

INSTITUTO TECNOLÓGICO DE AERONÁUTICA



Cássio dos Santos Sousa

Resolução otimizada de problemas com uso de algoritmos
evolutivos

*Trabalho de Graduação
2016*

Computação

Cássio dos Santos Sousa

**Resolução Otimizada de Problemas com uso de Algoritmos
Evolutivos**

Orientador
Prof. Dr. Carlos Henrique Quartucci Forster (ITA)

Engenharia de Computação

SÃO JOSÉ DOS CAMPOS
INSTITUTO TECNOLÓGICO DE AERONÁUTICA

2016

Dados Internacionais de Catalogação-na-Publicação (CIP)
Divisão de Informação e Documentação

Sousa, Cássio dos Santos

Resolução Otimizada de Problemas com uso de Algoritmos Evolutivos / Cássio dos Santos Sousa.

São José dos Campos, 2016.

15f.

Trabalho de Graduação – Divisão de Ciência da Computação –

Instituto Tecnológico de Aeronáutica, 2016. Orientador: Prof. Dr. Carlos Henrique Quartucci Forster.

1. Algoritmo Evolutivo.
 2. Inteligência Artificial.
- I. Instituto Tecnológico de Aeronáutica.
II. Resolução Otimizada de Problemas com uso de Algoritmos Evolutivos.

REFERÊNCIA BIBLIOGRÁFICA

SOUSA, Cássio dos Santos. **Resolução Otimizada de Problemas com uso de Algoritmos Evolutivos.** 2016. 15f. Trabalho de Conclusão de Curso. (Graduação) – Instituto Tecnológico de Aeronáutica, São José dos Campos.

CESSÃO DE DIREITOS

NOME DO AUTOR: Cássio dos Santos Sousa

TÍTULO DO TRABALHO: Resolução Otimizada de Problemas com uso de Algoritmos Evolutivos

TIPO DO TRABALHO/ANO: Graduação / 2016

É concedida ao Instituto Tecnológico de Aeronáutica permissão para reproduzir cópias deste trabalho de graduação e para emprestar ou vender cópias somente para propósitos acadêmicos e científicos. O autor reserva outros direitos de publicação e nenhuma parte desta monografia de graduação pode ser reproduzida sem a autorização do autor.

Cássio dos Santos Sousa

Cássio dos Santos Sousa

Pça Mal-do-Ar Eduardo Gomes, 50

12228-900 – São José dos Campos – SP

RESOLUÇÃO OTIMIZADA DE PROBLEMAS COM USO DE ALGORITMOS EVOLUTIVOS

Essa publicação foi aceita como Relatório Final de Trabalho de Graduação.

Cássio dos Santos Sousa
Autor

Prof. Dr. Carlos Henrique Quartucci Forster (ITA)
Orientador

Prof. Dr. Cecília de Azevedo Castro César
Coordenadora do Curso de Engenharia de Computação

São José dos Campos, _____ de _____ de 2016.

Dedico este trabalho à minha mãe Lucinez, que me trouxe suporte durante todo meu tempo no ITA, e ao Instituto Ismart, oportunidade que muda a minha vida desde 2006.

Agradecimentos

À minha família, que se esforçou tanto para que eu ficasse até o fim no ITA.

Ao meu professor orientador Carlos Forster, que me guiou por caminhos tortuosos até a conclusão deste trabalho de graduação.

Aos meus colegas de turma, que foram uma família a mais enquanto estive no ITA, e que despertaram em mim o desejo de sempre voar mais alto.

Aos demais professores da COMP, que nos acompanham desde antes do Curso Profissional, e que podem nos ver hoje como verdadeiros engenheiros.

E ao Instituto Ismart, que investe em minha formação acadêmica, profissional e pessoal desde 2006 com muito amor e carinho.

When you are inspired by some great purpose, some extraordinary project, all your thoughts break their bounds. Your mind transcends limitations, your consciousness expands in every direction and you find yourself in a new, great and wonderful world. Dormant forces, faculties and talents become alive, and you discover yourself to be a greater person by far than you ever dreamed yourself to be.

Patañjali, criador das práticas do Ioga.

Resumo

Algoritmos Evolutivos (AE) são de grande interesse na resolução de problemas complexos. Baseados nos conceitos biológicos de Evolução e na resolução por tentativa-e-erro, são capazes de obter boas respostas com performance superior a muitos algoritmos. Este trabalho promoveu a implementação e a análise de um Algoritmo Genético (AG), subgrupo dos AEs cuja informação é contida em genes, os quais compõem indivíduos que tentam resolver os problemas. O objetivo deste trabalho foi o de utilizá-lo na resolução otimizada de três problemas: OneMax Booleano, cujos genes são expressos por 0 ou 1; OneMax Real, cujos genes são expressos por uma variável real de 0 a 1; e uma adaptação do Problema do Caixeiro Viajante que permite utilizar atalhos entre as cidades. Para otimizar o AG, foram feitos dois acréscimos ao código. O primeiro deles foi o elitismo entre as gerações, mantendo o melhor indivíduo imune a variações. O segundo deles foi a implementação própria de um módulo extra chamado Algoritmo Genético Adaptativo (AGA), adicionado ao código do AG e responsável por atualizar o parâmetro de mutação de acordo com a evolução da população. Os três problemas foram simulados com uso do AG com parâmetros estáticos e com uso do AGA. As simulações decorrentes trouxeram diferenças pequenas do AGA comparado ao AG estático para os problemas OneMax com as mesmas entradas, e uma performance significativamente melhor do AGA para o problema do Caixeiro Viajante. Tais resultados nos levaram a concluir que o AGA utilizado é promissor na resolução de problemas.

Abstract

Evolutionary Algorithms (AEs) are of great interest in solving complex problems. Based on the biological concepts of Evolution and trial-and-error resolution, they are able to obtain good answers with performance superior to many algorithms. This work promoted the implementation and analysis of a Genetic Algorithm (GA), a subset of AEs whose information is contained in genes, which compose individuals who try to solve the problems. The objective of this work was to use it in the optimized resolution of three problems: Boolean OneMax, whose genes are expressed by 0 or 1; Real OneMax, whose genes are expressed by a real variable between 0 to 1; and an adaptation of the Traveling Salesman Problem that allows shortcuts between the cities. To optimize the GA, two additions were made to the code. The first was the elitism between generations, keeping the best individual immune to variations. The second one was the implementation of an extra module called Adaptive Genetic Algorithm (AGA), added to the GA code and responsible for updating the mutation parameter according to the evolution of the population. The three problems were simulated using both the GA with the static parameters and the AGA. The following simulations brought small differences from the AGA compared to the static GA for the OneMax problemas with the same inputs, and a significantly better performance of the AGA for the Traveling Salesman Problem. These results lead us to conclude the the AGA is promising in solving problems.

Sumário

Agradecimentos	p. 6
Resumo	p. 8
Abstract	p. 9
Lista de Figuras	p. 13
Lista de Tabelas	p. 15
Lista de Algoritmos	p. 16
Lista de Abreviaturas	p. 18
1 Introdução	p. 19
1.1 Objetivo	p. 20
1.2 Abordagem	p. 20
1.3 Plano de Trabalho	p. 21
1.4 Organização do Trabalho	p. 21
2 Problemas Escolhidos	p. 22
2.1 OneMax Booleano	p. 22
Gene	p. 22
Indivíduo	p. 23
Função de fitness	p. 23
2.2 OneMax Real	p. 23

Gene	p. 23
Indivíduo	p. 23
Função de fitness	p. 23
2.3 Caixeiro Viajante Adaptado	p. 24
Gene	p. 24
Indivíduo	p. 24
Função de fitness	p. 25
Algoritmo de Dijkstra	p. 26
3 Algoritmo Genético	p. 28
3.1 Parâmetros de entrada	p. 28
3.2 Seleção dos pais	p. 29
3.3 Recombinação / Crossover	p. 29
3.4 Mutação	p. 30
3.5 Sobrevivência	p. 30
4 Otimização do Algoritmo	p. 31
4.1 Elitismo	p. 31
4.2 Algoritmo Genético Adaptativo (AGA)	p. 32
5 Análises e Resultados	p. 35
5.1 Parâmetros analisados	p. 35
5.2 OneMax Booleano	p. 36
5.2.1 Caso Estático	p. 36
5.2.2 Caso Adaptativo	p. 36
5.3 OneMax Real	p. 39
5.3.1 Caso Estático	p. 40
5.3.2 Caso Adaptativo	p. 41

5.4 Caixeiro Viajante Adaptado	p. 41
5.4.1 Caso Estático ($p_m = 0.01$)	p. 43
5.4.2 Caso Estático ($p_m = 0.2$)	p. 45
5.4.3 Caso Adaptativo ($p_m = 0.01$)	p. 45
5.4.4 Caso Adaptativo ($p_m = 0.2$)	p. 47
5.5 Discussões	p. 50
6 Conclusões e Trabalhos Futuros	p. 53
6.1 Conclusões	p. 53
6.2 Trabalhos Futuros	p. 53
Referências	p. 55
Apêndice A – Genes Utilizados	p. 57
Apêndice B – Indivíduos dos problemas	p. 60
Apêndice C – Algoritmo Genético	p. 63
Apêndice D – Funções de fitness	p. 70
Apêndice E – Mapa das cidades - Caixeiro Viajante	p. 71
Apêndice F – Implementação do Algoritmo de Dijkstra	p. 74
Apêndice G – Funções de utilidade	p. 78

Lista de Figuras

1	Framework de um algoritmo evolutivo.	p. 19
2	Ação de crossover em dois pontos.	p. 29
3	Evolução dos valores de p_m e p_{m0} para o problema OneMax para valores diferentes de p_{m0} (em vermelho) ao longo de 500 gerações. Observa-se que o desvio (em verde) tenta sempre se igualar a p_{m0} , e p_m (em azul) varia de modo a permitir isso.	p. 34
4	Evolução do fitness para o problema do OneMax Booleano mostrando mínimo, máximo, valor médio e desvio padrão ($p_c = 0.9$, $p_m = 0.01$). Foram necessárias 78 gerações para que a solução ótima aparecesse, e 93 gerações para que a população inteira convergisse para ela.	p. 37
5	Evolução do fitness para o problema do OneMax Booleano Adaptativo mostrando mínimo, máximo, valor médio e desvio padrão ($p_c = 0.9$, $p_{m0} = 0.01$). Foram necessárias 140 gerações para que um indivíduo encontrasse a solução ótima.	p. 38
6	Evolução da probabilidade de mutação p_m ao longo das gerações, juntamente com o desvio do melhor valor de fitness com relação à média ($p_c = 0.9$, $p_{m0} = 0.01$).	p. 39
7	Evolução do fitness para o problema do OneMax Real mostrando mínimo, máximo e valor médio ($p_c = 0.9$, $p_m = 0.01$).	p. 40
8	Evolução do fitness para o problema do OneMax Real Adaptativo mostrando mínimo, máximo e valor médio ($p_c = 0.9$, $p_{m0} = 0.01$).	p. 43
9	Probabilidade de mutação ao longo das gerações para o problema do OneMax Real Adaptativo ($p_c = 0.9$, $p_{m0} = 0.01$).	p. 44
10	Evolução do fitness para o problema do Caixeiro Viajante Adaptado mostrando mínimo, máximo e valor médio ($p_c = 0.9$, $p_m = 0.01$). O menor caminho encontrado tem distância total de 2375.	p. 44

11	Desvio padrão ao longo das gerações para o problema do Caixeiro Viajante Adaptado ($p_c = 0.9, p_{m0} = 0.01$).	p. 45
12	Evolução do fitness para o problema do Caixeiro Viajante Adaptado mostrando mínimo, máximo e valor médio ($p_c = 0.9, p_m = 0.2$). O menor caminho encontrado tem distância total de 2300.	p. 46
13	Desvio padrão ao longo das gerações para o problema do Caixeiro Viajante Adaptado ($p_c = 0.9, p_{m0} = 0.2$).	p. 47
14	Evolução do fitness para o problema do Caixeiro Viajante Adaptado Adaptativo, mostrando mínimo, máximo e valor médio ($p_c = 0.9, p_{m0} = 0.01$). O menor caminho encontrado tem distância total de 2381.	p. 47
15	Desvio padrão ao longo das gerações para o problema do Caixeiro Viajante Adaptado Adaptativo ($p_c = 0.9, p_{m0} = 0.01$).	p. 48
16	Probabilidade de mutação ao longo das gerações para o problema do Caixeiro Viajante Adaptado Adaptativo ($p_c = 0.9, p_{m0} = 0.01$).	p. 48
17	Evolução do fitness para o problema do Caixeiro Viajante Adaptado Adaptativo, mostrando mínimo, máximo, valor médio e desvio padrão ($p_c = 0.9, p_{m0} = 0.2$). O menor caminho encontrado tem distância total de 2160.	p. 49
18	Probabilidade de mutação ao longo das gerações para o problema do Caixeiro Viajante Adaptado Adaptativo ($p_c = 0.9, p_{m0} = 0.2$).	p. 50

Lista de Tabelas

1	Dados coletados do problema do OneMax Booleano ($p_m = 0.01$).	p. 39
2	Dados coletados do problema do OneMax Real ($p_m = 0.01$).	p. 41
3	Dados coletados do problema do Caixeiro Viajante Adaptado.	p. 50

Lista de Algoritmos

1	Pseudocódigo do Algoritmo de Dijkstra.	p. 26
2	Pseudocódigo de um Algoritmo Evolutivo.	p. 28
3	Pseudocódigo do Algoritmo Genético Adaptativo (AGA).	p. 33

Lista de Listagens

5.1	Mapa de cidades para o problema do Caixeiro Viajante Adaptado.	p. 42
A.1	Genes Utilizados (genes.py).	p. 57
B.1	Indivíduos dos problemas (individuals.py).	p. 60
C.1	Algoritmo Genético (geneflow.py).	p. 63
D.1	Funções de fitness (fitness.py).	p. 70
E.1	Arquivo com mapa das cidades - Caixeiro Viajante (dijkstra17.py). . .	p. 71
F.1	Arquivo com algoritmo de Dijkstra (dijkstra.py).	p. 74
G.1	Funções de utilidade (utils.py)	p. 78

Listas de Abreviaturas

AE Algoritmo Evolutivo

AG Algoritmo Genético

AGA Algoritmo Genético Adaptativo

1 *Introdução*

A resolução de diversos problemas se dá na forma de algoritmos, de instruções bem definidas. No entanto, alguns algoritmos podem pedir inúmeras instruções até concluírem, o que pode até mesmo inviabilizar a solução encontrada. A Inteligência Artificial pode atuar sobre tais problemas de modo a interagir com o problema e aprender com ele a encontrar uma solução. Ótimos candidatos para esta tarefa são os chamados *Algoritmos Evolutivos (AE)*.

Algoritmos evolutivos são aqueles que se baseiam nos princípios de evolução natural da Biologia, e são aplicados em um modo particular de solução de problemas: o da tentativa-e-erro [6]. Tais algoritmos seguem um framework mais ou menos comum, atuante sobre diferente *gerações* de um problema, por meio de mudanças e combinações de *indivíduos* existentes numa *população*. Tal framework, conjuntamente com as ações evolutivas, pode ser visto na figura 1.



Figura 1: Framework de um algoritmo evolutivo.

Cada indivíduo está tentando resolver o problema durante a execução do algoritmo evolutivo. A população contém estes indivíduos e é o alvo de interesse do processo evolutivo, e uma geração é a população que sobrevive após um ciclo de processos evolutivos do Algoritmo Evolutivo (AE) sobre o problema. Similar ao processo evolutivo, seus com-

ponentes principais são as operações de variação (mutação e recombinação) e de seleção (seleção da geração pai e sobrevivência), chamadas aqui simplesmente de *operações de evolução* [5].

Este trabalho se utilizou de um grupo específico de Algoritmo Evolutivo, chamado de **Algoritmo Genético (AG)**. Um AG possui, como a menor estrutura de seus indivíduos, o *gene*. Um gene costuma ter apenas duas propriedades: uma expressividade, normalmente associada a um valor numérico, e uma forma de mudá-la aleatoriamente.

Para um AG, a *mutação* é uma mudança não controlada de um indivíduo feita a partir da mudança na expressividade de um ou mais genes. A *recombinação* envolve a mistura de genes vindos de dois indivíduos que são cruzados. A *seleção* envolve uma escolha a dedo dos melhores exemplares para cruzamento (a geração pai). A *sobrevivência* envolve a rejeição de indivíduos que não estejam aptos o suficiente para resolver o problema. Tal aptidão é normalmente associada a uma função definida conjuntamente com o problema, chamada de *função de fitness*.

1.1 Objetivo

Este trabalho propôs implementar um algoritmo genético capaz de resolver problemas determinados de diferentes complexidades e encontrar boas soluções após uma quantidade razoável de gerações. Apesar de se considerar performance, a prioridade aqui foi buscar boas soluções.

1.2 Abordagem

Em termos de implementação, o AG deve ser tal que, uma vez aplicado sobre um problema e uma população, as gerações se desenvolvam automaticamente. O trabalho foi então dividido em 4 etapas:

- [1] Escolha e implementação de problemas compatíveis com a aplicação do AE;
- [2] Implementação do AG e de suas operações de evolução;
- [3] Otimização do AG;
- [4] Análises e coleta de dados.

As primeiras duas etapas são interdependentes, e precisaram ser completadas primeiro e conjuntamente. As demais etapas foram feitas sequencialmente, e foi encima da última etapa que as conclusões foram feitas.

De modo a permitir a reutilização deste código em outros problemas, o AG foi feito de modo a ser compatível com mais de um problema. As ferramentas de análise e coleta de dados consideraram tanto métricas comuns da literatura quanto parâmetros específicos dos problemas abordados.

Optou-se por utilizar Python como linguagem principal do código feito para este trabalho.

1.3 Plano de Trabalho

Este trabalho foi planejado de tal forma que as etapas de otimização e análise do AG demorassem mais tempo. A validação do algoritmo foi feita com base na comparação de performance com otimização e sem otimização, de modo a se ter maior controle da análise final.

1.4 Organização do Trabalho

- **Capítulo 1:** Introdução
- **Capítulo 2:** Problemas Escolhidos
- **Capítulo 3:** Algoritmo Genético
- **Capítulo 4:** Otimização do Algoritmo
- **Capítulo 5:** Análises e Resultados
- **Capítulo 6:** Conclusões e Trabalhos Futuros

2 Problemas Escolhidos

Por mais que a estrutura básica de um algoritmo evolutivo seja capaz de resolver múltiplos problemas, é importante que ele seja validado por problemas de diferentes naturezas. Para isso, este trabalho focou sua atenção na resolução de três problemas com implementações diferentes para a construção do algoritmo genético.

Como o AG atua diretamente com populações, um problema deve defini-las de antemão, tanto em termos de indivíduo quanto em termos de gene. Conjuntamente, é necessário definir quão apto um indivíduo está frente à solução que ele propõe. Isso é feito por meio da *função de fitness*.

Três problemas foram escolhidos: OneMax Booleano [9], OneMax Real e uma adaptação do problema do Caixeiro Viajante [1]. Os dois primeiros foram escolhidos pela facilidade de implementação e de encontro de soluções ótimas, como será detalhado a seguir, o que permite testar hipóteses e heurísticas de modo bem mais simples. O terceiro problema foi escolhido por não só ter uma literatura rica, mas também por ser um problema complexo, cujos resultados podem ser analisados e comparados de modo mais rico.

2.1 OneMax Booleano

Dado um conjunto de 100 bits iniciados em 0, o AG deve ser capaz de tornar todos os bits iguais a 1.

Gene

Utilizou-se aqui o *BooleanGene*, um gene com expressividade booleana (0 ou 1). Sua operação de mutação consiste numa operação semelhante a jogar cara-e-coroa, trocando o valor expresso para 0 ou 1 aleatoriamente, com igual probabilidade.

Indivíduo

Cada indivíduo tentará resolver o problema, o que faz com que cada indivíduo tenha 100 genes do tipo BooleanGene.

Função de fitness

Conta-se o número de genes de um indivíduo que sejam iguais a 1. Quanto maior a contagem, melhor.

2.2 OneMax Real

Dado um conjunto de 100 variáveis reais iniciadas em 0, o AG deve ser capaz de tornar todas elas o mais próximo de 1. Este OneMax possui uma caminhada bem mais lenta que o anterior, pois um gene booleano possui apenas dois estados, o que faz com que as ações de mutação permitam uma evolução muito mais rápida, enquanto um gene real muda sua expressividade num espectro bem maior.

Gene

Utilizou-se aqui o *RealGene*, um gene com expressividade real entre 0.0 e 1.0. Sua operação de mutação consiste numa escolha aleatória de um número real no intervalo [0.0, 1).

Indivíduo

Cada indivíduo tentará resolver o problema, o que faz com que cada indivíduo tenha 100 genes do tipo RealGene.

Função de fitness

Soma-se a expressividade de todos os genes de um indivíduo. Quanto maior a contagem, melhor. Feita de maneira apropriada, esta função pode ser a mesma utilizada para o OneMax Booleano.

2.3 Caixeiro Viajante Adaptado

Dado um conjunto de cidades e as distâncias entre elas, o AG deve ser capaz de descobrir qual o menor caminho que possibilita a um caixeiro visitar todas as cidades e retornar à cidade original. Tal problema é NP-Hard, e avaliar se uma solução candidata é algo tão complexo quanto a resolução do problema em si.

Este problema é uma adaptação do original por não obrigar ao problema que as cidades sejam visitadas uma única vez. Isso permite que um indivíduo possa atravessar duas cidades através de um atalho que passe por outras cidades (com isso, o grafo não precisa ser completo).

Gene

Utilizou-se o *IntegerGene*, um gene com expressividade inteira entre 0 e K-1, com uma operação de mutação capaz de escolher aleatoriamente um valor inteiro neste intervalo. A inicialização deste gene possui K como parâmetro.

Indivíduo

No caso de um indivíduo do problema do Caixeiro Viajante, foi pensado que o mesmo deveria ser capaz de gerar, a partir da expressividade de seus genes, um percurso que passasse uma única vez por todas as cidades. Para isso, os genes aqui foram organizados de modo um pouco diferente dos problemas OneMax.

Digamos, por exemplo, que um caixeiro na cidade A precise passar pelas cidades [B, C, D, E, F] e voltar à cidade A. O indivíduo de tal problema teria então quatro genes (o número total de cidades, menos dois) criados da seguinte forma:

- O primeiro gene possui expressividade de 0 a 4;
- O segundo gene possui expressividade de 0 a 3;
- O terceiro gene possui expressividade de 0 a 2;
- O quarto gene possui expressividade de 0 a 1.

Digamos que um dos indivíduos do AG tenha, pela expressividade de seus genes, os valores [3, 0, 1, 0]. Para se calcular o percurso feito por tal indivíduo, escolhe-se a cidade

da lista naquela posição, a qual é removida antes de se escolher a próxima cidade. Ou seja:

- Gene 1: [3] mapeia a cidade E na lista [B, C, D, E, F]. Sem ela, a lista se torna [B, C, D, F];
- Gene 2: [0] mapeia a cidade B na lista [B, C, D, F]. Sem ela, a lista se torna [C, D, F];
- Gene 3: [1] mapeia a cidade D na lista [C, D, F]. Sem ela, a lista se torna [C, F];
- Gene 4: [0] mapeia a cidade C na lista [C, F]. Sem ela, a lista se torna [F].

Como [F] foi a única cidade que tais genes não escolheram, ela será visitada por último. Com isso, o indivíduo com genes [3, 0, 1, 0] traz o percurso $A \rightarrow E \rightarrow B \rightarrow D \rightarrow C \rightarrow F \rightarrow A$.

O percurso inicial terá sempre genes com expressividade zero. No exemplo fornecido, o caminho inicial (trazido por [0, 0, 0, 0]) de todos os indivíduos seria $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow F \rightarrow A$.

Função de fitness

A função de fitness aqui calcula a distância percorrida pelo caixeiro no trajeto completo trazido pelo indivíduo, considerando sempre o menor caminho a ser percorrido entre quaisquer duas cidades. Isso é trazido pelo uso do algoritmo de Dijkstra [4] no grafo constituído pelas cidades. Seu uso será detalhado melhor a seguir, mas quanto menor a distância que um indivíduo encontrar, melhor.

Como parte do problema do Caixeiro Viajante é o de encontrar um trajeto de menor custo, não é dito à função de fitness qual é a menor distância que o grafo possui.

Algoritmo de Dijkstra

O algoritmo de Dijkstra possui o seguinte pseudocódigo [2]:

Algoritmo 1: Pseudocódigo do Algoritmo de Dijkstra.

```

Dijkstra(Grafo, cidade) begin
    Inicializar( $Q$ );
    Inicializar( $dist$ );
    Inicializar( $prev$ );
    foreach  $cidade v$  no  $Grafo$  do
         $dist[v] \leftarrow \infty$ ;
         $prev[v] \leftarrow desconhecido$ ;
         $Q.adicionar(v)$ ;
    end
     $dist[cidade] \leftarrow 0$ ;
    while  $Q$  não estiver vazio do
         $u \leftarrow VerticeEmQComMenorDist(Q, dist)$ ;
         $Q.remover(u)$ ;
        foreach vizinho  $w$  de  $u$  do
             $atalho \leftarrow dist[u] + DistanciaEntre(u, w)$ ;
            if  $atalho < dist[w]$  then
                 $dist[w] \leftarrow atalho$ ;
                 $prev[w] \leftarrow u$ ;
            end
        end
    end
    return  $dist, prev$ ;
end

```

A complexidade de tal algoritmo, por possuir um loop dentro de outro, é, para o pior caso, $O(N^2)$, sendo N o número de cidades. No entanto, ele só descobre a menor distância tendo como referência a cidade utilizada como parâmetro. Por conta disso, o algoritmo precisa ser rodado uma vez para cada cidade, trazendo uma complexidade total $O(N^3)$ para o seu uso.

O requerimento para este algoritmo convergir é o de que a distância entre quaisquer duas cidades seja maior que zero, e que a distância de uma cidade para ela mesma (se exis-

tir) seja zero. Para ser utilizada com o AG, considerou-se também um grafo inicial conexo, de tal forma que qualquer percurso sugerido por um indivíduo fosse sempre possível.

O algoritmo de Dijkstra buscará atalhos entre as cidades. Um indivíduo que propuser um trajeto, a princípio, não saberá dizer se há atalhos. Se houver um, ele será apresentado na tela, mas a função de fitness não irá considerar as cidades do atalho como visitadas. Esperou-se que, ao longo das gerações, tais cidades fossem escolhidas naturalmente pela população.

3 Algoritmo Genético

O pseudocódigo tradicional de um algoritmo evolutivo pode ser dado por [10]:

Algoritmo 2: Pseudocódigo de um Algoritmo Evolutivo.

```

AG(fitness) begin
    P  $\leftarrow$  InicializarPopulacao(fitness);
    P.fitness();
    t = 0;
    while t < número de gerações do
        Pais  $\leftarrow$  Selecao(P);
        Filhos  $\leftarrow$  Crossover(Pais);
        P  $\leftarrow$  P  $\cup$  Filhos;
        Mutacao(P);
        P.fitness();
        P  $\leftarrow$  Sobrevida(P);
    end
end

```

Cada uma destas funções principais pode ser feita de diferentes maneiras. No entanto, tais implementações independem do problema a ser resolvido, o que torna sua confecção e manutenção bem mais simples.

Para que este trabalho não fugisse muito de seu escopo, que é o de resolver problemas, as decisões feitas para o AG tentaram ser as mais simples possíveis frente à literatura existente.

3.1 Parâmetros de entrada

O AG possui uma série de algoritmos de entrada. No entanto, excetuando-se os parâmetros específicos de cada problema (como os associados à função de fitness ou ao

tipo de indivíduo que comporá a população), restam um total de 4 parâmetros:

- O número de indivíduos na população (μ);
- O número de gerações ($NGEN$);
- A probabilidade de crossover (p_c);
- A probabilidade de mutação (p_m).

As variáveis p_c e p_m serão discutidas a seguir. O valor de $NGEN$ variou de acordo com o problema e com sua complexidade, mas foi padronizado um valor de 200 gerações para cada execução do AG. O tamanho da população μ ficou padronizado como 100.

3.2 Seleção dos pais

A escolha dos pais foi feita ordenando a geração em função de seu valor de fitness, com os pais sendo pareados sequencialmente, dois a dois.

3.3 Recombinação / Crossover

Escolhidos os pais, avalia-se, com probabilidade p_c , se eles serão cruzados. Se sim, seus materiais genéticos serão misturados por uma ação chamada *crossover*.

O crossover envolve a troca de genes entre dois indivíduos. Este trabalho se utilizou do crossover de dois pontos, o qual divide as sequências em duas partes distintas, ocorrendo troca do material genético entre estas partes. Tal ação é mostrada na figura 2.



Figura 2: Ação de crossover em dois pontos.

Toda ação bem sucedida de crossover gera duas sequências de genes (filhas) a partir das sequências originais (pais), as quais são transformadas em indivíduos e adicionadas à população.

O valor normalmente associado à variável p_c na literatura costuma ficar entre 0.6 e 0.95 [3,5,14]. Padronizou-se o valor de 0.9 neste trabalho. Ele precisa ser alto para que os filhos tenham maiores chances de variar seus genes e sugerir melhores respostas ao longo das gerações.

Os problemas deste trabalho foram organizados de tal forma que os genes de cada indivíduo fossem capazes de interagir com o problema separadamente. Por conta disso, o crossover de dois pontos pode ser aplicado sem problemas nos indivíduos.

3.4 Mutação

A ação de mutação atua sobre todos os genes da população (pais e filhos), a qual avalia, com probabilidade p_m , se a expressividade destes genes será alterada. Se sim, o valor deste gene é alterado de acordo com a assinatura de mutação do gene, tal como foi explicado no capítulo anterior. Normalmente, esta mudança de valor é aleatória.

O valor associado à variável p_m deve ser baixo, pois um valor muito alto não seria capaz de permitir a convergência das soluções [3,5,14]. O valor padronizado neste trabalho foi de 0.01.

3.5 Sobrevivência

A ação de sobrevivência aqui também tentou ser relativamente simples. Os indivíduos (pais e filhos) após a mutação são ordenados de acordo com a função de fitness, e apenas os μ melhores indivíduos são mantidos. Ou seja, a população não aumenta de tamanho ao longo das gerações.

4 Otimização do Algoritmo

Como proposta deste trabalho, pensou-se em como seria possível otimizar um AG de modo a encontrar boas soluções em poucas gerações. Dado o pseudocódigo de um AE, é possível propor uma série de otimizações, desde aquelas voltadas à melhoria de processamento, como o processamento paralelo de indivíduos em uma dada geração, àquelas que otimizam cada uma das quatro operações evolutivas principais (seleção, recombinação, mutação e sobrevivência).

De modo geral, o que traz novas soluções ao problema são as operações de variação (recombinação e mutação), e por conta disso, foram as mais analisadas neste capítulo. Otimizações mais simples discutidas em outros trabalhos, mas que se mostraram eficazes no encontro ou manutenção de boas soluções, também foram discutidas neste capítulo.

A manutenção de boas soluções foi obtida pelo uso de *elitismo*. A otimização nas operações de variação foi obtida pelo uso dos chamados Algoritmos Genéticos Adaptativos (AGA).

4.1 Elitismo

O elitismo é a manutenção do indivíduo mais adaptado de uma geração, deixando-o imune a mutações para que a melhor solução não seja perdida [13]. Um indivíduo elitista ainda pode ser considerado para recombinação e geração de filhos, uma vez que as operações de mutação e recombinação são independentes. É possível também criar um grupo elitista, mantendo-se uma certa quantidade ou porcentagem de indivíduos imune a mutações.

Este trabalho utilizou elitismo para o melhor indivíduo em todas as execuções do AG. Tal propriedade pode ser desativada no código.

4.2 Algoritmo Genético Adaptativo (AGA)

A forma mais tradicional de implementação de um AG atribui valores estáticos aos parâmetros de entrada, incluindo os parâmetros de crossover e mutação. No entanto, os indivíduos buscarão soluções de acordo com estes dois parâmetros, e deixá-los estáticos pode limitar o alcance do AG e impedi-lo de encontrar soluções melhores.

Se fosse possível modificar tais parâmetros enquanto o AG é executado, de modo a se adaptar às mudanças de fitness dos próprios indivíduos, teríamos uma solução. Um bom candidato para isso são os chamados Algoritmos Genéticos Adaptivos (AGAs) [15].

O conceito por trás de um AGA envolve implementar em cima de um AG de modo a modificar os parâmetros de crossover e/ou mutação ao longo do tempo. Não obstante, é possível moldar um AGA de modo a tratar crossover e mutação com probabilidades diferentes para cada indivíduo, de acordo com seus valores de fitness.

Para este trabalho, optou-se por trabalhar com versões adaptadas de outros AGAs [11, 15, 16] e implementar uma versão própria, explicada a seguir:

- Apenas o parâmetro de mutação é modificado ao longo das gerações, uma vez que o crossover seja sempre um valor alto (como 0.9, padronizado neste trabalho);
- A adaptação de p_m acontece apenas depois que um ciclo de operações de evolução acontecer;
- O que decidirá se p_m mudará será o desvio do melhor valor de fitness f_{best} em comparação com o fitness médio \bar{f} , como na equação a seguir:

$$\left| \frac{f_{best} - \bar{f}}{\bar{f}} \right| \quad (4.1)$$

- Se este desvio for menor que um valor pp_{m0} , isso significa que a mutação está fraca e os indivíduos estão se aproximando de uma mesma solução, que pode não ser necessariamente a melhor. Para contornar isso, p_m irá aumentar;
- Caso contrário, as soluções estarão se desviando muito, o que pode ser resultado de uma mutação intensa. Para resolver isso, p_m irá diminuir;
- O valor de p_{m0} é o valor inicial de p_m para o valor inicial de p_{m0} , de modo a servir de termômetro para o valor do desvio;

- No entanto, p_m não pode ser igual a zero nem maior que 1, dado que representa uma probabilidade. Seguindo a linha de outros trabalhos [12], p_m será limitado ao intervalo [0.001, 0.5] (se p_m tentar extrapolar estes limites, ele retornará ao valor extremo mais próximo);
- O incremento/decremento para p_m será linear e igual a 0.001;
- Como há uma divisão por \bar{f} , se este valor for zero ou muito próximo de zero para alguma geração, este AGA não será executado.

Traduzindo-se a explicação para um algoritmo, chegamos ao código mostrado no algoritmo 3. Este trabalho avaliará o desempenho deste AGA comparando a evolução da população com e sem o uso do AGA para um mesmo valor inicial de p_m . O intuito não foi o de encontrar um AGA ideal, mas sim o de avaliar se o uso dele ajudaria ou não no encontro de soluções melhores.

Algoritmo 3: Pseudocódigo do Algoritmo Genético Adaptativo (AGA).

```

begin
     $p_{m0} \leftarrow$  (valor inicial de  $p_m$  na inicialização do AG);
     $\epsilon \leftarrow 0.0001$ ;
    foreach ciclo de operações de evolução do
         $\bar{f} \leftarrow$  (média dos valores de fitness);
        if  $\bar{f} < \epsilon$  then
            | return
        end
         $f_{best} \leftarrow$  (melhor valor de fitness na população);
         $desvio \leftarrow \left| \frac{f_{best} - \bar{f}}{\bar{f}} \right|$ ;
        if  $desvio \leq p_{m0}$  then
            |  $p_m \leftarrow \min(0.5, p_m + 0.001)$ ;
        end
        else
            |  $p_m \leftarrow \max(0.001, p_m - 0.001)$ ;
        end
    end
end

```

Para exemplificar o funcionamento do AGA, foram feitas simulações para diferentes valores de p_{m0} para o OneMax Booleano, na figura 3. É possível ver que p_m tenta sempre manter o desvio entorno de p_{m0} , incentivando a busca de soluções diferentes para o sistema.

A ideia por trás de uma implementação própria foi a de testar a implementação de um AGA a partir de conceitos mais simples. Se a ideia de adaptação de um AGA, conforme

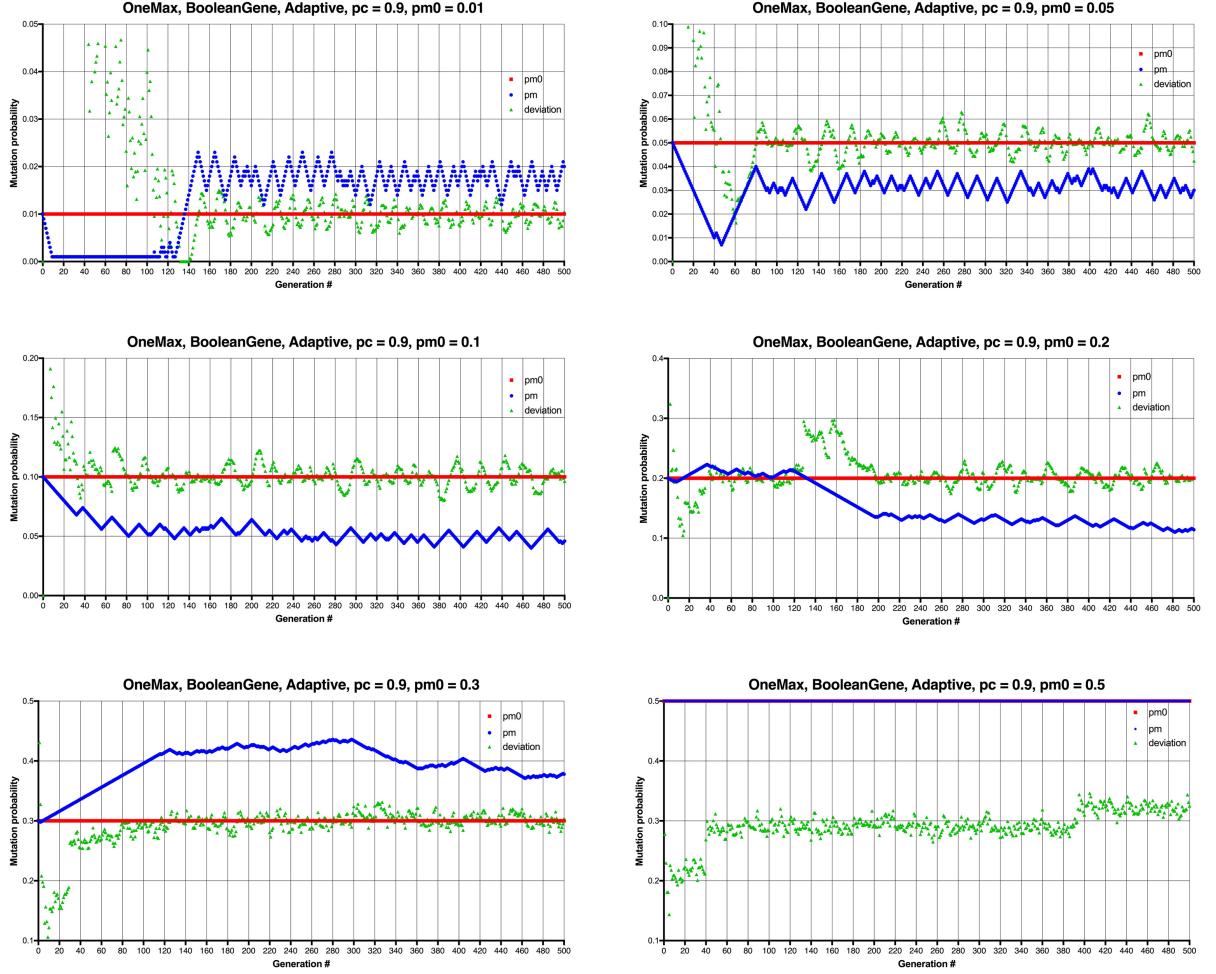


Figura 3: Evolução dos valores de p_m e p_{m0} para o problema OneMax para valores diferentes de p_{m0} (em vermelho) ao longo de 500 gerações. Observa-se que o desvio (em verde) tenta sempre se igualar a p_{m0} , e p_m (em azul) varia de modo a permitir isso.

vista na literatura, for tão simples quanto a base evolutiva do AG, sua implementação também deve buscar algo simples.

5 Análises e Resultados

5.1 Parâmetros analisados

Como um AG trabalha com tentativa-e-erro, não há uma geração exata para a qual pode ser prevista uma determinada solução (ou proximidade à solução). Para se analisar a evolução dos indivíduos ao longo das gerações, este trabalho focou em analisar os seguintes parâmetros:

- Valor mínimo de fitness em um indivíduo;
- Valor máximo de fitness em um indivíduo;
- Valor médio de fitness entre todos os indivíduos;
- Desvio padrão dos valores de fitness.

O desvio padrão utilizado aqui foi o amostral, dado por:

$$\sigma = \sqrt{\frac{1}{N-1} \sum_{i=1}^N (x_i - \bar{x})^2} \quad (5.1)$$

Os gráficos criados tiveram como base uma única simulação do AG para o problema e parâmetros de entrada escolhidos. Não é possível ter expectativas de valores ou gerações, uma vez que o AG trabalha com tentativa-e-erro, então criá-las ou assumi-las seria difícil de quantizar.

Como já foi comentado antes, se não for mencionado o contrário (mesmo para o caso adaptativo), utilizou-se 0.9 para p_c e 0.01 para p_m . Todas as execuções utilizaram 100 indivíduos, 200 gerações e elitismo do melhor indivíduo. Se um problema convergiu completamente para a melhor solução antes das 200 gerações completarem, os gráficos foram reduzidos para melhor visualização, uma vez que os pontos extras não trariam novas informações para nós.

As simulações feitas com uso do AGA foram utilizadas para comparação com a implementação estática do AG. Além disso, foi avaliado o comportamento adaptativo de p_m ao longo das gerações.

5.2 OneMax Booleano

5.2.1 Caso Estático

Este problema foi considerado simples de ser resolvido por um AG, uma vez que poucas mutações em um gene já permitem que ele ficasse igual a 1. A figura 4 demonstra exatamente isso, com convergência completa dos indivíduos para a solução ótima após 93 gerações, sendo que a própria solução ótima foi encontrada com a melhor solução aparecendo após 78 gerações.

O comportamento estático para os valores padronizados de p_c e p_m mostrou um crescimento aproximadamente linear dos valores de fitness, tanto para o pior quanto para o melhor indivíduo. Isso demonstra que tais valores demonstram um crescimento equilibrado da população, e a mutação existente é suficiente para evoluir os indivíduos.

5.2.2 Caso Adaptativo

Com o AGA ativado, ficou bem mais complicado para o OneMax atingir a solução ótima. Como mostrado na figura 5, foram necessárias 140 gerações para que o melhor indivíduo atingisse a solução ótima, e a média dos valores de fitness oscilou entre 98 e 99 para as últimas gerações. Em termos de desvio padrão, o valor nunca passou de 2.0 (de um máximo de 100), o que nos diz que a população não se dispersou completamente pelo efeito adaptativo.

Um efeito interessante pode ser visto na figura 6 com relação à evolução de p_m e p_{m0} . É possível ver que as primeiras gerações ainda eram muito dispersas, o que fez com que p_m começasse a cair. No entanto, como mencionado antes, um valor baixo de p_m se mostrou mais do que suficiente para que a população do caso estático fosse capaz de encontrar a solução ótima.

Por conta disso, p_{m0} começou a aumentar enquanto a população evoluía e ficava mais homogênea. O AGA, para uma população homogênea, interpreta isso como se a população tivesse travado em uma solução. Por conta disso, após cerca de 70 gerações, a homogeneização da população e o aumento de p_{m0} se encontraram, e p_m começou a

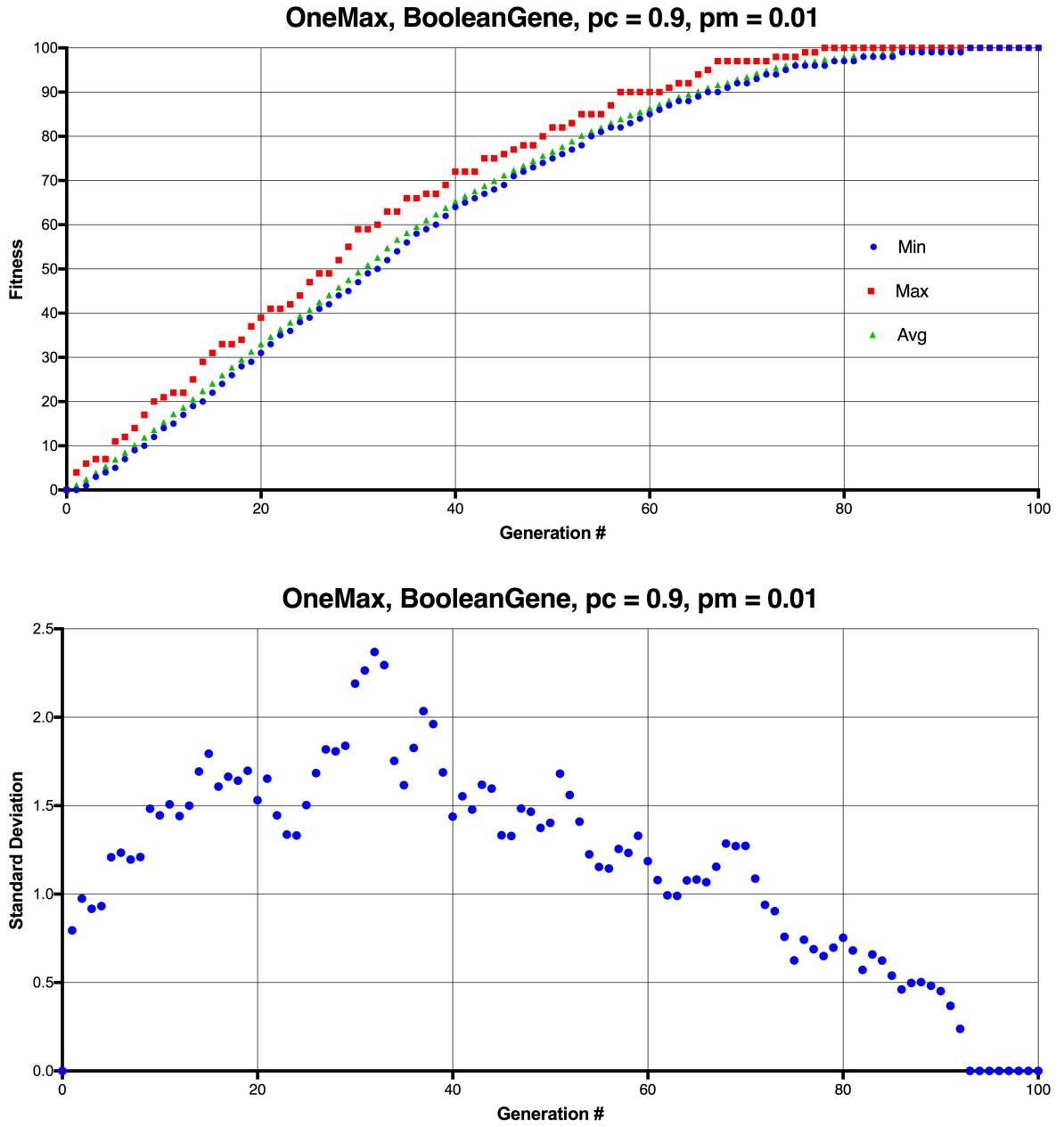


Figura 4: Evolução do fitness para o problema do OneMax Booleano mostrando mínimo, máximo, valor médio e desvio padrão ($p_c = 0.9$, $p_m = 0.01$). Foram necessárias 78 gerações para que a solução ótima aparecesse, e 93 gerações para que a população inteira convergisse para ela.

aumentar.

Quando p_m parou de crescer (por volta de 110 gerações), ele estabilizou em torno de um valor mais alto que o inicial (por volta de 0.03). Tal aumento (tanto de p_m quanto de p_{m0}) se mostrou suficiente para que a população fosse incapaz de convergir conjuntamente para a solução ótima.

Foi possível tirar desta simulação que, mesmo que o intuito inicial deste AGA tivesse

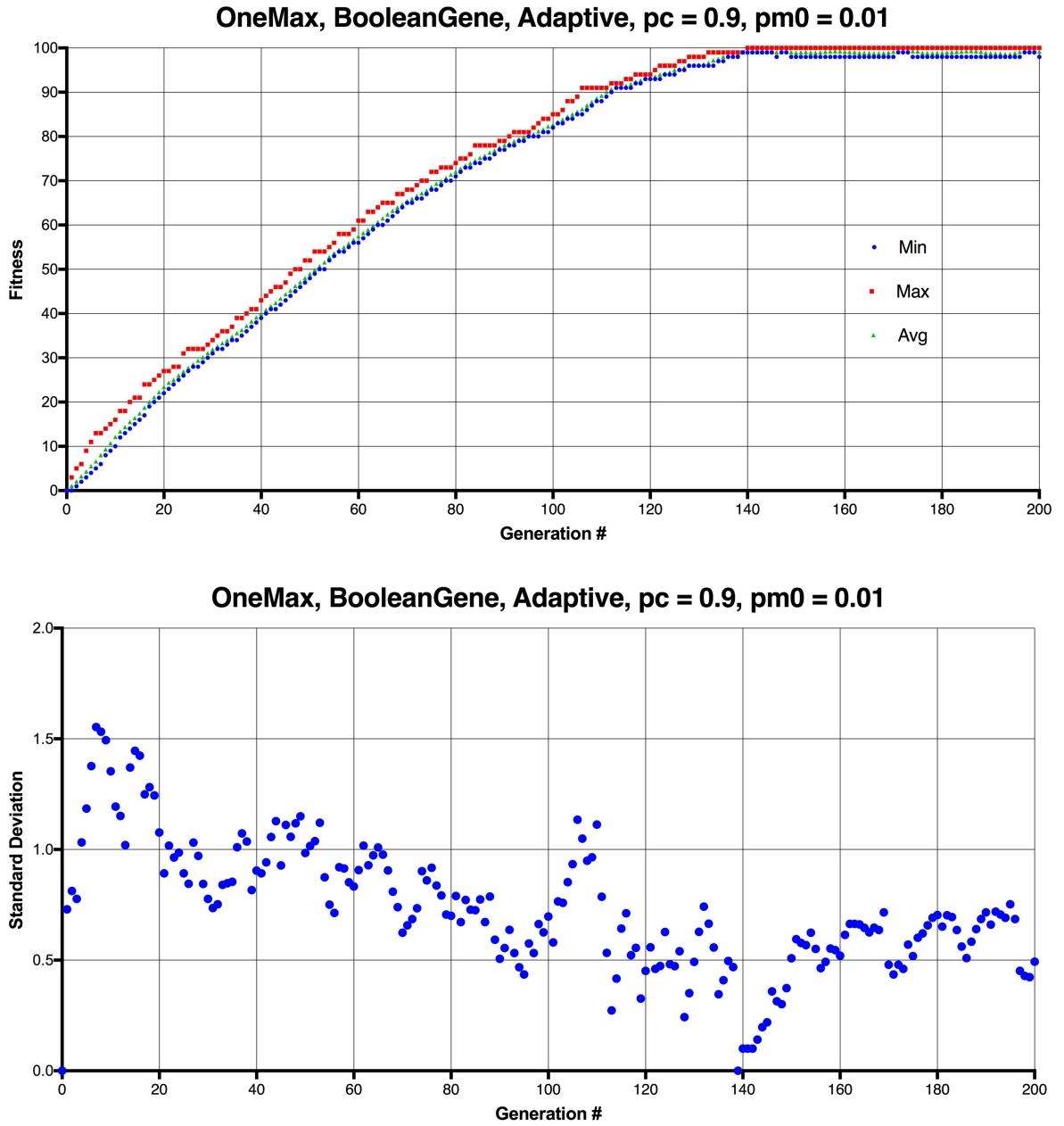


Figura 5: Evolução do fitness para o problema do OneMax Booleano Adaptativo mostrando mínimo, máximo, valor médio e desvio padrão ($p_c = 0.9$, $p_{m0} = 0.01$). Foram necessárias 140 gerações para que um indivíduo encontrasse a solução ótima.

sido o de fugir de momentos em que o AG travasse em uma solução, ele veio com um custo, que foi o de evitar que a população fosse capaz de convergir conjuntamente para a solução ótima. Se o intuito de usar o AG for o de encontrar uma boa solução no final, esse custo é baixo.

Os dados de interesse vindos destas simulações podem ser encontrados na tabela 1.

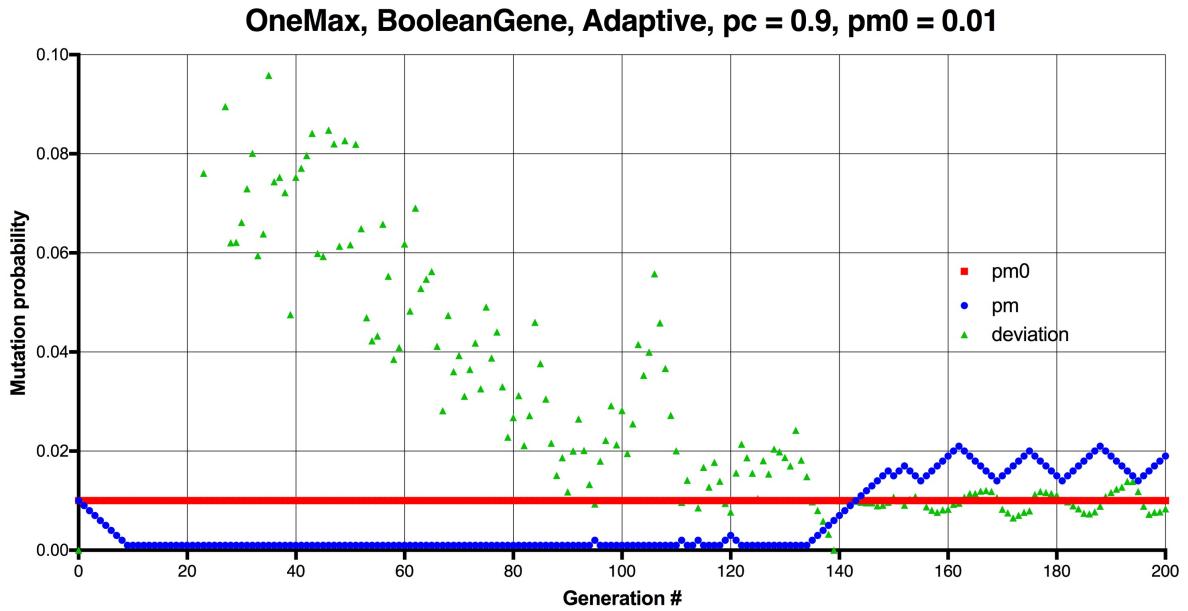


Figura 6: Evolução da probabilidade de mutação p_m ao longo das gerações, juntamente com o desvio do melhor valor de fitness com relação à média ($p_c = 0.9$, $p_{m0} = 0.01$).

5.3 OneMax Real

Para o OneMax Real, é virtualmente impossível chegar ao valor máximo de fitness (100.0), uma vez que a mutação para um número aleatório trabalha no intervalo $[0, 1]$. Por conta disso, as análises feitas aqui focaram no comportamento da curva e na diferença de comportamento frente aos resultados do OneMax Booleano.

Tabela 1: Dados coletados do problema do OneMax Booleano ($p_m = 0.01$).

Algoritmo analisado (AG = caso estático)	AG	AGA
Solução ótima encontrada?	Sim	Sim
Gerações p/solução ótima	78	140
Convergência da população (gerações)	93	--
Fitness médio após 100 gerações	100	82.67
Fitness médio após 200 gerações	100	99.17
Valor final de p_m	0.01	0.019
Valor mínimo de p_m	0.01	0.001
Valor máximo de p_m	0.01	0.021
Valor médio de p_m	0.01	0.00594
Valor médio de p_m (últimas 100 gerações)	0.01	0.0105

5.3.1 Caso Estático

Para o OneMax Real, foi possível ver, na figura 7, que a população, mesmo não sendo capaz de atingir o fitness máximo, conseguiu se homogeneizar e continuar crescendo ao longo das gerações, o que foi demonstrado também pela queda do desvio padrão.

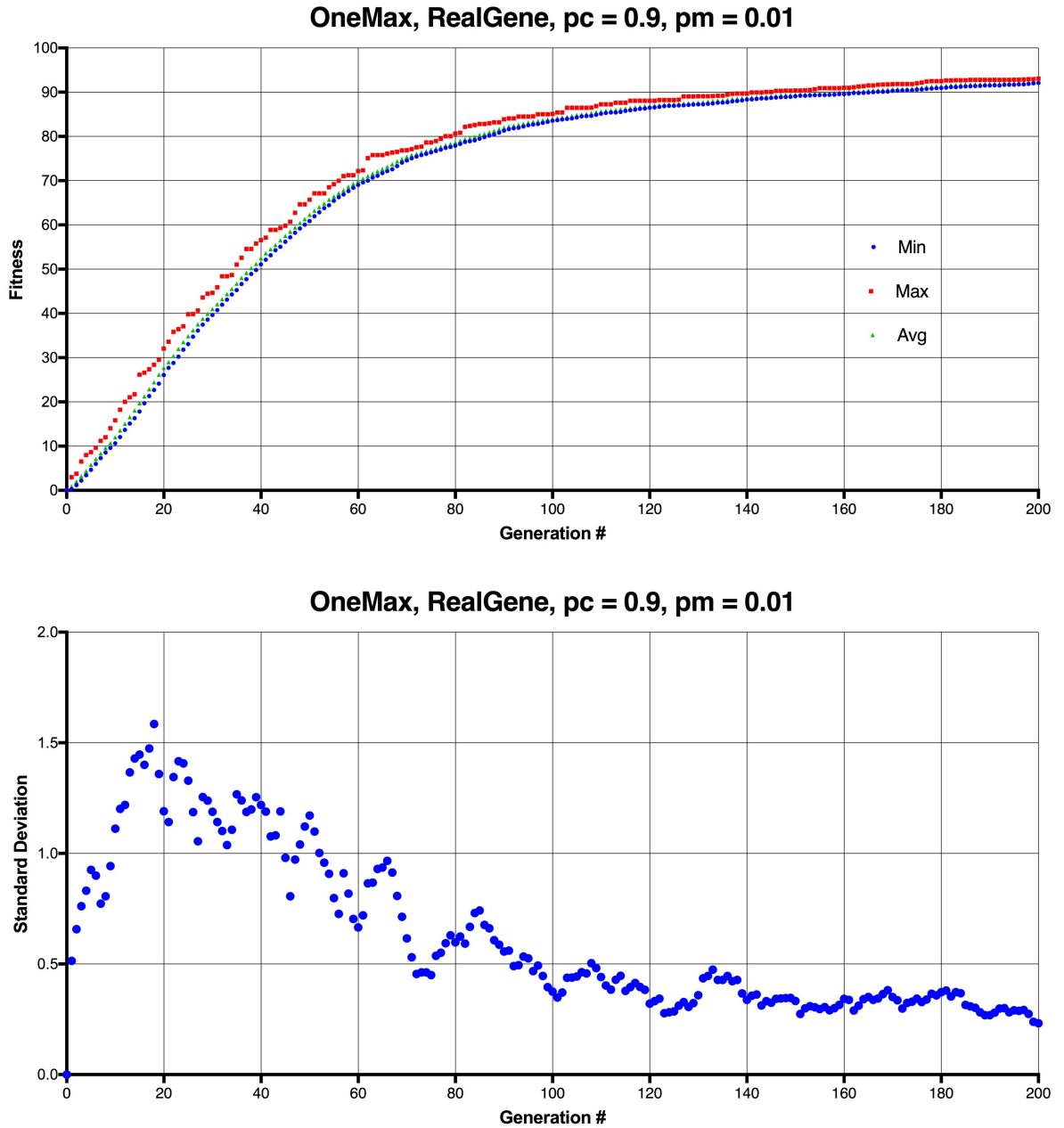


Figura 7: Evolução do fitness para o problema do OneMax Real mostrando mínimo, máximo e valor médio ($p_c = 0.9$, $p_m = 0.01$).

Os pontos de qualquer uma das medições (média, mínimo e máximo) parecem formar uma curva. Descobri-la fugiu do escopo deste trabalho, mas tal formato pode estar associado à probabilidade de se aumentar a expressividade de um gene. Na execução deste AG,

Tabela 2: Dados coletados do problema do OneMax Real ($p_m = 0.01$).

Algoritmo analisado (AG = caso estático)	AG	AGA
Fitness máximo após 100 gerações	85.115	72.452
Fitness médio após 100 gerações	84.048	70.186
Fitness máximo após 200 gerações	93.083	90.585
Fitness médio após 200 gerações	92.355	89.507
Valor final de p_m	0.01	0.009
Valor mínimo de p_m	0.01	0.001
Valor máximo de p_m	0.01	0.012
Valor médio de p_m	0.01	0.00300
Valor médio de p_m (últimas 100 gerações)	0.01	0.00458

cada gene é levado para mutação com probabilidade p_m . Se um gene tiver uma expressividade $0 < x < 1$, a mutação (assumida uniforme) possui uma probabilidade $(1 - x)$ de aumentá-la. Logo, a expressividade aumentará em uma dada geração com probabilidade:

$$p_m(1 - x) \quad (5.2)$$

5.3.2 Caso Adaptativo

Os efeitos notados para o OneMax Booleano se repetiram aqui. Como se pode ver nas figuras 8, a população se estabilizou após cerca de 110 gerações, o que indica que a mutação se estabilizou em um determinado valor.

Vemos também que o desvio padrão dos valores de fitness apareceu crescer ao longo das gerações. Isso se deu mais por conta do aumento de p_m após 73 gerações, mostrado na figura 9. Comparado ao OneMax Booleano, o p_m atingiu valores maiores próximo às últimas gerações (oscilando em torno de 0.04), o que pode ter contribuído para o aumento do desvio padrão, mesmo com a média dos valores de fitness se mantendo relativamente no mesmo valor.

Os dados de interesse vindos destas simulações podem ser encontrados na tabela 2.

5.4 Caixeiro Viajante Adaptado

Para as simulações deste problema, utilizou-se então o grafo do programa Open-Source DEAP com 17 cidades [7,8]. Tal grafo é conexo, completo e bidirecionado, e sua distância mínima (começando na primeira cidade) é 2085. Ele está presente na listagem 5.1.

Listagem 5.1: Mapa de cidades para o problema do Caixeiro Viajante Adaptado.

```
[0, 633, 257, 91, 412, 150, 80, 134, 259, 505, 353, 324, 70, 211,
 268, 246, 121],
[633, 0, 390, 661, 227, 488, 572, 530, 555, 289, 282, 638, 567, 466,
 420, 745, 518],
[257, 390, 0, 228, 169, 112, 196, 154, 372, 262, 110, 437, 191, 74,
 53, 472, 142],
[91, 661, 228, 0, 383, 120, 77, 105, 175, 476, 324, 240, 27, 182,
 239, 237, 84],
[412, 227, 169, 383, 0, 267, 351, 309, 338, 196, 61, 421, 346, 243,
 199, 528, 297],
[150, 488, 112, 120, 267, 0, 63, 34, 264, 360, 208, 329, 83, 105,
 123, 364, 35],
[80, 572, 196, 77, 351, 63, 0, 29, 232, 444, 292, 297, 47, 150, 207,
 332, 29],
[134, 530, 154, 105, 309, 34, 29, 0, 249, 402, 250, 314, 68, 108,
 165, 349, 36],
[259, 555, 372, 175, 338, 264, 232, 249, 0, 495, 352, 95, 189, 326,
 383, 202, 236],
[505, 289, 262, 476, 196, 360, 444, 402, 495, 0, 154, 578, 439, 336,
 240, 685, 390],
[353, 282, 110, 324, 61, 208, 292, 250, 352, 154, 0, 435, 287, 184,
 140, 542, 238],
[324, 638, 437, 240, 421, 329, 297, 314, 95, 578, 435, 0, 254, 391,
 448, 157, 301],
[70, 567, 191, 27, 346, 83, 47, 68, 189, 439, 287, 254, 0, 145, 202,
 289, 55],
[211, 466, 74, 182, 243, 105, 150, 108, 326, 336, 184, 391, 145, 0,
 57, 426, 96],
[268, 420, 53, 239, 199, 123, 207, 165, 383, 240, 140, 448, 202, 57,
 0, 483, 153],
[246, 745, 472, 237, 528, 364, 332, 349, 202, 685, 542, 157, 289,
 426, 483, 0, 336],
[121, 518, 142, 84, 297, 35, 29, 36, 236, 390, 238, 301, 55, 96, 153,
 336, 0]
```

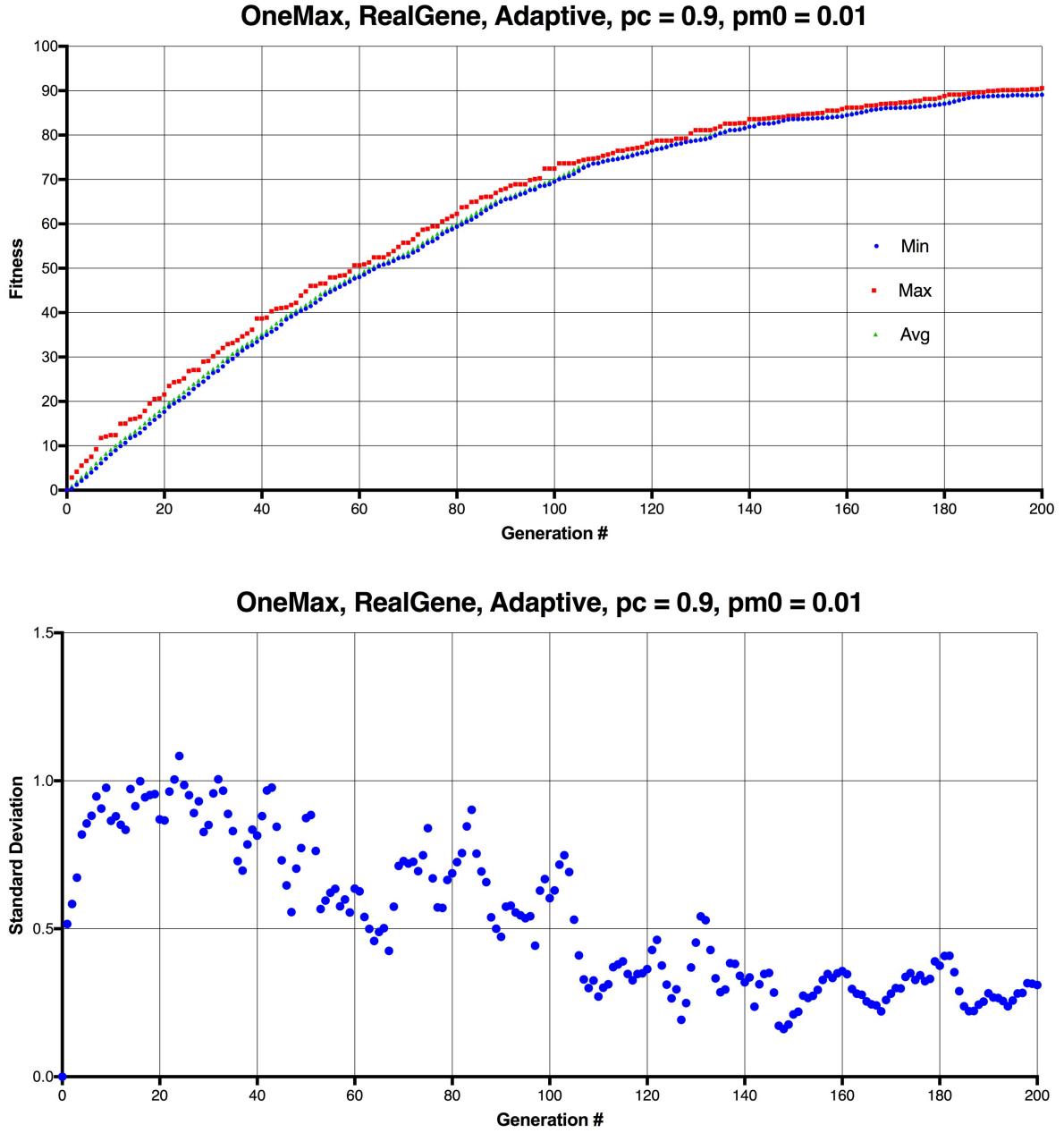


Figura 8: Evolução do fitness para o problema do OneMax Real Adaptativo mostrando mínimo, máximo e valor médio ($p_c = 0.9$, $p_{m0} = 0.01$).

5.4.1 Caso Estático ($p_m = 0.01$)

Para o caso estático com $p_m = 0.01$, mostrado na figura 10, vemos que este valor de p_m incentiva pouco o encontro de soluções melhores, demonstrado pela melhor solução (pontos azuis) ter mantido o mesmo valor por mais de 100 gerações. Em termos de desvio padrão, na figura 11, percebeu-se que $p_m = 0.01$ foi capaz apenas de igualar os valores de fitness de modo rápido (uma vez que há apenas 15 genes a serem recombinados).

Uma possível interpretação foi a de que o AG travou em um mínimo local, e um valor

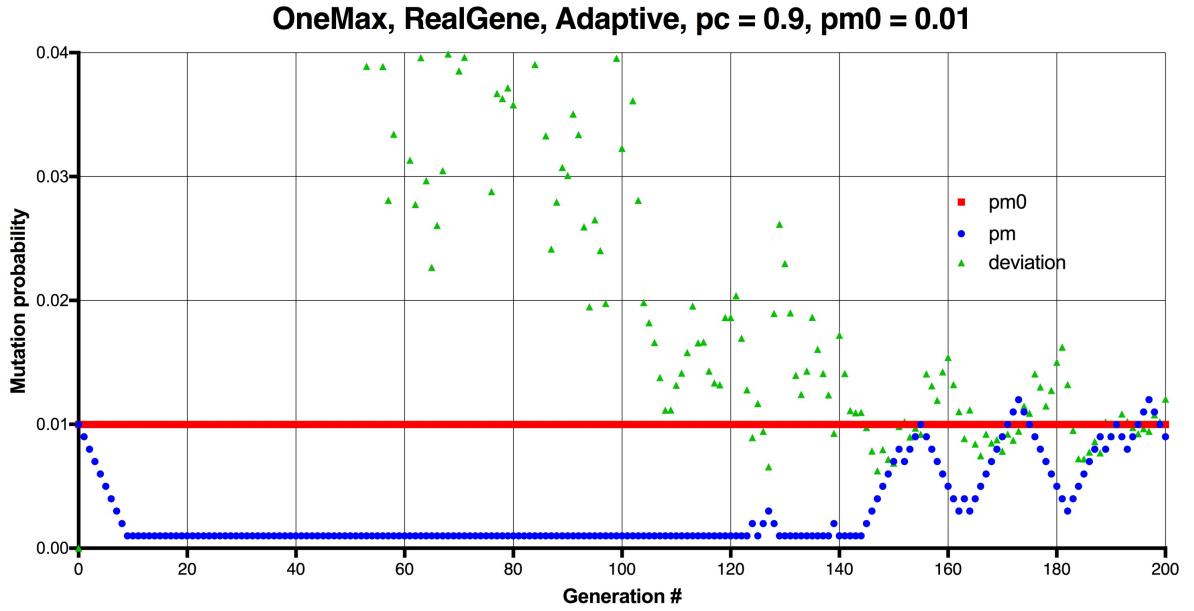


Figura 9: Probabilidade de mutação ao longo das gerações para o problema do OneMax Real Adaptativo ($p_c = 0.9$, $p_{m0} = 0.01$).

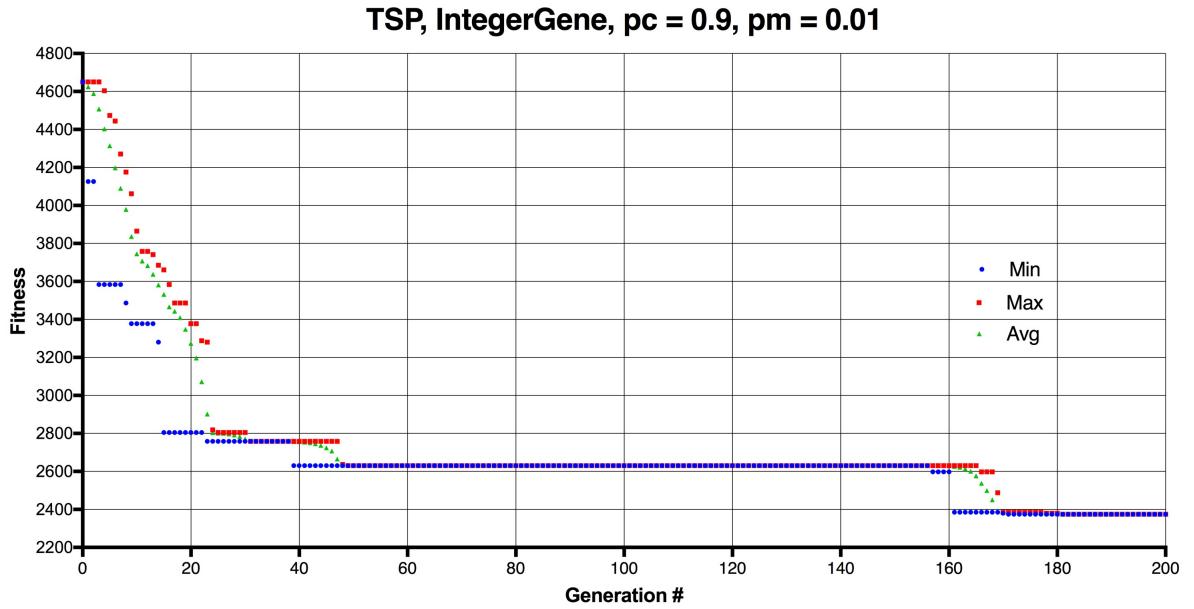


Figura 10: Evolução do fitness para o problema do Caixeiro Viajante Adaptado mostrando mínimo, máximo e valor médio ($p_c = 0.9$, $p_m = 0.01$). O menor caminho encontrado tem distância total de 2375.

maior de p_m poderia resolver este problema. Optou-se então por testar $p_m = 0.2$.

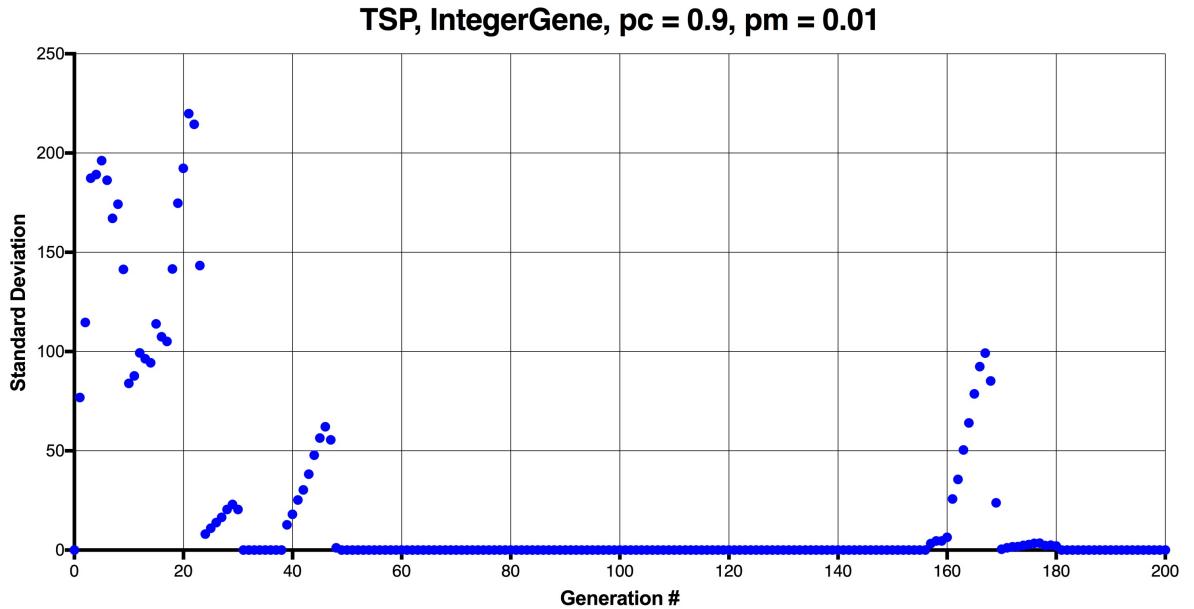


Figura 11: Desvio padrão ao longo das gerações para o problema do Caixeiro Viajante Adaptado ($p_c = 0.9$, $p_{m0} = 0.01$).

5.4.2 Caso Estático ($p_m = 0.2$)

Para o caso estático com $p_m = 0.2$, mostrado na figura 12, vemos que $p_m = 0.2$ conseguiu chegar a uma solução melhor (2300 contra 2375). No entanto, o comportamento do valor mínimo de fitness ainda foi muito próximo daquele visto para $p_m = 0.01$ (agora mantendo a melhor solução por mais de 120 gerações).

O desvio padrão, mostrado na figura 13, mostrou que uma mutação mais intensa incentiva o surgimento de muito mais soluções. No entanto, tais soluções são igualmente imprevisíveis aos olhos do problema, o que fez crer que aumentar p_m (para 20%, que já é absurdamente alto) não fez diferença para o caso estático.

5.4.3 Caso Adaptativo ($p_m = 0.01$)

O caso adaptativo para $p_m = 0.01$ mostrou uma evolução bem diferente do caso estático. Como visto na figura 14, os indivíduos também conseguiram igualar seus valores de fitness de modo relativamente rápido, mas a melhor solução pôde mudar bem mais rápido. A melhor solução ficou travada por não mais que 76 gerações (região azul entre as gerações 80 e 140), e mesmo assim, os demais indivíduos foram capazes de variar nesse intervalo, o que parece ter ajudado no encontro de soluções melhores. Tal comportamento das soluções pode ser observado nos desvios padrões, mostrados na figura 15.

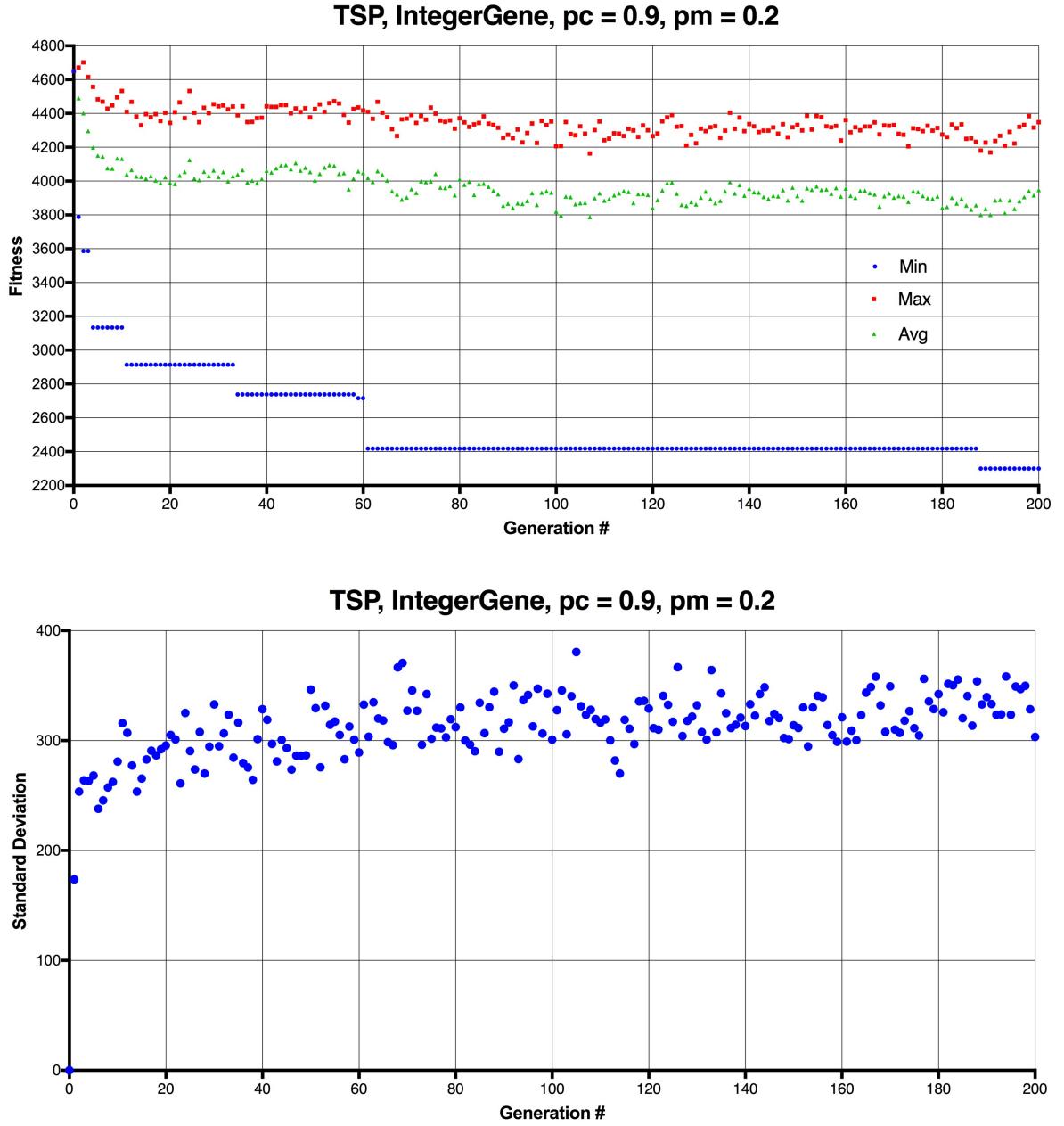


Figura 12: Evolução do fitness para o problema do Caixeiro Viajante Adaptado mostrando mínimo, máximo e valor médio ($p_c = 0.9$, $p_m = 0.2$). O menor caminho encontrado tem distância total de 2300.

O funcionamento do AGA ficou bem mais evidente na figura 16, com momentos de aumento e de redução de p_m alternados com maior frequência. Fora isso, p_{m0} convergiu para um valor bem menor no final da execução (0.024). No entanto, ao contrário dos problemas OneMax, p_m oscilou bastante também ao final das 200 gerações. Isso nos leva a crer que 0.01 talvez não seja o melhor valor para inicializar p_{m0} . Similar ao caso estático, simulou-se também $p_{m0} = 0.2$.

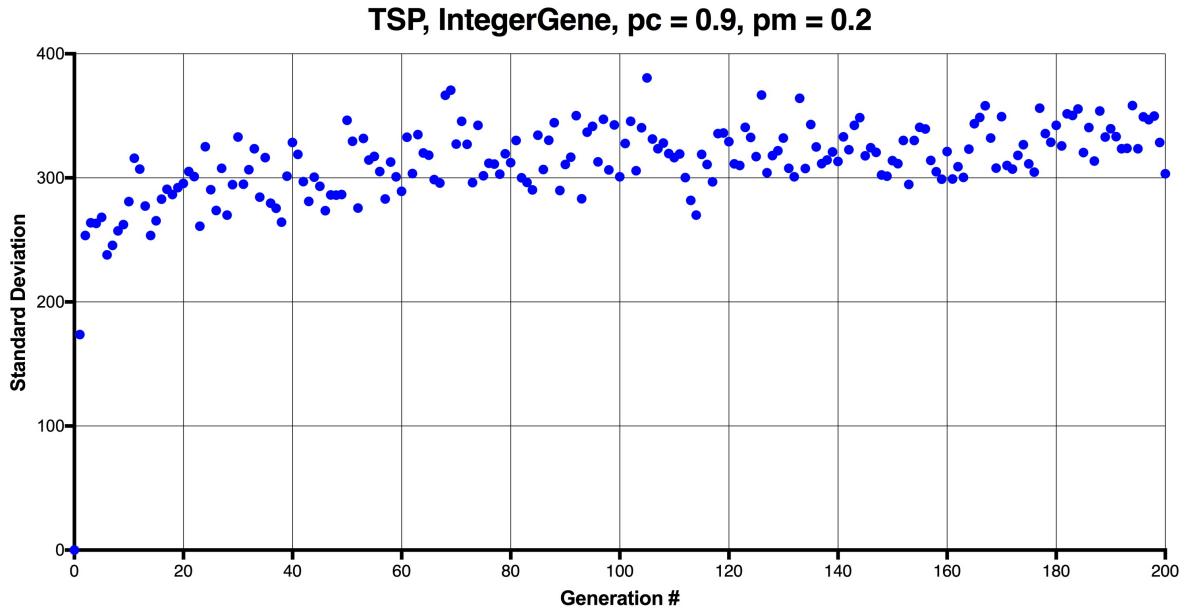


Figura 13: Desvio padrão ao longo das gerações para o problema do Caixeiro Viajante Adaptado ($p_c = 0.9$, $p_{m0} = 0.2$).

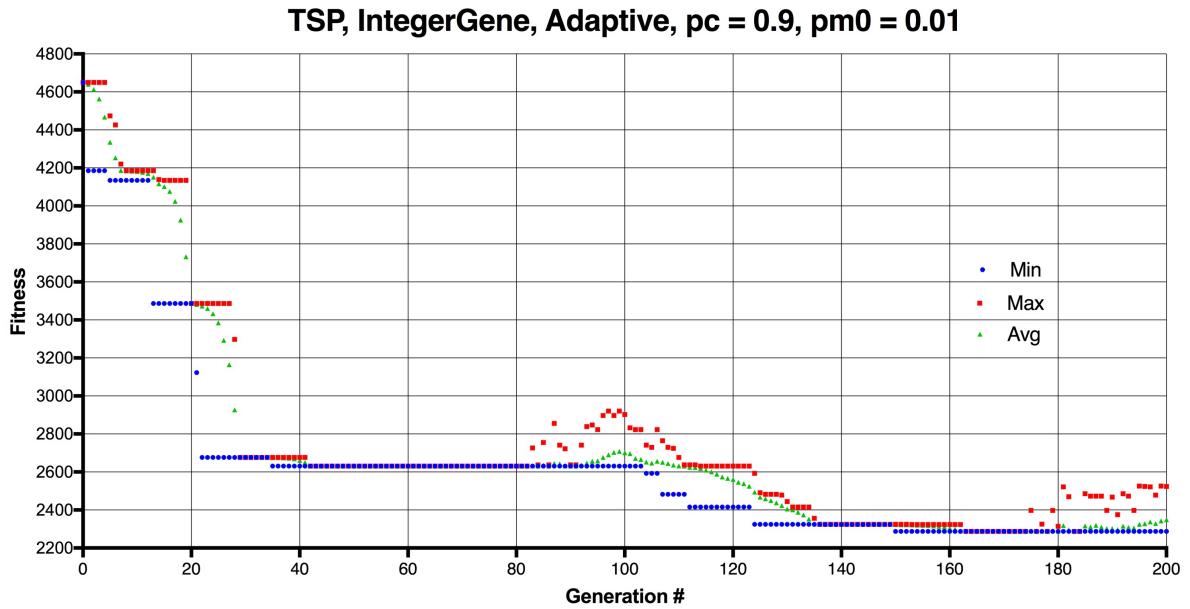


Figura 14: Evolução do fitness para o problema do Caixeiro Viajante Adaptado Adaptativo, mostrando mínimo, máximo e valor médio ($p_c = 0.9$, $p_{m0} = 0.01$). O menor caminho encontrado tem distância total de 2381.

5.4.4 Caso Adaptativo ($p_m = 0.2$)

O gráfico da figura 17 encontrou o menor percurso dentre as simulações feitas aqui (2160). Não só isso, as melhores soluções pareceram ficar travadas por bem menos tempo (o pior caso, próximo do final da simulação, durou 34 gerações). Um outro efeito interes-

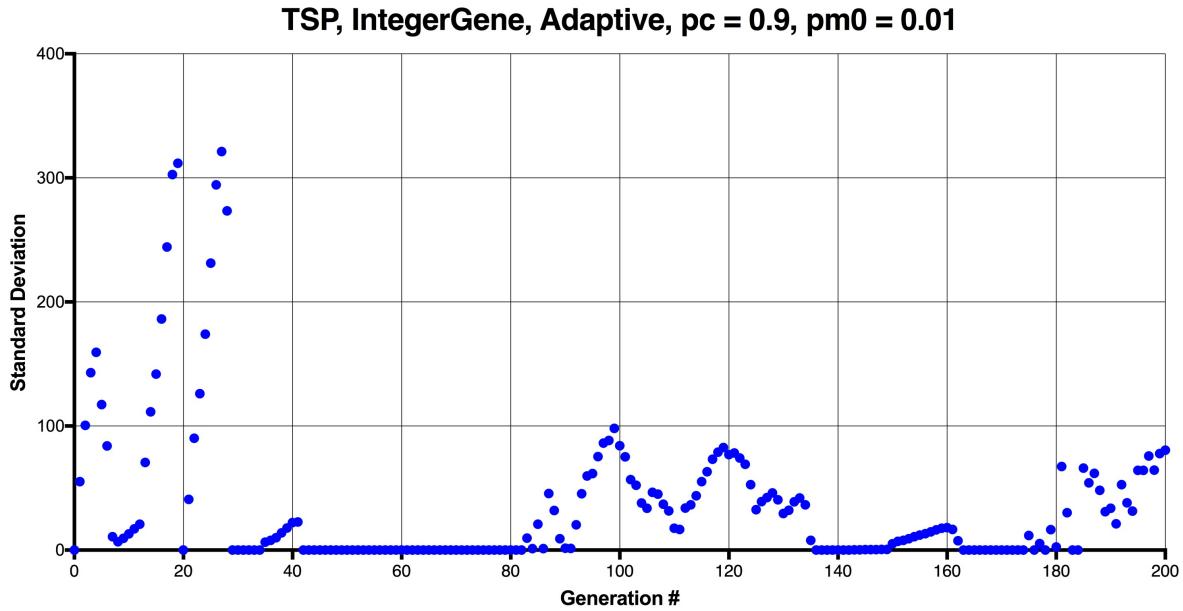


Figura 15: Desvio padrão ao longo das gerações para o problema do Caixeiro Viajante Adaptado Adaptativo ($p_c = 0.9$, $p_{m0} = 0.01$).

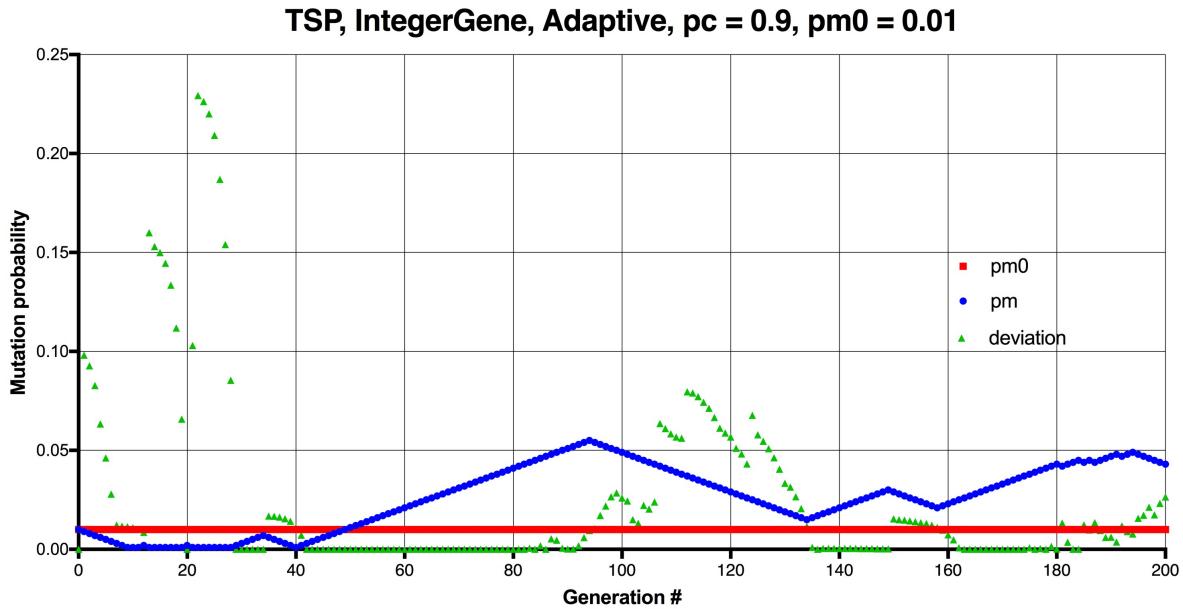


Figura 16: Probabilidade de mutação ao longo das gerações para o problema do Caixeiro Viajante Adaptado Adaptativo ($p_c = 0.9$, $p_{m0} = 0.01$).

sante também aconteceu: todas as curvas (mínimo, máximo e média) evoluíram de formas semelhantes, mesmo com uma mutação inicial intensa. Isso deu para observar também na evolução do desvio padrão, que conseguiu diminuir com o tempo e ficar mais ou menos estável ao longo das gerações.

A curva de interesse aqui foi certamente a da figura 18. O valor de p_{m0} não mudou

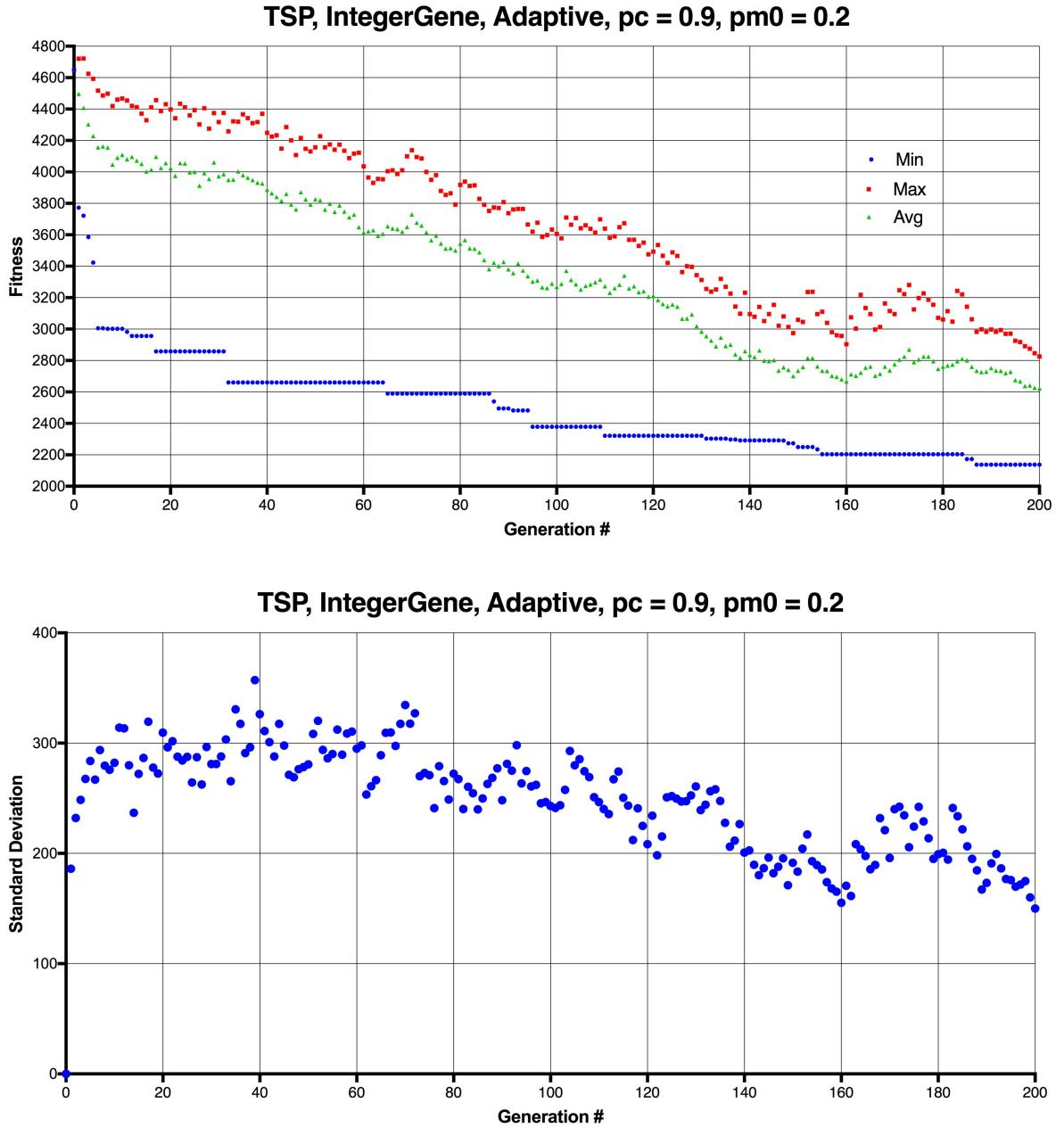


Figura 17: Evolução do fitness para o problema do Caixeiro Viajante Adaptado Adaptativo, mostrando mínimo, máximo, valor médio e desvio padrão ($p_c = 0.9$, $p_{m0} = 0.2$). O menor caminho encontrado tem distância total de 2160.

em momento algum, e p_m se estabilizou entorno de seu valor final (0.084). Esta é a curva que melhor demonstra o potencial do AGA desenvolvido aqui: mesmo começando num valor desfavorável para convergência, p_m mudou de patamar e foi estabilizado, trazendo a melhor resposta e a melhor evolução para este problema em 200 gerações.

Os dados de interesse destas quatro simulações estão presentes na tabela 3.

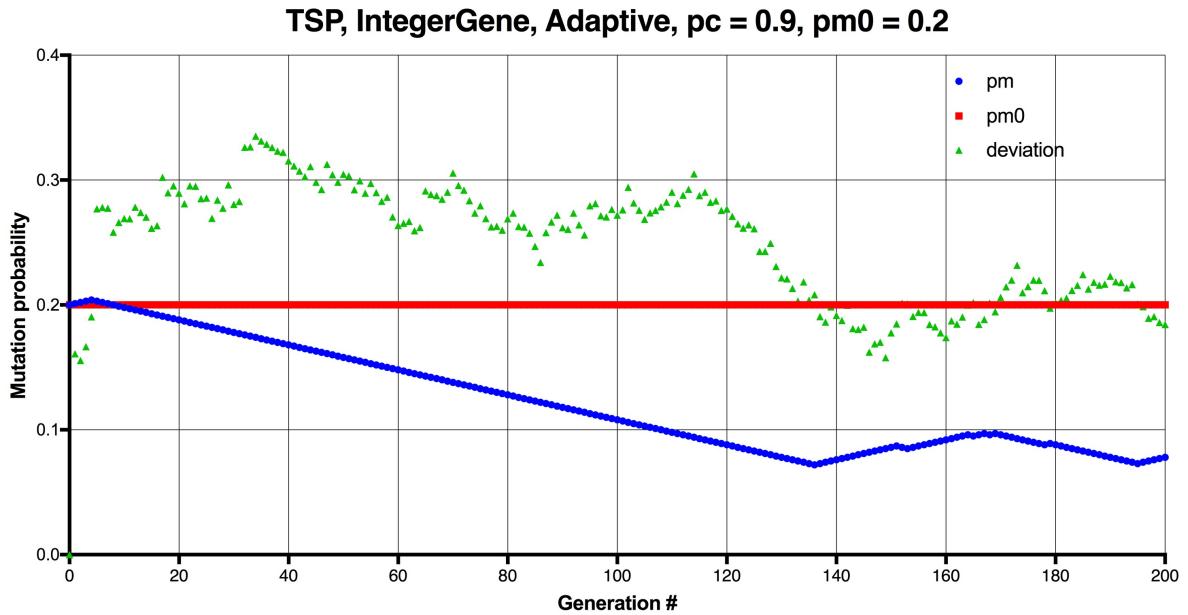


Figura 18: Probabilidade de mutação ao longo das gerações para o problema do Caixeiro Viajante Adaptado Adaptativo ($p_c = 0.9$, $p_{m0} = 0.2$).

5.5 Discussões

Os três problemas foram simulados tanto pelo AG estático quanto pelo AGA. Para os problemas OneMax, foi possível observar uma evolução muito melhor para o caso estático do que para o caso adaptativo. Para o problema do Caixeiro Viajante Adaptado, no entanto, observou-se o oposto. Seria possível validar o AGA a partir destes experimentos?

Pensemos nas operações de variação de um AG. Se associarmos probabilidade de crossover p_c (alta) ao fator de homogeneização do fitness da população, então a probabilidade de mutação p_m , combinada com elitismo, pode atuar de duas formas:

Tabela 3: Dados coletados do problema do Caixeiro Viajante Adaptado.

Algoritmo analisado (AG = caso estático)	0.01		0.2	
	AG	AGA	AG	AGA
Fitness mínimo após 100 gerações	2631	2631	2418	2379
Fitness médio após 100 gerações	2631	2701.2	3819.5	3267.64
Fitness mínimo após 200 gerações	2375	2287	2300	2138
Fitness médio após 200 gerações	2375	2349.11	3948.11	2621.33
# máximo de gerações "travado"(mínimo)	118(2631)	69(2631)	127(2418)	33(2661)
Valor final de p_m	0.01	0.043	0.2	0.078
Valor mínimo de p_m	0.01	0.001	0.2	0.072
Valor máximo de p_m	0.01	0.055	0.2	0.204
Valor médio de p_m	0.01	0.0261	0.2	0.122
Valor médio de p_m (últimas 100 gerações)	0.01	0.0327	0.2	0.0866

- Tornando a população heterogênea;
- **Melhorando o fitness do melhor indivíduo.**

O efeito que será mais predominante dependerá do valor de p_m e do problema. Como visto no OneMax, um valor de 0.01 ou de 0.001 foi suficiente para priorizar o segundo efeito sobre o primeiro, evoluindo gradualmente o melhor indivíduo.

No entanto, o problema do Caixeiro Viajante Adaptado não conseguiu se beneficiar destes efeitos nem para $p_m = 0.01$, nem para $p_m = 0.2$. Não que o AG estático não possua um valor ideal para convergência deste problema em específico - isto apenas atesta que o parâmetro ideal para um problema pode não ser ideal para outro problema.

Se tentássemos estressar o problema do Caixeiro, poderíamos eventualmente encontrar um valor ideal para p_{m_0} . No entanto, tal problema (mesmo adaptado) ainda é NP-Hard, e um valor estático de p_m pode não ser suficiente para encontrar o mínimo global. O que resolveria este problema, quando um valor muito baixo e um valor muito alto de p_{m_0} não são suficientes?

Tal questionamento trouxe a este trabalho a ideia de se usar um algoritmo adaptativo. Algo a se acrescentar ao AG que permitisse adaptar o valor de p_m ao longo das gerações. Mesmo que o uso do AGA não trouxesse melhores soluções mais rapidamente, ele se mostrou melhor para o encontro de percursos cada vez menores. No entanto, o uso do AGA não trouxe o mesmo benefício para os problemas OneMax. Repensar o AGA é uma ideia válida, mas usar o AGA será sempre uma ideia melhor?

Para isso, é necessário considerar os teoremas "No Free Lunch" (NFL - em português, teriam a mesma origem da expressão "Não existe almoço grátis") [17]. De maneira simples, estes teoremas dizem que, para um algoritmo de busca e otimização (como é o caso do AG e do AGA), uma performance elevada para um grupo de problemas (como os problemas OneMax) tem como preço uma queda de performance para todos os outros grupos de problemas (como o problema do Caixeiro Viajante). O AGA tentaria ser a adaptação do AG para se adequar a outros problemas, e mesmo assim, ele veio com uma perda de performance para os problemas OneMax.

Não é possível utilizar o mesmo algoritmo para todos os problemas. Mesmo que não utilizássemos um AGA, ainda haveria formas melhores de se otimizar o AG para um grupo de problemas, incluindo o Caixeiro Viajante. Isso, no entanto, não tira o mérito do AGA.

Pelos experimentos feitos neste trabalho, ele foi capaz de encontrar soluções melhores

para um problema mais complexo que os OneMax. Melhor performance é sempre interessante, mas ser capaz de encontrar soluções cada vez melhores sem ficar travado em extremos locais é, ao olhos deste trabalho, um benefício muito melhor. Restaria então verificar se outros problemas conseguiram se beneficiar do AGA da mesma forma, e se o algoritmo do AGA poderia ser melhorado também.

6 Conclusões e Trabalhos Futuros

6.1 Conclusões

Este trabalho teve como objetivo o desenvolvimento de um Algoritmo Genético (AG) para a resolução de diferentes problemas de modo otimizado. Foram escolhidos três problemas: OneMax Booleano (variável booleana), OneMax Real (variável real) e Caixeiro Viajante Adaptado (permitindo atalhos). De modo a melhorar a performance do AG, foram implementadas duas otimizações: o elitismo do melhor indivíduo, e o uso de um Algoritmo Genético Adaptativo (AGA) com implementação própria, focada na adaptação do parâmetro de mutação.

Chegou-se à conclusão que utilizar um mesmo algoritmo para resolver um grupo diferente de problemas sem perda de performance é virtualmente impossível. No entanto, o uso do AGA implementado se mostrou mais eficiente no encontro de melhores soluções para o problema do Caixeiro Viajante, um problema NP-Hard. Por conta disso, o uso deste AGA é fortemente incentivado em outros problemas, buscando sempre validações e melhorias. Se isso não for possível, o conceito por trás do desenvolvimento de um algoritmo adaptativo é muito forte e próximo do conceito básico do AG, e seu uso deve ser considerado em outros algoritmos evolutivos.

6.2 Trabalhos Futuros

O desenvolvimento feito neste trabalho resultaram na criação de um algoritmo adaptativo e de implementações próprias tanto do algoritmo genético quanto dos problemas escolhidos. A proposta deste trabalho foi audaz, e muitos conceitos foram desenvolvidos ao mesmo tempo, conceitos que precisam ser quebrados e maturados ainda mais. Sugere-se então as seguintes frentes de trabalho futuro:

- Análise mais cuidadosa do problema do OneMax Real, considerando modelagens

matemáticas e sua proximidade com o OneMax Booleano;

- Análise do problema do Caixeiro Viajante com a distribuição genética deste trabalho, comparada com implementações mais tradicionais (este trabalho distribuiu os genes deste problema de tal forma que o crossover de dois pontos ainda fosse possível);
- Evolução dos conceitos por trás do AGA desenvolvido aqui, buscando formalizar os conceitos por trás de seu desenvolvimento e, onde for possível, melhorá-lo (tanto para performance quanto para busca de soluções);
- Testar o AGA junto a outros problemas, para verificar prós, contras e limitações;
- O algoritmo em si foi pouco trabalhado (mesmo tendo muita coisa sendo discutida aqui), dado o tempo utilizado para este trabalho. Recomenda-se uma melhoria drástica deste algoritmo de modo a comportar outras implementações dos problemas, dos operadores de evolução e de outros algoritmos adaptativos.

Referências

- 1 D. L. Applegate, R. E. Bixby, V. Chvatal, and W. J. Cook. *The Traveling Salesman Problem: a computational study*. Princeton university press, 2011.
- 2 T. H. Cormen, C. E. Leiserson, R. L. Rivest, et al. *Section 24.3: Dijkstra's algorithm*. MIT Press and McGraw-Hill, 2001.
- 3 K. DeJong. An analysis of the behavior of a class of genetic adaptive systems. *Ph. D. Thesis, University of Michigan*, 1975.
- 4 E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.
- 5 A. E. Eiben and S. K. Smit. Parameter tuning for configuring and analyzing evolutionary algorithms. *Swarm and Evolutionary Computation*, 1(1):19–31, 2011.
- 6 A. E. Eiben and J. E. Smith. *Introduction To Evolutionary Computing*, volume 53. Springer-Verlag, 2003.
- 7 Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. Deap examples - tsp. <https://github.com/DEAP/deap/blob/master/examples/ga/tsp/gr17.json>. [Online; acessado em 19 de novembro de 2016].
- 8 Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13:2171–2175, jul 2012.
- 9 P. Giguere and D. E. Goldberg. Population sizing for optimum sampling with genetic algorithms: A case study of the onemax problem. *Genetic Programming*, 98:496–503, 1998.
- 10 ICMC-USP. Algoritmos genéticos. <http://conteudo.icmc.usp.br/pessoas/andre/research/genetic/>. [Online; acessado em 17 de novembro de 2016].
- 11 D. Jakobović and M. Golub. Adaptive genetic algorithm. *CIT. Journal of computing and information technology*, 7(3):229–235, 1999.
- 12 K. Matthias, T. Severin, and H. Salzwedel. Variable mutation rate at genetic algorithms: introduction of chromosome fitness in connection with multi-chromosome representation. *International Journal of Computer Applications*, 72(17), 2013.
- 13 M. Mitchell. *An introduction to genetic algorithms*. MIT press, 1998.

- 14 M. Obitko. Introduction to genetic algorithms - xiii. recommendations.
<http://www.obitko.com/tutorials/genetic-algorithms/recommendations.php/>. [Online; acessado em 17 de novembro de 2016].
- 15 M. Srinivas and L. M. Patnaik. Adaptive probabilities of crossover and mutation in genetic algorithms. *IEEE Transactions on Systems, Man, and Cybernetics*, 24(4):656–667, 1994.
- 16 L. Wang and T. Shen. Improved adaptive genetic algorithm and its application to image segmentation. In *Multispectral Image Processing and Pattern Recognition*, pages 115–120. International Society for Optics and Photonics, 2001.
- 17 D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82, 1997.

APÊNDICE A - Genes Utilizados

Listagem A.1: Genes Utilizados (genes.py).

```

import numpy.random as random
from utils import coin_toss

def new_gene(gene_name, args=None):
    constructor = globals()[gene_name]
    return constructor() if args is None else constructor(args)

class Gene:
    def __init__(self):
        self.val = None

    def __repr__():
        return 'Gene {value: %s}' % (self.val)

    def value(self):
        return self.val

    def mutate(self):
        pass

class BooleanGene(Gene):
    def __init__(self, val=False):
        Gene.__init__(self)
        self.val = val

```

```

def __repr__(self):
    return 'BooleanGene {value: %s}' % (self.value())

def value(self):
    return 1 if self.val else 0

def mutate(self):
    self.val = coin_toss()

class RealGene(Gene):
    def __init__(self, val=0.0):
        Gene.__init__(self)
        self.val = val

    def __repr__(self):
        return 'RealGene {value: %s}' % (self.val)

    def mutate(self):
        self.val = random.random()

class IntegerGene(Gene):
    def __init__(self, k=2):
        if k <= 1:
            raise ValueError('IntegerGene should have a range greater
                           than 1.')
        Gene.__init__(self)
        self.val = 0
        self.k = k

    def __repr__(self):
        return 'IntegerGene {value: %s, k: %s}' % (self.val, self.k)

    def mutate(self):
        current_val = self.val
        self.val = random.randint(self.k)

```

```
while self.val == current_val:  
    self.val = random.randint(self.k)
```

APÊNDICE B – Indivíduos dos problemas

Listagem B.1: Indivíduos dos problemas (individuals.py).

```

from genes import new_gene
from utils import arrow_list_str
from dijkstra import tsp_dist, tsp_path, tsp_full_path,
    dijkstra_gsize
import numpy.random as random

def new_individual(ind_name, gene_type=None, gsize=None):
    constructor = globals()[ind_name]
    if gene_type is None:
        return constructor()
    elif gsize is None:
        return constructor(gene_type)
    else:
        return constructor(gene_type, gsize)

def from_genes(ind_name, genes, gene_type=None):
    ind = new_individual(ind_name, gene_type)
    ind.genes = genes
    return ind

class Individual:
    def __init__(self, gene_type, gsize):
        self.gene_type = gene_type
        self.genes = [new_gene(gene_type) for x in range(
            gsize)]

```

```

    self.fitness = 0.0

def __len__(self):
    return len(self.genes)

def __repr__(self):
    return 'Individual {Gene: %s, Count: %d}' % (self.
                                                gene_type, len(self))

class OneMaxIndividual(Individual):
    def __init__(self, gene_type='BooleanGene', gsize=100):
        Individual.__init__(self, gene_type, gsize)
        self.gene_type = gene_type
        self.fitness = 0.0

    def __repr__(self):
        return 'OneMaxIndividual {Gene: %s, Count: %d}' % (
            self.gene_type, len(self))

    def __str__(self):
        return str([x.value() for x in self.genes])

class TSPIndividual(Individual):
    def __init__(self, gene_type='IntegerGene', gsize=
                dijkstra_gsize):
        Individual.__init__(self, gene_type, gsize)
        self.gene_type = gene_type
        self.genes = [new_gene(gene_type, csize) for csize in
                     reversed(range(2, gsize))]
        self.fitness = 0.0

    def __repr__(self):
        return 'TSPIndividual {Gene: %s, Count: %d}' % (self.
                                                       gene_type, len(self))

    def __str__(self):

```

```
sequence = [x.value() for x in self.genes]
string = '\nDistance : ' + str(tsp_dist(
    sequence))
string += '\nCities visited : ' + arrow_list_str(
    tsp_path(sequence))
string += '\nFull path : ' + arrow_list_str(
    tsp_full_path(sequence))
return string
```

APÊNDICE C – Algoritmo Genético

Listagem C.1: Algoritmo Genético (geneflow.py).

```

import numpy.random as random
import fitness
import copy
import os
from utils import *
from individuals import new_individual, from_genes
from individuals import Individual

class GeneFlow:
    def __init__(self, ind_type, gene_type, ffit=None, pc=0.9, pm
                 =0.01, mu=100, ngen=200,
                 print_stats=True, maximum=True, elitism=True,
                 adaptive=False):
        self.fitness = ffit
        self.population = [new_individual(ind_type, gene_type) for x
                           in range(mu)]
        self.pm = pm
        self.pm0 = pm
        self.pc = pc
        self.mu = mu
        self.ngen = ngen
        self.print_stats = print_stats
        self.maximum = maximum
        self.elitism = elitism
        self.adaptive = adaptive
        self.start_writing(ind_type, gene_type)

```

```

def start_writing(self, ind_type, gene_type):
    file_name = '../out/' + ind_type + '_' + gene_type + ('' if
        not self.adaptive else '_adaptive')
    new_path = os.path.relpath(file_name + '.csv', os.path.
        dirname(__file__))
    self.file = open(new_path, 'w')
    self.file.write('Generation,Min,Max,Avg,Std\n')

    if self.adaptive:
        new_path = os.path.relpath(file_name + '_pm.csv', os.path.
            dirname(__file__))
        self.adapt_file = open(new_path, 'w')
        self.adapt_file.write('Generation,pm,pm0,deviation,
            best_fitness\n')

def calculate_all_fitness(self):
    self.parent_fitness()
    self.offspring_fitness()

def parent_fitness(self):
    for individual in self.population:
        individual.fitness = self.fitness(individual)

def offspring_fitness(self):
    for individual in self.offspring:
        individual.fitness = self.fitness(individual)

def min_fitness(self):
    return min([x.fitness for x in self.population])

def max_fitness(self):
    return max([x.fitness for x in self.population])

def best_fitness(self):
    return self.max_fitness() if self.maximum else self.
        min_fitness()

```

```

def avg_fitness(self):
    return mean([x.fitness for x in self.population])

def std_fitness(self):
    return std([x.fitness for x in self.population])

def stats(self):
    if self.print_stats is False:
        return

    print( 'Min      : %.6f' % self.min_fitness())
    print( 'Max      : %.6f' % self.max_fitness())
    print( 'Average: %.6f' % self.avg_fitness())
    print( 'Std      : %.6f' % self.std_fitness())
    print( 'Best individual is: ' + str(self.population[0]))

def generate(self):
    if self.print_stats:
        print( 'Generation 0:')

        self.parent_fitness()
        self.stats()
        self.write_to_file(0)

    for i in range(self.ngen):
        if self.print_stats:
            print( '\nGeneration %s:' % (i+1))
            self.update()
            self.stats()
            self.write_to_file(i+1)

def update(self):
    self.selection()
    self.crossover()
    self.mutation()
    self.survival()
    pass

```

```

# The offspring is selected based on the best fitness values
def selection(self):
    self.population.sort(key=lambda ind:ind.fitness, reverse=self.
        .maximum)
    self.offspring = []

# Offspring is paired and crossed over with probability pc – two
parents generate two children
def crossover(self):
    for parent1, parent2 in zip(self.population[::2], self.
        population[1::2]):
        if random.random() < self.pc:
            child1, child2 = self.cross(parent1, parent2)
            self.offspring.append(child1)
            self.offspring.append(child2)

# Two-point crossover – two children mix their genes, based in a
two-point section switch of their genes
def cross(self, ind1, ind2):
    length = len(ind1)
    ind_type = ind1.__class__.__name__
    gene_type = ind1.gene_type

    genes1 = copy.deepcopy(ind1.genes)
    genes2 = copy.deepcopy(ind2.genes)

# Two distinct points are chosen
    rand1 = random.randint(length + 1)
    rand2 = random.randint(length + 1)

while (rand1 == rand2) or (rand1 == 0 and rand2 == length) or
    (rand2 == 0 and rand1 == length):
    rand2 = random.randint(length + 1)

    rand1, rand2 = (rand1, rand2) if rand1 < rand2 else (rand2,
        rand1)
    genes1[rand1:rand2], genes2[rand1:rand2] = genes2[rand1:rand2]
        , genes1[rand1:rand2]

```

```

return from_genes(ind_type, genes1, gene_type), from_genes(
    ind_type, genes2, gene_type)

# Mutation acts over all genes (except the elite), with
probability pm

def mutation(self):
    if self.elitism:
        self.population.sort(key=lambda ind:ind.fitness, reverse=
            self.maximum)
        self.elite = copy.deepcopy(self.population[0])
        self.population = self.population[1:]

    for ind in self.population:
        for gene in ind.genes:
            if random.random() < self.pm:
                gene.mutate()

    for ind in self.offspring:
        for gene in ind.genes:
            if random.random() < self.pm:
                gene.mutate()

# Only the best mu individuals survive

def survival(self):
    if self.elitism:
        self.population.append(self.elite)

        self.calculate_all_fitness()
        self.population = self.population + self.offspring
        self.population.sort(key=lambda ind:ind.fitness, reverse=self
            .maximum)
        self.population = self.population[:self.mu]

    if self.adaptive:
        self.adapt()

# Deviation of the best fitness to the average

```

```

def deviation(self):
    if abs(self.avg_fitness()) < 0.0000001:
        return 0.0

    return abs((self.best_fitness() - self.avg_fitness()) / (self.avg_fitness()))

# Adaptive Genetic Algorithm (AGA) module
def adapt(self):

    # In order to avoid division by zero
    if abs(self.avg_fitness()) < 0.0000001:
        return

    # Increment pm if the deviation is lower than pm0
    if self.deviation() <= self.pm0:
        self.pm = min(0.5, self.pm + 0.001)

    # Decrement pm, otherwise
    else:
        self.pm = max(0.001, self.pm - 0.001)

    if self.print_stats:
        print self.deviation(), self.pm, self.pm0, self.best_fitness()

def write_to_file(self, n):
    self.file.write(str(n))
    self.file.write(',')
    self.file.write(str(self.min_fitness()))
    self.file.write(',')
    self.file.write(str(self.max_fitness()))
    self.file.write(',')
    self.file.write(str(self.avg_fitness()))
    self.file.write(',')
    self.file.write(str(self.std_fitness()))
    self.file.write('\n')

```

```

if self.adaptive:
    self.adapt_file.write(str(n))
    self.adapt_file.write(',')
    self.adapt_file.write(str(self.pm))
    self.adapt_file.write(',')
    self.adapt_file.write(str(self.pm0))
    self.adapt_file.write(',')
    self.adapt_file.write(str(self.deviation()))
    self.adapt_file.write(',')
    self.adapt_file.write(str(self.best_fitness()))
    self.adapt_file.write('\n')

# Uncomment one of the next three lines to simulate the algorithm

#GeneFlow('OneMaxIndividual', 'BooleanGene', fitness.onemax, adaptive=True, print_stats=True, pm=0.01).generate()
#GeneFlow('OneMaxIndividual', 'RealGene', fitness.onemax, adaptive=True, print_stats=True, pm=0.01).generate()
GeneFlow('TSPIndividual', 'IntegerGene', fitness.tsp, maximum=False, adaptive=True, print_stats=True, pm=0.2).generate()

```

APÊNDICE D – Funções de fitness

Listagem D.1: Funções de fitness (fitness.py).

```
from dijkstra import tsp_dist, all_cities
import copy

# Fitness function for OneMax
def onemax(individual):
    return sum(gene.value() for gene in individual.genes)

# Fitness function for TSP
def tsp(individual):
    return tsp_dist([gene.value() for gene in individual.genes])
```

APÊNDICE E - Mapa das cidades - Caixeiro Viajante

Listagem E.1: Arquivo com mapa das cidades - Caixeiro Viajante (dijkstra17.py).

```
# Standard Dijkstra with 17 cities

dijkstra17 = [
    [0, 633, 257, 91, 412, 150, 80, 134, 259, 505, 353, 324, 70, 211,
     268, 246, 121],
    [633, 0, 390, 661, 227, 488, 572, 530, 555, 289, 282, 638, 567,
     466, 420, 745, 518],
    [257, 390, 0, 228, 169, 112, 196, 154, 372, 262, 110, 437, 191,
     74, 53, 472, 142],
    [91, 661, 228, 0, 383, 120, 77, 105, 175, 476, 324, 240, 27, 182,
     239, 237, 84],
    [412, 227, 169, 383, 0, 267, 351, 309, 338, 196, 61, 421, 346,
     243, 199, 528, 297],
    [150, 488, 112, 120, 267, 0, 63, 34, 264, 360, 208, 329, 83, 105,
     123, 364, 35],
    [80, 572, 196, 77, 351, 63, 0, 29, 232, 444, 292, 297, 47, 150,
     207, 332, 29],
    [134, 530, 154, 105, 309, 34, 29, 0, 249, 402, 250, 314, 68, 108,
     165, 349, 36],
    [259, 555, 372, 175, 338, 264, 232, 249, 0, 495, 352, 95, 189,
     326, 383, 202, 236],
    [505, 289, 262, 476, 196, 360, 444, 402, 495, 0, 154, 578, 439,
     336, 240, 685, 390],
    [353, 282, 110, 324, 61, 208, 292, 250, 352, 154, 0, 435, 287,
     184, 140, 542, 238],
```

```

[324, 638, 437, 240, 421, 329, 297, 314, 95, 578, 435, 0, 254,
 391, 448, 157, 301],
[70, 567, 191, 27, 346, 83, 47, 68, 189, 439, 287, 254, 0, 145,
 202, 289, 55],
[211, 466, 74, 182, 243, 105, 150, 108, 326, 336, 184, 391, 145,
 0, 57, 426, 96],
[268, 420, 53, 239, 199, 123, 207, 165, 383, 240, 140, 448, 202,
 57, 0, 483, 153],
[246, 745, 472, 237, 528, 364, 332, 349, 202, 685, 542, 157, 289,
 426, 483, 0, 336],
[121, 518, 142, 84, 297, 35, 29, 36, 236, 390, 238, 301, 55, 96,
 153, 336, 0]
]

def create_graph(dijk_map):
    if dijk_map is None or len(dijk_map) is 0:
        return None

    graph = {}
    length = len(dijk_map)

    for i in range(length):
        first_city = chr(i + ord('A'))
        for j in range(length):
            second_city = chr(j + ord('A'))
            distance = dijk_map[i][j]

            if graph.get(first_city) is None:
                graph[first_city] = {}

            if graph.get(second_city) is None:
                graph[second_city] = {}

            graph[first_city][second_city] = distance

    return graph

dijkstra17_graph = create_graph(dijkstra17)

```

APÊNDICE F – Implementação do Algoritmo de Dijkstra

Listagem F.1: Arquivo com algoritmo de Dijkstra (dijkstra.py).

```

import copy
from dijkstra17 import dijkstra17_graph

# Functions that apply Dijkstra's algorithm

def dijkstra_all(graph):
    full_map = {}
    for source in graph:
        dist, prev = dijkstra(graph, source)
        full_map[source] = {}
        full_map[source]['dist'] = dist
        full_map[source]['prev'] = prev
    return full_map

def dijkstra(graph, source):
    Q = set()
    dist = {}
    prev = {}

    for vertex in graph:
        dist[vertex] = float("inf")
        prev[vertex] = None
        Q.add(vertex)

    dist[source] = 0

```

```

while len(Q) is not 0:
    u = min_value(Q, dist)
    Q.remove(u)

    for v in graph[u]:
        alt = dist[u] + graph[u][v]
        if alt < dist[v]:
            dist[v] = alt
            prev[v] = u

return dist, prev

def min_value(Q, dist):
    u = next(iter(Q))
    for v in Q:
        if dist[v] < dist[u]:
            u = v
    return u

# Global variables

complete_graph = dijkstra_all(dijkstra17_graph)
all_cities = [node for node in complete_graph]
all_cities.sort()

# Functions to calculate distance

def dist_between(a, b, dijkstra_graph=complete_graph):
    return dijkstra_graph[a]['dist'][b]

def tsp_dist(sequence):
    copy_cities = copy.copy(all_cities)
    init = copy_cities.pop(0)
    prev = init
    next = None

```

```

tsp_sum = 0
for i in sequence:
    prev = prev if next is None else next
    next = copy_cities.pop(i)
    tsp_sum += dist_between(prev, next)
tsp_sum += dist_between(next, copy_cities[0])
tsp_sum += dist_between(copy_cities[0], init)
return tsp_sum

# Functions to find path

def path_between(a, b, dijkstra_graph=complete_graph):
    path = [b]
    prev = dijkstra_graph[a]['prev'][b]
    while prev is not None:
        path.insert(0, prev)
        prev = dijkstra_graph[a]['prev'][prev]
    return path

def tsp_path(sequence):
    copy_sequence = copy.copy(sequence)
    copy_sequence.append(0)

    copy_cities = copy.copy(all_cities)
    init = copy_cities.pop(0)
    tsp_path = [init]

    for i in copy_sequence:
        tsp_path.append(copy_cities.pop(i))

    tsp_path.append(init)
    return tsp_path

def tsp_full_path(sequence):
    path = tsp_path(sequence)
    full_path = []

```

```
for first, second in zip(path, path[1:]):
    full_path = full_path[:-1] + path_between(first,
                                              second)
return full_path

dijkstra_gszie = len(all_cities)
```

APÊNDICE G – Funções de utilidade

Listagem G.1: Funções de utilidade (utils.py)

```

import numpy.random as random

# Boolean variable on 50% chance
def coin_toss():
    return random.random() >= 0.5

# Mean value between numbers
def mean(numbers):
    return float(sum(numbers)) / max(len(numbers), 1)

# Sum of squares
def sum2(numbers):
    return sum(x*x for x in numbers)

# Standard deviation
def std(numbers):
    n = len(numbers)
    sum_sum = sum2(numbers)
    avg = mean(numbers)
    return abs((sum_sum - n*avg*avg) / (n - 1.0)) ** 0.5

# Adds arrows between values of a list
def arrow_list_str(some_list):
    return str(some_list).replace(' , ', ' → ').replace(' [ ', ' ')
        .replace(' ] ', ' ').replace('"', ' ')

```

FOLHA DE REGISTRO DO DOCUMENTO				
1. CLASSIFICAÇÃO/TIPO TC	2. DATA 22 de novembro de 2016	3. REGISTRO N° DCTA/ITA/TC-060/2016	4. N° DE PÁGINAS 80	
5. TÍTULO E SUBTÍTULO: Resolução otimizada de problemas com uso de algoritmos evolutivos.				
6. AUTOR(ES): Cássio dos Santos Sousa				
7. INSTITUIÇÃO(ÕES)/ÓRGÃO(S) INTERNO(S)/DIVISÃO(ÕES): Instituto Tecnológico de Aeronáutica – ITA				
8. PALAVRAS-CHAVE SUGERIDAS PELO AUTOR: <u>Algoritmo, Evolutivo, Genético, Adaptativo, Otimização.</u>				
9. PALAVRAS-CHAVE RESULTANTES DE INDEXAÇÃO: Algoritmos genéticos; Otimização; Identificação de parâmetros; Controle adaptativo; Problema do caixeiro viajante; Computação.				
10. APRESENTAÇÃO: X Nacional Internacional ITA, São José dos Campos. Curso de Graduação em Engenharia de Computação. Orientador: Carlos Henrique Quartucci Forster. Publicado em 2016.				
11. RESUMO: Algoritmos Evolutivos (AE) são de grande interesse na resolução de problemas complexos. Baseados nos conceitos biológicos de Evolução e na resolução por tentativa-e-erro, são capazes de obter boas respostas com performance superior a muitos algoritmos. Este trabalho promoveu a implementação e a análise de um Algoritmo Genético (AG), subgrupo dos AEs cuja informação é contida em genes, os quais compõem indivíduos que tentam resolver os problemas. O objetivo deste trabalho foi o de utilizá-lo na resolução otimizada de três problemas: OneMax Booleano, cujos genes são expressos por 0 ou 1; OneMax Real, cujos genes são expressos por uma variável real de 0 a 1; e uma adaptação do Problema do Caixeiro Viajante que permite utilizar atalhos entre as cidades. Para otimizar o AG, foram feitos dois acréscimos ao código. O primeiro deles foi o elitismo entre as gerações, mantendo o melhor indivíduo imune a variações. O segundo deles foi a implementação própria de um módulo extra chamado Algoritmo Genético Adaptativo (AGA), adicionado ao código do AG e responsável por atualizar o parâmetro de mutação de acordo com a evolução da população. Os três problemas foram simulados com uso do AG com parâmetros estáticos e com uso do AGA. As simulações decorrentes trouxeram diferenças pequenas do AGA comparado ao AG estático para os problemas OneMax com as mesmas entradas, e uma performance significativamente melhor do AGA para o problema do Caixeiro Viajante. Tais resultados nos levaram a concluir que o AGA utilizado é promissor na resolução de problemas.				
12. GRAU DE SIGILO: <input checked="" type="checkbox"/> OSTENSIVO <input type="checkbox"/> RESERVADO <input type="checkbox"/> SECRETO				