INSTITUTO TECNOLÓGICO DE AERONÁUTICA

Cássio dos Santos Sousa

Resolução otimizada de problemas com uso de algoritmos evolutivos

Trabalho de Graduação
2016

Computação

Número da CDU (tamanho 10)

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.

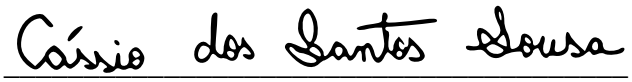
CESSÃO DE DIREITOS

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

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. Juliana de Melo Bezerra
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

Este trabalho focou no uso de Algoritmos Evolutivos (AE), mais especificamente de um Algoritmo Genético (AG), para a resolução de diferentes problemas. Tais algoritmos trabalham com populações onde a menor parte de um indivíduo dela é o gene, o qual contém informação suficiente para que o indivíduo tente resolver o problema. O AG desenvolvido foi utilizado para três problemas: OneMax Booleano, com genes expressados por 0 ou 1; OneMax Real, com genes expressados no intervalo [0, 1); e uma variação do Problema do Caixeiro Viajante que permite utilizar atalhos entre as cidades. De modo a otimizar o AG, utilizou-se duas implementações extras. A primeira delas foi a de elitismo entre as gerações, mantendo o melhor indivíduo imune a variações. A segunda foi o uso não só da versão estática do AG, na qual os parâmetros de entrada são fixos durante a execução, mas também de um módulo extra chamado Algoritmo Genético Adaptativo (AGA), o qual foi adicionado ao código do AG e permite que parâmetros de variação (recombinação e mutação) mudem durante a execução. A conclusão deste trabalho foi a de que, por se utilizar a mesma implementação do AG para diferentes problemas (com exceção das entradas específicas de cada problema), o tratamento que facilitaria a convergência de um problema para a(s) melhore(s) resposta(s) acabou não sendo o melhor para outros problemas. Em termos de otimização, o uso do AGA se mostrou eficaz no encontro de boas soluções, e deve ser dado mais atenção a ele. Para trabalhos futuros, sugere-se, quando o foco for trabalhar em um problema específico, moldar o AG para se beneficiar desta escolha ao máximo, e utilizar também alguma versão de AGA em sua implementação.

Abstract

This work focused on using Evolutionary Algorithms (EA), more specifically Genetic Algorithms (GA), to solve different problems. Such algorithms work with populations where the smallest portion of an individual is the gene, which contains enough information for the individual to try to solve the problem. The GA developed was used for three problems: Boolean OneMax, with genes expressed by 0 or 1; Real OneMax, with genes expressed in the interval [0, 1); and a variation of the Traveling Salesman Problem, in which it's possible to go through shortcuts between the cities. In order to optimize the GA, two extra implementations were used. The first one was the elitism between generations, keeping the best individual immune to variations. The second one was using not only a static version of the GA, in which the input parameters are fixed during the execution, but also an extra module called Adaptive Genetic Algorithm (AGA), which was added to the GA's code and allows variation parameters (recombination and mutation) to change during the execution. The conclusion of this work was that, since the same GA was used to different problems (with the exception of problem-specific inputs), the treatment that would ease convergence for one problem towards the best solution(s) is not the same for other problems. In terms of optimizations, using an AGA turned out to be effective for finding good solutions. For future works, we suggest, when the focus is to solve a specific problem, to mold the GA to benefit from this choice to the fullest, and use some version of AGA in its implementation, too.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Lista de Listagens

Lista 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.

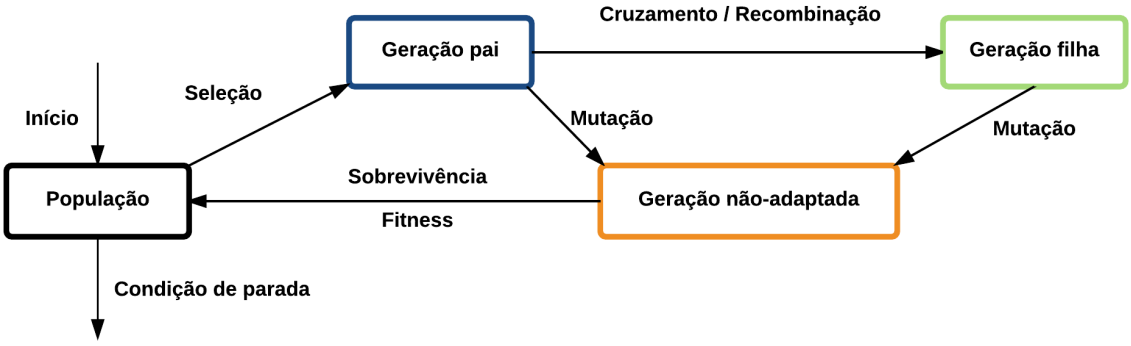


Figura 1: Framework de um algoritmo evolutivo.

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 [7], 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] ← ∞;
        prev[v] ← desconhecido;
        Q.adicionar(v);
    end
    dist[cidade] ← 0;
    while Q não estiver vazio do
        u ← VerticeEmQComMenorDist(Q, dist);
        Q.remover(u);
        foreach vizinho w de u do
            atalho ← dist[u] + DistanciaEntre(u, w);
            if atalho < dist[w] then
                dist[w] ← atalho;
                prev[w] ← 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 [8]:

Algoritmo 2: Pseudocódigo de um Algoritmo Evolutivo.

```
AG(fitness) begin
    P ← InicializarPopulacao(fitness);
    P.fitness();
    t = 0;
    while t < número de gerações do
        Pais ← Selecao(P);
        Filhos ← Crossover(Pais);
        P ← P ∪ Filhos;
        Mutacao(P);
        P.fitness();
        P ← Sobrevivem(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.

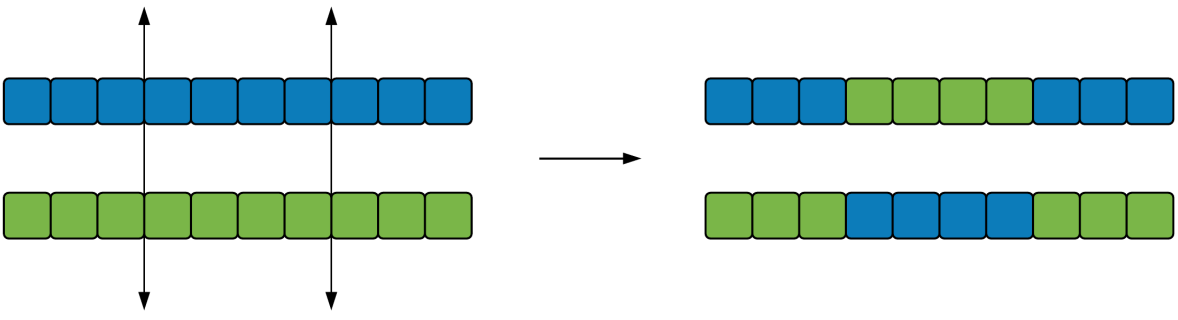


Figura 2: Ação de crossover em dois pontos.

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 [3,5,12]. 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 [3,5,12]. 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 [11]. 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) [13].

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 [9, 14] e implementar uma versão única, 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:

$$\left| \frac{f_{best} - \bar{f}}{\bar{f}} \right| \tag{4.1}$$

- Se este desvio for menor que um valor p_{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 quão grande deve ser o 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 [10], p_m será limitado ao intervalo [0.001, 0.5] (se p_m tentar extrapolar estes limites, ele retornará ao valor extremo mais próximo);

- Mesmo assim, pode acontecer de p_m ficar travado em um dos extremos e o AG não entender o que fazer. Para resolver isto, se p_m ficar travado em um destes limites, p_{m0} começará a variar na direção oposta. Isto é, se p_m atingir o valor de 0.5, p_{m0} começará a diminuir em direção ao valor 0.001, e vice-versa.

- O incremento/decremento para p_m e p_{m0} 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 a seguir. 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 ← (valor inicial de p_m na inicialização do AG);
    ε ← 0.0001;
    foreach ciclo de operações de evolução do
        f̄ ← (média dos valores de fitness);
        if f̄ < ε then
            return
        end
        f_best ← (melhor valor de fitness na população);
        desvio ← |(f_best − f̄)/f̄|;
        if desvio < p_m0 then
            p_m ← min(0.5, p_m + 0.001);
            if p_m == 0.5 then
                p_m0 ← max(0.001, p_m0 − 0.001);
            end
        end
        else
            p_m ← max(0.001, p_m − 0.001);
            if p_m == 0.001 then
                p_m0 ← min(0.5, p_m0 + 0.001);
            end
        end
    end
end
```

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:

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

Os gráficos criados tiveram como base uma única simulação do AG para o problema e parâmetros de entrada escolhidos.

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 fariam com que ele trouxesse o valor máximo. As figuras 3 e 4 demonstram exatamente isso, com convergência completa dos indivíduos para a solução ótima após 93 gerações.

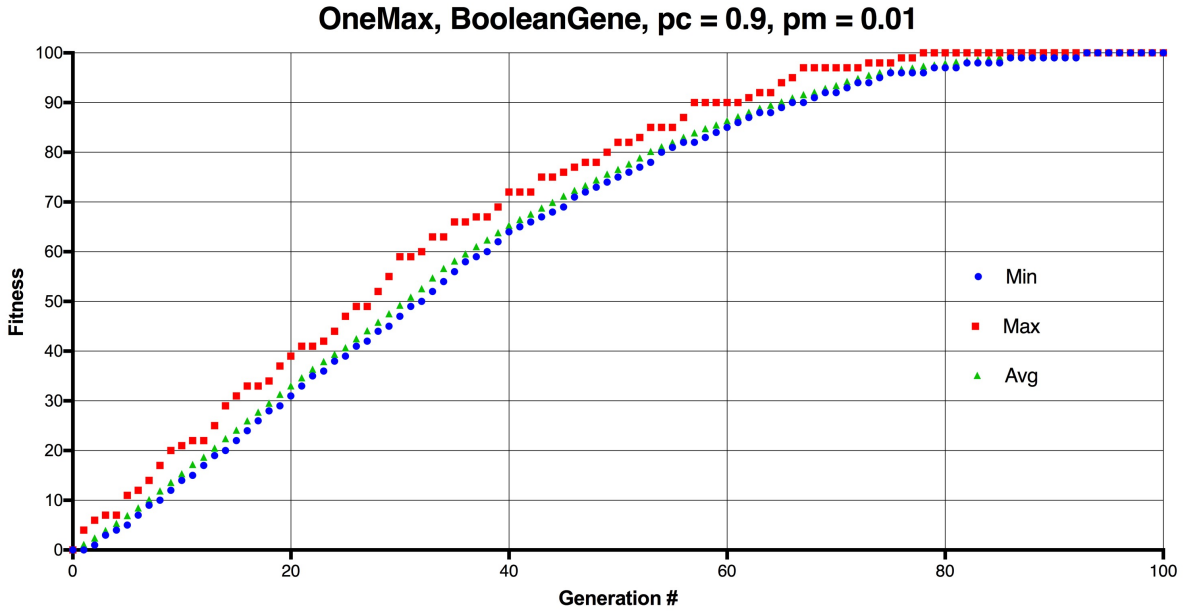


Figura 3: Evolução do fitness para o problema do OneMax Booleano mostrando mínimo, máximo e valor médio ($p_c = 0.9$, $p_m = 0.01$).

5.2.2 Caso Adaptativo

5.3 OneMax Real

Texto.

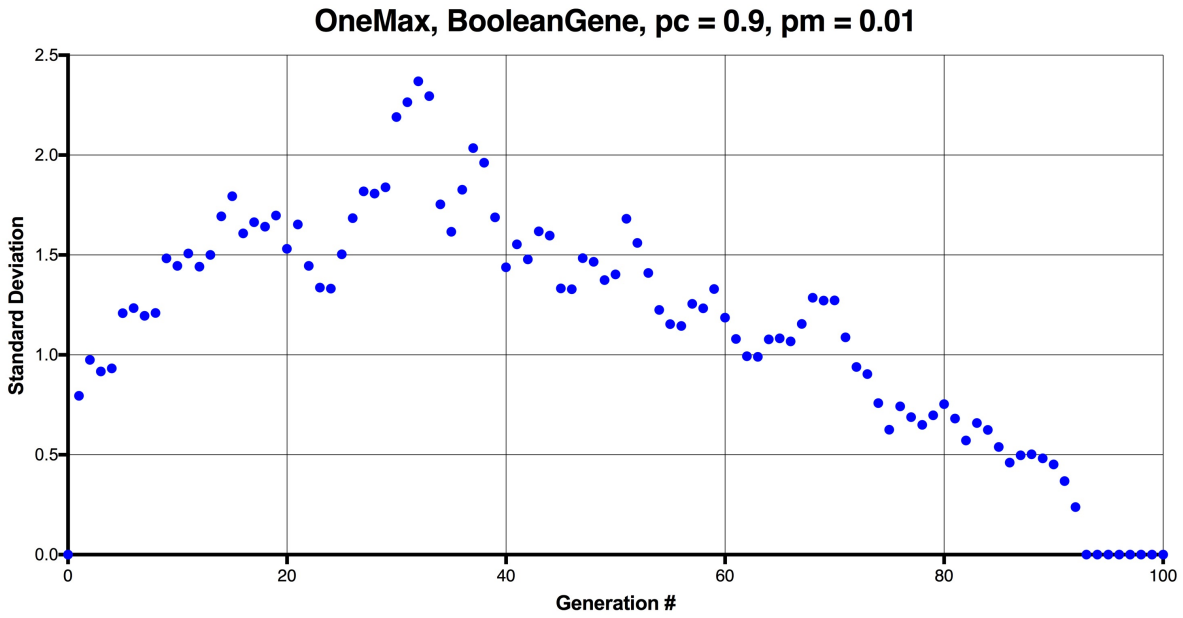


Figura 4: Desvio padrão ao longo das gerações para o problema do OneMax Booleano ($p_c = 0.9$, $p_m = 0.01$).

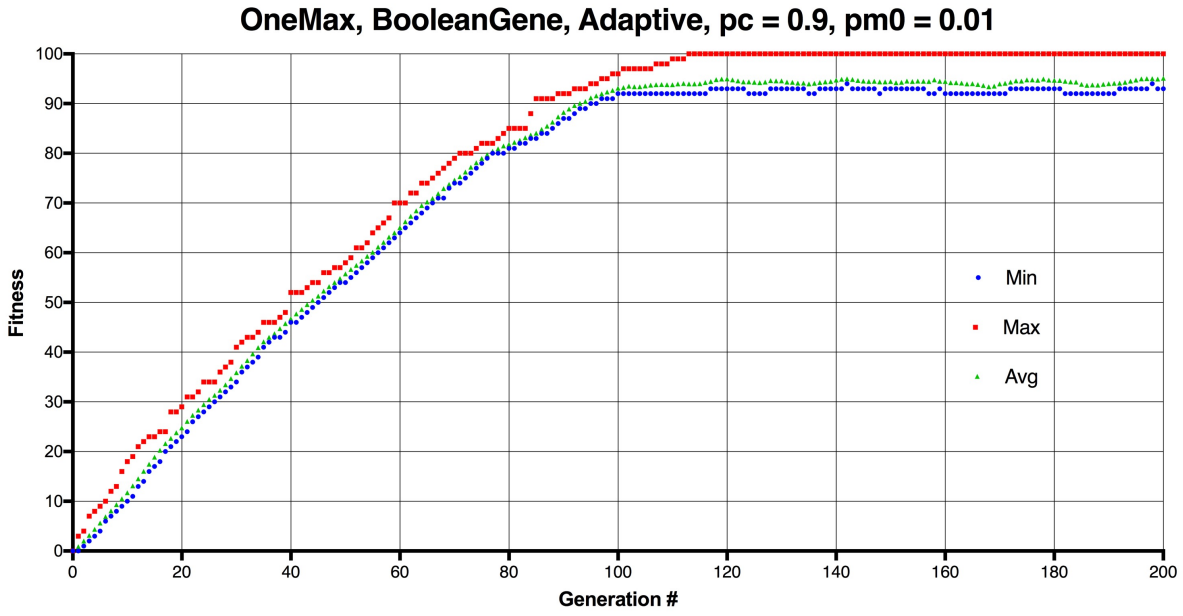


Figura 5: Evolução do fitness para o problema do OneMax Booleano Adaptativo mostrando mínimo, máximo e valor médio ($p_c = 0.9$, $p_{m0} = 0.01$).

5.4 Caixeiro Viajante Adaptado

Texto.

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]
```

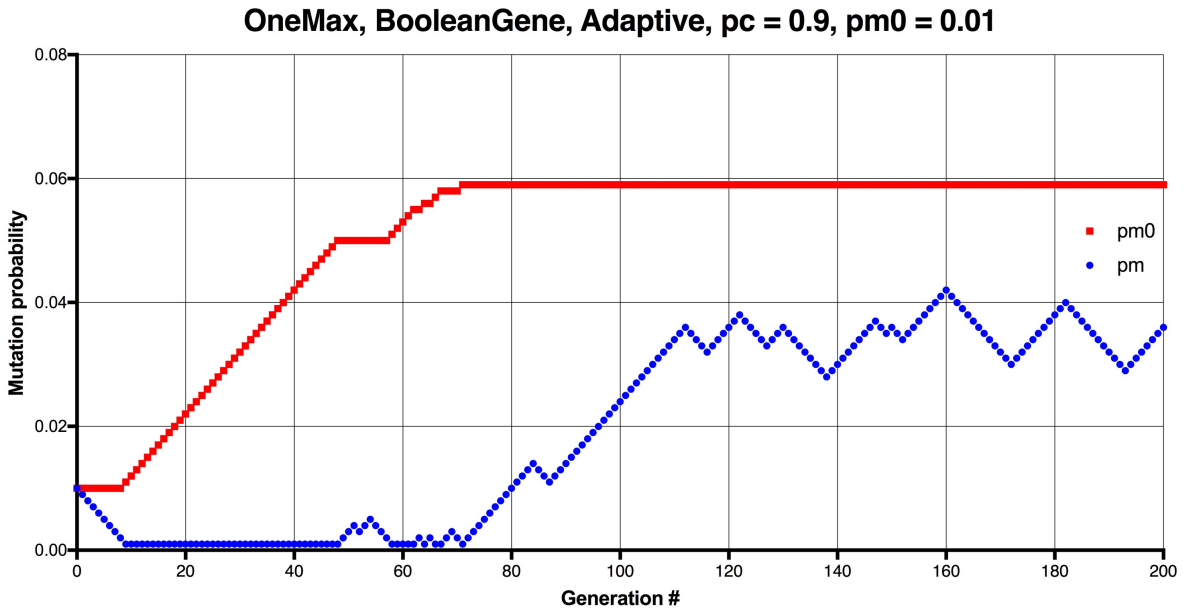


Figura 6: Probabilidade de mutação ao longo das gerações para o problema do OneMax Booleano Adaptativo ($p_c = 0.9$, $p_{m0} = 0.01$).

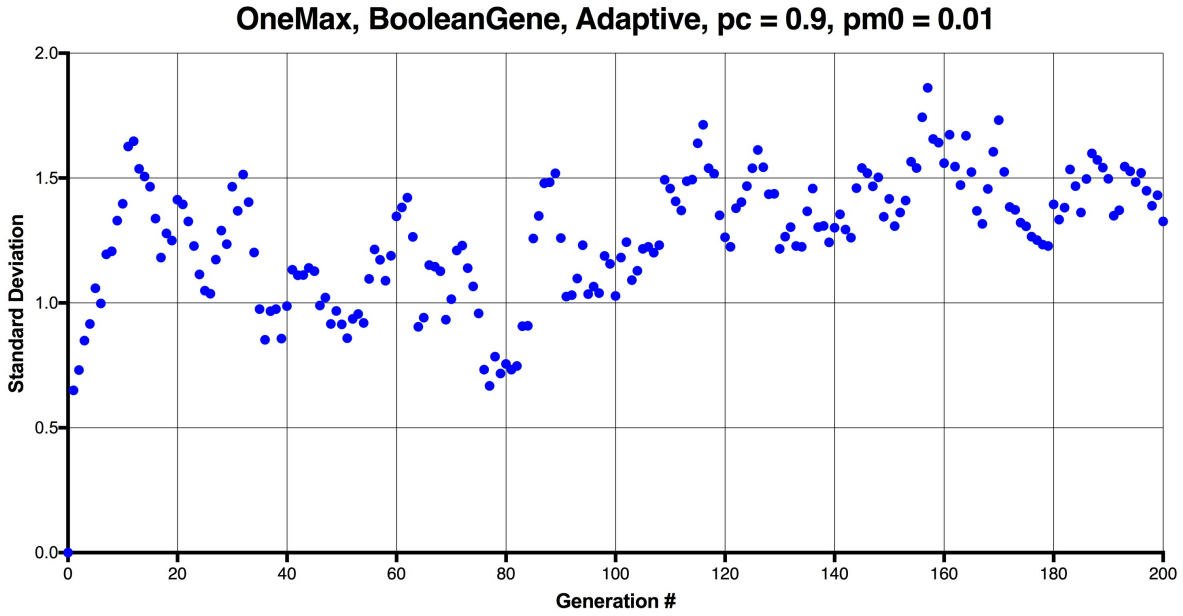


Figura 7: Desvio padrão ao longo das gerações para o problema do OneMax Booleano Adaptativo ($p_c = 0.9$, $p_{m0} = 0.01$).

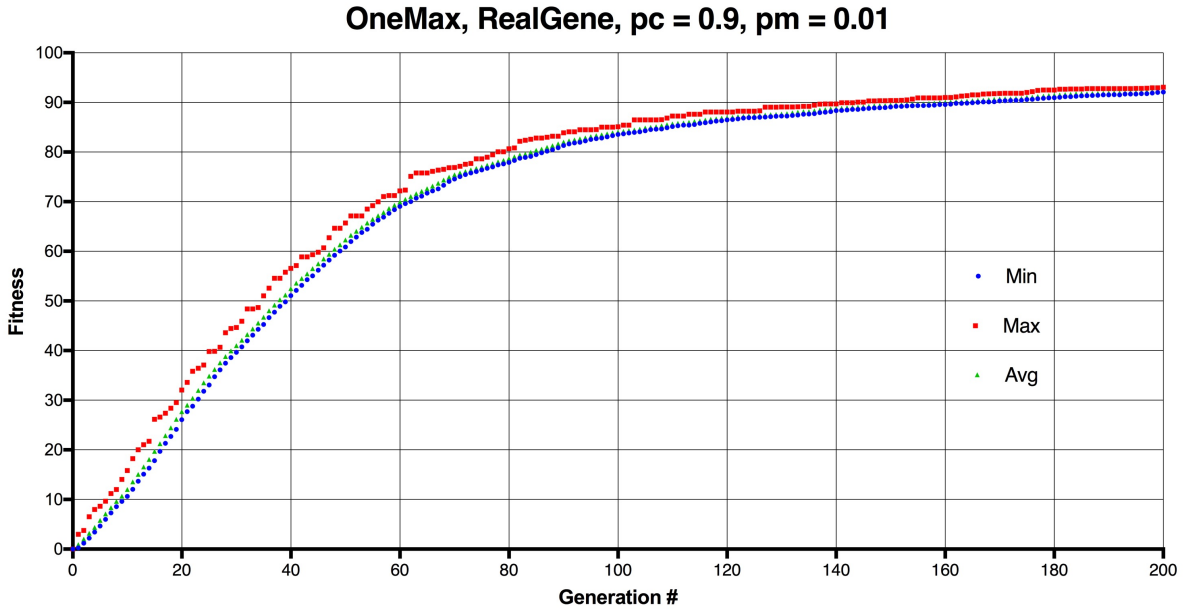


Figura 8: Evolução do fitness para o problema do OneMax Real com mínimo, máximo e valor médio, com $p_c = 0.9$ e $p_m = 0.01$.

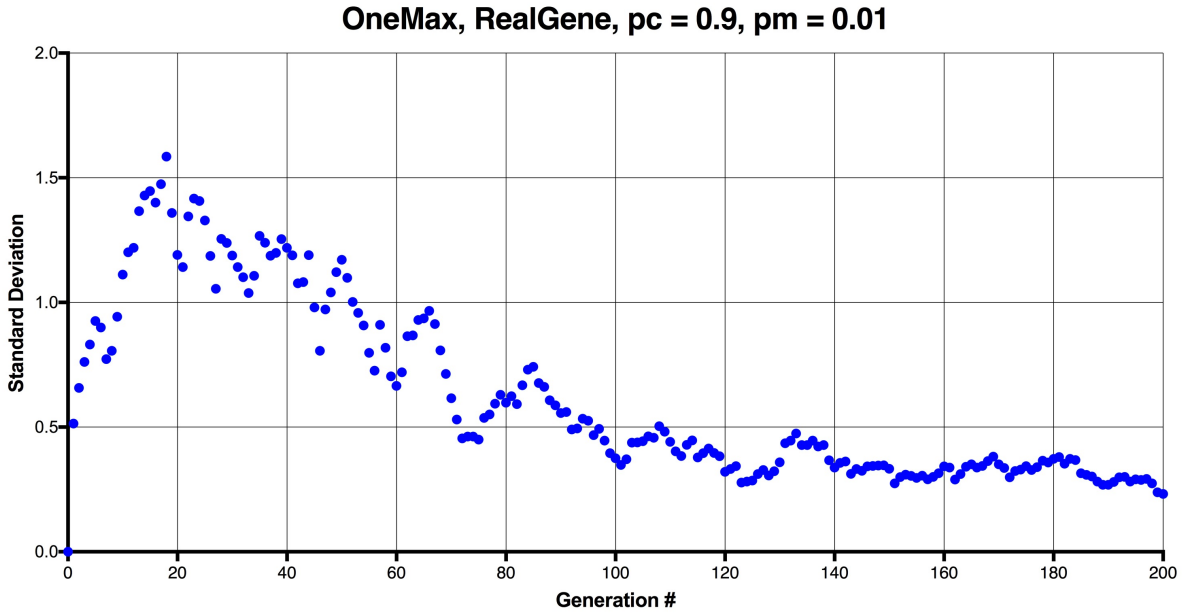


Figura 9: Desvio padrão ao longo das gerações para o problema do OneMax Real, com $p_c = 0.9$ e $p_m = 0.01$.

6 Conclusões e Trabalhos Futuros

Texto aqui.

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 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.

8 ICMC-USP. Algoritmos genéticos. `http://conteudo.icmc.usp.br/pessoas/andre/research/genetic/`. [Online; acessado em 17 de novembro de 2016].

9 D. Jakobović and M. Golub. Adaptive genetic algorithm. *CIT. Journal of computing and information technology*, 7(3):229–235, 1999.

10 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.

11 M. Mitchell. *An introduction to genetic algorithms.* MIT press, 1998.

12 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].

13 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.

14 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.

APÊNDICE A – 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__(self):
        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 para os 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 (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\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()

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

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

        # Deviation of the best fitness to the average
        deviation = abs((self.best_fitness() - self.avg_fitness()) /
           (self.avg_fitness()))

        # Reduce pm if deviation is high
        if deviation <= self.pm0:
            self.pm = min(0.5, self.pm + 0.001)
            if self.pm is 0.5:
                self.pm0 = max(0.001, self.pm0 - 0.001)

        # Increase it, otherwise
        else:
            self.pm = max(0.001, self.pm - 0.001)
            if self.pm is 0.001:
                self.pm0 = min(0.5, self.pm0 + 0.001)

        if self.print_stats:
            print 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('\n')

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

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

APÊNDICE D – 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 – 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 – 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_gsize = len(all_cities)
```

APÊNDICE G – 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	2. DATA	3. REGISTRO N°	4. N° DE PÁGINAS
TC			15

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:

Algoritmo Evolutivo, Inteligência Artificial.

9.PALAVRAS-CHAVE RESULTANTES DE INDEXAÇÃO:

Solicite preenchimento dos campos 2, 3 e 9 – envie este formulário para doc.pt@ita.br

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:

Insira algum texto interessante aqui.

12. GRAU DE SIGILO:

(X) OSTENSIVO **( ) RESERVADO** **( ) SECRETO**