UNIVERSIDADE FEDERAL DE SANTA CATARINA
CAMPUS REITOR JOÃO DAVID FERREIRA LIMA
PROGRAMA DE GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO

Augusto Silva de Oliveira

SACI - UMA FERRAMENTA PARA EXTRAÇÃO DE ALGORITMOS E SUAS COMPLEXIDADES DE TEMPO E ESPAÇO

Florianópolis, Santa Catarina – Brasil
2023

Augusto Silva de Oliveira

SACI - UMA FERRAMENTA PARA EXTRAÇÃO DE ALGORITMOS E SUAS COMPLEXIDADES DE TEMPO E ESPAÇO

Trabalho de Conclusão de Curso submetido ao Programa de Graduação em Sistemas de Informação da Universidade Federal de Santa Catarina para a obtenção do Grau de Bacharel em Sistemas de Informação.
Orientador(a): Carina Friedrich Dorneles, Dr.
Coorientador(a): Ricardo José Pfitscher, Dr.

Florianópolis, Santa Catarina – Brasil
2023

Catalogação na fonte pela Biblioteca Universitária da Universidade Federal de Santa Catarina.
Arquivo compilado às 00:36h do dia 8 de julho de 2024.

Augusto Silva de Oliveira
SACI - Uma ferramenta para extração de algoritmos e suas complexidades de tempo e espaço / Augusto Silva de Oliveira; Orientador(a), Carina Friedrich Dorneles, Dr.; Coorientador(a), Ricardo José Pfitscher, Dr. – Florianópolis, Santa Catarina – Brasil, 05 de Julho de 2024.
148 p.

Trabalho de Conclusão de Curso – Universidade Federal de Santa Catarina, INE – Departamento de Informática e Estatística, CTC – Centro Tecnológico, Programa de Graduação em Sistemas de Informação.

Inclui referências

1. Scraper, 2. Dataset, 3. Complexidade, I. Carina Friedrich Dorneles, Dr. II. Ricardo José Pfitscher, Dr. III. Programa de Graduação em Sistemas de Informação IV. SACI - Uma ferramenta para extração de algoritmos e suas complexidades de tempo e espaço

CDU 02:141:005.7

Augusto Silva de Oliveira

SACI - UMA FERRAMENTA PARA EXTRAÇÃO DE ALGORITMOS E SUAS COMPLEXIDADES DE TEMPO E ESPAÇO

Este(a) Trabalho de Conclusão de Curso foi julgado adequado(a) para obtenção do Título de Bacharel em Sistemas de Informação, e foi aprovado em sua forma final pelo Programa de Graduação em Sistemas de Informação do INE – Departamento de Informática e Estatística, CTC – Centro Tecnológico da Universidade Federal de Santa Catarina.

Florianópolis, Santa Catarina – Brasil, 05 de Julho de 2024.

Álvaro Junio Pereira Franco, Dr.
Coordenador(a) do Programa de Graduação em Sistemas de Informação

Banca Examinadora:

Carina Friedrich Dorneles, Dr.
Orientador(a)
Universidade Federal de Santa Catarina – UFSC

Ricardo José Pfitscher, Dr.
Coorientador(a)
Universidade Federal de Santa Catarina – UFSC

Roberto Willrich, Dr.
Universidade Federal de Santa Catarina – UFSC

Alexandre Gonçalves da Silva, Dr.
Universidade Federal de Santa Catarina – UFSC

AGRADECIMENTOS

Primeiramente, gostaria de expressar minha profunda gratidão à minha orientadora, Carina Friedrich Dorneles, por sua orientação, paciência e apoio contínuos ao longo deste trabalho. Sua expertise, conselhos valiosos e incentivo constante foram fundamentais para a realização deste projeto.

Agradeço também ao meu coorientador, Ricardo José Pfitscher, por seu suporte e contribuições significativas. Suas sugestões e feedbacks foram essenciais para o desenvolvimento deste trabalho.

À minha família, expresso minha mais sincera gratidão pelo amor incondicional, paciência e encorajamento durante toda a minha jornada acadêmica.

Aos meus amigos, que estiveram ao meu lado nos momentos de dificuldades e de conquistas, meu muito obrigado. Obrigado por todas as conversas, risadas. Vocês tornaram essa jornada mais leve e divertida.

RESUMO

Determinar a complexidade de tempo de execução e de espaço em um código escrito é uma tarefa que pode ser complexa e que muitas vezes é essencial, considerando-se que deseje-se obter uma aplicação eficiente. Apesar de ser um aspecto importante na formação do desenvolvedor, a análise de algoritmos quanto a sua complexidade é uma tarefa difícil que por vezes é negligenciada. Algumas ferramentas são capazes de fazer sugestões para o preenchimento automático do código conforme o programador esteja escrevendo, mas, até o momento atual, ela infelizmente não possui capacidade de considerar as complexidades em suas sugestões. Por isso é necessário que o desenvolvedor tenha conhecimento sólido sobre análise de algoritmos para garantir que as soluções propostas sejam eficientes. Tendo isso em vista, esse trabalho propõe a criação de uma ferramenta de extração de dados, capaz de extrair códigos de algoritmos juntos de suas respectivas complexidades a fim de formar uma base de dados. Essa base de dados poderá ser usada para o treinamento de ferramentas, seja através de aprendizado de máquina ou outras técnicas, que visam determinar a complexidade de um trecho de código e auxiliar os desenvolvedores em suas rotinas. Ao fim do trabalho obteve-se uma base de dados com 501 algoritmos distintos acompanhados de suas complexidades, com boa precisão e qualidade dos dados, comprovado pelas métricas utilizadas.

Palavras-chaves: Scraper. Dataset. Complexidade.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE CÓDIGOS

LISTA DE ALGORITMOS

SUMÁRIO

1 INTRODUÇÃO

A complexidade de algoritmos mede o consumo de recursos computacionais, como tempo de execução e o espaço ocupado em memória, necessários para execução de um programa ou algoritmo em relação ao tamanho da entrada; ou seja, é a medida de eficiência desse programa (CORMEN *et al.*, 2009). Trata-se de um tema muito importante durante o ciclo de desenvolvimento de uma aplicação porque está diretamente atrelado à forma como será seu desempenho. Uma vez conhecendo a complexidade de certo trecho de código, pode-se prever sua eficiência e otimizá-lo (SEDGEWICK, 2009); isso garante o melhor uso possível dos recursos computacionais, visto que os mesmos são finitos. Infelizmente, por falta de hábito, prática ou conhecimento, a tarefa de identificar a complexidade acaba sendo relegada ou pode tornar-se uma tarefa difícil para o programador, embora existam diversas variáveis que norteiam essa identificação, como tempo de execução, tamanho de entrada, espaço ocupado na memória, análises de melhor e pior caso, dentre outras (HALL; CHAPMAN, 2015). Considerar tudo isso pode ser demasiado complexo; então, por que não usar uma ferramenta que auxilie nessa tarefa?

A carência de dados nesse âmbito pode ser corroborada com uma breve pesquisa sobre o tema 'base de dados de algoritmos e complexidades', a qual retorna poucos resultados. Sikka et al. (SIKKA *et al.*, 2020a) trabalham com um banco de dados restrito, composto por 931 códigos em Java. O baixo número de amostras utilizado prejudica não somente o desempenho dos métodos de classificação, mas principalmente seu poder de generalização; além disso, o banco de dados não contempla algoritmos de ordem superior a On^2. A baixa amostragem de classes ineficientes dificulta a separação binária de eficiência do classificador, que é mais relevante do que a separação entre várias classes eficientes. Pfitscher et al. (PFITSCHER *et al.*, 2023) ampliam o total de códigos do banco de dados para 1325 amostras a partir de raspagem de dados por web scraping do site Geeks for Geeks [1], que contém códigos em Java acompanhados de sua complexidade. O SACI busca ampliar ainda mais as bases de dados e amostras de algoritmos disponíveis para a comunidade científica e desenvolvedores, criando um repositório robusto e abrangente.

A ferramenta proposta neste trabalho é construída para extrair dados sobre fontes pré-selecionadas, das quais os algoritmos são extraídos junto de suas respectivas complexidades. O resultado da extração é um *dataset* composto de algoritmos e suas complexidades que é disponibilizado de forma aberta em plataforma online, como o Kaggle e o Github, onde esses dados podem vir a servir de base para análises e trabalhos futuros. Dentre os possíveis trabalhos pode-se citar uma ferramenta que, dado um determinado código, ela retorne sua respectiva complexidade. Hoje em dia

[1] https://www.geeksforgeeks.org/

já existem ferramentas que fazem tarefas similares a essa, como é o caso do GitHub Copilot [2] que sugere trechos de código em tempo real durante a escrita de código; o qual, além de seu atual funcionamento, também poderia sugerir novos trechos com melhoras em relação à complexidade e ao desempenho. Assim sendo, o trabalho aqui descrito visa construir uma ferramenta chamada SACI, capaz de extrair as informações necessárias e gerar uma base de dados de algoritmos e suas complexidades a partir do site Geeks for Geeks [3]. O SACI é desenvolvido na filosofia de *web scraping* (ZHAO, 2017), uma técnica para extrair informações da internet e salvá-las em um formato estruturado para análises futuras.

Um exemplo interessante de implementação prática é apresentado por Rodenbusch (RODENBUSCH, 2023), onde foi desenvolvida uma API especificamente para prever padrões em dados complexos, chamada RTCC (*Running Time Complexity Calculator*). Em seu trabalho, foram utilizados os resultados gerados pelo SACI, além de outras bases de dados, para criar uma nova base que, combinada com ferramentas de aprendizado de máquina para o treinamento de modelos, resultou em uma capacidade de predição com 79,05% de acurácia para classe de complexidade e 88,02% para classe de eficiência. Uma ferramenta capaz de inferir tais complexidades necessita ser provida de uma inteligência e, nesse caso, uma Inteligência Artificial (IA) poderia ser treinada com essa finalidade; mas, para tanto, são necessários dados, ou seja, algoritmos e complexidades. Os dados são parte crucial do processo de treinamento, pois fornecem as informações necessárias para que a IA possa aprender e melhorar continuamente (WANG *et al.*, 2022).

1.1 MOTIVAÇÃO

Atualmente, não existe no mercado uma ferramenta amplamente estabelecida a ponto de ser considerada um padrão da indústria para tal fim e que compartilhe todas estas variáveis de identificação; somente algumas que geram sugestões de trechos de código sem considerar a complexidade.

A criação de uma ferramenta como o SACI, que extrai algoritmos e suas complexidades, pode preencher essa lacuna. Além de facilitar a identificação e análise das complexidades, a base de dados resultante da ferramenta contém informações valiosas para a comunidade de pesquisadores e desenvolvedores. Tal base, pode ser usada para construção de modelos preditivos que estimem a complexidade de algoritmos, pode ajudar a otimizar algoritmos já existentes ou criar novos, também pode ser utilizada para realizar um análise comparativa dos algoritmos e suas complexidade e para futuros estudos teóricos.

[2] https://github.com/features/copilot

[3] https://www.geeksforgeeks.org/

1.2 OBJETIVOS GERAL E ESPECÍFICOS

Desenvolver uma ferramenta de *scraping* capaz de extrair algoritmos e suas complexidades, com o objetivo de gerar uma base de dados com o conteúdo extraído.

Objetivos específicos:

1. Realizar a extração com expressões regulares
2. Realizar a extração com OpenAI
3. Tratar os dados extraídos e padronizá-los
4. Caracterizar os dados extraídos

1.3 ESTRUTURA DO TRABALHO

Este trabalho está organizado como segue. Na Seção 2 são descritos alguns conceitos importantes utilizados no trabalho. Na Seção 3 são apresentados alguns trabalhos relacionados, envolvendo o conceito de *scraper*. Na Seção 4 é apresentado o Saci, o *web Scraper* desenvolvido para extrair algoritmos e suas complexidades. Na Seção 5 são descritos as métricas e os experimentos realizados. Finalmente, na Seção 6 são apresentadas algumas conclusões e os trabalhos futuros.

2 CONCEITOS FUNDAMENTAIS

Durante a elaboração, deste trabalho, foram utilizados alguns conceitos que são definidos neste capítulo: tais como os temas sobre *web scraping*, que é a base para a construção da ferramenta. Complexidade de algoritmos, um dos principais atributos que extraídos das páginas. Modelos de linguagem em grande escala que foram utilizados duran o processo de *scraping* e, por fim, expressões regulares que foram muito utilizadas para definir padrões e ajudar na extração.

2.1 WEB SCRAPING

A web é uma fonte valiosa de informação para diversos setores. Nela, encontramos muita informação útil e, ao mesmo tempo, muita coisa dispensável (DIOUF *et al.*, 2019a). Portanto, é preciso uma forma de coletar e extrair utilidade desses dados. Nesse sentido, a técnica de *web scraping*, largamente conhecida como uma maneira útil de coleta de *big data* (MOONEY; WESTREICH; EL-SAYED, 2015), e comumente descrita como o ato de extrair dados da internet e salvá-los em um arquivo ou base de dados para futuro uso e/ou análises, certamente ajudará nessa tarefa.

Há diversas maneiras de realizar essas extrações. Esse processo pode até ser feito manualmente por uma pessoa, mas isso costuma não ser eficiente porque demanda muito tempo. Hoje existem diversas ferramentas customizáveis para tornar o processo cem por cento automatizado, sendo capazes de simular a interação de um humano com o conteúdo (ZHAO, 2017). Tal simulação de comportamento humano costuma ser necessária em alguns casos, visto que diversas páginas costumam bloquear o acesso quando percebem que o usuário está agindo de forma estranha, identificando-o como um *scraper*. Técnicas como *fingerprinting* e análise de comportamento são comumente utilizadas para diferenciar pessoas reais de *scrapers*; algumas medidas também podem ser tomadas para dificultar a vida dos *scrapers*, como páginas com conteúdo dinâmico e o uso de *captcha*.

O processo de *web scraping* costuma ser dividido em duas etapas: a coleta dos dados puros e não estruturados e o tratamento e extração de informação interessante a partir desses dados; como, por exemplo, podemos fazer uma requisição HTTP a alguma página e depois navegar pelos elementos do DOM(*Document Object Model*, representação dos objetos que formam um documento HTML.) extraindo informações relevantes para depois tratar os dados e salvá-los em algum banco de dados.

É possível citar uma grande variedade de cenários em que o *web scraping* é utilizado, desde aplicações simples até uma análise de preço de ações automatizada até um *scraper* de mensagens em redes sociais. Muitas aplicações que fazem uso de algum modelo de IA contam com uma etapa de extração de dados. Nessa etapa, em alguns casos, temos o uso de *scraping*. Se analisarmos o contexto da web em alto nível,

é bem provável que boa parte dos metadados das páginas web sejam coletados por *scrapers*, por exemplo, quase todas as páginas web estão sofrendo ação de *scrapers* constantemente para construção dos motores de busca, como o Google (SNYDER, 2003).

Existem diversas abordagens que podem ser utilizadas para construir um scraper; abaixo estão algumas abordagens citadas na obra de (DIOUF *et al.*, 2019b):

1. Mimetismo: Funciona através de um conjunto de regras pré-definidas e customizáveis para o *scraper*. A localização dos dados a serem coletados é pré configurada no *scraper* e as ações acontecem normalmente através de seletores DOM(expressões usadas para manipular e/ou selecionar os objetos que formam um documento HTML.). É uma estratégia relativamente eficiente, mas não é recomendada para ser usada em diversos sites diferentes, não lida bem com heterogeneidade.

2. Medidas de peso: Baseada em um algoritmo genérico que analisa a árvore DOM da página e define um peso para cada palavra em cada galho. Através de dedução o algoritmo consegue extrair o texto de todos os nodos. A principal vantagem desse método é que não precisa de nenhuma especificidade, ou seja, se adapta a múltiplas páginas, mas os dados resultantes costumam ter muito ruído.

3. Diferencial: Baseia-se no argumento de que duas páginas do mesmo site só irão divergir em conteúdo do corpo ou em inglês *body*. Seguindo essa lógica, barras de menu, colunas e outros elementos seriam idênticos entre páginas, assim só o conteúdo que fosse diferente seria coletado. Funciona bem quando o site é estruturado em um padrões, por exemplo, utilizando *templates*, ou seja, modelos pré-definidos que definem a disposição dos elementos na página, como cabeçalhos, rodapés, barras de navegação e colunas. Em sites de notícias, por exemplo, esses *templates* garantem uma uniformidade visual e funcional, permitindo que apenas o conteúdo específico de cada página varie. Dessa forma, a coleta de dados pode focar diretamente nas áreas de conteúdo que mudam, otimizando a extração de informações e aumentando a precisão do processo.

4. Aprendizado de máquina: Tem como princípio treinar um algoritmo utilizando uma base com uma grande quantidade de páginas web analisadas à mão. Assim, após treinamento o algoritmo seria capaz de identificar onde está o conteúdo desejado e extraí-lo.

2.2 COMPLEXIDADE DE ALGORITMOS

Temas como algoritmos e estrutura de dados são frequentemente relegados por alguns programadores, seja por desinformação ou por acharem que não os afeta

diretamente. Enquanto isso, debates de qual linguagem é mais rápida se tornou uma pauta altamente discutida, mas totalmente sem importância se a construção do código for feita alheia à noção de algoritmos.

Mas o que são algoritmos? Algoritmos podem ser definidos como ferramentas ou um conjunto de regras que resolvem algum problema, seja esse problema calcular uma rota mais curta, organizar uma lista de nomes em ordem alfabética, filtrar algum histórico etc. (CORMEN *et al.*, 2001). Existem muitos problemas que os algoritmos nos ajudam a resolver. Para isso, é preciso saber qual técnica utilizar, qual categoria o problema se encaixa. Alguns exemplos de categoria de algoritmos: algoritmo de busca, algoritmo recursivo, algoritmo de ordenação, algoritmo guloso etc.

A complexidade de algoritmos busca analisar a eficiênci a de um algoritmo ignorando a linguagem ou hardware em que ele está implementado; comparar baseado em tempo não é uma boa maneira de medir isso. Por exemplo, é possível que um código em JavaScript execute mais rápido do que em Python, isso ocorre pelas diferenças estruturais entre as linguagens de programação. Tal fato perde importância se o algoritmo selecionado para uso for ruim em termos de eficiência. Portanto, a complexidade de algoritmos é um aspecto crucial no estudo e construção de códigos bem estruturados e computacionalmente eficientes; sendo uma medida do desempenho de uma solução para um determinado problema (SIKKA *et al.*, 2020a).

Em função disso é que foram definidos os conceitos de complexidade de tempo e espaço; um algoritmo é mais eficiente que outro se apresentar uma menor complexidade de tempo e espaço. Para determinar estes dois fatores precisamos entender alguns conceitos como análise do código baseado na velocidade que ele executa suas computações, contagem de instruções fundamentais que o código executa, análise de pior caso, análise de comportamento assintótico, entre outros.

2.2.1 Tempo de execução polinomial

A partir da análise de algoritmos discretos e discussões em meados de 1960 surgiu um conceito sobre como quantificar um tempo de execução 'razoável'. Para certos problemas um bom algoritmo é aquele que possui uma propriedade de escalonamento melhor, que significa que conforme o tamanho de entrada cresça com base em um fator constante, o algoritmo desacelera conforme outro fator constante. Pode-se formular essa propriedade de escalonamento da seguinte maneira: suponha que em um algoritmo, para cada entrada de tamanho N, seu tempo de execução seja limitado por cN^d etapas computacionais, onde c e d são constantes positivas. Em outras palavras, o tempo de execução é, no máximo, proporcional a N^d. Esse comportamento é conhecido como tempo de execução limitado a uma função polinomial. Assim, a definição proposta é que um algoritmo é eficiente se tiver um tempo de execução polinomial (KLEINBERG; TARDOS, 2006).

No entanto essa definição pode ser muito específica, nem todo algoritmo com tempo de execução proporcional a um polinômio é considerado eficiente. Além disso, um algoritmo com tempo de execução não polinomial, como $O2^n$, pode ser aceitável dependendo do caso em que está aplicado. A definição de tempo polinomial é comumente mais usada porque há uma grande diferença entre o crescimento de funções polinomiais e exponenciais. Isso ajuda a dizer se existem algoritmos eficientes para certos problemas, avaliar o desempenho dos algoritmos e a viabilidade dos problemas de forma mais objetiva e aceita por todos. Além disso, permite investigar cientificamente se existem ou não algoritmos eficientes (KLEINBERG; TARDOS, 2006).

2.2.2 Análise assintótica

A análise assintótica trata da análise de desempenho de algoritmos à medida que o tamanho da entrada aumenta; em vez de se concentrar em medidas exatas de tempo de execução, ela se preocupa com o comportamento geral do algoritmo à medida que o tamanho da entrada tende ao infinito. Ela é baseada na ideia de que, para tamanhos de entrada suficientemente grandes, o fator dominante que determina o desempenho do algoritmo é a taxa de crescimento da função que descreve o tempo de execução em relação ao tamanho da entrada; essa taxa de crescimento é chamada de ordem de crescimento assintótica.

A ordem de crescimento assintótica é geralmente expressa usando a notação Big O; por exemplo, se um algoritmo tem uma ordem de crescimento assintótica de $O(n)$, isso significa que o tempo de execução do algoritmo cresce proporcionalmente ao tamanho da entrada. Se a ordem de crescimento assintótica for $O(n^2)$, o tempo de execução crescerá quadraticamente com o tamanho da entrada, e assim por diante. Isso nos permite fazer comparações entre diferentes algoritmos; algoritmos com ordens de crescimento assintóticas menores são considerados mais eficientes, pois seu tempo de execução cresce mais lentamente à medida que o tamanho da entrada aumenta. Existem diversas classes de complexidades, abaixo estão alguns exemplos das complexidades mais utilizadas usando a notação Big O:

1. Constante: $O(1)$ - significa que o tempo de execução do algoritmo não aumenta à medida que o tamanho da entrada aumenta. Algoritmos com complexidade de tempo constante são eficientes, pois executam em tempo constante, independentemente do tamanho da entrada.

2. Linear: $O(n)$ - é quando, o tempo de execução do algoritmo aumenta de forma proporcional ao tamanho da entrada. Algoritmos com complexidade de tempo linear têm um desempenho que aumenta de maneira proporcional ao tamanho da entrada. À medida que o tamanho da entrada aumenta, o tempo de execução do algoritmo também aumenta na mesma proporção.

3. Exponencial: $O(2^n)$ - neste caso, o tempo de execução do algoritmo cresce exponencialmente com o tamanho da entrada. Algoritmos com complexidade de tempo exponencial se tornam rapidamente inviáveis para entradas grandes, pois o tempo de execução aumenta drasticamente à medida que o tamanho da entrada aumenta exponencialmente.

4. Polinomial: $O(n^k)$ - onde n é o tamanho da entrada e k é uma constante positiva. Algoritmos com complexidade de tempo polinomial têm um desempenho aceitável para entradas moderadamente grandes, pois o tempo de execução aumenta de acordo com um polinômio do tamanho da entrada. No entanto, é importante notar que a mesma ainda pode se tornar impraticável para entradas muito grandes, especialmente para valores grandes de k.

5. Fatorial: $O(n!)$ - significa que o tempo de execução do algoritmo cresce de forma fatorial em relação ao tamanho da entrada. Algoritmos com complexidade fatorial são considerados inviáveis pois o tempo de execução aumenta extremamente rápido à medida que o tamanho da entrada aumenta.

6. Logarítmica: $O(logn)$ - aquela em, que o tempo de execução do algoritmo cresce de forma logarítmica em relação ao tamanho da entrada. Algoritmos com complexidade de tempo logarítmica são considerados muito eficientes, pois o tempo de execução aumenta de forma relativamente lenta à medida que o tamanho da entrada aumenta, pode-se afirmar que é uma das melhores complexidades em termos de eficiência e escalabilidade para problemas computacionais.

2.3 EXPRESSÕES REGULARES

A grande maioria dos programadores já se deparou com algo chamado expressões regulares em algum momento de sua vida, onde precisou filtrar padrões em um bloco de texto. Expressão regular, popularmente conhecida como *regex*, é um método conciso e flexível de se especificar um padrão e identificar uma cadeia de caracteres na qual se tem interesse (SERVIAN, 2020). Para definir uma expressão não é necessário listar todos os elementos do conjunto; existem algumas operações que ajudam nessa tarefa e dentre elas citam-se:

- Alternância: Representada por uma barra vertical ‘|’ possibilita-nos separar duas alternativas; por exemplo, ‘cara|casa’ identifica cara ou casa.

- Agrupamento: Representado por parênteses, define a ordem de precedência e o escopo; utilizando o exemplo anterior, ‘ca(r|s)a’ identificaria o mesmo padrão.

- Quantificação: Quantificadores são caracteres que especificam quantas vezes o elemento anterior pode acontecer ou repetir. Representados por 3 símbolos, o ‘?’

faz com que o elemento ocorra zero ou uma vez, o '*' faz com que o elemento ocorra zero ou mais vezes, o '+' faz que com o elemento ocorra uma ou mais vezes.

As expressões regulares constituem-se um poderoso meio de buscar, substituir e validar padrões de texto. Elas são amplamente suportadas em diversas linguagens de programação e aplicadas em uma variedade de tarefas, desde validação de formulários até análise de dados textuais complexos. Dominar o uso das expressões regulares pode aumentar a eficiência e a flexibilidade do processo de desenvolvimento de software.

2.4 MODELOS DE LINGUAGEM DE GRANDE ESCALA

Os Modelos de Linguagem de Grande Escala (LLMs, do inglês 'Large Language Models'), como o GPT-4, representam o estado da arte no campo da Inteligência Artificial para o processamento de linguagem natural. Estes modelos são treinados em vastos conjuntos de dados textuais através de técnicas de *deep learning* (KERNER, 2023) e são capazes de realizar uma ampla gama de tarefas relacionadas à linguagem, desde responder perguntas e compor textos até tradução de idiomas e geração de código, entre outras tarefas como:

- Compreensão de leitura
- Geração de texto
- Tradução de idiomas
- Sumarização
- Resposta a perguntas
- Análise de sentimentos
- Autocompletar texto

Os LLMs (Modelos de Linguagem de Grande Escala) são baseados em redes neurais chamadas de transformadores, que contém bilhões de parâmetros treinados a partir de grandes bases de dados textuais. Essas redes conseguem analisar toda a sequência do texto, aprendendo padrões complexos e relações contextuais dentro dos dados, resultando em respostas mais precisas e relevantes. Além disso, eles são capazes de usar 'transfer learning', o que significa que podem aplicar conhecimento de uma tarefa para realizar outras tarefas de linguagem, mesmo com poucos ou nenhum dado adicional específico para essas novas tarefas. No entanto, por serem tão poderosos, esses modelos também levantam questões éticas e desafios, como a possibilidade

de gerar desinformação, a necessidade de minimizar viés e garantir a equidade, e o desafio de gerenciar direitos autorais e a originalidade do conteúdo gerado (ZHAO *et al.*, 2023).

Todas essas funcionalidades tornam os LLMs ferramentas muito úteis para integrar em aplicações. Em Python, uma biblioteca chamada Langchain facilita essa integração, atuando como um intermediário que simplifica a interação entre os modelos de linguagem e o software do desenvolvedor. Com a API do OpenAI integrada ao Langchain, os desenvolvedores podem criar aplicações que utilizam o poder do GPT-4 para entender e responder a consultas, realizar tarefas automatizadas e até gerar conteúdo dinâmico. Usar LLMs em um projeto não só amplia as funcionalidades de uma aplicação, mas também abre novas possibilidades para interfaces de usuário mais naturais e interativas, análises de dados mais aprofundadas e automação de processos relacionados à linguagem.

2.5 CONSIDERAÇÕES FINAIS

Neste capítulo, foram abordados conceitos fundamentais que sustentam a construção da ferramenta proposta neste trabalho. Abordaram-se as técnicas de *web scraping*, fundamentais para a coleta automatizada de dados da web. A importância da complexidade de algoritmos, apresentando como a análise assintótica e a compreensão dos tempos de execução polinomial são essenciais para a eficiência computacional. Discutiu-se a utilização de expressões regulares e seu papel na identificação de padrões textuais. Por fim, examinou-se os Modelos de Linguagem de Grande Escala e suas possíveis aplicações.

Esses conceitos fornecem uma base sólida para o desenvolvimento e implementação das técnicas e ferramentas descritas nos próximos capítulos. A compreensão profunda desses temas é crucial para a análise dos trabalhos relacionados, que será discutida no próximo capítulo. Nele, revisaremos estudos e pesquisas relevantes, contextualizando e comparando-os com a proposta deste trabalho.

3 TRABALHOS RELACIONADOS

Na elaboração da ferramenta de extração, objeto desse estudo, considerou-se as abordagens de quatro obras de desenvolvimento de ferramentas que, embora não realizem a extração das complexidades, possuem relação com a que está sendo elaborada, visto que as mesmas também têm um objetivo, características específicas e dificuldades inerentes. Assim sendo, detalhar resumidamente em que consiste cada uma dessas quatro ferramentas e traçar um comparativo de suas características com as da ferramenta apresentada neste estudo.

3.1 LEARNING BASED METHODS FOR CODE RUNTIME COMPLEXITY PREDICTION

Este trabalho de (SIKKA *et al.*, 2020b) aborda a predição da complexidade de tempo de execução de um código de programação. Os autores destacam que prever a complexidade de tempo, até mesmo para humanos, é uma tarefa difícil, exigindo uma análise sutil e amplo conhecimento de algoritmos; por isso propõem abordar esse problema como uma tarefa de aprendizado de máquina e verificam sua viabilidade por meio de análises detalhadas. Como não há conjuntos de dados de código aberto disponíveis para essa tarefa, os autores propõem seu próprio conjunto de dados chamado 'CoRCoD: Code Runtime Complexity Dataset', que foi extraído de plataformas de programação online.

Os resultados apresentaram problemas de acurácia devido à base de dados criada ser pequena em relação ao tamanho considerado padrão atualmente; o modelo treinado teve dificuldades em prever as complexidades devido à falta de dados e variedade, um problema comum para algoritmos de *deep learning* que tendem a ter melhores resultados com mais dados.

Soluções como essa, potencialmente aplicam-se em avaliação automática de tarefas de programação, ferramentas integradas de análise de código estático, dentre outros. Este trabalho contribui com a disponibilização de um novo conjunto de dados anotado, bem como com a aplicação de técnicas de aprendizado de máquina para a previsão da complexidade de tempo de execução de código, proporcionando um avanço nesse campo.

3.2 QFEX - UM CRAWLER PARA BUSCA E EXTRAÇÃO DE QUESTIONÁRIOS DE PESQUISA EM DOCUMENTOS HTML

De acordo com Mathias e Dorneles (2021), o (MATHIAS; DORNELES, 2021) é um *web crawler* construído para coletar da internet questionários de pesquisa e extrair as questões a fim de formar uma base de dados. Um dos desafios encontrados na

confecção desse *crawler* é um problema muito comum: a grande variedade de layouts e formas que as páginas web são desenhadas; logo, esse trabalho focou apenas na extração em páginas com conteúdo estático. Para lidar com a diversidade das páginas foi utilizada duas heurísticas; uma para a identificação dos questionários e outra para a extração, focando-se em procurar por palavras relacionadas como *survey* ou 'questionário', elementos HTML de formulário, entre outras coisas.

Sua implementação foi feita em Java com o auxílio de uma biblioteca chamada Crawler4j que possui métodos para navegar nos elementos HTML e trabalhar com JSON. Como resultado foram encontrados 2262 links de questionários, os quais passaram por uma limpeza e, ao fim, restaram 510 links totalizando 5765 perguntas. Todos os dados extraídos foram inicialmente postos em JSON e armazenados em um banco de dados relacional; a precisão foi boa, maior que 94 por cento; ou seja, 2137 dos links realmente possuíam questionários.

Os resultados desse trabalho podem ser usados em diversas áreas como enriquecer bases de conhecimento, avaliar a qualidade de questionários de pesquisa, gerar novos questionários etc.

3.3 FERRAMENTA PARA COLETA E COMPARAÇÃO DE DADOS DE PUBLICAÇÕES ACADÊMICAS DOS PROFESSORES COM O CURRÍCULO LATTES

(BRANCO, 2018) destaca que esta ferramenta visa criar um *crawler* capaz de extrair informações de outros sites como o ResearchGate para complementar o currículo Lattes com dados que não existem no mesmo. Assim, a aplicação iria integrar e unificar os dados do Lattes com as novas obras que forem encontradas. O currículo Lattes é uma poderosa ferramenta dentro do mundo acadêmico; através dela é possível avaliar o nível de um pesquisador e diversas instituições a utilizam quando da seleção de candidatos para ocupar suas vagas, daí a importância de se manter o currículo sempre atualizado.

O *crawler* desenvolvido possui uma arquitetura simples; ele se conecta a um servidor *proxy* para utilizar as ferramentas de pesquisa, acessa os repositórios e extrai os dados das publicações, faz a análise do currículo Lattes para comparar com os dados extraídos, e grava os dados em uma base de dados. Para acessar o Lattes era necessário resolver um *captcha*; então o autor optou por fazer isso manualmente e deixar os dados pré baixados em formato XML.

A implementação foi feita em JAVA, com auxílio da IDE Eclipse, e foi utilizado o PostgreSQL como banco de dados. As principais bibliotecas utilizadas foram 'jsoup', uma biblioteca desenvolvida para trabalhar com páginas HTML, com uma estrutura de manipulação e extração de dados; e 'Java-string-similarity', uma biblioteca que implementa algumas maneiras de medir a similaridade e distância entre strings.

A aplicação final foi desenvolvida especificamente para certos sites, portanto, não

funciona bem de forma genérica, sendo, um dos problemas, o fato de que pequenas mudanças no site podem implicar no mal funcionamento do código. Também enfrentou-se alguns problemas com relação a bloqueio no acesso ao site devido à detecção do *crawler*. Para trabalhos futuros, seria interessante o uso de uma IA na parte de comparação de dados a fim de obter melhores resultados e que demandem menos análise manual.

3.4 CRAWLEX: UMA FERRAMENTA PARA EXTRAÇÃO DE DADOS NA WEB CONFIGURÁVEL ATRAVÉS DE EXEMPLOS

Segundo (LESSA, 2022), o CrawlEX é uma ferramenta que busca facilitar a vida do usuário com a criação de um *crawler* genérico e configurável para a procura e extração de artigos da internet. Ele possui uma interface visual com diversas telas desenvolvidas com Python; na tela de início são exibidas algumas instruções quanto ao uso de cada opção, a tela de fornecer exemplos é onde o usuário entra com os artigos que serão a base utilizada pelo *crawler*, a tela de carregar fonte permite ao usuário selecionar um site já configurado e verificar os dados coletados, a tela de executar um *crawler* permite a seleção de páginas web já coletadas e o envio de argumentos que devem ser procurados nas mesmas, a tela de configurações apresenta algumas opções quanto ao envio de relatórios.

A implementação foi feita em Python, utilizou de bibliotecas como ‘pyqt’, ‘beautifulsoup’, ‘sqlite’, ‘html5lib’, entre outras. Após fornecer os exemplos, o programa gera uma, uma expressão regular, identifica automaticamente uma possível URL que possua mais artigos e a extração é feita a partir de *tags* do ‘beautifulsoup’ pré-definidas.

Experimentos iniciais foram feitos passando 1, 2 e 3 exemplos para o programa. Com 1 exemplo o *crawler* teve dificuldade em encontrar os artigos; com 2 o desempenho foi bom, mais de 78 por cento de efetividade; já com 3, esse número abaixou um pouco e um dos motivos para isso é que, com mais exemplos ocorre uma maior generalização na pesquisa dos URLs.

3.5 DESENVOLVIMENTO DE API PARA PREDIÇÃO DA COMPLEXIDADE DE TEMPO DE EXECUÇÃO DE CÓDIGOS POR MEIO DE AUTOML

Em (RODENBUSCH, 2023), o autor realiza a combinação de duas bases de dados resultantes de *scraping* e gera uma base chamada de ’Crawleds’, logo após ocorre outra combinação gerando uma terceira nomeada de ’Mergeds’, que consta com 1668 amostras as quais são utilizadas como base principal para o trabalho. A partir do uso de métodos de aprendizado de máquina e a base foram gerados modelos de classificação, voltados para predição de complexidade em trechos de código, tais modelos atingiram 88,38% de acurácia para cálculo de eficiência e 77,58% para cálculo

de classe de complexidade. A partir destes foi desenvolvida então uma API a fim de disponibilizar uma interface para que o usuário final possa fazer fácil uso dos modelos treinados, na API é possível enviar trechos de código em Java e obter a classe de eficiência e complexidade predita.

3.6 ANÁLISE DOS TRABALHOS

Ao comparar quatro das obras apresentadas que implementaram alguma ferramente de extração e contrastando-as com o trabalho atual, é perceptível que, embora haja distintos focos de extração, a maioria dos trabalhos é direcionada de forma especializada. A Tabela 1 ilustra essas diferenças.

O CrawlEX se destaca por tentar abordar uma perspectiva mais generalizada; no entanto, ainda necessita de configurações prévias, o que pode influenciar sua eficiência que não atinge a totalidade. Independentemente da abordagem, é comum que todos utilizem bibliotecas especializadas para navegar na estrutura DOM da página e nos elementos HTML. Essa dependência de configuração, seja ela feita pelo usuário ou já incorporada, é crucial, visto que cada página da web possui suas particularidades.

A presença ou ausência de uma interface gráfica parece estar diretamente ligada ao público-alvo de cada trabalho. Trabalhos como ECL e CrawlEX, que possuem interface gráfica, parecem ser orientados para usuários finais. Por outro lado, obras como o qFEx, CoRCoD e o SACI estão mais voltadas para a extração de dados para alimentar bases, não necessitando, portanto, de um elemento visual.

Em relação às linguagens de programação adotadas, duas predominam: Python e Java. O que é particularmente interessante é que, independentemente da linguagem escolhida, as bibliotecas auxiliares empregadas, como ‘beautifulsoup’ e ‘jsoup’, visam solucionar problemas semelhantes, demonstrando um padrão de desafios comuns na área de extração de dados web.

Comparação de trabalhos					
Trabalho	Tipo	Item Alvo	Técnica	Algoritmo de Extração	Interface Gráfica
qFEx	Focado	Questionários	Heurísticas	Do autor	Não
ECL	Focado	Publicações	Similaridade textual	Do autor	Sim
CrawlEX	Genérico	Artigos	Comparação textual	Do autor e Ratcliff/Obershelp	Sim
CoRCoD	Focado	Algoritmos e Complexidades	API e Scraping	Do autor	Não
SACI	Focado	Algoritmos e Complexidades	Expressões regulares e OpenAI	Do autor	Não

Tabela 1 – Comparação de trabalhos

4 SACI - SCRAPER DE ALGORITMOS E COMPLEXIDADES

Este capítulo apresenta o SACI (Scraper de algoritmos e complexidades), um *web scraper* desenvolvido para a extração de algoritmos e suas complexidades. Inicialmente, é apresentada uma visão geral de seu funcionamento com códigos em alto nível, em seguida são mostrados detalhes da implementação sobre a fonte de dados usada na extração, o Geeks4Geeks [1], e, por fim, são apresentados alguns experimentos que mostram alguns números interessantes da extração realizada. Os arquivos com o código fonte podem ser encontrados no repositório[2] do GitHub.

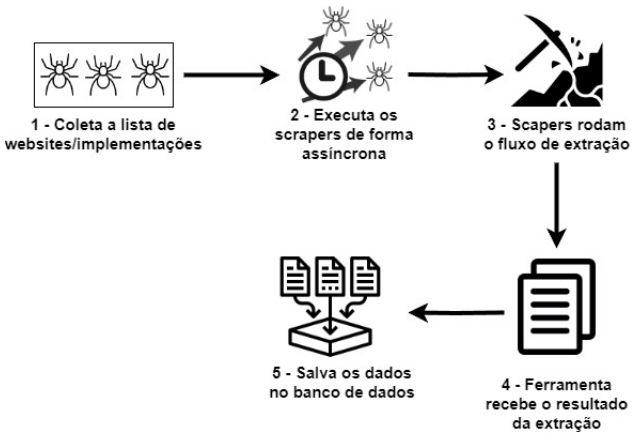
4.1 VISÃO GERAL

Esta seção apresenta o fluxo da ferramenta e de extração do *scrapers* desenvolvidos na ferramenta SACI.

4.1.1 Fluxo do SACI

Para o desenvolvimento da base da dados foi construída uma ferramenta de extração utilizando conceitos de *web scraping*. Essa ferramenta é capaz de extrair os dados desejados, como os códigos e complexidades dos algoritmos das páginas escolhidas para serem extraídas e que tiveram suas implementações realizadas. A Figura 1 demonstra o fluxo no qual a ferramenta atua:

Figura 1 – Fluxo de atuação da ferramenta



Fonte: elaborada pelo autor

Como já mencionado, a ferramenta realiza a extração somente das páginas web

[1] https://www.geeksforgeeks.org/

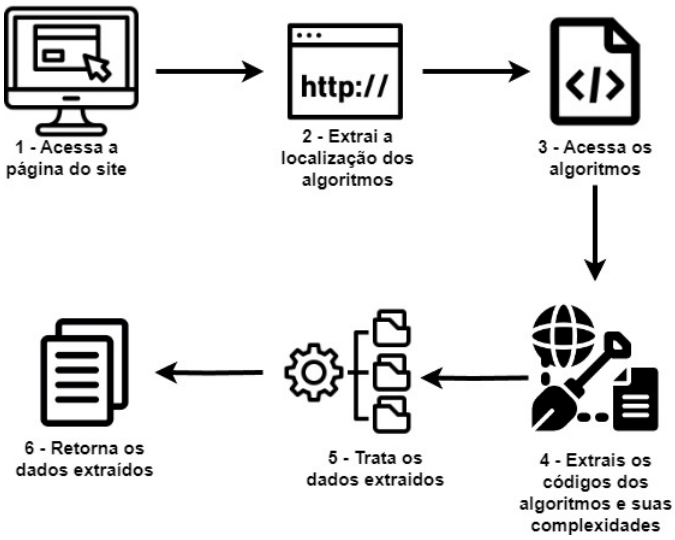
[2] https://github.com/Augustives/SACI

para as quais foi feita a implementação. Isso, devido ao fato que cada página contém suas peculiaridades exigindo, assim, certa especificidade para ser possível realizar a extração dos dados. Para tanto, existe uma lista de 'strings' representando websites, definida no arquivo '__init__.py', mais detalhes nas Sub-Seções 4.3 e 4.3.1. Conforme a Figura 1. a ferramenta começa lendo a lista de websites na etapa 1, executa cada implementação de forma assíncrona com o intuito de diminuir o tempo necessário, etapas 2 e 3, e, por fim, retorna todos os resultados em conjunto na etapa 4. Os dados retornados e já padronizados podem ser salvos em uma base de dados MongoDB ou em um arquivo JSON local durante a etapa 5.

4.1.2 Fluxo de extração

A Figura 2 apresenta o fluxo de extração executado, individualmente, em cada uma das páginas.

Figura 2 – Fluxo de tarefas de extração



Fonte: elaborada pelo autor

Conforme a Figura 2, o fluxo de extração dos dados para cada página/*website* se resume em acessar a página do site na etapa 1, localizar onde estão os alvos da extração e coletar os URLs que direcionam para esses dados, etapa 2 e 3, acessar cada URL e extrair os dados (códigos dos algoritmos, suas complexidades), etapa 4. Após extraídos, os dados são tratados e retornados nas etapas 5 e 6.

4.2 IMPLEMENTAÇÃO

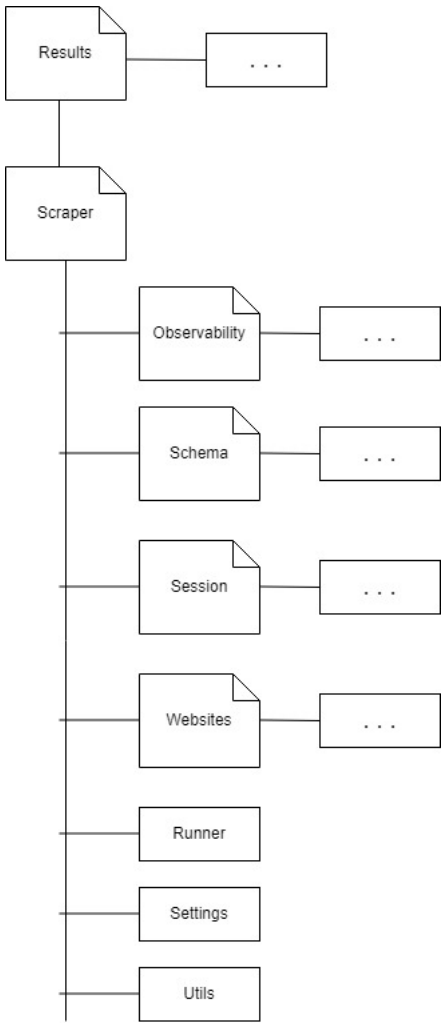
Para facilitar reprodutibilidade, SACI é explicado em detalhes conforme foi implementado. Primeiramente, é descrita a estrutura de pastas, e em seguida como cada

uma está organizada e o seu papel dentro da arquitetura do SACI.

O SACI foi desenhado de forma que é possível a inclusão de novas páginas. Dependendo do interesse do desenvolvedor, basta realizar a implementação seguindo os moldes definidos na Sub-Seção 4.3. A Figura 3 demonstra a estrutura da ferramenta em alto nível, onde cada nota, por exemplo 'Results, representa uma pasta e cada bloco representa um arquivo:

4.3 ESTRUTURA GERAL

Figura 3 – Estrutura do SACI



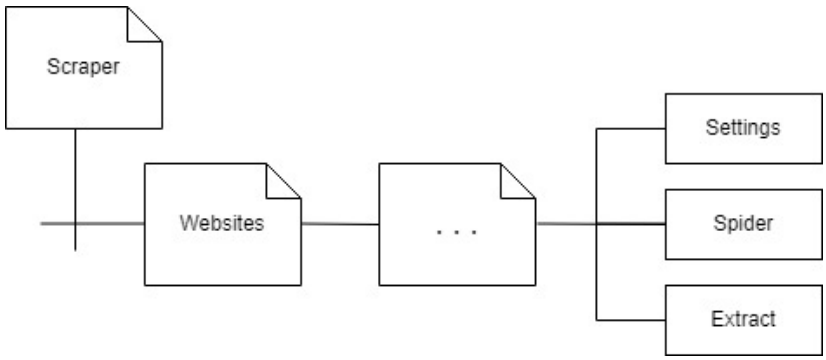
Fonte: elaborada pelo autor

A pasta 'Results', é destinada a armazenar os resultados e análises de cada página localmente. Dentro de 'Observability' estão os códigos responsáveis pelos 'logs' da ferramenta e o código responsável pela geração de métricas e análise do desempenho da extração; por exemplo, quantos algoritmos tiveram suas complexidades corretamen-

te extraídas. Em ’Schema’ encontra-se o esquema JSON, desenvolvido com auxílio da biblioteca Pydantic na qual os dados resultantes são validados. A pasta ’Session’ contém a implementação de uma sessão HTTP, responsável por todas as requisições feitas no código e pela construção do objeto response, que é o retorno para cada requisição feita. Na pasta ’Websites’ está a lógica específica para cada página que foi implementada. Até o momento final deste trabalho apenas o site Geeks4Geeks foi implementado. No arquivo ’runner’, está a lógica para executar todos os algoritmos de extração para cada página de forma assíncrona e armazenar os resultados de cada um na base de dados ou localmente. Em ’settings’, estão descritas algumas configurações globais, como a variável que define se os resultados ficam localmente em JSON ou são enviados para o banco de dados. Por último, em ’utils’, existem códigos que auxiliam no *scraping* de forma geral, por exemplo, a função de ’retry’, que executa novamente funções caso alguma exceção seja levantada. Essa função é útil para refazer requisições que apresentem algum problema, seja por fatores de segurança da página ou outros.

4.3.1 Spiders individuais

Figura 4 – Estrutura das páginas implementadas



Fonte: elaborada pelo autor

Cada página implementada deve ter seu código em uma pasta inserida dentro de ’websites’. Apesar de não ser obrigatório, é recomendado que essas implementações sigam o padrão definido na Figura 4 e contenham os seguintes três arquivos: ’settings’, ’spider’ e ’extract’.

Em ’settings’, estão alguns detalhes específicos para cada site, como URLs, cabeçalhos, expressões regulares etc. Em ’extract’, estão os métodos de extração, funções para extrair os algoritmos, as complexidades, os títulos etc. No arquivo ’spider’, fica a lógica geral da extração, onde são feitas as requisições e, em seguida, chamados os métodos de ’extract’ para realizar a coleta dos dados.

Nada impede que mais arquivos sejam adicionados, caso o desenvolvedor deseje deixar o código mais modular e organizado. A única obrigatoriedade é realizar o registro da implementação no arquivo '__init__.py', dentro da pasta 'websites', conforme o exemplo no Algoritmo 1. Apesar de ser chamada de lista, o objeto em código é um dicionário onde ficam associados o nome de cada página com o método que roda o fluxo de extração e retorna os dados.

Algoritmo 1 Código mostra a lista de páginas em alto nível

```
Import executarSpider from página1 as executarPágina1
Import executarSpider from página2 as executarPágina2

websites ← {'página1': executarPágina1, 'página2': executarPágina2}
```

4.3.2 Results

A pasta 'results' é apenas um espaço para serem guardados os arquivos gerados pela ferramenta que são os resultados das extrações e os resultados das análises feitas sobre cada extração, todos em JSON. O Código 1 apresenta um exemplo, onde são mostrados alguns dos valores e características que são retornados com os resultados da execução do código, contém valores fictícios.

```
##### RESULTS #####

'Total Algorithms': 20,
'Time Complexity Extracted': '10 extracted - 50%',
'Space Complexity Extracted': '10 extracted - 50%',
'URLs with problem': [
    'https://www.geeksforgeeks.org/interval-tree/',
    ...
],
    'Distinct Time Complexities': 10,
    'Distinct Space Complexities': 10,
    'Non Trusted Time Complexities': 2,
    'Non Trusted Space Complexities': 2,
    'Time Complexity': {
      'True Positive': 10,
      'False Positive': 0,
      'True Negative': 0,
      'False negative': 0,
      'Accuracy': 1.0,
      'Precision': 1.0,
      'Recall': 1.0,
      'F1 Score': 1.0
    },
```

```
    'Space Complexity': {
      'True Positive': 10,
      'False Positive': 0,
      'True Negative': 0,
      'False negative': 0,
      'Accuracy': 1.0,
      'Precision': 1.0,
      'Recall': 1.0,
      'F1 Score': 1.0
    },
    'Time Complexity Classification': {
      'Constant complexity': 2,
      'Linear complexity': 2,
      'Exponential complexity': 2,
      'Polynomial complexity': 2,
      'Factorial complexity': 1,
      'Log complexity': 1
    },
    'Space Complexity Classification': {
      'Constant complexity': 2,
      'Linear complexity': 2,
      'Exponential complexity': 2,
      'Polynomial complexity': 2,
      'Factorial complexity': 1,
      'Log complexity': 1
    }
  }
]
```

Código 1 – Exemplo de arquivo de análises

Os dados que formam a base de dados também ficam dentro da pasta 'results' e são retornados em formato JSON, formam uma lista de objetos que representam os dados extraídos, o Código 2 exemplifica isso.

```
[
  {
    "name": "Knapsack Problem",
    "time_complexity": "O(2N)",
    "trustable_time_complexity": true,
    "space_complexity": "O(N)",
    "trustable_space_complexity": true,
    "url": "https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/",
    "codes": {
      "C": {
        "code": "...",
        "comments": "..."
      },
```

```
      "C#": {
        "code": "...",
        "comments": "..."
      },
      "C++": {
        "code": "...",
        "comments": "..."
      },
      "Python3": {
        "code": "...",
        "comments": "..."
      },
      "Python": {
        "code": "...",
        "comments": "..."
      },
      "PHP": {
        "code": "...",
        "comments": "..."
      },
      "Java": {
        "code": "...",
        "comments": "..."
      },
      "Javascript": {
        "code": "...",
        "comments": "..."
      }
    }
  },
```

Código 2 – Exemplo de arquivo de dados

Descrição das variáveis do Código 2:

1. **name**: Nome/Título do algoritmo.
2. **time_complexity**: Complexidade de tempo do algoritmo.
3. **trustable_time_complexity**: Booleano indicando se essa complexidade é confiável, foi corretamente extraída, houve ou não uso do método de 'fallback'.
4. **space_complexity**: Complexidade de espaço do algoritmo.
5. **trustable_space_complexity**: Booleano indicando se essa complexidade é confiável, foi corretamente extraída, houve ou não uso do método de 'fallback'.
6. **url**: URL de onde os dados foram extraídos.

7. **codes**: Dicionário contendo o código do algoritmo em diferentes linguagens de programação.

4.3.3 Observability

Dentro de 'observability', existe um *logger* implementado, Algortimo 2, responsável por produzir mensagens de erro ou avisos, de forma a ajudar no momento de depurar o código e obter mais informações em tempo de execução. Também há códigos para geração de métricas de avaliação, por exemplo o Algoritmo 3, onde se verifica quantos algoritmos foram coletados, a complexidade de tempo e espaço de cada um, o total que poderia ter sido extraído e quais deles tiveram erros, gera-se uma matriz de confusão, histogramas e outros gráficos.

Algoritmo 2 Código mostra o logger em alto nível

```
Function constroiLogger(nomeLogger)
Cria o logger
Configura o logger
return logger

logger ← constroiLogger('scraperLogger')
```

Algoritmo 3 Código calcula métricas de desempenho em alto nível

```
Function calculaMetricas(verdadeiroPositivo, falsoPositivo, verdadeiroNegativo, falsoNegativo)
Calcula a acurácia: (verdadeiroPositivo + verdadeiroNegativo) / (verdadeiroPositivo + verdadeiroNegativo + falsoPositivo + falsoNegativo)
Calcula a precisão: verdadeiroPositivo / (verdadeiroPositivo + falsoPositivo)
Calcula a sensibilidade (recall): verdadeiroPositivo / (verdadeiroPositivo + falsoNegativo)
Calcula a medida F1: (2 * precisão * recall) / (precisão + recall)
return métricas
```

4.3.4 Schema

Os dados retornados por cada página implementada devem seguir um esquema pré-definido. Na pasta 'schema', ele foi implementado com o auxílio da biblioteca Pydantic, uma biblioteca que fornece de forma prática maneiras de gerar esquemas de dados e validá-los. Antes do retorno do resultado definitivo, todos os dados passam por uma validação no formato apresentado no Código 3.

```python
class ScrapedCode(BaseModel):
    '''The scraped codes of a given algorithm'''

    code: str = Field(
        default='', description='The code of the algorithm in a certain language'
    )
    comments: str = Field(default='', description='The first few comments in the code')


class ScrapedAlgorithm(BaseModel):
    '''The final result of the scraping operations, an extracted algorithm with its attributes'''

    name: str = Field(default='', description='The name of the scraped algorithm')
```

```
    time_complexity: Optional[str] = Field(
        default='', description='The time complexity of the scraped
            algorithm'
    )
    trustable_time_complexity: bool = Field(
        default=True,
        description='If the given time complexity used a trustable
            extraction method',
    )
    space_complexity: Optional[str] = Field(
        default='', description='The space complexity of the algorithm'
    )
    trustable_space_complexity: bool = Field(
        default=True,
        description='If the given space complexity used a trustable
            extraction method',
    )
    url: str = Field(default='', description='The space complexity of the
        algorithm')
    codes: dict[str, ScrapedCode] = Field(
        default=dict(), description='The algorithm implementations'
    )
```

Código 3 – Esquema dos dados

Com a criação destes dois esquemas conseguimos atingir um nível de validação que evita problemas nos resultados. Isso é fundamental para garantir que os dados coletados durante o processo sejam consistentes e estruturados em um padrão.

4.3.5 Session

Na pasta 'session', está grande parte do funcionamento da ferramenta que é baseado em requisições HTTP. Para isso, foi implementado um tipo de sessão, apresentada em alto nível no Algoritmo 4, que utiliza como base a biblioteca Aiohttp que, por sua vez, é capaz de lidar com requisições assíncronas. Nela, ocorrem algumas validações de URL, um meio de definir os cabeçalhos que serão utilizados, um método de fazer requisições para páginas dinâmicas (que precisam de JavaScript) e um métodos para requisições normais. Também foi definido um objeto 'response' para cada resposta de requisição recebida onde são executados alguns métodos para se ter os dados da resposta em mãos. Com o objeto resposta é possível realizar buscas através de XPATH e através dos elementos HTML da resposta.

Algoritmo 4 Código mostra a sessão http em alto nível

```
1: Class SessãoHttp
2: Init(sessão, jsSessão, cabeçalhosPadrão)
3: self.sessão ← sessão
4: self.jsSessão ← jsSessão
5: self.cabeçalhosPadrão ← cabeçalhosPadrão

6: Getter cabeçalhosPadrão
7: Setter cabeçalhosPadrão

8: StaticMethod validarUrl(url)
9: if not urlValida() then
10:     raise Erro
11: end if

12: Method validaArgumentosRequest(**kwargs)
13: Extrai metodo(GET, POST...) dos kwargs
14: if metodo is None or metodo not in ARGUMENTOS_NECESSARIOS then
15:     raise Erro
16: end if
17: validarUrl(url)

18: Async Method request(**kwargs)
19: Valida argumentos
20: Realiza HTTP request
21: if request falhou then
22:     Log erro
23: end if
24: Cria objeto resposta

25: Async Method jsRequest(**kwargs)
26: Valida argumentos
27: Realiza JS request
28: Cria objeto resposta
```

Algoritmo 5 Código mostra o objeto resposta em alto nível

```
1: Class Resposta
2: Init(url, status, conteudo, cabeçalhos, resposta, soup)
3: self.url ← url
4: self.status ← status
5: self.conteudo ← conteudo
6: self.cabeçalhos ← cabeçalhos
7: self.resposta ← resposta
8: self.soup ← BeautifulSoup(content)
```

4.3.6 Runner

O arquivo 'runner', apresentado no Algoritmo 6, apesar de conter pouco código, é parte essencial do SACI. Nele fica a implementação responsável por gerenciar a extração de todas as páginas implementadas de forma assíncrona, tornando, assim, a tarefa mais rápida.

Algoritmo 6 Código mostra a função principal do arquivo runner em alto nível

```
1: Async Function rodarScrapers(url)
2: resultadosScrapers ← Executa scrapers e agrega resultados em uma lista
3: Salva os resultados em arquivo local ou em uma base de dados
```

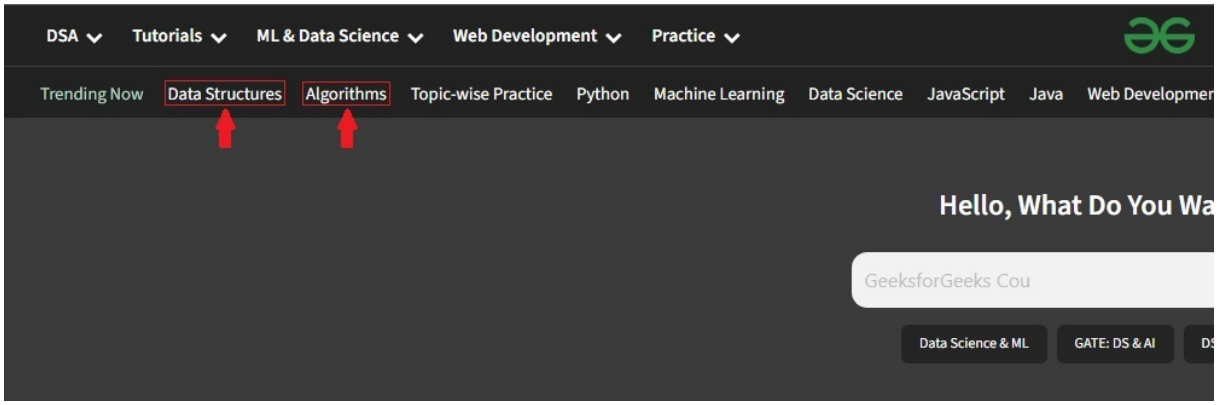
4.4 GEEKS4GEEKS

O Geeks4Geeks [3] é um site com conteúdo voltado para programadores com uma variedade de materiais para aprendizado e, também, de entretenimento. repleto de artigos, tutoriais, aulas, seminários, é um ótimo lugar para quem procura aprender algo novo.

No topo do site, Figura **??**, ficam evidenciados os principais tópicos: algoritmos de estrutura de dados, tutoriais, ciência de dados e aprendizado de máquina, desenvolvimento web, entre outros. Destes, o que interessa para o trabalho são os referentes a algoritmos e estrutura de dados. Dentro deles existem subtópicos e links, Figura **??**, que direcionam para páginas com implementações de código, em múltiplas linguagens, acompanhadas das complexidades de tempo e espaço Figura **??**, Figura 8 e Figura 9. Por conta dessas características, essas duas seções, apontadas na Figura **??**, foram escolhidas como fontes de dados a serem extraídos pelo SACI. Nessas, existe uma variedade de links de algoritmos, organizados conforme seus tipos e temas.

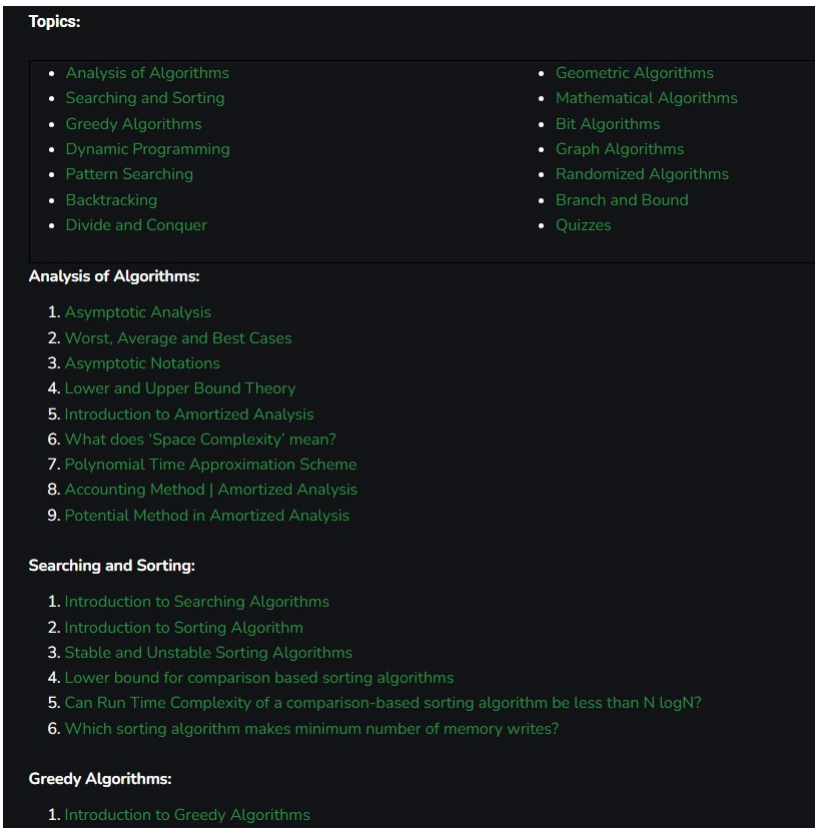
[3] https://www.geeksforgeeks.org/

Figura 5 – Menu Geeks4Geeks



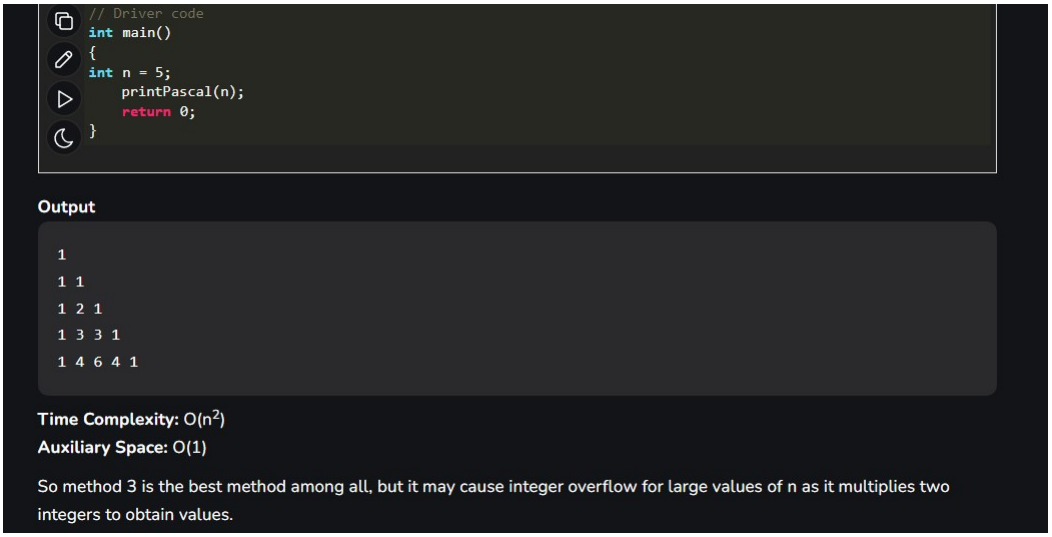
Fonte: elaborada pelo autor

Figura 6 – Página de algoritmos



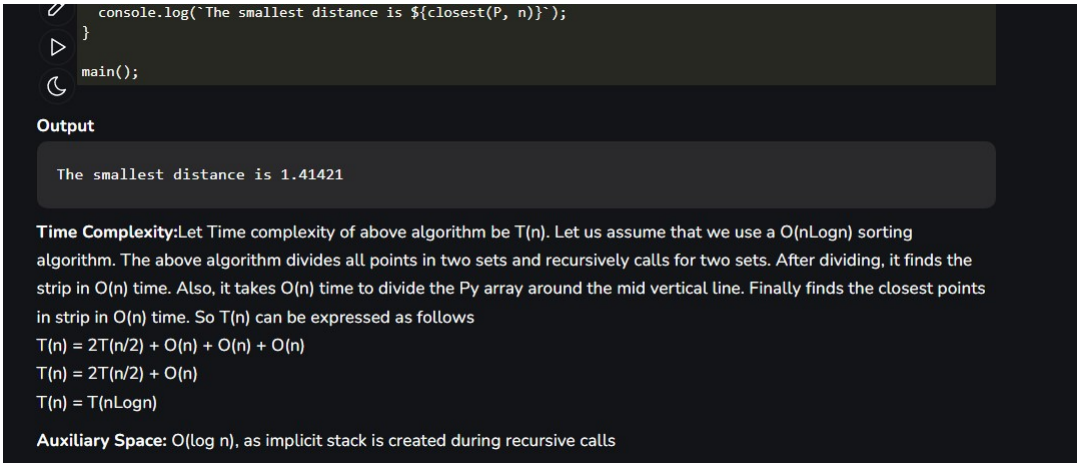
Fonte: elaborada pelo autor

Figura 7 – Complexidades formato 1



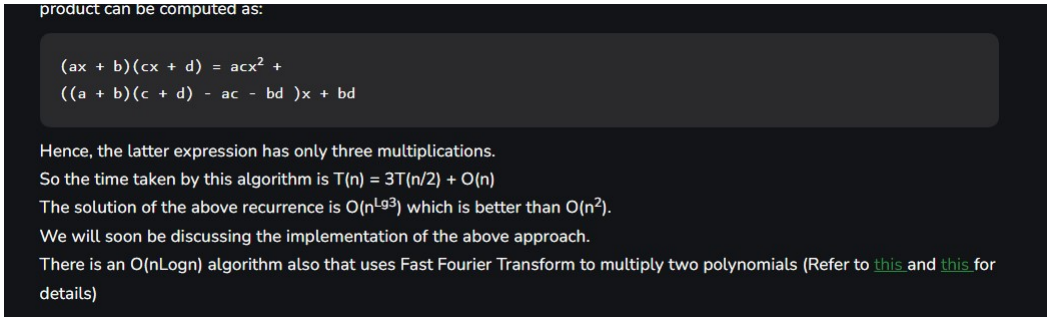
Fonte: elaborada pelo autor

Figura 8 – Complexidades formato 2



Fonte: elaborada pelo autor

Figura 9 – Complexidades formato 3



Fonte: elaborada pelo autor

Ao acessar esses links, tem-se as implementações dos algoritmos e suas complexidades. Em geral as complexidades são apresentadas após os blocos de código com suas respectivas implementações e se apresentam de diversas formas, podendo ser de uma maneira mais direta como na Figura **??** ou dentro de textos como na Figura 8 e na Figura 9.

4.4.1 Settings

Dentro do arquivo 'settings' tem-se, além das expressões regulares, os cabeçalhos que utilizados nas requisições HTTP, os URLs do site e a lista de linguagens de programação coletadas. Os cabeçalhos são muito importantes, pois são utilizados em todas as requisições e transmitem informações sobre a fonte da requisição e como ela deve ser tratada. Quando algum serviço de *scraping* faz uma requisição a um servidor, ele pode ser detectado e bloqueado com base em seus cabeçalhos. Vários sites têm mecanismos de defesa para identificar padrões e bloquear tais serviços que não se comportam como navegadores padrões. Por isso, os cabeçalhos são modificados para se parecerem com navegadores legítimos, 'disfarçando' sua verdadeira natureza e obtendo mais chances de acessar o conteúdo do site sem ser bloqueado.

```
HEADERS = {
    'Host': 'www.geeksforgeeks.org',
    'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,image/
        avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=
        b3;q=0.9',
    'Accept-Encoding': 'gzip, deflate',
    'Upgrade-Insecure-Requests': '1',
    'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit
        /537.36 (KHTML, like Gecko) Chrome/104.0.0.0 Safari/537.36',
    'Sec-Fetch-Site': 'none',
    'Sec-Fetch-Mode': 'navigate',
    'Sec-Fetch-User': '?1',
```

```
    'Sec-Fetch-Dest': 'document',
    'Sec-Ch-Ua': ''Chromium';v='105', 'Not)A;Brand';v='8'',
    'Sec-Ch-Ua-Mobile': '?0',
    'Sec-Ch-Ua-Platform': ''Windows'',
}
```

Código 4 – Cabeçalhos utilizados

Os cabeçalhos, apresentados no Código 4, são modificados para garantir que a requisição HTTP se pareça com a de um navegador legítimo, usado no dia a dia por um ser humano, para isso seguiu-se estas estratégias:

1. User-Agent: Identifica o cliente que está fazendo a requisição. Utilizou-se uma string de User-Agent de um navegador popular, como o Google Chrome, para fazer a requisição parecer que vem de um usuário humano, em vez de um serviço de *scraping* automatizado.

2. Accept e Accept-Encoding: Indicam os tipos de conteúdo que o cliente pode processar. Configurou-se esses cabeçalhos para aceitarem uma variedade de tipos de conteúdo e codificações comuns para navegadores, como HTML, XML, e imagens.

3. Sec-Fetch-Site, Sec-Fetch-Mode, Sec-Fetch-User, Sec-Fetch-Dest: Fazem parte da política de segurança dos navegadores modernos e informam ao servidor o contexto da navegação. Ajustou-se estes para imitar os padrões de navegação de um usuário humano.

4. Host e Upgrade-Insecure-Requests: Especifica o domínio ao qual a requisição está sendo feita, enquanto Upgrade-Insecure-Requests indica ao servidor para aceitar a conexão através de HTTPS. Ajustou-se estes para se alinhar com as práticas de navegação segura.

Ao ajustar esses cabeçalhos para simular um navegador padrão, é possível evitar muitos bloqueios e defesas automáticas implementadas pelos sites, aumentando significativamente a eficácia do *scraper*.

4.4.2 Scraper

Para o site do Geeks4Geeks, a extração seguiu em linhas gerais o fluxo já apresentado na Figura 2 e representado em alto nível no Algoritmo 7. A função principal do arquivo inicializa a sessão HTTP, faz o login na página a fim de evitar problemas como a detecção do *scraper* ou restrições a usuários não cadastrados, procura os URLs do algoritmo e, em seguida, faz o acesso aos algoritmos.

Algoritmo 7 Código mostra a função principal do arquivo spider em alto nível

```
1: Async Function executarSpider(url)
2: sessão ← SessãoHttp()
3: fazLogin(sessão)
4: urlsAlgoritmos ← encontratUrlsAlgoritmos(sessão)
5: return acessarTodosAlgoritmos(sessão, urlsAlgoritmos)
```

A extração dos URLs se dá em três funções, apresentadas na Figura 8. Uma responsável por acessar a página do site que contém os URLs, uma para fazer a extração e uma para filtrar os URLs extraídos.

Algoritmo 8 Código mostra as funções responsáveis por extrair URLs em alto nível

```
1: Function extrairUrlsAlgoritmos(response)
2: Encontra e retorna todos os URLs que contém algoritmos

3: Function filtraUrlsAlgoritmos(urlsAlgoritmos)
4: return Lista de URLs filtradas, sem URLs desnecessárias

5: Async Function encontratUrlsAlgoritmos(sessão)
6: urlsAlgoritmos ← Acessa página que contém as URLs
7: urlsAlgoritmos ← extrairUrlsAlgoritmos(urlsAlgoritmos)
8: urlsAlgoritmos ← filtraUrlsAlgoritmos(urlsAlgoritmos)
9: return Lista de URLs de algoritmos
```

A extração dos algoritmos é dividida em quatro etapas: primeiro, as URLs são percorridas e é chamada a função 'acessarAlgoritmo', linha 12 do Algoritmo 9, sobre cada uma, a qual faz a extração e adéqua os dados ao esquema, e por último os resultados são agregados em uma lista. A função 'extrairDados', linha 5 do Algoritmo 9, é encontrada no arquivo 'Extract', descrito na Seção 4.5.

Algoritmo 9 Código mostra as funções responsáveis por extrair algoritmos em alto nível

```
1: Function extrairDadosAlgoritmo(response)
2: if not response then
3:     return [ ]
4: end if
5: dados ← extrairDados(response)
6: if not dados then
7:     raise Erro
8: end if
9: return Dados brutos dos algortimos

10: Function algoritmosParaEsquema(dadosBrutosAlgoritmo)
11: return Lista de Algoritmos adequados ao esquema

12: Async Function acessarAlgoritmo(sessão, url)
13: response ← sessão.request(url)
14: dadosBrutosAlgoritmo ← extrairDadosAlgoritmo(response)
15: algoritmo ← algoritmosParaEsquema(dadosBrutosAlgoritmos)
16: return Algoritmo dentro do esquema definido

17: Async Function acessarTodosAlgoritmos(sessão, urlsAlgoritmos)
18: lista ← [ ]
19: for url in urlsAlgoritmos do
20:     algoritmo ← acessarAlgoritmo(sessão, url)
21:     Insere algoritmo na lista
22: end for
23: return Lista com os dados do algoritmo
```

4.5 EXTRACT

Foram desenvolvidas duas implementações para a extração. Uma das versões faz uso de expressões regulares como o principal meio de identificação e extração das complexidades. Já a outra, no momento da extração, utiliza inteligência artificial através da API(*Application Programming Interface*) do OpenAI para a identificação e extração das complexidades. Todos os resultados das extrações podem ser encontrados no repositório[4] do GitHub e também no Kaggle[5].

4.5.1 Extração com expressões regulares

Cada página do site pode conter mais de um código/implementação. Esses códigos ficam localizados em um elemento HTML chamado aqui de bloco. Cada bloco

[4] https://github.com/Augustives/SACI

[5] https://www.kaggle.com/datasets/augustives/algorithms-and-their-complexities

contém o algoritmo codificado em diversas linguagens de programação e uma complexidade associada ao algoritmo. Ao acessar à página o método procura pelas seções que contém esses blocos, linha 2 do Algoritmo 10, guarda as referências de posição desses blocos na árvore DOM junto dos códigos. Em seguida, procura em volta de cada referência pelas complexidades de tempo e espaço, cada uma através de uma seleção prévia de expressões regulares, linhas 7 a 18 do Algoritmo 10. Por fim, extraímos o título do algoritmo na linha 19 do Algoritmo 10 e adicionamos um objeto resultado a uma lista, linhas 20 a 32 do Algoritmo 10. Para a extração dos títulos dos códigos, linha 19, também utiliza-se a referência de localização dos blocos de código. É extraído o título principal da página que geralmente é o título do primeiro algoritmo e os restantes, para retornar uma lista de dicionários Python contendo o nome do algoritmo, a complexidade de tempo, a complexidade de espaço, o URL e o código em todas as linguagens, linhas 20 a 32. Essa lista é tratada pelo *scraper* na última etapa retornando os dados de acordo com o esquema JSON já definido na Sub-Seção 4.3.4. As chaves 'tempoConfiavel' e 'espaçoConfiavel' do objeto resultado tem relação com o uso ou não do método 'fallback' apresentado mais a frente. Caso o método seja utilizado essas chaves recebem como valor o booleano 'True', caso contrário 'False'.

Algoritmo 10 Código mostra a função de extração principal em alto nível

```
1:  Function extrairDados(response)

2:  codigos, referencisaDom ← extrairCodigosReferencias(response)
3:  if not codigos then
4:      Log aviso
5:      return []
6:  end if

7:  complexidades ← []
8:  for ref in referencisaDom do
9:      try:
10:     complexidadeTempo ← extrairComplexidadeDeReferencia(ref, regexTempo)
11:     catch Erro:
12:     complexidadeTempo ← extrairComplexidadeComFallback(ref)

13:     try:
14:     complexidadeEspaço ← extrairComplexidadeDeReferencia(ref, regexEspaço)
15:     catch Erro:
16:     complexidadeEspaço ← extrairComplexidadeComFallback(ref)

17:     Insere complexidades na lista
18: end for

19: titulos ← extrairTitulos(ref)

20: resultados ← []
21: for (nome, complexidade, codigo) in zip(nomes, complexidades, codigos) do
22:     resultado ← {
23:             'nome': nome,
24:             'complexidadeTempo': complexidade['complexidadeTempo'],
25:             'tempoConfiavel': complexidade['tempoConfiavel'],
26:             'complexidadeEspaço': complexidade['complexidadeEspaço'],
27:             'espaçoConfiavel': complexidade['espaçoConfiavel'],
28:             'url': url,
29:             'codigo': codigo,
30:     }
31:     Insere resultado na lista
32: end for

33: return resultados
```

Para extrair os blocos de código e suas referências, existe a função 'extrairCodigosReferencias', utilizada na linha 2 do Algoritmo 10. Esses blocos podem ser localizados com a ajuda de uma biblioteca chamada 'BeautifulSoup' que recebe conteúdo HTML e o transforma em 'soups', objetos que podem ser usados para fazer pesquisas e navegar no DOM da página.

Das linhas 4 a 7 do Algoritmo 11, através do objeto response e sua 'soup', é feita uma pesquisa com a função 'find_all' na qual retornam todas as 'divs' que possuem a classe 'responsive-tabs'; esses são os blocos de código. Em seguida, é percorrido cada bloco e, também, uma lista de linguagens (['C', 'C#', 'C++', 'Python3'...]).

Das linhas 8 a 15 do Algoritmo 11, procuramos em cada bloco o conteúdo com o algoritmo através da função 'find_next', estes localizados em elementos 'h2'com a classe 'tabtitle'. Caso exista o algoritmo, o código é localizado em elementos 'td' com a classe 'code' e extraído. Além do código, são coletados os comentários que ficam dentro do mesmo.

Algoritmo 11 Código mostra a função 'extrairCodigosReferencias' em alto nível

```
1:  Function extrairCodigosReferencias(response)
2:  codigos ← [ ]
3:  referencias ← [ ]

4:  blocosCodigo ← response.soup.find_all('div', {'class': 'responsive-tabs'})
5:  for bloco in blocosCodigo do
6:      listaLinguagens ← [ ]
7:      Insere referencia do bloco na lista de referencias

8:      for linguagem in linguagensProcuradas do
9:          algoritmo ← bloco.find_next('h2', {'class': 'tabtitle'}, string=lang)
10:         if algoritmo then:
11:             codigo ← extrairCodigo(algoritmo.find_next('td', {'class': 'code'}))
12:             comentarios ← extrairComentarios(codigo)
13:         end if

14:         Insere codigo na lista de linguagens
15:     end for

16:     Insere código em suas linguagens na lista de codigos
17: end for

18: return codigos, referencias
```

As funções que extraem o código, Algoritmo 12, e os comentários, Algoritmo 13, são simples: apenas percorrem os elementos e suas linhas e coletam o desejado. Ambas se utilizam da função 'find_all', linhas 3 e 5, pertencente à biblioteca 'BeautifulSoup', que resumidamente encontra todos os elementos dentro do 'soup', os quais correspondem aos parâmetros de pesquisa usados.

Código mostra a função 'extrairCodigo' em alto nível

```
1:  Function extrairCodigo(elementoCodigo)
2:  codigo ← ''
3:  linhasCodigo ← elementoCodigo.find_all('div', {'class': 'line'})

4:  for linha in linhasCodigo do
5:      trechosCodigo ← linha.find_all('code')
6:      for trecho in trechosCodigo do
7:          codigo ← codigo + trecho
8:      end for

9:  end for

10: return codigo
```

Código mostra a função 'extrairComentarios' em alto nível

```
1:  Function extrairComentarios(codigo)
2:  comentariosCodigo ← ''

3:  for linha in codigo do
4:      comentariosCodigo ← comentariosCodigo + comentario
5:  end for

6:  return comentariosCodigo
```

No momento da extração das complexidades, são utilizados a referência da localização dos códigos, linhas 10 e 14 do Algoritmo 10, métodos do 'BeautifulSoup' e expressões regulares. A extração conta com dois caminhos: o padrão com regras mais específicas e um que chamamos de 'fallback', no qual são utilizadas regras mais generalizadas e inclusivas a fim de conseguir extrair a informação caso ocorra alguma falha durante o fluxo padrão.

O método padrão, representado no Algoritmo 14, procura por todas as palavras *Time Complexity* ou *Space Complexity* nas proximidades do bloco de códigos, linha 4. Para cada palavra existe uma lista de expressões regulares, para cada palavra encontrada elas são inseridas em uma lista de 'matches', linha 3 e 6. Então, procura o 'match' mais próximo da referência, linha 8, com um cálculo simples: para cada 'match' da lista subtrai-se a linha em que a referência está com a linha que o 'match' encontra-se, dessa forma, o menor valor dentre todos os calculados é o mais próximo. A seguir, o método percorre outra lista de expressões regulares, dessa vez focada em capturar os valores das complexidades. Para cada expressão regular de valor, itera os elementos HTML, para os quais é feita uma pesquisa, linha 11. Caso nenhuma complexidade consiga ser extraída, é levantado um erro de extração, linha 17.

Sendo o erro levantado, o fluxo de 'fallback', representado no Algoritmo 15, tenta extrair novamente a complexidade, e uma vez utilizado esse método, uma 'flag' é adicionada aos resultados indicando seu uso.

Algoritmo 14 Código mostra a função 'extrairComplexidadeDeReferencia' em alto nível

```
1:  Function extrairComplexidadeDeReferencia(referencia, regex)
2:  for regex in regexes['palavra'] do
3:      matches ← [ ]
4:      palavra ← referencia.find_next(string=compile(regex))

5:      if palavra then:
6:          Insere palavra em matches
7:      end if

8:      match_mais_proximo ← Pega o match mais próximo da ref

9:      for regex in regexes['valor'] do
10:         for elemento in elementosHTML['valor'] do
11:             complexidade ← busca_regex(regex, match_mais_proximo)

12:             if complexidade then:
13:                 return complexidade
14:             end if
15:         end for
16:     end for

17:     raise FalhaExtraçaoComplexidade
18: end for
```

Algoritmo 15 Código mostra a função 'fallback' em alto nível

```
1: Function fallback(referencias, regex)
2: for referencia in referencias do
3:     if buscaComRegex(regex, referencia) then:
4:         return complexidade
5:     end if
```

A Figura 5 apresenta as expressões regulares criadas e utilizadas na extração dos dados. Aquelas que buscam pelas palavras, linhas 3, 16 e 17, tentam encontrar *time complexity*, *space complexity* e suas variações de escrita. As responsáveis pelos valores, linhas 5 a 10 e 19 a 28, são uma combinação da palavra com o valor da complexidade, buscando, por exemplo, algo como *Time Complexity*: O(log N). Todas foram criadas com base nos diversos padrões com que as complexidades são descritas nos textos das páginas do site. Por fim, tem-se a que é usada no método

'fallback, linhas 11 e 29.

```
TIME_COMPLEXITY_REGEX = {
    'word': [
        r'[Tt]ime [Cc]omplexit\w*[\s]?:|[Tt]ime [Cc]omplexit\w*[\s]?',
    ],
    'value': [
        r'[Tt]ime [Cc]omplexit\w*\s?:.*?(O\s*\(.*?\))',
        r'[Tt]ime [Cc]omplexit\w*\s?.*?(O\s*\(.*?\))',
        r'[Tt]ime [Cc]omplexit\w*\s.*?\sis\s.*?(O\s*\(.*?\))',
        r'[Tt]ime [Cc]omplexit\w*\s?:?.*?\sis\s(\w*)',
    ],
    'fallback': r'O\s*\(.*?\)',
}

AUXILIARY_SPACE_REGEX = {
    'word': [
        r'[Aa]uxiliary [Ss]pace\s?:|[Aa]uxiliary [Ss]pace\s?',
        r'[Ss]pace [Cc]omplexit\w*\s?:|[Ss]pace [Cc]omplexit\w*\s?',
    ],
    'value': [
        r'[Aa]uxiliary [Ss]pace\s?[\[Cc\]omplexit\w*]?:.*?(O\s*\(.*?\))',
        r'[Aa]uxiliary [Ss]pace\s?[\[Cc\]omplexit\w*]?.*?(O\s*\(.*?\))',
        r'[Ss]pace [Cc]omplexit\w*\s?:.*?(O\s*\(.*?\))',
        r'[Ss]pace [Cc]omplexit\w*\s?.*?(O\s*\(.*?\))',
        r'[Aa]uxiliary [Ss]pace\s?[\[Cc\]omplexit\w*]?.*?\sis\s.*?(O\s
            *\(.*?\))',
        r'[Ss]pace [Cc]omplexit\w*\s.*?\sis\s.*?(O\s*\(.*?\))',
        r'[Aa]uxiliary [Ss]pace\w*\s?:?.*?\sis\s(\w*)',
        r'[Ss]pace [Cc]omplexit\w*\s?:?.*?\sis\s(\w*)',
    ],
    'fallback': r'O\s*\(.*?\)',
}
```

Código 5 – Regex de extração

4.5.2 Extração com OpenAI

Para a extração com OpenAI utilizou-se da API oferecida [6], e para sua implementação foi utilizada uma biblioteca do Python chamada de Langchain [7]. Ela oferece uma integração fácil com modelos de linguagem ou LLMs(*Large Language Models*), como o GPT-3 e outros similares, além de oferecer aos desenvolvedores um conjunto de

[6] https://openai.com/index/openai-api/

[7] https://www.langchain.com/

ferramentas e abstrações que facilitam a criação de aplicações interativas e complexas. A premissa central da biblioteca Langchain é permitir a construção de correntes de 'prompts' e respostas; ou seja, sequências de interações com modelos de linguagem que podem ser encadeadas para realizar tarefas sofisticadas. Dentro do arquivo *utils* do SACI está definida uma classe que faz o uso desta biblioteca e realiza essa comunicação com a OpenAI.

Algoritmo 16 Código mostra a sessão http em alto nível

```
1: Class LlmSearcher
2: throttler ← Throttler(rate_limit=100, period=60)

3: Init()
4: self.LLM ← OpenAI(openai_api_key=...)
5: self.PROMPT ← PromptTemplate.from_template(template_do_prompt)
6: self.LLM_CHAIN ← LLMChain(prompt=self.PROMPT, llm=self.LLM)

7: Method search(complexidade, texto)
8: async with throttler:
9: resposta ← self.LLM_CHAIN.run(complexity=complexidade, text=texto)
10: return json.loads(resposta)
        =0
```

A classe, representada no Algoritmo **??**, atua como um *singleton*, isto é, só existe uma instância sua durante a vida útil da aplicação. Dentro dela estão definidos o LLM que conecta com o modelo de linguagem da OpenAI, o 'PROMPT' que define como o modelo irá responder e a LLM_CHAIN que une o 'PROMPT' com o modelo em si. Além disso, existe um 'throttler' que limita quantas 'requests' são feitas por minuto, pois, a API da OpenAI possui certas restrições.

```
I am using you during a scraping operation in which I need to extract from
    a piece of text the values of complexity/code complexity.
Consider that sometimes the space complexity is also called as auxiliary
    space.
Consider that both complexitys often come in this format: 'Some type of
    complexity: value of complexity'.
But they can also come inside the text like: 'The type of complexity of the
     given ... is ...'.
Give the answer in JSON format with no line breaks, with a key called '
    complexity' and the value for the key is your answer.
If you cant determine the answer give the json with a null in the value.
What is the {complexity} that is written in the following text: '{text}'?
```

Código 6 – Prompt

O 'PROMPT' utilizado é descrito no Algoritmo 6, com ele é possível explicar o

contexto em que o SACI está, demonstrando alguns exemplos de como os dados irão chegar, seus formatos e diferenças, sendo ainda explicado como a resposta deverá ser retornada e qual pergunta é feita. Com isso o modelo de linguagem consegue entender que está sendo feito a pesquisa pelos valores de complexidade do texto que ele está recebendo.

O método de extração das complexidades sobre as referências, representado pelo Algoritmo 17, também sofreu uma pequena alteração. Não é mais iterado a lista de 'regexes' responsáveis pela extração dos valores de complexidade, como era feito no Algoritmo 14, pois esse trabalho agora é feito pelo LLM na linha 10.

Algoritmo 17 Código mostra a função 'extrairComplexidadeDeReferencia' em alto nível

```
1:  Function extrairComplexidadeDeReferencia(referencia, regex)
2:  for regex in regexes['palavra'] do
3:      matches ← [ ]
4:      palavra ← referencia.find_next(string=compile(regex))

5:      if palavra then:
6:          Insere palavra em matches
7:      end if

8:      match_mais_proximo ← Pega o match mais próximo da ref

9:      for elemento in elementosHTML['valor'] do
10:         complexidade ← llm_searcher.search(complexidade, match_mais_proximo)

11:         if complexidade then:
12:             return complexidade
13:         end if
14:     end for

15:     raise FalhaExtraçaoComplexidade
```

5 EXPERIMENTOS

Para cada página em que foi implementada um *scraper* foram realizadas avaliações a fim de avaliar se a extração das informações teve um bom resultado ou não. Tentou-se garantir que a quantidade de extrações fosse boa e que as informações extraídas estivessem corretas. A seguir, é apresentada a configuração do ambiente que executou o *scraper*, descrevem as métricas utilizadas, caracterizam os dados extraído e detalham os experimentos e seus respectivos resultados a partir do uso do SACI.

5.1 CONFIGURAÇÃO DO AMBIENTE

Os experimentos foram realizados em uma máquina local, com o uso de Python e Visual Studio Code para rodar o *scraper* e com o auxílio do Libre Office para cálculos estatísticos e geração de gráficos. Configurações da máquina: Sistema operacional Microsoft Windows 10, processador AMD Ryzen 5600X, placa de Vídeo: GTX 1060 3GB, 16GB RAM, 1TB SSD.

5.2 MÉTRICAS

Para caracterizar a fonte de dados foram criados histogramas e gráficos para exibir o comportamento e a distribuição dos dados extraídos. Esses gráficos ajudam a visualizar padrões e tendências que podem não ser imediatamente aparentes através de uma simples observação.

Para melhor entender os resultados de cada extração foram selecionadas métricas de avaliação. A partir dos resultados é criado uma matriz de confusão e calculada a acurácia, revocação, precisão e F1-Score. Utilizam melhor essas métricas em conjunto do que individualmente, pois cada uma oferece uma perspectiva única sobre a performance do processo. Estas serão usadas para comparar a extração realizada com expressões regulares e a feita com OpenAi.

5.2.1 Histogramas

Um histograma é uma representação gráfica da distribuição de um conjunto de dados. Ele é utilizado para visualizar a frequência com que os diferentes valores ou intervalos de valores ocorrem em um conjunto de dados. Um histograma é composto por barras retangulares (também chamadas de *bins*) onde a altura de cada barra representa a frequência ou a contagem de dados que se enquadram em um determinado intervalo.

5.2.2 Matriz confusão

A matriz confusão é uma tabela que resume o desempenho de um modelo em dados de teste; com ela é possível avaliar sua qualidade, pois fornece uma visão detalhada dos acertos e erros do modelo. Ela é organizada em quatro quadrantes quando usada a classificação binária, que representam as quatro possíveis combinações de previsão e realidade:

Verdadeiro Positivo (VP): a previsão do modelo é positiva e a classe real também é positiva. Falso Positivo (FP): a previsão do modelo é positiva, mas a classe real é negativa. Falso Negativo (FN): a previsão do modelo é negativa, mas a classe real é positiva. Verdadeiro Negativo (VN): a previsão do modelo é negativa e a classe real também é negativa.

A partir da matriz de confusão na Figura 10, podem ser calculadas diversas métricas de avaliação, tais como acurácia, precisão, revocação e F1-Score.

Figura 10 – Matriz confusão

		Valor Predito	
		Sim	**Não**
Real	**Sim**	Verdadeiro Positivo (TP)	Falso Negativo (FN)
	Não	Falso Positivo (FP)	Verdadeiro Negativo (TN)

Fonte: elaborada pelo autor

5.2.3 Acurácia

A acurácia mede a proporção de classificações corretas em relação ao total de classificações feitas pelo modelo. Em outras palavras, a acurácia mede a taxa de sucesso geral do modelo em classificar as amostras corretamente. Por exemplo, se um *scraper* é configurado para extrair informações de um conjunto de 100 páginas da web e consegue obter corretamente os dados de 80 delas, então a acurácia desse seria de 80%.

Vale ressaltar que a acurácia é apenas uma métrica superficial e deve ser avaliada em conjunto com outras métricas. Por exemplo, suponha que o *scraper* esteja coletando dados de duas categorias diferentes, A e B, onde a categoria A representa 90% das páginas e a categoria B representa apenas 10%. Nesse caso, um *scraper*

que sempre extrai informações da categoria A terá uma acurácia de 90%, mas não será útil na prática, já que não está realmente obtendo os dados corretos da categoria B. Nesse cenário, a acurácia alta pode ser enganosa, pois o *scraper* parece estar funcionando bem ao considerar todas as páginas, mas na verdade está falhando em capturar informações importantes da categoria B.

```
acurácia = (número de classificações corretas) / (total de amostras)
```

5.2.4 Revocação

A revocação ou 'recall', é uma métrica usada para medir a proporção de verdadeiros positivos que foram identificados corretamente em relação ao total de amostras positivas. A revocação é especialmente relevante em cenários em que os falsos negativos têm um alto custo; ou seja, em casos em que classificar erroneamente uma amostra negativa como positiva pode ter consequências graves.

Consideremos um cenário no qual temos um *scraper* que faz a extração de preços de ações, em que uma amostra pode ser classificada como apta a ser comprada (positivo) ou não (negativo). O custo de um falso negativo é alto, pois pode resultar em uma compra indesejada e causar prejuízos significativos ao usuário. Nesse caso, é crucial que o sistema tenha uma alta revocação; ou seja, que identifique corretamente a maioria dos preços.

```
revocação = (amostras positivas classificadas corretamente) / (total de
    amostras)
```

5.2.5 Precisão

A precisão é uma métrica utilizada para avaliar a proporção de amostras classificadas como positivas que são realmente positivas em relação ao total de amostras classificadas como positivas. A precisão é também importante em situações em que o custo de um falso positivo é alto.

Vamos considerar um cenário em que temos um sistema de *scraper* de mensagens com o objetivo de detectar spam, no qual uma amostra pode ser classificada como spam (positivo) ou não (negativo). O custo de um falso positivo é alto porque pode resultar em mensagens importantes sendo filtradas erroneamente. Nesse caso, é crucial que o sistema tenha uma alta precisão; isto é, que classifique

corretamente a maioria das mensagens positivas como spam.

```
precisão = (amostras classificadas como positivas corretamente) / (amostras
    classificadas como positivas)
```

5.2.6 F1-Score

O F1-Score é uma métrica que combina a precisão e a revocação (recall) em uma única medida. Essa métrica é especialmente útil em problemas de classificação desbalanceados, nos quais a acurácia pode ser enganosa quando uma classe é muito mais frequente do que a outra.

Por exemplo, em um conjunto de dados com 99 amostras corretamente extraídas por um *scraper* e apenas 1 amostra errônea, poderíamos dizer que o *scraper* em questão teria uma acurácia de 99%; mas, seu desempenho real na extração seria muito ruim. Nesse caso, o F1-Score seria uma medida mais apropriada para ser utilizada, pois leva em consideração tanto a precisão quanto a revocação do modelo, proporcionando uma visão mais equilibrada da sua eficácia na classificação das amostras positivas corretamente.

```
F1-Score = 2 * (precisão * revocação) / (precisão + revocação)
```

5.3 RESULTADOS

A partir da análise do SACI na versão com expressões regulares, considerando as métricas citadas anteriormente, pode-se observar que os resultados obtidos mostraram um desempenho promissor na extração de ambas as complexidades, conforme demonstram as matrizes de confusão para tempo e espaço das Figuras 11 e 12.

		Valor Predito	
		Sim	**Não**
Real	**Sim**	391	11
	Não	88	11

Figura 11 – Matriz confusão, tempo

Fonte: elaborada pelo autor

		Valor Predito	
		Sim	**Não**
Real	**Sim**	419	1
	Não	47	34

Figura 12 – Matriz confusão, espaço

Fonte: elaborada pelo autor

A validação dos resultados foi feita de forma manual onde cada URL, onde ocorreu alguma extração de algoritmo, foi acessada e comparados os valores que foram extraídos com os valores presentes na página web.

Nota-se um bom desempenho na identificação de verdadeiros positivos para ambos os tipos de complexidade, com uma quantidade significativa de registros corretamente identificados pelo SACI. Porém, na complexidade de tempo observa-se um número considerável de falso-positivos, totalizando 88 casos. Isso indica que alguns dados errôneos foram incluídos na base de dados. A explicação para existência desse número um pouco mais elevado de falso-positivos é o uso do método de 'fallback', já que o mesmo parte do princípio de ser genérico e coletar a qualquer custo alguma complexidade.

Para falso-negativos, nota-se também sua existência significativa apenas para a complexidade de tempo, com 11 registros, o que sugere que alguns dados foram ignorados durante a extração. A baixa quantidade de verdadeiro-negativos na complexidade de tempo pode significar uma menor capacidade de filtrar dados irrelevantes.

Por outro lado, a complexidade de espaço apresentou um desempenho superior. A quantidade de verdadeiros negativos foi maior, com 34 registros, e os falsos negativos foram mínimos, com apenas 1 registro, mostrando uma melhor performance em comparação com a complexidade de tempo. Isso se deve ao fato de que a complexidade de espaço é apresentada de forma mais padronizada em todas as páginas, resultando em uma melhor identificação dos dados.

Na versão OpenAI notamos uma melhora geral em todos os parâmetros se comparados com a versão padrão, mas alguns apresentaram diferenças significativas.

		Valor Predito	
		Sim	**Não**
Real	**Sim**	451	6
	Não	32	12

Figura 13 – Matriz confusão OpenAI, tempo

Fonte: elaborada pelo autor

		Valor Predito	
		Sim	**Não**
Real	**Sim**	424	1
	Não	37	39

Figura 14 – Matriz confusão OpenAI, espaço

Fonte: elaborada pelo autor

Para a complexidade de tempo observou-se um grande incremento nos verdadeiro-positivos, de 391 para 451, além de uma diminuição significativa nos falso-positivos que reduziram de 88 para 32, sugerindo a diminuição da inclusão de dados incorretos na base. Para a complexidade de espaço notou-se também um aumento dos verdadeiro-positivos, de 419 para 424, além da redução dos

falso-positivos, de 47 para 37, e o aumento dos verdadeiro-negativos, de 34 para 39, evidenciando, assim, uma significativa melhora na capacidade da nova versão em evitar a inclusão de dados incorretos e discernir quando uma complexidade de espaço não está presente.

Tabela 2 – Métricas das complexidades de tempo e espaço

Métrica	**Tempo Regex**	**Espaço Regex**	**Tempo OpenAI**	**Espaço OpenAI**
Acurácia	0,8023	0,9041	0,9241	0,9241
Precisão	0,8162	0,8991	0,9337	0,9197
Revocação	0,9726	0,9976	0,9868	0,9976
F1 Score	0,8876	0,9458	0,9595	0,9571

Fonte: elaborada pelo autor

Os resultados alcançados pelo SACI são motivo de encorajamento. Eles evidenciaram alta precisão, revocação e acurácia em todas as complexidades analisadas. Notadamente, a implementação das técnicas da OpenAI trouxe melhorias substanciais sobre a versão padrão em vários aspectos críticos. Contudo, é importante reconhecer a dependência desses resultados à estrutura da fonte de dados. Mudanças futuras nos sites podem influenciar o desempenho da ferramenta.

Ao examinar as métricas de desempenho, confirma-se que o SACI se destaca como uma solução robusta e eficiente na coleta de complexidades. As altas taxas de precisão e revocação asseguram a habilidade em extrair informações com o mínimo de erros, um ponto crucial para a integridade dos dados coletados, garantindo a validade das informações e evitando dados imprecisos ou irrelevantes.

Além disso, a elevada acurácia e os índices de F1-Score reforçam a capacidade de realizar classificações precisas, otimizando o equilíbrio entre a detecção de dados relevantes e a exclusão de falso-positivos. A adoção de tecnologias avançadas em processamento de linguagem natural e aprendizado de máquina da OpenAI foi decisiva nesse avanço, proporcionando uma compreensão contextual mais refinada e uma extração mais acertada das complexidades.

A versão aprimorada do SACI, equipada com os modelos da OpenAI, mostrou uma redução expressiva tanto em falso-positivos quanto em falso-negativos, um diferencial que potencializa a confiabilidade dos dados extraídos. Tal performance sugere que, mesmo diante de estruturas de dados variáveis nos sites-alvo, a ferramenta possui a versatilidade para se adaptar e manter alto padrão de qualidade.

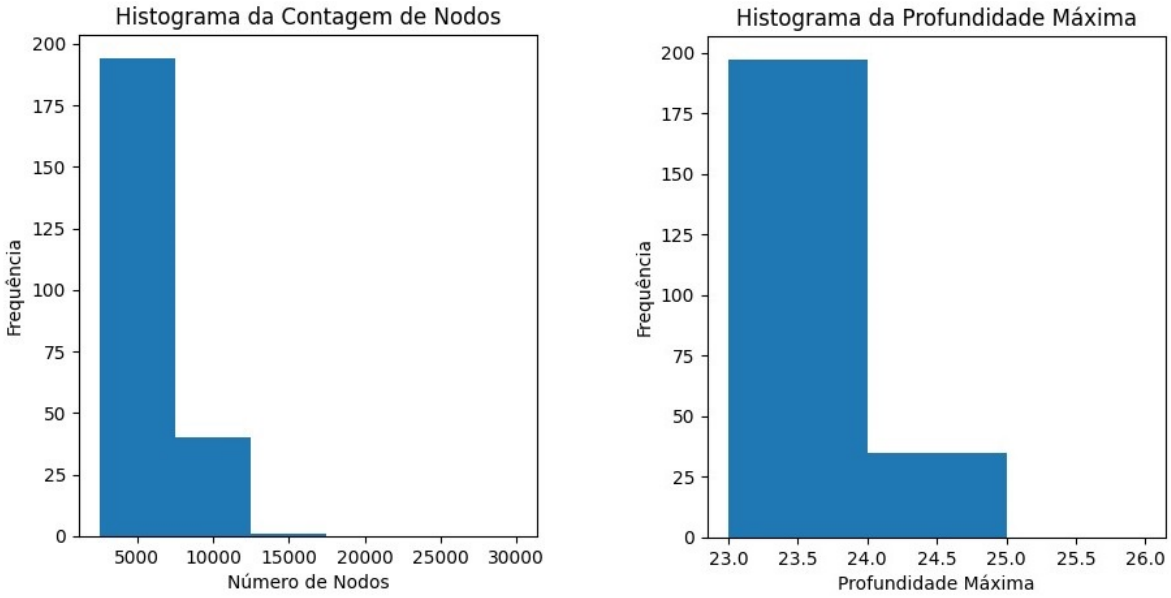
As métricas obtidas solidificam a confiança na ferramenta e pavimentam o caminho para seu contínuo desenvolvimento, com foco na adaptação para diversos domínios e na capacidade de enfrentar mudanças estruturais nos sites de origem.

5.4 FONTE DE DADOS

Todos os dados usados neste trabalho tiveram sua extração no dia '24/08/2023', conforme descrito previamente na Seção 4.4. Foi realizada uma análise sobre cada página onde houve a extração de uma ou mais complexidades. De um total de 235 URLs, com 501 algoritmos disponíveis através delas, para cada página, executou-se uma função que percorria todos os nodos da árvore HTML e retornava qual a profundidade máxima encontrada e quantidade de nodos existentes.

Tal análise foi feita porque o número de nodos HTML em uma página tem impacto direto na velocidade das operações de extração; quanto maior o número de nós, maior será o tempo necessário para percorrer a árvore de elementos e extrair as informações desejadas. A maioria das profundidades máximas fica em 24 nodos e, no caso da quantidade de nodos, entre 5 e 15 mil nodos é o mais frequente.

Figura 15 – Caracterização dos Nodos



Fonte: elaborada pelo autor

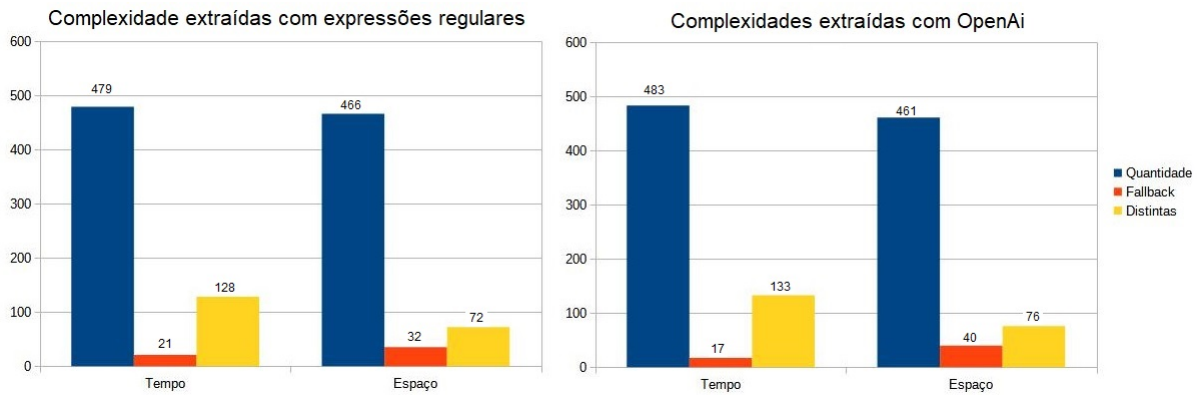
5.4.1 Conjunto de dados extraídos

A caracterização do conjunto de dados extraídos foi feita, primeiramente, avaliando o quanto a ferramenta de coleta utilizou da técnica de 'fallback' para extrair as complexidades. Os títulos dos algoritmos e os outros atributos coletados não fazem

parte da caracterização. Conforme a Figura **??**, 479 algoritmos tiveram suas complexidades de tempo extraídas e 466 suas complexidades de espaço; dentre estes, 21 fizeram uso do 'fallback' para extrair a complexidade de tempo e 32 para a de espaço. Foram obtidas 128 complexidades de tempo distintas e 72 complexidades de espaço distintas.

Um dos motivos para não se ter todas as complexidades de tempo extraídas com sucesso é a falta de um padrão na página. Em algumas páginas as complexidades estão escritas de maneira diferente, em locais diferentes, fazendo, assim, com que a expressão regular não as identifique. No caso da complexidade de espaço, alguns algoritmos realmente não a possuem e, em outros casos, ocorreu o mesmo problema que com a extração da complexidade de tempo.

Figura 16 – Complexidades extraídas



Fonte: elaborada pelo autor

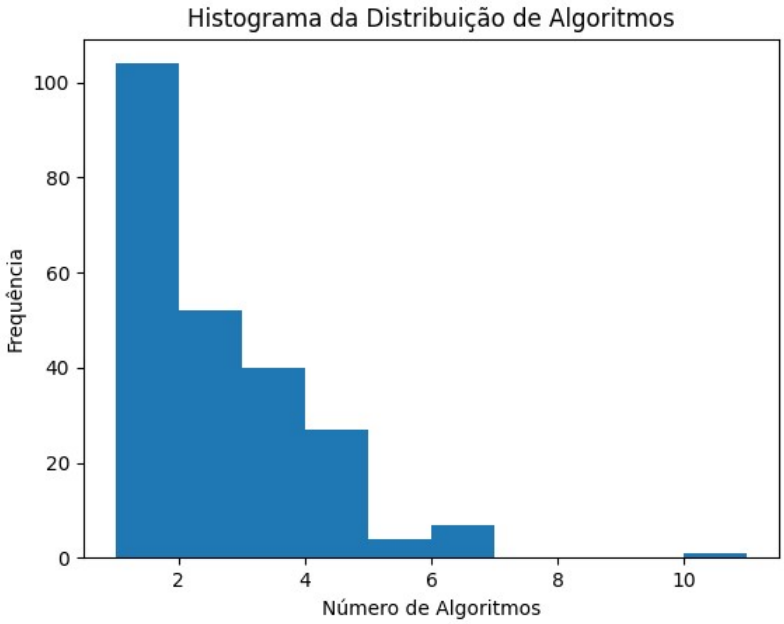
Na nova versão do método de extração, que incorpora as capacidades da OpenAI, observou-se um incremento na eficiência da extração de dados. Como exibido na Figura **??**, esta versão avançada conseguiu extrair com sucesso as complexidades de tempo de 483 algoritmos, e as complexidades de espaço de 461 algoritmos, superando um pouco a marca anterior.

Além disso, a dependência da técnica de 'fallback' foi reduzida para apenas 17 casos na extração das complexidades de tempo indicando, com isso, uma melhoria. Nas complexidades de espaço houve um pequeno aumento de 32 para 40 casos. Em termos de diversidade, o SACI identificou 133 complexidades de tempo distintas e 76 de espaço, um aumento em relação às 128 e 72 variantes respectivamente encontradas na versão que utilizava expressões regulares.

O número de URLs se manteve o mesmo em ambas as versões e é menor que o número de algoritmos extraídos, pois, cada página pode conter mais de um algoritmo; para exemplificar, foi gerado um histograma na Figura 17. Existem em média 2.27

algoritmos em cada página, sendo o máximo 10 e o mínimo 1.

Figura 17 – Distribuição de Algoritmos



Fonte: elaborada pelo autor

Ao realizar um teste de uma das URLs do site[1] com ambas as versões, notamos que o *scraper* retorna como valores de complexidade 'O(N)' para a de tempo e 'O(1)' para a de espaço. Acessando o site através da própria URL exposta no retorno, pode-se confirmar que os dados foram extraídos com sucesso, conforme os valores da página exibida na Figura 18.

[1] https://www.geeksforgeeks.org/linear- search/

Figura 18 – Linear Search

Below is the implementation of the above approach:

C C++ Java Python3 C# PHP Javascript

```python
# Python3 code to linearly search x in arr[].
# If x is present then return its location,
# otherwise return -1


def search(arr, N, x):

    for i in range(0, N):
        if (arr[i] == x):
            return i
    return -1


# Driver Code
if __name__ == "__main__":
    arr = [2, 3, 4, 10, 40]
    x = 10
    N = len(arr)

    # Function call
    result = search(arr, N, x)
    if(result == -1):
        print("Element is not present in array")
    else:
        print("Element is present at index", result)
```

Output

```
Element is present at index 3
```

Time complexity: O(N)

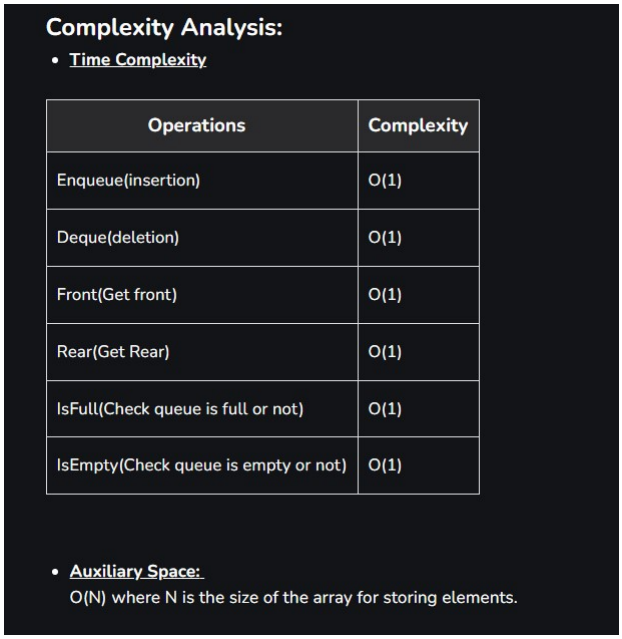
Auxiliary Space: O(1)

Fonte: elaborada pelo autor

Um exemplo de caso com problemas seria o do algoritmo encontrado no site [2], onde as complexidades de tempo aparecem em uma lista, cada uma associada a uma determinada operação, o que causa problemas quando o código tenta realizar a extração, apresentado na Figura 19.

[2] https://www.geeksforgeeks.org/introduction-and-array-implementation-of- queue/

Figura 19 – Problema na extração



Complexity Analysis:

- **Time Complexity**

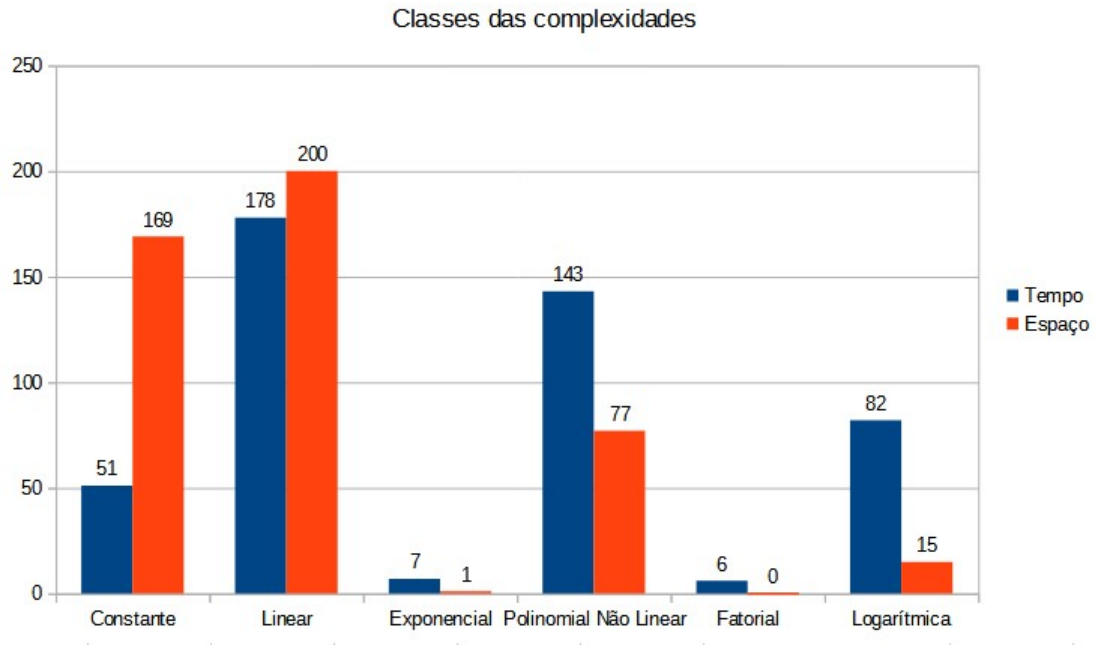
Operations	Complexity
Enqueue(insertion)	O(1)
Deque(deletion)	O(1)
Front(Get front)	O(1)
Rear(Get Rear)	O(1)
IsFull(Check queue is full or not)	O(1)
IsEmpty(Check queue is empty or not)	O(1)

- **Auxiliary Space:**
  O(N) where N is the size of the array for storing elements.

Fonte: elaborada pelo autor

Para se ter um panorama mais detalhado, também foi analisado a distribuição das classes de complexidade extraídas e chegou-se ao resultado apresentado na Tabela 20.

Figura 20 – Classe das complexidades



Fonte: elaborada pelo autor

Conforme Tabela 20, ao analisar as complexidades de tempo, observa-se que a classe *linear* é a predominante com 178 entradas. Ela é seguida pelas classes *polinomial* com 143 entradas; *logarítmica* com 82; e *constante* com 51. As classes *fatorial* e *exponencial* apresentam números mais baixos, com 6 e 7 entradas, respectivamente. Isso sugere que, embora haja um esforço para representar todas as classes na base de dados, algumas têm uma presença reduzida no site analisado.

Um fato notável é que as complexidades *fatorial* e *exponencial* são, geralmente, menos eficientes, resultando em sua menor utilização em aplicações práticas. Esta tendência pode ser interpretada como uma inclinação da comunidade em favor de abordagens que evitem tais complexidades ineficientes. Ao explorar as classes de complexidade de espaço, conforme indicado na mesma tabela, percebe-se que a classe *linear* se sobressai, contabilizando 200 entradas. A classe *constante* também apresenta uma boa representatividade, com 169 entradas, seguida pela *polinomial*, com 77. As classes *exponencial* e *fatorial* são menos frequentes, ambas com apenas 1 e 0 entrada, respectivamente, enquanto a *logarítmica* registra 15.

Esta distribuição reforça a ideia de que há uma preferência por complexidades mais práticas e gerenciáveis em termos de uso de espaço. A baixa representatividade das complexidades *exponencial* e *fatorial* reflete a natureza ineficiente destas quando se trata de consumo de memória. A presença mais discreta da complexidade *logarítmica*, por sua vez, pode indicar sua aplicação em contextos mais específicos.

Em resumo, os dados mostram uma clara inclinação da base para soluções

que equilibram eficiência e praticidade, priorizando algoritmos que otimizem o uso de recursos. A versão com OpenAI não apresentou diferenças significativas. Algumas entradas diminuíram e outras aumentaram, mas a ordem de representatividade se manteve.

A classificação das complexidades em ambas versões foi feita através de expressões regulares, exibidas no Código 7 utilizando-se de um bloco de código de condicionais. Caso a expressão capturasse algo na complexidade, significa que tal complexidade é pertencente a tal categoria.

```
1    Constante = r'O\(1\)'
     Linear = r'O\(\w\)'
3    Exponencial = r'O\(\d\^?\w\)'
     Fatorial = r'O\(.*!.*\)'
5    Logarítmica = r'O\(.*log.*\)'
     Polinomial = r'O\(\w\s*[\^\*\+]?\s*[\d\w\s]+.*\)' e r'O\(\d+\(\w\*\w\)'
```

Código 7 – Regex de classificação de complexidades

6 CONCLUSÕES E TRABALHOS FUTUROS

A criação de ferramentas capazes de identificar a complexidade de um algoritmo e, através disso, gerar *insights* para o usuário final sobre o código que está sendo analisado, requer uma base de dados com algoritmos e suas complexidades, de forma que tais ferramentas possam aproveitar-se e aprender com eles a partir de técnicas de inteligência artificial.

O SACI apresentou resultados bastantes satisfatórios. A quantidade de códigos e complexidades coletadas das páginas do site formam uma base de dados adequada e as métricas calculadas comprovam a eficácia da ferramenta.

Durante o desenvolvimento, foi perceptível a dificuldade existente na construção de um *scraper* genérico, principalmente devido à grande diversidade de layouts de páginas. No entanto, a forma como se desenhou a ferramenta permitiu a reutilização de certos trechos de códigos, facilitando, com isso, a implementação futura para outras páginas. O maior obstáculo, porém, consistia em detectar e extrair os elementos desejados. Neste caso, a estratégia escolhida foi o uso de expressões regulares juntamente com o método 'fallback', utilizado quando as expressões regulares não obtinham êxito na extração.

Quanto a questões de impedimentos por 'anti-bots' ou sistemas do gênero, estes não implicaram em problemas para a página coletada, bastando o simples envio dos cabeçalhos adequados e a criação de login para evitar futuros transtornos. Todavia, é preciso atentar-se para a quantidade de requisições enviadas simultaneamente, pois a ferramenta faz isso de forma assíncrona e pode ser facilmente notado caso exceda um número normal. Assim sendo, criou-se um argumento para limitar as requisições e, também, para adicionar um intervalo entre as mesmas, mitigando, assim, possíveis eventos de alerta.

Por questões de tempo ou por escolhas quando da implementação, algumas funções da ferramenta não foram desenvolvidas completamente, estando, portanto, passíveis de melhorias em futuros projetos e desenvolvimentos. São elas:

1. Melhorar as métricas: Atualmente o SACI só está levando em questão as complexidades de tempo e espaço em suas métricas, é possível adicionar mais atributos a serem acompanhados.

2. Melhorar as técnicas de extração: As expressões regulares e lógicas utilizadas nas extrações dos dados desejados podem ser redesenhadas e melhoradas a fim de obter mais sucesso na tarefa. Também é possível usar inteligência artificial além do Chat GPT para as extrações, treinando um modelo especificamente para o caso de uso que identifique os dados relevantes da página e usar isso para extraí-los.

3. Fonte de dados: Realizar a implementação em mais fontes de dados a fim de coletar mais códigos e complexidades, gerando uma base de dados mais rica.

Além destas melhorias, a base de dados proveniente dos resultados já existentes na ferramenta contém informações valiosas para pesquisadores da área, que podem utilizar os 501 algoritmos extraídos em trabalhos que dependam de uma base de dados com tais características e, desta forma, melhorá-los, otimizá-los ou desenvolver algo novo a partir destes. Dentre os possíveis usos para essa base destacam-se:

1. Análise Comparativa de Algoritmos: Uma das aplicações mais óbvias que utiliza uma base de dados de algoritmos e suas complexidades, é a análise comparativa de algoritmos. Os dados fornecidos podem ser usados para comparar o desempenho de diferentes algoritmos em termos de tempo de execução e uso de espaço. Isso permite identificar algoritmos mais eficientes para determinados problemas e fornece *insights* sobre as características de desempenho de cada algoritmo.

2. Otimização de Algoritmos: É possível realizar trabalhos de otimização de algoritmos com as informações obtidas. Os dados podem revelar oportunidades para melhorar algoritmos existentes, seja reduzindo a complexidade de tempo e espaço ou refinando suas implementações. Por exemplo, é possível identificar partes do algoritmo que podem ser substituídas por abordagens mais eficientes ou adaptar o algoritmo para melhor lidar com entradas específicas para casos específicos.

3. Previsão de Desempenho de Algoritmos: A base de dados pode ser utilizada para construir modelos preditivos que estimem o desempenho de algoritmos com base em suas características e nos tamanhos dos conjuntos de dados de entrada. Isso pode ser útil para tomar decisões sobre o uso de um determinado algoritmo em uma situação específica. Os modelos preditivos podem ser usados para prever o tempo de execução ou o consumo de espaço de um algoritmo com base em dados históricos e características conhecidas do problema.

4. Estudo Teórico de Algoritmos: Também pode-se usar a base para fins teóricos. É possível realizar estudos para investigar as características dos algoritmos em relação a suas complexidades de tempo e espaço. Esses estudos podem envolver a análise de classes de algoritmos, a identificação de limites teóricos para determinados problemas computacionais ou a análise de propriedades gerais dos algoritmos em relação às suas complexidades.

5. Desenvolvimento de Novos Algoritmos: Os dados disponíveis na base de dados podem servir como inspiração e orientação para o desenvolvimento de novos

algoritmos. Ao estudar as complexidades de tempo e espaço de algoritmos existentes, é possível identificar lacunas ou problemas não resolvidos e propor novas soluções. Por exemplo, gerar um novo algoritmo de *sorting* baseado nas melhores características dos já existentes.

A base de dados construída oferece uma riqueza de informações que podem ser exploradas em diferentes áreas. A análise comparativa de algoritmos, a otimização, a previsão de desempenho, o estudo teórico e o desenvolvimento de novos algoritmos são apenas alguns exemplos dos trabalhos possíveis que podem ser realizados a partir dessa base de dados.

REFERÊNCIAS

BRANCO, Arthur Machado. Ferramenta para coleta e comparação de dados de publicações acadêmicas dos professores com o currículo lattes. *In:* UFSC. Citado na p. 23.

CORMEN, Thomas H. *et al.* **Algoritmos: Estruturas de Dados e Algoritmos para Resolução de Problemas**. Rio de Janeiro: LTC, 2009. Citado na p. 12.

CORMEN, Thomas H. *et al.* **Introduction to Algorithms**. 2nd. [*S.l.*]: The MIT Press, 2001. ISBN 0262032937. Disponível em: <http://www.amazon.com/Introduction-Algorithms-Thomas-H-Cormen/dp/0262032937%3FSubscriptionId%3D13CT5CVB80YFWJEPWS02%26tag%3Dws%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D0262032937>. Citado na p. 17.

DIOUF, Rabiyatou *et al.* Web Scraping: State-of-the-Art and Areas of Application. *In:* 2019 IEEE International Conference on Big Data (Big Data). [*S.l.*: *s.n.*], 2019. P. 6040–6042. DOI: 10.1109/BigData47090.2019.9005594. Citado na p. 15.

DIOUF, Rabiyatou *et al.* Web scraping: state-of-the-art and areas of application. *In:* IEEE. 2019 IEEE International Conference on Big Data (Big Data). [*S.l.*: *s.n.*], 2019. P. 6040–6042. Citado na p. 16.

HALL, Mark; CHAPMAN, Barry. Measuring algorithm complexity. **ACM SIGPLAN Notices**, ACM Press, v. 48, n. 1, p. 16–23, 2015. Citado na p. 12.

KERNER, Sean Michael. **What are large language models (LLMs)?** 2023. Disponível em: <https://www.techtarget.com/whatis/definition/large-language-model-LLM>. Acesso em: 5 nov. 2022. Citado na p. 20.

KLEINBERG, J.; TARDOS, É. **Algorithm Design**. [*S.l.*]: Pearson/Addison-Wesley, 2006. (Alternative Etext Formats). ISBN 9780321295354. Disponível em: <https://books.google.com.br/books?id=OiGhQgAACAAJ>. Citado nas pp. 17, 18.

LESSA, Marcos Aurélio. CrawlEX: uma ferramenta para extração de dados na web configurável através de exemplos. *In:* UFSC. Citado na p. 24.

MATHIAS, Gilney; DORNELES, Carina. qFEx - um crawler para busca e extração de questionários de pesquisa em documentos HTML. *In:* ANAIS do III Dataset Showcase Workshop. Rio de Janeiro: SBC, 2021. P. 1–10. DOI: 10.5753/dsw.2021.17409. Disponível em: <https://sol.sbc.org.br/index.php/dsw/article/view/17409>. Citado na p. 22.

MOONEY, Stephen J; WESTREICH, Daniel J; EL-SAYED, Abdulrahman M. Epidemiology in the era of big data. **Epidemiology (Cambridge, Mass.)**, NIH Public Access, v. 26, n. 3, p. 390, 2015. Citado na p. 15.

PFITSCHER, Ricardo J. *et al.* Estimating Code Running Time Complexity with Machine Learning. *In:* NALDI, Murilo C.; BIANCHI, Reinaldo A. C. (Ed.). **Intelligent Systems**. Cham: Springer Nature Switzerland, 2023. P. 400–414. Citado na p. 12.

RODENBUSCH, Gabriel Braun. **Desenvolvimento de API para predição da complexidade de tempo de execução de códigos por meio de AutoML**. [*S.l.*: *s.n.*], 2023. TCC (graduação) - Universidade Federal de Santa Catarina, Campus Joinville, Engenharia Mecatrônica. Disponível em: <https://repositorio.ufsc.br/handle/123456789/252875>. Acesso em: 7 dez. 2023. Citado nas pp. 13, 24.

SEDGEWICK, Robert. Analyzing algorithms. **Communications of the ACM**, ACM Press, v. 52, n. 1, p. 139–148, 2009. Citado na p. 12.

SERVIAN, Alexandre. **Regex: Um guia prático para expressões regulares**. 2020. Disponível em: <https://medium.com/xp-inc/regex-um-guia-pratico-para-express%C3%B5es-regulares-1ac5fa4dd39f>. Acesso em: 10 dez. 2022. Citado na p. 19.

SIKKA, Jagriti *et al.* Learning Based Methods for Code Runtime Complexity Prediction. *In:* SPRINGER. EUROPEAN Conference on Information Retrieval. [*S.l.*: *s.n.*], 2020. P. 313–325. Citado nas pp. 12, 17.

SIKKA, Jagriti *et al.* Learning Based Methods for Code Runtime Complexity Prediction. *In:* JOSE, Joemon M. *et al.* (Ed.). **Advances in Information Retrieval**. Cham: Springer International Publishing, 2020. P. 313–325. Citado na p. 22.

SNYDER, Russell. **Web search engine with graphic snapshots**. [*S.l.*]: Google Patents, nov. 2003. US Patent 6,643,641. Citado na p. 16.

WANG, Meng *et al.* A Survey on Large-Scale Machine Learning. **IEEE Transactions on Knowledge and Data Engineering**, v. 34, n. 6, p. 2574–2594, 2022. DOI: 10.1109/TKDE.2020.3015777. Citado na p. 13.

ZHAO, Bo. Web scraping. **Encyclopedia of big data**, Springer Living ed. Cham, p. 1–3, 2017. Citado nas pp. 13, 15.

ZHAO, Wayne Xin *et al.* **A Survey of Large Language Models**. [*S.l.*: *s.n.*], 2023. arXiv: 2303.18223 [cs.CL]. Citado na p. 21.

Apêndice A - Código Fonte Versão Expressões Regulares

- saci/requirements.txt

```
aiohttp
beautifulsoup4
black
pandas
pydantic
pymongo
pyppeteer
requests
scikit-learn
scipy
seaborn
mlxtend
requests-html
pip-tools
openai
langchain
asyncio-throttle
```

- saci/makefile

```
PYTHON :=
ifeq ($(OS),Windows_NT)
  PYTHON = venv/Scripts/python
  PIP = venv/Scripts/pip
else
  PYTHON = venv/bin/python3
  PIP = venv/bin/pip
endif

requirements: requirements.txt
  $(PIP) install -r requirements.txt
```

```
13 run_scraper:
     $(PYTHON) -m main
15
   scrape_url:
17   $(PYTHON) -m scraper.scripts --script=scrape_url --url=$(url) --scraper=$
       (scraper)

19 results_analysis:
     $(PYTHON) -m scraper.scripts --script=results_analysis --scraper=$(
       scraper)
21
   manual_results_boilerplate:
23   $(PYTHON) -m scraper.scripts --script=manual_results_boilerplate --
       scraper=$(scraper)
```

- saci/main.py

```
1 from asyncio import get_event_loop

3 from scraper.runner import run_scrapers

5 if __name__ == "__main__":
      loop = get_event_loop()
7     loop.run_until_complete(run_scrapers())
```

- saci/scraper/__init__.py
- saci/scraper/database.py

```
1 import os
  from typing import Dict, List
3
  from pymongo import MongoClient
5 from pymongo.collection import Collection
  from pymongo.operations import UpdateOne
7
  from scraper.settings import DATABASE_NAME
9
  FIELDS_TO_UPDATE = [
11     "name",
       "time_complexity",
```

```
    "trustable_time_complexity",
    "space_complexity",
    "trustable_space_complexity",
]


class ScraperDatabase:
    def __init__(self):
        self.client: MongoClient = MongoClient(
            os.environ.get("MONGO_CONNECTION_STRING"),
                serverSelectionTimeoutMS=1
        )
        self.database = self.client[DATABASE_NAME]

    def _get_collection(self, collection_name: str) -> Collection:
        return self.database[collection_name]

    def update_database(self, collection_name: str, data: List[Dict[str,
        any]]):
        collection = self._get_collection(collection_name)
        operations = [
            UpdateOne(
                filter={"url": item["url"], "codes": item["codes"]},
                update={"$set": {field: item[field] for field in
                    FIELDS_TO_UPDATE}},
                upsert=True,
            )
            for item in data
        ]
        collection.bulk_write(operations)
```

- saci/scraper/exceptions.py

```
class InvalidUrlException(Exception):
    def __init__(self, url):
        super().__init__(f"The provided URL '{url}' is not valid.")


class TooManyRetrysException(Exception):
    pass


class FailedExtraction(Exception):
    """Failed the extraction proccess"""
```

```
    pass


class FailedComplexityExtraction(Exception):
    """Failed to extract the complexitys"""

    pass
```

- saci/scraper/runner.py

```
from asyncio import gather

from scraper.database import ScraperDatabase
from scraper.settings import USE_MONGO_DATABASE
from scraper.utils import write_results_to_json
from scraper.websites import websites


async def run_scrapers():
    scrapers_results = await gather(*[scraper() for scraper in websites.
        values()])
    for name, data in zip(websites.keys(), scrapers_results):
        if USE_MONGO_DATABASE:
            database = ScraperDatabase()
            database.update_database(name, [result.dict() for result in
                data])
        else:
            write_results_to_json(
                name,
                sorted([result.dict() for result in data], key=lambda alg:
                    alg["url"]),
            )
```

- saci/scraper/scripts.py

```
import argparse
import pprint
from asyncio import get_event_loop
from importlib import import_module
from scraper.observability.metrics import (
    make_results_analysis,
    make_manual_results_boilerplate,
```

```
)


def extract_single_url(url: str, scraper: str):
    module = import_module(f"scraper.websites.{scraper}")
    spider = getattr(module, "spider")
    loop = get_event_loop()
    result = loop.run_until_complete(spider.run(url))
    pprint.PrettyPrinter(indent=4).pprint(result)


def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("--script")
    parser.add_argument("--url", help="URL to be extracted.")
    parser.add_argument("--scraper", help="Scraper to be used.")
    args = parser.parse_args()

    actions = {
        "scrape_url": lambda: extract_single_url(args.url, args.scraper),
        "results_analysis": lambda: make_results_analysis(args.scraper),
        "manual_results_boilerplate": lambda:
            make_manual_results_boilerplate(
             args.scraper
        ),
    }

    action = actions.get(args.script)
    if action:
        action()
    else:
        print(f"Unknown script: {args.script}")


if __name__ == "__main__":
    main()
```

- saci/scraper/settings.py

```
DATABASE_NAME = "COMPLEXITY_SCRAPER"

USE_MONGO_DATABASE = False
```

- saci/scraper/utils.py

```python
import json

from scraper.exceptions import TooManyRetrysException
from scraper.observability.log import scraper_log as log


def retry(
    times: int,
    raise_exception=True,
    return_value=None,
):
    def func_wrapper(f):
        async def wrapper(*args, **kwargs):
            for _ in range(times):
                try:
                    return await f(*args, **kwargs)
                except Exception:
                    pass

            if not raise_exception:
                return return_value
            else:
                raise TooManyRetrysException

        return wrapper

    return func_wrapper


def remove_duplicates(data: list) -> list:
    return list(set(data))


def open_results_from_json(file_path: str) -> dict:
    with open(file_path, "r") as file:
        file_contents = file.read()
        results = json.loads(file_contents)
    return results


def write_results_to_json(file_path: str, data: list):
    with open(f"{file_path}.json", "w") as file:
        json.dump(data, file)
```

- saci/observability/__init__.py

- saci/observability/log.py

```
import platform

from logging import DEBUG, Filter, Formatter, StreamHandler, getLogger

from scraper.observability.settings import SCRAPER_LOG_FORMAT,
    SESSION_LOG_FORMAT


class AddScraperName(Filter):
    def filter(self, record):
        if not hasattr(record, "scraper"):
            split_char = "\\" if "Windows" in platform.platform() else "/"
            record.scraper = record.pathname.split(split_char)[-2]
        return True


def build_scraper_log(logger_name):
    log_handler = StreamHandler()
    log_handler.setLevel(DEBUG)
    log_handler.setFormatter(Formatter(SCRAPER_LOG_FORMAT))

    log = getLogger(logger_name)
    log.addFilter(AddScraperName())
    log.addHandler(log_handler)
    return log


def build_session_log(logger_name):
    log_handler = StreamHandler()
    log_handler.setLevel(DEBUG)
    log_handler.setFormatter(Formatter(SESSION_LOG_FORMAT))

    log = getLogger(logger_name)
    log.addHandler(log_handler)
    return log


scraper_log = build_scraper_log("scraper_log")
session_log = build_session_log("session_log")
```

- saci/observability/metrics.py

```
import math
import re
from collections import Counter
from typing import Any, Dict, List, Tuple, Union

import matplotlib.pyplot as plt
import numpy
import requests
from bs4 import BeautifulSoup, NavigableString, Tag
from mlxtend.plotting import plot_confusion_matrix

from scraper.utils import (
    open_results_from_json,
    remove_duplicates,
    write_results_to_json,
)


def make_results_caracterization(scraper: str) -> Dict[str, int]:
    results = open_results_from_json(f"./results/{scraper}.json")

    return {
        "Distinct Time Complexities": len(
            {algorithm["time_complexity"] for algorithm in results}
        ),
        "Distinct Space Complexities": len(
            {algorithm["space_complexity"] for algorithm in results}
        ),
        "Non Trusted Time Complexities": sum(
            [1 for algorithm in results if not algorithm["
                trustable_time_complexity"]]
        ),
        "Non Trusted Space Complexities": sum(
            [1 for algorithm in results if not algorithm["
                trustable_space_complexity"]]
        ),
    }


def make_completition_rate(scraper: str) -> Dict[str, int]:
    results = open_results_from_json(f"./results/{scraper}.json")

    total = len(results)
    time_complexity, space_complexity = sum(
        1 for algorithm in results if not algorithm["time_complexity"]
    ), sum(1 for algorithm in results if not algorithm["space_complexity"])
```

```
    urls_with_problem = remove_duplicates(
        [
            algorithm["url"]
            for algorithm in results
            if not algorithm["time_complexity"] or not algorithm["
               space_complexity"]
        ]
    )

    return {
        "Total Algorithms": total,
        "Time Complexity Extracted": f"{total - time_complexity} extracted
           - {100 - (time_complexity / total) * 100}%",
        "Space Complexity Extracted": f"{total - space_complexity}
           extracted - {100 - (space_complexity / total) * 100}%",
        "URLs with problem": urls_with_problem,
    }


def calculate_metrics(
    true_positive: int, false_positive: int, true_negative: int,
       false_negative: int
) -> Dict[str, float]:
    accuracy = (true_positive + true_negative) / (
        true_positive + true_negative + false_positive + false_negative
    )
    precision = true_positive / (true_positive + false_positive)
    recall = true_positive / (true_positive + false_negative)
    f1_score = (2 * precision * recall) / (precision + recall)

    return {
        "True Positive": true_positive,
        "False Positive": false_positive,
        "True Negative": true_negative,
        "False negative": false_negative,
        "Accuracy": accuracy,
        "Precision": precision,
        "Recall": recall,
        "F1 Score": f1_score,
    }


def plot_confusion(
    title: str,
    true_positive: int,
    false_negative: int,
    false_positive: int,
    true_negative: int,
```

```
):
    array = numpy.array(
        [[true_positive, false_negative], [false_positive, true_negative]]
    )

    fig, ax = plot_confusion_matrix(conf_mat=array)
    plt.title("Space Complexity Confusion Matrix")
    plt.show()


def make_confusion_matrix(scraper: str) -> Dict[str, Dict[str, Any]]:
    manual_results = open_results_from_json(f"./results/manual_{scraper}.
        json")
    results = open_results_from_json(f"./results/{scraper}.json")

    time_metrics = {
        "true_positive": 0,
        "false_positive": 0,
        "true_negative": 0,
        "false_negative": 0,
    }

    space_metrics = {
        "true_positive": 0,
        "false_positive": 0,
        "true_negative": 0,
        "false_negative": 0,
    }

    for result, manual_result in zip(results, manual_results):
        # Time complexity
        if time_complexity := result["time_complexity"]:
            if time_complexity == manual_result["time_complexity"]:
                time_metrics["true_positive"] += 1
            else:
                time_metrics["false_positive"] += 1
        else:
            if not manual_result["time_complexity"]:
                time_metrics["true_negative"] += 1
            else:
                time_metrics["false_negative"] += 1

        # Space complexity
        if space_complexity := result["space_complexity"]:
            if space_complexity == manual_result["space_complexity"]:
                space_metrics["true_positive"] += 1
            else:
                space_metrics["false_positive"] += 1
```

```
136         else:
                if not manual_result["space_complexity"]:
138                 space_metrics["true_negative"] += 1
                else:
140                 space_metrics["false_negative"] += 1

142     # Calculating metrics
        time_complexity_metrics = calculate_metrics(**time_metrics)
144     space_complexity_metrics = calculate_metrics(**space_metrics)

146     # Plotting the confusion matrices
        plot_confusion("Time Complexity Confusion Matrix", **time_metrics)
148     plot_confusion("Space Complexity Confusion Matrix", **space_metrics)

150     # Return result
        return {
152         "Time Complexity": time_complexity_metrics,
            "Space Complexity": space_complexity_metrics,
154     }


156
    def make_manual_results_boilerplate(scraper: str) -> None:
158     results = open_results_from_json(f"./results/{scraper}.json")

160     manual_results = [
            {"url": alg["url"], "time_complexity": " ", "space_complexity": " "
                }
162         for alg in results
        ]
164
        write_results_to_json(f"./results/manual_{scraper}", manual_results)
166


168 def make_complexitys_classification(scraper: str) -> Dict[str, Dict[str,
    int]]:
        results = open_results_from_json(f"./results/{scraper}.json")
170
        time_classification = classify_complexity(
172         [result["time_complexity"] for result in results]
        )
174     space_classification = classify_complexity(
            [result["space_complexity"] for result in results]
176     )

178     return {
            "Time Complexity Classification": time_classification,
180         "Space Complexity Classification": space_classification,
        }
```

```
182
184 def classify_complexity(complexities: List[str]) -> Dict[str, int]:
        constant, linear, exponential, polynomial, factorial, logarithmic = 0,
            0, 0, 0, 0, 0
186     for complexity in complexities:
            if complexity:
188             if re.match(r"O\(1\)", complexity):
                    constant += 1
190             elif re.match(r"O\(\w\)", complexity):
                    linear += 1
192             elif re.match(r"O\(.*!.*\)", complexity):
                    factorial += 1
194             elif re.match(r"O\(.*log.*\)", complexity):
                    logarithmic += 1
196             elif re.match(r"O\(\d\^?\w\)", complexity):
                    exponential += 1
198             elif re.match(r"O\(\w\s*[\^\*\+]?\s*[\d\w\s]+.*\)", complexity)
                    or re.match(
                    r"O\(\d+\(\w\*\w\)", complexity
200             ):
                    polynomial += 1
202
        return {
204         "Constant complexity": constant,
            "Linear complexity": linear,
206         "Exponential complexity": exponential,
            "Polynomial complexity": polynomial,
208         "Factorial complexity": factorial,
            "Log complexity": logarithmic,
210     }


212
    def make_algorithms_histogram(scraper: str):
214     results = open_results_from_json(f"./results/{scraper}.json")

216     count = Counter([result["url"] for result in results]).values()
        plt.hist(count)
218     plt.xlabel("Número de Algoritmos")
        plt.ylabel("Frequência")
220     plt.title("Histograma da Distribuição de Algoritmos")
        plt.show()
222


224 def traverse_tree(
        node: Union[Tag, NavigableString], depth: int, depths: Dict[str, List[
            int]]
226 ) -> None:
```

```
    """
228     Traverse the BeautifulSoup tree and populate the depths dictionary with
        tag names and their depths.

230     Args:
            node: The current BeautifulSoup node.
232         depth: Current depth of traversal.
            depths: Dictionary to populate with depths.
234     """
        if isinstance(node, Tag):  # Using isinstance to check if node is a Tag
            type
236         if node.name not in depths:
                depths[node.name] = []
238         depths[node.name].append(depth)

240         for child in node.children:
                traverse_tree(child, depth + 1, depths)
242

244 def calculate_html_nodes_depth(scraper: str) -> None:
        results = open_results_from_json(f"./results/{scraper}.json")
246     urls = list(set(result["url"] for result in results))

248     all_max_depths: List[int] = []
        all_num_nodes: List[int] = []
250
        for url in urls:
252         response = requests.get(url)
            soup = BeautifulSoup(response.content, "html.parser")
254
            depths: Dict[str, List[int]] = {}
256         traverse_tree(soup, 0, depths)

258         max_depth = max(max(depths[tag]) for tag in depths)
            num_nodes = sum(len(depths[tag]) for tag in depths)
260
            all_max_depths.append(max_depth)
262         all_num_nodes.append(num_nodes)

264     _, axs = plt.subplots(1, 2, figsize=(10, 5))
        axs[0].hist(all_max_depths, bins=[23, 24, 25, 26])
266     axs[0].set_xlabel("Profundidade Máxima")
        axs[0].set_ylabel("Frequência")
268     axs[0].set_title("Histograma da Profundidade Máxima")

270     axs[1].hist(all_num_nodes, bins=[2500, 7500, 12500, 17500, 22500,
            30000])
        axs[1].set_xlabel("Número de Nodos")
```

```
    axs[1].set_ylabel("Frequência")
    axs[1].set_title("Histograma da Contagem de Nodos")

    plt.suptitle("Combined Histogram for All URLs")
    plt.show()


def make_results_analysis(scraper: str):
    write_results_to_json(
        f"./results/{scraper}_analysis",
        [
            {
                **make_completition_rate(scraper),
                **make_results_caracterization(scraper),
                **make_confusion_matrix(scraper),
                **make_complexitys_classification(scraper),
            }
        ],
    )
    # calculate_html_nodes_depth(scraper)
    # make_algorithms_histogram(scraper)
```

- saci/observability/settings.py

```
SCRAPER_LOG_FORMAT = (
    "%(asctime)s - %(levelname)s - Scraper: %(scraper)s - Message: %(
        message)s"  # noqa
)
SESSION_LOG_FORMAT = (
    "%(asctime)s - %(levelname)s - Session - Message: %(message)s"  # noqa
)
```

- saci/schema/__init__.py

```
```

- saci/schema/data_schemas.py

```
from typing import Optional

```

```python
from pydantic import Field
from pydantic import BaseModel


class ScrapedCode(BaseModel):
    """The scraped codes of a given algorithm"""

    code: str = Field(
        default="", description="The code of the algorithm in a certain
            language"
    )
    comments: str = Field(default="", description="The first few comments
        in the code")


class ScrapedAlgorithm(BaseModel):
    """The final result of the scraping operations, an extracted algorithm
        with its attributes"""

    name: str = Field(default="", description="The name of the scraped
        algorithm")
    time_complexity: Optional[str] = Field(
        default="", description="The time complexity of the scraped
            algorithm"
    )
    trustable_time_complexity: bool = Field(
        default=True,
        description="If the given time complexity used a trustable
            extraction method",
    )
    space_complexity: Optional[str] = Field(
        default="", description="The space complexity of the algorithm"
    )
    trustable_space_complexity: bool = Field(
        default=True,
        description="If the given space complexity used a trustable
            extraction method",
    )
    url: str = Field(default="", description="The space complexity of the
        algorithm")
    codes: dict[str, ScrapedCode] = Field(
        default=dict(), description="The algorithm implementations"
    )
```

- saci/session/__init__.py

- saci/session/exceptions.py

```
class InvalidUrlException(Exception):
    def __init__(self):
        self.message = "Please provide a valid URL string!"
        super().__init__(self.message)


class MissingMethodException(Exception):
    def __init__(self):
        self.message = "Please provide a request method!"
        super().__init__(self.message)


class UnsupportedMethodException(Exception):
    def __init__(self):
        self.message = "Please provide a supported method!"
        super().__init__(self.message)


class MissingArgumentException(Exception):
    def __init__(self, argument):
        self.message = f"Please provide the missing argument: {argument}!"
        super().__init__(self.message)


class InvalidArgumentType(Exception):
    def __init__(self, type):
        self.message = f"Please provide the correct argument type: {type}!"
        super().__init__(self.message)
```

- saci/session/http_session.py

```
import asyncio
import inspect

from aiohttp import ClientSession
from requests_html import AsyncHTMLSession
from yarl import URL
```

```
from scraper.exceptions import TooManyRetrysException
from scraper.observability.log import session_log as log
from scraper.session.exceptions import (
    InvalidArgumentType,
    InvalidUrlException,
    MissingArgumentException,
    MissingMethodException,
    UnsupportedMethodException,
)
from scraper.session.response import Response
from scraper.session.settings import MAX_REDIRECTS, REQUIRED_REQUEST_ARGS
from scraper.utils import retry


class HttpSession:
    def __init__(self):
        self._session = ClientSession()
        self._js_session = AsyncHTMLSession()
        self._default_headers = {}

    @property
    def default_headers(self):
        return self._default_headers

    @default_headers.setter
    def default_headers(self, headers: dict):
        if not isinstance(headers, dict):
            raise InvalidArgumentType(dict)
        self._default_headers = headers

    @staticmethod
    def validate_url(url: str):
        if not URL(url).is_absolute():
            raise InvalidUrlException(url)

    def _validate_request_args(self, **kwargs):
        method = kwargs.get("method")
        if not method:
            raise MissingMethodException
        if method not in REQUIRED_REQUEST_ARGS:
            raise UnsupportedMethodException
        for arg in REQUIRED_REQUEST_ARGS[method]:
            if arg not in kwargs:
                raise MissingArgumentException(arg)
        self.validate_url(kwargs["url"])

    def _make_headers(self, headers: dict) -> dict:
        return {**self._default_headers, **(headers or {})}
```

```
56
    @retry(times=3)
58  async def _http_request(self, **kwargs):
        response = await self._session.request(
60          max_redirects=MAX_REDIRECTS,
            **kwargs,
62          headers=self._make_headers(kwargs.get("headers")),
        )
64      await asyncio.sleep(2)
        return response
66
    async def request(self, **kwargs) -> Response:
68      self._validate_request_args(**kwargs)
        callbacks = kwargs.pop("callbacks", None)
70
        try:
72          aiohttp_response = await self._http_request(**kwargs)
        except TooManyRetrysException:
74          log.error(f'Request failed. URL: {kwargs["url"]}')
            return Response()
76
        response = await Response.create_response_object(aiohttp_response)
78
        if callbacks:
80          for callback in callbacks:
                if inspect.iscoroutinefunction(callback):
82                  response = await callback(response)
                else:
84                  response = callback(response)

86      return response

88  async def js_script_request(self, **kwargs):
        self._validate_request_args(**kwargs)
90      script = kwargs.get("script")

92      response = await self._js_session.request(kwargs)
        if script:
94          return await response.html.arender(script=script)
        else:
96          await response.html.arender()
            return response.html.full_text
```

- saci/session/response.py

```python
from aiohttp import ClientResponse
from bs4 import BeautifulSoup


class Response:
    def __init__(
        self,
        url: str,
        status: int,
        content: str,
        headers: dict,
        original_response: ClientResponse,
        soup: BeautifulSoup = None,
    ):
        self.url = url
        self.status = status
        self.content = content
        self.headers = headers
        self.original_response = original_response
        self.soup = soup or BeautifulSoup(content, "html.parser")

    @classmethod
    async def create_response_object(cls, aiohttp_response: ClientResponse)
        :
        content = await aiohttp_response.text()

        return cls(
            url=aiohttp_response.url,
            status=aiohttp_response.status,
            headers=aiohttp_response.headers,
            content=content,
            original_response=aiohttp_response,
        )
```

- saci/session/settings.py

```python
AVAILABLE_METHODS = ["get", "post"]

REGEX = {
    "url": r"(https?:\/\/(?:www\.|(?!www))[a-zA-Z0-9][a-zA-Z0-9-]+[a-zA-Z0
        -9]\.[^\s]{2,}|www\.[a-zA-Z0-9][a-zA-Z0-9-]+[a-zA-Z0-9]\.[^\s]{2,}|
        https?:\/\/(?:www\.|(?!www))[a-zA-Z0-9]+\.[^\s]{2,}|www\.[a-zA-Z0
        -9]+\.[^\s]{2,})"  # noqa
}
```

```
6
REQUIRED_REQUEST_ARGS = {"get": ["url"], "post": ["url", "data"]}
8
MAX_REDIRECTS = 5
```

- saci/session/utils.py

```
1 from enum import Enum


3
class Methods(Enum):
5     GET = "get"
      POST = "post"
```

- saci/websites/__init__.py

```
```

- saci/websites/geeks_for_geeks/__init__.py

```
```

- saci/websites/geeks_for_geeks/extract.py

```
  from re import IGNORECASE, compile, search
2 from typing import Dict, List, Optional, Tuple, Union

4 from bs4.element import Tag

6 from scraper.exceptions import (
      FailedComplexityExtraction,
8 )
  from scraper.observability.log import scraper_log as log
10 from scraper.session.response import Response
  from scraper.websites.geeks_for_geeks.settings import (
12     AUXILIARY_SPACE_REGEX,
       COMMENTS_STARTING_STRINGS,
14     HTML_ELEMENTS_NAMES,
```

```
    LANGUAGES,
    TIME_COMPLEXITY_REGEX,
)


def get_sourceline(element):
    return element.sourceline if isinstance(element, Tag) else element.
        parent.sourceline


def search_regex(pattern: str, text: str, group_num: int = 0) -> Optional[
    str]:
    match = search(pattern, text)
    return match.group(group_num) if match else None


def extract_code(code_table: Tag) -> str:
    code_text = ""
    code_lines = code_table.find_all("div", {"class": "line"})
    for line in code_lines:
        code_pieces = line.find_all("code")
        for code in code_pieces:
            code_text += code.text
        code_text += "\n"
    return code_text


def extract_code_comments(algorithm: str) -> str:
    algorithm_comments = ""
    for line in algorithm.splitlines():
        if line[:2] in COMMENTS_STARTING_STRINGS:
            algorithm_comments += line
            algorithm_comments += "\n"
        else:
            return algorithm_comments


def extract_complexity_from_reference(
    reference: Tag, regex_map: Dict[str, List[str]]
) -> str:
    matches = [
        reference.find_next(string=compile(regex)) for regex in regex_map["
            word"]
    ]

    matches = [match for match in matches if match]

    closest_match = min(
```

```
60          matches,
            key=lambda match: abs(get_sourceline(reference) - get_sourceline(
                match)),
62      )

64      for regex in regex_map["value"]:
            for element in HTML_ELEMENTS_NAMES:
66              complexity = search_regex(
                    regex, closest_match.find_previous(element).text, 1
68              )
                if complexity:
70                  return complexity

72      raise FailedComplexityExtraction


74
def extract_name(dom_reference: Tag) -> str:
76      name = dom_reference.find_previous("h2")
        if name and "tabtitle" not in name.get("class", ""):
78          return name.text


80
def fallback_search(reference_list: List[str], pattern: str) -> Optional[
    str]:
82      for result in reference_list:
            if search(pattern, result, flags=IGNORECASE):
84              return search_regex(TIME_COMPLEXITY_REGEX["fallback"], result)
        return None
86


88  def extract_complexity_with_fallback(
        dom_reference: Tag,
90  ) -> Tuple[Optional[str], Optional[str]]:
        search_func = lambda x: search_regex(TIME_COMPLEXITY_REGEX["fallback"],
            x.text)
92      next_results = [p.text for p in dom_reference.find_all_next("p") if
           search_func(p)]
        prev_results = [
94          p.text for p in dom_reference.find_all_previous("p") if search_func
                (p)
        ]
96
        time_complexity = fallback_search(next_results, r"time") or
           fallback_search(
98          prev_results, r"time"
        )
100     space_complexity = fallback_search(
            next_results, r"auxiliary|space"
```

```
    ) or fallback_search(prev_results, r"auxiliary|space")

    return time_complexity, space_complexity


def extract_codes_and_references(
    response: Response,
) -> Tuple[List[Dict[str, str]], List[Tag]]:
    codes, references = [], []
    code_tabs = response.soup.find_all("div", {"class": "responsive-tabs"})

    for tab in code_tabs:
        code_map = {}
        references.append(tab)

        for lang in LANGUAGES:
            algorithm = tab.find_next("h2", {"class": "tabtitle"}, string=
                lang)
            if algorithm:
                code = extract_code(algorithm.find_next("td", {"class": "
                    code"}))
                comments = extract_code_comments(code)

                code_map[lang] = {
                    "code": code,
                    "comments": comments,
                }

        codes.append(code_map)

    return codes, references


def extract_data(
    response: Response,
) -> List[Dict[str, Union[str, bool, Dict[str, str]]]]:
    codes, dom_references = extract_codes_and_references(response)
    if not codes:
        log.warning(f"Failed to find codes. URL: {response.url}")
        return []

    complexities = []
    for ref in dom_references:
        try:
            time_complexity = extract_complexity_from_reference(
                ref, TIME_COMPLEXITY_REGEX
            )
        except FailedComplexityExtraction:
```

```
            time_complexity, _ = extract_complexity_with_fallback(ref)

        try:
            space_complexity = extract_complexity_from_reference(
                ref, AUXILIARY_SPACE_REGEX
            )
        except FailedComplexityExtraction:
            _, space_complexity = extract_complexity_with_fallback(ref)

        complexities.append(
            {
                "time_complexity": time_complexity,
                "trustable_time_complexity": bool(time_complexity),
                "space_complexity": space_complexity,
                "trustable_space_complexity": bool(space_complexity),
            }
        )

    main_name = response.soup.find("div", {"class": "article-title"}).text
    names = [main_name] + [
        extract_name(ref) if extract_name(ref) else main_name
        for ref in dom_references[1:]
    ]

    return [
        {
            "name": name,
            "time_complexity": complexity["time_complexity"],
            "trustable_time_complexity": complexity["
                trustable_time_complexity"],
            "space_complexity": complexity["space_complexity"],
            "trustable_space_complexity": complexity["
                trustable_space_complexity"],
            "url": str(response.url),
            "codes": code,
        }
        for name, complexity, code in zip(names, complexities, codes)
    ]
```

- saci/websites/geeks_for_geeks/login.py

```
import os

from scraper.session.http_session import HttpSession
from scraper.session.response import Response
```

```
from scraper.session.utils import Methods
from scraper.websites.geeks_for_geeks.settings import LOGIN_URL


def get_login_payload():
    return {
        "reqType": "Login",
        "user": os.getenv("USERNAME"),
        "pass": os.getenv("PASSWORD"),
        "rem": False,
        "to": "https://auth.geeksforgeeks.org/?to=https://www.geeksforgeeks
            .org/",
        "rem": "on",
        "g-recaptcha-response": None,
        "browserInfo": {},
    }


async def follow_login(session: HttpSession):
    await session.request(
        method=Methods.GET.value,
        data=get_login_payload(),
        url=LOGIN_URL,
    )
```

- saci/websites/geeks_for_geeks/settings.py

```
BASE_URL = "https://www.geeksforgeeks.org"

LOGIN_URL = "https://auth.geeksforgeeks.org/auth.php"

ALGORITHMS_LOCATION_URLS = [
    f"{BASE_URL}/fundamentals-of-algorithms",
    f"{BASE_URL}/data-structures",
]

HEADERS = {
    "Host": "www.geeksforgeeks.org",
    "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/
        avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=
        b3;q=0.9",
    "Accept-Encoding": "gzip, deflate",
    "Upgrade-Insecure-Requests": "1",
    "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit
        /537.36 (KHTML, like Gecko) Chrome/104.0.0.0 Safari/537.36",
```

```
    "Sec-Fetch-Site": "none",
    "Sec-Fetch-Mode": "navigate",
    "Sec-Fetch-User": "?1",
    "Sec-Fetch-Dest": "document",
    "Sec-Ch-Ua": '"Chromium";v="105", "Not)A;Brand";v="8"',
    "Sec-Ch-Ua-Mobile": "?0",
    "Sec-Ch-Ua-Platform": '"Windows"',
}

ALGORITHM_LINKS_JS_SCRIPT = """
    () => {
        const algorithm_links = []

        const ordered_lists = document.getElementsByTagName('ol');
        for (ol of ordered_lists) {
            const hrefs = []
            const a_tags = ol.getElementsByTagName('a')
            for (a of a_tags) {
                hrefs.push(a.getAttribute('href'))
            }
            algorithm_links.push(hrefs)
        }

        return algorithm_links
    }
"""

TIME_COMPLEXITY_REGEX = {
    "word": [
        r"[Tt]ime [Cc]omplexit\w*[\s]?:|[Tt]ime [Cc]omplexit\w*[\s]?",
    ],
    "value": [
        r"[Tt]ime [Cc]omplexit\w*\s?:.*?(O\s*\(.*?\))",
        r"[Tt]ime [Cc]omplexit\w*\s?.*?(O\s*\(.*?\))",
        r"[Tt]ime [Cc]omplexit\w*\s.*?\sis\s.*?(O\s*\(.*?\))",
        r"[Tt]ime [Cc]omplexit\w*\s?:?.*?\sis\s(\w*)",
    ],
    "fallback": r"O\s*\(.*?\)",
}

AUXILIARY_SPACE_REGEX = {
    "word": [
        r"[Aa]uxiliary [Ss]pace\s?:|[Aa]uxiliary [Ss]pace\s?",
        r"[Ss]pace [Cc]omplexit\w*\s?:|[Ss]pace [Cc]omplexit\w*\s?",
    ],
    "value": [
        r"[Aa]uxiliary [Ss]pace\s?[\[Cc\]omplexit\w*]?:.*?(O\s*\(.*?\))",
        r"[Aa]uxiliary [Ss]pace\s?[\[Cc\]omplexit\w*]?.*?(O\s*\(.*?\))",
```

```
        r"[Ss]pace [Cc]omplexit\w*\s?:.*?(O\s*\(.*?\))",
        r"[Ss]pace [Cc]omplexit\w*\s?.*?(O\s*\(.*?\))",
        r"[Aa]uxiliary [Ss]pace\s?[\[Cc\]omplexit\w*]?.*?\sis\s.*?(O\s
            *\(.*?\))",
        r"[Ss]pace [Cc]omplexit\w*\s.*?\sis\s.*?(O\s*\(.*?\))",
        r"[Aa]uxiliary [Ss]pace\w*\s?:?.*?\sis\s(\w*)",
        r"[Ss]pace [Cc]omplexit\w*\s?:?.*?\sis\s(\w*)",
    ],
    "fallback": r"O\s*\(.*?\)",
}

HTML_ELEMENTS_NAMES = ["p", "li", "ul", "i"]

COMMENTS_STARTING_STRINGS = ["//", "# ", "/*"]

LANGUAGES = ["C", "C#", "C++", "Python3", "Python", "PHP", "Java", "
    Javascript"]
```

- saci/websites/geeks_for_geeks/spider.py

```
from asyncio import gather

from scraper.exceptions import FailedExtraction
from scraper.observability.log import scraper_log as log
from scraper.schema.data_schemas import ScrapedAlgorithm
from scraper.session.http_session import HttpSession
from scraper.session.response import Response
from scraper.session.utils import Methods
from scraper.utils import remove_duplicates, retry
from scraper.websites.geeks_for_geeks.extract import extract_data
from scraper.websites.geeks_for_geeks.login import follow_login
from scraper.websites.geeks_for_geeks.settings import
    ALGORITHMS_LOCATION_URLS, HEADERS


def parse_algorithm_schema(algorithms_data: list) -> list:
    return [ScrapedAlgorithm(**algorithm) for algorithm in algorithms_data]


def filter_algorithms_urls(algorithms_urls: list) -> list:
    return [href for href in algorithms_urls if "geeksquiz" not in href]


async def extract_algorithm_data(response: Response) -> list:
    if not response:
```

```
        return []

    data = await extract_data(response)
    if not data:
        raise FailedExtraction

    return data


def extract_algorithms_urls(response: Response) -> list:
    page_content = response.soup.find("div", {"class": "page_content"})
    return [
        a.attrs.get("href")
        for ol in page_content.find_all("ol")[1:]
        for a in ol.find_all("a")
        if a.attrs.get("href")
    ]


@retry(times=3, raise_exception=False, return_value=[])
async def follow_algorithm(session: HttpSession, url: str):
    return await session.request(
        method=Methods.GET.value,
        url=url,
        callbacks=[extract_algorithm_data, parse_algorithm_schema],
    )


async def follow_algorithms(session: HttpSession, algorithms_urls: list) ->
    list:
    algorithms = await gather(
        *[follow_algorithm(session, url) for url in algorithms_urls]
    )
    return sum(algorithms, [])


async def follow_algorithms_urls(session: HttpSession) -> list:
    algorithm_location_urls = await gather(
        *[
            session.request(
                method=Methods.GET.value,
                url=url,
                callbacks=[
                    extract_algorithms_urls,
                    filter_algorithms_urls,
                    remove_duplicates,
                ],
            )
```

```
72              for url in ALGORITHMS_LOCATION_URLS
            ]
74      )
        return sum(algorithm_location_urls, [])
76
78  async def run(url: str = None) -> list:
        session = HttpSession()
80      session.default_headers = HEADERS

82      log.info('Starting "Geeks for Geeks" algorithms extraction')
        await follow_login(session)
84
        algorithms_urls = [url] if url else await follow_algorithms_urls(
            session)
86      return await follow_algorithms(session, algorithms_urls)
```

Apêndice B - Código Fonte Versão OpenAI

- saci/requirements.txt

```
aiohttp
beautifulsoup4
black
pandas
pydantic
pymongo
pyppeteer
requests
scikit-learn
scipy
seaborn
mlxtend
requests-html
pip-tools
openai
langchain
asyncio-throttle
```

- saci/makefile

```
PYTHON :=
ifeq ($(OS),Windows_NT)
  PYTHON = venv/Scripts/python
  PIP = venv/Scripts/pip
else
  PYTHON = venv/bin/python3
  PIP = venv/bin/pip
endif

requirements: requirements.txt
  $(PIP) install -r requirements.txt

run_scraper:
```

```
  $(PYTHON) -m main

scrape_url:
  $(PYTHON) -m scraper.scripts --script=scrape_url --url=$(url) --scraper=$
      (scraper)

results_analysis:
  $(PYTHON) -m scraper.scripts --script=results_analysis --scraper=$(
      scraper)

manual_results_boilerplate:
  $(PYTHON) -m scraper.scripts --script=manual_results_boilerplate --
      scraper=$(scraper)
```

- saci/main.py

```
from asyncio import get_event_loop

from scraper.runner import run_scrapers

if __name__ == "__main__":
    loop = get_event_loop()
    loop.run_until_complete(run_scrapers())
```

- saci/scraper/__init__.py
- saci/scraper/database.py

```
import os
from typing import Dict, List

from pymongo import MongoClient
from pymongo.collection import Collection
from pymongo.operations import UpdateOne

from scraper.settings import DATABASE_NAME

FIELDS_TO_UPDATE = [
    "name",
    "time_complexity",
    "trustable_time_complexity",
    "space_complexity",
```

```
    "trustable_space_complexity",
]


class ScraperDatabase:
    def __init__(self):
        self.client: MongoClient = MongoClient(
            os.environ.get("MONGO_CONNECTION_STRING"),
                serverSelectionTimeoutMS=1
        )
        self.database = self.client[DATABASE_NAME]

    def _get_collection(self, collection_name: str) -> Collection:
        return self.database[collection_name]

    def update_database(self, collection_name: str, data: List[Dict[str,
        any]]):
        collection = self._get_collection(collection_name)
        operations = [
            UpdateOne(
                filter={"url": item["url"], "codes": item["codes"]},
                update={"$set": {field: item[field] for field in
                    FIELDS_TO_UPDATE}},
                upsert=True,
            )
            for item in data
        ]
        collection.bulk_write(operations)
```

- saci/scraper/exceptions.py

```
class InvalidUrlException(Exception):
    def __init__(self, url):
        super().__init__(f"The provided URL '{url}' is not valid.")


class TooManyRetrysException(Exception):
    pass


class FailedExtraction(Exception):
    """Failed the extraction proccess"""

    pass
```

```
15
class FailedComplexityExtraction(Exception):
17    """Failed to extract the complexitys"""

19    pass
```

- saci/scraper/runner.py

```
1  from asyncio import gather

3  from scraper.database import ScraperDatabase
   from scraper.settings import USE_MONGO_DATABASE
5  from scraper.utils import write_results_to_json
   from scraper.websites import websites
7

9  async def run_scrapers():
       scrapers_results = await gather(*[scraper() for scraper in websites.
           values()])
11     for name, data in zip(websites.keys(), scrapers_results):
           if USE_MONGO_DATABASE:
13             database = ScraperDatabase()
               database.update_database(name, [result.dict() for result in
                   data])
15         else:
               write_results_to_json(
17                 name,
                   sorted([result.dict() for result in data], key=lambda alg:
                       alg["url"]),
19             )
```

- saci/scraper/scripts.py

```
1  import argparse
   import pprint
3  from asyncio import get_event_loop
   from importlib import import_module
5  from scraper.observability.metrics import (
       make_results_analysis,
7      make_manual_results_boilerplate,
   )
9
```

```
def extract_single_url(url: str, scraper: str):
    module = import_module(f"scraper.websites.{scraper}")
    spider = getattr(module, "spider")
    loop = get_event_loop()
    result = loop.run_until_complete(spider.run(url))
    pprint.PrettyPrinter(indent=4).pprint(result)


def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("--script")
    parser.add_argument("--url", help="URL to be extracted.")
    parser.add_argument("--scraper", help="Scraper to be used.")
    args = parser.parse_args()

    actions = {
        "scrape_url": lambda: extract_single_url(args.url, args.scraper),
        "results_analysis": lambda: make_results_analysis(args.scraper),
        "manual_results_boilerplate": lambda:
            make_manual_results_boilerplate(
             args.scraper
        ),
    }

    action = actions.get(args.script)
    if action:
        action()
    else:
        print(f"Unknown script: {args.script}")


if __name__ == "__main__":
    main()
```

- saci/scraper/settings.py

```
DATABASE_NAME = "COMPLEXITY_SCRAPER"

USE_MONGO_DATABASE = False
```

- saci/scraper/utils.py

```python
import json
import os

from asyncio_throttle import Throttler
from langchain.chains import LLMChain
from langchain.llms import OpenAI
from langchain.prompts import PromptTemplate

from scraper.exceptions import TooManyRetrysException
from scraper.observability.log import scraper_log as log


def retry(
    times: int,
    raise_exception=True,
    return_value=None,
):
    def func_wrapper(f):
        async def wrapper(*args, **kwargs):
            for _ in range(times):
                try:
                    return await f(*args, **kwargs)
                except Exception:
                    pass

            if not raise_exception:
                return return_value
            else:
                raise TooManyRetrysException

        return wrapper

    return func_wrapper


def remove_duplicates(data: list) -> list:
    return list(set(data))


def open_results_from_json(file_path: str) -> dict:
    with open(file_path, "r") as file:
        file_contents = file.read()
        results = json.loads(file_contents)
    return results


def write_results_to_json(file_path: str, data: list):
```

```
    with open(f"{file_path}.json", "w") as file:
49      json.dump(data, file)


51
class LlmComplexitySearcher:
53      _instance = None
        throttler = Throttler(rate_limit=100, period=60)
55
        def __init__(self):
57          self.LLM = OpenAI(openai_api_key=os.environ.get("OPENAI_KEY", ""))
            self.PROMPT = PromptTemplate.from_template(
59              """I am using you during a scraping operation in which I need
                    to extract from a piece of text the values of complexity/
                    code complexity.
                Consider that sometimes the space complexity is also called as
                    auxiliary space.
61              Consider that both complexitys often come in this format: 'Some
                     type of complexity: value of complexity'.
                But they can also come inside the text like: 'The type of
                    complexity of the given ... is ...'.
63              Give the answer in JSON format with no line breaks, with a key
                    called "complexity" and the value for the key is your answer
                    .
                If you cant determine the answer give the json with a null in
                    the value.
65              What is the {complexity} that is written in the following text:
                     "{text}"?"""
            )
67          self.LLM_CHAIN = LLMChain(prompt=self.PROMPT, llm=self.LLM)

69      def __new__(cls, *args, **kwargs):
            if not cls._instance:
71              cls._instance = super().__new__(cls)
            return cls._instance
73
        @retry(times=3, raise_exception=False, return_value={"complexity": None
            })
75      async def search(self, complexity: str, text: str) -> dict:
            async with self.throttler:
77              answer = self.LLM_CHAIN.run(complexity=complexity, text=text)
                return json.loads(answer)
```

- saci/observability/__init__.py

- saci/observability/log.py

```
import platform

from logging import DEBUG, Filter, Formatter, StreamHandler, getLogger

from scraper.observability.settings import SCRAPER_LOG_FORMAT,
    SESSION_LOG_FORMAT


class AddScraperName(Filter):
    def filter(self, record):
        if not hasattr(record, "scraper"):
            split_char = "\\" if "Windows" in platform.platform() else "/"
            record.scraper = record.pathname.split(split_char)[-2]
        return True


def build_scraper_log(logger_name):
    log_handler = StreamHandler()
    log_handler.setLevel(DEBUG)
    log_handler.setFormatter(Formatter(SCRAPER_LOG_FORMAT))

    log = getLogger(logger_name)
    log.addFilter(AddScraperName())
    log.addHandler(log_handler)
    return log


def build_session_log(logger_name):
    log_handler = StreamHandler()
    log_handler.setLevel(DEBUG)
    log_handler.setFormatter(Formatter(SESSION_LOG_FORMAT))

    log = getLogger(logger_name)
    log.addHandler(log_handler)
    return log


scraper_log = build_scraper_log("scraper_log")
session_log = build_session_log("session_log")
```

- saci/observability/metrics.py

```
import math
```

```python
import re
from collections import Counter
from typing import Any, Dict, List, Tuple, Union

import matplotlib.pyplot as plt
import numpy
import requests
from bs4 import BeautifulSoup, NavigableString, Tag
from mlxtend.plotting import plot_confusion_matrix

from scraper.utils import (
    open_results_from_json,
    remove_duplicates,
    write_results_to_json,
)


def make_results_caracterization(scraper: str) -> Dict[str, int]:
    results = open_results_from_json(f"./results/{scraper}.json")

    return {
        "Distinct Time Complexities": len(
            {algorithm["time_complexity"] for algorithm in results}
        ),
        "Distinct Space Complexities": len(
            {algorithm["space_complexity"] for algorithm in results}
        ),
        "Non Trusted Time Complexities": sum(
            [1 for algorithm in results if not algorithm["
                trustable_time_complexity"]]
        ),
        "Non Trusted Space Complexities": sum(
            [1 for algorithm in results if not algorithm["
                trustable_space_complexity"]]
        ),
    }


def make_completition_rate(scraper: str) -> Dict[str, int]:
    results = open_results_from_json(f"./results/{scraper}.json")

    total = len(results)
    time_complexity, space_complexity = sum(
        1 for algorithm in results if not algorithm["time_complexity"]
    ), sum(1 for algorithm in results if not algorithm["space_complexity"])
    urls_with_problem = remove_duplicates(
        [
            algorithm["url"]
```

```
            for algorithm in results
            if not algorithm["time_complexity"] or not algorithm["
                space_complexity"]
        ]
    )

    return {
        "Total Algorithms": total,
        "Time Complexity Extracted": f"{total - time_complexity} extracted
            - {100 - (time_complexity / total) * 100}%",
        "Space Complexity Extracted": f"{total - space_complexity}
            extracted - {100 - (space_complexity / total) * 100}%",
        "URLs with problem": urls_with_problem,
    }


def calculate_metrics(
    true_positive: int, false_positive: int, true_negative: int,
        false_negative: int
) -> Dict[str, float]:
    accuracy = (true_positive + true_negative) / (
        true_positive + true_negative + false_positive + false_negative
    )
    precision = true_positive / (true_positive + false_positive)
    recall = true_positive / (true_positive + false_negative)
    f1_score = (2 * precision * recall) / (precision + recall)

    return {
        "True Positive": true_positive,
        "False Positive": false_positive,
        "True Negative": true_negative,
        "False negative": false_negative,
        "Accuracy": accuracy,
        "Precision": precision,
        "Recall": recall,
        "F1 Score": f1_score,
    }


def plot_confusion(
    title: str,
    true_positive: int,
    false_negative: int,
    false_positive: int,
    true_negative: int,
):
    array = numpy.array(
        [[true_positive, false_negative], [false_positive, true_negative]]
```

```
92      )
94      fig, ax = plot_confusion_matrix(conf_mat=array)
        plt.title("Space Complexity Confusion Matrix")
96      plt.show()
98
    def make_confusion_matrix(scraper: str) -> Dict[str, Dict[str, Any]]:
100     manual_results = open_results_from_json(f"./results/manual_{scraper}.
            json")
        results = open_results_from_json(f"./results/{scraper}.json")
102
        time_metrics = {
104         "true_positive": 0,
            "false_positive": 0,
106         "true_negative": 0,
            "false_negative": 0,
108     }
110     space_metrics = {
            "true_positive": 0,
112         "false_positive": 0,
            "true_negative": 0,
114         "false_negative": 0,
        }
116
        for result, manual_result in zip(results, manual_results):
118         # Time complexity
            if time_complexity := result["time_complexity"]:
120             if time_complexity == manual_result["time_complexity"]:
                    time_metrics["true_positive"] += 1
122             else:
                    time_metrics["false_positive"] += 1
124         else:
                if not manual_result["time_complexity"]:
126                 time_metrics["true_negative"] += 1
                else:
128                 time_metrics["false_negative"] += 1

130         # Space complexity
            if space_complexity := result["space_complexity"]:
132             if space_complexity == manual_result["space_complexity"]:
                    space_metrics["true_positive"] += 1
134             else:
                    space_metrics["false_positive"] += 1
136         else:
                if not manual_result["space_complexity"]:
138                 space_metrics["true_negative"] += 1
```

```
            else:
                space_metrics["false_negative"] += 1

    # Calculating metrics
    time_complexity_metrics = calculate_metrics(**time_metrics)
    space_complexity_metrics = calculate_metrics(**space_metrics)

    # Plotting the confusion matrices
    plot_confusion("Time Complexity Confusion Matrix", **time_metrics)
    plot_confusion("Space Complexity Confusion Matrix", **space_metrics)

    # Return result
    return {
        "Time Complexity": time_complexity_metrics,
        "Space Complexity": space_complexity_metrics,
    }


def make_manual_results_boilerplate(scraper: str) -> None:
    results = open_results_from_json(f"./results/{scraper}.json")

    manual_results = [
        {"url": alg["url"], "time_complexity": " ", "space_complexity": " "
            }
        for alg in results
    ]

    write_results_to_json(f"./results/manual_{scraper}", manual_results)


def make_complexitys_classification(scraper: str) -> Dict[str, Dict[str,
    int]]:
    results = open_results_from_json(f"./results/{scraper}.json")

    time_classification = classify_complexity(
        [result["time_complexity"] for result in results]
    )
    space_classification = classify_complexity(
        [result["space_complexity"] for result in results]
    )

    return {
        "Time Complexity Classification": time_classification,
        "Space Complexity Classification": space_classification,
    }


def classify_complexity(complexities: List[str]) -> Dict[str, int]:
```

```
    constant, linear, exponential, polynomial, factorial, logarithmic = 0,
        0, 0, 0, 0, 0
    for complexity in complexities:
        if complexity:
            if re.match(r"O\(1\)", complexity):
                constant += 1
            elif re.match(r"O\(\w\)", complexity):
                linear += 1
            elif re.match(r"O\(.*!.*\)", complexity):
                factorial += 1
            elif re.match(r"O\(.*log.*\)", complexity):
                logarithmic += 1
            elif re.match(r"O\(\d\^?\w\)", complexity):
                exponential += 1
            elif re.match(r"O\(\w\s*[\^\*\+]?\s*[\d\w\s]+.*\)", complexity)
                or re.match(
                r"O\(\d+\(\w\*\w\)", complexity
            ):
                polynomial += 1

    return {
        "Constant complexity": constant,
        "Linear complexity": linear,
        "Exponential complexity": exponential,
        "Polynomial complexity": polynomial,
        "Factorial complexity": factorial,
        "Log complexity": logarithmic,
    }


def make_algorithms_histogram(scraper: str):
    results = open_results_from_json(f"./results/{scraper}.json")

    count = Counter([result["url"] for result in results]).values()
    plt.hist(count)
    plt.xlabel("Número de Algoritmos")
    plt.ylabel("Frequência")
    plt.title("Histograma da Distribuição de Algoritmos")
    plt.show()


def traverse_tree(
    node: Union[Tag, NavigableString], depth: int, depths: Dict[str, List[
        int]]
) -> None:
    """
    Traverse the BeautifulSoup tree and populate the depths dictionary with
        tag names and their depths.
```

```
230     Args:
            node: The current BeautifulSoup node.
232         depth: Current depth of traversal.
            depths: Dictionary to populate with depths.
234     """
        if isinstance(node, Tag):  # Using isinstance to check if node is a Tag
            type
236         if node.name not in depths:
                depths[node.name] = []
238         depths[node.name].append(depth)

240         for child in node.children:
                traverse_tree(child, depth + 1, depths)
242

244 def calculate_html_nodes_depth(scraper: str) -> None:
        results = open_results_from_json(f"./results/{scraper}.json")
246     urls = list(set(result["url"] for result in results))

248     all_max_depths: List[int] = []
        all_num_nodes: List[int] = []
250

        for url in urls:
252         response = requests.get(url)
            soup = BeautifulSoup(response.content, "html.parser")
254

            depths: Dict[str, List[int]] = {}
256         traverse_tree(soup, 0, depths)

258         max_depth = max(max(depths[tag]) for tag in depths)
            num_nodes = sum(len(depths[tag]) for tag in depths)
260

            all_max_depths.append(max_depth)
262         all_num_nodes.append(num_nodes)

264     _, axs = plt.subplots(1, 2, figsize=(10, 5))
        axs[0].hist(all_max_depths, bins=[23, 24, 25, 26])
266     axs[0].set_xlabel("Profundidade Máxima")
        axs[0].set_ylabel("Frequência")
268     axs[0].set_title("Histograma da Profundidade Máxima")

270     axs[1].hist(all_num_nodes, bins=[2500, 7500, 12500, 17500, 22500,
            30000])
        axs[1].set_xlabel("Número de Nodos")
272     axs[1].set_ylabel("Frequência")
        axs[1].set_title("Histograma da Contagem de Nodos")
274
```

```
    plt.suptitle("Combined Histogram for All URLs")
    plt.show()


def make_results_analysis(scraper: str):
    write_results_to_json(
        f"./results/{scraper}_analysis",
        [
            {
                **make_completition_rate(scraper),
                **make_results_caracterization(scraper),
                **make_confusion_matrix(scraper),
                **make_complexitys_classification(scraper),
            }
        ],
    )
    # calculate_html_nodes_depth(scraper)
    # make_algorithms_histogram(scraper)
```

- saci/observability/settings.py

```
SCRAPER_LOG_FORMAT = (
    "%(asctime)s - %(levelname)s - Scraper: %(scraper)s - Message: %(
        message)s"  # noqa
)
SESSION_LOG_FORMAT = (
    "%(asctime)s - %(levelname)s - Session - Message: %(message)s"  # noqa
)
```

- saci/schema/__init__.py

```
```

- saci/schema/data_schemas.py

```
from typing import Optional

from pydantic import Field
from pydantic import BaseModel
```

```
6
class ScrapedCode(BaseModel):
8     """The scraped codes of a given algorithm"""

10    code: str = Field(
          default="", description="The code of the algorithm in a certain
              language"
12    )
      comments: str = Field(default="", description="The first few comments
          in the code")
14


16 class ScrapedAlgorithm(BaseModel):
      """The final result of the scraping operations, an extracted algorithm
          with its attributes"""
18
      name: str = Field(default="", description="The name of the scraped
          algorithm")
20    time_complexity: Optional[str] = Field(
          default="", description="The time complexity of the scraped
              algorithm"
22    )
      trustable_time_complexity: bool = Field(
24        default=True,
          description="If the given time complexity used a trustable
              extraction method",
26    )
      space_complexity: Optional[str] = Field(
28        default="", description="The space complexity of the algorithm"
      )
30    trustable_space_complexity: bool = Field(
          default=True,
32        description="If the given space complexity used a trustable
              extraction method",
      )
34    url: str = Field(default="", description="The space complexity of the
          algorithm")
      codes: dict[str, ScrapedCode] = Field(
36        default=dict(), description="The algorithm implementations"
      )
```

- saci/session/__init__.py

- saci/session/exceptions.py

```
class InvalidUrlException(Exception):
    def __init__(self):
        self.message = "Please provide a valid URL string!"
        super().__init__(self.message)


class MissingMethodException(Exception):
    def __init__(self):
        self.message = "Please provide a request method!"
        super().__init__(self.message)


class UnsupportedMethodException(Exception):
    def __init__(self):
        self.message = "Please provide a supported method!"
        super().__init__(self.message)


class MissingArgumentException(Exception):
    def __init__(self, argument):
        self.message = f"Please provide the missing argument: {argument}!"
        super().__init__(self.message)


class InvalidArgumentType(Exception):
    def __init__(self, type):
        self.message = f"Please provide the correct argument type: {type}!"
        super().__init__(self.message)
```

- saci/session/http_session.py

```
import asyncio
import inspect

from aiohttp import ClientSession
from requests_html import AsyncHTMLSession
from yarl import URL

from scraper.exceptions import TooManyRetrysException
from scraper.observability.log import session_log as log
from scraper.session.exceptions import (
    InvalidArgumentType,
    InvalidUrlException,
```

```
    MissingArgumentException,
    MissingMethodException,
    UnsupportedMethodException,
)
from scraper.session.response import Response
from scraper.session.settings import MAX_REDIRECTS, REQUIRED_REQUEST_ARGS
from scraper.utils import retry


class HttpSession:
    def __init__(self):
        self._session = ClientSession()
        self._js_session = AsyncHTMLSession()
        self._default_headers = {}

    @property
    def default_headers(self):
        return self._default_headers

    @default_headers.setter
    def default_headers(self, headers: dict):
        if not isinstance(headers, dict):
            raise InvalidArgumentType(dict)
        self._default_headers = headers

    @staticmethod
    def validate_url(url: str):
        if not URL(url).is_absolute():
            raise InvalidUrlException(url)

    def _validate_request_args(self, **kwargs):
        method = kwargs.get("method")
        if not method:
            raise MissingMethodException
        if method not in REQUIRED_REQUEST_ARGS:
            raise UnsupportedMethodException
        for arg in REQUIRED_REQUEST_ARGS[method]:
            if arg not in kwargs:
                raise MissingArgumentException(arg)
        self.validate_url(kwargs["url"])

    def _make_headers(self, headers: dict) -> dict:
        return {**self._default_headers, **(headers or {})}

    @retry(times=3)
    async def _http_request(self, **kwargs):
        response = await self._session.request(
            max_redirects=MAX_REDIRECTS,
```

```
            **kwargs,
            headers=self._make_headers(kwargs.get("headers")),
        )
        await asyncio.sleep(2)
        return response

    async def request(self, **kwargs) -> Response:
        self._validate_request_args(**kwargs)
        callbacks = kwargs.pop("callbacks", None)

        try:
            aiohttp_response = await self._http_request(**kwargs)
        except TooManyRetrysException:
            log.error(f'Request failed. URL: {kwargs["url"]}')
            return Response()

        response = await Response.create_response_object(aiohttp_response)

        if callbacks:
            for callback in callbacks:
                if inspect.iscoroutinefunction(callback):
                    response = await callback(response)
                else:
                    response = callback(response)

        return response

    async def js_script_request(self, **kwargs):
        self._validate_request_args(**kwargs)
        script = kwargs.get("script")

        response = await self._js_session.request(kwargs)
        if script:
            return await response.html.arender(script=script)
        else:
            await response.html.arender()
            return response.html.full_text
```

- saci/session/response.py

```
from aiohttp import ClientResponse
from bs4 import BeautifulSoup


class Response:
```

```
    def __init__(
        self,
        url: str,
        status: int,
        content: str,
        headers: dict,
        original_response: ClientResponse,
        soup: BeautifulSoup = None,
    ):
        self.url = url
        self.status = status
        self.content = content
        self.headers = headers
        self.original_response = original_response
        self.soup = soup or BeautifulSoup(content, "html.parser")

    @classmethod
    async def create_response_object(cls, aiohttp_response: ClientResponse)
        :
        content = await aiohttp_response.text()

        return cls(
            url=aiohttp_response.url,
            status=aiohttp_response.status,
            headers=aiohttp_response.headers,
            content=content,
            original_response=aiohttp_response,
        )
```

- saci/session/settings.py

```
AVAILABLE_METHODS = ["get", "post"]

REGEX = {
    "url": r"(https?:\/\/(?:www\.|(?!www))[a-zA-Z0-9][a-zA-Z0-9-]+[a-zA-Z0
        -9]\.[^\s]{2,}|www\.[a-zA-Z0-9][a-zA-Z0-9-]+[a-zA-Z0-9]\.[^\s]{2,}|
        https?:\/\/(?:www\.|(?!www))[a-zA-Z0-9]+\.[^\s]{2,}|www\.[a-zA-Z0
        -9]+\.[^\s]{2,})"  # noqa
}

REQUIRED_REQUEST_ARGS = {"get": ["url"], "post": ["url", "data"]}

MAX_REDIRECTS = 5
```

- saci/session/utils.py

```
from enum import Enum


class Methods(Enum):
    GET = "get"
    POST = "post"
```

- saci/websites/__init__.py

```
```

- saci/websites/geeks_for_geeks/__init__.py

```
```

- saci/websites/geeks_for_geeks/extract.py

```
from re import IGNORECASE, compile, search
from typing import Dict, List, Optional, Tuple, Union

from bs4.element import Tag

from scraper.exceptions import FailedComplexityExtraction
from scraper.observability.log import scraper_log as log
from scraper.session.response import Response
from scraper.utils import LlmComplexitySearcher
from scraper.websites.geeks_for_geeks.settings import (
    AUXILIARY_SPACE_REGEX,
    COMMENTS_STARTING_STRINGS,
    HTML_ELEMENTS_NAMES,
    LANGUAGES,
    TIME_COMPLEXITY_REGEX,
)

llm_searcher = LlmComplexitySearcher()


def get_sourceline(element):
```

```
22      return element.sourceline if isinstance(element, Tag) else element.
            parent.sourceline


24
def search_regex(pattern: str, text: str, group_num: int = 0) -> Optional[
    str]:
26      match = search(pattern, text)
        return match.group(group_num) if match else None
28


30  def extract_code(code_table: Tag) -> str:
        code_text = ""
32      code_lines = code_table.find_all("div", {"class": "line"})
        for line in code_lines:
34          code_pieces = line.find_all("code")
            for code in code_pieces:
36              code_text += code.text
            code_text += "\n"
38      return code_text


40
def extract_code_comments(algorithm: str) -> str:
42      algorithm_comments = ""
        for line in algorithm.splitlines():
44          if line[:2] in COMMENTS_STARTING_STRINGS:
                algorithm_comments += line
46              algorithm_comments += "\n"
            else:
48              return algorithm_comments


50
async def extract_complexity_from_reference(
52      reference: Tag, regex_map: Dict[str, List[str]]
) -> str:
54      matches = [
            reference.find_next(string=compile(regex)) for regex in regex_map["
                word"]
56      ]

58      matches = [match for match in matches if match]

60      closest_match = min(
            matches,
62          key=lambda match: abs(get_sourceline(reference) - get_sourceline(
                match)),
        )
64

        for element in HTML_ELEMENTS_NAMES:
```

```
66          search_result = await llm_searcher.search(
                regex_map["value"], closest_match.find_previous(element).text
68          )

70          complexity = search_result.get("complexity")
            if complexity:
72              return complexity

74      raise FailedComplexityExtraction


76
def extract_name(dom_reference: Tag) -> str:
78      name = dom_reference.find_previous("h2")
        if name and "tabtitle" not in name.get("class", ""):
80          return name.text


82
def fallback_search(reference_list: List[str], pattern: str) -> Optional[
    str]:
84      for result in reference_list:
            if search(pattern, result, flags=IGNORECASE):
86              return search_regex(TIME_COMPLEXITY_REGEX["fallback"], result)
        return None
88


90  def extract_complexity_with_fallback(
        dom_reference: Tag,
92  ) -> Tuple[Optional[str], Optional[str]]:
        search_func = lambda x: search_regex(TIME_COMPLEXITY_REGEX["fallback"],
            x.text)
94      next_results = [p.text for p in dom_reference.find_all_next("p") if
           search_func(p)]
        prev_results = [
96          p.text for p in dom_reference.find_all_previous("p") if search_func
               (p)
        ]
98
        time_complexity = fallback_search(next_results, r"time") or
           fallback_search(
            prev_results, r"time"
100     )
        space_complexity = fallback_search(
            next_results, r"auxiliary|space"
104     ) or fallback_search(prev_results, r"auxiliary|space")

106     return time_complexity, space_complexity


108
```

```
def extract_codes_and_references(
    response: Response,
) -> Tuple[List[Dict[str, str]], List[Tag]]:
    codes, references = [], []
    code_tabs = response.soup.find_all("div", {"class": "responsive-tabs"})

    for tab in code_tabs:
        code_map = {}
        references.append(tab)

        for lang in LANGUAGES:
            algorithm = tab.find_next("h2", {"class": "tabtitle"}, string=
                lang)
            if algorithm:
                code = extract_code(algorithm.find_next("td", {"class": "
                    code"}))
                comments = extract_code_comments(code)

                code_map[lang] = {
                    "code": code,
                    "comments": comments,
                }

        codes.append(code_map)

    return codes, references


async def extract_data(
    response: Response,
) -> List[Dict[str, Union[str, bool, Dict[str, str]]]]:
    codes, dom_references = extract_codes_and_references(response)
    if not codes:
        log.warning(f"Failed to find codes. URL: {response.url}")
        return []

    complexities = []
    for ref in dom_references:
        try:
            time_complexity = await extract_complexity_from_reference(
                ref, TIME_COMPLEXITY_REGEX
            )
        except FailedComplexityExtraction:
            time_complexity, _ = extract_complexity_with_fallback(ref)

        try:
            space_complexity = await extract_complexity_from_reference(
                ref, AUXILIARY_SPACE_REGEX
```

```
            )
        except FailedComplexityExtraction:
            _, space_complexity = extract_complexity_with_fallback(ref)

        complexities.append(
            {
                "time_complexity": time_complexity,
                "trustable_time_complexity": bool(time_complexity),
                "space_complexity": space_complexity,
                "trustable_space_complexity": bool(space_complexity),
            }
        )

    main_name = response.soup.find("div", {"class": "article-title"}).text
    names = [main_name] + [
        extract_name(ref) if extract_name(ref) else main_name
        for ref in dom_references[1:]
    ]

    return [
        {
            "name": name,
            "time_complexity": complexity["time_complexity"],
            "trustable_time_complexity": complexity["
                trustable_time_complexity"],
            "space_complexity": complexity["space_complexity"],
            "trustable_space_complexity": complexity["
                trustable_space_complexity"],
            "url": str(response.url),
            "codes": code,
        }
        for name, complexity, code in zip(names, complexities, codes)
    ]
```

- saci/websites/geeks_for_geeks/login.py

```
import os

from scraper.session.http_session import HttpSession
from scraper.session.response import Response
from scraper.session.utils import Methods
from scraper.websites.geeks_for_geeks.settings import LOGIN_URL


def get_login_payload():
```

```
    return {
        "reqType": "Login",
        "user": os.getenv("USERNAME"),
        "pass": os.getenv("PASSWORD"),
        "rem": False,
        "to": "https://auth.geeksforgeeks.org/?to=https://www.geeksforgeeks.org/",
        "rem": "on",
        "g-recaptcha-response": None,
        "browserInfo": {},
    }


async def follow_login(session: HttpSession):
    await session.request(
        method=Methods.GET.value,
        data=get_login_payload(),
        url=LOGIN_URL,
    )
```

- saci/websites/geeks_for_geeks/settings.py

```
BASE_URL = "https://www.geeksforgeeks.org"

LOGIN_URL = "https://auth.geeksforgeeks.org/auth.php"

ALGORITHMS_LOCATION_URLS = [
    f"{BASE_URL}/fundamentals-of-algorithms",
    f"{BASE_URL}/data-structures",
]

HEADERS = {
    "Host": "www.geeksforgeeks.org",
    "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9",
    "Accept-Encoding": "gzip, deflate",
    "Upgrade-Insecure-Requests": "1",
    "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/104.0.0.0 Safari/537.36",
    "Sec-Fetch-Site": "none",
    "Sec-Fetch-Mode": "navigate",
    "Sec-Fetch-User": "?1",
    "Sec-Fetch-Dest": "document",
    "Sec-Ch-Ua": '"Chromium";v="105", "Not)A;Brand";v="8"',
```

```
    "Sec-Ch-Ua-Mobile": "?0",
    "Sec-Ch-Ua-Platform": '"Windows"',
}

ALGORITHM_LINKS_JS_SCRIPT = """
    () => {
        const algorithm_links = []

        const ordered_lists = document.getElementsByTagName('ol');
        for (ol of ordered_lists) {
            const hrefs = []
            const a_tags = ol.getElementsByTagName('a')
            for (a of a_tags) {
                hrefs.push(a.getAttribute('href'))
            }
            algorithm_links.push(hrefs)
        }

        return algorithm_links
    }
"""

TIME_COMPLEXITY_REGEX = {
    "word": [
        r"[Tt]ime [Cc]omplexit\w*[\s]?:|[Tt]ime [Cc]omplexit\w*[\s]?",
    ],
    "value": "time complexity",
    "fallback": r"O\s*\(.*?\)",
}

AUXILIARY_SPACE_REGEX = {
    "word": [
        r"[Aa]uxiliary [Ss]pace\s?:|[Aa]uxiliary [Ss]pace\s?",
        r"[Ss]pace [Cc]omplexit\w*\s?:|[Ss]pace [Cc]omplexit\w*\s?",
    ],
    "value": "auxiliary/space complexity",
    "fallback": r"O\s*\(.*?\)",
}

HTML_ELEMENTS_NAMES = ["p", "li", "ul", "i"]

COMMENTS_STARTING_STRINGS = ["//", "# ", "/*"]

LANGUAGES = ["C", "C#", "C++", "Python3", "Python", "PHP", "Java", "
    Javascript"]
```

- saci/websites/geeks_for_geeks/spider.py

```python
from asyncio import gather

from scraper.exceptions import FailedExtraction
from scraper.observability.log import scraper_log as log
from scraper.schema.data_schemas import ScrapedAlgorithm
from scraper.session.http_session import HttpSession
from scraper.session.response import Response
from scraper.session.utils import Methods
from scraper.utils import remove_duplicates, retry
from scraper.websites.geeks_for_geeks.extract import extract_data
from scraper.websites.geeks_for_geeks.login import follow_login
from scraper.websites.geeks_for_geeks.settings import
    ALGORITHMS_LOCATION_URLS, HEADERS


def parse_algorithm_schema(algorithms_data: list) -> list:
    return [ScrapedAlgorithm(**algorithm) for algorithm in algorithms_data]


def filter_algorithms_urls(algorithms_urls: list) -> list:
    return [href for href in algorithms_urls if "geeksquiz" not in href]


async def extract_algorithm_data(response: Response) -> list:
    if not response:
        return []

    data = await extract_data(response)
    if not data:
        raise FailedExtraction

    return data


def extract_algorithms_urls(response: Response) -> list:
    page_content = response.soup.find("div", {"class": "page_content"})
    return [
        a.attrs.get("href")
        for ol in page_content.find_all("ol")[1:]
        for a in ol.find_all("a")
        if a.attrs.get("href")
    ]


@retry(times=3, raise_exception=False, return_value=[])
async def follow_algorithm(session: HttpSession, url: str):
```

```
    return await session.request(
        method=Methods.GET.value,
        url=url,
        callbacks=[extract_algorithm_data, parse_algorithm_schema],
    )


async def follow_algorithms(session: HttpSession, algorithms_urls: list) ->
    list:
    algorithms = await gather(
        *[follow_algorithm(session, url) for url in algorithms_urls]
    )
    return sum(algorithms, [])


async def follow_algorithms_urls(session: HttpSession) -> list:
    algorithm_location_urls = await gather(
        *[
            session.request(
                method=Methods.GET.value,
                url=url,
                callbacks=[
                    extract_algorithms_urls,
                    filter_algorithms_urls,
                    remove_duplicates,
                ],
            )
            for url in ALGORITHMS_LOCATION_URLS
        ]
    )
    return sum(algorithm_location_urls, [])


async def run(url: str = None) -> list:
    session = HttpSession()
    session.default_headers = HEADERS

    log.info('Starting "Geeks for Geeks" algorithms extraction')
    await follow_login(session)

    algorithms_urls = [url] if url else await follow_algorithms_urls(
        session)
    return await follow_algorithms(session, algorithms_urls)
```

Apêndice C - Artigo SBC

SACI - UMA FERRAMENTA PARA EXTRAÇÃO DE ALGORITMOS E SUAS COMPLEXIDADES DE TEMPO E ESPAÇO

Augusto Silva de Oliveira[1], Carina Friedrich Dorneles[2], Ricardo José Pfitscher[3]

[1]Universidade Federal de Santa Catarina (UFSC)

Departamento de Informática e Estatística (INE) – Florianópolis – SC – Brazil

augusto.so@grad.ufsc.br, carina.dorneles@ufsc.br, ricardo.pfitscher@ufsc.br

Abstract. Determining the time and space complexity of a written code is a task that can be complex and is often essential, especially when aiming to achieve an efficient application. Despite being an important aspect in the education of developers, the analysis of algorithms regarding their complexity is a difficult task that is sometimes neglected. Some tools can make suggestions for code auto-completion as the programmer writes, but, unfortunately, they currently do not have the capability to consider complexities in their suggestions. Therefore, it is necessary for developers to have a solid understanding of algorithm analysis to ensure that the proposed solutions are efficient. With this in mind, this work proposes the creation of a data extraction tool capable of extracting algorithm codes along with their respective complexities in order to form a database. This database can be used for training tools, whether through machine learning or other techniques, aimed at determining the complexity of a piece of code and assisting developers in their routines. By the end of the work, a database with 501 distinct algorithms accompanied by their complexities was obtained, with good precision and data quality, as evidenced by the metrics used.

Resumo. Determinar a complexidade de tempo de execução e de espaço em um código escrito é uma tarefa que pode ser complexa e que muitas vezes é essencial, considerando-se que deseje-se obter uma aplicação eficiente. Apesar de ser um aspecto importante na formação do desenvolvedor, a análise de algoritmos quanto a sua complexidade é uma tarefa difícil que por vezes é negligenciada. Algumas ferramentas são capazes de fazer sugestões para o preenchimento automático do código conforme

o programador esteja escrevendo, mas, até o momento atual, ela infelizmente não possui capacidade de considerar as complexidades em suas sugestões. Por isso é necessário que o desenvolvedor tenha conhecimento sólido sobre análise de algoritmos para garantir que as soluções propostas sejam eficientes. Tendo isso em vista, esse trabalho propõe a criação de uma ferramenta de extração de dados, capaz de extrair códigos de algoritmos juntos de suas respectivas complexidades a fim de formar uma base de dados. Essa base de dados poderá ser usada para o treinamento de ferramentas, seja através de aprendizado de máquina ou outras técnicas, que visam determinar a complexidade de um trecho de código e auxiliar os desenvolvedores em suas rotinas. Ao fim do trabalho obteve-se uma base de dados com 501 algoritmos distintos acompanhados de suas complexidades, com boa precisão e qualidade dos dados, comprovado pelas métricas utilizadas.

1 INTRODUÇÃO

A complexidade de algoritmos mede o consumo de recursos computacionais, como tempo de execução e espaço em memória, em relação ao tamanho da entrada, sendo uma medida de eficiência essencial para o desempenho de uma aplicação (CORMEN, et al., 2009). Conhecer a complexidade de um trecho de código permite prever sua eficiência e otimizá-lo (SEDGEWICK, 2009). No entanto, identificar a complexidade pode ser uma tarefa difícil para muitos programadores devido à falta de hábito, prática ou conhecimento (HALL; CHAPMAN, 2015).

Há uma carência de dados sobre algoritmos e suas complexidades, como evidenciado por Sikka et al. (SIKKA et al., 2020a), que trabalharam com um banco de dados restrito de 931 códigos em Java. Pfitscher et al. (PFITSCHER et al., 2023) ampliaram esse banco de dados para 1325 amostras com dados extraídos do site Geeks for Geeks. O SACI visa expandir ainda mais esses dados, criando um repositório robusto e abrangente de algoritmos e suas complexidades, disponibilizado online.

A ferramenta proposta extrai algoritmos e suas complexidades de fontes pré-selecionadas, resultando em um dataset aberto para a comunidade científica e desenvolvedores. Exemplos de aplicação incluem ferramentas que retornam a complexidade de um código, como o GitHub Copilot, que poderia sugerir melhorias de desempenho. O SACI utiliza técnicas de *web scraping* para construir essa base de dados (ZHAO, 2017).

Um exemplo prático é apresentado por Rodenbusch (RODENBUSCH, 2023), com a API RTCC (*Running Time Complexity Calculator*), que utiliza dados do SACI para prever padrões de complexidade com alta precisão. Ferramentas de IA treinadas com esses dados são cruciais para melhorar a capacidade de predição e eficiência (WANG et al., 2022).

2 OBJETIVOS GERAL E ESPECÍFICOS

Desenvolver uma ferramenta de *scraping* capaz de extrair algoritmos e suas complexidades, com o objetivo de gerar uma base de dados com o conteúdo extraído.

Objetivos específicos:

1. Realizar a extração com expressões regulares
2. Realizar a extração com OpenAI
3. Tratar os dados extraídos e padronizá-los
4. Caracterizar os dados extraídos

3 TRABALHOS RELACIONADOS

Este estudo considerou quatro abordagens relacionadas ao desenvolvimento de ferramentas, apesar de não focarem na extração de complexidades, pois compartilham objetivos e características específicas.

3.1 Learning Based Methods for Code Runtime Complexity Prediction

Este trabalho de (SIKKA et al., 2020b) aborda a previsão da complexidade de tempo de execução de códigos utilizando aprendizado de máquina. Os autores criaram um conjunto de dados próprio, 'CoRCoD', devido à falta de dados abertos, mas enfrentaram problemas de acurácia devido ao tamanho reduzido do conjunto de dados.

3.2 qFEx - um crawler para busca e extração de questionários de pesquisa em documentos HTML

O (MATHIAS; DORNELES, 2021) é um web crawler desenvolvido para coletar questionários de pesquisa da internet. Utiliza heurísticas para identificar e extrair questionários, enfrentando desafios devido à diversidade de layouts de páginas web. Foi implementado em Java usando a biblioteca Crawler4j.

3.3 Ferramenta para coleta e comparação de dados de publicações acadêmicas dos professores com o currículo Lattes

Desenvolvido por (BRANCO, 2018), este *scraper* complementa o currículo Lattes com dados de sites como ResearchGate. Implementado em Java, utiliza bibliotecas como 'jsoup' e 'Java-string-similarity', mas enfrenta problemas de bloqueio e manutenção devido a mudanças nos sites.

3.4 CrawlEX: uma ferramenta para extração de dados na web configurável através de exemplos

O CrawlEX, descrito por (LESSA, 2022), é uma ferramenta configurável para extração de artigos da web, com uma interface visual em Python. Usa bibliotecas como 'beautifulsoup' e 'sqlite' para a extração baseada em exemplos fornecidos pelo usuário.

3.5 Desenvolvimento de API para predição da complexidade de tempo de execução de códigos por meio de AutoML

Em (RODENBUSCH, 2023), o autor combina bases de dados de *scraping* para criar modelos de classificação de complexidade de código, resultando em uma API que permite a predição de classes de eficiência e complexidade para códigos em Java, com acurácia de até 88,38%.

3.6 Análise dos trabalhos

Ao comparar quatro das obras apresentadas que implementaram alguma ferramente de extração e contrastando-as com o trabalho atual, é perceptível que, embora haja distintos focos de extração, a maioria dos trabalhos é direcionada de forma especializada. A Tabela 1 ilustra essas diferenças.

O CrawlEX se destaca por tentar abordar uma perspectiva mais generalizada; no entanto, ainda necessita de configurações prévias, o que pode influenciar sua eficiência que não atinge a totalidade. Independentemente da abordagem, é comum que todos utilizem bibliotecas especializadas para navegar na estrutura DOM da página e nos elementos HTML. Essa dependência de configuração, seja ela feita pelo usuário ou já incorporada, é crucial, visto que cada página da web possui suas particularidades.

A presença ou ausência de uma interface gráfica parece estar diretamente ligada ao público-alvo de cada trabalho. Trabalhos como ECL e CrawlEX, que possuem interface gráfica, parecem ser orientados para usuários finais. Por outro lado, obras como o qFEx, CoRCoD e o SACI estão mais voltadas para a extração de dados para alimentar bases, não necessitando, portanto, de um elemento visual.

Em relação às linguagens de programação adotadas, duas predominam: Python e Java. O que é particularmente interessante é que, independentemente da linguagem escolhida, as bibliotecas auxiliares empregadas, como 'beautifulsoup' e 'jsoup', visam solucionar problemas semelhantes, demonstrando um padrão de desafios comuns na área de extração de dados web.

Tabela 1 - Comparação de trabalhos

Comparação de trabalhos					
Trabalho	Tipo	Item Alvo	Técnica	Algoritmo de Extração	Interface Gráfica
qFEx	Focado	Questionários	Heurísticas	Do autor	Não
ECL	Focado	Publicações	Similaridade textual	Do autor	Sim
CrawlEX	Genérico	Artigos	Comparação textual	Do autor e Ratcliff/Obershelp	Sim
CoRCoD	Focado	Algoritmos e Complexidades	API e Scraping	Do autor	Não
SACI	Focado	Algoritmos e Complexidades	Expressões regulares e OpenAI	Do autor	Não

4 SACI - SCRAPER DE ALGORITMOS E COMPLEXIDADES

4.1 Visão Geral

A ferramenta SACI extrai dados de algoritmos e suas complexidades de páginas web específicas, utilizando conceitos de *web scraping*. O fluxo da ferramenta é executado de forma assíncrona para aumentar a eficiência, e os dados extraídos são armazenados em uma base de dados MongoDB ou em um arquivo JSON local.

4.2 Implementação

A estrutura da ferramenta é modular, permitindo a inclusão de novas páginas. As principais pastas são:

1. **Results**: Armazena os resultados das extrações.
2. **Observability**: Contém os códigos responsáveis pelos logs e métricas de desempenho.
3. **Schema**: Define os esquemas JSON para validação dos dados.
4. **Session**: Implementa a sessão HTTP para requisições assíncronas.
5. **Websites**: Contém a lógica específica para cada página implementada.

4.3 Geeks4Geeks

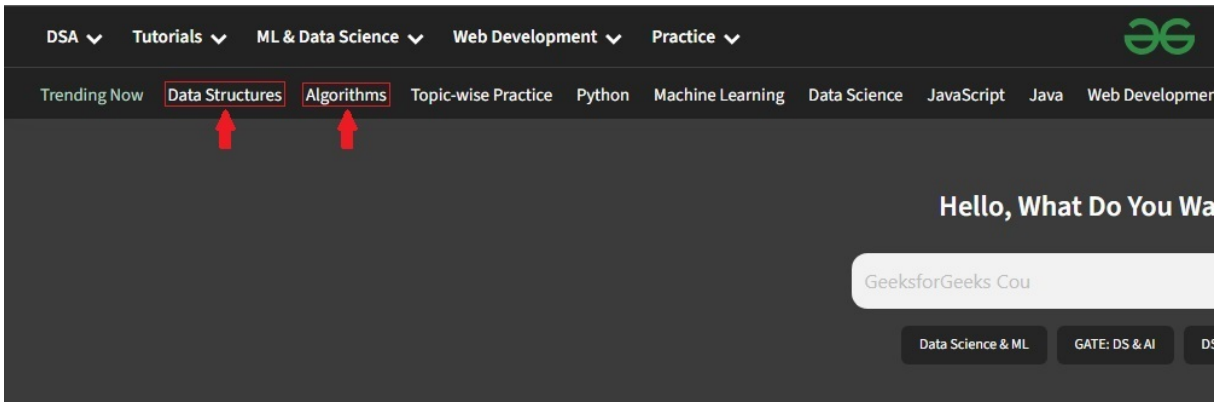
O Geeks4Geeks [1] é um site com conteúdo voltado para programadores com uma variedade de materiais para aprendizado e, também, de entretenimento. repleto de

[1] https://www.geeksforgeeks.org/

artigos, tutoriais, aulas, seminários, é um ótimo lugar para quem procura aprender algo novo.

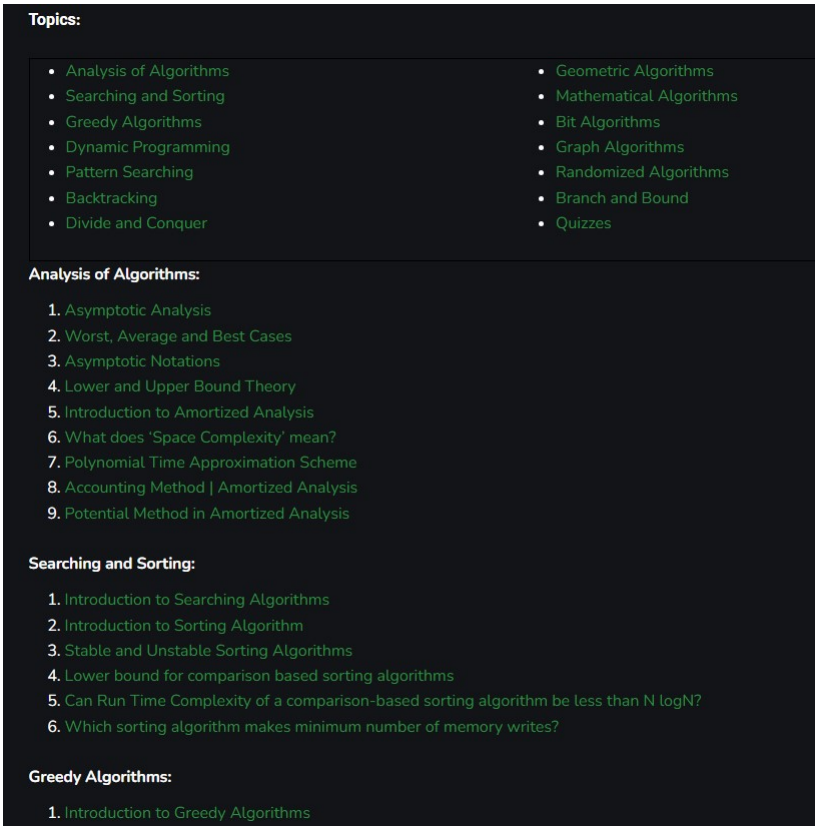
No topo do site, Figura 1, ficam evidenciados os principais tópicos: algoritmos de estrutura de dados, tutoriais, ciência de dados e aprendizado de máquina, desenvolvimento web, entre outros. Destes, o que interessa para o trabalho são os referentes a algoritmos e estrutura de dados. Dentro deles existem subtópicos e links, Figura 2, que direcionam para páginas com implementações de código, em múltiplas linguagens, acompanhadas das complexidades de tempo e espaço Figura . Por conta dessas características, essas duas seções, apontadas na Figura 3, foram escolhidas como fontes de dados a serem extraídos pelo SACI. Nessas, existe uma variedade de links de algoritmos, organizados conforme seus tipos e temas.

Figura 1 - Menu Geeks4Geeks



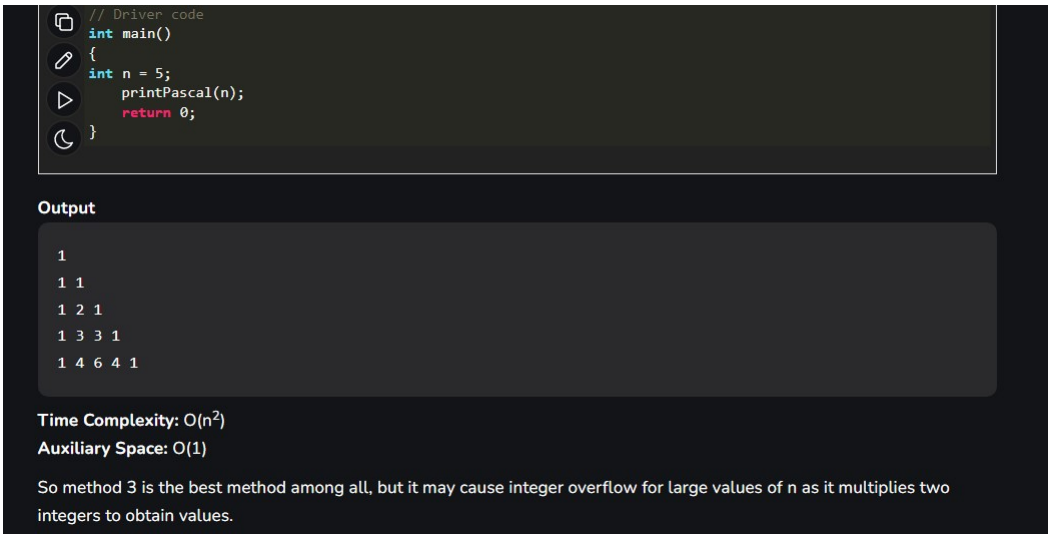
Fonte: elaborada pelo autor

Figura 2 - Página de algoritmos



Fonte: elaborada pelo autor

Figura 3 - Complexidades



Fonte: elaborada pelo autor

4.4 Scraper

O scraper segue um fluxo para localizar e extrair os códigos dos algoritmos e suas complexidades das páginas do Geeks4Geeks, utilizando expressões regulares e métodos 'fallback' para garantir a extração eficiente.

4.5 Extract

Foram desenvolvidas duas implementações para a extração. Uma das versões faz uso de expressões regulares como o principal meio de identificação e extração das complexidades. Já a outra, no momento da extração, utiliza inteligência artificial através da API(*Application Programming Interface*) do OpenAI para a identificação e extração das complexidades. Todos os resultados das extrações podem ser encontrados no repositório[2] do GitHub e também no Kaggle[3].

4.6 Extração com expressões regulares

Cada página do site pode conter mais de um código/implementação. Esses códigos ficam localizados em um elemento HTML chamado aqui de bloco. Cada bloco contém o algoritmo codificado em diversas linguagens de programação e uma complexidade associada ao algoritmo. Ao acessar à página o método procura pelas seções que contém esses blocos, linha 2 do Algoritmo 1, guarda as referências de posição desses blocos na árvore DOM junto dos códigos. Em seguida, procura em volta de cada referência pelas complexidades de tempo e espaço, cada uma através de uma seleção prévia de expressões regulares, linhas 7 a 18 do Algoritmo 1. Por fim, extraímos o título do algoritmo na linha 19 do Algoritmo 1 e adicionamos um objeto resultado a uma lista, linhas 20 a 32 do Algoritmo 1. Para a extração dos títulos dos códigos, linha 19, também utiliza-se a referência de localização dos blocos de código. É extraído o título principal da página que geralmente é o título do primeiro algoritmo e os restantes, para retornar uma lista de dicionários Python contendo o nome do algoritmo, a complexidade de tempo, a complexidade de espaço, o URL e o código em todas as linguagens, linhas 20 a 32. Essa lista é tratada pelo *scraper* na última etapa retornando os dados de acordo com o esquema JSON. As chaves 'tempoConfiavel' e 'espaçoConfiavel' do objeto resultado tem relação com o uso ou não do método 'fallback' apresentado mais a frente. Caso o método seja utilizado essas chaves recebem como valor o booleano 'True', caso contrário 'False'.

---

[2] https://github.com/Augustives/SACI

[3] https://www.kaggle.com/datasets/augustives/algorithms-and-their-complexities

Algoritmo 1 - Código mostra a função de extração principal em alto nível

```
1: Function extrairDados(response)

2: codigos, referencisaDom ← extrairCodigosReferencias(response)
3: if not codigos then
4:     Log aviso
5:     return []
6: end if

7: complexidades ← []
8: for ref in referencisaDom do
9:     try:
10:    complexidadeTempo ← extrairComplexidadeDeReferencia(ref, regexTempo)
11:    catch Erro:
12:    complexidadeTempo ← extrairComplexidadeComFallback(ref)

13:    try:
14:    complexidadeEspaço ← extrairComplexidadeDeReferencia(ref, regexEspaço)
15:    catch Erro:
16:    complexidadeEspaço ← extrairComplexidadeComFallback(ref)

17:    Insere complexidades na lista
18: end for

19: titulos ← extrairTitulos(ref)

20: resultados ← []
21: for (nome, complexidade, codigo) in zip(nomes, complexidades, codigos) do
22:    resultado ← {
23:            'nome': nome,
24:            'complexidadeTempo': complexidade['complexidadeTempo'],
25:            'tempoConfiavel': complexidade['tempoConfiavel'],
26:            'complexidadeEspaço': complexidade['complexidadeEspaço'],
27:            'espaçoConfiavel': complexidade['espaçoConfiavel'],
28:            'url': url,
29:            'codigo': codigo,
30:    }
31:    Insere resultado na lista
32: end for

33: return resultados =0
```

No momento da extração das complexidades, são utilizados a referência da localização dos códigos, linhas 10 e 14 do Algoritmo 1, métodos do 'BeautifulSoup' e expressões regulares. A extração conta com dois caminhos: o padrão com regras mais específicas e um que chamamos de 'fallback', no qual são utilizadas regras mais generalizadas e inclusivas a fim de conseguir extrair a informação caso ocorra alguma falha

durante o fluxo padrão.

O método padrão, representado no Algoritmo 2, procura por todas as palavras *Time Complexity* ou *Space Complexity* nas proximidades do bloco de códigos, linha 4. Para cada palavra existe uma lista de expressões regulares, para cada palavra encontrada elas são inseridas em uma lista de 'matches', linha 3 e 6. Então, procura o 'match' mais próximo da referência, linha 8, com um cálculo simples: para cada 'match' da lista subtrai-se a linha em que a referência está com a linha que o 'match' encontra-se, dessa forma, o menor valor dentre todos os calculados é o mais próximo. A seguir, o método percorre outra lista de expressões regulares, dessa vez focada em capturar os valores das complexidades. Para cada expressão regular de valor, itera os elementos HTML, para os quais é feita uma pesquisa, linha 11. Caso nenhuma complexidade consiga ser extraída, é levantado um erro de extração, linha 17.

Sendo o erro levantado, o fluxo de 'fallback', tenta extrair novamente a complexidade, e uma vez utilizado esse método, uma 'flag' é adicionada aos resultados indicando seu uso.

Algoritmo 2 - Código mostra a função 'extrairComplexidadeDeReferencia' em alto nível

```
 1: Function extrairComplexidadeDeReferencia(referencia, regex)
 2: for regex in regexes['palavra'] do
 3:     matches ← [ ]
 4:     palavra ← referencia.find_next(string=compile(regex))

 5:     if palavra then:
 6:         Insere palavra em matches
 7:     end if

 8:     match_mais_proximo ← Pega o match mais próximo da ref

 9:     for regex in regexes['valor'] do
10:         for elemento in elementosHTML['valor'] do
11:             complexidade ← busca_regex(regex, match_mais_proximo)

12:             if complexidade then:
13:                 return complexidade
14:             end if
15:         end for
16:     end for

17:     raise FalhaExtraçaoComplexidade
18: end for
```

A Figura 4 apresenta as expressões regulares criadas e utilizadas na extração dos dados. Aquelas que buscam pelas palavras, linhas 3, 16 e 17, tentam encontrar *time complexity*, *space complexity* e suas variações de escrita. As responsáveis

pelos valores, linhas 5 a 10 e 19 a 28, são uma combinação da palavra com o valor da complexidade, buscando, por exemplo, algo como *Time Complexity*: O(log N). Todas foram criadas com base nos diversos padrões com que as complexidades são descritas nos textos das páginas do site. Por fim, tem-se a que é usada no método 'fallback, linhas 11 e 29.

Figura 4 - Regex de extração

```
TIME_COMPLEXITY_REGEX = {
    'word': [
        r'[Tt]ime [Cc]omplexit\w*[\s]?:|[Tt]ime [Cc]omplexit\w*[\s]?',
    ],
    'value': [
        r'[Tt]ime [Cc]omplexit\w*\s?:.*?(O\s*\(.*?\))',
        r'[Tt]ime [Cc]omplexit\w*\s?.*?(O\s*\(.*?\))',
        r'[Tt]ime [Cc]omplexit\w*\s.*?\sis\s.*?(O\s*\(.*?\))',
        r'[Tt]ime [Cc]omplexit\w*\s?:?.*?\sis\s(\w*)',
    ],
    'fallback': r'O\s*\(.*?\)',
}

AUXILIARY_SPACE_REGEX = {
    'word': [
        r'[Aa]uxiliary [Ss]pace\s?:|[Aa]uxiliary [Ss]pace\s?',
        r'[Ss]pace [Cc]omplexit\w*\s?:|[Ss]pace [Cc]omplexit\w*\s?',
    ],
    'value': [
        r'[Aa]uxiliary [Ss]pace\s?[\[Cc\]omplexit\w*]?:.*?(O\s*\(.*?\))',
        r'[Aa]uxiliary [Ss]pace\s?[\[Cc\]omplexit\w*]?.*?(O\s*\(.*?\))',
        r'[Ss]pace [Cc]omplexit\w*\s?:.*?(O\s*\(.*?\))',
        r'[Ss]pace [Cc]omplexit\w*\s?.*?(O\s*\(.*?\))',
        r'[Aa]uxiliary [Ss]pace\s?[\[Cc\]omplexit\w*]?.*?\sis\s.*?(O\s
            *\(.*?\))',
        r'[Ss]pace [Cc]omplexit\w*\s.*?\sis\s.*?(O\s*\(.*?\))',
        r'[Aa]uxiliary [Ss]pace\w*\s?:?.*?\sis\s(\w*)',
        r'[Ss]pace [Cc]omplexit\w*\s?:?.*?\sis\s(\w*)',
    ],
    'fallback': r'O\s*\(.*?\)',
}
```

4.7 Extração com OpenAI

Para a extração com OpenAI utilizou-se da API oferecida [4], e para sua implementação foi utilizada uma biblioteca do Python chamada de Langchain [5]. Ela oferece uma integração fácil com modelos de linguagem ou LLMs(*Large Language Models*), como o GPT-3 e outros similares, além de oferecer aos desenvolvedores um conjunto de ferramentas e abstrações que facilitam a criação de aplicações interativas e complexas. A premissa central da biblioteca Langchain é permitir a construção de correntes de 'prompts' e respostas; ou seja, sequências de interações com modelos de linguagem que podem ser encadeadas para realizar tarefas sofisticadas. Dentro do arquivo *utils* do SACI está definida uma classe que faz o uso desta biblioteca e realiza essa comunicação com a OpenAI.

Algoritmo 3 - Código mostra a sessão http em alto nível

```
1: Class LlmSearcher
2: throttler ← Throttler(rate_limit=100, period=60)

3: Init()
4: self.LLM ← OpenAI(openai_api_key=...)
5: self.PROMPT ← PromptTemplate.from_template(template_do_prompt)
6: self.LLM_CHAIN ← LLMChain(prompt=self.PROMPT, llm=self.LLM)

7: Method search(complexidade, texto)
8: async with throttler:
9: resposta ← self.LLM_CHAIN.run(complexity=complexidade, text=texto)
10: return json.loads(resposta)
```

A classe, representada no Algoritmo 3, atua como um *singleton*, isto é, só existe uma instância sua durante a vida útil da aplicação. Dentro dela estão definidos o LLM que conecta com o modelo de linguagem da OpenAI, o 'PROMPT' que define como o modelo irá responder e a LLM_CHAIN que une o 'PROMPT' com o modelo em si. Além disso, existe um 'throttler' que limita quantas 'requests' são feitas por minuto, pois, a API da OpenAI possui certas restrições.

Figura 5 - Prompt do LLM

```
I am using you during a scraping operation in which I need to extract from
    a piece of text the values of complexity/code complexity.
Consider that sometimes the space complexity is also called as auxiliary
    space.
```

[4] https://openai.com/index/openai-api/

[5] https://www.langchain.com/

```
Consider that both complexitys often come in this format: 'Some type of
    complexity: value of complexity'.
But they can also come inside the text like: 'The type of complexity of the
    given ... is ...'.
Give the answer in JSON format with no line breaks, with a key called '
    complexity' and the value for the key is your answer.
If you cant determine the answer give the json with a null in the value.
What is the {complexity} that is written in the following text: '{text}'?
```

O 'PROMPT' utilizado é descrito na Figura 5, com ele é possível explicar o contexto em que o SACI está, demonstrando alguns exemplos de como os dados irão chegar, seus formatos e diferenças, sendo ainda explicado como a resposta deverá ser retornada e qual pergunta é feita. Com isso o modelo de linguagem consegue entender que está sendo feito a pesquisa pelos valores de complexidade do texto que ele está recebendo.

O método de extração das complexidades sobre as referências também sofreu uma pequena alteração. Não é mais iterado a lista de 'regexes' responsáveis pela extração dos valores de complexidade, como era feito no antes, pois esse trabalho agora é feito pelo LLM.

5 EXPERIMENTOS

Para cada página em que foi implementada um *scraper* foram realizadas avaliações a fim de avaliar se a extração das informações teve um bom resultado ou não. Tentou-se garantir que a quantidade de extrações fosse boa e que as informações extraídas estivessem corretas. A seguir, é apresentada a configuração do ambiente que executou o *scraper*, descrevem as métricas utilizadas, caracterizam os dados extraído e detalham os experimentos e seus respectivos resultados a partir do uso do SACI.

5.1 Métricas

Para caracterizar a fonte de dados foram criados histogramas e gráficos para exibir o comportamento e a distribuição dos dados extraídos. Esses gráficos ajudam a visualizar padrões e tendências que podem não ser imediatamente aparentes através de uma simples observação.

Para melhor entender os resultados de cada extração foram selecionadas métricas de avaliação. A partir dos resultados é criado uma matriz de confusão e calculada a acurácia, revocação, precisão e F1-Score. Utilizam melhor essas métricas em conjunto do que individualmente, pois cada uma oferece uma perspectiva única sobre a performance do processo. Estas serão usadas para comparar a extração realizada com expressões regulares e a feita com OpenAi.

5.2 Resultados

A partir da análise do SACI na versão com expressões regulares, considerando as métricas citadas anteriormente, pode-se observar que os resultados obtidos mostraram um desempenho promissor na extração de ambas as complexidades, conforme demonstram as matrizes de confusão para tempo e espaço da Figura 6.

Figura 6 - Resultados Regex

		Valor Predito	
		Sim	Não
Real	Sim	391	11
	Não	88	11

Fonte: elaborada pelo autor

		Valor Predito	
		Sim	Não
Real	Sim	419	1
	Não	47	34

Fonte: elaborada pelo autor

A validação dos resultados foi feita de forma manual onde cada URL, onde ocorreu alguma extração de algoritmo, foi acessada e comparados os valores que foram extraídos com os valores presentes na página web.

Nota-se um bom desempenho na identificação de verdadeiros positivos para ambos os tipos de complexidade, com uma quantidade significativa de registros corretamente identificados pelo SACI. Porém, na complexidade de tempo observa-se um número considerável de falso-positivos, totalizando 88 casos. Isso indica que alguns dados errôneos foram incluídos na base de dados. A explicação para existência desse número um pouco mais elevado de falso-positivos é o uso do método de 'fallback', já que o mesmo parte do princípio de ser genérico e coletar a qualquer custo alguma complexidade.

Para falso-negativos, nota-se também sua existência significativa apenas para a complexidade de tempo, com 11 registros, o que sugere que alguns dados foram ignorados durante a extração. A baixa quantidade de verdadeiro-negativos na complexidade de tempo pode significar uma menor capacidade de filtrar dados irrelevantes.

Por outro lado, a complexidade de espaço apresentou um desempenho superior. A quantidade de verdadeiros negativos foi maior, com 34 registros, e os falsos negativos foram mínimos, com apenas 1 registro, mostrando uma melhor performance em comparação com a complexidade de tempo. Isso se deve ao fato de que a complexidade de espaço é apresentada de forma mais padronizada em todas as páginas, resultando em uma melhor identificação dos dados.

Na versão OpenAI notamos uma melhora geral em todos os parâmetros se comparados com a versão padrão, mas alguns apresentaram diferenças significativas.

Figura 7 - Resultados OpenAI

		Valor Predito	
		Sim	Não
Real	Sim	451	6
	Não	32	12

Fonte: elaborada pelo autor

		Valor Predito	
		Sim	Não
Real	Sim	424	1
	Não	37	39

Fonte: elaborada pelo autor

Tabela 2 - Métricas das complexidades de tempo e espaço

Métrica	**Tempo Regex**	**Espaço Regex**	**Tempo OpenAI**	**Espaço OpenAI**
Acurácia	0,8023	0,9041	0,9241	0,9241
Precisão	0,8162	0,8991	0,9337	0,9197
Revocação	0,9726	0,9976	0,9868	0,9976
F1 Score	0,8876	0,9458	0,9595	0,9571

Fonte: elaborada pelo autor

Os resultados alcançados pelo SACI são motivo de encorajamento. Eles evidenciaram alta precisão, revocação e acurácia em todas as complexidades analisadas. Notadamente, a implementação das técnicas da OpenAI trouxe melhorias substanciais sobre a versão padrão em vários aspectos críticos. Contudo, é importante reconhecer a dependência desses resultados à estrutura da fonte de dados. Mudanças futuras nos sites podem influenciar o desempenho da ferramenta.

A versão aprimorada do SACI, equipada com os modelos da OpenAI, mostrou uma redução expressiva tanto em falso-positivos quanto em falso-negativos, um diferencial que potencializa a confiabilidade dos dados extraídos. Tal performance sugere que, mesmo diante de estruturas de dados variáveis nos sites-alvo, a ferramenta possui a versatilidade para se adaptar e manter alto padrão de qualidade.

5.3 Fonte de dados

Todos os dados usados neste trabalho tiveram sua extração no dia ’24/08/2023’, conforme descrito previamente na Seção 4.4. Foi realizada uma análise sobre cada página onde houve a extração de uma ou mais complexidades. De um total de 235

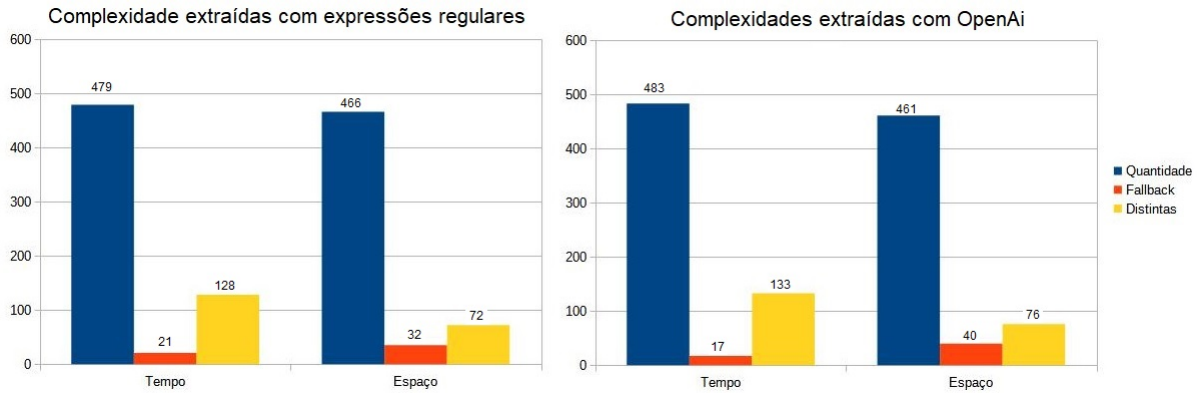
URLs, com 501 algoritmos disponíveis através delas, para cada página, executou-se uma função que percorria todos os nodos da árvore HTML e retornava qual a profundidade máxima encontrada e quantidade de nodos existentes.

5.4 Conjunto de dados extraídos

A caracterização do conjunto de dados extraídos foi feita, primeiramente, avaliando o quanto a ferramenta de coleta utilizou da técnica de 'fallback' para extrair as complexidades. Os títulos dos algoritmos e os outros atributos coletados não fazem parte da caracterização. Conforme a Figura 8, 479 algoritmos tiveram suas complexidades de tempo extraídas e 466 suas complexidades de espaço; dentre estes, 21 fizeram uso do 'fallback' para extrair a complexidade de tempo e 32 para a de espaço. Foram obtidas 128 complexidades de tempo distintas e 72 complexidades de espaço distintas.

Um dos motivos para não se ter todas as complexidades de tempo extraídas com sucesso é a falta de um padrão na página. Em algumas páginas as complexidades estão escritas de maneira diferente, em locais diferentes, fazendo, assim, com que a expressão regular não as identifique. No caso da complexidade de espaço, alguns algoritmos realmente não a possuem e, em outros casos, ocorreu o mesmo problema que com a extração da complexidade de tempo.

Figura 8 - Complexidades extraídas



Fonte: elaborada pelo autor

Na nova versão do método de extração, que incorpora as capacidades da OpenAI, observou-se um incremento na eficiência da extração de dados. Como exibido na Figura 8, esta versão avançada conseguiu extrair com sucesso as complexidades de tempo de 483 algoritmos, e as complexidades de espaço de 461 algoritmos, superando um pouco a marca anterior.

Além disso, a dependência da técnica de 'fallback' foi reduzida para apenas 17 casos na extração das complexidades de tempo indicando, com isso, uma melhoria. Nas complexidades de espaço houve um pequeno aumento de 32 para 40 casos. Em termos de diversidade, o SACI identificou 133 complexidades de tempo distintas e 76 de espaço, um aumento em relação às 128 e 72 variantes respectivamente encontradas na versão que utilizava expressões regulares.

A classificação das complexidades em ambas versões foi feita através de expressões regulares, utilizando-se de um bloco de código de condicionais. Caso a expressão capturasse algo na complexidade, significa que tal complexidade é pertencente a tal categoria.

```
    Constante = r'O\(1\)'
    Linear = r'O\(\w\)'
    Exponencial = r'O\(\d\^?\w\)'
    Fatorial = r'O\(.*!.*\)'
    Logarítmica = r'O\(.*log.*\)'
    Polinomial = r'O\(\w\s*[\^\*\+]?\s*[\d\w\s]+.*\)' e r'O\(\d+\(\w\*\w\)'
```

6 CONSIDERAÇÕES FINAIS E TRABALHOS FUTUROS

A criação de ferramentas para identificar a complexidade de algoritmos e gerar *insights* para o usuário requer uma base de dados de algoritmos e suas complexidades. O SACI apresentou resultados satisfatórios, formando uma base de dados adequada com eficácia comprovada pelas métricas calculadas.

Durante o desenvolvimento, foram enfrentadas dificuldades na construção de um *scraper* genérico devido à diversidade de layouts de páginas. A reutilização de trechos de código facilitou futuras implementações. A detecção e extração dos elementos desejados foram realizadas com expressões regulares e um método *fallback*. Problemas com *anti-bots* foram mitigados com o envio de cabeçalhos adequados e limitações de requisições.

Para trabalhos futuros, melhorias podem incluir:

1. **Melhorar as métricas:** Adicionar mais atributos além das complexidades de tempo e espaço.
2. **Melhorar as técnicas de extração:** Refinar expressões regulares e usar inteligência artificial para identificar e extrair dados relevantes.
3. **Expandir fontes de dados:** Implementar a coleta de mais códigos e complexidades de diversas fontes.

A base de dados resultante possui 501 algoritmos que podem ser utilizados por pesquisadores para diversos fins, como:

1. **Análise Comparativa de Algoritmos:** Comparar o desempenho de diferentes algoritmos.
2. **Otimização de Algoritmos:** Identificar oportunidades para melhorar algoritmos existentes.
3. **Previsão de Desempenho de Algoritmos:** Construir modelos preditivos para estimar o desempenho dos algoritmos.
4. **Estudo Teórico de Algoritmos:** Investigar características teóricas dos algoritmos.
5. **Desenvolvimento de Novos Algoritmos:** Inspirar e orientar o desenvolvimento de novos algoritmos.

A base de dados construída oferece uma riqueza de informações que podem ser exploradas em diversas áreas, proporcionando avanços significativos em análise, otimização e desenvolvimento de algoritmos.

7 REFERÊNCIAS

CORMEN, Thomas H. et al. Algoritmos: Estruturas de Dados e Algoritmos para Resolução de Problemas. Rio de Janeiro: LTC, 2009.

SEDGEWICK, Robert. Analyzing algorithms. Communications of the ACM, ACM Press, v. 52, n. 1, p. 139–148, 2009.

HALL, Mark; CHAPMAN, Barry. Measuring algorithm complexity. ACM SIGPLAN Notices, ACM Press, v. 48, n. 1, p. 16–23, 2015.

SIKKA, Jagriti et al. Learning Based Methods for Code Runtime Complexity Prediction. In: SPRINGER. EUROPEAN Conference on Information Retrieval. [S.l.: s.n.], 2020. P. 313–325.

PFITSCHER, Ricardo J. et al. Estimating Code Running Time Complexity with Machine Learning. In: NALDI, Murilo C.; BIANCHI, Reinaldo A. C. (Ed.). Intelligent Systems. Cham: Springer Nature Switzerland, 2023. P. 400–414.

ZHAO, Bo. Web scraping. Encyclopedia of big data, Springer Living ed. Cham, p. 1–3, 2017.

RODENBUSCH, Gabriel Braun. Desenvolvimento de API para predição da complexidade de tempo de execução de códigos por meio de AutoML. [S.l.: s.n.], 2023. TCC (graduação) - Universidade Federal de Santa Catarina, Campus Joinville, Engenharia Mecatrônica.

WANG, Meng et al. A Survey on Large-Scale Machine Learning. IEEE Transactions on Knowledge and Data Engineering, v. 34, n. 6, p. 2574–2594, 2022. DOI: 10.1109/TKDE.2020.3015777.

SIKKA, Jagriti et al. Learning Based Methods for Code Runtime Complexity Prediction. In: JOSE, Joemon M. et al. (Ed.). Advances in Information Retrieval. Cham: Springer International Publishing, 2020. P. 313–325.

MATHIAS, Gilney; DORNELES, Carina. qFEx - um crawler para busca e extração de questionários de pesquisa em documentos HTML. In: ANAIS do III Dataset Showcase Workshop. Rio de Janeiro: SBC, 2021. P. 1–10. DOI: 10.5753/dsw.2021.17409.

BRANCO, Arthur Machado. Ferramenta para coleta e comparação de dados de publicações acadêmicas dos professores com o currículo lattes.

LESSA, Marcos Aurélio. CrawlEX: uma ferramenta para extração de dados na web configurável através de exemplos.