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:

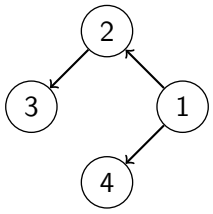

Figura 2.1: Um grafo com $V := \{1, 2, 3, 4\}$ e $A := \{(1, 2), (1, 4), (2, 3)\}$.

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:

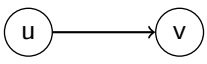

Figura 2.2: Um grafo com $V := \{u, v\}$ e $A := \{(u, v)\}$.

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:

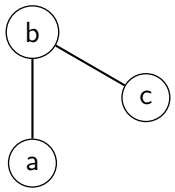

Figura 2.3: Um grafo não direcionado com $V := \{a, b, c\}$ e $A := \{ab, ba, bc, cb\}$

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.

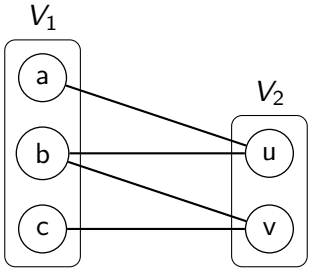

Figura 2.4: Um grafo bipartido com $V_1 := \{a, b, c\}$ e $V_2 := \{u, v\}$

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

Methodology

We mentioned in Chapter 1 that a project report's structure could follow a particular paradigm. Hence, the organization of a report (effectively the Table of Content of a report) can vary depending on the type of project you are doing. Check which of the given examples suit your project. Alternatively, follow your supervisor's advice.

3.1 Examples of the sections of a methodology chapter

A general report structure is summarised (suggested) in Table 3.1. Table 3.1 describes that, in general, a typical report structure has three main parts: (1) front matter, (2) main text, and (3) end matter. The structure of the front matter and end matter will remain the same for all the undergraduate final year project report. However, the main text varies as per the project's needs.

Tabela 3.1: Undergraduate report template structure

Frontmatter		Title Page Abstract Acknowledgements Table of Contents List of Figures List of Tables List of Abbreviations
Main text	Chapter 1	Introduction
	Chapter 2	Literature Review
	Chapter 3	Methodology
	Chapter 4	Results
	Chapter 5	Discussion and Analysis
	Chapter 6	Conclusions and Future Work
	Chapter 7	Refection
End matter		References Appendices (Optional) Index (Optional)

3.1.1 Example of a software/Web development main text structure

Notice that the "methodology" Chapter of Software/Web development in Table 3.2 takes a standard software engineering paradigm (approach). Alternatively, these suggested sections can be the chapters of their own. Also, notice that "Chapter 5" in Table 3.2 is "Testing and Validation" which is different from the general report template mentioned in Table 3.1. Check with your supervisor if in doubt.

Tabela 3.2: Example of a software engineering-type report structure

Chapter 1	Introduction	
Chapter 2	Literature Review	
Chapter 3	Methodology	
		Requirements specifications
		Analysis
		Design
		Implementations
Chapter 4	Testing and Validation	
Chapter 5	Results and Discussion	
Chapter 6	Conclusions and Future Work	
Chapter 7	Reflection	

3.1.2 Example of an algorithm analysis main text structure

Some project might involve the implementation of a state-of-the-art algorithm and its performance analysis and comparison with other algorithms. In that case, the suggestion in Table 3.3 may suit you the best.

Tabela 3.3: Example of an algorithm analysis type report structure

Chapter 1	Introduction	
Chapter 2	Literature Review	
Chapter 3	Methodology	
		Algorithms descriptions
		Implementations
		Experiments design
Chapter 4	Results	
Chapter 5	Discussion and Analysis	
Chapter 6	Conclusion and Future Work	
Chapter 7	Reflection	

3.1.3 Example of an application type main text structure

If you are applying some algorithms/tools/technologies on some problems/datasets/etc., you may use the methodology section prescribed in Table 3.4.

Tabela 3.4: Example of an application type report structure

Chapter 1	Introduction	
Chapter 2	Literature Review	
Chapter 3	Methodology	
		Problems (tasks) descriptions
		Algorithms/tools/technologies/etc. descriptions
		Implementations
		Experiments design and setup
Chapter 4	Results	
Chapter 5	Discussion and Analysis	
Chapter 6	Conclusion and Future Work	
Chapter 7	Reflection	

3.1.4 Example of a science lab-type main text structure

If you are doing a science lab experiment type of project, you may use the methodology section suggested in Table 3.5. In this kind of project, you may refer to the "Methodology" section as "Materials and Methods."

Tabela 3.5: Example of a science lab experiment-type report structure

Chapter 1	Introduction	
Chapter 2	Literature Review	
Chapter 3	Materials and Methods	
		Problems (tasks) description
		Materials
		Procedures
		Implementations
		Experiment set-up
Chapter 4	Results	
Chapter 5	Discussion and Analysis	
Chapter 6	Conclusion and Future Work	
Chapter 7	Reflection	

3.1.5 Ethical considerations

This section addresses ethical aspects of your project. This may include: informed consent, describing how participants will be informed about the study's purpose, procedures, risks, and benefits. You should detail the process used for obtaining consent and ensuring participants understand their rights.

- **Informed Consent**: If data was collected from participant, detail the process for obtaining consent and ensuring participants understand their rights.
- **Confidentiality and Privacy**: Explain measures taken to protect participants' data and maintain confidentiality. Discuss how data is stored, who will have access, and how anonymity will be preserved.

- **Risk Assessment**: Identify potential risks to participants and outline strategies to minimize them.
- **Vulnerable Populations**: If applicable, address how you will protect vulnerable groups (e.g., children, elderly, or marginalized communities) involved in your project.
- **Research Integrity**: Highlight your commitment to honesty and transparency in research. Discuss how you will avoid plagiarism, fabrication, and falsification of data.
- **Compliance with Regulations**: Mention relevant ethical guidelines and regulations that your project will adhere to.
- **Impact on Society**: Reflect on the broader implications of your project. Discuss how the outcomes may affect communities, stakeholders, or the environment, and how you plan to address any potential negative consequences.
- **Feedback Mechanisms**: Describe how you incorporate feedback from participants and stakeholders to improve the ethical conduct of the project throughout its duration.

3.2 Example of an Equation in LaTeX

Eq. 3.1 [note that this is an example of an equation's in-text citation] is an example of an equation in LaTeX. In Eq. (3.1), s is the mean of elements $x_i \in \mathbf{x}$:

$$s = \frac{1}{N} \sum_{i=1}^{N} x_i. \tag{3.1}$$

Have you noticed that all the variables of the equation are defined using the **in-text** maths command $.$, and Eq. (3.1) is treated as a part of the sentence with proper punctuation? Always treat an equation or expression as a part of the sentence.

3.3 Example of a Figure in LaTeX

Figure 3.1 is an example of a figure in LaTeX. For more details, check the link:
wikibooks.org/wiki/LaTeX/Floats,_Figures_and_Captions.
Keep your artwork (graphics, figures, illustrations) clean and readable. At least 300dpi is a good resolution of a PNG format artwork. However, an SVG format artwork saved as a PDF will produce the best quality graphics. There are numerous tools out there that can produce vector graphics and let you save that as an SVG file and/or as a PDF file. One example of such a tool is the "Flow algorithm software". Here is the link for that: flowgorithm.org.

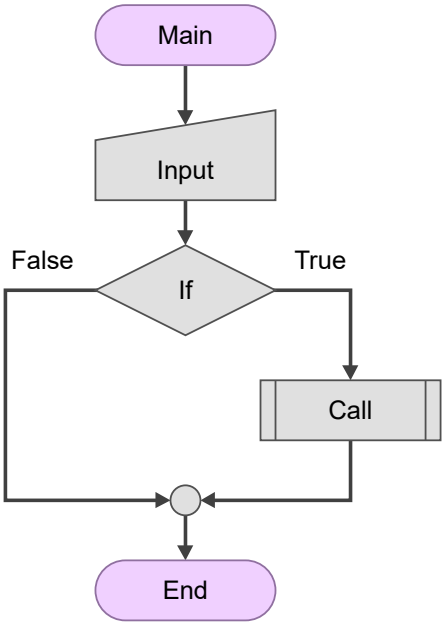

Figura 3.1: Example figure in LaTeX.

3.4 Example of an algorithm in LaTeX

Algorithm 3 is a good example of an algorithm in LaTeX.

Algorithm 1 Example caption: sum of all even numbers

Input: $\mathbf{x} = x_1, x_2, \ldots, x_N$
Output: $EvenSum$ (Sum of even numbers in $\mathbf{x}$)

```
1: function EvenSummation(x)
2:     EvenSum ← 0
3:     N ← length(x)
4:     for i ← 1 to N do
5:         if x_i mod 2 == 0 then          ▷ Check whether a number is even.
6:             EvenSum ← EvenSum + x_i
7:         end if
8:     end for
9:     return EvenSum
10: end function
```

3.5 Example of code snippet in LaTeX

Code Listing 3.1 is a good example of including a code snippet in a report. While using code snippets, take care of the following:

- do not paste your entire code (implementation) or everything you have coded. Add code snippets only.
- The algorithm shown in Algorithm 3 is usually preferred over code snippets in a technical/scientific report.
- Make sure the entire code snippet or algorithm stays on a single page and does not overflow to another page(s).

Here are three examples of code snippets for three different languages (Python, Java, and CPP) illustrated in Listings 3.1, 3.2, and 3.3 respectively.

```python
import numpy as np

x  = [0, 1, 2, 3, 4, 5] # assign values to an array
evenSum = evenSummation(x) # call a function

def evenSummation(x):
    evenSum = 0
    n = len(x)
    for i in range(n):
        if np.mod(x[i],2) == 0: # check if a number is even?
            evenSum = evenSum + x[i]
    return evenSum
```

Listing 3.1: Code snippet in LaTeX and this is a Python code example

Here we used the "\clearpage" command and forced-out the second listing example onto the next page.

```java
public class EvenSum{
    public static int evenSummation(int[] x){
        int evenSum = 0;
        int n = x.length;
        for(int i = 0; i < n; i++){
            if(x[i]%2 == 0){ // check if a number is even?
                evenSum = evenSum + x[i];
            }
        }
        return evenSum;
    }
    public static void main(String[] args){
        int[] x  = {0, 1, 2, 3, 4, 5}; // assign values to an array
        int evenSum = evenSummation(x);
        System.out.println(evenSum);
    }
}
```

Listing 3.2: Code snippet in LaTeX and this is a Java code example

```cpp
int evenSummation(int x[]){
    int evenSum = 0;
    int n = sizeof(x);
    for(int i = 0; i < n; i++){
        if(x[i]%2 == 0){ // check if a number is even?
            evenSum = evenSum + x[i];
      }
    }
    return evenSum;
}

int main(){
    int x[]  = {0, 1, 2, 3, 4, 5}; // assign values to an array
    int evenSum = evenSummation(x);
    cout<<evenSum;
    return 0;
}
```

Listing 3.3: Code snippet in LaTeX and this is a C/C++ code example

3.6 Example of in-text citation style

3.6.1 Example of the equations and illustrations placement and reference in the text

Make sure whenever you refer to the equations, tables, figures, algorithms, and listings for the first time, they also appear (placed) somewhere on the same page or in the following page(s). Always make sure to refer to the equations, tables and figures used in the report. Do not leave them without an **in-text citation**. You can refer to equations, tables and figures more them once.

3.6.2 Example of the equations and illustrations style

Write **Eq.** with an uppercase "Eq" for an equation before using an equation number with (\eqref{.}). Use "Table" to refer to a table, "Figure" to refer to a figure, "Algorithm" to refer to an algorithm and "Listing" to refer to listings (code snippets). Note that, we do not use

the articles "a," "an," and "the" before the words Eq., Figure, Table, and Listing, but you may use an article for referring the words figure, table, etc. in general.

For example, the sentence "A report structure is shown in **the** Table 3.1" should be written as "A report structure is shown **in** Table 3.1."

3.7 Summary

Write a summary of this chapter.

Note: In the case of **software engineering** project a Chapter "**Testing and Validation**" should precede the "Results" chapter. See Section 3.1.1 for report organization of such project.

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.

Cada algoritmo é expresso de forma estruturada e independente de linguagem, destacando apenas o fluxo essencial das operações. Essa abordagem facilita a compreensão dos procedimentos, sem se prender a detalhes sintáticos da linguagem de implementação.

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)

Grafos não direcionados são fundamentais para modelar problemas onde a direção não é relevante.

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 2 Busca em Profundidade

Input: $\mathbf{G}(\mathbf{V}, \mathbf{A})$, $v \in \mathbf{V}$
Output: $predecessor = [\,]$ (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 3 Busca em Largura

Input: $\mathbf{G}(\mathbf{V}, \mathbf{A})$, $v \in \mathbf{V}$
Output: $predecessor = [\,]$ (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 do dígrafo:

- **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.

O algoritmo também permite analisar a conectividade crítica do grafo, identificando vértices e arestas cuja remoção aumentaria o número de componentes conectadas.

Algorithm 4 Componentes Bicônexas

Input: $\mathbf{G}(\mathbf{V}, \mathbf{A})$
Output: Conjunto das componentes bicônexas de $\mathbf{G}$

```
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

Discussion and Analysis

Depending on the type of project you are doing, this chapter can be merged with "Results" Chapter as " Results and Discussion" as suggested by your supervisor.

In the case of software development and the standalone applications, describe the significance of the obtained results/performance of the system.

5.1 A section

The Discussion and Analysis chapter evaluates and analyses the results. It interprets the obtained results.

5.2 Significance of the findings

In this chapter, you should also try to discuss the significance of the results and key findings, in order to enhance the reader's understanding of the investigated problem

5.3 Limitations

Discuss the key limitations and potential implications or improvements of the findings.

5.4 Summary

Write a summary of this chapter.

Capítulo 6

Conclusions and Future Work

6.1 Conclusions

Typically a conclusions chapter first summarizes the investigated problem and its aims and objectives. It summaries the critical/significant/major findings/results about the aims and objectives that have been obtained by applying the key methods/implementations/experiment set-ups. A conclusions chapter draws a picture/outline of your project's central and the most signification contributions and achievements.

A good conclusions summary could be approximately 300–500 words long, but this is just a recommendation.

A conclusions chapter followed by an abstract is the last things you write in your project report.

6.2 Future work

This section should refer to Chapter **??** where the author has reflected their criticality about their own solution. Concepts for future work are then sensibly proposed in this section.

Guidance on writing future work: While working on a project, you gain experience and learn the potential of your project and its future works. Discuss the future work of the project in technical terms. This has to be based on what has not been yet achieved in comparison to what you had initially planned and what you have learned from the project. Describe to a reader what future work(s) can be started from the things you have completed. This includes identifying what has not been achieved and what could be achieved.

A good future work summary could be approximately 300–500 words long, but this is just a recommendation.

Capítulo 7

Reflection

Write a short paragraph on the substantial learning experience. This can include your decision-making approach in problem-solving.

Some hints: You obviously learned how to use different programming languages, write reports in LaTeXand use other technical tools. In this section, we are more interested in what you thought about the experience. Take some time to think and reflect on your individual project as an experience, rather than just a list of technical skills and knowledge. You may describe things you have learned from the research approach and strategy, the process of identifying and solving a problem, the process research inquiry, and the understanding of the impact of the project on your learning experience and future work.

Also think in terms of:

- what knowledge and skills you have developed
- what challenges you faced, but was not able to overcome
- what you could do this project differently if the same or similar problem would come
- rationalize the divisions from your initial planed aims and objectives.

A good reflective summary could be approximately 300–500 words long, but this is just a recommendation.

Note: The next chapter is "**References**," which will be automatically generated if you are using BibTeX referencing method. This template uses BibTeX referencing. Also, note that there is difference between "References" and "Bibliography." The list of "References" strictly only contain the list of articles, paper, and content you have cited (i.e., refereed) in the report. Whereas Bibliography is a list that contains the list of articles, paper, and content you have cited in the report plus the list of articles, paper, and content you have read in order to gain knowledge from. We recommend to use only the list of "References."

Referências Bibliográficas

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

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*

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