



Adriano Batista Prieto

Quociente de Rayleigh e Algoritmos Genéticos: estudo de caso para o cálculo de Autovalores de Matrizes Simétricas.

Limeira

2016



UNIVERSIDADE ESTADUAL DE CAMPINAS
Faculdade de Tecnologia

Adriano Batista Prieto

Quociente de Rayleigh e Algoritmos Genéticos: estudo de caso para o cálculo de Autovalores de Matrizes Simétricas.

Dissertação apresentada à Faculdade de Tecnologia da Universidade Estadual de Campinas como parte dos requisitos exigidos para a obtenção do título de Mestre em Tecnologia, na área de Tecnologia e Inovação.

Orientador: Prof. Dr. Vitor Rafael Coluci

Este exemplar corresponde à versão final da dissertação defendida pelo aluno Adriano Batista Prieto e orientada pelo Prof. Dr. Vitor Rafael Coluci.

Limeira

2016

INCLUA AQUI O PDF COM A FICHA CATALOGRÁFICA FORNECIDA.

A banca examinadora composta pelos membros abaixo aprovou esta dissertação.

Prof. Dr. Vitor Rafael Coluci (Presidente).
Unicamp, Faculdade de Tecnologia.

Prof. Dr. Varese Salvador Timoteo.
Unicamp, Faculdade de Tecnologia.

Dr. Ronaldo Giro.
IBM, Brazil Research Lab.

“A Ata da Defesa, assinada pelos membros da Comissão Examinadora, consta no processo de vida acadêmica do aluno”.

À minha mãe, Sandra, e ao meu filho, Heitor.

Resumo

Um programa (disponível no GitHub) foi desenvolvido em Linguagem C para estudar um método que transforma, por meio de Algoritmos Genéticos (GAs), a obtenção de alguns autovalores de Matrizes Simétricas em um problema de Otimização Combinatória. A análise das Funções de Avaliação (*fitness*) mostrou em quais condições o autovalor mínimo é encontrado, gerando a hipótese de que um artigo de 2004, publicado em periódico internacional e indexado, está parcialmente incorreto. Justificativas matemáticas, baseadas nas propriedades do Quociente de Rayleigh, e experimentos computacionais, executados com matrizes de Coope–Sabo, confirmaram a hipótese.

Palavras-chaves: Autovalores; Quociente de Rayleigh; Algoritmos Genéticos; Otimização Combinatória; Linguagem C.

Abstract

A program (available on GitHub) was developed in C language to study a method that transforms, by means of Genetic Algorithms (GAs), the calculation of some *eigenvalues* of Symmetric Matrices in a Combinatorial Optimization problem. The analysis of Evaluation Functions (*fitness*) showed in which conditions the minimum *eigenvalue* can be found, generating the hypothesis that a 2004 article, published in an international and indexed periodic, is partially incorrect. Mathematical justifications, based on properties of Rayleigh Quotient, and computational experiments, executed with Coope–Sabo matrices, confirmed the hypothesis.

Keywords: Eigenvalues; Rayleigh Quotient; Genetic Algorithms; Combinatorial Optimization; C Language.

Listas de ilustrações

Figura 1 – Fluxograma do algoritmo genético típico.	19
Figura 2 – Exemplo de instabilidade do <i>fitness</i> . Enquanto em (a) o <i>fitness</i> cresce de maneira contínua e depois se estabiliza, em (b) há alguns pontos onde o comportamento da nota sofre uma mudança razoavelmente brusca. Gráficos retirados de (NANDY <i>et al.</i> , 2004).	21
Figura 3 – Gráfico da função $y(x) = \sin(x)/x$ para $x = [-20, 20]$. Se desejamos encontrar os máximos (global ou locais), uma boa função de avaliação seria a própria $y(x)$.	22
Figura 4 – Gráfico da função $y(x) = -x^2 + 36$ para $x = [-1, 1]$.	24
Figura 5 – Roleta criada a partir dos dados da tabela 6. Note que o indivíduo $x = 7$ foi descartado porque sua avaliação foi um número menor do que zero.	25
Figura 6 – Exemplo de código em Linguagem C para o método da Roleta.	26
Figura 7 – Exemplo de função em Linguagem C que implementa a Seleção por Torneio.	28
Figura 8 – Definição de Ponto de <i>Corte</i> e o <i>Crossover</i> de ponto único. Com esse operador conseguimos gerar até dois filhos para cada par de pais.	29
Figura 9 – Exemplo do <i>crossover</i> entre os indivíduos 011 e 101 para o primeiro ponto de corte.	31
Figura 10 – Código para a Reprodução. Nesse exemplo o algoritmo gera apenas um descendente. Detalhes da função <i>CrossOver()</i> estão na figura 11.	32
Figura 11 – Detalhes da função <i>CrossOver()</i> .	32
Figura 12 – Representação gráfica de uma mutação. Nesse exemplo a mutação no último <i>bit</i> levou à solução ótima para o máximo da função $y(x) = -x^2 + 36$.	34
Figura 13 – Exemplo de código para o operador Mutação.	35
Figura 14 – Exemplo do <i>crossover</i> de (NANDY <i>et al.</i> , 2004). Indivíduos antes da reprodução.	41
Figura 15 – Exemplo do <i>crossover</i> de (NANDY <i>et al.</i> , 2004). Indivíduos depois da reprodução.	42
Figura 16 – Exemplo do <i>crossover</i> de (NANDY <i>et al.</i> , 2011). Indivíduos antes da reprodução.	43
Figura 17 – Exemplo do <i>crossover</i> de (NANDY <i>et al.</i> , 2011). Indivíduos depois da reprodução.	44
Figura 18 – Fluxo algoritmo genético	48
Figura 19 – ONEMAX: representação cromossomial, <i>fitness</i> e objetivo.	49
Figura 20 – Uma solução para o Problema das 8 Rainhas.	50
Figura 21 – Arena de batalha do Robocode.	50

Figura 22 – Exemplo de <i>script</i> Windows para fazer execuções variando o número de genes.	52
Figura 23 – Diagrama do <i>crossover</i> implementado. Na esquerda: <i>crossover</i> acontece; na direita: <i>crossover</i> acontece.	68
Figura 24 – Comportamento do <i>fitness</i> $f_i = e^{-\beta \nabla\rho_i ^2}$ para $N = 10$. Na primeira geração o melhor <i>fitness</i> é pequeno, aproximadamente 0,1, cresce rapidamente e a partir da décima geração está próximo de 1.	75
Figura 25 – Comportamento de ρ (Quociente de Rayleigh) para uma matriz de Coope–Sabo de ordem 10.	75
Figura 26 – Execuções $N = 10$	80
Figura 27 – Execuções $N = 20$	81
Figura 28 – Execuções $N = 30$	82
Figura 29 – Execuções $N = 40$	83
Figura 30 – Comportamento do <i>fitness</i> para as execuções zero do Hamiltoniano de ordem 10, semente 1445738835. A primeira usa o <i>fitness</i> $f_i = e^{-\beta(\rho_i - E_L)^2}$, que chega ao autovalor mínimo, enquanto a segunda utiliza o $f_i = e^{-\beta\ \nabla\rho_i\ ^2}$	87
Figura 31 – Comportamento do ρ para as execuções zero do Hamiltoniano de ordem 10, semente 1445738835. A primeira usa o <i>fitness</i> $f_i = e^{-\beta(\rho_i - E_L)^2}$, que chega ao autovalor mínimo, enquanto a segunda utiliza o $f_i = e^{-\beta\ \nabla\rho_i\ ^2}$	88
Figura 32 – Cenário 1. Execução para a semente 1445738835. E_L um pouco acima de E_0 no <i>fitness</i> $f_i = e^{-\beta(\rho_i - E_L)^2}$	89
Figura 33 – Execução para a semente 1445738835. E_L muito acima de E_0 no <i>fitness</i> $f_i = e^{-\beta(\rho_i - E_L)^2}$	89
Figura 34 – Execução para a semente 1445738835. E_L muito abaixo de E_0 no <i>fitness</i> $f_i = e^{-\beta(\rho_i - E_L)^2}$. Até geração 500.	90
Figura 35 – Execução para a semente 1445738835. E_L muito abaixo de E_0 no <i>fitness</i> $f_i = e^{-\beta(\rho_i - E_L)^2}$. Geração entre 30.000 e 40.000.	90
Figura 36 – Execuções para $N = 10$ com o <i>fitness</i> $f_i = e^{-\beta(\rho_i - E_L)^2}$	92
Figura 37 – Execuções para $N = 20$ com o <i>fitness</i> $f_i = e^{-\beta(\rho_i - E_L)^2}$	93
Figura 38 – Execuções para $N = 30$ com o <i>fitness</i> $f_i = e^{-\beta(\rho_i - E_L)^2}$	94
Figura 39 – Execuções para $N = 40$ com o <i>fitness</i> $f_i = e^{-\beta(\rho_i - E_L)^2}$	95
Figura 40 – Cenário 1. Várias execuções com o E_L um pouco acima de E_0 no <i>fitness</i> $f_i = e^{-\beta(\rho_i - E_L)^2}$. Semente 1445738835, $N = 10$	96
Figura 41 – Execuções com o E_L um pouco abaixo de E_0 no <i>fitness</i> $f_i = e^{-\beta(\rho_i - E_L)^2}$. Semente 1445738835, $N = 10$	97
Figura 42 – Execuções com o E_L muito acima de E_0 no <i>fitness</i> $f_i = e^{-\beta(\rho_i - E_L)^2}$. Semente 1445738835, $N = 10$	98

Figura 43 – Execuções com o E_L muito abaixo de E_0 no fitness $f_i = e^{-\beta(\rho_i - E_L)^2}$. Semente 1445738835, N = 10.	99
Figura 44 – Simetria de $f_i = e^{-\beta(\rho - E_L)^2}$. Nesse caso $E_L = 0$.	101
Figura 45 – Deslocamento do máximo de $f_i = e^{-\beta(\rho - E_L)^2}$	102
Figura 46 – Efeito da mudança de β em $f_i = e^{-\beta(\rho - E_L)^2}$	102
Figura 47 – E central é linear	103
Figura 48 – Exemplo de um grafo. Fonte: Wikipedia.	114
Figura 49 – ONEMAX, um problema clássico nos Algoritmos Genéticos.	116
Figura 50 – Execução do ONEMAX paralelo. Apenas as chamadas dos kernels acontecem na CPU, enquanto o restante (execução + dados) está na GPU.	117
Figura 51 – ONEMAX paralelo. Ganho de velocidade em função do número de indivíduos da população.	119
Figura 52 – ONEMAX paralelo. Ganho de velocidade em função do tamanho do cromossomo.	119

Lista de tabelas

Tabela 1 – Sistemas Naturais x Sistemas Artificiais	18
Tabela 2 – Exemplo de representação cromossomial	20
Tabela 3 – Valores de x gerados aleatoriamente para a função $y(x) = \sin(x)/x$. A própria $y(x)$ pode ser usada como função de avaliação.	22
Tabela 4 – Representação cromossomial para os pontos $x = 0$ até $x = 7$ dentro do problema de máximo da função $y(x) = -x^2 + 36$	23
Tabela 5 – População inicial para o problema de máximo da função $y(x) = -x^2 + 36$. O valor de máximo ocorre em $x = 0$, ou $x = 000$ na representação binária.	24
Tabela 6 – Todas as notas dos indivíduos para o exemplo da seção 3.6.1. Lembre-se que para obter o máximo da função $y(x) = -x^2 + 36$ podemos utilizar, com algumas restrições, a própria $y(x)$ como função de avaliação $f_c(x)$ (seção 3.5.1).	25
Tabela 7 – Valores obtidos aleatoriamente para <i>vlrRoleta</i> e os respectivos indivíduos selecionados. Note que o cromosso com <i>fitness</i> zero foi eliminado e o <i>fitness</i> médio aumentou.	27
Tabela 8 – Geração antes e depois do <i>Crossover</i> . O melhor indivíduo dos descendentes possui o melhor <i>fitness</i> entre todas as gerações anteriores. Além disso, o <i>fitness</i> médio ($< f >$) também aumentou.	32
Tabela 9 – Representação cromossomial para os indivíduos que passaram pela Seleção e pelo <i>Crossover</i>	33
Tabela 10 – Lista de autovalores para matrizes de Coope–Sabo de ordem 10, 20, 30 e 40.	79
Tabela 11 – Execuções para matrizes de Coope–Sabo.	84
Tabela 12 – Execuções novo <i>Fitness</i>	86
Tabela 13 – Variando E_L para a execução da semente 1445738835. Os tipos de teste são: cenário 1 : E_L um pouco acima de E_0 ; cenário 2 : E_L um pouco abaixo de E_0 ; cenário 3 : E_L muito acima de E_0 ; cenário 4 : E_L muito abaixo de E_0	91
Tabela 14 – Cinco execuções para cada tipo de teste de variação de E_L em torno de E_0 no fitness $f_i = e^{-\beta(\rho_i - E_L)^2}$	100
Tabela 15 – Valores de $(E_{central} - E_0)$	105

Sumário

1	Introdução	12
2	Quociente de Rayleigh	15
3	Algoritmos Genéticos	17
3.1	Aspectos históricos	17
3.2	Terminologia	18
3.3	Fluxograma de um algoritmo genético simples	18
3.4	Representação Cromossomial	18
3.5	Função de Avaliação	20
3.5.1	Exemplo: Máximos de $y(x) = \sin(x)/x$	21
3.6	Seleção	23
3.6.1	Exemplo: Variabilidade Genética	23
3.6.2	O método da Roleta	24
3.6.3	Exemplo: máximo da função $y(x) = -x^2 + 36$	26
3.6.4	Seleção por torneio	28
3.7	Reprodução (<i>Crossover</i>)	29
3.7.0.1	Exemplo: <i>crossover</i> para $y(x) = -x^2 + 36$	29
3.8	Mutação	33
4	Autovalores de Matrizes Simétricas com Algoritmos Genéticos	36
4.1	Representação Cromossomial	37
4.2	População	38
4.3	Funções de Avaliação (<i>fitness</i>)	38
4.4	Seleção	40
4.5	Reprodução	40
4.5.1	<i>Crossover</i> em (NANDY <i>et al.</i> , 2004)	40
4.5.2	<i>Crossover</i> em (NANDY <i>et al.</i> , 2011)	42
4.5.3	<i>Crossover</i> utilizado	44
4.6	Mutação	46
4.7	Fluxograma do algoritmo implementado	48
5	Metodologia	49
5.1	Algoritmos Genéticos	49
5.2	<i>Software</i>	51
5.2.1	Como utilizar o programa	51
5.2.2	Informações de saída	52
5.2.3	Lista dos arquivos fonte	53
5.2.4	Lista dos parâmetros de execução	54
5.2.5	Estruturas de Dados	57

5.2.5.1	parametrosPrograma	57
5.2.5.2	parametros_Metodo	58
5.2.5.3	parametros	58
5.2.5.4	individual	59
5.2.5.5	generation	60
5.2.6	População inicial	61
5.2.7	<i>Fitness</i>	62
5.2.8	Seleção	67
5.2.9	<i>Crossover</i>	68
5.2.10	Mutação	70
5.2.11	Fluxo principal	71
6	Resultados e discussão	74
6.1	Problemas com o mínimo global	74
6.2	Outro <i>fitness</i> para encontrar o mínimo global	85
6.3	Análise do <i>fitness</i> e equação empírica para β	101
7	Conclusões	106
Referências	107
Apêndices	110
APÊNDICE A Autovalores do século XVIII ao XXI	111
APÊNDICE B Resultados preliminares na GPU	116

1 Introdução

O problema dos autovalores e autovetores pode ser definido brevemente da seguinte maneira: dada uma matriz A n por n , o escalar λ é chamado de **autovalor** de A se existe um vetor \mathbf{u} não nulo tal que

$$A\mathbf{u} = \lambda\mathbf{u}. \quad (1.1)$$

O vetor \mathbf{u} é chamado de **autovetor** de A associado a λ . Reescrevendo a equação 1.1 chegamos a

$$\begin{aligned} A\mathbf{u} - \lambda\mathbf{u} &= 0 \\ (A - \lambda I)\mathbf{u} &= 0, \end{aligned} \quad (1.2)$$

onde I é a matriz identidade. Pela teoria das equações lineares, a equação acima só tem soluções se

$$\det(A - \lambda I) = 0. \quad (1.3)$$

A equação 1.3 é chamada de **Equação Característica** de A , e leva a um **Polinômio Característico** de ordem n . Portanto, A pode ter até n autovalores.

Mas, de onde vieram os autovalores? O que são? Por que são importantes? Muitos estudantes de ciências exatas e engenharia fazem essas três perguntas durante as disciplinas introdutórias de Álgebra Linear. Uma descrição mais detalhada da importância histórica dos autovalores, além dos métodos numéricos clássicos, pode ser encontrada no Apêndice A.

A primeira aparição dos autovalores aconteceu no século XVIII. A partir de então grandes nomes da matemática se debruçaram sobre o estudo da sua natureza, como D'Alembert, Lagrange, Laplace, Sturm e Cauchy, culminando no final do século XIX na Teoria Espectral das Matrizes. No início do século XX os autovalores foram fundamentais dentro da Mecânica Quântica, uma das grandes revoluções científicas da nossa era. Com o advento do computador eletrônico por volta de 1950, a criação de métodos numéricos tornou-se muito ativa ([HAWKINS, 1975](#)).

No século XXI a relevância continua. A busca pela palavra *eigenvalue* em periódicos como *Nature* e *Science* revela que os autovalores continuam muito utilizados. Google, Facebook e Twitter, por exemplo, estão fortemente relacionados aos autovalores. Dado o número de usuários desses sistemas, no mínimo da ordem de centenas de mi-

lhões (EDIGER *et al.*, 2010), passou-se a buscar métodos com alto potencial para uso de algoritmos paralelos e computação de alto desempenho.

Apresento nesta dissertação o estudo de um método com esse potencial. Em uma série de artigos, com (NANDY *et al.*, 2004) sendo o primeiro, é proposta uma maneira de encontrar o autovalor mínimo de uma matriz simétrica. Para isso fazem uso de Algoritmos Genéticos (GAs) (MITCHELL, 1998), que são intrinsecamente paralelos (LINDEN, 2008) e, dependendo do problema modelado e da arquitetura de computação paralela escolhida, altamente escaláveis. Essas características já foram exploradas de maneira sistemática em arquiteturas tradicionais (CANTU-PAZ, 2000), e mais recentemente bons desempenhos foram atingidos com GAs simples executados em placas de vídeo (GPUs) com arquitetura NVIDIA CUDA. Em (DEBATTISTI *et al.*, 2009) o GA paralelizado na GPU chegou a ser até trinta vezes mais rápido.

No entanto, encontrei uma falha em (NANDY *et al.*, 2004). O artigo afirma que o método obtém sempre o autovalor mínimo de matrizes simétricas. Com base em resultados experimentais próprios, em conhecidas propriedades do Quociente de Rayleigh e em um erro lógico no artigo, argumento que não há garantias de se encontrar o menor autovalor. É possível que os autores tenham omitido algum procedimento específico.

Dos cinco artigos que utilizam o método, o último (NANDY *et al.*, 2011) possui uma diferença significativa: a Função de Avaliação (*fitness*) do Algoritmo Genético foi alterada. Essa função tem papel fundamental em todo algoritmo genético, pois é a maneira utilizada pelos GAs para determinar a qualidade de um indivíduo como solução do problema (LINDEN, 2008). Isso me motivou a estudar (NANDY *et al.*, 2011) em detalhes e identifiquei que, justamente por causa do novo *fitness*, o erro lógico dos trabalhos anteriores não foi cometido.

Dado o contexto, formulei as seguintes hipóteses:

Hipótese 1: a impossibilidade de se obter o menor autovalor com (NANDY *et al.*, 2004) não reside em uma falha do método, mas apenas na má definição da Função de Avaliação.

Hipótese 2: se a Hipótese 1 é verdadeira, é possível encontrar o menor autovalor com a Função de Avaliação de (NANDY *et al.*, 2011).

A fim de verificá-las, implementei um GA contendo os principais elementos comuns entre (NANDY *et al.*, 2004) e (NANDY *et al.*, 2011). A ideia foi, mantendo uma única estrutura, comparar apenas o efeito da mudança do *fitness*.

Os resultados obtidos foram satisfatórios, e trouxeram as seguintes novidades:

1. As duas hipóteses foram confirmadas;

2. Há base para contradizer ([NANDY *et al.*, 2004](#));
3. Restrita a um caso específico, encontrei uma equação para um importante parâmetro do *fitness* de ([NANDY *et al.*, 2011](#));

2 Quociente de Rayleigh

Com referência à equação 1.3 da Introdução, seja \mathbf{A} , a partir de agora, uma matriz auto-adjunta ($A^\dagger = A$). Todos os seus autovalores λ_i são reais e, consequentemente, podem ser ordenados do menor para o maior:

$$\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n. \quad (2.1)$$

Para \mathbf{A} , que opera sobre os vetores \mathbf{u} do espaço euclidiano ε^n , define-se o **Quociente de Rayleigh** $\rho(\mathbf{u})$:

$$\rho(\mathbf{u}) \equiv \rho(\mathbf{u}; \mathbf{A}) \equiv \frac{\mathbf{u}^\dagger \mathbf{A} \mathbf{u}}{\mathbf{u}^\dagger \mathbf{u}}, \quad (2.2)$$

onde \mathbf{u}^\dagger é o complexo conjugado de \mathbf{u} , que por sua vez deve ser sempre diferente de zero ($\mathbf{u} \neq 0$). Todo vetor \mathbf{u} possui um $\rho(\mathbf{u})$.

O Quociente de Rayleigh está diretamente relacionado aos autovalores e autovetores. Suponha que \mathbf{w}_i seja um dos n autovetores de \mathbf{A} . A equação 2.2 fica

$$\rho(\mathbf{w}_i) = \frac{\mathbf{w}_i^\dagger \mathbf{A} \mathbf{w}_i}{\mathbf{w}_i^\dagger \mathbf{w}_i} = \frac{\mathbf{w}_i^\dagger \lambda_i \mathbf{w}_i}{\mathbf{w}_i^\dagger \mathbf{w}_i} = \frac{\lambda_i \mathbf{w}_i^\dagger \mathbf{w}_i}{\mathbf{w}_i^\dagger \mathbf{w}_i} = \lambda_i, \quad (2.3)$$

ou seja, o quociente de Rayleigh de um autovetor é o autovalor associado:

$$\rho(\mathbf{w}_i) = \lambda_i. \quad (2.4)$$

Portanto, se temos um autovetor de uma matriz auto-adjunta, não é necessário resolver o sistema linear 1.3 para obter o autovalor, basta efetuar as multiplicações de 2.2.

O vetor gradiente de ρ é (WILKINSON, 1965)

$$\nabla \rho(\mathbf{u}) = \frac{2[\mathbf{A} - \rho(\mathbf{u})]\mathbf{u}}{\mathbf{u}^\dagger \mathbf{u}}. \quad (2.5)$$

Ele é nulo se, e somente se, \mathbf{u} é um autovetor \mathbf{w}_i de \mathbf{A} :

$$\nabla \rho(\mathbf{w}_i) = \frac{2[\mathbf{A} - \rho(\mathbf{w}_i)]\mathbf{w}_i}{\mathbf{w}_i^\dagger \mathbf{w}_i} = \frac{2[\mathbf{A} - \lambda_i]\mathbf{w}_i}{\mathbf{w}_i^\dagger \mathbf{w}_i} = \frac{2[\mathbf{A}\mathbf{w}_i - \lambda_i\mathbf{w}_i]}{\mathbf{w}_i^\dagger \mathbf{w}_i} = \frac{2[0]}{\mathbf{w}_i^\dagger \mathbf{w}_i} = \mathbf{0}, \quad (2.6)$$

$$\nabla \rho(\mathbf{w}_i) = \mathbf{0}. \quad (2.7)$$

Além disso, para todos os vetores \mathbf{u} n –dimensionais diferentes de zero, $\rho(\mathbf{u})$ é limitado no intervalo $[\lambda_1, \lambda_n]$ entre o menor e o maior autovalor da matriz \mathbf{A} (PARLETT, 1998). Em seguida agrupo as três propriedades de $\rho(\mathbf{u})$ citadas acima.

Algumas propriedades de $\rho(\mathbf{u})$

1. Se \mathbf{w}_i é um autovetor, $\rho(\mathbf{w}_i) = \lambda_i$.
2. **Fronteira:** $\rho(\mathbf{u})$ é limitado no intervalo $[\lambda_1, \lambda_n]$.
3. **Estacionaridade:** o gradiente de $\rho(\mathbf{u})$ é nulo se \mathbf{u} é um autovetor.

A segunda e terceira propriedades podem ser utilizadas para tratar o problema dos autovalores como um problema de otimização matemática. A propriedade de fronteira, por exemplo, permite transformar o problema de encontrar o menor (ou maior) autovalor de uma matriz auto–adjunta em um problema de minimização (ou maximização) de $\rho(\mathbf{u})$. A estacionaridade pode, inclusive, auxiliar na definição da função objetivo. Minimizar $\nabla\rho(\mathbf{u})$ também leva a um autovetor, entretanto, note que, quando $\nabla\rho(\mathbf{u}) = \mathbf{0}$, \mathbf{u} pode ser qualquer um dos n autovetores $(\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_n)$ de \mathbf{A} . Ou seja, não há garantia que o autovalor obtido é o menor ou maior.

A primeira propriedade permite que a obtenção dos autovalores seja transformada em um método de busca de autovetores. O espaço de busca é o conjunto de todos os vetores \mathbf{u}_i , e o espaço de soluções é composto pelos n autovetores $(\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_n)$.

Nesta dissertação estudei um método que cria um Algoritmo Genético para fazer essa busca. Ele faz uso das três propriedades.

3 Algoritmos Genéticos

3.1 Aspectos históricos

Dá-se o nome *Algoritmos Genéticos* a uma técnica de busca baseada numa metáfora da Evolução. Na Natureza muitos animais competem entre si por recursos limitados, e os vencedores de uma dada população têm mais chance de procriação. Seu êxito vem das características que os tornam melhor adaptados para os desafios apresentados pelo ambiente onde vivem. Seus filhos levarão essas características adiante.

Esse mecanismo de seleção dos melhores foi descoberto por Darwin, que o chamou de Seleção Natural. A partir dele foi possível explicar como a natureza gera organismos complexos e capazes de solucionar problemas difíceis. Darwin não sabia como as características eram transmitidas e como novas surgiam.

Fazendo experimentos com plantas, Mendel determinou como se dá a transmissão. Há uma unidade básica de informação associada às características dos organismos, que ele chamou de Gene. Definiu como *Fenótipo* as características externas de um organismo, e *Genótipo* o conjunto de genes associados a um dado fenótipo. Na reprodução sexuada os filhos possuem uma mistura entre os genes do pai e da mãe. Por isso, não são idênticos a nenhum dos seus progenitores, mas são semelhantes a eles.

Sabe-se hoje que toda a informação de um organismo está codificada no DNA, que contém os genes. A combinação entre os genes dos pais acontece durante a reprodução, num evento chamado de Cruzamento Cromossômico, ou *Crossing—Over*, onde há troca de informação entre os cromossomos. Durante a cópia do DNA ocorrem, com baixíssima probabilidade, erros eventuais, fazendo com que o filho tenha um novo atributo.

Se a nova característica for muito boa, após várias gerações ela estará espalhada por toda população. O indivíduo será beneficiado na competição e provavelmente se reproduzirá mais, assim como seus filhos. A cada nova geração, mais indivíduos possuirão o novo fenótipo, perpetuando a transmissão dos genes associados.

Apesar de não ter sido o primeiro a utilizar ideias da Evolução em Ciência da Computação, John Holland é considerado o pai dos Algoritmos Genéticos. Ele estudou formalmente, do ponto de vista matemático, a adaptação na natureza e propôs uma heurística baseada nesse estudo. Seu objetivo era simular a Evolução em computadores. Em 1975 publicou o livro *Adaptation in Natural and Artificial Systems*. A partir de então muitos passaram a usar GAs na solução de problemas de diversas áreas.

3.2 Terminologia

Os GAs usam terminologia baseada na Seleção Natural e Genética, conforme tabela abaixo:

Tabela 1 – Sistemas Naturais x Sistemas Artificiais

Sistema Natural	Sistema Artificial
gene	caractere
alelo	valor do caractere
cromossomo	cadeia de caracteres (indivíduo)
locus	posição do gene na cadeia de caracteres
ambiente	problema a ser solucionado

3.3 Fluxograma de um algoritmo genético simples

O algoritmo genético mais básico é composto por cinco processos (figura 1)

- **Gerar população inicial:** População inicial gerada aleatoriamente.
- **Avaliação:** Cada indivíduo recebe uma nota. Maiores notas indicam indivíduos mais aptos (soluções mais próximas do objetivo). Esta medida é conhecida como *Fitness*.
- **Teste do critério de parada:** Com todos os indivíduos avaliados, é possível verificar se algum deles representa uma boa solução e se o algoritmo pode ser finalizado. Um número máximo de gerações também pode ser utilizado.
- **Seleção:** A Seleção escolhe os sobreviventes da população atual que comporão a próxima população. Um bom processo de seleção atribui maior chance aos indivíduos com melhores *fitness*.
- **Reprodução (*crossover*):** Com probabilidade alta ($> 70\%$), indivíduos são selecionados aleatoriamente e geram filhos através da combinação de seus genes.
- **Mutação:** Com baixíssima probabilidade ($\approx 1\%$), genes são escolhidos para sofrer alterações em seus valores.

3.4 Representação Cromossomial

O cromossomo é uma cadeia de caracteres (genes) de comprimento L , que representa um indivíduo candidato à solução do problema proposto.

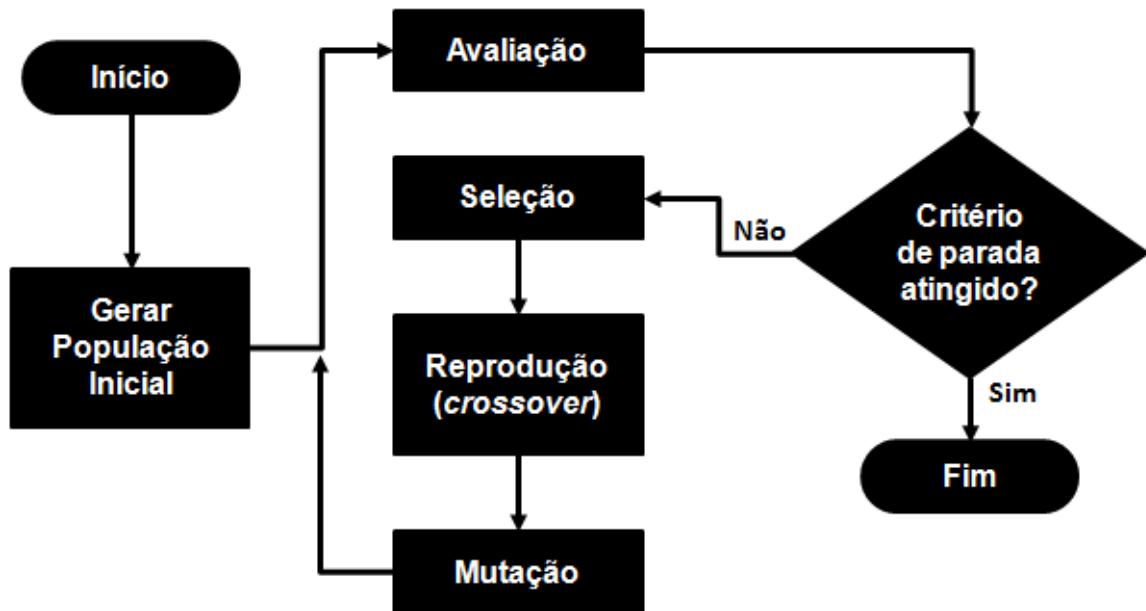


Figura 1 – Fluxograma do algoritmo genético típico.

Esta representação comprehende parte de grande importância num GA, pois é a partir dessa estrutura que os indivíduos serão avaliados e também sofrerão a atuação dos operadores de seleção e variação (*crossover* e mutação). Uma boa codificação do cromossomo influi diretamente no sucesso da aplicação de um GA.

Uma boa definição segue os princípios abaixo (LINDEN, 2008):

- Deve ser a mais simples possível.
- Soluções proibidas não devem ser representadas.
- Condições de qualquer tipo devem estar implícitas na representação.

A representação binária introduzida por Holland é a mais comum entre os GAs. Um dos motivos é facilitar a utilização dos operadores genéticos e ter fácil manipulação.

Caso se faça necessário, um cromossomo pode conter mais de uma variável em sua cadeia, sendo que estas serão concatenadas para representar o indivíduo. Na tabela 2 é exibido um exemplo com três possíveis indivíduos com cromossomo multivariável (variáveis x_1 e x_2) de comprimento $L = 5$ e alelos que variam entre 1 e 0. Porém, é possível implementar soluções com k diferentes alelos para cada locus do cromossomo, contudo, a dificuldade em manipular essa estrutura será maximizada.

Os exemplos desta seção utilizam representação binária. Mas pode-se utilizar outros tipos, como decimal, inteira, símbolos etc.

Tabela 2 – Exemplo de representação cromossomial

Indivíduo	x_1	x_2
1	10010	01101
2	00110	11100
3	11101	01001

3.5 Função de Avaliação

A função de avaliação tem um papel fundamental em um algoritmo genético. Junto com a representação cromossomial, ela é o elo entre o algoritmo e o problema que tentamos resolver no mundo real. Aliás, geralmente a principal diferença entre dois GAs reside na função de avaliação. Ela deve conter todo o conhecimento do problema, incluindo suas condições e restrições.

Mas, afinal, o que é a função de avaliação? Uma vez definido o problema e, em seguida, o objetivo do algoritmo, ela representa nesse contexto a qualidade de um indivíduo. Em outras palavras, através da função de avaliação devemos ser capazes de identificar se um cromossomo leva ou não a uma boa solução. Assim, ela deve refletir a meta que desejamos atingir.

Ao aplicarmos a função de avaliação¹ em um indivíduo obtemos uma nota associada àquele cromossomo. Essa nota é um número, um escalar, que pode ser discreto (inteiro) ou contínuo (real):

$$f_i = f_c(\text{cromossomo}_i) \quad (3.1)$$

Então, já podemos apresentar a primeira característica de uma função de avaliação: quanto maior a nota, melhor o indivíduo. Pensando em termos da Seleção Natural de Darwin, maiores notas exprimem indivíduos mais adaptados ao ambiente (metas). Além disso, na equação 3.1, f_i é uma métrica que deve identificar o quão próximo o cromossomo i está de uma boa solução.

Por exemplo, suponha que o cromossomo c_1 tem nota $n_1 = 10$, enquanto $n_2 = 9,7$ é atribuída ao cromossomo c_2 . Imaginando hipoteticamente que uma boa solução está próxima de $n_{boa} = 11$, podemos concluir que ambos são bons, mas c_1 é melhor. Sintetizando, uma boa função de avaliação deve quantificar, dentre boas soluções, quais são as melhores.

Outro número importante é o *fitness* médio ($\langle f \rangle$), ou seja, a razão entre a

¹ Essa função também pode ser chamada de Função Custo, por isso o f_c na equação 3.1.

soma das notas de todos os indivíduos e o número de indivíduos na geração (N):

$$\langle f \rangle = \frac{\sum_{i=1}^N f_i}{N} \quad (3.2)$$

Quando não sabemos qual será a maior nota possível para o nosso problema, temos a opção de usar a estabilidade de $\langle f \rangle$ como critério de parada. Se a média das notas não muda muito com o passar do tempo, podemos concluir que o material genético disponível indivíduo a indivíduo é muito semelhante, e essa ausência de variabilidade faz com que uma geração futura se pareça com a passada. Em linguagem mais técnica, estamos confinados a uma região específica no espaço de soluções.

Com relação ao comportamento da função de avaliação, é desejável que seja suave e regular. Se um indivíduo é levemente superior a outro, sua nota deve ser apenas um pouco maior. Infelizmente, na maioria das vezes isso não acontece, e o impacto pode surgir em forma de instabilidade do *fitness* médio. Na figura 2 encontramos dois exemplos.

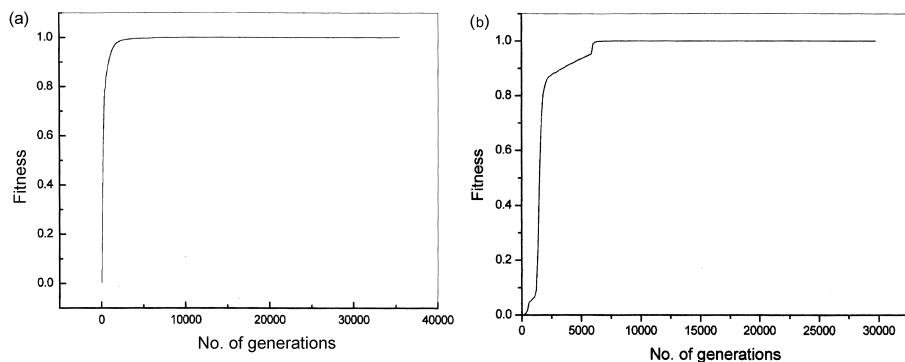


Figura 2 – Exemplo de instabilidade do *fitness*. Enquanto em (a) o *fitness* cresce de maneira contínua e depois se estabiliza, em (b) há alguns pontos onde o comportamento da nota sofre uma mudança razoavelmente brusca. Gráficos retirados de (NANDY *et al.*, 2004).

Na próxima seção discutiremos um exemplo de definição de uma função de avaliação.

3.5.1 Exemplo: Máximos de $y(x) = \sin(x)/x$

A função

$$y(x) = \frac{\sin(x)}{x} \quad (3.3)$$

aparece em várias áreas da matemática. Observando seu gráfico na figura 3, vemos que ela tem um máximo global em $x = 0$ e vários máximos locais. Imaginando que quiséssemos

obter os valores de x onde esses máximos aparecem, como definir uma boa função de avaliação?

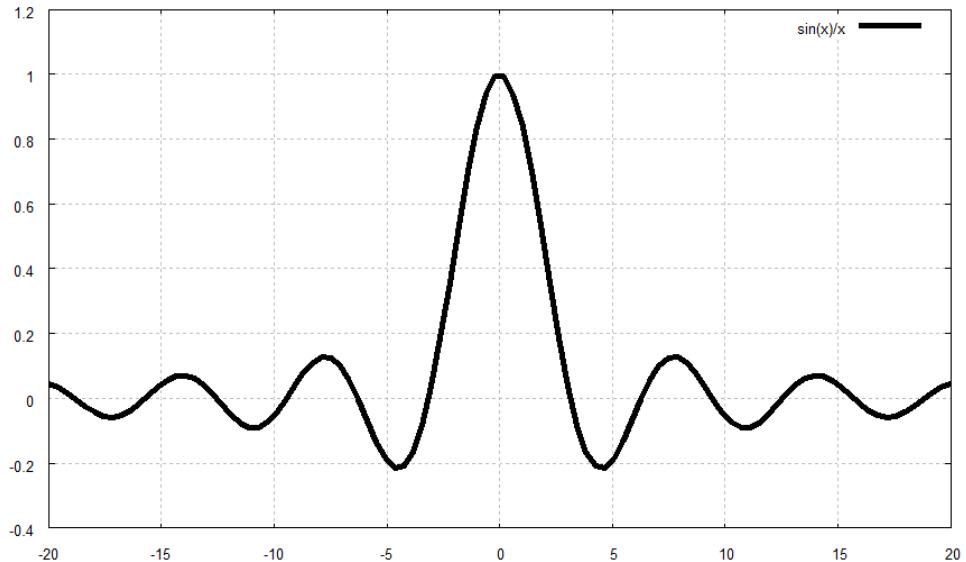


Figura 3 – Gráfico da função $y(x) = \sin(x)/x$ para $x = [-20, 20]$. Se desejamos encontrar os máximos (global ou locais), uma boa função de avaliação seria a própria $y(x)$.

Nesse caso podemos utilizar a própria $y(x)$ (equação 3.3), pois ela possui boas características. Pelo gráfico 3 concluímos que ela é suave e, se compararmos dois valores bem próximos de x , o resultado de $y(x)$ também é. Por exemplo, $f(0,5) = 0,9589$ e $f(0,51) = 0,9572$, evidenciando que o ponto $x = 0,5$ é um indivíduo levemente melhor.

Tabela 3 – Valores de x gerados aleatoriamente para a função $y(x) = \sin(x)/x$. A própria $y(x)$ pode ser usada como função de avaliação.

Indivíduo	x	y(x)
01	2	0,455
02	3	0,047
03	-9	0,046
04	-8	0,124
05	19	0,008
Soma:		0,680

Se escolhermos vários valores para x aleatoriamente, basta aplicar a $y(x)$ e selecionar os maiores². Na tabela 3 listamos cinco pontos como exemplo, assim como a soma dos resultados de $y(x)$. Como veremos na próxima seção, os pontos 01 e 04 têm,

² Em geral deve-se impor algumas restrições na função de avaliação. Veremos na seção 3.6 que valores negativos são proibidos.

respectivamente, $0,455/0,680 = 66,9\%$ e $0,124/0,680 = 18,2\%$ de chance de serem selecionados no processo de Seleção.

3.6 Seleção

A parte do algoritmo genético que chamamos de *Seleção*³ tem como objetivo simular o processo de Seleção Natural da Evolução. Basicamente, os mais aptos, leia-se “com maior *fitness*”, devem gerar mais descendentes.

Porém, exatamente como na natureza, os indivíduos avaliados com notas menores não devem ser totalmente descartados, e há bons motivos para isso. Em primeiro lugar, esses cromossomos, apesar de mal avaliados, podem conter informação genética importante, senão fundamental, para uma boa solução. Em segundo, a seleção apenas dos melhores, chamada de Elitismo, pode levar à uma convergência precoce e com soluções não tão boas.

3.6.1 Exemplo: Variabilidade Genética

Como exemplo da importância da variabilidade genética, imagine o problema de encontrar o valor máximo da função $y(x) = -x^2 + 36$ no intervalo (discreto) $x = [0, 7]$. Assim, uma representação cromossomial binária com 3 bits é suficiente, pois $0 = 000$ e $7 = 111$ (tabela 4).

Tabela 4 – Representação cromossomial para os pontos $x = 0$ até $x = 7$ dentro do problema de máximo da função $y(x) = -x^2 + 36$.

x (decimal)	x (representação binária)
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Vemos na figura 4 que a resposta correta é $x = 0$, cujo valor é $f(0) = 36$. Suponha agora que os dados da tabela 5 representem a população inicial do algoritmo. Se escolhêssemos apenas os indivíduos com as melhores notas, ou seja, $x = 2$ e $x = 3$, descartaríamos o indivíduo $x = 5$ e perderíamos uma importante característica genética: o zero no bit central.

³ Alguns autores chamam esse módulo de Seleção de Pais, enquanto outros o definem exatamente como na biologia, Seleção Natural. Para evitar mal entendidos, optamos por chamá-lo apenas de Seleção.

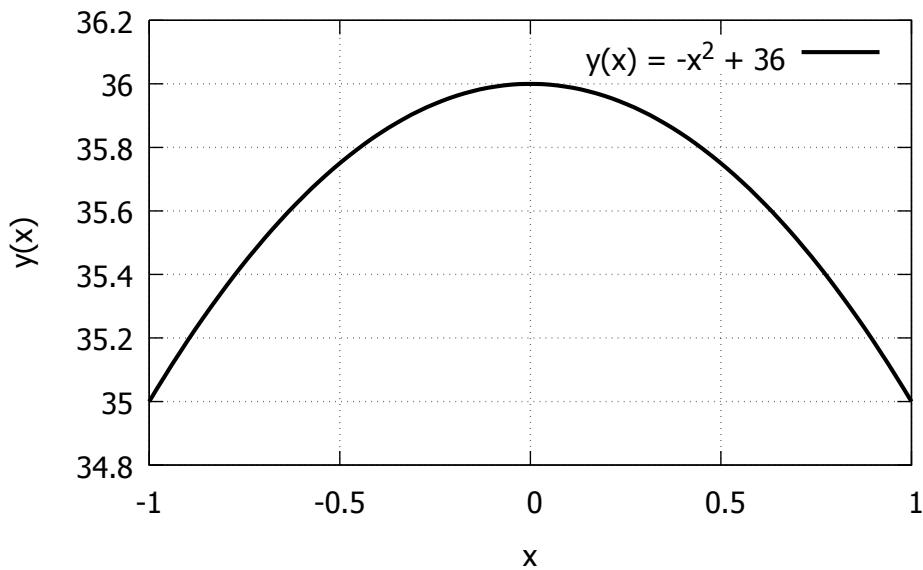


Figura 4 – Gráfico da função $y(x) = -x^2 + 36$ para $x = [-1, 1]$.

Tabela 5 – População inicial para o problema de máximo da função $y(x) = -x^2 + 36$. O valor de máximo ocorre em $x = 0$, ou $x = 000$ na representação binária.

x (decimal)	x (binário)
2	010
7	111
5	101
3	011

O que aconteceria depois? O mais próximo do valor máximo que o algoritmo conseguiria chegar seria $y(2) = 32$, independentemente do *crossover* entre os indivíduos restantes. Portanto, esse resultado final seria considerado uma **Convergência Genética** prematura. Em outras palavras, se apenas os melhores indivíduos se reproduzirem, as novas gerações chegarão rapidamente a um estado em que os cromossomos são muito semelhantes entre si, minando a diversidade genética e impedindo que a evolução prossiga satisfatoriamente.

Vários métodos foram criados para executar a Seleção de maneira coerente, ou seja, privilegiando os indivíduos com alta função de avaliação, mas não desprezando totalmente os de menor nota. Dois deles serão apresentados: o método da Roleta e a Seleção por Torneio.

3.6.2 O método da Roleta

A ideia do método é simular uma roleta parecida com as utilizadas em cassinos. Entretanto, há duas diferenças. Enquanto na roleta tradicional temos o mesmo

tamanho de fatia para cada número, na roleta para os algoritmos genéticos a fatia que cada indivíduo ganha é proporcional a sua avaliação.

Uma vez calculados os ângulos associados a cada cromossomo, “giramos a roleta” e selecionamos o indivíduo. Nesse ponto reside a segunda diferença. Nos cassinos, o número é selecionado por uma esfera que também está em movimento. Já nos algoritmos genéticos, o cromossomo é selecionado por um marcador fixo em uma roleta.

Na tabela 6 temos os valores dessas fatias (em porcentagem e graus) para o exemplo da seção 3.6.1. Note que o indivíduo 2 foi descartado, e esse é um ponto muito importante no método da roleta. Indivíduos com avaliações negativas não podem ocupar a roleta, e o motivo é simples: notas negativas levam a pedaços negativos na roleta, como, por exemplo, - 15% ou -125°.

Tabela 6 – Todas as notas dos indivíduos para o exemplo da seção 3.6.1. Lembre-se que para obter o máximo da função $y(x) = -x^2 + 36$ podemos utilizar, com algumas restrições, a própria $y(x)$ como função de avaliação $f_c(x)$ (seção 3.5.1).

Indivíduo	x	y(x)	$f_c(\text{Indivíduo})$	Ocupação (%)	Ângulo (°)
01	2	32	32	46	165
02	7	-13	0	0	0
03	5	11	11	16	57
04	3	27	27	39	139
Totais:			70	100	360

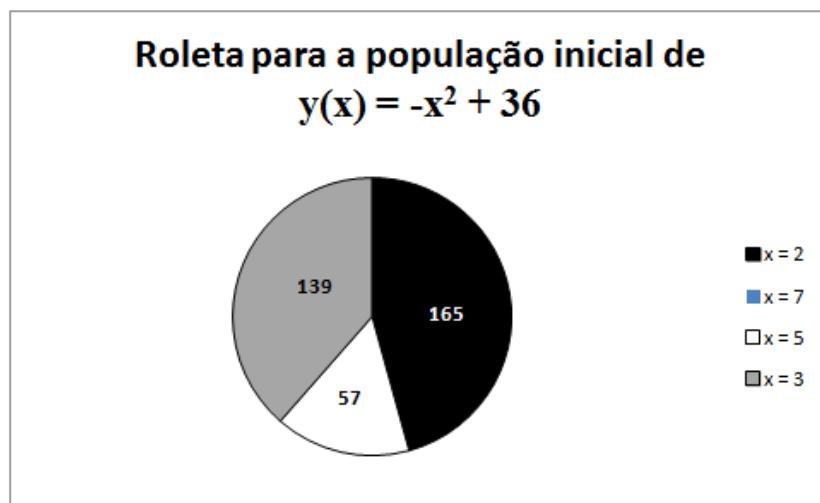


Figura 5 – Roleta criada a partir dos dados da tabela 6. Note que o indivíduo $x = 7$ foi descartado porque sua avaliação foi um número menor do que zero.

Um exemplo em Linguagem C de como implementar tal roleta encontra-se na figura 6. Na linha 226 um *for* é definido de modo que as suas instruções sejam executadas

numIndividuos vezes. Dessa maneira, a população de indivíduos mantém-se fixa, e podemos trabalhar com vetores constantes. A próxima linha de código armazena em *vlrRoleta* um número aleatório entre 0 e a soma das notas de todos os indivíduos (*sumFitness*). Essa soma também é exibida na tabela 6. A variável *iSelecionado* armazenará o índice do indivíduo selecionado.

No laço *do ... while* a roleta começa de fato a girar. A partir do primeiro indivíduo, somamos o *fitness* de cada um em *sumFitness*. Quando essa variável atingir um valor maior do que *vlrRoleta*, o indivíduo com índice *iSelecionado* na geração atual será transferido para a posição *iIndividuo* da próxima geração (linha 238 da figura 6).

Apesar do caráter randômico do método, ele prevê, estatisticamente, que a quantidade de vezes que um cromossomo aparecerá na próxima geração é proporcional à sua nota. Portanto, não despreza completamente os indivíduos com menor *fitness*, ao mesmo tempo que privilegia os mais aptos (LINDEN, 2008).

```

225 // Seleção via método da Roleta
226 for (iIndividuo = 0; iIndividuo < numIndividuos; iIndividuo++) {
227
228     vlrRoleta = Randomico(0, sumFitness);
229     iSelecionado = -1;
230     sumRoleta = 0;
231
232     do {
233         iSelecionado++;
234         sumRoleta = sumRoleta + geracao[0].individuo[iSelecionado].fitness;
235     } while (sumRoleta <= vlrRoleta);
236
237
238     geracao[1].individuo[iIndividuo] = geracao[0].individuo[iSelecionado];
239 }

```

Figura 6 – Exemplo de código em Linguagem C para o método da Roleta.

3.6.3 Exemplo: máximo da função $y(x) = -x^2 + 36$

Usaremos os dados da tabela 6 para montar a roleta da figura 5 e mostrar, passo a passo, o funcionamento do método.

Como temos quatro indivíduos na população,

$$numIndividuos = 4.$$

Devemos obter um número aleatório entre 0 e 70 (soma dos fitness). Imagine que esse primeiro número tenha sido 65:

$$vlrRoleta = 65.$$

Entramos no *for* e *iSelecionado* é inicializada com -1:

$$iSelecionado = -1.$$

Já dentro do laço **do ... while** o primeiro indivíduo na geração é o $x = 2$, com *fitness* igual a 32. Então *sumRoleta* passa a ser também 32, mas continua menor do que *vlrRoleta*:

$$sumRoleta = 32 < vlrRoleta = 65.$$

Ainda não é possível sair do laço **do ... while**, e os passos são repetidos para o segundo indivíduo, $x = 7$. Entretanto, esse cromossomo obteve nota nula na função de avaliação, fazendo com que o valor de *sumRoleta* não seja alterado pela soma da linha 234 (figura 6).

O terceiro cromossomo possui nota 11, fazendo com que ainda permaneçamos dentro do laço:

$$sumRoleta = 43 < vlrRoleta = 65.$$

Chegamos ao último indivíduo da população. Seu *fitness* é 27, permitindo a seleção:

$$sumRoleta = 70 > vlrRoleta = 65.$$

Na tabela 7 encontra-se uma comparação entre a geração inicial e final após a seleção via método da roleta. Os valores de *vlrRoleta* foram obtidos utilizando a função ALEATÓRIOENTRE do Microsoft Excel.

Tabela 7 – Valores obtidos aleatoriamente para *vlrRoleta* e os respectivos indivíduos selecionados. Note que o cromosso com *fitness* zero foi eliminado e o *fitness* médio aumentou.

Geração inicial		Roleta	Geração final	
x	fitness (n)	vlrRoleta	x	fitness (n)
2	32	65	3	27
7	0	27	2	32
5	11	70	3	27
3	27	41	5	11
$\langle f \rangle = 17,5$		-	$\langle f \rangle = 24,25$	

Dois fatos interessantes aconteceram. O indivíduo com *fitness* zero foi eliminado conforme esperado. Ele não ocupou espaço na roleta e, obviamente, não poderia

participar do processo de seleção. Além disso, o *fitness* médio ($< f >$) da nova população é superior ao da anterior, indicando que a Seleção cumpriu o seu papel em manter os mais aptos.

3.6.4 Seleção por torneio

Nesse tipo de seleção k indivíduos são selecionados aleatoriamente. O que possui maior *fitness* vence e é levado para a próxima geração. O parâmetro k é chamado de **Tamanho do Torneio**.

Assim como na Roleta, nada impede que um indivíduo seja escolhido mais de uma vez. Porém, uma desvantagem do Torneio é que o pior indivíduo será escolhido apenas se competir com cópias dele mesmo. Quanto maior o tamanho da população e do torneio, menor é a probabilidade de isso acontecer.

O Torneio e a Roleta levam a resultados completamente diferentes, mas há pontos positivos em usar o primeiro. Há indícios empíricos que o Torneio leva a resultados melhores. Ao contrário da Roleta, não há no Torneio favorecimento dos melhores. A única vantagem deles é que, se escolhidos, têm mais chance de vencer e prosseguir. Um superindivíduo, aquele com *fitness* muito maior que a média da população, não é privilegiado (LINDEN, 2008). O torneio também pode ser facilmente paralelizado.

```
//=====================================================================
void Selecao_Por_Torneio_serial( struct generation *PopulacaoAntes,
                                 struct generation *PopulacaoDepois,
                                 struct parametros *parametrosGA) {
//=====================================================================

    unsigned short int iIndividuo;
    char iTamanhoDoTorneio;
    unsigned short int iIndividuo_para_Torneio;
    struct individual *Individuo;
    double melhorFitness;

    for (iIndividuo = 0; iIndividuo < parametrosGA->numIndividuos; iIndividuo++) {

        melhorFitness = -1.0F;

        for (iTamanhoDoTorneio = 0; iTamanhoDoTorneio < parametrosGA->tamanho_torneio; iTamanhoDoTorneio++) {

            iIndividuo_para_Torneio = Randomico_int(0,parametrosGA->numIndividuos);

            Individuo = &PopulacaoAntes->individuo[iIndividuo_para_Torneio];

            if (Individuo->fitness > melhorFitness) {
                melhorFitness = Individuo->fitness;
                PopulacaoDepois->individuo[iIndividuo] = *Individuo;
            };
        };
    };
}
```

Figura 7 – Exemplo de função em Linguagem C que implementa a Seleção por Torneio.

3.7 Reprodução (*Crossover*)

A Reprodução consiste em combinarmos a informação genética de dois cromossomos da população e gerar um ou mais descendentes, até que o número de indivíduos da nova população seja atingido. Uma maneira simples é através do ***Crossover de ponto único*** (figura 8).

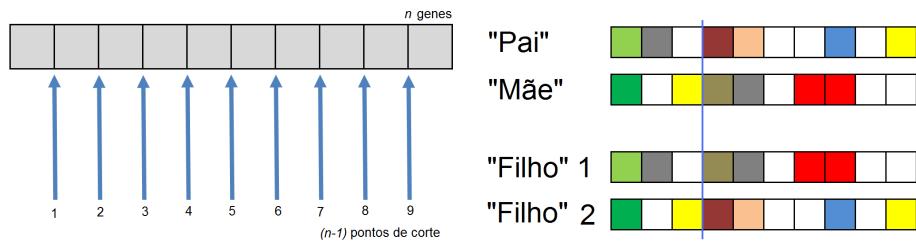


Figura 8 – Definição de Ponto de Corte e o *Crossover* de ponto único. Com esse operador conseguimos gerar até dois filhos para cada par de pais.

O primeiro passo é escolher dois cromossomos que farão o papel dos pais. Qualquer critério pode ser usado para compor esse par, como indivíduos que são diferentes geneticamente, ou agrupando os melhores e os piores separadamente. Porém, a escolha aleatória é a utilizada na maioria dos livros e aplicações, e, dada a sua simplicidade, será ela a abordada aqui. Tendo os pais em mãos, obtemos também aleatoriamente o ponto de corte, e estamos prontos para a reprodução (figura 8).

Entretanto, voltando à Natureza, sabemos que a reprodução não acontece necessariamente sempre, e o nosso algoritmo deve contemplar esse efeito. Fazemos isso através de uma probabilidade p_c associada à ocorrência do *Crossover*. Digamos que $p_c = 70\%$. Após escolhidos os pais e o ponto de corte, sorteamos um número p_{aux} entre 0 e 1 e, se $p_{aux} \leq p_c$, o *Crossover* acontece.

Esse processo deve ser repetido até que o número de indivíduos desejado seja atingido. Nas figuras 10 e 11 há um exemplo de código.

3.7.0.1 Exemplo: *crossover* para $y(x) = -x^2 + 36$

Para que o funcionamento do operador *crossover* fique claro, faremos uma execução manual do código contido na figura 10. Os indivíduos que utilizaremos serão aqueles selecionados pela Roleta, presentes na tabela 7. Ao final teremos quatro indivíduos provenientes de oito pares de cromossomos.

Entretanto, antes de continuarmos, explicaremos o algoritmo. A segunda linha, 248, inicializa a variável *flagCrossOver* com zero, ou *falso* na linguagem C. A função dela é garantir que em todas as iterações do laço *for* sairemos com um descendente.

Entramos no `while` e chamamos a função `GeraPontoDeCorte()`, cujo trabalho é definir o ponto de corte e armazená-lo em uma variável global. Conforme pode ser visto na figura 11, essa variável (`PontoDeCorte`) é utilizada na função `CrossOver()`.

Cada elemento do vetor `geracao` (linhas 251 e 252) armazena uma população com um número fixo de indivíduos, definida em `numIndividuos`. Então, para obter um cromossomo aleatoriamente, geramos, através da função `Randomico()`, um valor entre zero e (`numIndividuos` - 1). Esse número será a posição do indivíduo escolhido dentro da geração. As variáveis `indPai` e `indMae` receberão os dois indivíduos.

Em `probAux` guardamos um valor entre 0 e 1, obtido novamente de maneira aleatória. Se ele for menor ou igual a `probCrossOver` (probabilidade de ocorrer a Reprodução), os indivíduos `indPai` e `indMae` gerarão um filho, que será armazenado na posição `iIndividuo` da `geracao_auxiliar`. A `flagCrossOver` recebe `verdadeiro`, indicando que temos um descendente e podemos sair do `while`.

Voltemos à execução manual do nosso exemplo. Começamos definindo que a probabilidade de ocorrência da Reprodução será de 70%⁴:

$$\text{probCrossOver} = 0,7.$$

Através da tabela 7 lembramos que

$$\text{numIndividuos} = 4.$$

Ao entrar no `for` a `flagCrossOver` recebe `falso`:

$$\text{flagCrossOver} = 0.$$

Assim que chegamos ao `while` a função `GeraPontoDeCorte()` é executada. Como a nossa representação cromossomial possui três genes, há apenas dois pontos de corte possíveis (figura 8). Digamos que a função escolha

$$\text{PontoDeCorte} = 1,$$

e que

$$\text{probAux} = 0,86.$$

Para esse caso, a condição `probAux <= probCrossOver` é falsa, obviamente não entramos no `if` e voltamos ao início do `while`.

⁴ Não há um valor ótimo e absoluto para p_c . Para cada caso deve-se ajustar esse parâmetro e verificar a qualidade das soluções. Podemos afirmar apenas que p_c deve ser alta, entre 70% e 100%.

Entretanto, suponha que agora os seguintes valores tenham sido obtidos:

$$PontoDeCorte = 1 \text{ e } probAux = 0,65.$$

Na linha 251 *Randomico(0, numIndividuos)* retorna 0, e isso significa que *indPai* recebeu $x = 3 = 011$:

$$indPai = 011.$$

Em seguida, *Randomico(0, numIndividuos)* retorna 3^5 , e isso implica

$$indMae = 101.$$

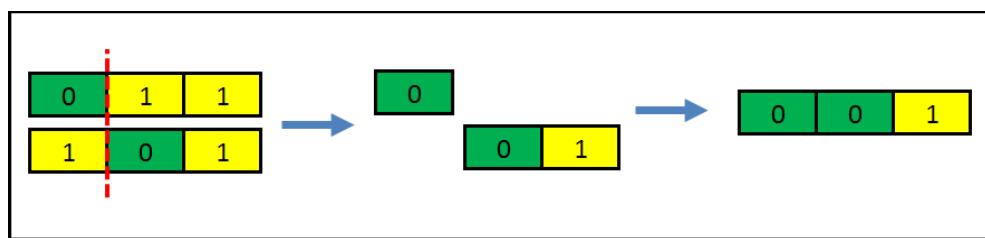


Figura 9 – Exemplo do *crossover* entre os indivíduos 011 e 101 para o primeiro ponto de corte.

O resultado da reprodução, veja a figura 9, é armazenado na primeira posição da *geracao_auxiliar*:

$$geracao_auxiliar[0] = 001.$$

Na geração atual (*Geração Final* na tabela 7) o indivíduo com melhor *fitness* é o $x = 2$, enquanto $x = 5$ possui a pior nota e $x = 3$ obteve uma avaliação intermediária. Por causa do caráter completamente aleatório do algoritmo, o melhor indivíduo não foi selecionado para gerar descendentes nessa reprodução. Poderíamos supor, num primeiro momento, que esse não tenha sido o melhor caminho. Entretanto, o resultado do *crossover* entre $x = 5$ e $x = 3$ gerou um indivíduo com o maior *fitness* desde a geração inicial. Usando $x = 1$ na $y(x)$ chegamos a $f_c(1) = 35$, maior que o $f(2) = 32$.

A tabela 8 apresenta os dados do nosso processo de Reprodução feito manualmente. É interessante notar que, apesar dessa nova geração ter perdido o indivíduo $x = 2$, a maior avaliação da população anterior, o *fitness* médio aumentou. Não só isso, mas também o melhor cromossomo tem uma nota maior do que todos os indivíduos anteriores.

⁵ Lembre-se de que, na linguagem C, o primeiro elemento de um vetor possui índice $i = 0$. Por isso o número 3 retornou o quarto elemento.

Tabela 8 – Geração antes e depois do *Crossover*. O melhor indivíduo dos descendentes possui o melhor *fitness* entre todas as gerações anteriores. Além disso, o *fitness* médio ($< f >$) também aumentou.

Geração Selecionada		Descendentes	
x	fitness (f)	x	fitness (f)
3	27	1	35
2	32	5	11
3	27	3	27
5	11	3	27
$< f > = 24,25$		$< f > = 25,00$	

Apesar de simples, o exemplo acima exibiu o papel fundamental do *Crossover* na busca pelas melhores soluções. Na próxima seção mostraremos como funciona a Mutação, essencial para a variabilidade genética.

```

247 | for (iIndividuo = 0; iIndividuo < numIndividuos; iIndividuo++) {
248 |   flagCrossOver = 0;
249 |   while (flagCrossOver = 0) {
250 |     GeraPontoDeCorte();
251 |     indPai = geracao[1].individuo[Randomico(0,numIndividuos)];
252 |     indMae = geracao[1].individuo[Randomico(0,numIndividuos)];
253 |     probAux = rand();
254 |     if (probAux <= probCrossOver) {
255 |       geracao_auxiliar.individuo[iIndividuo] = CrossOver(indPai, indMae);
256 |       flagCrossOver = 1;
257 |     };
258 |   };
259 | };

```

Figura 10 – Código para a Reprodução. Nesse exemplo o algoritmo gera apenas um descendente. Detalhes da função `CrossOver()` estão na figura 11.

```

76 | struct individual CrossOver(struct individual Pai, struct individual Mae) {
77 |   unsigned int iAux;
78 |   struct individual indAux;
79 |
80 |   for (iAux = 0; iAux < PontoDeCorte[0]; iAux++) {
81 |     indAux.gene[iAux] = Pai.gene[iAux];
82 |   };
83 |
84 |   for (iAux = PontoDeCorte[0]; iAux < numGenes; iAux++) {
85 |     indAux.gene[iAux] = Mae.gene[iAux];
86 |   };
87 |
88 |   return indAux;
89 | };
90 |

```

Figura 11 – Detalhes da função `CrossOver()`.

3.8 Mutação

Chegamos à última operação de um algoritmo genético típico, a Mutação. Assim como no *Crossover*, existe uma probabilidade p_m associada ao acontecimento da Mutação. Porém, nesse momento, não operaremos sobre um par de cromossomos, mas na estrutura interna de cada cromosso: os genes.

Se nossos indivíduos possuem dez genes, para cada um dos dez *locus* testamos a condição $p_{aux} \leq p_m$, onde p_{aux} é um valor entre zero e um, escolhido aleatoriamente. Caso a desigualdade seja verdadeira, invertemos o *bit* daquela posição e partimos para o próximo *locus*.

Seguindo com o exemplo da seção anterior, a população atual, depois da Seleção e do *Crossover*, possui o melhor indivíduo comparado com os anteriores, além do *fitness* médio também ter crescido (tabela 8 na página 32). Na tabela 9 listamos esses indivíduos e a sua representação cromossomial.

Tabela 9 – Representação cromossomial para os indivíduos que passaram pela Seleção e pelo *Crossover*.

x (decimal)	x (representação binária)
1	001
5	101
3	011
3	011

Do ponto de vista da informação genética, o indivíduo que mais se aproxima de $x = 0 = 000$, a solução ideal, é o $x = 1 = 001$. Portanto, bastaria evoluir essa população e, em algum momento, obteríamos a melhor solução, certo? Errado.

Nenhum indivíduo possui um zero na última posição. Então, mesmo fazendo todas as combinações possíveis entre os cromossomos, nunca chegaríamos ao indivíduo $x = 0$. O único que possuía tal característica era o $x = 2$, presente na população inicial mas “extinto” ao longo da evolução.

É aí que entra a Mutação: ainda que baixa, geralmente em torno de 1%, há uma chance do último *bit* de algum indivíduo sofrer mutação e ser alterado para zero. E esse é um dos pontos que torna a Mutação tão importante: ela insere uma nova informação genética que, ou foi perdida, ou não estava presente na população inicial (Heurística Exploratória).

Obviamente, existe a possibilidade de bons esquemas serem destruídos com a mudança em algum “*bit* errado”. Mesmo assim, essa chance é equivalente à probabilidade de transformarmos um péssimo esquema em um excelente espaço de soluções. Nessa linha,



Figura 12 – Representação gráfica de uma mutação. Nesse exemplo a mutação no último *bit* levou à solução ótima para o máximo da função $y(x) = x^2 + 36$.

a Mutação é a responsável por manter a diversidade e evitar a Convergência Genética (seção 3.6.1).

Qual deve ser o valor de p_m para uma Mutação eficiente? Não há uma resposta absoluta. Muitos pesquisadores e usuários dos GAs utilizam valores entre 0,5% e 1%, simplesmente porque nos primeiros trabalhos bons resultados foram obtidos com eles. Sabe-se que há um valor ideal, mas ele é diferente para cada problema e para cada representação cromossomial.

Apesar disso, há um consenso de que o valor de p_m deve ser pequeno, bem menor do que p_c . Em parte porque na genética natural essa probabilidade é de fato pequena, mas principalmente, no contexto da computação, porque valores grandes fariam com que a busca por soluções se comportasse de maneira semelhante ao *Random Walk*.

Partindo para o lado prático, o código para implementar a mutação é mais simples, e um exemplo encontra-se na figura 13. Na linha 281 um determinado indivíduo recebe ele próprio após o operador Mutação. Não há *if* ou outra condição, ou seja, aplicamos o operador em todos os indivíduos da população.

Mas, afinal, onde está a probabilidade p_m ? Lembre que a mutação, caso aconteça, deve ocorrer isoladamente em cada gene. Por isso usamos p_m dentro da função *Mutacao()*. Ela recebe como parâmetro um indivíduo que possui um número de genes igual a *numGenes*, uma constante definida de maneira global.

Entramos em um *for* que irá varrer todos os genes do *Individuo* e, para cada um, faz o teste $p_{aux} \leq p_m$. Se a condição retornar *verdadeiro*, a Mutação é finalmente expressa como a inversão do *bit* no *locus* atual.

Depois que todos os indivíduos da população passarem pelo operador Mutação, o ciclo estará fechado. A nova geração está pronta para passar por todo o processo: Avaliação, Seleção, Reprodução e Mutação.

```
281 geracao_auxiliar.individuo[iIndividuo] =
282         Mutacao(geracao_auxiliar.individuo[iIndividuo]);
283 }

91 struct individual Mutacao(struct individual Individuo) {
92     unsigned int iGene, probAux;
93     const float probMutacao = (float)0.01;
94
95     for (iGene = 0; iGene < numGenes; iGene++) {
96         probAux = rand();
97         //printf("\n<<TESTE>> probAux = %d", probAux);
98         if (probAux <= probMutacao) {
99             //printf("\n<<TESTE>> Mutacao no gene %d", iGene);
100            if (Individuo.gene[iGene] == 1) {
101                Individuo.gene[iGene] = 0;
102            }
103            else {
104                Individuo.gene[iGene] = 1;
105            }
106        }
107    }
108 }
```

Figura 13 – Exemplo de código para o operador Mutação.

4 Autovalores de Matrizes Simétricas com Algoritmos Genéticos

Os métodos que estudei para esta dissertação estão contidos em uma série de artigos ([NANDY et al., 2002](#); [NANDY et al., 2004](#); [SHARMA et al., 2006](#); [SHARMA et al., 2008](#); [NANDY et al., 2009](#); [NANDY et al., 2011](#)). O objetivo é encontrar, sequencialmente, do menor para o maior, os autovalores de uma matriz simétrica. Essa matriz é o Hamiltoniano presente na formulação matricial da equação de Schrödinger independente do tempo¹. A estratégia é transformar o problema do autovalor em um problema de otimização, buscando um escalar a_i e um vetor X_i de modo que a equação $H X_i = a_i X_i$ seja satisfeita.

Os algoritmos apresentados em ([NANDY et al., 2004](#)) e ([NANDY et al., 2011](#)) são mais simples. Por exemplo, em ([NANDY et al., 2002](#)) o Hamiltoniano é alterado por algumas transformações ortogonais e, só então, o *fitness* é calculado. Nos artigos ([SHARMA et al., 2006](#)), ([SHARMA et al., 2008](#)) e ([NANDY et al., 2009](#)) o espaço vetorial é dividido em duas partes de dimensões diferentes, levando a um Hamiltoniano que contém alguns autovalores de interesse². Isso não ocorre com os GAs das publicações de 2004 e 2011. Nelas o Hamiltoniano original é sempre mantido.

Pode-se dizer que ([NANDY et al., 2011](#)) é a continuação de ([NANDY et al., 2004](#)). A representação cromossomial e os operadores de seleção, *crossover* e mutação são os mesmos. No entanto, ele adiciona dois operadores genéticos. O primeiro é complementar à mutação, e atua para criar mais diversidade na população. O segundo acentua a pressão seletiva via Elitismo. Não há justificativa para os novos operadores. Acredito que o intuito tenha sido melhorar a qualidade dos resultados, mas, infelizmente, não há comparação com os obtidos em 2004. E, de fato, isso seria impossível, pois em 2011 há uma mudança drástica: a função de avaliação foi alterada. Além disso, ([NANDY et al., 2011](#)) paraleliza o GA e compara os desempenhos.

Assim, optei por seguir uma combinação entre ([NANDY et al., 2004](#)) e ([NANDY et al., 2011](#)). Isso permitiu, por exemplo, comparar com segurança os resultados das duas funções de avaliação, algo que os autores não fizeram. A partir do próximo parágrafo descrevo o conteúdo dos dois artigos e como fiz a combinação entre os dois.

É possível expandir as soluções ψ da Equação de Schrödinger independente do

¹ Ela é importante para a física moderna pois está associada à quantização da energia na Mecânica Quântica.

² *Partitioned matrix eigenvalue problem*.

tempo em termos dos vetores geradores ϕ_k de uma base ortonormal finita ϕ :

$$\psi = \sum_k c_k \phi_k. \quad (4.1)$$

Isso leva ao problema de autovalores

$$HC = EC, \quad (4.2)$$

onde H é uma matriz real e simétrica, construída na base ϕ , que representa o operador Hamiltoniano. A diagonalização de H encontra os autovalores E_n correspondentes às energias possíveis do sistema quântico. Com isso, é possível obter os autovetores (autoestados) C_n associados.

No artigo ([NANDY et al., 2004](#)) é apresentada uma maneira de reduzir o problema de autovalores a um problema de busca. Dados todos os vetores C existentes na base ϕ , conjunto chamado de **Espaço de Busca**, o objetivo é encontrar os autovetores C_n que satisfaçam a equação 4.2. O conjunto de todos os autovetores C_n é o **Espaço de Soluções**. O mecanismo de busca é um algoritmo genético (GA).

Conforme visto no capítulo 3, o elo entre o GA e o problema a ser resolvido está na Representação Cromossomial e na Função de Avaliação. O cromossomo deve codificar a solução desejada na forma de um *string*, seja de caracteres, símbolos, números inteiros ou reais. O *fitness* deve ser capaz de definir, objetivamente, a qualidade de todos os indivíduos da população, de modo que seja possível comparar cada um com as soluções desejadas. Quanto mais próximo um cromossomo está da solução, mais alto deve ser seu *fitness*.

4.1 Representação Cromossomial

Como a solução pretendida é um autovetor, o cromossomo codifica um vetor. Cada indivíduo i da população é um vetor ψ_i candidato a autovetor na forma

$$\psi_i = \sum_{p=1}^m c_{pi} \phi_p, \quad i = 1, 2, \dots, n \quad (4.3)$$

Na equação acima, i varia de 1 até o número máximo de indivíduos na população do GA, mantida fixa ao longo de toda a execução. O índice p é tomado de 1 até a m , que é ordem da matriz H (ou a dimensão do espaço vetorial).

O cromossomo é definido como uma cadeia de números reais, cujos valores são os coeficientes c_{pi} . O *string* S_i , codificação para o membro ψ_i , é dado por

$$S_i \equiv (c_{1i}, c_{2i}, \dots, c_{pi}, \dots, c_{mi}) = C_i^\dagger, \quad (4.4)$$

enquanto que para outro membro ψ_k da população, o *string* S_k é

$$S_k \equiv (c_{1k}, c_{2k}, \dots, c_{pk}, \dots, c_{mk}) = C_k^\dagger. \quad (4.5)$$

4.2 População

A população inicial é gerada aleatoriamente. Não há nenhuma preferência com relação aos valores iniciais de cada gene.

Seu tamanho (número de indivíduos) é sempre mantido fixo a cada nova geração.

4.3 Funções de Avaliação (*fitness*)

O *fitness* proposto em (NANDY *et al.*, 2004) é obtido em três passos:

1. Cálculo do quociente de Rayleigh ρ_i associado ao i -ésimo indivíduo C_i ;
2. Cálculo do gradiente de ρ_i ;
3. Cálculo da função de avaliação.

Retomando o capítulo 2, o quociente de Rayleigh é dado por

$$\rho_i = \frac{C_i^\dagger H C_i}{C_i^\dagger C_i}, \quad (4.6)$$

e seu gradiente por

$$\nabla \rho_i = \frac{2[H - \rho_i]C_i}{C_i^\dagger C_i}. \quad (4.7)$$

O *fitness* é então definido como ³

$$f_i = e^{-\beta(\nabla \rho_i)^\dagger (\nabla \rho_i)}. \quad (4.8)$$

³ No artigo (NANDY *et al.*, 2004) a equação originalmente apresentada é $f_i = e^{-\beta(\nabla \rho_i)^\dagger (\nabla \rho_i)}$. Acredito que tenha sido um erro de digitação por dois motivos. O primeiro é que tal definição não leva, necessariamente, ao comportamento esperado: $f_i \rightarrow 1$ quando $\nabla \rho_i \rightarrow \mathbf{0}$. Em segundo lugar, nos artigos (SHARMA *et al.*, 2006), (SHARMA *et al.*, 2008) e (NANDY *et al.*, 2009), que seguem o mesmo método, a função de avaliação é definida como $f_i = e^{-\beta(\nabla \rho_i)^\dagger (\nabla \rho_i)}$. Portanto, segui com a definição de f_i da equação 4.8.

Lembrando que o módulo de um vetor V é dado por $|V| = \sqrt{V^\dagger V}$, a equação 4.8 fica

$$f_i(\nabla \rho_i) = e^{-\beta |\nabla \rho_i|^2}. \quad (4.9)$$

A função de avaliação f_i está limitada entre zero e um, $f_i = (0,1]$. Se $|\nabla \rho_i|^2 \gg 1$, $f_i \rightarrow 0$, indicando que C_i possui baixa qualidade, está longe da solução. Por outro lado, se $\nabla \rho_i \rightarrow \mathbf{0}$, $f_i \rightarrow 1$, e C_i é uma boa aproximação para um autovetor. O parâmetro β é escolhido para não haver *over flow* ou *under flow* da função exponencial.

De acordo com os autores, a equação 4.9 leva ao autovalor mínimo. Se algum C_i , em algum momento, é o autovetor fundamental C_0 , o $\nabla \rho$ é nulo. Eles afirmam que “Claramente, $f_i \rightarrow 1$ quando $\nabla \rho_i \rightarrow 0$, sinalizando que a evolução atingiu o verdadeiro autovetor fundamental de H em C_i ”⁴.

Uma vez que C_0 foi encontrado, o próximo passo é obter o autovalor E_0 associado. Na verdade ele já foi calculado, e é simplesmente o valor do quociente de Rayleigh para C_i :

$$\rho_0 = \frac{C_i^\dagger H C_i}{C_i^\dagger C_i} = E_0. \quad (4.10)$$

Quando o algoritmo chega nesse estágio tem-se o par (C_0, E_0) .

Como já dito anteriormente, a função de avaliação foi alterada em ([NANDY et al., 2011](#)), e é dada por

$$f_i(\rho_i) = e^{-\beta(\rho_i - E_L)^2}, \quad (4.11)$$

onde E_L é um limite inferior para o autovalor mínimo procurado.

Ela compartilha algumas propriedades com a equação 4.9. Está limitada no conjunto $f_i(\rho) = (0, 1]$ e, quanto maior seu valor, melhor a qualidade do indivíduo. O parâmetro β tem exatamente a mesma função. A busca utilizando $f_i(\rho)$ também termina quando $f_i \rightarrow 1$. “Se $\rho_i \rightarrow E_L$ durante a busca, $f_i \rightarrow 1$ e C_i se aproxima do autovetor fundamental de H ”⁵. Novamente, como já temos C_0 , o autovalor E_0 é simplesmente o ρ_i já calculado. O cálculo de $f_i(\rho)$ executa os passos 1 e 3 de $f_i(\nabla \rho)$.

A não ser por acidente, a condição $f_i \rightarrow 1$ não é satisfeita logo na primeira população. É necessário evoluir os indivíduos por meio dos operadores genéticos de seleção, reprodução e mutação. Eles serão apresentados nas seções seguintes.

⁴ Tradução livre de “Clearly, $f_i \rightarrow 1$, as $\nabla \rho_i \rightarrow 0$, signalling that the evolution has hit the true ground state eigenvector of H in the vector C_i ”.

⁵ Tradução livre de “If $\rho_i \rightarrow E_L$ during the search, $f_i \rightarrow 1$ and C_i approaches the ground eigenvector of H ”.

4.4 Seleção

O operador de seleção utilizado tanto em (NANDY *et al.*, 2004) quanto em (NANDY *et al.*, 2011) é o da Roleta, com fatias proporcionais aos valores do *fitness* (LINDEN, 2008). Se a população possui q indivíduos, a roleta é “girada” q vezes, de modo a criar a nova população com os q cromossomos selecionados.

Entretanto, utilizei a seleção por Torneio pelos motivos apresentados na seção 3.6.4. Mantive o tamanho da população fixa.

4.5 Reprodução

A operação de reprodução (*crossover*) é aplicada na nova população após a Seleção. Há diferença entre os operadores de reprodução de (NANDY *et al.*, 2004) e (NANDY *et al.*, 2011). Apresentarei ambos e, ao final, justificarei porque escolhi como base o segundo.

4.5.1 Crossover em (NANDY *et al.*, 2004)

Suponha que tenham sido escolhidos, aleatoriamente, um par de cromossomos (S_k , S_l) dentre todos os N indivíduos da população:

$$\begin{aligned} S_k &= (c_{k1}, c_{k2}, \dots, c_{kn}) \\ S_l &= (c_{l1}, c_{l2}, \dots, c_{ln}) \end{aligned} \quad (4.12)$$

Em seguida, um inteiro p é obtido, também aleatoriamente, entre 1 e $n-1$ [$p = [1, n]$]. Lembre-se que n é a ordem da matriz H e, portanto, a quantidade de coeficientes no *string* S_i . A função de p é determinar em qual posição (*locus*) do cromossomo acontecerá a alteração.

O operador cria o novo par (S'_k , S'_l):

$$\begin{aligned} S'_k &= (c_{k1}, c_{k2}, \dots, c'_{kp} c_{k,p+1}, \dots, c_{kn}) \\ S'_l &= (c_{l1}, c_{l2}, \dots, c'_{lp} c_{l,p+1}, \dots, c_{ln}), \end{aligned} \quad (4.13)$$

onde

$$\begin{aligned} c'_{kp} &= f c_{kp} + (1-f) c_{lp} \\ c'_{lp} &= (1-f) c_{kp} + f c_{lp}. \end{aligned} \quad (4.14)$$

O parâmetro f faz o papel da mistura que cria nova informação. Ele é gerado aleatoriamente com valores entre zero e um. Nesse caso os valores limite não estão inclusos [$f = (0,1)$]. Dessa maneira há garantia de mistura.

Esse operador só age em uma certa fração da população, dada por uma probabilidade p_c que, em geral, é grande (70–75%). O restante da população não é alterado.

Exemplo

Na figura 14 há dois cromossomos de tamanho $n = 6$. A posição onde ocorrerá a alteração dos valores foi obtida aleatoriamente entre $p = [1, 6)$, e, para esse caso, vale $p = 4$. O valor de S_k na posição $p = 4$ é $c_{k4} = 0,80$, e para S_l é $c_{l4} = 0,39$. Para $p + 1$ os valores são $c_{k5} = 0,15$ e $c_{l5} = 0,89$.

	1	2	3	4	5	6
S_k	0,50	0,47	0,52	0,80	0,15	0,38
S_l	0,58	0,33	0,37	0,39	0,89	0,13

Figura 14 – Exemplo do *crossover* de (NANDY *et al.*, 2004). Indivíduos antes da reprodução.

O parâmetro f teve seu valor determinado aleatoriamente como $f = 0,62$. Os valores de c'_{k4} e c'_{l4} ficam:

$$\begin{aligned} c'_{kp} &= fc_{kp} + (1 - f)c_{lp} \\ c'_{k4} &= 0,62c_{k4} + (1 - 0,62)c_{l4} \\ c'_{k4} &= 0,62 * 0,80 + 0,38 * 0,39 \\ c'_{k4} &= 0,496 + 0,1482 \\ c'_{k4} &= 0,6442 \end{aligned} \quad (4.15)$$

$$\begin{aligned} c'_{lp} &= (1 - f)c_{kp} + fc_{lp} \\ c'_{l4} &= (1 - 0,62)c_{k4} + 0,62c_{l4} \\ c'_{l4} &= 0,38 * 0,80 + 0,62 * 0,39 \\ c'_{l4} &= 0,304 + 0,2418 \\ c'_{l4} &= 0,5458 \end{aligned} \quad (4.16)$$

Finalmente, os valores para a posição $p = 4$ nos novos indivíduos S'_k e S'_l são

$$\begin{aligned} \text{Novo valor na posição 4 de } S'_k &= c'_{kp}c_{k,p+1} \\ &= c'_{k4}c_{k5} \\ &= 0,6442 * 0,15 \\ &= 0,09663 \\ &\approx 0,1 \end{aligned} \quad (4.17)$$

$$\begin{aligned}
 \text{Novo valor na posição 4 de } S'_l &= c'_{lp} c_{l,p+1} \\
 &= c'_{l4} c_{l5} \\
 &= 0,5458 * 0,89 \\
 &= 0,485762 \\
 &\approx 0,49
 \end{aligned} \tag{4.18}$$

Na figura 15 há os dois novos indivíduos.

	1	2	3	4	5	6
S'ₖ	0,50	0,47	0,52	0,10	0,15	0,38
S'ₗ	0,58	0,33	0,37	0,49	0,89	0,13

Figura 15 – Exemplo do *crossover* de (NANDY *et al.*, 2004). Indivíduos depois da reprodução.

4.5.2 Crossover em (NANDY *et al.*, 2011)

A representação cromossomial usada em (NANDY *et al.*, 2011) é a mesma de (NANDY *et al.*, 2004), portanto, o par (S_k, S_l) é igual, assim como a probabilidade p_c :

$$\begin{aligned}
 S_k &= (c_{k1}, c_{k2}, \dots, c_{kn}) \\
 S_l &= (c_{l1}, c_{l2}, \dots, c_{ln})
 \end{aligned} \tag{4.19}$$

Diferentemente de (NANDY *et al.*, 2004), utiliza-se o *crossover* de dois pontos. Aleatoriamente escolhe-se dois inteiros, o e p , cuja função é determinar a região do cromossomo que sofrerá miscigenação. O valor em o indica o primeiro gene, e p o último. Portanto, todos os genes entre os dois, incluindo eles próprios, sofrerão a ação do *crossover* ($o \leq c_i \leq p$, $p \geq o$). Os novos indivíduos são (S'_k, S'_l)

$$\begin{aligned}
 S'_k &= (c_{k1}, c_{k2}, \dots, c'_{ko}, \dots, c'_{kp}, c_{k,p+1}, \dots, c_{kn}) \\
 S'_l &= (c_{l1}, c_{l2}, \dots, c'_{lo}, \dots, c'_{lp}, c_{l,p+1}, \dots, c_{ln}).
 \end{aligned} \tag{4.20}$$

Para todos genes selecionados, a transformação ocorre da seguinte maneira ($i = o, o + 1, \dots, p$):

$$\begin{aligned}
 c'_{ki} &= f_c c_{ki} + (1 - f_c) c_{li} \\
 c'_{li} &= (1 - f_c) c_{ki} + f_c c_{li}
 \end{aligned} \tag{4.21}$$

onde f_c é dado por

$$f_c = 0,75 + 0,25r, \tag{4.22}$$

sendo r um número aleatório ($0 \leq r \leq 1$). Assim como o parâmetro f de (NANDY *et al.*, 2004), f_c faz o papel da mistura que cria nova informação.

Exemplo.

Usei os mesmos indivíduos da figura 14. Os parâmetros o e p foram escolhidos no intervalo $[1,6]$ e têm valores $o = 2$ e $p = 4$. Portanto, no *string* S_k os elementos que sofrerão alteração são $c_{k2} = 0,47$, $c_{k3} = 0,52$ e $c_{k4} = 0,80$. Eles se misturarão com os elementos $c_{l2} = 0,33$, $c_{l3} = 0,37$ e $c_{l4} = 0,39$ de S_l . Aleatoriamente obtive $r = 0,2$, que leva a $f_c = 0,80$.

	1	2	3	4	5	6
S_k	0,50	0,47	0,52	0,80	0,15	0,38
S_l	0,58	0,33	0,37	0,39	0,89	0,13

Figura 16 – Exemplo do *crossover* de (NANDY *et al.*, 2011). Indivíduos antes da reprodução.

Os elementos c'_{ki} são:

$$\begin{aligned}
 c'_{k2} &= f_c c_{k2} + (1 - f_c) c_{l2} \\
 &= 0,80 * 0,47 + (1 - 0,80) * 0,33 \\
 &= 0,376 + 0,2 * 0,33 \\
 &= 0,376 + 0,066 \\
 &= 0,442 \\
 &\approx 0,44
 \end{aligned} \tag{4.23}$$

$$\begin{aligned}
 c'_{k3} &= f_c c_{k3} + (1 - f_c) c_{l3} \\
 &= 0,80 * 0,52 + (1 - 0,80) * 0,37 \\
 &= 0,416 + 0,2 * 0,37 \\
 &= 0,416 + 0,074 \\
 &= 0,49
 \end{aligned} \tag{4.24}$$

$$\begin{aligned}
 c'_{k4} &= f_c c_{k4} + (1 - f_c) c_{l4} \\
 &= 0,80 * 0,80 + (1 - 0,80) * 0,39 \\
 &= 0,64 + 0,2 * 0,39 \\
 &= 0,64 + 0,078 \\
 &= 0,718 \\
 &\approx 0,72
 \end{aligned} \tag{4.25}$$

Os elementos c'_{li} são:

$$\begin{aligned}
 c'_{l2} &= (1 - f_c)c_{k2} + f_c c_{l2} \\
 &= (1 - 0,80) * 0,47 + 0,80 * 0,33 \\
 &= 0,2 * 0,47 + 0,264 \\
 &= 0,094 + 0,264 \\
 &= 0,358 \\
 &\approx 0,36
 \end{aligned} \tag{4.26}$$

$$\begin{aligned}
 c'_{l3} &= (1 - f_c)c_{k3} + f_c c_{l3} \\
 &= (1 - 0,80) * 0,52 + 0,80 * 0,37 \\
 &= 0,2 * 0,52 + 0,296 \\
 &= 0,104 + 0,296 \\
 &= 0,40
 \end{aligned} \tag{4.27}$$

$$\begin{aligned}
 c'_{l4} &= (1 - f_c)c_{k4} + f_c c_{l4} \\
 &= (1 - 0,80) * 0,80 + 0,80 * 0,39 \\
 &= 0,2 * 0,80 + 0,312 \\
 &= 0,16 + 0,312 \\
 &= 0,472 \\
 &\approx 0,47
 \end{aligned} \tag{4.28}$$

Os novos indivíduos estão na figura 17.

	1	2	3	4	5	6
S_k	0,50	0,44	0,49	0,72	0,15	0,38
S_I	0,58	0,36	0,40	0,47	0,89	0,13

Figura 17 – Exemplo do *crossover* de (NANDY *et al.*, 2011). Indivíduos depois da reprodução.

4.5.3 Crossover utilizado

A quantidade de nova informação não é escalável com o tamanho do cromossomo no *crossover* de (NANDY *et al.*, 2004). Ele altera apenas uma posição (locus) de cada indivíduo, independentemente do tamanho do cromossomo. Por exemplo, se a ordem do Hamiltoniano for $n = 100$, o *string* terá 100 elementos, mas apenas um sofrerá alteração. O mesmo aconteceria para $n = 1.000, 10.000$ e assim por diante.

Ainda sobre o artigo de 2004, é impossível haver troca de informação na última posição. O parâmetro p , obtido aleatoriamente, dá a posição na qual ocorrerá o *crossover*.

Porém, parte do cálculo envolve os elementos $c_{k,p+1}$ e $c_{l,p+1}$. Note o índice $p + 1$. Como a posição máxima é n , o maior índice possível para estes elementos é $p + 1 = n$, impondo o limite $p \leq n - 1$. Nos exemplos apresentados, $n = 6$, $p \leq 5$ e, portanto, a posição 6 nunca será escolhida.

Isso pode dificultar a busca. Suponha que uma solução precise do valor 23 no último coeficiente do cromossomo ($c_n = 0,23$). Se nenhum indivíduo da população inicial já tiver nascido com ele, esse valor nunca aparecerá por meio do *crossover*. Resta como esperança a Mutação, porém, por definição, ela tem baixa probabilidade (LINDEN, 2008). Também não há garantia de que, após muitas gerações com sucessivas operações de *crossover*, mutação e seleção, haverá um indivíduo na população que, além de ter 0,23 na última posição, possua os outros $(n - 1)$ primeiros elementos necessários.

O *crossover* de (NANDY *et al.*, 2004) é um caso especial do *crossover* de (NANDY *et al.*, 2011). O operador de reprodução de (NANDY *et al.*, 2011) é o clássico *crossover* de dois pontos (LINDEN, 2008). Nele, $p \geq o$, e, se $o = p$, há mistura em apenas uma posição do *string*, exatamente o que acontece em (NANDY *et al.*, 2004). Como $1 \leq o \leq n$ e $1 \leq p \leq n$, o último gene também está sujeito à operação, e o problema citado no parágrafo anterior não existe.

Pelo que expus acima, escolhi utilizar o *crossover* de (NANDY *et al.*, 2011), mas com uma pequena modificação. No cálculo dos novos coeficientes c'_i , em vez do parâmetro f_c , usei o f conforme definido em (NANDY *et al.*, 2004). O f pode variar entre 0 e 1, enquanto o f_c , além de necessitar do parâmetro adicional r , é limitado apenas entre 0,75 e 1. Assim, acredito que f seja mais abrangente como parâmetro de mistura e criação de nova informação.

Crossover utilizado

A seguir apresento sinteticamente a estrutura do *crossover* utilizado. Um par de indivíduos (S_k, S_l)

$$\begin{aligned} S_k &= (c_{k1}, c_{k2}, \dots, c_{kn}) \\ S_l &= (c_{l1}, c_{l2}, \dots, c_{ln}) \end{aligned} \tag{4.29}$$

é obtido aleatoriamente da população. Com probabilidade p_c , a operação de *crossover* acontece. Dois pontos de corte o e p são obtidos, também aleatoriamente, gerando os novos *strings* S'_k e S'_l na forma

$$\begin{aligned} S'_k &= (c_{k1}, c_{k2}, \dots, c'_{ko}, \dots, c'_{kp}, c_{k,p+1}, \dots, c_{kn}) \\ S'_l &= (c_{l1}, c_{l2}, \dots, c'_{lo}, \dots, c'_{lp}, c_{l,p+1}, \dots, c_{ln}), \end{aligned} \tag{4.30}$$

onde

$$\begin{aligned} c'_{ki} &= fc_{ki} + (1 - f)c_{li} \\ c'_{li} &= (1 - f)c_{ki} + fc_{li} \end{aligned} \quad (4.31)$$

e

$$0 \leq f \leq 1 \text{ (obtido aleatoriamente)} \quad (4.32)$$

4.6 Mutação

O operador Mutação é semelhante nos dois artigos. Todos os cromossomos estão sujeitos à mutação. Se um indivíduo k sofre mutação no gene q , o antigo valor c'_{kq} é alterado para c''_{kq} com a seguinte equação:

$$c''_{kq} = c'_{kq} + (-1)^L r \Delta, \quad (4.33)$$

onde L é um número inteiro, r um número aleatório ($0 \leq r \leq 1$) e Δ é a intensidade da mutação.

Não ficou claro para mim qual a relação entre a probabilidade p_m e como será a mutação no artigo (NANDY *et al.*, 2004). Os autores dizem que todos os *indivíduos* estão sujeitos ao operador, mas não citam quais *genes* podem sofrer mutação. Essa informação está explícita em (NANDY *et al.*, 2011), que permite mutação, caso aconteça, em apenas um gene de cada indivíduo. Ou seja, se houver mutação logo no primeiro gene de um cromossomo, o algoritmo parte para o próximo indivíduo.

Assumi que no artigo (NANDY *et al.*, 2004) os autores utilizaram o operador clássico. Conforme a literatura (MITCHELL, 1998; LINDEN, 2008) o operador clássico de mutação age com probabilidade p_m em cada gene, de maneira individual e independente. Ou seja, se um gene $c_{k,j}$ sofreu ou não mutação, essa informação não é utilizada para avaliar a ocorrência de mutação no próximo gene $c_{k,j+1}$. Portanto, a probabilidade de cada *gene* sofrer mutação é p_m , mas a probabilidade P_m de um *indivíduo* sofrer mutação é, na verdade, $P_m = n * p_m$. Quanto maior o n , mais provável ocorrer a mutação, o que me parece razoável. Mutação é causada principalmente por erros de cópia do DNA. Quanto maior a fita, mais chance de acontecer erro.

Há diferença na intensidade da mutação Δ . No trabalho de 2004 ela é pequena ($10^{-2} - 10^{-3}$) e mantida constante. Já no segundo artigo ela é proporcional ao melhor *fitness* f_t da geração atual t , e dada pela equação 4.34. Como o *fitness* começa pequeno e

termina próximo de 1, $\Delta \rightarrow 0$ à medida que $f \rightarrow 1$.

$$\Delta_m^{(t)} = 1 - f_t. \quad (4.34)$$

Não fiquei confortável com a definição da equação 4.34, por isso usei a intensidade Δ constante de (NANDY *et al.*, 2004). No início de um GA a criação de variabilidade genética é dominada pelo *crossover*. No final, como os indivíduos são parecidos, fica a cargo da mutação fazer isso. A intensidade da mutação deve ser pequena e, se não for constante, é desejável que cresça com o tempo, justamente porque o papel da mutação passa a ser dominante na variabilidade comparado com o *crossover* (LINDEN, 2008). Os autores de (NANDY *et al.*, 2011) fazem o inverso. Como dito anteriormente, Δ diminui com o tempo, mas eles não justificam porque optaram por esse comportamento.

Com relação à probabilidade de mutação, optei novamente a favor de (NANDY *et al.*, 2004). Ambos os trabalhos utilizam valor constante de p_m . Entretanto, em (NANDY *et al.*, 2011) p_m é inversamente proporcional ao tamanho do cromossomo n (equação 4.35). Na prática, quanto maior a ordem n do Hamiltoniano, menor a probabilidade de mutação, e no artigo não há justificativa para essa escolha. Fiquei com a probabilidade constante, discutida em livros tradicionais de GA (MITCHELL, 1998; LINDEN, 2008).

$$p_m = 4.0/n \quad (4.35)$$

Mutação utilizada.

Abaixo resumo a estrutura da Mutação implementada na dissertação.

- Probabilidade de mutação p_m

- Constante.
- Um dos parâmetros do algoritmo.
- Pode ter qualquer valor $0 \leq p_m \leq 1$.

- Equação da mutação.

- No indivíduo S , transforma o valor c'_{kq} do gene q no novo valor c''_{kq} .
- Equação: $c''_{kq} = c'_{kq} + (-1)^L r \Delta$
- Parâmetro L : inteiro aleatório.
- Parâmetro r : aleatório ($0 \leq r \leq 1$).
- Intensidade de mutação Δ : Constante. Valores pequenos ($10^{-3} - 10^{-2}$).

4.7 Fluxograma do algoritmo implementado

O algoritmo genético seguiu a estrutura de um GA básico (MITCHELL, 1998; LINDEN, 2008). Após gerar a população inicial, Avaliação, Seleção, *Crossover* e Mutação são executados nessa sequência até que uma condição de parada seja atingida.

Na figura 18 está o fluxograma. Ele não é um diagrama completo do *software* desenvolvido, mas é útil para ter a visão geral do algoritmo.

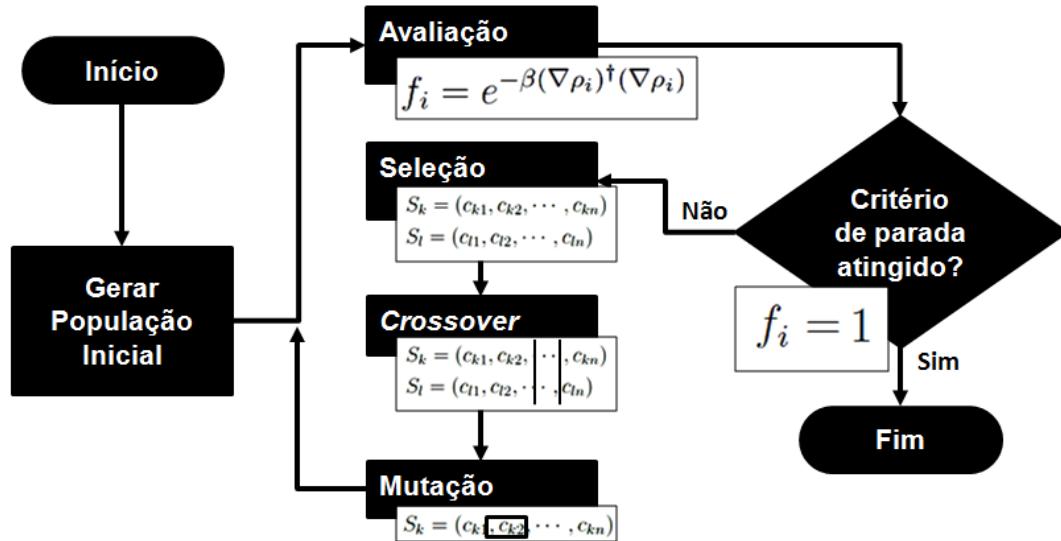


Figura 18 – Fluxo algoritmo genético

5 Metodologia

A metodologia possuiu dois pilares: sólida base em Algoritmos Genéticos e desenvolvimento de um *software* confiável para execução do modelo. Ambos são apresentados nas seções seguintes.

5.1 Algoritmos Genéticos

Iniciei os estudos em Algoritmos Genéticos pelos livros ([MITCHELL, 1998](#)) e ([LINDEN, 2008](#)) e, antes de partir para o trabalho da dissertação em si, trabalhei com três problemas completamente distintos.

O primeiro foi o ONEMAX ([DEBATTISTI et al., 2009](#)), desenvolvido “do zero”, em Linguagem C. Considerado o *Hello world* do GA, tem representação cromossomial binária, o *fitness* é a soma dos *bits* de cada indivíduo e o objetivo é obter um indivíduo com o maior número de ‘1’ possível. Com ele pude estudar os parâmetros de um GA simples, como número de indivíduos e a probabilidade de *crossover*, e verificar a influência de cada um na qualidade da solução, convergência, desempenho, evolução do *fitness* etc. Uma versão paralelizada em CUDA foi apresentada em evento de computação de alto desempenho ([PRIETO; COLUCI, 2012](#)) e encontra-se no apêndice B.

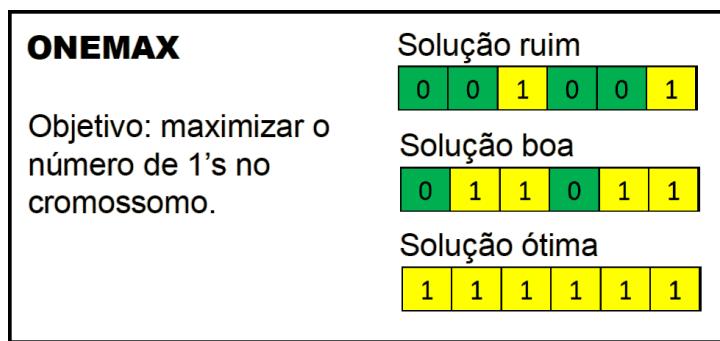


Figura 19 – ONEMAX: representação cromossomial, *fitness* e objetivo.

Tendo a estrutura básica do ONEMAX, o próximo passo foi tentar fazer um modelo. Abordei o Problema das Oito Rainhas, que consiste em posicionar oito rainhas em um tabuleiro de xadrez de modo que não se ataquem. Sem nenhuma referência externa, propus um modelo de GA que conseguiu chegar em algumas soluções ([PRIETO, 2012](#)). Uma delas está na figura 20.

Tratando o tabuleiro de xadrez como um plano cartesiano, a representação cromossomial era um *string* com os oito pares de coordenadas (x, y) . Cada coordenada gerava uma matriz característica de 0’s e 1’s que, quando somadas, resultava em uma



Figura 20 – Uma solução para o Problema das 8 Rainhas.

matriz com a informação da quantidade c de ataques que cada rainha sofreu. O objetivo, portanto, era encontrar um indivíduo que levasse a $c = 0$. A função de avaliação foi definida como $f = 1/(1 + c)$.

Criar um modelo para o Oito Rainhas foi importante para o entendimento do elo entre um GA e o problema a ser resolvido. Essa ligação encontra-se na representação cromossomial e na função de avaliação. Defini ambas de maneira adequada, mas a representação cromossomial apresentou um problema: nada impede de haver pontos (x, y) repetidos no cromossomo. Em outras palavras, ela permite que duas rainhas sejam colocadas na mesma posição do tabuleiro, o que é proibido no xadrez.

Por fim, utilizei GA na criação de um robô, chamado *Genético*, para ser testado contra os campeões do torneio de Robocode da Faculdade de Tecnologia da Unicamp¹. Robocode é um jogo², cujo objetivo é programar um tanque de guerra robô para competir contra outros robôs em uma arena de batalha. Ele começou como um projeto pessoal no ano 2000 e depois foi incorporado pela IBM³. Atualmente é um projeto de código aberto.



Figura 21 – Arena de batalha do Robocode.

¹ <http://torneiorobocode.orgfree.com/torneio-ft.php>

² <http://robocode.sourceforge.net/>

³ <http://robocode.sourceforge.net/docs/ReadMe.html>

Baseei-me no artigo (HONG; CHO, 2004), publicado em um congresso de computação evolutiva da IEEE. As primeiras versões do *Genético* não travaram boas batalhas. Alterei o *fitness* e o processo de treinamento. Na versão final ele foi capaz de vencer os robôs que ficaram em primeiro e terceiro lugares no torneio daquele ano (o segundo lugar não estava disponível para *download*). O *Genético* vencia, inclusive, contra os dois simultaneamente (PRIETO, 2010).

5.2 Software

O *software* utilizado foi totalmente desenvolvido por mim, e há razões metodológicas que justificam essa escolha. Obtive bons resultados criando os próprios programas nos estudos iniciais de GA. Eles foram fundamentais para que eu soubesse exatamente o que estava acontecendo durante a execução. Quis ter esse controle total também sobre o programa que executaria o GA proposto para esta dissertação. Ele foi escrito em Linguagem C, utilizando apenas quatro bibliotecas padrão: `stdio.h`, `stdlib.h`, `time.h` e `math.h`. Portanto, é totalmente portável para os sistemas operacionais que possuem um compilador C++. Além disso, C é a linguagem nativa da arquitetura CUDA, o que facilitará sua paralelização.

Tive muito cuidado com a confiabilidade dos resultados. Quando opta-se por não gerá-la automaticamente, a *semente* dos números pseudo-aleatórios é um dos parâmetros de entrada. Duas execuções com os mesmos parâmetros, incluindo a semente, levam a exatamente os mesmos resultados. Por isso nas tabelas e gráficos deixei explícito seu valor. Os gráficos encontrados em (NANDY *et al.*, 2004) e (NANDY *et al.*, 2011) referem-se ao maior *fitness* (e ρ associado) de cada geração. O programa gera essa informação, mas também exibe as médias dessas variáveis. De acordo com (MITCHELL, 1998), uma teoria geral para entender e prever o comportamento dos GAs seria análoga à Mecânica Estatística na Física. Ao contrário de lidar com grande número de componentes do sistema, como a composição genética exata de cada população, tal abordagem trabalha com uma estatística mais “macroscópica”, como o *fitness* médio da população. Portanto, tanto os critérios de parada do programa quanto a minha análise foram baseadas em médias.

5.2.1 Como utilizar o programa

Para utilizar o *software* basta compilar o arquivo `Serial_novo.c` (será necessário alterar o diretório no *include* das bibliotecas que desenvolvi). O código está disponível na internet para qualquer um utilizar, testar e, inclusive, melhorar⁴. Caso isso aconteça, peço apenas que cite esta dissertação.

⁴ https://github.com/prietoab/msc_code

A execução dá-se via linha de comando passando os parâmetros necessários. Porém, como são muitos parâmetros, é aconselhável a utilização de um arquivo de *script*. Assim é possível criar processos de varredura para variar os parâmetros desejados ou repetir várias execuções. Fiz isso para automatizar o estudo e ter dados suficientes para análise. Na figura 22 há um exemplo de *script* Windows para fazer execuções variando o número de genes.

```

REM G: quantidade de genes
for /l %%G in (20, 10, 100) do (
    @ECHO Processando quantidade de genes = %%G
    for /l %%E in (1, 1, 5) do (
        @ECHO .... Execucao %%E de 5
        Serial_novo %Maquina% %Serial_Paralelo% %%G
    )
)

```

Figura 22 – Exemplo de *script* Windows para fazer execuções variando o número de genes.

Não é necessário ter uma matriz para execução. Por meio do parâmetro *Tamanho do cromossomo* (seção 5.2.4) o programa gera automaticamente uma matriz de Coope (COOPE; SABO, 1977), definida na equação 5.1. Essa matriz, que não possui significado físico, foi escolhida pois também é utilizada nos testes de (NANDY *et al.*, 2011).

$$H(i, i) = 2i - 1 \quad , \quad i = 1, 2, \dots, n.$$

$$\begin{aligned}
H(i, j) &= H(j, i) = 1 \quad , \quad i \neq j; \\
&\quad i = 1, 2, \dots, n; \\
&\quad j = 1, 2, \dots, n.
\end{aligned} \tag{5.1}$$

5.2.2 Informações de saída

Há cinco grupos de informações na saída do programa. Usados em conjunto dão um panorama do algoritmo genético. Para apenas um deles (Estatística) um arquivo texto é gerado automaticamente. Os outros são exibidos na tela. Para alterar a saída padrão para um arquivo texto, é necessário utilizar o caractere de redirecionamento da saída padrão⁵.

1. Cabeçalho

Contém todos os parâmetros de execução recebidos na linha de comando. Impresso na tela.

⁵ No Windows, Linux e Unix esse caractere é o “>”. No Linux, por exemplo, o comando “ls > dirs.txt” lista o conteúdo do diretório atual e armazena no arquivo dirs.txt.

2. Comportamento do *fitness*

Imprime na tela, para cada geração, além de alguns parâmetros de execução, as seguintes informações:

- ρ mínimo
- ρ médio ($\langle \rho \rangle$)
- *fitness* médio ($\langle \text{fitness} \rangle$)
- Maior *fitness*
- ρ associado ao maior *fitness*
- $\langle |\nabla \rho|^2 \rangle$
- Posição do melhor indivíduo

3. Tempos de processamento

Imprime na tela uma estimativa para o tempo de processamento (em *clocks*) para cada operador. Se o programa foi executado por 200 gerações, haverá 200 tempos de processamento para o *fitness*, seleção, *crossover* e mutação.

4. Geração final

Impressão de todos os indivíduos da última geração.

5. Estatística

A cada execução é criado (ou atualizado caso já exista) um arquivo chamado `estatistica.txt`. Nele, além de todos os parâmetros, há informações relacionadas à geração que atingiu algum critério de parada. Por exemplo, além do próprio número da última geração, há o *fitness* médio, o maior *fitness*, o ρ associado ao maior *fitness*, o ρ médio e o tempo total de processamento do programa.

5.2.3 Lista dos arquivos fonte

O programa é pequeno (≈ 2500 linhas), e está distribuído em seis arquivos, um principal e cinco bibliotecas:

- **Serial_novo.c**

Arquivo principal. Contém o fluxo do GA (figura 18).

- **Estruturas.h**

Contém as estruturas de dados e uma função que retorna automaticamente o parâmetro β do *fitness* (seção 6.3).

- **Algebra_Linear_serial.h**

Multiplicação e subtração de matrizes.

- **Estatistica.h**

Média, variância e desvios do Quociente de Rayleigh.

- **Auxiliares_serial.h**

Números pseudo-aleatórios, alocação de memória para indivíduos na população.

- **GA_Serial.h**

Contém as funções do Algoritmo Genético. A maioria do código está nessa biblioteca.

5.2.4 Lista dos parâmetros de execução

1. **Código da máquina.**

Número inteiro, utilizado para identificar o computador que o programa foi executado. Útil para comparar as execuções em computadores diferentes. Por exemplo, se há quatro computadores A, B, C e D, é possível classificá-los como A = 0, B = 1, C = 2, D = 3.

2. **Serial ou Paralelo?**

Número inteiro. Determina se a execução será serial (= 0) ou paralela (= 1). A atual versão permite apenas execução serial. Portanto, esse parâmetro pode ser fixado em zero.

3. **Tamanho do cromossomo (ordem da matriz de Coope).**

Número inteiro. Determina a ordem da matriz de Coope. Inserir 200 nesse parâmetro gera uma matriz de Coope de tamanho 200 x 200.

4. **Quantidade máxima de gerações.**

Número inteiro. Um dos critérios de parada. Para evitar que o programa entre em um *loop* infinito caso não haja convergência para uma solução. Se definida como 100, o programa executará, no máximo, 100 iterações de Avaliação, Seleção, Crossover e Mutação.

5. **Quantidade de indivíduos na população.**

Número inteiro. Determina a quantidade de indivíduos em cada população.

6. **Tipo do Fitness.**

Número inteiro. O programa pode trabalhar com cinco tipos de *fitness*. Dois deles são os apresentados na seção 4.3. Qualquer código diferente dos apresentados abaixo faz com que o *fitness* seja definido como $f = -1$.

- Tipo 0 ([NANDY et al., 2011](#)):

$$f = e^{-\beta(\rho - E_L)^2} \quad (5.2)$$

- Tipo 1 ([NANDY et al., 2004](#)):

$$f = e^{-\beta|\nabla\rho|^2} \quad (5.3)$$

- Tipo 2:

$$f = e^{-\beta[(\rho - E_L)^2 + |\nabla\rho|^2]} \quad (5.4)$$

- Tipo 3:

$$f = e^{-\beta|\nabla\rho|} \quad (5.5)$$

- Tipo 4:

$$f = e^{-\beta[(\rho - E_L)^2 + |\nabla\rho|]} \quad (5.6)$$

7. Tipo do Fitness Paralelo.

Número inteiro. A atual versão permite apenas execução serial. Portanto, esse parâmetro pode ser fixado em zero.

8. Tamanho do Torneio

Número inteiro. Define a quantidade de indivíduos selecionados para o torneio na Seleção.

9. Probabilidade do *Crossover*.

Número real. Define, em porcentagem, a probabilidade de *Crossover*. Exemplo: 80.5 = 80,5%.

10. Quantidade de Pontos de Corte.

Número inteiro. Na versão atual a quantidade de pontos de corte foi fixada em dois conforme o operador de *crossover* definido na seção 4.5.3. Pode ser mantido como zero.

11. Probabilidade de Mutação.

Número real. Define, em porcentagem, a probabilidade de mutação. Exemplo: 12.7 = 12,7%.

12. Intensidade da Mutação - Δ .

Número real. Define o parâmetro Δ utilizado no *crossover*. É dividido por dez. Exemplo: 1.2 no parâmetro \rightarrow 0, 12 no Δ .

13. Valor para β .

Número real. Define o valor do parâmetro β do *fitness*. Se for configurado como -1 , um valor adequado para β é gerado automaticamente.

14. Valor para E_L .

Número real. Define um limite inferior para o autovalor mínimo (E_0) nos *fitness* de tipo 0, 2 e 4.

15. Precisão - ξ .

Número real. Define a precisão dos critérios de parada. O programa termina se a variável alvo é menor ou igual a ξ . Depende do tipo de *fitness*.

Condições de parada em função do *fitness* ($\langle x \rangle$ significa valor médio de x):

- Fitness tipos 0, 2 e 4:

$$\begin{aligned} \langle |\nabla \rho| \rangle &\leq \xi \quad \text{ou} \\ |\langle \rho \rangle - E_L| &\leq \xi \end{aligned} \tag{5.7}$$

- Fitness tipos 1 e 3:

$$\langle |\nabla \rho| \rangle \leq \xi \tag{5.8}$$

16. Imprime comportamento do *Fitness*?

Se configurado como Verdadeiro (1), imprime na saída padrão as variáveis de interesse para estudar o comportamento do *fitness*.

0: *Falso*. Não imprime.

1: *Verdadeiro*. Imprime.

17. Imprime tempos de execução?

Número inteiro. Se configurado como Verdadeiro (1), imprime na saída padrão os tempos estimados de execução (em *clocks* do processador) das funções e operadores.

0: *Falso*. Não imprime.

1: *Verdadeiro*. Imprime.

18. Gera nova semente para números pseudo-aleatórios?

Número inteiro. Se configurado como Verdadeiro (1), o programa cria uma nova semente para os números pseudo-aleatórios. Caso contrário, a semente definida no próximo parâmetro é utilizada.

0: *Falso*. Utiliza semente definida no parâmetro *Semente*.

1: *Verdadeiro*. Cria uma nova semente. O parâmetro *Semente* é ignorado.

19. Semente dos números pseudo-aleatórios.

Número inteiro. Define a semente dos números pseudo-aleatórios. Depende do parâmetro anterior.

20. Tipo do cálculo de $\nabla\rho$.

Deve ser configurado como 1.

Nas versões iniciais $\nabla\rho$ era calculado literalmente como na equação 4.7, exigindo o uso de uma matriz identidade (I) da ordem do Hamiltoniano. A equação foi reescrita internamente de modo que I não fosse necessária, liberando memória e fazendo menos operações. Usar ou não I leva aos mesmos resultados.

0: Utiliza matriz identidade.

1: Não utiliza matriz identidade (libera memória e faz menos operações).

5.2.5 Estruturas de Dados

Cinco estruturas de dados formam a base do programa. Elas estão definidas no arquivo Estruturas.h.

5.2.5.1 parametrosPrograma

Responsável por armazenar os parâmetros de execução, descritos na seção 5.2.4.

```

1 struct parametrosPrograma {
2     unsigned short int parmMaquina;
3     unsigned short int parmSerial_ou_Paralelo;
4     unsigned short int parmQtdeGenes;
5     unsigned long int parmQtdeMaxGeracoes;
6     unsigned short int parmQtdeIndividuos;
7     char parmTipoFitnessEquacao;
8     char parmTipoFitnessParalelo;
9     char parmTipoCalculoGradRho;
```

```

10     char parmTamanhoTorneio;
11     double parmProbCrossOver;
12     unsigned short int parmQtdePontosCorte;
13     double parmProbMutacao;
14     double parmIntensidadeMutacao;
15     double parmLambda;
16     double parmRhoMinimo;
17     double parmTolerancia;
18     unsigned short int parmFlagImprimeComportamentoFitness;
19     unsigned short int parmFlagImprimeTempo;
20     unsigned short int parmFlagNovaSemente;
21     unsigned long int parmSemente;
22 };

```

5.2.5.2 parametros_Metodo

Armazena os parâmetros do *fitness*, β e E_0 .

```

1 struct parametros_Metodo {
2     double lambda;
3     double rho_minimo;
4 };

```

5.2.5.3 parametros

Parâmetros do algoritmo genético.

```

1 struct parametros {
2     unsigned short int numGenes;
3     unsigned short int numIndividuos;
4     double probCrossOver;
5     double probMutacao;
6     double intensidadeMutacao;
7     char tamanho_torneio;
8     unsigned short int qtde_Pontos_de_Corte;
9 };

```

5.2.5.4 individual

Essa estrutura representa o cromossomo, o indivíduo, do algoritmo genético.

cociente_Rayleigh: ρ_i .

grad_elevado_ao_quadrado: $|\nabla \rho_i|^2$.

rho_menos_rho0_ao_quadrado: $(\rho_i - E_0)^2$.

Numerador: Recebe o retorno do cálculo $C_i^\dagger H C_i$, utilizado na equação 4.6, seção 4.3.

CtC: Recebe o produto escalar de $C_i^\dagger C_i$, utilizado no cálculo da equação 4.6, seção 4.3.

inverso_de_CtC: $(C_i^\dagger C_i)^{-1}$.

fitness: recebe o valor do *fitness* do indivíduo, independente do tipo.

pontos_de_corte(2): Vetor de tamanho fixo, cuja função é armazenar as duas posições que determinarão a posição inicial e final do *crossover*.

***gene**: Ponteiro de memória para um vetor que armazenará os valores dos genes do cromossomo. O tamanho desse vetor é dado pelo parâmetro **numGenes** da estrutura **parametros**.

***gradRho**: Ponteiro de memória para um vetor que armazenará os valores dos componentes do vetor $\nabla \rho_i$. O tamanho desse vetor é dado pelo parâmetro **numGenes** da estrutura **parametros**.

```

1 struct individual {
2     double cociente_Rayleigh;
3     double grad_elevado_ao_quadrado;
4     double rho_menos_rho0_ao_quadrado;
5     double Numerador;
6     double CtC;
7     double inverso_de_CtC;
8     double fitness;
9     unsigned short int pontos_de_corte[2];
10    double *gene;
11    double *gradRho;
12 };

```

5.2.5.5 generation

A estrutura **generation** representa a população dos indivíduos em um dado instante de tempo. O programa possui apenas duas, utilizadas alternadamente a cada aplicação de um operador genético. Por exemplo, Seleção(geracao0) → geracao1, e em seguida Crossover(geracao1) → geracao0.

sumFitness: Soma do *fitness* de todos os indivíduos da população. Utilizada para obter o *fitness* médio $\langle f \rangle$.

FitnessMedio: $\langle f \rangle$.

sumRho: Soma do ρ de todos os indivíduos da população. Utilizada para obter o quociente de Rayleigh médio $\langle \rho \rangle$.

rhoMedio: $\langle \rho \rangle$.

difRho: $\langle \rho \rangle - E_L$

sumGradRho: Soma de todos os $|\nabla \rho_i|$. Utilizada para obter $\langle |\nabla \rho| \rangle$.

gradRhoMedio: $\langle |\nabla \rho| \rangle$

Maior_fitness: Maior *fitness* da população.

Melhor_cociente_Rayleigh: ρ associado ao maior *fitness* da população.

idxMelhorIndividuo: Posição do melhor indivíduo no vetor de indivíduos. Por exemplo, se há 10 cromossomos na população, seu valor pode ser 0, 1, 2, ..., 9.

***individuo**: Ponteiro de memória para o vetor que contém os indivíduos da população.

```

1 struct generation {
2     double sumFitness;
3     double FitnessMedio;
4     double rhoMedio;
5     double sumRho;
6     double difRho;
7     double sumGradRho;
8     double gradRhoMedio;
9     double Maior_fitness;
10    double Melhor_cociente_Rayleigh;
11    unsigned short int idxMelhorIndividuo;
12    struct individual *individuo; // alocar constNumIndividuos;
13 };

```

5.2.6 População inicial

Está no arquivo **GA_serial.h**.

O objetivo da **GeraPopulacaoInicial_serial** é criar os valores aleatórios para os genes de todos os indivíduos. As outras variáveis da geração e dos indivíduos são apenas inicializadas. Ela como parâmetros dois ponteiros (linhas 3 e 4), um para a população e outro para os parâmetros do GA.

O *loop* principal varre todos os indivíduos da população, e o laço interno caminha pelos genes de cada indivíduo (linha 35). Um inteiro L, com valor 0 ou 1, é escolhido aleatoriamente, e utilizado para definir o sinal positivo (+1) ou negativo (-1). Na linha 38 outro número aletório é obtido, mas agora é real e entre 0 e 1. Ele é multiplicado pelo sinal e armazenado no gene **iGene** do indivíduo **iIndividuo**.

Portanto, os genes dos indivíduos estão limitados no intervalo [-1,1].

```

1 //-----
2 void GeraPopulacaoInicial_serial(
3     struct generation *GeracaoInicial ,
4     struct parametros *parametrosGA ) {
5 //-----
6
7     unsigned short int iIndividuo , qtdePontosCorte , iPontoCorte;
8     unsigned long int iGene;
9
10    GeracaoInicial->FitnessMedio = -1.0F;
11    GeracaoInicial->idxMelhorIndividuo = 99;
12    GeracaoInicial->Maior_fitness = -1.0F;
13    GeracaoInicial->Melhor_cociente_Rayleigh = -1.0F;
14    GeracaoInicial->sumFitness = -1.0F;
15
16    unsigned short int L;
17    short int sinal;
18
19    for (iIndividuo = 0; iIndividuo < parametrosGA->numIndividuos ;
20        iIndividuo++) {
21
22        GeracaoInicial->individuo [iIndividuo].cociente_Rayleigh = -1.0F
23                ;
24        GeracaoInicial->individuo [iIndividuo].CtC = -1.0F;
25        GeracaoInicial->individuo [iIndividuo].fitness = -1.0F;
26        GeracaoInicial->individuo [iIndividuo].grad_elevado_ao_quadrado

```

```

    = -1.0F;
25 GeracaoInicial->individuo[iIndividuo].inverso_de_CtC = -1.0F;
26 GeracaoInicial->individuo[iIndividuo].Numerador = -1.0F;
27 GeracaoInicial->individuo[iIndividuo].
    rho_menos_rho0_ao_quadrado = -1.0F;
28
29 qtdePontosCorte = parametrosGA->qtde_Pontos_de_Corte;
30
31 for (iPontoCorte = 0; iPontoCorte < qtdePontosCorte;
      iPontoCorte++) {
32     GeracaoInicial->individuo[iIndividuo].pontos_de_corte[
      iPontoCorte] = 11111;
33 }
34
35 for (iGene = 0; iGene < parametrosGA->numGenes; iGene++) {
36     L = Randomico_int(0, 2);
37     sinal = (short int)pow(-1.0, (double)L);
38     GeracaoInicial->individuo[iIndividuo].gene[iGene] = (double)
      sinal * rand() / RAND_MAX;
39 }
40 }
41 }

```

5.2.7 Fitness

Está no arquivo **GA_serial.h**.

Na função **Fitness_Serial** são calculados:

- ρ_i

Usa a função **Calcula_Quociente_Rayleigh**, linha 30, que contém, basicamente, multiplicações de matrizes.

- $\nabla \rho_i$.

Dependendo do tipo do cálculo escolhido, usa **Gradiente_de_Rho_semI** na linha 40 ou **Gradiente_de_Rho** na linha 49.

- $|\nabla \rho_i|^2$.

Multiplicação de matrizes na linha 59–63.

- $(\rho_i - E_L)^2$

Linha 67.

- *fitness*

Dependendo do tipo escolhido.

- Tipo 0, $f = e^{-\beta(\rho-E_L)^2}$: linha 71
- Tipo 1, $f = e^{-\beta|\nabla\rho|^2}$: linha 75
- Tipo 2, $f = e^{-\beta[(\rho-E_L)^2+|\nabla\rho|^2]}$: linha 79
- Tipo 3, $f = e^{-\beta|\nabla\rho|}$: linha 83
- Tipo 4, $f = e^{-\beta[(\rho-E_L)^2+|\nabla\rho|]}$: linha 87
- Se nenhum dos tipos acima foi escolhido, define o *fitness* como -1 na linha 91.

- $\langle f \rangle$

Na linha 96 está a soma de todos os *fitness*, e na 105 o cálculo da média sobre os indivíduos.

- $\langle \rho \rangle$

Soma dos ρ : linha 36. Média: linha 107.

- $\langle \rho \rangle - E_L$

Linha 108.

- **Atributos do melhor indivíduo da geração.**

- Identificação do melhor indivíduo da geração: linha 98.
- Maior *fitness*: linha 99.
- Melhor ρ : linha 100.
- Posição na geração: linha 101.

```

1 //=====
2 void Fitness_Serial(
3     unsigned short int tipoFitness ,
4     char TipoCalculoGradRho ,
5     struct generation *geracao ,
6     struct parametros *parametrosGA ,
7     struct parametros_Metodo *parms_Metodo ,
8     double *hamiltoniano ,

```

```
9  const unsigned short int ordem_hamiltoniano ,
10 double lambda ,
11 double rho_minimo ,
12 double *matriz_Identidade) {
13 //=====
14
15 unsigned short int      iIndividuo;
16
17 geracao->rhoMedio = -1.0F;
18 geracao->sumRho = 0.0F;
19 geracao->FitnessMedio = -1.0F;
20 geracao->Maior_fitness = -1.0F;
21 geracao->sumFitness = 0.0F;
22 geracao->Maior_fitness = -1.0F;
23 geracao->Melhor_cociente_Rayleigh = -1.0F;
24
25 geracao->sumGradRho = 0.0F;
26 geracao->gradRhoMedio = -1.0F;
27
28 for (iIndividuo = 0; iIndividuo < parametrosGA->numIndividuos ;
29     iIndividuo++) {
30
31     Calcula_Quociente_Rayleigh(
32         geracao->individuo[iIndividuo].gene ,
33         hamiltoniano ,
34         ordem_hamiltoniano ,
35         &geracao->individuo[iIndividuo].cociente_Rayleigh);
36
37     geracao->sumRho = geracao->sumRho + geracao->individuo [
38         iIndividuo].cociente_Rayleigh;
39
40     if (TipoCalculoGradRho == 1){
41
42         Gradiente_de_Rho_semI(
43             hamiltoniano ,
44             ordem_hamiltoniano ,
45             geracao->individuo[iIndividuo].gene ,
46             geracao->individuo[iIndividuo].cociente_Rayleigh ,
47             geracao->individuo[iIndividuo].gradRho
48         );
49     }
50     else {
51
52         Gradiente_de_Rho_comI(
53             geracao->individuo[iIndividuo].gene ,
54             geracao->individuo[iIndividuo].cociente_Rayleigh ,
55             geracao->individuo[iIndividuo].gradRho
56         );
57     }
58 }
```

```
49     Gradiente_de_Rho(
50         hamiltoniano,
51         ordem_hamiltoniano,
52         geracao->individuo[iIndividuo].gene,
53         geracao->individuo[iIndividuo].cociente_Rayleigh,
54         geracao->individuo[iIndividuo].gradRho,
55         matriz_Identidade
56     );
57 }
58
59 Multiplica_Matrices(
60     geracao->individuo[iIndividuo].gradRho, 1, ordem_hamiltoniano
61     ,
62     geracao->individuo[iIndividuo].gradRho, ordem_hamiltoniano,
63     1,
64     &geracao->individuo[iIndividuo].grad_elevado_ao_quadrado
65 );
66
67 geracao->sumGradRho = geracao->sumGradRho + sqrt(geracao->
68     individuo[iIndividuo].grad_elevado_ao_quadrado);
69
70 geracao->individuo[iIndividuo].rho_menos_rho0_ao_quadrado = pow
71     ( (geracao->individuo[iIndividuo].cociente_Rayleigh -
72     rho_minimo) ,2);
73
74 switch (tipoFitness) {
75     case 0: {
76         geracao->individuo[iIndividuo].fitness = exp( (-1)*lambda*
77             geracao->individuo[iIndividuo].rho_menos_rho0_ao_quadrado
78             ));
79     }
80     break;
81     case 1: {
82         geracao->individuo[iIndividuo].fitness = exp( (-1)*lambda*
83             geracao->individuo[iIndividuo].grad_elevado_ao_quadrado))
84             ;
85     }
86     break;
87     case 2: {
88         geracao->individuo[iIndividuo].fitness = exp( (-1)*lambda*
89             geracao->individuo[iIndividuo].rho_menos_rho0_ao_quadrado
90             + geracao->individuo[iIndividuo].
```

```
        grad_elevado_ao_quadrado));
80    }
81    break;
82    case 3: {
83        geracao->individuo[iIndividuo].fitness = exp( (-1)*lambda*
84            sqrt(geracao->individuo[iIndividuo].
85            grad_elevado_ao_quadrado) ) );
86    }
87    break;
88    case 4: {
89        geracao->individuo[iIndividuo].fitness = exp( (-1)*lambda*
90            geracao->individuo[iIndividuo].rho_menos_rho0_ao_quadrado
91            + sqrt(geracao->individuo[iIndividuo].
92            grad_elevado_ao_quadrado)));
93    }
94    break;
95    default: {
96        geracao->individuo[iIndividuo].fitness = -1;
97    }
98    break;
99 }
100}
101geracao->sumFitness = geracao->sumFitness + geracao->individuo[
102    iIndividuo].fitness;
103if (geracao->individuo[iIndividuo].fitness > geracao->
104    Maior_fitness) {
105    geracao->Maior_fitness = geracao->individuo[iIndividuo].
106    fitness;
107    geracao->Melhor_cociente_Rayleigh = geracao->individuo[
108        iIndividuo].cociente_Rayleigh;
109    geracao->idxMelhorIndividuo = iIndividuo;
110}
111}
112
113geracao->FitnessMedio = geracao->sumFitness / parametrosGA->
114    numIndividuos;
115geracao->gradRhoMedio = geracao->sumGradRho / parametrosGA->
116    numIndividuos;
117geracao->rhoMedio = geracao->sumRho / parametrosGA->numIndividuos
118    ;
119geracao->difRho = abs(geracao->rhoMedio - parms_Metodo->
```

```

    rho_minimo);

109 }
```

5.2.8 Seleção

Está no arquivo **GA_serial.h**.

O laço externo, linha 14, garante a seleção de `parametros.numIndividuos` para serem armazenados na `PopulacaoDepois`. O laço interno, na linha 18, busca aleatoriamente os indivíduos, um a um (linhas 20 e 21), para participarem do torneio. Se o *fitness* do indivíduo atual for maior do que o anterior (linha 23), ele é levado para a população seguinte (linha 25).

```

1 //=====
2 void Selecao_Por_Torneio_serial(
3     struct generation *PopulacaoAntes,
4     struct generation *PopulacaoDepois,
5     struct parametros *parametrosGA) {
6 //=====

7
8     unsigned short int iIndividuo;
9     char iTamanhoDoTorneio;
10    unsigned short int iIndividuo_para_Torneio;
11    struct individual *Individuo;
12    double melhorFitness;
13
14    for (iIndividuo = 0; iIndividuo < parametrosGA->numIndividuos;
15        iIndividuo++) {
16
17        melhorFitness = -1.0F;
18
19        for (iTamanhoDoTorneio = 0; iTamanhoDoTorneio < parametrosGA->
20            tamanho_torneio; iTamanhoDoTorneio++) {
21
22            iIndividuo_para_Torneio = Randomico_int(0, parametrosGA->
23                numIndividuos);
24            Individuo = &PopulacaoAntes->individuo[
25                iIndividuo_para_Torneio];
```

```

23     if (Individuo->fitness > melhorFitness) {
24         melhorFitness = Individuo->fitness;
25         PopulacaoDepois->individuo[iIndividuo] = *Individuo;
26     };
27 };
28 };
29 };

```

5.2.9 Crossover

Está no arquivo **GA_serial.h**.

A cada iteração do laço principal (linha 14) é criado um filho. São escolhidos aleatoriamente:

- Pai e mãe: linhas 16–20.
- Probabilidade auxiliar que será comparada com a probabilidade de *crossover*: linha 22.
- Dois pontos de corte do cromossomo: linhas 26 e 27.
- Parâmetro f da equação 4.31: linha 31.

A condição para ocorrência do *crossover*, $p_{aux} < \text{parametros.probCrossOver}$, está na linha 29. Se ela for falsa, o pai é transferido sem nenhuma alteração para a próxima geração (linhas 48–50). Se for verdadeira, entre os pontos de corte há combinação entre os genes do pai e da mãe. Fora dos pontos de corte os genes são iguais ao do pai (linhas 33–43). Veja diagrama na figura 23.

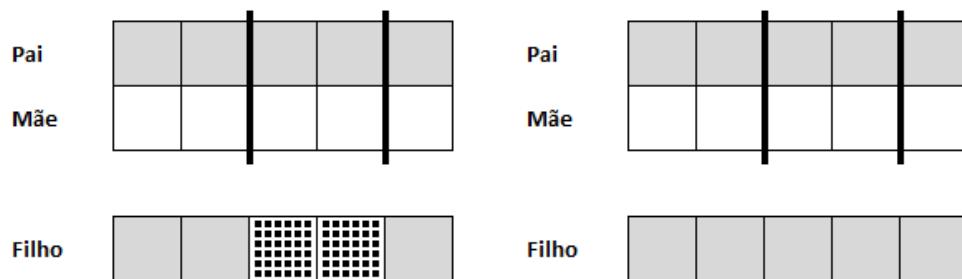


Figura 23 – Diagrama do *crossover* implementado. Na esquerda: *crossover* acontece; na direita: *crossover* acontece.

```
1 //=====
2 void CrossOver2Pontos_serial(
3     struct generation *g0,
4     struct generation *g1,
5     struct parametros *parametrosGA) {
6 //=====
7
8     unsigned short int iIndividuo, idx_Pai, idx_Mae;
9     unsigned long int iGene;
10
11    struct individual *Pai, *Mae;
12    double f, p_aux;
13
14    for (iIndividuo = 0; iIndividuo < parametrosGA->numIndividuos;
15        iIndividuo = iIndividuo + 1) {
16
17        idx_Pai = Randomico_int(0, parametrosGA->numIndividuos - 1);
18        idx_Mae = Randomico_int(0, parametrosGA->numIndividuos - 1);
19
20        Pai = &g0->individuo[idx_Pai];
21        Mae = &g0->individuo[idx_Mae];
22
23        p_aux = (double)((double)rand() / (double)RAND_MAX);
24
25        f = -1.0F;
26
27        g1->individuo[iIndividuo].pontos_de_corte[0] = Randomico_int(0,
28                           (parametrosGA->numGenes-1));
29        g1->individuo[iIndividuo].pontos_de_corte[1] = Randomico_int(g1
30                           ->individuo[iIndividuo].pontos_de_corte[0], (parametrosGA->
31                           numGenes-1));
32
33        if (p_aux <= parametrosGA->probCrossOver) {
34
35            f = (double)((double)rand()/(double)RAND_MAX);
36
37            for (iGene = 0; iGene < g1->individuo[iIndividuo].
38                  pontos_de_corte[0]; iGene++) {
39                g1->individuo[iIndividuo].gene[iGene] = Pai->gene[iGene];
40            }
41
42            for (iGene = g1->individuo[iIndividuo].pontos_de_corte[0];
```

```

    iGene < g1->individuo[iIndividuo].pontos_de_corte[1]; iGene
    ++) {
38     g1->individuo[iIndividuo].gene[iGene] = f*Pai->gene[iGene]
        + (1 - f)*Mae->gene[iGene];
39 }
40
41     for (iGene = g1->individuo[iIndividuo].pontos_de_corte[1];
        iGene < parametrosGA->numGenes; iGene++) {
42         g1->individuo[iIndividuo].gene[iGene] = Pai->gene[iGene];
43     }
44 }
45
46 else {
47
48     for (iGene = 0; iGene < parametrosGA->numGenes; iGene++) {
49         g1->individuo[iIndividuo].gene[iGene] = g0->individuo[
            iIndividuo].gene[iGene];
50     }
51 }
52 }
53 }
```

5.2.10 Mutação

Está no arquivo **GA_serial.h**.

Todos os genes de cada indivíduo estão sujeitos à mutação (laços das linhas 14 e 16). A condição de ocorrência é semelhante à do *crossover*. Se uma probabilidade auxiliar, obtida aleatoriamente na linha 18, for menor que a probabilidade de mutação, o gene é alterado na linhas linhas 25–27 conforme equação 4.33. Os parâmetros necessários são obtidos nas linhas 21–23.

```

1 //=====
2 void Mutacao_Serial(
3     struct generation *geracao,
4     struct parametros *parametrosGA) {
5 //=====
6
7 unsigned short int iIndividuo;
```

```

8  unsigned long iGene;
9  unsigned short int L;
10 int termo_do_L;
11 double r, pAux;
12 double intensidadeMutacao = 10*geracao->Maior_fitness;
13
14 for (iIndividuo = 0; iIndividuo < parametrosGA->numIndividuos;
15     iIndividuo++) {
16
17     for (iGene = 0; iGene < parametrosGA->numGenes; iGene++) {
18
19         pAux = (double)((double)rand() / (double)RAND_MAX);
20
21         if (pAux <= parametrosGA->probMutacao) {
22             L = Randomico_int(0, 10);
23             r = (double)((double)rand() / (double)RAND_MAX) + 0.000002;
24             termo_do_L = (int)pow((double)(-1),(int)L);
25
26             geracao->individuo[iIndividuo].gene[iGene] =
27                 geracao->individuo[iIndividuo].gene[iGene] +
28                 (double)(termo_do_L * r * intensidadeMutacao);
29         }
30     }
31 }
```

5.2.11 Fluxo principal

Abaixo listo o código, presente no arquivo **Serial_novo.c**, referente ao fluxo principal do GA (figura 18). Os operadores são aplicados na ordem: *Fitness* (linha 7), Seleção (linha 44), *Crossover* (linha 55) e Mutação (linha 66).

O programa termina quando atinge o número máximo de gerações, testado na condição do `while` da linha 2, ou quando algum dos critérios de parada de precisão é atingido (linhas 37–40), definidos pelas equações 5.7 e 5.8. Note que elas são testados logo após o *fitness*, afinal, se uma geração contém boas soluções, não deve ser alterada pelos outros operadores genéticos.

```
2 while (iGeracao < parmsPrograma.parmQtdeMaxGeracoes) {
3
4 // FITNESS
5 time_i = clock();
6
7 Fitness_Serial(
8     parmTipoFitnessEquacao,
9     TipoCalculoGradRho,
10    geracao0,
11    &parametrosGA,
12    &parametrosMetodo,
13    Hamiltoniano,
14    parametrosGA.numGenes,
15    parametrosMetodo.lambda,
16    parametrosMetodo.rho_minimo,
17    MatrizIdentidade
18 );
19
20 time_f = clock();
21
22 if (flagImprimeTempo == 1) {
23     imprimeTempo(
24         1, 0, 0, iGeracao, 4, 2, time_i, time_f,
25         &parametrosGA, &parametrosMetodo, &parmsPrograma
26     );
27 }
28
29 if (flagImprimeComportamentoFitness == 1) {
30     imprimeComportamentoFitness(
31         1, codMaquina, tipoPrograma, 0, iGeracao, geracao0,
32         &parametrosMetodo, &parametrosGA, &parmsPrograma
33     );
34 }
35
36 // CRITERIOS DE PARADA
37 flagAtingiuTolerancia = atingiuCriterioDeParada(
38     parmTipoFitnessEquacao, tolerancia, geracao0);
39
40 if (flagAtingiuTolerancia == 1)
41     break;
42 // SELECAO
```

```
43     time_i = clock();
44     Selecao_Por_Torneio_serial(geracao0, geracao1, &parametrosGA);
45     time_f = clock();
46     if (flagImprimeTempo == 1) {
47         imprimeTempo(
48             1, 0, 0, iGeracao, 5, 2, time_i, time_f,
49             &parametrosGA, &parametrosMetodo, &parmsPrograma
50         );
51     }
52
53 // Crossover
54 time_i = clock();
55 CrossOver2Pontos_serial(geracao1, geracao0, &parametrosGA);
56 time_f = clock();
57 if (flagImprimeTempo == 1) {
58     imprimeTempo(
59         1, 0, 0, iGeracao, 6, 2, time_i, time_f,
60         &parametrosGA, &parametrosMetodo, &parmsPrograma
61     );
62 }
63
64 // Mutacao
65 time_i = clock();
66 Mutacao_Serial(geracao0, &parametrosGA);
67 time_f = clock();
68 if (flagImprimeTempo == 1) {
69     imprimeTempo(
70         1, 0, 0, iGeracao, 7, 2, time_i, time_f,
71         &parametrosGA, &parametrosMetodo, &parmsPrograma
72     );
73 }
74
75 iGeracao++;
76 }
```

6 Resultados e discussão

6.1 Problemas com o mínimo global

No capítulo 4 mostrei que o *fitness* utilizado no artigo ([NANDY et al., 2004](#)) foi

$$f_i = e^{-\beta|\nabla\rho_i|^2}, \quad (6.1)$$

onde f_i é o *fitness* do i -ésimo indivíduo da população, β é um parâmetro para evitar o estouro do *fitness* e $|\nabla\rho_i|^2$ é o módulo ao quadrado do vetor gradiente de ρ , dado por

$$\nabla\rho_i = \frac{2[H - \rho_i]C_i}{C_i^t C_i}, \quad (6.2)$$

em que C_i é um vetor candidato à solução do problema do autovalor

$$HC = EC. \quad (6.3)$$

Além disso, se C_i é de fato um dos autovetores, ρ é o autovalor associado E_i :

$$\rho_i = \frac{C_i^t HC_i}{C_i^t C_i} = E_i. \quad (6.4)$$

A fim de reproduzir os resultados, testei o método com matrizes de Coope–Sabo (equação 5.1) de ordem 10, 20, 30 e 40, utilizando os mesmos parâmetros encontrados em ([NANDY et al., 2004](#)): probabilidade de crossover $p_c = 75\%$, probabilidade de mutação $p_m = 50\%$ e intensidade de mutação $\Delta = 0,01$. Com um bom ajuste de β , que será discutido em detalhes posteriormente, o *fitness* comportou-se conforme o esperado em todos os casos. Um exemplo está na figura 24, que apresenta o melhor *fitness* de cada geração para uma matriz de ordem $N = 10$. Na primeira geração o melhor *fitness* é pequeno, aproximadamente 0,1, cresce rapidamente e a partir da décima geração está próximo de 1.

O passo seguinte foi verificar o comportamento de ρ , o Quociente de Rayleigh, e, especificamente, sua convergência para o menor autovalor E_0 . Ainda conforme ([NANDY et al., 2004](#)), obteríamos uma curva semelhante à da figura 24, mas invertida, ou seja, os primeiros valores de ρ seriam grandes e, rapidamente, diminuiriam até haver convergência para o autovalor mínimo. Na figura 25 há um exemplo. Os gráficos exibem os valores de ρ para a mesma execução apresentada na figura 24. Note no primeiro gráfico que até a geração 20 o quociente ρ teve caráter oscilatório e, então, aparentemente estabilizou-se

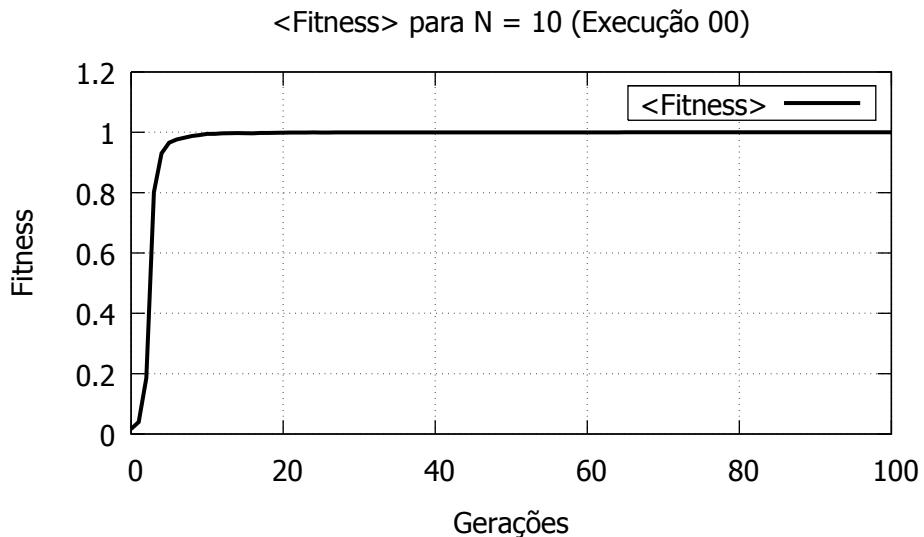


Figura 24 – Comportamento do fitness $f_i = e^{-\beta|\nabla\rho_i|^2}$ para $N = 10$. Na primeira geração o melhor fitness é pequeno, aproximadamente 0,1, cresce rapidamente e a partir da décima geração está próximo de 1.

entre 6 e 8, valores muito superiores ao autovalor mínimo para essa matriz, $E_0 = 0,38675$. Entretanto, ainda no primeiro gráfico, observa-se que há uma tendência de queda do ρ entre as gerações 40 e 50 e, portanto, existiria a possibilidade de o algoritmo convergir para E_0 . Porém, para esse exemplo especificamente, isso não aconteceu, como pode ser visto no segundo gráfico da figura 25. Para garantir a estabilidade, o programa foi executado até a geração 400.000, e o valor médio obtido foi $\langle \rho \rangle = 6,572898$. Para minha surpresa, além do valor obtido de $\langle \rho \rangle$ não ser o mínimo, ele não é um valor qualquer, mas corresponde, com erro menor que 0,00002%, ao quarto autovalor da matriz, $E_3 = 6,572897$. Imaginei, então, que poderia haver algo de errado com o programa.

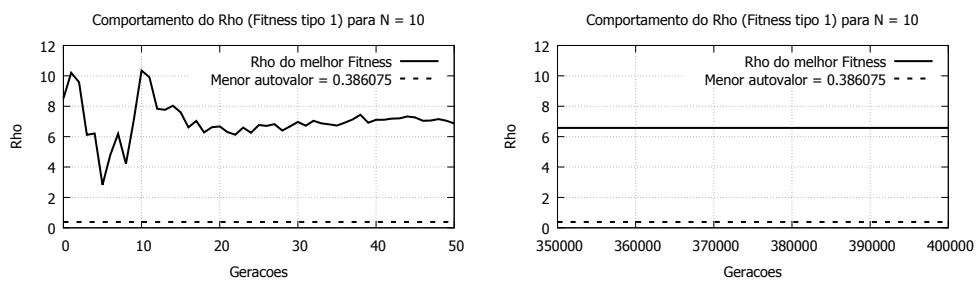


Figura 25 – Comportamento de ρ (Quociente de Rayleigh) para uma matriz de Coope–Sabo de ordem 10.

Após esses resultados preliminares executei uma validação cuidadosa do programa, testando cada uma de suas quase 2500 linhas e comparando os resultados das operações e cálculos com o Microsoft Excel e SciLab. A hipótese era a de que erros numéricos, principalmente nas funções de álgebra linear e nos operadores genéticos, pudessem ter levado ao comportamento incorreto da não convergência para o menor autovalor. Nenhum erro significativo foi encontrado.

Os testes com a versão corrigida do programa estão nas figuras 26, 27, 28 e 29. Visando brevidade, apresentarei dados para matrizes de Coope–Sabo de ordem 10, 20, 30 e 40 apenas, sem perda de generalidade. Foram cinco execuções para cada matriz, até a geração 400.000, gerando sempre dois gráficos, um do *fitness* médio ($\langle \text{fitness} \rangle$) e outro do Quociente de Rayleigh médio ($\langle \rho \rangle$), ambos em função do número de gerações, e dando ênfase às primeiras 100 gerações. Essas escolhas, número máximo da geração e uso de médias sobre cada população, visaram garantir, respectivamente, a convergência genética e boa precisão. A exibição de apenas as primeiras 100 gerações tem como objetivo olhar em detalhe (com *zoom*) o período em que o *crossover* tem mais peso, ou seja, onde há geralmente os saltos no espaço de soluções de um Algoritmo Genético. Em todos os gráficos de $\langle \rho \rangle$ há indicado nas legendas o autovalor mínimo E_0 e o autovalor obtido após as 400.000 gerações (E_{obtido}). Na tabela 10 há a lista de todos os autovalores. Por exemplo, para uma matriz de ordem $N = 10$, o menor autovalor é $E_0 = 0,386075$, e o quinto autovalor para $N = 30$ é $E_4 = 8,450274$.

Há características comuns encontradas em todas as execuções. Em qualquer gráfico do *fitness* observa-se estabilidade do comportamento conforme esperado pelo método: no início seu valor é baixo, próximo de zero, cresce rapidamente nas primeiras gerações e fica estável próximo de $\langle \text{fitness} \rangle = 1$. Com relação ao ρ , há sempre oscilações, sejam pequenas variações em torno de uma clara linha de tendência, como na execução 02 para $N = 10$, ou grandes saltos, como nas execuções 05 de $N = 20$ e 05 de $N = 30$. Novamente, o menor autovalor não foi obtido em nenhuma execução, contradizendo os resultados de (NANDY *et al.*, 2004), mas, por outro lado, o algoritmo sempre encontrou algum autovalor.

De fato, verificando os dados da tabela 11, concluí que tais valores não devem ser coincidência. Para todas as execuções o *fitness* médio chegou ao valor máximo ($\langle f \rangle = 1,000000$). As médias de ρ sobre todos os indivíduos da última população possuem baixo desvio padrão ($\sigma < 0,0001$), indicando que eles são muito parecidos entre si e que o algoritmo convergiu. Ou seja, não há variabilidade genética suficiente na população para alterar o rumo da busca de modo a atingir o menor autovalor, ou o mínimo global. Portanto, o algoritmo chegou em um mínimo local, corroborado pelos baixos erros relativos de $\langle \rho \rangle$ quando comparado com o autovalor mais próximo. Por exemplo, para $N = 30$, execução 4, $\langle \rho \rangle = 40,772447$, correspondendo, com erro relativo absoluto

menor que 0,001%, ao vigésimo primeiro autovalor, $E_{20} = 40,772850$. Apesar das evidências descritas acima, até esse ponto ainda havia dúvidas sobre a validade do programa e, obviamente, dos resultados produzidos. Então, busquei embasamento mais rigoroso.

De acordo com (NANDY *et al.*, 2004), se algum C_i , em algum momento, é o autovetor fundamental (associado ao menor autovalor), o $\nabla\rho$ é nulo. Com o *fitness* da equação (6.1) os autores afirmam que “*Claramente, $f_i \rightarrow 1$ quando $\nabla\rho_i \rightarrow 0$, sinalizando que a evolução atingiu o verdadeiro autovetor fundamental de H em C_i* ”¹. Há duas relações distintas de causalidade nessa frase, e acredito que nelas residam a explicação dos resultados obtidos até agora.

A primeira relação de causalidade refere-se à afirmação “ $f_i \rightarrow 1$ quando $\nabla\rho_i \rightarrow \mathbf{0}$ ”, que está absolutamente correta. Retomando a seção 4.3, o *fitness* definido pela equação 6.1 é limitado ao intervalo $(0,1]$ e, como $\beta > 0$, só chega ao seu valor máximo quando $\nabla\rho_i = \mathbf{0}$. Em outras palavras, $\nabla\rho_i \rightarrow \mathbf{0}$ implica $f_i \rightarrow 1$.

Na afirmação “(...) sinalizando que a evolução atingiu o verdadeiro autovetor fundamental de H em C_i ” reside a segunda relação de causalidade:

$$\text{Se } f_i \rightarrow 1, C_i = C_0. \quad (6.5)$$

Ou seja, sempre que algum indivíduo C_i , de qualquer população, possuir *fitness* muito próximo de 1, isso implica que, além de ter uma excelente “nota”, ele também é um vetor especial, o autovetor fundamental C_0 . Portanto, possui autovalor associado E_0 , o autovalor mínimo (conforme equação 6.4). Grossso modo, $f_i(C_i) = 1$ implica que $C_i = C_0$ e que podemos obter $E_0(C_0)$:

$$f_i(C_i) = 1 \rightarrow C_i = C_0 \rightarrow E_0(C_0). \quad (6.6)$$

As relações de causa e efeito da equação acima estão erradas. Em sua obra clássica sobre o problema de autovalores em matrizes simétricas, (PARLETT, 1998) abre o capítulo introdutório frisando que “*em muitos lugares no livro, é feita referência a fatos mais ou menos bem conhecidos sobre a teoria de matrizes*”. Conforme já dito no capítulo 2, um desses fatos diz que $\rho(u)$ é estacionário, ou seja, $\nabla\rho(u) = \mathbf{0}$, apenas se o vetor u é um autovetor w de $HC = EC$. Consequentemente, o encadeamento correto se apresenta como:

$$C_i \text{ é um autovetor} \rightarrow \nabla\rho(C_i) = \mathbf{0} \rightarrow f_i = 1. \quad (6.7)$$

¹ Tradução livre de “*Clearly, $f_i \rightarrow 1$, as $\nabla\rho_i \rightarrow 0$, signalling that the evolution has hit the true ground state eigenvector of H in the vector C_i .*”

Então, se $f_i = 1$, o máximo que podemos concluir é que C_i é *algum* autovetor, e não necessariamente *o* autovetor fundamental.

Acredito que o programa não contém erros. Ao final de todos os testes o *fitness* médio foi $\langle f \rangle = 1$, a população final era composta por autovetores e foi possível, com boa precisão, obter os autovalores relacionados (não necessariamente o autovalor mínimo). Os dados, portanto, confirmaram a matemática.

Apesar de não chegar ao mínimo, o método pode ser utilizado de maneira exploratória com relativa facilidade, bastando extrair ρ sempre que $f_i \rightarrow 1$ e $\nabla \rho \rightarrow \mathbf{0}$.

Resta a dúvida: afinal, como o autovalor mínimo foi obtido com o *fitness* definido pela equação 6.1? Não sei. Esse *fitness* foi utilizado não só em (NANDY *et al.*, 2004), mas também em (SHARMA *et al.*, 2006), (SHARMA *et al.*, 2008) e (NANDY *et al.*, 2009), seguindo exatamente o argumento resumido pela equação 6.6. Não identifiquei nada nesses quatro artigos que pudesse levar à resposta. Segui o estudo com uma nova definição do *fitness* encontrada em (NANDY *et al.*, 2011).

Tabela 10 – Lista de autovalores para matrizes de Coope–Sabo de ordem 10, 20, 30 e 40.

#	10	20	30	40
0	0,386075	0,341237	0,319737	0,306086
1	2,461056	2,397247	2,36844	2,350583
2	4,518931	4,436173	4,401134	4,379909
3	6,572897	6,468521	6,427419	6,4031
4	8,628524	8,497626	8,450274	8,42294
5	10,69057	10,52507	10,47105	10,44068
6	12,76574	12,55178	12,4905	12,457
7	14,86753	14,57845	14,50908	14,47232
8	17,03654	16,60562	16,52713	16,48692
9	22,07215	18,63385	18,54488	18,501
10		20,6637	20,56255	20,5147
11		22,69588	22,5803	22,52816
12		24,73127	24,59828	24,54146
13		26,77114	26,61667	26,55469
14		28,81733	28,6356	28,56792
15		30,87288	30,65527	30,58122
16		32,94325	32,67586	32,59466
17		35,04014	34,6976	34,60831
18		37,19805	36,72077	36,62223
19		45,2308	38,74571	38,63648
20			40,77285	40,65114
21			42,80277	42,6663
22			44,83625	44,68204
23			46,87444	46,69846
24			48,91902	48,71568
25			50,97274	50,73385
26			53,04052	52,75311
27			55,13271	54,77369
28			57,27946	56,79581
29			68,37101	58,81981
30				60,84608
31				62,87517
32				64,90781
33				66,94504
34				68,98845
35				71,04053
36				73,10578
37				75,19353
38				77,33102
39				91,50634

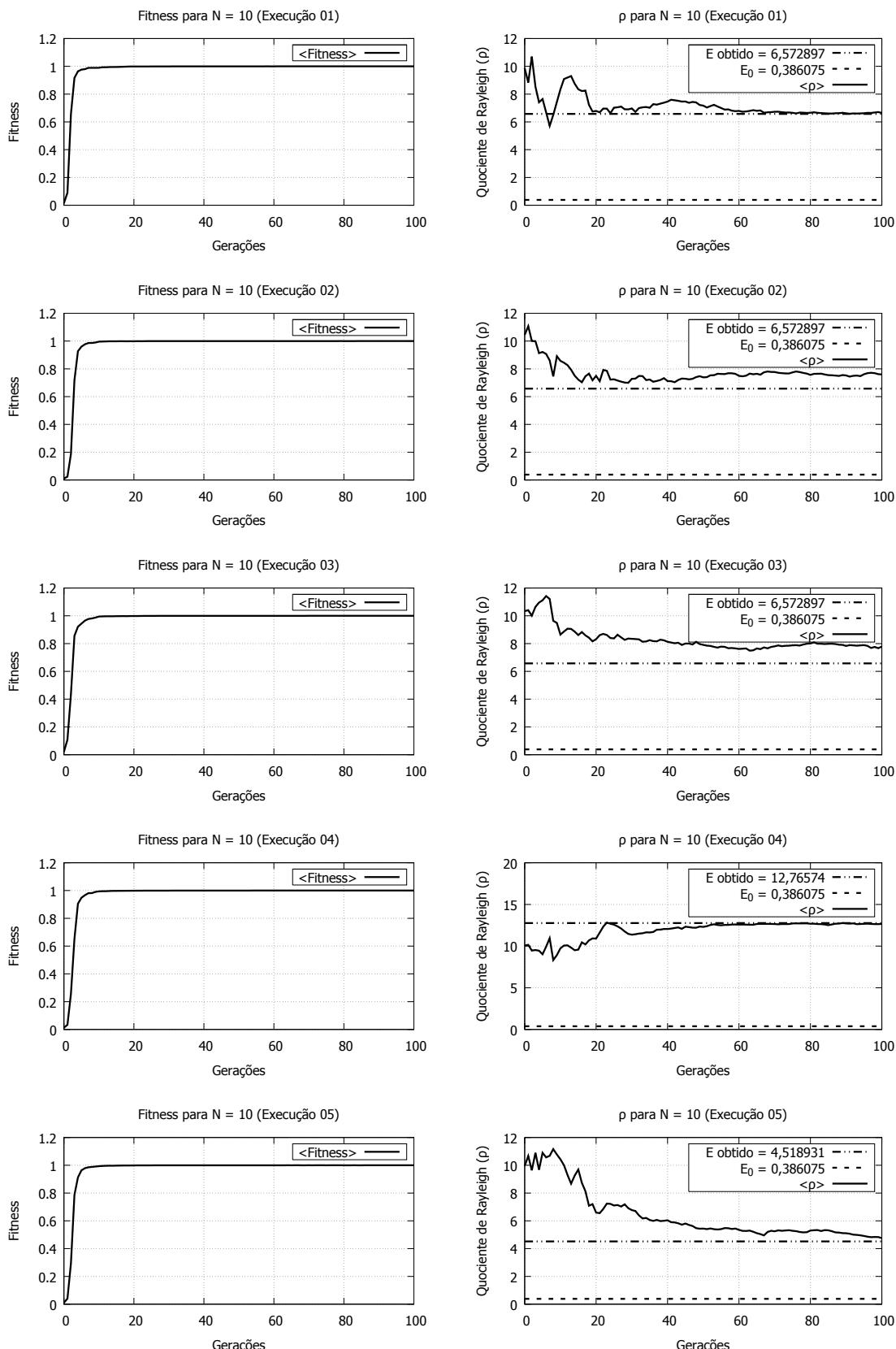


Figura 26 – Execuções N = 10.

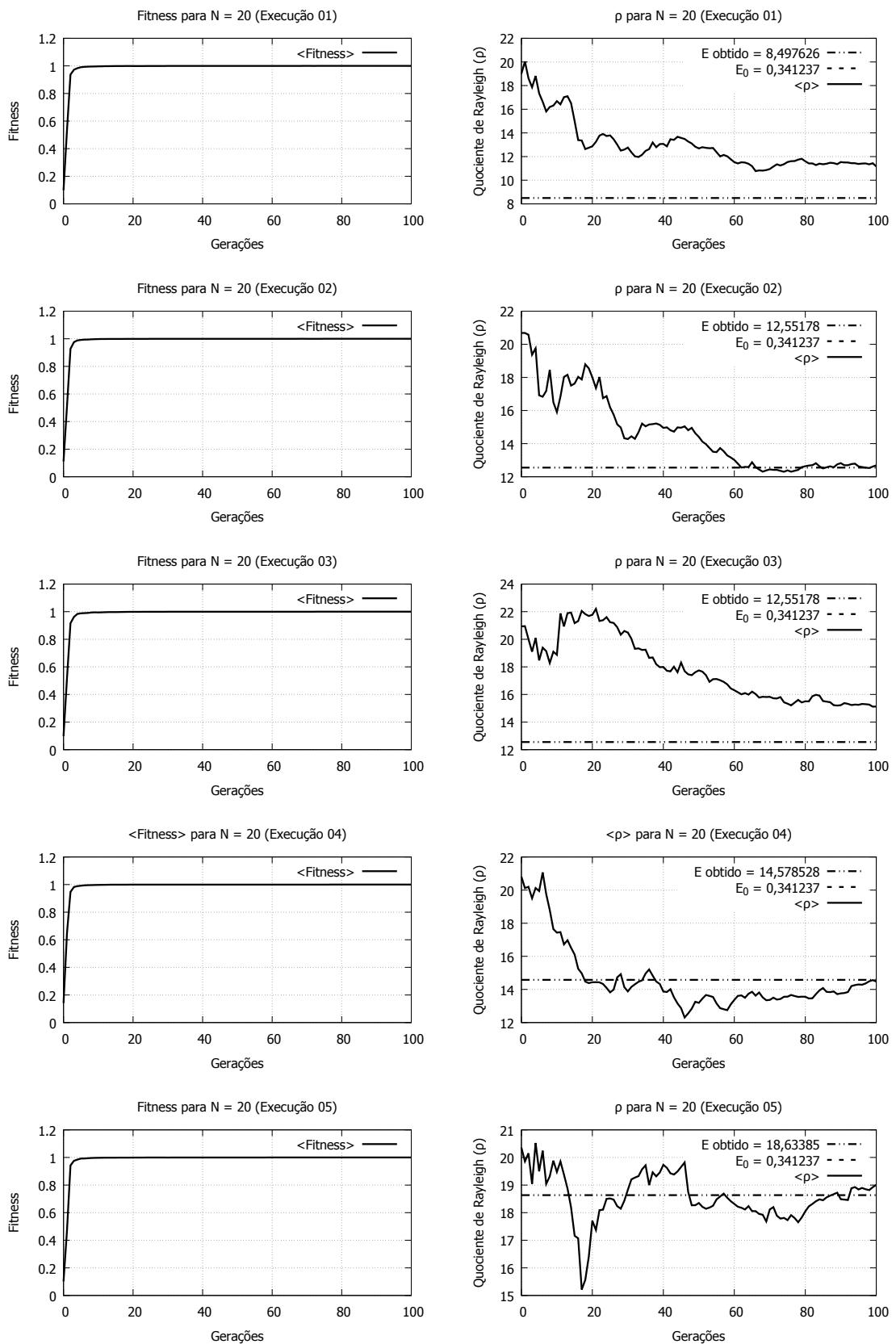
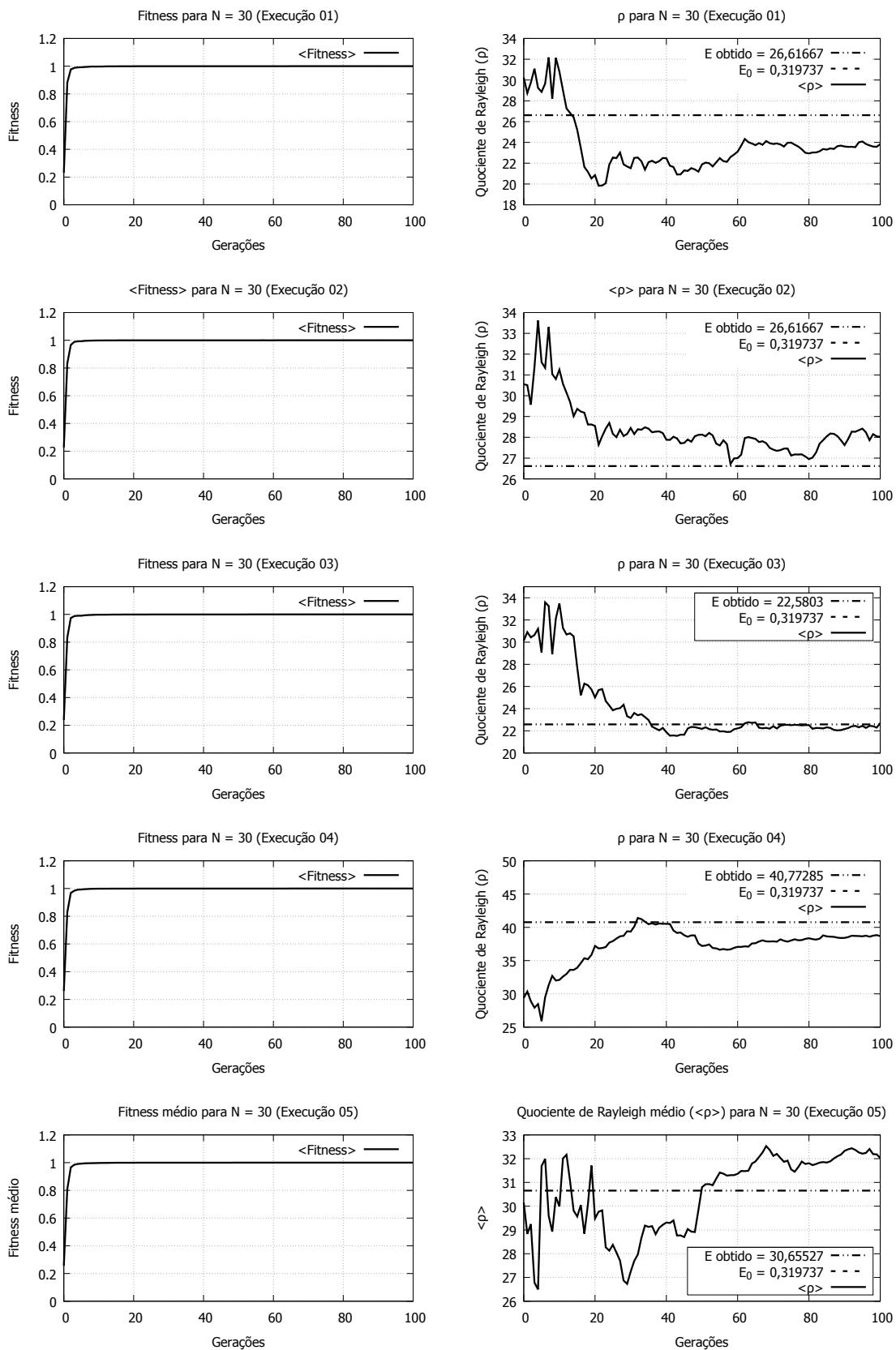


Figura 27 – Execuções N = 20.

Figura 28 – Execuções $N = 30$.

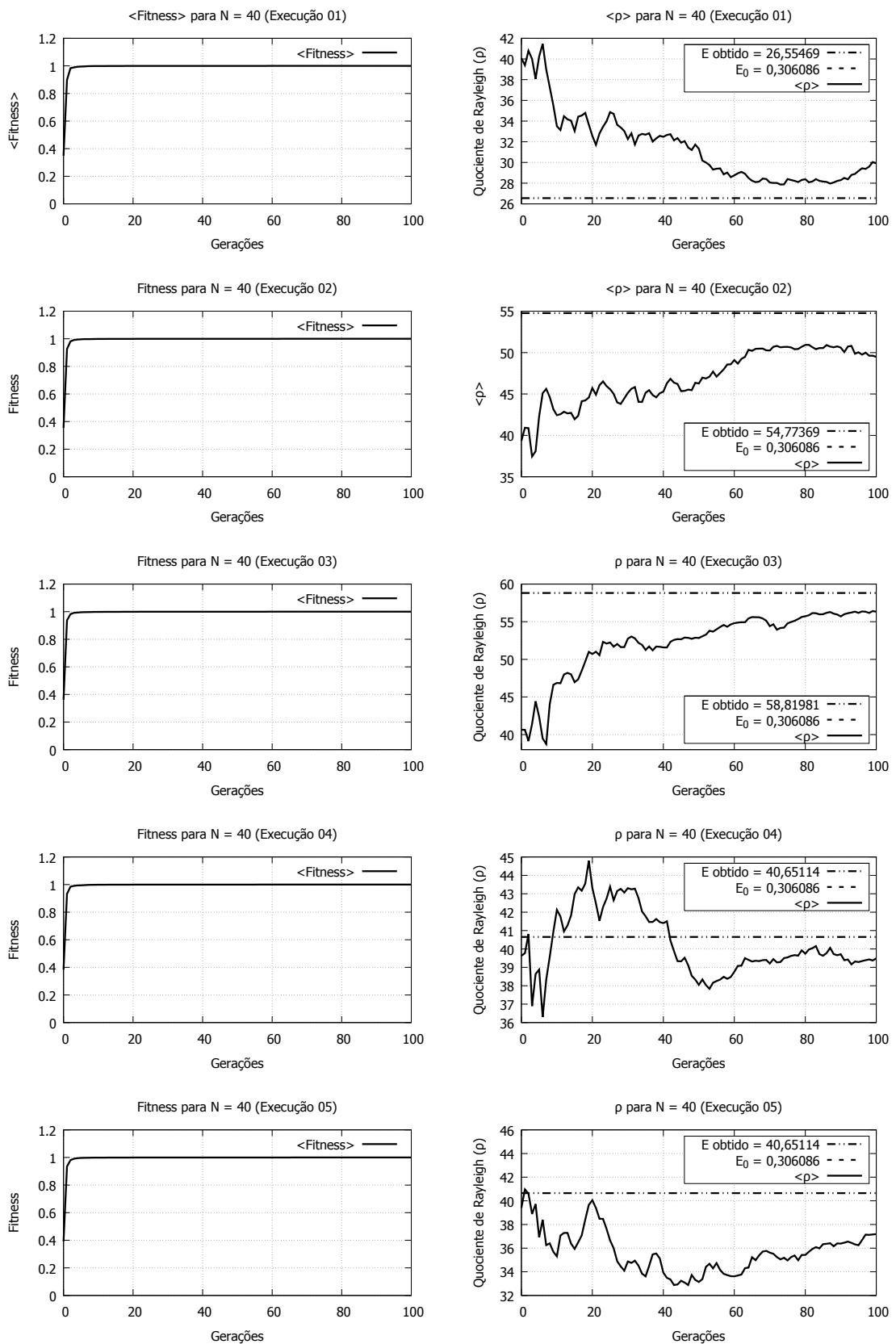
Figura 29 – Execuções $N = 40$.

Tabela 11 – Execuções para matrizes de Coop–Sabo.

N	Execução	Semente	β	$<Fitness>$	$<\rho>$	σ	# autovector	Autovector	Erro relativo
10	0	1445738835	0,128788	1,000000	2,461122	0,000023	1	2,461056	0,003%
10	1	1445780626	0,128788	1,000000	6,572898	0,000013	3	6,572897	0,00001%
10	2	1445780762	0,128788	1,000000	6,572883	0,000015	3	6,572897	-0,0002%
10	3	1445780907	0,128788	1,000000	6,572910	0,000016	3	6,572897	0,0002%
10	4	1445781049	0,128788	1,000000	12,765701	0,000016	6	12,765740	-0,0003%
10	5	1445781195	0,128788	1,000000	4,518952	0,000012	2	4,518931	0,0005%
20	1	1445795292	0,026665	1,000000	8,498192	0,000052	4	8,497626	0,007%
20	2	1445795501	0,026665	1,000000	12,551830	0,000018	6	12,551780	0,0004%
20	3	1445795718	0,026665	1,000000	12,551878	0,000020	6	12,551780	0,0008%
20	4	1445795953	0,026665	1,000000	14,578527	0,000035	7	14,578450	0,0005%
20	5	1445796166	0,026665	1,000000	18,634220	0,000062	9	18,633850	0,002%
30	1	1445796378	0,011171	1,000000	26,616790	0,000065	13	26,616670	0,0005%
30	2	1445796746	0,011171	1,000000	26,616595	0,000029	13	26,616670	-0,0003%
30	3	1445797109	0,011171	1,000000	22,580060	0,000051	11	22,580300	-0,001%
30	4	1445797473	0,011171	1,000000	40,772447	0,000071	20	40,772850	-0,001%
30	5	1445797882	0,011171	1,000000	30,655283	0,000022	15	30,655270	0,00004%
40	1	1445798248	0,006105	1,000000	26,554758	0,000040	13	26,554690	0,0003%
40	2	1445798838	0,006105	1,000000	54,773734	0,000078	27	54,773690	0,00008%
40	3	1445799429	0,006105	1,000000	58,819413	0,000087	29	58,819810	-0,0007%
40	4	1445800091	0,006105	1,000000	40,651473	0,000077	20	40,651140	0,0008%
40	5	1445800683	0,006105	1,000000	40,650764	0,000061	20	40,651140	-0,0009%

6.2 Outro *fitness* para encontrar o mínimo global

O novo *fitness*, apresentado em (NANDY *et al.*, 2011), é dado por

$$f_i = e^{-\beta(\rho_i - E_L)^2}, \quad (6.8)$$

e contém semelhanças com o definido pela equação 6.1. Há uso de uma exponencial, o parâmetro β foi mantido e possui exatamente o mesmo papel, f_i depende apenas de ρ e, como $(\rho_i - E_L)^2$ é claramente positivo, o *fitness* continua limitado ao conjunto $(0,1]$. As diferenças estão na ausência do $\nabla\rho$ e na inclusão do parâmetro E_L , que representa um limite inferior para o menor autovalor². Por exemplo, se soubermos de antemão que o autovalor *mínimo* é maior que zero, poderíamos definir $E_L = 0$.

A justificativa para o funcionamento do método em (NANDY *et al.*, 2011) segue a mesma estrutura de (NANDY *et al.*, 2004): “Se $\rho_i \rightarrow E_L$ durante a busca, $f_i \rightarrow 1$ e C_i está próximo do autovetor fundamental de H ”³. Parece que, novamente, não há garantia de que, se $f_i \rightarrow 1$, ρ tende, necessariamente, ao autovalor fundamental. E aqui há um agravante: nada na equação 6.8 está diretamente associado aos autovalores de H . Lembre que o *fitness* anterior (equação 6.1) contém $\nabla\rho$, que possui relação direta com os autovalores de H quando $\nabla\rho = 0$.

O autovalor mínimo foi encontrado, mas a qualidade foi inferior. Repeti as execuções da tabela 11 alterando apenas o *fitness* e configurando o parâmetro E_L para $E_L = 0$, um pouco abaixo dos autovalores mínimos. Os resultados estão na página 86, e os gráficos da evolução do *fitness* e do quociente de Rayleigh estão nas páginas 92, 93, 94 e 95. Surpreendentemente, apesar do que foi dito no parágrafo anterior, o programa encontrou o menor autovalor em **todos** os casos. Assim como nas primeiras execuções, o desvio padrão (σ) de $\langle \rho \rangle$ na última geração (400.000) foi pequeno, indicando convergência genética. Entretanto, essa foi a única semelhança. Os próprios valores de σ são uma ordem de grandeza menores, sugerindo que os indivíduos são mais semelhantes entre si. O *fitness* médio só atingiu seu valor máximo para a matriz de ordem $N = 40$. Aliás, especificamente para E_L fixado em $E_L = 0$, o $\langle fitness \rangle$ final diminui com N , pois E_L está mais distante de E_0 na matriz de ordem 10 do que na de ordem 40. Os erros relativos não ultrapassaram 1%, mas foram substancialmente maiores comparados aos obtidos com o primeiro *fitness*. Enquanto nos testes anteriores seus valores permaneceram estáveis, agora os erros relativos apresentaram tendência de crescimento com N .

² L de *lower*.

³ Tradução livre de “If $\rho_i \rightarrow E_L$ during the search, $f_i \rightarrow 1$ and C_i approaches the ground eigenvector of H ”.

Tabela 12 – Execuções novo *Fitness*.

N	Execução	Semente	β	$<\text{Fitness}>$	$<\rho>$	σ	# autovalor	Autovalor	Erro relativo (%)
10	0	1445738835	0,128788	0,999044	0,386176	0,00005	0	0,3860745	0,03%
10	1	1445780626	0,128788	0,999044	0,386169	0,00003	0	0,3860745	0,02%
10	2	1445780762	0,128788	0,999045	0,386132	0,00002	0	0,3860745	0,01%
10	3	1445780907	0,128788	0,999044	0,386175	0,00005	0	0,3860745	0,03%
10	4	1445781049	0,128788	0,999043	0,386211	0,00003	0	0,3860745	0,04%
10	5	1445781195	0,128788	0,999044	0,386183	0,00005	0	0,3860745	0,03%
20	1	1445795292	0,026665	0,999954	0,341484	0,00005	0	0,3412367	0,07%
20	2	1445795501	0,026665	0,999954	0,341693	0,0001	0	0,3412367	0,1%
20	3	1445795718	0,026665	0,999954	0,34147	0,00006	0	0,3412367	0,07%
20	4	1445795953	0,026665	0,999954	0,341689	0,0001	0	0,3412367	0,1%
20	5	1445796166	0,026665	0,999954	0,34153	0,00007	0	0,3412367	0,09%
30	1	1445796378	0,011171	0,999995	0,320582	0,0001	0	0,319737	0,3%
30	2	1445796746	0,011171	0,999995	0,320772	0,0002	0	0,319737	0,3%
30	3	1445797109	0,011171	0,999995	0,320699	0,0001	0	0,319737	0,3%
30	4	1445797473	0,011171	0,999995	0,320755	0,0001	0	0,319737	0,3%
30	5	1445797882	0,011171	0,999995	0,320274	0,00007	0	0,319737	0,2%
40	1	1445798248	0,006105	1	0,306968	0,0001	0	0,306086	0,3%
40	2	1445798838	0,006105	1	0,307128	0,0001	0	0,306086	0,3%
40	3	1445799429	0,006105	1	0,307297	0,0002	0	0,306086	0,4%
40	4	1445800091	0,006105	1	0,307816	0,0002	0	0,306086	0,6%
40	5	1445800683	0,006105	1	0,30765	0,0002	0	0,306086	0,5%

Apesar das diferenças dos valores finais, o comportamento do *fitness* e do ρ ao longo da busca não foi alterado significativamente. Na figura 30 estão os gráficos referentes à execução zero para o Hamiltoniano de ordem 10, semente 1445738835. A primeira usa o *fitness* $f_i = e^{-\beta(\rho_i - E_L)^2}$, que chega ao autovalor mínimo, enquanto a segunda utiliza o $f_i = e^{-\beta\|\nabla\rho_i\|^2}$. Ambos iniciam com valores muito baixos e convergem para 1, entretanto, o da esquerda é muito ruidoso e, aparentemente, essa é a causa da convergência mais lenta. Quando a curva da direita já está estável em $\langle f \rangle \approx 1$ em torno da geração de número 15, a da esquerda ainda não ultrapassou o $\langle f \rangle = 0,1$. A princípio, não podemos comparar os dois comportamentos diretamente, visto que cada um chegou a um autovalor diferente. A execução da direita, lembre-se, obteve apenas um mínimo local ($E_1 = 2,461056$, tabela 11).

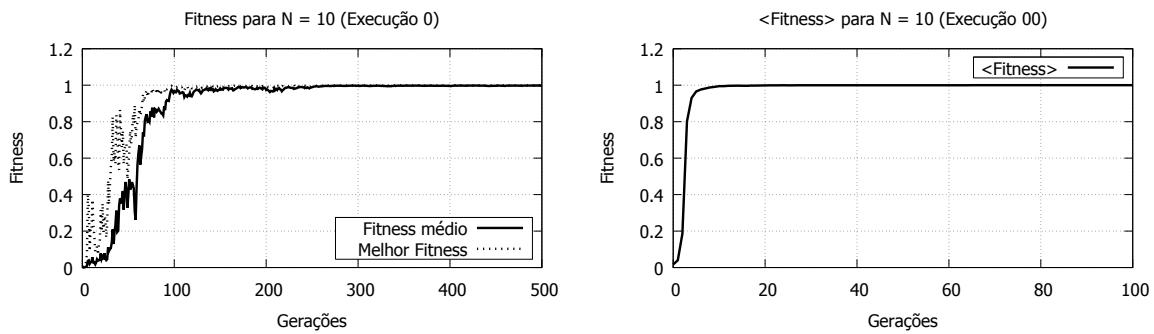


Figura 30 – Comportamento do *fitness* para as execuções zero do Hamiltoniano de ordem 10, semente 1445738835. A primeira usa o *fitness* $f_i = e^{-\beta(\rho_i - E_L)^2}$, que chega ao autovalor mínimo, enquanto a segunda utiliza o $f_i = e^{-\beta\|\nabla\rho_i\|^2}$.

De todo modo, as duas execuções estão conectadas pois, como partiram da mesma semente de números pseudoaleatórios, a população inicial foi a mesma. Inclusive, na primeira geração, em ambas as execuções, os valores para $\langle \rho \rangle$ e para o melhor ρ foram, respectivamente, 9,876075 e 9,557892, igualmente distantes do autovalor mínimo $E_0 = 0,386075$. Os gráficos da figura 31 permitem comparar a evolução do $\langle \rho \rangle$ nos dois casos. Assim como na figura anterior, a imagem da esquerda refere-se ao uso do *fitness* $f_i = e^{-\beta(\rho_i - E_L)^2}$.

Alguém poderia afirmar que a causa de uma execução ter sido mais lenta do que a outra foi porque percorreu um caminho mais longo ao sair de $\langle \rho \rangle = 9,876075$, passar por $E_1 = 2,461056$ e continuar até encontrar $E_0 = 0,386075$, enquanto a mais rápida saiu do mesmo $\langle \rho \rangle$ e parou logo que encontrou E_1 . Infelizmente essa conclusão estaria incorreta. A maneira como os Algoritmos Genéticos percorrem o espaço de soluções tem base estocástica e, portanto, qualquer comparação linear é extremamente arriscada, quiçá impossível. Objetivamente, posso apenas concluir que os valores finais encontrados por cada *fitness* condizem com a construção de cada função objetivo: $\nabla\rho_i$ leva a qualquer autovalor; $\rho_i - E_L$, com E_L configurado apropriadamente, encontra o autovalor mínimo.

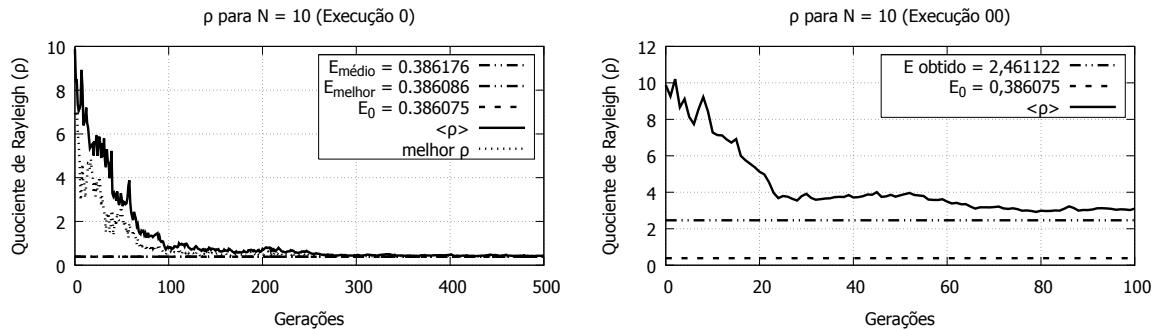


Figura 31 – Comportamento do ρ para as execuções zero do Hamiltoniano de ordem 10, semente 1445738835. A primeira usa o *fitness* $f_i = e^{-\beta(\rho_i - E_L)^2}$, que chega ao autovalor mínimo, enquanto a segunda utiliza o $f_i = e^{-\beta\|\nabla\rho_i\|^2}$.

O limite inferior E_L é o responsável pelo funcionamento do *fitness* (equação 6.8), pois o termo $(\rho_i - E_L)$ minimiza diretamente ρ . Se $(\rho_i - E_L)$ é grande, f_i é pequeno, e isso significa que o vetor C_i associado a ρ_i está distante do autovetor fundamental C_0 . Portanto, indivíduos com $(\rho_i - E_L)$ menores (e *fitness* maiores) serão selecionados mais vezes. De acordo com a equação 2.1, o processo fica estável quando é impossível diminuir ρ_i , ou seja, quando $\langle \rho \rangle \rightarrow E_0$.

À primeira vista, esse *fitness* não parece útil. Os valores finais do *fitness* só foram próximos de 1 porque escolhi E_L próximo de E_0 . Ou seja, eu tinha conhecimento prévio do menor autovalor. Se o objetivo é encontrar E_0 e eu não tenho conhecimento sobre a região onde ele se encontra, como escolher apropriadamente E_L ? Os autores de (NANDY *et al.*, 2011) não falam nada a respeito.

Então, para a mesma semente 1445738835, $\beta = 0,128788$ e $N = 10$, cujo $E_0 = 0,386075$, testei quatro cenários. O objetivo foi verificar em quais condições o autovalor mínimo é encontrado. O resumo dos resultados está na tabela 13.

- **Cenário 1:** E_L um pouco acima de E_0 .
- **Cenário 2:** E_L um pouco abaixo do E_0 .
- **Cenário 3:** E_L muito acima de E_0 .
- **Cenário 4:** E_L muito abaixo de E_0 .

Não há surpresa no **Cenário 2**. Como citado no início desta seção, a escolha de E_L um pouco abaixo de E_0 garante o mínimo global. Nos testes obtive o menor autovalor em todas as execuções, com *fitness* médio próximo de 1 e $\langle |\nabla\rho| \rangle$ próximo de zero. Portanto, o **Cenário 2** é o melhor cenário possível.

Apesar de não obter E_0 , o **Cenário 1** dá uma informação importante. O algoritmo termina sempre quando $\langle \rho \rangle$ fica próximo de E_L , mas não obtém E_0 . Como

pode ser visto na figura 32, $\langle f \rangle$ chega ao valor máximo 1 (gráfico da esquerda), mas o valor final médio para ρ foi $E_{\text{médio}} = 0,387001$ (gráfico da direita). Então, se o algoritmo parar em E_L , significa que o autovalor mínimo é, com certeza, menor: $E_0 < E_L$.

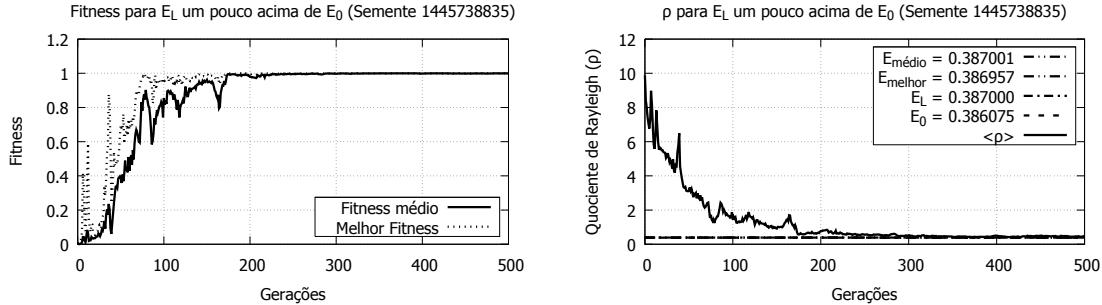


Figura 32 – Cenário 1. Execução para a semente 1445738835. E_L um pouco acima de E_0 no fitness $f_i = e^{-\beta(\rho_i - E_L)^2}$.

Com relação ao **Cenário 3** (*muito acima*), novamente o algoritmo chegou ao E_L em todas as execuções. Entretanto, na última geração $|\nabla \rho|$ foi mais que trinta vezes maior comparado com o **Cenário 1** ($0,003/0,00009 \approx 33$). Retomando a condição de estacionariedade do Quociente de Rayleigh (equação 2.7), isso significa que $\langle \rho \rangle$ está mais distante de algum autovalor.

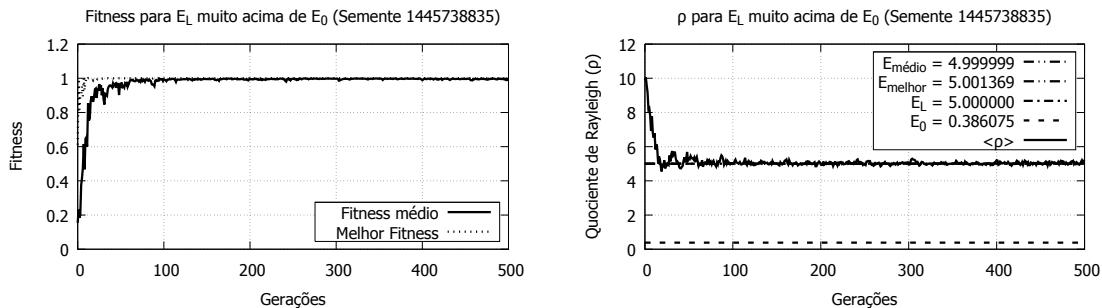


Figura 33 – Execução para a semente 1445738835. E_L muito acima de E_0 no fitness $f_i = e^{-\beta(\rho_i - E_L)^2}$.

O valor de $\beta = 0,128788$, utilizado em todos os cenários anteriores, não foi adequado para os parâmetros do **Cenário 4**. Logo no início, tanto o $\langle \text{fitness} \rangle$ quanto o maior fitness foram praticamente zero. Nesse regime não há como, a princípio, distinguir os indivíduos, pois todos possuem avaliação muito próxima. Veja no segundo gráfico da figura 34 que $\langle \rho \rangle$ rapidamente fica estagnado um pouco abaixo de 6. Especificamente, na geração 500, o Quociente de Rayleigh médio era $\langle \rho \rangle = 5,651846$, que não corresponde a nenhum autovalor para uma matriz de Coope de ordem 10 (tabela 10). Esse é um exemplo de *underflow* do fitness.

Entretanto, após várias gerações, houve convergência para o autovalor mínimo (figura 35). Um pouco antes da geração 32.000 aconteceu um salto no fitness (gráfico da

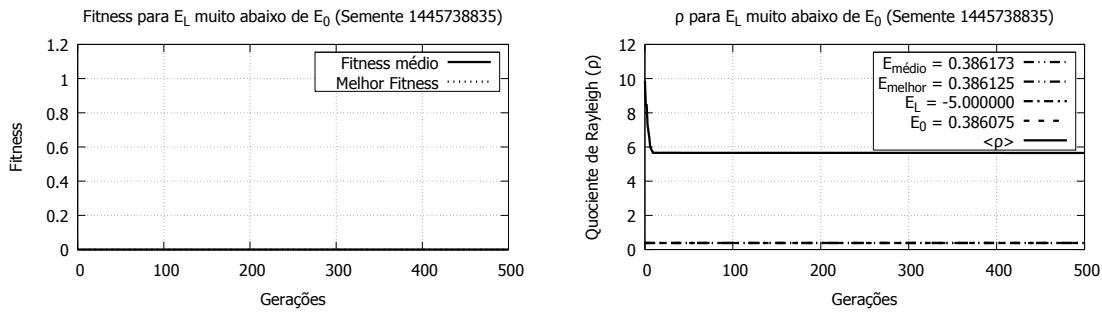


Figura 34 – Execução para a semente 1445738835. E_L muito abaixo de E_0 no fitness $f_i = e^{-\beta(\rho_i - E_L)^2}$. Até geração 500.

esquerda), consequência de uma queda brusca do $\langle \rho \rangle$ (gráfico da direita), que só pode ter sido possível com um salto na variabilidade genética na população, descontinuidade característica da Mutação. Apesar de o fitness médio continuar pequeno após a mudança ($\langle f_i \rangle < 0,025$), o crossover foi capaz de atuar com a nova informação genética e criou variabilidade suficiente para chegar ao autovalor mínimo.

Mesmo se não houvesse a descontinuidade, acredito que a queda de $\langle \rho \rangle$ continuaria e haveria convergência para $\langle \rho \rangle \approx E_0$. Veja no segundo gráfico da figura 35 que entre o intervalo $0 \leq \langle \rho \rangle \leq 30.000$ há uma queda lenta, porém sistemática, de $\langle \rho \rangle$. Ou seja, apesar de pequeno, muito próximo de zero, o fitness de cada indivíduo foi suficiente para permitir distinção na Seleção e reprodução no Crossover. Portanto, se não ocorresse a queda brusca do $\langle \rho \rangle$ em torno de $\langle \rho \rangle = 32.000$, a diminuição seguiria lenta e o algoritmo chegaria ao autovalor mínimo.

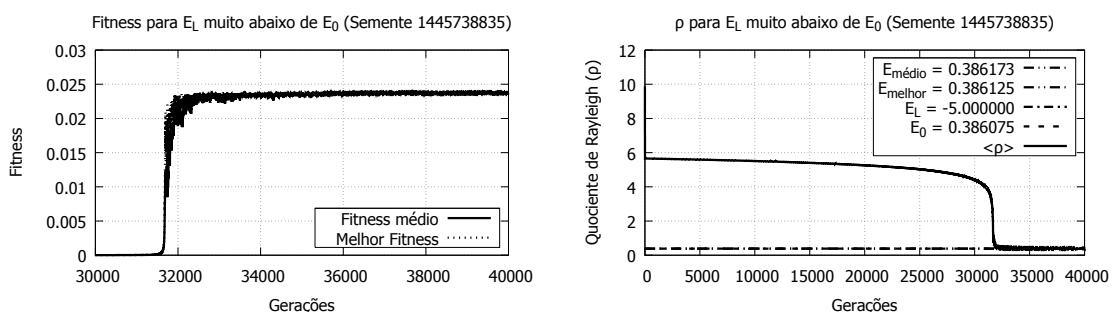


Figura 35 – Execução para a semente 1445738835. E_L muito abaixo de E_0 no fitness $f_i = e^{-\beta(\rho_i - E_L)^2}$. Geração entre 30.000 e 40.000.

Na tabela 13 há os valores desses testes. Como nas tabelas anteriores, os valores médios de ρ e do fitness ($\langle \rho \rangle$ e $\langle \text{fitness} \rangle$) foram calculados na geração final, ou seja, na população que atingiu algum dos critérios de parada. O $\langle \rho \rangle$ foi comparado com $E_0 = 0,386075$ para calcular o erro relativo (coluna Erro do $\langle \rho \rangle$ (%)).

Tabela 13 – Variando E_L para a execução da semente 1445738835. Os tipos de teste são:
cenário 1: E_L um pouco acima de E_0 ; **cenário 2:** E_L um pouco abaixo de E_0 ;
cenário 3: E_L muito acima de E_0 ; **cenário 4:** E_L muito abaixo de E_0 .

Teste	E_L	Geração final	$\langle \rho \rangle$	σ	Erro do $\langle \rho \rangle$ (%)	$ \nabla \rho $	$\langle \text{Fitness} \rangle$
1	0,387000	42.577	0,3870	0,0004	0,2%	0,00009	1,000000
2	0,385000	400.000	0,38615	0,00003	0,02%	0,000006	1,000000
3	5,000000	9.622	5,00	0,02	1195%	0,003	0,999966
4	-5,000000	400.000	0,38617	0,00003	0,03%	0,0003	0,023843

Em função dos resultados obtidos no cenário 4, analisei o *fitness* de ([NANDY et al., 2011](#)). O objetivo foi verificar como a variação dos parâmetros β e E_L alteram a forma de $f = e^{-\beta(\rho-E_L)^2}$, e quais as consequências para o algoritmo genético.

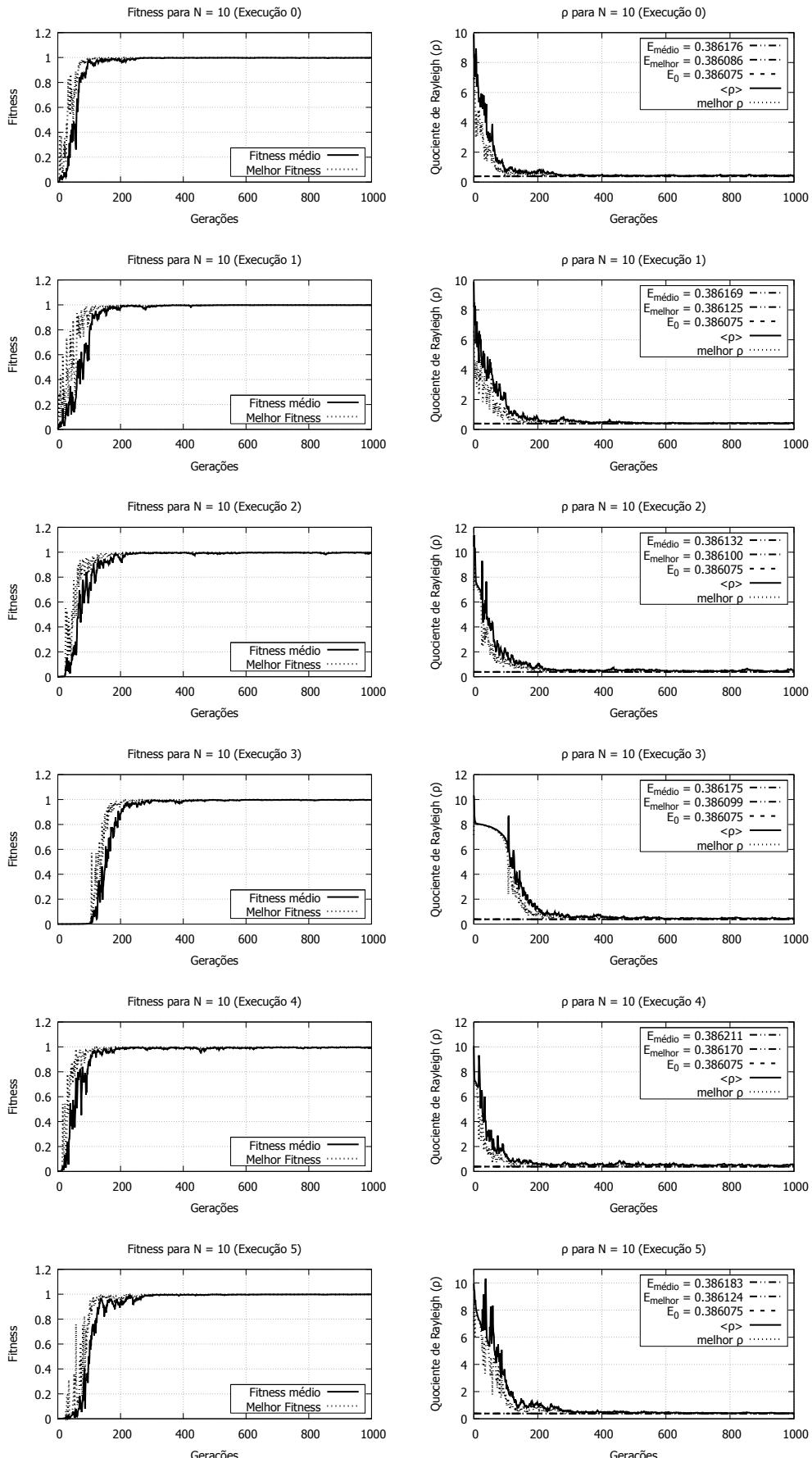


Figura 36 – Execuções para $N = 10$ com o fitness $f_i = e^{-\beta(\rho_i - E_L)^2}$.

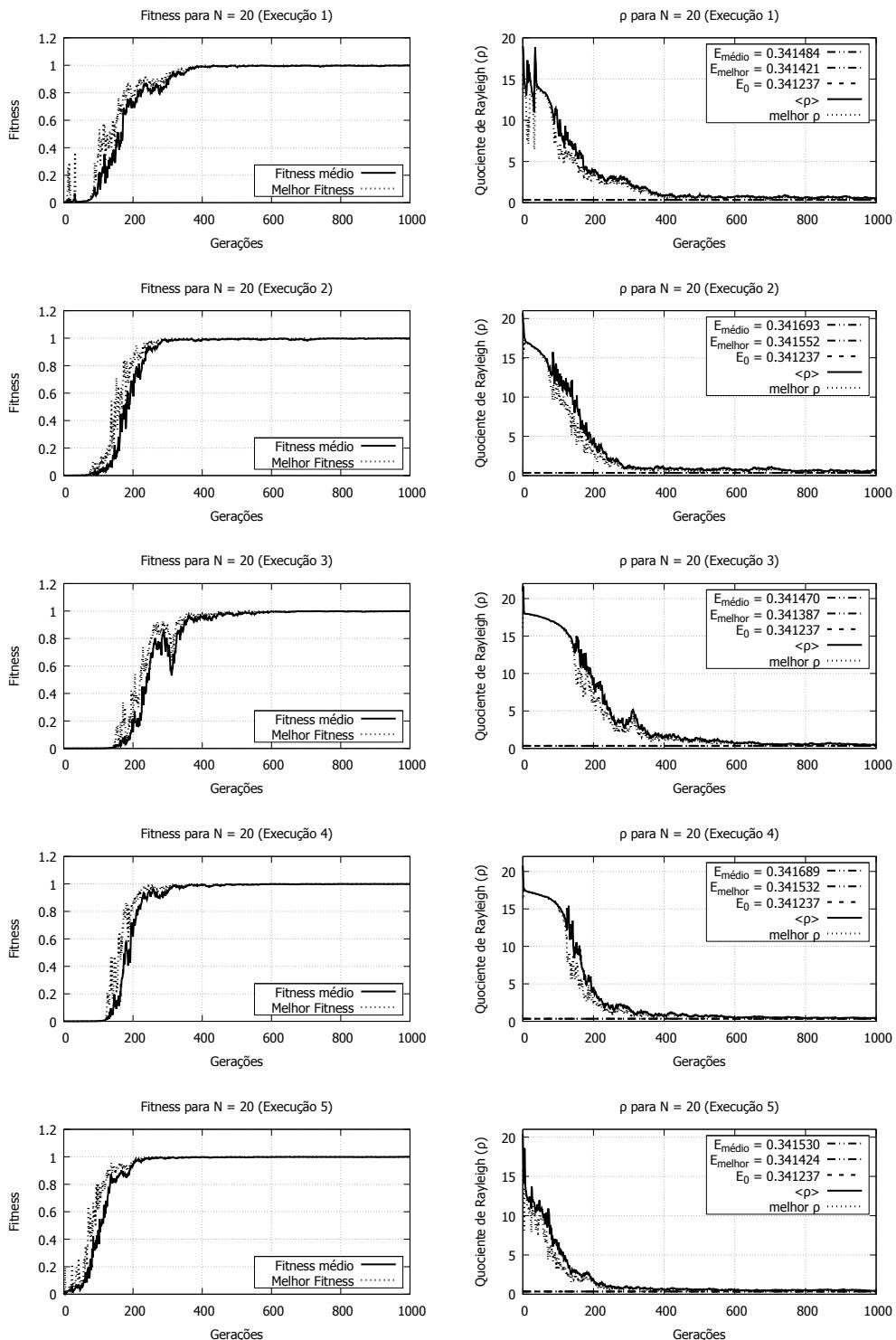


Figura 37 – Execuções para $N = 20$ com o fitness $f_i = e^{-\beta(\rho_i - E_L)^2}$.

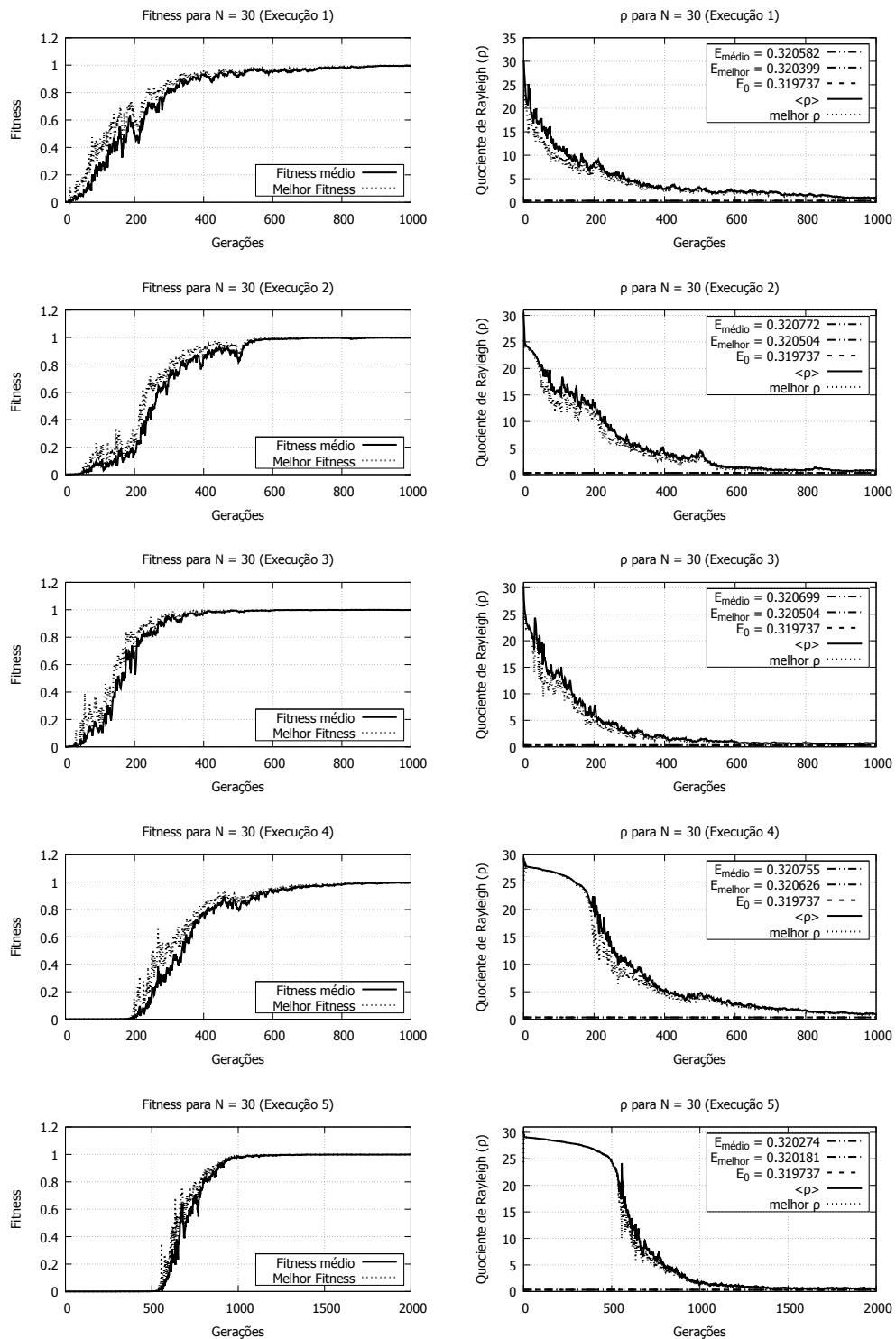


Figura 38 – Execuções para $N = 30$ com o fitness $f_i = e^{-\beta(\rho_i - E_L)^2}$.

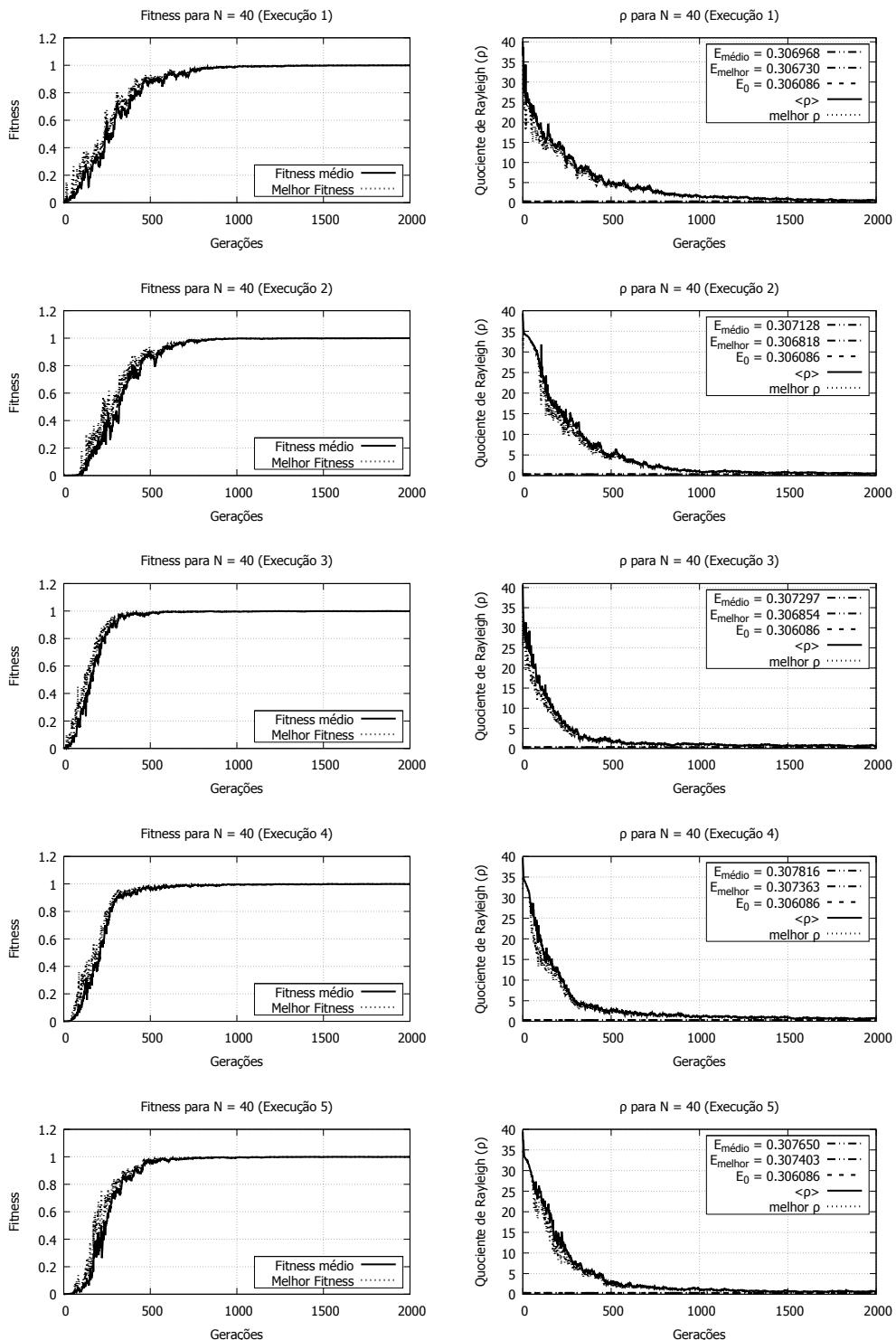


Figura 39 – Execuções para $N = 40$ com o fitness $f_i = e^{-\beta(\rho_i - E_L)^2}$.

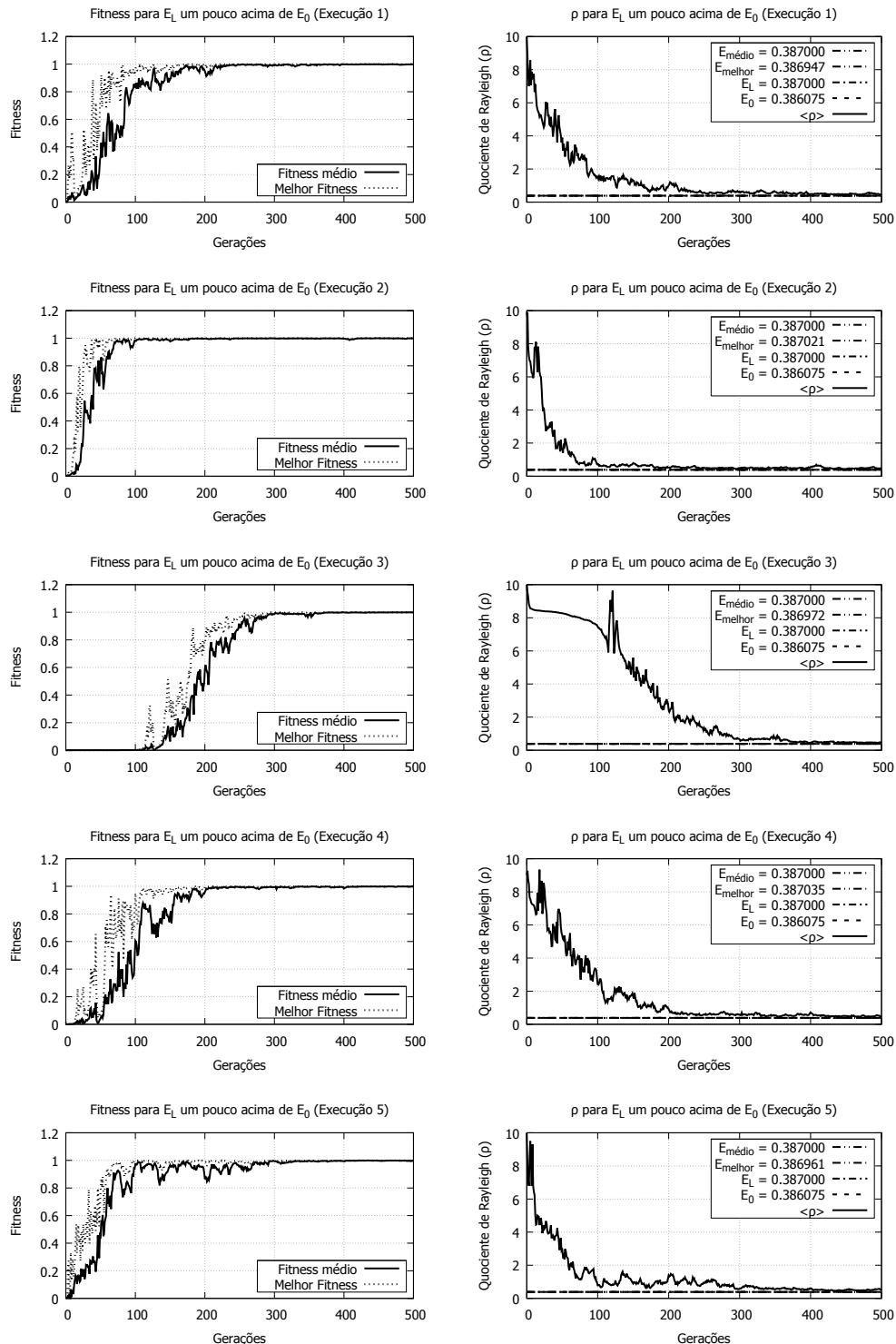


Figura 40 – Cenário 1. Várias execuções com o E_L um pouco acima de E_0 no fitness $f_i = e^{-\beta(\rho_i - E_L)^2}$. Semente 1445738835, $N = 10$.

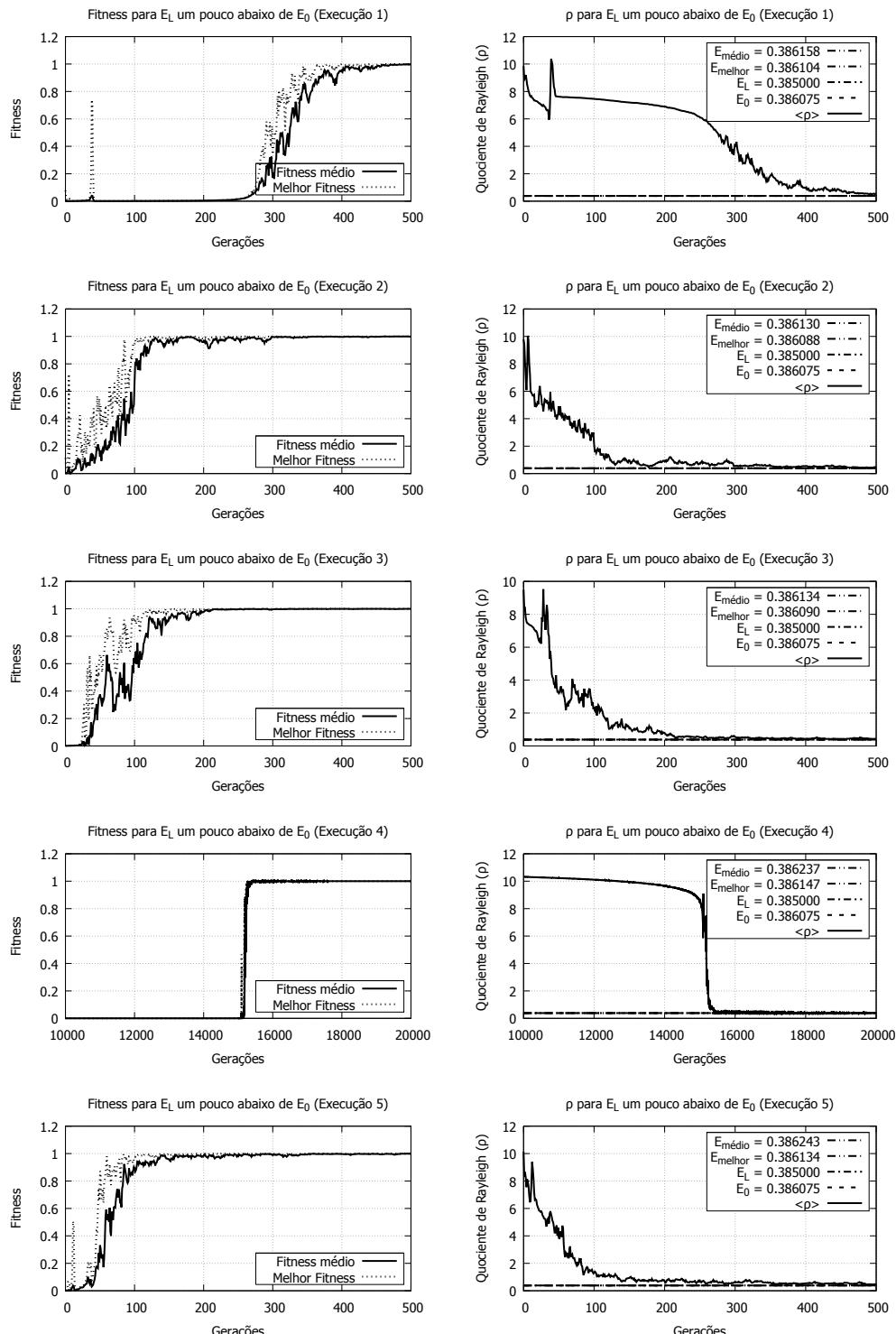


Figura 41 – Execuções com o E_L um pouco abaixo de E_0 no fitness $f_i = e^{-\beta(\rho_i - E_L)^2}$. Semente 1445738835, $N = 10$.

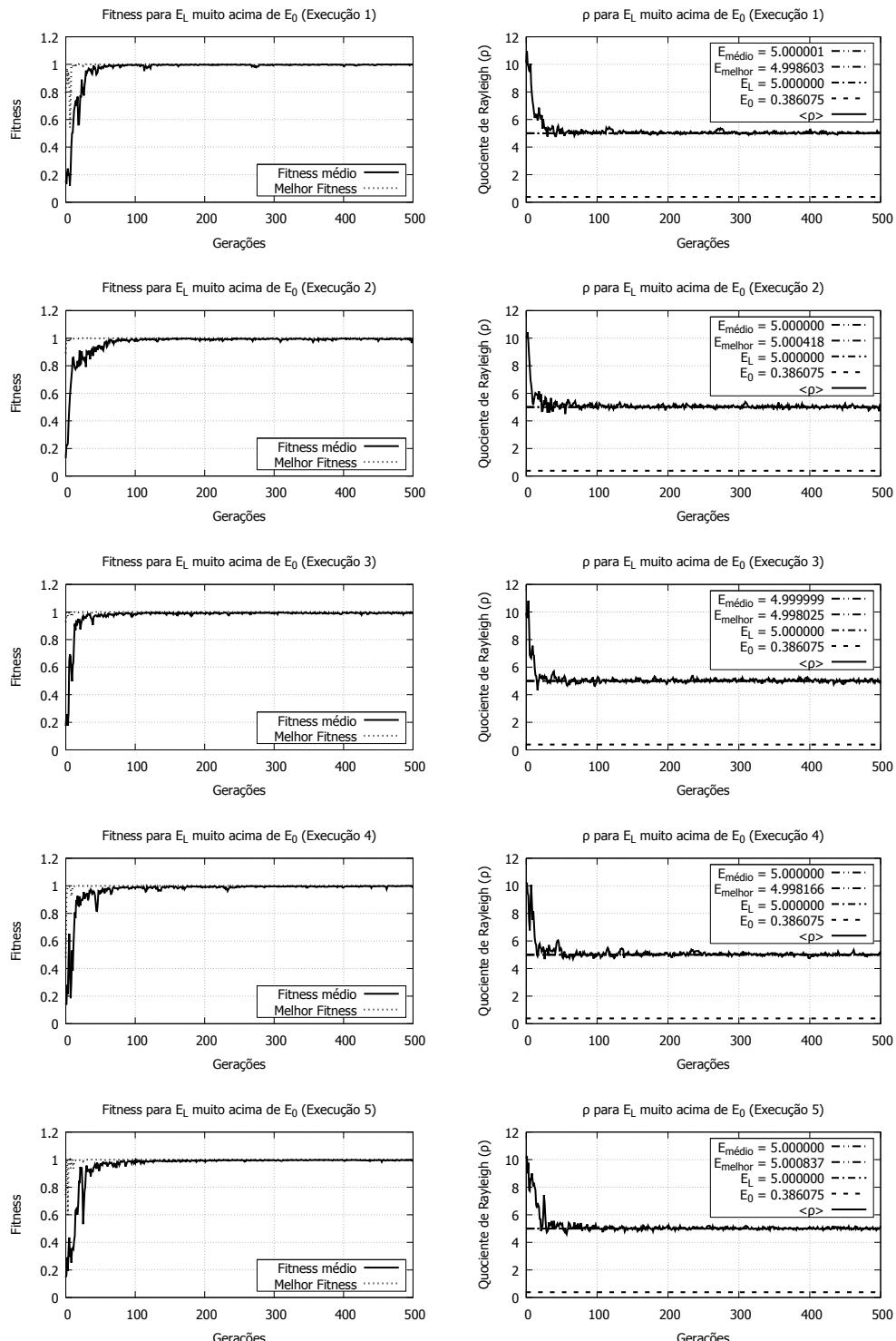


Figura 42 – Execuções com o E_L muito acima de E_0 no fitness $f_i = e^{-\beta(\rho_i - E_L)^2}$. Semente 1445738835, $N = 10$.

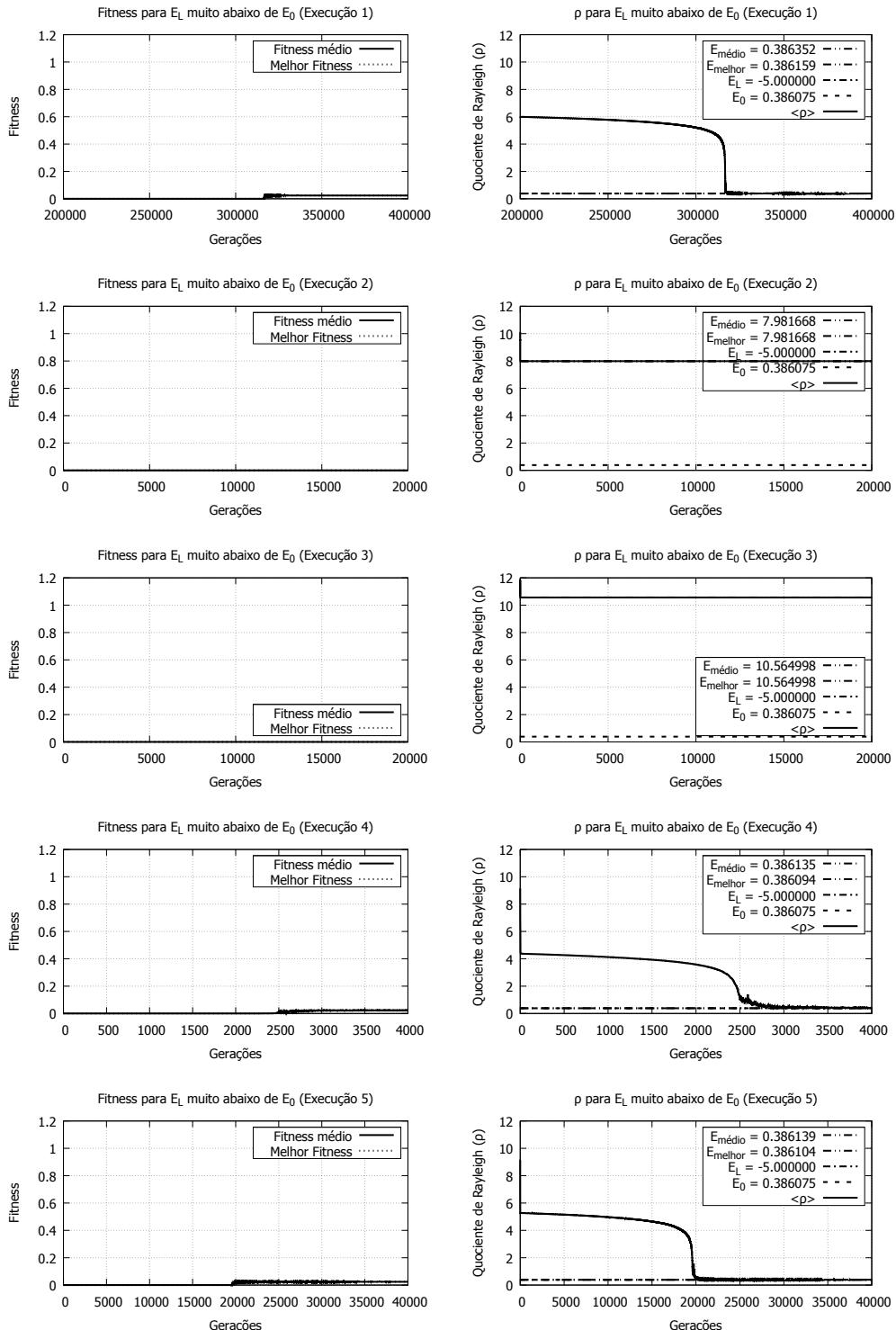


Figura 43 – Execuções com o E_L muito abaixo de E_0 no fitness $f_i = e^{-\beta(\rho_i - E_L)^2}$. Semente 1445738835, N = 10.

Tabela 14 – Cinco execuções para cada tipo de teste de variação de E_L em torno de E_0 no fitness $f_i = e^{-\beta(\rho_i - E_L)^2}$.

Teste	Execução	Semente	Geração final	$\langle \rho \rangle$	σ	Erro do $\langle \rho \rangle$ (%)	$ \nabla \rho $	$\langle Fitness \rangle$
1	1	1448150274	47.945	0,3870	0,0005	0,00005%	0,00008	1,000000
1	2	1448150289	24.128	0,3870	0,0004	-0,00004%	0,00008	1,000000
1	3	1448150298	40.795	0,3870	0,0003	0,000007%	0,00008	1,000000
1	4	1448150315	17.047	0,3870	0,0005	-0,0001%	0,0001	1,000000
1	5	1448150321	16.284	0,3870	0,0003	0,00002%	0,00008	1,000000
2	1	1448150327	400.000	0,38616	0,00003	0,02%	0,00009	1,000000
2	2	1448150472	400.000	0,38613	0,00002	0,01%	0,00005	1,000000
2	3	1448150600	400.000	0,38613	0,00002	0,02%	0,00005	1,000000
2	4	1448150704	400.000	0,38624	0,00008	0,04%	0,00002	1,000000
2	5	1448150809	400.000	0,38624	0,00007	0,04%	0,00001	1,000000
3	1	1448150912	8.074	5,00	0,05	0,00002%	0,007	0,999750
3	2	1448150914	14.604	5,00	0,03	-0,000005%	0,009	0,999889
3	3	1448150918	41.659	5,00	0,02	-0,00002%	0,003	0,999954
3	4	1448150929	9.775	5,00	0,03	0,000009%	0,006	0,999886
3	5	1448150932	12.637	5,00	0,03	-0,0000006%	0,005	0,999904
4	1	1448150935	400.000	0,3864	0,0001	0,07%	0,001	0,023837
4	2	1448151040	400.000	7,98166818	0,00000001	1967%	6,0	0,000000
4	3	1448151146	400.000	10,564998429558	0,0000000002	2637%	8,6	0,000000
4	4	1448151251	400.000	0,38613	0,00002	0,02%	0,0003	0,023844
4	5	1448151357	400.000	0,38614	0,00003	0,02%	0,0003	0,023844

6.3 Análise do *fitness* e equação empírica para β

Nos artigos (NANDY *et al.*, 2004) e (NANDY *et al.*, 2011) os autores dizem que o β do *fitness* deve ser escolhido cuidadosamente, mas não justificam com detalhes essa afirmação. Nesta sessão apresento uma análise do *fitness* de (NANDY *et al.*, 2011) e algumas justificativas para essa importância. Isso me permitiu definir uma equação empírica para o parâmetro β . Seu uso é restrito a matrizes de Coope–Sabo e ao *fitness* do (NANDY *et al.*, 2011). Porém, com um pequeno ajuste, ela mostrou-se adequada também para o uso no *fitness* de (NANDY *et al.*, 2004).

A função de avaliação de (NANDY *et al.*, 2011) é dada por

$$f_i = e^{-\beta(\rho_i - E_L)^2}. \quad (6.9)$$

Na figura 44 é possível verificar que f é simétrica em torno de E_L , e possui máximo quando $\rho = E_L$. Alterar E_L causa um deslocamento do máximo, mantendo a simetria (figura 45). Essas propriedades são importantes, pois permitem que mudemos E_L , limite inferior para o menor autovalor, sem prejuízo do comportamento da função de avaliação.

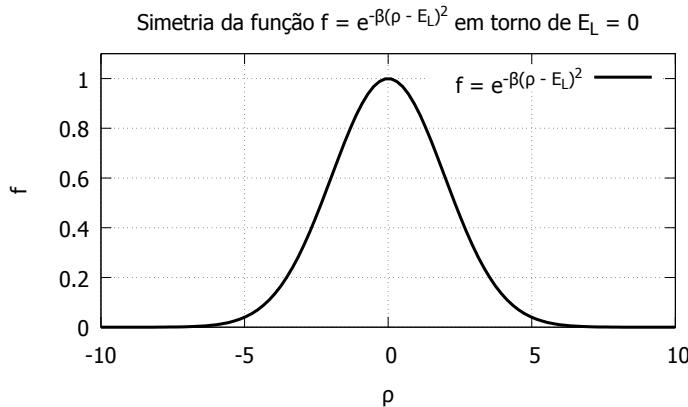


Figura 44 – Simetria de $f_i = e^{-\beta(\rho - E_L)^2}$. Nesse caso $E_L = 0$.

Na figura 46 f é exibida, com $E_L = 0$, para cinco valores de β . No eixo das abscissas estão os valores de ρ , e nas ordenadas, os de f . Note que alterar β mantém o centro da função, mas muda seu formato. Isso causa impacto na Seleção em dois pontos.

O primeiro é como isso pode influenciar a comparação dos indivíduos quando ρ se aproxima de E_L . Note que o máximo de f é muito mais acentuado para $\beta_f = 0,1 = 10^{-1}$ que para $\beta_a = 0,000001 = 10^{-6}$, fazendo com que o *fitness* com β_a pareça uma horizontal $f \approx 1$ no intervalo $\rho = [-20, 20]$. Isso significa que um indivíduo ruim, com ρ distante de $E_L = 0$, digamos $\rho = 10$, terá praticamente a mesma nota de um cromossomo com

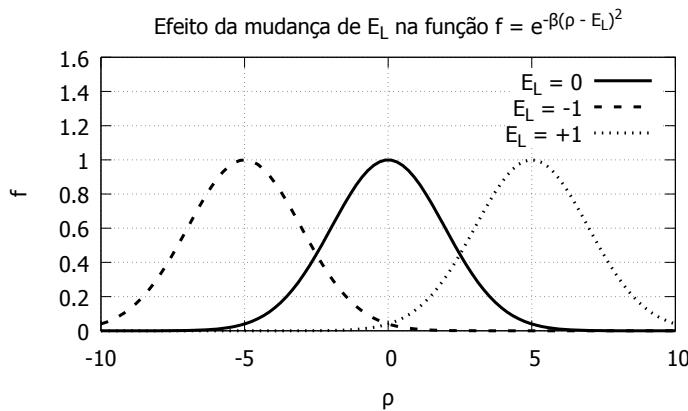


Figura 45 – Deslocamento do máximo de $f_i = e^{-\beta(\rho - E_L)^2}$

Efeito da mudança de β na função $f = e^{-\beta(\rho - E_L)^2}$

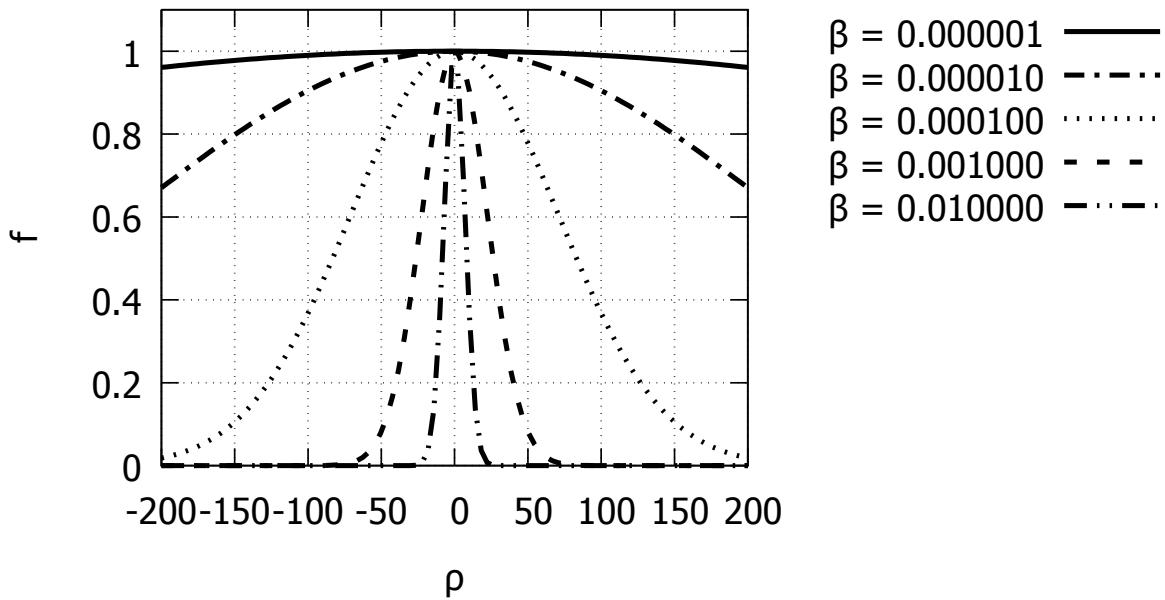
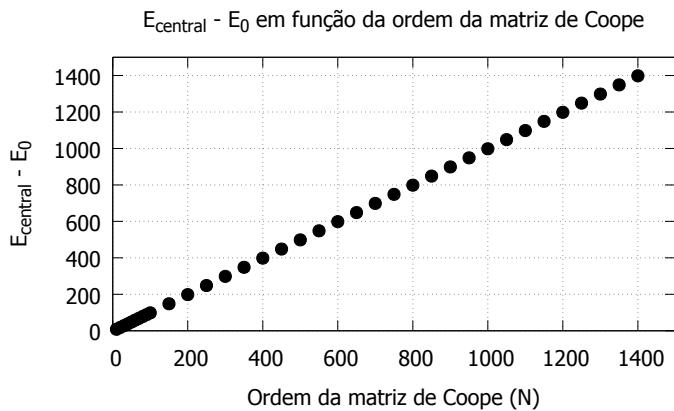


Figura 46 – Efeito da mudança de β em $f_i = e^{-\beta(\rho - E_L)^2}$

$\rho = 0,01$, próximo de E_L e, portanto, de boa qualidade. Assim, β deve ser escolhido para que a largura de f seja pequena.

Por outro lado, se a largura da função de avaliação for muito pequena, o mesmo tipo de problema acontecerá, mas para $f = 0$. Suponha que β_f tenha sido escolhido, e que na população inicial alguns indivíduos possuam ρ em torno de $\rho \approx 10$. Veja na figura 46 que todos eles terão nota $f = 0$, e certamente seriam preteridos em qualquer tipo de Seleção. Lembre-se que mesmo os piores indivíduos podem conter informação genética valiosa e, por isso, não devem ser sempre descartados (MITCHELL, 1998). Então, a escolha de β deve garantir também que, desde a primeira geração, os piores indivíduos

Figura 47 – $E_{central}$ é linear

tenham *fitness* maior que zero.

Para as matrizes de Coope utilizadas neste trabalho, verifiquei uma regularidade na população inicial. Em geral o Quociente de Rayleigh médio $\langle \rho \rangle$ ficava em torno dos autovalores centrais. Por exemplo, para matrizes com ordem $N = 10$, $\langle \rho \rangle \approx E_4 = 8,6285$ (veja tabela 10). Talvez a causa dessa regularidade esteja relacionada com a maneira como a matriz de Coope é definida (equação 5.1). Com essa informação, pude construir uma equação empírica para β .

A estratégia foi escolher um β que faça o *fitness* ser sempre muito pequeno, $f \approx 0 = 0,000001$, na região dos autovalores centrais, e que dependesse apenas das características da matriz. No parâmetro E_L usei E_0 , assim, a função de avaliação fica limitada ao intervalo $(0,1]$. Isolando β na equação 6.9 temos

$$\beta = -\frac{\ln f_i}{(\rho_i - E_L)^2}, \quad (6.10)$$

e incluindo os valores chega-se a

$$\beta = -\frac{\ln(0,000001)}{(E_{central} - E_0)^2}. \quad (6.11)$$

Os valores de $(E_{central} - E_0)$ foram calculados para matrizes de ordem $N = 10$ até $N = 1.400$, e estão na tabela 15. Observe na figura 47 que $E_{central} - E_0 = f(N)$ é claramente linear. A regressão dá-nos

$$E_{central} - E_0 = 1,00001N - 1,6507. \quad (6.12)$$

Utilizando a equação 6.12 na 6.11 cheguei finalmente a

$$\beta = -\frac{\ln(0,000001)}{(1,0001N - 1,6507)^2}. \quad (6.13)$$

Inseri um termo 0,65 de ajuste para que a equação 6.13 também fosse adequada para o *fitness* de (NANDY *et al.*, 2004):

$$\beta = -0,65 \frac{\ln(0,000001)}{(1,0001N - 1,6507)^2} \quad (6.14)$$

O uso dessa equação permitiu automatizar completamente os testes para matrizes de Coope, além de facilitar a comparação entre os *fitness* de (NANDY *et al.*, 2004) e (NANDY *et al.*, 2011). Todas as execuções apresentadas nesta dissertação utilizaram a equação 6.14.

Tabela 15 – Valores de $(E_{central} - E_0)$.

N	E_0	$E_{CENTRAL}$	$E_{CENTRAL} - E_0$
10	0,386075	8,628524	8,242449
20	0,341237	18,633845	18,292608
30	0,319737	28,635603	28,315866
40	0,306086	38,636479	38,330393
50	0,296280	48,637004	48,340724
60	0,288722	58,637353	58,348631
70	0,282625	68,637602	68,354977
80	0,277547	78,637789	78,360242
90	0,273215	88,637934	88,364719
100	0,269451	98,638050	98,368599
150	0,255873	148,638398	148,382525
200	0,247028	198,638572	198,391544
250	0,240570	248,638676	248,398106
300	0,235535	298,638746	298,403211
350	0,231435	348,638795	348,407360
400	0,227996	398,638833	398,410837
450	0,225044	448,638862	448,413818
500	0,222467	498,638885	498,416418
550	0,220184	548,638904	548,418720
600	0,218140	598,638919	598,420779
650	0,216293	648,638933	648,422640
700	0,214609	698,638944	698,424335
750	0,213065	748,638954	748,425889
800	0,211640	798,638963	798,427323
850	0,210318	848,638970	848,428652
900	0,209087	898,638977	898,429890
950	0,207935	948,638983	948,431048
1000	0,206853	998,638989	998,432136
1050	0,205835	1048,638994	1048,433159
1100	0,204873	1098,638998	1098,434125
1150	0,203962	1148,639002	1148,435040
1200	0,203098	1198,639006	1198,435908
1250	0,202275	1248,639010	1248,436735
1300	0,201491	1298,639013	1298,437522
1350	0,200742	1348,639016	1348,438274
1400	0,200025	1398,639019	1398,438994

7 Conclusões

O método apresentado nos artigos ([NANDY et al., 2004](#)) e ([NANDY et al., 2011](#)) é eficaz para obter autovalores de matrizes simétricas pequenas ($n \leq 40$). Seus resultados estão de acordo com a construção das funções objetivo.

As hipóteses levantadas na Introdução estão confirmadas. Como é possível encontrar o autovalor mínimo com o *fitness* de ([NANDY et al., 2011](#)), a segunda hipótese é verdadeira, portanto, não há problemas fundamentais no método. Consequentemente, a primeira hipótese também é correta, e concluo que é impossível usar o *fitness* de ([NANDY et al., 2004](#)) para obter o autovalor mínimo. Há base para contradizer, parcialmente, ([NANDY et al., 2004](#)).

As duas funções de avaliação podem ser utilizadas em conjunto. A de ([NANDY et al., 2011](#)) encontra o autovalor mínimo, porém, é necessário conhecimento prévio sobre a região onde ele se encontra. O *fitness* de ([NANDY et al., 2004](#)) é adequado para identificar autovalores intermediários, e essa informação pode ser útil para auxiliar na definição daquela região.

A configuração do parâmetro β merece cuidado. Ele deve ser escolhido de modo que a função de avaliação seja estreita, mas garantindo que os piores indivíduos da população inicial tenham *fitness* maior do que zero.

Referências

- ABREU, N.; DEL-VECCHIO, R.; TREVISAN, V.; VINAGRE, C. Teoria espectral de grafos – uma introdução. 2014. Disponível em: <<http://www.sbm.org.br/docs/coloquios/SU3-06.pdf>>.
- CANTU-PAZ, E. *Efficient and Accurate Parallel Genetic Algorithms*. [S.l.]: Springer, 2000.
- COOPE, J. A. R.; SABO, D. W. A new approach to the determination of several eigenvectors of a large hermitian matrix. *Journal of Computational Physics*, v. 23, p. 404–424, 1977.
- DEBATTISTI, S.; MARLAT, N.; MUSSI, L.; CAGNONI, S. Implementation of a simple genetic algorithm within the cuda architecture. *GPUs for Genetic and Evolutionary Computation, CECCO2009*, 2009. Disponível em: <<http://www.gpgpu.com/gecco2009/3.pdf>>.
- EDIGER, D.; JIANG, K.; RIEDY, J.; BADER, D.; CORLEY, C.; FARBER, R.; REYNOLDS, W. *Massive Social Network Analysis: Mining Twitter for Social Good*. [S.l.], 2010. Disponível em: <<http://www.cc.gatech.edu/~bader/papers/MassiveTwitter.html>>.
- EIDINI, M.; PAULINO, G. Unraveling metamaterial properties in zigzag-base folded sheets. *Science Advances*, AAAS – American Association for the Advancement of Science, September 2015. Disponível em: <<http://advances.sciencemag.org/content/1/8/e1500224.full>>.
- GOLUB, G.; VORST, H. van der. Eigenvalue computation in the 20th century. *Journal of Computational and Applied Mathematics*, Elsevier, p. 25–65, November 2000. Disponível em: <<http://www.sciencedirect.com/science/article/pii/S0377042700004131>>.
- HAEGEMAN, J.; ZAUNER, V.; SCHUCH, N.; VERSTRAETE, F. Shadows of anyons and the entanglement structure of topological phases. *Scientific Reports*, Nature Publishing Group, a division of Macmillan Publishers Limited. All Rights Reserved, October 2015. Disponível em: <<http://dx.doi.org/10.1038/ncomms9284>>.
- HAWKINS, T. Cauchy and the spectral theory of matrices. *Historia Mathematica*, Elsevier, v. 2, n. 1, p. 1–29, February 1975. Disponível em: <<http://www.sciencedirect.com/science/article/pii/0315086075900324>>.
- HONG, J.-H.; CHO, S.-B. Evolution of emergent behaviors for shooting game characters in robocode. *Proceedings of the 2004 IEEE Congress on Evolutionary Computation*, IEEE, 2004.
- KUANG, Q.; XU, X.; LI, R.; DONG, Y.; LI, Y.; HUANG, Z.; LI, Y.; LI, M. An eigenvalue transformation technique for predicting drug-target interaction. *Scientific Reports*, Macmillan Publishers Limited, September 2015. Disponível em: <<http://www.nature.com/articles/srep13867>>.

- LINDEN, R. *Algoritmos Genéticos. Uma importante ferramenta da Inteligência Computacional.* [S.l.]: BRASPORT, 2008.
- LUO, J.; PAN, Y.; XU, L.; GRISHAM, M.; ZHANG, H.; QUE, Y. Rational regional distribution of sugarcane cultivars in china. *Scientific Reports*, Macmillan Publishers Limited, October 2015. Disponível em: <<http://www.nature.com/articles/srep15721>>.
- MITCHELL, M. *An Introduction to Genetic Algorithms.* [S.l.]: MIT Press, 1998.
- NANDY, S.; CHAUDHURY, P.; BHATTACHARYYA, S. P. Stochastic diagonalization of hamiltonian: A genetic algorithm-based approach. *International Journal of Quantum Chemistry*, Wiley Periodicals, Inc, v. 90, p. 188–194, 2002.
- NANDY, S.; CHAUDHURY, P.; BHATTACHARYYA, S. P. Diagonalization of a real-symmetric hamiltonian by genetic algorithm: A recipe based on minimization of rayleigh quotient. *J. Chem. Sci*, Indian Academy of Sciences, v. 116, n. 5, p. 285–291, September 2004.
- NANDY, S.; CHAUDHURY, P.; BHATTACHARYYA, S. P. Workability of a genetic algorithm driven sequential search for eigenvalues and eigenvectors of a hamiltonian with or without basis optimization. In: YU, W.; SANCHEZ, E. (Ed.). *Advances in Computational Intelligence*. Berlin: Springer-Verlag, 2009. p. 259–268.
- NANDY, S.; SHARMA, R.; BHATTACHARYYA, S. P. Solving symmetric eigenvalue problem via genetic algorithms: Serial versus parallel implementation. *Applied Soft Computing*, Elsevier, v. 11, p. 3946–3961, 2011.
- NEWMAN, M. The structure and function of complex networks. March 2003. Disponível em: <<http://arxiv.org/pdf/cond-mat/0303516.pdf>>.
- PAGE, L.; BRIN, S. *The PageRank Citation Ranking: Bringing Order to the Web.* East Lansing, Michigan, 1998. 17 p. Disponível em: <<http://ilpubs.stanford.edu:8090/422/1/1999-66.pdf>>.
- PARLETT, D. N. *The Symmetric Eigenvalue Problem.* 2. ed. Philadelphia, USA: SIAM - Society for Industrial and Applied Mathematics, 1998. (Classics in Applied Mathematics).
- PRIETO, A. B. Robocode com algoritmos genéticos. *Disciplina FT011 Inteligência Artificial*, Faculdade de Tecnologia da Unicamp, Dezembro 2010.
- PRIETO, A. B. Diagonalização de matrizes hermitianas por meio de algoritmos genéticos paralelizados em gpus. *Exame de Qualificação de Mestrado*, Faculdade de Tecnologia da Unicamp, Novembro 2012.
- PRIETO, A. B.; COLUCI, V. R. Aceleração de desempenho de algoritmos genéticos com cuda. *III Escola Regional de Alto Desempenho de São Paulo*, ERAD2012, 2012.
- SHARMA, R.; NANDY, S.; BHATTACHARYYA, S. P. On solving energy-dependent partitioned eigenvalue problem by genetic algorithm: The case of real symmetric hamiltonian matrices. *PRAMANA Journal of Physics*, Indian Academy of Sciences, v. 66, n. 6, p. 1125–1130, June 2006.

SHARMA, R.; NANDY, S.; BHATTACHARYYA, S. P. On solving energy-dependent partitioned real symmetric matrix eigenvalue problem by a parallel genetic algorithm. *Journal of Theoretical and Computational Chemistry*, World Scientific Publishing Company, v. 7, n. 6, p. 1103–1120, 2008.

WILKINSON, J. H. *The Algebraic Eigenvalue Problem*. Clarendon, Oxford: [s.n.], 1965.

Apêndices

APÊNDICE A – Autovalores do século XVIII ao XXI

A primeira aparição do que hoje é chamado de autovalor aconteceu em 1743 ([HAWKINS, 1975](#)). Estudando o problema de várias massas ligadas umas às outras por molas, D'Alembert chegou a um sistema de equações diferenciais. Ao fazer algumas transformações de variáveis ele foi capaz de reduzir o estudo a apenas uma equação:

$$\frac{d^2u}{dt^2} + \lambda u = 0, \quad (\text{A.1})$$

sendo u uma soma envolvendo o produto das velocidades e posições de cada massa, e λ um escalar. D'Alembert aplicou o novo método a sistemas com duas e três massas ($n = 2$ ou $n = 3$) e, com argumentos relacionados à Física do problema, afirmou que λ só poderia ser real. A partir de então, diversos matemáticos se dedicaram ao assunto.

Na metade do século XVIII, D'Alembert, aproveitando trabalho anterior de Euler, demonstrou que as soluções gerais da equação A.1 são da forma $ge^{-\lambda t}$, g sendo um escalar, e que λ está associado com a estabilidade do sistema massa–mola. Lagrange estende a solução para n massas e escreve a equação polinomial característica, mas naquele tempo nada se sabia sobre a natureza de suas raízes. Em 1775 ele aplica seu método para a rotação de corpos rígidos desenvolvida por Euler dez anos antes, e é a primeira vez que autovalores são utilizados fora do contexto massa–mola. Em seguida, em 1778, o mesmo Lagrange mostra que a mecânica celestial pode ser escrita como um sistema de equações diferenciais, e conclui que λ está ligado à natureza das órbitas e à estabilidade do Sistema Solar.

Entra em cena Laplace, descobrindo em 1784 que λ depende *apenas* dos coeficientes A_{ij} envolvidos nos sistemas de equações diferenciais. Quatro anos depois mostra que um sistema discreto de massas próximo do equilíbrio pode ser escrito como

$$\mathbf{B}\mathbf{X} = \lambda \mathbf{A}\mathbf{X}, \quad (\text{A.2})$$

com as matrizes \mathbf{B} e \mathbf{A} ligadas, respectivamente, à Energia Potencial e Energia Cinética do sistema. Embasado na Conservação da Energia, argumenta que os autovalores λ são reais, positivos e distintos. Finalmente, em 1789, Laplace percebe as simetrias envolvidas para construir o primeiro teorema, completo e com demonstração, da natureza dos autovalores.

A partir do século XIX o problema dos autovalores e autovetores começa a tomar a forma que conhecemos hoje. Cauchy desenvolve em 1815 a Teoria dos Deter-

minantes e em 1829 prova, com argumentos puramente matemáticos, que os autovalores de uma matriz simétrica são reais. Matrizes simétricas são quadradas, com elementos a_{ij} reais, e $a_{ij} = a_{ji}$ para $i \neq j$. Nesse mesmo ano Sturm usa autovalores na Condução de Calor, levando as aplicações para além da Mecânica Clássica. Em 1839 Cauchy cunha o termo “Equação Característica”. Em torno de 1855 os resultados obtidos por Cauchy tornam-se “matemática básica” entre os matemáticos da época.

No artigo (GOLUB; VORST, 2000) há uma revisão sobre o desenvolvimento do cálculo de autovalores no século XX¹. Impulsionado pelo advento do computador eletrônico na década de 1950, o alvo desse desenvolvimento foi a criação de métodos numéricos com convergência rápida e resultados precisos. Esponho aqui os dois tipos principais, os de potência e os que reduzem a matriz principal a uma forma mais eficiente.

Os Métodos de Potência (*Power Methods*) são mais simples. A ideia é multiplicar a matriz A repetidas vezes por um vetor inicial x bem escolhido, de modo que um de seus componentes, o que está na direção do autovetor associado ao maior autovalor em valor absoluto, é aumentado em relação aos outros componentes. Assim, obtém-se o maior autovalor. Uma variação mais efetiva é o Método da Potência Inverso (*Inverse Power Method*), que trabalha com a matriz $(A - \mu I)^{-1}$, onde μ é um valor de deslocamento em torno de A a cada iteração. Tais algoritmos não são mais competitivos, mas continuam sendo estudados pois formam a base de métodos modernos.

Um deles é o Método da Iteração do Quociente de Rayleigh (*Rayleigh Quotient Iteration*). Inspirado num algoritmo utilizado por Lord Rayleigh em 1870, usa um quociente de Rayleigh (equação 2.2 do capítulo 2) para o deslocamento μ . O atual é muito rápido, e possui convergência cúbica [$O(n^3)$].

Com relação aos métodos de redução, todos partem da ideia central que matrizes podem ser reduzidas a uma forma mais eficiente para as computações subsequentes, utilizando um número finito de passos em transformações ortogonais. Por exemplo, foi possível aproveitar a seguinte propriedade das matrizes simétricas: para qualquer matriz simétrica A sempre existe uma matriz Q de modo a fazer uma transformação do tipo $Q^\dagger A Q = D$, em que Q^\dagger é a transposta de Q , D é diagonal e seus elementos são os autovalores de A . O Método de Jacobi, desenvolvido em 1846, faz isso por meio de uma série de rotações. Originalmente não garantia convergência, problema que foi corrigido apenas em 1949.

Em 1931 Kyrlov sugeriu um método baseado no fato de que toda matriz satisfaz sua equação (ou polinômio) característica(o). Ele usou os vetores x , Ax , A^2x (...) gerados pelo método da potência para determinar os coeficientes dessa equação. A técnica não foi bem aceita porque era instável, pois pequenas modificações em A levam a grandes

¹ É importante salientar que o problema de autovalores e autovetores foi fundamental em uma das grandes revoluções científicas e culturais da nossa era, a Mecânica Quântica.

mudanças nos coeficientes do polinômio. Entretanto, ele teve sua importância pois inspirou os famosos métodos de Householder e Lanczos. O último, por exemplo, a partir de 1980 era o preferido para grandes matrizes simétricas e esparsas (com muitos zeros).

De acordo com ([GOLUB; VORST, 2000](#)), o Método QR era um dos mais populares e mais poderosos do ano 2000. Ele é capaz de calcular *todos* os autovalores e autovetores de uma matriz simétrica e densa (não esparsa), sempre com convergência cúbica [$O(n^3)$].

Porém, por volta de 1970 o rumo da pesquisa na área mudou. Naquela época o problema padrão para o cálculo numérico de autovalores (equação [1.3](#)) foi visto como essencialmente resolvido para matrizes não muito grandes ($n \leq 25$). Então, além de tratar problemas generalizados e, consequentemente, mais complexos, o interesse voltou-se para matrizes maiores.

Em 1981 Cuppen apresenta o primeiro algoritmo paralelo para matrizes tridiagonais de tamanho moderado, com $n > 25$ e menor do que alguns milhares. Da classe de algoritmos do tipo “Divida e Conquiste”, a ideia foi dividir a matriz original em dois blocos com metade do tamanho original, além de gerar uma matriz que ele chamou de Matriz de Atualização. Cuppen mostrou como o problema de autovalores para cada um dos blocos poderia ser combinado para resolver o problema principal, e reconheceu que seu algoritmo era assintoticamente muito mais rápido que o QR. Novamente, problemas de instabilidade, principalmente relacionados aos autovetores de autovalores próximos, fizeram com que o método não fosse considerado competitivo para matrizes pequenas. Entretanto, ele continuou a ser desenvolvido pois apresentava propriedades paralelas interessantes. Após uma correção publicada em 1995 o método foi aceito pela comunidade.

O século XX chega ao seu fim com *software* consolidado, seja em forma de bibliotecas para uso de programadores, seja em ambientes numéricos comerciais e de código aberto. A biblioteca LINPACK cobriu soluções numéricas para sistemas lineares, enquanto a EISPACK se concentrou nos problemas de autovalores. A EISPACK foi substituída em 1995 pela LAPACK, que possui uma versão paralela, ScaLAPACK, cuja meta é fornecer *software* para arquiteturas paralelas modernas. O ambiente MATLAB, comercial, está no estado da arte da computação para álgebra linear numérica, e tornou-se padrão na década de 1990. Boas alternativas não comerciais estão disponíveis, como o Octave e o SciLab.

Autovalores continuam importantes no século XXI. A busca pela palavra *eigenvalue* em periódicos como *Nature* e *Science* leva a vários artigos em inúmeras áreas diferentes. Restringindo a pesquisa apenas ao ano de 2015, encontramos autovalores na descoberta de novos fármacos ([KUANG et al., 2015](#)), cultivo de cana de açúcar na China ([LUO et al., 2015](#)), física teórica ([HAEGEMAN et al., 2015](#)) e ciência de materiais ([ELIDINI; PAULINO, 2015](#)). No jornal PLOS ONE é possível navegar por artigos associados

especificamente à palavra-chave *eigenvalue*².

Mas o destaque não está limitado apenas à ciência. O algoritmo *PageRank*, base do mecanismo de busca do Google, tem em seu núcleo uma formulação do problema de autovalores e autovetores (PAGE; BRIN, 1998). Em uma versão simplificada, define-se uma matriz quadrada \mathbf{A} de modo que suas linhas e colunas representam páginas da *Web*. Os elementos $A_{u,v}$ são definidos de tal maneira que, se não houver um *hyperlink* entre u e v , $A_{u,v} = 0$, caso contrário, $A_{u,v}$ é inversamente proporcional ao número total de *hyperlinks* que u possui apontando para quaisquer outras páginas (uma característica, então, que depende apenas de u). A relevância das páginas (*rank*) é definida como

$$\mathbf{R} = c\mathbf{AR}, \quad (\text{A.3})$$

onde \mathbf{R} é o autovetor de \mathbf{A} com autovalor associado c . O objetivo é encontrar o autovetor dominante, ou seja, aquele associado ao autovalor de maior valor absoluto. Ele terá as informações da ordem de relevância das páginas associadas à busca, da mais relevante para a menos. Ou seja, a ordem das páginas exibidas em uma busca no Google é a expressão direta de \mathbf{R} na equação acima.

Outra aplicação fundamental dos autovalores na atualidade está presente na Teoria Espectral dos Grafos, que “*busca analisar propriedades estruturais de grafos através de matrizes e seus espectros, ou seja, dos autovalores das matrizes associadas a eles*” (ABREU *et al.*, 2014). Um grafo (ou rede) é um conjunto de itens, chamados de vértices ou nós, com conexões entre eles, chamadas de arestas. Na figura 48 há um exemplo. Há várias matrizes associadas a um grafo, e tem-se descoberto que seus autovalores trazem informações importantes sobre a estrutura da rede.

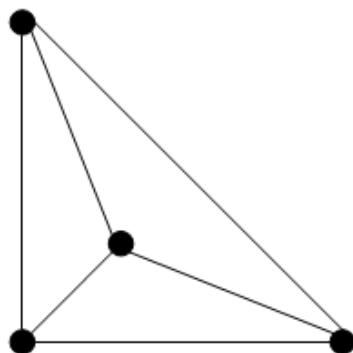


Figura 48 – Exemplo de um grafo. Fonte: Wikipedia.

Vários sistemas tomam a forma de redes, como a *Web* e as Redes Sociais digitais (NEWMAN, 2003). No caso do Facebook e Twitter, por exemplo, as redes são

² <http://www.plosone.org/browse/eigenvalues>

enormes, atingindo facilmente centenas de milhões de nós (usuários), levando a matrizes de dimensão equivalente a essa ordem de grandeza (EDIGER *et al.*, 2010). Nesses casos, extrair informações estruturais por meio dos seus autovalores é uma tarefa desafiadora.

Então, acredito que a pesquisa teórica e computacional, assim como das aplicações dos autovalores, continuarão ativas por um bom tempo.

APÊNDICE B – Resultados preliminares na GPU

O GA aqui desenvolvido buscou a solução do problema ONEMAX, cujo objetivo é encontrar uma sequência de N bits com a maior quantidade possível de “1” a partir de uma sequência aleatória de “1” e “0”. O ONEMAX é especialmente indicado para o início dos estudos em GA. Além de permitir simples implementação, possui representação cromossomial binária que, junto com o crossover de ponto único, forma a base da teoria original de Holland (LINDEN, 2008).

O programa paralelizado foi uma tradução literal do seu equivalente serial para a sintaxe do CUDA C. Ou seja, não houve nenhuma mudança estrutural no código, seja nas variáveis e estruturas de dados, seja na ordem de execução das funções e procedimentos. Apenas o preenchimento aleatório da população inicial é executado na CPU, de forma que o núcleo do programa é executado inteiramente na GPU (DEBATTISTI *et al.*, 2009).

A população do GA era constituída por indivíduos formados por cromossomos com *numGenes* elementos do tipo `char`. Cada elemento do vetor (gene) podia ter um valor “1” ou “0”. Dentro do problema ONEMAX, os melhores indivíduos foram os que apresentaram maior número de genes iguais a “1”.

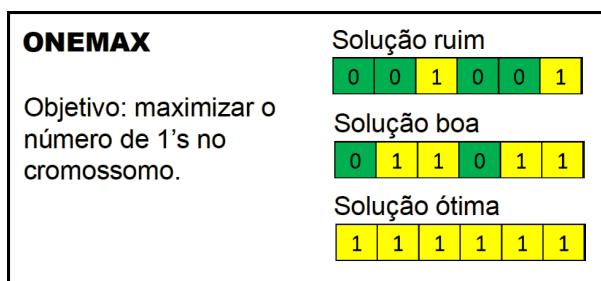


Figura 49 – ONEMAX, um problema clássico nos Algoritmos Genéticos.

Implementei um *kernel* (função que tem sua execução feita pela GPU) para cada um dos quatro passos do GA: cálculo da função avaliadora (*fitness*), seleção, *crossover* e mutação. Eles são chamados um após o outro até que um número máximo de gerações seja atingido. Isso é feito dentro do *loop* principal (realizado na CPU), mas sem troca de informação entre CPU e GPU.

No início do programa duas gerações são alocadas na memória global da GPU, as quais são usadas alternadamente como *input* e *output* dos kernels. Apenas as chamadas dos kernels acontecem na CPU, enquanto o restante (execução + dados) está na GPU.

Todos os *kernels* tinham como *input* e *output* uma estrutura do tipo Geração. Ou seja, as funções operaram sobre toda a população do GA, levando-nos a adotar como estratégia o paralelismo no nível dos indivíduos.

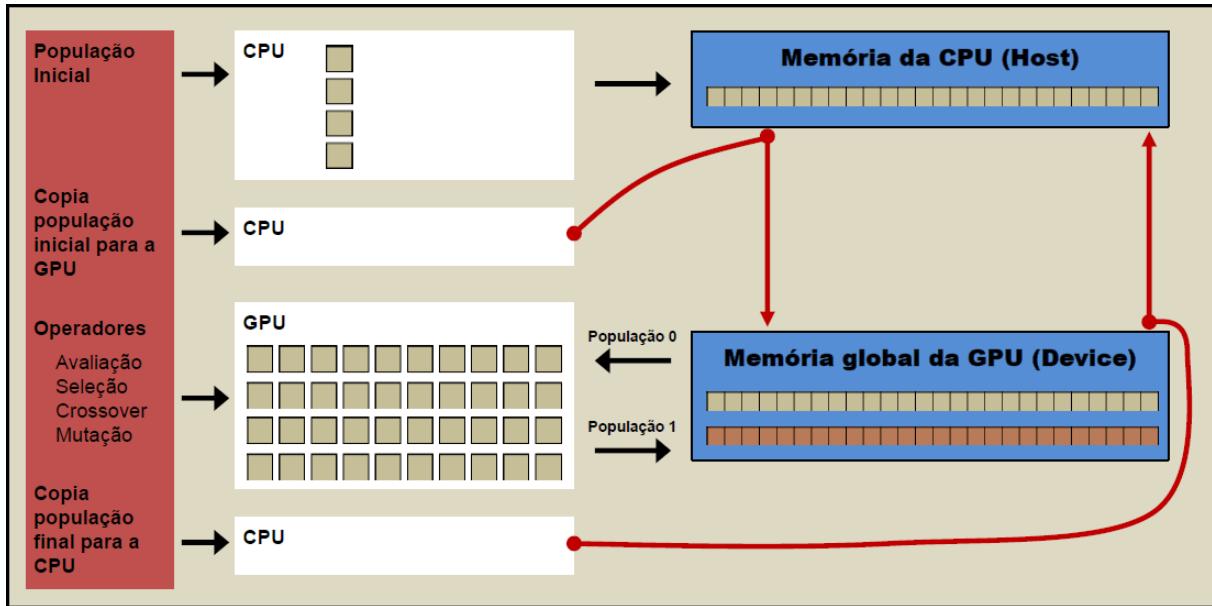


Figura 50 – Execução do ONEMAX paralelo. Apenas as chamadas dos kernels acontecem na CPU, enquanto o restante (execução + dados) está na GPU.

No cálculo do *fitness*, o *input* foi uma população com *numIndividuos* e o *output* uma população com os mesmos *numIndividuos* e suas respectivas notas. O cálculo do *fitness* de cada indivíduo foi realizado por meio da soma dos valores de seus genes. Por exemplo, para um indivíduo formado por um cromossomo de 6 genes (*numGenes* = 6) com a configuração “010101”, o valor do *fitness* é 3 e a solução ótima para este caso seria “111111” (figura 49). No código serial a programação é simples e envolve apenas um laço **for** que percorre o cromossomo e soma os bytes. Porém, note que toda informação necessária para esse cálculo está contida no próprio cromossomo, ou seja, a obtenção do *fitness* de um dado indivíduo não depende do restante da população. Assim, a paralelização do cálculo do *fitness* deu-se por meio da associação de uma *thread* para cada indivíduo.

Para o operador de seleção, optei pela seleção via torneio com o tamanho do torneio fixo e igual a dois. Novamente, o *input* e o *output* são populações. Na entrada há *numIndividuos* e suas notas. Os mais aptos (maiores *fitness*) têm maiores chances de serem selecionados, e compõem os *numIndividuos* da população na saída. Assim como no cálculo do *fitness*, o paralelismo acontece no nível dos indivíduos. Para cada indivíduo na população de saída há uma *thread*, que seleciona aleatoriamente dois cromossomos na população da entrada (memória global) e fica com o de maior *fitness*.

O *input* do crossover é a população resultante da seleção. Utilizei o crossover

de dois pontos, independentemente da quantidade de genes do cromossomo, com probabilidade $p_C = 90\%$. Na implementação serial, apenas um indivíduo é gerado ao término do *crossover*. Isso garantiu que, na versão paralela, a chamada da função de *crossover* fosse configurada com exatamente o mesmo número de *threads* dos operadores anteriores (avaliação e seleção): uma *thread* para cada indivíduo na população de saída, que recebe um cromossomo resultante do *crossover*.

Após o *crossover* todos os indivíduos passam por uma mutação simples, onde cada gene do cromossomo tem baixa probabilidade (0,01%) de ser invertido ($0 \rightarrow 1$ ou $1 \rightarrow 0$). Logo, semelhante ao cálculo do *fitness*, a mutação em um dado indivíduo é independente do restante da população. Mais uma vez, uma *thread* foi associada a cada *iIndividuo* cromossomo na saída, que recebe os genes modificados do *iIndividuo* na entrada.

Os experimentos foram executados em um laptop equipado com uma CPU Intel Core 2 Duo T6600 - 2,2 GHz. A placa de vídeo utilizada foi uma GeForce G 130M, com quatro multiprocessadores a 1,5 GHz e memória global total de 466 MB. A versão da API CUDA foi a 4.0, programada com o Microsoft Visual C++ Express 2008.

A placa G 130M possui capacidade de computação 1.1 (que indica a versão do hardware de computação presente na GPU). Comparada com a primeira versão (arquitetura original da G80), ela adiciona suporte à operações na memória global que permitem que múltiplas *threads* executem, sem conflito, operações ler-modificar-escrever na memória. Como o suporte às operações de ponto flutuante com precisão dupla só foi disponibilizado na versão 1.3, tanto o programa serial quanto o paralelo utilizaram precisão simples.

As medidas de desempenho foram feitas com o objetivo de observar a influência de dois parâmetros do GA: i) número de indivíduos na população e ii) tamanho do cromossomo. O ganho na velocidade foi calculado como a razão entre o tempo de execução do programa serial e o tempo de execução do programa paralelo.

Verifiquei que o ganho de desempenho da versão paralela do GA cresce com o aumento do número de indivíduos (figura 51). O programa serial é mais rápido (ganho < 1) para populações pequenas (< 50). Porém, a partir de uma população de 50 indivíduos, o programa paralelo apresenta desempenho superior. Com 600 indivíduos, a versão paralela é oito vezes mais rápida para um cromossomo de tamanho 10, e dez vezes para um cromossomo de tamanho 300.

Ao analisar a influência do tamanho do cromossomo, verifiquei um comportamento aproximadamente constante do ganho (figura 52). Isso era esperado, pois a paralelização ocorreu no nível dos indivíduos e não no nível dos cromossomos. Com 600 indivíduos o ganho fica em torno de nove vezes para qualquer tamanho de cromossomo.

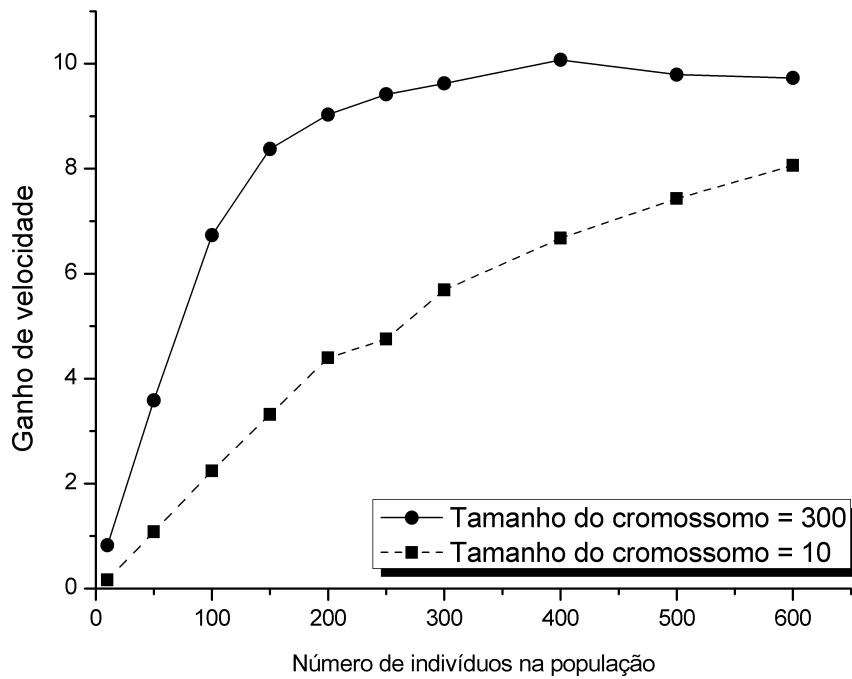


Figura 51 – ONEMAX paralelo. Ganho de velocidade em função do número de indivíduos da população.

O comportamento repete-se com uma população de 10 indivíduos, mas, nesse caso, o programa serial sempre é mais rápido, mesmo para cromossomos muito pequenos (ganho sempre < 1).

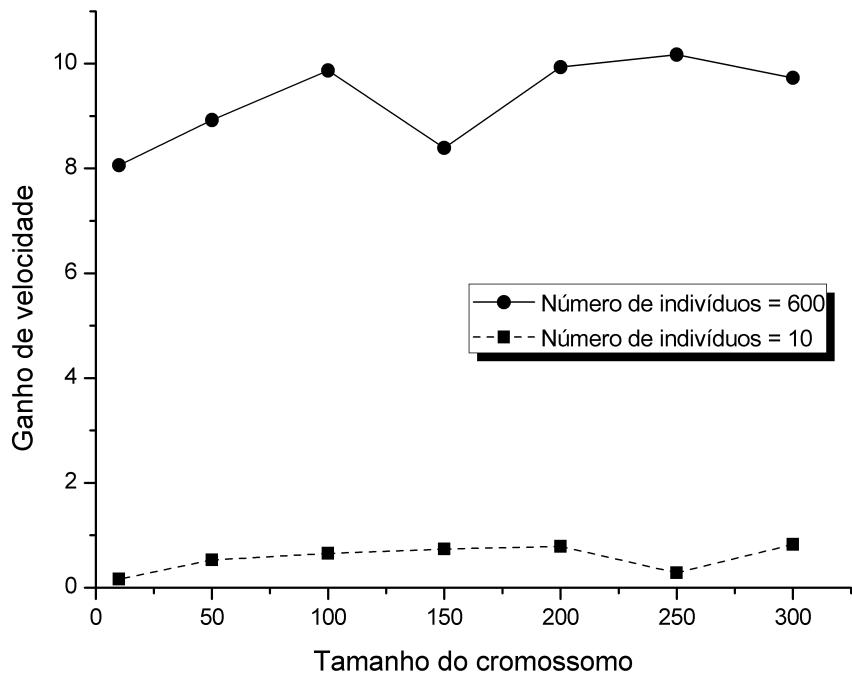


Figura 52 – ONEMAX paralelo. Ganho de velocidade em função do tamanho do cromossomo.