UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

Uso de Busca Local no Desafio Regex Golf

André de Almeida Farzat

Orientador

Prof. Márcio de Oliveira Barros

RIO DE JANEIRO, RJ - BRASIL
SETEMBRO DE 2019

USO DE BUSCA LOCAL NO DESAFIO REGEX GOLF

ANDRÉ DE ALMEIDA FARZAT

DISSERTAÇÃO APRESENTADA COMO REQUISITO PARCIAL PARA OBTENÇÃO DO TÍTULO DE MESTRE PELO PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA DA UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO (UNIRIO). APROVADA PELA COMISSÃO EXAMINADORA ABAIXO ASSINADA.

Aprovada por:

Márcio de Oliveira Barros, D.Sc. — UNIRIO

Adriana Cesário de Faria Alvim, D.Sc. — UNIRIO

Toacy Cavalcante de Oliveira, D.Sc. — UNIRIO

RIO DE JANEIRO, RJ - BRASIL
SETEMBRO DE 2019

Catalogação informatizada pelo(a) autor(a)

F219

Farzat, André de Almeida
Uso de busca local no desafio regex golf / André de Almeida Farzat. -- Rio de Janeiro, 2019.
66

Orientador: Márcio de Oliveira Barros.
Dissertação (Mestrado) - Universidade Federal do Estado do Rio de Janeiro, Programa de Pós-Graduação em Informática, 2019.

1. Expressões regulares. 2. Regex Golf. 3. Code Golf. 4. Busca local. 5. Busca heurística. I. Barros, Márcio de Oliveira, orient. II. Título.

Agradecimentos

A Deus, por me prover as oportunidades e forças para enfrentar todos os desafios que a vida me trouxe.

Ao meu orientador, Prof. Márcio Barros, por ter acreditado em minha capacidade e me guiado em todos dos momentos difíceis durante a jornada que foi o mestrado.

A minha mãe Aleuda, por todo o seu amor e por ser minha inspiração de vida. Por ter me mostrado que no final tudo vale a pena sim.

Ao meu irmão mais velho, Fabio Farzat, minha fonte de inspiração e meu maior exemplo de como ser uma pessoa melhor.

Aos meus grandes amigos e sócios Luiz Glaucio e Pedro Monte Mor, graças ao suporte de vocês pude concentrar meus esforços em terminar este trabalho.

À Coordenação de Aperfeiçoamento de Pessoal de Nível Superior pela bolsa de estudos do Mestrado.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001

Farzat, André de Almeida. **Uso de Busca Local no desafio Regex Golf**. UNIRIO, 2019. 66 páginas. Dissertação de Mestrado. Departamento de Informática Aplicada, UNIRIO.

RESUMO

Expressão regular é uma tecnologia bastante utilizada no desenvolvimento de software para extração de dados de documentos textuais, validação da estrutura de documentos textuais ou mesmo formatação destes dados. Porém, existem poucos trabalhos com o intuito de reduzir o esforço manual para se criar uma expressão regular eficiente e com qualidade. Regex Golf é um desafio que consiste em encontrar a menor expressão regular possível dado um conjunto de frases para realizar *matches* e outro conjunto para não realizar *matches*. Um algoritmo capaz de vencer o desafio Regex Golf é uma contribuição relevante para a área de extração de dados de documentos semi-estruturados utilizando expressões regulares. Regex Golf é considerado um problema NP-completo e heurísticas são métodos bem aceitos para a resolução de problemas de análise combinatória com esse nível de complexidade. O objetivo deste trabalho é propor uma nova técnica heurística baseada em busca local, em conjunto com um algoritmo compactador de expressões regulares, para encontrar resultados válidos para instâncias do desafio Regex Golf. Um estudo experimental foi conduzido para avaliar e comparar a técnica proposta com um algoritmo exato e um algoritmo de programação genética criados para para o desafio Regex Golf.

Palavras-chave: Expressões Regulares, Regex Golf, Code Golf, Busca Local, Busca heurística.

ABSTRACT

Regular expression is a technology widely used in software development for extracting textual data, validating textual structure or formatting data. However, there has been little work on reducing the manual effort involved in generating efficient and high-quality regular expressions for information extraction tasks. Regex Golf is a challenge that consists of finding the smallest possible regular expression given a set of phrases to perform matches and another set to perform unmatches. An algorithm capable of meeting the Regex Golf requirements is a relevant contribution to the area of semi-structured document data extraction using regular expressions. Regex Golf is considered an NP-complete problem and heuristics are well accepted methods for solving combinatorial analysis problems of this level of complexity. In this paper, we propose a new heuristic technique based on local search, combined with a regular expression compressor algorith, to find valid results for the Regex Golf challenge instances. An experimental assessment was conducted to evaluate and compare the proposed technique with an exact algorithm and a genetic programming algorithm designed for the Regex Golf challenge.

Keywords: Regex, Regex Golf, Code Golf, Regular Expressions, Local Search, Heuristics.

Sumário

1 Introdução **1**

1.1 Expressões regulares 1

1.2 Regex golf 4

1.3 Problema 5

1.3.1 Questão de pesquisa 6

1.4 Objetivos da pesquisa 6

1.5 Metodologia da pesquisa 7

1.6 Estrutura da dissertação 8

2 Estudos relacionados **9**

2.1 Introdução 9

2.2 Algoritmos exatos 11

2.3 Algoritmos heurísticos 14

2.3.1 Busca Local 16

2.3.2 Algoritmos Genéticos 18

2.3.3 Programação Genética 20

2.3.4 Evolução Gramatical 23

2.4 Aprendizado de máquina 26

2.5 Considerações finais 30

3 Algoritmo de busca local para construir expressões regulares 31

3.1 Introdução 31

3.2 Representação da solução do problema 32

3.3 Função objetivo 33

3.4 Vizinhança 34

3.5 Geração aleatória de uma solução 39

3.6 Compactador 39

3.7 Considerações finais 41

4 Avaliação do algoritmo de busca local proposto 43

4.1 Introdução 43

4.2 Projeto do estudo experimental 43

4.2.1 Instâncias selecionadas 43

4.2.2 Ambiente computacional 44

4.3 Parametrização do algoritmo 44

4.3.1 Profundidade da árvore 45

4.3.2 Critério de parada 48

4.4 Comparação entre os resultados 49

4.4.1 Comparação com o algoritmo de programação genética 53

4.4.2 Comparação com o algoritmo exato 54

4.4.3 Comparação com expressões regulares desenvolvidas por humanos 55

4.5 Problemas encontrados 56

4.6 Considerações finais 57

5 Conclusão **58**

5.1 Considerações finais 58

5.2 Contribuições 59

5.3 Resposta para a questão de pesquisa 59

5.4 Limitações 60

5.5 Perspectivas futuras de trabalho 60

Referências Bibliográficas **61**

Lista de Figuras

2.1 Etapas efetuadas na nossa busca na literatura 10

3.1 Fluxograma da busca local 31

3.2 Representação em árvore sintática da expressão regular $\hat{}ab[c]$ 32

4.1 Bloxplot comparando os resultados do algoritmo de busca loca, do algoritmo de programação genética e do algoritmo exato 52

Lista de Tabelas

1.1 Lista com as frases de exemplo de instância do problema 4

2.1 Artigos que tratam da geração automática de expressões regulares, encontrados na busca na literatura 10

3.1 Nós função e suas representações 33

3.2 *match char list*: Lista com os caracteres extraídos e ordenados da *match list* 35

3.3 *unmatch char list*: Lista com os caracteres extraídos e ordenados *unmatch list* 35

3.4 *inclusive char list*: Lista com os caracteres da *match char list* que não pertencem a *unmatch char list* 35

3.5 *exclusive char list*: Lista com os caracteres da *unmatch char list* que não pertencem a *match char list* 35

4.1 Instâncias selecionadas para o estudo experimental 44

4.2 Médias das pontuações das instâncias de acordo com o parâmetro de profundidade da árvore 47

4.3 Resultado médio do número de avaliações necessárias até encontrar a melhor solução e pontuação da melhor solução encontrada 48

4.4 As melhores expressões regulares resultantes das execuções do algoritmo de busca local para cada instância 50

4.5 Os resultados do algoritmo exato (Norvig-exato), do algoritmo de programação genética (Bartoli-PG) e de 30 execuções do algoritmo de busca local 51

4.6 As melhores expressões regulares desenvolvidas por humanos 53

1. Introdução

1.1 Expressões regulares

Uma expressão regular (*regex*) é uma sequência de caracteres (*string*) que define um padrão de busca. Normalmente esse padrão é usado para realizar buscas em *strings* com o intuito de encontrar ou substituir partes de um texto. Jeffrey Friedl [14] dá um exemplo prático para o uso de expressões regulares: imagine que você precise analisar uma grande quantidade de arquivos para confirmar se todos contém o texto "SetSize" exatamente na mesma quantidade que o texto contém a palavra "ResetSize". E, como agravante, é necessário tratar caixa alta e caixa baixa da mesma forma (ou seja, "setSIZE" deve ser contado como "SetSize"). Mesmo utilizando uma ferramenta de "encontrar por palavra-chave" em um editor de texto, esta tarefa exige um trabalho árduo se feito manualmente. Por outro lado, com apenas um comando baseado em expressões regulares é possível verificar todos os arquivos e confirmar a contagem desejada.

Uma expressão regular é composta por uma sequência de operadores, muitos dos quais recebendo parâmetros na forma de outros operadores ou caracteres literais. Os operadores e seus parâmetros são representados na forma de *strings* usando uma sintaxe específica. Uma expressão regular é utilizada para fazer buscas em documentos de texto e seus operadores servem para designar variantes dos termos especificados por seus literais. Quando a sequência representada em uma expressão regular é encontrada em um texto, dizemos que ocorreu um *match*. A existência ou contagem de *matches* é útil para diversas tarefas, incluindo a captura de dados do documento texto e a verificação da sua estrutura.

O termo expressão regular é derivado do trabalho do matemático Stephen Cole Kleene [18], que desenvolveu as expressões regulares como uma notação ao que ele chamava de álgebra de conjuntos regulares. Seu trabalho serviu de base para os primeiros algoritmos computacionais de busca e para algumas das mais antigas ferramentas de tratamento de texto da plataforma Unix. Hoje em dia, as expressões regulares também são utilizadas

por ferramentas de busca e substituição em editores de texto, ferramentas de busca em arquivos (como *awk* e *grep*) e ferramentas de substituição de texto (como *sed*) [41].

Muitas linguagens de programação oferecem recursos para o uso de expressões regulares diretamente em sua máquina virtual ou via biblioteca. Cada linguagem ou ferramenta tem sua própria sintaxe e conjunto de operadores para as expressões regulares, apresentando pequenas variações entre elas. Dentre os padrões existentes, *Perl Compatible Regular Expressions*[1] (PCRE) e *POSIX extended regular expressions* (POSIX ERE) são os mais comumente utilizados por linguagens de programação [14]. Nesta dissertação utilizaremos a sintaxe e o conjunto de operadores definidos pelo padrão ECMAscript[2], que é baseado no padrão POSIX ERE. Este padrão compreende os seguintes operadores:

- *Character class*: Realiza *match* apenas nos caracteres apresentados como parâmetro do operador e na ordem especificada. Ou seja, o operador *character class* com o valor `ab` realizará *match* em "abcedf", mas não realizará em "uvwxyz". Exemplos: `a`, `b`, `cd` e `04`;

- *Character set*: Este operador tem como valor um conjunto de caracteres. Realiza *match* com qualquer caractere que pertença ao conjunto. O conjunto de caracteres pode ser representado por um intervalo, como o valor `[0-9]`, que realizará *match* em qualquer dígito. Alternativamente, qualquer conjunto de caracteres apresentado entre colchetes será considerado um conjunto válido para *match*. Exemplos: `[a9z]`, `[a-z]` e `[A-Z0-9]`;

- *Negative character set*: Este operador é o inverso do *character set*. Não realiza *matches* com os caracteres que pertençam ao conjunto. Por exemplo a expressão regular `[^abc]` não realiza *match* com os caracteres "a", "b" ou "c", enquanto a expressão regular `[^5-9]` não realiza *match* com dígitos entre 5 e 9;

- *Shorthand Character Class*: Como certos *character set* são usados com frequência, algumas abreviações estão disponíveis. Entre os casos mais conhecidos, `\d` é abreviação de `[0-9]` e `\w` é abreviação de `[a-zA-Z0-9_]`;

- *Dot*: O operador *dot* (`.`) realiza *match* em qualquer caractere que não seja quebra de linha (`\n`);

- *Start anchor* e *End anchor*: Estes operadores realizam *match* no início (`^`) e fim (`$`) de linha, respectivamente;

[1] http://www.pcre.org/

[2] https://www.ecma-international.org/ecma-262/9.0/index.htm

- *Word boundaries*: O operador `\b` marca o início ou o fim de uma palavra. Por exemplo, a expressão regular `\babc` realiza *match* na *string* "xyz abc", mas não realiza *match* na *string* "xyzabc";

- *Alternation*: O operador `|` realiza *match* em um dos operadores que estejam antes ou depois do mesmo. Por exemplo, a expressão regular `a|\d` é similar a `[a0-9]`;

- *Optional*: O operador `?` torna opcional o operador precedente, permitindo que o *match* com este operador aconteça ou não. Por exemplo, a expressão regular `ab\d?d` realiza *matches* em ambas *strings* "abd" e "ab4d";

- *Quantifier*: O operador `*` torna o operador anterior uma repetição, ou seja, determina que o operador anterior precisa realizar zero ou mais *matches* . O operador `+` determina que o operador anterior precisa realizar um ou mais *matches* . Quando a sintaxe for `{n}`, a repetição determina que o operador anterior precisa realizar `n` *matches* . Finalmente, quando a sintaxe for `{n,m}`, a repetição determina que o operador anterior precisa realizar entre `n` e `m` *matches*;

- *Possessive quantifier*: Realiza *matches* similar ao operador *quantifier*, porém utilizando uma estratégia preguiçosa (*lazy*) na repetição. Por exemplo, a expressão `.*` realizará *match* na *string* inteira, não deixando nenhum caractere para realizar *match* com outro operador da expressão regular. Uma vez que o *possessive quantifier* não realiza *backtracking* (ou seja, ele não volta nos caracteres para realizar outros testes para o *match* em conjunto de outros operadores), a realização do *match* de uma expressão com outros operadores falhará;

- *Group*: Este operador agrupa operadores, permitindo aplicar outros operadores ao grupo. Por exemplo, é possível aplicar o operador *quantifier* em um grupo de *character class* (como em `(abc){2,3}`, que realiza *matches* nas *strings* "abcabc" e "abcabcabc") ou o operador *alternation* a uma parte da expressão regular (como em `(abc)|(def)`, que realiza *matches* em ambas *strings* "abc" e "def");

- *Backreference*: O operador `\1` realiza *match* utilizando o mesmo texto do primeiro operador *group* que realizou *match* anteriormente. Por exemplo, a expressão regular `(abc)def\1` realiza *match* na *string* "abcdefabc";

- *Lookbehind*: Este operador funciona em conjunto ao próximo operador, que realizará o *match* somente se este operador também realizar na *string* anterior. Por exemplo, a expressão regular `(?<=tel:)\d+` realiza *match* na *string* "tel:123" e não realiza *match* na *string* "cell:123";

- *Negative lookbehind*: Apresenta a função contrária do operador *lookbehind*, ou seja, realiza o *match* em conjunto com o próximo operador somente se este operador não realizar na *string* anterior. Por exemplo, a expressão regular `(?<!tel:)\d+` realiza *match* na *string* "cell:123" e não realiza *match* na *string* "tel:123";
- *Lookahead*: Similar ao operador *Lookbehind*, este operador funciona em conjunto ao operador anterior, que realizará o *match* somente se este operador também realizar na *string* seguinte. Por exemplo, a expressão regular `\d(?=st|nd|rd|th)` realiza *match* nas *strings* "1st", "2nd", "3rd" e "4th";
- *Negative lookahead*: Apresenta a função contrária do operador *lookahead*, realiza o *match* em conjunto ao operador anterior somente se este operador não realizar na *string* seguinte. Por exemplo, a expressão regular `\d(?!=st|nd|rd|th)` realiza *match* nas *strings* "1", "2" e "3", mas não nas *strings* "1st", "2nd" e "3rd".

1.2 Regex golf

Regex Golf é um tipo específico de desafio conhecido como *Code Golf*: uma competição na qual o objetivo é resolver um problema escrevendo o menor código possível. O desafio *Regex Golf* consiste em criar a menor expressão regular que faça *match* em todas as frases de uma lista (*match list*) e **não** faça *match* em nenhuma frase de uma segunda lista (*unmatch list*) [4]. Na Tabela 1.1 temos um exemplo de instância do problema: uma *match list* com os nomes de todos os livros da série "Harry Potter" e uma *unmatch list* com os nomes dos livros da série "O guia do mochileiro da galáxia", ambas em inglês.

Match list	***Unmatch list***
the philosopher's stone	the hitchhiker's guide to the galaxy
the chamber of secrets	the restaurant at the end of the universe
the prisoner of azkaban	life, the universe and everything
the goblet of fire	so long, and thanks for all the fish
the order of the phoenix	mostly harmless
the half-blood prince	and another thing
the deathly hallows	

Tabela 1.1: Lista com as frases de exemplo de instância do problema

Uma solução dessa instância do desafio é a expressão regular `^the\s[^r][^i]`, na qual `^` é o marcador de início de linha, usado para determinar que a expressão regular

precisa procurar pela *string* que comece no início da linha, `the` simboliza a *string* literal, `\s` marca um espaço em branco, `[^r]` indica que o caractere seguinte não pode ser a letra `r` e `[^i]` indica que o próximo caractere não pode ser a letra `i`. Esta instância do problema será utilizada nos exemplos das próximas seções.

Uma forma prática de gerar sistematicamente uma expressão regular capaz de cumprir as exigências do desafio consiste em construir uma expressão regular com todas as frases da lista com que se deseja ter os *matches* separadas pelo operador *alternation* (caractere `|`). Porém, a ideia do desafio é gerar a **menor** expressão regular possível. Então, quanto menor a solução, melhor será a sua pontuação no desafio. É esperado que a expressão regular seja compatível com a máquina virtual de alguma linguagem de programação, como Javascript, Java, PHP, Python, entre outras.

O desafio *Regex Golf* foi idealizado por Peter Norvig em 2014 [3], quando o próprio apresentou uma solução para o desafio na forma de um algoritmo exato. Esta solução será discutida no Capítulo 2.

1.3 Problema

Expressões regulares são uma tecnologia bastante utilizada no desenvolvimento de software para extração de dados de documentos textuais, validação da estrutura de documentos textuais ou mesmo formatação destes dados. Apesar de suas diversas aplicações, existem poucos trabalhos com o intuito de reduzir o esforço manual para se criar uma expressão regular eficiente e eficaz [24].

As poucas propostas para a criação automática de expressões regulares geralmente criam expressões a partir de listas de exemplos de informações que desejam encontrar. Porém, estas estratégias tendem a gerar expressões regulares muito gerais (por exemplo, `.*`) pela falta de restrições e exemplos de trechos de texto que não podem ser aceitos pela expressão regular que está sendo construída.

Peter Norvig aponta que desafios como *Regex Golf* são problemas NP-completos por sua complexidade e dificuldade de resolução. Acreditamos que um algoritmo capaz de vencer o desafio *Regex Golf* é uma contribuição relevante para a área de extração de textos de documentos semi-estruturados utilizando expressões regulares. A intenção é usar trechos de texto de outras partes do documento como contra-exemplo para criar a expressão

[3] http://nbviewer.jupyter.org/url/norvig.com/ipython/xkcd1313.ipynb

regular, formando uma *unmatch list*. Por fim, a redução do tamanho da expressão regular tem como vantagem torná-la mais simples de processar e entender por um desenvolvedor.

1.3.1 Questão de pesquisa

Para este trabalho, foi definida a seguinte questão de pesquisa que estabelece o escopo do estudo experimental.

RQ1: O algoritmo de busca local proposto encontra melhores resultados em menor tempo de execução quando comparado a um algoritmo genético e um algoritmo exato?

A expectativa é que o algoritmo de busca local encontre resultados melhores ou iguais aos encontrados pelo algoritmo genético e pelo algoritmo exato consumindo menos tempo de execução do que estes algoritmos.

1.4 Objetivos da pesquisa

Este trabalho de pesquisa tem como objetivo investigar e desenvolver uma técnica heurística para a geração automática de expressões regulares para um problema proposto no desafio *Regex Golf*. Esta técnica será representada por uma heurística de busca local atrelada a um algoritmo compactador de expressões regulares. A busca local terá como objetivo explorar o espaço de soluções, buscando expressões regulares que façam *match* na *match list* e não façam *match* na *unmatch list*, enquanto o compactador de expressões regulares terá a finalidade de reduzir o número de caracteres de uma expressão regular sem alterar o número de *matches* (ou a falta de *matches*) nas listas.

A intenção é comparar a técnica proposta com outros algoritmos heurísticos utilizados para a geração de expressões regulares, em especial os algoritmos genéticos. A comparação levará em consideração tanto a capacidade de encontrar boas soluções, quanto o tempo de processamento necessário para tal. A busca local é um algoritmo heurístico simples e que tipicamente consome menos tempo de processamento do que algoritmos mais complexos, como os algoritmos genéticos. Em contrapartida, a capacidade de exploração do espaço de soluções de um algoritmo de busca local é limitada, fazendo com que os resultados de algoritmos deste tipo sejam, em geral, inferiores aos gerados por algoritmos mais complexos para certas classes de problemas.

Acreditamos que o resultado deste trabalho pode contribuir para o desenvolvimento de software de extração de dados baseada em exemplos, uma vez que existe uma quan-

tidade crescente de trabalhos que exploram a geração automática de expressões regulares [7, 9, 24, 36]. Neste contexto, o desempenho do algoritmo é um aspecto importante, dada a crescente disponibilidade de exemplos para a configuração (ou treinamento) de um algoritmo de extração de dados. Neste sentido, esta Dissertação analisa se há perda de eficácia (qualidade dos resultados) e se há ganho de eficiência (tempo de processamento) quando uma busca local é utilizada.

1.5 Metodologia da pesquisa

A metodologia de pesquisa para este trabalho começou com um estudo bibliográfico sobre técnicas e algoritmos de geração automática de expressões regulares (Capítulo 2) com o objetivo de entender o estado da arte da área e descobrir técnicas e ferramentas que pudessem apoiar a implementação do algoritmo de busca local proposto (Capítulo 3) e servir de base para uma comparação entre as ferramentas (Capítulo 4).

A técnica proposta foi implementada utilizando a linguagem TypeScript[4] em ambiente NodeJS[5]. TypeScript é uma extensão da linguagem JavaScript com o propósito de facilitar o desenvolvimento de sistemas JavaScript de larga escala [35]. Esta linguagem foi escolhida uma vez que um programa escrito em TypeScript pode ser convertido para JavaScript e executado em ambiente NodeJS e em navegadores *web* modernos (como Google Chrome[6] e Microsoft Edge[7]). Isso permite a sua execução via linha de comando, assim como a criação de uma interface *web* para a execução do algoritmo. Expressões regulares escritas em JavaScript também são compatíveis com um padrão muito abrangente (POSIX ERE), o que torna os resultados generalizáveis para outras linguagens que adotem o mesmo padrão.

Em seguida, foi desenvolvido um plano experimental para comparar o algoritmo de busca local aqui proposto a um algoritmo de programação genética e a um algoritmo exato em quinze instâncias do desafio *Regex Golf* (Capítulo 3). Executamos um conjunto de experimentos para aprimorar os operadores de vizinhança com o intuito de entender quais os tipos de adições ou remoções de operadores da expressão regular são melhores aplicados para chegar encontrar boas soluções. Como parte destes experimentos, realizamos uma comparação entre algoritmos de busca local e algoritmos genéticos no que tange à sua capacidade de encontrar boas soluções nesse espaço de busca. Realizamos uma aná-

[4]https://www.typescriptlang.org/
[5]https://nodejs.org
[6]https://www.google.com/chrome/
[7]https://www.microsoft.com/en-us/windows/microsoft-edge

lise de variações de buscas locais até concluirmos que a adição de um compactador de expressões regulares como causador de perturbações seria uma boa estratégia. Por fim, obtivemos os resultados dos ambos algoritmos exato e de programação genética para as 15 instâncias selecionadas para conseguirmos compara-los ao nosso algoritmo de busca local.

1.6 Estrutura da dissertação

Este trabalho está organizado em cinco capítulos. O primeiro capítulo compreende esta introdução. O segundo capítulo apresenta a revisão da literatura realizada sobre técnicas e algoritmos para a geração automática de expressões regulares, compreendendo não apenas algoritmos heurísticos de busca, mas também algoritmos de aprendizado de máquina e algoritmos exatos.

O terceiro capítulo descreve o desenvolvimento do algoritmo de busca local e do compactador de expressões regulares utilizados pela técnica de geração de expressões regulares proposta. São apresentadas as dificuldades e as decisões técnicas tomadas durante o desenvolvimento das ferramentas.

O quarto capítulo descreve o planejamento e execução do experimento realizado para avaliar a técnica proposta. São apresentados como foi realizada a execução do algoritmo, como os dados foram coletados e, no final, uma discussão sobre os resultados obtidos na sua avaliação, separando-os em resultados qualitativos e quantitativos, nos quais são analisados as modificações encontradas pela técnica proposta. Além disso, o capítulo aborda as ameaças à validade do experimento realizado.

Por fim, o quinto capítulo contém as considerações finais e contribuições da pesquisa reportada nesta Dissertação.

2. Estudos relacionados

2.1 Introdução

Algumas áreas do conhecimento possuem estudos de geração automática de expressões regulares com diferentes objetivos, como por exemplo, gerar uma expressão regular para extrair uma informação específica de um grande conjunto de dados. Existem também os casos em que é preciso obter uma expressão regular para validar algum tipo de informação. Áreas como aprendizado de máquina, otimização, Programação Genética e Evolução Gramatical apresentam estudos que descrevem o uso de heurísticas para a criação de expressões regulares.

Realizamos uma busca na literatura por estudos sobre técnicas e algoritmos para a geração automática de expressões regulares. A Figura 2.1 demostra as etapas efetuadas como parte da busca na literatura. A busca foi efetuada através da plataforma Scopus[1] utilizando a *query string* abaixo:

```
TITLE-ABS-KEY(((regular AND expression) OR regex) AND
              (generation OR creation OR build))
AND (LIMIT-TO (SUBJAREA, "COMP"))
```

A busca utilizando esta *query string* retornou 434 artigos. Em uma primeira análise, que consistiu na leitura do *abstract* destes artigos, foram descartados 384 artigos que não tratam da geração automática de expressões regulares e dois artigos não foram encontrados, restando 48 artigos. Após a leitura dos artigos, foram encontrados onze estudos que apresentam algoritmos para a geração automática de expressões regulares. Embora não tenha sido encontrado na busca realizada na Scopus, adicionamos o estudo do proponente do desafio *Regex Golf*, Peter Norvig [28], aos artigos selecionados por conter um algoritmo exato desenvolvido para o desafio.

[1] https://www.scopus.com/

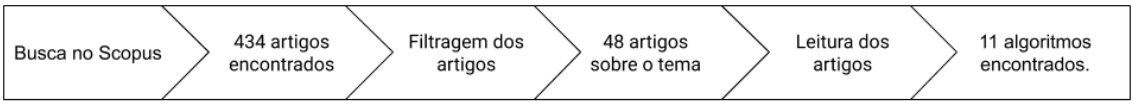

Figura 2.1: Etapas efetuadas na nossa busca na literatura

Os doze estudos resultantes da busca na literatura serão apresentados neste capítulo e estão listados na Tabela 2.1. Dentre os estudos encontrados, apenas dois são diretamente relacionados ao desafio *Regex Golf*: o estudo de Bartoli et al. [4], que usa um algoritmo genético adaptado para o problema, e o estudo supracitado de Peter Norvig [28].

Tabela 2.1: Artigos que tratam da geração automática de expressões regulares, encontrados na busca na literatura

Algoritmos exatos	
Design and development of generic web based framework for log analysis	Rastogi et al. [36]
Generating Evil Test Strings for Regular Expressions	Larson et al. [23]
Automatic generation of valid and invalid test data for string validation routines using web searches and regular expressions	Shahbaz et al. [40]
Regex Golf	Peter Norvig [28]
Algoritmos heurísticos	
Regular Expression Learning for Information Extraction	Yunyao Li et al. [24]
Automatic Synthesis of Regular Expressions from Examples	Bartoli et al. [3]
Playing Regex Golf with Genetic Programming	Bartoli et al. [4]
Inference of Regular Expressions for Text Extraction from Examples	Bartoli et al. [7]
Regular Expression Generation through Grammatical Evolution	Ahmet Cetinkaya [11]
Analysis of grammatical evolutionary approaches to regular expression induction	Antonio Gonzalez-Pardo e David Camacho [15]
Aprendizado de máquina	
A Semi-Supervised Active Learning Algorithm for Information Extraction from Textual Data	Wu et al. [46]
Active learning approaches for learning regular expressions with genetic programming	Bartoli et al. [5]

2.2 Algoritmos exatos

Um algoritmo é considerado exato quando seus dados de entrada (*input*) sempre geram os mesmos de saída (*output*), não importando a quantidade de vezes ou em qual momento no tempo ou ainda em qual ambiente (sistema operacional, hardware, etc) o algoritmo é executado. Na busca na literatura encontramos estudos que utilizam algoritmos exatos para geração automática de expressões regulares com o objetivo de validar ou extrair informações, além de um algoritmo exato criado para o desafio *Regex Golf*.

Rastogi et al. [36] criaram um sistema web de análise de *logs* que se utiliza de geração automática de expressões regulares para extrair dados dos *logs* e gerar relatórios personalizados para o usuário. O sistema é uma *Single Page Application* (SPA), um sistema *web* de apenas uma página, no qual o usuário pode enviar arquivos de *log* que serão processados e apresentados como registros em tela. O usuário, por sua vez, seleciona a parte de um registro do *log* referente ao *timestamp* e o sistema cria uma expressão regular para extrair o *timestamp* de todos os outros registros, utilizando a seleção do usuário como base. O usuário também pode selecionar outros tipos de dados nos registros para que sejam criadas expressões regulares para a sua extração.

Para gerar as expressões regulares baseadas no *timestamp*, por exemplo, é utilizado o seguinte algoritmo: para cada caractere no texto selecionado, se for um dígito, adicionar `\d` à expressão regular; se for uma letra, adicionar `\w`; e se for um caractere especial, adicionar `\W`. Como exemplo, o *input* "ah-65" resultaria na expressão regular `\w\w\W\d\d`. Para gerar as expressões regulares dos outros tipos de dados é utilizado outro algoritmo: para cada caractere no texto selecionado, se for um dígito, adicionar `\d` à expressão regular; se for uma letra do alfabeto, adicionar a letra em si; e se for um caractere especial, adicionar um *backslask*(`\`) e o caractere em si, para a expressão regular interpretar o caractere especial como um literal. Como exemplo, a entrada "192.16/AF" resultaria na expressão regular `\d\d\d\.\d\d\/AF`. A saída do sistema são gráficos gerados utilizando o Google Chart API[2] para exibir os relatórios das ocorrências dos *matches* realizados. O estudo não apresenta nenhum experimento, apenas indicações de uso com relatórios e imagens dos gráficos gerados pelo sistema.

Para a área de testes, Larson et al. [23] criaram um algoritmo, chamado EGRET, que cria textos para serem utilizados como contra-exemplos na validação de uma expressão regular. O EGRET trabalha convertendo a expressão regular em uma máquina de estados finita não-determinística (AFND). A AFND resultante é percorrida para se obter um con-

[2]https://developers.google.com/chart/

junto de caminhos básicos. As *strings* que correspondem a esses caminhos formam um conjunto inicial de *strings* de teste para a expressão regular. Essas *strings* serão processadas posteriormente para gerar *strings* adicionais. Para cada operador *character class*, *character set* e *dot*, um conjunto de caracteres é criado e uma *string* de teste é adicionada para cada caractere deste conjunto. Para cada operador *quantifier*, *strings* de teste são geradas utilizando diferentes contadores de iteração. Por exemplo, se o operador *quantifier* possuir o valor `*`, o contador pode ser selecionado entre 0 e 10; se for `+`, o contador pode ser selecionado entre 1 e 10; e se for `{2,5}`, o contador pode ser selecionado entre 2 e 5.

As decisões de *design* do EGRET concentram-se em erros comuns que programadores cometem ao criar expressões regulares. Em geral, a linguagem de expressão regular faz com que seja fácil escrever expressões regulares que aceitam mais *strings* do que deveriam. Por exemplo, uma expressão regular ruim que aceita um número de ponto flutuante é "`[0-9,.]+`". Ela aceitará corretamente todos os números de ponto flutuante, mas também aceitará sequências como "1,,00", "3.4.5" e ".". Escrever uma expressão regular que rejeite corretamente entradas mal formadas é mais difícil. O EGRET ajuda a encontrar esses problemas, gerando sequências que provavelmente estão incorretas. Ele foi capaz de gerar *strings* problemáticas como ".@." para uma expressão regular de *e-mail* e "200)00000000" para uma expressão regular de número de telefone.

Nos experimentos reportados pelos autores, EGRET foi comparado a outro algoritmo gerador de expressões regulares, chamado EXREX [42]. Foi criado um conjunto de testes contendo expressões regulares simples com uma ou duas expressões regulares com *bugs* de algum tipo conhecido pelos autores. Ambos algoritmos foram executados nesse conjunto de testes e as *strings* geradas foram analisadas manualmente. EGRET gerou *strings* capazes de encontrar os *bugs* inseridos nas expressões regulares, enquanto EXREX, apesar de ter gerado uma quantidade maior de *strings*, falhou ao tentar encontrar *bugs* que envolvem os operadores *start anchor* e *end anchor*, uma vez que ignora estes operadores.

Apesar de não ser um algoritmo de geração de expressões regulares, mas um algoritmo para gerar *strings* de contra-exemplo de uma expressão regular, EGRET apresentou bons exemplos de como gerar mutações de uma expressão regular, como a transformação de caracteres repetidos no operador *range*, a troca do operador *character set* para o operador *dot* e a transformação de dígitos para o operador *character class* "`\d`". Essas transformações foram utilizadas no trabalho de pesquisa descritos nesta Dissertação.

Zhang et al. [47] criaram um algoritmo chamado ARROW para gerar expressões regulares para detectar possíveis ataques de *download* automáticos (*drive-by download attack*). Um ataque de *download* automático ocorre quando um usuário visita uma pá-

gina *web* que tenta fazer um *download* automático de *malware* sem o consentimento do usuário. Às vezes, os invasores usam uma rede de distribuição de *malware* (MDN) para gerenciar um grande número de páginas maliciosas, *exploits* e executáveis de *malware*. ARROW apresenta um método para detectar essas MDNs, gerando uma expressão regular para identificar um endereço *web* (URL) marcado como potencial ameaça. Essa expressão regular se torna uma assinatura para o endereço *web*.

Para gerar essa expressão regular, ARROW utiliza um algoritmo que divide o endereço *web* em pequenos fragmentos chamados de *tokens*. Esses fragmentos podem ser uma parte do caminho da URL, a extensão do arquivo, a chave ou o valor de parâmetros da URL. Em seguida, o algoritmo cria uma expressão regular para cada um desses *tokens* com o intuito gerar uma expressão regular que será considerada a assinatura da MDN. Um exemplo de expressão regular gerada é apresentado abaixo.

```
domain.com/[a-z]{3,5}/default/[a-z]{5,6}.php?id=[0-9]{1}\&category=[0-9]{1}
```

O algoritmo ARROW é bastante simples, uma vez que ele gera uma expressão regular para cada pequena parte e depois constrói uma expressão regular maior referente ao endereço *web* completo. Uma vez que um MDN possui vários endereços (equivalente a várias páginas *web* de um mesmo domínio), ARROW consegue criar expressões regulares enxutas para realizar *matches*. Porém, o algoritmo não tem como objetivo verificar *unmatches*, o que torna este algoritmo incompleto para o desafio *Regex Golf*.

Peter Norvig [28] apresenta um algoritmo exato desenvolvido para o desafio *Regex Golf*. Este algoritmo constrói fragmentos de expressões regulares utilizando caracteres da *match list*. Estes fragmentos são criados da seguinte forma: para cada palavra na *match list*, é criada uma expressão regular que faça *match* na palavra inteira. Por exemplo, a palavra "abc" gerará a expressão regular "`^abc$`". Depois, são gerados fragmentos (*n-gram*) de 1 a 4 caracteres dessa expressão regular. Por exemplo, "`^abc$`" gerará os fragmentos "`^`", "`a`", "`b`", "`c`", "`$`", "`^a`", "`ab`", "`bc`", "`c$`", "`^ab`", "`abc`", "`bc$`", "`^abc`" e "`abc$`".

Para cada palavra na *match list* também são gerados fragmentos substituindo os caracteres da palavra pelo operador *dot*. Por exemplo, a palavra "abc" gerará os fragmentos "`.bc`", "`a.c`", "`ab.`", "`..c`", "`.b.`", "`a..`" e "`...`". Destes fragmentos, apenas os que não realizam *match* na *unmatch list* são mantidos. Uma função avaliadora é responsável por dar uma pontuação para os fragmentos. Um fragmento ganha quatro pontos para cada *match* realizado na *match list* e perde um ponto para cada caractere em seu tamanho.

Em seguida, o algoritmo ordena os fragmentos pela pontuação que cada um recebeu. Então, verifica se algum fragmento conseguiu realizar *match* em todos os itens da *match list*. Caso não encontre, o algoritmo criará outros fragmentos concatenando dois fragmentos da lista com o operador *alternation*. O primeiro fragmento será concatenado com o segundo, depois com o terceiro, e assim até o último. Depois, o segundo fragmento será concatenado com o terceiro, com o quarto, e segue-se esta sequência até o último fragmento da lista. Os novos fragmentos são adicionados a lista, são avaliados para receber uma pontuação e são novamente ordenados. Em seguida, o algoritmo verifica se algum fragmento conseguiu realizar *match* em todos os itens da *match list*. Este ciclo é repetido até uma solução ser encontrada.

Norvig testou seu algoritmo com algumas *match list* e *unmatch list* criadas com dados arbitrários (como os subtítulos dos filmes de *Star Wars* na *match list* e subtítulos dos filmes de *Star Trek* na *unmatch list*) e sempre conseguiu encontrar uma expressão regular que realize *match* em todos itens da *match list* e não realize nenhum *match* nos itens da *unmatch list*. O autor ressalta que foram utilizados poucos mecanismos da expressão regular: apenas os operadores *dot*, *start anchor*, *end anchor* e *alternation*. Portanto, um algoritmo mais sofisticado poderia utilizar mais recursos da linguagem para construir expressões regulares menores com o mesmo objetivo.

2.3 Algoritmos heurísticos

Muitos problemas de decisão em negócios e economia, em especial aqueles relacionados com atividades de fabricação, logística e distribuição, podem ser formulados como problemas de otimização combinatória [44]. Devido à complexidade de muitos desses problemas de otimização combinatória, algoritmos exatos normalmente apresentam fraco desempenho, em alguns casos levando dias ou meses para encontrar uma solução viável, quanto mais soluções ótimas, mesmo para instâncias pequenas dos problemas. Nestes cenários, algoritmos heurísticos são preferíveis por apresentarem soluções viáveis a um custo computacional razoável. Algoritmos heurísticos utilizam heurísticas para gerar e avaliar soluções para um problema. Heurísticas são critérios, métodos ou princípios para decidir quais dentre vários cursos alternativos de ação prometem ser os mais eficazes para atingir um determinado objetivo [33].

Problemas de otimização combinatória surgem em situações em que escolhas discretas devem ser feitas e resolvê-las equivale a encontrar uma solução ótima dentre um número finito ou infinito de alternativas [1]. A otimização se refere a algum critério de custo,

que fornece uma medida quantitativa da qualidade de cada solução. Muitos problemas de otimização combinatória são NP-difíceis. Acredita-se que os problemas NP-difíceis não podem ser resolvidos em tempo de computação polinomial. Consequentemente, há muito interesse em algoritmos heurísticos que possam encontrar soluções quase ótimas com tempos de execução aceitáveis para os problemas sob análise. Stefan Voß [44] distingue algoritmos heurísticos em três classes:

- **Métodos construtivos gulosos**: São heurísticas simples, disponíveis para qualquer tipo de problema de otimização combinatória [44]. Normalmente, uma heurística de construção gulosa começa com uma dada solução incompleta e a completa passo a passo. Em cada iteração, há várias opções alternativas (movimentos) que podem ser feitas para transformar a solução. A partir destas alternativas, que consistem na correção (ou alteração) de uma ou mais variáveis, é feita uma escolha gulosa, isto é, a melhor alternativa de acordo com uma dada medida de avaliação é escolhida até que tais transformações não sejam mais possíveis. Como cada alternativa tem que ser avaliada, é definido algum tipo de medida heurística que é seguida iterativamente até que uma solução completa seja construída;

- **Busca Local**: O princípio básico da Busca Local é que as soluções sofram transformações sucessivamente, realizando movimentos que as alterem *localmente* [44]. Transformações válidas são definidas por vizinhanças que apresentam soluções que podem ser alcançadas com um movimento. Movimentos devem ser avaliados por alguma medida heurística para guiar a busca. Normalmente usa-se a mudança do valor da função objetivo, que pode fornecer informações razoáveis sobre a vantagem (local) dos movimentos. Seguindo uma estratégia gulosa, em cada iteração é selecionado o melhor movimento baseado no resultado da função objetivo até que a busca pare em um ótimo local. Como a qualidade da solução dos ótimos locais pode ser insatisfatória, também existem mecanismos que guiam a busca para superar a otimização local. Uma estratégia simples, chamada busca local iterada, é reiniciar o processo de busca local depois que um ótimo local foi obtido, o que requer algum esquema de perturbação para gerar uma nova solução inicial (por exemplo, executando alguns movimentos aleatórios). É claro que formas mais estruturadas de superar a otimização local podem ser vantajosas;

- **Meta-heurística**: Uma meta-heurística é um processo mestre iterativo que orienta e modifica as operações de heurísticas subordinadas para produzir com eficiência soluções de alta qualidade [45]. Pode manipular uma solução única completa (ou incompleta) ou uma coleção de soluções em cada iteração. As heurísticas subordinadas podem ser procedimentos de alto (ou baixo) nível, uma busca local ou um

método construtivo. A família de meta-heurísticas inclui, mas não está limitada a, procedimentos de memória adaptativa, *tabu search*, colônia de formigas, *greedy randomized adaptive search*, busca de vizinhança variável, métodos evolutivos, algoritmos genéticos, *scatter search* e *simulated annealing*.

Na busca realizada na literatura, encontramos um estudo de Yunyao Li [24] que apresenta um algoritmo de busca local para a geração automática de expressões regulares baseadas em exemplos positivos e negativos. Também encontramos meta-heurísticas criadas por Ahmet Cetinkaya [11], Antonio Gonzalez-Pardo e David Camacho [15] e Bartoli et al. [3, 7], incluindo um algoritmo de programação genética para o desafio *Regex Golf* [4].

2.3.1 Busca Local

Busca local [13] é uma classe de algoritmos de busca baseados em um ponto que usa uma sistemática para percorrer a vizinhança de um determinado indivíduo procurando por um vizinho melhor (ou seja, um vizinho com maior valor para a função objetivo). Um vizinho de um determinado indivíduo A é o resultado da aplicação de uma mutação sobre A, como a alteração de uma característica individual. A vizinhança de um indivíduo A é definida pelo conjunto de vizinhos distintos que podem ser gerados pela aplicação de uma determinada mutação sobre A.

Quando um vizinho melhor é encontrado, ou seja, quando o valor da função objetivo é melhor do que o valor da mesma função para a solução de origem, o procedimento de pesquisa é repetido para examinar a vizinhança deste vizinho. Algoritmos de busca local favorecem a região da solução atual ao invés de uma exploração mais ampla do espaço de busca, concentrando a investigação na região vizinha de um dado indivíduo em detrimento do restante do espaço de busca.

Algoritmos de busca local são métodos computacionalmente simples, tendo baixo custo de execução se comparados a algoritmos de busca mais complexos. Eles tendem a ser eficientes e eficazes se partem de uma boa solução, ou seja, de uma solução que esteja próxima da solução ótima. Existem vários tipos de busca local, sendo o algoritmo de *Hill Climbing* o mais utilizado entre eles [16]. *Hill Climbing* é um algoritmo guloso que possui duas variantes principais: a subida mais íngreme (*Last-ascent Hill Climbing*) e subida mais próxima (*First-ascent Hill Climbing*). Na subida mais íngreme todos os vizinhos são avaliados e a heurística se move para o melhor deles. Na subida mais próxima, a heurística se move para o primeiro melhor vizinho encontrado sem avaliar o restante.

O estudo de Yunyao Li et al. [24] apresenta ReLIE, um algoritmo de *Hill Climbing* para gerar uma expressão regular baseada em exemplos positivos e negativos. O algoritmo recebe como parâmetros de entrada uma expressão regular inicial (R), uma coleção de documentos (D) e uma lista de exemplos (M) marcados como positivos ou negativos (ou seja, ao qual a expressão regular deve ou não deve realizar *match*, respectivamente).

No ReLIE, a vizinhança é gerada por um conjunto de transformações aplicadas à expressão regular representada pela solução atual. Essas transformações utilizam caracteres ou palavras extraídas da lista de exemplos. As transformações incluem, mas não se limitam a: (1) adicionar operadores *negative lookahead* contendo uma ou mais palavras (utilizando o operador *alternation* em conjunto) marcadas como negativas; (2) substituição de algum caractere alfanumérico pelo seu respectivo *character class* (por exemplo, um caractere "a" é substituído por `\w`, enquanto um caractere "8" é substituído por `\d`); (3) introdução de uma restrição nos operadores *quantifier* para realizar *match* no tamanho máximo de palavras na lista de exemplos marcadas como positivas (por exemplo, se a maior palavra possuir 5 caracteres, esta transformação alterará a construção `a+` para `a{1,5}`); e (4) adicionar operadores *negative character set* contendo um ou mais caracteres das palavras marcadas como negativas.

Para avaliar as expressões regulares da vizinhança, ReLIE usa uma função objetivo baseada em *F-measure* [39] para comparar a qualidade da extração das diferentes expressões regulares candidatas no espaço de busca. Utilizando $M_p(R,D)$ e $M_n(R,D)$ para expressar o conjunto de *matches* positivos e negativos, respectivamente, de uma expressão regular R, e $M_p(R_0,D)$ para expressar o conjunto de *matches* positivos da expressão regular R_0, a fórmula da função objetivo é definida da seguinte maneira:

$$precision(R,D) = \frac{M_p(R,D)}{M_p(R,D) + M_n(R,D)}$$

$$recall(R,D) = \frac{M_p(R,D)}{M_p(R_0,D)}$$

$$F(R,D) = \frac{2 \times precision(R,D) \times recall(R,D)}{precision(R,D) + recall(R,D)}$$

A execução do algoritmo ReLIE é uma busca local padrão na qual as transformações são aplicadas à expressão regular vigente para gerar a vizinhança e o algoritmo se move para o vizinho com a melhor pontuação na função objetivo.

O algoritmo foi testado utilizando 50 mil páginas *web* e uma coleção de 10 mil endereços de *e-mail* como documentos de entrada. O objetivo no teste era gerar expressões regulares para extrair nomes de software, nome de cursos e números de telefone das páginas *web*. As expressões regulares geradas pelo algoritmo conseguiram taxa de sucesso de 92%, 69% e 35%, respectivamente.

Este algoritmo se mostrou um bom candidato para o desafio *Regex Golf*, pois trabalha com um sistema similar ao de *match* e *unmatch lists*. Porém, o algoritmo utiliza poucos operadores para a geração de vizinhança das suas expressões regulares e possui poucas transformações, o que acarreta em gerar vizinhanças pequenas no espaço de busca.

2.3.2 Algoritmos Genéticos

Algoritmo Genético (AG) é uma meta-heurística baseada nos princípios da genética e seleção natural. Um AG permite que uma população composta por muitos indivíduos evolua sob regras de seleção especificadas para um estado que maximiza a "adequação" (isto é, minimiza a função objetivo) [17]. Um AG genérico é um processo iterativo no qual as iterações são chamadas de gerações e os membros da população são chamados de cromossomos (ou indivíduos), devido a seus análogos na evolução natural. Uma rodada de execução de AG consiste em uma busca heurística e estocástica no espaço de soluções, procurando uma solução próxima ao ótimo. Para este fim, uma população inicial é construída, geralmente de forma aleatória, e um procedimento iterativo é inicializado. A cada iteração são realizadas as seguintes ações: (1) construir novos cromossomos a partir dos existentes, usando operadores genéticos; (2) adicionar novos cromossomos na população; e (3) descartar os piores cromossomos. O processo termina quando a população satisfaz alguma condição pré-determinada ou um certo número de gerações foi excedido [7].

Para a criação de novos cromossomos, existem três tipos de operadores genéticos: seleção, cruzamento (*crossover*) e mutação [34]. Estes operadores também são análogos aos operadores que atuam na evolução natural das espécies e existem muitas versões destes operadores pela literatura. O objetivo do procedimento de seleção é reproduzir mais cópias de indivíduos cujos valores de aptidão são maiores do que aqueles cujos valores de aptidão são baixos. O processo de seleção tem uma influência significativa na condução da busca para uma área promissora para encontrar boas soluções em um curto

espaço de tempo. No entanto, a diversidade da população deve ser mantida para evitar a convergência prematura e alcançar a solução ideal global.

Em AGs há principalmente três procedimentos de seleção: seleção proporcional, seleção baseada em *ranking* e seleção por torneio. A seleção proporcional é geralmente chamada de "roleta", já que seu mecanismo é uma associação à operação de uma roleta. Os valores de aptidão dos indivíduos representam as larguras dos *slots* na roleta. Após um giro aleatório da roleta para selecionar um indivíduo para a próxima geração, os indivíduos em seções mais largas, representando valores altos de aptidão, terão uma chance maior de serem selecionados. Uma forma de evitar a convergência rápida é limitar o número de tentativas alocadas a um indivíduo, de modo que nenhum indivíduo gere muitos filhos. O procedimento de produção baseado em *ranking* baseia-se nessa ideia. De acordo com este procedimento, cada indivíduo gera um número esperado de descendentes, que é baseado na classificação do seu valor de aptidão e não na magnitude. A seleção por torneio escolhe aleatoriamente um número predeterminado de indivíduos da população e seleciona o melhor indivíduo desta conjunto para a próxima população. Este processo é repetido até que o tamanho desejado da população seja atingido.

O operador de cruzamento (*crossover*) é considerado o operador que diferencia os AGs de outros tipos de algoritmos. Ele é usado para criar dois novos indivíduos (filhos) a partir de dois indivíduos existentes (pais) escolhidos da população atual pela operação de seleção. Existem várias maneiras de gerar os novos indivíduos. Algumas operações comuns de cruzamento são o *crossover* de um ponto, *crossover* de dois pontos, *crossover* de ciclo e *crossover* uniforme. O *crossover* de um ponto é a operação de *crossover* mais simples. Dois indivíduos (pais) são selecionados a partir do resultado do procedimento de seleção e cortados em um ponto escolhido aleatoriamente. As caudas, que são as partes após o ponto de corte, são trocadas e dois novos indivíduos (filhos) são produzidos.

Já na operação de mutação, os indivíduos selecionados sofrem uma alteração aleatória, como, por exemplo, inverter o valor de um *bit* ou de alguma propriedade. Ao contrário do *crossover*, esta operação necessita apenas de um indivíduo, ou seja, um filho é produzido a partir de um único pai. O operador de mutação força o algoritmo a procurar novas áreas no espaço de solução, ajudando o AG a evitar a convergência prematura e encontrar soluções mais próximas ao máximo global.

Existem diversas variações dos algoritmos genéticos, mas os ingredientes cruciais são a maneira pela qual a função objetivo orienta a busca, a natureza recombinatória e populacional do processo e a representação de um cromossomo para o uso com os operadores genéticos. Na busca realizada na literatura, não encontramos nenhum trabalho

sobre geração de expressões regulares utilizando AGs genéricos. Porém, encontramos três estudos utilizando Programação Genética e dois estudos utilizando Evolução Gramatical para a geração automática de expressões regulares. Ambos algoritmos são variantes de algoritmos genéticos.

2.3.3 Programação Genética

Programação genética (PG) [21] é uma variante de algoritmos genéticos na qual o cromossomo é uma árvore de sintaxe abstrata de um programa de computador. Bartoli et al. [7] definem Programação Genética como um paradigma computacional evolutivo em que soluções candidatas para um problema, chamadas de indivíduos, são representadas como árvores de sintaxe abstrata nas quais um nó intermediário é uma função vinda do um conjunto de funções extraídas do programa e um nó folha é um valor possível de variável, vindo de um conjunto de literais também extraídos do programa. A função objetivo é uma função numérica para quantificar a habilidade de cada indivíduo em resolver o problema sob análise. Essa função geralmente é implementada como um índice de desempenho do indivíduo em um conjunto predefinido de instâncias de problemas, chamado conjunto de aprendizado. Uma rodada de execução da PG consiste em uma busca heurística e estocástica no espaço de soluções, procurando uma solução com maior adequação.

Bartoli et al. [3] criaram um algoritmo de Programação Genética para a geração automática de expressões regulares para a extração de dados com base em exemplos. Cada exemplo é composto por uma *string* `t` e a *substring* `s` de `t`, que foi extraída pela expressão regular desejada. Sem perda de generalidade, é assumido que cada exemplo contenha no máximo uma *substring* a ser extraída. Os exemplos em que `s` está vazio são chamados de exemplos negativos. Um indivíduo é representado por uma árvore de sintaxe abstrata, na qual o nó folha é um elemento de um conjunto de terminais predefinidos para a execução. Este conjunto pode conter letras maiúsculas e minúsculas do alfabeto, dígitos, pontuação, caracteres especiais, intervalos da expressão regular ("`a-z`", "`A-Z`" e "`0-9`"), dois *character classes* ("`\w`" e "`\d`"), além do *dot* ("`.`"). Um nó intermediário é um elemento de um conjunto de funções predefenidas que consiste nos seguintes operadores de expressão regular: *possessive quantifiers*, *group*, *character classes* e *negative character classes*.

A execução da PG começa a partir de uma população inicial de indivíduos gerados aleatoriamente e consiste em uma busca evolutiva como segue:

- Gerar uma população intermediária com a seguinte composição: 10% dos indivíduos são gerados aleatoriamente e 10% dos indivíduos são gerados aplicando o operador genético de mutação sobre um indivíduo selecionado da população atual;
- A seleção é realizada com um torneio de tamanho 7, ou seja, 7 indivíduos são selecionados aleatoriamente e, em seguida, o indivíduo com maior aptidão neste conjunto é selecionado;
- 80% dos indivíduos da população intermediária são gerados aplicando o operador genético de cruzamento a um par de indivíduos selecionados da população atual, cada qual selecionado através de um torneio independente;
- Construir uma nova população composta por indivíduos com maior aptidão entre aqueles presentes na população atual e na população intermediária (uma estratégia chamada de elitismo).

Essas etapas constituem uma geração. Este processo é iterado até que uma solução com perfeita aptidão seja encontrada ou um número máximo de gerações tenha evoluído. O tamanho da população é constante em todas as gerações.

Após a geração de um novo indivíduo, é realizada uma verificação da expressão regular correspondente. Se a verificação falhar, o indivíduo é descartado e um novo é gerado. A classificação dos indivíduos que passam pela verificação é realizada por meio de um algoritmo genético multiobjetivo padrão, o NSGA-II [12]. As funções objetivo deste algoritmo são: minimizar a soma das distâncias de Levenshtein (também chamadas de distâncias de edição) entre cada *string* detectada e a *string* desejada correspondente e minimizar o tamanho da expressão regular.

Nos experimentos realizados pelos autores, o algoritmo proposto foi testado em 12 diferentes tarefas de extração de dados (como endereços de *e-mail*, endereços IP, URLs, números de telefone, entre outros). Três destas tarefas foram extraídas de outros dois trabalhos com a intenção de comparar os resultados. O algoritmo proposto obteve resultados acima dos 90% de precisão em todas as tarefas.

Na comparação ao algoritmo de busca local de Yunyao Li et al. [24] (ReLIE apresentado na Seção 2.3.1), os autores indicaram que o algoritmo proposto obteve resultados similares ao algoritmo de busca local, porém não informaram os resultados. Na comparação ao algoritmo de evolução gramatical de Ahmet Cetinkaya [11] (que será apresentado na Seção 2.3.4), os autores alegaram que o algoritmo proposto obteve resultados superiores ao algoritmo de evolução gramatical, obtendo resultados próximos a 100% de precisão

nas duas tarefas comparadas, enquanto o algoritmo de evolução gramatical obteve precisão de 76%.

Em outro estudo, Bartoli et al. [4] propuseram um algoritmo de PG para a geração de expressões regulares para o desafio *Regex Golf*. Este algoritmo utilizou o trabalho de Peter Norvig [28] como base para aperfeiçoar a geração de expressões regulares para o *Regex Golf*. Os autores testaram seu algoritmo utilizando 16 instâncias do desafio extraídas do *site https://alf.nu/RegexGolf*, que disponibiliza uma versão *online* do *Regex Golf* em que qualquer pessoa pode sugerir expressões regulares para várias instâncias do desafio. O algoritmo apresentou soluções melhores do que as sugeridas por humanos, porém não atingiu a pontuação máxima em todas as instâncias.

O experimento primeiramente comparou o algoritmo baseado em PG com as melhores expressões regulares geradas por humanos (dados extraídos do próprio *site* em que as instâncias foram extraídas). Nesta comparação, o algoritmo de PG gerou resultados melhores para todas as instâncias. Já na comparação com o algoritmo exato de Peter Norvig (apresentado na Seção 2.2), o algoritmo de PG obteve melhores resultados em seis instâncias, empatou em duas, e obteve resultados inferiores em oito instâncias.

No estudo seguinte, Bartoli et al. [7] aperfeiçoaram o seu algoritmo PG para a geração de expressões regulares adicionando o operador *alternation* ("|") ao conjunto de funções predefinidas para os nós intermediários da árvore sintática. Isto possibilitou uma estratégia de "dividir para conquistar", na qual é possível dividir a expressão regular em partes que realizam *matches* em diferentes palavras. Os autores também adicionaram os operadores *lookahead* e *lookbehind* para utilizar os caracteres presentes antes ou depois de uma parte da *string* para melhor a chance de realizar o *match*.

Assim como o algoritmo anterior [4], o novo algoritmo voltado para o *Regex Golf* foi inicialmente comparado com expressões regulares criadas por pessoas e, depois, com o algoritmo apresentado no trabalho dos mesmos autores voltado para a extração de dados [3]. Para obter as expressões regulares das pessoas, Bartoli et al. realizaram um estudo [6] no qual 73 usuários (60% considerados novatos, 20% intermediários e 20% experientes com expressões regulares) criaram expressões regulares para 20 tarefas de extração de dados (12 tarefas do trabalho anterior e 8 tarefas adicionadas neste estudo).

O novo algoritmo obteve melhores resultados em 19 das 20 tarefas selecionadas em comparação as expressões regulares criadas pelas pessoas. Na comparação com o algoritmo do estudo anterior, foram realizados testes com as mesmas 20 tarefas e o novo algoritmo obteve melhores resultados em todas as instâncias.

2.3.4 Evolução Gramatical

Evolução Gramatical (EG) é um sistema baseado em algoritmos genéticos que desenvolve programas em qualquer linguagem usando uma definição de *Backus Naur Form* (BNF) [32] da linguagem. BNF é um conjunto de regras de produção que são usadas para expressar gramáticas de idiomas. Gramáticas BNF consistem em símbolos terminais, que são os itens que podem aparecer na linguagem, e não-terminais, que podem ser expandidos para um ou mais símbolos terminais ou não-terminais. Elas podem expressar apenas um subconjunto de um idioma que atenda às necessidades de um problema [11].

Em GE, os indivíduos são representados por sequências de 8 bits chamadas de codões. Cada codão codifica um número inteiro de 8 bits. Esta representação é conhecida como o genótipo. Estes números podem ser combinados, formando as características reais do indivíduo, ou fenótipo. Esta representação é independente do problema, pois uma vez encontrada a representação em vetores binários, as operações padrão podem ser utilizadas, facilitando o seu emprego em diferentes classes de problemas [38].

Diferentemente da Programação Genética, a EG não realiza o processo evolutivo nos programas reais, mas nas sequências de 8 bits. A partir do início do genótipo, o valor inteiro de cada codão é utilizado para calcular o índice de regras de produção atual. Por exemplo, para a regra `<meta-character>` abaixo, existem duas regras de produção para selecionar. Se o valor inteiro do codão que está sendo processado for `231`, então a operação `(231 MOD 2) = 1` selecionaria a regra (1), ou `<meta-character> :: = +`.

```
<meta-character> :: = "*"     (0)
                    | "+"     (1)
```

Números inteiros diferentes podem representar a mesma regra de produção devido ao uso da operação de resto da divisão, como em `13 MOD 2 = 1` e `23 MOD 2 = 1`. Neste exemplo, todos os números ímpares selecionariam a regra `<meta-character> :: = +`. Este efeito é chamado de degeneração do código genético [31].

Antes que o fenótipo possa ser completamente construído, o processo de mapeamento pode chegar ao final da sequência do códão. A solução para este problema é chamada de "embrulhamento" e consiste em voltar ao começo e reusar os mesmos códões. O embrulhamento representa a propriedade dos "genes sobrepostos" da evolução, observada para muitos organismos. O número máximo de embrulhamentos é um parâmetro do processo de mapeamento. Se o mapeamento de um indivíduo atingir o número máximo sem pro-

duzir um fenótipo válido, o indivíduo é classificado como inválido. Indivíduos inválidos recebem o menor valor da função objetivo [31], sendo descartados no processo evolutivo.

Evolução Gramatical tem sido aplicada com sucesso em vários domínios de problemas nos quais algoritmos genéticos são mais relevantes. No caso do problema do *Santa Fe Trail*, os algoritmos de Evolução Gramatical obtiveram resultados superiores aos da PG [29]. *Santa Fe Trail* [43] é a instância mais comum do problema *Artificial Ant*, que consiste em encontrar um programa para controlar uma formiga artificial, limitada a uma quantidade de movimentos, com o objetivo de encontrar todas as migalhas de comida deixadas num percurso irregular em um plano de 32x32 espaços. Esta instância do problema *Artificial Ant* foi desenvolvida por Christopher Langton [19] e vem sendo utilizada como padrão de comparação entre diferentes tipos de algoritmos genéticos [43].

Na busca realizada na literatura, encontramos dois estudos que utilizam EG para a geração automática de expressões regulares: um de Ahmet Cetinkaya [11] e outro de Antonio Gonzalez-Pardo e David Camacho [15].

O algoritmo de Evolução Gramatical de Ahmet Cetinkaya [11] utiliza uma lista de frases de exemplos para serem utilizadas pela função objetivo. Antes da avaliação dos indivíduo pela função objetivo, cada indivíduo é passado pelo processo de mapeamento, que resulta em uma expressão regular para cada indivíduo. Então, o valor de aptidão de cada expressão regular é calculado. Para isso, os usuários fornecem um arquivo de n linhas onde m linhas estão marcadas, $m < n$. O resultado da função objetivo é o número de linhas marcadas em que a expressão regular realizou *match* somado ao número de linhas não marcadas em que a expressão regular não realizou *match*. Com esta estratégia de avaliação, o valor de aptidão mínimo é zero e o máximo é a quantidade de frases na lista de exemplos. Indivíduos inválidos recebem o valor de aptidão -1.

Para avaliar a técnica proposta, Ahmet Cetinkaya utilizou o código fonte de uma página *web* que continha 266 linhas como frases de exemplos. Oitenta e três linhas que continham uma *tag* âncora (e.g.: `<a href="otherpage.html">link to a page</a>`) foram marcadas. Seu objetivo era desenvolver uma expressão regular que realizasse *match* apenas em linhas que continham um *link* para uma outra página. Uma população inicial de 100 indivíduos de até 100 caracteres foi construída. O número de avaliações foi limitado em 250.000 e o operador de embrulhamento da EG foi implementado para o máximo de 5 vezes. A cada iteração foi realizada uma seleção de torneio (tamanho igual a 2) e *crossover* de dois pontos com probabilidade de 90%. Para a mutação, foram considerados os operadores *random-reset mutation* e de *creep mutation*. O operador *random-reset mutation* define, aleatoriamente, um novo número inteiro para cada gene do indivíduo sendo

processado com uma probabilidade definida previamente. *Creep mutation* adiciona $+1$ ou -1 para cada gene com a probabilidade de $1/N$, onde N é a quantidade de números inteiros no indivíduo. A tática de elitismo é utilizada para garantir que o melhor indivíduo da geração anterior participará na próxima geração. O algoritmo foi executado em 20 rodadas para o mesmo arquivo. Ele foi capaz de gerar expressões regulares satisfatórias: o autor testou estas expressões no código fonte de outras páginas *web* e obteve o resultado esperado de realizar *match* apenas nas linhas que continham uma *tag* âncora.

Antonio Gonzalez-Pardo e David Camacho [15] realizaram um estudo experimental para analisar o desempenho de diferentes abordagens de Evolução Gramatical em tarefas de indução de linguagem. Indução de linguagem é um problema conhecido de aprendizado de máquina que consiste em aprender uma gramática a partir de um conjunto de exemplos. O objetivo desses experimentos foi extrair expressões regulares que descrevem um conjunto de exemplos usando a EG sobre diferentes tipos de gramáticas. Ou autores testaram quatro diferentes tipos de gramáticas:

- *Context-free grammar* (CFG): Todas as regras de produção são similares a `NT` $\rightarrow$ `s` onde `NT` é um não-terminal e `s` é um conjunto de terminais e não-terminais. Este tipo de gramática é chamada de *Context-Free* porque `NT` pode ser substituído pelo conjunto `s` sem levar em conta o contexto de `NT`;
- *Context-free grammar with a penalized fitness function*: Uma extensão das CFG que utiliza o resultado da função objetivo para determinar a validade um indivíduo, ou seja, um indivíduo com valor baixo de adaptação (normalmente zero) é tratado como inválido;
- *Extensible context-free grammar*: Uma extensão da CFG que evita gerar indivíduos inválidos alterando as regras da CFG que possam gerar tais indivíduos;
- *Christiansen grammars*: Inicialmente denominadas Gramáticas Gerativas, permitem descrever idiomas extensíveis, como linguagens de programação, expressões regulares e outros idiomas que permitem ao usuário enriquecê-las com novos construtos linguísticos [30].

No estudo, foi utilizado um conjunto de dados com 38.000 endereços de páginas *web* (URL) e aplicaram a Lei de Zipf para extrair subconjuntos de símbolos (palavras e caracteres especiais) para a definição do vocabulário para a gramática do algoritmo de Evolução Gramatical. A lei de Zipf é uma lei de potência que ocorre quando a probabilidade de medir um determinado valor de alguma quantidade varia inversamente como uma potência

desse valor [27]. Em cada rodada, a população foi composta por 200 indivíduos e evoluiu durante 100 gerações. O número máximo de codões por indivíduo foi estabelecido como 50, enquanto o mínimo foi 20. O embrulhamento foi limitado a duas vezes. O operador de "*crossover*" foi aplicado com probabilidade de 95% e o operador de mutação foi aplicado a 4% dos indivíduos. Cada rodada do experimento testou 100 gramáticas e uma rodada foi considerada bem-sucedida se obteve ao menos um indivíduo que realize *match* em ao menos 90% das URLs de teste. O experimento mostrou que *Christiansen grammars* fornecem os melhores resultados entre as abordagens de EG, conseguindo atingir 100% de *match* em algumas rodadas.

Ambos estudos de Evolução Gramatical encontrados na busca realizada na literatura se aplicam ao desafio *Regex Golf*. O algoritmo de Ahmet Cetinkaya [11] tem o objetivo de gerar expressões regulares para realizar *matches* e *unmatches* em um conceito parecido com os das *match list* e *unmatch list*. O segundo estudo ([15]) apresenta um experimento em EG que pode ser utilizado para aprimorar tanto o algoritmo de Ahmet Cetinkaya quanto outros algoritmos genéticos.

2.4 Aprendizado de máquina

Aprendizado de Máquina (*Machine Learning*) é o uso de algoritmos e modelos estatísticos para desempenhar atividades específicas baseadas no reconhecimento de padrões e inferências ao invés de uma sequência de instruções previamente definidas [20]. É uma sub-área de Inteligência Artificial (AI). Algoritmos de aprendizado de máquina constroem modelos matemáticos baseados em dados já existentes, conhecidos como "dados de treinamento" (*training data*) para realizar predições ou tomar decisões sem ser explicitamente programados para realizar estas tarefas.

Aprendizado de máquina utiliza estatística na construção de modelos matemáticos para a tarefa de inferir a partir dos dados de treinamento. No treinamento, são utilizados algoritmos eficientes para resolver o problema de otimização em que se traduz a construção dos modelos, bem como para armazenar e processar a enorme quantidade de dados que geralmente é consumida por esses estudos. Segundo, uma vez que um modelo é "aprendido", sua representação e solução algorítmica de inferência também precisam ser eficientes. Em certas aplicações, a eficiência do algoritmo de aprendizagem pode ser tão importante quanto sua precisão preditiva [2].

Os tipos de algoritmos de Aprendizado de Máquina diferem em sua abordagem, no tipo de dados que eles consomem e produzem e no tipo de tarefa ou problema que eles

procuram resolver. Para situações em que dados não-classificados são abundantes e a classificação manual é cara, existe um tipo de algoritmo de aprendizado de máquina que questiona o usuário quando não consegue classificar a informação que está sendo processada. Quando o usuário classifica esta informação, o algoritmo adiciona a informação aos dados de treinamento e "aprende" para as próximas tarefas. Este tipo de algoritmo de aprendizado de maquina é chamado *Active Learning* [46]. Na busca realizada na literatura, encontramos dois algoritmos de *Active Learning* que geram expressões regulares: o algoritmo semi-supervisionado criado por Wu et al. [46] e o algoritmo atrelado ao um algoritmo de Programação Genética criado por Bartoli et al. [5].

Wu et al. [46] propuseram um algoritmo de *Active Learning* semi-supervisionado para a criação de expressões regulares para extrair dados de documentos. Trata-se de um algoritmo de aprendizado semi-supervisionado que utiliza exemplos (dados de treinamento) categorizados como relevantes ou irrelevantes para determinado tipo de dado especificado pelo usuário. Um algoritmo de *Active Learning* é utilizado para auxiliar o algoritmo de aprendizado semi-supervisionado com o intuito de reduzir o esforço em gerar os dados de treinamento. O usuário informa um exemplo positivo para um determinado tipo de dado e o algoritmo adiciona automaticamente mais exemplos positivos baseados no exemplo informado. Estes exemplos gerados são podados manualmente pelo usuário durante esta fase inicial (chamada pelos autores de "*active learning phase*") e o algoritmo de aprendizado semi-supervisionado automaticamente descobre expressões regulares para cada tipo de dado. Esta fase inicial é composta de três passos:

- Segmentação: Cada documento nos dados de treinamento é dividido em segmentos. Cada segmento se torna uma instância no sistema. Estes seguimentos são frases ou bloco de textos que são definidos utilizando o algoritmo de identificação de término de sentenças de Reymar e Ratnaparkhi [37];

- Rotulagem dos segmentos: Os usuários devem identificar os tipos de dados (atributos) que serão extraídos. Por exemplo, se o objetivo é extrair descrições físicas de pessoas, a lista deve incluir altura, peso, cor dos olhos, etc. Após esta rotulagem, cada atributo terá duas listas de segmentos: a *True set*, que possui elementos com dados considerados verdadeiros, e a *False set*, cujos elementos são considerados falso-positivos para o tipo de dado definido;

- Marcações da *part-of-speech*: Para cada palavra nos dados de treinamento deve ser atribuída uma marcação de *part-of-speech* antes do início do processo de aprendizagem. É utilizada a lógica de Eric Brill [10] para atribuir as marcações às palavras. Estas marcações diferenciam palavras comuns de possíveis tipos. Por exemplo,

palavras marcadas como pronomes (“eu”, “ele”, etc) não são utilizadas no treinamento.

O objetivo do algoritmo é descobrir sequências de palavras ou marcações de *part-of-speech* que tenham muitas recorrências na *True set*. Assim como a função objetivo do algoritmo ReLIE de Yunyao Li et al. [24], o algoritmo de aprendizagem usa uma função de avaliação baseada em *F-measure* para avaliar uma expressão regular candidata. Utilizando TP para expressar o conjunto de *matches* verdadeiros positivos, FP para expressar o conjunto de *matches* falso-positivos, e FN para expressar o conjunto de *matches* falsos negativos. A fórmula da função de avaliação (F_β) é definida da seguinte maneira:

$$P = precision = \frac{TP}{TP+FP}$$

$$R = recall = \frac{TP}{TP+FN}$$

$$F_\beta = \frac{(\beta^2+1)PR}{\beta^2 P+R}$$

Após a fase inicial, o algoritmo inicia a fase de aprendizado que utiliza um processo dividido em quatro fases para a geração da expressão regular:

- Encontrar um elemento raiz: Nesta primeira fase do algoritmo, cada palavra ou marcação de *part-of-speech* na lista *True set* é convertida para uma expressão regular e avaliada contra todos os segmentos que foram extraídos na fase inicial. O elemento que obter melhor pontuação será selecionado como elemento raiz;

- Processo de aprendizagem “AND”: Após encontrar um elemento raiz, o algoritmo cria novas expressões regulares concatenando palavras ou marcações de *part-of-speech* no início ou no final da expressão regular composta pelo elemento raiz. Estas novas expressões regulares são avaliadas e a que obter melhor pontuação substituirá a expressão regular atual criada pelo elemento raiz;

- Processo de aprendizagem “GAP”: Nesta fase, o algoritmo busca lacunas da expressão regular em relação aos exemplos marcados como relevantes. Essas lacunas são preenchidas com palavras ou marcações de *part-of-speech* até encontrar uma

expressão regular capaz de, não só realizar o *match*, mas também de extrair a informação para o atributo;

- Processo de aprendizagem do início e fim da expressão regular: Nesta fase, o algoritmo testa adicionar ambos operadores *start anchor* e *end anchor* (um de cada vez e depois juntos) à expressão regular atual. Primeiramente, é inserido o operador *start anchor* no início da expressão. Se o resultado da avaliação for igual ou melhor do que os resultados da expressão regular atual, esta nova expressão substituirá a atual. A mesma operação é realizada com o operador *end anchor* sendo adicionada no final da expressão.

Este processo de geração da expressão regular é executado para cada atributo definido na fase inicial. O algoritmo de *Active Learning* semi-supervisionado foi testado utilizando 100 relatórios de incidentes policiais, contendo 1404 seguimentos depois da fase inicial. Os atributos definidos para serem extraídos foram cor de cabelo, idade, altura, etnia e peso de uma pessoa, e também a data e hora do relatório. As expressões regulares resultantes conseguiram *matches* em mais de 95% em quase todos os atributos. Porém os atributos de data, hora e peso ficaram abaixo dos 80% dos *matches* esperados.

Em um outro estudo, Bartoli et al. [5] propuseram um algoritmo de *Active Learning* atrelado a um algoritmo de programação genética (baseado no algoritmo de programação genética de Bartoli et al. apresentado na Seção 2.3.3) que também possui o objetivo de gerar expressões regulares para extrair informações de documentos. Foi desenvolvido um sistema no qual o usuário inicialmente seleciona um trecho do texto do documento como a informação a ser extraída. Um algoritmo de aprendizagem baseado em programação genética constrói uma solução (uma expressão regular que faça *match* no trecho selecionado) e depois começa uma investigação no texto do documento para selecionar o trecho mais apropriado a ser usado para melhorar a solução atual (*extraction query*). Um oráculo responde à questão especificando se o trecho selecionado deve ser extraído ou não e o processo continua iterativamente, melhorando a solução a cada consulta.

O oráculo inicialmente é o usuário do sistema. Após a primeira iteração, o oráculo passa a aprender. Ele é composto por três componentes: o *Solver*, que gera uma expressão regular adequada aos *matches* e *unmatches* anotados pelo oráculo; o *Query builder*, que constrói as *extraction queries* e determina qual delas deve ser apresentada ao oráculo; e o *Query trigger*, que determina quando uma *extraction query* deve ser apresentada ao oráculo. A *extraction query* enviada para o oráculo é composta da expressão regular atual e de duas listas de trechos do texto: uma de *matches* e outra de *unmatches*. O oráculo então utiliza o algoritmo de Programação Genética para gerar a expressão regular que

realize *match* em todos os trechos do texto da primeira lista e que não realize *matches* nos trechos do texto da segunda lista (ou seja, realize *unmatches*). A lista de *unmatches* pode ser uma lista vazia, mas a de *matches* necessariamente precisa possuir ao menos um trecho de texto.

O algoritmo foi testado com diferentes configurações para determinar qual configuração teria o melhor desempenho. Os autores escolheram 11 tarefas de extração de dados do estudo em que o algoritmo de PG foi apresentado [3] e, para cada uma dessas tarefas e para cada variação da configuração, foram selecionadas aleatoriamente cerca de 100 linhas de texto dos dados de treinamento. Depois, foi definido aleatoriamente um trecho a ser extraído para simular o comportamento do usuário. Foram executadas 15 rodadas de testes. Os autores concluem que o melhor resultado entre os algoritmos e configurações diferentes foi capaz de gerar expressões regulares tão boas quanto o algoritmo de PG, porém com redução drástica do trabalho manual necessário para realizar a tarefa.

2.5 Considerações finais

Neste capítulo foram apresentados trabalhos relacionados à geração automática de expressões regulares divididos nas áreas de algoritmos exatos, algoritmos heurísticos e algoritmos de aprendizado de máquina. Os trabalhos realizados com algoritmos exatos possuem um foco maior em validação de dados ou validação da própria expressão regular. Já os algoritmos heurísticos e de aprendizado de máquina são focados em gerar expressões regulares para extração de dados de documentos.

O algoritmo de Programação Genética de Bartoli et al. [4] e o algoritmo exato de Peter Norvig [28] são os algoritmos criados especificamente para o desafio *Regex Golf*. Porém, o algoritmo de busca local de Yunyao Li et al. [24], o algoritmo de evolução gramatical de Ahmet Cetinkaya [11] e o algoritmo de *active learning* atrelado ao algoritmo de PG de Bartoli et al. [5] são bons algoritmos candidatos para o desafio por utilizarem exemplos e contraexemplos (similar as *match list* e *unmatch list*) para a geração de expressão regulares. Por outro lado, estes algoritmos não possuem a característica de gerar a menor expressão regular possível para o problema, conforme estabelecido no *Regex Golf*.

No próximo capítulo descreveremos como foi desenvolvido o algoritmo de busca local atrelado a um compactador de expressões regulares para o desafio *Regex Golf* proposto nesta Dissertação.

3. Algoritmo de busca local para construir expressões regulares

3.1 Introdução

O desafio *Regex Golf* é considerado um problema NP-difícil [4, 28] e heurísticas de busca são bons métodos para gerar resultados para problemas de análise combinatória com esse nível de complexidade. Conforme apresentado no capítulo 2, existem algoritmos de geração automática de expressões regulares que podem ser adaptados para o desafio *Regex Golf*, em especial o algoritmo exato de Peter Norvig [28] e o algoritmo de programação genética de Bartoli et al. [4] que foram criados especificamente para o desafio. Para este trabalho de pesquisa, apresentaremos uma heurística de busca local atrelada a um algoritmo compactador de expressões regulares para o desafio *Regex Golf*. A Figura 3.1 apresenta o fluxograma desta nova busca local.

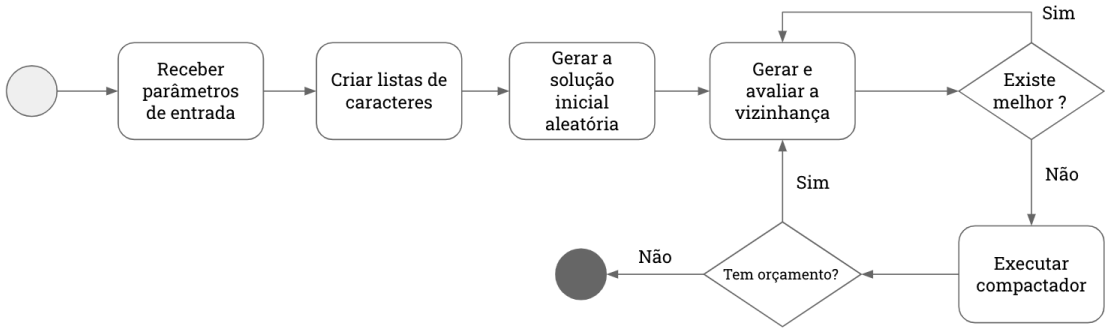

Figura 3.1: Fluxograma da busca local

A heurística de busca local recebe como parâmetros de entrada a *match list* e a *unmatch list*. Em sua fase inicial, o algoritmo cria as listas de caracteres (Seção 3.4) que irão ajudar a gerar a vizinhança de uma solução. O algoritmo é iniciado com uma solução gerada aleatoriamente (Seção 3.5). A vizinhança desta solução é gerada seguindo as

transformações apresentadas na Seção 3.4. A heurística de busca local usa a estratégia *Last-ascent Hill Climbing*, ou seja, são avaliadas todas as soluções que compõem a vizinhança e a heurística se move para a melhor delas em cada uma das interações. Se a vizinhança não apresentar nenhuma solução melhor do que a solução atual, é executado o algoritmo compactador de expressões regulares (Seção 3.6) para gerar uma solução derivada da solução atual. Se a solução gerada for melhor, a heurística se move para ela. Caso a expressão regular resultante do algoritmo compactador não seja melhor que a solução atual, uma nova solução será gerada aleatoriamente para a continuidade da execução do algoritmo.

3.2 Representação da solução do problema

Para a heurística de busca local proposta para o desafio *Regex Golf*, uma solução é representada por uma árvore onde os nós terminais são os caracteres (o operador *character class* com o valor sendo o caractere em si) e os nós função são os operadores da expressão regular. Os nós terminais são representados pelo caractere que eles possuem, enquanto os nós função apresentam os caracteres que representam o operador da expressão regular contendo um *placeholder*, o caractere •, que representa o local em que os caracteres dos nós terminais ou expressões representadas em ramos da árvore ocuparão quando a expressão regular for convertida para o formato texto. Este modelo é baseado na representação da solução criada para o algoritmo de Programação Genética de Bartoli et al. [4]. A Figura 3.2 demostra a expressão regular `^ab[^c]` no formato de árvore sintática.

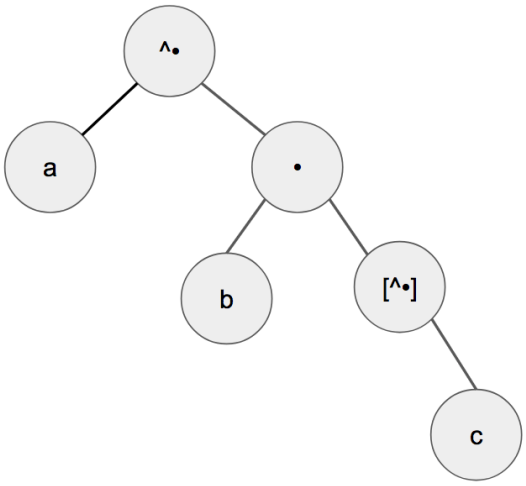

Figura 3.2: Representação em árvore sintática da expressão regular $\hat{}ab[\hat{}c]$

Alguns operadores da expressão regular são representados por mais de um tipo de nó função. O operador *quantifier* possui três nós função (*Star quantifier*, *Plus quantifier* e *Repetition*) e o operador *character set* possui dois nós função (*list* e *range*) para representar as diferentes formas que os operadores possuem na expressão regular. O nó função *concatenation* (um nó binário que concatena os seus ramos) foi adicionado para ajudar na representação e nas mutações realizadas na árvore sintática. A Tabela 3.1 apresenta todos os possíveis nós função e as suas representações.

Nó Função	**Representação**
Concatenation	•
Alternation	•|•
Negative char set	[^•]
Start anchor	^•
End anchor	•$
Star Quantifier	•*
Plus Quantifier	•+
Repetition	•#
List	[•]
Range	[•-•]
Optional	•?
Group	(•)
Dot	.•
Backreference	\#•

Tabela 3.1: Nós função e suas representações

3.3 Função objetivo

Uma função objetivo (*fitness function*) é utilizada para determinar o valor de uma solução. Uma solução é considerada viável quando ela possui uma expressão regular que faça *match* em todas as frases da *match list* e não faça *match* em nenhuma das frases da *unmatch list*. Utilizaremos a mesma função objetivo usada no trabalho de Bartoli et al. [4] que, por sua vez, utilizou a função avaliadora do *website* `https://alf.nu/RegexGolf` de onde as instâncias dos seus experimentos foram coletadas.

Para calcular o valor de uma solução, a expressão regular é executada para cada frase da *match list* e para cada frase da *unmatch list*. Quando ocorre um *match* em uma frase da *match list*, a solução ganha um ponto. Quando ocorre um *match* em uma frase da *unmatch list*, a solução perde um ponto. O resultado é multiplicado por um valor estático definido para cada instância que representa o peso da instância. O peso da instância é um fator de multiplicação que é aplicado a quantidade de *matches* e representa o grau de dificuldade da solução. Finalmente, substrai-se a quantidade de caracteres da expressão regular do resultado da operação anterior, de modo a favorecer expressões regulares menores. A função objetivo é definida pela seguinte fórmula:

$$W(M-U) - length(R)$$

onde R é a expressão regular candidata à solução, M é a quantidade de *matches* na *match list* da expressão regular R, U é a quantidade de *matches* na *unmatch list* da expressão regular R e W é o peso da instância, conforme estabelecido no *website*. Caso duas soluções possuam a mesma pontuação, os critérios de desempate são a quantidade de caracteres na solução (quanto menos caracteres tiver a expressão, melhor), a quantidade de *matches* na *match list* (desconsiderando os *matches* da *unmatch list*) e a ordem de criação (a primeira a ser gerada tem maior prioridade).

A pontuação máxima que uma solução pode atingir é a quantidade de frases na *match list* multiplicada por W. É importante ressaltar que os valores mínimo e máximo dos resultados desta função objetivo são relacionados à instância do problema e não ao problema em si e que é possível haver resultados negativos.

3.4 Vizinhança

Para a construção da vizinhança de uma solução, são efetuadas transformações que geram novas soluções. Estas transformações foram baseadas em ambos algoritmos exato [28] e de programação genética [4] criados para o desafio *Regex Golf*. Estas transformações se utilizam de quatro listas de caracteres, formadas a partir das frases da instância do problema. As listas de caracteres criadas a partir da instância do problema são:

- *match char list*: Uma lista de caracteres composta pelos caracteres existentes na *match list*. São extraídos todos os caracteres válidos da *match list*, removendo os

duplicados para compor uma lista de caracteres únicos. Os caracteres são ordenados baseado na quantidade de vezes em que aparecem nas frases. A Tabela 3.2 apresenta essa lista para a instância do problema usada como exemplo na introdução (Seção 1.2);

t	\s	o	e	h	r	i	f	l	p	n	b	a
s	d	c	m	z	k	g	’	x	-	y	w	

Tabela 3.2: *match char list*: Lista com os caracteres extraídos e ordenados da *match list*

- *unmatch char list*: Uma lista de caracteres similar a *match char list*, porém baseada na *unmatch list*. Ela contém todos os caracteres válidos das frases da *unmatch list* e segue o mesmo critério de ordenação baseado nos caracteres que mais se apresentam nestas frases. A Tabela 3.3 apresenta essa lista para a instância do problema usada como exemplo;

t	\s	a	r	e	h	s	o	i	d	g	n
l	f	y	u	k	v	,	x	’	c	m	

Tabela 3.3: *unmatch char list*: Lista com os caracteres extraídos e ordenados *unmatch list*

- *inclusive char list*: Uma lista com todos os caracteres que pertencem a *match char list* e que não estejam presentes na *unmatch char list*. A Tabela 3.4 apresenta essa lista para a instância do problema usada como exemplo;

p	b	z	-	w

Tabela 3.4: *inclusive char list*: Lista com os caracteres da *match char list* que não pertencem a *unmatch char list*

- *exclusive char list*: Uma lista com todos os caracteres que pertencem a *unmatch char list* e que não estejam presentes na *match char list*. A Tabela 3.5 apresenta essa lista para a instância do problema usada como exemplo.

u	v	,

Tabela 3.5: *exclusive char list*: Lista com os caracteres da *unmatch char list* que não pertencem a *match char list*

As transformações são pequenas alterações realizadas sobre uma solução que resultam em novas soluções. A vizinhança de uma solução é composta por todas as soluções geradas a partir das transformações apresentadas a seguir:

- *Swap*: Realiza a troca de um nó da árvore por um nó terminal com valor de um caractere da *inclusive char list*. Esta transformação é efetuada N vezes para cada nó da solução atual, sendo N a quantidade de itens na *inclusive char list*. O resultado são $N \times L$ soluções geradas, sendo L a quantidade de nós da solução atual. Por exemplo, dada a *inclusive char list* "(t, x)" e a expressão regular `^ab[^c]`, são geradas as seguintes soluções: `tab[^c]`, `^tb[^c]`, `^at[^c]`, `^abt`, `^ab[^t]`, `xab[^c]`, `^xb[^c]`, `^ax[^c]`, `^abx` e `^ab[^x]`;

- *Concatenation*: Realiza a troca de um nó da árvore por um nó função do tipo *concatenation* contendo um nó terminal com um valor de um caractere da *inclusive char list* e um nó extraído da solução em seus ramos. Esta transformação é efetuada $2N$ vezes para cada nó da solução atual, sendo N a quantidade de itens na *inclusive char list*. O resultado são $2N \times L$ soluções geradas, sendo L a quantidade de nós da solução atual. Por exemplo, dada a *inclusive char list* "(t, x)" e a expressão regular `^ab[^c]`, são geradas as seguintes soluções válidas: `t^ab[^c]`, `^tab[^c]`, `^atb[^c]`, `^abt[^c]`, `^ab[^ct]`, `^ab[^c]t`, `x^ab[^c]`, `^xab[^c]`, `^axb[^c]`, `^abx[^c]`, `^ab[^cx]` e `^ab[^c]x`. A transformação é efetuada duas vezes para cada nó da solução atual para garantir que o nó terminal adicional seja posicionado em ambos os ramos;

- *Start anchor*: Adiciona um nó do tipo *start anchor* como raiz da árvore da solução. Por exemplo, dada a expressão regular `abc`, será gerada a solução `^abc`. Esta transformação também ocorre nos ramos dos nós função do tipo *alternation*. Por exemplo, dada a expressão regular `ab|cd|ef`, são geradas as soluções `^ab|cd|ef`, `ab|^cd|ef` e `ab|cd|^ef`. O resultado são $X + 1$ soluções geradas, sendo X a quantidade de nós *alternation* presentes na solução atual;

- *End anchor*: Realiza a troca do último nó da árvore (o nó com a maior profundidade) por um nó função do tipo *concatenation*, contendo em um dos ramos o nó substituído e no outro ramo um nó do tipo *end anchor*. Por exemplo, dada a expressão regular `abc`, será gerada a solução `abc$`. Assim como a transformação *start anchor*, esta transformação também ocorre nos ramos dos nós funções do tipo *alternation*. Por exemplo, dada a expressão regular `ab|cd|ef`, são geradas as so-

luções `ab|cd|ef$`, `ab|cd$|ef` e `ab$|cd|ef`. O resultado são $X+1$ soluções geradas, sendo X a quantidade de nós *alternation* presentes na solução atual;

- *Alternative*: Substitui um nó da árvore por um nó função do tipo *alternative* contendo os ramos do nó substituído e o adicionando em um desses ramos. Esta transformação é efetuada uma vez para cada nó da solução atual, porém muitas vezes o resultado gerado é inválido e acaba sendo descartado. Por exemplo, dada a expressão regular `^ab[^c]`, serão geradas as seguintes soluções válidas: `^a|b[^c]` e `^ab|[^c]`;

- *Negation*: Realiza a troca de um nó da árvore por um nó função do tipo *negative char set* com valor de um caractere da *exclusive char list*. Esta transformação é efetuada N vezes para cada nó da solução atual e N vezes adicionando o nó como raiz da árvore, sendo N a quantidade de itens na *exclusive char list*. O resultado são $(N \times L)+N$ soluções geradas, sendo L a quantidade de nós da solução atual. Se o nó a ser trocado for um nó do tipo *negative char set*, o valor do nó que seria adicionado a árvore é adicionado ao nó presente na árvore, resultando em um nó do tipo *negative char set* com um caractere a mais. Por exemplo, dada a *exclusive char list* "(x, y)" e a expressão regular `ab[^c]`, são geradas as soluções `[^x]ab[^c]`, `[^x]b[^c]`, `a[^x]b[^c]`, `a[^x][^c]`, `ab[^x][^c]`, `ab[^cx]`, `ab[^c][^x]`, `[^y]ab[^c]`, `[^y]b[^c]`, `a[^y]b[^c]`, `a[^y][^c]`, `ab[^y][^c]`, `ab[^cy]` e `ab[^c][^y]`;

- *Range*: Primeiramente, é criada uma lista de nós função do tipo *range* com os valores de todas as combinações possíveis de dois caracteres da *inclusive char list*. Por exemplo, se a *inclusive char list* for "(a, h, m, x)", são criados os nós `[a-h]`, `[a-m]`, `[a-x]`, `[h-m]`, `[h-x]` e `[m-x]`. Após a criação desta lista, é realizada a troca de um nó da árvore por um nó função desta lista. Esta transformação é efetuada N vezes para cada nó da solução atual e N vezes adicionando o nó como raiz da árvore, sendo N a quantidade de itens na lista recém criada. O resultado são $(N \times L)+N$ soluções geradas, sendo L a quantidade de nós da solução atual. Por exemplo, dada a *inclusive char list* "(a, m, y)" e a expressão regular `xyz`, são geradas as soluções `[a-m]xyz`, `[a-m]yz`, `x[a-m]yz`, `x[a-m]z`, `xy[a-m]z`, `xy[a-m]`, `xyz[a-m]`, `[a-m]xyz`, `[a-m]yz`, `x[a-m]yz`, `x[a-m]z`, `xy[a-m]z`, `xy[a-m]`, `xyz[a-m]`, `[m-y]xyz`, `[m-y]yz`, `x[m-y]yz`, `x[m-y]z`, `xy[m-y]z`, `xy[m-y]` e `xyz[m-y]`;

- *Star quantifier*: Realiza a troca de um nó da árvore por um nó função do tipo *concatenation* contendo em seus ramos um nó função do tipo *star quantifier* e o nó extraído. Esta transformação é efetuada apenas nos nós terminais e nos nós função dos tipos *list*, *range*, *group* e *negation*. O resultado são N soluções geradas, sendo N a quantidade nós terminais somada a quantidade de nós função dos tipos aceitos por esta transformação. Por exemplo, dada a expressão regular `ab[h-z]`, são geradas as soluções `a*b[h-z]`, `ab*[h-z]` e `ab[h-z]*`;

- *Plus quantifier*: Esta transformação tem o mesmo comportamento da transformação *star quantifier*, porém utiliza um nó do tipo *plus quantifier*. Por exemplo, dada a expressão regular `ab[h-z]`, são geradas as soluções `a+b[h-z]`, `ab+[h-z]` e `ab[h-z]+`;

- *Repetition*: Realiza a troca de um nó da árvore por um nó função do tipo *concatenation* contendo em seus ramos um nó função do tipo *repetition* com o valor 2 e o nó extraído. Assim como a transformação *star quantifier*, esta transformação é efetuada apenas nos nós terminais e nos nós função dos tipos *list*, *range*, *group* e *negation*. Por exemplo, dada a expressão regular `ab[^z]`, são geradas as soluções `a{2}b[^z]`, `ab{2}[^z]` e `ab[^z]{2}`. Caso já exista algum nó função do tipo *repetition* na solução, o valor deste nó é incrementado em um. Por exemplo, a transformação na expressão regular `ab{2}[^z]` gera a solução `ab{3}[^z]`. O resultado são $N+L$ soluções geradas, sendo N a quantidade nós terminais somada a quantidade de nós função dos tipos aceitos por esta transformação e L a quantidade de nós do tipo *repetition*;

- *Dot*: Substitui um nó da árvore por um nó função do tipo *concatenation* contendo um nó função do tipo *dot* e o nó extraído da solução em seus ramos. Esta transformação é efetuada N vezes para cada nó da solução atual e uma vez adicionando o nó *dot* como raiz da árvore, sendo N a quantidade nós na solução atual. Como é possível gerar soluções inválidas, o resultado é de até $N+1$ soluções geradas. Por exemplo, dada a expressão regular `ab[^c]`, são geradas as seguintes soluções válidas: `.ab[^c]`, `.b[^c]`, `a.b[^c]`, `a.[^c]`, `ab.[^c]` e `ab.`;

- *Optional*: Esta transformação tem o mesmo comportamento da transformação *star quantifier*, porém utiliza um nó do tipo *optional*. Por exemplo, dada a expressão regular `ab[h-z]`, são geradas as soluções `a?b[h-z]`, `ab?[h-z]` e `ab[h-z]?`;

- *Extraction*: Remove um nó da árvore. Esta transformação é efetuada uma vez para cada nó da solução atual. Por exemplo, dada a expressão regular `a[b-h][^k]$`, são geradas as soluções `[b-h][^k]$`, `a[^k]$`, `a[b-h]$` e `a[b-h][^k]`.

3.5 Geração aleatória de uma solução

Para gerar uma solução aleatória para iniciar ou reiniciar a busca, o algoritmo gera um número entre um e o valor do parâmetro da profundidade da árvore para servir de profundidade da árvore da solução. Esta profundidade inicial foi definida através dos testes realizados durante os nossos experimentos. Para compor a solução inicial, é criado um nó função do tipo *concatenation* e um nó terminal contendo um caractere aleatório da *match char list* é adicionado como ramo. Um novo nó função do tipo *concatenation* é criado e um novo nó terminal contendo outro caractere aleatório da *match char list* é adicionado como ramo deste novo nó função. Este processo é repetido $X - 1$ vezes, sendo X o número gerado para servir de profundidade da árvore. Para a geração dos valores aleatórios, utilizamos uma implementação em JavaScript do algoritmo *Mersenne Twister*[1].

Mersenne Twiester [25] é um algoritmo de geração de números pseudo-aleatórios. Ele utiliza uma entrada de dados (*seed*) para gerar um estado inicial. Toda vez que se é solicitado um novo número para o algoritmo, o estado atual é passado para uma função de transformação de sentido único que recebe esse estado, retorna um número aleatório e gera um novo estado para o algoritmo. *Mersenne Twiester* é considerado um algoritmo pseudo-randômico exatamente pela sua característica de gerar os mesmos números na mesma ordem se baseando em um *seed*. Nos aproveitamos dessa característica e utilizamos os mesmos *seeds* nas execuções das diferentes instâncias do problema selecionadas durante os nossos experimentos (que serão apresentados no Capítulo 4) para assim termos uma melhor comparação nos resultados.

3.6 Compactador

O algoritmo compactador de expressões regulares tem o intuito de reduzir a quantidade de caracteres de uma expressão regular mantendo a mesma quantidade de *matches* na *match list* e a mesma quantidade de *unmatches* na *unmatch list*. Este algoritmo aplica transformações em cada nó da árvore sintática, tentando transformar em um outro nó

[1] https://github.com/boo1ean/mersenne-twister

que realize um *match* equivalente, porém com uma quantidade menor de caracteres. As possíveis transformações são:

- *Remover operadores redundantes*: Uma expressão regular com dois ou mais operadores *start anchor* ou *end anchor* é uma expressão válida, porém os operadores em excesso são desnecessários e podem ser removidos. Exemplos: `^^abc` e `abc$$` podem ser transformados, respectivamente, em `^abc` e `abc$`;

- *Remover valores duplicados*: Os nós dos tipos *list* e *negative char set* podem conter valores repetidos desnecessariamente. Esta transformação remove esses valores duplicados. Exemplos: as expressões regulares `abc[eefg]` e `a[^bbc]` podem ser transformadas em `abc[efg]` e `a[^bc]`, respectivamente;

- *Mesclar repetitions*: Quando nós do tipo *repetition* estão próximos na árvore, existe a possibilidade de mescla-los em apenas um nó sem afetar a quantidade de *matches* e *unmatches* da solução. Por exemplo, a expressão regular `a{1,2}a{2,3}` pode ser transformada em `a{3,5}`;

- *Converter para repetition*: Quando dois ou mais nós terminais ou nós função dos tipos *concatenation*, *list*, *range*, *group* e *negative char set* apresentam os mesmos valores e estão próximos um do outro, é possível substitui-los por um nó do tipo *repetition* contendo o valor destes nós. Por exemplo, as expressões regulares `abcccccdef`, `[a-d][a-d]` e `[^z][^z]` podem ser transformados em `abc{5}def`, `[a-d]{2}` e `[^z]{2}`, respectivamente;

- *Mesclar quantifiers*: Nós do tipo *star quantifier* ou *plus quantifier* que possuam os mesmos valores podem ser mesclados em apenas um nó. Por exemplo, as expressões regulares `a+a+`, `b*b*` e `c+c*` podem ser transformadas em `a+`, `b*` e `c+` respectivamente;

- *Simplificar alternation*: Um conjunto de nós do tipo *alternation* pode ser substituído por um nó do tipo *list* ou *range*. É necessário que sejam três ou mais nós do tipo *alternation* para que o nó resultante possua menos caracteres do que os atuais. Por exemplo, as expressões regulares `a|e|h|o` e `a|b|c|d` podem ser transformadas em `[aeho]` e `[a-d]`, respectivamente;

- *Remover alternation duplicados*: Uma sequência de dois ou mais nós do tipo *alternation* que possuam os mesmos valores podem ser removidos sem alterar a quan-

tidade de *matches* e *unmatches*. Por exemplo, a expressão regular `ab|ab` pode ser transformada em `ab`;

- *Simplificar ranges*: Um nó função do tipo *range* pode ter o valor alterado para diminuir a quantidade de caracteres e ainda efetuar os mesmos *matches* e *unmatches*. Quando o nó possuir apenas um caractere de valor, é possível substituí-lo por um nó terminal com este valor. Por exemplo, as expressões regulares `a[b-ed-h]c` e `[a-a]` podem ser transformadas em `a[b-h]c` e `a`, respectivamente;

- *Converter range para shorthand*: Um nó função do tipo *range* que contenha valores que equivalem a um *shorthand character class* pode substituído por um nó terminal que contenha o *shorthand character class*. Por exemplo, as expressões regulares `[a-zA-Z0-9]` e `[0-9]` podem ser transformadas em `\w` e `\d`, respectivamente;

- *Adição de backrefs*: Quando dois ou mais nós da árvore possuírem os mesmos valores e os valores forem mais que três caracteres em tamanho, é possível converter o primeiro nó para *group* que contenha os mesmos valores e os outros nós podem ser substituídos por nós do tipo *backref* fazendo referência ao nó *group* adicionado anteriormente. Por exemplo, as expressões regulares `abcdefabc` e `defabcdefabcabc` podem ser transformadas em `(abc)def\1` e `(def)(abc)\1\2\2`, respectivamente;

- *Simplificar negation*: Um nó do tipo *negation* que não possua valor pode ser substituído por um nó do tipo *dot* sem afetar a quantidade de *matches* e *unmatches* da solução. Por exemplo a expressão regular `a[^]bc` pode ser transformada em `a.bc`.

3.7 Considerações finais

As implementações inicias realizadas com fins de analise sofreram alterações durante os experimentos até ser concluída como a forma apresentada neste capítulo. A necessidade de adicionar o algoritmo compactador de expressões regulares ocorreu quando percebemos que a solução tende a crescer em número de caracteres durante a busca. Então, adicionamos este algoritmo para realizar a perturbação para quando a busca local encontrar um melhor local e também adicionamos transformações que removam nós da árvore durante a geração da vizinhança.

O código da implementação que utilizamos para os nossos experimentos (descritos no próximo capítulo) está hospedado no serviço *GitHub*[2] e pode ser acessado no endereço web[3]. No próximo capítulo, apresentaremos os resultados dos nossos experimentos com as 15 instâncias selecionadas e executadas no algoritmo de busca local, no algoritmo de programação genética criado por Bartoli e tal. [4], no algoritmo exato criado por Peter Norvig [28] e expressões regulares desenvolvidas por humanos.

[2]http://github.com

[3]'http://github.com/andrefarzat/regex-golf'

4. Avaliação do algoritmo de busca local proposto

4.1 Introdução

Nesse capítulo apresentaremos um estudo experimental com o intuito de avaliar o algoritmo de busca local proposto com relação à sua capacidade de encontrar boas expressões regulares para o desafio *Regex Golf*. Também apresentaremos os estudos realizados para definir os parâmetros de configuração utilizados no estudo experimental.

4.2 Projeto do estudo experimental

Esta seção apresenta as questões de pesquisa que direcionaram o estudo experimental que avaliou o algoritmo de busca local proposto nesta Dissertação, as instâncias do desafio *Regex Golf*, as ferramentas e o ambiente computacional utilizados na avaliação.

4.2.1 Instâncias selecionadas

Para comparar o algoritmo proposto nesta Dissertação com os algoritmos supracitados, selecionamos quinze instâncias que foram utilizadas num estudo experimental anterior realizado por Bartoli et al. [4] que, por sua vez, extraiu as instâncias do *website* `https://alf.nu/RegexGolf`. O *website* apresenta desafios populares para o *Regex Golf*, tendo sido utilizado tanto por algoritmos que tentam vencer o desafio, quanto por desenvolvedores de software que tentam encontrar soluções melhores que os algoritmos. Também utilizamos a mesma função objetivo (apresentada na Seção 3.3) usada no estudo anterior e no *website* para que os resultados sejam comparáveis.

A Tabela 4.1 apresenta as instâncias e a pontuação máxima possível para cada uma delas segundo a função objetivo. Conforme discutido na Seção 3.3 a pontuação máxima

de uma instância é a quantidade de frases na *match list* multiplicada pelo peso definido para cada instância. A instância "Long count v2", mencionada no estudo anterior, foi descartada por não ter sido encontrada no *website*.

Nome da instância	**Itens na *match list***	**Itens na *unmatch list***	**Peso**	**Pontuação máxima**
Plain strings	20	20	10	210
Anchors	21	21	10	210
Ranges	21	21	10	210
Backrefs	21	21	10	210
Abba	21	22	10	210
A man, a plan	19	21	10	190
Prime	20	20	15	300
Four	21	21	10	210
Order	21	21	20	210
Triples	21	21	30	630
Glob	21	21	20	420
Balance	32	32	10	320
Powers	11	11	10	110
Long count	1	20	270	270
Alphabetical	17	17	20	340
Total				**4.050**

Tabela 4.1: Instâncias selecionadas para o estudo experimental

4.2.2 Ambiente computacional

Os testes conduzidos durante a construção do algoritmo de busca local e o algoritmo compactador de expressões regulares foram executados em computadores comuns, com pouca variação em seu poder de processamento e quantidade de memória RAM. Para os resultados apresentados neste capítulo, foi utilizado um Macbook Pro com processador Intel Core i5 2,7 GHz e 8GB 1867 MHz DDR3 de memória RAM.

4.3 Parametrização do algoritmo

O algoritmo de busca local precisa que sejam definidos valores para dois parâmetros para que possa ser executado: a profundidade da árvore que representa a solução inicial

da busca e o critério de parada. Os valores destes parâmetros foram definidos com base em um conjunto de estudos, que são relatados a seguir.

4.3.1 Profundidade da árvore

A profundidade da árvore determina o número de nós no maior caminho desde a raiz até uma folha da árvore sintática abstrata que representa uma solução inicial para o algoritmo de busca local. A profundidade da árvore também é utilizada para gerar uma nova solução aleatória quando o algoritmo encontra uma solução ótima local e o algoritmo de compactação não produz uma expressão regular melhor. Este parâmetro determina o tamanho da árvore e, consequentemente, o tamanho da expressão regular que é gerada aleatoriamente nestas situações.

Para determinar o melhor valor de profundidade da árvore para o algoritmo proposto, executamos o algoritmo 30 vezes para cada instância do problema (Tabela 4.1) e para cada um dos valores de profundidade a seguir: 1, 3, 5, 7, 10, 13, 15, 18 e 20. Os critérios de parada utilizados nestas execuções foram de 500 mil avaliações da função objetivo e dez minutos de tempo máximo de execução (*timeout*). Para observar o impacto da variação do parâmetro de profundidade da árvore no algoritmo, comparamos o valor da função objetivo da expressão regular resultante em cada rodada de cada cenário e anotamos o número de avaliações necessárias para chegar a este resultado. Em seguida, utilizamos o teste de Kruskal-Wallis [22] com a hipótese nula de que a qualidade do resultado é insenível à profundidade da árvore para verificar se existe diferença nos resultados dos diferentes valores testados para o parâmetro.

McKight et al. [26] apresentam o teste de Kruskal-Wallis como um teste estatístico não-paramétrico que avalia as diferenças entre três ou mais amostras independentes sobre uma única variável contínua que não segue uma distribuição gaussiana. O teste de Kruskal-Wallis foi aplicado para determinar se há diferença significativa no valor da função objetivo das expressões regulares encontradas durante as variações dos valores de profundidade da árvore para cada instância selecionada. O resultado do teste é apresentado como um *p-value*, um valor entre 0 e 1 que indica a diferença entre as amostras, sendo que o valor 0 reseprenta o maior grau de diferença possível e valor 1 representa a menor diferença possível, ou seja que as amostra são iguais.

Nas instâncias consideradas fáceis (“Plain Strings”, “Anchors” e “Ranges”, que admitem soluções com menos do que 10 caracteres) e na instância “Long count” o teste retornou um *p-value* igual a 1.0, indicando que não há diferença significativa nessas instâncias. As instâncias fáceis apresentam expressões regulares diferentes como resultados,

mas sempre com a mesma pontuação, caracterizando a falta de diferença significativa entre os valores do parâmetro. A instância “Long count”, uma instância considerada difícil (instâncias que possuem menos de três caracteres diferentes na *inclusive char list*), retornou *p-value* igual a 1.0 por gerar a mesma expressão regular em todas as execuções com diferentes valores de profundidade de árvore testados.

As instâncias “Abba”, “A man, a plan”, “Triples” e “Balance” apresentaram *p-values* entre 0.8 e 0.9, enquanto as instâncias “Four”, “Alphabetical”, “Backrefs” e “Order” apresentaram *p-values* entre 0.3 e 0.6. As instâncias “Prime”, “Glob” e “Powers”, consideradas instâncias difíceis, retornaram os *p-values* 0.24, 0.16 e 0.01, respectivamente, e nenhuma das expressões regulares para as profundidades de árvore testadas para estas instâncias foi semelhante a outra expressão regular. O resultado mostra que não podemos rejeitar a hipótese nula de que a qualidade do resultado do algoritmo é insensível à profundidade da árvore utilizada para gerar a solução inicial (exceto em “Powers”).

Por outro lado, observamos que o tamanho da solução resultante é influenciado pela profundidade inicial da árvore. Se uma instância precisar que uma solução seja grande (com mais de 20 caracteres) e o parâmetro da profundidade de árvore for pequeno (inferior a 10 caracteres), o algoritmo não é capaz de se deslocar no espaço de busca e encontrar uma solução adequada. Isso nos indicou que a profundidade da árvore precisa ser determinada para cada instância.

Realizamos um segundo teste para definir qual o melhor valor para a profundidade da árvore para uma instância. Neste teste, o valor para a profundidade da árvore é definido pela quantidade de caracteres do maior item da *match list*. Ou seja, o algoritmo será capaz de gerar soluções aleatórias até o tamanho da maior frase com a qual ele precisa realizar *matches*. Assim, acreditamos que o algoritmo terá a flexibilidade necessária para percorrer no espaço de busca até encontrar uma solução viável. Executamos o algoritmo 30 vezes com a definição de profundidade da árvore por instância. Utilizamos as mesmas sementes aleatórias e o mesmo critério de parada do primeiro teste para podermos comparar os resultados com o teste anterior. A Tabela 4.2 apresenta a média da pontuação para cada valor testado para o parâmetro de profundidade da árvore e a média da pontuação com a profundidade definida de forma dinâmica (DN) para cada instância.

Na comparação dos valores das profundidades estáticas com a profundidade dinâmica, podemos observar que ao utilizar o tamanho da maior frase da *match list* como o valor da profundidade da árvore as médias das pontuações das instâncias “Backrefs”, “Abba”, “Prime”, “Four”, “Triples”, “Glob”, “Long count” e “Alphabetical” sobem consideravelmente. Isso indica que o algoritmo foi capaz de percorrer uma área maior do espaço

Instâncias	**1**	**3**	**5**	**7**	**10**	**13**	**15**	**18**	**20**	**DN**
Plain strings	207	207	207	207	207	207	207	207	207	207
Anchors	208	208	208	208	208	208	208	208	208	178
Ranges	201	201	201	201	201	201	201	201	201	196
Backrefs	123	127	121	128	125	122	126	124	123	168
Abba	142	141	141	141	141	141	140	140	142	299
A man, a plan	135	132	134	134	130	131	130	130	134	132
Prime	11	13	12	9	14	14	14	17	17	142
Four	145	146	145	144	144	147	147	146	144	182
Order	194	194	192	193	191	193	194	191	191	190
Triples	333	328	329	337	328	333	330	334	335	487
Glob	302	291	292	298	295	295	300	295	292	375
Balance	260	259	260	259	259	259	259	259	259	237
Powers	15	15	15	15	14	15	15	15	15	5
Long count	-1	-1	-1	-1	-1	-1	-1	-1	-1	212
Alphabetical	211	209	199	206	208	208	209	203	214	299

Tabela 4.2: Médias das pontuações das instâncias de acordo com o parâmetro de profundidade da árvore

de busca e encontrou resultados melhores. Em contrapartida, as instâncias “Anchors” e “Ranges” tiveram uma queda em suas médias, posto que as melhores expressões regulares possíveis são muito menores que a maior frase da *matchlist*. As instâncias “A man, a plan”, “Order” e “Balance” também tiveram a média da pontuação da profundidade dinâmica menor em comparação com as profundidades estáticas.

A instância “Powers” possui apenas uma frase de 1024 caracteres em sua *match list*, o que resulta em soluções grandes que geram vizinhanças gigantes e provocam execuções muito longas. Vizinhanças gigantes geram muitas avaliações para cada movimento do algoritmo, reduzindo o seu alcance no espaço de busca. Em nosso teste, todas as execuções desta instância foram terminadas por atingir o *timeout* de dez minutos, o que resultou na queda de sua média da pontuação em relação às outras configurações. A instância “Plain strings” é a única que não apresentou diferenças em nenhuma das configurações testadas.

A configuração dinâmica obteve melhores resultados em oito instâncias, empatou em três instâncias e perdeu em três das quinze instâncias. Com esse resultado, definimos que o parâmetro de profundidade da árvore será um valor dinâmico baseado na maior frase presente da *match list* da instância.

4.3.2 Critério de parada

O algoritmo de busca local possui dois critérios de parada: um limite de avaliações e um limite de tempo (*timeout*). Ao alcançar um dos limites, o algoritmo retorna a melhor solução encontrada até o momento como o seu resultado. Para encontrar valores adequados para o algoritmo de busca local, realizamos experimentos com limites de avaliação de 300 mil, 500 mil, 800 mil e 1 milhão de avaliações e comparamos os resultados obtidos com estes limites. Definimos o limite de dez minutos de *timeout*. Executamos o algoritmo 30 vezes em cada instância selecionada para o problema e cada um dos limites de avaliações. Durante a execução destes testes, observamos que em grande parte das execuções o algoritmo encontra a melhor solução antes de realizar 200 mil avaliações.

Para testar um terceiro critério de parada, adicionamos um limite de avaliações sem melhorias. Assim, o algoritmo seria interrompido se consumisse um determinado número de avaliações, menor do que o número total de avaliações, sem apresentar melhoria na função objetivo. Executamos o algoritmo 30 vezes com o limite de 1 milhão de avaliações, o limite de 200 mil avaliações sem melhorias e *timeout* de 10 minutos. A Tabela 4.3 apresenta as médias dos números de avaliações necessárias até encontrar a melhor solução e a pontuação da melhor solução encontrada em cada configuração.

Instâncias	**300 mil**		**500 mil**		**800 mil**		**1 milhão**		**1 milhão (200 mil)**	
	pontos	avaliações	pontos	avaliações	pontos	avaliações	pontos	avaliações	pontos	avaliações
Plain strings	207	2.927	207	2.927	207	2.927	207	2.927	207	2.927
Anchors	164	73.741	173	134.560	178	175.786	178	175.670	161	59.538
Ranges	196	110.870	196	189.124	196	209.356	196	177.632	195	95.608
Backrefs	168	130.513	168	130.513	168	129.946	168	129.946	168	130.513
Abba	187	72530	187	72530	187	72530	187	72530	187	72530
A man, a plan	129	85.908	132	129.831	133	141.840	132	129.831	128	83.164
Prime	142	88.404	142	96.639	142	109.076	142	96.639	142	81.193
Four	182	44.803	182	44.803	182	44.803	182	448.03	182	44.803
Order	190	31.959	190	31.959	190	31.959	190	31.959	190	31.959
Triples	471	59.499	486	130.887	486	130.533	487	128.443	467	31.611
Glob	368	139.838	371	183.068	376	176.101	375	186.454	366	123.920
Balance	237	6.584	237	6.584	237	6.584	237	6.584	237	6.584
Powers	4	37.803	5	42.693	5	40.069	5	41.364	5	56.289
Long count	212	1.064	212	1.064	212	1.064	212	1.064	212	1.064
Alphabetical	299	16.935	299	16.935	299	16.935	299	16.935	299	16.935
Total		**60.225**		**80.941**		**85.967**		**82.852**		**55.909**

Tabela 4.3: Resultado médio do número de avaliações necessárias até encontrar a melhor solução e pontuação da melhor solução encontrada

Observando a Tabela 4.3 podemos concluir que quanto mais avaliações são efetuadas, mais o espaço de busca é percorrido e melhores resultados podem ser encontrados. Como

o reinício do algoritmo pode ser feito de forma aleatória (se o compactador não gerar uma solução melhor) com soluções de tamanhos distintos (dependendo do parâmetro da profundidade da árvore), a execução configurada para 1 milhão de avaliações e limitada a 200 mil avaliações sem melhorias apresentou um desempenho inferior à execução configurada para 1 milhão de avaliações sem a limitação de avaliações sem melhorias. Todas as execuções de todas as instâncias terminaram em menos de cinco minutos, com a exceção da instância "Powers" que excedeu o limite de dez minutos em todas as configurações, exceto na configuração de 1 milhão de avaliações com 200 mil avaliações sem melhoria.

Com este resultado, concluímos que 1 milhão de avaliações sem limitação de avaliações sem melhoria é o melhor critério de parada para o algoritmo proposto. Definimos o *timeout* para dez minutos, tempo superior a todas as médias de tempo de execução, pois temos o objetivo de usar o número de avaliações como a condição primária de parada e o *timeout* como condição secundária para evitar longas execuções desnecessárias.

4.4 Comparação entre os resultados

Nesta seção comparamos os resultados do algoritmo de busca local proposto no Capítulo 3 com os resultados do algoritmo de programação genética criado por Bartoli et al. [4] e com os resultados do algoritmo exato criado por Peter Norvig [28]. Também comparamos com expressões regulares desenvolvidas por seres humanos.

Para gerar os resultados do algoritmo de busca local, executamos o algoritmo 30 vezes para cada instância selecionada com o parâmetro de profundidade inicial da árvore definido na Seção 4.3.1, com os critérios de parada definidos na Seção 4.3.2 e 30 sementes aleatórias diferentes. A Tabela 4.4 apresenta as melhores expressões regulares encontradas pelo algoritmo de busca local entre todas as execuções. A Tabela 4.5 apresenta os melhores resultados de todos os algoritmos e das expressões regulares desenvolvidas por humanos por instância.

Para obtermos os resultados do algoritmo exato, implementamos o algoritmo apresentado por Peter Norvig em seu *website* [28] e o executamos para cada instância selecionada. A implementação foi desenvolvida na linguagem de programação *Python*[1], a mesma utilizada pelo autor. O único problema na implementação de Peter Norvig foi de não realizar *escape* quando caracteres que são operadores da expressão estão presentes na *match list* ou na *unmatch list*. Este problema ocorreu apenas quando executamos a implementação

[1] https://python.org/

Nome da instância	Expressão regular
Plain strings	`foo`
Anchors	`k$`
Ranges	`^[a-f]*$`
Backrefs	`ho|ea|oc|ry|rp|te|tr|en|^[lm]|rou|.la*$`
Abba	`rit|rs|st|z|ph|.u|te*$`
A man, a plan	`mu|ic|ev|oo|x|^[^cims]*$`
Prime	`x{33}|^xx.?$`
Four	`lit|ev|de|vi|ara|o.o`
Order	`ce|ch|lo|^.[^ar]*$`
Triples	`0{9}|472|775|0[12]5|003|900|009|06|5[45]|2[34]|[01]2$`
Glob	`ro|rr|eat|lle|tl|de|co|gen|ow|fa|lo`
Balance	`>>><>><<|<<>><>>|<>{4}<<|<>{4}$`
Powers	`^..?$`
Long count	`0{4} 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 101`
Alphabetical	`ar t[ -e]|a t|r sn|rt r|t tes|esen|e .r`

Tabela 4.4: As melhores expressões regulares resultantes das execuções do algoritmo de busca local para cada instância

do algoritmo exato com a instância "Glob". Após este problema ter sido corrigido, obtivemos todas as expressões regulares encontradas por Peter Norvig para todas as instâncias.

Os resultados do algoritmo de programação genética foram obtidos do trabalho de Bartoli et al. [4], que também apresenta os resultados das expressões regulares desenvolvidas por seres humanos. Porém, estas expressões regulares poderiam estar obsoletas uma vez que se passaram alguns anos desde a publicação do trabalho. Para encontrar as expressões regulares desenvolvidas por seres humanos mais atuais, realizamos uma pesquisa no *google*[2] utilizando as palavras-chaves *"regex golf answers"* e encontramos muitas expressões regulares em diversos *websites*. A Tabela 4.6 apresenta as melhores expressões regulares desenvolvidas por diversos seres humanos para as instâncias selecionadas.

Observamos que as expressões regulares encontradas por seres humanos representam sempre o melhor resultado, restando aos algoritmos competir entre si para ver qual mais se aproxima destas soluções. Os três algoritmos empatam em "Plain strings" e "Anchors" por serem as instâncias que possuem as menores soluções entre as instâncias seleciona-

[2]http://google.com

Nome da instância	Norvig-exato	Bartoli-PG	Busca local
Plain strings	**207**	**207**	**207**
Anchors	**208**	**208**	**208**
Ranges	191	195	**202**
Backrefs	**175**	138	171
Abba	186	184	**188**
A man, a plan	157	136	**166**
Prime	-398	**188**	153
Four	**192**	183	190
Order	190	186	**192**
Triples	**589**	430	577
Glob	**392**	340	385
Balance	-1457	130	**249**
Powers	-1969	**51**	15
Long count	189	191	**212**
Alphabetical	294	132	**301**

Tabela 4.5: Os resultados do algoritmo exato (Norvig-exato), do algoritmo de programação genética (Bartoli-PG) e de 30 execuções do algoritmo de busca local

das. O algoritmo de busca local conseguiu o melhor resultado entre os três algoritmos nas instâncias “Abba”, “A man, a plan”, “Ranges”, “Order” e “Alphabetical”, além das instâncias “Long count” e “Balance”, que são instâncias consideradas difíceis. Na instância “Ranges”, o algoritmo de busca local conseguiu encontrar uma solução compatível com a melhor solução desenvolvida por humanos. O algoritmo exato obteve os melhores resultados nas instâncias “Backrefs”, “Four”, “Triples” e “Glob” e o algoritmo de programação genética obteve os melhores resultados nas instâncias “Powers” e “Prime”, ambas consideradas instâncias difíceis. A Figura 4.1 apresenta os resultados de forma gráfica.

Na maioria dos resultados dos três algoritmos, o operador da expressão regular mais utilizado é o *alternation*, por proporcionar uma estratégia de “dividir para conquistar”. Uma vez que é difícil conseguir uma expressão regular enxuta que realize os *matches* e *unmatches* necessários nas listas, torna-se mais fácil criar expressões regulares com poucos caracteres com o foco em realizar alguns *matches* na *match list* e nenhum *match* na *unmatch list*, utilizando o operador *alternation* para concatena-las em uma expressão regular. Peter Norvig [28] chama essas pequenas expressões regulares de *n-grams*. Esta é uma forma eficiente para encontrar uma solução para o *Regex Golf*. Por isso, embora os

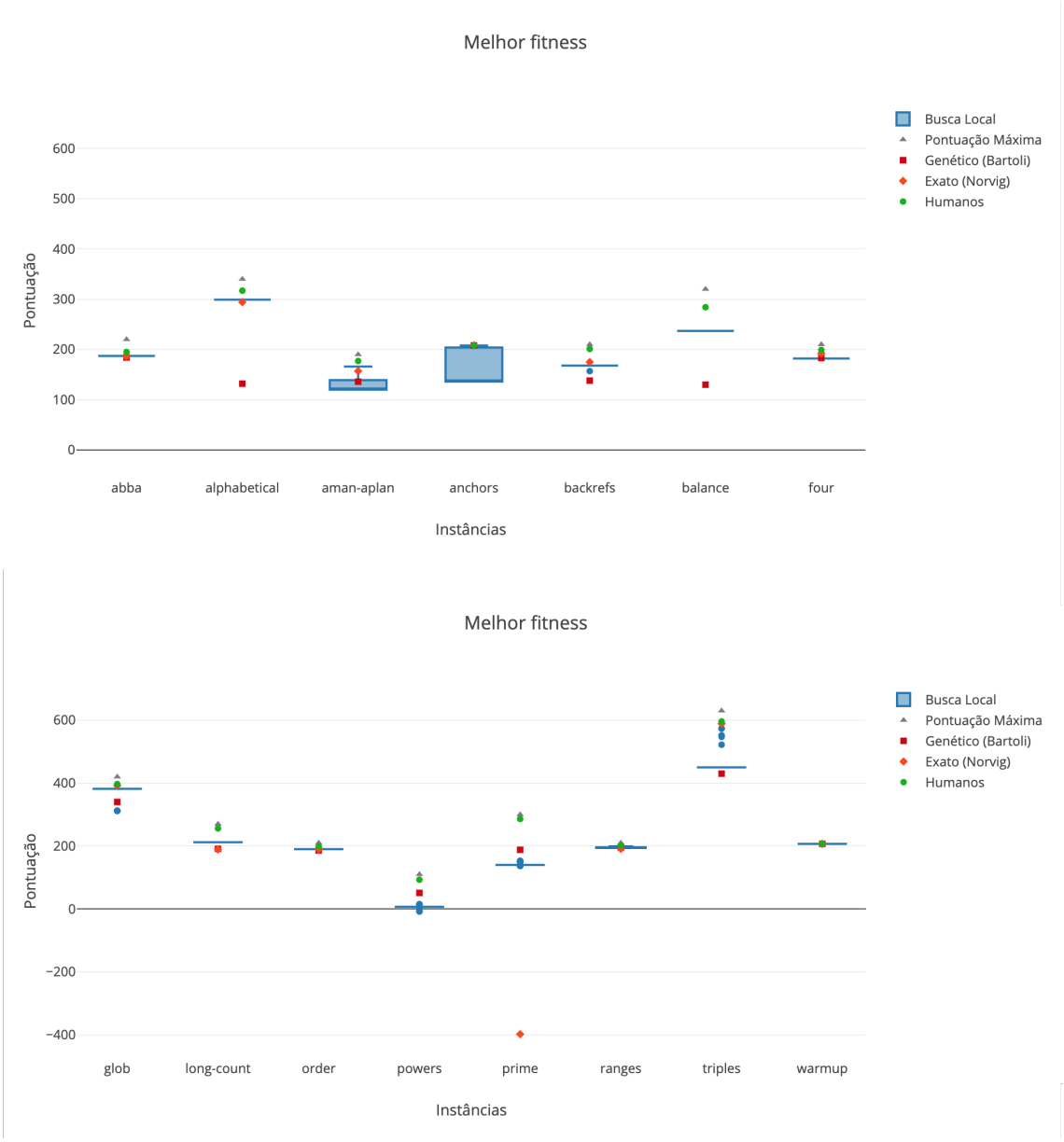

Figura 4.1: Bloxplot comparando os resultados do algoritmo de busca loca, do algoritmo de programação genética e do algoritmo exato

algoritmos heurísticos não sigam esta estratégia, muitos dos resultados destes algoritmos se assemelham ao resultado do algoritmo exato.

O algoritmo de busca local tende a apresentar melhor desempenho quando existe uma variedade de caracteres diferentes em ambas *match list* e *unmatch list*. Instâncias com poucos caracteres únicos, como as instâncias “Prime”, “Powers” e “Long count”, apresentam desempenho ruim por não conseguirem gerar vizinhanças boas e por dependerem das listas de caracteres derivadas das *match list* e *unmatch list*. A busca local só conseguiu obter um bom resultado na instância “Long count” porque o algoritmo compactador de expressões regulares foi executado na única frase da *match list*, que foi utilizada quando o

Nome da instância	Expressão regular	Pontuação
Plain strings	`foo`	207
Anchors	`k$`	208
Ranges	`^[a-f]*$`	202
Backrefs	`(...).*\1`	201
Abba	`^(?!.*(.)\1)|ef`	195
A man, a plan	`^(.)[^p].*\1$`	177
Prime	`^(?!(..+)\1+$)`	286
Four	`(.)(.\1){3}`	199
Order	`^.{5}[^e]?$`	199
Triples	`00($|3|6|9|12|15)|4.2|.1.+4|55|.17`	596
Glob	`ai|c$|^p|[bcnrw][bnopr]`	397
Balance	`.{37}|^(<(..(?!<.>$))*>)*$`	294
Powers	`^(?!(.(..)+)\1*$)`	93
Long count	`((.+)0\2+1){8}`	256
Alphabetical	`.r.{32}r|a.{10}te|n.n..`	317

Tabela 4.6: As melhores expressões regulares desenvolvidas por humanos

algoritmo precisou ser reiniciado. Nas instâncias “Balance” e “Powers” ocorreram muitos *timeouts* devido a alguns itens da *match list* possuírem mais de 100 caracteres, o que acarreta no parâmetro da profundidade da árvore ser muito elevado e faz com que cada rodada da vizinhança seja desnecessariamente longa, uma vez que muitos vizinhos não conseguirão ser melhores que a solução atual.

O algoritmo compactador de expressões regulares foi um grande diferencial nas instâncias que possuem frases muito grandes na *match list*, como as instâncias “Prime”, “Triples”, “Powers” e “Long count”. Nestas instâncias, apenas a execução do algoritmo de busca local com reinício aleatório não seria suficiente para gerar as expressões regulares obtidas ao final como as suas melhores soluções, pois este algoritmo não é capaz de gerar alguns dos nós que favoreceram a busca, como, por exemplo, o nó função do tipo *quantifier* `0{9}` gerado para a instância “Triples”.

4.4.1 Comparação com o algoritmo de programação genética

O algoritmo de busca local vence o algoritmo de programação genética em onze instâncias, perde em duas e empata em duas instâncias. Nas instâncias “Ranges” e “Long

count", a busca local obteve melhores resultados por conseguir gerar soluções com os operadores *quantifier* derivados do algoritmo compactador de expressões regulares. Ou seja, as expressões regulares resultantes em ambos algoritmos foram muito próximas para estas instâncias, mas o algoritmo compactador de expressões regulares conseguiu remover alguns caracteres, levando o algoritmo de busca local à vitória por poucos pontos de diferença.

Nas instâncias "Four", "Backrefs", "Triples" e "Glob" ambos algoritmos apresentaram soluções que usaram muitas vezes o operador *alternation*. Porém, as expressões regulares da programação genética apresentam muitos caracteres redundantes ou repetidos desnecessariamente. Por exemplo, o algoritmo genético gerou soluções que apresentam o mesmo *n-gram* duas vezes ou um operador *group* que não melhora a capacidade de *match* da expressão regular. Estas redundâncias não aparecem na busca local porque o algoritmo compactador remove os operadores em excesso, aprimorando a solução.

Diferente da busca local, o algoritmo de programação genética consegue percorrer diferentes pontos do espaço de busca encontrando expressões regulares bastantes diferentes. Isto mostra que o algoritmo de programação genética utiliza melhor os operadores de *backref*, *lookahead* e *lookbehind*, apresentando uma vantagem nas instâncias em que existem poucos caracteres únicos, como as instâncias "Prime" e "Powers". Porém, as expressões regulares geradas para compor a população possuem poucos operadores de expressão regular, o que gera poucas variações e limita o percurso do espaço de busca.

Outra vantagem do algoritmo de busca local sobre o algoritmo de programação genética é o tempo de execução. Em seu trabalho, Bartoli et al. [4] informa que o tempo de execução do algoritmo de programação genética ultrapassa os 60 minutos em quase todas as instâncias, enquanto o algoritmo de busca local tem o seu tempo médio de execução abaixo dos cinco minutos, com exceção da instância "Powers" que ultrapassa dez minutos em todas as execuções.

4.4.2 Comparação com o algoritmo exato

O algoritmo exato tem em sua configuração gerar *n-grams* de tamanho máximo de quatro caracteres. Isto limitou a capacidade de busca do algoritmo ao ponto do resultado ser representado por todos os itens da *match list* concatenados com operadores *alternation* nas instâncias que precisam de expressões regulares grandes para conseguirem um bom *match* ou *unmatch*. Isto gera expressões regulares muito grandes, resultando muitas vezes em pontuação negativa.

Como explicado na Seção 2.2, o algoritmo exato tem a característica de sempre gerar uma expressão regular capaz de vencer o desafio. Porém, a expressão regular resultante não é penalizada pela quantidade de caracteres, apenas pela quantidade de *matches* e *unmatches* realizados. Isso acarreta em expressões regulares muito grandes que podem influenciar na pontuação. Como o algoritmo exato não gera expressões regulares que possuam operadores *quantifier*, a pontuação nas instâncias "Prime", "Balance" e "Powers", que são instâncias que precisam desses operadores para apresentar soluções mais robustas, ficam abaixo de zero por gerar uma solução contendo conjuntos de operadores *alternation* com caracteres ou mesmo palavras completas da *match list*, resultando em soluções grandes que, mesmo realizando *match* em todos os itens da *match list* e nenhum *match* na *unmatch list*, tem pontuação prejudicada pelo seu tamanho.

Essas pontuações ruins fizeram com que o algoritmo de busca local vencesse o algoritmo exato em sete instâncias, empatasse em três e perdesse em cinco instâncias. Nas instâncias "Backrefs" e "Four", a busca local perdeu por apenas seis e dois pontos de diferença, respectivamente. Isto indica que ainda é possível melhorar o algoritmo de vizinhança para que ele possa se movimentar até uma solução com dois caracteres a menos.

Não é possível comparar os tempos de execuções dos algoritmos uma vez que a busca local mais rápida levou 28 segundos para realizar um milhão de avaliações, enquanto o algoritmo exato realiza uma quantidade de avaliações inferior a qualquer vizinhança avaliada pela busca local.

4.4.3 Comparação com expressões regulares desenvolvidas por humanos

As expressões regulares desenvolvidas por seres humanos foram criadas e evoluídas por anos nos *websites*[3] em que foram encontradas. Não é possível determinar o nível de conhecimento das pessoas que participaram do desenvolvimento das expressões regulares, mas o resultado em todas as instâncias é bastante próximo da pontuação máxima possível. Comparar os resultados do algoritmo de busca local com estas expressões regulares nos mostra o quão próximo ou distante o algoritmo de busca local está de alcançar os melhores resultados conhecidos.

O algoritmo de busca local encontra as mesmas expressões regulares desenvolvidas por humanos nas instâncias "Plain string", "Anchors" e "Ranges". Porém, estas são as únicas instâncias em que os humanos não vencem a busca local. Em todas as outras instâncias, a diferença na pontuação não chega a ser muito grande, mas a quantidade de caracteres das expressões regulares desenvolvidas por pessoas pode ser menor que a

[3] https://gist.github.com/jpsim/8057500 e https://gist.github.com/Davidebyzero/9221685

metade da encontrada pela busca local, como nos casos das instâncias “Backrefs”, “Long count” e “A man, a plan”. Essas expressões regulares são complexas a ponto de não ser possível gera-las pelo algoritmo de vizinhança, que não possui transformações para gerar operadores da expressão regular aninhados a outros operadores de forma concisa, sendo esta a principal limitação do algoritmo de busca local.

4.5 Problemas encontrados

Durante os experimentos do algoritmo de busca local ocorreram alguns problemas. Muitos dos problemas encontrados foram relacionados as transformações realizadas na solução atual para gerar sua vizinhança. Algumas dessas transformações geravam muitas expressões regulares inválidas ou mesmo repetidas, gerando um “ruído” que atrapalhava tanto na métrica de avaliações necessárias para encontrar a melhor solução possível quanto no tempo de execução do algoritmo.

Para reduzir a quantidade de expressões regulares inválidas geradas, foi desenvolvida uma ferramenta que, dada uma expressão regular, apresenta todas as expressão regulares geradas por cada uma das transfomações. Desta forma, foi possível corrigir a implementação das transformações responsáveis pela geração da vizinhança e reduzir em mais de 90% a quantidade de expressões regulares geradas incorretamente e em 100% a quantidade de expressões regulares geradas mais de uma vez na mesma vizinhança. O código desta ferramenta se encontra no repositório do algoritmo de busca local[4].

Um dos problemas encontrados de mais difícil correção foi na função objetivo do algoritmo. Quando a solução atual empatava com a solução anterior, gerava um *loop* infinito saltando exclusivamente entre as duas soluções até acabar o orçamento de avaliações. Percebemos este problema quando executamos os testes nas instâncias que geram uma *match list* pequena, de um ou dois caracteres apenas, como nas instâncias “Long count’ e “Prime”, o que acarreta em vizinhanças relativamente pequenas e espaço de busca reduzido. Este problema foi resolvido adicionando uma lista de soluções já visitadas e não permitindo que o algoritmo salte mais de uma vez para a mesma solução.

Outro problema encontrado durante os experimentos foi que algumas expressões regulares geradas pela busca local tinham um tempo de execução muito elevado devido à quantidade de verificações realizadas pelos operadores *quantifiers* quando dois ou mais estão presentes na expressão regular. Este problema é conhecido como *catastrophic backtracking* [8] e as implementações das *engines* de expressão regular atuais não são progra-

[4] https://github.com/andrefarzat/regex-golf

madas para tratar este problema. Isto causava um impacto no tempo de execução em instâncias onde eram gerados muitos indivíduos com operadores *quantifiers*. As instâncias "Balance", "Prime" e "Powers" são exemplo destes problemas.

A primeira forma que encontramos para resolver este problema foi trocar o uso dos operadores *quantifiers* para os operadores *possessive quantifiers*, que não causariam *catastrophic backtracking*. Porém, essa solução tem duas desvantagens: os operadores *possessive quantifiers* possuem um caractere a mais do que os operadores *quantifiers* e a diferença de *matches* entre os operadores resultou em uma piora significativa nas expressões regulares resultantes. A solução encontrada foi implementar um *timeout* para o cálculo da função objetivo. Se o tempo de avaliar uma expressão regular for superior a 10 ms, a expressão regular é considerada inválida e é descartada da vizinhança.

Por fim, o algoritmo compactador de expressões regulares, em alguns poucos casos, gerava uma expressão regular com um caractere que não pertencia a nenhuma das listas de caracteres utilizadas para gerar as soluções vizinhas da solução atual. Isto "poluía" a solução, acarretando em uma vizinhança ruim que atrapalhava o desempenho do algoritmo. A solução foi garantir que a expressão regular gerada pelo algoritmo compactador de expressões regulares possua a mesma quantidade de *matches* na *match list* e *unmatches* na *unmatch list*. Caso isso não ocorra, a expressão regular é considerada inválida e é gerada uma expressão regular aleatória no seu lugar. Isso garante que a expressão regular resultante seja confiável para o reinício do algoritmo de busca local.

4.6 Considerações finais

Neste capítulo foi apresentado um estudo experimental que comparou o algoritmo de busca local com o algoritmo exato e com o algoritmo de programação genética em suas eficácias em gerar boas expressões regulares para o desafio *Regex Golf*. O estudo também ajudou a descobrir problemas de lógica na implementação do algoritmo de busca local e do algoritmo compactador de expressões regulares. Ele também foi útil para calibrar o parâmetro de profundidade da árvore e os critérios de parada do algoritmo.

Ao final do estudo experimental, pudemos observar que o algoritmo de busca local é um bom candidato ao desafio *Regex Golf*, encontrando as mesmas soluções que os outros algoritmos em duas instâncias selecionadas e soluções melhores que os outros algoritmos em sete das quinze instâncias selecionadas em uma fração do tempo de execução do algoritmo de programação genética.

5. Conclusão

5.1 Considerações finais

Nesta Dissertação foi apresentada uma nova técnica heurística de busca com o objetivo de gerar expressões regulares para um problema proposto no desafio *Regex Golf*. Esta nova heurística é um algoritmo de busca local atrelado a um algoritmo compactador de expressões regulares apresentados no Capítulo 3. O desafio *Regex Golf* foi escolhido por sua ênfase em encontrar a menor expressão regular possível para um problema.

O objetivo principal desta nova técnica heurística é colaborar com desenvolvimento de softwares de extração de dados baseada em exemplos e de softwares que utilizam expressões regulares para validação de dados. Normalmente, os algoritmos de geração automática de expressões regulares possuem o foco na eficácia, ou seja, no número de *matches* realizados, e não priorizam a eficiência da expressão regular. Como vimos no Capitulo 2, a maioria dos algoritmos utilizam técnicas simples, com poucos operadores de expressão regular, para compor as soluções. Isso leva a expressões regulares com grande número de caracteres, o que pode demandar um tempo de processamento elevado na busca dos padrões em textos longos.

Um estudo experimental foi apresentado no Capítulo 4. Foram selecionadas quinze instâncias do problema e o algoritmo de busca local foi executado 30 vezes para cada instância. Este experimento comparou os resultados gerados pelo algoritmo de busca local com os outros dois algoritmos desenvolvidos para o desafio *Regex Golf*: um algoritmo exato e um algoritmo de programação genética. O algoritmo de busca local encontrou soluções melhores na maioria das instâncias selecionadas em um tempo menor que o algoritmo de programação genética.

5.2 Contribuições

As principais contribuições desta Dissertação são:

- A definição e implementação de uma heurística para a geração automática de expressões regulares para o desafio *Regex Golf*;
- Um algoritmo compactador que mantém a capacidade de *match* de expressões regulares;
- Um ferramental para execução em escala do algoritmo de busca local e do algoritmo exato criados para o *Regex Golf*;
- O planejamento, execução e análise dos resultados de um estudo experimental para avaliar o algoritmo proposto.

O algoritmo de busca local e o algoritmo compactador foram implementados na linguagem TypeScript e estão disponíveis publicamente em um repositório no GitHub[1].

5.3 Resposta para a questão de pesquisa

Após a análise dos resultados do estudo experimental apresentado no Capítulo 4, é possível responder a questão de pesquisa proposta nesta Dissertação:

RQ1: O algoritmo de busca local proposto encontra melhores resultados em menor tempo de execução quando comparado a um algoritmo genético e um algoritmo exato?

O algoritmo de busca local encontrou os melhores resultados em sete das quinze instâncias selecionadas com tempo de execução inferior a 10 minutos, com exceção de uma instância que atinge 10 minutos de execução. O algoritmo de programação genética obteve os melhores resultados em duas instâncias e seu tempo médio de execução excede 60 minutos. O algoritmo exato apresenta o melhor desempenho no critério tempo, tendo o tempo máximo de execução de cinco segundos, e obteve os melhores resultados em quatro instâncias. Por fim, os três algoritmos encontraram os mesmos resultados em duas das quinze instâncias.

Na comparação entre o algoritmo de busca local e o algoritmo de programação genética, o algoritmo de busca local encontrou melhores resultados em onze instâncias, o

[1] `http://github.com/andrefarzat/regex-golf`

algoritmo de programação genética encontrou melhores resultados em duas e os algoritmos empataram em duas das quinze instâncias.

Na comparação entre o algoritmo de busca local e o algoritmo exato, o algoritmo de busca local encontrou melhores resultados em nove instâncias, o algoritmo exato encontrou melhores resultados em quatro e empataram em duas das quinze instâncias.

Assim, concluímos que o algoritmo de busca local obtém os melhores resultados na maioria das instâncias do problema em um tempo superior ao do algoritmo exato e inferior ao do algoritmo de programação genética.

5.4 Limitações

A principal limitação é que a técnica heurística foi observada em um pequeno conjunto de instâncias, sendo estas as instâncias que possuem resultados para o algoritmo de programação genética e algoritmo exato extraídas do trabalho [4]. No *website*[2] do desafio *Regex Golf* existe um número maior de instâncias, porém não haveria como comparar com os outros algoritmos por falta de resultados. Seria possível comparação apenas com resultados obtidos desenvolvidas por humanos.

O uso do padrão ECMAScript para os operadores da expressão regular também é uma limitação, uma vez que é possível que a expressão gerada pelo algoritmo não seja compatível com a máquina virtual em que ela será utilizada.

5.5 Perspectivas futuras de trabalho

Sempre é possível continuar a evoluir um trabalho de pesquisa e a criação de uma técnica heurística de busca não é exceção. Algumas sugestões de possíveis evoluções deste trabalho incluem:

- A criação de uma interface *web* ou *desktop* para o algoritmo de busca local;
- O algoritmo compactador de expressões regulares pode ser incrementado com mais operadores de compressão;
- Realização de um estudo experimental para avaliar instâncias com *match list* que possuam frases muito grandes, como a instância “Powers”;

[2] https://alf.nu/RegexGolf

- Adicionar operadores de expressão regular ainda não utilizados, como *lookahead*, *negative lookahead*, *lookbehind* e *negative lookbehind*, para gerar mais soluções na vizinhança.

Referências Bibliográficas

[1] Emile Aarts, Emile HL Aarts, and Jan Karel Lenstra. *Local search in combinatorial optimization*. Princeton University Press, 2003.

[2] Ethem Alpaydin. *Introduction to machine learning*. MIT press, 2009.

[3] Alberto Bartoli, Giorgio Davanzo, Andrea De Lorenzo, Marco Mauri, Eric Medvet, and Enrico Sorio. Automatic generation of regular expressions from examples with genetic programming. In *Proceedings of the 14th annual conference companion on Genetic and evolutionary computation*, pages 1477–1478. ACM, 2012.

[4] Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, and Fabiano Tarlao. Playing regex golf with genetic programming. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, pages 1063–1070. ACM, 2014.

[5] Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, and Fabiano Tarlao. Active learning approaches for learning regular expressions with genetic programming. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pages 97–102. ACM, 2016.

[6] Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, and Fabiano Tarlao. Can a machine replace humans in building regular expressions? a case study. *IEEE Intelligent Systems*, 31(6):15–21, 2016.

[7] Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, and Fabiano Tarlao. Inference of regular expressions for text extraction from examples. *IEEE Transactions on Knowledge and Data Engineering*, 28(5):1217–1230, 2016.

[8] Martin Berglund, Frank Drewes, and Brink Van Der Merwe. Analyzing catastrophic backtracking behavior in practical regular expression matching. *arXiv preprint arXiv:1405.5599*, 2014.

[9] Falk Brauer, Robert Rieger, Adrian Mocan, and Wojciech M Barczynski. Enabling information extraction by inference of regular expressions from sample entities. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 1285–1294. ACM, 2011.

[10] Eric Brill. Transformation-based error-driven learning and natural language processing: A case study in part-of-speech tagging. *Computational linguistics*, 21(4):543–565, 1995.

[11] Ahmet Cetinkaya. Regular expression generation through grammatical evolution. In *Proceedings of the 9th annual conference companion on Genetic and evolutionary computation*, pages 2643–2646. ACM, 2007.

[12] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[13] Márcio de Oliveira Barros Fabio de Almeida Farzat, Guilherme Horta Travassos. *Otimização para Reduzir o Tamanho de Código-Fonte Javascript*. PhD thesis, Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, Universidade Federal do Rio de Janeiro, 2018.

[14] Jeffrey EF Friedl. *Mastering Regular Expressions: Understand Your Data and Be More Productive*. "O'Reilly Media, Inc.", 2006.

[15] Antonio Gonzalez-Pardo and David Camacho. Analysis of grammatical evolutionary approaches to regular expression induction. In *2011 IEEE Congress of Evolutionary Computation (CEC)*, pages 639–646. IEEE, 2011.

[16] Mark Harman. The current state and future of search based software engineering. In *2007 Future of Software Engineering*, pages 342–357. IEEE Computer Society, 2007.

[17] Randy L Haupt and Sue Ellen Haupt. Practical genetic algorithms. *Wiley Online Library*, 2004.

[18] Stephen Cole Kleene. Representation of events in nerve nets and finite automata. Technical report, RAND PROJECT AIR FORCE SANTA MONICA CA, 1951.

[19] John R Koza. Genetic evolution and co-evolution of computer programs. *Artificial life II*, 10:603–629, 1991.

[20] John R Koza, Forrest H Bennett, David Andre, and Martin A Keane. Automated design of both the topology and sizing of analog electrical circuits using genetic programming. In *Artificial Intelligence in Design'96*, pages 151–170. Springer, 1996.

[21] John R Koza and John R Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

[22] William H Kruskal and W Allen Wallis. Use of ranks in one-criterion variance analysis. *Journal of the American statistical Association*, 47(260):583–621, 1952.

[23] Eric Larson and Anna Kirk. Generating evil test strings for regular expressions. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 309–319. IEEE, 2016.

[24] Yunyao Li, Rajasekar Krishnamurthy, Sriram Raghavan, Shivakumar Vaithyanathan, and HV Jagadish. Regular expression learning for information extraction. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 21–30. Association for Computational Linguistics, 2008.

[25] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

[26] Patrick E McKight and Julius Najab. Kruskal-wallis test. *The corsini encyclopedia of psychology*, pages 1–1, 2010.

[27] Mark EJ Newman. Power laws, pareto distributions and zipf's law. *Contemporary physics*, 46(5):323–351, 2005.

[28] Peter Norvig. Regex golf with peter norvig. `https://www.oreilly.com/learning/regex-golf-with-peter-norvig`, 2014. Visitado em: 26-05-2018.

[29] Michael O'Neill and Conor Ryan. Under the hood of grammatical evolution. In *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation-Volume 2*, pages 1143–1148. Morgan Kaufmann Publishers Inc., 1999.

[30] Alfonso Ortega, Marina De La Cruz, and Manuel Alfonseca. Christiansen grammar evolution: grammatical evolution with semantics. *IEEE Transactions on Evolutionary Computation*, 11(1):77–90, 2007.

[31] Michael O'Neil and Conor Ryan. Grammatical evolution: Evolutionary automatic programming in an arbitrary language. *Norwell, MA: Kluwer Academic Publishers. doi*, 10:978–1, 2003.

[32] Michael O'Neill and Conor Ryan. Evolving multi-line compilable c programs. In *European Conference on Genetic Programming*, pages 83–92. Springer, 1999.

[33] Judea Pearl. Heuristics: intelligent search strategies for computer problem solving. *Web*, 1984.

[34] Duc Pham and Dervis Karaboga. *Intelligent optimisation techniques: genetic algorithms, tabu search, simulated annealing and neural networks*. Springer Science & Business Media, 2012.

[35] Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe & efficient gradual typing for typescript. In *ACM SIGPLAN Notices*, volume 50, pages 167–180. ACM, 2015.

[36] Raghav Rastogi, S Akash, G Shobha, G Poonam, D Pratiba, and Ankit Singh. Design and development of generic web based framework for log analysis. In *2016 IEEE Region 10 Conference (TENCON)*, pages 232–236. IEEE, 2016.

[37] Jeffrey C Reynar and Adwait Ratnaparkhi. A maximum entropy approach to identifying sentence boundaries. In *Proceedings of the fifth conference on Applied natural language processing*, pages 16–19. Association for Computational Linguistics, 1997.

[38] Conor Ryan, John James Collins, and Michael O Neill. Grammatical evolution: Evolving programs for an arbitrary language. In *European Conference on Genetic Programming*, pages 83–96. Springer, 1998.

[39] Yutaka Sasaki et al. The truth of the f-measure. *Teach Tutor mater*, 1(5):1–5, 2007.

[40] Muzammil Shahbaz, Phil McMinn, and Mark Stevenson. Automatic generation of valid and invalid test data for string validation routines using web searches and regular expressions. *Science of Computer Programming*, 97:405–425, 2015.

[41] Mark G Sobell. *A practical guide to Linux commands, editors, and shell programming*. Pearson Education, 2009.

[42] A. Tauber. Exrex - regular expression string generator. `https://github.com/asciimoo/exrex`, 2012. Visitado em: 18-05-2019.

[43] Paulo Urbano and Loukas Georgiou. Improving grammatical evolution in santa fe trail using novelty search. In *Artificial Life Conference Proceedings 13*, pages 917–924. MIT Press, 2013.

[44] Stefan Voß. Meta-heuristics: The state of the art. In *Workshop on Local Search for Planning and Scheduling*, pages 1–23. Springer, 2000.

[45] Stefan Voß, Silvano Martello, Ibrahim H Osman, and Catherine Roucairol. *Meta-heuristics: Advances and trends in local search paradigms for optimization*. Springer Science & Business Media, 2012.

[46] Tianhao Wu and William M Pottenger. A semi-supervised active learning algorithm for information extraction from textual data. *Journal of the Association for Information Science and Technology*, 56(3):258–271, 2005.

[47] Junjie Zhang, Christian Seifert, Jack W Stokes, and Wenke Lee. Arrow: Generating signatures to detect drive-by downloads. In *Proceedings of the 20th international conference on World wide web*, pages 187–196. ACM, 2011.