Universidade Federal do Rio Grande do Norte

Departamento de Informática e Matemática Aplicada

Relatório da 1^a Unidade - Grafos

Alexandre Dantas, Andriel Vinicius, Gabriel Carvalho, Maria Paz e Vinicius de Lima

Professor: Matheus Menezes

10 de outubro de 2025

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Este projeto tem como objetivo implementar uma API de algoritmos relacionados a Grafos na linguagem de programação Rust. A especificação da API segue as funcionalidades descritas na definição da Avaliação 01 da disciplina de Grafos, sob o Departamento de Informática e Matemática Aplicada (DIMAp) da Universidade Federal do Rio Grande do Norte (UFRN).

Nesta implementação, a especificação da API se materializa em grande parte como *traços* (*traits*) de Rust. Tal conceito é semelhante a noção de *interface* em linguagens como Java e Go e praticamente idêntico ao conceito de *typeclasses* em linguagens como Haskell. O uso de traços permite que a API seja flexível e mantenha fidelidade as definições algébricas, além do mais, permite, por meio da implementação a nível de traço, que o código seja genérico e eficiente para as diversas representações de grafos.

Além dos traços, fizemos grande uso do conceito de *iteradores*, que também surgem a partir de traços, essa abstração permite que o código fique ainda mais reutilizável, sem que a eficiência seja comprometida.

Neste relatório vamos discutir a implementação e apresentar essas vantagens em detalhes. Além disso, vamos demonstrar empiricamente a partir de testes de benchmark que apesar do alto nível de abstração, a eficiência dos algoritmos ainda fica a par de implementações ótimas feitas em C++.

Para assegurar o comportamento esperado dos algoritmos, também implementamos mais de 50 testes unitários que testam as validam as implementações. Entretanto, por causa da extensão dos testes, vamos disponibilizá-los apenas através do código fonte da API.

1.1 Especificação da API

De acordo com a definição da Avaliação 01 a API deve possuir as seguintes funcionalidades:

1.1.1 Para Grafos Não Direcionados

1. Criação do Grafo a partir da Lista de Adjacências.
2. Criação do Grafo a partir da Matriz de Adjacências.
3. Criação do Grafo a partir da Matriz de Incidência.
4. Conversão de matriz de adjacência para lista de Adjacências e vice-versa.
5. Função que calcula o grau de cada vértice.
6. Função que determina se dois vértices são adjacentes.

7. Função que determina o número total de vértices.
8. Função que determina o número total de arestas.
9. Inclusão de um novo vértice usando Lista de Adjacências e Matriz de Adjacências.
10. Exclusão de um vértice existente usando Lista de Adjacências e Matriz de Adjacências.
11. Função que determina se um grafo é conexo ou não.
12. Determinar se um grafo é bipartido.
13. Busca em Largura, a partir de um vértice específico.
14. Busca em Profundidade, com determinação de arestas de retorno, a partir de um vértice em específico.
15. Determinação de articulações e componentes biconexos, utilizando obrigatoriamente a função `lowpt`.

1.1.2 Para Grafos Direcionados (ou Dígrafos)

16. Representação do Dígrafo a partir da Matriz de Adjacências.
17. Representação do Dígrafo a partir da Matriz de Incidência.
18. Determinação do Grafo subjacente.
19. Busca em largura.
20. Busca em profundidade, com determinação de profundidade de entrada de saída de cada vértice, e arestas de árvore, retorno, avanço e cruzamento.

1.2 Solução proposta

1.2.1 Traços

Como discutido brevemente no início do capítulo (1), nossa solução se baseia na criação de traços e iteradores que vão implementar os algoritmos descritos na especificação da API (1.1). Um traço define a funcionalidade que um tipo tem e que pode compartilhar com outros tipos (Klabnik et al., 2025*b*). No nosso caso específico, haverá um traço para grafos não direcionados, chamado de `Graph`, e um traço para grafos direcionados, chamado de `UndirectedGraph`. O traço `UndirectedGraph` herdará as propriedades do traço `Graph`, permitindo que os algoritmos definidos em `Graph` funcionem em `UndirectedGraph`. Essa escolha faz sentido porque na nossa implementação o grafo não direcionado funciona melhor como uma extensão do grafo direcionado.

1.2.2 Iteradores

Quanto ao uso de iteradores, eles serão implementados em algoritmos que envolvem algum tipo de travessia no grafo, como a Busca em Profundidade (DFS) dos items 14 e 20, e a Busca em Largura (BFS) dos items 13 e 19. A vantagem de implementar iteradores para esses algoritmos é que outros algoritmos, como a Determinação de Componentes Biconexos, do item 15, fica derivável dessa implementação. Além disso, a implementação em forma de

iteradores permite que possíveis usuários da API tenham acesso ao extenso framework de iteradores que a biblioteca padrão de Rust oferece. Alguns exemplos de uso de iteradores para além da implementação da API, seriam, a determinação de ciclos no grafo, a determinação de menor caminho entre dois vértices, a busca por um vértice específico, uma filtragem de vértices durante a busca, uma modificação do grafo durante a busca, algoritmos de backtracking e outros, tudo isso apenas compondo a implementação inicial. É importante ressaltar, também, que o iterador de Rust não perde desempenho em relação a uma implementação tradicional com loops (Klabnik et al., 2025*a*; *Zero-cost abstractions: performance of for-loop vs. iterators — stackoverflow.com*, 2018), inclusive podendo ser mais ótimo devido a otimizações do compilador. Essa capacidade lhe dá o apelido de abstração de custo zero.

1.3 Organização do relatório

No capítulo 2, vamos apresentar as definições que vamos usar durante o relatório e construir a base necessária para que o leitor consiga acompanhar a parte teórica em sua totalidade. Nessa seção revisaremos conceitos como definição de grafo, vértices, arestas, adjacência, conectividade e outros.

No capítulo 3, vamos revisar as representações de grafos nas estruturas de dados que temos disponíveis na linguagem de programação, estas são, Lista de Adjacência, Matriz de Adjacência e Matriz de Incidência. Também vamos discorrer um pouco sobre as vantagens e desvantagens de cada uma.

No capítulo 4, vamos descrever em uma linguagem de pseudocódigo a implementação dos algoritmos da especificação da API. Novamente, para preparar o leitor teoricamente para a implementação em Rust.

No capítulo 5, vamos apresentar e discorrer sobre as partes relevantes do código em Rust que implementa a API e consequentemente as representações e algoritmos descritos nos capítulos 3 e 4.

No capítulo 6, vamos demonstrar porque as abstrações utilizadas na implementação é zero custo comparando sua performance com uma implementação tradicional em C++.

No capítulo 7, vamos discutir as possíveis melhorias a nossa API e quais podem ser os futuros próximos passos.

No apêndice A, você poderá consultar as atividades desenvolvidas por cada integrante.

Capítulo 2

Revisão teórica

As definições utilizadas neste projeto foram, em grande parte, retiradas de Diestel (2025), com algumas modificações. Como a implementação dos algoritmos está em inglês, também apresentaremos o correspondente de cada definição em inglês, entre parênteses.

Definição 1 (Grafo)**.** Um *grafo* (*graph*) é uma estrutura $G := (V, A)$ tal que $A \subseteq V^2$ e V é um conjunto de um tipo qualquer. Os elementos de V são denominados vértices (*nodes*) e os elementos de A são denominados de arestas (*edges*). O jeito tradicional de visualizar um grafo é como uma figura composta de bolas e setas:

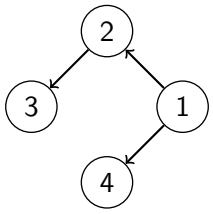


Figura 2.1: Um grafo com $V := \{1, 2, 3, 4\}$ e $A := \{(1, 2), (1, 4), (2, 3)\}$.

Definição 2 (Ordem e Tamanho)**.** O número de vértices de um grafo G é chamado de *ordem* (*order*) e é denotado por $|G|$ – o número de arestas é chamado de *tamanho* (*size*) e é denotado por $||G||$. Por exemplo, na Figura 2.1, $|G| = 4$ e $||G|| = 3$.

Definição 3 (Adjacência)**.** Dizemos que um vértice v é *adjacente*, ou *vizinho*, de um vértice u (*neighbor*) se somente se $(u, v) \in A$, também, denotaremos (u, v) como uv. Visualmente, enxergamos isso como:

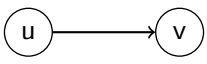


Figura 2.2: Um grafo com $V := \{u, v\}$ e $A := \{(u, v)\}$.

Definição 4 (Conjunto de adjacentes)**.** Num grafo G, o conjunto de todos os vértices adjacentes de u (*neighbors*) é denotado por $A_G(u) := \{v \in V \mid uv \in A\}$. Já o conjunto de todos os vértices que em que v é adjacente será denotado por $\bar{A}_G(u) := \{v \in V \mid vu \in A\}$.

Definição 5 (Grau de um vértice)**.** O *grau de um vértice* v (*node degree*) é o valor correspondente da soma $|A_G(v)| + |\bar{A}_G(v)|$. Também denotamos $|A_G(v)|$ como $d^+(v)$, $|\bar{A}_G(v)|$ como $d^-(v)$ e sua soma como $d(v)$.

Definição 6 (Caminho)**.** Um *caminho* (*path*) é um grafo $C := (V, A)$ que tem a forma:

$$V := \{x_0, x_1, ..., x_k\} \qquad A := \{x_0x_1, x_1x_2, ..., x_{k-1}x_k\}$$

Dizemos que C é um caminho de x_0 a x_k. Normalmente nos referimos ao caminho como a sequência dos seus vértices, $x_0x_1...x_k$.

Definição 7 (Conectividade)**.** Dizemos que um grafo G é conexo (*connected*) se somente se para quaisquer dois vértices u e v, existe um caminho entre eles.

Definição 8 (Grafo não direcionado)**.** Dizemos que um grafo G é *não direcionado* (*undirected*) se somente se A é simétrico, ou seja, se $uv \in A$ então $vu \in A$. O nome não direcionado vem da ideia de que os grafos que viemos discutindo até agora são denomidados de *direcionados*, ou simplesmente *dígrafos*. Na literatura é comum apresentar grafo não direcionado como grafo e depois o direcionado como dígrafo, resolvemos inverter a ordem pois assim se traduz melhor nas representações de grafos que vamos implementar. Um grafo não direcionado pode ser visualizado sem a ponta das setas:

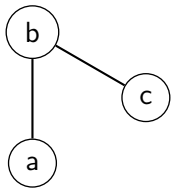


Figura 2.3: Um grafo não direcionado com $V := \{a, b, c\}$ e $A := \{ab, ba, bc, cb\}$

Também é comum omitir a simetria das arestas se pelo contexto for claro que está sendo tratado de um grafo não direcionado, na Figura 2.3, o conjunto de arestas A seria escrito como $\{ab, bc\}$.

Definição 9 (Grafo subjacente)**.** O grafo subjacente (*underlying*) de um grafo direcionado $G := (V, A)$ é o grafo $S := (V, A')$ onde $A' := \{vu \mid uv \in A\} \cup A$.

Definição 10 (Grafo Bipartido)**.** Dizemos que um grafo $G := (V, A)$ é bipartido (*bipartite*) se somente se existem dois conjuntos disjuntos V_1, V_2 tal que $V_1 \cup V_2 = V$, e que para todo $(u, v) \in A$, ($u \in V_1$ e $v \in V_2$) ou ($u \in V_2$ e $v \in V_1$), isto é, toda aresta em A conecta um vértice de V_1 a um vértice de V_2.

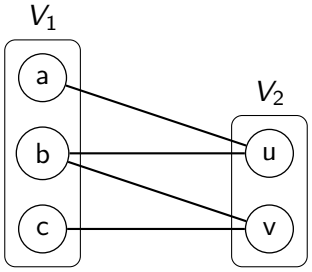


Figura 2.4: Um grafo bipartido com $V_1 := \{a, b, c\}$ e $V_2 := \{u, v\}$

Definição 11 (Grafo Biconexo)**.** Dizemos que um grafo $G := (V, A)$ é biconexo (*biconnected*) se somente se $|G| > 2$ e para todo $v \in V$, o subgrafo de G resultante da remoção de v é conexo, isto é, $G - \{v\}$ é conexo.

Capítulo 3

Representações de Grafos

Existem diversas formas de representar um grafo computacionalmente. Nesta seção serão explanadas as 3 estruturas que usamos ao decorrer do projeto, suas características e casos em que selecionar cada tipo é interessante. As representações de Lista de Adjacência, Matriz de Adjacência e Matriz de Incidência foram retiradas de **?** e de **?**.

Definição 12 (Matriz de Adjacência)**.** Uma matriz quadrada M é uma Matriz de Adjacência de um Grafo $G := \{V, A\}$ quando:

$$M_{ij} = \begin{cases} 1 & \text{se a aresta } (i,j) \in A, \\ 0 & \text{caso contrário} \end{cases}$$

Definição 13 (Matriz de Incidência)**.** Uma matriz M de dimensões $|V| \times |A|$ é uma Matriz de Incidência de um Grafo $G := \{V, A\}$ quando:

$$M_{ij} = \begin{cases} -1 & \text{se a aresta } j \text{ sai do vértice } i, \\ 1 & \text{se a aresta } j \text{ entra no vértice } i, \\ 0 & \text{caso contrário.} \end{cases}$$

Definição 14 (Lista de Adjacência)**.** Uma Lista de Adjacência L de um Grafo $G := \{V, A\}$ consiste em um vetor de n vetores, um para cada $v \in V$, onde cada "sub-vetor"guarda os vértices u com os quais $(v, u) \in A$. Assim, cada $L[v]$ contém todos os vértices adjacentes a v.

Em **?** a definição de Lista de Adjacência obriga que os "sub-vetores"armazenem ponteiros para vértices. Contudo, neste projeto optamos por simplificar a estrutura destes "sub-vetores"para armazenar diretamente os vértices.

3.1 Comparação

Nesta seção iremos analisar a eficiência das diferentes representações de grafos, utilizando as notações abstratas de Big O e Theta. Nesta comparação será excluída a representação por Matriz de Incidência, tendo em vista sua pouca utilização e praticidade para o andamento do projeto.

Tabela 3.1: Representações de grafos

Aspecto	**L.A**	**M.A**
Busca de aresta (x, y)	$O(n)$	$\Theta(1)$
Inserção / Remoção de aresta	$O(n)$	$\Theta(1)$
Busca em grafo	$\Theta(V + E)$	$\Theta(V^2)$
Grau de um vértice	$\Theta(1)$	$O(V)$
Uso de memória	$\Theta(V + E)$	$\Theta(V^2)$

Podemos concluir, portanto, que a Matriz de Adjacência tem como vantagem a eficiência para busca, inserção e deleção de arestas ($\Theta(1)$) pois acessa diretamente a posição em memória, mas não é adequada para grafos grandes dado o custo exponencial de espaço.

Por outro lado, a Lista de Adjacência tem como vantagem o melhor desempenho para travessia e maior economia em espaço, sendo uma escolha adequada para grafos de diversos tamanhos, mas apresenta desempenho pior para operações que alteram a estrutura do grafo.

A Matriz de Incidência, em geral, acaba sendo a escolha menos eficiente para qualquer situação. Portanto, ela obterá pouco destaque neste trabalho e em nossas implementações.

Capítulo 4

Descrição em Pseudocódigo dos Algoritmos da API

Neste capítulo, apresentamos a descrição, em linguagem de pseudocódigo, dos principais algoritmos definidos na especificação da API. O objetivo é proporcionar uma visão clara do funcionamento lógico das rotinas antes de sua implementação em Rust, estabelecendo um elo entre o modelo conceitual e o código-fonte.

Os algoritmos apresentados baseiam-se em uma abstração comum de estruturas de grafos, contemplando tanto grafos não dirigidos quanto grafos dirigidos (dígrafos). Enquanto os primeiros modelam relações bidirecionais entre vértices, os dígrafos representam relações assimétricas, nas quais as arestas possuem direção definida.

4.1 Dígrafos

O `trait Graph` é responsável por explicitar qual assinatura as funções que devem ser implementadas. Tal abordagem é coerente pois, como grafo se trata de um Tipo Abstrato de Dado, a implementação é uma particularização de cada Estrutura de Dados. E dentre a funções solicitadas para que uma estrutura possa implementar o trait grafo temos:

- Criar um grafo vazio
- Ordem e tamanho do grafo
- Nós do grafo(através de um iterador)
- Adicionar vértices e arestas
- Remover nós e arestas
- Vizinhos de um nós(através de um iterador)
- Determinar se o grafo é bipartido
- Grafo subjacente
- Se dois nós compartilham uma aresta
- Grau de um nó
- Iterador a partir de uma BFS

- Iterador a partir de uma DFS
- Classificar arestas

4.2 Grafos Não Direcionados

O `trait UndirectedGraph` estende o conceito de `Graph`, representando grafos em que as arestas não possuem direção. E uma vez que só se pode implementar caso `Graph` já tenha sido implementado anteriormente, a implementação das novas funções é trivial na maioria dos casos, pois recorre às implementações de `Graph`. A API define métodos específicos para:

- Adicionar vértices e arestas;
- Remover vértices e arestas;
- Obter vizinhos de um vértice;
- Determinar se o grafo é conexo
- Grau de um nó
- Iterador a partir de uma BFS
- Iterador a partir de uma DFS
- Identificar componentes biconexas(através de um iterador)

4.3 Descrição em Pseudocódigo dos Algoritmos

A seguir, apresentamos os algoritmos da API em pseudocódigo. O objetivo é destacar sua lógica fundamental, sem referência direta à sintaxe de Rust, mas mantendo correspondência com o comportamento esperado dos iteradores.

4.3.1 Busca em Profundidade (DFS)

A busca em profundidade percorre o grafo a partir de um vértice inicial, explorando recursivamente os caminhos até o limite de cada ramo.

Essa lógica permite identificar ciclos, classificar arestas e analisar conectividade de forma eficiente.

4.3.2 Busca em Largura (BFS)

A busca em largura percorre o grafo por camadas, processando primeiro todos os vértices a uma mesma distância do ponto inicial antes de avançar para os níveis seguintes. O pseudocódigo é apresentado abaixo:

O BFS é útil em problemas que requerem a descoberta de caminhos mínimos ou a análise de níveis de distância em grafos.

Algorithm 1 Busca em Profundidade

Entrada $\mathbf{G}(\mathbf{V}, \mathbf{A})$, $v \in \mathbf{V}$
Saída $predecessor = [\,]$ (Lista do vizinho de cada vértice)

```
function DFS(G(V, A), v)
    predecessor ← [ ]
    predecessor[v] ← nulo

    visitado ← [ ]
    visitado[v] ← 1

    pilha ← [ ]
    pilha ← empilhar(v)

    while pilha.tamanho() > 0 do
        u ← topo(pilha)
        if ∃ uw ∈ A(G) e visitado[w] ≠ 1 then
            predecessor[w] ← u
            visitado[w] ← 1
            pilha ← empilhar(w)
        else
            pilha ← desempilhar()
        end if
    end while
    return predecessor
end function
```

Algorithm 2 Busca em Largura

Entrada $\mathbf{G}(\mathbf{V}, \mathbf{A})$, $v \in \mathbf{V}$
Saída $predecessor = [\,]$ (Lista do vizinho de cada vértice)

```
1:  function BFS(G(V, A), v)
2:      predecessor ← [ ]
3:      predecessor[v] ← nulo

4:      visitado ← [ ]
5:      visitado[v] ← 1

6:      fila ← [ ]
7:      fila ← enfileirar(v)

8:      while fila.tamanho() > 0 do
9:          u ← começo(fila)
10:         fila ← desenfileirar()
11:         for v ∈ u.vizinhos() do
12:             if visitado[v] ≠ 1 then
13:                 predecessor[v] ← u
14:                 visitado[v] ← 1
15:                 fila ← enfileirar(v)
16:             end if
17:         end for
18:     end while
19:     return predecessor
20: end function
```

4.3.3 Classificação de Arestas

Durante a DFS, é possível classificar as arestas em diferentes categorias, que auxiliam na análise estrutural de dígrafos (grafos orientados):

- **Tree Edge (Árvore):** Conecta um vértice a um filho na DFS.
- **Back Edge (Retorno):** Conecta um vértice a um ancestral, indicando ciclo.
- **Forward Edge (Avanço):** Conecta um vértice a um descendente não filho.
- **Cross Edge (Cruzamento):** Conecta vértices em diferentes ramos da DFS.

4.3.4 Componentes Biconexas

O algoritmo permite identificar componentes biconexas em grafos não direcionados. Mantém pilhas de arestas e tempos de descoberta para determinar subconjuntos de vértices que formam componentes biconexas.

No capítulo seguinte os conceitos abordados neste serão implementados para Lista de Adjacência, Matrix de Adjacência e Matriz de Incidência.

Algorithm 3 Componentes Bicônexas

Entrada **G**(**V**, **A**)
Saída Conjunto das componentes bicônexas de **G**

```
1:  visitado ← [ ]
2:  descoberta ← [ ]
3:  baixo ← [ ]
4:  pilha ← [ ]
5:  componentes ← [ ]
6:  tempo ← 0

7:  function biconnected(G(V, A))
8:      for v ∈ V do
9:          if visitado[v] ≠ 1 then
10:             dfs_biconnected(G, v, pai = nulo)
11:         end if
12:     end for
13:     return componentes
14: end function

15: function dfs_biconnected(G, v, pai)
16:     visitado[v] ← 1
17:     tempo ← tempo + 1
18:     descoberta[v] ← tempo
19:     baixo[v] ← tempo

20:     for u ∈ v.vizinhos() do
21:         if visitado[u] ≠ 1 then
22:             pilha ← empilhar((v,u))
23:             dfs_biconnected(G, u, v)
24:             baixo[v] ← min(baixo[v], baixo[u])
25:             if baixo[u] ≥ descoberta[v] then
26:                 temp ← [ ]
27:                 while pilha.topo() ≠ (v, u) do
28:                     pilha ← desempilhar()
29:                     temp ← adicionar((v',u'))
30:                 end while
31:                 componentes ← temp
32:             end if
33:         else if u ≠ pai  e  descoberta[u] < descoberta[v] then
34:             pilha ← empilhar((v,u))
35:             baixo[v] ← min(baixo[v], descoberta[u])
36:         end if
37:     end for
38: end function
```

Capítulo 5

Implementação

Nesse capítulo serão explicadas as particularidades de cada implementacão das funcões mencionadas no capítulo anterior. Também serão elucidados alguns aspectos exclusivos da linguagem Rust, primeiramente abordando algumas decisões de arquitetura e em seguida falando sobre as 3 estruturas de dados implementadas.

5.1 Arquitetura

Como já citado no capítulos anterior, dado que haveriam implementações de funções em comum para estruturas de dados diferentes, grafo e dígrafo foram implementados através de `traits`, da seguinte forma:

```
pub trait Graph<Node: Eq + Hash + Copy> {
    fn new_empty() -> Self;

    fn order(&self) -> usize;

    fn size(&self) -> usize;

    fn node_degrees(&self, n: Node) -> (usize, usize);

    fn nodes(&self) -> impl Iterator<Item = Node>;

    fn add_node(&mut self, n: Node);

    fn remove_node(&mut self, n: Node);

    fn add_edge(&mut self, n: Node, m: Node);

    fn remove_edge(&mut self, n: Node, m: Node);

    type Neighbors<'a>: Iterator<Item = Node>
    where
        Self: 'a,
        Node: 'a;
    fn neighbors<'a>(&'a self, n: Node) -> Self::Neighbors<'a>;

    fn biparted(&self) -> bool;

    fn underlying_graph(&self) -> Self;

    fn has_edge(&self, n: Node, m: Node) -> bool {
        self.neighbors(n).any(|neighbor| neighbor == m)
```

```
32      }
33
34      fn dfs(&self, start: Node) -> DfsIter<'_, Node, Self>
35      where
36          Self: Sized,
37      {
38          DfsIter::new(self, start)
39      }
40
41      fn bfs(&self, start: Node) -> BfsIter<'_, Node, Self>
42      where
43          Self: Sized,
44      {
45          BfsIter::new(self, start)
46      }
47
48      fn classify_edges(&self, start: Node) -> DfsEdgesIter<'_, Node, Self>
49      where
50          Self: Sized,
51      {
52          DfsEdgesIter::new(self, start)
53      }
54  }
```

Código 5.1: Implementação do trait Graph

As particularidades da sintaxe acima são que, `Graph` pode ser implementado para qualquer tipo genérico `N`, desde que este implemente os seguintes `traits`:

- **Eq**: Que dá ao tipo a propriedade de igualdade através do operador $==$
- **Hash**: Propriedade necessária para que aquele tipo possa ser usado em estruturas de dados como `HashMap/HashSet`. Que serão usadas em trabalhos futuros.
- **Copy**: Que dá ao tipo a propriedade de ser copiável através do operador $=$, o que apesar de parecer trivial, não é. E isso acontece pois Rust introduz o conceito de Klabnik et al. (2025*c*), que faz com que nem todo tipo seja copiável.

Feito isso, basta com que o tipo implementa as funções com as assinaturas solicitadas e ele terá o `trait Graph`. E além dele, também temos outro `trait` que tem ele como pré-requisito para implementação, o `UndirectedGraph`:

```
1  pub trait UndirectedGraph<Node: Copy + Eq + Hash>: Graph<Node> {
2      fn undirected_size(&self) -> usize;
3
4      fn connected(&self) -> bool;
5
6      fn biconnected_components(&self, start: Node) ->
       BiconnectedComponentsIter<'_, Node, Self>
7      where
8          Self: Sized,
9      {
10         BiconnectedComponentsIter::new(self, start)
11     }
12
13     fn add_undirected_edge(&mut self, n: Node, m: Node) {
14         self.add_edge(n, m);
15         self.add_edge(m, n);
16     }
17
```

```
18      fn remove_undirected_edge(&mut self, n: Node, m: Node) {
19          self.remove_edge(n, m);
20          self.remove_edge(m, n);
21      }
22
23      fn undirected_node_degree(&self, n: Node) -> usize {
24          self.neighbors(n).count()
25      }
26
27      fn classify_undirected_edges<'a>(&'a self, start: Node) -> impl
        Iterator<Item = Edge<Node>>
28      where
29          Self: Sized,
30          Node: 'a,
31      {
32          DfsEdgesIter::new(self, start)
33              .filter(|edge| matches!(edge, Edge::Tree(_, _) | Edge::Back(_,
         _)))
34      }
35  }
```

Código 5.2: Implementação do trait UndirectedGraph

Note que o `trait` já tem várias implementações, e isso só é possível pois ele usa as funcões já implementadas em `Graph`. Por isso que implementar `UndirectedGraph`, antes, é necessário implementar esse.

Mas feitas as considerações iniciais acerca da arquitetura da implementação, podemos abordar as particularidades de cada Estrutura da Dados.

5.2 Matriz de Adjacência

A Matriz de Adjacência, conforme falado anteriormente, é uma das formas mais comuns de se implementar um grafo. Muito disso em decorrência de sua baixa complexidade na inserção e no acesso nos elementos. Sendo assim, ela é implementada da seguinte forma:

```
1  #[derive(Debug, Clone)]
2  pub struct AdjacencyMatrix(pub Vec<Vec<usize>>);
```

Código 5.3: Implementação da Estrutura de Dados Matriz de Adjacência

No código acima, o `derive` é uma instrução para o compilador inferir como adicionar as pripriedades `Debug` e `Clone` na estrutura abaixo. Estrutura essa que consiste em um vetor de vetores de `usize`(inteiros não-negativos, tipo usado para fins de simplificação, uma vez que temos mais garantias sobre as propridades dele dessa forma, diferente de em tipos genéricos). E abaixo segue a implementação das funções solicitadas pelo `trait UndirectedGraph`.

```
1       fn undirected_size(&self) -> usize {
2           let mut size = 0;
3           for i in 0..self.order() {
4               for j in 0..=i {
5                   if self.0[i][j] > 0 {
6                       size += 1;
7                   }
8               }
9           }
10          size
11      }
12
13      fn connected(&self) -> bool {
```

```
        let n = self.order();
        if n == 0 {
            return true;
        }

        let mut visited = vec![false; n];
        let mut stack = vec![0];
        visited[0] = true;

        while let Some(u) = stack.pop() {
            for (v, &is_edge) in self.0[u].iter().enumerate() {
                if is_edge > 0 && !visited[v] {
                    visited[v] = true;
                    stack.push(v);
                }
            }
        }

        visited.into_iter().all(|v| v)
    }

    fn undirected_node_degree(&self, node: usize) -> usize {
        if let Some(row) = self.0.get(node) {
            row.iter().filter(|&&val| val != 0).count()
        } else {
            0
        }
    }
}
```

Código 5.4: Implementação de UndirectedGraph na Estrutura de Dados Matriz de Adjacência

Note que ele só define as funções que não tem implementação padrão. E algo que também importante para se pontuar é que a palavra `mut`(abreviação de "mutável") é requerida sempre que desejamos alterar o valor da variável em questão, pois devemos fazer isso de forma explícita, e isso acontece por conta do conceito de Klabnik et al. (2025*c*). E abaixo segue a implementação de `Graph`:

```
impl Graph<usize> for AdjacencyMatrix {
    fn new_empty() -> Self {
        AdjacencyMatrix(vec![])
    }

    fn order(&self) -> usize {
        self.0.len()
    }

    fn size(&self) -> usize {
        self.0
            .iter()
            .enumerate()
            .map(|(i, _)| self.neighbors(i).count())
            .sum()
    }

    fn node_degrees(&self, n: usize) -> (usize, usize) {
        let out_deg = self.0[n].iter().filter(|&&v| v != 0).count();
        let in_deg = self.0.iter().filter(|row| row[n] != 0).count();
        (in_deg, out_deg)
    }

```

```
24      fn nodes(&self) -> impl Iterator<Item = usize> {
25          0..self.order()
26      }
27
28      fn add_node(&mut self, _n: usize) {
29          self.0.push(Vec::new());
30          let new_order = self.order();
31
32          for r in &mut self.0 {
33              while r.len() < new_order {
34                  r.push(0);
35              }
36          }
37      }
38
39      fn remove_node(&mut self, n: usize) {
40          if n < self.0.len() {
41              self.0.remove(n);
42              for row in self.0.iter_mut() {
43                  for idx in n + 1..row.len() {
44                      row[idx - 1] = row[idx];
45                  }
46                  row.pop();
47              }
48          }
49      }
50
51      fn add_edge(&mut self, n: usize, m: usize) {
52          if let Some(edges) = self.0.get_mut(n)
53              && let Some(edge) = edges.get_mut(m)
54          {
55              if *edge == 1 {
56                  return;
57              }
58              *edge = 1;
59          }
60      }
61
62      fn remove_edge(&mut self, n: usize, m: usize) {
63          if let Some(edges) = self.0.get_mut(n)
64              && let Some(edge) = edges.get_mut(m)
65          {
66              *edge = 0;
67          }
68      }
69
70      type Neighbors<'a> = std::iter::FilterMap<
71          std::iter::Enumerate<std::slice::Iter<'a, usize>>,
72          fn((usize, &'a usize)) -> Option<usize>,
73      >;
74
75      fn neighbors<'a>(&'a self, n: usize) -> Self::Neighbors<'a> {
76          fn filter_fn((i, &weight): (usize, &usize)) -> Option<usize> {
77              if weight != 0 { Some(i) } else { None }
78          }
79          match self.0.get(n) {
80              Some(row) => row.iter().enumerate().filter_map(filter_fn),
81              None => [].iter().enumerate().filter_map(filter_fn),
82          }
83      }
84
```

```rust
    fn biparted(&self) -> bool {
        let n = self.order();
        if n == 0 {
            return true;
        }

        let mut side = vec![None; n]; // None = uncolored, Some(0/1) =
    partition
        let mut queue = std::collections::VecDeque::new();

        for start in 0..n {
            // skip already colored components
            if side[start].is_some() {
                continue;
            }

            side[start] = Some(0);
            queue.push_back(start);

            while let Some(u) = queue.pop_front() {
                let u_side = side[u].unwrap();

                for (v, &is_edge) in self.0[u].iter().enumerate() {
                    if is_edge == 0 {
                        continue;
                    }

                    if side[v].is_none() {
                        side[v] = Some(1 - u_side);
                        queue.push_back(v);
                    } else if side[v] == Some(u_side) {
                        return false; // adjacent nodes with same color
                    }
                }
            }
        }

        true
    }

    fn underlying_graph(&self) -> Self {
        let mut matrix: AdjacencyMatrix =
            AdjacencyMatrix(vec![vec![0; self.0.len()]; self.0.len()]);

        for (idx_r, row) in self.0.iter().enumerate() {
            for (idx_c, col) in row.iter().enumerate() {
                if *col == 1 && !matrix.has_edge(idx_c, idx_r) {
                    matrix.add_undirected_edge(idx_r, idx_c);
                }
            }
        }

        matrix
    }
}
```

Código 5.5: Implementação de Graph na Estrutura de Dados Matriz de Adjacência

Perceba que funções como `underlying graph` recorrem a funções definidas em `UndirectedGraph`, e isso ocorre pois os `traits` tem acesso mútuo entre as funções definidas pelos tipos que os implementam.

5.3 Lista de Adjacência

Juntamente com a Matriz de Adjacência, a lista de Adjacência é uma das implementações mais comum usadas para grafos. Sua natureza de tamanho dinâmico é usada principalmente para problemas que envolvem muitos vértices e arestas, sendo assim, sua implementação em Rust se dá da seguinte forma:

```rust
#[derive(Debug, Clone, Default)]
pub struct AdjacencyList(pub Vec<Vec<usize>>);
```

Código 5.6: Implementação de Graph na Estrutura de Dados Matriz de Adjacência

Perceba que é análoga à matriz de adjacência, com exceção do `Default` que cria um comportamento padrão de instanciar a estrutura através de um vetor vazio. E sua implementação de `UndirectedGraph` é a seguinte:

```rust
impl UndirectedGraph<usize> for AdjacencyList {
    fn undirected_size(&self) -> usize {
        let mut self_loops = 0;
        let regular_edges: usize = self
            .0
            .iter()
            .enumerate()
            .map(|(i, _)| {
                self.neighbors(i)
                    .filter(|&n| {
                        let is_self_loop = n == i;
                        self_loops += is_self_loop as usize;
                        !is_self_loop
                    })
                    .count()
            })
            .sum();
        regular_edges / 2 + self_loops
    }

    fn connected(&self) -> bool {
        for i in 0..self.order() {
            if self
                .dfs(i)
                .filter(|event| matches!(event, DfsEvent::Discover(_, _)))
                .count()
                != self.order()
            {
                return false;
            }
        }
        true
    }

    fn undirected_node_degree(&self, node: usize) -> usize {
        self.0
            .get(node)
            .map(|neighbors| neighbors.len())
            .unwrap_or(0)
    }
}
```

Código 5.7: Implementação de Graph na Estrutura de Dados Matriz de Adjacência

Perceba que, dada a natureza de continuidade de "valores significativos"(ao contrário da Matriz de Adjacência, que, por vezes, tem várias ocorrências de 0), a Estrutura de Dados

em questão é muito iterável. Sendo assim, o conteúdo da maioria das funções consiste na chamada de vários métodos em sequência. E o mesmo se aplica para a implementacão de `Graph`, que recorer muito pouco a laços:

```rust
impl Graph<usize> for AdjacencyList {
    fn new_empty() -> Self {
        AdjacencyList(vec![])
    }

    fn order(&self) -> usize {
        self.0.len()
    }

    fn size(&self) -> usize {
        self.0.iter().map(|neighbors| neighbors.len()).sum()
    }

    fn node_degrees(&self, n: usize) -> (usize, usize) {
        let out_deg = self.0.get(n).map_or(0, |neighbors| neighbors.len())
    ;
        let in_deg = self
            .0
            .iter()
            .filter(|neighbors| neighbors.contains(&n))
            .count();
        (in_deg, out_deg)
    }

    fn nodes(&self) -> impl Iterator<Item = usize> {
        0..self.order()
    }

    fn add_node(&mut self, _n: usize) {
        self.0.push(Vec::new());
    }

    fn remove_node(&mut self, n: usize) {
        if n < self.0.len() {
            self.0.remove(n);
            for neighbors in self.0.iter_mut() {
                neighbors.retain(|&x| x != n);
                for x in neighbors.iter_mut() {
                    if *x > n {
                        *x -= 1;
                    }
                }
            }
        }
    }

    fn add_edge(&mut self, n: usize, m: usize) {
        if self.0.get(m).is_some()
            && let Some(n_edges) = self.0.get_mut(n)
            && !n_edges.contains(&m)
        {
            n_edges.push(m);
        }
    }

    fn remove_edge(&mut self, n: usize, m: usize) {
        if let Some(edges) = self.0.get_mut(n) {
```

```
57              edges.retain(|&x| x != m);
58          }
59      }
60
61      type Neighbors<'a> = std::iter::Copied<std::slice::Iter<'a, usize>>;
62
63      fn neighbors<'a>(&'a self, n: usize) -> Self::Neighbors<'a> {
64          match self.0.get(n) {
65              Some(edges) => edges.iter().copied(),
66              None => [].iter().copied(),
67          }
68      }
69
70      fn biparted(&self) -> bool {
71          let n = self.order();
72          if n == 0 {
73              return true;
74          }
75
76          let mut side = vec![None; n];
77
78          for start in 0..n {
79              if side[start].is_some() {
80                  continue;
81              }
82              side[start] = Some(0);
83              let mut queue = std::collections::VecDeque::new();
84              queue.push_back(start);
85
86              while let Some(u) = queue.pop_front() {
87                  let u_side = side[u].unwrap();
88                  for v in self.neighbors(u) {
89                      if side[v].is_none() {
90                          side[v] = Some(1 - u_side);
91                          queue.push_back(v);
92                      } else if side[v] == Some(u_side) {
93                          return false;
94                      }
95                  }
96              }
97          }
98
99          true
100     }
101
102     fn underlying_graph(&self) -> Self {
103         let mut list = AdjacencyList(vec![Vec::new(); self.0.len()]);
104
105         for (idx_r, row) in self.0.iter().enumerate() {
106             for &col in row.iter() {
107                 if !list.has_edge(idx_r, col) {
108                     list.add_undirected_edge(idx_r, col);
109                 }
110             }
111         }
112         list
113     }
114 }
```

Código 5.8: Implementação de Graph na Estrutura de Dados Matriz de Adjacência

Algo válido de se ressaltar é que o uso de `Some` e `None` ocorre pois, por vezes, funções retornam um `Result` que pode ou não conter um valor válido, e esses são os contrutores do tipo. Sendo assim, a linguagem opta por encapsular dessa forma o resultado de certas funções, ao invés de lançar uma exceção(o que geralmente é feito por outras linguagens).

5.4 Matriz de Incidência

Por motivos já citados, esta se trata de uma das abordagens menos interessantes. Ela tem o consumo de memória semelhante a uma Matriz de Adjacência e uma acesso com custo de uma Lista de Adjacência. Mesmo assim, segue a implementação:

```
impl UndirectedGraph<usize> for IncidenceMatrix {
    fn undirected_size(&self) -> usize {
        self.0.len()
    }

    fn connected(&self) -> bool {
        todo!()
    }

    fn undirected_node_degree(&self, vertex: usize) -> usize {
        if self.0.is_empty() || vertex >= self.0[0].len() {
            return 0;
        }

        self.0.iter().filter(|row| row[vertex] != 0).count()
    }
}
```

Código 5.9: Implementação de UndirectedGraph na Estrutura de Dados Matriz de Incidência

```
impl Graph<usize> for IncidenceMatrix {
    fn new_empty() -> Self {
        IncidenceMatrix(Vec::new())
    }

    fn order(&self) -> usize {
        if self.0.is_empty() {
            0
        } else {
            self.0[0].len()
        }
    }

    fn size(&self) -> usize {
        self.0.len()
    }

    fn node_degrees(&self, _n: usize) -> (usize, usize) {
        todo!()
    }

    fn nodes(&self) -> impl Iterator<Item = usize> {
        0..self.order()
    }

    fn add_node(&mut self, _n: usize) {
        todo!()
    }
```

```
29
30      fn remove_node(&mut self, _n: usize) {
31          todo!()
32      }
33
34      fn add_edge(&mut self, _n: usize, _m: usize) {
35          todo!()
36      }
37
38      fn remove_edge(&mut self, _n: usize, _m: usize) {
39          todo!()
40      }
41
42      type Neighbors<'a> = std::iter::FilterMap<
43          std::iter::Enumerate<std::slice::Iter<'a, usize>>,
44          fn((usize, &'a usize)) -> Option<usize>,
45      >;
46
47      fn neighbors<'a>(&'a self, _n: usize) -> Self::Neighbors<'a> {
48          todo!()
49      }
50
51      fn bipartite(&self) -> bool {
52          let n = self.order();
53          if n == 0 {
54              return true;
55          }
56
57          let mut adj = vec![Vec::new(); n];
58
59          for edge in &self.0 {
60              let endpoints: Vec<usize> = edge
61                  .iter()
62                  .enumerate()
63                  .filter_map(|(v, &x)| if x != 0 { Some(v) } else { None })
64                  .collect();
65
66              match endpoints.as_slice() {
67                  [u, v] => {
68                      adj[*u].push(*v);
69                      adj[*v].push(*u);
70                  }
71                  [u] => {
72                      adj[*u].push(*u);
73                  }
74                  _ => {}
75              }
76          }
77
78          let mut partition = vec![None; n];
79          let mut queue = std::collections::VecDeque::new();
80
81          for start in 0..n {
82              if partition[start].is_some() {
83                  continue;
84              }
85
86              partition[start] = Some(0);
87              queue.push_back(start);
88
89              while let Some(u) = queue.pop_front() {
```

```
90                  let u_side = partition[u].unwrap();
91
92                  for &v in &adj[u] {
93                      if partition[v].is_none() {
94                          partition[v] = Some(1 - u_side);
95                          queue.push_back(v);
96                      } else if partition[v] == Some(u_side) {
97                          return false;
98                      }
99                  }
100             }
101         }
102
103         true
104     }
105
106     fn underlying_graph(&self) -> Self {
107         todo!()
108     }
109 }
```

Código 5.10: Implementação de Graph na Estrutura de Dados Matriz de Incidência

As funções que tem como conteúdo `todo!()` têm implementação pendente. Mas algo relevante a se falar é que todas as Estruturas de Dados tem funções de conversão entre si, ou seja:

```
1     pub fn IncidenceMatrix::from_adjacency_matrix(matrix: &AdjacencyMatrix
      ) -> Self;
2     pub fn IncidenceMatrix::from_adjacency_list(matrix: &AdjacencyList) ->
       Self;
```

Código 5.11: Funções de Conversão

Dito isso, através de composição de funções, todos os resultados de uma instância de `IncidenceMatrix` poderiam ser obtivos através disso.

Capítulo 6

Testes de performance

Nessa seção vamos comparar o desempenho da busca em profundidade implementada em Rust com uma versão tradicional implementada em C++, usando bibliotecas *benchmarking* escritas em Rust. O objetivo é mostrar que apesar do alto nível de abstração e flexibilidade da nossa implementação, o desempenho ainda fica próximo de versões tradicionais do algoritmo que não tem a mesma flexibilidade, implementadas em C++.

6.1 Metodologia

Os testes realizados vivem sob duas principais categorias, *Micro Benchmarking* de alta precisão e *Micro Benchmarking* estatístico. Micro benchmarking de alta precisão executa apenas uma vez pedaço de código e mensura estatísticas como, quantidade de instruções realizadas pelo processador, acesso ao cache e uso de RAM. Enquanto, o micro benchmarking estatístico, se refere a múltiplas execuções de um pedaço de código, onde é realizada a coletas de várias amostras, nesse tipo de teste, é mensurado medidas de tendência central e dispersão, como a média e desvio padrão do tempo de execução. Para facilitar a realização desses testes e a coleta de dados, importamos duas bibliotecas públicas escritas em Rust, *Criterion* (n.d.) e *Gungraun* (n.d.) para o Micro Benchmark estatístico e Micro Benchmark de alta precisão respectivamente.

Para viabilizar a coleta de dados de funções em C++ usando as bibliotecas implementadas em Rust, foi necessário criar uma FFI (Foreign Function Interface) para permitir que código C++ interaja com o código das bibliotecas em Rust. Essa implementação, se deu especificando um bloco `extern "C"` para permitir que as funções sejam identificadas unicamente pelo seu nome, por exemplo:

```
1 extern "C" {
2 void* mk_adjacency_list(size_t node_amt) { ... }
3
4 void add_edge_unchecked(void* graph, size_t n, size_t m) { ... }
5
6 void dfs(void* graph, size_t start) { ... }
7 }
```

Código 6.1: Exemplo de interface FFI escrita em C++

Feito isso, fomos capazes de importar e usar essa interface em código Rust e implementar um encapsulador:

```
1 unsafe extern "C" {
2     fn mk_adjacency_list(node_amt: usize) -> *mut std::ffi::c_void;
```

```
3       fn add_edge_unchecked(graph: *mut std::ffi::c_void, n: usize, m: usize
        );
4       fn dfs(graph: *mut std::ffi::c_void, start: usize);
5   }
6
7   pub struct AdjacencyListCpp {
8       ptr: *mut std::ffi::c_void,
9   }
10
11  impl AdjacencyListCpp {
12      pub fn new(node_amt: usize) -> Self { ... }
13
14      pub fn add_edge_unchecked(&self, n: usize, m: usize) { ... }
15
16      pub fn dfs(&self, start: usize) { ... }
17  }
```

Código 6.2: Exemplo de uso de interface FFI escrita em Rust

Exemplos completos de código são acessíveis no caminho https://github.com/mpazmarcato/Trabalho-U1-Grafos/tree/main/crates/cpp-api.

6.1.1 Micro Benchmarking estatístico

O processo de Micro Benchmarking implementado pela biblioteca *Criterion* (n.d.) funciona em quatro etapas, Aquecimento, Medição, Análise e Comparação com o resultado anterior (*Analysis Process - Criterion.rs Documentation — bheisler.github.io*, n.d.). Entretanto, não levaremos em conta o resultado da última etapa. A primeira etapa serve para aquecer o cache da CPU, de forma que seja reduzida a quantidade de casos atípicos na amostra (*Analysis Process - Criterion.rs Documentation — bheisler.github.io*, n.d.). A segunda etapa realiza a medição dos tempos de execução e a terceira etapa realiza o cálculo de estatísticas como média, moda e mediana dos dados medidos.

A biblioteca também gera gráficos automaticamente usando o programa Gnuplot. A biblioteca também argumenta que as amostras de bom benchmark deve formam uma linha de regressão quando plotadas num gráfico (*Analysis Process - Criterion.rs Documentation — bheisler.github.io*, n.d.).

6.1.2 Micro Benchmarking de alta precisão

O processo de Micro Benchmarking implementado pela biblioteca *Gungraun* (n.d.), funciona de uma maneira bastante diferente do Criterion, este executa um trecho de código apenas uma vez e coleta as estatísticas usando o Valgrind. Estas estatísticas incluem quantidade estimada de ciclos da CPU, quantidade de instruções realizadas, quantos acertos de cache L1 e L2 e outros. O processo de configuração e uso é praticamente idêntico ao do Criterion.

6.2 Implementação dos testes

Vamos realizar testes apenas para a Busca em profundidade. Todos os testes serão realizados numa lista de adjacência em grafos completos, ou seja, que há arestas entre todos os nós. Essa escolha advém da ideia de que queremos maximizar o uso de recursos e quantidade de instruções, possivelmente evidenciando as diferenças que favorecem a implementação em C++.

Como não foi apresentada anteriormente, aqui está a implementação da busca em profundidade em C++. Definimos uma classe para a lista de adjacência em C++ e realizamos a

implementação tradicional da busca, tomando o cuidado para pré-alocar a pilha e o dicionário de visitados, assim como a implementação em Rust e para evitar alocações no meio da busca.

```cpp
class AdjacencyList {
   private:
    std::vector<std::vector<size_t>> data;

   public:
    AdjacencyList(size_t node_amt) : data(node_amt) {}

    size_t order() { return data.size(); }

    std::vector<size_t>& neighbors(const size_t node) {
        static std::vector<size_t> empty;
        return (node < order()) ? data[node] : empty;
    }

    void dfs(const size_t start) {
        std::vector<size_t> stack;
        std::unordered_set<size_t> visited;

        stack.reserve(order());
        visited.reserve(order());

        stack.push_back(start);
        visited.insert(start);

        while (!stack.empty()) {
            const auto current = stack.back();
            stack.pop_back();

            for (const auto& neighbor : neighbors(current)) {
                if (visited.insert(neighbor).second) {
                    stack.push_back(neighbor);
                }
            }
        }
    }
};
```

Código 6.3: Implementação da busca em profundidade em C++

O código fonte da geração do grafo e da criação dos testes é a seguinte:

```rust
pub fn create_complete_graph(size: usize) -> AdjacencyMatrix {
    let mut matrix = vec![vec![0; size]; size];
    for (i, row) in matrix.iter_mut().enumerate() {
        for (j, val) in row.iter_mut().enumerate() {
            if i != j {
                *val = 1;
            }
        }
    }
    AdjacencyMatrix(matrix)
}
```

Código 6.4: Código fonte da geração do grafo completo

6.2.1 Micro Benchmarking estatístico

Realizamos 3 testes performando a busca num grafo completo, os testes se diferenciam pela ordem do grafo gerado. Foram gerados grafos com 500 nós, 1000 e 2000 nós.

```rust
fn bench_dfs_comparison(c: &mut Criterion) {
    let sizes = vec![500, 1000, 2000];

    let mut group = c.benchmark_group("dfs");

    for size in sizes {
        let g = create_complete_graph(size);
        let rust_list = AdjacencyList::from_adjacency_matrix(&g);
        let cpp_list = AdjacencyListCpp::from_adjacency_matrix(&g);

        group.bench_with_input(
            BenchmarkId::new("rust", size),
            &size,
            |b, _| b.iter(|| rust_list.dfs(0).for_each(|_| ()))
        );

        group.bench_with_input(
            BenchmarkId::new("cpp", size),
            &size,
            |b, _| b.iter(|| cpp_list.dfs(0))
        );
    }

    group.finish();
}

criterion_group! {
    name = benches;
    config = Criterion::default()
        .measurement_time(std::time::Duration::from_secs(30));
    targets = bench_dfs_comparison
}

criterion_main!(benches);
```

Código 6.5: Código fonte dos testes de Micro Benchmark estatístico da DFS.

6.2.2 Micro Benchmarking de alta precisão

No Micro Benchmark de alta precisão, foi realizado apenas um teste da busca em profundidade para um grafo completo com 1000 nós. É importante ressaltar que a parte do código que é responsável pela inicialização do grafo também é mensurada no teste, pois é uma limitação da biblioteca. O código que implementa o Micro Benchmark é o seguinte:

```rust
fn setup() -> AdjacencyList {
    AdjacencyList::from_adjacency_matrix(&create_complete_graph(1000))
}

fn cpp_setup() -> AdjacencyListCpp {
    AdjacencyListCpp::from_adjacency_matrix(&create_complete_graph(1000))
}

#[library_benchmark]
fn single_shot_dfs() {
    setup().dfs(black_box(0)).for_each(|_| ())
}

#[library_benchmark]
fn single_shot_cpp_dfs() {
    cpp_setup().dfs(black_box(0))
```

```
17 }
18
19 library_benchmark_group! {
20     name = bench_dfs_group;
21     benchmarks = single_shot_dfs, single_shot_cpp_dfs
22 }
23
24 main!(library_benchmark_groups = bench_dfs_group);
```

Código 6.6: Código dos testes de Micro Benchmark de alta precisão da implementação da DFS.

Na código 6.6, Note o uso da função `black_box` na chamada das duas buscas (linhas 11 e 16), ela serve para que o compilador não pré-calcule a computação realizada pela DFS.

O mesmo não precisa no Micro Benchmark estatístico pois o tipo da variável `group` na código 6.5 já garante isso automaticamente.

6.3 Resultados

6.3.1 Especificações da máquina usada nos testes

Para realizar os testes, usamos um computador com as seguintes configurações:

Tabela 6.1: Especificações do sistema de teste

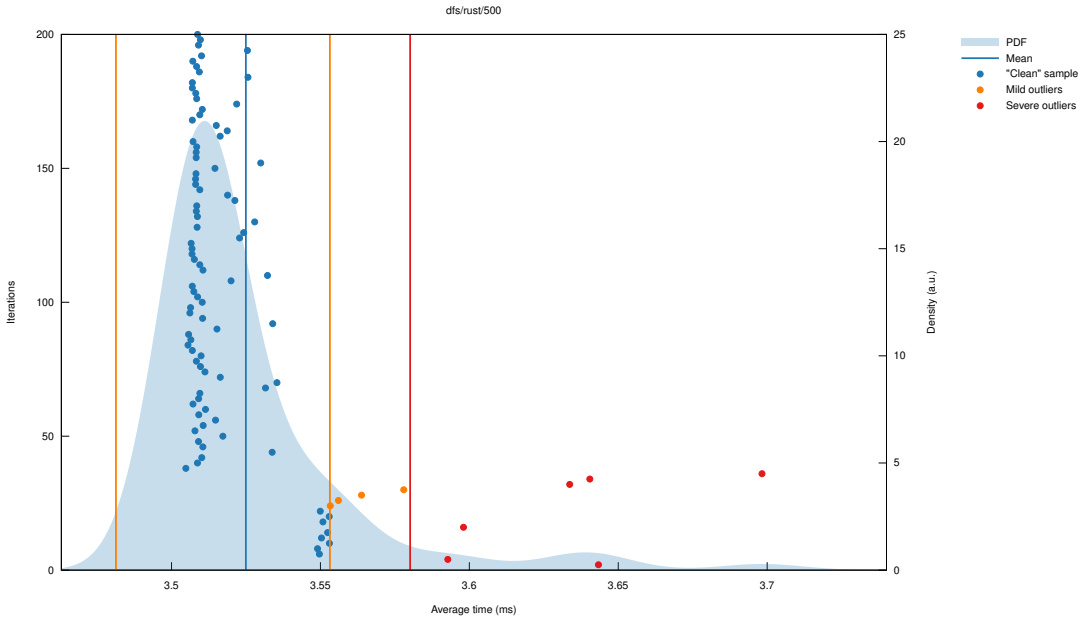
Hardware	Processador	AMD Ryzen 9 5900XT
	GPU	Nvidia Geforce RTX 4060 Ti
	RAM	32GB RAM
Software	Sistema Operacional	Arch Linux x86_64
	Kernel	Linux 6.17.1-arch-1-1
	Compilador Rust	rustc 1.90.0 (2025-09-14)
	Versão do Cargo	1.90.0 (2025-07-30)
	Compilador C++	clang 20.1.8
	Versão do Valgrind	3.25.1

6.3.2 Busca em profundidade

Micro Benchmark estatístico

Começaremos pelos testes de Micro Benchmarking estatístico, que exibem estatísticas sobre o tempo de execução dos algoritmos coletadas de várias iterações, ou amostras. Como as proporções das versões de um grafo com 500, 1000 e 2000 nós foram parecidas, nesse texto exibiremos apenas o resultado do benchmark de um grafo com 500 nós. Portanto, tivemos:

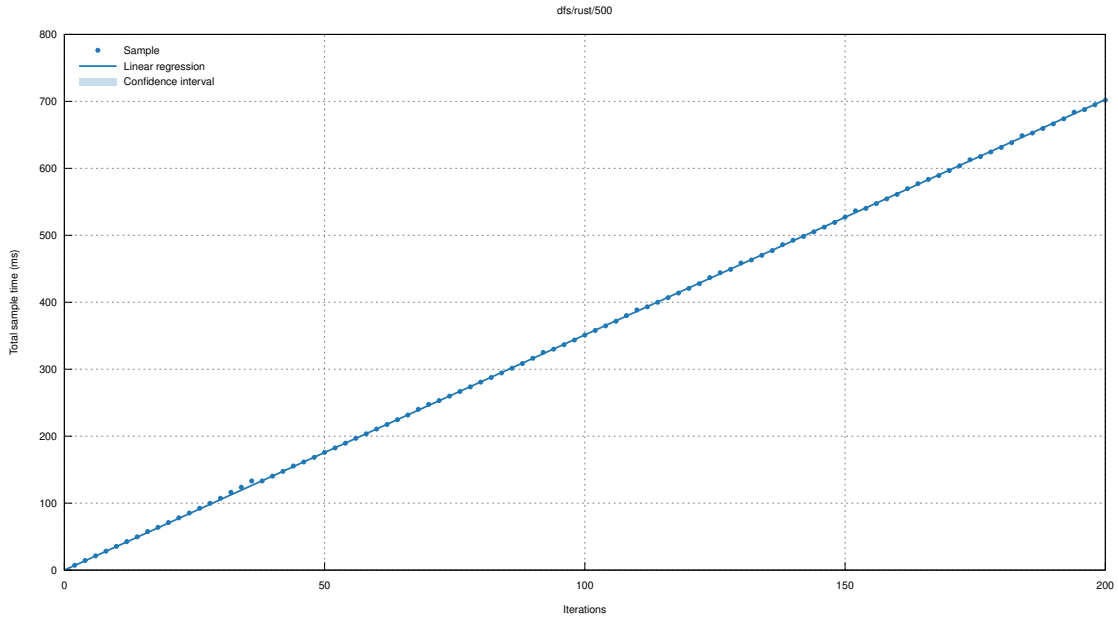
Figura 6.1: Gráfico de inclinação do Micro Benchmark estatístico da DFS em Rust.



Na figura 6.1, os pontos azuis são amostras "limpas", os pontos vermelhos representam casos atípicos sevéros, os pontos amarelos representam casos atípicos médios, seguindo o método de Tukey (*Criterion*, n.d.), as retas amarelas e vermelhas representam as cotas superiores e inferiores dessas medidas. O fundo azul representa a função de probabilidade desse teste e a reta azul representa a média, nesse caso, em torno de 3.54 ms.

Para validar se o benchmark foi bom, analisemos a reta de regressão das amostras desse teste:

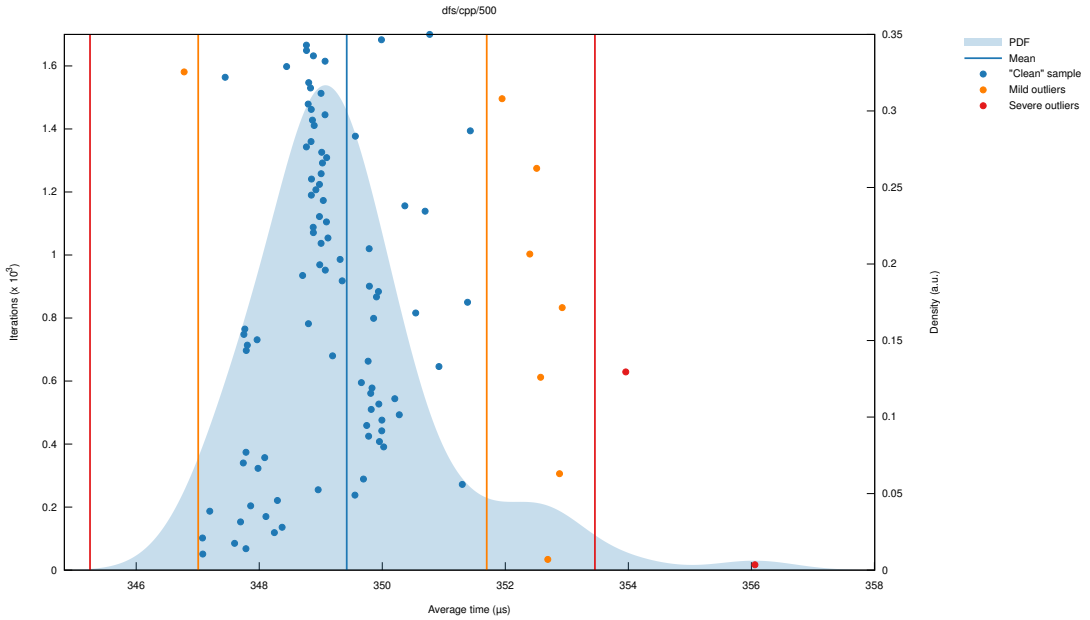
Figura 6.2: Gráfico de regressão das amostras do Micro Benchmark estatístico da DFS em Rust.



Na figura 6.2, os pontos azuis representam as amostras coletadas e a linha azul representa a reta de regressão sobre as amostras. Se o teste produz uma reta de regressão que grosseiramente passa por todas as amostras, o teste é considerado um bom benchmark (*Criterion*, n.d.), o que nesse caso é verdade.

Quanto aos teste realizado na implementação em C++, obtivemos o seguinte:

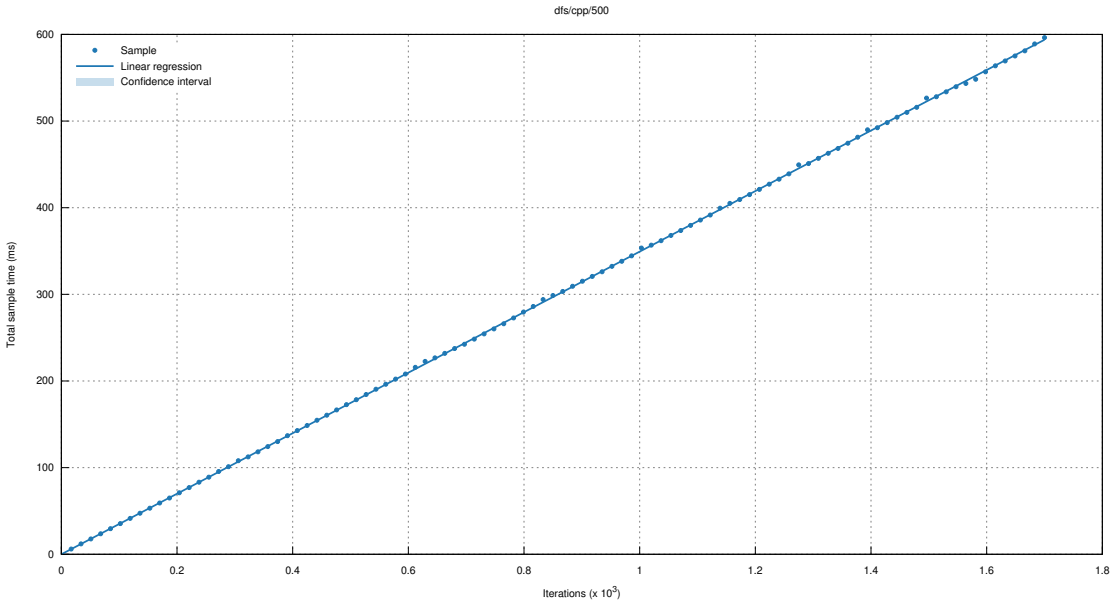
Figura 6.3: Gráfico de inclinação do Micro Benchmark estatístico da DFS em C++.



O gráfico da figura 6.3 segue as mesmas especificações do gráfico da figura 6.1, discutida anteriormente. Nesse caso, o tempo médio de execução do algoritmo é aproximadamente 349 ms, sendo 10 vezes mais rápida do que a versão em Rust, algo que se mantém consistente independente do tamanho do grafo.

Quanto ao gráfico de regressão das amostras, ele segue o padrão do teste realizado no código em Rust:

Figura 6.4: Gráfico de regressão das amostras do Micro Benchmark estatístico da DFS em C++.



Micro Benchmark de alta precisão

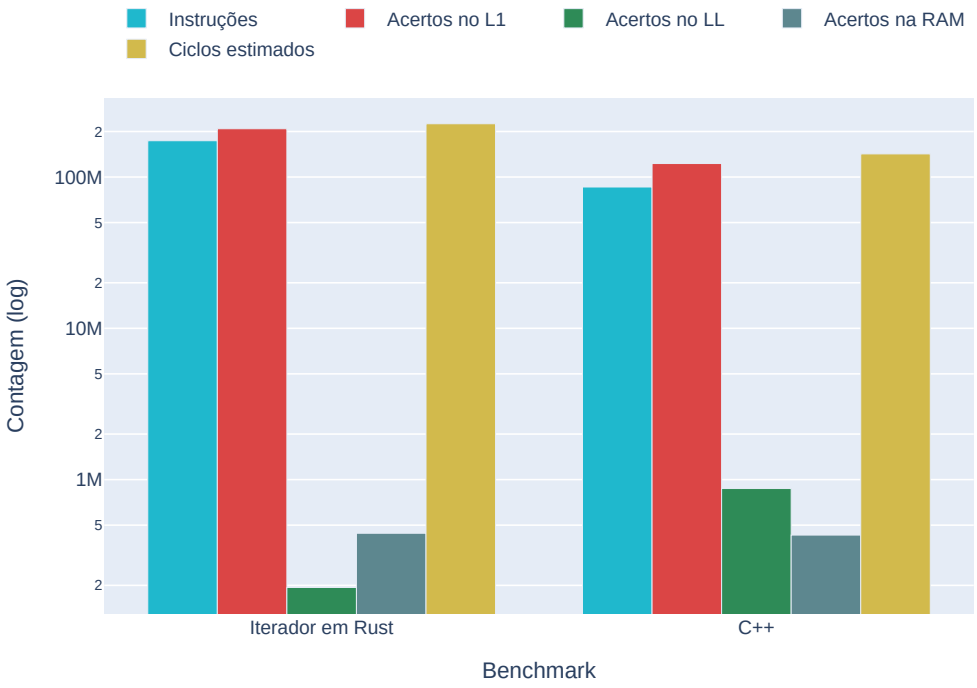
Nos testes de Micro Benchmarking de alta precisão tivemos os seguintes resultados:

Tabela 6.2: Resultados do Micro Benchmark de alta precisão da DFS.

Métricas	C++	Iterador em Rust
Quantidade de instruções	85926683	173801339
Acertos no Cache L1	122803778	208565341
Acertos no Cache LL	873133	193603
Acertos na RAM	429925	441938
Quantidade de ciclos estimada	142216818	225001186

Em forma de gráfico, tivemos:

Figura 6.5: Gráfico do Micro Benchmark de alta precisão da DFS.



Na tabela 6.2 é possível perceber que a versão em C++ realiza em torno de 2 vezes menos a quantidade de instruções que a versão em Rust faz. Entretanto também é possível notar que a versão em Rust usa mais o cache L1, em torno de 1.7 vezes mais, e bem menos o cache LL em relação a versão em C++, em torno de 4.5 vezes menos, indicando que o uso de cache é mais otimizado na versão em Rust. Também é possível notar que a versão em C++ precisa de 1.5 vezes menos ciclos na CPU para terminar sua computação. Isso significa que, mesmo aproveitando o cache, a versão da DFS em forma de iteradores em Rust precisou de mais instruções para completar, o que é razoável dado a flexibilidade do uso dessa implementação.

Capítulo 7

Conclusão e Melhorias

Implementar uma API de Grafos em Rust sem dúvidas oferece diversas vantagens, além da segurança de memória, as abstrações permitem flexibilidade e reutilização de código, enquanto o baixo nível da linguagem permite um alto nível de controle e de performance sobre a implementação.

Os traços permitem que as implementações sejam fiéis a especificação algébrica de grafos e permitem que as diversas representações compartilhem comportamento entre si.

As implementações padrões nos traços permitem que os algoritmos sejam reutilizados dentre as representações sem oferecer nenhum custo adicional.

A implementação de travessias como iteradores dá flexibilidade ao usuário por se integrar com a biblioteca padrão, e consequentemente permitir a composição com funções já existentes.

O uso do Cargo permite que a biblioteca seja facilmente exportada publicamente e também permite que exemplos e benchmarks de código sejam facilmente configurados e executados independentemente da aplicação principal.

Entretanto, naturalmente ainda há diversas possíveis melhorias que poderiam ser testadas e incorporadas na nossa implementação, dentre elas teríamos:

- **Representação de um grafo usando tabelas hash**: Uma coisa a se testar que pode ser interessante, é implementar um grafo como uma tabela hash, isto é:

```
1       struct AdjacencyList<T>(HashMap<T, HashSet<T>>);
2
```

  Uma implementação como essa poderia se aproveitar do acesso e remoção médio em $O(1)$. Além disso, tal abordagem poderia se valer da visitação de nós adjacentes de maneira rápida em grafos esparsos, aproveitando tanto as vantagens do uso de uma lista de adjacência quanto as de uma matriz de adjacência.

- **Mapear usos ótimos do iterador na DFS**: Nos testes de performance da DFS, apenas comparamos estatísticas relacionadas a execução de uma busca num grafo completo, sem nenhuma aplicação específica e sem que o compilador possa fazer suposições e realizar otimizações mais agressivas. Seria interessante identificar, trazer e analisar cenários em que o compilador consiga fazer essas previsões e otimizar o código mais agressivamente. É possível, inclusive, que tais otimizações tornem o código com o iterador mais veloz que a versão tradicional em C++ ou Rust.

- **Testes de performance para a busca em largura e classificação de arestas**: Seria interessante trazer testes de benchmark para outras travessias implementadas na base de código, como a Busca em Largura, Classificação de Arestas e iteração sobre vizinhos de um nó.

- **Restrições estruturais sobre um grafo**: Na nossa implementação o usuário tem total liberdade sobre se as funções que vai usar num grafo não direcionado ou direcionado, podendo possivelmente invalidar a estrutura interna dessas representações. Adicionando arestas direcionadas num grafo não direcionado, por exemplo. Seria interessante investigar se é possível impor restrições nesse tipo de uso, para garantir a consistência da estrutura em tempo de compilação. Um caminho para alcançar esse objetivo talvez seja tipos de tamanho 0 e *type state patterns*.
- **Iterador da DFS/BFS sem eventos de aresta de restorno e de finalização:** Uma possibilidade a se considerar seria implementar separadamente uma BFS e uma DFS que não se importam com eventos de encontro de arestas de retorno e finalização de um nó. Implementar um algoritmo separado dessa forma emplacaria melhor performance e manteria a flexibilidade que um iterador promove, mas perderia em termos do que pode ser feito durante uma busca.

Referências Bibliográficas

Analysis Process - Criterion.rs Documentation — bheisler.github.io (n.d.).
URL: *https://bheisler.github.io/criterion.rs/book/analysis.html*

Criterion (n.d.).
URL: *https://github.com/bheisler/criterion.rs*

Diestel, R. (2025), *Graph theory*, Vol. 173, Springer Nature.

Gungraun (n.d.).
URL: *https://github.com/gungraun/gungraun*

Klabnik, S., Nichols, C., Krycho, C. and Rust Community (2025*a*), 'Comparing Performance: Loops vs. Iterators - The Rust Programming Language'.
URL: *https://doc.rust-lang.org/book/ch13-04-performance.html*

Klabnik, S., Nichols, C., Krycho, C. and Rust Community (2025*b*), 'Traits: Defining Shared Behavior - The Rust Programming Language'.
URL: *https://doc.rust-lang.org/book/ch10-02-traits.html*

Klabnik, S., Nichols, C., Krycho, C. and Rust Community (2025*c*), 'Understanding Ownership — doc.rust-lang.org'.
URL: *https://doc.rust-lang.org/book/title-page.html*

Zero-cost abstractions: performance of for-loop vs. iterators — stackoverflow.com (2018).
URL: *https://stackoverflow.com/questions/52906921/zero-cost-abstractions-performance-of-for-loop-vs-iterators*

Apêndice A

Atividades desenvolvidas por cada integrante