UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DEPARTAMENTO ACADÊMICO DE COMPUTAÇÃO
ENGENHARIA DE COMPUTAÇÃO

FERNANDO AUGUSTO ALVES SANCHES CARDOSO NEVES

ANÁLISE DA EFICIÊNCIA DE UM ALGORITMO GENÉTICO APLICADO AO SUDOKU

TRABALHO DE CONCLUSÃO DE CURSO

CORNÉLIO PROCÓPIO
2020

FERNANDO AUGUSTO ALVES SANCHES CARDOSO NEVES

ANÁLISE DA EFICIÊNCIA DE UM ALGORITMO GENÉTICO APLICADO AO SUDOKU

Trabalho de Conclusão de Curso apresentado ao Engenharia de Computação da Universidade Tecnológica Federal do Paraná, como requisito parcial para a obtenção do título de Engenheiro de Computação.

Orientador: Danilo Sipoli Sanches
Universidade Tecnológica Federal do Paraná

CORNÉLIO PROCÓPIO
2020

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Câmpus Cornélio Procópio
Diretoria de Graduação
Coordenação de Engenharia de Computação
Engenharia de Computação

TERMO DE APROVAÇÃO

Análise da Eficiência de Um Algoritmo Genético Aplicado ao Sudoku

por

Fernando Augusto Alves Sanches Cardoso Neves

Este Trabalho de Conclusão de Curso de graduação foi julgado adequado para obtenção do Título de Bacharel em Engenharia de Computação e aprovado em sua forma final pelo Programa de Graduação em Engenharia de Computação da Universidade Tecnológica Federal do Paraná.

Cornélio Procópio, 25/11/2020

_____

Prof. Dr. Danilo Sipoli Sanches

_____

Prof. Dr. Henrique Yoshikazu Shishido

_____

Prof. Dr. Silvio Ricardo Rodrigues Sanches

"A Folha de Aprovação assinada encontra-se na Coordenação do Curso"

AGRADECIMENTOS

Dedico este trabalho à minha mãe Maria Aparecida por ter sempre me apoiado, tanto financeiramente quanto psicologicamente. Não posso esquecer também de agradecer minha namorada Mirian Quatroque por estar sempre do meu lado, nos momentos fáceis, difíceis e até mesmo nos momentos que me destruiam. Devo toda a minha vida a estas mulheres!

Além disso, agradeço a UTFPR e aos professores do curso de engenharia de computação por terem me proporcionado tudo que proporcionaram e por me guiarem pelo caminho correto. Sei que o profissional que me tornei foi resultado de um bom trabalho deles e só posso agradecê-los por isso.

A vida nunca é uma linha reta, sempre tem altos e baixos. O importante é aprender com as dificuldades e sempre ver nas dificuldades uma oportunidade. (LEMANN, Jorge Paulo, 1998).

RESUMO

NEVES, Fernando. Análise da eficiência de um algoritmo genético aplicado ao Sudoku. 2020. 29 f. Trabalho de Conclusão de Curso – Engenharia de Computação, Universidade Tecnológica Federal do Paraná. Cornélio Procópio, 2020.

Este trabalho tem como principal objetivo estudar como funciona um algoritmo genético e se ele pode ser eficiente em um determinado contexto. O contexto aplicado neste trabalho é um problema combinatório, no caso, o jogo Sudoku. Para atingir tal objetivo, primeiramente será implementado um algoritmo genético e, a partir dos *outputs* de diversos jogos de Sudoku, será realizado uma análise estatística para podermos descrever se o algoritmo genético é uma boa alternativa para análises combinatórias.

Palavras-chave: Algoritmo Genético. Problema Combinatório. Análise de eficiência. Sudoku.

ABSTRACT

NEVES, Fernando. . 2020. 29 f. Trabalho de Conclusão de Curso – Engenharia de Computação, Universidade Tecnológica Federal do Paraná. Cornélio Procópio, 2020.

This work has as main objective to study how a genetic algorithm works and if it can be efficient in a given context. The context applied in this work is a combined problem, in this case, the Sudoku game. To achieve this goal, a genetic algorithm will be implemented and, based on textit outputs of several Sudoku games, a statistical analysis will be carried out in order to describe whether the genetic algorithm is a good alternative for combined combinations.

Keywords: Genetic Algorithm. Combinatorial Problem. Efficiency analysis. Sudoku.

LISTA DE FIGURAS

LISTA DE ALGORITMOS

SUMÁRIO

1 INTRODUÇÃO

Sudoku é um tipo de jogo japonês de quebra-cabeça que recentemente ficou muito popular na Europa e na América do Norte. Mesmo o jogo sendo considerado japonês, sua primeira aparição foi em uma revista norte americana em 1979 e só depois começou a rodear o Japão, onde ficou popular em 1986. Mais tarde, em 2005 começou a ficar famoso no ocidente (PERCÍLIA, 2019).

Este jogo é um quebra-cabeça que se baseia na concordância racional de números, proveniente da expressão japonesa "Suuji wa dokushin ni kagiru" traduzida"para "Os números devem ser únicos" (PERCÍLIA, 2019). O jogo é composto por uma espécie de grade de nove blocos de altura por nove blocos de largura (9x9) que são divididas em outras nove sub-grades de três de altura por três de largura (3x3), contabilizando um total de 81 posições. Os jogos fornecem números aleatórios, fixos, espalhados por esta grade logo de início, onde a aleatoriedade desses números é descrita de acordo com a dificuldade escolhida pelo jogador.

A finalidade do jogo é preencher todas essas 81 posições com números de uma forma que cada linha, coluna e sub-grade contenham os números de um a nove uma e somente uma vez. O jogo, mesmo parecendo muito simples devido as suas regras, é bem desafiador, porém não passa de matemática, lógica e análise. Devido a isso, podemos dizer que seja possível aplicar algum algoritmo para resolver o problema combinatório intríseco ao Sudoku. Existem diversos algoritmos que podem resolver um Sudoku, onde um deles é o algoritmo genético.

O princípio básico de todas as abordagens de algoritmos evolucionários é que dada uma população de indivíduos ou um conjunto de soluções, pressões ambientais criam processos de seleção natural. Processos que privilegiam os melhores indivíduos encontrados ou até mesmo as melhores soluções encontradas, o que por sua vez, causa um acréscimo na adequação dos indivíduos para determinado problema.(EIBEN A. E.; SMITH, 2003) O algoritmo genético, sendo um algoritmo evolucionário segue esta mesma lógica. No nosso AG, essa adequação é medida através de uma equação chamada *fitness*, que mede a qualidade das soluções candidatas, atribuindo um valor que mede sua adequação.

Com base no valor atribuído a adequação da solução, as melhores soluções são selecionadas para darem origem a uma nova população através dos processos de crossover e mutation, estes serão explicados mais especificamente. Ao fim destes processos, esses novos candidatos competem com os candidatos da geração anterior, levando em consideração o valor obtido pela equação fitness, para que seja definido a nova geração. Este processo é repetido até que a solução seja qualificada o suficiente ou até que o número de gerações específico seja obtido (AKEMI, 2018).

Algoritmos Genéticos são, basicamente, os algoritmos evolucionários mais comuns. É uma operação computacional que usufrui de todos os processos citados acima que fora proposto por John Holland em 1975 em seu livro Adaption in Natural and Artificial Systems.

(HOLLAND, 1975)

1.1 Análise de eficiência

Um tópico muito importante dentro da ciência da computação é a análise de eficiência dos algoritmos. Este tópico examina vários aspectos de um algoritmo, entre eles a precisão e velocidade.

A precisão diz respeito a qualidade da resposta, ou seja, o quão perto o *output* está de uma solução real para o problema tratado pelo algoritmo. Enquanto a velocidade é, como o próprio nome já diz, a velocidade em que o algoritmo chega nesta resposta. Existem diversas formas de realiza essa análise, neste trabalho realizaremos uma análise empírica. Esta análise consiste em, nada mais nada menos, que rodar o algoritmo de forma exaustiva, obtermos dados o suficiente para fazermos análise de velocidade e qualidade, e comparar cada velocidade obtida para cada série de parâmetros usado (COUTINHO, 2017).

Outros aspéctos podem ser avaliados quando queremos analisar eficiência, tais quais: uso de memória primária ao compilar o programa, uso de memória secundária para armazenar o programa,uso de processamento do processador do computador, entre outros.

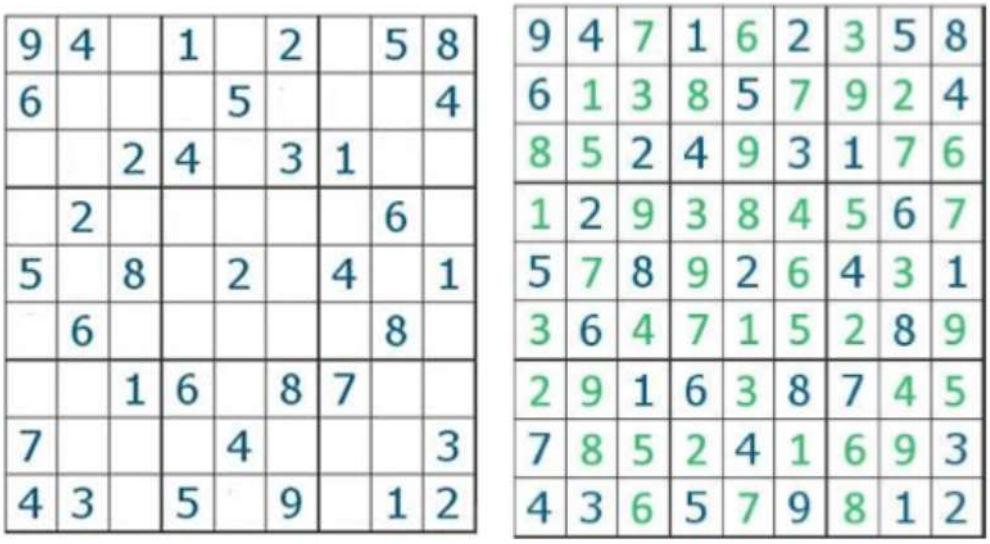
2 REVISÃO DE LITERATURA

Para prosseguir com a metodologia, é bom explorarmos um pouco mais, de forma mais aprofundada, trabalhos que seguem a mesma perspectiva que a deste trabalho. Logo, essa seção é destinada a fundamentação teórica, explicando conceitos do Sudoku, e abrindo uma concepção mais específica dos Algoritmos Genéticos e como eles podem ser aplicados a resolução de um Sudoku.

2.1 Solução do Sudoku

Como dito previamente, o Sudoku é um jogo de quebra cabeça que consiste em uma matriz 9x9 divididas em nove matrizes de 3x3. São dados alguns números fixos dentre dessas matrizes e as regras são simples: cada linha, coluna e matriz 3x3 deve conter os números de um a nove uma e somente uma vez.

Figura 1 – Exemplo de solução do Sudoku



Fonte: Autoria própria

Foi comprovado por Lawler e Rinnooy (LAWLER; RINNOOY, 1985) que, na versão 9x9 de Sudoku, existem cerca 6,671 x 10^{21} soluções possíveis, o que torna o Sudoku em um problema NP-Completo. O número de soluções possíveis é diferenciado de acordo com o nível de dificuldade do quebra-cabeça. (SCHIFF, 2015)

Devido a isso, este trabalho propõe analisar a eficiência de um algoritmo genético aplicado a diversas dificuldades do Sudoku. O algoritmo será testado variando seus parâmetros, para termos noção do que o faz mais influencia na velocidade de processamento, precisão e consumo de memória.

2.2 Heurística e Meta-Heurística

Heurísticas são procedimentos que tratam problemas de otimização sem dispor de garantias teóricas, nem que a solução ótima exata seja obtida. Heurística pode ser tratada como um procedimento de busca de boas soluções (GASPAR-CUNHA; TAKAHASHI, 2012). Já a meta-heurística é definida como estratégias de alto nível que guiam uma heuristica subjacente, buscando aumentar o desempenho desta, tendo como objetivo principal não se manter em ótimos locais (BLUM, 2003).

2.3 Algoritmos Genéticos

Os Algoritmos Genéticos (AG) são meta-heurísticas associadas com a evolução, como citado anteriormente. Esta evolução é alcançada através da criação de novas gerações com uma qualificação melhor perante aos seus antecessores. Essa característica foi inspirada na forma como os seres vivos sobrevivem e passam seu material genético para as próximas gerações, utilizando os princípios de seleção natural propostos por Charles Darwin (AKEMI, 2018).

2.3.1 Estrutura básica de um AG, relacionando com Sudoku

Devido ao AG ser inspirado ao processo evolutivo natural, ele é estruturado de uma forma que as informações codificadas do sistema, em geral, possam ser comparadas aos cromossomos biológicos.

O cromossomo é a estrutura primordial de um AG. É nele que consiste todas as informações necessárias para a solução do problema em questão. São inúmeras formas existentes de representa-lo, a mais comum é feita através de uma String de números binários. Mas a sua representação, que também pode ser chamado de alfabeto da AG, pode mudar de acordo com o problema a ser resolvido (AKEMI, 2018). A figuras a seguir é um exemplo de representações do cromossomo para a solução do Sudoku:

Figura 2 – Exemplo de cromossomo aplicado ao Sudoku

Fonte: (MANTERE; KOLJONEN, 2006)

Tendo em mente a estrutura de um cromossomo, uma população destes indivíduos é iniciada com genes aleatórios logo no início do algoritmo. O tamanho desta população é definido de acordo com o que problema necessita. Existem muitas divergências com a recomendação de um tamanho de população, se ela for pequena demais a cobertura pode ser pequena e se for grande demais, será necessário muito recurso computacional (MANTERE; KOLJONEN, 2006), devido a isso será realizado diversos teste a fins de aumentar a eficiência do AG.

Para que a recombinação ocorra, devem ser previamente selecionados os cromossomos pais da nova geração. Essa escolha deve ser feita de uma forma que os pais desejados tenham

algo a contribuir com a solução, isto é feito através da equação Fitness. Uma equação que define o problema em questão e que normalmente deve ser maximizada ou minimizada. A seguir são expressados alguns exemplos de equação fitness aplicados ao Sudoku:

$$F_{ij}(x) = 45 - \sum_{i=1}^{9} x_{i,j} \quad F_{ij}(x) = 45 - \sum_{j=1}^{9} x_{i,j} \tag{1}$$

$$F_{ij}(x) = 9! - \prod_{i=1}^{9} x_{i,j} \quad F_{ij}(x) = 9! - \prod_{j=1}^{9} x_{i,j} \tag{2}$$

O primeiro exemplo mostra que a soma de cada coluna e linha deve ser igual a 45, já a segunda mostra que o produto de cada coluna e linha deve ser igual a nove fatorial. (MANTERE; KOLJONEN, 2006)

Mesmo existindo essa tal avaliação, não devem ser escolhidos apenas os melhores valores, a fim de evitar a convergência total no máximo local. Em outras palavras alguns valores parecem ser os melhores quando comparados aos outros indivíduos, porém não são efetivamente os melhores para a solução do problema (AKEMI, 2018).

Para tratar este problema alguns métodos de seleção são usados, como por exemplo o Roullete Wheel, um dos métodos mais utilizados (MCCAFFREY, 2017). O método aplica uma probabilidade para que o cromossomo em evidência faça a recombinação, cuja probabilidade é definida pelo valor do fitness do cromossomo.

Alguns algoritmos genéticos contam também com um processo de repetição de cromossomos com as melhores soluções. Este processo ocorre devido ao elitismo criado a fim de que o algoritmo convirja mais eficientemente para a solução.

Após a seleção destes indivíduos inicia-se o processo de recombinação, também chamado de cruzamento ou crossover. Neste estágio os genes de dois indivíduos, também chamados de pais, são recombinados, gerando novos membros.

Existem vários métodos de realizar a recombinação, dentre elas, a mais comum é o crossover de um ou mais pontos: primeiramente é determinado uma porcentagem para a probabilidade de realização de crossover (caso o cruzamento não seja realizado os pais permanecem estáveis para a próxima geração) e logo após é definido um ponto onde será realizado um "corte" e, subsequentemente ocorrer a recombinação. (MANTERE; KOLJONEN, 2006)

Enquanto o Crossover cuida de troca de informações entre dois cromossomos a etapa da mutação realiza a alteração de informação genética de um indivíduo só, com o intuito de criar uma maior diversidade de genes. Essa técnica é explorada a fim de criar uma válvula de escape para soluções ótimas locais. (AKEMI, 2018)

Há diversas formas de tratar esta mutação, a mais comum é a mutação única, que consiste em atrelar uma probabilidade de mutação para cada gene, onde somente ele deve ser afetado pela mutação. (MANTERE; KOLJONEN, 2006)

3 METODOLOGIA

Este trabalho baseou-se em uma estratégia quantitativa de pesquisa, que tem caráter exploratório por meio da otimização de um algoritmo genético através da busca exaustiva de parâmetros mais adequados, a fim de aperfeiçoar a qualidade do algoritmo. Neste capítulo pretendemos demonstrar os procedimentos metodológicos utilizados no trabalho.

3.1 DELINEAMENTO DA PESQUISA

Este trabalho, como dito antes, tem uma abordagem quantitativa, ou seja é um estudo que será realizado através de proporções numéricas, dados coletados e análise de gráficos. Além disso terá como objetivo um aprofundamento exploratório, que, em outras palavras, é uma pesquisa a fim de buscar maior familiaridade com o problema em questão, no caso, o algoritmo genético.

3.2 IMPLEMENTAÇÃO DO ALGORITMO

A implementação do algoritmo seguiu a metodologia ágil, mais especificamente, seguindo o framework Scrum. Nesta metodologia, os projetos são divididos em ciclos (tipicamente mensais) chamados de *Sprint*. O Sprint representa o tempo no qual um conjunto de atividades devem ser executados. Para o desenvolvimento e implementação do algoritmo foram feitas no total 12 *Sprint* de duas semanas cada. O algoritmo foi construido inteiramente em Python 3 devido a suas bibliotecas que facilitam o manuseio e transformação de conjuntos e listas de dados.

3.2.1 Início da implementação do Algoritmo

Logo na primeira *Sprint* o algoritmo genético começou a ser desenvolvido. O trabalho no início do projeto foi o de construir uma função que lê um arquivo que se encontra no mesmo diretório de onde o algoritmo está. Este arquivo se trata do Sudoku em formato matricial que é passado como entrada para o algoritmo. Além disso, foi implementado outras duas funções: uma que em que avalia o arquivo e indica se ele se encontra no formato desejado e outra que transforma a matriz em três diferentes dimensões, linhas, colunas e em sub-grades para subsequente avaliação.

3.2.2 Implementação das funções de um algoritmo genético

O próximo passo, após o algoritmo ler o arquivo de entrada, é o de gerar uma solução aleatória para o problema, também conhecido como cromossomo. A função para gerar essa solução começou a ser estruturada e implementada logo na terceira *Sprint*.

As soluções geradas por essa função são realizadas da seguinte forma: para cada subgrade adiciona-se números de 1 a 9 (contendo somente uma cópia para cada número) em ordem aleatória, excluindo somente aqueles que já foram dados dentro de cada subgrade. Este método foi adotado a fim de diminuir o número de gerações, tendo em vista de que o número de repetições dentro de cada subgrade já será minimizada o máximo possível logo na primeira geração.

Tendo essa função implementada, podemos gerar um conjunto de soluções para criar nossa população inicial. Essa população inicial é a base do algoritmo, são os antecessores de todos os outros cromossomos, como dito na seção anterior. O número de cromossomos para a população varia de projeto para projeto, tendo como paradigma eficiência e performance. Como o intuito deste trabalho é analisar eficiência, iremos variar a população entre 4 a 100, com intervalo de 4 em 4 indivíduos.

Tendo todos estas funções iniciais implementadas, podemos partir para os métodos mais específicos de um algoritmo genético.

3.2.2.1 Função *Fitness*

O método adotado para avaliar o cromossomo pertence a função *Fitness*. A metodologia utilizada no trabalho é representada pelos seguintes pseudocódigos:

Algoritmo 1: Equação Fitness para as Linhas

Input: O indivíduo A com L linhas e C colunas
Output: A nota N atribuida às linhas do cromossomo A
$index \leftarrow 0$
while *index* $<=$ *L* **do**
  $auxiliar \leftarrow 0$
  for $auxiliar \in C$ **do**
    if *O valor está repetido* **then**
      $N \leftarrow N + 1$
    end
  end
end

Algoritmo 2: Equação Fitness para as Colunas

Input: O indivíduo A com L linhas e C colunas
Output: A nota N atribuida às colunas do cromossomo A
$index \leftarrow 0$
while *index* $<=$ *C* **do**
  $auxiliar \leftarrow 0$
  for $auxiliar \in L$ **do**
    if *O valor está repetido* **then**
      $N \leftarrow N + 1$
    end
  end
end

Algoritmo 3: Equação Fitness para as sub-grades

Input: S Sub-grades em um array de X elementos
Output: A nota N atribuida a cada Sub-grade do cromossomo
$index \leftarrow 0$
while *index* $<=$ *S* **do**
  $auxiliar \leftarrow 0$
  for $auxiliar \in X$ **do**
    if *O valor está repetido* **then**
      $N \leftarrow N + 1$
    end
  end
end

O Fitness do problema é calculado usando a soma das três funções exemplificadas, onde cada uma calcula o número valores duplicados para cada linha, coluna e sub-grade dentro do Sudoku. Logo, quanto mais próximo o valor de zero, melhor a solução encontrada pelo cromossomo. Esta função começou a ser desenvolvida na sexta *Sprint*.

3.2.2.2 Função para Seleção

Assim que temos os valores das notas de cada cromossomo dadas pela função *Fitness*, no início da nona *Sprint*, partimos para a seleção dos indivíduos com melhor nota, mas ainda de forma probabilística, possibilitando, assim, que todos os cromossomos tenham a oportunidade de se reproduzir. Mesmo que sua avaliação seja muito ruim, ele ainda terá a remota chance de ser usado para a próxima geração para que a solução não estagne em um máximo local. O modelo adotado segue a equação:

$$P[i] = \frac{\frac{1}{fitness[i]}}{\sum_{j=0}^{N} fitness[j]} \tag{3}$$

Onde $fitness[i]$ é o valor de fitness para o indivíduo, N é o número de indivíduos na

população e P é a probabilidade do indivíduo em si.

Em outras palavras, a probabilidade é proporcional ao fitness do cromossomo comparado a somatória do total. Este método, também conhecido como *Roulette Wheel*, foi escolhido por sua maior robustez e também devido a curiosidade no quanto esse método seria impactado assim que aumentassemos o número da população.

3.2.2.3 Função para *Crossover*

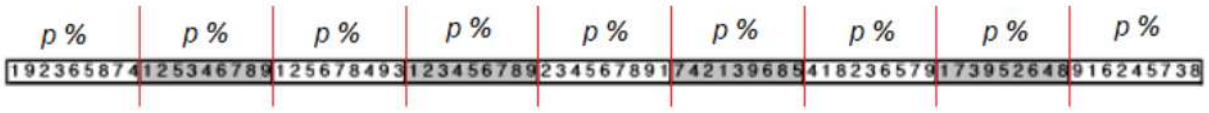
Agora, tendo em mãos toda a população já probabilísticamente selecionada para seguir para a próxima etapa, podemos realizar o *Crossover* delas. Essa etapa, também conhecida como cruzamento, é a fase em que utilizamos dois cromossomos aleatórios dentre os selecionados e cruzamos suas informações para a criação de filhos. Cada dois cromossomos geram dois novos cromossomos, substituindo os pais na próxima geração ou não, a depender do parâmetro de taxa de seleção (caso haja *crossover* os filhos substitem os pais, caso contrário os pais permanecem para a próxima geração).

Este parâmetro será uma das métricas utilizadas. Propriedade na qual irá variar entre 30% a 70% com intervalo de 10% para cada variação.

A metodologia utilizada para essa etapa será o *Single-Point Crossover*. Método no qual escolhe-se somente um ponto de corte em ambas as soluções pais e as recombinam - cada parte é fundida com a parte par da outra solução pai.

A imagem a seguir mostra os pontos com probabilidade de corte para a realização do crossover:

Figura 3 – Exemplo de crossover no cromossomo



Fonte: Autoria Própria

Como a figra ilustra, o *crossover* só poderá ocorrer, entre linhas ou colunas ou até mesmo entre sub-grades, a fim de que o *crossover* não impacte negativamente na equação *Fitness*. Caso haja a possibilidade de cruzarmos cada gene, o algoritmo provavelmente levará mais tempo de processamento. Devido a isso, usaremos as linhas como pontos de corte para o *crossover*, somando um total de 8 possíveis locais de cortes diferentes.

Tendo isso em mente, a probabilidade de corte para cada célula será fixa em $1/8$ para cada linha. Como a probabilidade será fixa, ela não será uma métrica para a análise de eficiência.

3.2.2.4 Função para Mutação

Após a realização do *crossover* e criação da nova geração, é o momento de realizar a mutação. A mutação se trata de mudar elementos de lugar, porém diferentemente do *crossover*, a mudança ocorre somente dentro das linhas, colunas ou sub-grades de somente um cromossomo. Neste caso, como o crossover realiza o corte tendo como base as intersecções das linhas do Sudoku, a mutação ocorrerá entre as linhas, para que tenhamos uma maior variança de informações.

A estratégia adotada na mutação foi a de usar a otimização *Hill Climbing*, a fim de que cada interação seja a mais benéfica possível para a solução. Esse método foi adotado devido ao alto consumo de tempo que seria utilizado caso a mutação fosse randômica, uma vez que parte do trabalho consiste em realizar uma grande quantidade de testes.

Além de tudo isso, será aplicado uma probabilidade para ocorrer a mutação. Essa medida foi tomada a fim de evitar um máximo local após a implementação do *hill climbing*, o que é um problema comum desta otimização. Essa probabilidade será uma das métricas avaliadas para a eficiência do algoritmo e variará entre 30% até 90%, com intervalo de 20%.

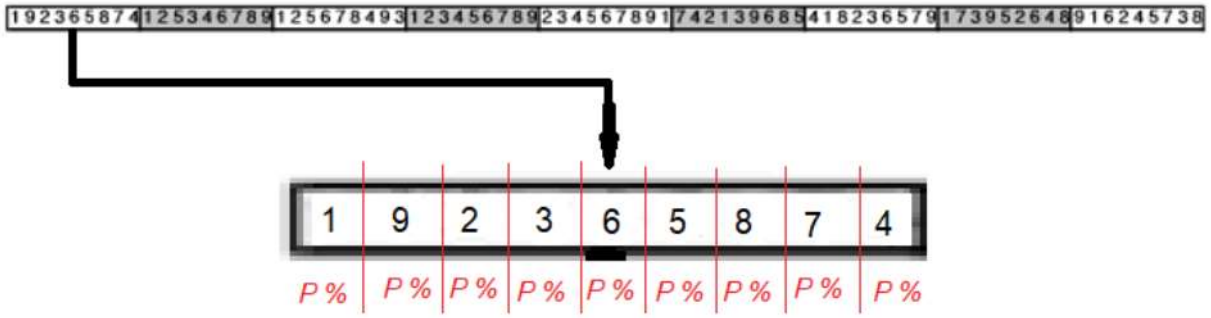
As imagens a seguir ilustram um exemplo do processo de mutação por completo:

Figura 4 – Linhas selecionadas pela probabilidade para sofrer a mutação

Fonte: Autoria Própria

A figura 4 mostra em destaque avermelhado todas as linhas selecionadas para sofrer a mutação. Essas linhas foram selecionadas pela probabilidade que será avaliada no trabalho.

Figura 5 – Probabilidade de cada bit sofrer mutação



Fonte: Autoria Própria

A figura 5 mostra que cada gene possui a mesma chance de sofrer mutação, apenas um gene sofrerá mutação dentro todos os 9. Assim cada gene terá a probabilidade fixa de 1/9, semelhante a probabilidade fixa do crossover.

A figura 6 ilustra o próximo passo, em que o *hill climbing* atua. Um bit aleatório é escolhido e ele troca de lugar com outro. Essa troca não é randômica, como dito antes, ele escolhe o lugar onde melhor impactará no *Fitness* do cromossomo.

Figura 6 – Ação do *Hill Climbing* na linha

Fonte: Autoria Própria

E assim a mutação perdurará, até que todas as linhas de todas os cromossomos da população sofram (ou não) a mutação.

3.2.2.5 Algoritmo Evolucionário

Tendo todas as funções construidas, agora basta implementar a condição de parada do algoritmo. Então na décima segunda *Sprint* estas condições foram implementadas, nas quais são: chegar na solução (com a melhor nota *Fitness* possível, a nota zero) ou o algoritmo deve parar quando o número de gerações chegasse a cem.

3.3 COLETA E TRATAMENTO DE DADOS

O método adotado para a coleta de dados foi a de construir um script em que realiza uma bateria de testes, sendo estes com variação de tamanho da população, variação na taxa de *crossover* e variação na taxa de mutação, como citados nas seções anteriores. O número de experimentos a serem realizados são de 18 mil segundo a equação:

$$N = x \times p \times c \times m \times d \times e \tag{4}$$

Substuindo os valores, onde x é o número de testes por cada condição (a fim de evitar *outliers* possíveis), p é o número de diferentes populações, c é o número de diferentes taxas de *crossover*, m é o número de difentes taxas de mutação, d é o número de diferentes dificuldades a serem analisadas e e é o número de exemplos diferentes para cada dificuldade:

$$N = 4 \times 25 \times 5 \times 4 \times 3 \times 3 = 18000 \tag{5}$$

Tendo essa quantidade de testes em mente sabe-se que serão feitos 6000 experimentos para cada dificuldade. Cada dificuldade terá três problemas distintos, somando 2000 testes para cada problema (9 problemas no total, somando 18000 execuções). Vale lembrar que a condição de parada do algoritmo é quando o *Fitness* chega a 0 ou quando o limite de 100 gerações é atingido.

As fontes de problemas do Sudoku foram retiradas do site 'https://sudoku.com/pt'. Estes problemas são armazenados em um arquivo ".txt"em uma estrutura matricial e em seguida são lidos pelo algoritmo.

Para mais informações sobre o algoritmo e/ou curiosidade de como ele funciona, ele se encontra no link: 'https://github.com/ifertz/SudokuSolver'.

4 ANÁLISE E DISCUSSÃO DOS RESULTADOS

Neste capítulo será discutido os resultados encontrados a partir das 18 mil execuções de teste, usando a metodologia mencionada anteriormente.

4.0.1 Ambiente

O ambiente utilizado para realização de testes foi o mesmo para todas as dificuldades e permutações de parâmetros. Segue configuração do computador ambiente:

- Intel Core i7-8550U @ 1.8GHZ
- 16GB RAM
- Windows 10 64-bit OS

4.1 Resultados

Os resultados obtidos através deste trabalho foram recompensadoras. A implementação foi completamente trabalhosa, porém os resultados foram bons, vide a tabela a seguir:

Tabela 1 – Resultados com o uso de hillclimbing

	Ótimos Encontrados Com Hill Climbing	Porcentagem com relação ao Total (%)
Fácil	6000	100.00
Médio	5996	99.93
Difícil	5997	99.95

Fonte: Autoria Própria

Como a tabela mostra, das 18 mil execuções de teste apenas sete delas falharam. Todas as execuções que falharam chegaram muito próximo a uma solução (fitness muito próximo de 0) mas foram barradas pelo número limite de gerações.

4.1.1 Sem uso de Hill-Climbing

Como dito na seção anterior, o trabalho utiliza a técnica de busca local *Hill-Climbing* dentro da mutação. Esse procedimento foi necessário devido a alta demanda de tempo que o algoritmo exigiria para solucionar os problemas, Além disso havia a possibilidade de estagnação em ótimos locais. A seguir estão os resultados utilizando toda a metologia predeterminada com exceção do *Hill-Climbing* para mostrarmos o quão impactante a busca local é:

Tabela 2 – Resultados sem o uso de hillclimbing

	Ótimos Encontrados sem Hill Climbing	Porcentagem com relação ao Total (%)
Fácil	6000	100.00
Médio	954	15.9
Difícil	654	10.9

Fonte: Autoria Própria

Vide a tabela, os resultados foram desastrosos. Poucas execuções obtiveram resultado ótimo. Isso acontece devido ao alto número de gerações necessárias para solução, e como o limite é de apenas cem, a execução é interrompida.

4.2 Análise dos resultados

Antes de iniciar uma análise da eficiência do algoritmo, é interessante analisar os resultados em si, quais deles falharam e tentar entender o motivo deles terem falhados.

A iniciativa tomada para analisar os resultados foi a de construir um gráfico de pontos para cada dificuldade, usando todos os parâmetros do algoritmo como pontos de avaliação. Os resultados serão, então, distribuidos em três gráficos, um pada cada dificuldade, inciando do mais fácil ao mais difícil (cada subgráfico representa uma combinação de *crossover* e mutação):

Figura 7 – Resultados para a dificuldade fácil, distribuidos através dos parâmetros do algoritmo

Fonte: Autoria Própria

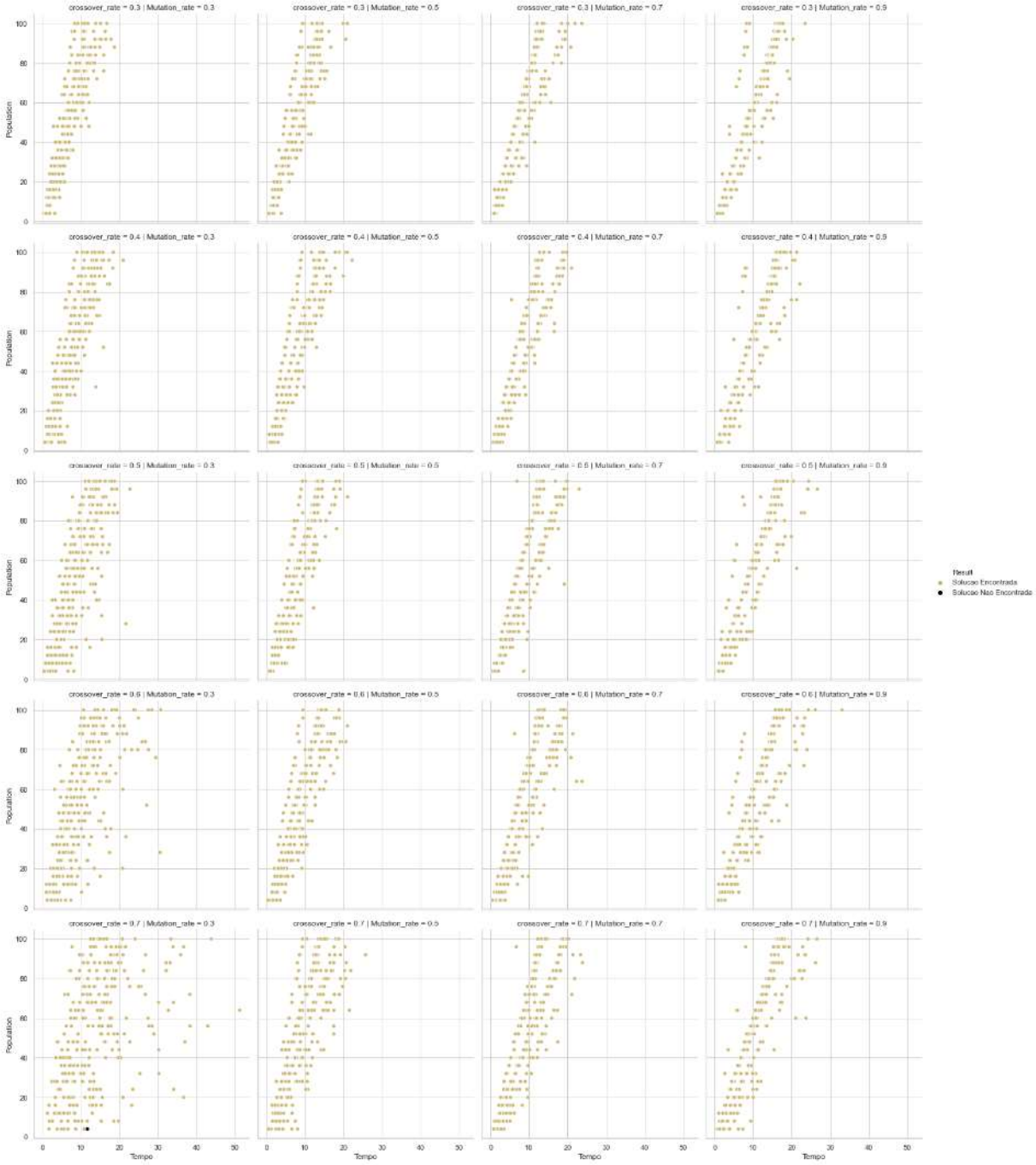
Como mostrado previamente, todos os testes para a dificuldade fácil foram resolvidos com soluções ótimas. Continuaremos com as outras dificuldades, agora:

Figura 8 – Resultados para a dificuldade médio, distribuidos através dos parâmetros do algoritmo

Fonte: Autoria Própria

Para a dificuldade médio encontra-se quatro testes que não encontraram resultados ótimos, todas elas possuem os parâmetros muito parecidos, menor população, menor taxa de mutação e maiores taxas de *crossover* utilizadas nos testes. Agora resta construir o mesmo gráfico para a última dificuldade para que possamos concluir que o problema está na escolha de parâmetros.

Figura 9 – Resultados para a dificuldade difícil, distribuidos através dos parâmetros do algoritmo



Fonte: Autoria Própria

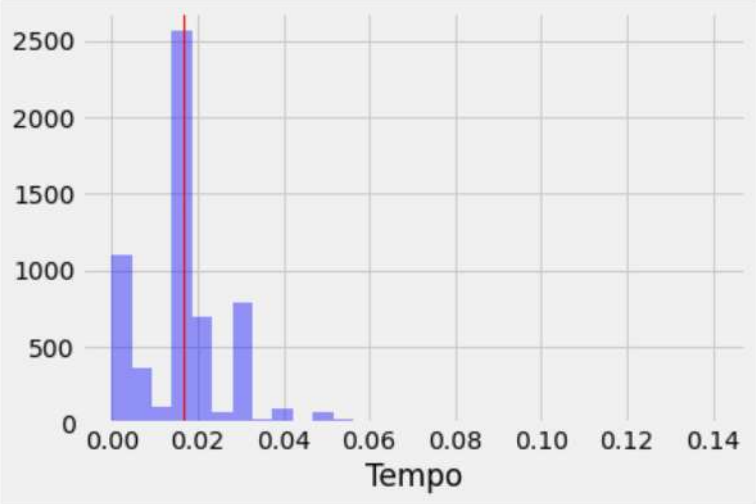
Novamente, assim como para os testes com a dificuldade media, os testes com a dificuldade difícil tiveram um gargalo quando submetidos à alguns parâmetros específicos. Os parâmetros que influenciaram para os testes sem resposta foram os mesmo que influenciaram nas falhas para a dificuldade media: menor população, menor taxa de mutação e maiores taxas de *crossover* utilizadas nos testes. Concluimos que essa combinação pode não ser benéfica para nosso problema. Será explorado melhor esta combinação na próxima seção.

4.3 Análise da eficiência

Agora finalmente podemos analisar a eficiência do algoritmo que, mesmo com bons resultados, pode ter sido baixa.

A eficiência foi intuitivamente diferente para cada dificuldade. Para a dificuldade 'Fácil', em específico, obteve-se de longe a melhor eficiência: os resultados foram explêndidos para os três diferentes problemas com a dificuldade em questão, onde todos os problemas foram resolvidos apenas com a geração inicial. Isso só foi possível através da metodologia implementada para criação da primeira geração (devido a isso, os resultados encontrados sem a utilização do *Hill-Climbing* para a dificuldade fácil, foram os mesmo de quando usado).

Figura 10 – Distribuição do Tempo na dificuldade Fácil



Fonte: Autoria Própria

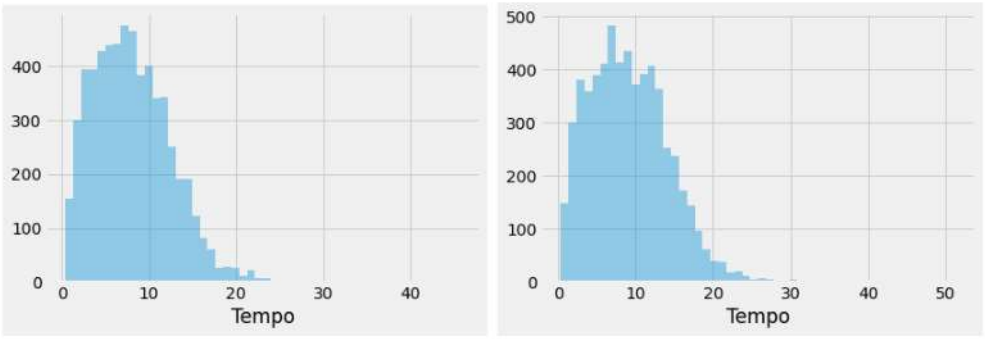
A imagem mostra bem o que foi dito previamente, bastaram um pouco menos de 0.02 segundos, em média (como mostra a linha vermelha), para a solução dos problemas. A distribuição do tempo também não foge muito dessa média.

O trabalho continuará analisando a eficiência para as diferentes dificuldades tendo em vista que os parâmetros do algoritmo não influenciam no tempo para a dificuldade 'Fácil', já que logo na primeira geração o problema é resolvido.

4.3.1 Tempo

O tempo de processamento é um bom indicador para avaliar eficiência de um algoritmo. Nesse sentido, foram avaliados os tempos de execução para cada nível de dificuldade do sudoku. A figura 11 mostra a distribuição de tempo de processamento para cada dificuldade através de um histograma. Nota-se que se trata de uma distribuição assimétrica esquerda para ambas as dificuldades, o que nos força a entender que a média da variável tempo é maior que a mediana e moda. Isso, na maioria das vezes, quer dizer que existem alguns outliers, ou seja, dados com valores absurdos para tempo quando comparado com a maioria dos outros valores.

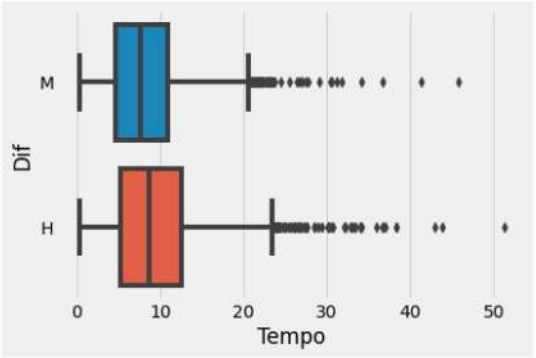
Figura 11 – Distribuição do Tempo para a dificuldade Médio e Difícil



Fonte: Autoria própria

O diagrama de caixas na imagem 12 confirma o ponto pressuposto de que existem outliers. Com isso em mente, o trabalho irá explorar um pouco mais a fundo no motivo desses outliers estarem surgindo.

Figura 12 – Distribuição do Tempo em diagrama de caixa para cada dificuldade



Fonte: Autoria Própria

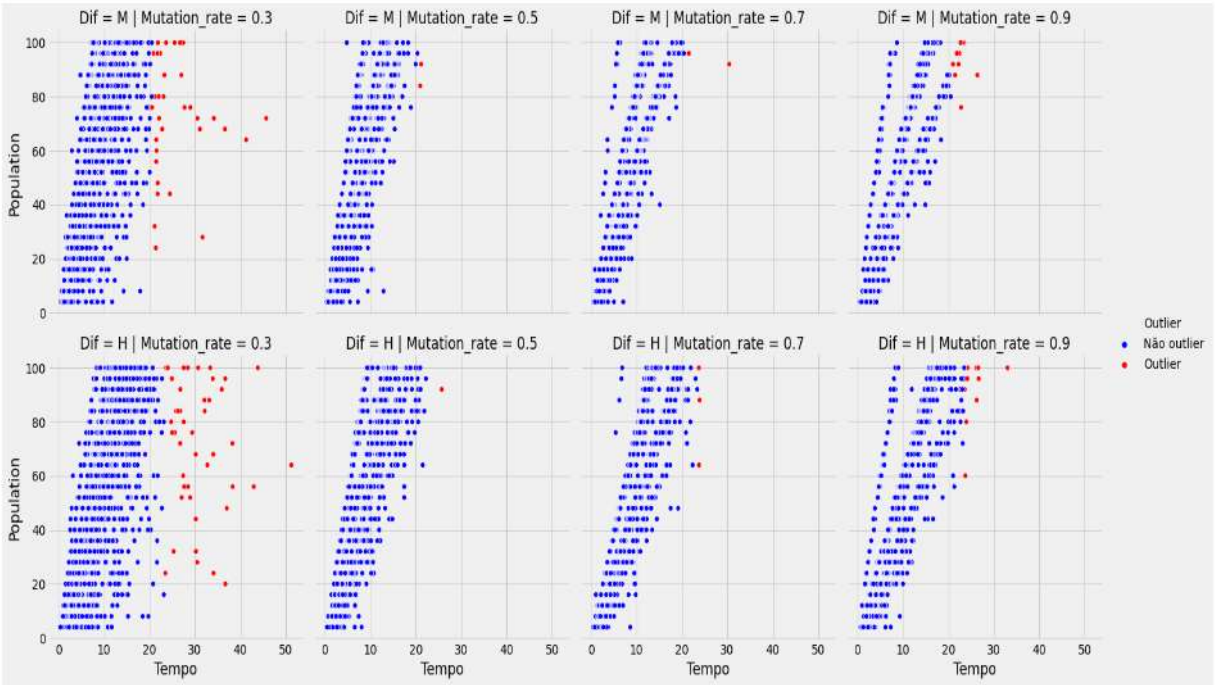
Tabela 3 – Quantidade de outliers

	Médio	Dificil
Quantiade de não outliers	5940	5937
Quantidade de outliers	60	63

Fonte: Autoria Própria

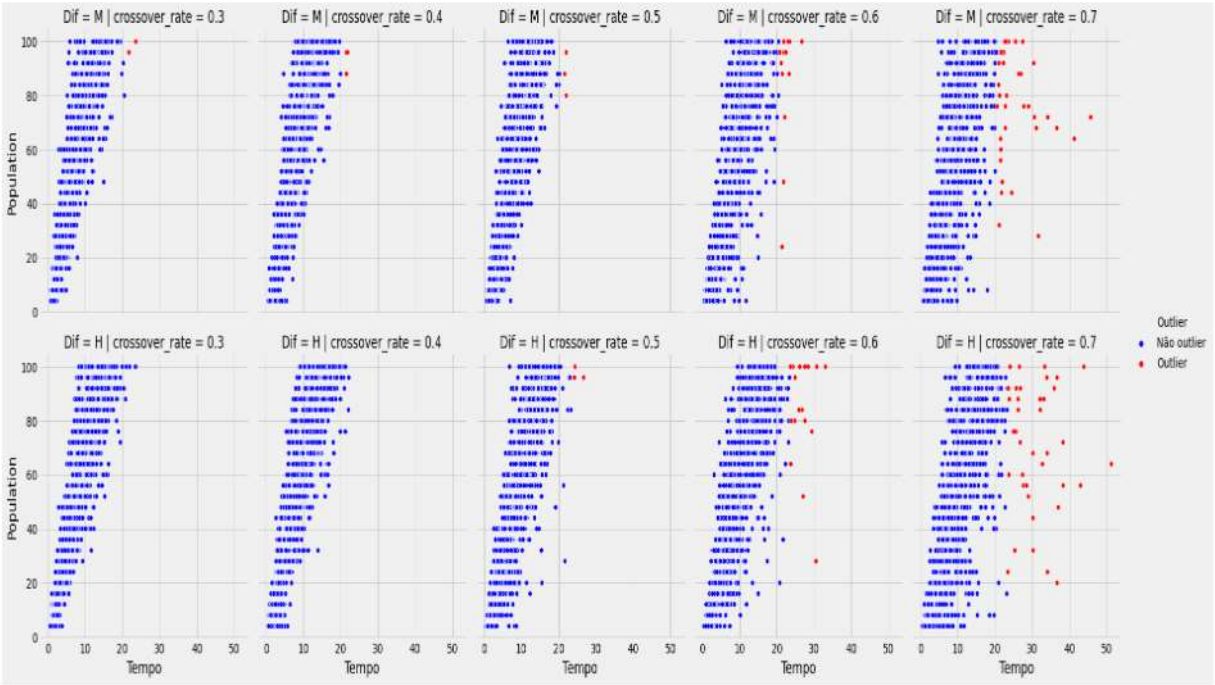
No total são 123 outliers, onde 60 são proveniente da dificuldade médio e 63 da difícil. Com o uso de um diagrama de pontos podemos analisar melhor de onde estão surgindo os outliers:

Figura 13 – Distribuição do Tempo para cada dificuldade variando a mutação



Fonte: Autoria Própria

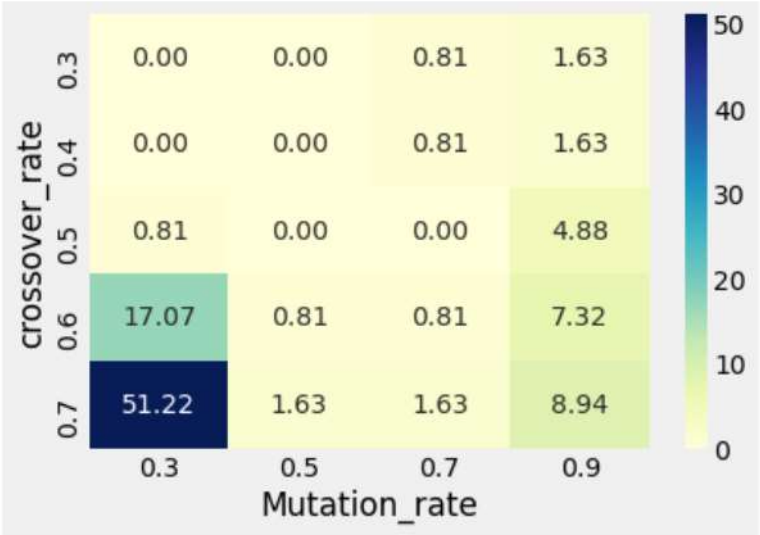
Figura 14 – Distribuição do Tempo para cada dificuldade variando o crossover



Fonte: Autoria Própria

Com estes simples gráficos de pontos, percebe-se que há uma possível correlação entre os parâmetros do algoritmo e os outliers. Eles geralmente ocorrem, para ambas as dificuldades, quando a taxa de mutação é baixa e há alta taxa de *cross-over*.

Figura 15 – Gráfico de calor para quantidade de outliers (%) variando *crossover* e mutação



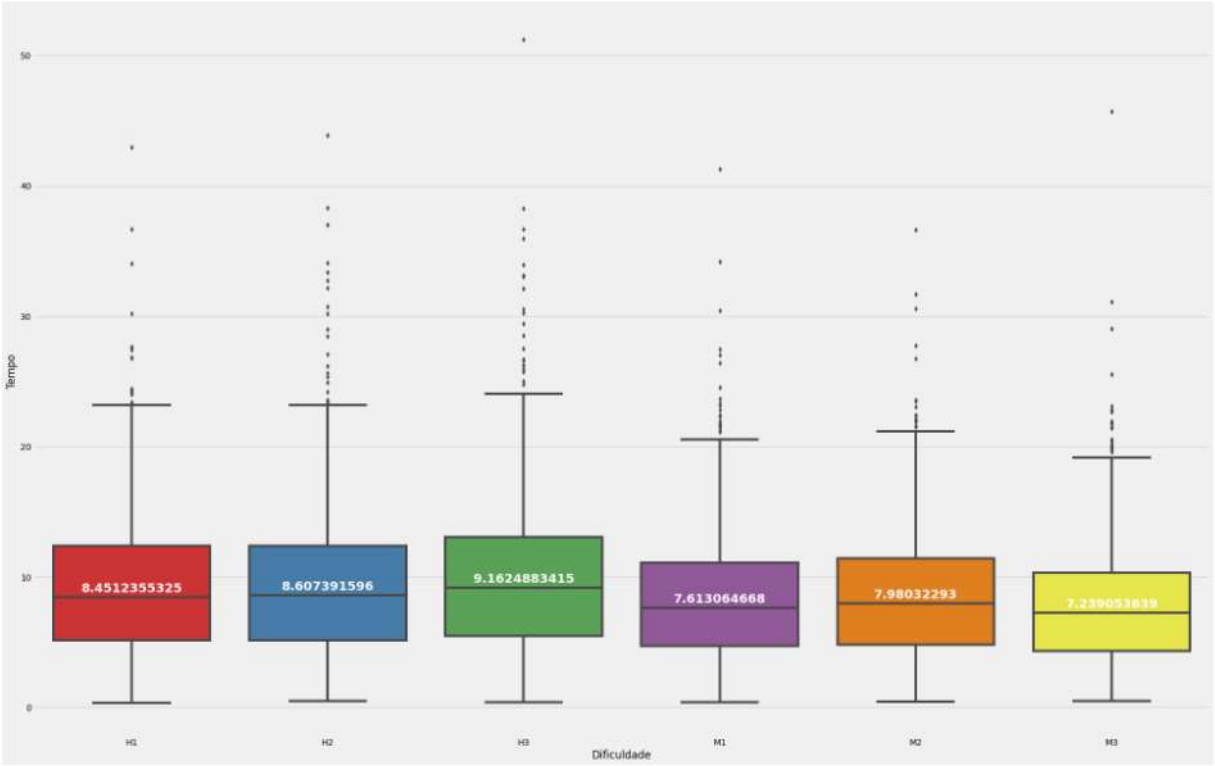
Fonte: Autoria Própria

O gráfico de calor comprova o fato pressuposto, e ainda mostra que quando há alta mutação e alto cross-over, o mesmo fenômeno tem leve tendência a acontecer.

Isso ocorre, muito provavelmente, devido a contribuição para a aleatoriedade que esses parâmetros possuem. Por exemplo, quanto maior a taxa *cross-over*, maior a variação da população. Isso ajuda diretamente na saída de ótimos locais, porém pode influenciar no número de gerações necessárias para a solução ideal, impactando no tempo. A mutação também serve de exemplo para isso. Isso é a explicação, inclusive, do porque não foram encontradas as soluções para os sete testes que falharam.

Agora que melhor explorado o motivo dos outliers estarem surgindo, podemos começar a explorar a distribuição da métrica de tempo correlacionando com os parâmetros. Começando pela distribuição do tempo através de um diagrama de caixa para cada problema:

Figura 16 – Distribuição do Tempo em diagrama de caixa para cada problema



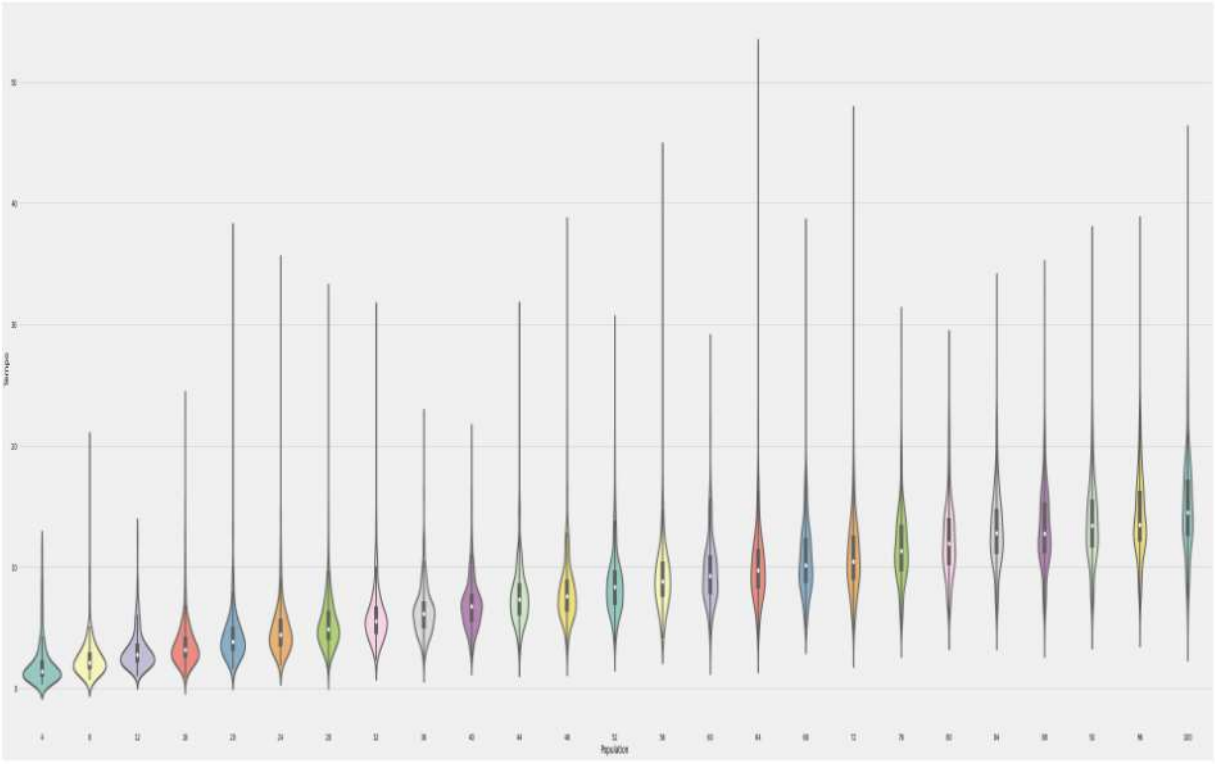
Fonte: Autoria Própria

Podemos ver através da imagem 16 que a distribuição de tempo segue semelhante e quase linear para todos os problemas, não importando a dificuldade. Tendo isso em mente podemos exluir a variável dificuldade como um influenciador no tempo (pelo menos para dificuldades diferentes de fácil).

O próximo passo será o de explorar todos os outros parâmetros, os correlacionando com o indicador de tempo. O primeiro parâmetro a ser explorado será o tamanho da popoulação, para isso será usado um diagrama de violino. O diagrama de violino é muito semelhante ao

diagrama de caixas porém mostra também a densidade dos dados para cada valor no eixo das ordenadas.

Figura 17 – Distribuição do Tempo com aumento da população



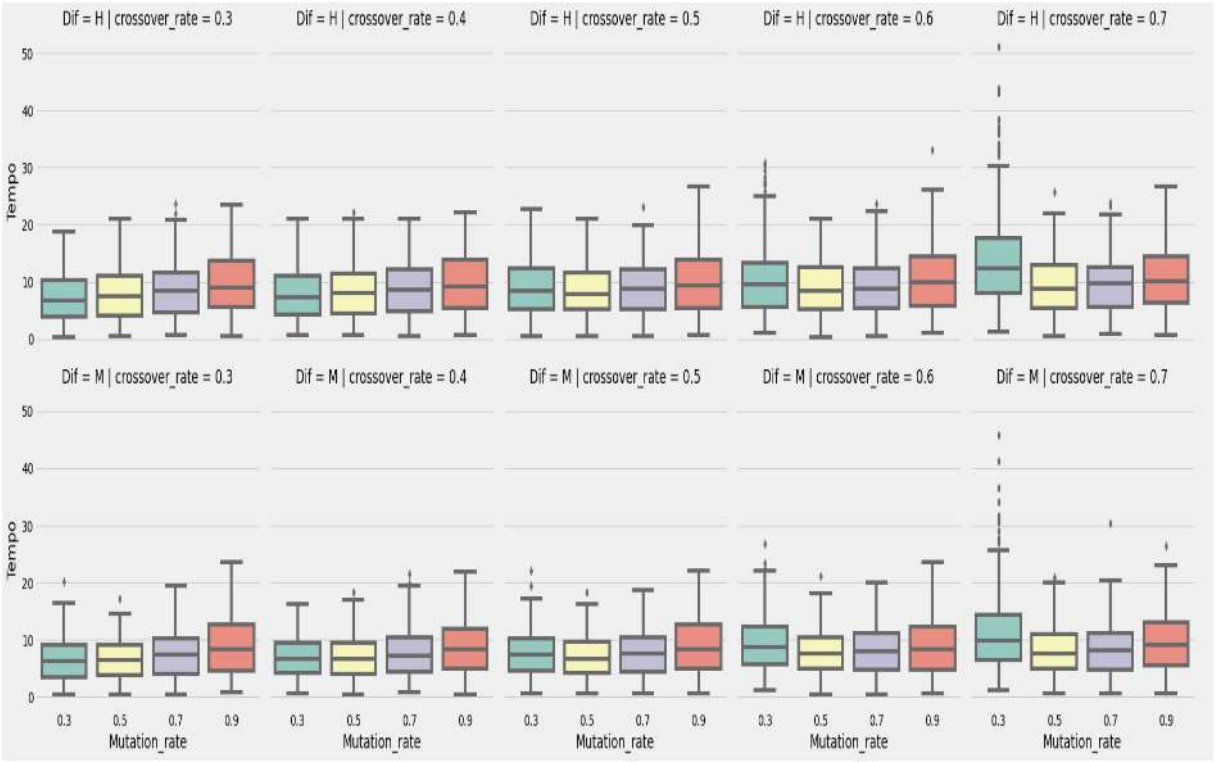
Fonte: Autoria Própria

A figura mostra que conforme a população cresce, a variança de tempo também cresce, além disso, nota-se também que a densidade de dados é sempre mais próximo da mediana em populações menores. Disso podemos concluir que quanto menor a população, melhor o tempo será, na maioria das vezes, quando comparado com populações maiores.

Essa conclusão pode ser bem intuitiva, mas existem mais peças nesse quebra-cabeça que são necessárias explorar, como por exemplo: Todas as execuções que não obtiveram resposta correta, vieram de execuções que eram constituidas da menor população possível. Se o número de gerações limite fosse maior, esse problema não existiria. Podemos, por isso, confirmar que é sempre bom pensarmos em como os parâmetros se interferem.

Agora resta explorar os outros parâmetros para observar se eles são influenciados ou se influênciam nos outros parâmetros e métricas assim como a população atinge diretamente no tempo.

Figura 18 – Variação do tempo para cada combinação de *crossover* e mutação



Fonte: Autoria Própria

Através da figura 18 nota-se que há um padrão para a variança de tempo: conforme é aumentado ou diminuido a taxa de mutação e taxa de *crossover* a variança tende a aumentar ou diminuir. Isso acontece para ambas as dificuldades e não depende da população, portanto, sabe-se que esse fato acontecerá sempre, independentemente da população ou dificuldade. Tendo isso em mente conclue-se que existem combinações específicas de taxa de mutação e taxa de crossover que impactam diretamente no tempo.

Um exemplo disso é o mesmo exemplo que foi identificado quando foi analisado o motivo do surgimentos dos outlier: quando há baixa mutação e um valor alto de taxa de *crossover*, o tempo tenderá a variar muito. Isso acontece devido a variação muito mais aleatória da população, fugindo da implementação do *hill climbing* e tornando a melhoria de *fitness* mais aleatória a cada geração.

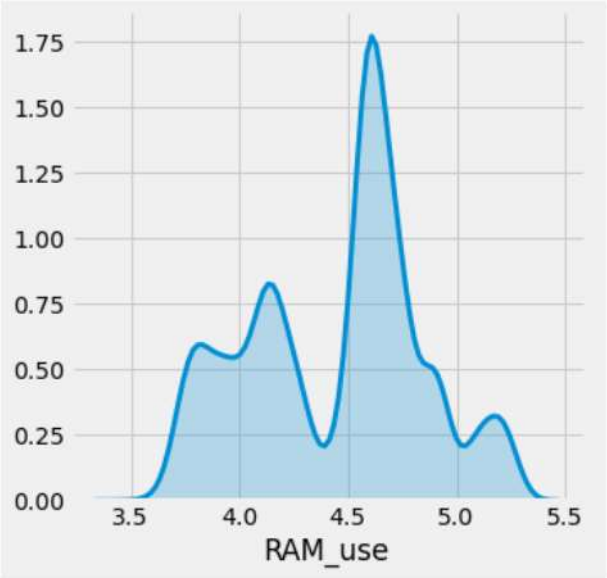
Em suma, o tempo é realmente bastante impactado pela taxa de *crossover*, taxa de mutação e pela população. Caso não seja feito algum estudo prévio, o tempo de processamento pode decolar com uso indevido destes parâmetros: no caso de estudo do trabalho seria melhor usar a menor população possível devido ao uso do *hill climbing*, mas com cuidado para o número limite de gerações não barrar a solução. Além disso, seria melhor usar um valor mediano para taxa de *crossover* e taxa de mutação, tendo em vista que eles tendem a variar muito quando estão altos e variam de dificuldade para dificuldade quando estão muito baixos.

4.3.2 Uso de RAM

Outra métrica de eficiência de um algoritmo é o uso de memória principal, também conhecida como memória RAM. O motivo desta métrica ser usada é simples e intuitivo, algoritmos que utilizam menor RAM podem ser utilizados em diversos computadores com hardwares inferiores, barateando a operação.

A análise se iniciará da mesma forma como fora feito com o tempo, através da distribuição dos valores:

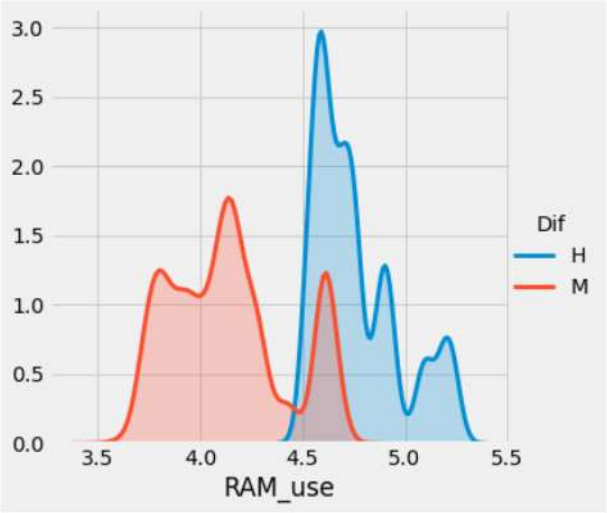
Figura 19 – Distribuição do uso de RAM



Fonte: Autoria Própria

Percebe-se, através da imagem 19, que a distribuição do uso de RAM é bimodal. Em outras palavras, é uma distribuição de probabilidade contínua com duas modas diferentes, aproximadamente em 4,55 e em 4,20. Isso pode estar acontecendo por estarmos explorando os resultados de duas dificuldades diferentes.

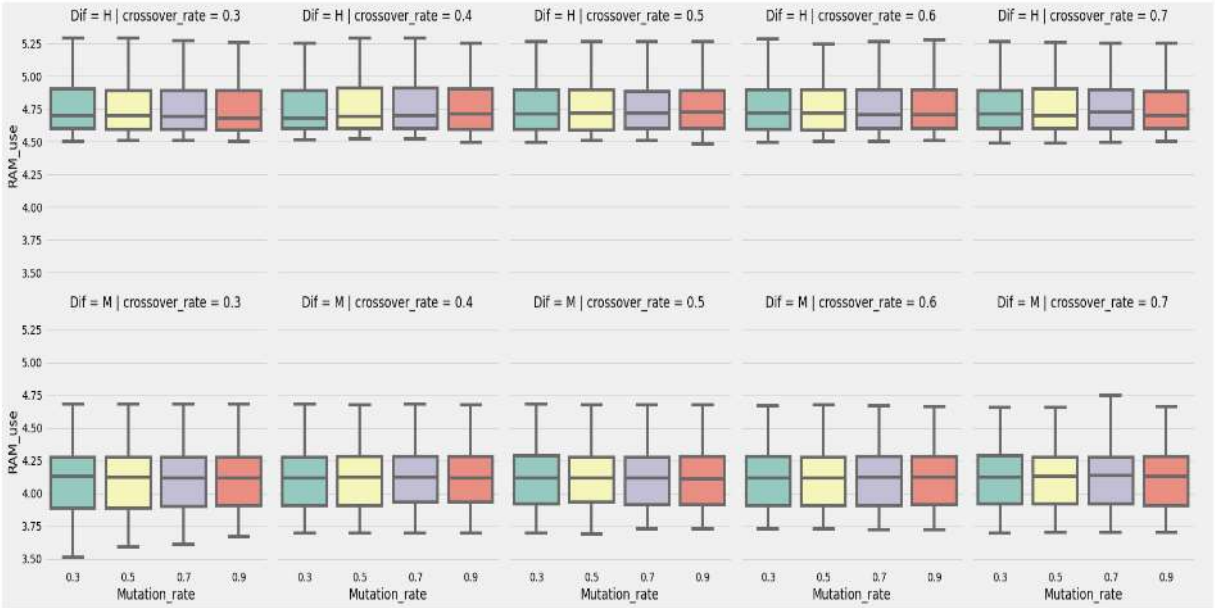
Figura 20 – Distribuição do uso de RAM diferenciado pela dificuldade



Fonte: Autoria Própria

Como pensado anteriormente, a RAM é realmente interferida pela dificuldade do jogo. Pode se observar melhor essa correlação de dificuldade e uso de RAM através da imagem:

Figura 21 – Diagrama de caixas de uso de RAM para cada problema solucionado



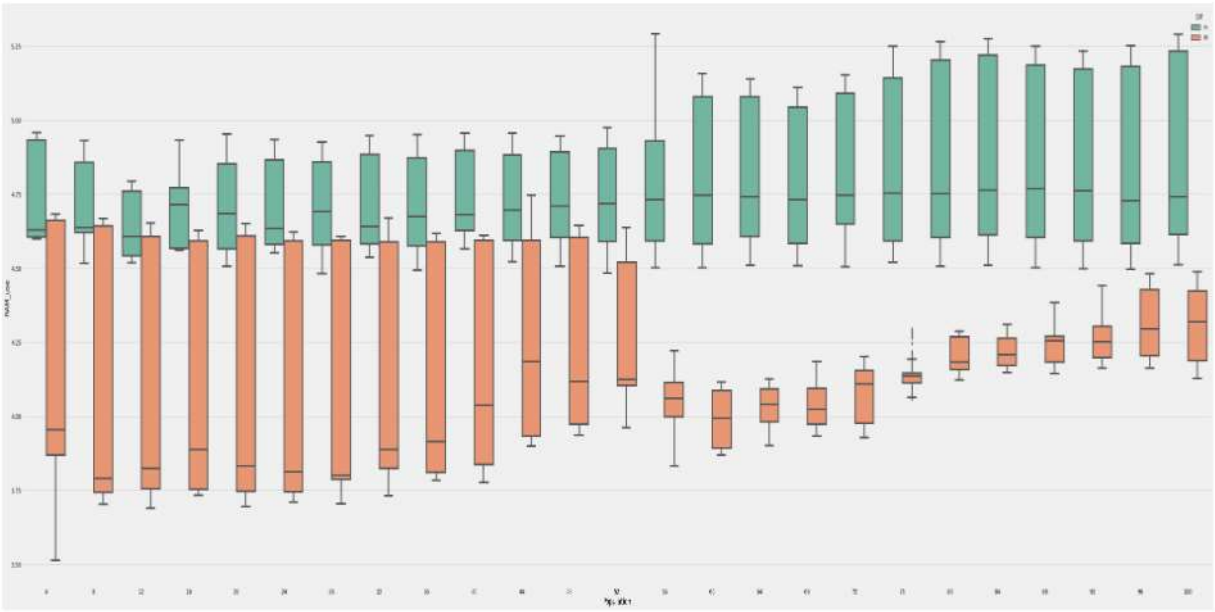
Fonte: Autoria Própria

Nota-se que realmente a RAM é influenciada pela dificuldade do jogo. É discrepante a diferença de consumo de RAM das dificuldades mais fáceis para as mais difíceis. Tendo em mente que todos os testes foram realizados com o mesmo ambiente, logo não há variações externas de RAM, a explicação para isso seria de que, em dificuldades mais fáceis, são necessárias menos gerações para a solução, logo menos cromossomos seriam criados e menos RAM seria utilizada.

Outro ponto que observa-se com a imagem 17 é de que a taxa de *crossover* e taxa de mutação não impactam no consumo de RAM tendo em vista que todos as caixas seguem paralelos ao eixo X e de que não há distorção nas medianas.

Resta agora explorar a variança da memória principal de acordo com o crescimento da população:

Figura 22 – Diagrama de caixas de uso de RAM para cada população diferenciado pela dificuldade



Fonte: Autoria Própria

Por último, a figura 22 mostra que o uso de RAM varia, levemente, conforme a população cresce. Isso não é algo repentino, já era de se esperar que isso aconteceria partir do momento que cada indivíduo na população possui 81 dígitos no formato *INT*, totalizando 324 bytes por indivíduo. Assim, quanto maior a população, maior número de indivíduos, e maior o consumo de RAM.

Em resumo, conclui-se que a RAM é principalmente impactada pela dificuldade, ou seja, o número de possíveis combinações impacta diretamente o consumo de memória principal. Além disso,o número de interações que a taxa de mutação e o *crossover* causam, não impactam o consumo de RAM, já a população sim.

5 CONCLUSÃO

O desenvolvimento do estudo possibilitou uma análise de como um algoritmo genético se comporta perante um problema combinatório, em específico, um Sudoku. A conclusão que pode-se tirar desta análise é de que um algoritmo genético consegue sim resolver o problema combinatório em questão, porém, não é o caminho mais eficiente para alcançar tal objetivo.

A implementação é uma proposta muito interessante, porém não é muito simplória e demanda muito tempo. Além disso, utilizar o algoritmo genético como única meta-heurística para o algoritmo não gerou bons resultados: o tempo demandado para solução é muito grande e varia muito. Para diminuir o tempo de execução houve a implementação do algoritmo de otimização *hill-climbing* no processo de mutação, o que adicionou uma nova meta-heurística ao trabalho.

Esses problemas são pertinentes porém o maior problema consiste na engenharia por detrás do algoritmo. Existem infinitas possíveis combinações de parâmetros para o algoritmo. Descobrir qual é a combinação mais eficiente nem sempre é uma tarefa fácil, e essa combinação pode variar de problema a problema ou dificuldade a dificuldade. Em outras palavras, para o algoritmo ser mais eficiente sempre deverá haver um estudo durante a implementação para especificar os parâmetros do algoritmo.

Além do processo de *tunning* dos parâmetros, existem também diversas outras metodologias que poderiam ser adotadas para equação fitness, ou para o *crossover*, ou até mesmo outros algoritmos de otimização dentro da mutação, o que torna o trabalho bastante complexo e desafiador.

5.1 TRABALHOS FUTUROS

Outros trabalhos poderiam ser realizados a fim de complementar este: trabalhos que exploram outras metaheurísticas e/ou operadores genéticos. Além disso, outra abordagem que pode ser investigada é a utilização de diferentes técnicas de busca local para auxiliar na etapa de mutação, contribuindo para o processo de *exploitation* do algoritmo.

Referências

AKEMI, P. **Introdução a algorítmos genéticos**. [S.l.], 2018. Disponível em: <<https://www.ime.usp.br/~gold/cursos/2009/mac5758/PatriciaGenetico.pdf>>. Citado 3 vezes nas páginas 1, 4 e 5.

BLUM, C. A. R. **Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison, vol. 3.** [S.l.], 2003. Citado na página 4.

COUTINHO, D. **Complexidade de Algoritmos**. [S.l.], 2017. Disponível em: <https://docente.ifrn.edu.br/demetrioscoutinho/disciplinas/algoritmos/03-complexidade>. Citado na página 2.

EIBEN A. E.; SMITH, J. E. **Introduction to evolutionary computing.** [S.l.], 2003. Citado na página 1.

GASPAR-CUNHA, A.; TAKAHASHI, R. **Manual de computação evolutiva e metaheurística, vol. 1.** [S.l.], 2012. Citado na página 4.

HOLLAND, J. **Adaptation in Natural and Artificial Systems**. [S.l.], 1975. Citado na página 2.

LAWLER, E.; RINNOOY, K. **The Traveling Salesman Problem: A guided Tour of Combinatorial Optimization.** [S.l.], 1985. Citado na página 3.

MANTERE, T.; KOLJONEN, J. **Solving and Rating Sudoku Puzzles with Genetic Algorithms**. [S.l.], 2006. Citado 2 vezes nas páginas 4 e 5.

MCCAFFREY, J. D. **Roulette Wheel Selection Algorithm**. [S.l.], 2017. Disponível em: <https://jamesmccaffrey.wordpress.com/2017/12/01/roulette-wheel-selectionalgorithm/>. Citado na página 5.

PERCÍLIA, E. **Sudoku**. [S.l.], 2019. Disponível em: <https://brasilescola.uol.com.br/curiosidades/sudoku.htm>. Citado na página 1.

SCHIFF, K. **An Ant Algorithm for the Sudoku Problem.** [S.l.], 2015. Citado na página 3.