \cdot x_{21} \cdot x_{12} \cdot \ldots \cdot x_{K2}}{a_1 \cdot x_{11} \cdot x_{22} \cdot \ldots \cdot x_{K2} + a_2 \cdot x_{21} \cdot x_{12} \cdot \ldots \cdot x_{K2} + \ldots + a_K \cdot x_{K1} \cdot x_{12} \cdot \ldots \cdot x_{K2}}$$

$$n_k = \lceil m \cdot \overline{taxa_K} \rceil, \text{ com}$$

$$\overline{taxa_K} = \frac{a_K \cdot x_{K1} \cdot x_{12} \cdot \ldots \cdot x_{K2}}{a_1 \cdot x_{11} \cdot x_{22} \cdot \ldots \cdot x_{K2} + a_2 \cdot x_{21} \cdot x_{12} \cdot \ldots \cdot x_{K2} + \ldots + a_K \cdot x_{K1} \cdot x_{12} \cdot \ldots \cdot x_{K2}}$$

O valor de L é calculado como:

$$L = \sum_{i \in OP} g_i = \sum_{i \in OP} \left\lceil \frac{\lfloor m \cdot \overline{taxa_i} \rfloor}{a_i} \right\rceil$$

Portanto,

$$L = \sum_{i \in OP} \left\lceil \frac{\lfloor n_{ind} \cdot taxa_{subs} \cdot \overline{taxa_i} \rfloor}{a_i} \right\rceil \tag{7.4}$$

A seguir é exemplificado o cálculo de L, dada sua importância para o cálculo da complexidade.

Suponha OP = { op_1, op_2, op_3}.

Então, as probabilidades normalizadas de cada operador são as seguintes:

$$\overline{taxa_{op_1}} = \frac{a_1.x_{11}.x_{22}.x_{32}}{a_1.x_{11}.x_{22}.x_{32} + a_2.x_{21}.x_{12}.x_{32} + a_3.x_{31}.x_{12}.x_{22}}$$

$$\overline{taxa_{op_2}} = \frac{a_2.x_{21}.x_{12}.x_{32}}{a_1.x_{11}.x_{22}.x_{32} + a_2.x_{21}.x_{12}.x_{32} + a_3.x_{31}.x_{12}.x_{22}}$$

$$\overline{taxa_{op_3}} = \frac{a_3.x_{31}.x_{12}.x_{22}}{a_1.x_{11}.x_{22}.x_{32} + a_2.x_{21}.x_{12}.x_{32} + a_3.x_{31}.x_{12}.x_{22}}$$

Logo, o número médio de elementos gerados por cada operador é dado por:

$$n_{op_1} = \lceil m \cdot \overline{taxa_{op_1}} \rceil$$

$$\mathrm{n}_{op_2} = \left\lceil \mathrm{m} \cdot \ \overline{\mathrm{taxa}_{op_2}} \ \right\rceil$$

$$\mathrm{n}_{op_3} = \left\lceil \mathrm{m} \cdot \ \overline{\mathrm{taxa}_{op_3}} \ \right\rceil$$

Portanto, pela equação 7.4, o valor de L é:

$$\mathrm{L} = \left\lceil \frac{\mathrm{m} \cdot \ \overline{\mathrm{taxa}_{op_1}}}{a_1} \right\rceil \ + \left\lceil \frac{\mathrm{m} \cdot \ \overline{\mathrm{taxa}_{op_2}}}{a_2} \right\rceil \ + \left\lceil \frac{\mathrm{m} \cdot \ \overline{\mathrm{taxa}_{op_3}}}{a_3} \right\rceil$$

Agora, particulariza-se o cálculo de L com o objetivo de deixar mais claro o entendimento, tem-se três operadores de reprodução: crossover, mutação e outro; então, OP = { crossover, mutação, outro}.

Suponha ainda que: $\mathrm{n}_{ind} = 50$; $\mathrm{taxa}_{subs} = 0.6$ (logo, $\mathrm{m} = \mathrm{n}_{ind} \ . \ \mathrm{taxa}_{subs} = 30$); $\mathrm{taxa}_{cross} = \frac{1}{3}$ (isto é, o indivíduo tem $\frac{1}{3}$ de chance de ser operado por *crossover*), $\mathrm{taxa}_{mut} = \frac{1}{10}$ (análogo ao taxa_{cross}) e $\mathrm{taxa}_{outro} = \frac{1}{5}$ (idem).

Recordando, $\mathrm{a}_{cross} = 2$ e $\mathrm{a}_{mut} = 1$; e, supondo, $\mathrm{a}_{outro} = 1$.

Então, as probabilidades normalizadas de cada operador são as seguintes:

$\overline{\mathrm{taxa}_{cross}} = \frac{2.1.10.5}{(2.1.10.5)+(1.1.3.5)+(1.1.3.10)} = \frac{100}{145}$

$\overline{\mathrm{taxa}_{mut}} = \frac{1.1.3.5}{(2.1.10.5)+(1.1.3.5)+(1.1.3.10)} = \frac{15}{145}$

$\overline{\mathrm{taxa}_{outro}} = \frac{1.1.3.10}{(2.1.10.5)+(1.1.3.5)+(1.1.3.10)} = \frac{30}{145}$

Logo, o número médio de elementos gerados por cada operador é dado por:

$$\mathrm{n}_{cross} = \left\lceil 30 \cdot \ \frac{100}{145} \right\rceil = 21$$

$$\mathrm{n}_{mut} = \left\lceil 30 \cdot \ \frac{15}{145} \right\rceil = 4$$

$$\mathrm{n}_{outro} = \left\lceil 30 \cdot \ \frac{30}{145} \right\rceil = 7$$

Pela equação 7.4, o valor de L é:

$$\mathrm{L} = \left\lceil \frac{21}{2} \right\rceil + \left\lceil \frac{4}{1} \right\rceil + \left\lceil \frac{7}{1} \right\rceil = 22$$

7.4 Conclusão

Neste capítulo foi desenvolvida uma equação de complexidade, partindo de um maior nível de abstração, ou seja, do procedimento mais geral AG (equação 7.1), depois para a análise da complexidade do procedimento reprodução (equação 7.2), propondo o estudo em separado das partes de sua complexidade, a partir do cálculo de L (equação 7.4), e depois do módulo específico de geração (equação 7.3). A partir deste estudo pode-se determinar alguns conselhos e conclusões mostrados a seguir.

Pode-se afirmar que a complexidade do procedimento reprodução, e principalmente os valores n_i, são fundamentais para o cálculo da complexidade total, pois a estimativa do número de vezes que é executado o enquanto influi diretamente na complexidade esperada do procedimento. A complexidade é fundamentalmente dada por n_i e n_{ger}, sendo n_i calculado a partir dos parâmetros n_{ind} e lista_{taxa}.

Ao analisar as equações 7.2 e 7.4 pode-se dizer que ao se utilizarem módulos com complexidades aproximadas (mesma ordem de complexidade) deve-se determinar taxas maiores (lista_{taxa}) para operadores com valores grandes para a_i, acarretando um valor menor para L, ou seja, permitir que os procedimentos que geram mais elementos por vez sejam melhor utilizados. Caso contrário, se forem utilizados módulos com complexidades diferentes, se poderia aconselhar a utilização de operadores com complexidades menores, através do aumento de suas respectivas taxas, ou se for o caso, utiliza-se operadores com complexidades maiores e com valores grandes para a_i, aumentando-se a taxas destes. Neste último caso, apesar de se promover o uso de módulos com complexidades maiores, se poderia ter compensação no número de elementos gerados por este, ou seja, se precisaria menos iterações (menor L) para ser encerrada a reprodução.

Aconselha-se aos projetistas de algoritmos utilizarem operadores genéticos que tenham diferentes taxas, $\mathrm{taxa}_{op_1} \neq \mathrm{taxa}_{op_2} \neq \ldots \neq \mathrm{taxa}_{op_K}$. é fácil observar na equação 7.4 que, ao terem taxas iguais, os operadores, em média, dividem igualmente entre si o trabalho de gerar indivíduos e impossibilitam a exploração do funcionamento de determinados operadores que possam ter complexidades menores. Uma alternativa para explorar os módulos com menor complexidade seria através do aumento das respectivas taxas.

Ainda, sugere-se a utilização de $\mathrm{a}_i > 1$, visto que, operadores que geram apenas um indivíduo por vez degradam o desempenho do algoritmo. A observação, refere-se diretamente ao número de vezes que o enquanto do procedimento de reprodução é executado (ver equação 7.4), assim, os operadores gerando apenas um elemento por vez elevam a complexidade do algoritmo superestimando aqueles que possuem maior complexidade, ou seja, os operadores serão utlizados em média o mesmo número de vezes permitindo que operadores mais *complexos* sejam utilizados frequentemente.

Pode-se observar na equação 7.4, que o pior nível de degradação do desempenho seria a ocorrência das duas situações simultaneamente, $\text{taxa}_{op_1} = \text{taxa}_{op_2} = \ldots = \text{taxa}_{op_K}$ e usar $a_i = 1$, onde levaria executar o enquanto m vezes. Ou seja, os valores n_i destes operadores seria $\frac{m}{k}$, e sendo $L = \sum_{i \in OP} \frac{n_i}{a_i}$, implicaria em:

$$L = \frac{\frac{m}{k}}{1} + \frac{\frac{m}{k}}{1} + \ldots + \frac{\frac{m}{k}}{1}, \text{K somas. Então: } L = m$$

Da equação 7.1 vê-se que a componente mais importante da complexidade de um Algoritmo Genético é C(reprodução), além, obviamente de n_{ger}. Examinando a equação 7.2 nota-se que o somatório tem K termos e está sendo multiplicado por $\frac{1}{k}$. Os procedimentos escolhe_i e mod_i dependem do operador, isto é, são pontos chaves na equação.

Além desses fatores mais óbvios, tem-se L como fator multiplicador. L deve ser um fator bem ponderado, para estudá-lo recorre-se à equação 7.4. Do exame conclui-se que $n_{ind} \cdot \text{taxa}_{subs}$ é o fator mais importante. Sugere-se que $n_{ind} \cdot \text{taxa}_{subs}$ não exceda às funções lineares a n, isto é, o número de indivíduos gerados/substituídos em cada iteração seja no máximo proporcional ao problema.

Se o objetivo é um algoritmo polinomial $n_{ger} \cdot n_{ind} \cdot \text{taxa}_{subs}$ deve ser polinomial em n. Recomenda-se avaliar esses três parâmetros com cuidado para atingir uma exatidão satisfatória sem sobrecarregar a complexidade.

Um estudo dos efeitos desses três parâmetros na exatidão da solução é desejável, sem esquecer que estes devem ser postos como função de n.

8 Conclusão Final

O tema abordado possibilitou o estudo de todas as etapas necessárias para a criação de um modelo para o desenvolvimento de algoritmos, tendo consciência das definições formais de algoritmos e problemas, levando em consideração os aspectos de computabilidade e complexidade computacional e algorítmica. Além disso, criou-se uma maneira metodizada para análise da complexidade de um Algoritmo Genético, permitindo, assim, conhecer e/ou estimar o grau de viabilidade da utilização do algoritmo em problemas de otimização. O método foi criado com o propósito de sistematizar e qualificar o desenvolvimento de Algoritmos Genéticos, utilizando-se um conjunto de axiomas que dão a semântica do algoritmo abstrato cuja correção e terminação do programa foi devidamente demonstrada a partir destes.

Segundo [REI 89], algoritmos randômicos oferecem uma alternativa muito pragmática (aos algoritmos determinísticos). Além de serem mais simples que seus *parentes* determinísticos, eles têm usualmente constantes muito menores. Além disso, eles proporcionam uma ligação entre os desenvolvedores de algoritmos para modelos abstratos e suas implementações realísticas em arquiteturas plausíveis.

Inicialmente (no capítulo 2), foi apresentada a descrição dos tipos de problemas, com a definição formal de problema introduzindo conceitos muito importantes para a Teoria da Computabilidade e a definição de problema no ponto de vista computacional como problemas de otimização e decisão.

O capítulo 3 é resultado de um estudo sobre os tipos de complexidade e suas principais medidas de complexidade – inclusive para algoritmos randômicos – que introduz as classes de complexidade P, NP, NP-Completo, NPI, NP-Difícil e as relações entre as principais classes.

No capítulo 4 mostrou-se a definição de algoritmo aproximativo e das medidas de qualidade para este tipo de algoritmo e suas categorias, as classes de problemas de otimização e o conceito de redutibilidade.

O capítulo 5 apresentou um estudo abrangente sobre os fundamentos dos Algoritmos Genéticos englobando a história, a fundamentação, o funcionamento, a constituição dos módulos (composição), as opções de otimização, e a aplicabilidade dos Algoritmos Genéticos. Este capítulo teve destaque pela importância no auxílio da formalização e na composição do método de desenvolvimento de Algoritmos Genéticos.

A proposta do método de desenvolvimento de Algoritmos Genéticos propriamente foi apresentado no capítulo 6, e está baseado na formalização de um MDA em [TOS 88], através de um programa abstrato e um conjunto de axiomas, que dão a semântica ao algoritmo cujas funções e procedimento são definidas sintaticamente pelo diagrama sintático.

O desenvolvimento de uma equação de complexidade para o Algoritmo Genético, apresentada no capítulo 7, foi embasado no estudo formal efetuado, permitindo a análise da complexidade do algoritmo abstrato.

Quanto ao método de desenvolvimento de Algoritmos Genéticos, pôde-se concluir que sua utilização fornece subsídio muito importante ao estudo da viabilidade do uso de Algoritmos Genéticos para a solução de problemas. No que tange à análise da complexidade deste tipo de algoritmo, pode-se saber, de antemão o custo médio das operações presentes no algoritmo abstrato. O desenvolvimento do al-

goritmo pode ser sistematizado e o grau de otimização do algoritmo é melhorado, assegurando-se que o conjunto de axiomas resultantes da formalização são suficientes para a semântica e para o funcionamento do algoritmo, o que é demonstrado no capítulo 6.

Dois fatores são importantes, para um algoritmo aproximativo, a exatidão e a complexidade. Infelizmente, são fatores conflitantes, pois quanto maior a exatidão, pior (mais alta) é a complexidade, e vice-versa. Assim, um estudo da qualidade de um Algoritmo Genético, considerado um algoritmo aproximativo, só estaria completa com a consideração destes dois fatores. Este trabalho levantou os pontos importantes para o estudo da complexidade. Um estudo da exatidão de resposta e sua relação com parâmetros do Algoritmo Genético Abstrato faz-se necessário para uma avaliação da qualidade. A exigência de uma solução a cada geração mais adaptada é uma condição de qualidade, mesmo que incompleta, no sentido que não evita ótimos locais. é talvez a condição mais simples de ser verificada.

A condição mais-adaptado(pop_{i+1}, pop_i, p) na asserção de saída da reprodução indica que a cada iteração a solução atingida é melhor (está mais adaptada). Entretanto, às vezes não é conveniente ter-se esta condição, pois pode-se estar aproximando de um ótimo local que pode estar longe do ótimo global. Então é conveniente numa iteração ir de uma solução melhor para uma mais adaptada para sair da proximidade do ótimo local e poder aproximar-se do ótimo global. Nestes casos a condição mais-adaptado(reprodução(pop_i, p), p) nem sempre é atingido e a seqüência pop_1, pop_2, ..., $\text{pop}_{n_{ger}}$ de que fala a asserção de saída de AG não satisfaz mais-adaptado(pop_{i+1}, pop_i, p). Como aprimoramento do trabalho, sugere-se que a condição (mais-adaptado(pop_{i+1}, pop_i, p)), e somente esta, possa ser relaxada e o resultado (o algoritmo) continuará sendo um Algoritmo Genético.

Na figura 8.1, pode-se visualizar um dos objetivos que está por trás de todo este estudo formal sobre Algoritmos Genéticos. A figura apresenta o procedimento AG, composto pelas suas características próprias e por módulos que são *encaixados* ou *embutidos* ao algoritmo inicial. Esta característica permite que sejam utilizados outros operadores não necessariamente genéticos, mas qualquer tipo de rotina de busca de solução, ótima ou aproximativa, com suas respectivas complexidades.

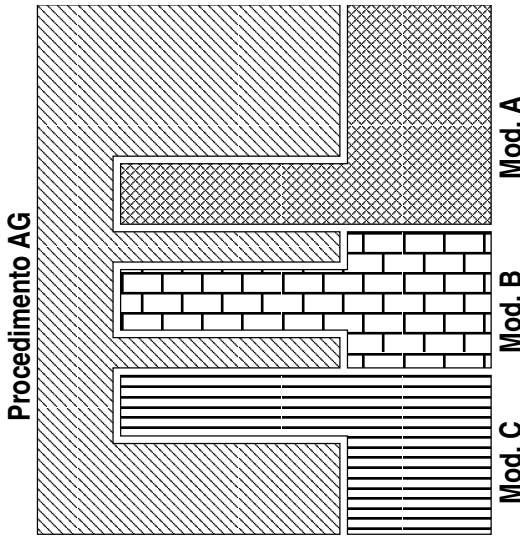

Figura 8.1 – Visão da Modularidade do Método

Assim permite-se um estudo mais dinâmico do Algoritmo Genético. Os módulos que serão aproveitados como operadores passam a contribuir com suas soluções como sendo indivíduos para a população daquela geração. A dinamicidade é favorecida pela utilização de rotinas já conhecidas, amplamente divulgadas e estudadas pelos projetistas, para compor esta modularização do Algoritmo Genético.

Por isso, como aprimoramento do trabalho, é lógico que seria interessante o estudo das possibilidades de hibridização do Algoritmo Genético, o que ficaria viável devido a generalidade do estudo da complexidade efetuado. Deixou-se em aberto, mas indicado para futura verificação, a complexidade dos módulos genéricos e sugeriu-se interpretações para certas estruturas, como: condição de término do algoritmo e operadores opcionais aos freqüentemente encontrados na literatura. Ainda, como trabalho futuro sugere-se a definição das condições que um Algoritmo Genético deve satisfazer para atingir um patamar de qualidade definido.

Anexo 1 Programa Exemplo

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <time.h>
4.  #include <conio.h>
5.  #include <math.h>
6.  #include <graphics.h>

7.  #define ESC 0x1b

8.  void grafico (void);
9.  void carro_bola (void);
10. void inferencia (void);
11. void resultado (void);
12. void inicializa (void);
13. void ajuste (void);
14. void roleta (void);
15. void reproducao (void);

16. #define DIM_X getmaxx()
17. #define DIM_Y getmaxy()
18. #define GRAF_X 260
19. #define GRAF_Y 80
20. #define EX 2
21. #define EY 1.33
22. #define GAP 15
23. #define R 2

24. int dy;
25. int conta;
26. int contador;
27. double ymin = 0.0, ymax = 1.0;
28. char str[40];
29. int xi, xf, yi, yf;
30. int switchpages;
31. int visualpage = 1;
32. int passo_desloc = 25;
33. int vetcro[100][31];
34. int vetnov[100][31];
35. int graf;
36. double vetaju[100];
37. double vetacu[100];
38. double teta_pos[200];
39. int jaux, numpop, numint;
40. double x_obtido = 0, y_obtido = 0, fi_obtido = 0, teta_obtido = 0;
41. double mutacao;
```

```
42. double alpha_x;
43. double alpha_beta[7];

44. void
45. ini_graf ()
46. {
47.   if (graf > 0)
48.     {
49.       int graphdriver, graphmode;
50.       detectgraph (&graphdriver, &graphmode);
51.       graphdriver = VGA;
52.       graphmode = VGAMED;
53.       initgraph (&graphdriver, &graphmode, "");
54.       setbkcolor (BLACK);
55.       setcolor (WHITE);
56.       switch (graphmode)
57.       {
58.       case VGAHI:
59.         switchpages = 0;
60.         break;
61.       case VGAMED:
62.         switchpages = 1;
63.         break;
64.       case VGALO:
65.         switchpages = 1;
66.         break;
67.       }
68.       if (switchpages)
69.       {
70.         setvisualpage (0);
71.         setactivepage (0);
72.       }
73.     }
74. }

75. void
76. grafico ()
77. {
78.   if (graf > 0)
79.     {
80.       teta_pos[contador] = teta_obtido;
81.       contador++;
82.       xi = DIM_X / 2 + 90;
83.       yi = 0;
84.       xf = DIM_X;
85.       yf = DIM_Y;
86.       setviewport (xi, yi, xf, yf, 0);
87.       clearviewport ();
```

```
88.       setcolor (WHITE);
89.       line (10, 1, 10, yf);
90.       line (10, yf / 2, xf - 30, yf / 2);
91.       outtextxy (3, yf / 2, "0");
92.       outtextxy (10, 0, "+30");
93.       outtextxy (10, yf - 7, "-30  CROMOSSOMOxINTERACOES");
94.       setcolor (LIGHTRED);
95.       for (conta = 0; conta < 200; conta++)
96.         line (11 + conta, yf / 2,
                 11 + conta, yf / 2 - teta_pos[conta] * 5.5);
97.     }
98. }

99. void
100. inferencia ()
101. {
102.   ajuste ();
103.   roleta ();
104.   reproducao ();
105. }

106. void
107. inicializa ()
108. {
109.   int j, k;
110.   srand (0);
111.   for (j = 0; j < numpop; j++)
112.     for (k = 0; k < 31; k++)
113.       vetcro[j][k] = (rand () % 2);
114. }

115. void
116. ajuste ()
117. {
118.   int j, k, cont;
119.   double tetaux, fiobt, xobt, yobt, ajuaux;
120.   for (j = 0; j < numpop; j++)
121.     {
122.       cont = 0;
123.       tetaux = 0;
124.       fiobt = 0;
125.       vetaju[j] = 0;
126.       vetacu[j] = 0;
127.       for (k = 0; k < 30; k++)
128.         cont += vetcro[j][k];
129.       if (vetcro[j][30] == 0)
130.         tetaux = cont;
131.       else
```

```
132.         tetaux = -cont;
133.       fiobt = fi_obtido + tetaux;
134.       xobt = x_obtido + R * cos ((fiobt * M_PI) / 180);
135.       yobt = y_obtido + R * sin ((fiobt * M_PI) / 180);
136.       xobt -= 50;
137.       if (xobt < 0)
138.         xobt *= -1;
139.       yobt -= 100;
140.       if (yobt < 0)
141.         yobt *= -1;
142.       vetaju[j] = 50000 - ((xobt * xobt) + (yobt * yobt));
143.       if (j > 0)
144.         vetacu[j] = vetacu[j - 1] + vetaju[j];
145.       else
146.         vetacu[j] = vetaju[j];
147.     }
148.   cont = 0;
149.   ajuaux = 0;
150.   for (j = 0; j < numpop; j++)
151.     if (ajuaux <= vetaju[j])
152.       {
153.         ajuaux = vetaju[j];
154.         jaux = j;
155.       }
156.   for (k = 0; k < 30; k++)
157.     cont += vetcro[jaux][k];
158.   if (vetcro[jaux][30] == 0)
159.     teta_obtido = cont;
160.   else
161.     teta_obtido = -cont;
162. }

163. void
164. roleta ()
165. {
166.   long tempo;
167.   double aleatorio, randomico;
168.   int j, k, l, conov, conta;
169.   time (&tempo);
170.   srand (tempo);
171.   conta = 0;
172.   conov = 0;
173.   for (l = 0; l < numpop; l++)
174.     {
175.       randomico = pow (rand (), 2);
176.       aleatorio = fmod (randomico, vetacu[numpop - 1]);
177.       for (j = 0; j < numpop; j++)
178.       if (vetacu[j] >= aleatorio)
```

```
179.         {
180.           conov++;
181.           for (k = 0; k < 31; k++)
182.             vetnov[conov][k] = vetcro[j][k];
183.           j = numpop;
184.         }
185.     }
186. }

187. void
188. reproducao ()
189. {
190.   long tempo;
191.   double aleatorio;
192.   int j, k;
193.   time (&tempo);
194.   srand (tempo);
195.   for (j = 0; j < numpop; j += 2)
196.     {
197.       aleatorio = fmod (rand (), 31);
198.       if (aleatorio > 31)
199.       printf ("\n%f", aleatorio);
200.       for (k = 0; k < aleatorio; k++)
201.         {
202.           vetcro[j][k] = vetnov[j + 1][k];
203.           vetcro[j + 1][k] = vetnov[j][k];
204.         }
205.       for ( = aleatorio; k < 31; k++)
206.         {
207.           vetcro[j][k] = vetnov[j][k];
208.           vetcro[j + 1][k] = vetnov[j + 1][k];
209.         }
210.       for (k = 0; k < 31; k++)
211.         {
212.           aleatorio = fmod (rand (), 1000);
213.           if (aleatorio <= mutacao)
214.            {
215.              if (vetcro[j][k] == 1)
216.                vetcro[j][k] = 0;
217.              else
218.                vetcro[j][k] = 1;
219.            }
220.         }
221.     }
222. }

223. void
224. carro_bola ()
```

```
225. {
226.   if (graf > 0)
227.     {
228.       xi = 0;
229.       yi = 0;
230.       xf = 200 * EX;
231.       yf = 200 * EY;
232.       setviewport (xi, yi, xf, yf, 0);
233.       setcolor (LIGHTBLUE);
234.       rectangle (0, 0, 200 * EX, 198 * EY);
235.       rectangle (100 * EX - 10, 198 * EY, 100 * EX + 8,
                                                206 * EY);
236.       setcolor (BLACK);
237.       line (100 * EX - 10, 198 * EY, 100 * EX + 8,
                                           198 * EY);
238.       setcolor (LIGHTRED);
239.       if (x_obtido > 0 && x_obtido < 100 && y_obtido > 0
                                              && y_obtido < 100)
240.         circle (2 * x_obtido * EX, 2 * y_obtido * EY, 7);
241.     }
242. }

243. void
244. resultado ()
245. {
246.   if (graf > 0)
247.     {
248.       int y = 5;
249.       int desl = 13;
250.       xi = 0;
251.       yi = DIM_Y - GRAF_Y - 2 * GAP + 30;
252.       xf = DIM_X;
253.       yf = DIM_Y;
254.       setviewport (xi, yi, xf, yf, 0);
255.       clearviewport ();
256.       setcolor (WHITE);
257.       sprintf (&str[0], "Populacao:%d", numpop);
258.       outtextxy (5, y += desl + desl, str);
259.       sprintf (&str[0], "Geracoes: %d   X:%g     Y:%g", contador,
                                                  x_obtido, y_obtido);
260.       outtextxy (5, y += desl, str);
261.       sprintf (&str[0], "Fi: %g   Teta:%g", fi_obtido, teta_obtido);
262.       outtextxy (5, y += desl, str);
263.       sprintf (&str[0], "Cromossomo: %d   Ajuste:%g", jaux,
                                                    vetaju[jaux]);
264.       outtextxy (5, y += desl, str);
265.     }
266. }
```

```
267. void
268. controlador ()
269. {
270.   int nger = 0;
271.   while (nger < numint)
272.     {
273.       nger++;
274.       resultado ();
275.       grafico ();
276.       carro_bola ();
277.       inferencia ();
278.       fi_obtido += teta_obtido;
279.       x_obtido = x_obtido + R * cos ((fi_obtido * M_PI) / 180);
280.       y_obtido = y_obtido + R * sin ((fi_obtido * M_PI) / 180);
281.       if (switchpages)
282.         {
283.           visualpage = 1 - visualpage;
284.           setvisualpage (visualpage);
285.           setactivepage (1 - visualpage);
286.         }
287.     }
288. }

289. void
290. main (void)
291. {
292.   int i, j, k;
293.   int td = 0;
294.   int to = 0;
295.   int tf = 0;
296.   int tt = 0;
297.   int tm = 0;
298.   contador = 0;
299.   mutacao = 0;
300.   graf = 1;
301.   clrscr ();
302.   printf ("\n Populacao(maximo 100):"); scanf ("%d", &numpop);
303.   printf ("\n Numero de Iteracoes:"); scanf ("%d", &numint);
304.   printf ("Taxa de Mutacao:");
305.   scanf ("%d", &tm);
306.   printf ("Posicao X:");
307.   scanf ("%d", &td);
308.   printf ("Posicao Y:");
309.   scanf ("%d", &to);
310.   printf ("FI obtido:");
311.   scanf ("%d", &tf);
312.   printf ("Teta Obtido:");
```

```
313.   scanf ("%d", &tt);
314.   for (k = 0; k < tm; k++)
315.     mutacao++;
316.   for (k = 0; k < td; k++)
317.     x_obtido++;
318.   for (k = 0; k < to; k++)
319.     y_obtido++;
320.   for (k = 0; k < tf; k++)
321.     fi_obtido++;
322.   for (k = 0; k < tt; k++)
323.     teta_obtido++;
324.   if (tf < 0)
325.     for (k = 0; k > tf; k--)
326.       fi_obtido--;
327.   if (tt < 0)
328.     for (k = 0; k > tt; k--)
329.       teta_obtido--;
330.   inicializa ();
331.   ini_graf ();
332.   controlador ();
333. }
```

Bibliografia

[ADL 77] ADLEMAN, L.; MANDERS, K. Reductibility, randomness and untractability. In: ACM STOC, 9. 1977. **Proceedings...** [S.l.:s.n.], 1977. p. 151-163.

[AGU 96] AGUIAR, M. S. **Tratamento de problemas NP-completo:** trabalho individual. Porto Alegre:CPGCC da UFRGS, 1996. (TI-570)

[AGU 97a] AGUIAR, M. S.; TOSCANI, L. V. Avaliação qualitativa de algoritmos genéticos em problemas de otimização. In: SEMANA ACADÊMICA DO CPGCC, 2., 1997, Porto Alegre. **Anais...** Porto Alegre: CPGCC da UFRGS, 1997. p. 187-190.

[AGU 97b] AGUIAR, M. S.; TOSCANI, L. V. Algoritmos Genéticos. In: WORKSHOP SOBRE MéTODOS FORMAIS E QUALIDADE DE SOFTWARE, 1., 1997, Porto Alegre. **Anais...** Porto Alegre: CPGCC da UFRGS, 1997. p. 78-87.

[AGU 97c] AGUIAR, M. S.; TOSCANI, L. V. Avaliação Qualitativa de Algoritmos Genéticos em Problemas de Otimização. In: CONGRESSO NACIONAL DE MATEMáTICA APLICADA E COMPUTACIONAL, 20., 1997, Gramado. **Resumos...** Rio de Janeiro:SBMAC, 1997. p. 448-449.

[AHO 74] AHO, A. V.; HOPCROFT, J. E.; ULLMAN, J. D. **The design and analysis of computer algorithms.** Readings: Addison-Wesley, 1974. 470p.

[AND 95] ANDERSEN, H. C.; TSOI, A. C. **A constructive algorithm for the training of a multilayer perceptron based on the genetic algorithm**. 1995. Disponível por WWW em http://www.elec.uq.edu.au/∼andersen/SpeciesGANN.html.

[BOV 94] BOVET, D. P.; CRESCENZI, P. **Introduction to the theory of complexity.** [S.l.]: Prentice Hall, 1994. 282p.

[BUR 95] BURGESS, C. J. **A genetic algorithm for the optimization of a multiprocessor computer architecture.** 1995. Disponível por WWW em http://www.compsci.bristol.ac.uk/Tools/Reports/Abstracts/1995-burgess.html.

[COO 83] COOK, S. A. An overview of computational complexity. **Communications of the ACM**, New York, v. 16, n. 6, p. 401-407, 1983.

[CSI 90] CSIKSZENTMIHALYI, M. **Floq: The psycology of optimal experience.** New York: Harper & Row, 1990.

[DAV 91] DAVIS, L. **Handbook of genetic algorithms.** New York: Van Nostrand Reinhold, 1991.

[GAG 94] GAGE, P. **Multidisciplinary optimization methods for aircraft preliminary design**. 1994. Disponível por WWW em http://-aero.stanford.edu/Reports/MDO94.html.

[GAR 79] GAREY, M. R.; JOHNSON, D. S. **Computers and intractability. A guide to the theory of NP-Completeness.** San Francisco: W. H. Freeman, 1979.

[GOL 89] GOLDBERG, D. **Genetic algorithms in search, optimization & machine learning.** [S.l.]: Addison Wesley, 1989.

[GOL 93] GOLDBERG, D. Making genetic algorithms fly: A lesson from the Wright Brothers. **Advanced Technology for Developers,** [S.l.], n.2, p.1-8, 1993.

[GOL 94a] GOLDBERG, D. Genetic and evolutionary algorithms come of age. **Communications of the ACM**, New York, v.37, n.3, p. 113-119, 1994.

[GOL 94b] GOLDBERG, D. **The existencial pleasures of genetic algorithms.** Urbana-Champaign: University of Illinois, IlliGAL, 1994. 8p. (Report 94010).

[GRA 97] GRAY, G. J.; LI, Y. **Identification of nonlinear control laws using genetic programming.** 1997. Disponível por WWW em http://www.mech.gla.ac.uk/∼yunli/csc97012.htm.

[HAY 94] HAYWARD, T. J. **Adaptation of genetic-algorithm search for matched-field inversion of ocean bottom compressional wave speed profiles.** 1994. Disponível por WWW em http://sound.media/mit.edu/AUDITORY/asamtgs/asa94aus/2aA0/2aA04.html.

[HOL 75] HOLLAND, J. H. **Adaptation in natural and artificial systems.** Ann Arbor: The University of Michigan Press, 1975.

[HOR 78] HOROWITZ, E.; SAHNI, S. **Fundamentals of computer algorithms.** [S.l.]: Comp. Sci. Press, 1978.

[JAC 95] JACOB, B. L. **Composing with genetic algorithms.** 1995. Disponível por WWW em http://www.eecs.umich.edu/∼blj/papers/-icmc.95.html.

[JAN 93] JANIKOW, C. Z. **A knowledge-intensive genetic algorithm for supervised learning.** 1993. Disponível por WWW em http://mlis.-www.wkap.nl/contens/abstracts/absv13p189.htm.

[JON 93] JONG, K. A. Genetic algorithms are not function optimizers. **Foundations of Genetic Algorithms.** [S.l.:s.n.], 1993. p. 5-17.

[KIN 95] KING, J. P. et al. **Economic Optimization of River Management using Genetic Algorithms.** 1995. Disponível por WWW em http://wrri.nmsu.edu/publish/techrpt/abs295.html.

[KNU 83] KNUTH, D. E. Algorithm and program, information and data. **Communication of the ACM**, New York, v. 26, p.56, 1983.

[KOS 91] KOSKO, B. **Neural networks and fuzzy systems: a dynamical approach to machine intelligence.** Englewood Cliffs: Prentice Hall, 1991.

[LAD 75] LADNER, R. On the structure of polynomial time reducibility. **Journal of the ACM**, New York, v.22, p. 159-171, 1975.

[LAG 96] LAGUNA, M.; MOSCATO, P. Algoritmos geneticos. In: DIAZ, B. A. (Ed.). **Optimizacion heuristica y redes neuronales.** Madri: Paraninfo, 1996.

[LEA 97] LEAL, L. A. S.; CLAUDIO, D. M.; TOSCANI, L. V. **Teoria de Algoritmos Frente à Intratabilidade de Problemas.** Porto Alegre:CPGCC da UFRGS, 1997. (EQ-16).

[MIT 94a] MITCHELL, M. **Evolving cellular automata to perform computations: mechanisms and impediments.** 1994. Disponível por WWW em http://www.satafe.edu/projects/evca/evabstracts.html.

[MIT 94b] MITCHELL, M.; DAS, R.; CRUTCHFIELD, J. P. **A genetic algorithm discovers particle - based computation in cellular automata.** 1994. Disponível por WWW em http://www.satafe.edu/-projects/evca/evabstracts.html.

[MIT 96] MITCHELL, M. **An introduction to genetic algorithms.** [S.l.]: MIT Press, 1996.

[MUH 93] MÜHLENBEIN, H.; SCHLIERKAMP-VOOSEN, D. **Predictive models for the breeder genetic algorithm.** 1993. Disponível por WWW em http://borneo.gmd.de/∼dirk/publi/-gmd_as_ga-93_01.html.

[PAP 94] PAPADIMITRIOU, C. H. **Computational complexity.** [S.l.]: Addison-Wesley Publishing Company, 1994. 523p.

[RAB 76] RABIN, M. O. Probabilistics algorithms.**Algorithms and Complexity**. [S.l.]: Academic Press, 1976. p. 21-36.

[RAM 94] RAMOS, A. **Aplicação de algoritmos genéticos em problemas de otimização.**: trabalho individual. Porto Alegre: CPGCC da UFRGS, 1994. (TI-403).

[REI 89] REIF, J. H.; SEN, S. Randomization in parallel algorithms and its impact on computational geometry. In: GOOS, G.; HARTMANIS, J. (Eds.). **Optimal Algortihms.** Berlin: Springer-Verlag, 1989. p. 1-8. (Lecture Notes in Computer Science).

[ROS 97] ROSA, D. S. **Complexidade média algorítmica: uma metodologia para o seu cálculo.** Porto Alegre: CPGCC/UFRGS, 1997.

[SAR 93] SARGENT, T. J. **Bounded rationality in macroeconomics.** Oxford: Clarendon Press, 1993.

[SCH 95] SCHNECKE, V.; VORNBERGER, O. **Genetic design of VLSI-layouts.** 1995. Disponível por WWW em http://brahms.informatik.uni-osnabrueck.de/prakt_eng/pub/abstracts/galesia_95.html.

[SCH 96] SCHNECKE, V.; VORNBERGER, O. **A genetic algorithm for VLSI physical design automation.** 1996. Disponível por WWW em http://brahms.informatik.uni-osnabrueck.de/prakt/-pub/abstracts/ppsn_96.html.

[SER 96] SERRIDGE, B. **Building a genetic algorithm to control a walking robot.** 1996. Disponível por WWW em http://sls-www.lcs.-mit.edu/∼ben/brooks/brooks.html.

[SMI 92] SMITH, R. E; FORREST, S.; PERELSON, A. S. **Searching for diverse, cooperative populations with genetic algorithms.** 1992. Disponível por WWW em http://www.santafe.edu/sfi/publications/Abstracts/92-06-027abs.html.

[SOL 77] SOLOVAY, R.; STRASSEN, V. A fast Monte-Carlo test for primality. **SIAM Journal of Computing**, [S.l.], p. 84-85, 1977.

[SYE 97] SYED, O. **Applying genetic algorithms to recurrent neural networks for learning network parameters and architecture.** 1997. Disponível por WWW em http://www.lerc.nasa.gov/people/-OmarSyed/homepage/MSThesis/.

[SZW 84] SZWARCFITER, J. L. **Grafos e algoritmos computacionais.** Rio de Janeiro: Campus, 1984.

[TOS 86] TOSCANI, L. V.; SZWARCFITER, J. L. Algoritmos aproximativos: uma alternativa para problemas NP-Completo. In: PANEL/CLEI, 12., 1986, [S.l.] **Anales...** [S.l.:s.n.], 1986. p. 155-164.

[TOS 88] TOSCANI, L. V. **Métodos e desenvolvimento de algoritmos: especificação formal, análise comparativa e de complexidade.** Rio de Janeiro: PUC-RJ, 1988. Tese de Doutorado.

[VEL 84] VELOSO, P. A. S. Aspectos de uma teoria geral de problemas. **Cadernos de História e Filosofia da Ciência**, [S.l.], n. 7, p. 21-42, 1984.

[WEI 77] WEIDE, B. A survey of analysis techniques for discrete algorithms. **Computing Surveys**, [S.l.], v. 9, p. 291-313, 1977.