Universidade Federal do Rio Grande do Norte

Departamento de Informática e Matemática Aplicada

Relatório da 2ª Unidade - Grafos

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

Professor: Matheus Menezes

15 de novembro de 2025

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Capítulo 2

Revisão teórica

Todas as definições e teoremas são retirados diretamente ou readaptados para melhor clareza de Diestel (2025), George et al. (2010), Bang-Jensen and Gutin (2007) e Gersting (1993). Como os algoritmos são implementados em inglês, apresentaremos o correspondente ao termo em inglês.

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:

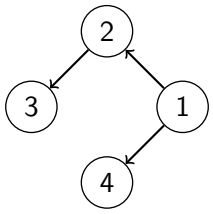

Figura 2.1: Um grafo com $V := \{1, 2, 3, 4\}$ e $A := \{(1, 2), (1, 4), (2, 3)\}$.

Definição 2 (Grafo rotulado)**.** Dizemos que um grafo $G := (V, A)$ é *rotulado* quando há informações de identificação (rótulos) nos vértices do grafo. Tais rótulos podem ser numéricos ou alfabéticos.

Definição 3 (Grafo ponderado)**.** Dizemos que um grafo $G := (V, A)$ é *ponderado* (*weighted*) quando há pesos associados a todas as arestas do grafo.

Definição 4 (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 5 (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:

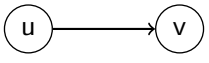

Figura 2.2: Um grafo com $V := \{u, v\}$ e $A := \{(u, v)\}$.

Definição 6 (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 7 (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 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:

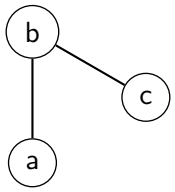

Figura 2.3: Um grafo não direcionado com $V := \{a, b, c\}$ e $A := \{ab, ba, bc, cb\}$

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 (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 10 (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 11 (Ciclo)**.** Dizemos que um grafo $G := (V, A)$ contém um *ciclo* (*cycle*) quando há um caminho possível do vértice v_i até ele próprio sem passar mais de uma vez por vértices intermediários. Quando não há ciclos no grafo, dizemos que ele é *acíclico*.

Definição 12 (Caminho Euleriano)**.** Um *Caminho Euleriano* (*Eulerian Path*) em um grafo $G := (V, A)$ é um caminho que usa cada uma das arestas de G exatamente 1 vez.

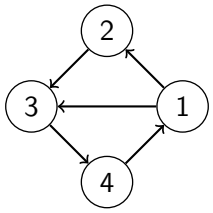

Figura 2.4: Um grafo com $V := \{1, 2, 3, 4\}$ e $A := \{(1, 2), (2, 3), (3, 4), (4, 1), (1, 3)\}$. O caminho $1 \to 2 \to 3 \to 4 \to 1 \to 3$ é um exemplo de caminho euleriano.

Teorema 1 (Teorema dos Caminhos Eulerianos)**.** Existe um caminho euleriano em um grafo não direcionado $G := (V, A)$ se e somente se existir exatamente 0 ou 2 vértices de grau ímpar no grafo. Existe um caminho euleriano em um grafo direcionado $G := (V_2, A_2)$ se e somente se existe no máximo 1 vértice com grau de saída maior que grau de entrada por 1 e no máximo 1 vértice com grau de entrada menor que grau de saída por 1.

Definição 13 (Ciclo Euleriano)**.** Um *Ciclo Euleriano* (*Eulerian Cycle*) em um grafo $G := (V, A)$ é um caso particular do Caminho Euleriano onde o Caminho inicia e termina no mesmo vértice v.

Definição 14 (Grafo Euleriano)**.** Dizemos que um grafo $G := (V, A)$ é um *Grafo Euleriano* (*Eulerian Graph*) se o grafo contém um Ciclo Euleriano.

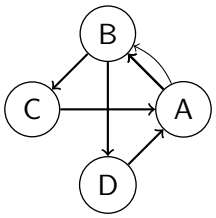

Figura 2.5: Um grafo euleriano com $V := \{A, B, C, D\}$ e $A := \{(A, B), (A, B), (B, C), (C, A), (B, D), (D, A)\}$. Exemplo de ciclo euleriano: $A \to B \to C \to A \to B \to D \to A$.

Teorema 2 (Teorema do Grafo Euleriano)**.** Um grafo conexo não orientado $G_1 := (V_1, A_1)$ é Euleriano se e somente se todos os vértices tem grau par. Um grafo conexo orientado $G_2 := (V_2, A_2)$ é Euleriano se e somente se todos os vértices tem o mesmo grau de entrada e saída.

Definição 15 (Caminho Mais Curto)**.** O *Caminho Mais Curto* (*Shortest Path*) em um grafo ponderado $G := (V, A)$ é o caminho entre dois vértices $v_i, v_j \in V$ que acumula o menor peso possível dentre todos os demais caminhos entre os dois vértices.

Definição 16 (Subgrafo)**.** Dizemos que $G' := (V', A')$ é *subgrafo* (*subgraph*) de um grafo $G := (V, A)$ quando G' consiste em um subconjunto de vértices e arestas do grafo original, mas preservando a adjacência entre os vértices.

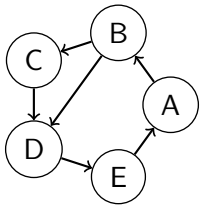

Figura 2.6: Grafo G com $V = \{A, B, C, D, E\}$ e $A = \{(A, B), (B, C), (C, D), (D, E), (E, A), (B, D)\}$.

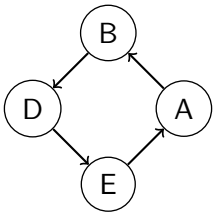

Figura 2.7: Subgrafo G', com $V' = \{A, B, D, E\}$ e $A' = \{(A, B), (B, D), (D, E), (E, A)\}$, preservando as adjacências de G.

Definição 17 (Árvore)**.** Dizemos que um grafo $G := (V, A)$ é uma *árvore* (*tree*) quando G é acíclico e conexo. Além disso, dizemos que a árvore é enraizada quando fixamos um vértice como *raiz* (*root*) ou não-enraizada quando não há raiz.

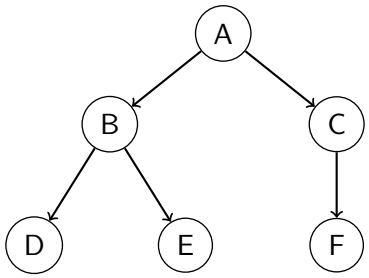

Figura 2.8: Árvore enraizada em A.

Definição 18 (Árvore Geradora)**.** Dizemos que $T := (V, A')$ é uma *árvore geradora* de $G := (V, A)$ quando T é um subgrafo de G conexo e acíclico. É uma forma de conectar todos os vértices de um grafo sem ciclos.

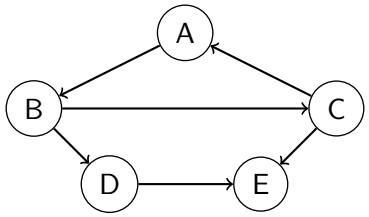

Figura 2.9: Grafo G, note a existência do ciclo $A \to B \to C \to A$

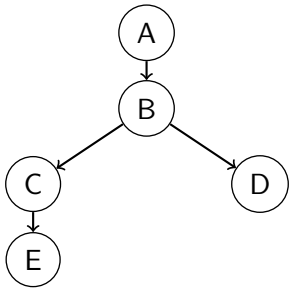

Figura 2.10: Árvore geradora T de G. Note que não há mais ciclos.

Definição 19 (Árvore Geradora Mínima)**.** Dizemos que $T := (V, A')$ é a árvore geradora mínima (ou AGM) de um grafo ponderado $G := (V, A)$ quando é a árvore geradora de menor custo dentre todas as geradoras de G.

Capítulo 3

Descrição em Pseudocódigo dos Algoritmos da API

Neste capítulo, apresentaremos a descrição dos algoritmos, na forma de pseudocódigos, que constam na especificação da API. Este capítulo tem por objetivo preparar o leitor para ler e compreender com clareza a implementação feita em Rust, trazendo descrições e explicações que elucidem o funcionamento de cada algoritmo.

3.1 Algoritmos de Árvore Geradora Mínima

3.2 Algoritmos de Caminho Mais Curto

3.2.1 Algoritmo de Dijkstra

O Algoritmo de Dijkstra (1959), proposto pelo cientista da computação Edsger Dijkstra, é um algoritmo extremamente útil para encontrar o caminho mais curto dentro de um grafo ponderado orientado ou não orientado sem arestas negativas.

O algoritmo acima escolhe o vértice para operar com base no que tem a menor distância alcançável. No início, ele visita o primeiro vértice e determina a distância para os vizinhos como sendo o peso de suas arestas; para os que não são alcançáveis, a distância é infinita. Então, enquanto existirem vértices não visitados, o algortimo seleciona o que tem a menor distância, o visita e então relaxa os seus vizinhos: o relaxamento consiste em mudar o caminho no qual o vizinho é visitado, caso isso melhore a distância até o vizinho. Isso é feito verificando se a distância do vizinho é pior do que a distância até o vértice atual + o peso da aresta que liga os dois.
Após o relaxamento, tudo se repete até que todos os vértices estejam visitados. Então, ao final das iterações, teremos a menor rota de um ponto de origem até todos os demais vértices.

3.3 Algoritmos em Grafos Eulerianos

Algorithm 1 Algoritmo de Dijkstra

Entrada $\mathbf{G}(\mathbf{V}, \mathbf{A}, \mathbf{W})$, $v \in \mathbf{V}$
Saída Sequência de $v \in \mathbf{V}$ ordenados que formam o caminho mais curto

```
function Dijkstra(G (V, A, W), s)
    predecessor ← []
    predecessor[s] ← nulo
    visitado ← []
    visitado[s] ← 1
    distancia ← []
    distancia[s] ← 0
    for v ∈ V do
        if v ∈ G.vizinhos() then
            predecessor[v] ← s
            distancia[v] ← w(sv)
        else
            predecessor[v] ← nulo
            distancia[v] ← inf
        end if
    end for
    while u ∈ V ∧ u ∉ visitado do
        v ← min V
        visitado[v] ← 1
        for n ∈ v.vizinhos() do
            if n ∉ visitado ∧ distancia[n] > distancia[v] + w(vn) then
                distancia[n] ← distancia[v] + w(vn)
                predecessor[n] ← v
            end if
        end for
    end while
    return (visitado, distancia)
end function
```

Capítulo 4

Implementação

Nesse capítulo serão mostradas as implementações reais de cada algoritmo, acrescidas de comentários que elucidem as escolhas de implementações tomadas.

4.1 Algoritmos de Árvore Geradora Mínima

4.2 Algoritmos de Caminho Mais Curto

4.2.1 Algoritmo de Dijkstra

Para a implementação de Dijkstra foi elaborado o seguinte iterador:

```
#[derive(Debug)]
pub struct DijkstraIter<'a, T, G>
where
    T: Node,
    G: Graph<T>,
{
    graph: &'a G,
    visited: HashSet<T>,
    distance: HashMap<T, i32>,
    parent: HashMap<T, Option<T>>,
}
```

Código 4.1: Iterador de Dijkstra

Este iterador, além de guardar o grafo original a ser percorrido, guarda um conjunto de vértices visitados, um map/dicionário que contém a distância de cada vértice e um map/dicionário com o predecessor de cada vértice, caso haja. Agora, partindo para a implementação do iterador, temos:

```
impl<'a, T, G> Iterator for DijkstraIter<'a, T, G>
where
    T: Node,
    G: Graph<T>,
{
    type Item = DijkstraEvent<T>;

    fn next(&mut self) -> Option<Self::Item> {
        let mut unvisited_node: Option<(T, i32)> = None;

        for node in self.graph.nodes() {
            if !self.visited.contains(&node)
                && let Some(distance) = self.distance.get(&node)
```

```
                && (unvisited_node.is_none()
                    || (unvisited_node.is_some() && distance < &
    unvisited_node.unwrap().1))
            {
                unvisited_node = Some((node, *distance));
            }
        }

        match unvisited_node {
            None => None,
            Some((node, node_weight)) => {
                self.visited.insert(node);

                if let Some(neighbors) = self.graph.neighbors(node) {
                    for (neighbor, edge_weight) in neighbors {
                        if !self.visited.contains(&neighbor) {
                            let new_distance = edge_weight + node_weight;

                            match self.distance.get(&neighbor) {
                                Some(&neighbor_distance) => {
                                    if neighbor_distance > new_distance {
                                        self.distance.insert(neighbor,
    new_distance);
                                        self.parent.insert(neighbor, Some(
    node));
                                    }
                                }
                                None => {
                                    self.distance.insert(neighbor,
    new_distance);
                                    self.parent.insert(neighbor, Some(node
    ));
                                }
                            }
                        }
                    }

                let mut parent: Option<T> = None;
                if let Some(opt) = self.parent.get(&node) {
                    parent = *opt;
                }

                Some(DijkstraEvent::Discover((node, node_weight, parent)))
            }
        }
    }
}
```

Código 4.2: Implementação do iterador de Dijkstra

A cada passo do iterador, ele calcula qual é o nó disponível com menor distância, adiciona-o no conjunto dos visitados, realiza o relaxamento de seus vizinhos e retorna o novo vértice do caminho mais curto junto de sua distância e de seu predecessor, caso haja. Caso não haja mais nós disponíveis no início do algoritmo, o iterador retorna `None`, indicando o fim do algoritmo.

Os consumidores deste iterador serão capazes de montar o caminho mais curto a partir do retorno de cada iteração, onde é retornada uma tripla com o nó que foi visitado, a distância até ele e o seu predecessor. Para encontrar o caminho mais curto entre dois nós, por exemplo,

basta salvar todos estes retornos, acessar o elemento que contém o nó final da busca, capturar seu predecessor e explorá-lo até encontrar o nó inicial.

4.3 Algoritmos de Grafos Eulerianos

Capítulo 5

Implementação

Referências Bibliográficas

Bang-Jensen, J. and Gutin, G. (2007), *Digraps: Theory, Algorithms and Applications*, Springer-Verlag.

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

Dijkstra, E. (1959), 'A note on two problems in connexion with graphs'.

George, P., Tarjan, R. E. and Woods, D. R. (2010), *Hamiltonian and Eulerian Paths*, Birkhäuser Boston, pp. 157–168.

Gersting, J. L. (1993), *Mathematical Structures for Computer Science*, W. H. Freeman and Company.

Apêndice A

Atividades desenvolvidas por cada integrante

- **Alexandre Dantas**:
- **Andriel Vinicius**:
- **Gabriel Carvalho:**
- **Maria Paz:**
- **Vinicius de Lima:**