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

17 de novembro de 2025

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Códigos

Capítulo 1

Introdução

O presente trabalho tem por objetivo descrever a implementação realizada para a avaliação prática da 2ª Unidade de DIM0549 Grafos. A avaliação mencionada consiste em implementar uma API para algoritmos de Grafos que envolvem a geração de uma árvore geradora mínima, a descoberta do caminho mais curto e a detecção de caminhos e ciclos eulerianos. Mais especificamente, serão estas as funções a serem implementadas (de forma obrigatória):

1. Árvores Geradoras Mínimas
   (a) Algoritmo de Kruskal
   (b) Algoritmo de Prim
   (c) Algoritmo de Boruvka (Opcional)
   (d) Algoritmo de Chu-Liu/Edmonds (Opcional)
2. Caminho Mais Curto
   (a) Algoritmo de Dijkstra
   (b) Algoritmo de Bellman-Ford
   (c) Algoritmo de Floyd-Warshall, incluindo a recuperação de caminhos através da árvore de caminhos mais curtos
3. Grafos Eulerianos
   (a) Algoritmo de Hierholzer para Detecção de Caminhos Eulerianos
   (b) Algoritmo de Hierholzer para Detecção de Ciclos Eulerianos (Opcional)

Dos algoritmos citados, todos os obrigatórios foram implementados, além do Algoritmo de Hierholzer para detectar caminhos eulerianos, que é opcional.

Todos os algoritmos foram implementados utilizando a linguagem de programação Rust.

1.1 Traços

Assim como no trabalho anterior, nossa implementação se materializa com o apoio dos *traços* de Rust. O uso de traços permite uma implementação genérica e eficiente para a API, de modo que há independência para os algoritmos da estrutura de dados que está sendo utilizada para representar os grafos.

Nesta implementação, existem dois tipos de traços que serão abordados abaixo.

1.1.1 Tipos

Temos dois traços que representam tipos genéricos na implementação:

1. `Node`: este traço representa todos os tipos que podem ser configurados como vértices de um grafo. Na implementação deste traço, restringimos estes a tipos que implementam os traços `Eq`, `Hash`, `Copy`, `Debug` e `Ord`. Na prática, teremos nós que são numéricos ou caracteres; note que não permitimos nós nós `String` pois em Rust este tipo não implementa `Copy`.

2. `Weight`: este traço representa os tipos aceitáveis para representar o peso das arestas. A necessidade desse traço se justifica para não tornar fixo o tipo do peso das arestas como `i32` ou `usize`. Para a criação desse traço utilizamos um pacote (`crate`) auxiliar: `num_traits`, que provém vários tipos úteis.

1.1.2 Grafos

Temos três traços que representam três tipos de grafos diferentes:

1. `Graph`: este traço representa grafos orientados e provém métodos para manipulação destes. É o tipo principal de grafos no nosso trabalho.

2. `UndirectedGraph`: este traço representa grafos não orientados e estende o traço `Graph`. Esta escolha é interessante para nossa implementação pois todas as operações dos grafos orientados são as mesmas ou compõem as operações de grafos não orientados, enquanto o contrário não necessariamente ocorre.

3. `WeightedGraph`: este traço representa grafos orientados ponderados e armazena os principais algoritmos desenvolvidos. Para representar corretamente, uma nova função de encontrar vizinhos é fornecida e todas as operações com arestas recebem um peso.

1.2 Iteradores vs Loops

Para os algoritmos propostos na especificação da API, utilizamos tanto iteradores quanto *loops* tradicionais. Em Rust, iteradores são abstrações de zero custo, ou seja, não adicionam custo extra em relação à memória, desempenho e instruções em comparação a laços de repetições convencionais. Para os algoritmos que encontram a AGM foram empregados os iteradores, enquanto os que determinam Ciclos/Caminhos Eulerianos e o Caminho Mais Curto utilizam-se de laços convencionais; não há vantagem explícita entre usar um método e outro, a escolha fica a cargo de cada componente.

1.3 Organização do relatório

No capítulo 02, iremos construir um sólido arcabouço teórico para que o leitor seja elucidado acerca dos problemas que os algoritmos têm como alvo. Serão apresentadas desde definições elementares de grafos até definições de Caminho mais Curto e Árvores Geradoras Mínimas.

No capítulo 03, serão apresentados ao leitor os pseudocódigos utilizados como base para a implementação computacional. Estes pseudocódigos são similares aos algoritmos abordados durante a dsiciplina.

No capítulo 04, exibiremos ao leitor todo o código relevante que implementa os algoritmos mencionados no capítulo 03, além de detalhar o funcionamento de trechos difíceis a fim de elucidar a compreensão da implementação.

No capítulo 05, refletiremos sobre a implementação feita e traremos conclusões e possíveis pontos de melhoria.

No apêndice A, é possível conferir a lista completa das atividades desenvolvidas por cada integrante.

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:

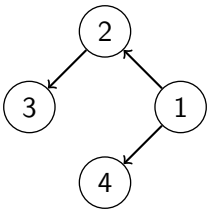

Figura 2.1: Um grafo com $V := \{1, 2, 3, 4\}$ e $A := \{(1, 2), (1, 4), (2, 3)\}$.

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:

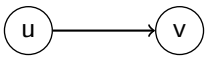

Figura 2.2: Um grafo com $V := \{u, v\}$ e $A := \{(u, v)\}$.

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:

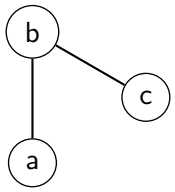

Figura 2.3: Um grafo não direcionado com $V := \{a, b, c\}$ e $A := \{ab, ba, bc, cb\}$

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.

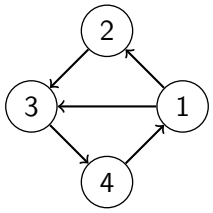

Figura 2.4: Um grafo com $V := \{1, 2, 3, 4\}$ e $A := \{(1, 2), (2, 3), (3, 4), (4, 1), (1, 3)\}$. O caminho $1 \to 2 \to 3 \to 4 \to 1 \to 3$ é um exemplo de caminho euleriano.

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.

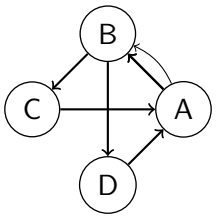

Figura 2.5: Um grafo euleriano com $V := \{A, B, C, D\}$ e $A := \{(A, B), (A, B), (B, C), (C, A), (B, D), (D, A)\}$. Exemplo de ciclo euleriano: $A \to B \to C \to A \to B \to D \to A$.

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.

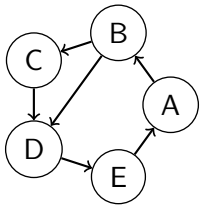

Figura 2.6: Grafo G com $V = \{A, B, C, D, E\}$ e $A = \{(A, B), (B, C), (C, D), (D, E), (E, A), (B, D)\}$.

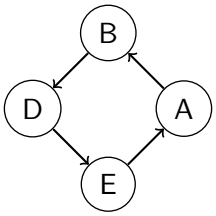

Figura 2.7: Subgrafo G', com $V' = \{A, B, D, E\}$ e $A' = \{(A, B), (B, D), (D, E), (E, A)\}$, preservando as adjacências de G.

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.

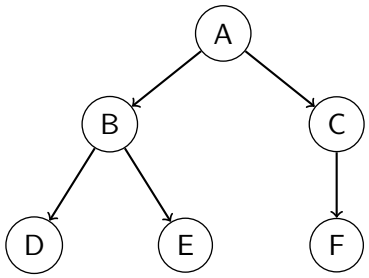

Figura 2.8: Árvore enraizada em A.

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.

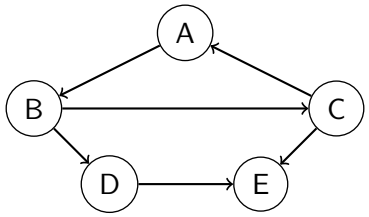

Figura 2.9: Grafo G, note a existência do ciclo $A \to B \to C \to A$

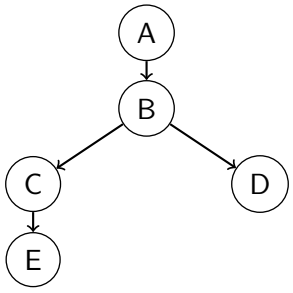

Figura 2.10: Árvore geradora T de G. Note que não há mais ciclos.

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

3.1.1 Algoritmo de Kruskal

O algoritmo de Kruskal, proposto por Joseph Kruskal em 1956, é um método clássico e eficiente para construir a Árvore Geradora Mínima (AGM) de um grafo ponderado. Seu funcionamento baseia-se em uma estratégia gulosa, selecionando iterativamente as arestas de menor peso que não formem ciclos.

Algorithm 1 Algoritmo de Kruskal

Entrada $G := (V, A, c)$
Saída Uma árvore geradora mínima $T \subseteq A$ tal que $c(T) \leq c(T\prime)$ para toda $T\prime \subseteq A$ que conecta V.

```
function Kruskal(G)
    T ← {}
    componentes ← {}
    for v ∈ V do
        componentes ← componentes ∪ {(v, {v})}
    end for
    for (u, v) ∈ A ordenado por c_uv crescente do
        if encontrar(u, componentes) ≠ encontrar(v, componentes) then
            T ← T ∪ {(u, v)}
            unir(u, v, componentes)
        end if
    end for
    return T
end function
```

O algoritmo de Kruskal inicia-se ordenando todas as arestas do grafo pelo seu peso. Em seguida, itera sobre essa lista ordenada, adicionando as arestas à árvore geradora mínima sempre que isso não resultar na formação de um ciclo.

3.1.2 Algoritmo de Prim

O algoritmo de Prim, desenvolvido por Robert C. Prim em 1957, é um método clássico para construir a Árvore Geradora Mínima (AGM) de um grafo ponderado e conexo. Assim como o algoritmo de Kruskal, utiliza uma estratégia gulosa; entretanto, enquanto Kruskal seleciona arestas globalmente pelo menor peso, Prim cresce a árvore gradualmente a partir de um vértice inicial, sempre escolhendo a aresta de menor custo que conecta a árvore parcial a um novo vértice. Quando utilizado com estruturas como filas de prioridade, o algoritmo alcança excelente desempenho em grafos densos.

Algorithm 2 Algoritmo de Prim

Entrada $G := (V, A, c)$, $r \in V$
Saída Uma árvore geradora mínima $T \subseteq A$ tal que $c(T) \leq c(T\prime)$ para toda $T\prime \subseteq A$ que conecta V.

```
function Prim(G, r)
    T ← {}
    visitado ← []
    visitado[r] ← 1
    while ∃(u, v) ∈ A : u ∈ visitado ∧ v ∉ visitado do
        (u, v) ← min{(u, v) ∈ A : u ∈ visitado ∧ v ∉ visitado}
        T ← T ∪ {(u, v)}
        visitado[v] ← 1
    end while
    return T
end function
```

Diferentemente do algoritmo de Kruskal, que começa com um conjunto vazio de arestas, o algoritmo de Prim inicia-se a partir de um vértice escolhido arbitariamente. A partir desse vértice, o algoritmo cresce a árvore geradora mínima, adicionando a aresta de menor peso que conecta um vértice da árvore a um vértice fora dela, até que todos os vértices estejam incluídos na árvore.

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.

Algorithm 3 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, 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 Bellman-Ford

O algoritmo em questão surgiu durante a segunda metade da década da 50 através de publicações feitas num período de tempo muito próximo. Seu nome homenageia os matemáticos americanos autores dos artigos que o conceberam, (BELLMAN, 1958) e (FORD, 1956). Esse se trata de um dos algoritmos mais conhecidos dentro da Teoria dos Grafos, especialmente dentro do estudo de dos problemas de caminho mínimo, e diferente de 3.2.1, não se trata de um algoritmo guloso. Além de ter a capacidade de lidar com arestas de peso negativo e identificar ciclos negativos.

Algorithm 4 Algoritmo de Bellman-Ford

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

```
function Bellman-Ford(G = (V, A, c), s)
    predecessor ← []
    distancia ← []
    for v ∈ V do                                          ▷ Inicialização
        predecessor[v] ← ∅
        distancia[v] ← ∞
    end for
    distancia[c] ← 0
    for i ∈ [1, n) do
        for (u, v) ∈ A do                                 ▷ Relaxamento de arestas
            if distancia[v] > distancia[u] + custo(u, v) then
                distancia[v] ← distancia[u] + +custo(u, v)
                predecessor[v] ← u
            end if
        end for
    end for
    ciclo ← falso
    for (u, v) ∈ A do                                     ▷ Verifica ciclo negativo
        if distancia[v] > distancia[u] + custo(u, v) then
            ciclo ← verdadeiro
        end if
    end for
    return (predecessor, distancia, ciclo)
end function
```

O algoritmo consiste em três partes: Inicialização, relaxamento de arestas e verificação de ciclo negativos. A primeira etapa envolve percorrer todos os vértices do grafo a fim de inicializar as variáveis de distância e predecessor. A segunda diz respeito ao relaxamento de arestas, ou seja, usar a desigualdade triangular para verificar se, dada uma aresta que liga os vértices A e B, é possível obter um custo menor de A para B percorrendo um terceiro vértice. E por fim, uma vez realizado o relaxamento de arestas, a desigualdade triangular será verificada novamente para toda a aresta, pois caso inda seja possível diminuir o custo ao percurrer alguma aresta do grafo há um ciclo negativo.

Note que este se trata de um algoritmo de complexidade $O(nm)$ em essência. Entretanto, a iteração a mais para verificar por ciclos negativos o faz ser $O(nm^2)$

3.2.3 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 o vértice como 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 5 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 a 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 FloydWarshall(G)
    D ← {d_ij | i, j ∈ V, d_ij := c_ij se (i, j) ∈ A ou d_ij := lim_{x→+∞} x caso contrário }
    P ← {p_ij | (i, j) ∈ A, p_ij := i}
    for k ∈ V do
        for i ∈ V do
            for j ∈ 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.

Árvore de caminhos mais curtos

A partir do retorno do algoritmo de Floyd-Warshall, é possível construir uma árvore tal que o caminho entre a raiz e qualquer vértice, é exatamente o menor caminho entre os dois.

O algoritmo é recursivo, e a ideia geral é: para cada vértice final de um caminho viável a partir de uma raiz, construir nó filho recursivamente se o vértice predecessor do caminho entre a raiz inicial (no caso base da recursão) e o vértice final for igual a raiz atual (raiz corrente na recursão). O algoritmo pode ser descrito em pseudocódigo da seguinte forma:

Algorithm 6 Algoritmo de construção da árvore de caminhos mais curtos

Entrada $G := (V, A, c)$, r e $\mathcal{F}_{\mathcal{W}} := (D, P)$. Onde $\mathcal{F}_{\mathcal{W}}$ é o resultado da aplicação de Floyd-Warshall em G e r, é a raiz escolhida previamente.

Saída Uma árvore $\mathcal{A} := (n, \mathcal{F})$. Onde $n \in V$ e $\mathcal{F}$ é uma lista de árvores como $\mathcal{A}$.

function ConstruirÁrvore($G, \mathcal{F}_{\mathcal{W}}, r, \mathcal{A} := (r, \mathcal{F} := \{\}), \Sigma := \{r\}$)
  for $j \in \{j \mid d_{ij} \in D, d_{ij} \neq \lim_{x \to +\infty} x\}$ **do**
    if $j \neq n \wedge p_{rj} = n$ **then**
      if $j \notin \Sigma$ **then**
        $\Sigma \leftarrow \Sigma \cup \{j\}$
        $\mathcal{F} \leftarrow \mathcal{F} \cup \{\text{ConstruirÁrvore}(G, \mathcal{F}_{\mathcal{W}}, r, (j, \{\}), \Sigma)\}$
      end if
    end if
  end for
  return $\mathcal{A}$
end function

Naturalmente, para recuperar o caminho mais curto entre r e qualquer outro, bastar percorrer a árvore.

3.3 Algoritmos em Grafos Eulerianos

Os algoritmos apresentados nesta seção têm por objetivo determinar a existência de Caminhos Eulerianos e Ciclos Eulerianos em grafos, além de produzir explicitamente o caminho (quando existente). A fundamentação teórica encontra-se no Capítulo 02, onde são descritas as condições necessárias e suficientes para que um grafo seja euleriano, tanto no caso orientado quanto não orientado.

O método utilizado para construir o caminho ou ciclo euleriano é o **Algoritmo de Hierholzer**, originalmente proposto em 1736 e considerado o procedimento padrão para percorrer todas as arestas de um grafo exatamente uma vez.

3.3.1 Algoritmo Principal de Hierholzer

Algorithm 7 Algoritmo de Hierholzer

```
1: function Hierholzer(G, is_directed)
2:     outDegree, inDegree ← CalcularGraus(G, is_directed)
3:     (v_0, hasPath, hasCycle) ← VerificarCondicoesEulerianas(outDegree, inDegree, is_directed)
4:     if não hasPath e não hasCycle then
5:         return (∅, falso, falso)
6:     end if
7:     if hasCycle e hasPath e |outDegree| = 1 then
8:         path ← lista com único vértice de outDegree
9:         return (path, verdadeiro, verdadeiro)
10:    end if
11:    stack ← ∅
12:    path ← ∅
13:    workGraph ← cópia de G
14:    u ← v_0
15:    while stack ≠ ∅ ou workGraph.vizinhos(u) ≠ ∅ do
16:        if workGraph.vizinhos(u) ≠ ∅ then
17:            empilhar(stack, u)
18:            v ← primeiro vizinho de u em workGraph
19:            if is_directed then
20:                workGraph.removerAresta(u, v)
21:            else
22:                workGraph.removerArestaNaoDirecionada(u, v)
23:            end if
24:            u ← v
25:        else
26:            adicionar u a path
27:            u ← desempilhar(stack)
28:            if stack = ∅ then
29:                adicionar u a path
30:            end if
31:        end if
32:    end while
33:    inverter path
34:    totalArestas ← ContarArestas(G, is_directed)
35:    caminhoValido ← (|path| = totalArestas + 1)
36:    return (path, hasCycle e caminhoValido, hasPath e caminhoValido)
37: 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

4.1.1 Algoritmo de Kruskal

Código 4.1 Construtor do Iterador de Kruskal

```rust
pub fn new(graph: &'a G) -> Self {
    let nodes: Vec<T> = graph.nodes().collect();
    let accepted_adj: HashMap<T, HashSet<T>> =
    ↪  HashMap::with_capacity(nodes.len());

    let mut seen: HashSet<(T, T)> = HashSet::with_capacity(nodes.len() * 2);
    let mut edges: Vec<(T, T, i32)> = Vec::new();

    for &u in &nodes {
        for (v, w) in graph.weighted_neighbors(u) {
            let (a, b) = if u <= v { (u, v) } else { (v, u) };
            if seen.insert((a, b)) {
                edges.push((a, b, w));
            }
        }
    }

    edges.sort_by(|(ua, va, wa), (ub, vb, wb)| {
        wa.cmp(wb).then_with(|| ua.cmp(ub)).then_with(|| va.cmp(vb))
    });

    KruskalIter {
        _graph: graph,
        edges,
        idx: 0,
        accepted_adj,
    }
}
```

Código 4.2 Iteração de Kruskal

```rust
impl<'a, T, G> Iterator for KruskalIter<'a, T, G>
where
    T: Node + Ord,
    G: UndirectedGraph<T> + WeightedGraph<T, i32> + ?Sized,
{
    type Item = KruskalEvent<T>;

    fn next(&mut self) -> Option<Self::Item> {
        if self.idx < self.edges.len() {
            let (u, v, w) = self.edges[self.idx];
            self.idx += 1;

            if !self.connected_by_accepted(u, v) {
                self.accepted_adj.entry(u).or_default().insert(v);
                self.accepted_adj.entry(v).or_default().insert(u);
                Some(KruskalEvent::EdgeAdded((u, v, w)))
            } else {
                Some(KruskalEvent::EdgeSkipped((u, v, w)))
            }
        } else {
            None
        }
    }
}
```

A função `new` prepara previamente todos os dados necessários para a execução incremental do Algoritmo de Kruskal. Ela coleta todas as arestas do grafo, organiza-as em um vetor e as ordena pelo peso, garantindo que possam ser examinadas da menor para a maior. Também é inicializada a estrutura `accepted_adj`, que representa o subgrafo formado apenas pelas arestas já selecionadas na construção da solução.

A lógica do algoritmo é realizada de forma incremental no método `next()`. A cada chamada, o iterador processa a próxima aresta da lista ordenada. Para decidir se ela deve ser incluída na árvore geradora, o método verifica se seus vértices já estão conectados no subgrafo parcial armazenado em `accepted_adj`. Caso ainda estejam conectados, isto é, caso a inclusão da aresta não forme um ciclo, a aresta é incorporada à solução e registrada na estrutura `accepted_adj`.

Ao final das iterações, as arestas acumuladas em `accepted_adj` constituem o resultado do algoritmo.

4.1.2 Algoritmo de Prim

Código 4.3 Construtor do Iterador de Prim

```
pub fn new(graph: &'a G) -> Self {
    let nodes: Vec<T> = graph.nodes().collect();
    let mut visited: HashSet<T> = HashSet::with_capacity(nodes.len());
    let mut heap = BinaryHeap::new();

    if let Some(&s) = nodes.first() {
        visited.insert(s);
        for (v, w) in graph.weighted_neighbors(s) {
            let (a, b) = if s <= v { (s, v) } else { (v, s) };
            heap.push(Reverse((w, a, b)));
        }
    }

    PrimIter {
        _graph: graph,
        visited,
        heap,
        nodes_len: nodes.len(),
    }
}
```

Código 4.4 Iteração de Prim

```rust
impl<'a, T, G> Iterator for PrimIter<'a, T, G>
where
    T: Node + Ord,
    G: UndirectedGraph<T> + WeightedGraph<T, i32> + ?Sized,
{
    type Item = PrimEvent<T>;

    fn next(&mut self) -> Option<Self::Item> {
        if self.visited.len() >= self.nodes_len {
            return None;
        }

        while let Some(Reverse((w, u, v))) = self.heap.pop() {
            let u_vis = self.visited.contains(&u);
            let v_vis = self.visited.contains(&v);

            if u_vis && v_vis {
                return Some(PrimEvent::EdgeSkipped(u, v, w));
            }

            if u_vis ^ v_vis {
                let new = if u_vis { v } else { u };
                let (a_out, b_out) = if u <= v { (u, v) } else { (v, u) };
                self.visited.insert(new);
                for (nv, w2) in self._graph.weighted_neighbors(new) {
                    let (aa, bb) = if new <= nv { (new, nv) } else { (nv, new)
                    ↪ };
                    self.heap.push(Reverse((w2, aa, bb)));
                }
                return Some(PrimEvent::EdgeAdded(a_out, b_out, w));
            }

            continue;
        }

        None
    }
}
```

A função `new()` prepara previamente todos os dados necessários para a execução incremental do Algoritmo de Prim. Inicialmente, escolhe-se um vértice arbitrário como ponto de partida (neste caso, o primeiro retornado por `graph.nodes()`). Esse vértice é marcado como visitado e todas as suas arestas incidentes são inseridas em um heap mínimo, que servirá como estrutura de prioridade para selecionar sempre a próxima aresta de menor peso que expande a árvore. São também inicializados os conjuntos `visited` e `heap`, que representarão, respectivamente, os vértices já incorporados à solução e o conjunto de arestas candidatas que conectam o conjunto visitado ao restante do grafo.

A lógica do algoritmo é realizada de forma incremental no método `next()`. A cada chamada, o iterador remove do heap a aresta de menor peso disponível. Se essa aresta conecta um vértice já visitado a um ainda não visitado, então ela é aceita na árvore geradora mínima. O vértice recém-incluído é adicionado a `visited`, e todas as suas arestas são inseridas no

heap, permitindo que novas expansões sejam consideradas. Caso contrário, se a aresta conecta dois vértices já visitados, ela é descartada, pois sua inclusão criaria um ciclo.

Ao final das iterações, quando todos os vértices tiverem sido incorporados ao conjunto `visited`, o método `next()` deixa de produzir eventos e a execução se encerra. As arestas aceitas ao longo do processo formam a árvore geradora mínima do grafo.

4.2 Algoritmos de Caminho Mais Curto

4.2.1 Algoritmo de Dijkstra

Para a implementação de Dijkstra foi elaborado a seguinte *struct*:

Código 4.5 Estrutura do iterador de Dijkstra

```
1 struct DijkstraResult<Node, Weight> {
2     route: HashMap<Node, (Weight, Option<Node>)>,
3 }
```

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.

Código 4.6 Implementação do Algoritmo de Dijkstra

```rust
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 }
```

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.

4.2.2 Algoritmo de Bellman-Ford

Já para a implementação de Bellman-Ford foi elaborado a seguinte *struct*:

Código 4.7 Estrutura que encapsula o resultado do Algoritmo de Bellman-Ford

```
pub struct BellmanFordResult<Node, Weight> {
    pub dist: HashMap<Node, Weight>,
    pub pred: HashMap<Node, Option<Node>>,
    pub has_negative_cycle: bool,
}
```

Tal estrutura contém três informações de relevância para a análise do resultado do algoritmo:

- **dist:** O dicionário que contém a relação de custo acumulado para percorrer o grafo até determinado vértice.
- **pred:** Dicionário que indica a ordem de precedência do melhor caminho dado um vértice inicial.
- **has-negative-cycle:** Um booleano correspondente há presença ou não de ciclo negativo no grafo.

E através dessa estrutura, é implementado o método `new` para na interface correspondente:

Código 4.8 Interface do algoritmo de Bellman-Ford

```
impl<N: Node, W: Weight> BellmanFordResult<N, W> {
    fn new(g: &(impl WeightedGraph<N, W> + ?Sized), start: N) -> Self;
}
```

E segue a implementação do método:

Código 4.9 Implementação do Algoritmo de Bellman-Ford

```rust
    pub fn new(g: &(impl WeightedGraph<N, W> + ?Sized), start: N) -> Self {
        let mut dist = HashMap::new();
        let mut pred = HashMap::new();

        for n in g.nodes() {
            pred.insert(n, None);
            dist.insert(n, W::max_value());
        }

        dist.insert(start, W::zero());

        for _i in 1..g.order() {
            for out_node in g.nodes() {
                for (in_node, weight) in g.weighted_neighbors(out_node) {
                    let new_dist = dist[&out_node].saturating_add(&weight);

                    if dist[&in_node] > new_dist {
                        dist.insert(in_node, new_dist);
                        pred.insert(in_node, Some(out_node));
                    }
                }
            }
        }

        let mut has_negative_cycle = false;

        for out_node in g.nodes() {
            for (in_node, weight) in g.weighted_neighbors(out_node) {
                if dist[&in_node] > dist[&out_node].saturating_add(&weight) {
                    has_negative_cycle = true;
                }
            }
        }

        Self {
            dist,
            pred,
            has_negative_cycle,
        }
    }
```

A execução do Algoritmo de Bellman-Ford se dá através da chamada da função *new*. Nela, dado um grafo e um vértice inicial, o algoritmo é excecutado e tem seu resultado encapsulado através do *BellmanFordResult*.

4.2.3 Algoritmo de Floyd-Warshall

Para o algoritmo de Floyd-Warshall (3.2.3), inicialmente definimos uma estrutura, que vai representar o retorno da função:

Código 4.10 Estrutura de resultado do Algoritmo Floyd-Warshall

```
struct FloydWarshallResult<Node, Weight> {
    dist: HashMap<Node, HashMap<Node, Weight>>,
    pred: HashMap<Node, HashMap<Node, Node>>,
}
```

Ela é genérica a qualquer tipo de vértice e de custos, identificados por `Node` e `Weight`. `dist` representa a matriz de distâncias do resultado do algoritmo e `pred` representa a matriz de predecessores. Para o resultado ser genérico a qualquer tipo, é necessário implementar a matriz através de tabelas hash que indexam vértices e armazenam outras tabelas hash, as quais indexam vértices e custo (no caso da matriz de distâncias). De forma que um acesso `dist[i][j]` represente a distância entre o vértice `i` e o vértice `j`.

Após definir a estrutura, definimos a interface do algoritmo como sendo o construtor dessa estrutura, nomeado de `new`:

Código 4.11 Interface do algoritmo de Floyd-Warshall

```
impl<N: Node, W: Weight> FloydWarshallResult<N, W> {
    fn new(g: &(impl WeightedGraph<N, W> + ?Sized)) -> Self;
}
```

O algoritmo espera a implementação de um grafo ponderado `g` que pode ter seu tamanho conhecido ou não em tempo de compilação (com a restrição de traço `?Sized`). `N` e `W` são o tipo do vértice e do custo, e são restritos pelos traços `Node` e `Weight` respectivamente.

Começando a implementação, definimos as duas estruturas que vão constituir o resultado e as inicializamos:

```
let mut dist = HashMap::with_capacity(g.order());
let mut pred = HashMap::with_capacity(g.order());
for n in g.nodes() {
    let mut neighbors_dist = HashMap::new();
    let mut neighors_pred = HashMap::new();

    neighbors_dist.insert(n, W::zero());
    neighors_pred.insert(n, n);

    for (neighbor, weight) in g.weighted_neighbors(n) {
        neighbors_dist.insert(neighbor, weight);
        neighors_pred.insert(neighbor, n);
    }

    dist.insert(n, neighbors_dist);
    pred.insert(n, neighors_pred);
}
```

Essa inicialização garante que os custos e predecessores dos vértices adjacentes já sejam incorporados na estrutura. Entretanto, note que as outras distâncias e predecessores não são inicializados, diferente de como é descrito no pseudocódigo 3.2.3. Note também que

explicitamente definimos `pred[i][i]` como `i` e `dist[i][i]` como `0` nessa implementação. Fizemos isso para manter a fidelidade a implementações mais clássicas do algoritmo, mas não é obrigatório.

Antes de partir para o cerne da implementação, definimos uma função anônima denominada `unwrap_dist`, para lidar com o acesso seguro as tabelas hash de rust, especificamente a `dist`. Pois o acesso usando operador `[]` pode causar pânicos se a chave não existir, o que no nosso caso não é garantido para o segundo acesso.

```
let unwrap_dist = |dist: &HashMap<N, HashMap<N, W>>, i, j| {
    dist[&i].get(&j).copied().unwrap_or(W::max_value())
};
```

No caso em que a segunda chave não exista, retornamos o valor máximo do tipo numérico do custo (`W::max_value()`) ao invés do valor que existiria no acesso.

Por fim, partimos para o cerne da implementação, o loop principal do algoritmo:

```
for k in g.nodes() {
    for i in g.nodes() {
        for j in g.nodes() {
            let dist_ik = unwrap_dist(&dist, i, k);
            let dist_kj = unwrap_dist(&dist, k, j);
            let dist_ij = unwrap_dist(&dist, i, j);
            if let Some(sum) = dist_ik.checked_add(&dist_kj)
                && sum < dist_ij
            {
                dist.entry(i).and_modify(|ds| {
                    ds.entry(j)
                        .and_modify(|d| *d = sum)
                        .or_insert(sum);
                });
                let pred_kj = pred[&k][&j];
                pred.entry(i).and_modify(|ps| {
                    ps.entry(j)
                        .and_modify(|p| *p = pred_kj)
                        .or_insert(pred_kj);
                });
            }
        }
    }
}

Self { dist, pred }
```

A implementação segue o algoritmo clássico em semântica, entretanto, com diversas checagens de segurança. Por exemplo, ao checar se `k` melhora a distância entre `i` e `j`, é primeiro verificado se a soma entre `dist[i][k]` e `dist[k][j]` causa overflow de inteiro, usando o método `checked_add` de um tipo numérico:

```
if let Some(sum) = dist_ik.checked_add(&dist_kj)
    && sum < dist_ij { ... }
```

Além disso, ao invés de alterar diretamente o predecessor e distância de ij quando é necessário, alteramos ou inserimos o novo valor, visto que não é garantido que `pred[i][j]` ou `dist[i][j]` existiam previamente.

```
dist.entry(i).and_modify(|ds| {
    ds.entry(j)
        .and_modify(|d| *d = sum)
        .or_insert(sum);
});
let pred_kj = pred[&k][&j];
pred.entry(i).and_modify(|ps| {
    ps.entry(j)
        .and_modify(|p| *p = pred_kj)
        .or_insert(pred_kj);
});
```

A implementação completa do algoritmo é a seguinte:

Código 4.12 Implementação do algoritmo de Floyd-Warshall

```
impl<N: Node, W: Weight> FloydWarshallResult<N, W> {
    pub fn new(g: &(impl WeightedGraph<N, W> + ?Sized)) -> Self {
        let mut dist = HashMap::with_capacity(g.order());
        let mut pred = HashMap::with_capacity(g.order());
        for n in g.nodes() {
            let mut neighbors_dist = HashMap::new();
            let mut neighors_pred = HashMap::new();

            neighbors_dist.insert(n, W::zero());
            neighors_pred.insert(n, n);

            for (neighbor, weight) in g.weighted_neighbors(n) {
                neighbors_dist.insert(neighbor, weight);
                neighors_pred.insert(neighbor, n);
            }

            dist.insert(n, neighbors_dist);
            pred.insert(n, neighors_pred);
        }

        let unwrap_dist = |dist: &HashMap<N, HashMap<N, W>>, i, j| {
            dist[&i].get(&j).copied().unwrap_or(W::max_value())
        };

        for k in g.nodes() {
            for i in g.nodes() {
                for j in g.nodes() {
                    let dist_ik = unwrap_dist(&dist, i, k);
                    let dist_kj = unwrap_dist(&dist, k, j);
                    let dist_ij = unwrap_dist(&dist, i, j);
                    if let Some(sum) = dist_ik.checked_add(&dist_kj)
                        && sum < dist_ij
                    {
                        dist.entry(i).and_modify(|ds| {
                            ds.entry(j)
                                .and_modify(|d| *d = sum)
                                .or_insert(sum);
                        });
                        let pred_kj = pred[&k][&j];
                        pred.entry(i).and_modify(|ps| {
                            ps.entry(j)
                                .and_modify(|p| *p = pred_kj)
                                .or_insert(pred_kj);
                        });
                    }
                }
            }
        }

        Self { dist, pred }
    }
}
```

4.2.4 Criação da árvore de caminhos mais curtos

Para a criação da árvore de menores caminhos, seguiremos uma abordagem parecida com a implementação do algoritmo de Floyd-Warshall (4.2.3). Primeiro definiremos uma estrutura de retorno, no caso a árvore:

Código 4.13 Estrutura de retorno do algoritmo de criação da árvore de caminhos mais curtos

```
struct ShortestPathTree<Node, Weight> {
    node: Node,
    childs: Vec<(Weight, ShortestPathTree<Node, Weight>)>,
}
```

Novamente, a estrutura é genérica a qualquer tipo de vértice e custo, identificados por `Node` e `Weight`. `node` representa o valor de um vértice no nó da árvore e `childs` representam os nós filhos da árvore. `childs` é uma lista de tuplas de custo e árvore, o custo nessa tupla representa o custo de percorrer a aresta para o nó filho no segundo argumento da tupla.

Quanto a interface do algoritmo, ele será implementado como o construtor da estrutura, de maneira similar a interface de Floyd-Warshall (4.11):

Código 4.14 Interface do algoritmo de criação da árvore de caminhos mais curtos

```
impl<N: Node, W: Weight> ShortestPathTree<N, W> {
    fn new(g: &(impl WeightedGraph<N, W> + ?Sized), root: N) -> Self;
}
```

Assim como o algoritmo de Floyd-Warshall, o construtor espera um grafo ponderado, que pode ter seu tamanho conhecido ou não em tempo de compilação. Além disso, o algoritmo também espera uma raiz escolhida previamente pelo utilizador da interface. Como em Rust não há suporte para argumentos com valor padrão, vamos definir uma função auxiliar que recebe uma árvore e um dicionário de vértices visitados durante a criação da árvore:

```
fn build_tree<N: Node, W: Weight>(
    visited: &mut HashSet<N>,
    floyd: &FloydWarshallResult<N, W>,
    tree: &mut ShortestPathTree<N, W>,
    root: N,
) {
    for (&k, &w) in &floyd.dist[&tree.node] {
        if k != tree.node && floyd.pred[&root][&k] == tree.node &&
        ↪  visited.insert(k) {
            let mut new_child = ShortestPathTree {
                node: k,
                childs: vec![],
            };
            build_tree(visited, floyd, &mut new_child, root);
            tree.childs.push((w, new_child));
        }
    }
}
```

A implementação segue fielmente seu pseudocódigo, a principal diferença está na condicional `visited.insert(k)`, que valora para verdadeiro somente se foi possível inserir o nó `k` no dicionário de visitados, ou seja, se não houve repetição.

Por fim, definimos a função principal como:

Código 4.15 Implementação do algoritmo da criação da árvore de menores caminhos

```
fn new(g: &(impl WeightedGraph<N, W> + ?Sized), root: N) -> Self {
    let floyd = g.floyd_warshall();
    let mut visited = HashSet::from([root]);
    let mut tree = ShortestPathTree {
        node: root,
        childs: vec![],
    };
    build_tree(&mut visited, &floyd, &mut tree, root);

    tree
}
```

4.2.5 Algoritmo de Hierholzer

Para o algoritmo de Hierholzer (3.3), inicialmente definimos uma estrutura, que vai representar o retorno da função:

Código 4.16 Estrutura de resultado do Algoritmo de Hierholzer

```
pub struct HierholzerResult<Node> {
    pub path: Vec<Node>,
    pub has_eulerian_path: bool,
    pub has_eulerian_cycle: bool,
}
```

Ela é genérica a qualquer tipo de nó, identificado por `Node`. `path` representa o caminho euleriano encontrado (se existir), `has_eulerian_path` indica se existe um caminho euleriano no grafo, e `has_eulerian_cycle` indica se existe um ciclo euleriano no grafo.

Após definir a estrutura, definimos a interface do algoritmo como sendo o construtor dessa estrutura, nomeado de `new`:

Código 4.17 Interface do algoritmo de Hierholzer

```
impl<N: Node> HierholzerResult<N> {
    pub fn new<G: UndirectedGraph<N>>(graph: &G, is_directed: bool) -> Self;
}
```

O algoritmo espera a implementação de um grafo `graph` que implementa a trait `UndirectedGraph` e um booleano `is_directed` indicando se o grafo deve ser tratado como direcionado. `N` é o tipo do nó e é restrito pelo traço `Node`.

Começando a implementação, definimos as estruturas que vão auxiliar no cálculo e as inicializamos:

```
let mut out_degree = HashMap::new();
let mut in_degree = HashMap::new();
Self::compute_every_node_degree(graph, &mut out_degree, &mut in_degree);

let (mut start_node, has_eulerian_path, has_eulerian_cycle) =
    Self::check_eulerian_conditions(&out_degree, &in_degree, is_directed);
```

Essa inicialização garante que os graus de todos os vértices sejam calculados e que as condições de Euler sejam verificadas antes de executar o algoritmo principal. A função `compute_every_node_degree` calcula os graus de entrada e saída de todos os vértices, enquanto `check_eulerian_conditions` determina se o grafo possui ciclo euleriano, caminho euleriano, ambos ou nenhum dos dois.

Em seguida, tratamos os casos especiais onde não existe caminho euleriano ou onde existe um ciclo trivial:

```
if !has_eulerian_path && !has_eulerian_cycle {
    return HierholzerResult {
        path: Vec::new(),
        has_eulerian_cycle,
        has_eulerian_path,
    };
} else if has_eulerian_cycle && has_eulerian_path && out_degree.len() == 1 {
    return HierholzerResult {
        path: out_degree.keys().copied().collect::<Vec<N>>(),
        has_eulerian_cycle,
        has_eulerian_path,
    };
}
```

O primeiro caso lida com grafos que não possuem nem caminho nem ciclo euleriano, retornando imediatamente. O segundo caso trata de ciclos triviais em grafos com apenas um vértice.

Para o algoritmo principal, inicializamos as estruturas de dados necessárias:

```
let mut stack = Vec::new();
let mut path = Vec::new();
let mut work_graph: G = graph.clone();
```

A `stack` é usada para armazenar vértices durante a exploração, o `path` armazena o caminho final, e `work_graph` é uma cópia do grafo original onde arestas são removidas conforme são visitadas.

Por fim, partimos para o cerne da implementação, o loop principal do algoritmo:

```
loop {
    if stack.is_empty() && work_graph.neighbors(start_node).count() == 0 {
        break;
    } else if work_graph.neighbors(start_node).count() > 0 {
        stack.push(start_node);

        let next_neighbor = work_graph.neighbors(start_node).next();
```

```
8
9           if let Some(neighbor) = next_neighbor {
10              if is_directed {
11                  work_graph.remove_edge(start_node, neighbor);
12              } else {
13                  work_graph.remove_undirected_edge(start_node, neighbor);
14              }
15
16              start_node = neighbor;
17          }
18      } else {
19          path.push(start_node);
20          if let Some(s) = stack.pop() {
21              start_node = s;
22          }
23
24          if stack.is_empty() {
25              path.push(start_node);
26          }
27      }
28  }
29  path.reverse();
```

O algoritmo segue uma estratégia baseada em pilha para construir o caminho euleriano. Enquanto houver vértices para processar, o algoritmo:

- Se o vértice atual tem arestas incidentes: empilha o vértice e segue uma aresta não visitada.
- Caso contrário: adiciona o vértice ao caminho e retrocede na pilha.

Ao final do loop, o caminho é revertido pois foi construído de trás para frente.

Finalmente, validamos o caminho encontrado e retornamos o resultado:

```
1   let total_edges: usize = if is_directed {
2       out_degree.values().sum()
3   } else {
4       out_degree.values().sum::<usize>() / 2
5   };
6
7   let valid_path = path.len() == total_edges + 1;
8
9   Self {
10      path: if valid_path { path } else { Vec::new() },
11      has_eulerian_cycle: valid_path && has_eulerian_cycle,
12      has_eulerian_path: valid_path && has_eulerian_path,
13  }
```

A validação verifica se o caminho encontrado tem comprimento adequado (número de arestas + 1) e atualiza os flags indicando a existência de ciclo e caminho euleriano de acordo.

Capítulo 5

Conclusão

Implementar uma API para Grafos em Rust fornece diversas vantagens, indo desde a segurança de memória, mecanismo poderoso e nativo da linguagem até o uso de tipos genéricos, que permite uma poderosa reusabilidade das implementações, além do alto controle sobre a implementação.

O uso de traços para representação de tipos de dados e Grafos permite que as estruturas sejam fiéis à especificação algébrica dos dados e grafos, além de permitir que diferentes implementações compartilhem métodos entre si. As implementações padrões dentro destes também permitem que, independente da estrutura de dados, certos algoritmos sejam reutilizados sem oferecer custos adicionais.

A criação dos exemplos presentes em `examples` permitem a rápida visualização das execuções dos algoritmos. Para além disso, a criação de testes unitários em cada arquivo permite o desenvolvedor atestar se o comportamento se mantém válido mesmo com mudanças.

A especificação da Lista de Adjacência como um `HashMap<T, Vec>` permite uma representação mais fiel e facilitada de grafos em comparação ao trabalho da Unidade 1, onde os nós eram todos numéricos e auto-gerenciados com índices. Desta forma, conseguimos obter comportamentos e dados mais próximos aos cenários reais. Adicionalmente, `HashMap` têm inserção e acessos como $O(1)$, o que é um avanço em relação à representação por `Vec`.

Mesmo com estas vantagens, reconhecemos algumas limitações neste projeto, onde a principal seria que não há nenhuma restrição estrutural e de tipagem que impeça um grafo orientado de usar funções de grafos não orientados, por exemplo. Isso pode bagunçar os dados do grafo e impedir que os algoritmos funcionem corretamente. Deveria ser feito um levantamento do que pode ser feito para o próximo projeto.

Referências Bibliográficas

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

BELLMAN, R. *On a Routing Problem**. 1958. Disponível em: <https://www.ams.org/journals/qam/1958-16-01/S0033-569X-1958-0102435-2/S0033-569X-1958-0102435-2.pdf>.

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

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

FORD, L. R. *Network Flow Theory*. 1956. Disponível em: <https://www.rand.org/pubs/papers/P923.html#document-details>.

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**: Algoritmo de Kruskal, Algoritmo de Prim, Relatório (Cap. 03, 04), documentação.
- **Andriel Vinicius**: Algoritmo de Dijkstra, Algoritmo de Hierholzer, Relatório (Cap. 01, 02, 03, 04, 05), revisão de código, documentação.
- **Gabriel Carvalho:** Algoritmo de Bellman-Ford, Relatório (Cap. 03, 04), documentação, vídeo demonstrativo.
- **Maria Paz:** Algoritmo de Hierholzer, Relatório (Cap. 02, 03, 04), documentação.
- **Vinicius de Lima:** Algoritmo de Floyd-Warshall e descoberta do caminho mais curto, Relatório (Cap. 02, 03, 04), revisão de código, documentação.