



Adriano Batista Prieto

Autovalores com Algoritmos Genéticos

Limeira

2015



UNIVERSIDADE ESTADUAL DE CAMPINAS
Faculdade de Tecnologia

Adriano Batista Prieto

Autovalores com Algoritmos Genéticos

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 tese defendida pelo aluno Adriano Batista Prieto, e orientada pelo Prof. Dr. Vitor Rafael Coluci

Limeira

2015

INCLUA AQUI O PDF COM A FICHA CATALOGRÁFICA FORNECIDA.

INCLUA AQUI A FOLHA DE APROVAÇÃO.

Dedico esta tese à todo mundo.

Agradecimentos

Agradecimentos aqui.

“Escreva aqui a sua epígrafe”
(Citação)

Resumo

Colocar o resumos aqui.

Palavras-chaves: palavra-chave 1; palavra-chave 2; palavra-chave 3.

Abstract

Put abstract here.

Keywords: keyword 1; keyword 2; keyword 3.

Listas de ilustrações

Figura 1 – Fluxograma do algoritmo genético típico.	16
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).	18
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 $y(x)$.	19
Figura 4 – Gráfico da função $y(x) = -x^2 + 36$ para $x = [-1, 1]$.	21
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.	23
Figura 6 – Exemplo de código em Linguagem C para o método da Roleta.	23
Figura 7 – Exemplo de função em Linguagem C que implementa a Seleção por Torneio.	26
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.	26
Figura 9 – Exemplo do <i>crossover</i> entre os indivíduos 011 e 101 para o primeiro ponto de corte.	28
Figura 10 – Código para a Reprodução. Nesse exemplo o algoritmo gera apenas um descendente. Detalhes da função <i>CrossOver()</i> estão na figura 11.	29
Figura 11 – Detalhes da função <i>CrossOver()</i> .	30
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 $y(x) = -x^2 + 36$.	31
Figura 13 – Exemplo de código para o operador Mutação.	32
Figura 14 – Representação esquemática da utilização de transistores na CPU e na GPU. Note que o número de transistores dedicados às operações matemáticas é muito superior, enquanto o controle de fluxo de dados e a memória <i>cache</i> estão naturalmente paralelizados. Retirado de (NVIDIA, 2011).	33
Figura 15 – Exemplo de código em CUDA C. A função <i>VecAdd</i> adiciona dois vetores A e B de ordem N e armazena o resultado no vetor C . Retirado de (NVIDIA, 2011).	34

Figura 16 – Estrutura hierárquica das <i>threads</i> . No lado esquerdo há uma representação da distribuição de blocos entre os núcleos. Esse processo, que é automático, permite a construção de programas altamente escaláveis. À direita está a disposição das <i>threads</i> dentro dos blocos. Retirado de (NVIDIA, 2011).	35
Figura 17 – Comportamento do <i>fitness</i> $f_i = e^{-\lambda \ \nabla \rho_i\ ^2}$ 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.	40
Figura 18 – Comportamento de ρ (Quociente de Rayleigh) para uma matriz de Coope–Sabo de ordem 10.	40
Figura 19 – Comportamento de ρ (Quociente de Rayleigh) para uma matriz de Coope–Sabo de ordem 10.	41
Figura 20 – Comportamento do <i>fitness</i> para as execuções zero do Hamiltoniano de ordem 10, semente 1445738835. A primeira usa o <i>fitness</i> $f_i = e^{-\lambda(\rho_i - E_L)^2}$, que chega ao autovalor mínimo, enquanto a segunda utiliza o $f_i = e^{-\lambda \ \nabla \rho_i\ ^2}$.	48
Figura 21 – Comportamento do ρ para as execuções zero do Hamiltoniano de ordem 10, semente 1445738835. A primeira usa o <i>fitness</i> $f_i = e^{-\lambda(\rho_i - E_L)^2}$, que chega ao autovalor mínimo, enquanto a segunda utiliza o $f_i = e^{-\lambda \ \nabla \rho_i\ ^2}$.	49
Figura 22 – Execução para a semente 1445738835. E_L um pouco acima de E_0 no <i>fitness</i> $f_i = e^{-\lambda(\rho_i - E_L)^2}$.	49
Figura 23 – Execução para a semente 1445738835. E_L muito acima de E_0 no <i>fitness</i> $f_i = e^{-\lambda(\rho_i - E_L)^2}$.	50
Figura 24 – Execução para a semente 1445738835. E_L muito abaixo de E_0 no <i>fitness</i> $f_i = e^{-\lambda(\rho_i - E_L)^2}$. Até geração 500.	50
Figura 25 – Execução para a semente 1445738835. E_L muito abaixo de E_0 no <i>fitness</i> $f_i = e^{-\lambda(\rho_i - E_L)^2}$. Geração entre 30.000 e 40.000.	51
Figura 26 – ONEMAX, um problema clássico nos Algoritmos Genéticos.	55
Figura 27 – Execução do ONEMAX paralelo. Apenas as chamadas dos kernels acontecem na CPU, enquanto o restante (execução + dados) está na GPU.	56
Figura 28 – ONEMAX paralelo. Ganho de velocidade em função do número de indivíduos da população.	58
Figura 29 – ONEMAX paralelo. Ganho de velocidade em função do tamanho do cromossomo.	58
Figura 30 – Execuções $N = 10$.	65
Figura 31 – Execuções $N = 20$.	66
Figura 32 – Execuções $N = 30$.	67
Figura 33 – Execuções $N = 40$.	68
Figura 34 – Execuções para $N = 10$ com o <i>fitness</i> $f_i = e^{-\lambda(\rho_i - E_L)^2}$.	70

Figura 35 – Execuções para N = 20 com o fitness $f_i = e^{-\lambda(\rho_i - E_L)^2}$.	71
Figura 36 – Execuções para N = 30 com o fitness $f_i = e^{-\lambda(\rho_i - E_L)^2}$.	72
Figura 37 – Execuções para N = 40 com o fitness $f_i = e^{-\lambda(\rho_i - E_L)^2}$.	73
Figura 38 – Execuções com o E_L um pouco acima de E_0 no fitness $f_i = e^{-\lambda(\rho_i - E_L)^2}$. Semente 1445738835, N = 10.	75
Figura 39 – Execuções com o E_L um pouco abaixo de E_0 no fitness $f_i = e^{-\lambda(\rho_i - E_L)^2}$. Semente 1445738835, N = 10.	76
Figura 40 – Execuções com o E_L muito acima de E_0 no fitness $f_i = e^{-\lambda(\rho_i - E_L)^2}$. Semente 1445738835, N = 10.	77
Figura 41 – Execuções com o E_L muito abaixo de E_0 no fitness $f_i = e^{-\lambda(\rho_i - E_L)^2}$. Semente 1445738835, N = 10.	78

Lista de tabelas

Tabela 1 – Sistemas Naturais x Sistemas Artificiais	15
Tabela 2 – Exemplo de representação cromossomial	17
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.	20
Tabela 4 – Representação cromossomial para os pontos $x = 0$ até $x = 7$ dentro do problema de máximo da função $y(x) = -x^2 + 36$	21
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.	21
Tabela 6 – Todas as notas dos indivíduos para o exemplo da seção 3.6.1. Lembre-se que para obter 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).	22
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.	25
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.	29
Tabela 9 – Representação cromossomial para os indivíduos que passaram pela Seleção e pelo <i>Crossover</i>	30
Tabela 10 – Execuções para matrizes de Coope–Sabo.	45
Tabela 11 – Execuções novo <i>Fitness</i>	47
Tabela 12 – Variando E_L para a execução da semente 1445738835. Os tipos de teste são: tipo 1 : E_L um pouco acima de E_0 ; tipo 2 : E_L um pouco abaixo de E_0 ; tipo 3 : E_L muito acima de E_0 ; tipo 4 : E_L muito abaixo de E_0	51
Tabela 13 – Cinco execuções para cada tipo de teste de variação de E_L em torno de E_0 no fitness $f_i = e^{-\lambda(\rho_i - E_L)^2}$	52
Tabela 14 – Lista de autovalores para matrizes de Coope–Sabo de ordem 10, 20, 30 e 40.	63

Sumário

1	Introdução	12
1.1	Autovalores e o Quociente de Rayleigh	12
1.2	Algoritmos Genéticos	12
1.2.1	ONEMAX	12
1.2.2	Robocode com GA	12
1.3	NVidia CUDA	12
1.4	Objetivos	12
2	Autovalores e o Quociente de Rayleigh	13
3	Algoritmos Genéticos	14
3.1	Aspectos históricos	14
3.2	Terminologia	15
3.3	Fluxograma de um algoritmo genético simples	15
3.4	Representação Cromossomial	16
3.5	Função de Avaliação	17
3.5.1	Exemplo: Máximos de $y(x) = \sin(x)/x$	19
3.6	Seleção	20
3.6.1	Exemplo: Variabilidade Genética	20
3.6.2	O método da Roleta (<i>Roulette Wheel</i>)	22
3.6.3	Exemplo: máximo da função $y(x) = -x^2 + 36$	24
3.6.4	Seleção por torneio	25
3.7	Reprodução (<i>Crossover</i>)	25
3.7.0.1	Exemplo: crossover para $y(x) = -x^2 + 36$	27
3.8	Mutação	30
4	CUDA	33
5	Materiais e Métodos	36
5.1	CUDA	36
5.2	Método	36
5.3	Framework/Programa Serial	36
6	Resultados e discussão	38
6.1	Problemas com o mínimo global	39
6.2	Outro <i>fitness</i> para encontrar o mínimo global	46
6.3	Por que o λ deve ser escolhido cuidadosamente?	53
6.4	Equação empírica para o λ	53
6.5	A mistura de $(\rho - \rho_0)^2$ com $\nabla\rho$ não leva a melhores resultados	54
6.6	$f_i = e^{[-\lambda\nabla\rho]}$ é mais rápido do que $f_i = e^{[-\lambda(\nabla\rho)^2]}$	54
6.7	Resultados preliminares na GPU	54

Referências	59
Apêndices	61
APÊNDICE A Lista de autovalores	62
APÊNDICE B Execuções para o <i>fitness</i> $f_i = e^{-\lambda \nabla\rho ^2}$	64
APÊNDICE C Execuções para o <i>fitness</i> $f_i = e^{-\lambda(\rho_i - E_L)^2}$	69
APÊNDICE D Execuções para a variação de E_L em torno de E_0	74
Anexos	79
ANEXO A Título do Anexo X	80
ANEXO B Título do Anexo Y	81

1 Introdução

1.1 Autovalores e o Quociente de Rayleigh

1.2 Algoritmos Genéticos

1.2.1 ONEMAX

Problema 1: ONEMAX, considerado o “*Hello World*” dos algoritmos genéticos.

Definir objetivo do problema.

Representação cromossomial: zeros e uns.

Fitness: soma dos genes.

Seleção: por roleta

Crossover: ponto único

Mutação: simples

Gráfico do comportamento do fitness.

Listar última geração, mostrando que o algoritmo chegou na solução.

1.2.2 Robocode com GA

Resumir o relatório final da disciplina de Inteligência Artificial.

1.3 NVidia CUDA

1.4 Objetivos

2 Autovalores e o Quociente de Rayleigh

Inseir apenas a álgebra linear necessária para o entendimento do método e da discussão.

3 Algoritmos Genéticos

O objetivo desse capítulo é dar uma breve introdução aos Algoritmos Genéticos (GAs). Apesar dos GAs terem como base uma teoria matemática formal, o foco está nos aspectos práticos. Espero que o leitor que nunca teve contato com GA possa ter uma boa ideia de seu funcionamento utilizando pouco tempo. Os mais experientes em GA ou outras técnicas de Computação Evolutiva podem pular o capítulo sem prejuízo.

3.1 Aspectos históricos

Da-se o nome *Algoritmos Genéticos* (GAs) 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 Articiais

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.

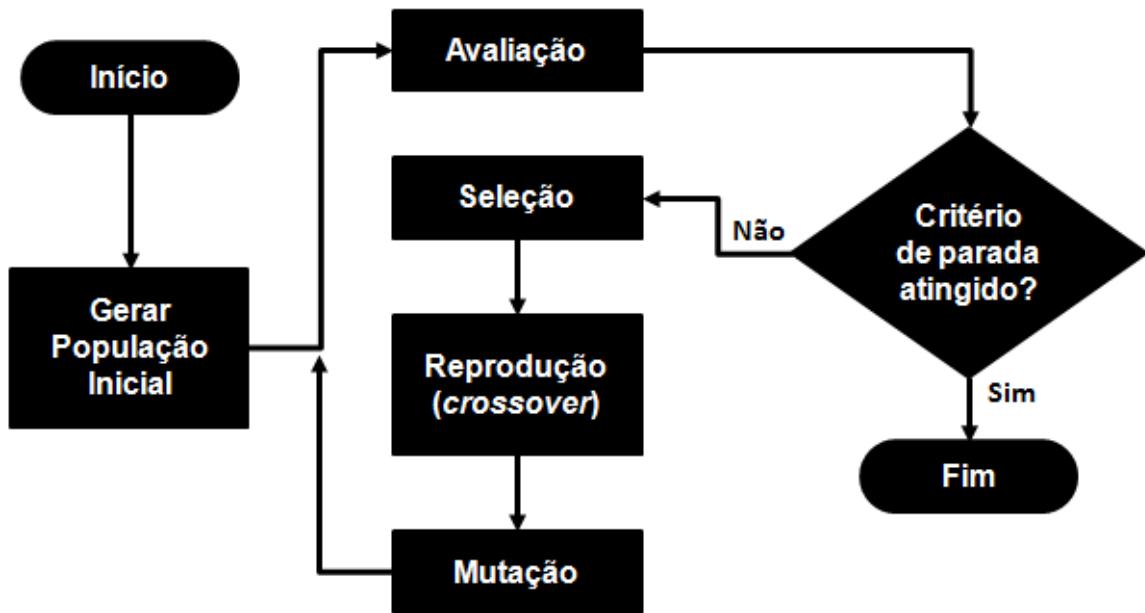


Figura 1 – Fluxograma do algoritmo genético típico.

3.4 Representação Cromossomial

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

Esta representação comprehende parte de grande importância num GA, pois é a partir desta 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 influí 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 esta estrutura será maximizada.

Tabela 2 – Exemplo de representação cromossomial

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

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

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 à 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 aquele 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, n é uma métrica que deve identificar o quão próximo um cromossomo 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

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

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 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.

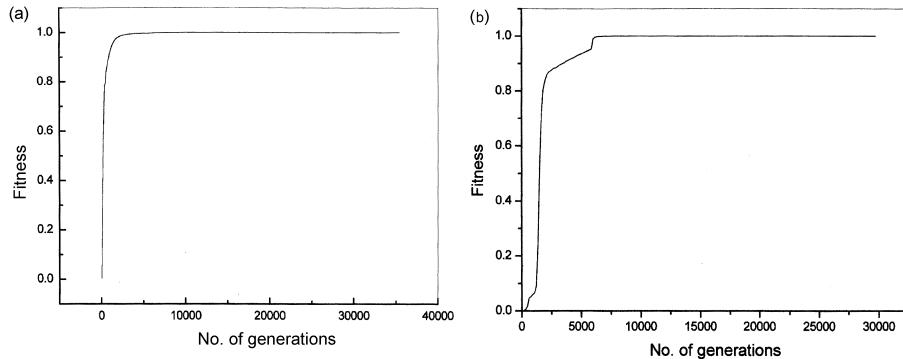


Figura 2 – Exemplo de instabilidade do *fitness*. Enquanto em (a) o *fitness* cresce de maneira contínua e depois se estabiliza, em (b) há alguns pontos onde o comportamento da nota sofre uma mudança razoavelmente brusca. Gráficos retirados de (NANDY *et al.*, 2004).

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?

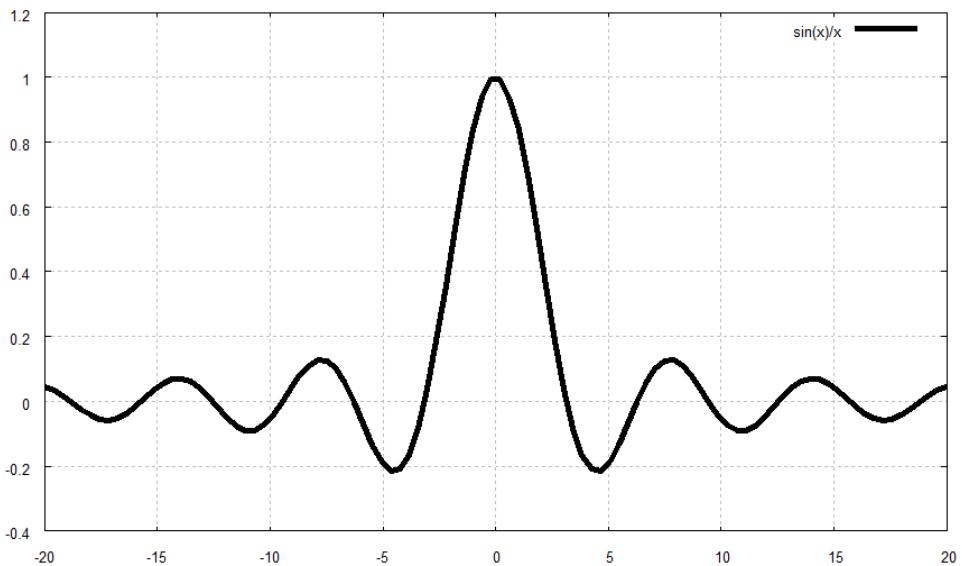


Figura 3 – Gráfico da função $y(x) = \sin(x)/x$ para $x = [-20, 20]$. Se desejamos encontrar os máximos (global ou locais), uma boa função de avaliação seria a própria $y(x)$.

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.

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

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

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

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).

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.

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

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

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

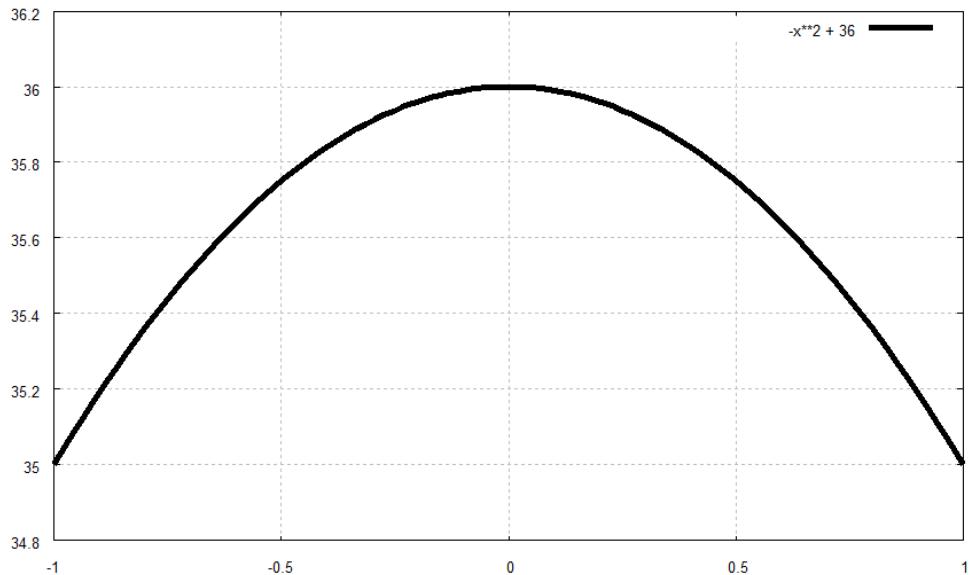


Figura 4 – Gráfico da função $y(x) = -x^2 + 36$ para $x = [-1, 1]$.

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

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 (*Roulette Wheel*)

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 à 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 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

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*).

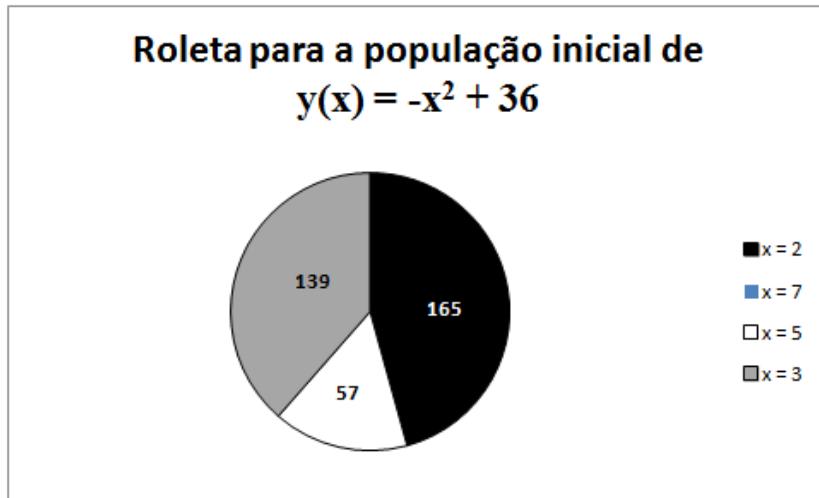


Figura 5 – Roleta criada a partir dos dados da tabela 6. Note que o indivíduo $x = 7$ foi descartado porque sua avaliação foi um número menor do que zero.

Essa soma também é exibida na tabela 6. A variável *iSeleccionado* 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 *iSeleccionado* 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     iSeleccionado = -1;
230     sumRoleta = 0;
231
232     do {
233         iSeleccionado++;
234         sumRoleta = sumRoleta + geracao[0].individuo[iSeleccionado].fitness;
235     } while (sumRoleta <= vlrRoleta);
236
237
238     geracao[1].individuo[iIndividuo] = geracao[0].individuo[iSeleccionado];
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 o esperado. Ele não ocupou espaço na roleta e, obviamente, não poderia participar do processo de seleção. Além disso, o *fitness* médio ($\langle f \rangle$) 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 disso 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 vencer e prosseguir. Um superindivíduo, aquele com *fitness* muito maior que a média da população, não é privilegiado (LINDEN, 2008). O torneio pode ser facilmente paralelizado.

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).

```

//=====
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.

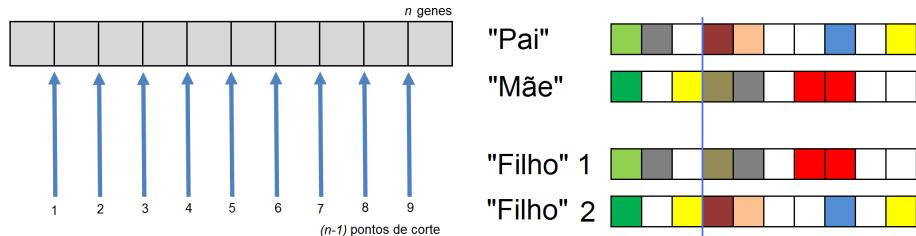


Figura 8 – Definição de Ponto de Corte e o Crossover de ponto único. Com esse operador conseguimos gerar até dois filhos para cada par de pais.

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*:

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

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%.

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

$$PontoDeCorte = 1,$$

e que

$$probAux = 0,86.$$

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

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 3⁵, e isso implica

$$indMae = 101.$$

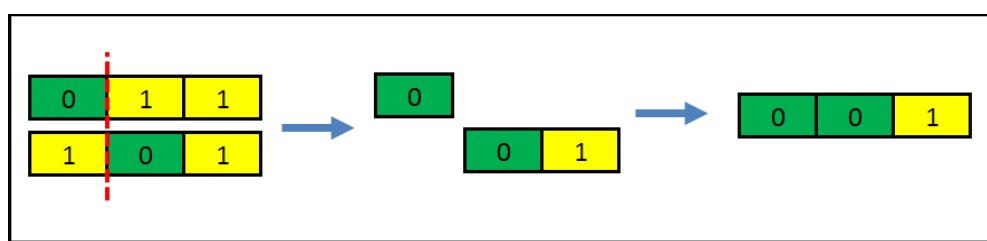


Figura 9 – Exemplo do *crossover* entre os indivíduos 011 e 101 para o primeiro ponto de corte.

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

$$geracao_auxiliar[0] = 001.$$

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

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 do que o $f(2) = 32$.

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 ($\langle f \rangle$) 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
$\langle f \rangle = 24,25$		$\langle f \rangle = 25,00$	

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.

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 29). 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).

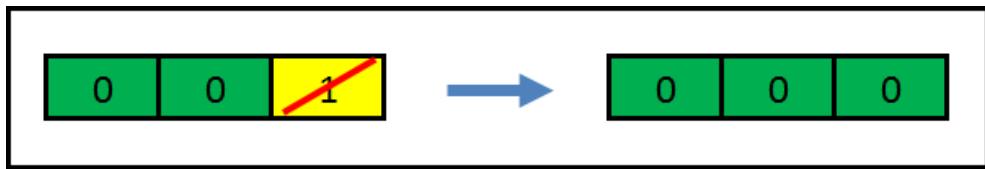


Figura 12 – Representação gráfica de uma mutação. Nesse exemplo a mutação no último *bit* levou à solução ótima para o máximo da função $y(x) = x^2 + 36$.

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, 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-se 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.

```

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    }

```

Figura 13 – Exemplo de código para o operador Mutação.

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.

4 CUDA

CUDA (*Compute Unified Device Architecture*) é a arquitetura de computação paralela da NVIDIA para GPUs (*Graphic Processor Unit* - Unidade de Processamento Gráfico). Graças à combinação de CPUs multinúcleo com as GPUs, os PCs e *laptops* entraram na era da computação na teraescala ([KIRK; HWU, 2010](#)), abrindo a possibilidade para pesquisadores e empresas de diversas áreas tomarem proveito da computação de alto desempenho. Muitos produtos comerciais já são compatíveis com CUDA, como o Mathematica e o MatLab.

A principal diferença na comparação com a CPU é o fato da GPU ser especializada em computação intensiva (ponto flutuante) e paralela - historicamente necessárias para rerenderização de imagens ([NVIDIA, 2011](#)). Do ponto de vista do *hardware*, a GPU utiliza mais transistores para operações matemáticas (figura 14).

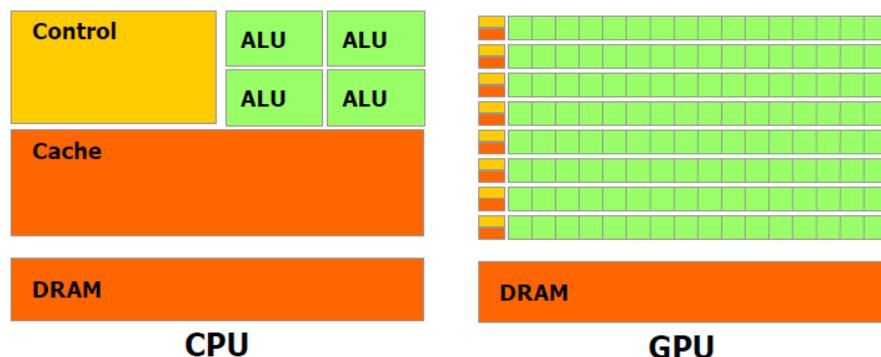


Figura 14 – Representação esquemática da utilização de transistores na CPU e na GPU.

Note que o número de transistores dedicados às operações matemáticas é muito superior, enquanto o controle de fluxo de dados e a memória *cache* estão naturalmente paralelizados. Retirado de ([NVIDIA, 2011](#)).

No modelo híbrido de programação, parte do código é executada na CPU, parte na GPU. A porção executada na placa de vídeo deve possuir características para tirar proveito da arquitetura CUDA, ou seja, ser paralelizável e necessitar de cálculo intensivo.

Para programar uma GPU compatível com CUDA utiliza-se o CUDA C, uma extensão do ANSI C disponibilizada gratuitamente pela NVIDIA. A CPU (*host*) executa o código de maneira sequencial, enquanto a GPU (*device*) utiliza o conceito de *Multithreading* para a computação paralela. Ou seja, na arquitetura CUDA, a CPU e a GPU são complementares.

Na figura 15 há um exemplo. O objetivo do programa é somar dois vetores,

tarefa que pode ser facilmente paralelizada pois a soma de cada elemento acontece de maneira independente. No início do código uma função chamada `VecAdd` é declarada. Seus parâmetros são três ponteiros para um número real e, no corpo da função, acontece a soma dos elementos do vetor:

$$C[i] = A[i] + B[i] \quad (4.1)$$

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
}
```

Figura 15 – . Exemplo de código em CUDA C. A função `VecAdd` adiciona dois vetores A e B de ordem N e armazena o resultado no vetor C . Retirado de ([NVIDIA, 2011](#)).

Naturalmente sente-se falta de um *loop* percorrendo todos os N elementos do vetor. Porém, dentro do CUDA C, esse *loop* é substituído por `int i = threadIdx.x`, indicando que, agora, há uma *thread* dedicada à soma de cada um dos i elementos.

Entretanto, dentro da função `VecAdd()` não há referência ao tamanho dos vetores sendo somados. Então, como `VecAdd()` sabe que precisa de N *threads* para executar a soma de todos os N elementos em paralelo? Essa informação está contida na chamada da função `VecAdd()` dentro da `main()`. A sintaxe `<<1, N>>` diz ao compilador que a função precisará de 1 bloco com N *threads/bloco*. Se for necessário lidar com um vetor maior, basta solicitar mais *threads*, o que exige o conhecimento da arquitetura CUDA. A cláusula `__global__` na definição de `VecAdd()` indica que essa função será executada na GPU.

O número de *threads* por bloco é uma característica de cada placa, e pode ser obtido em tempo de execução. Cada núcleo de processamento executa um bloco com suas N *threads/bloco*. Caso o número necessário de *threads* exija mais blocos do que os núcleos disponíveis, os blocos são distribuídos automaticamente pela CUDA (figura 16), tornando esse processo transparente para o programador.

Essa facilidade permite escrever códigos altamente escaláveis. Nota-se na figura 16 que um mesmo programa pode rodar em placas simples, com poucos núcleos, ou nas mais sofisticadas com centenas de núcleos. Isso vale também para a utilização de mais de uma placa.

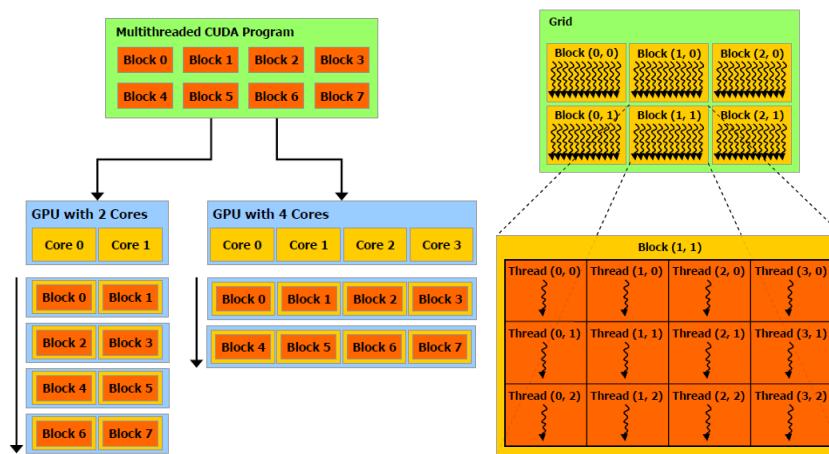


Figura 16 – Estrutura hierárquica das *threads*. No lado esquerdo há uma representação da distribuição de blocos entre os núcleos. Esse processo, que é automático, permite a construção de programas altamente escaláveis. À direita está a disposição das *threads* dentro dos blocos. Retirado de ([NVIDIA, 2011](#)).

5 Materiais e Métodos

Antes de atacar o método em si foi necessário estudo de Algoritmos Genéticos e Linguagem C, escolhida pensando em CUDA.

5.1 CUDA

Metodologia: revisão bibliográfica e aplicação no ONEMAX.

Material do ERAD.

5.2 Método

Artigo de 2004. Aproveitar conteúdo do segundo relatório de estudos dirigidos.

Apresentar o fitness *fitness* com $\rho - \rho_0$ utilizado no Artigo de 2006.

Apresentar e definir a matriz de Coope-Sabo (citar artigo original de 77), utilizada no pelos indianos no artigo de 2006.

5.3 Framework/Programa Serial

Optou-se por desenvolver do zero todo o programa. Motivo: controle sobre todas as características do método.

Linguagem C: linguagem nativa para CUDA.

Totalmente parametrizado.

Com isso será possível:

1. Estudo do método
2. Estudo de algoritmos genéticos
3. Novas aplicações como máximo de função
 - a) Mudança no fitness
 - b) Mudança nos critérios de parada

Descrição dos parâmetros.

Descrição de cada função, incluindo *print screen* das partes de código mais fundamentais:

1. Estruturas de dados
2. Geração de números pseudoaleatórios
3. Geração das Matrizes de Coope
4. Geração da População Inicial
5. Fitness: as várias equações
6. Fitness: cálculo de ρ_i
7. Fitness: cálculo de $\nabla\rho_i$
8. Seleção
9. Crossover
10. Mutação
11. Álgebra Linear: multiplicação de matrizes
12. Álgebra Linear: multiplicação de matriz por escalar
13. Álgebra Linear: subtração de matrizes

Qualidade dos números pseudo-aleatórios (base do GA)

1. Mostrar que a distribuição dos números segue o esperado para números aleatórios
2. Quanto maior a quantidade de pontos, melhor a distribuição
3. Gráfico: histograma de frequência.

Exemplo de execução no Windows.

Reprodutibilidade.

Exemplo de reproduibilidade.

Código disponível em

https://github.com/prietoab/msc_code

6 Resultados e discussão

1. Parágrafo de introdução do capítulo. Citar que, basicamente, o leitor encontrará no capítulo:
 - a) Resultados do ONEMAX, legitimando o uso do código para o programa mais complexo que foi utilizado no método dos indianos.
 - b) o estudo dos tipos de *fitness*, operador responsável pelo elo entre o algoritmo e o problema ([LINDEN, 2008](#)), que, para o nosso caso, é encontrar autovalores. Ponte para o próximo: para cada tipo de *fitness*, um resultado diferente.
2. Os dois tipos de fitness dos indianos. Ideia central: dois tipos, resultados diferentes. Com $\nabla\rho$ chegamos a um autovalor qualquer, com $(\rho - \rho_0)^2$ podemos chegar ao mínimo, mas dá mais trabalho. Ponte para o próximo: proposta de dois novos fitness.
3. Combinação de $\nabla\rho$ com $(\rho - \rho_0)^2$. Se cada forma leva a comportamentos diferentes, tentamos combinar os dois termos em um único fitness. Uma hipótese seria a melhoria da qualidade dos resultados. A hipótese não foi confirmada. Ponte para o próximo: a busca pela qualidade levou à verificação da importância do parâmetro λ .
4. Além do que os indianos disseram, que λ é escolhido para não estourar a função exponencial, ele tem influência na convergência do algoritmo e na precisão (ou resolução) do fitness. Se na primeira população, geração inicial, o fitness médio é alto, isso provoca convergência precoce, fazendo com que o resultado final seja ruim. Por outro lado, se no início o fitness médio é muito baixo, não há muita discriminação entre os indivíduos, o fitness não cresce e não chegamos a uma solução. A medida que o fitness se aproxima de 1, a discriminação entre os indivíduos fica difícil, levando ao problema da resolução. Ponte para o próximo: vários testes levaram ao desenvolvimento de uma equação empírica para λ , restrita às matrizes de Coope–Sabo ([COOPE; SABO, 1977](#)).
5. Fórmula empírica. Por já conhecermos de antemão os autovalores das matrizes de Coope–Sabo, foi possível criar uma fórmula empírica para λ . Ela garante que na primeira população o fitness médio é baixo, previnindo o *underflow* do fitness e a convergência prematura.

6.1 Problemas com o mínimo global

Na seção 5.2 vimos que o *fitness* utilizado no artigo (NANDY *et al.*, 2004) foi

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

onde f_i é o *fitness* do i -ésimo indivíduo da população, λ 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, testamos o método com matrizes de Coope-Sabo 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 λ , que será discutido em detalhes posteriormente, o *fitness* comportou-se conforme o esperado em todos os casos. Um exemplo está na figura 17, 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 próximo passo 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 17, mas invertida, ou seja, os primeiros valores de ρ seriam grandes e, rapidamente, diminuiriam até haver convergência para o autovalor mínimo. Na figura 18 há um exemplo. Os gráficos exibem os valores de ρ para a mesma execução apresentada na figura 17. Note no primeiro gráfico que até a geração 20 o quociente ρ teve caráter oscilatório e, então, aparentemente estabilizou-se 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 do algoritmo convergir para E_0 . Porém, para esse exemplo especificamente, isso não aconteceu, como pode ser visto no segundo gráfico da figura 18. Para garantir a estabilidade, o programa foi executado até a

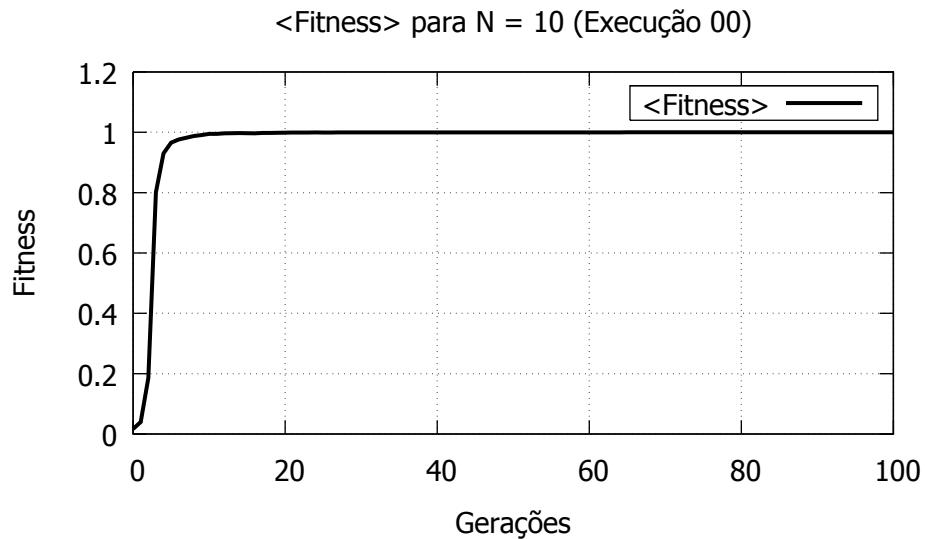


Figura 17 – Comportamento do fitness $f_i = e^{-\lambda \|\nabla \rho_i\|^2}$ para $N = 10$. Na primeira geração o melhor fitness é pequeno, aproximadamente 0,1, cresce rapidamente e a partir da décima geração está próximo de 1.

geração 400.000, e o valor médio obtido foi $\langle \rho \rangle = 6,572898$. Para nossa 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, $E_3 = 6,572897$. Um gráfico expandido dessa execução está na figura 19 da página 41. Pensamos, então, que poderia haver algo de errado com o nosso programa.

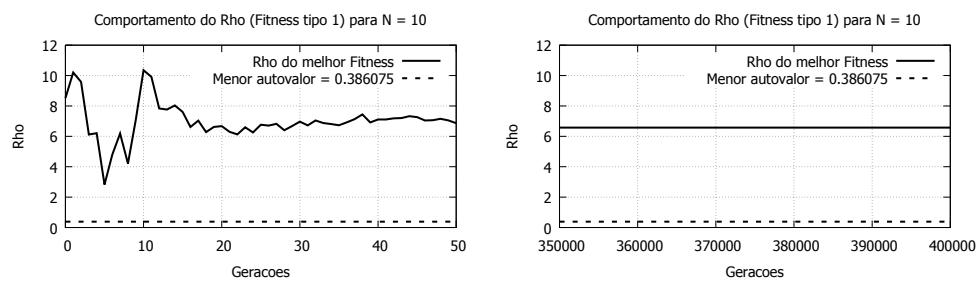


Figura 18 – Comportamento de ρ (Quociente de Rayleigh) para uma matriz de Coope–Sabo de ordem 10.

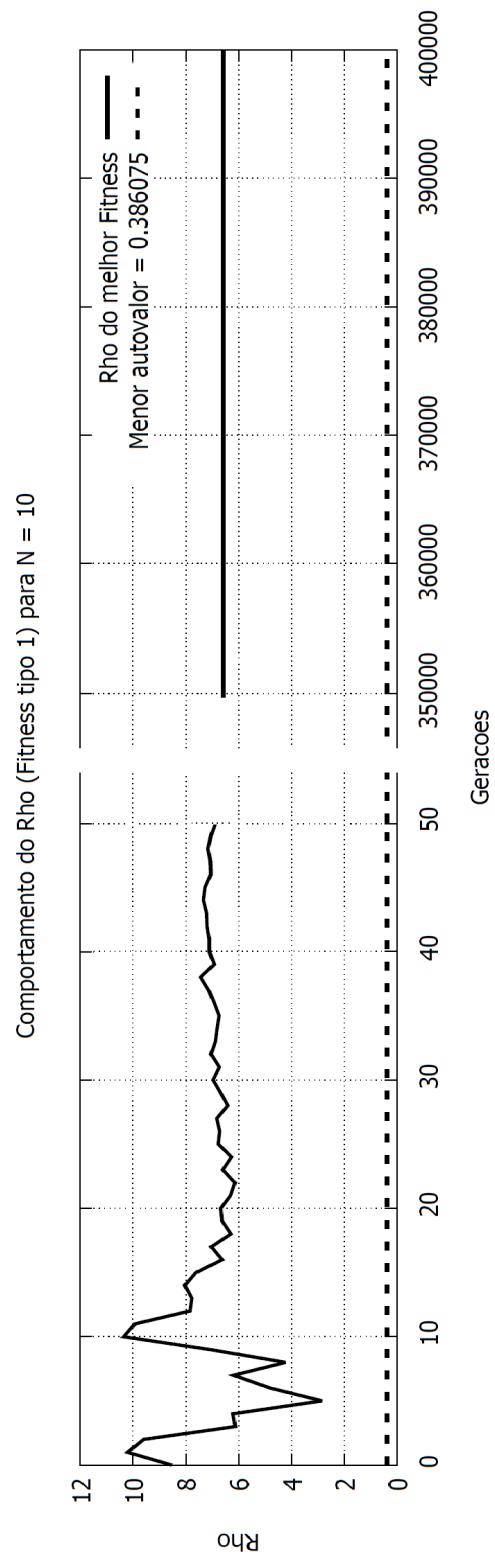


Figura 19 – Comportamento de ρ (Quociente de Rayleigh) para uma matriz de Coope–Sabo de ordem 10.

Após esses resultados preliminares executamos uma validação cuidadosa do programa, testando cada uma de suas quase 2500 linhas e comparando os resultados das operações e cálculos com os softwares Excel ([Microsoft Corporation, 2007](#)) e SciLab ([Scilab Enterprises, 2012](#)). 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. De fato alguns erros foram encontrados.

Discutiremos a seguir os testes com a versão corrigida do programa e exibidos nas figuras [30](#), [31](#), [32](#) e [33](#). Visando brevidade, apresentaremos dados para matrizes de CoopeSabo 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 [14](#) 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 $E_4 = 8,450274$.

Comecemos a discussão com o que foi encontrado 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 [10](#), concluímos 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 atingiu a convergência genética. 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 há dúvidas sobre a validade do nosso programa e, obviamente, dos resultados produzidos. Então, buscamos 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 acreditamos que nelas residam a explicação dos resultados obtidos por nós até agora.

A primeira relação de causalidade refere-se à afirmação “ $f_i \rightarrow 1$ quando $\nabla\rho_i \rightarrow 0$ ”, que está absolutamente correta. Retomando a seção 5, o *fitness* definido pela equação 6.1 é limitado no intervalo $(0,1]$ e, como $\lambda > 0$, só chega ao seu valor máximo quando $\nabla\rho_i = 0$. Em outras palavras, $\nabla\rho_i \rightarrow 0$ 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 que, apesar de sutil, é muito poderosa:

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

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, ainda por cima, é um vetor especial, o autovetor fundamental C_0 . Portanto, possui autovalor associado E_0 , o autovalor mínimo (conforme equação 6.4). Grosso 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*”². Conforme já dito no capítulo 2, um desses fatos diz que $\rho(u)$ é estacionário, ou seja, $\nabla\rho(u) = 0$, apenas se o vetor u é

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 .*”

² Tradução livre de “*At many places in the book, reference is made to more or less well known facts from matrix theory*”.

um autovetor w de $HC = EC$. Consequentemente, o encadeamento correto se apresenta como:

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

Então, se $f_i = 1$, o máximo que podemos concluir é que C_i é *algum* autovetor, e não necessariamente *o* autovetor fundamental. Ao fim de todos os nossos 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). Nossos dados confirmam a matemática e, assim, acreditamos que nosso programa não contém erros.

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 $\nabla\rho \rightarrow 0$.

Resta a dúvida: afinal, como o autovalor mínimo foi obtido com o *fitness* definido pela equação 6.1? Não sabemos. 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 identificamos nada nesses quatro artigos que pudesse levar à resposta. Seguimos o estudo com uma nova definição do *fitness* encontrada em (NANDY *et al.*, 2011).

Tabela 10 – Execuções para matrizes de Coop–Sabo.

N	Execução	Semente	λ	$\langle Fitness \rangle$	$\langle \rho \rangle$	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

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

e apresenta semelhanças com o definido pela equação 6.1. Há uso de uma exponencial, o parâmetro λ 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* autovalor que estamos procurando³. 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 ”⁴. Parece que, outra vez, 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-se que o *fitness* anterior (equação 6.1) contém $\nabla\rho$, que possui relação direta com os autovalores de H quando $\nabla\rho = 0$.

Repeti as execuções da tabela 10 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 47, e os gráficos da evolução do *fitness* e do quociente de Rayleigh estão nas páginas 70, 71, 72 e 73. 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) da média de ρ 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 .

³ L de *lower*.

⁴ Tradução minha para “If $\rho_i \rightarrow E_L$ during the search, $f_i \rightarrow 1$ and C_i approaches the ground eigenvector of H ”.

Tabela 11 – Execuções novo *Fitness*.

N	Execução	Semente	λ	$\langle \text{Fitness} \rangle$	$\langle \rho \rangle$	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%

As diferenças dos valores finais são indiscutíveis, sugerindo que o comportamento do *fitness* e do ρ ao longo da busca também deve ter sido alterado. Na figura 20 estão os gráficos referentes à execução zero para o Hamiltoniano de ordem 10, semente 1445738835. A primeira usa o *fitness* $f_i = e^{-\lambda(\rho_i - E_L)^2}$, que chega ao autovalor mínimo, enquanto a segunda utiliza o $f_i = e^{-\lambda\|\nabla\rho_i\|^2}$. Ambos saem de 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 em um autovalor diferente. A execução da direita, lembre-se, obteve apenas um mínimo local ($E_1 = 2,461056$, tabela 10).

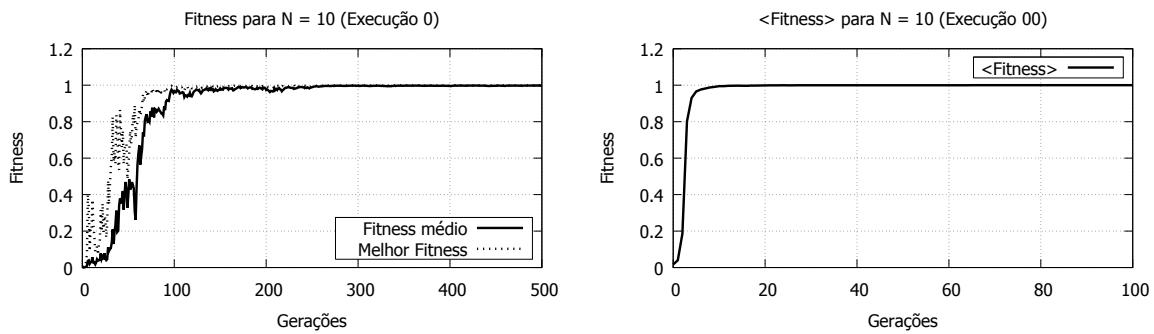


Figura 20 – Comportamento do *fitness* para as execuções zero do Hamiltoniano de ordem 10, semente 1445738835. A primeira usa o *fitness* $f_i = e^{-\lambda(\rho_i - E_L)^2}$, que chega ao autovalor mínimo, enquanto a segunda utiliza o $f_i = e^{-\lambda\|\nabla\rho_i\|^2}$.

De todo modo, as duas execuções estão conectadas pois, como partiram da mesma semente de números pseudoaleatórios, a população inicial foi *exatamente* 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 21 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* $f_i = e^{-\lambda(\rho_i - E_L)^2}$.

É tentador 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 viajam no espaço de soluções tem forte base estocástica e, portanto, qualquer comparação linear é extremamente arriscada, quiçá impossível. Objetivamente, posso apenas concluir que os valores finais encontrados por cada *fitness* estão condizentes com a construção de cada função objetivo: $\nabla\rho_i$ leva a qualquer autovalor; $\rho_i - E_L$ encontra o autovalor mínimo.

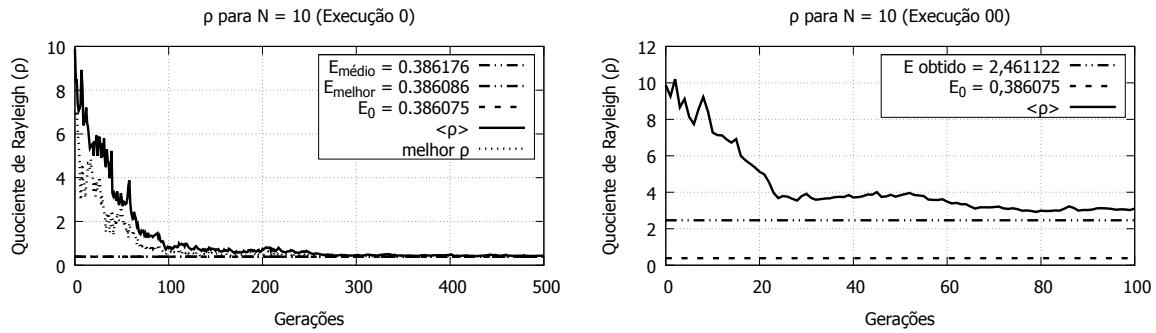


Figura 21 – Comportamento do ρ para as execuções zero do Hamiltoniano de ordem 10, semente 1445738835. A primeira usa o fitness $f_i = e^{-\lambda(\rho_i - E_L)^2}$, que chega ao autovalor mínimo, enquanto a segunda utiliza o $f_i = e^{-\lambda\|\nabla\rho_i\|^2}$.

Ok. Os dados mostraram que chega no menor autovalor. Mas, se não há relação direta no fitness, como isso acontece? Outra sutileza: E_L é um limite inferior para o menor autovalor. Roubada? Não parece muito útil, pois o fitness só foi próximo de 1 porque escolhi um E_L bem próximo de E_0 . Mas, o que aconteceria se eu não soubesse por onde anda ou autovalor mínimo? Quatro cenários para E_L . Cenário 1: com sorte, o E_L escolhido está um pouco abaixo de E_0 . Encontra o valor mínimo, conforme exemplos. Cenário 2: um pouco acima de E_0 . Cenário 3: muito abaixo de E_0 ; Cenário 4: muito acima de E_0 .

Semente 1445738835. Tipo 1. E_L um pouco acima. Sempre converge para E_L . “Passa” por todos os autovalores, mas não para em nenhum. Aconteceu em *todas* as execuções.

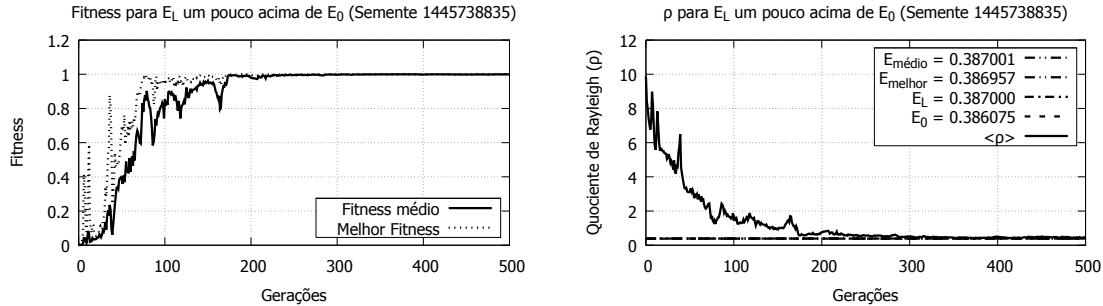


Figura 22 – Execução para a semente 1445738835. E_L um pouco acima de E_0 no fitness $f_i = e^{-\lambda(\rho_i - E_L)^2}$.

E_L um pouco abaixo. Já citado no início da seção. Chegou ao menor autovalor em todas as execuções. Fitness médio é próximo de 1 pois E_L está próximo de E_0 . Melhor cenário.

Semente 1445738835. Tipo 3. E_L muito acima. Novamente, chegou ao E_L em todas as execuções. Tendo cuidado com o λ , parece não haver diferenças entre *um pouco* acima e *muito* acima. Mas, aqui $\nabla\rho >> 0$, estamos longe de algum autovalor. O valor de

$\nabla\rho$ é próximo de zero pro “um pouco acima”, indicando que estamos próximo do autovalor mínimo. (verificar com a tabela de execuções). $\nabla\rho$ tem importância a cada iteração mesmo não estando no *fitness*. Mas, mesmo que, accidentalmente, E_L é escolhido como próximo de um autovalor, $\nabla\rho \approx 0$, e não podemos dizer que convergiemos para um autovalor. Diferente do caso do *fitness* com $\nabla\rho$.

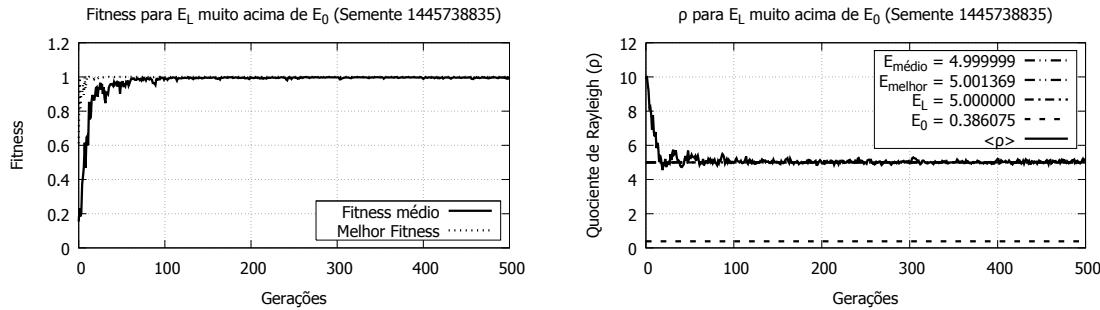


Figura 23 – Execução para a semente 1445738835. E_L muito acima de E_0 no *fitness* $f_i = e^{-\lambda(\rho_i - E_L)^2}$.

Semente 1445738835. Tipo 4. E_L muito abaixo. Hipótese: não encontrará, muito distante. O *fitness* ficou praticamente zero. Havia variabilidade no início, mas, como o *fitness* foi zero pra todos, não havia como distinguir os melhores indivíduos. Entre as gerações 0 e 500 houve convergência genética precoce, estabilizando a média dos ρ em aproximadamente 5.9 (verificar), que não é nenhum autovalor pra $N = 10$.

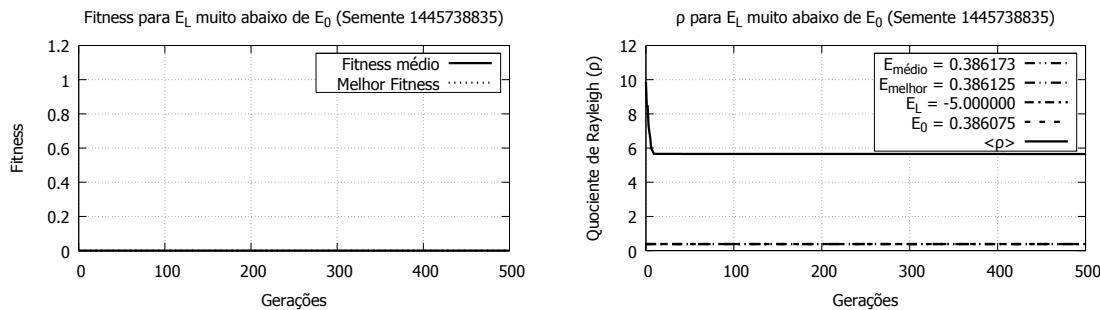


Figura 24 – Execução para a semente 1445738835. E_L muito abaixo de E_0 no *fitness* $f_i = e^{-\lambda(\rho_i - E_L)^2}$. Até geração 500.

Entretanto, houve convergência para o autovalor mínimo. Um pouco antes da geração 32.000 aconteceu um salto no *fitness*, causado possivelmente por mutações (argentar). Apesar do *fitness* médio ainda ser pequeno ($\langle f_i \rangle < 0.025$), o crossover com a nova informação genética criou variabilidade suficiente para chegar ao autovalor mínimo.

Na tabela ?? 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 $\langle \rho \rangle$ (%)).

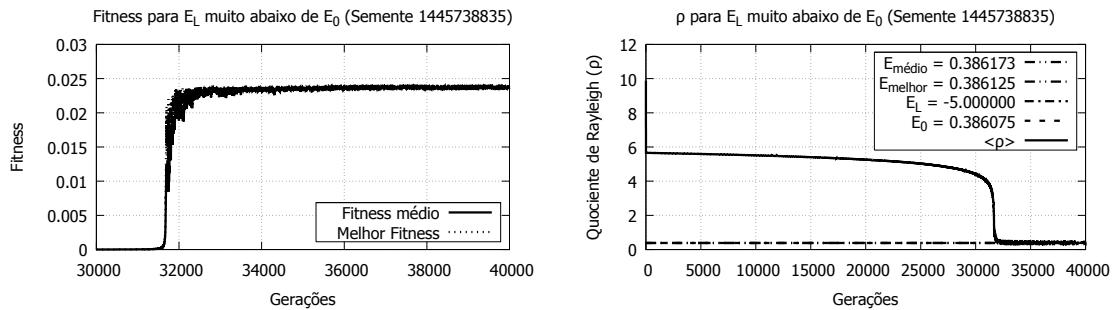


Figura 25 – Execução para a semente 1445738835. E_L muito abaixo de E_0 no fitness $f_i = e^{-\lambda(\rho_i - E_L)^2}$. Geração entre 30.000 e 40.000.

Tabela 12 – Variando E_L para a execução da semente 1445738835. Os tipos de teste são:
tipo 1: E_L um pouco acima de E_0 ; **tipo 2:** E_L um pouco abaixo de E_0 ; **tipo 3:** E_L muito acima de E_0 ; **tipo 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

Tabela 13 – Cinco execuções para cada tipo de teste de variação de E_L em torno de E_0 no fitness $f_i = e^{-\lambda(\rho_i - E_L)^2}$.

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 Por que o λ deve ser escolhido cuidadosamente?

Execuções para $N=10$ com diferentes λ 's. Com os gráficos, explicar o que o artigo de 2004 quis dizer com *fitness overflow/underflow*.

Gráficos com rho entre 0 e 250 (exemplo pra $N=10$), mas com cortes em diferentes rhos.

Explicar que uma boa escolha do λ deve cobrir todos os autovalores. Citar as execuções anteriores (boas e ruins em função de cada λ).

Gráfico com λ fazendo o fitness cortar em um ρ muito baixo. Discutir puxando as execuções anteriores.

Outro gráfico, mas com λ fazendo o fitness cortar em um ρ muito alto. Discutir puxando as execuções anteriores.

Gráfico com uma boa escolha de λ . Discutir puxando as execuções anteriores.

Após estimativa, refinar a obtenção do λ . Alterar o *lambda* (valores em torno da estimativa), executar o programa para verificar se o fitness médio da primeira população é baixo. (se a população inicial tem fitness muito grande, há convergência prematura).

Tabela com alguns *lambdas* encontrados dessa maneira (estimativa e refinamento).

Infelizmente, para cada matriz, um λ diferente.

Ponte pra equação empírica do λ .

6.4 Equação empírica para o λ

Delineamento da equação como feito na reunião de 29/09.

Isolar λ a partir da $f = e^{-\lambda*(\rho-\rho_0)^2}$

Fazer $f = 0.00001 \approx 0$.

Substituir $(\rho - \rho_0)^2$ por $E_{central} - E_{mínimo}$. Justificar.

Regressão linear para $E_{central} - E_{mínimo}$ com função apenas da ordem da matriz (N).

Inserir a Equação obtida na regressão na equação de λ .

Fator 0.65: obtido empiricamente de modo que o λ seja semelhante aos encontrados pelo processo de estimativa e refinamento.

Exemplo de execução com λ automático.

Explicitar que essa equação é válida apenas para matrizes de Coope–Sabo.

Apesar disso, foi importante para o estudo pois permitiu automação completa.

6.5 A mistura de $(\rho - \rho_0)^2$ com $\nabla\rho$ não leva a melhores resultados

Como em seção anterior verificamos que $f_i = e^{[-\lambda\nabla\rho]}$ é mais rápido do que $f_i = e^{[-\lambda(\nabla\rho)]}$, e que o $\nabla\rho$ está diretamente associado aos autovalores, pensei na seguinte hipótese: inserir $\nabla\rho$ ao fitness com $(\rho - \rho_0)^2$ traria resultados mais rápidos.

Justificativas para a hipótese:

1. Inserir $\nabla\rho$ no fitness puniria os ρ 's que, apesar de próximos de ρ_0 , não fossem autovalor. Em outras palavras, o termo $\rho - \rho_0 \approx 0$, mas $\nabla\rho \gg 0$ e, portanto, o fitness ficaria pequeno.
2. Como o fitness, a princípio, estaria diferenciamento melhor os bons indivíduos, o algoritmo teria uma taxa de convergência maior.

Executar 10 para o primeiro fitness, e, utilizando as mesmas dez sementes, executar outros 10 testes com ou outro fitness.

Comparação dos resultados: gráficos do comportamento do fitness e tabela comparando a velocidade de convergência (em que geração o critério de parada foi atingido), tempo de execução e erro relativo ao menor autovalor “exato” (obtido no SciLab).

6.6 $f_i = e^{[-\lambda\nabla\rho]}$ é mais rápido do que $f_i = e^{[-\lambda(\nabla\rho)^2]}$

Como um dos critérios de parada utiliza $\nabla\rho$ (sem quadrado), testamos essa forma no fitness.

Várias execuções.

Gráfico comparando o comportamento (um termina mais rápido)

Tabela com os detalhes explícitos do ganho.

Ponte pra falar sobre o outro fitness que encontra o mínimo.

6.7 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”.

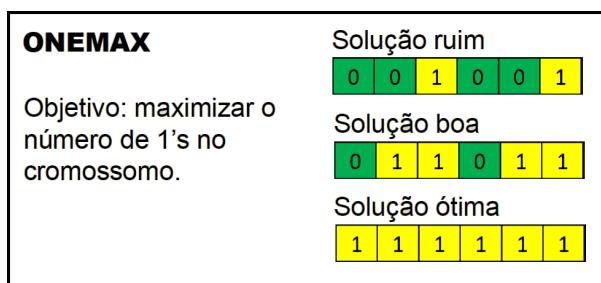


Figura 26 – ONEMAX, um problema clássico nos Algoritmos Genéticos.

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.

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 esse caso seria “111111” (figura 26). 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

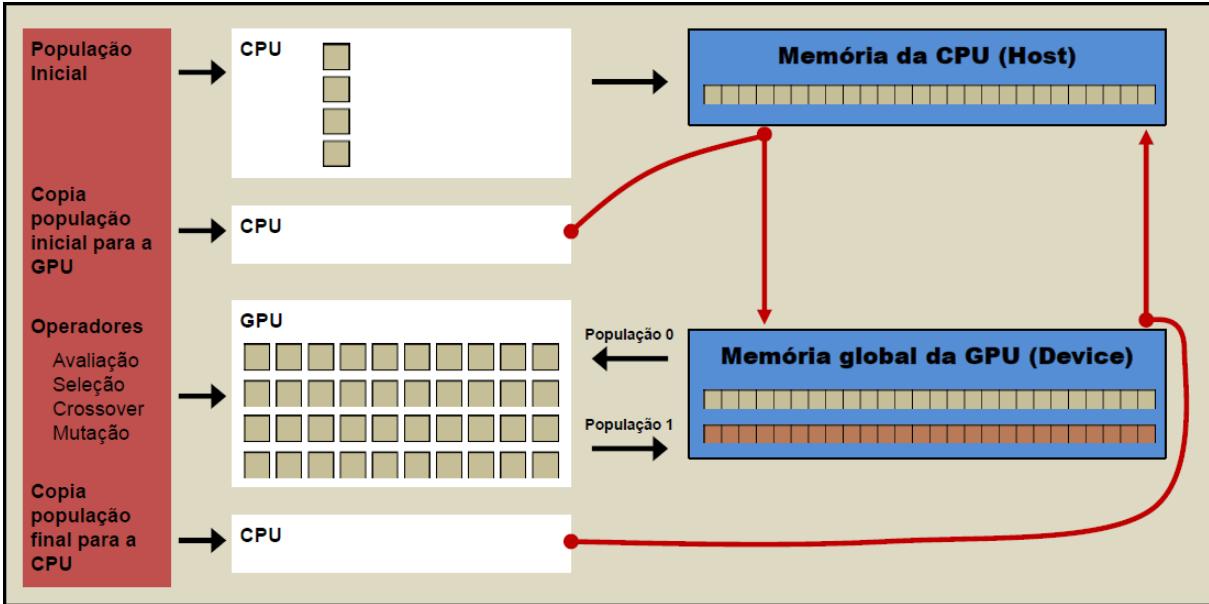


Figura 27 – Execução do ONEMAX paralelo. Apenas as chamadas dos kernels acontecem na CPU, enquanto o restante (execução + dados) está na GPU.

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 → 1 ou 1 → 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 28). 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 analisarmos a influência do tamanho do cromossomo verificamos um comportamento aproximadamente constante do ganho (figura 29). 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 9x para qualquer tamanho de cromossomo. O comportamento se repete com uma população de 10 indivíduos, mas, nesse caso, o programa serial sempre é mais rápido, mesmo para cromossomos muito pequenos (ganho sempre < 1).

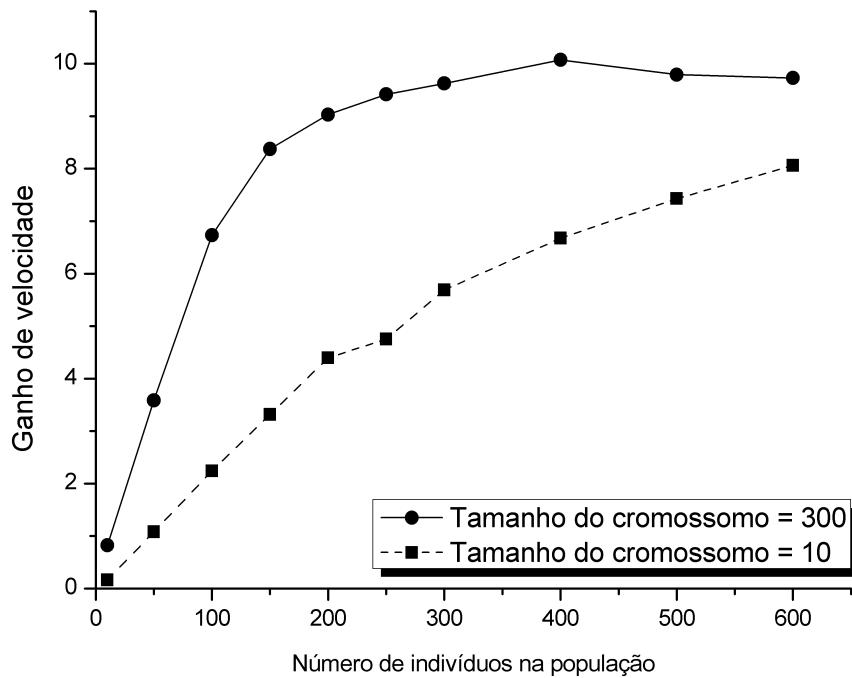


Figura 28 – ONEMAX paralelo. Ganho de velocidade em função do número de indivíduos da população.

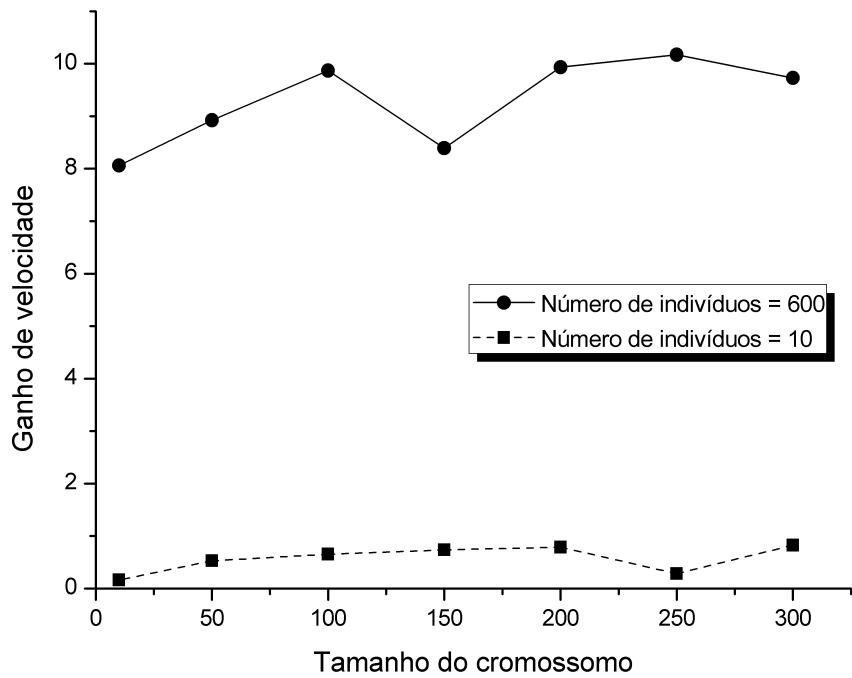


Figura 29 – ONEMAX paralelo. Ganho de velocidade em função do tamanho do cromossomo.

Referências

- 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.gpgppu.com/gecco2009/3.pdf>>.
- KIRK, D. B.; HWU, W. W. *Programming Massively Parallel Processors*. 1. ed. [S.l.]: Elsevier, 2010.
- LINDEN, R. *Algoritmos Genéticos. Uma importante ferramenta da Inteligência Computacional*. [S.l.]: BRASPORT, 2008.
- Microsoft Corporation. *Microsoft Excel 2007*. Redmond, Washington, 2007. Disponível em: <<https://products.office.com/pt-br/excel>>.
- NANDY, S.; CHAUDHRY, 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, 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.
- NVIDIA. *NVIDIA CUDA C Programming Guide. Version 4.0*. [S.l.], 2011. Disponível em: <<http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>>.
- PARLETT, D. N. *The Symmetric Eigenvalue Problem*. 2. ed. Philadelphia, USA: SIAM - Society for Industrial and Applied Mathematics, 1998. (Classics in Applied Mathematics).
- Scilab Enterprises. *Scilab: Free and Open Source software for numerical computation*. Orsay, France, 2012. Disponível em: <<http://www.scilab.org>>.
- 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, 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.

Apêndices

APÊNDICE A – Lista de autovalores

Tabela 14 – 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

APÊNDICE B – Execuções para o *fitness*

$$f_i = e^{-\lambda ||\nabla \rho||^2}$$

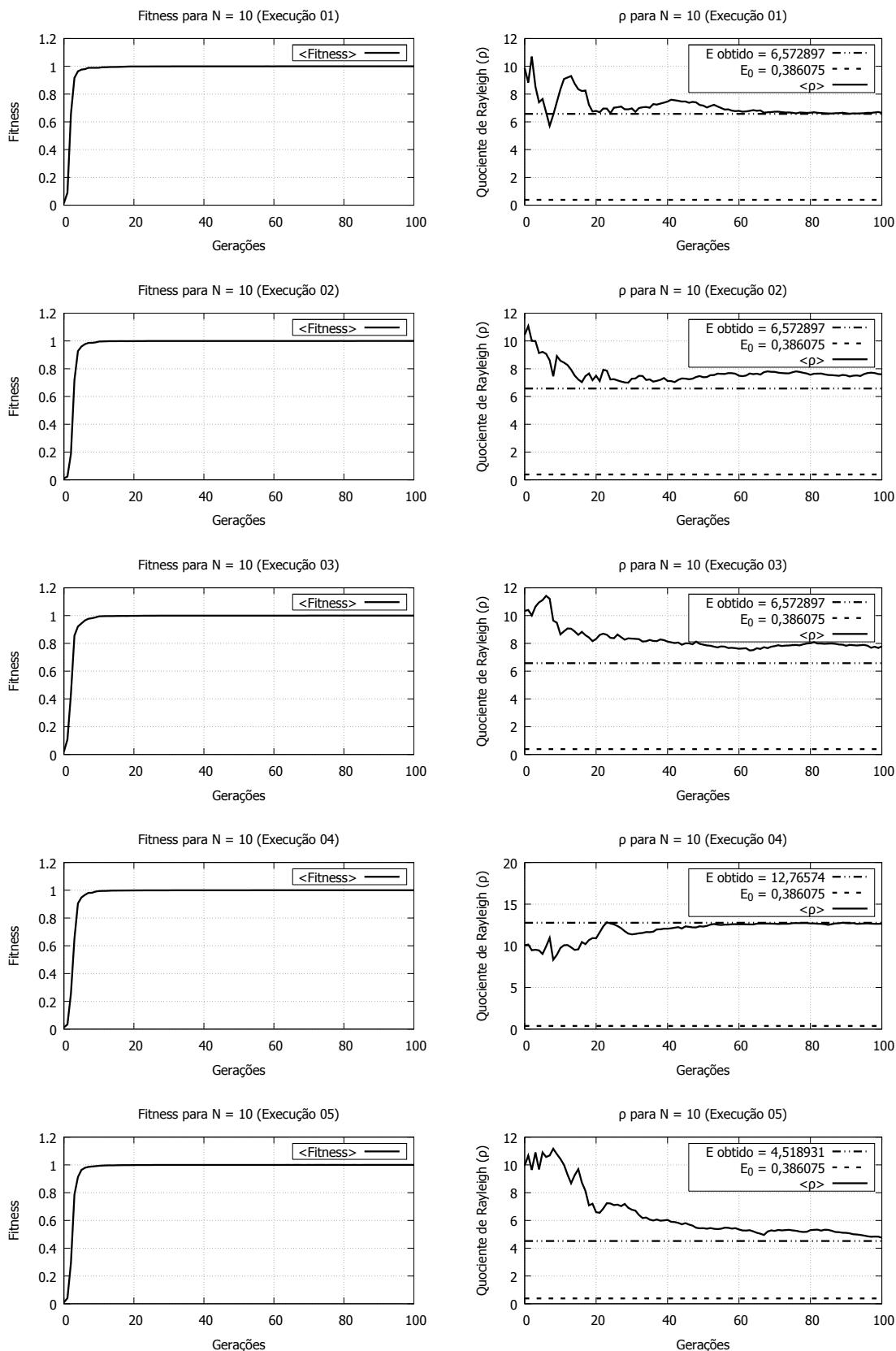


Figura 30 – Execuções N = 10.

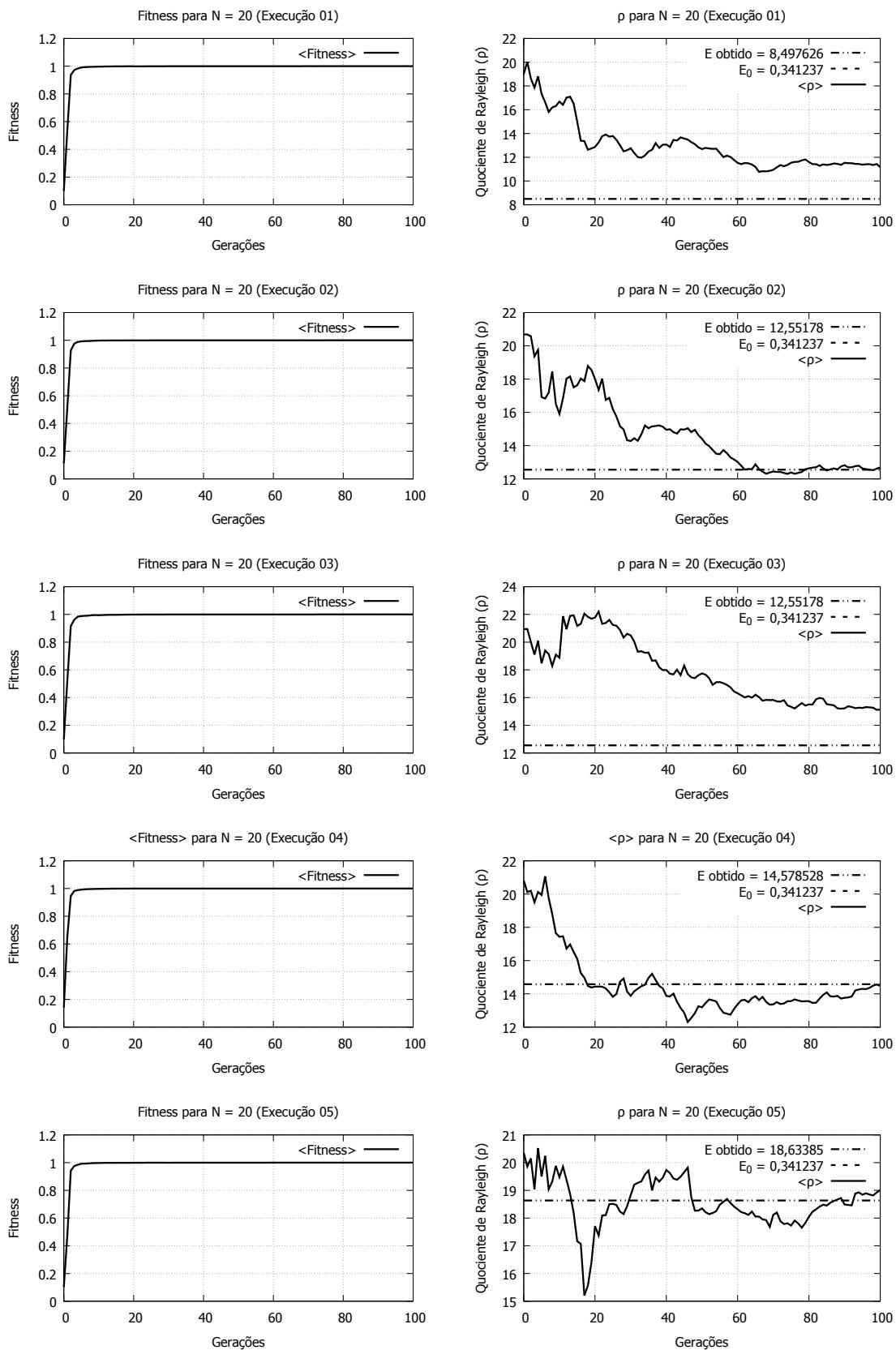


Figura 31 – Execuções N = 20.

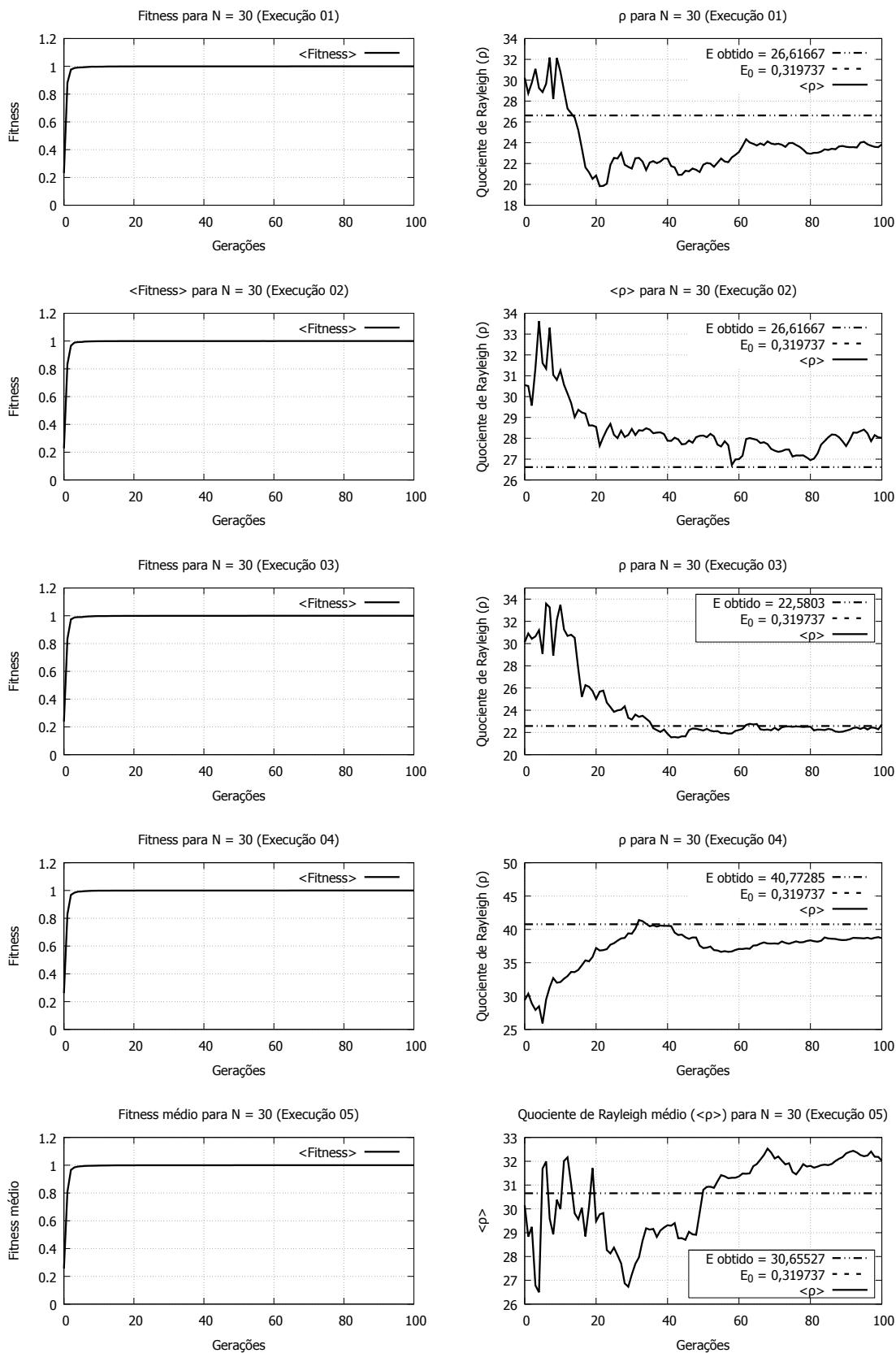


Figura 32 – Execuções N = 30.

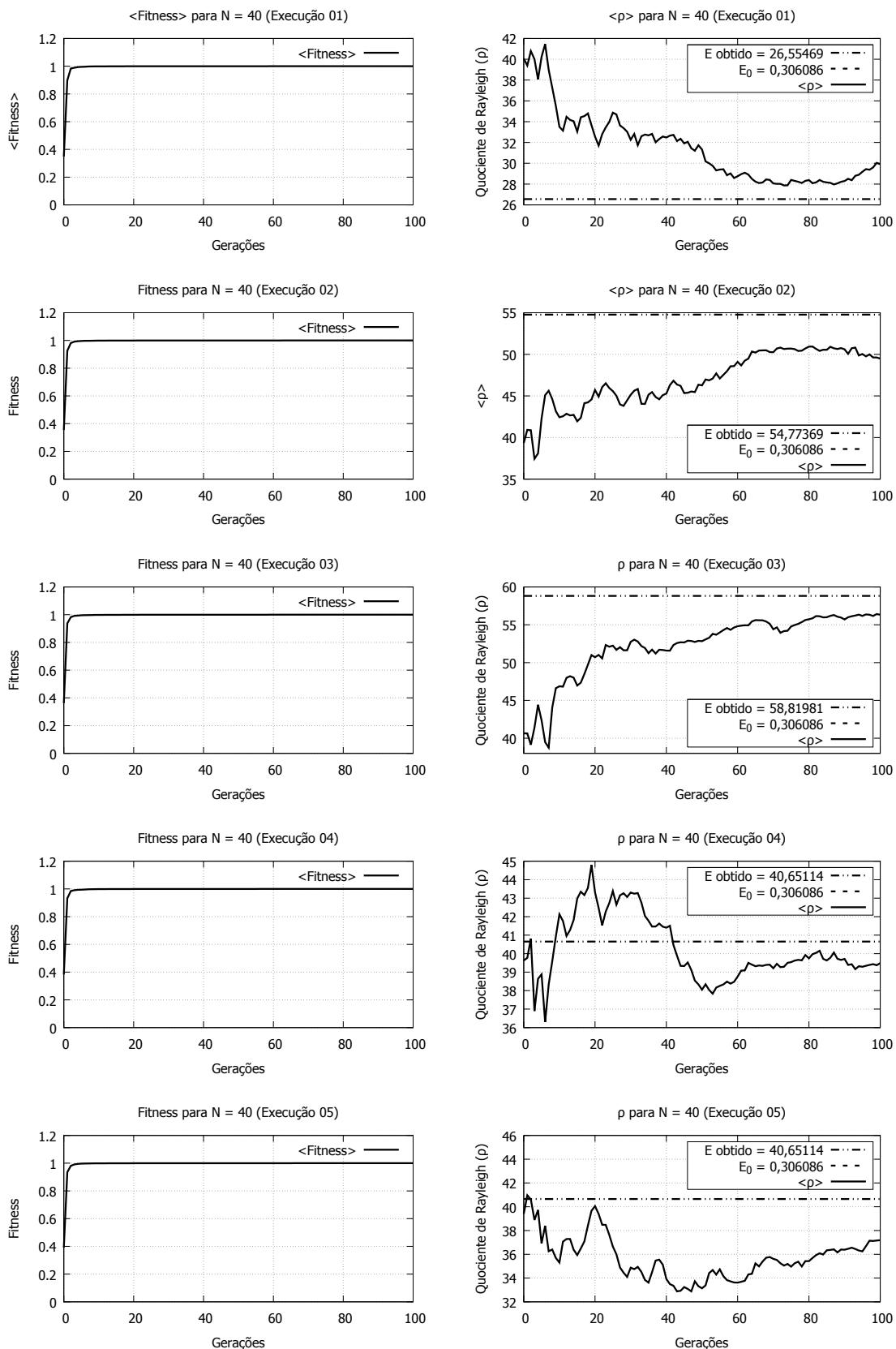
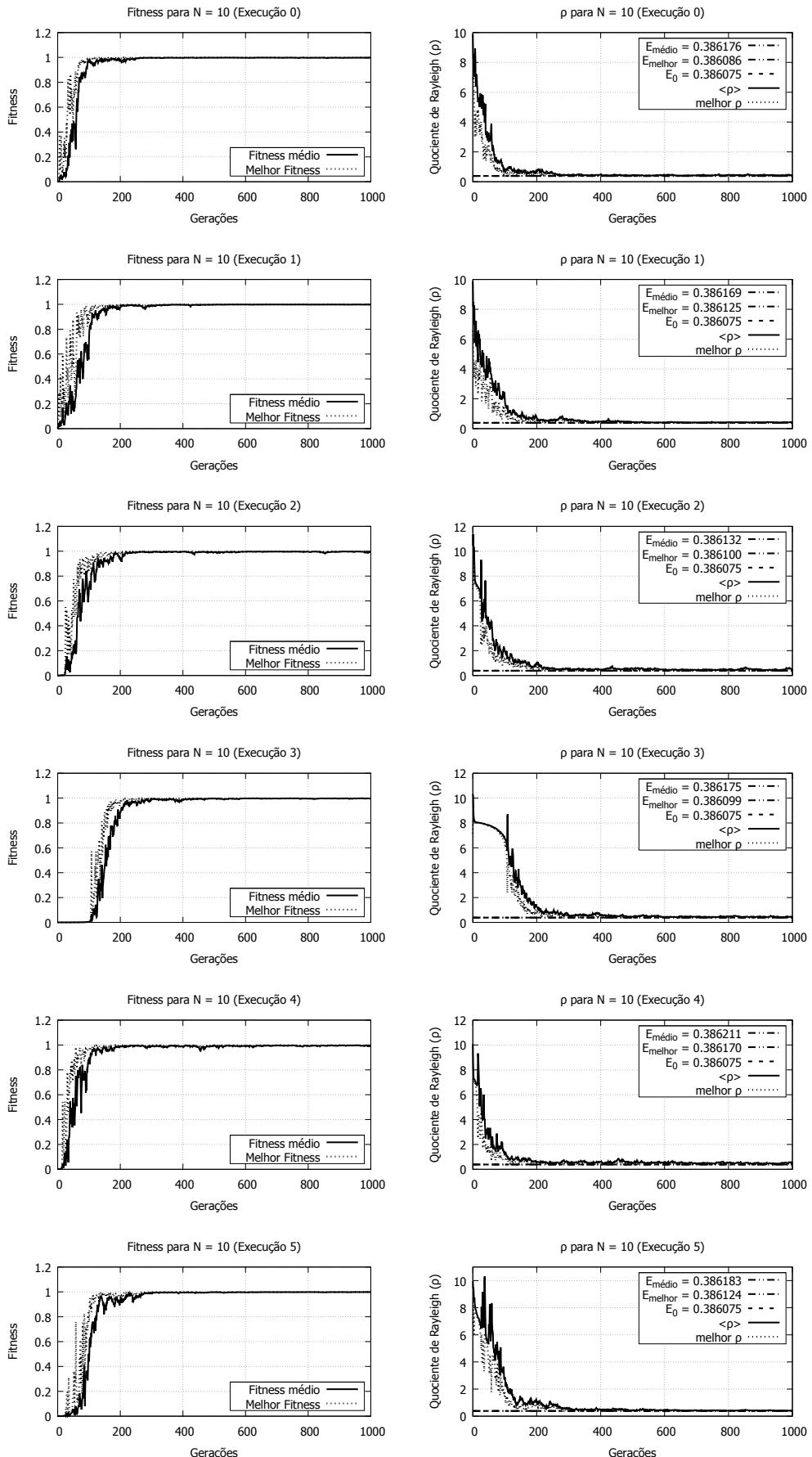
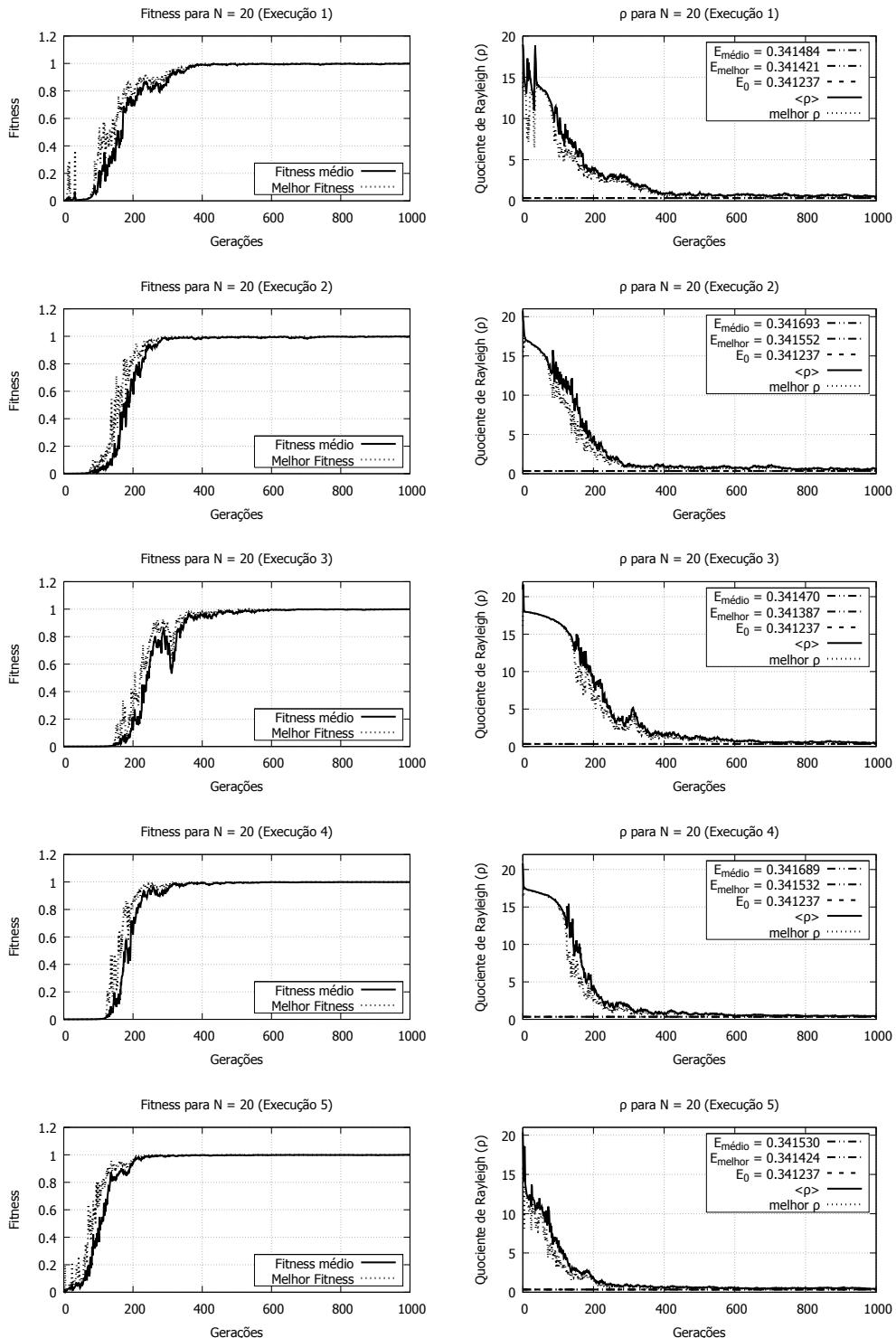


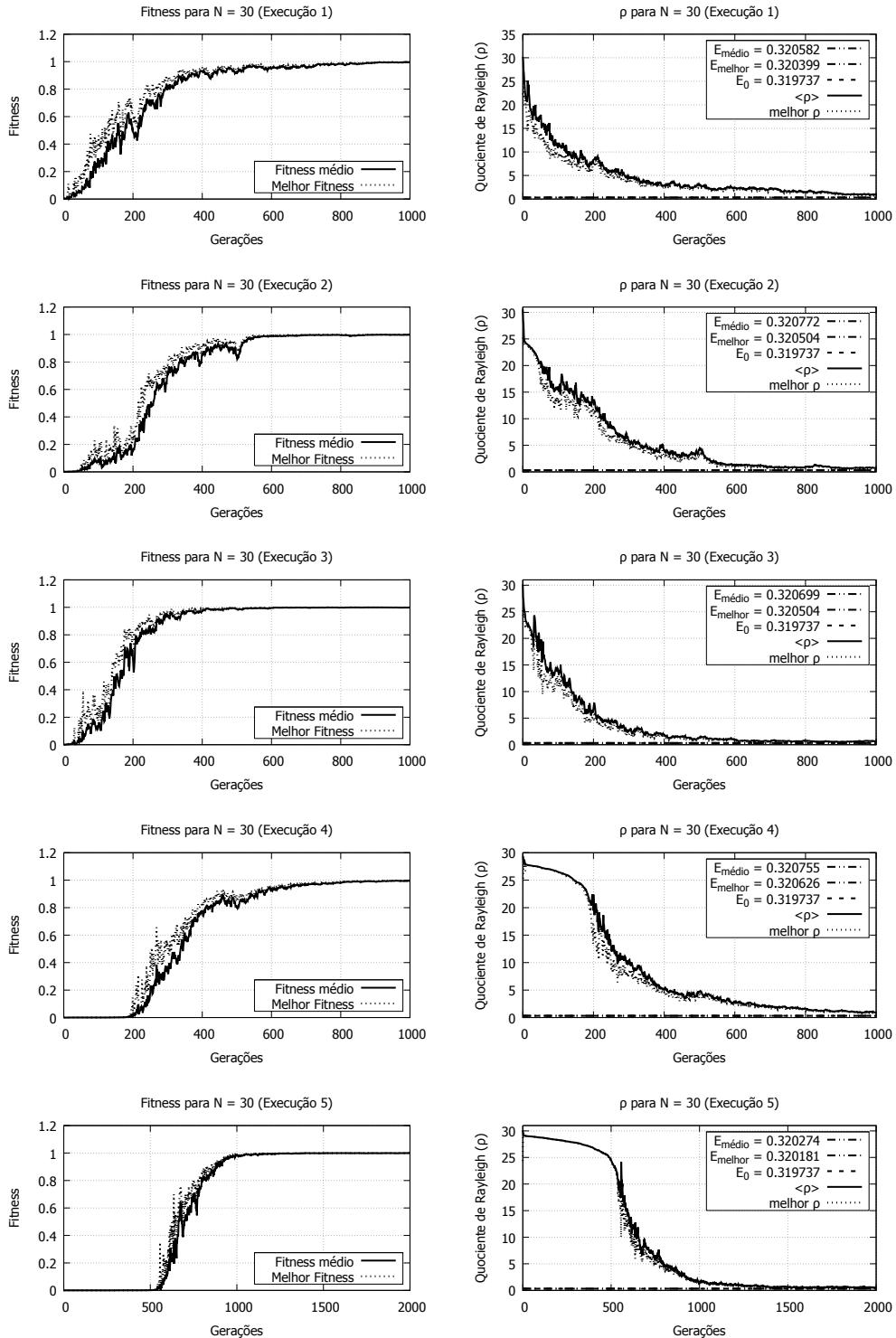
Figura 33 – Execuções N = 40.

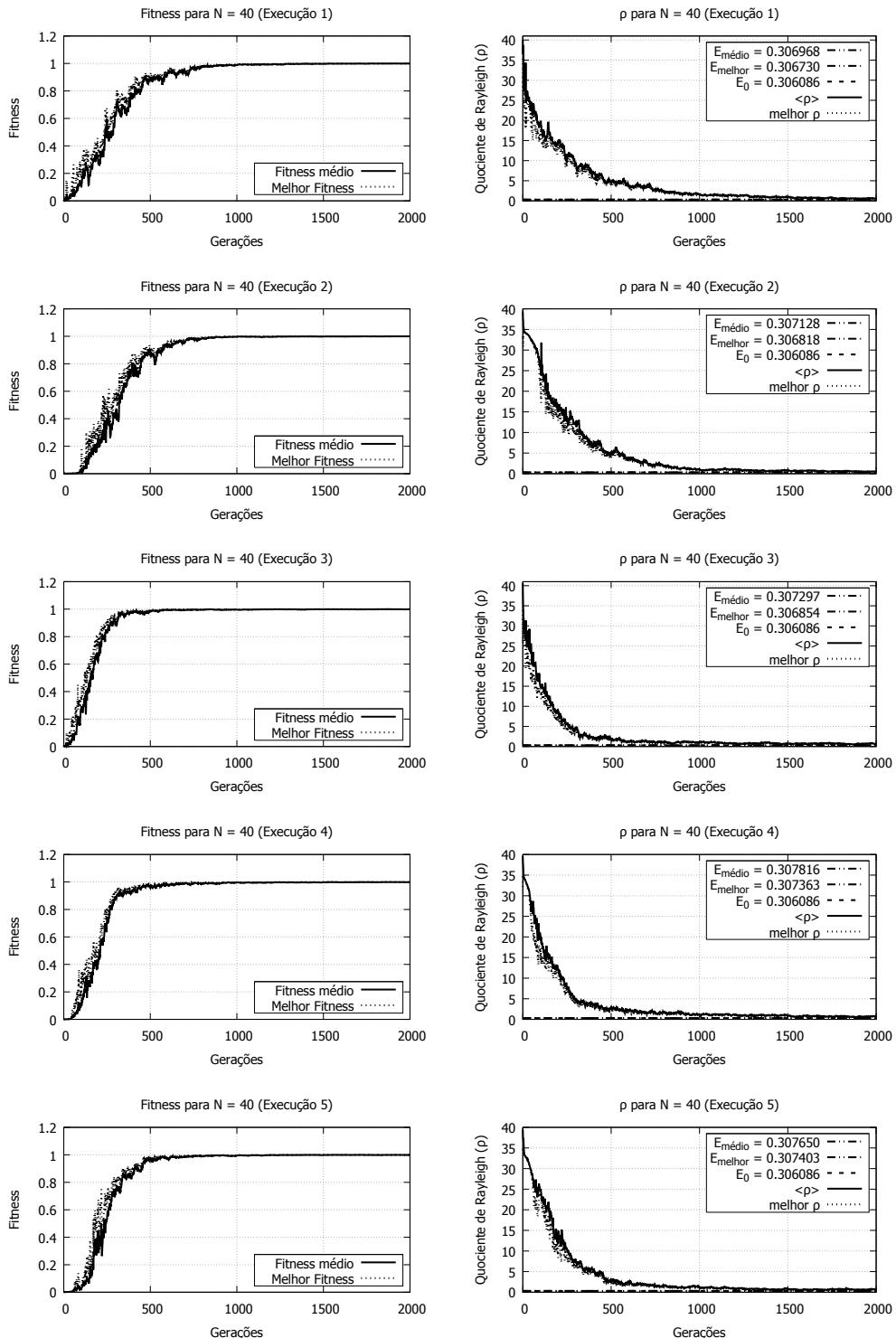
APÊNDICE C – Execuções para o *fitness*

$$f_i = e^{-\lambda(\rho_i - E_L)^2}$$

Figura 34 – Execuções para N = 10 com o fitness $f_i = e^{-\lambda(\rho_i - E_L)^2}$.

Figura 35 – Execuções para N = 20 com o fitness $f_i = e^{-\lambda(\rho_i - E_L)^2}$.

Figura 36 – Execuções para N = 30 com o fitness $f_i = e^{-\lambda(\rho_i - E_L)^2}$.

Figura 37 – Execuções para $N = 40$ com o fitness $f_i = e^{-\lambda(\rho_i - E_L)^2}$.

APÊNDICE D – Execuções para a variação de E_L em torno de E_0

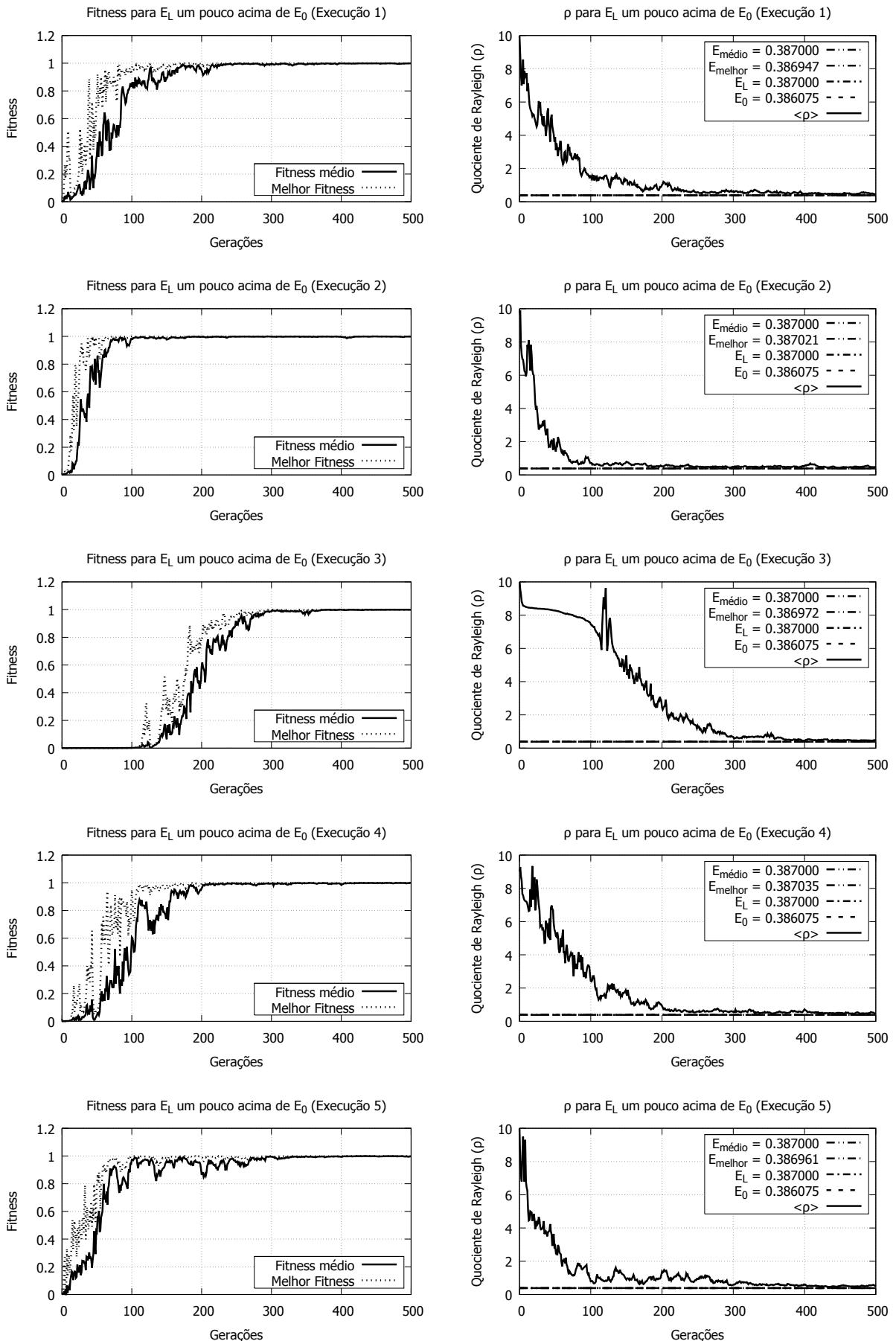


Figura 38 – Execuções com o E_L um pouco acima de E_0 no fitness $f_i = e^{-\lambda(\rho_i - E_L)^2}$. Semente 1445738835, $N = 10$.

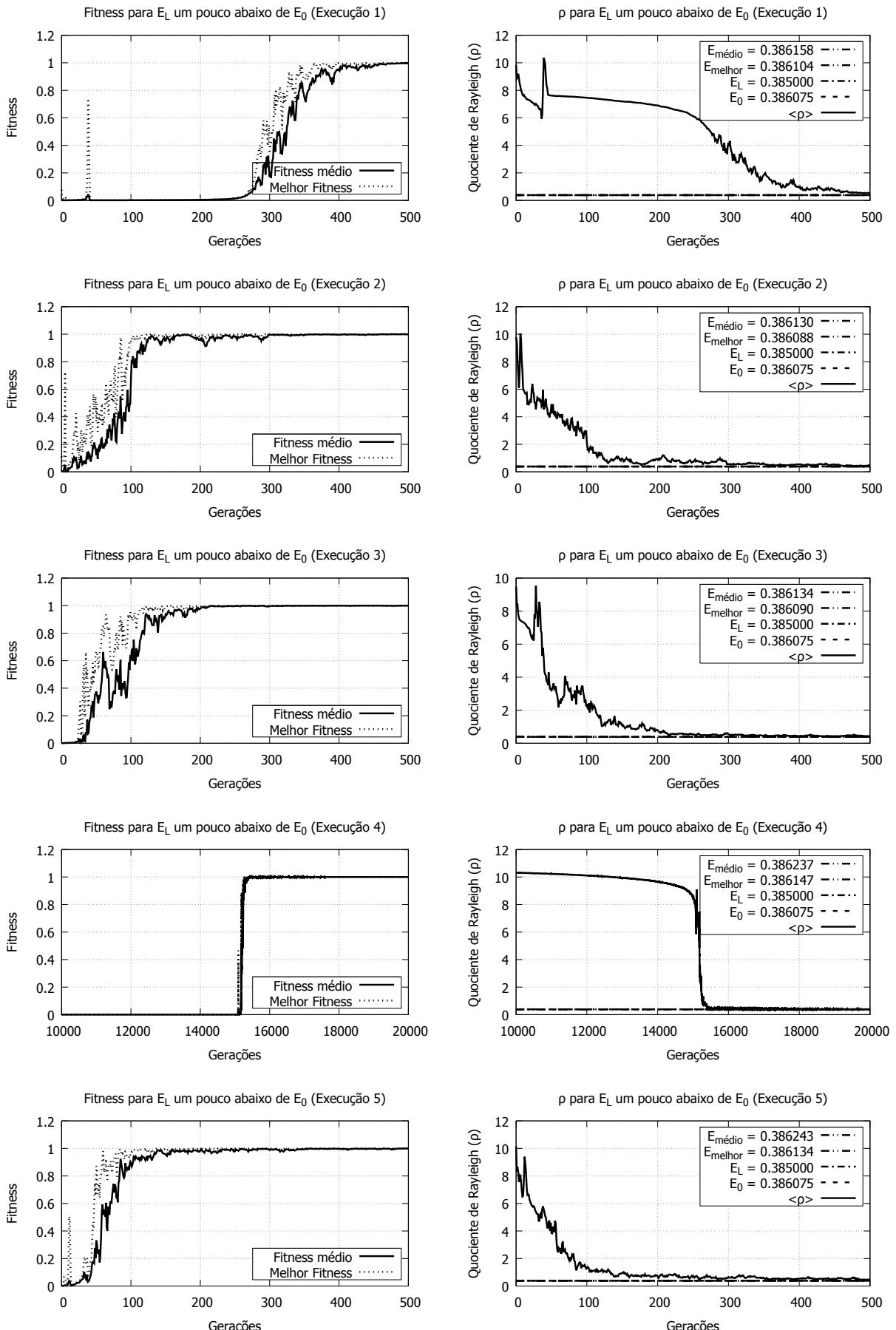


Figura 39 – Execuções com o E_L um pouco abaixo de E_0 no fitness $f_i = e^{-\lambda(\rho_i - E_L)^2}$. Semente 1445738835, N = 10.

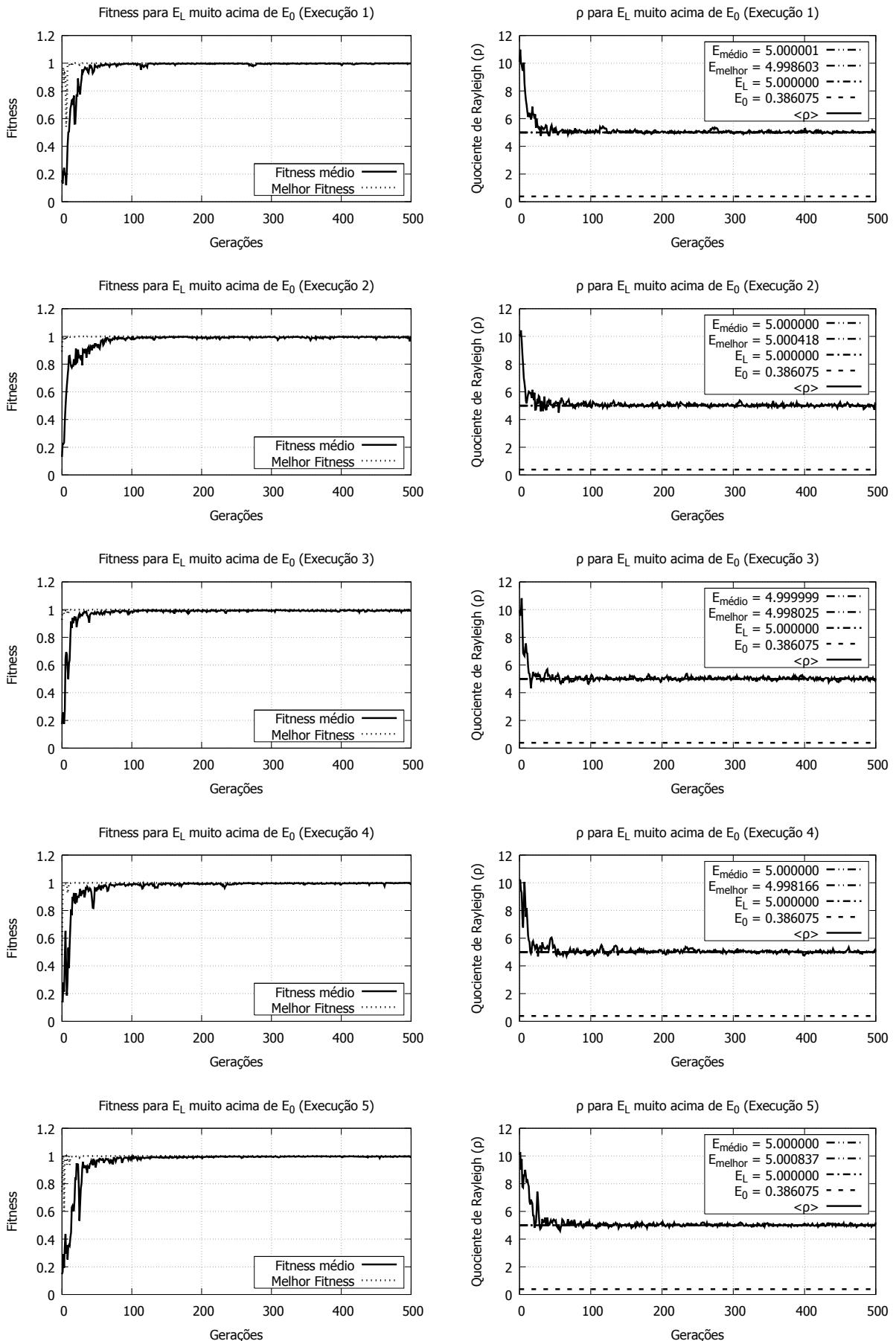


Figura 40 – Execuções com o E_L muito acima de E_0 no fitness $f_i = e^{-\lambda(\rho_i - E_L)^2}$. Semente 1445738835, N = 10.

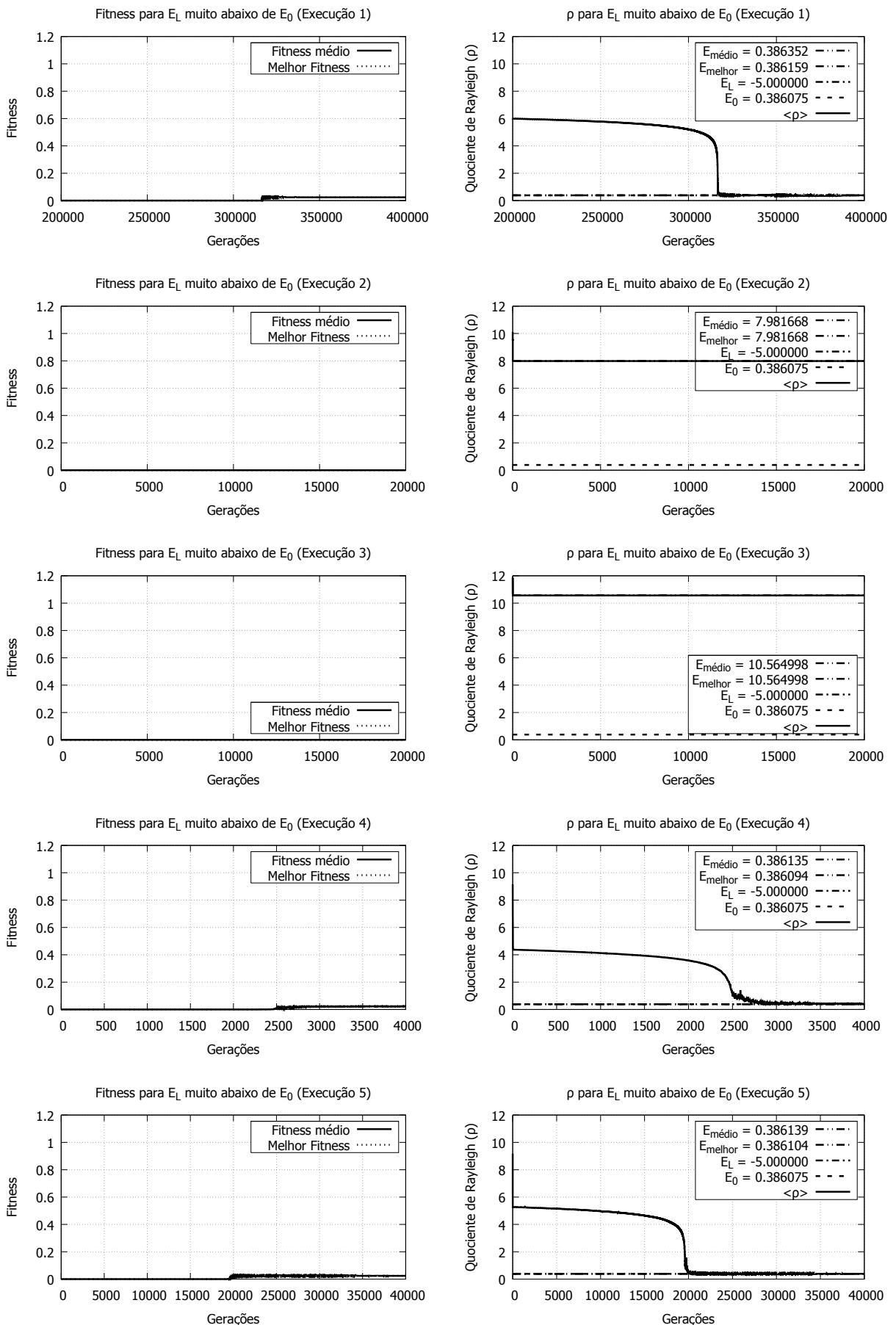


Figura 41 – Execuções com o E_L muito abaixo de E_0 no fitness $f_i = e^{-\lambda(\rho_i - E_L)^2}$. Semente 1445738835, N = 10.

Anexos

ANEXO A – Título do Anexo X

Texto aqui.

ANEXO B – Título do Anexo Y

Texto aqui.