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

14 de novembro de 2025

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Capítulo 2

Revisão teórica

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

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. 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

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

Apêndice A

Atividades desenvolvidas por cada integrante

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