Universidade Federal do Rio Grande do Norte

Departamento de Informática e Matemática Aplicada

Relatório da 2^a Unidade - Grafos

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

Professor: Matheus Menezes

16 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; TARJAN; WOODS, 2010), (BANG-JENSEN; 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:

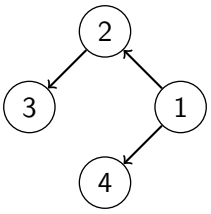

Figura 2.1: Um grafo com $V := \{1, 2, 3, 4\}$ e $A := \{(1, 2), (1, 4), (2, 3)\}$.

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 $G := (V, A, c)$ é *ponderado* (*weighted*) quando c é uma função $A \to \mathbb{R}$, onde c_α representa o custo de atrevessar uma aresta $\alpha \in A$.

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:

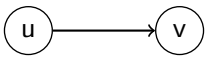

Figura 2.2: Um grafo com $V := \{u, v\}$ e $A := \{(u, v)\}$.

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:

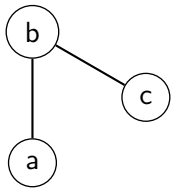

Figura 2.3: Um grafo não direcionado com $V := \{a, b, c\}$ e $A := \{ab, ba, bc, cb\}$

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.

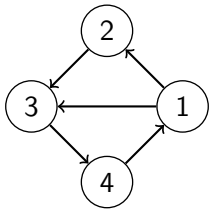

Figura 2.4: Um grafo com $V := \{1, 2, 3, 4\}$ e $A := \{(1, 2), (2, 3), (3, 4), (4, 1), (1, 3)\}$. O caminho $1 \to 2 \to 3 \to 4 \to 1 \to 3$ é um exemplo de caminho euleriano.

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.

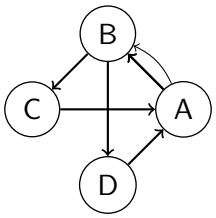

Figura 2.5: Um grafo euleriano com $V := \{A, B, C, D\}$ e $A := \{(A, B), (A, B), (B, C), (C, A), (B, D), (D, A)\}$. Exemplo de ciclo euleriano: $A \to B \to C \to A \to B \to D \to A$.

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)**.** Dizemos que $C := (V, A)$ é o *caminho mais curto* (*shortest path*) entre dois vértices u e v sse para todo caminho $C' := (V', A')$ entre u e v:

$$\sum_{\alpha \in A} c_\alpha \leq \sum_{\alpha \in A'} c_\alpha \tag{2.1}$$

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.

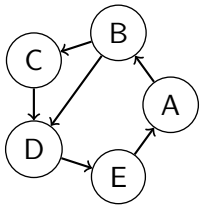

Figura 2.6: Grafo G com $V = \{A, B, C, D, E\}$ e $A = \{(A, B), (B, C), (C, D), (D, E), (E, A), (B, D)\}$.

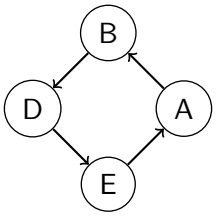

Figura 2.7: Subgrafo G', com $V' = \{A, B, D, E\}$ e $A' = \{(A, B), (B, D), (D, E), (E, A)\}$, preservando as adjacências de G.

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.

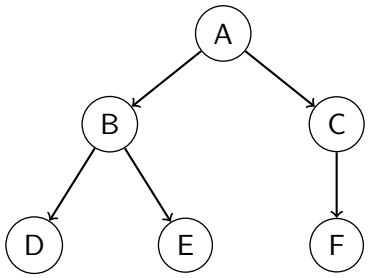

Figura 2.8: Árvore enraizada em A.

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.

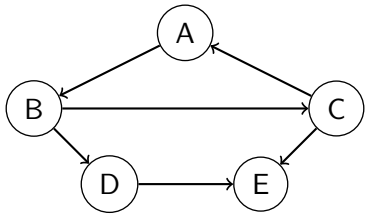

Figura 2.9: Grafo G, note a existência do ciclo $A \to B \to C \to A$

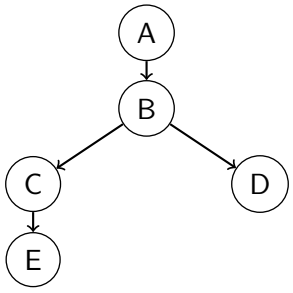

Figura 2.10: Árvore geradora T de G. Note que não há mais ciclos.

Definição 19 (Árvore Geradora Mínima)**.** Dizemos que $T := (V, A')$ é a árvore geradora mínima (ou AGM) de um grafo ponderado $G := (V, A, c)$ 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 (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.

Algorithm 1 Algoritmo de Dijkstra

Entrada $G = (V, A, c)$, $v \in V$
Saída Sequência de $v \in V$ ordenados que formam o caminho mais curto

```
function Dijkstra(G = (V, A, c), s)
    predecessor ← []
    predecessor[s] ← ∅
    visitado ← []
    visitado[s] ← 1
    distancia ← []
    distancia[s] ← 0
    for v ∈ V do
        if v ∈ vizinhos(v, G) then
            predecessor[v] ← s
            distancia[v] ← c_sv
        else
            predecessor[v] ← ∅
            distancia[v] ← lim_{x→+∞} x
        end if
    end for
    while u ∈ V ∧ ¬visitado[u] do
        v ← min V
        visitado[v] ← 1
        for n ∈ vizinhos(v, G) do
            if n ∉ visitado ∧ distancia[n] > distancia[v] + c_vn then
                distancia[n] ← distancia[v] + c_vn
                predecessor[n] ← v
            end if
        end for
    end while
    return (visitado, distancia)
end function
```

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.

3.2.2 Algoritmo de Floyd-Warshall

O Algoritmo de Robert Floyd e Stephen Warshall, comumente chamado de Floyd-Warshall (Wikipedia contributors, 2025) visa encontrar todos os caminhos mais curtos entre todos os vértices de um grafo ponderado. O algoritmo não detecta ciclos negativos, mas não é impedido por sua existência, diferentemente de 3.2.1.

A ideia geral do algoritmo é, para todos os vértices, adicionar um vértice intermediário no caminho entre outros dois se a distância final entre as extremidades for menor. Em termos de pseudocódigo, o algoritmo é descrito da seguinte forma:

Algorithm 2 Algoritmo de Floyd-Warshall

Entrada $G = (V, A, c)$

Saída Um par (D, P). Onde, para todo $i, j \in V$, $D = [d_{ij}]$ e d_{ij} representa as distância entre o vértices i e j. $P = [p_{ij}]$ e p_{ij} representa o vértice origem da última aresta no caminho de i até j.

function Floyd-Warshall(G = (V, A, c))
  $D \leftarrow \{d_{ij} \mid i, j \in V, d_{ij} = c_{ij}$ se $(i, j) \in A$ ou $d_{ij} = \lim_{x \to +\infty} x$ caso contrário $\}$
  $P \leftarrow \{p_{ij} \mid (i, j) \in A, p_{ij} = i\}$
  for $k \in V$ **do**
    for $i \in V$ **do**
      for $j \in V$ **do**
        if $d_{ik} + d_{kj} < d_{ij}$ **then**
          $d_{ij} = d_{ik} + d_{kj}$
          $p_{ij} = p_{kj}$
        end if
      end for
    end for
  end for
  return (D, P)
end function

O algoritmo primeiro começa inicializando a matriz de distâncias e predecessores com os valores conhecidos entre os vértices adjacentes. Após isso, o algoritmo, tenta inserir todo vértice k no caminho entre outros vértices i e j se for vantajoso, atualizando D e P no processo.

3.3 Algoritmos em Grafos Eulerianos

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 a seguinte *struct*:

```
pub struct DijkstraResult<Node, Weight> {
    pub route: HashMap<Node, (Weight, Option<Node>)>,
}
```

Código 4.1: Resultado de Dijkstra

Esta estrutura é responsável por armazenar um dicionário que contém o resultado do Algortimo de Dijkstra, onde cada chave é um nó que aponta para uma dupla, que indica o predecessor até o nó e também a distância dele da origem.

```
impl<N: Node, W: Weight> DijkstraResult<N, W> {
    pub fn new(graph: &(impl WeightedGraph<N, W> + ?Sized), start: N) ->
    Self {
        let mut route: HashMap<N, (W, Option<N>)> = HashMap::new();
        let mut visited: HashSet<N> = HashSet::new();
        let mut distance: HashMap<N, W> = HashMap::new();
        let mut pred: HashMap<N, Option<N>> = HashMap::new();
        distance.insert(start, W::zero());
        pred.insert(start, None);

        for (neighbor, weight) in graph.weighted_neighbors(start) {
            pred.insert(neighbor, Some(start));
            distance.insert(neighbor, weight);
        }

        loop {
            let mut unvisited_node: Option<(N, W)> = None;
            for node in graph.nodes() {
                if !visited.contains(&node)
                    && let Some(distance) = 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 => break,
                Some((node, node_weight)) => {
                    visited.insert(node);

                    for (neighbor, weight) in graph.weighted_neighbors(
    node) {
                        if !visited.contains(&neighbor) {
                            let new_distance = weight + node_weight;

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

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

                    route.insert(node, (node_weight, parent));
                }
            }
        }
        Self { route }
    }
}
```

Código 4.2: Implementação do Algoritmo de Dijkstra

A função *new* é responsável por executar o Algoritmo de Dijkstra e retornar seu resultado. Para manter o controle de vértices visitados, a distância até eles e também seus predecessores, criamos dicionários e conjuntos auxiliares para este processo.

O algoritmo inicia definindo a distância e o predecessor do nó inicial como 0 e *None*, respectivamente, para então definir os mesmos elementos para os seus vizinhos, mas sem marcar ninguém como visitado. Após isso, inicia-se o loop principal: a cada iteração, é buscado o vértice com menor distância, para que este seja visitado e os seus vizinhos sejam relaxados, ou seja, tenham sua distância e predecessor atualizados caso seja vantajoso; ao visitar um vértice, note que ele é salvo no dicionário `route`, onde o vértice é a chave que aponta para a dupla com a sua distância e também seu predecessor. Ao acabar os nós não

visitados, a função retorna a rota completa.

Para encontrar, então, o caminho entre dois vértices, o consumidor da função pode acessar o dicionário `route` a partir do vértice final e ir explorando seus predecessores até encontrar o nó inicial. Isso traz solidez e isolamento para o algoritmo, que é capaz de cumprir com eficácia seu objetivo principal sem se preocupar com o modo em que as informações serão usadas.

4.3 Algoritmos de Grafos Eulerianos

Capítulo 5

Implementação

Referências Bibliográficas

BANG-JENSEN, J.; GUTIN, G. *Digraps: Theory, Algorithms and Applications*. [S.l.]: Springer-Verlag, 2007.

DIESTEL, R. *Graph theory*. [S.l.]: Springer Nature, 2025. v. 173.

DIJKSTRA, E. *A Note on Two Problems in Connexion with Graphs*. 1959.

GEORGE, P.; TARJAN, R. E.; WOODS, D. R. Hamiltonian and eulerian paths. In: _____. *Notes on Introductory Combinatorics*. [S.l.]: Birkhäuser Boston, 2010. p. 157–168.

GERSTING, J. L. *Mathematical Structures for Computer Science*. [S.l.]: W. H. Freeman and Company, 1993.

Wikipedia contributors. *Floyd–Warshall algorithm*. 2025. Disponível em: <https://en.wikipedia.org/w/index.php?title=Floyd%E2%80%93Warshall_algorithm&oldid=1318399870>.

Apêndice A

Atividades desenvolvidas por cada integrante

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