Universidade Federal do ABC
Centro de Matemática, Computação e Cognição
Projeto de Graduação em Ciência da Computação

Graphcol: Biblioteca de Algoritmos para Coloração de Grafos

Wesley Lima de Araujo

Santo André - SP, Maio de 2023

Wesley Lima de Araujo

Biblioteca de Algoritmos para Coloração de Grafos

Projeto de Graduação em Computação apresentado ao Bacharelado em Ciência da Computação, como parte dos requisitos necessários para a obtenção do Título de Bacharel em Ciência da Computação.

Universidade Federal do ABC – UFABC

Centro de Matemática, Computação e Cognição

Bacharelado em Ciência da Computação

Orientador: Carla Negri Lintzmayer

Santo André - SP

Maio de 2023

Wesley Lima de Araujo
Biblioteca de Algoritmos para Coloração de Grafos/ Wesley Lima de Araujo. – Santo André - SP, Maio de 2023-
Orientador: Carla Negri Lintzmayer

Projeto de Graduação em Computação (PGC) – Universidade Federal do ABC – UFABC
Centro de Matemática, Computação e Cognição
Bacharelado em Ciência da Computação, Maio de 2023.

1. Grafos 2. Coloração de Grafos 3. Python I. Carla Negri Lintzmayer. II. Universidade Federal do ABC. III. Bacharelado em Ciência da Computação. IV. Graphcol: Biblioteca de Algoritmos para Coloração de Grafos

Aos meus pais

Agradecimentos

Gostaria acima de tudo agradecer aos meus pais pelo apoio e suporte durante os anos que passei na UFABC. Talvez vocês não saibam, mas nada foi mais importante que isso. Meu muito obrigado vai desde as palavras de conforto em situações difíceis até todo suporte material que recebi durante todos esses anos. Sem vocês dois essa monografia não seria possível.

Também cabe um agradecimento aos meus amigos, tanto aqueles que conheci na UFABC quanto aqueles que me acompanham desde o ensino médio. A companhia de vocês tornou os dias na universidade mais alegres com cada almoço no RU que tivemos.

Por fim, gostaria de agradecer aos corpo docente da UFABC que muito me ensinou enquanto aqui estive, o que se estende além dos professores de Ciência da Computação. Em especial, gostaria de agradecer minha orientadora Profa. Dra. Carla Negri Lintzmayer pelos anos de trabalho em conjunto que me fizeram compreender muito mais sobre nossa área de atuação.

“Que ninguém se engane,
só se consegue a simplicidade
através de muito trabalho.
(Lispector, Clarice)“

Resumo

A coloração de grafos é um problema que visa separar objetos (vértices) em diferentes grupos (cores) seguindo algum critério (arestas). Sua versão como problema de otimização, que é o foco desse trabalho, preocupa-se não apenas em encontrar uma coloração viável mas sim uma que contenha o menor número possível de cores. Trata-se de um problema computacional conhecidamente NP-difícil. Existem algoritmos de diferentes abordagens usados na construção e busca de boas soluções para esse problema. Nessa monografia, vamos apresentar alguns desses algoritmos e os conceitos por trás deles. Todos os algoritmos aqui apresentados foram implementados na linguagem Python e compõem um pacote da linguagem disponível *online* chamado Graphcol[1]. Ao final da construção do pacote também foram feitas algumas análises experimentais das implementações.

Palavras-chaves: Grafos, Python, Coloração, Heurísticas, Algoritmos.

[1] Disponível em https://pypi.org/project/graphcol/

Abstract

Graph coloring is a problem that aims to separate objects (vertices) into different groups (colors) respecting some criteria (edges). The optimization version of the problem, the main topic of this work, searches for a coloring that not only is feasible but also uses the smallest possible number of colors. Graph coloring is an NP-hard problem. To tackle this challenge, a multitude of strategies exist for constructing and exploring effective solutions. In this monography, we will present some of these algorithms and their main concepts. Every algorithm presented in this work was implemented as Python code and is available online as a single Python package named Graphcol[2]. At the end of the work, we also make some experimental analysis of the algorithms implemented.

Keywords: Graphs, Python, Coloring, Heuristics, Algorithms.

[2] Available at https://pypi.org/project/graphcol/

Sumário

Introdução

A área conhecida como otimização combinatória estuda problemas de otimização que trabalham com estruturas discretas e como encontrar as soluções ótimas para esses problemas. Normalmente temos uma quantidade gigantesca, mas finita, de soluções viáveis para os problemas de otimização combinatória, o que torna possível a criação algoritmos que os resolvam de forma exata (usando, por exemplo, força bruta). Apesar de funcionar bem para algumas classes específicas de entrada ou para entradas de tamanho limitado, na maioria dos casos esses algoritmos se tornam inviáveis conforme o tamanho da entrada cresce, sendo necessário o uso de outras técnicas para tratá-los.

Uma das estruturas combinatórias mais famosas usadas em problemas de otimização combinatória são os grafos. Grafos são estruturas que representam relações par-a-par entre objetos, sendo muito importantes na matemática e na computação. Por meio de grafos, é possível modelar e resolver inúmeros problemas e situações, como trajetos em mapas, conexidade em redes e relações de precedência.

Segundo Bondy e Murty [2], o grande desenvolvimento da Teoria dos Grafos se deu principalmente devido ao "Problema da Coloração de Mapas". Esse problema visa determinar qual o número mínimo de cores necessárias para colorir um mapa plano sem que duas áreas vizinhas tenham a mesma cor. Em 1852 foi conjecturado que o número máximo de cores necessárias para colorir qualquer mapa é quatro. Foi apenas em 1976 que Appel e Haken [1] provaram que essa conjectura vale, em um resultado que é conhecido como Teorema das Quatro Cores.

Problemas nos quais é necessário particionar um conjunto de elementos em vários grupos com determinadas características compatíveis entre os membros podem ser modelados e resolvidos por meio da coloração de vértices em grafos. Como exemplos, podemos citar problemas de escalonamento de tarefas, atribuição de frequências em redes, coloração de mapas e alocação de registradores. Muitos outros problemas ainda estão relacionados com coloração de vértices em grafos [2, 18].

O problema mais simples consiste em encontrar uma quantidade mínima de cores que possam ser atribuídas aos vértices de um grafo de tal forma que não existam vértices adjacentes com a mesma cor. Porém, encontrar tal valor mínimo é um problema NP-difícil [13]. Isso significa que, na prática, não temos esperança de desenvolver um algoritmo que resolva esse problema em tempo polinomial para qualquer grafo. Assim, as abordagens que temos acabam abrindo mão de fornecer essas características simultaneamente. Considerando o caráter difícil e as diversas aplicações relacionadas ao problema de coloração de grafos, existem vários trabalhos que tentam encontrar bons algoritmos para resolvê-lo, em

todas essas abordagens [18].

Entre os anos de 2019 e 2021 o aluno e a orientadora desenvolveram juntos dois projetos de iniciação científica (IC). O primeiro trabalho foi uma pesquisa bibliográfica sobre algoritmos de aproximação clássicos para diversos problemas de otimização, em sua maioria problemas que envolviam grafos. Já o segundo trabalho, também uma pesquisa bibliográfica, tinha como objetivo estudar algoritmos de diferentes abordagens para o problema de coloração de grafos. Ambos os trabalhos contaram com financiamento da Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP).

Para o PGC, procuramos envolver o projeto de algoritmos com uma boa diversidade de técnicas de projeto e que permitisse ao aluno entrar em contato com a parte experimental da área. Por isso a escolha foi por desenvolver um pacote Python com implementações de alguns dos algoritmos para coloração de grafos que foram estudados na IC. Assim, o projeto escolhido permitiu que o aluno entrasse em contato com conceitos de engenharia de software e os colocasse em prática enquanto desenvolvia o PGC. Alguns exemplos são Testes Automatizados, *Continuos Integration/Continuos Delivery*, *Gitflow*, entre outros.

É interessante destacar que muitas das ideias e dos conhecimentos técnicos aplicados no PGC tiveram de alguma forma influência do estágio que o aluno iniciou durante o curso de graduação. Isso mostra o valor e a aplicabilidade desse tipo de experiência no seu desenvolvimento enquanto cientista da computação, e como a relação com essa parte mais teórica acabou se dando de forma natural.

Após a implementação da biblioteca também foi trabalhada uma outra parte do PGC voltada à análise experimental dos algoritmos implementados. Para fazer isso foram realizados testes de execução deles sobre um *benchmark* com instâncias do problema de coloração de grafos. Com os dados obtidos foi feita uma análise exploratória.

No que segue, vamos apresentar a organização desse texto, que é baseada nas etapas de desenvolvimento do projeto, tentando assim reproduzir a mesma evolução de ideias e fatos que ocorreram durante a execução do projeto.

No Capítulo 1 apresentamos o problema central do projeto, isso é, o Problema de Coloração de Grafos, e apresentamos conceitos, notações e resultados importantes nas áreas de grafos, otimização e projeto de algoritmos que são fundamentais para melhor apresentar o problema e abordagens para tratá-lo.

No Capítulo 2 o assunto principal são os algoritmos que foram estudados, ou seja, as soluções estudadas para o problema abordado. Nessa parte vamos abordar bastante as diferentes técnicas de programação usadas no projeto de algoritmos para problemas de otimização, falando sobre algumas diferenças, conceitos e resultados que forem importantes.

Já no Capítulo 3 abordamos a parte mais técnica do projeto: a implementação da biblioteca e os experimentos realizados. Isso envolve apresentar as ferramentas, os conceitos

práticos, a implementação dos algoritmos, os testes realizados e a análise dos resultados obtidos. Sobre o experimento realizado apresentaremos a metodologia, um detalhamento dos grafos que foram usados no *benchmark*, entre outros detalhes técnicos relevantes sobre a execução e os resultados obtidos.

Por fim, fechamos o trabalho com as Considerações Finais, que vão apresentar uma revisão dos objetivos e avaliação dos mesmos: Se foram atingidos ou não, se a forma como foram trabalhados poderia ser melhor, entre outras questões. Também vamos falar de possíveis melhorias do projeto que já podem ser mapeadas.

1 Apresentando o Problema

Nesse capítulo pretendemos apresentar o embasamento teórico mais introdutório do projeto. Será com base nos conceitos apresentados nesse capítulo que as outras partes do trabalho vão se desenrolar. Afinal, as outras partes precisam do que será apresentado aqui para que o conjunto do trabalho faça sentido. Tudo que será apresentado nesse capítulo tem como referência livros didáticos, pois podem ser considerados resultados e conceitos clássicos do tema de coloração de grafos.

Vamos começar apresentando conceitos e noções de otimização combinatória na Seção 1.1. Depois vamos falar de grafos na Seção 1.2, essa será uma seção fundamental para o projeto, pois apresentaremos definições muito importantes para os algoritmos que usaremos e a notação que será usada para representar esses objetos que são o centro do problema de coloração. Encerramos o capítulo falando do problema de coloração de grafos em si na Seção 1.3, dando uma introdução, apresentando sua definição formal, resultados relevantes e problemas associados.

1.1 Otimização Combinatória

Nessa seção vamos apresentar alguns dos conceitos básicos da otimização combinatória com base nos livros "Algoritmos" [7] e "Introdução Sucinta aos Algoritmos de Aproximação" [6]. Como nesse trabalho não vamos nos aprofundar tanto na área de otimização combinatória, acreditamos que o mais importante é ter uma boa noção dos conceitos que serão apresentados nessa seção.

De maneira informal, um **problema** é uma tarefa ou uma questão que desejamos resolver, sendo que para resolvermos um computacionalmente é necessário descrever seus dados. Todo problema possui um conjunto de variáveis que define o universo de suas possíveis respostas, chamado de **entrada** do problema. Quando um problema é resolvido ele nos dá uma resposta para uma determinada configuração de suas variáveis. Chamamos uma configuração da entrada do problema de **instância** do problema. Cada problema quando resolvido devolve um conjunto de variáveis conhecido como **saída** do problema, sendo que cada configuração da saída do problema é conhecida como uma **solução** do problema. Podemos ainda definir uma série de limitações para os valores de saída do problema, que são chamadas de **restrições** do problema. Se o conjunto de valores devolvidos na solução respeita todas as restrições definidas para o problema dizemos que trata-se de uma **solução viável**, enquanto que uma solução que não respeita uma ou mais restrições é chamada de **solução inviável**.

Dizemos que um problema é de **decisão** se suas soluções possíveis são SIM ou NÃO. Já os problemas ditos de **otimização combinatória** buscam uma **solução ótima** dentro de um espaço de soluções discretas para uma instância. Podemos dizer que uma solução é ótima usando dois conceitos. O primeiro é o de **ótimo local**, que quer dizer que trata-se do melhor valor de solução para algum subespaço de soluções. O segundo é o de **ótimo global**, que refere-se a um valor ótimo não limitado apenas para um subespaço do problema mas sim para todo o espaço de soluções existente. Nesse trabalho sempre que usarmos a expressão solução ótima estaremos nos referindo ao ótimo global do problema, a não ser que seja especificado o contrário.

A otimalidade de uma solução é definida por uma função de maximização ou minimização, dependendo do problema. Aplicações práticas desse conceito podem ser encontradas em várias situações comuns (minimizar rotas de veículos, maximizar lucro, minimizar desperdício de material de produção, minimizar uso de recursos disponíveis, entre outros). Embora os problemas de otimização combinatória sejam muito diversos entre si, todos eles possuem alguns elementos em comum. O primeiro seria uma **instância** de entrada, isso é, um conjunto de dados para qual o problema será resolvido. O segundo é o **conjunto de todas as soluções** para aquele problema que sejam viáveis, ou seja, que respeitem as condições impostas para saída do problema. O terceiro é o que chamamos de **função objetivo**, que é a função de maximização ou minimização que citamos e avalia se uma solução é ótima ou não.

Para resolvermos um problema computacionalmente usamos **algoritmos**. Algoritmos podem ser definidos como um sequência de passos com regras bem definidas, que recebem uma entrada e produzem uma saída após essa sequência de passos ser realizada. Veja que podemos entender a entrada de um algoritmo como a instância de um problema, e sua saída pode ser entendida como uma possível solução para esse problema. Ao falarmos que um algoritmo está **correto**, queremos dizer que para qualquer instância possível o algoritmo devolve como saída uma solução para aquela instância.

Para resolver um problema computacional usamos um sistema computacional que possui recursos limitados. De maneira informal, dado um problema computacional e os algoritmos que o resolvem, a **complexidade computacional** desse problema refere-se à sua classificação segundo os recursos computacionais que seus algoritmos consomem. Um exemplo de recurso computacional é o espaço, onde a complexidade em relação a esse recurso é definida pela quantidade de memória que um algoritmo usa em sua execução. Da complexidade computacional também surge a ideia de **eficiência computacional**, sendo que um algoritmo é considerado mais eficiente que o outro em um determinado recurso se a quantidade usada de tal recurso for menor. Cada recurso computacional está associado a **classes de complexidade**, que são conjuntos de problemas que podem ser resolvidos usando uma quantidade bem definida de um determinado recurso computacional.

O recurso computacional que nos dá as classes de complexidade mais famosas é o **tempo**. Em suma, o tempo consumido por um algoritmo é uma função que determina quantos passos um algoritmo faz dependendo do tamanho de sua entrada. Dizemos que um algoritmo é **eficiente** se sua função de tempo no pior caso é um polinômio sobre o tamanho de sua entrada. As classes de complexidade mais famosas relacionadas ao consumo de tempo são:

- Classe $\mathcal{P}$, que contém todos os problemas para os quais existe um algoritmo eficiente;
- Classe $\mathcal{NP}$, que contém todos os problemas de decisão para os quais podemos verificar se uma solução é de fato uma solução usando um algoritmo eficiente.

Uma das maiores dúvidas em aberto na matemática é se $\mathcal{P} = \mathcal{NP}$. Veja que claramente $\mathcal{P}$ está contido em $\mathcal{NP}$, afinal se podemos encontrar a resposta de um problema em tempo polinomial, então também podemos verificar se uma solução qualquer é a nossa resposta. Saber se $\mathcal{P} = \mathcal{NP}$ é tão importante que esse é considerado um dos problemas do milênio[1].

É importante notar que para um problema pertencer a $\mathcal{NP}$ ele precisa ser de decisão. Problemas que possuam complexidade computacional equivalente à complexidade computacional dos problemas mais difíceis em $\mathcal{NP}$, independentemente de serem ou não problemas de decisão, pertencem a uma classe de problemas chamada $\mathcal{NP}$-difícil. Dizer que um problema pertence a $\mathcal{NP}$-difícil no fundo significa que não sabemos se existe ou não algoritmo eficiente para esse problema. Essa incerteza deriva de não sabermos se $\mathcal{P} = \mathcal{NP}$.

Aplicar força bruta em problemas de otimização combinatória é algo inviável na prática, mesmo para instâncias de tamanho moderado, devido à maioria dos problemas de otimização combinatória interessantes serem $\mathcal{NP}$-difíceis. Isso basicamente significa que não temos esperança em encontrar algoritmos eficientes que devolvam a solução ótima para esses problemas em tempo polinomial. Além disso, a menos que $\mathcal{P} = \mathcal{NP}$, não existem (i) algoritmos eficientes para resolver estes problemas de (ii) forma ótima (iii) para qualquer instância.

1.2 Grafos

Nessa seção apresentaremos algumas definições e resultados da área de Teoria dos Grafos. Grafos são objetos matemáticos muito usados para modelar problemas de otimização. A principal referência usada nessa seção foi o clássico livro sobre teoria dos grafos "*Graph Theory*" [2]. Também vale dizer que aqui a palavra grafo refere-se a grafos

[1] http://www.claymath.org/millennium-problems

simples, sem laços nem arestas paralelas. Por fim, ao longo de toda a seção vamos fazer referência a um grafo G que é um grafo genérico.

Então vamos começar por essa que é nossa definição mais básica. Um **grafo** é um par de conjuntos (V, E), onde V é um conjunto finito de elementos e E é um conjunto de pares não ordenados dos elementos de V. Dado um grafo $G = (V, E)$, chamamos os elementos de V de **vértices** do grafo e chamamos os elementos de E de **arestas** do grafo. Assim, V é o conjunto dos vértices de G e E é o conjunto das arestas de G. Ao longo desse texto, para um grafo $J = (A, B)$ qualquer vamos denotar por $V(J)$ seu conjunto de vértices e $E(J)$ seu conjunto de arestas. Dessa forma, podemos definir um grafo sem precisar explicitar os elementos do par. Além disso, dado um grafo G, denotaremos a aresta formada pelo par $u, v \in V(G)$ por uv.

Dados dois vértices $u, v \in V(G)$, sendo a aresta $uv \in E(G)$, dizemos que:

- u e v são **extremos** de uv;
- u e v são vértices **adjacentes** ou **vizinhos**;
- a aresta uv **incide** em u e em v.

Esses são termos importantes e amplamente usados na teoria dos grafos e ainda mais para nosso problema de coloração. Essas relações entre vértices adjacentes podem ser usadas para representar restrições em problemas de otimização.

Para representar graficamente grafos, o padrão é usar pontos e linhas. Os pontos representam os vértices e as linhas as arestas entre eles. Na Figura 1 temos um exemplo de representação gráfica de um grafo com 6 vértices (A, B, C, D, E, F) e 9 arestas (AB, AC, BC, CD, CE, CF, DE, DF, EF).

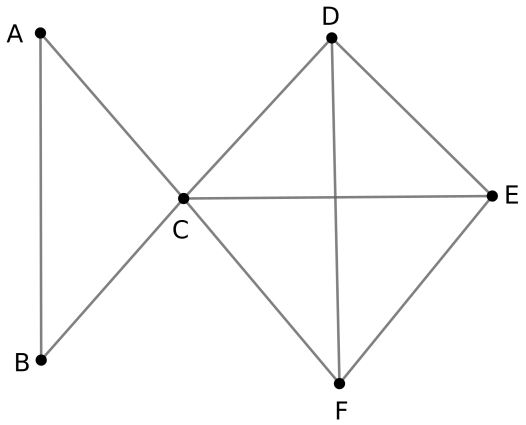

Figura 1 – Exemplo de representação gráfica de um grafo com 6 vértices e 9 arestas.

Outros conceitos importantes são o de grafo completo e subgrafo. Dado um grafo G com n vértices, se para todo $v \in V(G)$ temos que v é adjacente a todos os outros vértices de $V(G)$, então G é um **grafo completo** que pode ser representado por K_n. Agora, se temos os grafos G e H, dizemos que H é **subgrafo** de G se $V(H) \subseteq V(G)$ e $E(H) \subseteq E(G)$. Também denotamos essa relação por $H \subseteq G$. Note que todo grafo G com n vértices é subgrafo de K_n.

Dentro dessa relação de subgrafo temos tipos especiais de subgrafo que possuem características específicas em relação ao grafo original. Segue a definição de alguns dos principais tipos de subgrafo. Dados os grafos G e H, sendo H subgrafo de G, dizemos que:

- Se $H \neq G$, então H é **subgrafo próprio** de G. Podemos denotar essa relação por $H \subset G$;
- Se $V(H) = V(G)$, então H é **subgrafo gerador** de G;
- Se $E(H)$ é o conjunto de todas as arestas de $E(G)$ que têm ambos os extremos em $V(H)$, então H é **subgrafo induzido** por $V(H)$;
- Se $V(H)$ é o conjunto de todos os vértices de $V(G)$ que são extremos de alguma aresta de $E(H)$, então H é um **subgrafo induzido** por $E(H)$.

Dados um grafo G e $S \subseteq V(G)$, denotamos por $G[S]$ o **subgrafo induzido pelos vértices** de S de forma que $E(G[S]) = \{uv \in E(G) : u, v \in S\}$ e $V(G[S]) = S$. De forma equivalente, dado $T \subseteq E(G)$, denotamos por $G[T]$ o **subgrafo induzido pelas arestas** em T de forma que $V(G[T]) = \{v, x \colon vx \in T\}$ e $E(G[T]) = T$.

Grafos também têm algumas medidas e valores associados com significados importantes, que podem ser para o grafo todo ou somente para arestas e vértices. Considerando nosso grafo G, o valor $|V(G)|$ é a **ordem** de G, enquanto o valor $|E(G)|$ é o **tamanho** de G. Um outro valor importante na teoria dos grafos é o grau de um vértice. Dado $v \in V(G)$, o **grau** de v é a quantidade de vizinhos de v em G e é denotado por $d(v)$. O conjunto de todos os vértices vizinhos de um $v \in V(G)$ é representado por $N(v)$. Denotamos o valor do **grau máximo** em G por $\Delta(G)$ e denotamos o valor do **grau mínimo** em G por $\delta(G)$.

Um outro conjunto de definições importantes na área são referentes a sequências de vértices com características específicas. São eles os passeios, caminhos e ciclos. Para começarmos, dado um grafo G, chamamos de **passeio** em G uma sequência $W = (v_0, v_1, \ldots, v_k)$, onde para todo $0 \leq i \leq k$ vale que $v_i \in V(G)$ e para todo $0 \leq i \leq k-1$ vale que $v_i v_{i+1} \in E(G)$. Dado um passeio $W = (v_0, v_1, \ldots, v_k)$ em G, dizemos que W é **aberto** se $v_0 \neq v_k$ e é **fechado** se $v_0 = v_k$. O **comprimento do passeio** é igual ao total de arestas contidas no passeio. Um **caminho** é um passeio que não repete vértices. Por fim, um **ciclo** é um passeio fechado que não repete vértices internos.

Outro conceito importante da área é o de conexidade. Se para qualquer par $u, w \in V(G)$ podemos construir um passeio com início em u e término em w, então G é um grafo **conexo**. Uma propriedade, como conexidade, pode ser maximal ou não em um grafo. Dados os grafos G e H, sendo $H \subseteq G$, dizemos que H é **maximal** em relação a uma propriedade P se não existe $H' \subseteq G$ tal que $H \subset H'$ e H' possui a propriedade P. Então, por exemplo, se H for subgrafo de G e conexo, caso não haja nenhum outro subgrafo de G, digamos H', tal que H esteja contido em H', então H é maximal quanto a conexidade.

Dados os grafos G e H, sendo que H é subgrafo de G, se H é maximal em relação à conexidade, então H é uma **componente (conexa)** de G. Um grafo que não é conexo é composto por várias componentes conexas.

Dado um grafo F, se não existe nenhum ciclo em F, então F é uma **floresta**. Agora, dada uma floresta T, se T é um grafo conexo (ou, equivalentemente, contém apenas uma componente conexa), então T é uma **árvore**. Note que com essa definição é possível perceber que toda componente de uma floresta é uma árvore.

Se podemos particionar $V(G)$ em dois conjuntos X e Y onde cada aresta de $E(G)$ possui um extremo em X e outro extremo em Y, então dizemos que (X, Y) é **bipartição** de G, além de dizermos que G é um **grafo bipartido**. Denotamos que G é biparticionado em (X, Y) usando a notação $G[X, Y]$.

Para encerrar a seção, vamos apresentar três definições importantes para o problema de coloração e para os algoritmos que serão apresentados mais à frente no trabalho. Dado um grafo G, chamamos uma coleção de conjuntos $\mathcal{C} = \{C_1, C_2, \ldots, C_k\}$, onde cada $C_i \subseteq V(G)$, de **partição dos vértices** de G se cada $v \in V(G)$ está em exatamente um dos conjuntos C_i. Falamos que um subconjunto de vértices $U \subseteq V(G)$ é um **conjunto independente** em G, se para qualquer par de vértices $v, w \in U$ é verdade que $vw \notin E(G)$. Por fim, uma **clique** de um grafo G é um subconjunto $S \subseteq V(G)$ tal que $G[S]$ é grafo completo.

1.3 Coloração de Grafos

Nessa seção apresentamos o problema central do trabalho, **O Problema de Coloração de Vértices**. Não vamos somente defini-lo enquanto problema de otimização, mas também apresentaremos seu contexto histórico e alguns dos principais resultados. De maneira intuitiva, o problema de Coloração de Vértices (também chamado de coloração de grafos) visa rotular os vértices de um grafo com cores, de maneira que não haja **conflito** entre vértices vizinhos. Um conflito ocorre quando dois vértices vizinhos possuem a mesma cor. Dessa forma, cada aresta representa a possibilidade de um conflito.

Esse é um problema aplicável em diversas situações do mundo real onde desejamos modelar situações com conflitos, e para tal basta que representemos os objetos modelados

como vértices e os conflitos entre eles como arestas. Problemas de alocação são um bom exemplo de problema que pode ser modelado e resolvido por coloração de vértices. Por exemplo, imagine um problema de construção de calendário em que temos algum conjunto de atividades (palestras, aulas, jogos, etc) e um conjunto de horários em que essas atividades podem ocorrer. Desejamos alocar as atividades nesses horários mas somos limitados por um conjunto de restrições (pessoas, salas, etc). Veja que nesse caso podemos representar cada atividade como um vértice e cada restrição como uma aresta, sendo que nossa busca por marcar um horário compatível se resume a colorir o grafo que construímos, onde cada cor é um horário.

Vamos começar definindo precisamente o problema de coloração de vértices como um problema de otimização combinatória na Seção 1.3.1. Depois, vamos falar de alguns outros problemas de coloração semelhantes na Seção 1.3.2, em especial a coloração total e a coloração de arestas. Já na Seção 1.3.3 apresentamos resultados referentes ao número cromático de um grafo, tema de muito interesse para o problema de coloração. Por fim, fechamos com a Seção 1.3.4, onde apresentamos alguns conceitos sobre complexidade computacional e seu principal resultado na coloração de grafos.

1.3.1 Definindo a Coloração de Grafos

Uma partição $\mathcal{C}$ de $V(G)$ pode ser vista como uma **coloração dos vértices de** G, porque podemos observá-la como uma forma de colorir os vértices do grafo, ou seja, pensamos em cada conjunto da partição $\mathcal{C}$ como uma cor. Por isso, cada conjunto de $\mathcal{C}$ é também chamado de **classe de cor**. É comum usarmos números inteiros positivos para representarmos as cores da coloração.

Se para todo $vw \in E(G)$ vale que v e w fazem parte de diferentes conjuntos de $\mathcal{C}$, então chamamos $\mathcal{C}$ de **coloração própria dos vértices de** G. Em outras palavras, se $\mathcal{C}$ é uma partição de $V(G)$ em conjuntos independentes, então $\mathcal{C}$ é uma coloração própria. Normalmente, quando falamos de uma coloração qualquer estará implícito que estamos falando de uma coloração própria. Por isso, ao longo desse texto, a não ser que seja especificado o contrário, quando usarmos o termo coloração estaremos nos referindo a uma coloração própria. Se um grafo G aceita uma coloração que usa k cores, falamos que trata-se de uma k**-coloração**, além disso, falamos que G é um grafo k**-colorível**.

O problema de coloração de vértices passa a ser um problema de otimização combinatória quando desejamos não só encontrar uma coloração, mas sim uma coloração com o menor número possível de cores. Chamamos essa quantidade mínima de cores possível para uma coloração de um grafo G de **número cromático do grafo** G, denotado por $\chi(G)$. Chamamos uma coloração de um grafo G que use $\chi(G)$ cores de **coloração ótima de** G. Além disso, se para um grafo G temos que $\chi(G) = k$, falamos que G é um **grafo** k**-cromático**.

Segue a definição formal do problema de Coloração Mínima de Vértices, em que buscamos encontrar uma coloração ótima para um grafo.

Problema 1.1 (Coloração Mínima de Vértices (1))**.** *Dado um grafo G, no problema da Coloração Mínima de Vértices (CMV) desejamos associar a cada vértice $v \in V(G)$ um inteiro $c(v) \in \{1, \ldots, k\}$ de forma que:*

- *$c(v) \neq c(u)$, caso $vu \in E(G)$;*
- *k é mínimo.*

Cada valor inteiro em $\{1, \ldots, k\}$ é chamado de ***cor****.*

A definição para o CMV que acabamos de dar não é a única possível. Ela foi retirada do livro *"A Guide to Graph Colouring"* [17], que também dá outras definições, inclusive uma que usa o conceito de conjuntos independentes. Segue essa definição alternativa para o CMV.

Problema 1.2 (Coloração Mínima de Vértices (2))**.** *Dado um grafo G, desejamos encontrar uma coleção $\mathcal{S} = \{S_1, \ldots, S_k\}$, onde cada $S_i \in \mathcal{S}$ é um conjunto independente em G tal que*

$$\bigcup_{i=1}^{k} S_i = V(G) \tag{1.1}$$

$$S_i \cap S_j = \emptyset, \quad 1 \leq i, j \leq k \text{ e } i \neq j \tag{1.2}$$

$$\forall u, v \in S_i, uv \notin E(G), \quad 1 \leq i \leq k \tag{1.3}$$

sendo que k deve ser mínimo.

Cada conjunto independente S_i é chamado de ***classe de cor*** *enquanto cada i é chamado de* ***cor****.*

Observe que a definição alternativa define exatamente o mesmo problema, sendo a grande diferença a forma como uma solução é representada. A partir daqui, sempre que falarmos "problema da coloração de grafos" ou "problema da coloração de vértices", estaremos nos referindo ao CMV.

Como exemplo de coloração de um grafo temos a Figura 2. Os dois grafos apresentados na figura são iguais ao grafo da Figura 1. Veja que dois vértices vizinhos nunca possuem a mesma cor em ambos os exemplos. Mesmo sendo grafos iguais, as partições de cores de ambos são diferentes, como podemos ver. Além disso, as duas colorações apresentadas no nosso exemplo são ótimas. Então, para um mesmo grafo podemos ter mais de uma possibilidade de solução ótima.

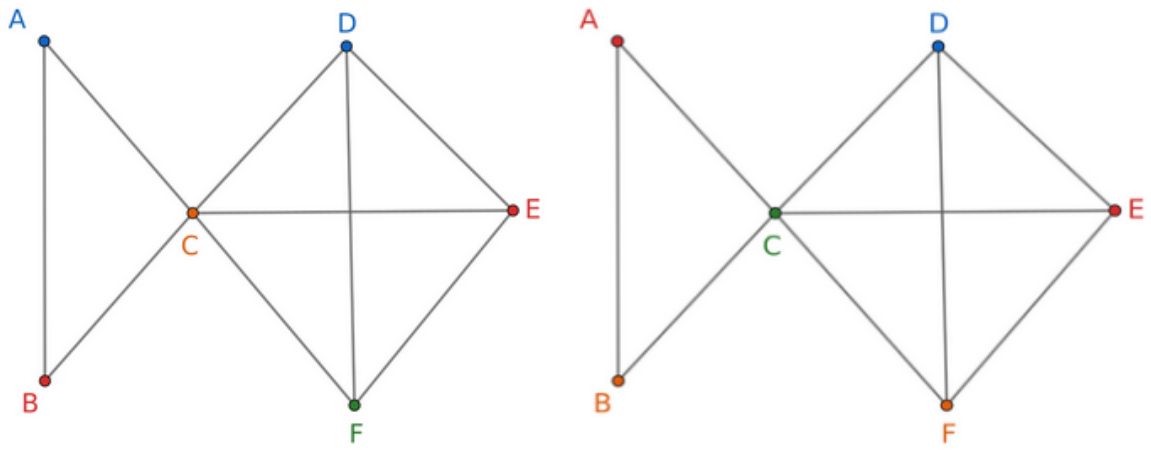

Figura 2 – Exemplos de coloração ótima para um mesmo grafo.

1.3.2 Outros Problemas de Coloração

Vamos apresentar alguns problemas de coloração associados ao problema da Coloração de Grafos. Mais especificamente, apresentaremos o Problema de Coloração de Arestas, o Problema de Coloração Total de grafos e alguns resultados associados. Nossa principal referência é a tese de Campos [5], que trata sobre o problema da Coloração Total de Vértices para classes específicas de grafos.

O problema da **Coloração de Arestas** nada mais é que uma rotulação das arestas de um grafo de forma que duas arestas que incidam em um mesmo vértice não possuam o mesmo rótulo ou a mesma cor. Segue a definição formal do problema da Coloração Mínima de Arestas.

Problema 1.3 (Coloração Mínima de Arestas)**.** *Dado um grafo G, desejamos associar a cada aresta $e \in E(G)$ um inteiro $c(e) \in \{1, \ldots, k\}$ de forma que:*

- *Para qualquer $vu, vw \in E(G)$ com $u \neq w$, vale que $c(vu) \neq c(vw)$;*
- *k é mínimo.*

Na coloração de vértices denotamos o número cromático de um grafo G por $\chi(G)$ e o definimos como a menor quantidade de cores possível que uma coloração de vértices pode assumir para G. Analogamente, para a coloração de arestas temos o **índice cromático do grafo**, representado por $\chi'(G)$, que é a menor quantidade de cores que qualquer coloração de arestas pode ter.

Podemos dizer que o principal resultado relacionado ao problema da Coloração Mínima de Arestas (a partir de agora chamaremos só de coloração de arestas) é o Teorema de Vizing [2], enunciado a seguir.

Teorema 1.1 (Teorema de Vizing [2])**.** *Dado um grafo G e seja $\Delta(G)$ o grau máximo de G, então vale que:*

- $\chi'(G) = \Delta(G)$, *ou*
- $\chi'(G) = \Delta(G) + 1$.

Baseando-se no Teorema de Vizing, os grafos são divididos em duas classes. A classe 1 é a classe dos grafos G em que $\chi'(G) = \Delta(G)$. Já a classe 2 é a classe de grafos G em que $\chi'(G) = \Delta(G) + 1$.

Da mesma forma que falamos de colorações só de arestas ou só vértices, também podemos falar de colorações que visem colorir ambos, arestas e vértices. Uma **Coloração Total** de um grafo G é uma rotulação de cores às arestas e aos vértices de G, de maneira que cada par de vértices adjacentes e cada par de arestas adjacentes tenham sempre cores distintas e, além disso, cada vértice deve ter cor distinta à das arestas que nele incidem. Formalmente, dados um grafo G e um valor $k \in \mathbb{N}$, chamamos de coloração total de G uma função $\Phi : V(G) \cup E(G) \to \{1, \ldots, k\}$ tal que:

- $\forall v \in V(G)$ e $\forall u \in N_G(v)$ vale que $\Phi(v) \neq \Phi(u)$;
- $\forall v \in V(G)$ e $\forall vx \in E(G)$ vale que $\Phi(v) \neq \Phi(vx)$;
- $\forall vx, vy \in E(G)$, onde $x \neq y$, vale que $\Phi(vx) \neq \Phi(vy)$.

Chamamos de **número cromático total** do grafo G o menor valor k para o qual G assume uma coloração total. Denotamos o número cromático total como $\chi_T(G)$. Qualquer coloração total de um grafo G precisa de ao menos $\Delta(G) + 1$ cores, pois o vértice de grau máximo tem $\Delta(G)$ arestas incidentes a ele e, além disso, também precisamos de mais uma cor para o próprio vértice. Esse fato estabelece um limitante inferior para $\chi_T(G)$.

Behzad e Vizing apresentaram a **Conjectura da Coloração Total** que afirma que para todo grafo G vale que $\chi_T(G) \leq \Delta(G) + 2$ [5]. Então, em conjunto com o que discutimos no último parágrafo, essa conjectura dá base para um resultado na coloração total semelhante ao Teorema de Vizing na coloração de arestas, que seria: dado grafo G, vale que $\chi_T(G) = \Delta(G)+1$ ou $\chi_T(G) = \Delta(G)+2$. É importante destacar que a Conjectura da Coloração Total já foi provada para diversas classes de grafos [5].

Para encerrar a seção, na Figura 3 temos exemplos de uma coloração de vértices, uma coloração de arestas e uma coloração total de um mesmo grafo.

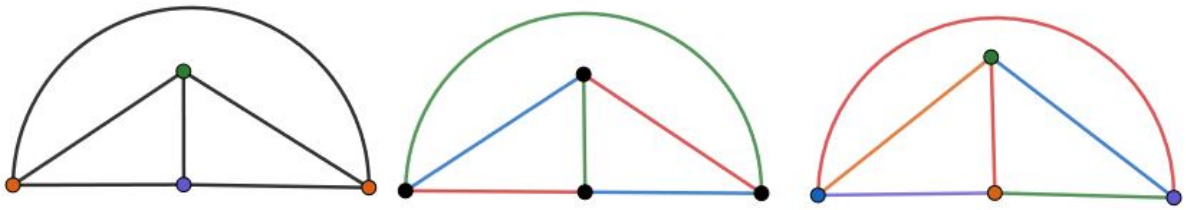

Figura 3 – Da esquerda para direita, mostramos uma coloração de vértices, uma coloração de arestas e uma coloração total de um grafo completo com 4 vértices.

1.3.3 Limitantes do Número Cromático

Conforme falamos na Seção 1.3.1, o número cromático de um grafo G ($\chi(G)$) representa a quantidade mínima de cores que uma coloração pode ter nesse grafo. Dito isso, fica claro como entender os limitantes do número cromático é algo que pode ser importante para projetar algoritmos para o problema de coloração de grafos, que não visa só encontrar o número cromático mas sim uma coloração. Nessa subseção vamos apresentar alguns dos resultados clássicos referentes a esse assunto. A principal referência dessa seção foi o livro *"Graph Theory"* de Bondy e Murty [2]. Esses resultados clássicos que citamos estão contidos na Proposição 1.1.

Proposição 1.1 ([2])**.** *Dado um grafo G, onde $n = |V(G)|$, vale que:*

- $\chi(G) = 1$ *se e somente se G é um grafo nulo;*
- $\chi(G) = 2$ *se e somente se G é um grafo bipartido;*
- *Se G é um ciclo de comprimento par, então $\chi(G) = 2$ (bipartido);*
- *Se G é um ciclo de comprimento ímpar, então $\chi(G) = 3$;*
- *Se G é um grafo completo, então $\chi(G) = n$;*
- $\chi(G) \geq \omega(G)$*, onde $\omega(G)$ é a cardinalidade da maior clique em G; e*
- $\chi(G) \geq \dfrac{n}{\alpha(G)}$*, onde $\alpha(G)$ é tamanho do maior conjunto independente em G.*

Os resultados acima são interessantes, mas podem ser considerados intuitivos. Em contrapartida, existem teoremas que estabelecem outros três limitantes para o valor de $\chi(G)$. O primeiro está associado à quantidade de arestas do grafo, o segundo está relacionado ao conceito de grafo k-crítico, e o terceiro é conhecido como Teorema de Brooks. Esses três teoremas são respectivamente apresentados aqui como Teorema 1.2, Teorema 1.3 e Teorema 1.4.

Teorema 1.2 ([2])**.** *Dado um grafo G, vale que $\chi(G) \leq \frac{1}{2} + \sqrt{2|E(G)| + \frac{1}{4}}$.*

Para o segundo limitante que iremos apresentar, vamos primeiro definir os conceitos de **grafo crítico** e **grafo k-crítico**, que são usados na demonstração do Teorema 1.3, que por sua vez implicará em dois corolários, sendo que é um desse corolários que realmente apresentará o limitante para $\chi(G)$. Dado um grafo G, dizemos que G é um **grafo crítico** se todo subgrafo H próprio de G respeita que $\chi(H) < \chi(G)$. Além disso, sendo $k = \chi(G)$, dizemos que G é um **grafo k-crítico** se ele é crítico e k-cromático.

Teorema 1.3 ([2])**.** *Se G é um grafo k-crítico, então $\delta(G) \geq k - 1$.*

Corolário 1.1 ([2])**.** *Todo grafo k-cromático possui ao menos k vértices com grau maior ou igual a $k - 1$.*

Corolário 1.2 ([2])**.** *Para todo grafo G vale que $\chi(G) \leq \Delta(G) + 1$.*

Por mais que o Corolário 1.2 nos dê um limitante superior para o valor de $\chi(G)$, esse limitante ainda pode ser muito distante do valor real de $\chi(G)$, sendo o exemplo mais fácil de visualizar isso um grafo bipartido completo. O Teorema de Brooks prova um limitante um pouco mais justo, porém, que ainda pode estar muito longe do valor real de $\chi(G)$.

Teorema 1.4 (Teorema de Brooks [2])**.** *Se G é um grafo conexo e não é um ciclo ímpar ou um grafo completo, então vale que $\chi(G) \leq \Delta(G)$.*

1.3.4 Complexidade do Problema

Conforme já dissemos, o problema de decisão da coloração de vértices é $\mathcal{NP}$-completo, enquanto o problema de otimização da coloração de vértices é $\mathcal{NP}$-difícil. Nessa seção, vamos discutir a $\mathcal{NP}$-completude do problema de decisão de coloração de vértices para 3 cores, conhecido como 3-COLORAÇÃO (Problema 1.4). Esse resultado implica na $\mathcal{NP}$-completude do problema de decisão da coloração de vértices para qualquer k, problema conhecido como k-COLORAÇÃO (Problema 1.5). As principais referências dessa seção são o artigo de Karp [13] e o livro "*Introduction to Algorithms*" [7]. Quaisquer termos usados nessa seção que não forem definidos aqui podem ter sua definição encontrada nesses trabalhos.

Problema 1.4 (3-COLORAÇÃO)**.** *Problema de decisão onde, dado um grafo G, devolvemos:*

- *SIM, caso exista 3-coloração própria para G;*
- *NÃO, caso não exista 3-coloração própria para G.*

Problema 1.5 (k-COLORAÇÃO)**.** *Problema de decisão onde, dado um grafo G e um número natural k, devolvemos:*

- *Sim, caso exista k-coloração própria para G;*
- *Não, caso não exista k-coloração própria para G.*

Para demonstrar que um problema é $\mathcal{NP}$-completo é necessário realizar uma redução de um outro problema de decisão que seja $\mathcal{NP}$-completo para ele. Então, para demonstrar que o 3-COLORAÇÃO é $\mathcal{NP}$-completo, precisamos tomar uma instância qualquer desse outro problema, adaptar ela para uma instância do 3-COLORAÇÃO e provar que a resposta do 3-COLORAÇÃO é SIM para essa instância adaptada se e somente se a resposta do problema para a instância original também for SIM. A ideia por trás disso é provar que a dificuldade em resolver ambos os problemas é igual.

Definição 1.1 (Redução)**.** *Sejam P_1 e P_2 dois problemas de decisão. Se existe uma forma de transformar uma instância qualquer α de P_1 para uma instância β de P_2, de forma que a resposta de P_2 para instância β é SIM se e somente se a resposta de P_1 para a instância α é SIM, então chamamos o processo de transformação de instância de* ***redução****.*

O primeiro problema provado $\mathcal{NP}$-completo foi o **Satisfatibilidade Booliana**, que aqui é apresentado no Problema 1.6, conhecido como SAT. Aqueles que descobriram esse resultado foram Stephen Cook e Leonid Levin, fazendo com que essa descoberta fosse conhecida como Teorema de Cook-Levin (Teorema 1.5).

Problema 1.6 (Satisfatibilidade Booliana)**.** *Problema de decisão onde, dada uma expressão booleana ϕ em sua forma normal conjuntiva, devolvemos:*

- *Sim, caso exista valoração dos literais de ϕ que faça seu valor ser verdade;*
- *Não, caso não exista valoração dos literais de ϕ que faça seu valor ser verdade.*

Caso a resposta seja Sim, dizemos que ϕ é ***satisfazível****, caso contrário dizemos que ϕ* ***não é satisfazível****.*

Teorema 1.5 (Teorema de Cook-Levin [7])**.** *O problema da Satisfatibilidade Booliana é $\mathcal{NP}$-completo. Além disso, qualquer outro problema $\mathcal{NP}$-completo pode ser reduzido em tempo polinomial para a Satisfatibilidade Booliana.*

Um outro problema $\mathcal{NP}$-completo, que é relacionado ao SAT, é o **3-Satisfatibilidade Booliana** (Problema 1.7), conhecido como 3-SAT. O 3-SAT é conhecidamente $\mathcal{NP}$-completo, sendo que esse resultado foi obtido por Karp [13] através de uma redução entre SAT e 3-SAT. A grande diferença entre o SAT e o 3-SAT está relacionada à quantidade de literais em cada cláusula da forma normal conjuntiva de uma expressão booleana que seja instância do problema.

Problema 1.7 (3-Satisfatibilidade Booliana)**.** *Problema de decisão onde, dada uma expressão booleana ϕ em sua forma normal conjuntiva onde cada cláusula possui exatamente 3 literais, devolvemos:*

- *SIM, caso exista valoração dos literais de ϕ que faça seu valor ser verdade;*
- *NÃO, caso não exista valoração dos literais de ϕ que faça seu valor ser verdade.*

Finalmente, para provar que o 3-COLORAÇÃO é $\mathcal{NP}$-completo podemos usar uma redução do 3-SAT para a 3-COLORAÇÃO, isso significa que vamos pegar uma instância do 3-SAT (ϕ) e transformá-la em uma instância do 3-COLORAÇÃO (G), e então provar que G é 3-colorável se e somente se ϕ é satisfazível.

Teorema 1.6 ([13])**.** *3-COLORAÇÃO é $\mathcal{NP}$-completo.*

Voltando a algo que foi dito no início dessa seção, é fácil perceber que o problema 3-COLORAÇÃO pode ser reduzido para a k-coloração, pois qualquer instância do primeiro é instância do segundo com $k = 3$, e a resposta devolvida pela k-coloração é sempre a mesma que a da 3-COLORAÇÃO. Dessa forma, k-coloração também é $\mathcal{NP}$-completo. O fato da k-coloração ser um problema $\mathcal{NP}$-completo implica diretamente que o problema da coloração mínima de vértices seja $\mathcal{NP}$-difícil.

Por tratar-se de um resultado muito importante e que foi estudado com certa profundidade para esse trabalho, como pode ser visto ao longo dessa seção onde apresentamos todos os conceitos e ferramental necessários (redução, SAT, 3-SAT), vamos apresentar a demonstração do Teorema 1.6 a seguir.

Demonstração. Considere uma instância ϕ do 3-SAT, onde $c_1, c_2, \ldots, c_m$ são as cláusulas definidas sobre as variáveis $\{x_1, x_2, \ldots, x_n\}$. Com base nessa instância do 3-SAT vamos construir um grafo G instância de 3-coloração. Para isso, a forma como vamos construir G vai considerar os seguintes aspectos:

1. A necessidade de estabelecer um paralelo entre os valores verdade de cada variável em ϕ com as cores de G.
2. A necessidade de capturar a satisfatibilidade de cada cláusula em ϕ.

Para que G considere os dois aspectos levantados vamos primeiro criar um triângulo em G com os vértices t, f, b, onde t significa verdadeiro (*true*), f significa falso (*false*) e b significa *base* (*base*). Cada um desses três vértices representa uma cor que vai ser usada para colorir G, afinal, como esse é um triângulo já sabemos que cada um terá uma cor diferente.

O próximo passo é adicionar dois vértices v_i e v'_i para cada variável x_i, sendo que cada par desses vértices formará um triângulo com o vértice base e representará os literais da variável. Portanto, impomos ao grafo G que se v_i tem a mesma cor que t então v'_i tem mesma cor que f, e vice versa. Essa construção de G permite um paralelo preciso entre os valores verdade de cada literal x_i em ϕ e a cor dos vértices v_i e v'_i em G. Dessa forma, o primeiro aspecto listado já está sendo considerado nessa construção de instância. Veja a construção de G até o momento na Figura 4.

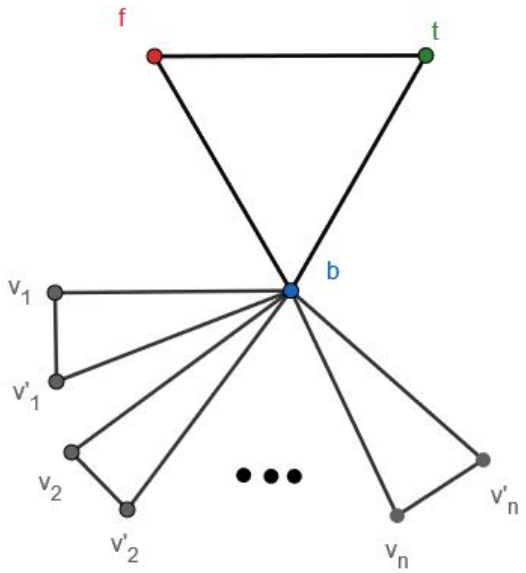

Figura 4 – Construção parcial da instância G criada a partir de ϕ.

Agora, vamos adicionar restrições a G de forma que o segundo aspecto citado também seja considerado. Para isso vamos usar o dispositivo de satisfatibilidade da cláusula, sendo que esse dispositivo nada mais é do que uma forma de expressar a operação lógica OR entre os três literais de cada cláusula em ϕ no grafo G. Para obter o resultados dessas operações usamos as cores já comentadas.

Para cada cláusula $c_j = (a \vee b \vee c)$, definimos o dispositivo de satisfatibilidade dessa cláusula como um grafo capaz de representar a operação OR entre os vértices que representam tais literais. Na Figura 5 temos um exemplo de construção do dispositivo descrito.

Perceba que essa construção baseada no dispositivo de satisfatibilidade tenta repetir duas vezes a mesma operação, primeiramente fazendo $(a \vee b)$ e depois $((a \vee b) \vee c)$.

É possível perceber que o dispositivo possui as seguintes propriedades:

- Se a, b, c possuem todos cor f, então $((a \vee b) \vee c)$ também tem que ser colorido com f. Dessa forma, o dispositivo nos indica quando alguma cláusula não é satisfazível.

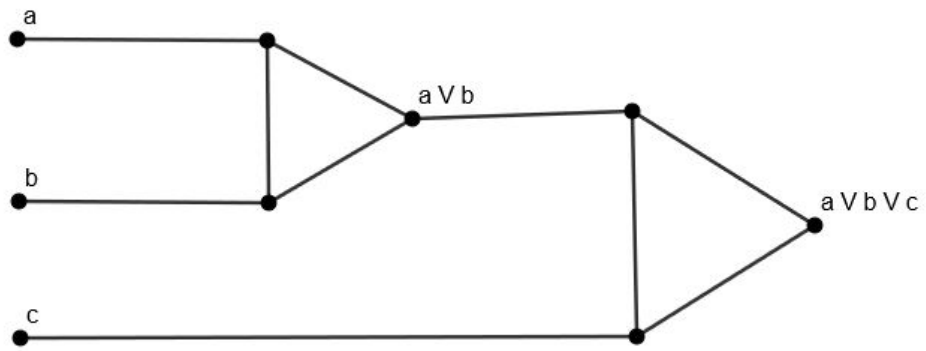

Figura 5 – Dispositivo de satisfatibilidade da cláusula $(a \vee b \vee c)$.

- Se algum dos vértices a, b, c possui cor t, então existe 3-coloração própria para o dispositivo, o que indica que a cláusula considerada possui valoração que a satisfaça.

Perceba então que os dispositivos capturam muito bem o segundo aspecto citado, porém, veja que eles indicam o resultado da operação OR entre os literais considerados. Para considerar a satisfatibilidade expressão booliana ϕ é necessário adicionar mais uma estrutura à cada dispositivo, como podemos ver na Figura 6 . Com a adição dessa nova estrutura, qualquer outra cor atribuída à operação final do dispositivo que não t indicará que não existe 3-coloração válida para o grafo e portanto ϕ não é satisfazível.

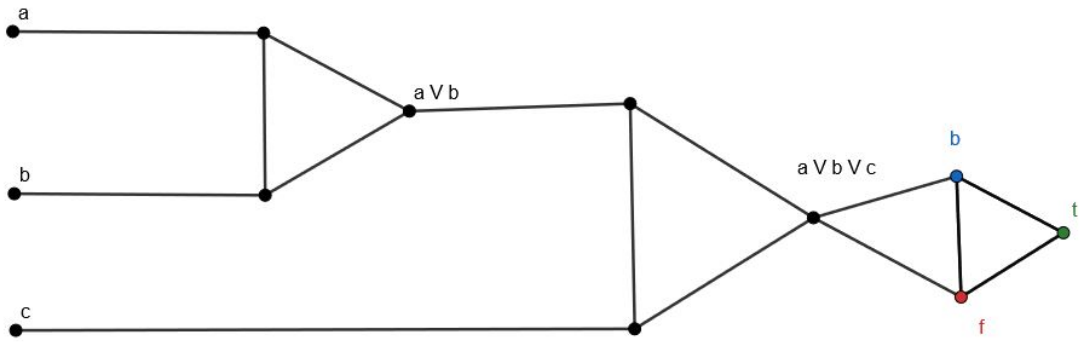

Figura 6 – Dispositivo de satisfatibilidade da cláusula em sua versão final.

Finalizamos nossa adaptação entre instâncias de forma que agora podemos demonstrar que ϕ é satisfazível se e somente se G é 3-colorável.

Vamos começar com a ida. Se ϕ é satisfazível então G é 3-colorável. Suponha que $\{y_1, y_2, \ldots, y_n\}$ seja uma valoração das variáveis $\{x_1, x_2, \ldots, x_n\}$ que satisfaça ϕ. Se $y_i = V$, então v_i é colorido com t e v_i' com f, sendo que caso $y_i = F$ o que acontece é o contrário. Como ϕ é satisfazível, toda cláusula c_j também deve ser satisfazível, isso nos diz que a, b ou c é colorido com t. Pela construção de G, em especial do dispositivo de satisfatibilidade, sabemos que o dispositivo correspondente a cada c_j é 3-colorível e o vértice de saída do dispositivo é colorido com t, o que nos garante que o grafo G possui 3-coloração própria.

Agora a volta. Suponha que G seja 3-colorível, vamos construir uma valoração das variáveis $\{x_1, x_2, \ldots, x_n\}$ da seguinte forma: se v_i é colorido com t, então x_i é valorada com V, caso seja colorido com f então x_i é valorada F. Por contradição, assuma que ϕ não seja satisfazível com a valoração construída, isso significa que ao menos uma cláusula não foi satisfeita com tal valoração. Então, pela construção de G, o vértice de saída dos dispositivos que representam as cláusulas não satisfeitas devem assumir valor f, porém, também pela construção de G, isso cria um conflito na coloração que faz com que G não seja 3-colorável, contradição com o que já assumimos. □

2 Soluções do Problema

No Capítulo 1 definimos o problema de coloração de grafos e apresentamos alguns outros conceitos e resultados importantes sobre o mesmo, o que serviu como uma introdução teórica ao tema. Agora, apresentamos nessa seção os algoritmos para o problema que foram implementados durante o PGC, isso é, os geradores de soluções para o problema, por isso o nome do capítulo. É importante destacar que ainda não vamos apresentar detalhes técnicos, apenas a definição conceitual desses algoritmos e pseudocódigos. Além de falar dos algoritmos em si também vamos apresentar as abordagens de projeto de algoritmos que cada um segue e o que diferencia cada uma delas.

Como explicado na Seção 1.1, para problemas em $\mathcal{NP}$-difícil, como a coloração de grafos, não temos esperança de desenvolver um algoritmo que seja computacionalmente eficiente e devolva uma solução ótima para qualquer instância. Para esse tipo de problema existem três principais abordagens de desenvolvimento de algoritmos:

- **Abordagem exata:** Nessa abordagem sempre garantimos que a solução devolvida pelo nosso algoritmo é ótima, porém, para problemas $\mathcal{NP}$-difícil esse tipo de algoritmo é ineficiente para instâncias que sejam muito grandes;
- **Abordagem heurística:** Um algoritmo é considerado heurístico se ele possuir um tempo de execução polinomial para toda instância do problema mas não possuir garantia em relação a sua qualidade de sua solução;
- **Abordagem por aproximação:** Falamos que um determinado algoritmo é de aproximação se para qualquer instância o tempo de execução é eficiente e existe certa garantia sobre a qualidade da solução, mas nem sempre a solução devolvida é ótima. Especificamente, um algoritmo é uma α-aproximação se para qualquer instância recebida ele devolva uma solução cujo valor está a um fator de no máximo α do valor da solução ótima.

No PGC implementamos sete algoritmos heurísticos, sendo quatro deles meta-heurísticos e três heurísticas gulosas. Além disso também implementamos dois algoritmos exatos. Começamos falando das heurísticas gulosas na Seção 2.1, depois passamos para as meta-heurísticas na Seção 2.2 e encerramos com os algoritmos exatos na Seção 2.3. Seguem os nomes e as classificações dos algoritmos implementados:

- Algoritmos Gulosos:
    1. Algoritmo Guloso Simples;
    2. Algoritmo *Degree of Saturation* (DSatur);
    3. Algoritmo *Recursive Largest First* (RLF);
- Algoritmos Meta-Heurísticos:
    1. Algoritmo Coloração Tabu (TabuCol);
    2. Algoritmo *Hill-Climbing*;
    3. Algoritmo Evolucionário;
    4. Algoritmo Colônia de Formigas (AntCol);
- Algoritmos Exatos:
    1. Algoritmo de Lawler;
    2. Algoritmo DSatur Exato.

É relevante dizer que as referências que permitiram o contato inicial com esses algoritmos foram o livro "*A Guide to Graph Colouring*" [17], com enfoque maior nas heurísticas, e a dissertação de Lima [19] sobre algoritmos exatos para coloração de grafos. Embora esses sejam materiais que sejam resultados de pesquisa bibliográfica, as fontes originais também foram consultadas e serão citadas ao longo do capítulo.

2.1 Algoritmos Gulosos

Quando falamos que um algoritmo é guloso queremos dizer que ele segue o que conhecemos como estratégia gulosa (*greedy strategy*). Algoritmos para problemas de otimização costumam ter etapas onde alguma decisão é tomada. Os algoritmos gulosos tomam a decisão que é considerada ótima naquele momento do algoritmo, sem levar em consideração as próximas etapas e a execução completa do algoritmo. Embora algoritmos gulosos nem sempre produzam soluções ótimas, eles acabam as produzindo em muitas situações [7].

Para trabalhar com problemas que sejam $\mathcal{NP}$-difíceis uma alternativa viável é o uso de heurísticas, ou seja, algoritmos com tempo de execução polinomial mas sem garantias sobre a qualidade da solução. Algoritmos eficientes gulosos sem garantia quanto ao resultado final são também algoritmos heurísticos. Nessa seção apresentaremos três algoritmos heurísticos gulosos. O primeiro contato com os três algoritmos foi o livro "*A Guide to Graph Colouring*" [17], que classifica os três algoritmos como construtivos, e ao longo das subseções vamos citar os trabalhos originais associados a esses algoritmos e que foram citados pelo autor do livro. Ao longo da seção vamos ver que, por mais que não existam garantias sobre a qualidade da solução devolvida por esse algoritmos para todos os grafos, existem classes de grafos nas quais podemos usá-los para encontrar a solução ótima.

2.1.1 Algoritmo Guloso

Vamos começar com aquela que provavelmente é a heurística mais simples e fundamental para o problema da coloração de vértices. Considerando que cada cor está associada a um número natural, o algoritmo funciona da seguinte forma: dada uma ordem pré-definida dos vértices do grafo de entrada, o algoritmo passa por cada vértice seguindo essa ordem, e então ele atribui a primeira cor disponível para esse vértice, isto é, atribui a ele a cor com valor mais baixo sem criar conflitos com a definição de coloração própria. Então, uma instância do algoritmo Guloso consiste no próprio grafo G e uma ordem π dos vértices de G. O Algoritmo 1 apresenta um pseudocódigo do algoritmo Guloso. Veja que como entrada temos um grafo G e opcionalmente uma ordem π, sendo que se ela não for passada o próprio algoritmo se encarrega de construí-la. Também temos $\mathcal{S}$, que é o conjunto de todas as cores disponíveis até um dado momento no algoritmo, que é inicializado com apenas uma possibilidade e dependendo dos conflitos aumenta.

O algoritmo Guloso produz soluções viáveis em tempo polinomial, e dependendo da ordem em que os vértices são passados ele pode produzir uma solução ótima ou uma solução muito ruim. Tome como exemplo um grafo bipartido. Se a ordem de vértices dada ao algoritmo tem todos os vértices de uma das duas partições em sequência, então

Algoritmo 1: GULOSO(G, π)

Entrada: Grafo G e opcionalmente uma ordem dos vértices π
Saída: Partição dos vértices de G em uma coloração viável

```
1  S_1 ← ∅
2  𝒮 ← {S_1}
3  se π não está definido então
4  |  Defina π usando algum método ou de maneira aleatória
5  Seja π ← (v_1, ..., v_n) em que n = |V(G)|
6  para i ← 1 até n faça
7  |  j ← 1
8  |  encontrou ← Falso
9  |  enquanto encontrou = Falso e j ≤ |𝒮| faça
10 |  |  se S_j ∪ {v_i} é conjunto independente então
11 |  |  |  S_j ← S_j ∪ {v_i}
12 |  |  |  encontrou ← Verdade
13 |  |  j ← j + 1
14 |  se encontrou = Falso e j > |𝒮| então
15 |  |  S_j ← {v_i}
16 |  |  𝒮 ← 𝒮 ∪ {S_j}
17 devolve 𝒮
```

a quantidade de cores usadas será muito maior do que o real valor de $\chi(G)$ que é 2. Isso só ressalta que ordem de vértices é muito importante. Existem algoritmos para tentar encontrar ordens de vértices que levariam a uma solução ótima, ou ao menos boa considerando certos critérios [17]. Dada a simplicidade e rapidez do algoritmo, uma estratégia comum ao usá-lo envolve executá-lo repetidas vezes, porém, com a ordem dos vértices da segunda execução em diante dependendo da coloração atribuída na execução anterior. A estratégia consiste em, na ordem da próxima execução, colocar os vértices que foram coloridos com a mesma cor em sequência. Inclusive, existem teoremas que dão certas garantias sobre essa estratégia, como o Teorema 2.1 e o Teorema 2.2.

Teorema 2.1 ([17])**.** *Seja $\mathcal{S}$ uma coloração viável do grafo G. Se todos os vértices de cada classe $S_i \in \mathcal{S}$ encontram-se em sequência na ordenação π, então quando a instância (G, π) é dada ao algoritmo* GULOSO*, a solução devolvida $\mathcal{S}'$ é viável e $|\mathcal{S}'| \leq |\mathcal{S}|$.*

Teorema 2.2 ([17])**.** *Seja G um grafo onde $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ é uma coloração de vértices ótima. Então existem, no mínimo, $k! \prod_{i=1}^{k} |S_i|!$ permutações de vértices que, quando passadas ao algoritmo* GULOSO*, levam o algoritmo a devolver uma solução ótima.*

Vamos agora fazer uma análise do tempo de execução do algoritmo. Seja $n = |V(G)|$ e $m = |E(G)|$. Consideramos aqui que a geração dos n elementos de π, caso o mesmo não esteja definido, consome $O(n)$. Veja que a parte principal do GULOSO é o laço **para** que começa na linha 6, que itera exatamente n vezes e, para cada uma dessas vezes, temos o outro laço **para** que inicia na linha 9 e itera pela quantidade de cores

já usadas até o momento de colorir o i-ésimo vértice, o que, no pior dos casos, itera n vezes também, por ser o máximo número de cores possível. Na linha 10 temos a avaliação se um conjunto de vértices é um conjunto independente. Para verificar se um conjunto S_j é independente com a adição de um v_i avaliamos todas as arestas de v_i. Fazer essa verificação para cada vértice significa visitar cada aresta do grafo duas vezes, uma vez para cada extremo da aresta independente das iterações dos laços externos. Assim, a quantidade de comparações para todo o algoritmo é $2m$. Já as outras operações dentro do laço **para** são consideradas constantes. Dessa forma, o tempo de execução do algoritmo é $O(n^2) + O(n) + O(m) = O(n^2 + m)$.

O pseudocódigo apresentado pode deixar a ideia do algoritmo clara, porém, existem implementações mais eficientes. Um exemplo de implementação mais eficiente pode considerar iterar por todos os vértices da sequência passada e, ao invés de iterar por todas as cores, avaliar os vizinhos do vértice sendo colorido e desconsiderar as cores usadas por eles. Essa troca na forma de avaliação das cores que podem ser usadas pode levar o algoritmo a ter tempo de execução $O(n + m)$.

2.1.2 Algoritmo DSatur

DSatur é uma abreviação para *degree of saturation*, que é traduzido como grau de saturação. O algoritmo DSatur foi uma heurística proposta por Brélaz [4]. No artigo original o autor destaca esse método de encontrar colorações como mais eficiente do que os outros algoritmos disponíveis na época. Além disso, no mesmo artigo o autor também mostra que um algoritmo exato para coloração de grafos também baseado na heurística do grau de saturação. Esse algoritmo exato será apresentado na Subseção 2.3.2.

O Dsatur tem funcionamento semelhante ao do algoritmo Guloso, sendo que a grande diferença entre esses dois reside na forma em que a ordem dos vértices a serem coloridos é determinada. Diferente do algoritmo Guloso que já recebe na entrada tal ordem, o algoritmo Dsatur usa o conceito de grau de saturação do vértice para escolher qual vai ser o próximo vértice a ser colorido na execução do algoritmo. Assim, uma instância do algoritmo Dsatur consiste simplesmente em um grafo G.

Definição 2.1 (Grau de Saturação)**.** *Seja G um grafo que está tendo uma coloração construída por um algoritmo. Dado $v \in V(G)$, dizemos que o* ***grau de saturação*** *de v naquele momento é igual à quantidade de cores diferentes atribuídas aos vizinhos de v. Denotamos o grau de saturação de v por $sat(v)$.*

Então, o Dsatur também é um algoritmo heurístico guloso para o problema da coloração. A ideia gulosa por trás dele é dar preferência aos vértices que já estão mais restritos naquele momento, sendo que o critério para avaliar isso é o grau de saturação. Como o procedimento de coloração do Dsatur é consideravelmente menos aleatório que

do Guloso, em algumas classes de grafos o Dsatur devolve soluções exatas. Segue um teorema que apresenta que o resultado devolvido pelo Dsatur para certas classes é exato. Perceba que na hora de escolher o vértice v com maior grau de saturação podemos ter empates e escolher como resolver empates é um ponto de variação na implementação do algoritmo. Vamos considerar aqui que quando empates ocorrem um vértice é escolhido aleatoriamente.

Teorema 2.3 ([17]). *Seja G um grafo bipartido, um ciclo ou uma roda. O algoritmo Dsatur devolve uma solução exata para G.*

Algoritmo 2: DSatur(G)

Entrada: Grafo G
Saída: Partição dos vértices de G em uma coloração viável

```
1  S ← {S_1}
2  U ← V(G)
3  enquanto U ≠ ∅ faça
4  |  Seja v o vértice de U com maior grau de saturação ou aquele selecionado pelos
   |    critérios de desempate
5  |  j ← 1
6  |  encontrou ← Falso
7  |  enquanto encontrou = Falso e j ≤ |S| faça
8  |  |  se S_j ∪ {v} é conjunto independente então
9  |  |  |  S_j ← S_j ∪ {v}
10 |  |  |  encontrou ← Verdade
11 |  |  j ← j + 1
12 |  se encontrou = Falso e j > |S| então
13 |  |  S_j ← {v_i}
14 |  |  S ← S ∪ {S_j}
15 |  U ← U \ {v}
16 devolve S
```

Apresentamos o pseudocódigo do DSatur no Algoritmo 2. Veja que iniciamos com um conjunto U, que contém todos os vértices ainda não coloridos (*Uncolored*), e com uma coleção $\mathcal{S}$, que representa todas as cores criadas até um certo momento. Basicamente, o controle de iterações não é baseado em passar por uma lista de vértices com ordem pré-definida, mas sim no controle de U estar ou não vazio. O próximo vértice a ser colorido é sempre o v com maior $sat(v)$.

A análise de tempo de execução do DSatur é muito semelhante à do Guloso feita na Seção 2.1.1. Seja $n = |V(G)|$ e $m = |E(G)|$. Veja que na linha 3 temos um laço **enquanto** que continuará executando até que U esteja vazio, isso sendo garantindo pelo esvaziamento do mesmo a cada iteração na linha 15. Assim já sabemos que tudo dentro desse laço será repetido n vezes. Calcular e manter a lista de grau de saturação demanda atualizar o grau de saturação dos vértices impactados a cada vértice colorido. Como no

início do algoritmo esse valor é zero para todos os vértices, iniciar a lista de grau de saturação é $O(n)$. Atualizar a lista de grau de saturação envolve verificar a cada vértice v colorido todos os seus vizinhos e atualizar o valor de grau de saturação deles. Somando para todos os vértices, temos uma operação de atualização executada ao todo duas vezes por aresta ou seja, a etapa de atualização executa em $O(m)$. A seleção do vértice com maior grau de saturação envolve buscar na lista de grau de saturação o vértice com maior valor, isso é uma busca simples em $O(n)$ que repetida a cada iteração resulta em uma complexidade de $O(n^2)$. Já o laço **enquanto** iniciado na linha 9 executa no máximo n vezes, o número máximo de cores possível. Verificar se $\mathcal{S}_j \cup \{v\}$ é conjunto independente, assim como no algoritmo Guloso, é uma operação que não precisa avaliar todas as arestas a cada iteração mas sim só aquelas arestas do vértice v. No total cada aresta é avaliada duas vezes nesse processo, então temos $O(m)$. Assim, ao todo o tempo de execução do DSatur é dado por $O(n^2) + O(m) + O(m) = O(n^2 + m)$.

Da mesma forma que para o Algoritmo Guloso, é importante ressaltar que embora o pseudocódigo apresentado para o DSatur facilite a compreensão da ideia existem formas computacionalmente mais eficientes de implementar ele. A alteração mais importante que podemos citar para o DSatur é o o uso da estrutura de dados *heap* para armazenar os graus de saturação de cada vértice, onde a prioridade é justamente definida pelo grau de saturação. A cada vértice colorido temos que atualizar o grau de saturação de seus vizinhos. Veja que a *heap* vai começar com um tamanho n, um elemento para cada vértice, e a cada vértice colorido vamos remover um elemento. Primeiro inserimos todos os vértices na *heap* com grau de saturação 0, o que possui complexidade $O(n)$. Quando vamos colori-los temos que remover o elemento correspondente na *heap*, o que leva $O(\log(n))$. Assim, a etapa de colorir os vértices leva tempo $O(n \log(n) + n) = O(n \log(n))$. Após colorir um vértice é necessário atualizar o grau de seus vizinhos. Percorrer os vizinhos e atualizar seus graus possui tempo de execução $O(m)$. Assim, o tempo de execução dessa implementação alternativa é $O(m + n \log(n))$.

2.1.3 Algoritmo RLF

O algoritmo *Recursive Largest First* (RLF) foi apresentado por Leighton (1979) em um artigo que tinha como objetivo apresentar uma nova heurística para problemas de escalonamento, uma possível aplicação da coloração de grafos. O artigo foca em mostrar a eficiência do algoritmo em encontrar boas soluções quando aplicado sobre algumas instâncias grandes do problema (vinte e sete grafos de cento e cinquenta vértices e doze grafos de quatrocentos e cinquenta vértices) com densidades de aresta variadas. O RLF devolve soluções ótimas para as mesmas classes de grafo citadas para o DSatur, como podemos ver no teorema enunciado a seguir.

Teorema 2.4 ([17])**.** *Seja G um grafo bipartido, um ciclo ou uma roda. O algoritmo RLF*

devolve uma solução exata para G.

Da mesma forma que os outros dois algoritmos apresentados anteriormente, o RLF é uma heurística gulosa. Porém, diferentemente dos outros, sua estratégia envolve construir uma cor por vez ao invés de colorir os vértices segundo um critério relacionado a uma ordem de vértices. Para fazer essa construção de uma cor por vez, considerando que uma coloração é uma coleção de conjuntos independentes, o algoritmo RLF constrói um conjunto independente e considera ele uma cor. Então, os vértices pertencentes a esse conjunto são desconsiderados na construção da próxima cor.

A heurística para a seleção do próximo vértice a ser adicionado ao conjunto independente (cor) considerado no momento não é única, mas o recomendado é seguir uma política semelhante ao DSATUR, isso é, priorizar a escolha de vértices que estejam mais restringidos. Então, por exemplo, poderíamos aplicar o conceito de grau de saturação na hora construir os conjuntos independentes e escolher o próximo vértice a ser adicionado na cor com base nesse critério. Nesse caso, seria necessária uma adaptação para também usar os vértices desconsiderados durante o processo do RLF na construção das cores.

Algoritmo 3: RLF(G)

Entrada: Grafo G
Saída: Partição dos vértices de G em uma coloração viável

```
1  S ← ∅
2  U ← V(G)
3  F ← ∅
4  i ← 0
5  enquanto U ≠ ∅ faça
6  |  i ← i + 1
7  |  S_i ← ∅
8  |  enquanto U ≠ ∅ faça
9  |  |  Escolha v pertencente a U
10 |  |  U ← U \ {v}
11 |  |  S_i ← S_i ∪ {v}
12 |  |  F ← F ∪ N(v)
13 |  |  U ← U \ F
14 |  S ← S ∪ {S_i}
15 |  U ← F
16 |  F ← ∅
17 devolve S
```

No Algoritmo 3 temos o pseudocódigo do RLF. Veja que, como no DSATUR, controlamos a iteração usando uma lista U de vértices ainda não coloridos (*uncolored*). Depois de criada uma cor S_i, escolhemos o primeiro vértice v para adicionar a ela, cuja escolha pode seguir um critério qualquer, mas vamos considerar um grau de saturação do vértice adaptado para o RLF. A diferença é que quando atualizamos o grau de saturação

dos vértices, consideramos os vértices que não podem ser adicionados à cor S_i como coloridos, assim eles também contabilizam para o grau de saturação. Uma vez escolhido esse vértice, criamos uma lista auxiliar F (*forbidden*) para listar os vértices não viáveis de adicionarmos à cor S_i, isso é, a vizinhança dos vértices já adicionados em S_i. Esse processo é repetido até que não tenhamos mais vértices a serem adicionados a S_i que mantenham a viabilidade da cor, momento em que F é vazio. Ao final da construção da cor, seus vértices são removidos de U e passamos para a construção da próxima cor. Lembrando que $N(v)$ indica a vizinhança do vértice v.

Sobre o tempo de execução do RLF, seja $n = |V(G)|$ e $m = |E(G)|$. Vemos que temos dois laços **enquanto** aninhados. No pior caso ambos executam em $O(n)$ por serem dependentes da quantidade de vértices. Por fim, temos a escolha do vértice v dentro dos dois laços aninhados. Como já dito, estamos considerando que essa escolha é feita por um grau de saturação adaptado que considera qualquer vértice que não esteja em U como colorido independente de estar colorido de fato ou de estar em F no momento de avaliação. A busca dos valores de grau de saturação é um processo que para cada vértice escolhido tem tempo de execução $O(n)$, já a atualização dos vértices pode ser feita em $O(m)$, como avaliado para o algoritmo DSatur. Por fim, consideramos que os processos de movimentar os vértices entre conjuntos pode ser feito em $O(1)$, isso se representarmos os conjuntos de vértices por vetores indexados pelos vértices. Então, podemos dizer que o tempo de execução do RLF é $O(n) \times O(n) \times (O(n) + O(1)) + O(m) = O(n^3 + m)$.

2.2 Algoritmos Meta-Heurísticos

Nessa seção também falaremos sobre alguns algoritmos heurísticos para o problema da coloração de grafos, porém, diferentemente dos algoritmos heurísticos apresentados na seção anterior, aqui focaremos em apresentar meta-heurísticas e em como elas são usadas para encontrar soluções do problema de coloração de grafos. As principais referências usadas nessa seção são os livros de Lewis [17], que apresenta ideias gerais de como cada meta-heurística aqui citada é aplicada no problema de coloração, e o livro "*Handbook of Metaheuristics*" [10], que é um livro focado no tema de meta-heurísticas. Como essas duas referências são livros didáticos que dão uma visão geral, também apresentaremos nas subseções os artigos ou referências originais de cada algoritmo.

Vamos começar relembrando que um algoritmo é dito heurístico se ele possuir tempo de execução polinomial para toda instância e se devolve sempre uma solução viável. O campo de pesquisa em algoritmos heurísticos pode ser considerado o mais frutífero para o problema da coloração dos grafos [17]. Geralmente, heurísticas devolvem soluções não exatas, podendo até não reconhecer uma solução ótima. Porém, algumas heurísticas podem devolver excelentes resultados quando aplicadas a grafos específicos, sendo inclusive mais recomendadas que algoritmos exatos em alguns casos onde o tamanho da instância é muito grande.

Não existe definição única de meta-heurística, porém, sua ideia é ser um padrão algorítmico desenvolvido para ser aplicável a diversos problemas de otimização sendo necessário apenas algumas alterações em sua implementação e projeto [10]. A grande vantagem das meta-heurísticas é justamente essa adaptabilidade a diversos problemas. Alguns exemplos de meta-heurísticas são busca tabu, *Simulated Annealing*, algoritmos evolucionários e algoritmos de colônias de formigas.

Geralmente, meta-heurísticas aprendem e exploram sobre o espaço de solução do problema enquanto executam. Dessa forma, podem ir melhorando a tomada de decisão até devolverem a melhor solução encontrada. Dito isso, algoritmos meta-heurísticos costumam ser feitos para explorar o maior número de soluções viáveis para o problema sem perder o controle do tempo de execução, além de geralmente usar uma solução momentaneamente tomada como entrada para a tomada de decisão, ou seja, o local onde o algoritmo se encontra no momento influencia nos próximos passos que ele vai tomar [17].

Nessa seção começaremos apresentando alguns conceitos importantes para entender as meta-heurísticas e suas diferenças, como os espaços de soluções e a vizinhança. Depois disso, apresentaremos algumas meta-heurísticas e como elas são aplicadas ao problema de coloração de grafos. As meta-heurísticas que apresentaremos nesse trabalho serão *Hill-Climbing*, Coloração Tabu, Algoritmos Evolucionários e Colônia de Formigas.

2.2.1 Espaços de Soluções

Quando falamos de espaços de solução nos quais meta-heurísticas podem trabalhar, temos que entender a ideia de como essas meta-heurísticas operam quanto a otimização. Uma meta-heurística sempre começa com alguma solução do problema e então ela vai explorando algum espaço de soluções possíveis em busca de uma melhor solução viável. É nessa ação que entra a ideia de otimização. Por exemplo, no caso da coloração imagine que uma meta-heurística qualquer começa com uma solução inviável e então ela vai alterando as cores dos vértices, seguindo alguma regra, até obter uma solução viável, e que depois disso, ela busca minimizar a quantidade de cores com o mesmo procedimento. Essa atividade caracteriza uma exploração do espaço de soluções.

- **Espaço de Soluções Viáveis -** Nesse tipo de espaço trabalha-se apenas com soluções viáveis. Os algoritmos de coloração de grafos que trabalham nesse espaço geralmente usam alguma variação do algoritmo Guloso apresentado em conjunto com o Teorema da Permutação (Teorema 2.1) para encontrar permutações de vértices que possam reduzir o número de cores. Veja que com esse procedimento o algoritmo nunca sai do espaço de soluções viáveis.

- **Espaço de Soluções Inviáveis -** É o espaço onde a maioria das meta-heurísticas trabalha e contém soluções viáveis e também inviáveis. Geralmente, os algoritmos de coloração que trabalham nesse espaço começam estabelecendo um número k de cores e então procuram uma coloração viável para o grafo instância com k cores. Caso encontrem tal coloração, eles diminuem o valor de k e repetem o processo. Caso não encontrem tal coloração podem aumentar o valor de k.

- **Espaço de Soluções Parciais -** Essa abordagem recebe menos atenção que as outras duas e tem como principal característica conter **soluções parciais** do problema em questão. Para o problema de coloração de grafos, soluções parciais são aquelas que apresentam apenas um subconjunto de vértices do grafo particionados em cores. Quando aplicada ao problema da coloração de grafos, geralmente, segue a estratégia de construir um conjunto de vértices ainda não coloridos U. Quando ocorre um conflito, descolorimos algum vértice e o adicionamos a esse conjunto U.

Em uma mesma estratégia de resolução podemos ainda combinar espaços de solução, sendo que diferentes espaços de solução são usados em diferentes etapas da resolução. Por trás dessas combinações temos o objetivo de diversificar nossa busca, já que estratégias trabalhando em espaços de solução diferentes dificilmente terão o mesmo ótimo local.

2.2.2 Vizinhança

Dada uma solução S para o problema da coloração de grafos, ou seja, uma coloração dos vértices do grafo instância, um **operador de vizinhança** é uma operação que quando recebe a solução S modifica ela seguindo alguma regra e devolve uma outra solução para o problema, digamos S'. Toda solução S' gerada dessa forma é chamada de **solução vizinha** de S, e o conjunto de todas as S' é chamado **vizinhança** de S.

No caso do problema de coloração de vértices, o operador mais comum é a troca de cor de um dos vértices. Dessa forma, quaisquer duas soluções que tenham somente a cor de um vértice como diferença são chamadas de vizinhas. Para mensurar a distância de uma solução a outra no espaço de soluções geralmente usamos a quantidade de vértices com cores diferentes.

2.2.3 Algoritmo Hill-Climbing

Chamamos de *Hill-Climbing* uma técnica de otimização genérica que tem um princípio simples: só fazer mudanças quando encontrar melhoria da solução atual, sem nunca piorar a solução mesmo que isso possibilitasse uma melhora em algumas iterações. Assim, dada uma solução S, se é feita alguma operação sobre essa solução resultando em uma solução vizinha S', distinta da original, uma nova avaliação do valor da função objetivo é realizado e, caso tenhamos uma melhora, o próximo passo é repetir a operação na solução S' em busca de novas melhorias. Agora, se tivermos uma piora no valor da função objetivo, retornamos para S e aplicamos a operação outra vez mas com outros parâmetros. Se nenhuma solução vizinha de S tiver resultado de função objetivo melhor que S, encerramos a execução do algoritmo.

Em 2009, Lewis [16] propôs em um artigo a aplicação do método *Hill-Climbing* nos problemas de coloração de vértices e empacotamento. Com relação ao problema de coloração, o algoritmo começa com uma solução inicial viável construída pelo DSatur, digamos $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$. Depois dividimos as cores dessa solução viável em duas soluções parciais $\mathcal{S}$ e $\mathcal{T}$. Aplicamos uma busca em $\mathcal{T}$ visando vértices que possam ser transferidos para as cores em $\mathcal{S}$ sem causar conflitos, ou seja, buscamos aumentar a cardinalidade das cores em $\mathcal{S}$ e esvaziar $\mathcal{T}$. Ao final do procedimento as cores em $\mathcal{T}$ que não forem vazias são devolvidas a $\mathcal{S}$, que volta a ser uma solução viável diferente da solução original.

Por se tratar de um procedimento relativamente barato computacionalmente, podemos realizar essa operação I vezes em busca de melhorar a solução inicialmente dada. Para aumentar a área de exploração no espaço de soluções e evitar cair em ótimos locais, algumas etapas entre iterações também são apresentadas por Lewis [16]. Em cada iteração as soluções parciais $\mathcal{S}$ e $\mathcal{T}$ seriam divididas de forma diferente. Além disso, após cada

iteração, com a solução S completa definiríamos uma ordem de vértices π onde os vértices de cada cor estariam em sequência. Depois usaríamos o algoritmo Guloso para criar uma nova solução usando a ordem π definida. Perceba que mesmo com esses procedimento a nunca teríamos uma piora na qualidade da solução devido ao Teorema 2.1.

Outro ponto muito importante do *Hill-Climbing* e das meta-heurísticas em geral são os **hiperparâmetros**. Tratam-se de parâmetros passados ao algoritmo para serem usados no controle o processo de exploração e otimização. O exemplo mais óbvio seria a quantidade I de iterações do algoritmo, usada para controlar o algoritmo como critério de parada. Para o *Hill-Climbing* alguns outros hiperparâmetros possíveis são:

- *split*, valor entre 0 e 1 que indica qual porcentagem das cores deve ser destinada a S e qual deve ser destinada a $\mathcal{T}$ na divisão;
- I_{sm}, que indica um limite para quantidade de iterações sem melhora, isso é, caso o algoritmo faça I_{sm} iterações e não apresente melhoras encerramos o algoritmo mesmo que não tenham sido atingidas I iterações.

No Algoritmo 4 temos o pseudocódigo do algoritmo descrito. Veja que começamos com uma solução inicial definida pelo DSatur, porém, após isso já usamos o algoritmo Guloso para verificar se com essa heurística podemos encontrar uma coloração com menos cores. Temos também algumas funções cujos passos não foram definidos de forma explicita, como a Divide, que recebe a coloração viável $\mathcal{S}$ e divide ela em duas soluções parciais, e a Transfere, responsável pela transferência de vértices das cores de $\mathcal{T}$ para $\mathcal{S}$.

Algoritmo 4: HillClimbing(G, I)

Entrada: Grafo G e número máximo de iterações I
Saída: Partição dos vértices de G em uma coloração viável

```
1 S ← DSatur(G)
2 π ← Ordem dos vértices V(G) onde os vértices de cada cor S_i ∈ S estão sempre
    em sequência
3 para i ← 1 até I faça
4 |   S ← Guloso(G, π)
5 |   S, T ← Divide(S)
6 |   S ← Transfere(S, T)
7 |   π ← Ordem dos vértices V(G) onde os vértices de cada cor S_i ∈ S estão
  |     sempre em sequência
8 devolve S
```

Sobre o tempo de execução, o hiperparâmetro I é fundamental devido ao laço **para** da linha 3. Seja $n = |V(G)|$ e $m = |E(G)|$. Também temos um DSatur sendo executado no início do algoritmo, na linha 1, e como vimos na Seção 2.1.2 é um algoritmo que executa em $O(n^2+m)$, assim como o Guloso executado na linha 4, executado dentro do laço **para**. A ordenação dos vértices em π é algo que pode ser feito em $O(n)$ dado que os vértices já estão particionados por cor. A sub-rotina Divide tem como principal ação a transferência de uma parte dos vértices de G para $\mathcal{T}$ fazendo com que ela e $\mathcal{S}$ sejam soluções parciais naquele ponto do algoritmo, portanto leva tempo $O(n)$. Já a função Transfere não somente faz uma movimentação dos vértices de $\mathcal{T}$ pra $\mathcal{S}$, operação limitada por $O(n)$, mas também avalia se a adição de um vértice de $\mathcal{T}$ em um subconjunto de $\mathcal{S}$ mantém $\mathcal{S}$ como partição viável dos vértices. Veja que durante a Transfere avalia-se todas as arestas de cada vértice de $\mathcal{T}$, o que é limitado por $O(m)$. Assim, o algoritmo HillClimbing tem como tempo de execução $O(n^2+m)+O(I)\times(O(n^2+m)+O(n)+O(m)) = O(In^2+Im)$.

2.2.4 Algoritmo TabuCol

Para começar a falar do Algoritmo TabuCol vamos falar sobre duas estratégias de exploração de espaço de soluções, a Descida e a Busca Tabu:

- **Descida -** Dada uma solução S, calculamos o custo de todas as soluções vizinhas de S. Vamos considerar que S' é um vizinho de S com menor custo. Assim, o algoritmo se movimenta para S' e repete o processo de calcular todos os vizinhos de S'. Caso não haja um vizinho com custo menor o algoritmo para na solução do momento. Esse algoritmo tem o claro problema de cair em ótimos locais. Caso estivéssemos falando de um problema de maximização usaríamos a **Subida**, que é um algoritmo análogo ao da descida [10]. Esse método de exploração do espaço de soluções é semelhante ao HillClimbing, porém, aqui é feita uma análise de todas as soluções vizinhas antes de realizar um movimento.

- **Busca Tabu -** Trata-se de uma especificação do algoritmo de descida, apresentando um mecanismo para escapar de ótimos locais. Basicamente, aplicamos a descida de forma que, quando identificamos que o algoritmo caiu em ótimo local (não há melhora no custo da solução), permite-se que ocorram movimentos para soluções piores. Para evitar rodar em círculos, é mantida uma lista de soluções chamadas **lista tabu**. O algoritmo é impedido de voltar a alguma solução nessa lista durante algumas iterações. A forma como a lista tabu é gerenciada (tempo de uma solução na lista em iterações) é algo parametrizável. É considerada uma das estratégias meta-heurísticas mais usadas e das que apresenta melhores resultados também [10].

Apresentadas essas duas estratégias de exploração vamos falar do TabuCol (Coloração Tabu). O TabuCol opera em um espaço de soluções inviáveis e sempre opera em função de diminuir a quantidade de colisões. Consideramos que uma **colisão** ocorre quando dois vértices vizinhos possuem a mesma cor. O TabuCol começa com um número k de cores, podendo a solução de partida ser própria ou imprópria.

A aplicação da Busca Tabu na coloração de grafos gerou diversos trabalhos, seja com foco exclusivo nesse procedimento, como em Hertz e de Werra [11], ou seja em trabalhos que apresentam estratégias híbridas de coloração, como em Thompson e Downsland [8]. Aqui vamos tratar de uma versão do TabuCol apresentada por Galliner e Hao [9], em um trabalho onde queriam explorar um algoritmo híbrido entre a Busca Tabu e o Algoritmo Evolucionário.

Dada uma partição $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ e seja $V(G) = \{v_1, v_2, \ldots, v_n\}$ com $n = |V(G)|$. Um movimento no espaço de soluções a partir de $\mathcal{S}$ é feito da seguinte forma: escolhemos um vértice $v \in S_i$ que esteja causando um conflito e movemos v para $S_j \neq S_i$. A lista tabu é mantida em uma matriz T, de dimensões $n \times k$. Digamos que em uma iteração ℓ o algoritmo movimente um vértice v_a de S_i para uma outra cor $S_j \neq S_i$. Então, na posição T_{ai} colocamos o valor $\ell + t$, onde t é o parâmetro que indica quantas iterações um movimento fica com *status* tabu.

Dada uma solução S e escolhido um vértice v_a, devemos calcular cada uma das $k - 1$ soluções vizinhas, que geralmente é o processo mais demorado do algoritmo. Após avaliar as soluções vizinhas, o movimento é realizado para o vizinho com maior decréscimo de conflitos ou com menor acréscimo. Empates são resolvidos aleatoriamente.

Através do Algoritmo 5 apresentamos as ideias descritas como um pseudocódigo. Veja que nesse pseudocódigo algumas funções estão implícitas, como a Conflitos, que devolve o conjunto de todos os conflitos como pares não ordenados de vértices, Vizinhas, que devolve todas as soluções vizinhas e quantos conflitos cada uma apresenta, Movimento, que troca a cor de um vértice para uma outra cor e devolve a solução vizinha resultante desse movimento. Lembramos que TabuCol não necessariamente devolve uma solução viável com as k cores passadas, mas caso encontre podemos salvar essa solução em uma variável auxiliar $\mathcal{S}'$. Além disso, toda vez que uma solução viável melhor for encontrada podemos diminuir o valor de k para a próxima iteração.

Dentre os hiperparâmetros do TabuCol podemos citar:

- t, o valor tabu que representa a quantidade de iterações que um movimento fica na lista tabu;
- k, valor máximo de cores que uma solução pode usar durante o algoritmo;
- I, número máximo de iterações que o algoritmo vai executar.

Algoritmo 5: TabuCol(G, t, k, I, $\mathcal{S}$)

Entrada: Grafo G, valor tabu t, máximo de cores k, número máximo de iterações I e uma solução inicial $\mathcal{S}$
Saída: Partição dos vértices de G em uma coloração viável ou inviável

1 $\ell \leftarrow 1$
2 Seja T uma a matriz $n \times k$ com todos os valores iniciados em 0
3 **se** $\mathcal{S} = \emptyset$ **então**
4 | Atribua aos vértices em $V(G)$ uma cor entre 1 e k, gerando uma coloração $\mathcal{S}$
5 $\mathcal{S}' \leftarrow \mathcal{S}$
6 $c \leftarrow$ Conflitos$(G, \mathcal{S})$
7 **enquanto** $c \neq \emptyset$ *ou* $\ell \leq I$ **faça**
8 | *vizinhas* $\leftarrow$ Vizinhas$(G, \mathcal{S}, v)$
9 | Seja o movimento do vértice $v \in S_i$ para $S_j \neq S_i$ aquele que leva à solução vizinha com o menor número de conflitos, também valendo que $T_{vj} < \ell$
10 | $T_{vi} \leftarrow \ell + t$
11 | $\mathcal{S} \leftarrow$ Movimento$(\mathcal{S}, v, j)$
12 | **se** $\mathcal{S}$ *é solução viável e* $|\mathcal{S}| \leq |\mathcal{S}'|$ **então**
13 | | $\mathcal{S}' \leftarrow \mathcal{S}$
14 | | $k \leftarrow k - 1$
15 | | Atribua aos vértices em $V(G)$ uma cor entre 1 e k, gerando uma coloração $\mathcal{S}$
16 | | $c \leftarrow$ Conflitos$(G, \mathcal{S})$
17 | $\ell \leftarrow \ell + 1$
18 **se** $\mathcal{S}' \neq \emptyset$ **então**
19 | $\mathcal{S} \leftarrow \mathcal{S}'$
20 **devolve** $\mathcal{S}$

Sobre o tempo de execução do algoritmo, seja $n = |V(G)|$ e $m = |E(G)|$. A atribuição aleatória de cores aos vértices para construir uma solução inicial é $O(n)$. Calcular os conflitos do grafo usando Conflitos é $O(n^2)$, por ter que avaliar par a par os vértices. O laço principal que inicia na linha 7 tem como um dos critérios de parada quando a quantidade de iterações tiver atingido o valor I. Dentro do laço temos primeiro a execução de Vizinhas, que pode executar até n^2 movimentos de troca de cor para mapear as vizinhas da solução atual, além também termos que calcular a quantidade de conflitos de cada uma dessas, que é $O(n^2)$. Então, a complexidade de Vizinhas no geral é $O(n^4)$. Podemos dizer que a busca pela melhor vizinha é $O(n)$ por ser uma busca simples. A execução da função Movimento leva tempo constante. Dentro do condicional que avalia se a solução é viável e melhor que a última salva, temos operações de atribuição constantes e um novo cálculo de Conflitos que é $O(n^2)$. Assim, o tempo de execução geral do algoritmo é $O(In^4)$, pois é dominado pela apuração da vizinhança.

2.2.5 Algoritmo Evolucionário

Algoritmos Evolucionários é uma nomenclatura usada para nos referirmos a uma meta-heurística bio-inspirada que baseia-se em alguns mecanismos evolucionários, em especial, recombinação, mutação e seleção. Trata-se de uma estratégia popular que tem sido muito utilizada para atacar problemas difíceis e complexos [17]. É uma estratégia que se baseia fortemente em aleatoriedade, especialmente nas fases relacionadas à mutação. A ideia dessa meta-heurística é começar com um conjunto de soluções inicial, onde as soluções são chamadas de **indivíduos** e o conjunto delas de **população**, e ir evoluindo essa população através da recombinação entre soluções usando operadores específicos nos processos de mutação aleatória e seleção da próxima geração. Para comparar soluções e mensurar a evolução usamos o valor de ***fitness***, que nada mais é que a função que dá valor a um indivíduo. Assim, evoluir uma população trata-se de buscar melhores soluções para o problema em questão, usando de certas operações. Algoritmos Evolucionários trabalham no espaço de soluções impróprias a não ser que sejam modificados para atuar no espaço de soluções próprias.

Além de Algoritmo Evolucionário um outro termo muito semelhante é Algoritmo Genético. Segundo Gendreau e Potvin [10], as ideias de mutação e seleção são originárias do que primeiro se chamava de Algoritmos Evolucionários, esses descritos desde a década de 1960. Embora o termo mais comum para meta-heurísticas que envolvam recombinação seja Algoritmos Genéticos, entende-se que Algoritmos Evolucionários seja um termo mais abrangente.

Já citamos que as três principais etapas dos Algoritmos Evolucionários são recombinação, mutação e seleção. Agora, vamos explicar detalhadamente a ideia por trás de cada uma dessas etapas:

1. **Recombinação ou *Crossover*:** Dados dois indivíduos da população, sendo que cada um deles representa de alguma forma uma solução (viável ou inviável), misturamos os elementos que representam essas soluções através de um operador específico e obtemos uma, ou até mais de uma, nova solução que carrega características das soluções que a geraram. Nessa situação os dois indivíduos originais são chamados de soluções mães, enquanto o indivíduo gerado é a solução filha.

2. **Mutação:** Após obtermos uma solução filha usamos aleatoriedade para fazermos alterações (mutações) nela, assim trazendo um "fato novo" para as soluções que temos para o problema, evitando possíveis vieses que podem levar a ótimos locais.

3. **Seleção:** Realizada a mutação, temos agora, além da população original, um conjunto de novos indivíduos e precisamos selecionar dentre todas essas soluções aquelas que continuarão no processo evolucionário, o que caracteriza uma pressão evolutiva.

Os operadores de recombinação geralmente são o principal diferencial entre diferentes implementações de Algoritmo Evolucionário pois são eles que determinam como a hereditariedade é herdada por soluções filhas.

Explicados esses conceitos sobre Algoritmos Evolucionários vamos focar no problema da coloração de grafos. Para o problema de coloração de grafos a função de custo (*fitness*) usada é o número de colisões da solução, de forma que soluções mais adaptadas são aquelas com menos colisões. A implementação que vamos nos aprofundar aqui será a do mesmo trabalho que usamos na subseção anterior sobre Coloração Tabu, o trabalho de Galliner e Hao [9], que visa apresentar um híbrido entre essas duas estratégias. Galliner e Hao apresentam o método de recombinação conhecido como **Greedy Partition Crossover (GPC)**.

A ideia do GPC é que o filho herde cores com muitos vértices de suas mães, herdando cores inteiras. Mas é importante destacar que cores inteiras não necessariamente tem muitos vértices. A primeira cor a ser herdada é aquela com mais vértices, considerando duas mães X e Y. Sem perda de generalidade, suponha que a mãe que possui a cor selecionada seja X. Para evitar duplicação de vértices, os vértices pertencentes à primeira cor adicionada são eliminados de ambas as mães. Em seguida, escolhemos a maior cor de Y e adicionamos à solução filha. Então, o processo de atualização de vértices e escolha da cor é repetido, alternando-se entre as mães, até que a solução filha contenha todos os vértices do grafo. Para o GPC podemos escolher um parâmetro k que determina quantas cores queremos que a solução filha tenha. Após obtermos a solução filha, usamos o TabuCol para melhorar localmente a solução. Na Tabela 1 temos um exemplo de aplicação do GPC.

Tabela 1 – Exemplo de aplicação do GPC na construção de uma filha.

Etapa	Mãe X	Mãe Y	Filha
1	$\{\{v_1, v_2, v_3\}, \{v_4, v_5, v_6, v_7\}, \{v_8, v_9, v_{10}\}\}$	$\{\{v_3, v_4, v_5, v_7\}, \{v_1, v_6, v_9\}, \{v_2, v_8, v_{10}\}\}$	$\{\ \}$
2	$\{\{v_1, v_2, v_3\}, \{v_8, v_9, v_{10}\}\}$	$\{\{v_3\}, \{v_1, v_9\}, \{v_2, v_8, v_{10}\}\}$	$\{\{v_4, v_5, v_6, v_7\}\}$
3	$\{\{v_1, v_3\}, \{v_9\}\}$	$\{\{v_3\}, \{v_1, v_9\}\}$	$\{\{v_4, v_5, v_6, v_7\}, \{v_2, v_8, v_{10}\}\}$
4	$\{\{v_9\}\}$	$\{\{v_9\}\}$	$\{\{v_4, v_5, v_6, v_7\}, \{v_2, v_8, v_{10}\}, \{v_1, v_3\}\}$
5	$\{\ \}$	$\{\ \}$	$\{\{v_4, v_5, v_6, v_7\}, \{v_2, v_8, v_{10}\}, \{v_1, v_3\}, \{v_9\}\}$

Para construir a população original usamos um algoritmo heurístico de construção de soluções. Aqui vamos usar o Dsatur. É interessante que a população original seja a mais diversa possível, e para evitar que o Dsatur sempre inicie sua coloração com um dos vértices de maior vizinhança, critério de desempate que apresentamos em sua seção, podemos passar como hiperparâmetro um vértice aleatório v para que o algoritmo comece a coloração construtiva com ele. Aqui vamos considerar essa versão alterada do Dsatur. Para melhorar a qualidade da resposta fornecida por Algoritmos Evolucionários podemos mesclar eles com procedimentos de busca mais agressivos e determinísticos, como

a Busca Tabu, como Galliner e Hao [9] fizeram. Assim como em outras meta-heurísticas, usamos iterações em busca de melhorar a resposta devolvida, mas no caso dos algoritmos evolucionários chamamos as iterações de *tuning* da solução.

Algoritmo 6: EVOLUCIONARIO(G, p, t)

Entrada: Grafo G, tamanho da população p e iterações de *tuning* t
Saída: Partição dos vértices de G em uma coloração viável

```
1  pop ← POPULACAO(G, p)
2  ℓ ← 1
3  enquanto ℓ ≤ t faça
4  |  m1, m2 ← MAES(pop)
5  |  f ← RECOMBINACAO(m1, m2)
6  |  f ← TABUCOL(G, f)
7  |  pop ← SELECAO(pop, m1, m2, f)
8  |  ℓ ← ℓ + 1
9  Seja S a solução de pop com melhor fitness fitting
10 devolve S
```

O Algoritmo 6 apresenta um pseudocódigo com as ideias do Algoritmo Evolucionário que usa o GPC. A função POPULACAO cria os p integrantes da primeira geração da população usando algum algoritmo capaz de gerar soluções para coloração de grafos de forma rápida, como GULOSO ou DSATUR apresentados nesse mesmo trabalho. Com a população inicial criada começamos as iterações com a execução de MAES, que seleciona as duas soluções mães dentro das soluções na população inicial usando um critério qualquer. Após isso realizamos a RECOMBINACAO dessas mães para gerar uma filha. Com a solução filha gerada usamos o TABUCOL em busca de melhorar essa solução (omitimos os parâmetros de TABUCOL mas eles também podem ser passado na chamada de EVOLUCIONARIO). Essa etapa do TABUCOL é nossa mutação. Por fim, com a nova filha definida temos que exercer a pressão evolutiva na população, o que é feito através da função SELECAO, que substitui a mãe com pior *fitness* pela filha criada, independente da mãe ter *fitness* melhor ou pior que a filha. Ao final do processo de *tuning* temos uma população com p soluções, mas devolvemos como resposta apenas aquela com melhor *fitness*.

O tempo de execução de EVOLUCIONARIO depende muito de suas sub-rotinas, afinal é um algoritmo praticamente só composto por elas. Seja $n = |V(G)|$ e $m = |E(G)|$ A primeira das subrotinas, POPULACOES, basicamente consiste em p execuções de um algoritmo heurístico. Vamos considerar aqui que seja o DSATUR que, como já mostrado, executa em $O(n^2 + m)$, logo $O(p(n^2 + m))$. Na linha 3 entramos em um laço **para** que pode ser executado até t vezes. A MAES simplesmente seleciona duas soluções dentro da população de forma aleatória, o que pode ser feito em tempo constante $O(1)$. Já em RECOMBINACAO temos a execução do GPC, que é uma operação que percorre as duas soluções mães e elimina de uma delas os vértices que foram selecionados na cor da outra, o que tem tempo $O(n)$. Também é em RECOMBINACAO que se constrói a solução

filha, e esse processo precisa, para cada adição de uma cor, verificar quais os vértices já presentes na solução até aquele momento, o que leva $O(n)$. Assim, RECOMBINACAO acaba sendo uma operação $O(n)$. A função TABUCOL usada não se diferencia daquela que apresentamos na Seção 2.2.4, assim, temos uma execução $O(n^4)$, um procedimento de otimização bem custoso para o algoritmo. Finalizando o laço **enquanto** temos a SELECAO, que simplesmente compara o *fitness* das mães e substitui a com pior *fitness* da população do algoritmo. Todo esse procedimento de seleção pode ser feito em $O(1)$, pois só envolve comparar um valor e substituir uma solução por outra, considerando uma representação das soluções que permite essa operação ser diretamente sobre elas e não sobre suas partições. Assim, o tempo de execução total do EVOLUCIONARIO é dominado pela execução do TABUCOL, que é $O(n^4)$.

2.2.6 Algoritmo ANTCOL

Otimização por Colônia de Formigas (*Ant Colony Optimization*) é uma meta-heurística com bio-inspiração na forma como formigas determinam caminhos mais eficientes entre fontes de comida e suas colônias. A ideia é que as formigas começam andando aleatoriamente em todas as direções em busca de comida. Quando a encontram, retornam para a colônia deixando uma trilha de feromônios que influencia as outras formigas. Agora, quando outras formigas saírem da colônia, a direção que vão seguir já não é mais aleatória, pois passa a ser impactada de alguma forma pelos feromônios deixados por outras formigas. Quanto mais formigas encontrarem uma mesma fonte de comida mais forte será a trilha de feromônios daquela fonte e maior será seu impacto na decisão das outras formigas ao deixarem a colônia. Trilhas não são eternas, pois a cada período de tempo os feromônios evaporam, enfraquecendo-a. Assim, trilhas mais longas tendem a ser perdidas enquanto trilhas mais próximas da colônia tendem a ser mais reforçadas. A tendência é que as trilhas das formigas venham a convergir em algum momento.

Desde que foi proposta, essa meta-heurística levou a muitos resultados significativos na área de otimização, sendo uma das estratégias preferidas e mais bem sucedidas na resolução de problemas complexos [10]. Também é comumente usada em estratégias híbridas com outras meta-heurísticas, como Algoritmos Evolucionários. Veja que algumas das principais ideias para otimização que essa bio-inspiração traz são a construção de soluções e suas avaliações impactarem nos próximos passos do algoritmo, isso é, na construção das próximas soluções, com estímulos e desencorajamentos a tomar certas escolhas. Enquanto computacionalmente as formigas seriam soluções, as trilhas de feromônios seriam parâmetros que impactariam uma função probabilística responsável pela construção de outras soluções. Nessa seção, vamos explorar como aplicar a *Ant Colony* (AntCol) para resolver o problema de coloração de grafos, e para isso vamos usar o trabalho de Downsland e Thompson [8], que apresenta uma proposta de aplicação dessa metaheurística ao problema

de coloração. E diferentemente das últimas subseções, vamos apresentar o Algoritmo 7 e depois explicaremos como ele é um uso da meta-heurística para o problema.

Algoritmo 7: AntCol(G, f, I, α, β, ρ)

Entrada: Grafo G, quantidade de formigas f, iterações I, parâmetro α, parâmetro β e taxa de evaporação ρ

Saída: Partição dos vértices de G em uma coloração viável ou inviável e *viavel* que indica se foi ou não encontrada uma solução viável

```
1  n ← |V(G)|
2  k ← n
3  Seja T uma matriz n × n com todos os valores iniciados em 0
4  S' ← ∅
5  para ℓ ← 1 até I faça
6  |  Seja γ uma matriz n × n com todos os valores iniciados em 0
7  |  melhor ← k
8  |  viavel ← Falso
9  |  para formiga ← 1 até f faça
10 |  |  S ← RLFColonia(G, k, T, α, β)
11 |  |  se S é solução parcial então
12 |  |  |  Colorir os vértices restantes com uma das k cores de forma aleatória
13 |  |  |  S ← TabuCol(G, S)
14 |  |  se S é viável então
15 |  |  |  viavel ← Verdadeiro
16 |  |  |  se |S| < melhor então
17 |  |  |  |  melhor ← |S|
18 |  |  |  |  S' ← S
19 |  |  para Toda posição uv da matriz γ faça
20 |  |  |  γ_uv ← γ_uv + Feromonio(S)
21 |  para Toda posição uv da matriz T faça
22 |  |  T_uv ← T_uv × ρ + γ_uv
23 |  se viavel = Verdadeiro então
24 |  |  k ← melhor − 1
25 se S' ≠ ∅ então
26 |  S ← S'
27 devolve S, viavel
```

No Algoritmo 7 temos dois laços **para** aninhados, sendo que cada execução do laço externo será chamado de **geração de formigas** e cada execução do interno de **formiga**. Assim, temos I gerações de f formigas. Veja que também temos duas matrizes $n \times n$, T e γ, onde $n = |V(G)|$. A matriz T é a que chamamos de **matriz global** e é responsável por armazenar informações de todas as gerações de formigas e consolidar essas informações de cada geração. Já a matriz γ representa a **matriz local** e é responsável por armazenar as informações de todas as formigas de uma geração, isso é, de todas as soluções construídas em uma geração.

A cada solução construída por uma formiga usamos o algoritmo RLFColonia, que é uma versão modificada do RLF que também aceita um parâmetro k que limita a quantidade de cores na solução $\mathcal{S}$ devolvida, mesmo que isso signifique RLFColonia devolver uma solução parcial. Aqui estamos usando na inicialização do algoritmo $k = n$ por simplicidade na implementação, mas cabe dizer que qualquer outro limitante superior poderia ser usado no algoritmo. Caso $\mathcal{S}$ seja parcial, colorimos os vértices restantes segundo algum critério e aplicamos TabuCol em busca de melhorar a solução. Agora, caso ela seja viável acionamos um *flag* para salvar essa informação e usamos a variável auxiliar $\mathcal{S}'$ para salvar essa solução. Depois, aplicamos a política de feromônios nessa solução. Ao final de uma geração usamos a matriz γ para atualizar a matriz T em conjunto com o hiperparâmetro ρ usado na evaporação dos valores.

Resta esclarecer como as funções RLFColonia e Feromonio funcionam e afetam a construção de novas soluções. Primeiro, perceba que as duas matrizes $n \times n$ têm essas dimensões porque cada posição delas representa um par de vértices $u, v \in V(G)$. A ideia é usar essas matrizes para indicar o quanto é vantajoso que esses dois vértices u, v pertençam à mesma cor, isso é, quanto maior os valores de T_{uv} e γ_{uv}, mais entende-se que eles devem estar na mesma cor. Dito isso, RLFColonia tem outra grande diferença do algoritmo RLF que já apresentamos, pois ele usa uma função de probabilidade P_{vi} para determinar se um vértice v deve ou não ser adicionado à cor S_i. Ou seja, mesmo que adicionar um vértice v à cor S_i não gere um conflito pode ser que ele ainda não seja adicionado. A adição ou não depende única e exclusivamente dessa função, cuja fórmula segue:

$$P_{vi} = \begin{cases} \dfrac{\tau_{vi}^{\alpha} \times \eta_{vi}^{\beta}}{\sum_{u \in U}(\tau_{ui}^{\alpha} \times \eta_{ui}^{\beta})}, & \text{se } v \text{ não estiver colorido} \\ 0, & \text{se } v \text{ já tiver sido colorido} \end{cases} \tag{2.1}$$

onde

$$\tau_{vi} = \frac{\sum_{u \in S_i} T_{uv}}{|S_i|} \tag{2.2}$$

enquanto η_{vi} é o grau do vértice v no grafo induzido pelos vértices ainda não coloridos e U é o conjunto de vértices ainda não coloridos, como apresentado no Algoritmo RLF.

Vamos entender um pouco melhor essas variáveis e suas influências no resultado final. Veja que η_{vi} indica o quanto colorir o vértice v com a cor i vai restringir outros vértices ainda não coloridos. Já τ_{vi} é a soma de todos os valores de feromônio daquele vértice com os outros vértices já coloridos com a cor i dividido pela pelo tamanho da cor i, de forma que a construção de cores grandes são desfavorecidas enquanto os valores da matriz global impactam diretamente na probabilidade daquele vértice ser colorido. Agora, analisando a fórmula de P_{vi} vemos que η_{vi} e τ_{vi} do vértice considerado são diretamente

proporcionais à probabilidade, enquanto esses mesmos parâmetros para outros vértices ainda não coloridos atuam inversamente ao valor da probabilidade. Na prática, o cálculo de P_{vi} envolve favorecer os vértices com maiores η_{vi} e τ_{vi} em comparação aos outros vértices. Além disso, também temos os hiperparâmetros α e β, que indicam a força de η_{vi} e τ_{vi} no cálculo da probabilidade.

Para fecharmos a explicação do algoritmo vamos falar da função FEROMONIO. Ela é executada sobre cada solução $\mathcal{S}$ construída e recebe como entrada justamente $\mathcal{S}$. O que essa função faz é devolver um valor que representa uma aplicação de alguma regra de favorecimento a soluções viáveis e sem conflitos. Segue o exemplo dado no livro de Lewis [17]:

$$\text{FEROMONIO}(\mathcal{S}) = \begin{cases} \dfrac{1}{\text{número de conflitos}}, & \text{se } \mathcal{S} \text{ não for viável} \\ 3, & \text{se } \mathcal{S} \text{ for viável} \end{cases} \tag{2.3}$$

Sobre o tempo de execução desse algoritmo, veja que a maioria das operações mais custosas são realizadas em RLFCOLONIA e TABUCOL, sendo essas executadas If vezes. Sejam $n = |V(G)|$ e $m = |E(G)|$. Como apresentado, o RLFCOLONIA é uma adaptação do RLF da Seção 2.1.3, onde a grande diferença é a possibilidade de não colorir um vértice v mesmo que sua adição à cor i não inviabilize a solução. O cálculo da função de probabilidade envolve calcular τ e η para cada um dos n vértices. Calcular η é calcular o grau do vértice v a ser colorido no subgrafo induzido $G[U]$, o que pode ser feito verificando os vizinhos em $O(m)$. Já calcular τ envolve passar por cada vértice u pertencente à cor S_i considerada e consultar o valor T_{uv}, o que é feito em $O(n)$. Então, calcular τ e η é $O(n+m)$. Mas veja que calcular P_{vi} também envolve calcular τ e η para os vértices contidos em U, assim temos que multiplicar esse valor por $O(n)$, resultando em $O(n^2 + m)$, sendo que o m se mantém linear na expressão porque no final cada aresta será consultada até duas vezes para calcular os graus de todos os vértices. Considerando o Algoritmo 3, o cálculo de P_{vi} seria feito sempre dentro do segundo laço **enquanto** aninhado, assim temos que multiplicar o tempo de execução já mostrado do RLF pelo $O(n^2+m)$ referente à função de probabilidade. Lembrando que o $O(m)$ considerando no RLF não é alterado pela função de probabilidade. Assim, RLFCOLONIA tem tempo de execução $O(n^5 + nm + m)$. Sendo que o termo nm surge por causa do cálculo de probabilidade ser executado para cada vértice considerado no RLFCOLONIA. Já o TABUCOL tem o mesmo tempo de execução que o apresentado na análise do Algoritmo 5, $O(n^4)$. Outras operações que valem ser destacadas no cálculo do tempo de execução de ANTCOL são as atualizações das matrizes γ e T, o que é proporcional sempre a $O(n^2)$. Concluindo, a operação com tempo dominante em ANTCOL é o RLFCOLONIA devido ao cálculo da probabilidade P_{vi}, ficando assim em $If \times O(n^5)$.

2.3 Algoritmos Exatos

Até o momento vimos algoritmos heurísticos, sejam gulosos ou meta-heurísticos, e a principal qualidade deles é a rápida execução e a diversidade de estratégias que permitem a realização de experimentos diversos em busca de boas soluções. Porém, estratégias heurísticas não nos garantem que vamos encontrar solução ótima para o problema, e mesmo quando encontram uma solução ótima não sabem que a encontraram e podem continuar sua execução. Isso não faz com que percam o valor, especialmente em casos onde buscamos uma boa solução que não necessariamente é a ótima para o problema.

Por outro lado, algoritmo exatos são aqueles em que temos sempre a garantia que a resposta final do algoritmo é a solução ótima para o problema. O grande problema desse tipo de algoritmo é que para problemas $\mathcal{NP}$-difíceis seu tempo de execução não é polinomial, fazendo com que na prática seu uso seja completamente inviável para instâncias maiores de um problema.

Nessa seção apresentamos os dois algoritmos exatos que foram implementados no pacote desenvolvido ao longo do projeto. Os dois algoritmos seguem estratégias diferentes, mas a ideia central é a mesma: testar de maneira construtiva soluções de coloração para o grafo instância e devolver apenas aquela que for a melhor solução. O primeiro contato com ambos os algoritmos foi através da dissertação de mestrado de Lima (2017), que trata justamente de algoritmos exatos para coloração de grafos.

O primeiro algoritmo que apresentamos é o Algoritmo de Lawler, que não devolve uma coloração para um grafo, mas sim o número cromático desse grafo usando programação dinâmica com base em uma recursão. Já o segundo algoritmo apresentado é o DSatur Exato, que é um algoritmo exato que usa o conceito de Grau de Saturação (2.1) para construir as soluções do problema de forma dinâmica, fazendo uso de *backtracking* para construir uma árvore de busca.

2.3.1 Algoritmo de Lawler

Lawler [14] apresentou um algoritmo exato que devolve o número cromático de um grafo. Esse algoritmo usa programação dinâmica para resolver o problema. O objetivo de Lawler nesse trabalho era justamente apresentar um algoritmo para o problema que também tivesse limites superiores de tempo de execução bem definidos, algo que não havia sido demonstrado em outros algoritmos para o problema apresentados até aquele momento [14].

Falando sobre a técnica de programação usada, a programação dinâmica é uma técnica de construção de algoritmos poderosa utilizada em problemas que podem ser resolvidos através de recursão, isso é, um problema que gera subproblemas que são resolvidos recursivamente. A característica mais marcante da programação dinâmica é

evitar resolver o mesmo subproblema diversas vezes, e para isso os algoritmos fazem uso de memória extra para armazenar as soluções dos subproblemas. Um algoritmo de programação dinâmica está associado a uma recursão para o problema sendo resolvido. No caso do Algoritmo de Lawler, encontram-se soluções para subgrafos induzidos do grafo original, que seriam os subproblemas. Esse algoritmo usa uma estratégia de programação dinâmica começando a construir as soluções a partir dos menores subgrafos e aumentando a complexidade, aproveitando sempre as soluções já construídas até aquele momento. Essa estratégia é conhecida como *bottom-up*.

A primeira ideia que sustenta o Algoritmo de Lawler é um resultado do trabalho de Wang [22], que também apresenta um algoritmo exato para encontrar o número cromático. O resultado em questão é de que para alguma coloração ótima do problema podemos encontrar uma cor que também é conjunto independente maximal. Formalizamos isso na Proposição 2.1.

Proposição 2.1 ([22])**.** *Dado um grafo G e, para todo vértice $v \in V(G)$, seja $\{I_1, I_2, \ldots, I_\ell\}$ a coleção de todos os conjuntos independentes maximais em G que contêm v. Existe uma $\chi(G)$-coloração ótima de G tal que uma de suas cores é I_i, para algum $1 \leq i \leq \ell$.*

Esse resultado é usado por Lawler [14] para fundamentar um resultado apresentado em seu trabalho que diz que para algum conjunto independente maximal, se retirarmos esse conjunto do grafo, obtemos um subgrafo com número cromático menor que o grafo original. Enunciamos esse resultado na Proposição 2.2.

Proposição 2.2 ([14])**.** *Dados um grafo G e um subconjunto $S \subseteq V(G)$ com $S \neq \emptyset$, existe uma coloração de $G[S]$ em que ao menos um conjunto independente maximal I em $G[S]$ é tal que*

$$\chi(G[S]) = 1 + \chi(G[S] - I).$$

Finalmente, com o último resultado apresentado chegamos à recursão que define o Algoritmo de Lawler. Tal recursão é apresentada no Teorema 2.5, que apresenta como caso base o caso em que o subconjunto S de vértices é vazio, caso e que o número cromático é zero. Já a chamada recursiva baseia-se na Proposição 2.2, realizando a próxima chamada para o subconjunto S de vértices sem um conjunto independente maximal de $G[S]$, o que é garantido pelo teorema.

Teorema 2.5 ([14])**.** *Para todo $S \subseteq V(G)$, o número cromático de $G[S]$ é dado pela recorrência*

$$\chi(G[S]) = \begin{cases} 0 & \text{se } S = \emptyset \\ 1 + \min\{\chi(G[S] - I)\}, \text{ para todo conjunto independente } I \subseteq V(G[S]) & \text{se } S \neq \emptyset\,. \end{cases}$$

O pseudocódigo que usa da recursão do Teorema 2.5 para chegar ao número cromático encontra-se no Algoritmo 8. Veja que é um algoritmo de poucas linhas, porque

seu funcionamento realmente é simples e envolve poucos passos distintos. A grande perda que temos com esse algoritmo é a sua eficiência, afinal trata-se de um algoritmo exato e ele acaba sendo exaustivo mesmo com melhoras em relação a um algoritmo de força bruta menos refinado. Falando sobre os passos do algoritmo, o que ele faz é percorrer todos os 2^n subgrafos induzidos de G, representados pelos seus conjuntos de vértices $S \subseteq V(G)$. Para cada subgrafo induzido $G[S]$ computam-se os seus conjuntos independentes maximais, operação demonstrada ser $O(3^{n/3})$ usando o algoritmo de Bron-Kerbosch [3] que encontra as cliques maximais de um grafo. Devido à relação entre cliques e conjuntos independentes maximais, basta aplicar Bron-Kerbosch no grafo complementar de $G[S]$ para encontrar todos os conjuntos independentes maximais.

Sejam $n = |V(G)|$ e $m = |E(G)|$. Observando o algoritmo LAWLER vemos que para cada subgrafo possível de G, que são 2^n, fazemos uma outra iteração aninhada que passa por cada conjunto independente maximal do subgrafo induzido, que podem ser até $3^{n/3}$. Estamos considerando que a sub-rotina usada para calcular esses conjuntos independentes maximais é o algoritmo de Bron-Kerbosch. Assim temos uma complexidade de tempo $O((2 + 3^{1/3})^n)$ para o algoritmo apresentado.

A característica de programação dinâmica do algoritmo envolve a ordem na qual os subconjuntos de $V(G)$ são percorridos. Temos que considerar uma ordenação onde para cada S todos os seus subconjuntos já tenham sido processados previamente. Essa ordem é atendida se todos os subgrafos de G com menos vértices que S já tiverem sido processados. Dessa forma, quando o algoritmo chegar na etapa de encontrar o mínimo entre $X[S]$ e $X[S - I] + 1$ os valores podem ser obtidos sem necessidade de reprocessar os cálculos.

Algoritmo 8: LAWLER(G)

Entrada: Grafo G
Saída: Número cromático $\chi(G)$

```
1 Seja X um vetor com 2^n posições
2 Cada posição de X é associada a um dos 2^n subgrafos induzidos G[S]
3 S ← ∅
4 X[S] ← 0
5 para Cada S ⊆ V(G), S ≠ ∅ faça
6 |   X[S] ← |V(G)|
7 |   para Todo conjunto independente maximal I ⊆ V(G[S]) faça
8 |   |   X[S] ← min{X[S], X[S − I] + 1}
9 devolve X[V(G)]
```

2.3.2 Algoritmo DSaturExato

Nessa seção vamos apresentar a versão exata do algoritmo heurístico guloso apresentado na Seção 2.1.2, o DSaturExato. Como já dito, Brélaz [4] mostrou em seu trabalho uma forma de usar o conceito de Grau de Saturação (2.1) para apresentar um algoritmo exato para o problema de coloração de grafos. Nessa seção vamos apresentar uma versão do algoritmo inspirada no que foi apresentado no DSatur apresentado no livro de Lewis [17] e na dissertação de Lima [19]. Para explicar onde a heurística de grau de saturação aparece, vamos antes falar da árvore de soluções usada em algoritmos exatos que usam *backtracking*.

O DSaturExato faz uso das técnicas de programação *branch-and-bound* e *backtracking*. O *backtracking* consiste em solucionar problemas de otimização combinatória através da enumeração das possíveis soluções, sendo que essa enumeração ocorre através da criação de uma árvore de soluções, onde as folhas da árvore são soluções viáveis e os nós internos são soluções parciais. A árvore de *backtracking* é construída usando busca em profundidade através de passos de ida (*forward*) e passos de volta (*backward*). Já o *branch-and-bound* consiste em explorar o espaço de possíveis soluções evitando que todas as possíveis soluções sejam percorridas. Ele gera uma árvore de busca onde a instância inicial é sucessivamente particionada em soluções menores, em um processo que é conhecido como etapa de ramificação (*branching*). No *branch-and-bound*, quando verificamos que as ramificações de um nó só levarão a soluções piores que a melhor solução encontrada até o momento, paramos a ramificação naquele nó, processo conhecido como poda (*bound*). Para realizar a etapa de ramificação é possível usar o *backtracking*.

Para o problema de coloração, a ideia de *backtracking* tem a ver com construir soluções passo a passo até chegar a uma coloração completa. O passo a passo citado é a coloração de vértice em vértice até chegar à solução citada, sendo que colorir um vértice é o movimento de ida. Com uma primeira solução construída é feito um movimento de volta, isso é, o último vértice colorido é descolorido e, caso faça sentido, ele é colorido com outra cor, mas outro vértice também pode ser descolorido em um outro movimento de volta. Veja que esses movimentos de ida e de volta permitem que soluções sejam construídas dependendo da ordem em que os vértices são percorridos. Para ilustrar melhor a estrutura de árvore construída por esse procedimento de *backtracking* veja a Figura 7, onde temos um exemplo de árvore de soluções. Cada nível da árvore refere-se à coloração de um vértice por uma cor. Ao chegarmos no último nível temos um nó onde todos os vértices estão coloridos. No momento não estamos focando na forma como a ordem de vértices ou as cores são definidas, estamos pensando apenas na ideia básica de um algoritmo de *backtracking*.

Já a aplicação do *branch-and-bound* no problema de coloração envolve, ao chegar em um determinado nó, isso é, na atribuição de uma determinada cor a um determinado vértice, ter certeza que qualquer solução final que parta desse nó vai ser pior do que a

melhor solução encontrada até o momento. Assim, evitamos passar por todas as soluções que uma árvore de soluções possa ter, fazendo uma poda de todos os nós que temos que certeza que nos levam a soluções piores que as já encontradas até o momento. Então, ao chegarmos em um determinado nó com solução parcial de 4 cores, por exemplo, sendo que já encontramos uma solução completa com 3 cores, podemos fazer o movimento de volta e desconsiderar as soluções que tinham esse nó como base.

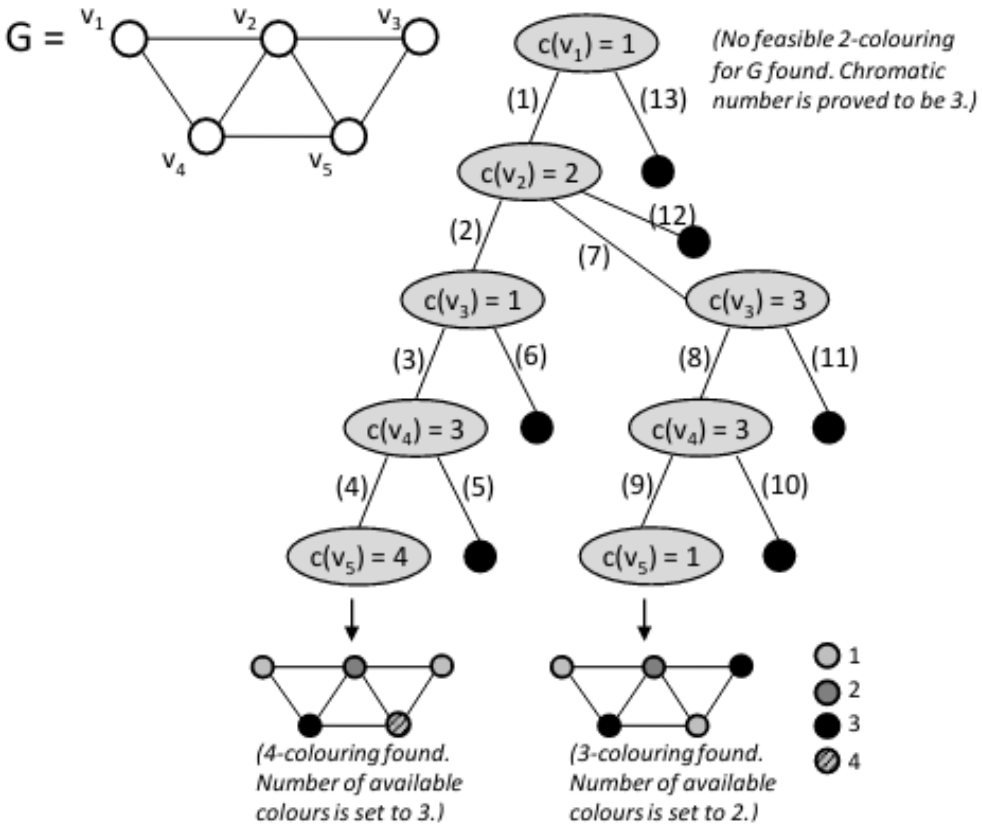

Figura 7 – Exemplo de coloração por *backtracking* que constrói árvore de soluções [17]. G é o grafo sendo colorido, $c(v_i)$ é a cor atribuída a cada vértice v_i e os número entre parênteses em cada ramo da árvore indica a ordem de ações na construção da árvore.

Agora, chegamos no ponto de definir a ordem na qual os vértices serão coloridos durante a exploração da árvore de soluções. Brélaz [4] propôs usar uma ordem dinâmica de vértices que fosse determinada pelo Grau Saturação (2.1) dos vértices na solução parcial construída até o momento. Assim, os níveis da árvore não estariam associados somente a um vértice, como no exemplo da Figura 7. O próximo vértice a ser colorido é sempre aquele com maior grau de saturação no momento.

Diferente dos outros algoritmos que vimos, aqui vamos usar dois pseudocódigos para representar o DSatur Exato em sua integralidade. O primeiro é o Algoritmo 9, o DSaturExato, que funciona como uma interface para iniciar a construção da árvore de soluções do problema. Agora, o DSaturRecursao é o pseudocódigo responsável

por fazer a construção de fato da árvore de soluções, de forma que cada execução sua representa um nó dessa árvore. O DSaturRecursao tem esse nome porque funciona através de chamadas recursivas, onde após um vértice ser colorido é feita uma chamada recursiva para colorir o próximo vértice da coloração sendo construída.

Algoritmo 9: DSaturExato(G)

Entrada: Grafo G
Saída: Partição dos vértices de G em uma coloração ótima
1 $melhor \leftarrow |V(G)|$
2 $coloracao \leftarrow \emptyset$
3 $resposta$ é uma coloração de G onde cada vértice possui uma cor
4 $\mathcal{S}, \chi \leftarrow$DSaturRecursao($G, cores, resposta, coloracao$)
5 **devolve** $\mathcal{S}$

O DSaturExato pode ser considerado um algoritmo de fácil compreensão. Podemos ver no Algoritmo 9 que recebido um grafo G, iniciamos as variáveis *melhor* com valor $|V(G)|$, *coloracao* como um conjunto vazio e *resposta* como uma coloração com o máximo de cores possível. Por fim, chamamos o DSaturRecursao e sua resposta é atribuída a $\mathcal{S}$, que é o retorno final desse algoritmo. A variável *melhor* é usada ao longo das chamadas recursivas e possui como valor a quantidade de cores da melhor solução completa encontrada até o momento. O valor de *melhor* é usado para cortar ramos da árvore de solução que levariam a soluções piores que a melhor já encontrada, o que na prática é evitar a chamada recursiva que levaria a um desses ramos. Já *coloracao* é a coloração parcial de G que está sendo construída durante a execução do DSaturRecursao do momento. A ideia é que, a cada chamada, o DSaturRecursao use o critério de grau de saturação para selecionar um vértice, atribua uma cor a esse vértice e faça a próxima chamada de DSaturRecursao se alguns critérios forem atendidos.

Agora, como podemos ver no Algoritmo 10, o DSaturRecursao é mais complexo, pois mistura uma parte do DSatur já apresentado com as ideias de recursão necessárias para construção da árvore de solução do *backtracking*. O primeiro passo do DSaturRecursao é construir o conjunto U (*Uncolored*) com os vértices de $V(G)$ ainda não coloridos em *coloracao*, na Linha 4. Esse conjunto U é usado para controle das tentativas de coloração daquela execução de DSaturRecursao. Isso é necessário porque uma execução de DSaturRecursao é um nó da árvore de soluções, então ele tem que ramificar as possibilidades de coloração dos vértices em U a partir da *coloracao* que foi passada como argumento da chamada. Então, se em uma chamada de DSaturRecursao é passada a *coloracao* com dois vértices de G ainda não coloridos, cabe ao algoritmo colorir esses dois vértices, mas cada um em uma coloração parcial diferente que será argumento da próxima chamada recursiva. Outra variável utilizada nesse controle é a T (Tentativas), que é inicializada como um conjunto vazio na Linha 5. Essa variável é usada dentro do primeiro laço **enquanto** do algoritmo para controlar as soluções parciais cujo ramo já foi

construído e "tentado". Então, a cada seleção de um vértice v usando grau de saturação, esse vértice é adicionado a T para demarcar que o ramo que considera *coloracao* e v já foi construído ou aparado.

Algoritmo 10: DSaturRecursao($G, cores, resposta, coloracao$)

Entrada: Grafo G, quantidade de cores da melhor solução completa até o momento *cores*, a melhor solução completa até o momento *resposta*, a coloração parcial construída até o momento *coloracao*

Saída: Partição dos vértices de G que é a melhor coloração até o momento e a quantidade de cores da melhor solução até o momento

```
1  U é o subconjunto dos vértices de V(G) não coloridos em coloracao
2  T ← ∅
3  enquanto |T| ≤ |U| faça
4      Seja v o vértice de U \ T com maior grau de saturação considerando coloracao
        ou aquele selecionado pelos critérios de desempate
5      T ← T ∪ {v}
6      S ← coloracao
7      j ← 1
8      encontrou ← Falso
9      enquanto encontrou = Falso e j ≤ |S| faça
10         se S_j ∪ {v} é conjunto independente então
11             S_j ← S_j ∪ {v}
12             encontrou ← Verdade
13         j ← j + 1
14     se encontrou = Falso e j > |S| então
15         S_j ← {v_i}
16         S ← S ∪ {S_j}
17     se S é coloração parcial com menos cores que a variável cores então
18         resposta_aux, cores_aux ← DSaturRecursao(G, cores, resposta, S)
19         se cores_aux < cores então
20             resposta ← resposta_aux
21             cores ← cores_aux
22     se S é coloração completa com menos cores que a variável cores então
23         devolve S, |S|
24 devolve resposta, cores
```

Entre as Linhas 6 e 16 temos um funcionamento quase igual ao do DSatur heurístico, mas com a diferença que o conjunto $\mathcal{S}$ tem seu valor "reiniciado" para *coloracao* em toda iteração. Usamos essa variável auxiliar $\mathcal{S}$ para não perdermos a referência da coloração parcial passada como parâmetro de entrada na chamada do algoritmo. Esse processo ocorre na Linha 6 e também visa fazer o controle da ramificação. Como dissemos, uma chamada de DSaturRecursao deve colorir um vértice por vez e passar para a próxima chamada a coloração parcial construída. O processo de colorir o vértice v escolhido ocorre dentro do segundo laço **enquanto**, que é igual ao do DSatur, passando por cada cor de $\mathcal{S}$ verificando se adicionar v a essa cor não faz $\mathcal{S}$ se tornar inviável.

Para finalizar a explicação do Algoritmo 10, temos as Linhas 17 e 22. Essas duas linhas fazem verificações referentes à solução construída $\mathcal{S}$ com o vértice v selecionado. Se a solução for parcial, isso é, se o v colorido não for o único vértice em U, então temos que realizar a próxima chamada recursiva para colorir os outros vértices até construir uma solução completa. A próxima chamada terá valor $\mathcal{S}$ para o parâmetro de *coloracao*, pois os próximos nós do ramo já devem considerar v colorido. Veja que além de verificar se é solução parcial, a Linha 17 também avalia se a quantidade de cores de $\mathcal{S}$ é menor que o valor de *cores*, a menor quantidade de cores que uma solução completa encontrou até aquele momento na execução. Essa verificação é feita como estratégia de *branch-and-bound*, pois poda os nós que temos certeza que vão gerar uma solução com número de cores igual ou maior que a melhor já encontrada. Agora, caso o v colorido seja o único vértice de U, então $\mathcal{S}$ é uma solução completa com todos os vértices coloridos. E caso essa solução completa $\mathcal{S}$ tenha quantidade de cores menor que o melhor encontrado até o momento, $\mathcal{S}$ é devolvida como resposta. Esse momento em que encontramos uma solução completa é o caso base da recursão. Caso a execução de DSaturRecursao não tenha encontrado nas iterações do primeiro laço **enquanto** nenhuma solução parcial ou completa melhor do que o já encontrado, ele devolve a solução e a quantidade de cores passadas como parâmetro na chamada.

É importante ressaltar que algoritmos exatos sempre vão possuir elevados tempos de execução por envolverem buscas de força bruta, e no caso do DSatur Exato não será diferente. A questão do Dsatur Exato é que existe uma heurística para selecionar a ordem dos vértices a serem coloridos que visa diminuir a quantidade de retrabalho de cálculo necessário usando cortes de ramos da árvore de solução. Considere que $n = |V(G)|$ e $m = |E(G)|$. Veja que o algoritmo DSaturExato possui tempo de execução constante para todas as suas linhas exceto, é claro, pela chamada a DSaturRecursao, que inicia construção da árvore de soluções. Focando agora em DSaturRecursao, temos o processo de coloração dos vértices que itera pelos vértices ainda não coloridos e pelas cores existentes, usando dois laços **enquanto**. Trata-se do mesmo processo de DSatur, exceto que sempre reiniciamos a coloração usando a coloração auxiliar $\mathcal{S}$, o que não muda o tempo de execução dessa parte que é $O(n^2 + m)$, onde o n^2 refere-se às iterações dos laços **enquanto** e m às verificações de conjunto independente. Já em relação à quantidade de chamadas de DSaturRecursao, sabemos que a árvore de soluções terá n níveis, um para cara vértice colorido durante a construção de soluções. Sobre a largura da árvore, veja que a raiz da árvore contém apenas um nó, o primeiro nível contém as n ramificações da raíz, o segundo nível as $n - 1$ ramificações de cada uma das n ramificações do primeiro nível, e assim por diante. Essa conta no final resulta em um tempo de execução que é $\sum_{i=0}^{n} n^i$, que no final é $O(n^{n+1})$. Então, o que temos na análise de tempo de execução são $O(n^{n+1})$ execuções do DSatur, ou seja, o tempo de execução de DSaturExato considerando as execuções de DSaturRecursao, no pior caso é $O(n^{n+3})$.

3 Implementação e Experimentação

Conforme falamos na Introdução, o principal objetivo do projeto é a construção de um pacote Python com a implementação de alguns algoritmos selecionados para o problema de coloração de grafos. Enquanto no Capítulo 1 apresentamos o problema de coloração de grafos em si, no Capítulo 2 apresentamos os algoritmos que foram implementados como parte do pacote. Assim, nos Capítulos 1 e 2 apresentamos as principais referências teóricas que nortearam o projeto. Ambos os capítulos foram fruto de uma extensa pesquisa bibliográfica que fundamentou a parte prática do projeto que apresentaremos nesse capítulo.

Nesse capítulo temos como objetivo apresentar as duas partes do projeto que são consideradas práticas. A primeira é a implementação em Python dos algoritmos apresentados no Capítulo 2. Vamos apresentar alguns detalhes técnicos do pacote, como as ferramentas usadas e a estrutura lógica, e também detalhes técnicos das implementações, em especial os parâmetros que devem ser usados para seu uso. Já a segunda parte é referente a testes realizados com esses algoritmos implementados, vamos dar mais detalhes sobre os métodos usados e a metodologia seguida, além de uma análise experimental dos resultados obtidos.

Na Seção 3.1 focaremos no desenvolvimento do pacote e sua estrutura técnica, abordando as principais ferramentas usadas e como elas foram usadas para atingirmos o resultado desejado. Já na Seção 3.2 falaremos das implementações dos algoritmos apresentados no Capítulo 2, não focando nas ideias gerais dos algoritmos que já foram apresentadas, mas sim apresentando os detalhes mais importantes da implementação. Essas duas seções cobrem as partes relacionadas à implementação.

Na Seção 3.3 será feito um resumo das referencias teóricas que consideramos na hora de desenhar o experimento. Como o experimento foi realizado é algo apresentado na Seção 3.4 que trará de forma detalhada ambiente, métodos e objetivos dessa experimentação. Dedicaremos a Seção 3.5 a apresentar os resultados obtidos após a realização do experimento e as análises dos mesmos.

3.1 Desenvolvimento do Pacote

Nessa seção pretendemos abordar os principais pontos técnicos do desenvolvimento do projeto. Vamos começar apresentando cada uma das ferramentas usadas, dando uma breve descrição envolvendo sua função e história. Apresentadas as ferramentas vamos passar para seu uso no projeto como um todo, onde pretendemos passar uma visão integrada de como essas ferramentas foram usadas para atingir nosso objetivo técnico e a razão de sua escolha. Também vamos nos aprofundar em alguns temas específicos como o uso de testes automatizados e ferramentas de *Continuos Integration/Continuos Development (CI/CD)*, conceito referente à automação de etapas de desenvolvimento para minimizar erros em etapas crucias.

3.1.1 Ferramentas

Python

Python[1] é uma linguagem de programação de alto nível, interpretada, multiparadigma, poderosa, com tipagem dinâmica e de fácil aprendizagem. Seu uso é gratuito e seu interpretador é facilmente obtido através de seu site oficial. Até o momento, sua última versão lançada foi a 3.11. Seu nome é inspirado no grupo humorístico britânico *Monty Python*.

Python Package Index (PyPi)

PyPi[2] é o repositório oficial de Pacotes de Distribuição de terceiros para a linguagem Python. Primariamente hospeda arquivos em formato *.wheel*. É considerado a fonte padrão dos instaladores de pacote do Python, como o Pip. Possui um site oficial onde usuários podem pesquisar sobre os pacotes disponíveis para instalação e suas diferentes versões.

Package Installer for Python (Pip)

É o instalador e gerenciador de pacotes de distribuição da linguagem Python mais popular, podendo ser usado para instalar e atualizar pacotes Python de fontes diferentes, incluindo o PyPi. Sua interface é feita através de linha de comando.

Pytest

Pytest[3] é um *framework* do Python com objetivo de facilitar a escrita de testes legíveis e escaláveis, sem deixar de ser capaz de executar testes complexos. É uma ferramenta

[1] Mais informações em https://www.python.org/
[2] Mais informações em https://pypi.org/
[3] Mais informações em https://docs.pytest.org/en/7.2.x/

de testes madura e popular entre os desenvolvedores Python. Sua forma de instalação mais comum é através do Pip.

Pipenv

Pipenv é um ferramenta de gerenciamento de ambientes virtuais e pacotes do Python. Com ela podemos criar ambientes virtuais com versões específicas de pacotes associadas a hashes e usar esses ambientes nos nossos desenvolvimentos. Isso evita problemas de conflito de versões na hora de desenvolver e facilita testes com diferentes combinações de pacotes sem prejudicar o ambiente Python geral do computador.

Poetry

Poetry[4] é uma ferramenta de desenvolvimento Python usada no controle do desenvolvimento, *build* e publicação de pacotes. Aqui falamos de *build* como a etapa técnica de construção do *.wheel* do pacote, o que se diferencia do desenvolvimento que é mais voltado ao código. Na prática o Poetry é uma ferramenta de linha de comando que facilita imensamente todas as etapadas do desenvolvimento de um pacote Python de forma integrada.

Igraph

O Igraph[5] é uma ferramenta de análise de redes com foco em eficiência e portabilidade. É uma ferramenta multilinguagem com uma distribuição como pacote Python, sendo esse o usado nesse projeto. Na prática atua como uma biblioteca com diversos algoritmos de grafos.

Git

Git[6] é um sistema de versionamento grátis e *open-source*, provavelmente o mais popular do mundo na categoria. É utilizável em projetos pequenos e grandes, sendo uma ferramenta de fácil aprendizagem e que contribui imensamente no desenvolvimento de projetos.

Github

O Github[7] é uma plataforma online para armazenamento de código fonte, arquivos de controle de versão, aplicação de práticas de *CI/CD*, armazenamento de pacotes, gerenciamento de projetos que dependam de código e também funciona como uma rede social com foco em desenvolvimento de código.

[4] Mais informações em https://python-poetry.org/
[5] Mais informações em https://igraph.org/
[6] Mais informações em https://git-scm.com/
[7] Mais informações em https://github.com/

Github Actions

O Github Actions[8] é um ferramenta do Github para facilitar o desenvolvimento e integração de etapas no desenvolvimento que vão além de escrever código, como *deploys*, testes, versionamento, lançamento de avisos, entre outras ações necessárias que possam ser automaticamente ativadas quando algum evento ocorrer.

3.1.2 Arquitetura da Solução

Como já comentado anteriormente no projeto, o pacote proposto foi desenvolvido em Python. Em específico, usamos o Python3 na versão 3.8. A escolha dessa linguagem de programação deve-se principalmente pela familiaridade do orientando com a linguagem e porque Python é uma linguagem com uma ampla comunidade e grande diversidade de ferramentas disponíveis através de pacotes. Dentro da linguagem Python o pacote fundamental para a construção do projeto foi o Igraph. Trata-se de uma ferramenta de análise de redes com distribuições para várias linguagens, como Python e R. Na prática o Igraph atua como um pacote Python com algoritmos e funcionalidades para análise de grafos e redes complexas. No nosso caso, o Igraph trouxe diversas comodidades por já ter implementado alguns algoritmos importantes que foram usados ao longo do projeto, como representação de grafos, verificação de conjuntos independentes, cálculo de subgrafos induzidos, entre outros. Claro que além do Igraph outros pacotes não nativos do Python também foram usados pontualmente, mas como nenhum foi tão relevante para o desenvolvimento do projeto vamos omitir sua descrição aqui.

Além do Python, no desenvolvimento local também usamos o Git para realizar o versionamento do código. Desenvolvemos o projeto usando duas *branches*: *develop* e *main*. A ideia por trás dessa escolha é que desenvolvimentos intermediários, como *checkpoints* durante a implementação de um algoritmo ou funcionalidade, seriam sempre *commitados* na *branch develop*. Enquanto isso, a *branch main* só seria atualizada quando uma nova versão do pacote final fosse lançada, e isso só aconteceria quando um algoritmo novo fosse totalmente implementado ou um erro identificado no código sofresse correção. Mesmo nesses casos essas alterações no código teriam que passar sempre pela *branch develop* e só depois passar para *main*. Para armazenar online de forma a suportar essa estrutura de versionamento do Git, usamos como repositório online de código um projeto no Github[9].

Construída uma versão do pacote, sua publicação envolve dois processos, que são a construção local do *.wheel* e a disponibilização da versão específica no repositório online PyPi. Para construir localmente o pacote usamos uma ferramenta do Python chamada Poetry que facilita imensamente o trabalho de construção e publicação. Para começar, com um simples comando dentro de uma pasta vazia, o Poetry constrói a estrutura de

[8] Mais informações em https://github.com/features/actions
[9] Disponível em https://github.com/WLAraujo/graphcol

pastas e os arquivos necessários para criação de um pacote publicável no Python, incluindo os arquivos *.lock* e *.toml*, responsáveis pelo hash de versões de dependência do pacote e as configurações básicas do pacote, respectivamente. Também com o Poetry é possível construir o *.wheel* do código desenvolvido e publicá-lo em um repositório de software. No nosso caso estamos realizando as publicações de versão do pacote no PyPi[10], assim, para instalar nosso pacote basta usar o comando '`pip install graphcol`'.

Por fim, duas outras ferramentas que usamos durante o desenvolvimento do projeto e estão mais relacionadas a boas práticas de desenvolvimento foram o Pytest e o Github Actions. O Pytest é talvez o *framework* de testes mais popular do Python, sendo amplamente usado para qualquer tipo de aplicação. Ele oferece uma forma simples de escrever os testes e os casos de validação, bem como fazer *assertions* e testar casos de sucesso e falha. No nosso projeto os testes realizados estão muito associados a avaliar se a solução devolvida pelos algoritmos era viável e se alguns teoremas que garantem resultados para certas classes de grafo são respeitados pelas nossas implementações. Já o Github Actions foi usado na automatização de alguns processos quando realizamos o *push* para o repositório remoto no Github ou quando realizamos um *Pull Request* (PR) da *branch develop* para a *main*. Então, o Github Actions ficou responsável por realizar algumas automatizações. Essas automatizações são baseadas em eventos que ocorrem com o repositório no Github, onde o código está armazenado. Os processos automatizados no projeto foram a execução de testes, sendo o evento de ativação a solicitação de PR, e a publicação de uma nova versão do pacote, com o evento de ativação sendo a aprovação do PR.

Para resumir tudo que foi descrito nessa subseção temos a Figura 8, que representa bem tanto o fluxo e os processos de desenvolvimento quanto as ferramentas usadas e suas posições na construção do pacote.

3.2 Implementações

Na seção anterior apresentamos como implementamos e desenvolvemos tecnicamente o produto final do projeto, que é um pacote com as implementações dos algoritmos disponíveis. Baseamos nossa explicação nos processos de desenvolvimento, nas ferramentas usadas e sua relação. Agora vamos focar em outra parte da dimensão técnica do projeto, que realmente fala da implementação dos algoritmos mostrados no Capítulo 2. Essa é uma parte chave do projeto, pois trata de um desafio técnico maior do que definir a arquitetura usada, afinal estamos falando de traduzir algoritmos com ideias e conceitos considerados sofisticados de seus pseudocódigos para códigos Python em uma estrutura que faça sentido para o usuário do pacote. É importante lembrar que um pacote Python tem o objetivo

[10] Disponível em https://pypi.org/project/graphcol/

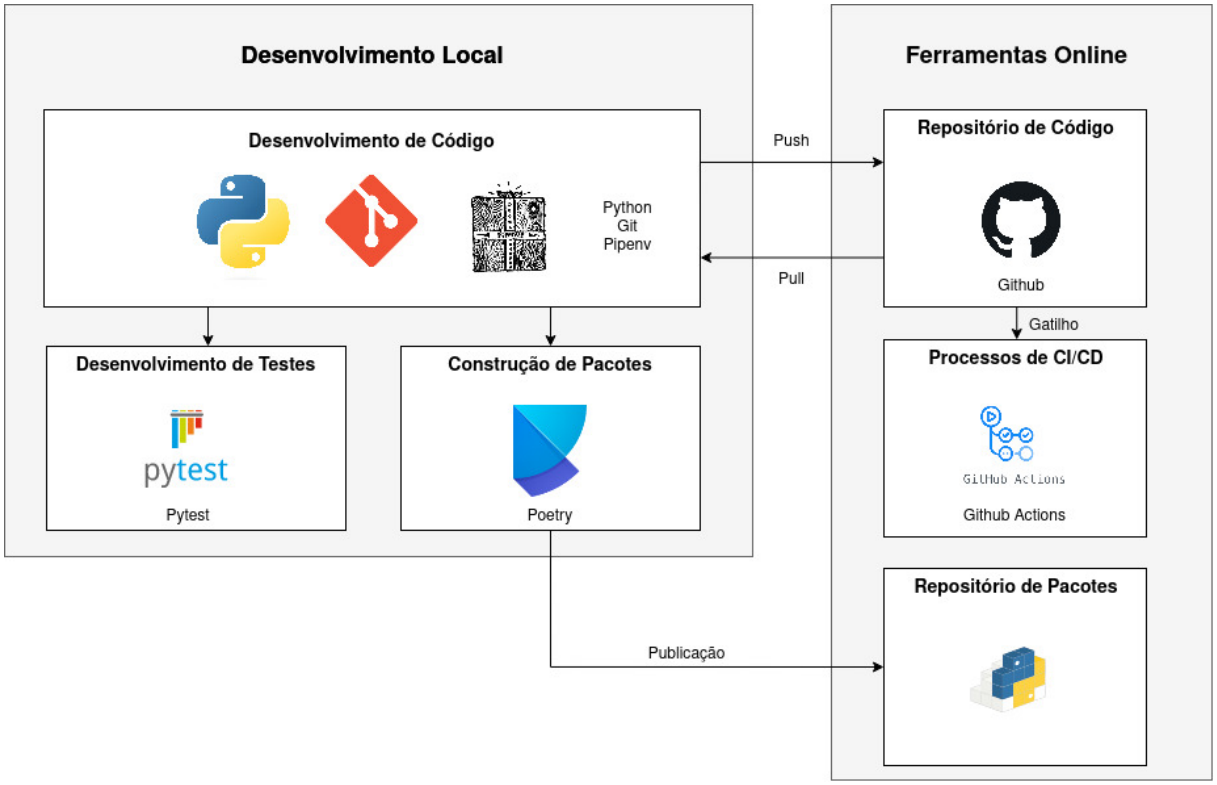

Figura 8 – Ferramentas usadas no projeto e a forma como interagem.

de ser usado, então nossas implementação não fazem sentido por si só, sendo também necessário falar de como usamos essas implementações no pacote.

Vamos começar apresentando a organização lógica do pacote, isso é, como as implementações estão organizadas, quais as divisões entre elas e quais são os métodos implementados. Depois disso vamos dar uma visão geral das implementações, focando especialmente nos parâmetros dos métodos. Também vamos apresentar exemplos de uso de cada uma das funções, com grafos de entrada gerados pelo método aleatório de Erdos-Reyni[11]. Visamos assim mostrar que o que está sendo implementado tem compatibilidade com o que é proposto nos algoritmos teóricos.

3.2.1 Organização Lógica

Como já discutimos, os algoritmos estudados podem ser divididos em três grupos: algoritmos gulosos; algoritmos meta-heurísticos; algoritmos exatos. Na nossa implementação levamos em conta essa divisão dos algoritmos para também dividi-los no pacote. O que fizemos foi criar três classes Python dentro do pacote, cada uma representando um desses grupos de algoritmos, assim, temos as classes `Gulosos`, `Metaheuristicas` e `Exatos`. Dentro de cada uma dessas classes temos funções referentes aos algoritmos apresentados no Capítulo 2. Na prática essas classes não possuem atributos, logo não faz sentido instanciá-las, sendo sua verdadeira utilidade a organização das funções que podem ser chamadas pelos

[11] https://python.igraph.org/en/stable/tutorials/erdos_renyi.html

usuários do pacote. Cada uma dessas classes está implementada em um arquivo diferente com mesmo nome das classes: `gulosos.py`, `metaheuristicas.py` e `exatos.py`. Essa é uma forma comum de organizar as funcionalidades de um pacote dentro da linguagem Python. Segue na Figura 9 um diagrama de como essas funções estão divididas dentro do pacote. Veja que trata-se da organização esperada, pois dividimos as implementações dos algoritmos por tipo de algoritmo usando as classes. Também vale ressaltar que o Python possui como recurso as *inner-functions*, que são funções definidas internamente a uma outra função, sendo assim somente utilizáveis dentro do escopo de execução da função principal que as engloba, ou seja, ocorre um encapsulamento da função dentro de um contexto. No projeto, todas as subrotinas dos algoritmos foram implementadas usando *inner-functions* e a forma como elas são usadas nas funções principais será apresentada nas próximas sessões.

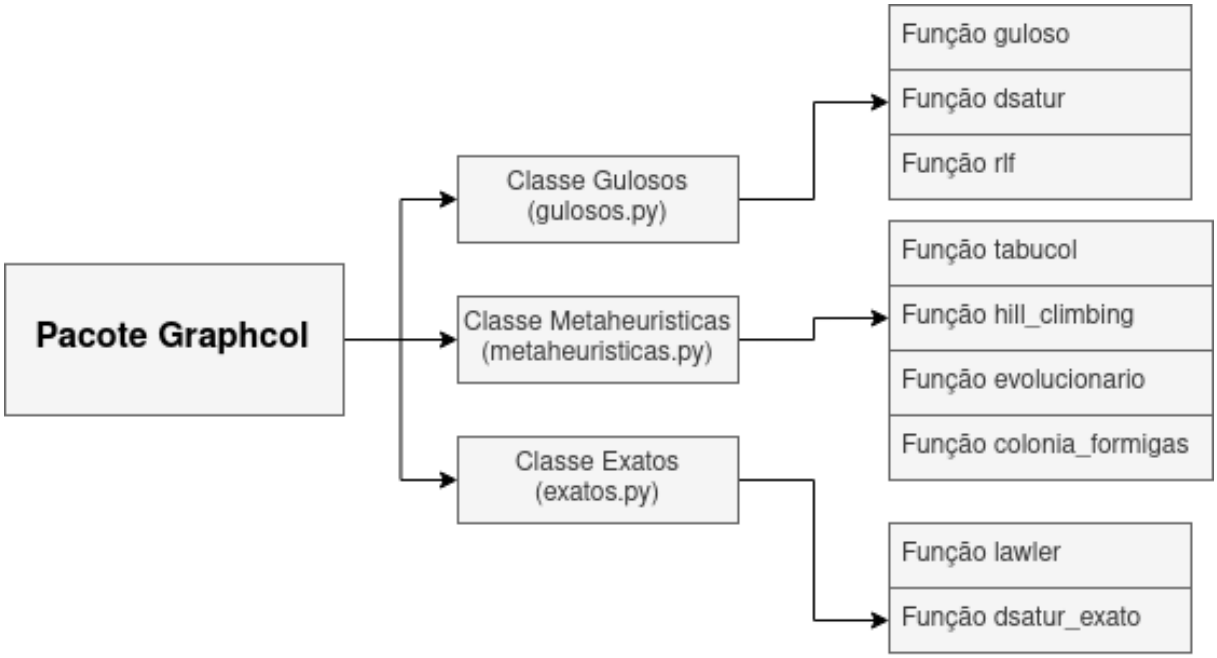

Figura 9 – Distribuição das funções dentro das classes.

Para deixar claro como essa organização impacta no uso das implementações temos um exemplo de uso da função `guloso` no código abaixo. Nesse primeiro momento o mais importante é entender a forma como fazemos a importação da classe `Gulosos` e como usamos a função `guloso`. Os parâmetros necessários para cada função serão mais bem detalhados mais à frente, mas podemos dizer que para todas as nossas funções um objeto `igraph.Graph()` deve ser passado na chamada, que é a representação de grafos nativa do pacote Igraph. Além disso, todas as nossas implementações devolvem o mesmo grafo passado na entrada, já que o pacote Igraph nos permite associar características a cada vértice de um objeto `igraph.Graph()`. Assim, devolvemos o mesmo grafo mas com o atributo `vs['cor']`, que é uma lista ordenada onde cada número da lista representa a cor do vértice na mesma posição, isso para todos as funções exceto a implementação do Algoritmo de Lawler, pois esse devolve o número cromático. Para acessar esse label use `grafo.vs["cor"]`.

```
from graphcol.gulosos import Gulosos
import igraph as ig

grafo = ig.Graph()
coloracao = Gulosos.guloso(grafo).vs["cor"]
```

3.2.2 Funções

Função `guloso`

A função `guloso(grafo, ordem = None)` é a implementação realizada do Algoritmo Guloso (1). Ela é disponibilizada no pacote GraphCol na classe `Gulosos`. Como **parâmetros de entrada** a função recebe:

- `grafo` (`igraph.Graph`) - Grafo a ser colorido,
- `ordem` (`list`) - Lista ordenada dos vértices do grafo.

Caso o parâmetro `ordem` não seja passado na chamada da função a própria cria uma lista com uma ordem aleatória dos vértices do grafo e usa essa ordem para colorir os vértices. Segue um exemplo de uso dessa função sobre um grafo aleatório com 5 vértices gerado pelo método Erdos-Renyi nativo do Igraph.

```
from graphcol.gulosos import Gulosos
import igraph as ig

grafo = ig.Graph().Erdos_Renyi(5, p=0.7)
coloracao = Gulosos.guloso(grafo, [1,2,0,4,3]).vs["cor"]
```

Função `dsatur`

A função `dsatur(grafo, v_inicial = None)` é a implementação realizada do Algoritmo DSatur (2). Ela é disponibilizada no pacote GraphCol na classe `Gulosos`. Como **parâmetros de entrada** a função recebe:

- `grafo` (`igraph.Graph`) - Grafo a ser colorido,
- `v_inicial` (`int`) - Inteiro associado a algum vértice do grafo.

Caso o parâmetro `v_inicial` não seja passado na chamada da função a própria define um vértice inicial para começar a coloração. Lembrando que isso é feito pois no início do algoritmo todos os vértices possuem Grau de Saturação igual a zero. Segue um exemplo de uso dessa função sobre um grafo aleatório com cinco vértices gerado pelo método Erdos-Renyi nativo do Igraph.

```
from graphcol.gulosos import Gulosos
import igraph as ig

grafo = ig.Graph().Erdos_Renyi(5, p=0.7)
coloracao = Gulosos.dsatur(grafo,3).vs["cor"]
```

Função `rlf`

A função `rlf(grafo)` é a implementação realizada do Algoritmo RLF (3). Ela é disponibilizada no pacote GraphCol na classe `Gulosos`. Como **parâmetros de entrada** a função recebe:

- `grafo` (`igraph.Graph`) - Grafo a ser colorido.

Veja que a função recebe apenas um parâmetro, o próprio grafo a ser colorido, ou seja, não há como o usuário influenciar na ordem em que a coloração será realizada, sendo um processo aleatório por parte do algoritmo. Segue um exemplo de uso dessa função sobre um grafo aleatório com 5 vértices gerado pelo método Erdos-Renyi nativo do Igraph.

```
from graphcol.gulosos import Gulosos
import igraph as ig

grafo = ig.Graph().Erdos_Renyi(5, p=0.7)
coloracao = Gulosos.rlf(grafo).vs["cor"]
```

Função `tabucol`

A função `tabucol(grafo, solucao_inicial=None, tabu=5, iteracoes_min=20, iteracoes_max=50, cores_max = None, iteracoes_s_mudanca = 10, msg_print = True)` é a implementação realizada do Algoritmo TabuCol (5). Ela é disponibilizada no pacote GraphCol na classe `Metaheuristicas`. Como **parâmetros de entrada** a função recebe:

- `grafo` (`igraph.Graph`) - Grafo a ser colorido
- `solucao_inicial` (`list`) - Lista de inteiros que contém solução inicial que será ponto de partida da busca tabu; caso nenhuma solução seja passada ele constrói uma solução inicial de partida de forma aleatória,
- `tabu` (`int`) - Valor tabu, quantidade de iterações que um movimento é mantido na lista tabu,
- `iteracoes_min` (`int`) - Número mínimo de iterações a serem realizadas antes que o algoritmo pare de executar,
- `iteracoes_max` (`int`) - Número máximo de iterações que o algoritmo executa antes de forçar a parada de execução,
- `cores_max` (`int`) - Número máximo de cores que serão usados na construção da solução; se vazio considera o número de vértices do grafo,
- `iteracoes_s_mudanca` (`int`) - Parâmetro de parada que estabelece quantas iterações o algoritmo pode executar sem melhora na solução; caso chegue a esse número de iterações ele é parado,
- `msg_print` (`bool`) - Valor booleano que indica se uma mensagem deve ser apresentada no caso do algoritmo não encontrar uma solução viável com os parâmetros passados.

Veja que a função recebe diversos parâmetros relacionados às restrições da busca. Trata-se da única função desenvolvida que pode não encontrar uma solução viável. Quando isso ocorre, ela imprime uma mensagem dizendo "Não foi possível encontrar solução viável com os parâmetros passados.". Segue um exemplo de uso dessa função sobre um grafo aleatório com 5 vértices gerado pelo método Erdos-Renyi nativo do Igraph.

```
from graphcol import gulosos
from graphcol.metaheuristicas import Metaheuristicas
import igraph as ig

grafo = ig.Graph().Erdos_Renyi(5, p=0.7)
coloracao = Metaheuristicas.tabucol(grafo).vs["cor"]
```

Função `hill_climbing`

A função `hill_climbing(grafo, divisao = 0.75, iteracoes_max=50, iteracoes_s_melhora = 10)` é a implementação do Algoritmo HILL-CLIMBING (4). Ela é disponibilizada no pacote GraphCol na classe `Metaheuristicas`. Como **parâmetros de entrada** a função recebe:

- `grafo` (`igraph.Graph`) - Grafo a ser colorido,
- `divisao` (`float`) - Número entre 0 e 1 que indica qual a divisão de cores inicial que será realizada; por exemplo, 0.75 indica que o algoritmo tentará transferir os vértices de 1/4 das cores existentes,
- `iteracoes_max` (`int`) - Número máximo de iterações que o algoritmo executa antes de forçar a parada de execução,
- `iteracoes_s_melhora` (`int`) - Parâmetro de parada que estabelece quantas iterações o algoritmo pode executar sem melhora na solução; caso chegue a esse número de iterações ele é parado.

Veja que a função recebe dois parâmetros relacionados às restrições da busca e um relacionado ao funcionamento da meta-heurística. Segue um exemplo de uso dessa função sobre um grafo aleatório com 5 vértices gerado pelo método Erdos-Renyi nativo do Igraph.

```
from graphcol import gulosos
from graphcol.metaheuristicas import Metaheuristicas
import igraph as ig

grafo = ig.Graph().Erdos_Renyi(5, p=0.7)
coloracao = Metaheuristicas.hill_climbing(grafo).vs["cor"]
```

Função `evolucionario`

A função `evolucionario(grafo, n_pop = 20, iteracoes_tuning = 20)` é a implementação do Algoritmo EVOLUCIONARIO (6). Ela é disponibilizada no pacote GraphCol na classe `Metaheuristicas`. Como **parâmetros de entrada** a função recebe:

- `grafo` (`igraph.Graph`) - Grafo a ser colorido,
- `n_pop` (`int`) - Quantidade de soluções que a população vai manter a cada geração,
- `iteracoes_tuning` (`int`) - Quantidade de iterações que o algoritmo vai realizar o processo de evolução.

Veja que a função recebe um parâmetro com o objetivo de restringir a busca e outro com a quantidade de soluções mantidas a cada geração do processo de evolução. Segue um exemplo de uso dessa função sobre um grafo aleatório com 5 vértices gerado pelo método Erdos-Renyi nativo do Igraph.

```
from graphcol import gulosos
from graphcol.metaheuristicas import Metaheuristicas
import igraph as ig

grafo = ig.Graph().Erdos_Renyi(5, p=0.7)
coloracao = Metaheuristicas.evolucionario(grafo).vs["cor"]
```

Função `colonia_formigas`

A função `colonia_formigas(grafo, n_formigas = 20, max_iteracoes = 20, alfa = 1, beta = 1, evaporacao = 0.75))` é a implementação do Algoritmo Ant-Col (7). Ela é disponibilizada no pacote GraphCol na classe `Metaheuristicas`. Como **parâmetros de entrada** a função recebe:

- `grafo` (`igraph.Graph`) - Grafo a ser colorido,
- `n_formigas` (`int`) - Representa a quantidade de formigas usadas em cada iteração, isso é, a quantidade de soluções construídas que contribuem com a construção da matriz de vértices por cor,
- `max_iteracoes` (`int`) - Número máximo de iterações que o algoritmo executa antes de forçar a parada de execução,
- `alfa` (`float`) - Hiperparâmetro que impacta na probabilidade de atribuir uma cor a um vértice,
- `beta` (`float`) - Hiperparâmetro que impacta na probabilidade de atribuir uma cor a um vértice,
- `evaporacao` (`float`) - Valor entre 0 e 1 que indica qual a perca do valor na matriz de vértice por cor para cada iteração.

Essa função possui alguns parâmetros diferentes das outras funções, que são o `alfa` e o `beta`, que influenciam em um fórmula específica e bem definida de probabilidade que é usada em uma atribuição de cor do algoritmo. Fora isso, os outros parâmetros estão mais associados a critérios de parada e quantidade de soluções consideradas na exploração. Segue um exemplo de uso dessa função sobre um grafo aleatório com 5 vértices gerado pelo método Erdos-Renyi nativo do Igraph.

```
from graphcol import gulosos
from graphcol.metaheuristicas import Metaheuristicas
import igraph as ig

grafo = ig.Graph().Erdos_Renyi(5, p=0.7)
coloracao = Metaheuristicas.colonia_formigas(grafo).vs["cor"]
```

Função `cromatico_lawler`

A função `cromatico_lawler(grafo)` é a implementação do Algoritmo Lawler (8). Ela é disponibilizada no pacote GraphCol na classe `Exatos`. Como **parâmetros de entrada** a função recebe apenas o parâmetro `grafo` (`igraph.Graph`), que é o grafo a ser colorido. Diferentemente dos demais algoritmos implementados, o Algoritmo de Lawler devolve apenas o número cromático, então a função `cromatico_lawler` devolve um inteiro e não um objeto grafo como as outras funções implementadas.

```
from graphcol.exatos import Exatos
from graphcol.exatos import Gulosos
import igraph as ig

grafo = ig.Graph().Erdos_Renyi(5, p=0.7)
num_cromatico = Exatos.cromatico_lawler(grafo)
```

Função `dsatur_exato`

A função `dsatur_exato(grafo)` é a implementação do Algoritmo DSaturExato (9). Ela é disponibilizada no pacote GraphCol na classe `Exatos`. Como **parâmetros de entrada** a função recebe apenas o parâmetro `grafo` (`igraph.Graph`), que é o grafo a ser colorido. Assim como os outros algoritmos que devolvem colorações, o `dsatur_exato` devolve um objeto `igraph.Graph` com a *label* cor para representar a coloração devolvida. Segue um exemplo de uso dessa função sobre um grafo aleatório com 5 vértices gerado pelo método Erdos-Renyi nativo do Igraph.

```
from graphcol.exatos import Exatos
from graphcol.exatos import Gulosos
import igraph as ig

grafo = ig.Graph().Erdos_Renyi(5, p=0.7)
coloracao = Exatos.dsatur_exato(grafo).vs["cor"]
```

3.3 Bases Metodológicas

Nessa seção vamos apresentar as bases metodológicas consideradas no projeto, na execução e na análise dos experimentos que visavam testar as implementações realizadas dos algoritmos apresentados. Como trata-se de analisar implementações de algoritmos, foi necessário entender em linhas gerais o que significa a análise experimental de algoritmos e quais seriam as boas práticas e até mesmo a razão da mesma existir. Para esse fim, usamos como referências principais os artigos "*A Theoretician's Guide to the Experimental Analysis of Algorithm*" [12] e "*Experimental Analysis of Algorithms*" [20], ambos com enfoque grande em fazer conexões e ao mesmo tempo diferenciar a análise teórica da experimental, especialmente para evidenciar qual seria a necessidade dessa última.

Para começar, entendemos que o **algoritmo** refere-se ao abstrato enquanto a **implementação** foca no físico, em uma relação que pode ser comparada ao que outras ciências tem entre os fenômenos estudados e os modelos desenvolvidos, só que na computação geralmente o algoritmo, o modelo, vem antes do fenômeno prático que é a implementação. Então, na computação estudamos o abstrato "perfeito" tendo como referência na vida real uma ferramenta de medida "imperfeita". Claro que estudar algoritmos apenas usando de análises de pior caso é algo que possui certa pureza devido ao rigor teórico-matemático que essa forma de estudo possui, e é muito claro que por causa dessa forma rigorosa de analisar algoritmos que vários avanços na computação foram feitos [12]. Porém, a análise teórica não conta toda a história quando temos que implementar um algoritmo para fins práticos e quando temos que analisar uma implementação.

Para entender o porquê alguém estudaria não os algoritmos mas as suas implementações de forma experimental, basta pensarmos que de um mesmo algoritmo várias implementações podem surgir e entender, avaliar e definir qual a "melhor" segundo alguma métrica é algo que possui valor. Especialmente quando avaliamos heurísticas para problemas $\mathcal{NP}$-completos, a análise experimental das implementações se mostra uma ferramenta poderosa. Afinal pode até ser que esses problemas possuam algoritmos eficientes que os resolvam, mas até encontrarmos esses algoritmos talvez nossa melhor alternativa sejam essas heurísticas, então avaliar de maneira científica as implementações é algo importante e relevante no campo da computação.

Johnson [12] dá uma série de sugestões baseadas em armadilhas a serem evitadas e boas práticas a serem seguidas quando vamos realizar análises experimentais de implementações de algoritmos e escrever artigos sobre esses. Seguem as dez principais sugestões que ele dá em seu artigo e que tentamos seguir na realização de experimentos e suas análises com nossas implementações:

1. **Performar experimentos interessantes** - Qual o interesse em analisar experimentalmente o problema teórico com esses algoritmos?
2. **Avaliar o que já existe na literatura** - Com base nos interesses em realizar o experimento o que já existe de análise experimental para o problema?
3. **Usar instâncias que levem a conclusões generalistas** - Quais instâncias são mais interessantes para seus objetivos, um *benchmark* já existente ou instâncias aleatórias?
4. **Usar modelos experimentais eficientes** - Como fazer com que seu experimento seja realizado de forma eficiente e que não prejudique os resultados obtidos?
5. **Usar implementações eficientes** - As implementações do algoritmo estudado e dos algoritmos auxiliares não vão prejudicar o tempo de execução e os seus resultados?
6. **Garantir reprodutibilidade** - Todo o ambiente e outras dependências necessárias usadas no seu experimento estão claras o suficiente a ponto de ser algo reproduzível?
7. **Garantir comparabilidade** - Seu trabalho pode ser usado por outras pessoas para fins de comparação?
8. **Contar a história inteira** - Você está mostrando o panorama inteiro do seu experimento e não somente um corte específico?
9. **Fazer conclusões bem justificadas** - Todas as conclusões e considerações levantadas são bem justificadas pelos dados obtidos?
10. **Apresentar os dados de maneira informativa** - Quem ler o seu trabalho conseguiria chegar às mesmas conclusões que você?

3.4 Experimento

Na Seção 3.3 apresentamos as principais referências que usamos para nos informar sobre a análise experimental de algoritmos. Com base nas informações e ideias dessas fontes foi desenhado um experimento para avaliar de maneira prática os algoritmos implementados no pacote desenvolvido para o projeto. Assim, nessa seção vamos apresentar o desenho experimental realizado nos baseando especialmente nos princípios apresentados na seção anterior. A ideia é detalhar ao máximo como os experimentos foram realizados e o porquê de certas decisões terem sido tomadas durante sua execução, assim, garantindo reprodutibilidade.

Para começar, vamos falar dos objetivos e das razões de realizar uma análise experimental com os algoritmos implementados. De maneira objetiva, enquanto trabalho final de graduação é importante que aqui sejam cobertos diversos aspectos do tema estudado. E quando falamos de implementar algoritmos para problemas $\mathcal{NP}$-completos a análise experimental é um fator relevante e interessante de ser estudado e aplicado. Os resultados da análise experimental podem levar a descobertas interessantes sobre as implementações realizadas quanto à eficiência. Além disso, no contexto já citado a própria realização de uma análise experimental é algo importante.

Com isso em mente, vamos falar agora de objetivos mais práticos para o experimento. Nosso objetivo aqui é realizar uma comparação entre os próprios algoritmos implementados no pacote, focando principalmente em duas métricas: tempo de execução e qualidade de solução. Quanto ao tempo, estamos nos referindo ao tempo entre a chamada de alguma das funções implementadas e a devolução da resposta dentro da função. Já em relação à qualidade, estamos interessados em avaliar quais dos algoritmos devolvem soluções com valor mais próximo do valor ótimo.

Nosso foco aqui será mensurar a execução dos algoritmos heurísticos gulosos (Guloso, *DSatur*, *RLF*) e dos algoritmos meta-heurísticos (*Hill-Climbing*, Colonia, *TabuCol*, Evolucionario). Essa foi uma decisão tomada durante o desenho do experimento por um interesse do aluno em focar na análise de algoritmos heurísticos.

Para os testes decidimos usar como *benchmark* para o experimento instâncias selecionadas do conjunto de instâncias para o problema de coloração de grafos do segundo desafio de implementações do DIMACS[12]. Essa escolha foi feita pensando em aproveitar instâncias conhecidas daqueles que investigam o problema de coloração de grafos e para basear menos o experimento em instâncias aleatórias que possam não dar tanta clareza quanto sua estrutura. Foram escolhidas instâncias que respeitassem as seguintes regras: não possuir mais de três mil e quinhentas arestas e possuir número cromático já descoberto.

A regra referente ao número de arestas foi imposta devido ao equipamento usado na execução dos testes, que foi o computador pessoal do aluno. Essa escolha foi feita por ser um ambiente mais controlável pelo aluno durante a execução dos testes do que as máquinas do laboratório de pesquisa que, quando foram usadas em uma primeira tentativa de realizar o experimento, apresentaram algumas problemas na execução dos testes que não puderam ser contornados com tanta velocidade quanto poderiam se o aluno tivesse um acesso mais fácil a elas.

Já sobre a regra do número cromático, é uma decisão que foi tomada pensando em uma das métricas que queríamos analisar, que é a distância para o ótimo. Porém, é importante ressaltar que na literatura de análises experimentais de algoritmos esse

[12] Disponível em https://mat.tepper.cmu.edu/COLOR/instances.html

não necessariamente é um fator decisivo para usar instâncias. Afinal, sem esse valor podem ser feitas análises comparativas entre algoritmos de forma mais competitiva e objetiva comparando qual algoritmo devolve a solução com o menor número de cores sem necessariamente saber se é o ótimo ou não.

Dentre as instâncias com as características citadas foram escolhidas as listadas na Tabela 2 que também contém outras características de cada instância, como vértices, arestas e quantidade de cores da solução ótima. Outra coisa sobre as instâncias é que elas podem ser agrupadas segundo sua origem e forma de construção. Segue a descrição dos grupos segundo o site[12] que contém as instâncias:

- **Grafos de livros** - Dado um livro, um grafo é criado, onde cada vértice é um personagem e dois vértices possuem uma aresta entre si se os personagens encontram-se no livro. São desse grupo as instâncias anna, david, homer, huck e jean.
- **Grafos de milhas (*miles*)** - Para esses grafos, cada vértice representa uma lista de cidades dos Estados Unidos e existe aresta entre os vértices se as cidades estão a uma certa distância em milhas. O nome da instância indica qual a distância considerada na construção do grafo. Essas instâncias são miles250, miles500, miles750 e miles1000.
- **Grafos de rainhas (*queens*)** - Dado um tabuleiro n por m, cada casa do tabuleiro é um vértice e se duas casas se conectarem horizontalmente, verticalmente ou diagonalmente existe aresta entre eles. No nome das instâncias são dados os valores de n e m. Elas são queen5_5, queen6_6, queen7_7, queen8_12, queen8_8 e queen9_9.
- **Grafos de Mycielski** - Grafos baseados na transformação Mycielskiana. Dado um grafo G e sua transformação Mycielskiana $L(G)$ vale que $\chi(G) + 1 = \chi(L(G))$, além disso $L(G)$ não possui triângulos [21]. Nas instâncias usadas o número no nome indica o número cromático antes da transformação Mycielskiana. Essas instâncias são myciel2, myciel3, myciel4, myciel5 e myciel6.

Como forma de executar os experimentos e coletar os resultados de maneira automática foi criado um *script* em Python que itera sobre as instâncias escolhidas uma a uma, executando todos os algoritmos, fazendo as variações de parâmetros necessárias e salvando os resultados em um csv[13]. A primeira coisa que o *script* faz é ler arquivos de texto com as informações das instâncias e criar um objeto `igraph.graph` com as mesmas propriedades. Para cada instância o *script* executa um mesmo algoritmo com determinados parâmetros dez vezes, o que busca evitar vieses que possam surgir em caso de poucas execuções de um mesmo algoritmo. No caso das meta-heurísticas onde há variação de parâmetros ele executa uma vez para cada combinação possível de parâmetros dentro da

[13] Disponível em https://github.com/WLAraujo/graphcol/blob/main/experimentos/testes_gulosos_metaheuristicos.py

Tabela 2 – Tabela com instâncias usadas no experimento e características de cada uma.

Instância	**Vértices**	**Arestas**	**Solução Ótima**
anna	138	493	11
david	87	406	11
games120	120	638	9
huck	74	301	11
jean	80	254	10
miles250	128	387	8
myciel2	5	5	3
myciel3	11	20	4
myciel4	23	71	5
myciel5	47	236	6
myciel6	95	755	7
queen5_5	25	160	5
queen6_6	36	290	7
queen7_7	49	476	7
queen8_12	96	1368	12
queen8_8	64	728	9
queen9_9	81	1056	10
miles500	128	1170	20
miles750	128	2113	31
miles1000	128	3216	42

lista definida. Na Tabela 3 temos os parâmetros e os valores que foram variados durante a execução do experimento. Veja que para cada meta-heurística foram escolhidos dois parâmetros para serem variados. Além disso, para cada um desses parâmetros foram estabelecidos quatro possíveis valores, totalizando dezesseis execuções de cada meta-heurística por iteração do *script*. Resumindo, para cada uma das 20 instâncias o *script* executou dez vezes a seguinte sequência:

- Uma vez o algoritmo *guloso*;
- Uma vez o algoritmo *DSatur*;
- Uma vez o algoritmo *RLF*;
- Dezesseis vezes o algoritmo *hill-climbing*;
- Dezesseis vezes o algoritmo *tabucol*;
- Dezesseis vezes o algoritmo *evolucionario*;
- Dezesseis vezes o algoritmo *colonia*.

Então, os resultados são 600 soluções geradas pelos testes dos algoritmos gulosos e 12800 soluções geradas pelos algoritmos meta-heurísticos. A coleta dos resultados foi

Tabela 3 – Tabela com variações de parâmetros das meta-heurísticas avaliadas.

Algoritmo	**Parâmetros Variados**	**Valores Possíveis**
tabucol	tabu, iteracoes_s_mudanca	(3,5,7,9), (8,10,12,14)
hill_climbing	div, iteracoes_max	(15,40,65,90), (40,45,50,55)
colonia	formigas, evolucionario	(10,15,20,25), (15,40,65,90)
evolucionario	populacao, geracoes	(10,15,20,15), (10,15,20,25)

feita de forma que cada solução pudesse ser coletado de forma tabular, assim, cada linha do csv construído deveria conter todas as informações importantes sobre a execução de um teste. Na verdade, dois csvs foram construídos durante os testes, o primeiro com os resultados dos testes dos algoritmos gulosos e o segundo com os resultados dos testes dos algoritmos meta-heurísticos. Essa separação foi necessária pois são necessárias menos informações para identificar uma execução de um algoritmo guloso, já que os algoritmos meta-heurísticos também apresentam as variações de parâmetros. No fim, cada linha deve conter a instância, sua quantidade de vértices, sua quantidade de arestas, a quantidade de cores da solução devolvida pela implementação, o início da execução e o fim da execução.

Para executar o código foi usada a máquina pessoal do aluno, um notebook Acer Aspire A515-52G com processador Intel Core i5 de oitava geração e 16GB de RAM. É uma máquina comum, o que dificulta a busca por performance, mas simula bem um ambiente de uso do pacote Python desenvolvido. Os testes dos algoritmos foram executados com a versão 0.3.2 do pacote *Graphcol*[14] com a máquina unicamente sendo usada para executar os algoritmos. Outro detalhe importante do ambiente é que o *script* do experimento foi executado dentro de um ambiente virtual python (venv) usando como base o arquivo de `requirements.txt` disponível na pasta de experimentos do repositório Github do projeto[15]. A versão do Python usada nos testes foi a 3.8, a mesma usada no desenvolvimento das implementações. O sistema operacional utilizado foi o Linux-Debian versão 5.10.

[14] Disponível em https://pypi.org/project/graphcol/0.3.2/
[15] Disponível em https://github.com/WLAraujo/graphcol/tree/main/experimentos

3.5 Resultados e Análises

Após a realização dos experimentos, temos os dados e nessa seção iremos apresentá-los e analisá-los. Esses resultados, em um primeiro momento, foram coletados e armazenados no formato de tabelas como discutido na Seção 3.4, e para apresentá-los da forma que desejamos será necessário usar ferramentas de análise de dados. Aqui usamos os pacotes Python Pandas e Seaborn. Pandas é um pacote de para tratamento e transformação de dados, especialmente para dados em formato tabular, permitindo cruzamento, construção de novas colunas e integração com diversos outros recursos. Seaborn é uma ferramenta de visualização de dados que permite criar visuais informativos e simples. Como ambiente de execução dessas análises usamos o *colab*, uma ferramenta de computação em nuvem da Google que permite executar notebooks Python. O notebook e os dados usados no experimento estão disponíveis também no repositório do pacote[16]. Nas visualizações usaremos como rótulo dos algoritmos nos gráficos os valores registrados durante os teste:

- Algoritmo DSatur - dsatur;
- Algoritmo RLF - rlf;
- Algoritmo Guloso - guloso;
- Algoritmo Hill-Climbing - hill climbing;
- Algoritmo Evolucionario - evolucionario;
- Algoritmo AntCol - colonia formigas;
- Algoritmo TabuCol - tabucol.

O principal objetivo dessa seção é realizar uma análise exploratória dos resultados do experimento feito. Vamos apresentar nessa seção uma série de tabelas e gráficos que tragam informações consolidadas sobre os resultados, facilitando as análises que tenham interesse em compreender como certas características dos algoritmos influenciaram nas métricas que estão sendo avaliadas. A seção está dividida em três partes. Na Seção 3.5.1 começamos as análises com um panorama geral dos resultados, o que permite uma comparação mais direta de todos os algoritmos com base nos valores deles para as duas métricas de interesse. Na Seção 3.5.2 vamos apresentar uma análise dos resultados obtidos para os experimentos referentes aos algoritmos gulosos. Já na Seção 3.5.3 vamos repetir o processo de análise mas para os algoritmos meta-heurísticos. Fechamos as análises com a Seção 3.5.4, onde apresentamos as conclusões com base nas análises realizadas.

No geral, para os dois grupos de algoritmos analisados, vamos focar nossas atenções no comportamento de duas métricas: diferença para solução ótima e tempo de execução. A

[16] Disponível em https://github.com/WLAraujo/graphcol/tree/main/experimentos

diferença para a solução ótima indica a "qualidade da solução", pois analisando esse valor podemos encontrar tendências sobre quão distantes do valor ótimo as soluções devolvidas estão para as instâncias estudadas. Assim, quanto menor os valores dessa métrica melhor a solução. Já o tempo de execução é a métrica clássica a ser analisada, pois nos dá uma boa noção sobre a qualidade da implementação e a viabilidade de sua aplicação a instâncias maiores. Para analisar de forma mais detalhada essas duas métricas vamos focar em abrir os resultados por algoritmo e por família de instância.

É importante entender os objetivos e limitações dessas análises. Afinal, estamos falando de um Projeto de Graduação em Computação que tem muito mais interesse em dar experiência para o aluno em implementações de algoritmos de otimização e na análise dessas implementações, do que necessariamente buscar implementações que sejam o estado da arte. Assim, nossas análises serão muito contidas no que foi apresentado até o presente momento nesse trabalho.

3.5.1 Resultados Gerais

Para começar nossas análises vamos fazer uma apresentação e análise dos resultados gerais por algoritmo e instância antes de focar com mais detalhes em cada grupo de algoritmos. Primeiro, na Tabela 4 temos a apresentação dos resultados por algoritmo, apresentando uma média das duas métricas que estamos avaliando: a diferença para o resultado ótimo e o tempo de execução.

Tabela 4 – Tabela resumo com valores médios das métricas por algoritmo.

	Diferença Resultado	**Tempo de Execucação (ms)**
Algoritmo		
AntCol	5,852	170395,428
DSatur	1,105	190,045
Evolucionario	2,084	3276,324
Guloso	1,540	269,935
Hill-Climbing	0,996	5730,052
RLF	1,625	0,105
Tabucol	36,105	33,167

Veja que, para a métrica "Diferença de Resultado", quatro algoritmos obtiveram em média resultados abaixo de 2, sendo que só um deles obteve uma média abaixo de 1, o Hill-Climbing. Os outros três algoritmos com média abaixo de 2 foram os algoritmos gulosos que apresentaram valores entre 1 e 2. Já o Evolucionário obteve uma média com valor bem próximo de 2. Por fim, o Antcol e o Tabucol obtiveram valores médios muito altos, sendo que o Tabucol especialmente obteve um resultado médio que dificilmente poderia ser comparado aos dos outros algoritmos. Mesmo o Antcol que possui a segunda maior média de "Diferença de Resultado'" possui valor aproximadamente 6 vezes menor

que o do Tabucol. Então, dentro dos algoritmos meta-heurísticos podemos dizer que só o Hill-Climbing e o Evolucionario mostraram resultados equiparáveis aos dos gulosos, enquanto o Antcol e o Tabucol se mostraram muito distantes dos outros resultados e de soluções que se aproximavam do ótimo.

Já para a métrica "Tempo de Execução", mensurada em milissegundos, vemos que o RLF se destaca bastante em relação aos outros algoritmos por ter em média um tempo de execução menor que 1ms. Outro que se destaca é o Tabucol, que também tem um tempo de execução consideravelmente inferior aos outros algoritmos, tendo média de 33ms. Se o RLF e o Tabucol destacam-se por um tempo médio baixo, o Antcol destaca-se por ter um tempo médio de execução elevado, tendo uma outra escala em relação aos outros algoritmos meta-heurísticos, aproximadamente 60 vezes o valor médio do tempo de execução do evolucionário. Tirando esses três algoritmos, os outros seguem o que havia se pensado sobre os tempos de execução. Os tempos de execução dos algoritmos gulosos DSatur e Guloso são próximos entre si, assim como os tempos de execução de Evolucionario e Hill-Climbing também podem ser considerados próximos por estarem na mesma escala de casa de milhar.

Agora, vamos também apresentar esses valores gerais cruzando as informações de instâncias e algoritmos. Segue na Tabela 5 um cruzamento que mostra os valores médios da métrica "Diferença para o Ótimo" para cada algoritmo quando executado para cada instância. Já na Tabela 6 temos um cruzamento semelhante mas os valores mostrados são referentes à métrica "Tempo de Execução", medida em milissegundos. É importante destacar que para ambas as tabelas estamos realizando um arredondamento e um corte na primeira casa decimal para facilitar a visualização dos resultados. Então, para os valores que forem apresentados como 0.0 na tabela de "Tempo de Execução", considere que a escala da média só não é considerável nessa análise por estar muito abaixo dos milissegundos e não que o tempo de execução foi nulo, o que seria impossível. O mesmo vale para a tabela de "Diferença para o Ótimo".

Sobre os resultados nessas tabelas, podemos ver que os algoritmos que mais destacam-se na "Diferença para o Ótimo" são o DSatur, o RLF, o Guloso e o Hill-Climbing, apresentando para uma quantidade razoável de instâncias valor 0.0. Outra coisa destacável é que esses algoritmos alternam entre ser o melhor para alguns grupos de instância. Por exemplo, veja que os valores de DSatur para as instâncias *book* são todos 0.0, mas para as instâncias *queen* o Hill-Climbing apresenta um desempenho melhor. Já em relação ao "Tempo de Execução" podemos perceber algo que já era esperado, os algoritmos gulosos apresentam tempos de execução consideravelmente menores que o dos algoritmos meta-heurísticos, exceto pelo Tabucol, que em muitas instâncias mostrou um tempo médio menor que o DSatur e o Guloso.

Tabela 5 – Valores médios de diferença para ótimo por instância e algoritmo.

Algoritmo	AntCol	DSatur	Evolu-cionario	Guloso	Hill Climbing	RLF	Tabucol
Instancia							
anna	4.9	**0.0**	1.4	0.2	0.1	0.6	72.1
david	4.1	**0.0**	1.8	0.9	0.3	0.7	40.8
games120	10.2	**0.0**	3.7	**0.0**	**0.0**	0.1	63.4
huck	2.8	**0.0**	2.1	**0.0**	**0.0**	**0.0**	32.4
jean	3.1	**0.0**	1.9	0.2	0.1	0.3	36.7
miles1000	6.5	0.7	2.4	3.3	2.1	4.0	45.5
miles250	8.4	0.1	2.3	1.6	0.7	1.5	68.6
miles500	6.9	0.4	2.2	2.0	1.4	2.3	59.8
miles750	6.5	0.2	2.5	2.6	2.2	3.6	52.4
myciel2	**0.0**	**0.0**	0.1	**0.0**	**0.0**	**0.0**	0.8
myciel3	0.3	**0.0**	0.3	**0.0**	**0.0**	**0.0**	2.2
myciel4	1.4	**0.0**	0.5	0.2	**0.0**	0.1	6.9
myciel5	4.1	**0.0**	0.8	**0.0**	**0.0**	0.3	20.3
myciel6	9.1	0.1	1.2	0.2	**0.0**	0.2	50.7
queen5_5	4.5	1.5	1.9	2.2	1.2	2.4	10.1
queen6_6	5.0	3.0	4.0	2.3	1.5	2.2	14.6
queen7_7	7.9	4.2	5.9	4.3	2.9	3.9	23.3
queen8_12	11.6	3.7	7.7	3.4	2.1	2.9	49.1

Tabela 6 – Valores médios de tempo de execução (ms) por instância e algoritmo.

Algoritmo	AntCol	DSatur	Evolu-cionario	Guloso	Hill Climbing	RLF	Tabucol
Instancia							
anna	353639.1	861.8	13013.1	1371.6	20365.5	**0.0**	27.5
david	96489.8	93.1	1790.5	158.3	3105.872	**0.0**	18.942
games120	372628.9	309.9	5798.1	399.0	7003.9	**0.0**	30.7
huck	59962.2	40.9	692.2	69.6	1196.5	**0.0**	15.1
jean	70259.6	50.9	977.4	96.7	1505.5	**0.0**	12.6
miles1000	554604.1	570.5	9855.9	827.6	23179.3	1.0	176.7
miles250	349647.7	314.7	5619.8	416.8	7860.1	**0.0**	20.0
miles500	531907.2	389.5	7342.3	592.2	15364.4	0.1	54.4
miles750	587470.4	470.1	7810.3	668.7	18456.6	1.0	96.2
myciel2	47.9	**0.0**	1.1	**0.0**	0.7	**0.0**	0.425
myciel3	251.6	**0.0**	2.9	**0.0**	2.1	**0.0**	1.1
myciel4	1680.3	0.7	19.7	0.1	17.4	**0.0**	3.2
myciel5	13513.6	17.5	293.9	18.0	285.6	**0.0**	10.9
myciel6	126695.7	285.0	4906.6	349.1	5141.5	**0.0**	32.6
queen5_5	1347.2	1.3	30.9	1.1	39.5	**0.0**	6.6
queen6_6	4367.9	7.5	110.8	5.3	147.8	**0.0**	12.4
queen7_7	12210.1	19.0	330.6	19.2	458.7	**0.0**	17.9

3.5.2 Análise dos Algoritmos Gulosos

Agora, vamos aprofundar nossas análises para os resultados dos algoritmos gulosos implementados. O primeira gráfico que vamos analisar é o da Figura 10, que mostra a

distribuição de quantidade de soluções devolvidas nos testes pela diferença de cores na solução para o valor ótimo da instância. Vemos que em 292 soluções devolvidas, pouco menos da metade, foram uma coloração ótima, indicado na coluna 0. Para as colunas entre 1 e 4 temos uma distribuição muito equilibrada de quantidades de soluções, embora em quantidade muito menor do que para a coluna 0. Já para as colunas 5 e 6 tivemos pouquíssimas soluções. Então, para as instâncias avaliadas, nossas implementações de algoritmos gulosos devolveram soluções que em geral não se afastam muito da solução ótima.

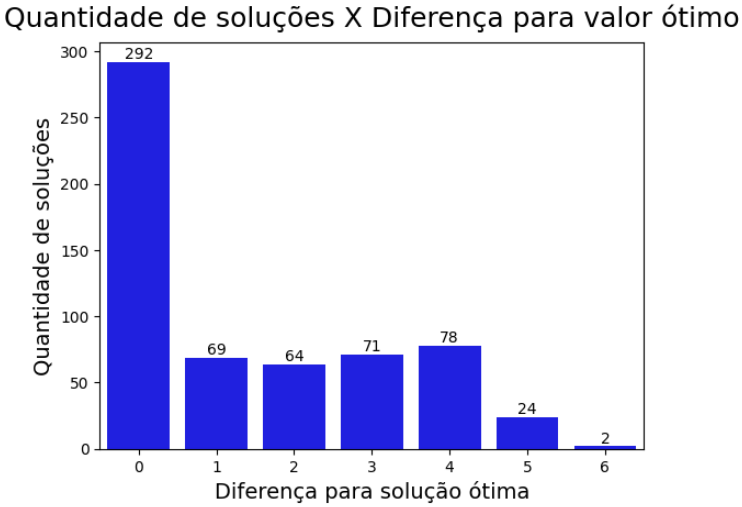

Figura 10 – Gráfico que mostra a quantidade de soluções que os algoritmos gulosos devolveram por diferença da solução ótima.

Os resultados apresentados na Figura 10 são expandidos por algoritmo implementado e testado na Figura 11. Veja que os algoritmos Guloso e RLF apresentaram distribuições mais semelhantes aos resultados gerais, isso é, por mais que tenham mais soluções ótimas ainda temos uma quantidade relevante de soluções com diferença para solução ótima entre 1 e 4. Já o algoritmo DSatur apresenta um desequilíbrio maior na distribuição, apresentando uma quantidade bem maior de soluções devolvidas que são ótimas em comparação às não ótimas, sendo inclusive o único algoritmo onde mais de 50% das soluções foram ótimas. Então, para as instâncias avaliadas o DSatur se mostrou um algoritmo com maior tendência em devolver soluções ótimas. Lembrando que para cada algoritmo foram executados 200 testes.

Além de avaliar a diferença da solução ótima por algoritmo também é nosso objetivo avaliar essa métrica para cada família de instâncias usada nos testes. Na Figura 12 mostramos a distribuição de soluções por diferença para a solução ótima, dividido por grupo de instância testada. Veja que os grupos de instâncias *games*, *book* e *mycielskian* apresentam um número consideravelmente maior de soluções devolvidas ótimas do que soluções não ótimas. Enquanto isso, os grupos de instância *miles* e *queens* apresentam distribuições que não tem como valor mais destacado o 0. Essas duas são distribuições

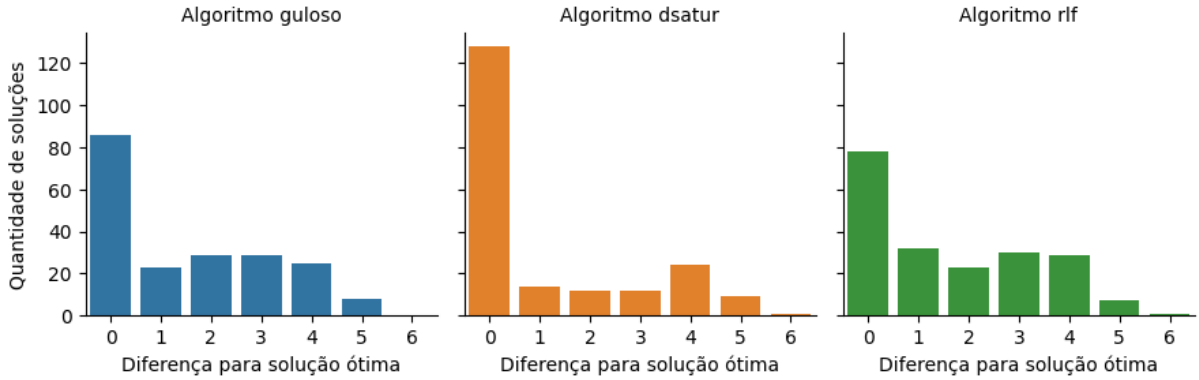

Figura 11 – Gráfico que mostra a quantidade de soluções que cada algoritmo guloso devolveu por diferença da solução ótima.

bem diferentes, pois *miles* é bem uniforme entre os valores 0 e 4 sem nenhum valor muito destacado, enquanto *queens* tem soluções mais concentradas nos valores 3 e 4 e quase nenhuma solução ótima. Então, com esses gráficos é bem visível que o grupo de instâncias influenciou bastante na qualidade das soluções devolvidas, sendo bem mais provável obter uma solução ótima para uma instância de *book* do que para uma instância de *queens*. Lembrando que podemos dizer os grafos de *book* são mais aleatórios que os de *queens* por não terem uma regra de criação de vértices e arestas tão determinística.

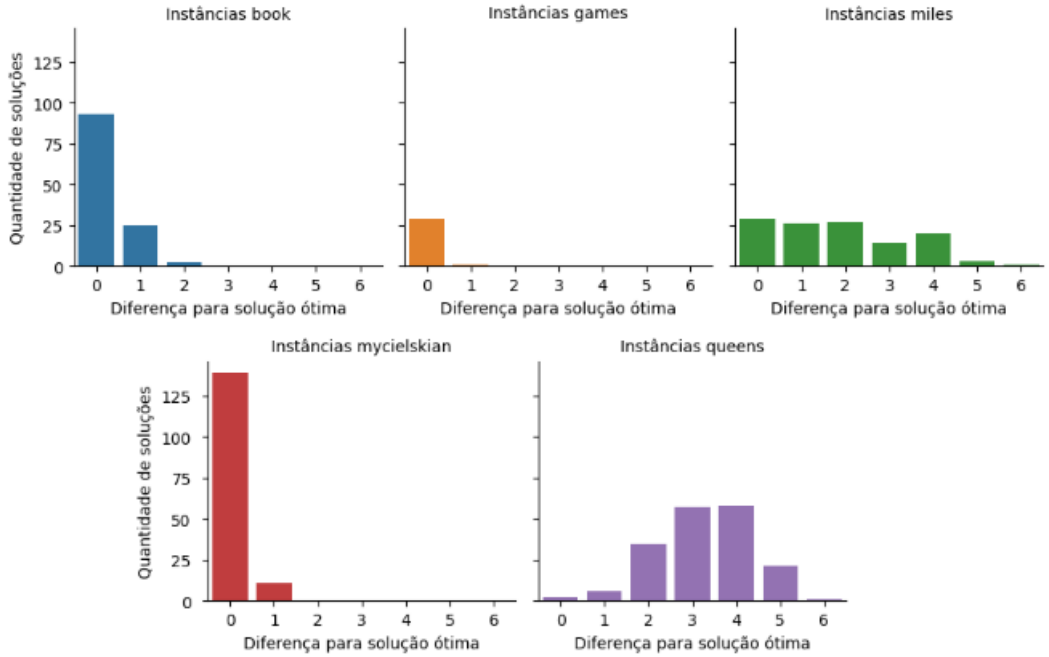

Figura 12 – Gráfico que mostra a quantidade de soluções devolvidas para cada grupo de instâncias por diferença da solução ótima.

Para fechar a análise da métrica de diferença para solução ótima vamos cruzar as informações referentes aos algoritmos e às instâncias. Para fazer essa análise usamos o gráfico na Figura 13, que mostra a média da métrica "diferença para solução ótima" para cada combinação de algoritmo e grupo de instância. A primeira coisa que podemos

destacar é que para todos os grupos de instâncias, exceto *queens*, o algoritmo DSatur apresentou um valor médio menor que os outros algoritmos. Veja que todos os algoritmos devolveram em média soluções com diferença da solução ótima maior que 3 para o grupo de instâncias *queens*, indicando que nenhum algoritmo se mostrou particularmente melhor no quesito de gerar soluções mais próximas do valor ótimo para esse grupo de instâncias. Porém, enquanto RLF e Guloso apresentam valores maiores que 2 para o grupo de instâncias *miles*, o Dsatur mostra um valor muito próximo de 0 podendo ser considerado um algoritmo que gera soluções de melhor qualidade para esse grupo de instâncias. Já para os grupos de instâncias *book*, *games* e *mycielskian*, embora o DSatur tenha apresentado valor 0 não há uma diferença que possa servir como prova de superioridade em relação aos outros algoritmos.

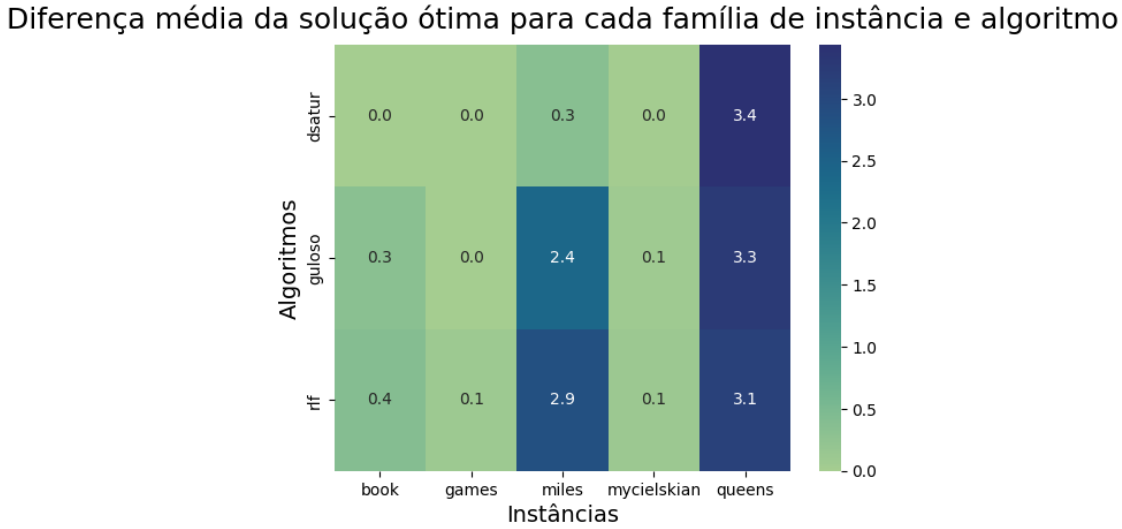

Figura 13 – Média da diferença da solução ótima das soluções devolvidas por um algoritmo para as instâncias de cada grupo.

Iniciamos agora a análise do tempo de execução dos algoritmos gulosos. Para basear nossas análises temos a Figura 14 e a Figura 15. Na Figura 14 vemos a distribuição dos tempos de execução dos algoritmos por quantidade de vértices da instância testada, e também temos uma separação visual dos algoritmos. Já na Figura 15 temos a distribuição dos tempos de execução por quantidade de arestas. Avaliando os dois gráficos, podemos perceber que parecem existir certas tendências no comportamento dos dados de cada algoritmo que relacionam a quantidade de vértices com o tempo de execução. Por exemplo, veja que os tempos referentes aos algoritmos Guloso e DSatur apresentam um comportamento de curva polinomial que se desenha conforme a quantidade de vértices cresce. Já os tempos referentes ao algoritmo RLF mostram-se praticamente constantes em comparação aos outros algoritmos. Isso pode ser considerado surpreendente, pois nas análises de tempo de execução realizadas no trabalho mostramos que o RLF possui o pior tempo de execução na análise de pior caso, mas para as instâncias testadas é o algoritmo que apresenta menores tempos de execução. Diferente da análise por quantidade de vértices,

a Figura 15, que mostra o cruzamento do tempo de execução com quantidade de arestas, não parece mostrar muita relação entre a quantidade de arestas e o tempo de execução. Isso pode ser considerado esperado já que nas análises de tempo de execução realizadas para os algoritmos mostramos que o termo referente à quantidade de arestas não contribui fortemente com o tempo de execução de nenhum desses algoritmos, fazendo com que a quantidade de vértices realmente seja mais relevante para o tempo de execução.

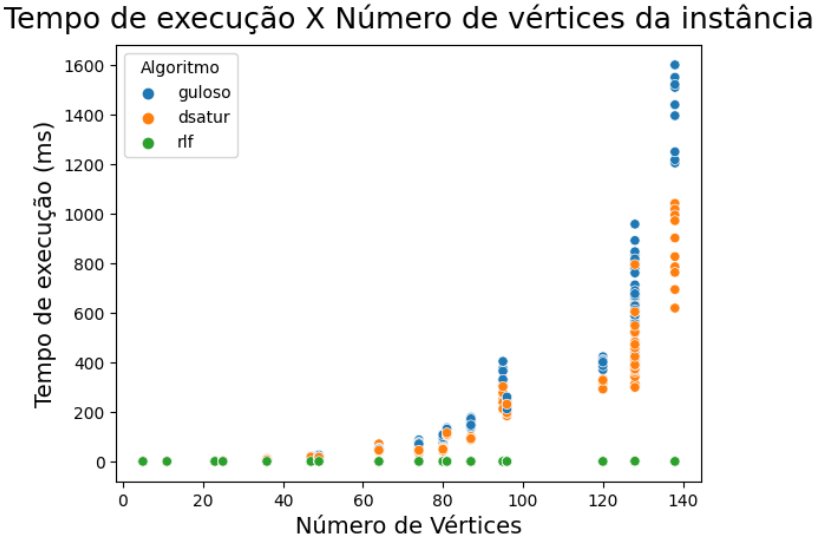

Figura 14 – Distribuição dos tempos de execução em milissegundos por quantidade de vértices da instância.

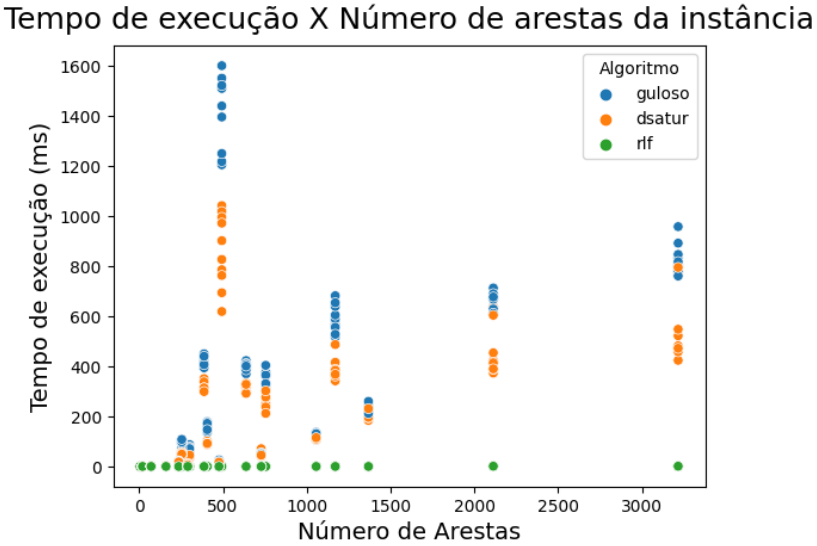

Figura 15 – Distribuição dos tempos de execução em milissegundos por quantidade de arestas da instância.

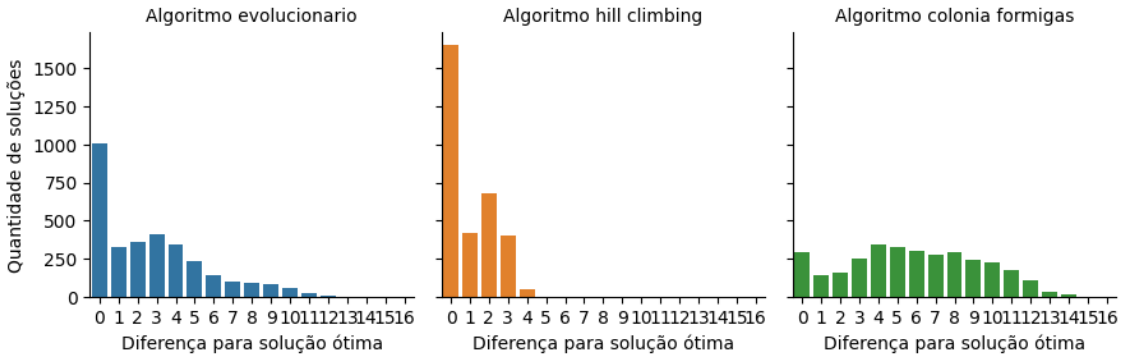

Figura 16 – Gráfico que mostra a quantidade de soluções que cada algoritmo meta-heurístico devolveu por diferença da solução ótima.

3.5.3 Análise de Algoritmos Meta-heurísticos

A análise de algoritmos gulosos na seção anterior tinha um facilitador em relação ao das meta-heurísticas, afinal como não haviam hiperparâmetros configuráveis executamos 200 vezes cada algoritmo, mas para os algoritmos meta-heurísticos vamos executar 200 vezes para cada uma das 16 combinações de hiperparâmetros. As combinações de valores de hiperparâmetros usadas são apresentadas na Seção 3.4. Vamos primeiro realizar as análises dos algoritmos Hill-Climbing, Evolucionario e AntCol, e mais a frente vamos explicar o porquê do Tabucol ser analisado à parte.

Vamos começar nossas análises da métrica "diferença para a solução ótima" com o gráfico da Figura 16, onde apresentamos a quantidade de soluções devolvidas por diferença para a solução ótima, dividido por algoritmo. De todos os algoritmos, claramente o Hill-Climbing apresentou melhores resultados mostrando uma moda muito significante no valor 0 com mais que o dobro de soluções do segundo maior valor que é 2. Já o algoritmo Evolucionario apresentou também moda em 0, mas a quantidade de soluções devolvidas com diferença para a solução ótima entre os valores 5 e 12 é muito mais representativa que do Hill-Climbing. Por fim, o AntCol apresentou uma distribuição mais uniforme entre diversos valores de diferença para solução, inclusive a moda do algoritmo na métrica de diferença para a solução ótima não é 0. Com isso, podemos dizer que o Hill-Climbing se mostrou o algoritmo que, por muito, devolveu uma maior quantidade de soluções ótimas para as instâncias testadas. Lembrando que para cada um desses algoritmos tivemos 3200 execuções.

Agora vamos voltar nossas atenções à variação de hiperparâmetros para o algoritmo Hill-Climbing. Na Figura 17 temos 16 gráficos e cada um deles indica um cruzamento dos dois hiperparâmetros variados para o Hill-Climbing: divisão da solução inicial no eixo y e quantidade máxima de iterações no eixo x. Vemos que quanto maior o valor do hiperparâmetro "Divisão", mais soluções ótimas foram devolvidas. Podemos ver que as execuções com valor de "Divisão" de 0,65 e 0,9 devolveram mais de 100 soluções ótimas.

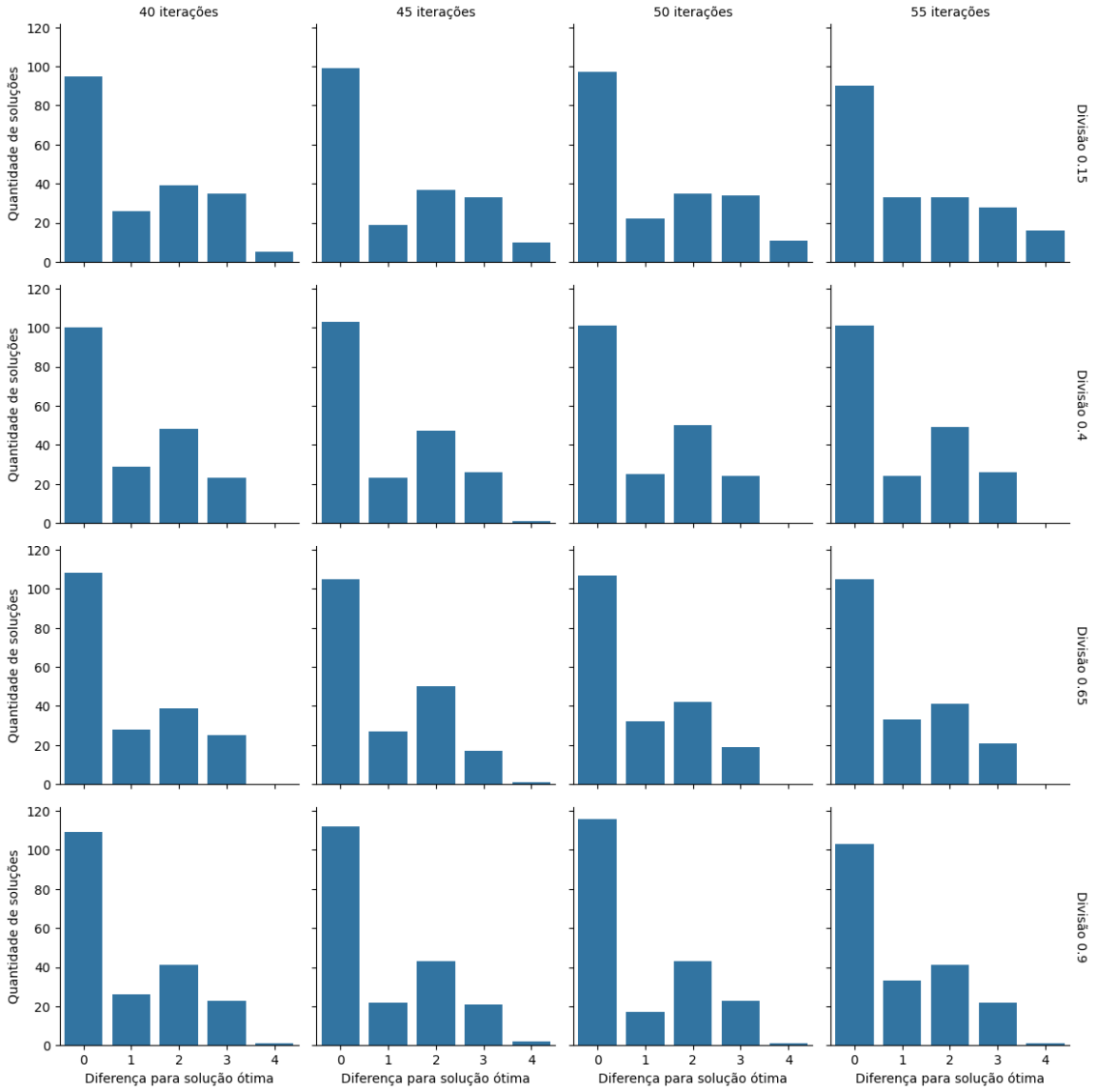

Figura 17 – Gráficos que mostram a quantidade de soluções que cada variação do HILL-CLIMBING devolveu por diferença da solução ótima.

No geral, podemos dizer que tivemos pouca variação da distribuição de quantidade de soluções pela métrica avaliada para os diferentes valores dos hiperparâmetros variados durante o experimento.

Vamos agora analisar as variações do hiperparâmetros do algoritmo EVOLUCIONARIO. Na Figura 18 temos um gráfico semelhante ao último analisado, mas com a variação de hiperparâmetros do algoritmo EVOLUCIONARIO: quantidade de membros no eixo x e quantidade de gerações no eixo y. Analisando os gráficos vemos que, quanto maior a quantidade de gerações usadas no algoritmo, menos soluções ótimas foram sendo devolvidas, sendo esse um comportamento observado para todas as quantidades de membros. Já em relação à quantidade de membros, vemos que essa quantidade se mostrou proporcional à qualidade das soluções devolvidas, pois quanto mais membros maior a quantidade de soluções ótimas devolvidas. Assim, a combinação de mais membros na população (25)

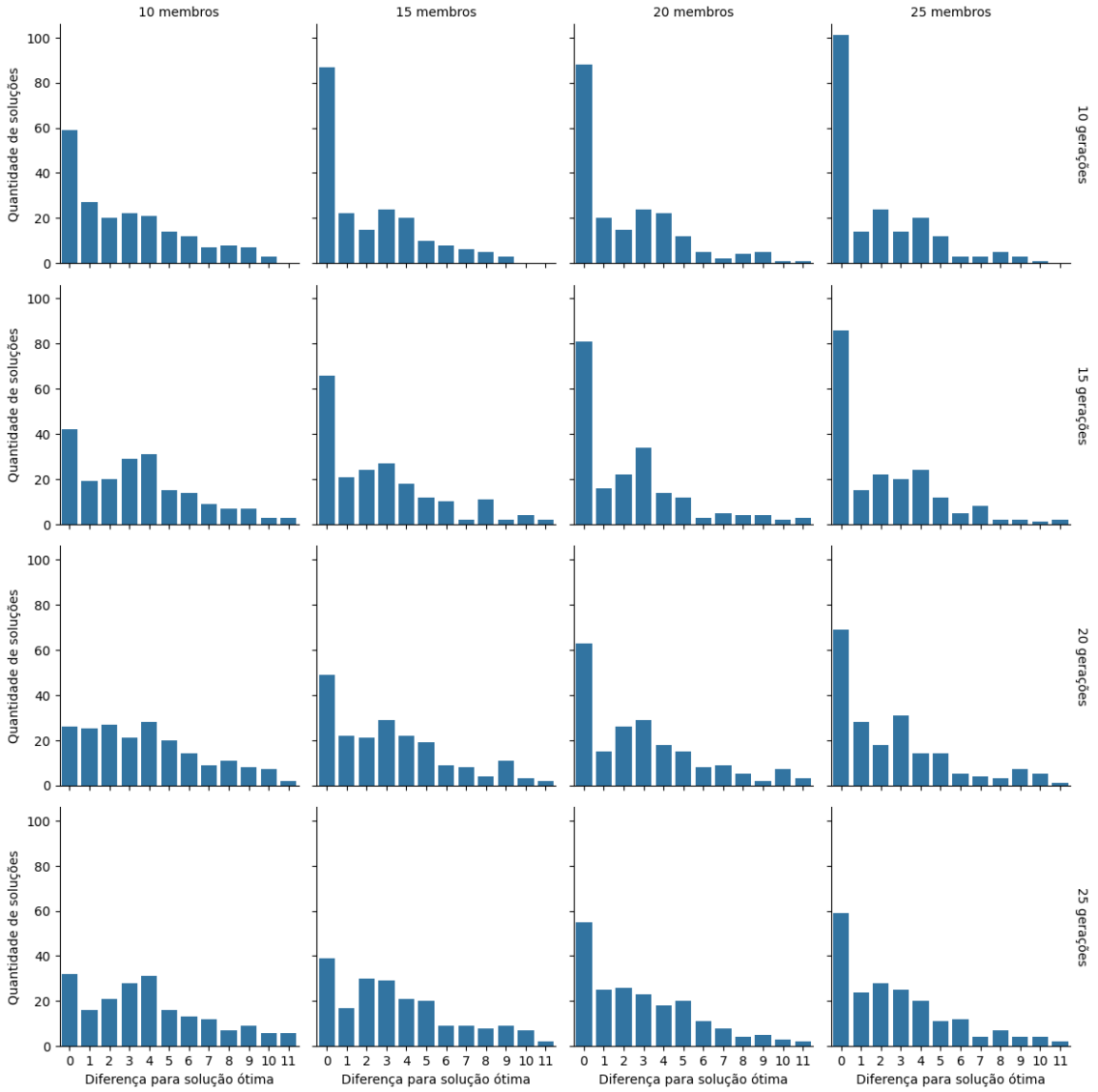

Figura 18 – Gráficos que mostram a quantidade de soluções que cada variação do EvolucionarIO devolveu por diferença da solução ótima.

com menor quantidade de gerações (10) mostrou-se a melhor em comparação às demais, uma vez que essa foi a única combinação com mais de 100 soluções ótimas devolvidas. Por outro lado, a combinação de poucos membros com muitas gerações parece ter um efeito contrário, mostrando uma tendência a ter distribuições achatadas e com uma quantidade bem menor de soluções ótimas devolvidas.

Para finalizar nossas análises de variação de hiperparâmetros, vamos avaliar os resultados devolvidos para a variação de quantidade de formigas e valor de evaporação no algoritmo AntCol, o que é mostrado na Figura 13. Diferente dos resultados obtidos para os outros algoritmos, é possível ver que o AntCol apresentou, em diversos cenários de hiperparâmetros, distribuições onde a moda da métrica avaliada não foi 0. Na realidade, é difícil fazer qualquer análise muito distante dessa constatação dado que as distribuições são todas muito parecidas no ponto que mais nos interessa, que é analisar a quantidade de

soluções ótimas devolvidas *versus* a quantidade de soluções não ótima devolvidas. Outra coisa que podemos afirmar é que a maior concentração de soluções encontram-se entre os valores 4 e 10.

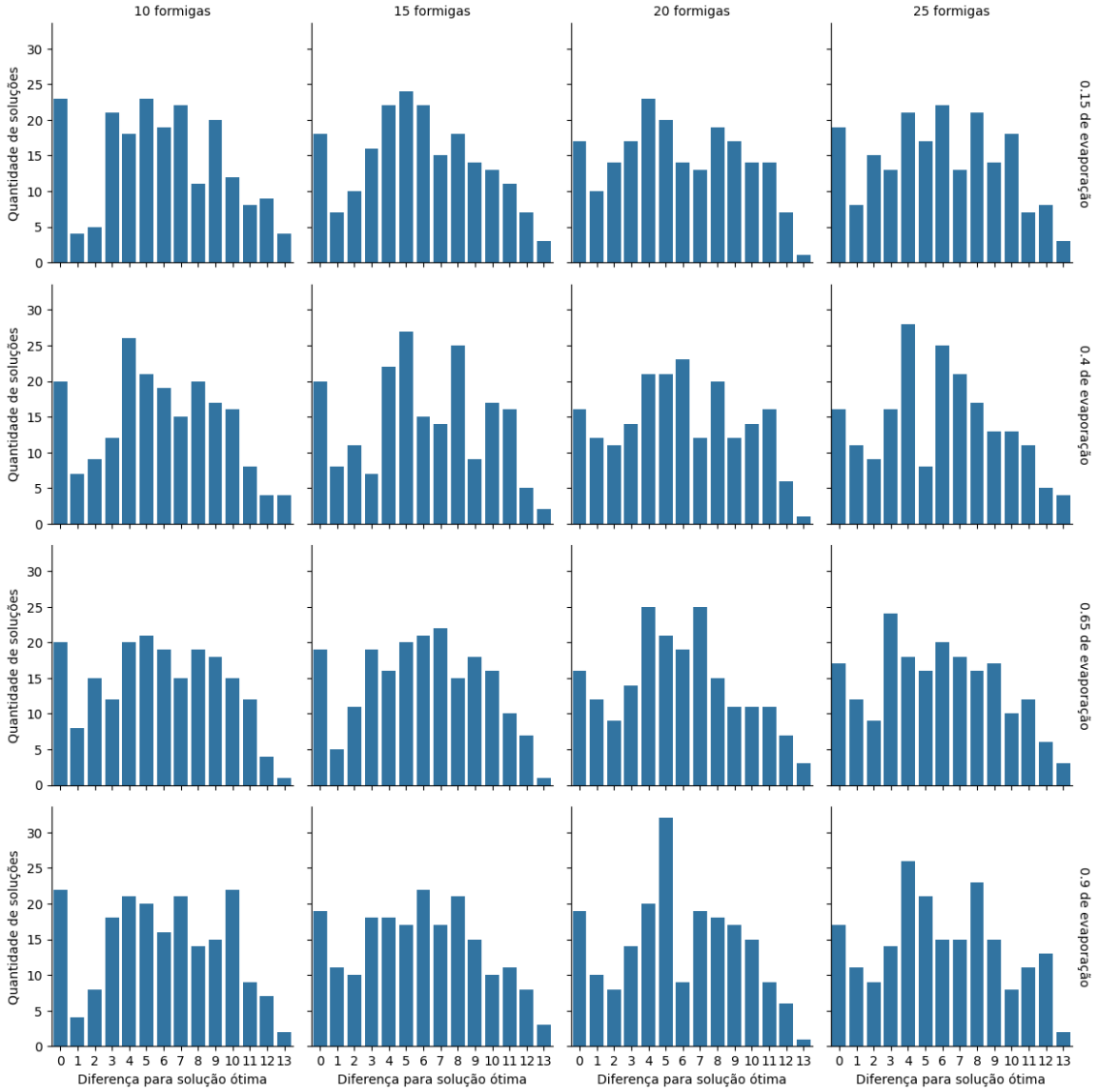

Figura 19 – Gráficos que mostram a quantidade de soluções que cada variação do AntCol devolveu por diferença da solução ótima.

Para finalizar nossas análises da métrica de diferença para a solução ótima vamos cruzar as informações de algoritmo e grupo de instância executada. Veja na Figura 20 as distribuições de cada algoritmo para cada grupo de instância. Com esses gráficos conseguimos ver com mais detalhes que o Hill-Climbing mostrou uma maior quantidade de soluções ótimas que os outros algoritmos muito em função das instâncias *mycielskian* e *book*. O AntCol apresentou uma distribuição parecida para todos os grupos de instância exceto *mycielskian*, todos com uma distribuição que apresenta moda no centro. Veja que para nenhum dos algoritmos as instâncias *queens* tiveram como principal resultado 0, pois tivemos muito mais soluções não ótimas para essas instâncias. Por outro lado, as soluções

devolvidas para as instâncias *mycielskian* apresentaram forte tendência de serem ótimas independente do algoritmo, embora alguns possam devolver soluções bem mais distantes do ótimo do que outros.

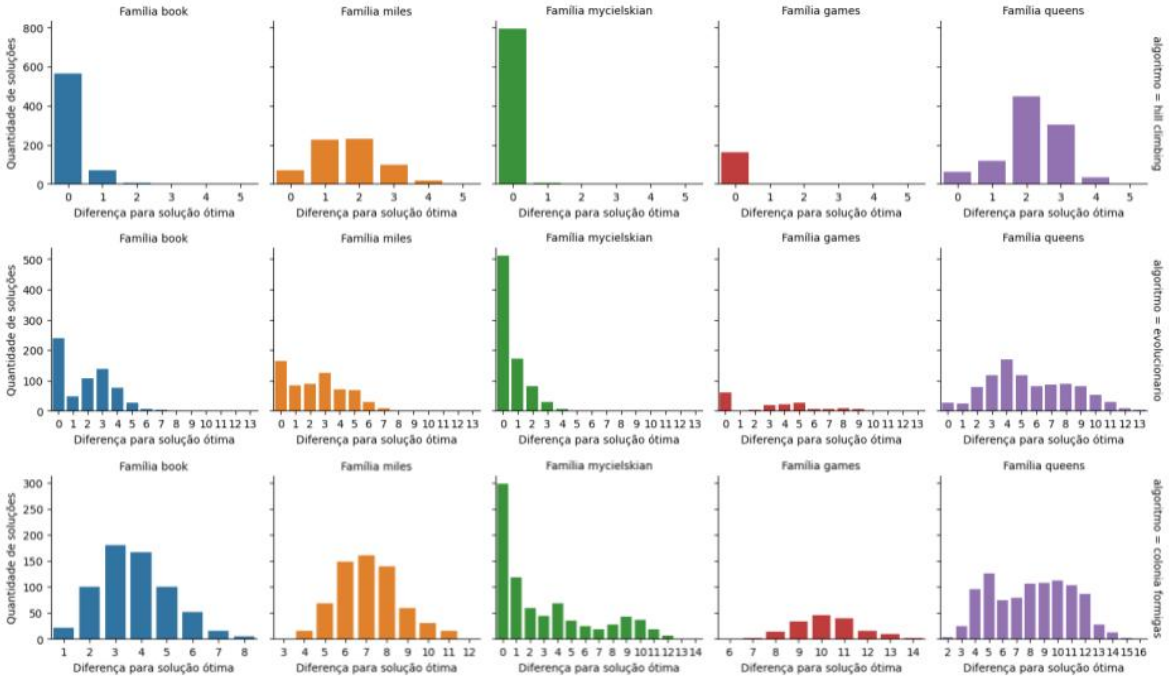

Figura 20 – Gráficos que mostra para cada combinação de grupo de instância e algoritmo a distribuição de quantidade de soluções por diferença para a solução ótima.

No geral podemos dizer que os resultados para a implementação do Hill-Climbing foram superiores aos dos demais algoritmos, especialmente por apresentar uma maior quantidade de soluções ótimas e um menor range de valores possíveis da métrica avaliada. Afinal, a pior solução apresentada pelo Hill-Climbing tem diferença 6 para a solução ótima, enquanto os outros dois algoritmos devolveram soluções com até o dobro ou mais de distância para a solução ótima. Porém, é importante ressaltar que para realizar afirmações mais generalistas sobre as implementações se faz necessário realizar mais testes de desempenho com outras instâncias de grafos e outros valores de hiperparâmetros. Ainda sobre a variação de hiperparâmetros, com exceção do AntCol, foi possível observar comportamentos e tendências com as variações de hiperparâmetros, indicando que as suas variações de fato impactaram na qualidade das soluções devolvidas.

Agora, vamos focar na outra métrica avaliada nas análises, o tempo de execução. As análises e os resultados de interesse foram muito semelhantes aos dos algoritmos gulosos, por isso vamos realizar fazer as análises da relações de tempo de execução com vértices e arestas de uma vez.

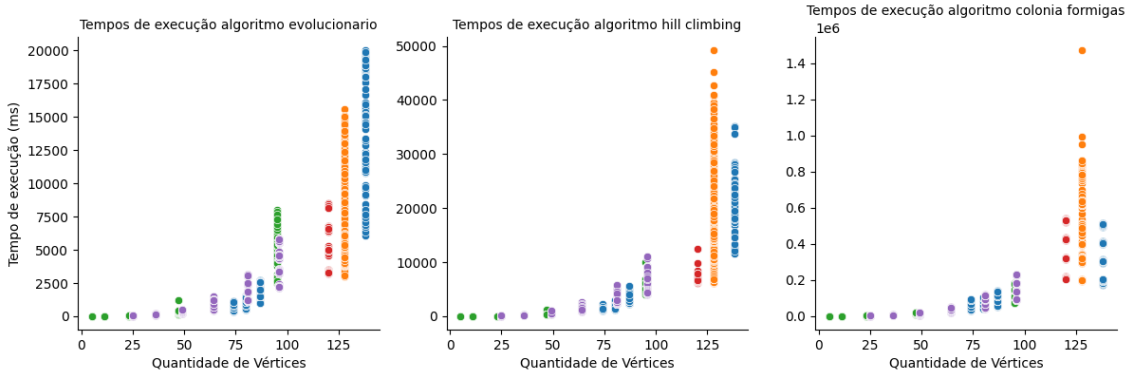

Figura 21 – Distribuição dos tempos de execução em milissegundos por quantidade de vértices da instância e algoritmo avaliado.

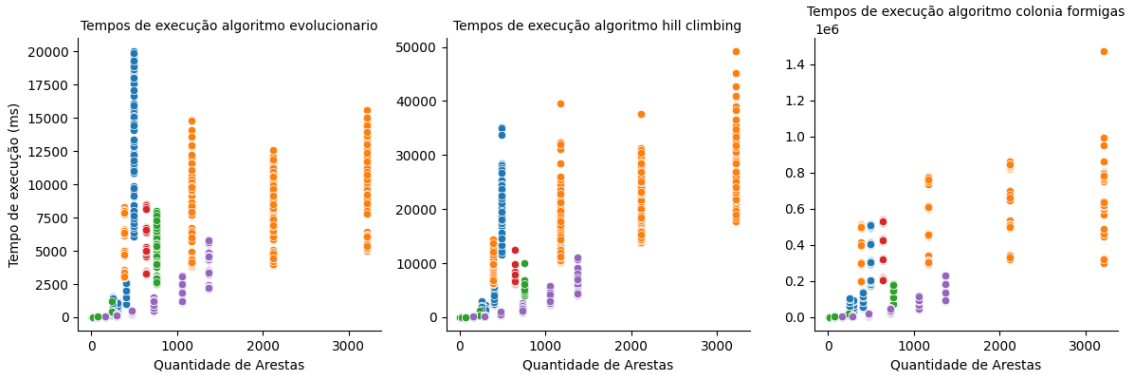

Figura 22 – Distribuição dos tempos de execução em milissegundos por quantidade de arestas da instância e algoritmo avaliado.

Nas Figuras 21 e 22 temos os tempos de execução dos testes realizados separados por algoritmo de execução. Analisando a relação com o número de vértices podemos ver que o tempo de execução aparenta mostrar relação polinomial com a quantidade de vértices para todos os algoritmos, algo previsto em nossas análises de tempo de execução. Além disso, é possível ver que para cada um dos algoritmos estamos lidando com uma escala diferente do tempo de execução, pois enquanto execuções do Hill-Climbing executaram em até 50000ms, execuções do Evolucionario executaram em até 20000ms e execuções do AntCol executaram em até 1400000ms. Considerando que, em nossas análises teóricas de tempo de execução, o AntCol realmente apresentava um tempo de pior caso como um polinômio na quantidade de vértices de grau maior que os polinômios de pior caso dos outros dois algoritmos, então ele apresentar um tempo de execução mais elevando é algo esperado. Agora, os resultados do Hill-Climbing mostraram tempos de execução mais elevados do que o algoritmo Evolucionario, o que não era o esperado, mas indica que talvez o algoritmo Evolucionario seja uma boa opção nesse quesito para os grupos de grafos usados como instância. Veja que diferentemente da relação com a quantidade de vértices, a relação entre o tempo de execução e a quantidade de arestas não parece ter causalidade, fato especialmente observável para instâncias com muitos vértices e poucas

arestas. Considerando nossas análises de pior caso para os algoritmos, vemos que não é um resultado inesperado que o valor de arestas afete menos que o de vértices, pois o tempo de execução apresenta dependência linear a esse valor.

Agora vamos falar brevemente do TABUCOL, justificando o porquê de seus resultados não terem sido apresentados e analisados em conjunto com os das outras meta-heurísticas. Para isso, vamos olhar a Figura 23. Veja que para todos os valores de tabu analisados tivemos soluções devolvidas com mais de 80 cores a mais que o valor ótimo. Ao observarmos esses resultados, decidimos por não compará-los com os dos outros para não prejudicar as análises devido às escalas devolvidas serem consideravelmente diferentes. Lembrando sempre que o TABUCOL foi o único dos algoritmos apresentados que também poderia devolver soluções inviáveis.

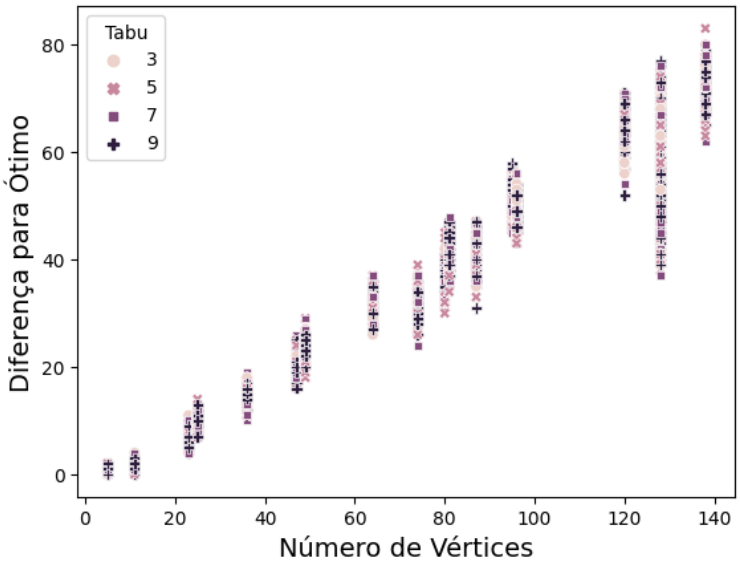

Figura 23 – Distribuição da quantidade de soluções devolvidas pelo TABUCOL por diferença para a solução ótima com diferenciação pelo valor tabu.

3.5.4 Conclusões

Após as análises dos experimentos sobre as implementações dos algoritmos gulosos e meta-heurísticos, entendemos que a experimentação prática se mostrou muito importante para a completude do trabalho por algumas razões.

Uma primeira razão que podemos apontar é nos momentos onde a análise experimental mostrou um resultado diferente do esperado, algo que pode ser especialmente destacado para a análise dos tempos de execução. Alguns exemplos são o RLF ter apresentado um tempo de execução menor para as instâncias analisadas do que os outros algoritmos, sendo ele $O(n^3)$ e os outros algoritmos $O(n^2)$. Mas também temos muito valor nas análises experimentais que vão de acordo com as expectativas inicias, como a relação entre os tempos de execução e a quantidade de arestas. Para todos os algoritmos analisados,

o pior caso era sempre proporcional a um termo linear da quantidade de arestas, o que podemos ver como a baixíssima relação aparente que a quantidade de arestas teve com tempos de execução experimental. Isso pode ser especialmente observado para as instâncias *miles* que apresentam muitos vértices mas poucas arestas.

Outro ponto é a avaliação da qualidade das implementações que os testes nos permitiram. Nosso maior exemplo é o TABUCOL, pois foi somente na consolidação dos dados coletados que foi possível perceber que deveriam ser feitos ajustes para garantir um desempenho do algoritmo que não o afastasse tanto dos ótimos locais, mesmo sendo um algoritmo com a ideia de também poder devolver soluções inviáveis. Também percebemos que no geral os algoritmos gulosos apresentaram uma maior tendência de devolver soluções ótimas ou com valor mais próximo do ótimo Acreditamos que esse fato se deve principalmente ao fato das instâncias testadas não possuírem um número tão grande de vértices e arestas. Acreditamos que por serem algoritmos mais complexos, as meta-heurísticas aparentaram não compensar tanto o uso para as instâncias avaliadas quando em comparação com os algoritmos gulosos. Também tivemos muitas análises que focaram nos grupos de instâncias usados nos testes. Ficou muito claro nos momentos em que cruzamos as informações de algoritmos e grupos de instâncias que as instâncias *queens* e *miles* foram as que se mostraram mais difíceis no quesito de encontrar soluções ótimas. Em especial as instâncias *queens*, pois podemos dizer que nenhum algoritmo apresentou resultado excelente para essas instâncias.

4 Considerações Finais

Na Introdução desse trabalho apresentamos como seu objetivo principal o desenvolvimento de um pacote Python com a implementação de certos algoritmos para o problema de coloração de grafos. Esse objetivo foi alcançado com sucesso e nesse trabalho conseguimos mostrar conceitos e etapas relevantes para alcançá-lo. O pacote desenvolvido foi disponibilizado no PyPI e seu código é *open-source*, proporcionando fácil acesso e simples utilização ao que já foi implementado enquanto possibilita que desenvolvedores interessados no tema possam contribuir com o projeto.

Ao longo do desenvolvimento, foram implementados três diferentes tipos de algoritmos: heurísticos gulosos, meta-heurísticos e exatos. Cabe aqui fazer uma consideração sobre a implementação de cada grupo de algoritmo, dado que esse foi o principal objetivo do projeto. A implementação dos algoritmos heurísticos gulosos foi a mais simples devido à sua lógica de funcionamento ser mais direta e possuir menos conceitos envolvidos. Por outro lado, os algoritmos meta-heurísticos apresentaram um desafio maior tanto em termos conceituais, por possuírem ideias referentes a meta-heurísticas sofisticadas, quanto em termos de programação, por possuírem etapas e passos mais complexos de serem estruturados. Já a implementação dos algoritmos exatos utilizou recursões que não encontramos tanta dificuldade de implementar, especialmente porque a estratégia principal para esse tipo de algoritmo é a força bruta. É importante ressaltar que implementar esses algoritmos em Python com pacotes próprios da linguagem não gerou empecilhos práticos no desenvolvimento, mas sim contribuiu devido ao amplo arsenal de ferramentas que o uso dessa linguagem proporciona, reforçando que a escolha da linguagem foi coerente.

Além das implementações, foram conduzidos testes utilizando um conjunto de instâncias de teste. Os resultados obtidos foram surpreendentes, revelando que os algoritmos heurísticos gulosos apresentaram um desempenho superior em média, no que toca qualidade de solução, em relação aos algoritmos meta-heurísticos, mesmo com variações nos hiperparâmetros desses últimos. Esse resultado acabou sendo surpreendente porque os algoritmos gulosos tem um funcionamento considerado mais simples do que meta-heurísticas, que possuem mecanismos de busca mais exaustivos dentro dos espaços de solução que trabalham. Os tempos de execução dos algoritmos heurísticos construtivos também foram notavelmente rápidos, destacando-se o algoritmo RLF como o mais eficiente experimentalmente, contrariando a expectativa teórica de ser o mais custoso. Também é importante ressaltar que a experimentação foi importante para visualizarmos possíveis problemas na implementação do Tabucol. De forma geral, podemos dizer que a experimentação é um recurso importante para avaliar heurísticas e meta-heurísticas, especialmente por serem um tipo de algoritmo que não oferece garantias quanto à qualidade de solução. É

importante ressaltar que os resultados registrados e apresentados nesse trabalho referem-se a um conjunto específico de instâncias, não sendo possível uma generalização muito mais abrangente desses resultados.

A realização desse projeto foi muito importante para fechar a formação de cientista da computação do aluno, servindo como um meio de aprofundamento e exploração de tópicos importantes do campo como engenharia de software, projetos de algoritmos, otimização combinatória, teoria dos grafos, entre outros. Essa visão abrangente, que cobriu o projeto desde o estudo das bases teóricas até a experimentação com os algoritmos, proporcionou uma completude para o projeto, permitindo que melhorias e novos assuntos se integrassem com o que estava sendo desenvolvido de maneira natural.

Sobre trabalhos futuros que poderiam ser realizados, acreditamos que existam duas linhas principais. A primeira seria a implementação de mais algoritmos para o problema de coloração de grafos, podendo esses pertencerem aos tipos de algoritmos apresentados ou pertencentes a grupos novos. Citamos como exemplo os algoritmos exatos que usam programação linear e que foram citados no trabalho de Lima [19]. Como são um tipo de algoritmo exato muito específico, talvez valeria a pena criar uma nova classe somente para eles. A segunda linha seria a realização de mais testes experimentais, sendo que esses novos testes poderiam focar em avaliar métricas diferentes, como quantidade de comparações realizadas, ou usar as mesmas implementações mas em instâncias de teste diferentes e maiores para reforçar ou discordar das conclusões aqui apresentadas sobre os nossos resultados.

Referências

[1] K. Appel and W. Haken. Every planar map is four colorable. Part I: Discharging. *Illinois Journal of Mathematics*, 21(3):429–490, 1977. doi:10.1215/ijm/1256049011.

[2] J. A. Bondy and U. S. R. Murty. *Graph Theory*. Springer Publishing Company, Incorporated, 1st edition, 2008. ISBN 1846289696.

[3] C. Bron and J. Kerbosch. Algorithm 457: Finding all cliques of an undirected graph. *Commun. ACM*, 16(9):575–577, 1973. doi:10.1145/362342.362367.

[4] D. Brélaz. New methods to color the vertices of a graph. *Commun. ACM*, 22(4): 251–256, 1979. doi:10.1145/359094.359101.

[5] C. N. Campos. *O problema da coloração total em classes de grafos*. PhD thesis, Instituto de Computação - Universidade Estadual de Campinas, 2006.

[6] M. H. Carvalho, M. R. Cerioli, R. Dahab, P. Feofiloff, C. G. Fernandes, F. C. E., K. S. Guimarães, F. K. Miyazawa, J. C. Pina Jr., J. A. R. Soares, and Y. Wakabayashi. *Uma introdução sucinta a algoritmos de aproximação*. 2001. ISBN 85-244-0184-2.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and S. C. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009. ISBN 0262033844.

[8] K. A. Dowsland and J. M. Thompson. An improved ant colony optimisation heuristic for graph colouring. *Discrete Applied Mathematics*, 156(3):313–324, 2008. doi:10.1016/j.dam.2007.03.025.

[9] P. Galinier and J.-K. Hao. Hybrid evolutionary algorithms for graph coloring. *Journal of Combinatorial Optimization*, 3:379–397, 01 1999. doi:10.1023/A:1009823419804.

[10] M. Gendreau and J.-Y. Potvin. *Handbook of Metaheuristics*. Springer Publishing Company, Incorporated, 2nd edition, 2010. ISBN 1441916636.

[11] A. Hertz and D. Werra. Using tabu search techniques for graph coloring. computing 39, 345-351. *Computing*, 39:345–351, 12 1987. doi:10.1007/BF02239976.

[12] D. Johnson. A theoretician's guide to the experimental analysis of algorithms, 12 2001.

[13] R. Karp. *Reducibility among combinatorial problems*. Plenum Press, 1972. doi:10.1007/978-1-4684-2001-2.

[14] E. L. Lawler. A note on the complexity of the chromatic number problem. *Information Processing Letters*, 5(3):66–67, 1976. doi:https://doi.org/10.1016/0020-0190(76)90065-X.

[15] F. T. Leighton. A graph coloring algorithm for large scheduling problems. *J. Res. Natl. Bur. Stand.*, 84(6):489–506, 1979. doi:10.6028/jres.084.024.

[16] R. Lewis. A general-purpose hill-climbing method for order independent minimum grouping problems: A case study in graph colouring and bin packing. *Computers & Operations Research*, 36(7):2295–2310, 2009. doi:10.1016/j.cor.2008.09.004.

[17] R. M. R. Lewis. *A Guide to Graph Colouring*. Springer International Publishing, 1st edition, 2016. ISBN 978-3-319-25730-3.

[18] R. R. Lewis. *A Guide to Graph Colouring: Algorithms and Applications*. Springer Publishing Company, Incorporated, 1st edition, 2015. ISBN 3319257285.

[19] A. M. Lima. Algoritmos exatos para o problema da coloração de grafos. Master's thesis, Universidade Federal do Paraná, 2017.

[20] C. C. McGeoch. *Experimental Analysis of Algorithms*, pages 489–513. Springer US, 2002. ISBN 978-1-4757-5362-2. doi:10.1007/978-1-4757-5362-2_14.

[21] K. Mirajkar and V. Mathad. Miscellaneous properties of line mycielskian graph of a graph. *International Journal of Applied Engineering Research*, 14:4552–4556, 01 2020. ISSN 0973-4562.

[22] C. C. Wang. An algorithm for the chromatic number of a graph. *Journal of the ACM*, 21(3):385–391, 1974. doi:10.1145/321832.321837.