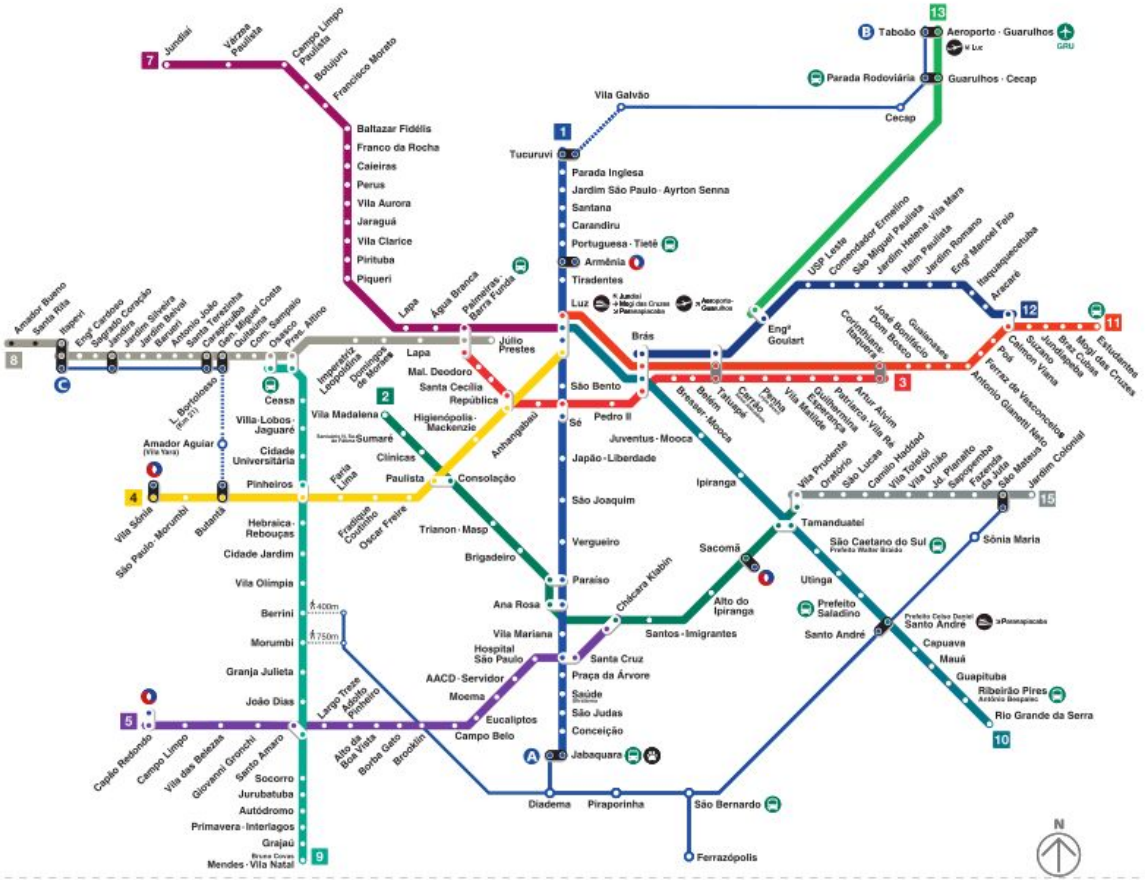
Algoritmo SPP Dijkstra em Rust

Disciplina: Projeto e Complexidade de algoritmos 2022/1
Professor: Dr. Rodrigo Bonifácio
Alunos:
Otho Teixeira Komatsu - 221108111
Luiz Antônio Borges Martins - 210025981

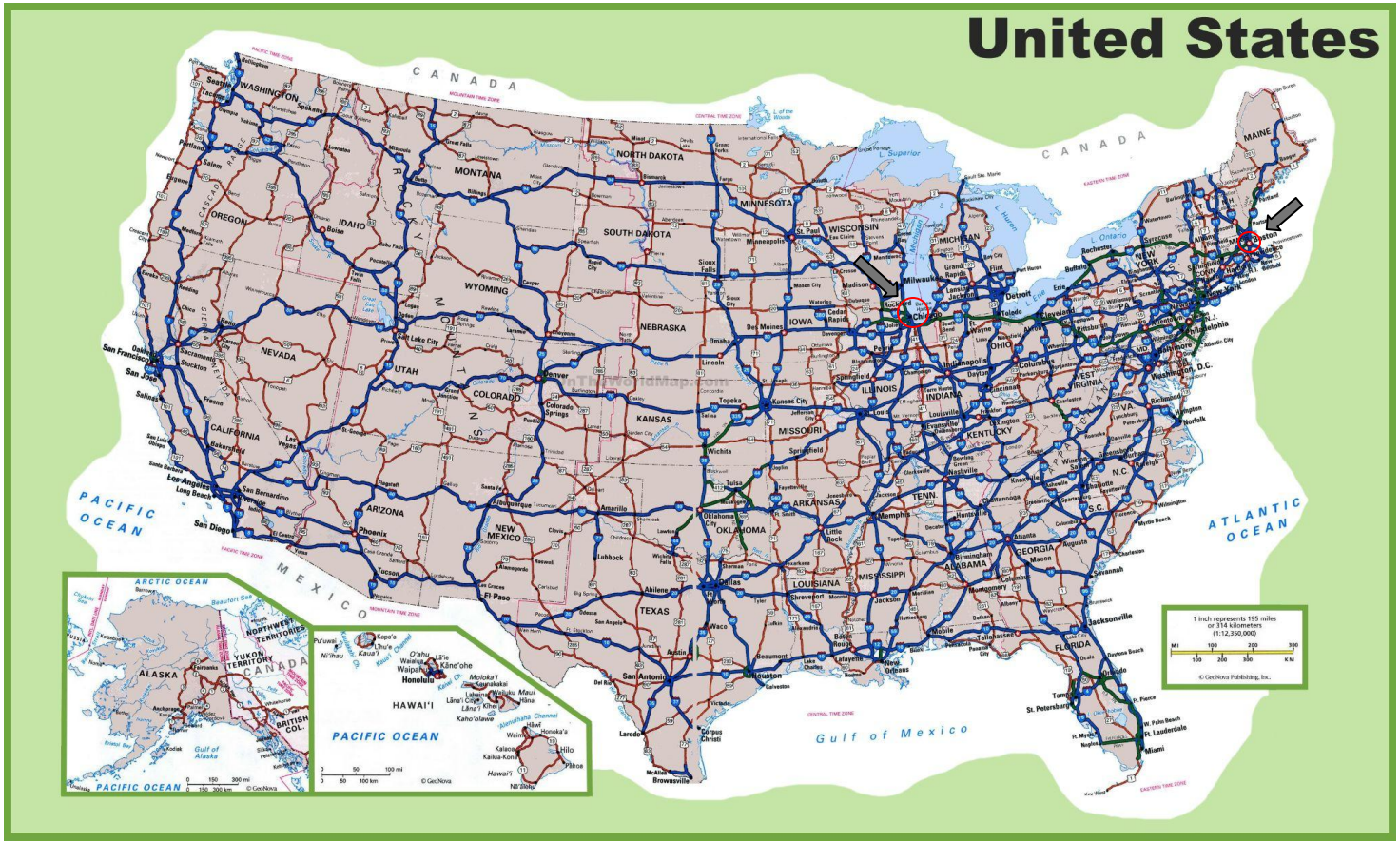
Descrição do Problema

Mapa do Transporte Metropolitano

Metropolitan Transport Network

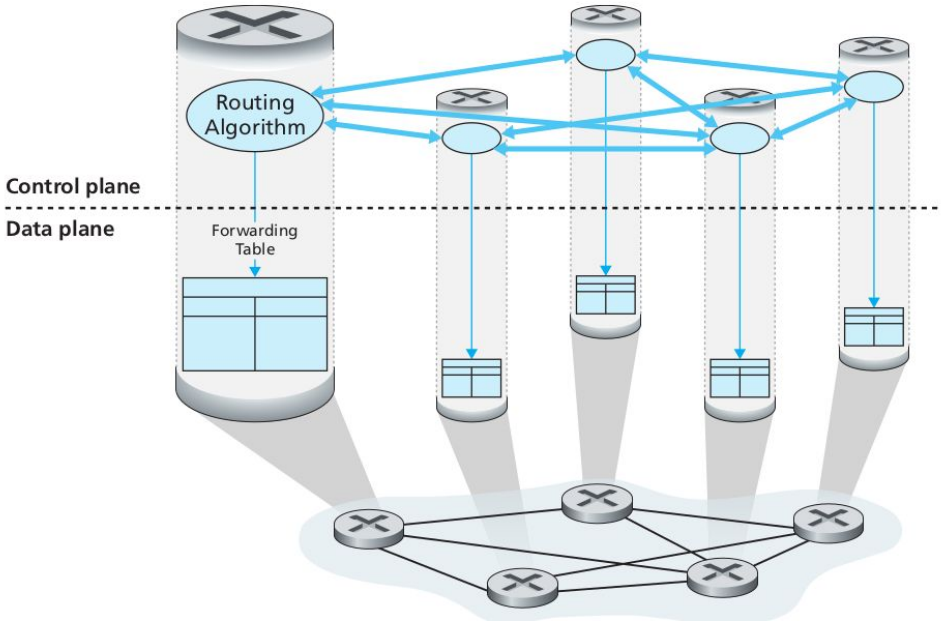


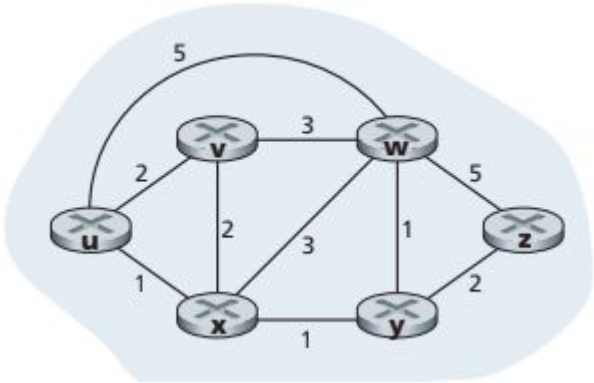
Boston para Chicago



Boston para Chicago

Algoritmos de Roteamento em redes





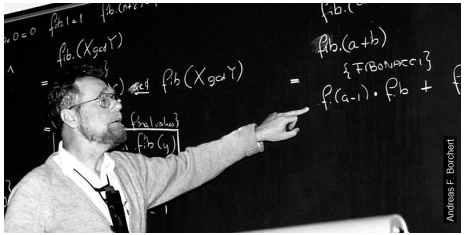
Shortest Path Problem

- Modelo: Grafo direcionado **G = (V, E),** em que **V** é conjunto de vértices e **E** conjunto de arestas, com pesos w.
- Preço do caminho: $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$.

- Objetivo: Menor caminho entre 2 vértices
- No contexto do metrô: Peso $\longrightarrow$ Distância, lotação, qualidade do trem

Algoritmo

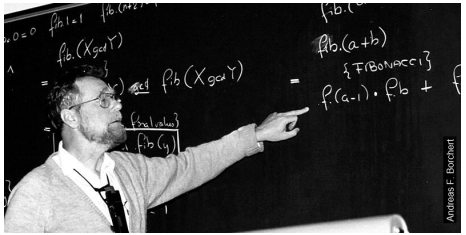
Djikstra's Algorithm



- Para shortest-paths problems single source
- Pesos não negativos
- **Greedy Strategy**
- "Relaxamento dos vértices"

$$
\begin{array}{ll}
\textsc{Relax}(u, v, w) & \\
1 & \textbf{if } d[v] > d[u] + w(u, v) \\
2 & \quad \textbf{then } d[v] \leftarrow d[u] + w(u, v) \\
3 & \quad\quad \pi[v] \leftarrow u
\end{array}
$$

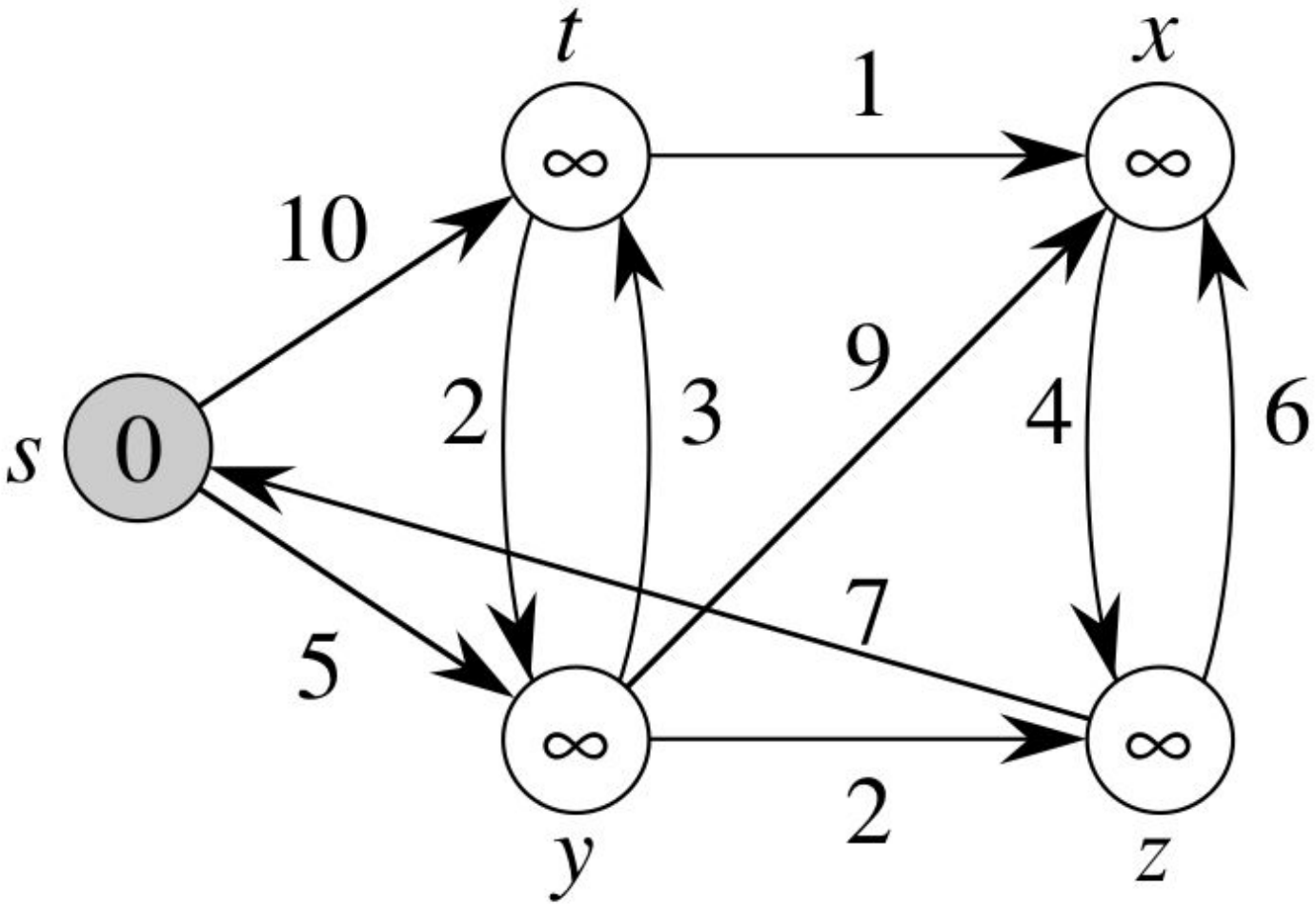
Djikstra's Algorithm

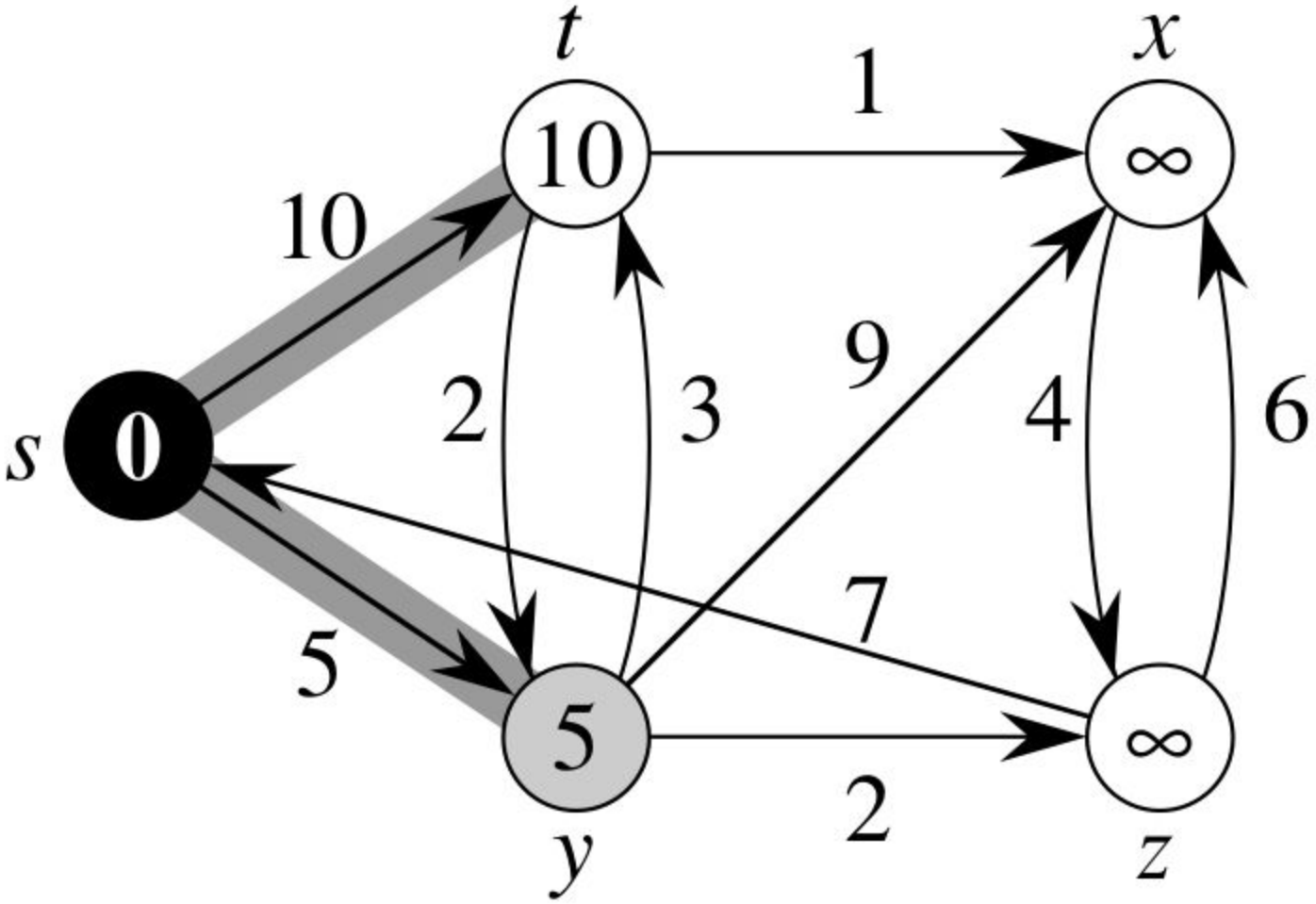


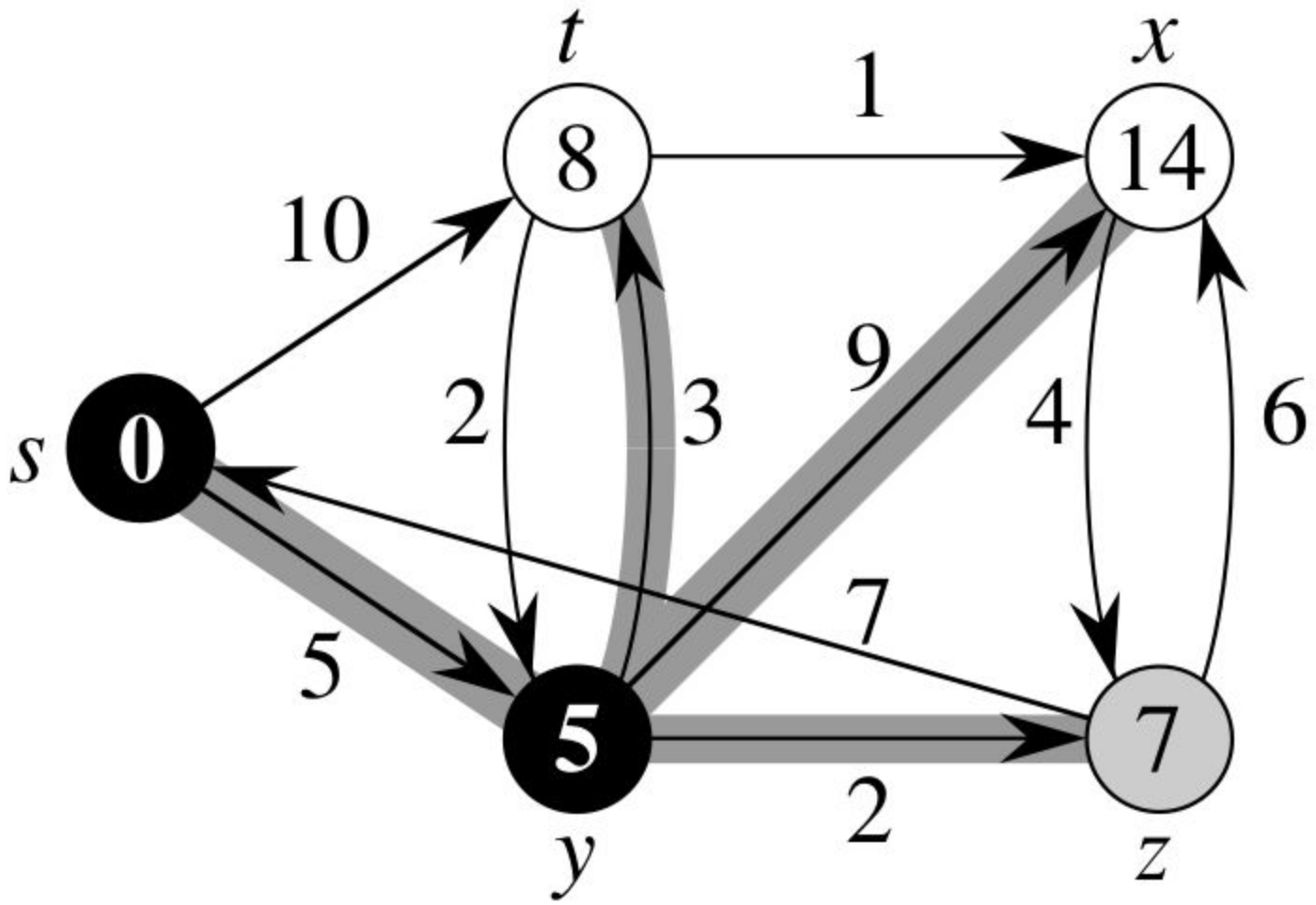
- Inicialização de vértices

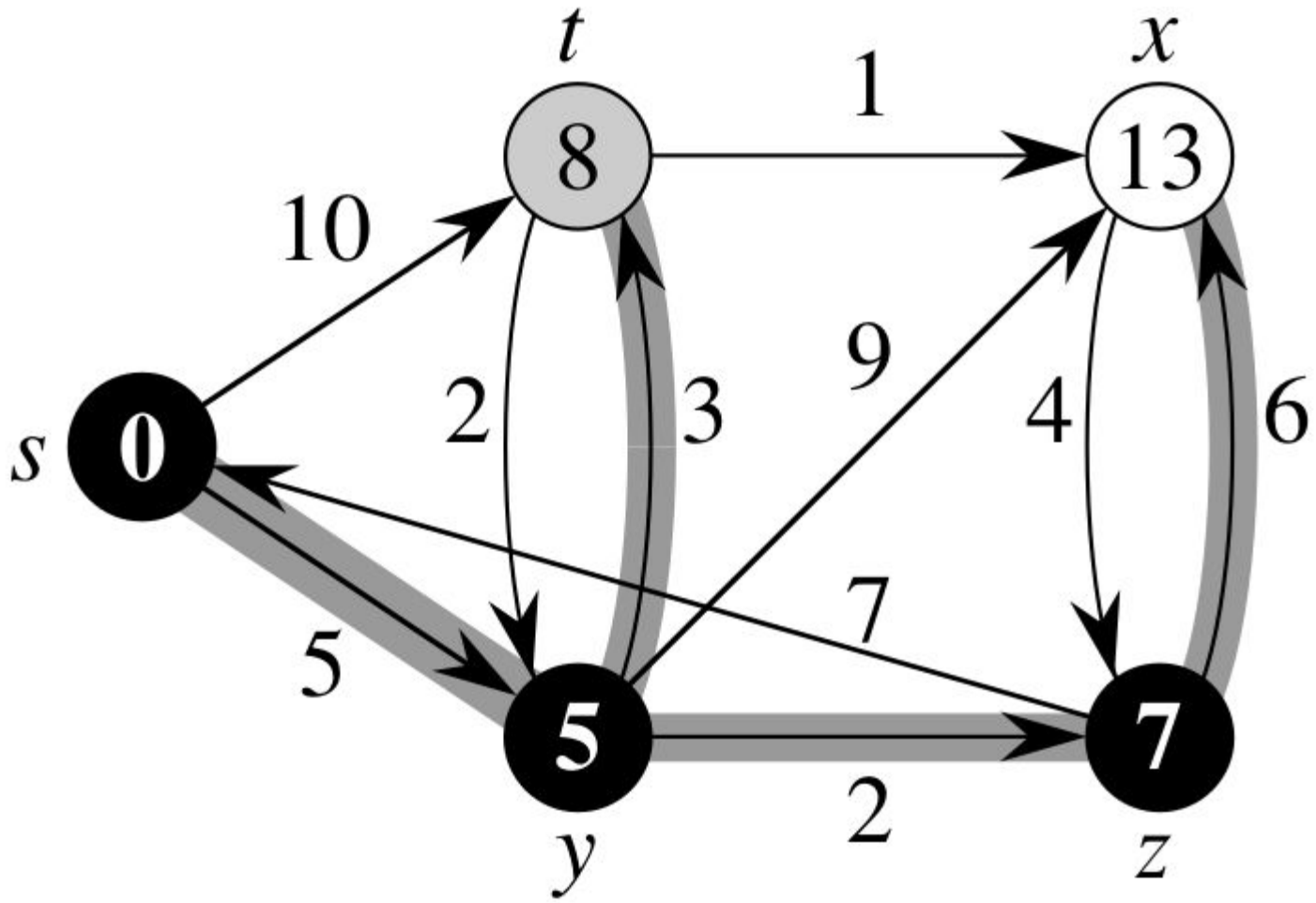
```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ V[G]
2        do d[v] ← ∞
3           π[v] ← NIL
4  d[s] ← 0
```

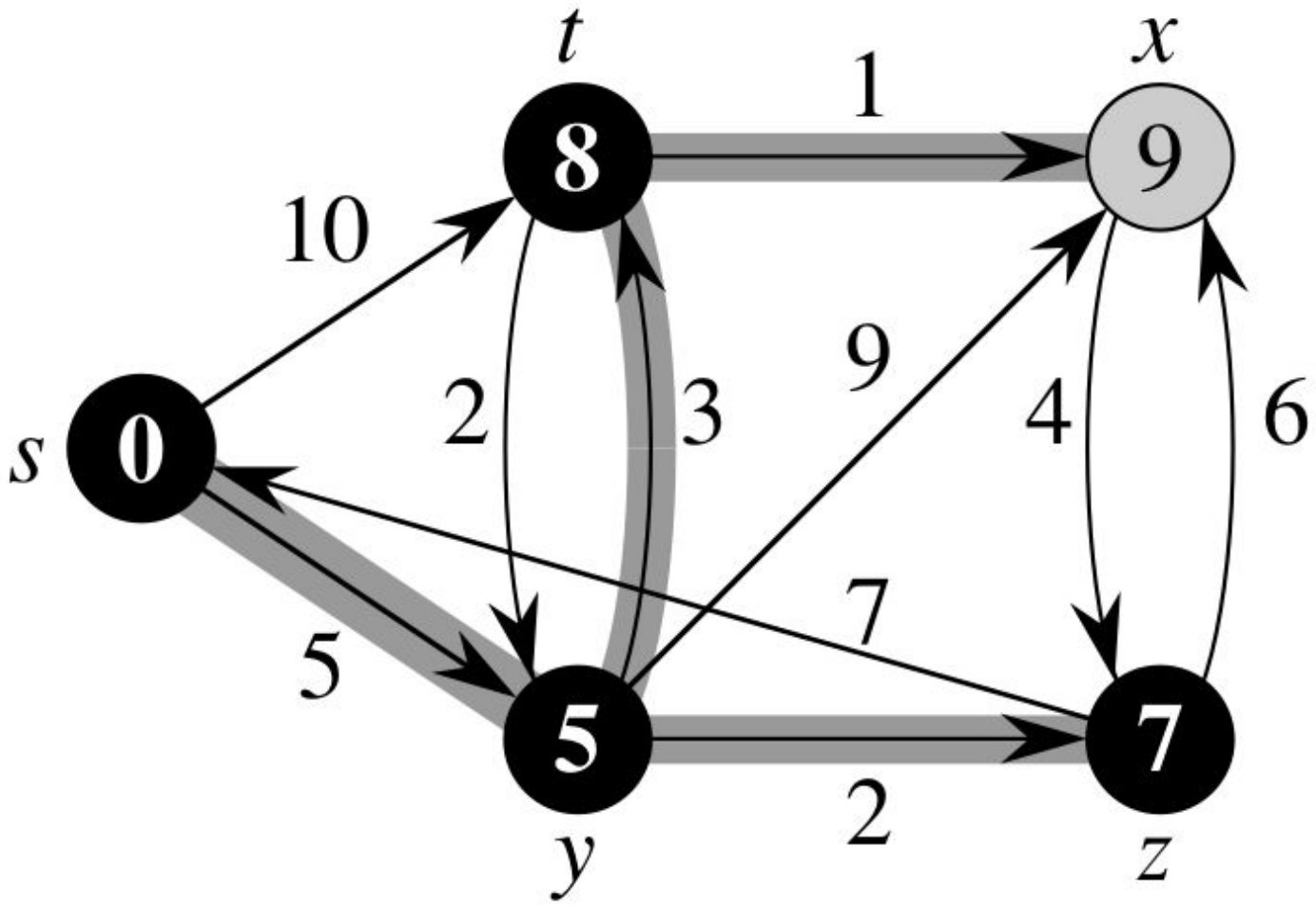
```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S ← ∅
3  Q ← V[G]
4  while Q ≠ ∅
5       do u ← EXTRACT-MIN(Q)
6          S ← S ∪ {u}
7          for each vertex v ∈ Adj[u]
8               do RELAX(u, v, w)
```

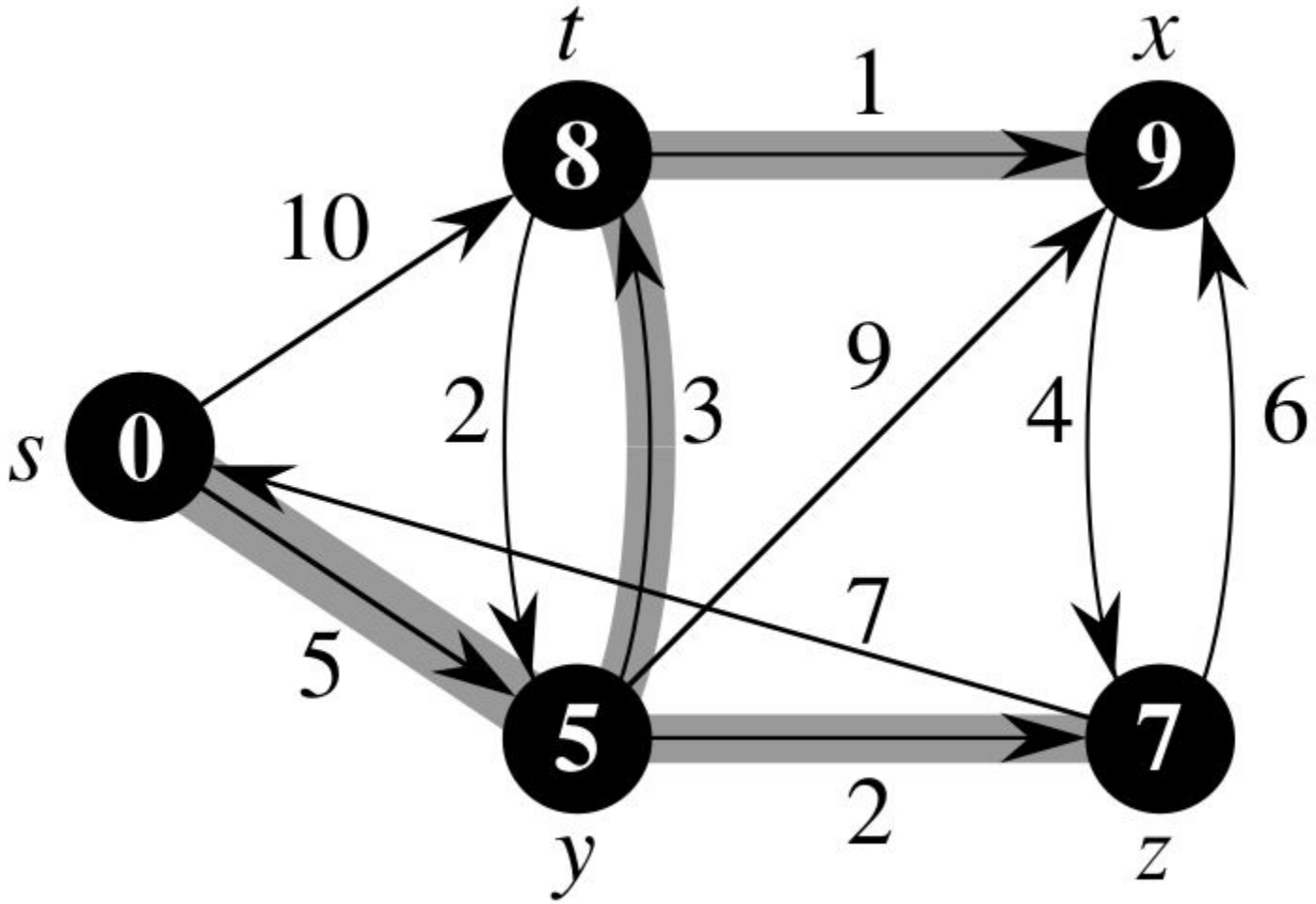

t
x
1
∞
∞
10
9
2
3
4
6
s
0
7
5
∞
∞
2
y
z


t
x
10
1
∞
10
s
0
2
3
9
4
6
7
5
5
∞
2
y
z


t
x
1
8
14
10
s
0
2
3
9
4
6
7
5
5
7
2
y
z


t
x
8
13
1
10
2
3
9
4
6
s
0
7
5
5
7
2
y
z


t
x
1
8
9
10
9
2
3
4
6
s
0
7
5
5
7
2
y
z


t
x
1
8
9
10
s
0
2
3
9
4
6
7
5
5
7
2
y
z

Análise de algoritmo

Linha por linha

Parâmetros

- |V|, número de vértices
- |E|, número de arestas
- $|E_V|$, número de arestas por vértice, $|E_V| = |E| / |V|$
- T_{POP}, tempo para obter o vértice de menor distância
- T_{PUSH}, tempo para adicionar/atualizar distância de um vértice na fila de prioridade.

```rust
// Runs |V|+1 times, takes Tpop
while let Some((u: usize, _)) = pq.pop() {
    // Runs |V| times, takes 1
    if visited[u] {
        // Runs |V| times, takes 1
        continue;
    }

    // Runs |V| times, takes 1
    visited[u] = true;

    // Runs |V| * (|Ev|+1) times, takes 1
    for &(v: usize, w: u64) in &graph.vertexes[u] {
        // Runs |V| * |Ev| times, takes 1
        if dist[v] > dist[u] + w {
            // Runs |V| * |Ev| times, takes 1
            dist[v] = dist[u] + w;

            // Runs |V| * |Ev| times, takes 1
            prev[v] = Some(u);

            // Runs |V| * |Ev| times, takes Tpush
            pq.push(item: v, priority: Reverse(dist[v]));
        }
    }
}
```

Linha por linha

Complexidade:

$O(|V| * T_{POP} + |V| * |E_V| * T_{PUSH})$

$O(|V| * T_{POP} + |E| * T_{PUSH})$

Em nossa implementação,

$T_{POP} = 1$, $T_{PUSH} = \lg |V|$, logo

$O(|V| + |E| * \lg |V|)$

```
// Runs |V|+1 times, takes Tpop
while let Some((u: usize, _)) = pq.pop() {
    // Runs |V| times, takes 1
    if visited[u] {
        // Runs |V| times, takes 1
        continue;
    }

    // Runs |V| times, takes 1
    visited[u] = true;

    // Runs |V| * (|Ev|+1) times, takes 1
    for &(v: usize, w: u64) in &graph.vertexes[u] {
        // Runs |V| * |Ev| times, takes 1
        if dist[v] > dist[u] + w {
            // Runs |V| * |Ev| times, takes 1
            dist[v] = dist[u] + w;

            // Runs |V| * |Ev| times, takes 1
            prev[v] = Some(u);

            // Runs |V| * |Ev| times, takes Tpush
            pq.push(item: v, priority: Reverse(dist[v]));
        }
    }
}
```

—

Linha por linha

Implementações ainda mais eficientes usando Fibonacci heap conseguem chegar a:

O(|V| lg |V| + |E|)

Para grafos densos, $|E|=|V|^2$:

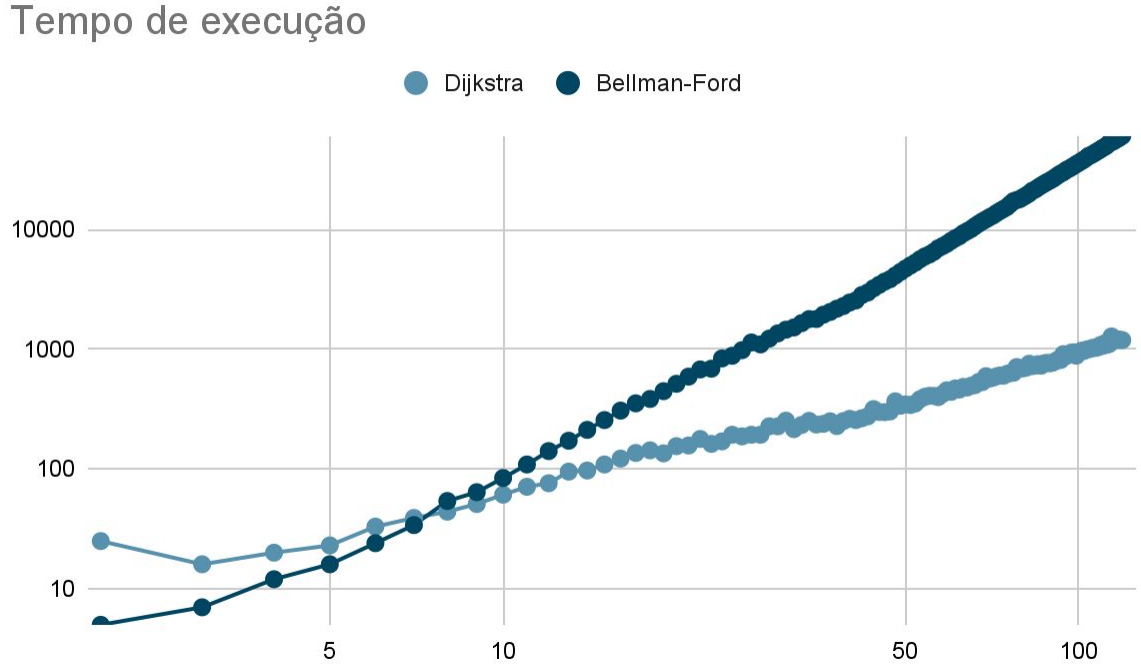
$O(|V|^2)$

```
// Runs |V|+1 times, takes Tpop
while let Some((u: usize, _)) = pq.pop() {
    // Runs |V| times, takes 1
    if visited[u] {
        // Runs |V| times, takes 1
        continue;
    }

    // Runs |V| times, takes 1
    visited[u] = true;

    // Runs |V| * (|Ev|+1) times, takes 1
    for &(v: usize, w: u64) in &graph.vertexes[u] {
        // Runs |V| * |Ev| times, takes 1
        if dist[v] > dist[u] + w {
            // Runs |V| * |Ev| times, takes 1
            dist[v] = dist[u] + w;

            // Runs |V| * |Ev| times, takes 1
            prev[v] = Some(u);

            // Runs |V| * |Ev| times, takes Tpush
            pq.push(item: v, priority: Reverse(dist[v]));
        }
    }
}
```

Comparação com outros algoritmos

Dijkstra	Bellman-Ford	Floyd-Warshall
$O(\|V\|^2)$	**$O(\|V\|\|E\|)$**	**$O(\|V\|^3)$**

Comparação com outros algoritmos



Implementação

Mostrar no VSCode

Conclusão

Dijkstra

- Rust é uma linguagem que permite facilmente o desenvolvimento do algoritmo de Dijkstra
- Algoritmo voltado para o shortest path problem (SPB) em grafos
- Tipo de algoritmo Greedy Strategy (guloso)
- Complexidade média $O(|V|^2)$ mais eficiente em tempo do que o de Bellman-Ford, mas exige mais memória por conta da fila de prioridade

Referências

- Imagens de www.stockfreeimages.com
- T. Cormen, C. Leiserson, R. Rivest, and C. Stein. T, Introduction to Algorithms,The MIT Press, 3rd edition, (2009)
- James F. Kurose, Ross - Computer Networking A Top-Down Approach (7th edition, 2016, Addison-Wesley)
- Mapa de estradas EUA: https://ontheworldmap.com/usa/usa-road-map.html

Obrigado!