Caminhos mínimos

Algoritmo de Bellman-Ford

Prof. Edson Alves

2018

Faculdade UnB Gama

Sumário

Algoritmo de Bellman-Ford

Caminhos mínimos

- Seja $G(V, E)$ um grafo e $u, v \in V$. Um caminho de u a v é uma sequência de M arestas $p = \{(a_0, a_1), (a_1, a_2), \ldots, (a_{M-1}, a_M)\}$ tal que $a_0 = u, a_M = v$ e, para cada par a, b de arestas consecutivas de p, o segundo vértice de a é igual ao primero vértice de b
- O conjunto C de todos os caminhos de u a v é dado por

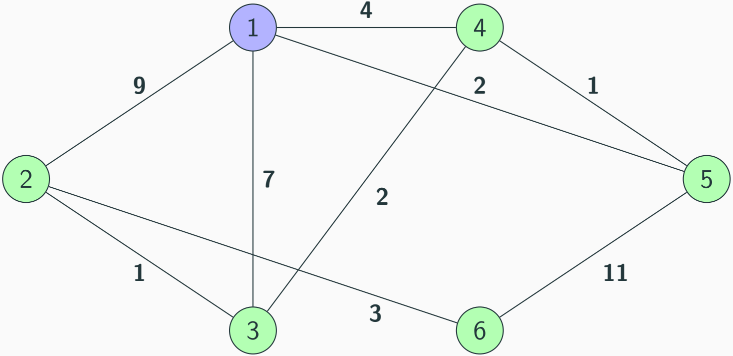
$$C(u, v) = \{p \subset E \mid p \text{ é caminho de } u \text{ a } v\}$$

- Se $C(u, v) \neq \emptyset$, o caminho de custo mínimo, ou simplesmente caminho mínimo, de u a v é um caminho, é o elemento de $m \in C$ tal que a soma dos pesos da arestas da sequência m é a menor possível
- Se o grafo não é ponderado, o caminho mínimo entre u e v pode ser obtido através de uma BFS

Algoritmo de Bellman-Ford

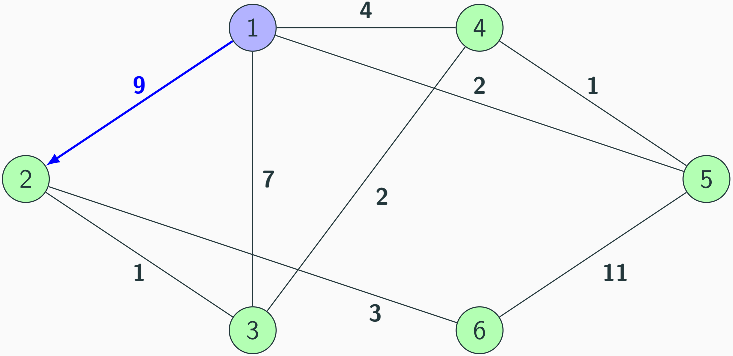
- O algoritmo de Bellman-Ford computa o caminho mínimo de todos os vértices de um grafo a um nó s dado
- É um algoritmo versátil, que pode processar grafos cujas arestas podem ter pesos negativos
- O único tipo de grafo que ele não processa são grafos com ciclos negativos, mas é capaz de detectar tais grafos
- Primeiramente ele inicializa a distância de s a s como zero e igual a infinito para todos os demais nós
- A cada iteração, ele visita todas as arestas na tentativa de encurtar um caminho já existente, até que não seja mais possível esta redução
- A complexidade é $O(VE)$, pois o número de arestas máximo em um caminho mínimo é igual a $V - 1$

Visualização do algoritmo de Bellman-Ford



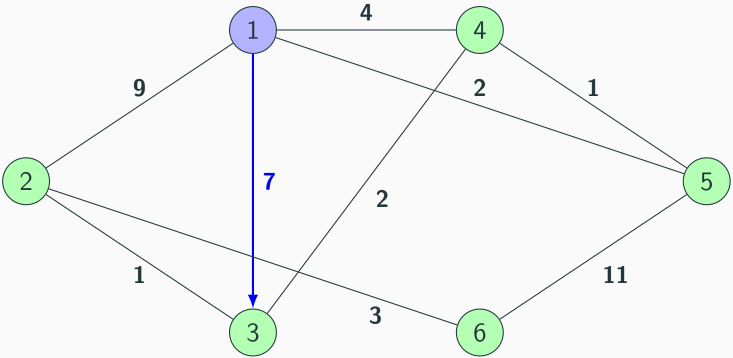
	1	2	3	4	5	6
Distâncias:	0	∞	∞	∞	∞	∞

Visualização do algoritmo de Bellman-Ford



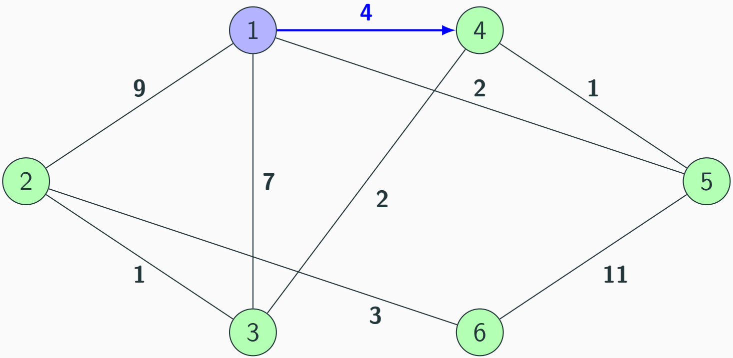
	1	2	3	4	5	6
Distâncias:	0	9	∞	∞	∞	∞

Visualização do algoritmo de Bellman-Ford



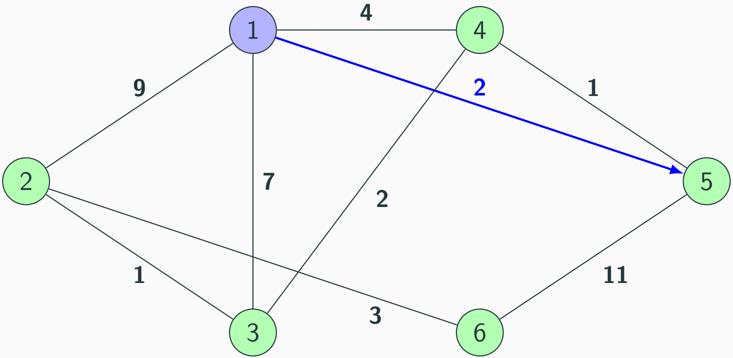
	1	2	3	4	5	6
Distâncias:	0	9	7	∞	∞	∞

Visualização do algoritmo de Bellman-Ford



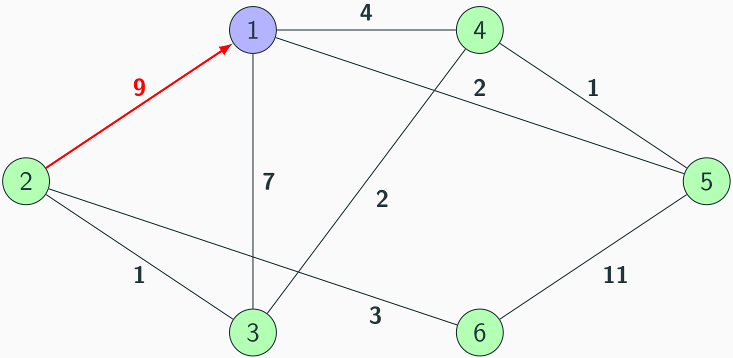
	1	2	3	4	5	6
Distâncias:	0	9	7	4	∞	∞

Visualização do algoritmo de Bellman-Ford



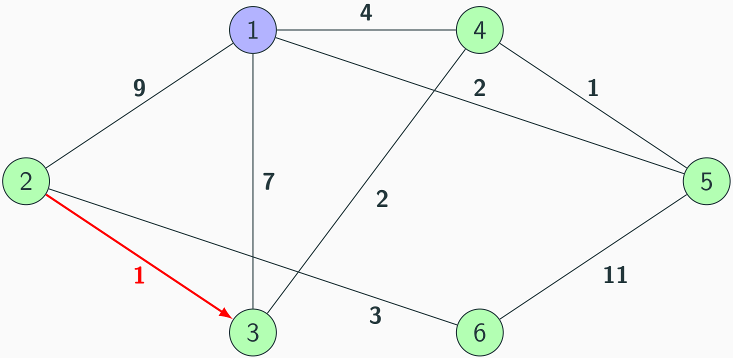
	1	2	3	4	5	6
Distâncias:	0	9	7	4	2	∞

Visualização do algoritmo de Bellman-Ford



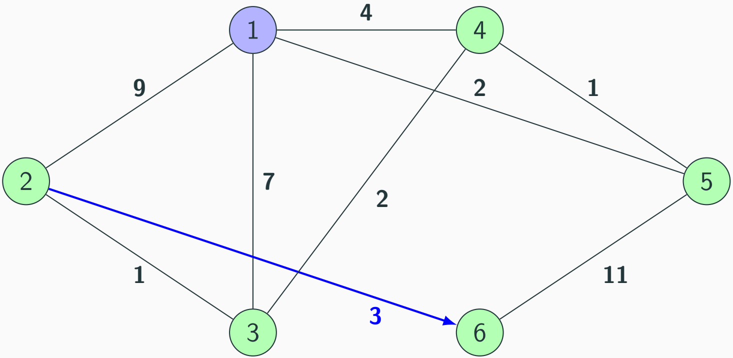
	1	2	3	4	5	6
Distâncias:	0	9	7	4	2	∞

Visualização do algoritmo de Bellman-Ford



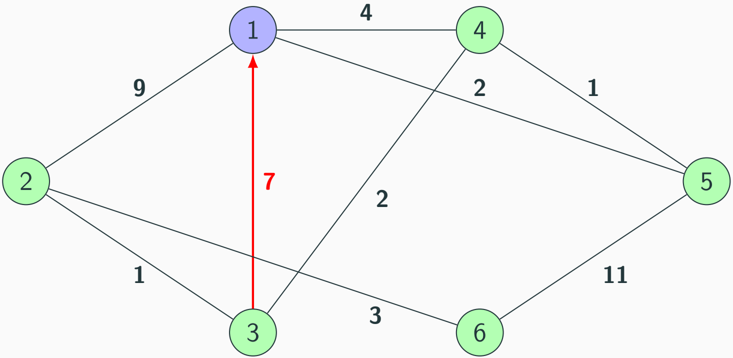
	1	2	3	4	5	6
Distâncias:	0	9	7	4	2	∞

Visualização do algoritmo de Bellman-Ford



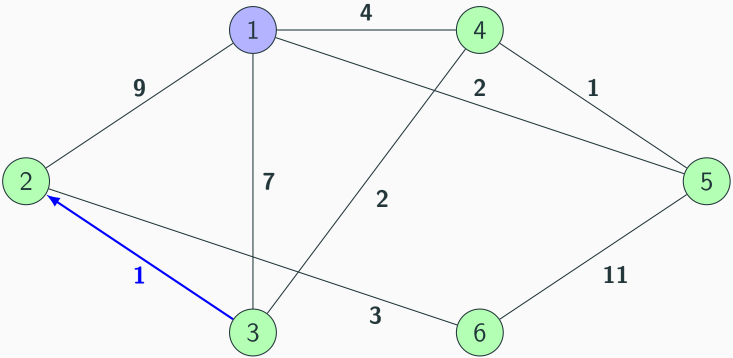
	1	2	3	4	5	6
Distâncias:	0	9	7	4	2	12

Visualização do algoritmo de Bellman-Ford



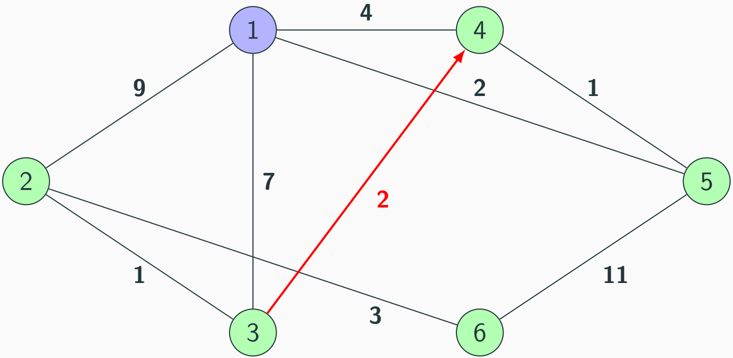
	1	2	3	4	5	6
Distâncias:	0	9	7	4	2	12

Visualização do algoritmo de Bellman-Ford



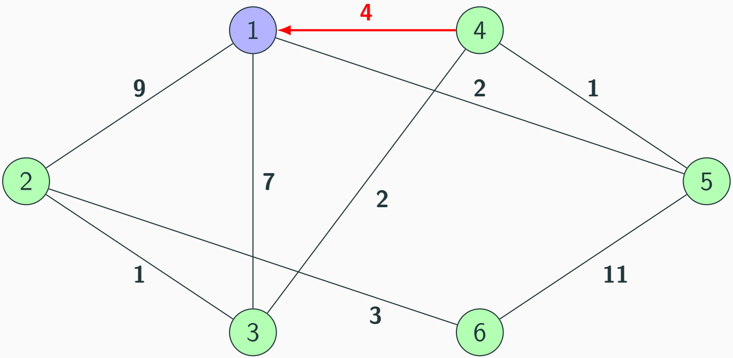
	1	2	3	4	5	6
Distâncias:	0	8	7	4	2	12

Visualização do algoritmo de Bellman-Ford



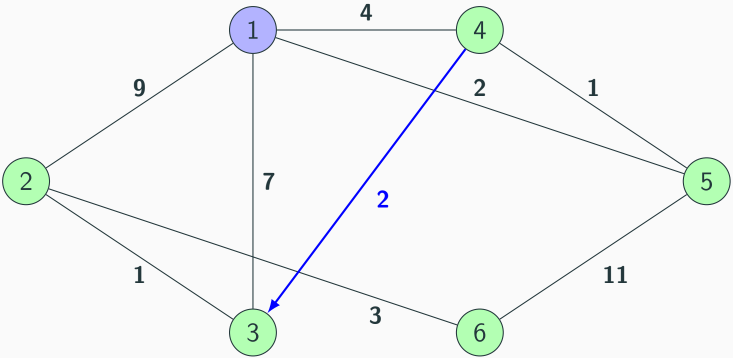
	1	2	3	4	5	6
Distâncias:	0	8	7	4	2	12

Visualização do algoritmo de Bellman-Ford



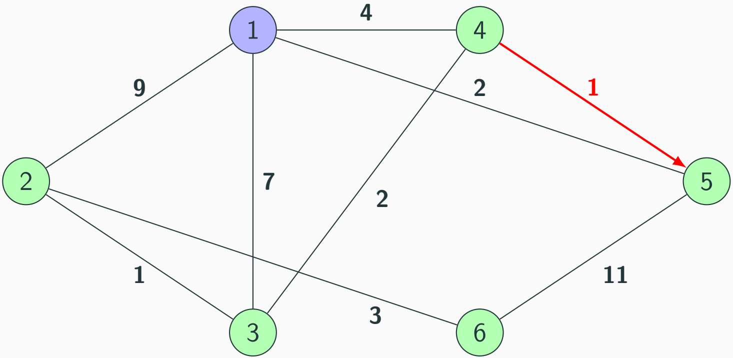
	1	2	3	4	5	6
Distâncias:	0	8	7	4	2	12

Visualização do algoritmo de Bellman-Ford



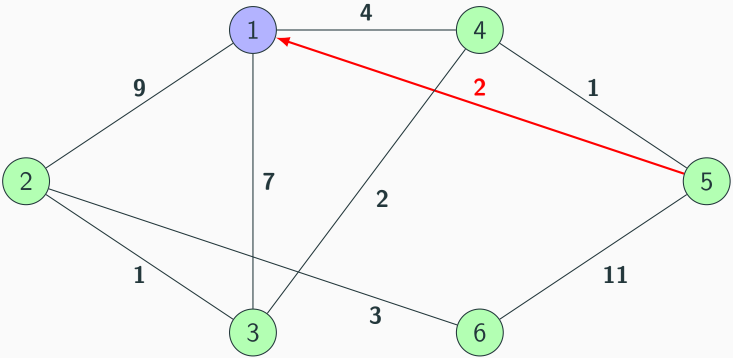
	1	2	3	4	5	6
Distâncias:	0	8	6	4	2	12

Visualização do algoritmo de Bellman-Ford



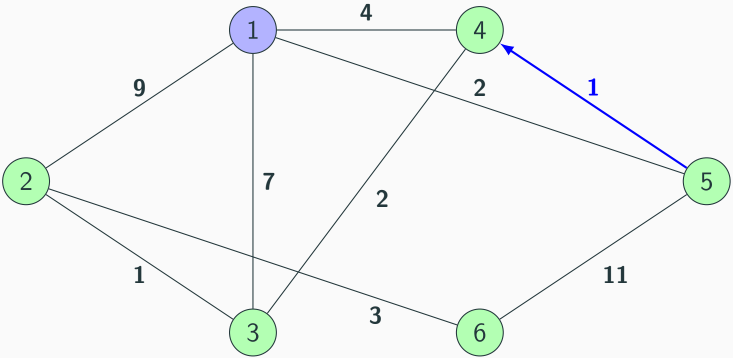
	1	2	3	4	5	6
Distâncias:	0	8	6	4	2	12

Visualização do algoritmo de Bellman-Ford



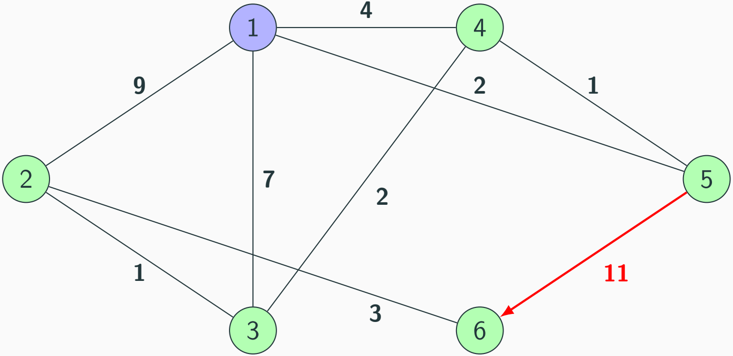
	1	2	3	4	5	6
Distâncias:	0	8	6	4	2	12

Visualização do algoritmo de Bellman-Ford



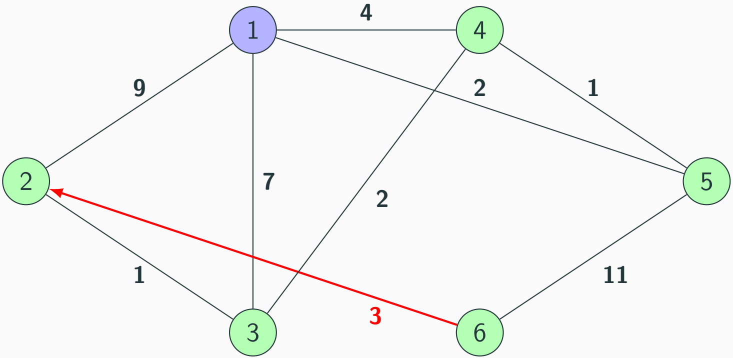
	1	2	3	4	5	6
Distâncias:	0	8	6	3	2	12

Visualização do algoritmo de Bellman-Ford



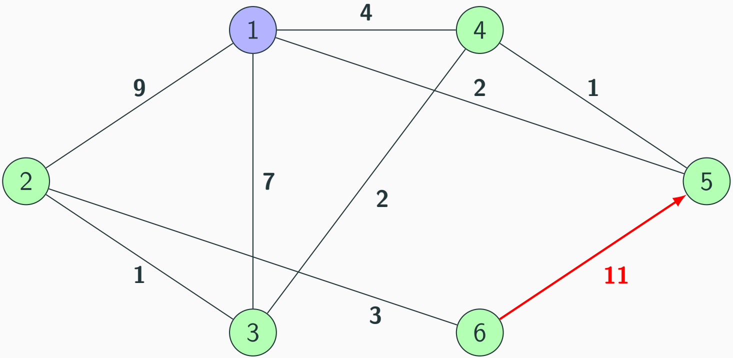
	1	2	3	4	5	6
Distâncias:	0	8	6	3	2	12

Visualização do algoritmo de Bellman-Ford



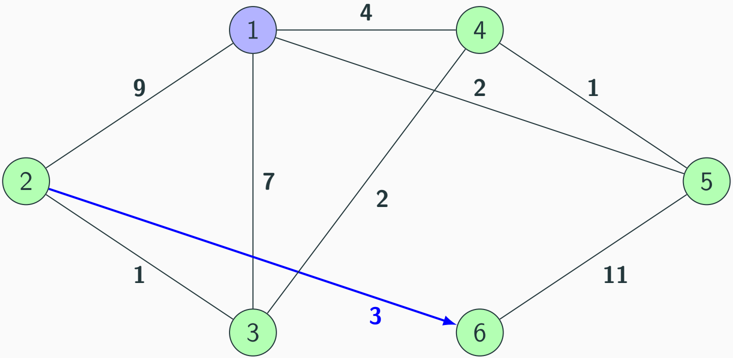
	1	2	3	4	5	6
Distâncias:	0	8	6	3	2	12

Visualização do algoritmo de Bellman-Ford



	1	2	3	4	5	6
Distâncias:	0	8	6	3	2	12

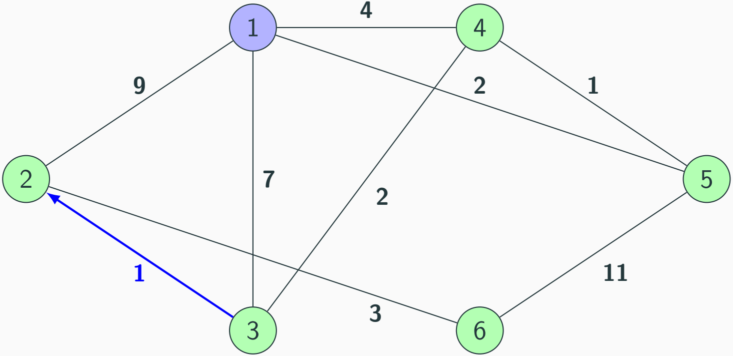
Visualização do algoritmo de Bellman-Ford



Round #2

	1	2	3	4	5	6
Distâncias:	0	8	6	3	2	11

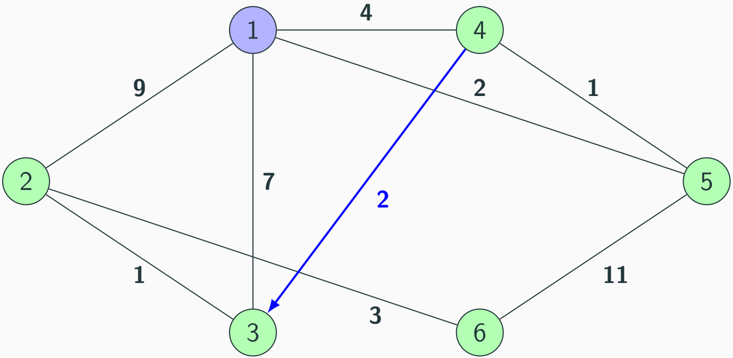
Visualização do algoritmo de Bellman-Ford



Round #2

	1	2	3	4	5	6
Distâncias:	0	7	6	3	2	11

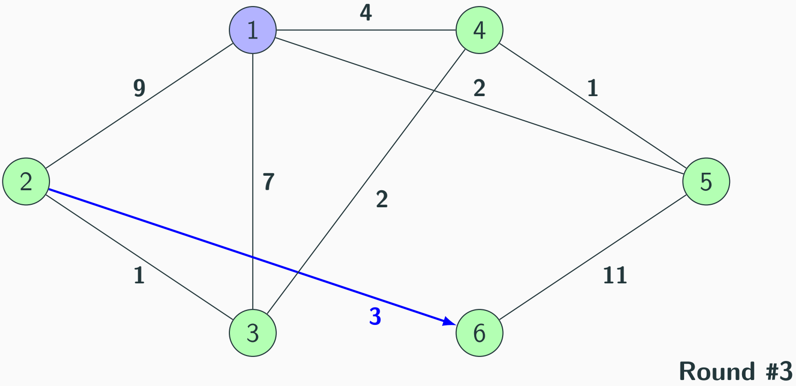
Visualização do algoritmo de Bellman-Ford



Round #2

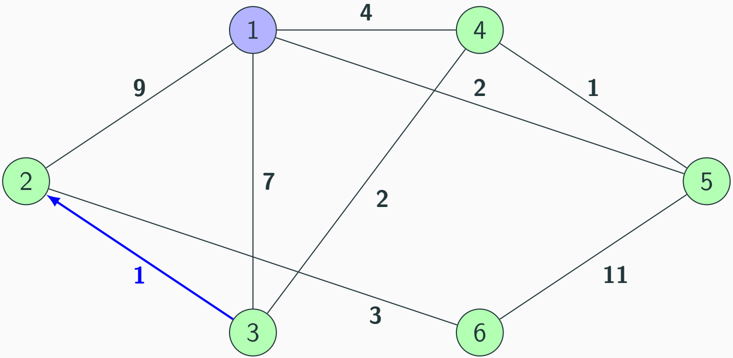
	1	2	3	4	5	6
Distâncias:	0	7	5	3	2	11

Visualização do algoritmo de Bellman-Ford



	1	2	3	4	5	6
Distâncias:	0	7	5	3	2	10

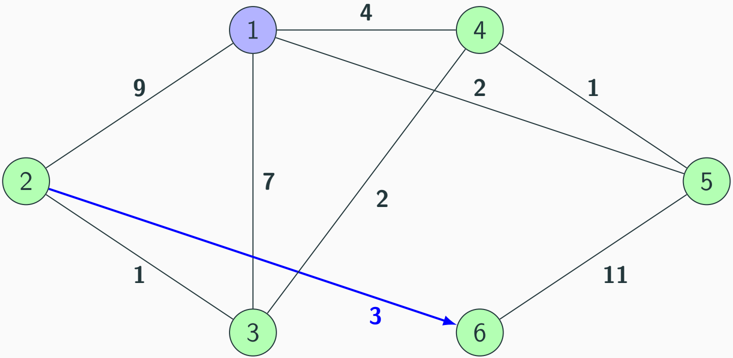
Visualização do algoritmo de Bellman-Ford



Round #3

	1	2	3	4	5	6
Distâncias:	0	6	5	3	2	10

Visualização do algoritmo de Bellman-Ford



Round #4

	1	2	3	4	5	6
Distâncias:	0	6	5	3	2	9

Implementação de Bellman-Ford em C++

```
#include <bits/stdc++.h>

using namespace std;
using edge = tuple<int, int, int>;

const int MAX { 100010 }, oo { 1000000010 };
int dist[MAX];

void bellman_ford(int s, int N, const vector<edge>& edges)
{
    for (int i = 1; i <= N; ++i)
        dist[i] = oo;

    dist[s] = 0;

    for (int i = 1; i <= N - 1; i++)
        for (const auto& [u, v, w] : edges)
            dist[v] = min(dist[v], dist[u] + w);
}

```

Implementação de Bellman-Ford em C++

```cpp
int main()
{
    vector<edge> edges { edge(1, 2, 9), edge(1, 3, 7), edge(1, 4, 4),
        edge(1, 5, 2), edge(2, 3, 1), edge(2, 6, 3), edge(3, 4, 2),
        edge(4, 5, 1), edge(5, 6, 11) };

    for (int i = edges.size() - 1; i >= 0; --i)
    {
        const auto& [u, v, w] = edges[i];
        edges.push_back(edge(v, u, w));
    }

    bellman_ford(1, 6, edges);

    for (int u = 1; u <= 6; ++u)
        cout << "Distância minima de 1 a " << u << ": " << dist[u] << '\n';

    return 0;
}
```

Identificação do caminho mínimo

- A implementação do algoritmo de Bellman-Ford apresentada computa a distância mínima entre qualquer vértice u conectado ao vértice s, mas não determina qual seria este caminho
- Para determinar o caminho, é preciso manter o vetor `pred`, onde `pred[u]` é o nó que antecede u no caminho mínimo que vai de s a u
- Inicialmente, todos os elementos deste vetor devem ser iguais a um valor sentinela, exceto o vértice s, que terá `pred[s] = s`
- Se a aresta (u, v) atualizar a distância `dist[v]`, então o predecessor deve ser atualizado também: `pred[v] = u`
- Deste modo, o caminho pode ser recuperado, passando por todos os predecessores até se atingir o nó um
- Se o predecessor de u for o valor sentinela, não há caminho de s a u no grafo

Implementação da identificação do caminho mínimo em C++

```cpp
#include <bits/stdc++.h>

using namespace std;
using edge = tuple<int, int, int>;

const int MAX { 100010 }, oo { 1000000010 };
int dist[MAX], pred[MAX];

void bellman_ford(int s, int N, const vector<edge>& edges)
{
    for (int i = 1; i <= N; ++i) {
        dist[i] = oo;
        pred[i] = -1;
    }

    dist[s] = 0; pred[s] = s;

```

```cpp
    for (int i = 1; i <= N - 1; i++)
        for (const auto& [u, v, w] : edges)
            if (dist[v] > dist[u] + w) {
                dist[v] = dist[u] + w;
                pred[v] = u;
            }
}

int main()
{
    vector<edge> edges { edge(1, 2, 9), edge(1, 3, 7), edge(1, 4, 4),
        edge(1, 5, 2), edge(2, 3, 1), edge(2, 6, 3), edge(3, 4, 2),
        edge(4, 5, 1), edge(5, 6, 11) };

    for (int i = edges.size() - 1; i >= 0; --i)
    {
        const auto& [u, v, w] = edges[i];
        edges.push_back(edge(v, u, w));
    }

    bellman_ford(1, 6, edges);
```

Implementação da identificação do caminho mínimo em C++

```cpp
    for (int u = 1; u <= 6; ++u)
    {
        cout << "dist(1," << u << ") = " << dist[u] << endl;

        vector<int> path;
        auto p = u;

        while (p != 1) {
            path.push_back(p);
            p = pred[p];
        }

        path.push_back(1);
        reverse(path.begin(), path.end());

        for (size_t i = 0; i < path.size(); ++i)
            cout << path[i] << (i + 1 == path.size() ? "\n" : " -> ");
    }

    return 0;
}
```

Ciclos negativos

- Um grafo G tem um ciclo negativo quando a soma dos pesos das arestas de um ciclo resultam em um valor menor do que zero
- A presença de um ciclo negativo faz com que, a cada iteração, o algoritmo de Bellman-Ford atualize ao menos uma distância
- Desta forma, o próprio algoritmo pode ser usado para detectar tais ciclos
- Basta iterar o algoritmo mais uma vez após o seu término: caso o algoritmo faça alguma atualização nas distâncias, há um ciclo negativo no grafo
- Tal estratégia identificará tais ciclos no componente conectado do grafo, independentemente do nó inicial escolhido

Implementação da identificação de ciclos negativos

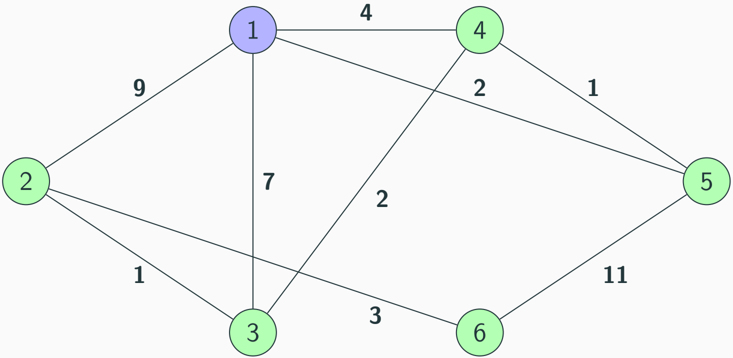
```cpp
bool has_negative_cycle(int s, int N, const vector<edge>& edges)
{
    for (int i = 1; i <= N; ++i)
        dist[i] = oo;

    dist[s] = 0;

    for (int i = 1; i <= N - 1; i++)
        for (const auto& [u, v, w] : edges)
            dist[v] = min(dist[v], dist[u] + w);

    for (const auto& [u, v, w] : edges)
        if (dist[v] > dist[u] + w)
            return true;

    return false;
}
```

SPFA

Shortest Path Fast Algorithm

- O SPFA (*Shortest Path Fast Algorithm*) é uma variante do algoritmo de Bellman-Ford, que reduz o tempo de execução através de uma escolha diferente das arestas a serem processadas
- É criada uma fila de nós a serem processados, e inicialmente o nó s é inserido na fila
- A fila é processada um nó por vez, e caso uma aresta (u, v) reduza a distância até v, o nó v é inserido na fila
- Embora tenha melhor tempo de execução do que o algoritmo de Bellman-Ford, a complexidade no pior caso ainda é de $O(VE)$.

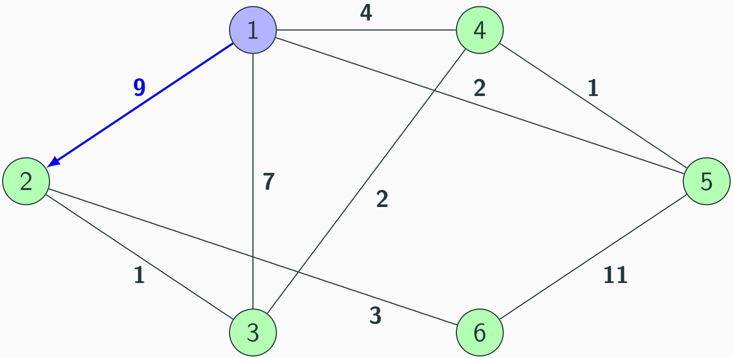
Visualização do algoritmo SPFA



Fila: 1

	1	2	3	4	5	6
Distâncias:	0	∞	∞	∞	∞	∞

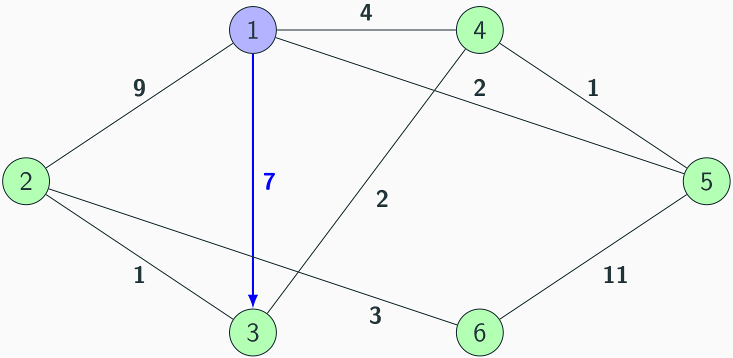
Visualização do algoritmo SPFA



Fila: 2

	1	2	3	4	5	6
Distâncias:	0	9	∞	∞	∞	∞

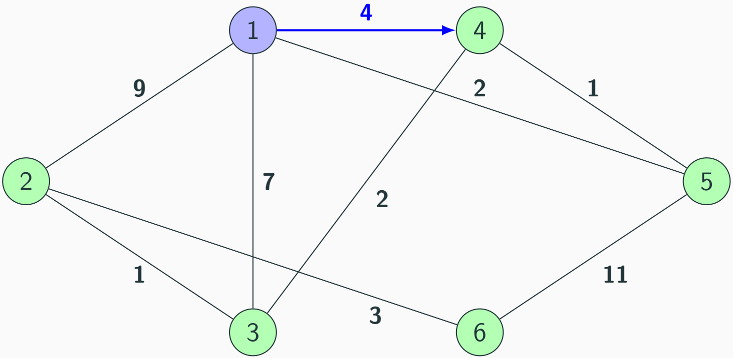
Visualização do algoritmo SPFA



Fila: 2 3

	1	2	3	4	5	6
Distâncias:	0	9	7	∞	∞	∞

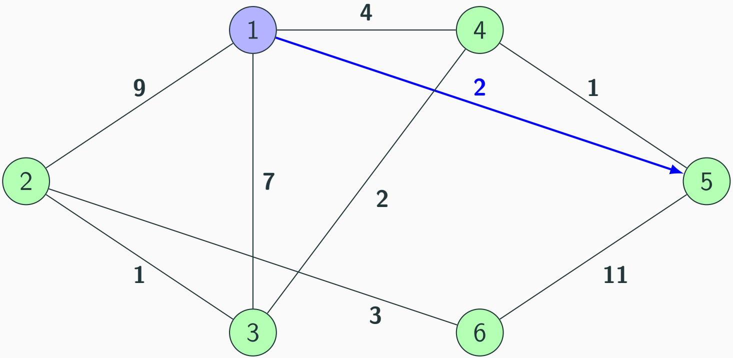
Visualização do algoritmo SPFA



Fila: 2 3 4

	1	2	3	4	5	6
Distâncias:	0	9	7	4	∞	∞

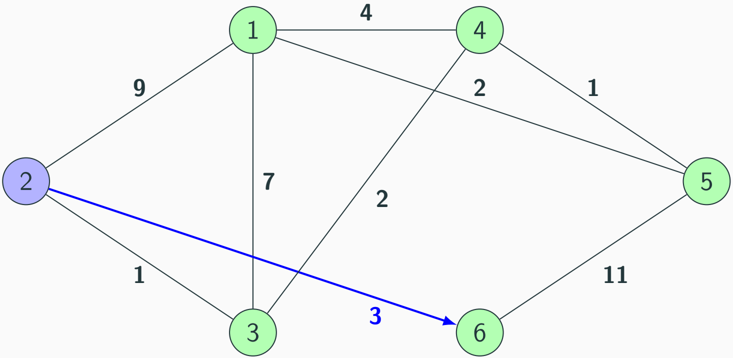
Visualização do algoritmo SPFA



Fila: 2 3 4 5

	1	2	3	4	5	6
Distâncias:	0	9	7	4	2	∞

Visualização do algoritmo SPFA

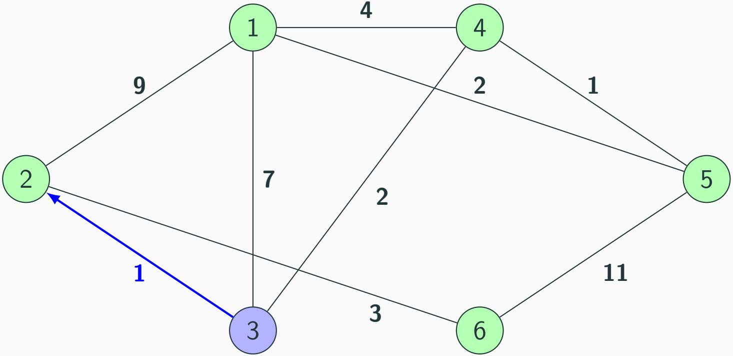


Fila: 3 4 5 6

Distâncias:

1	2	3	4	5	6
0	9	7	4	2	12

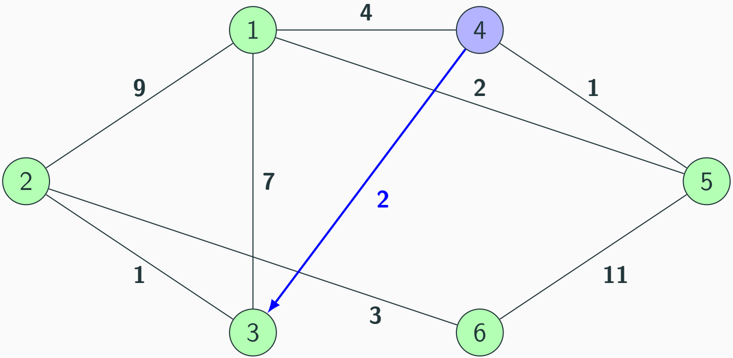
Visualização do algoritmo SPFA



Fila: 4 5 6 2

	1	2	3	4	5	6
Distâncias:	0	8	7	4	2	12

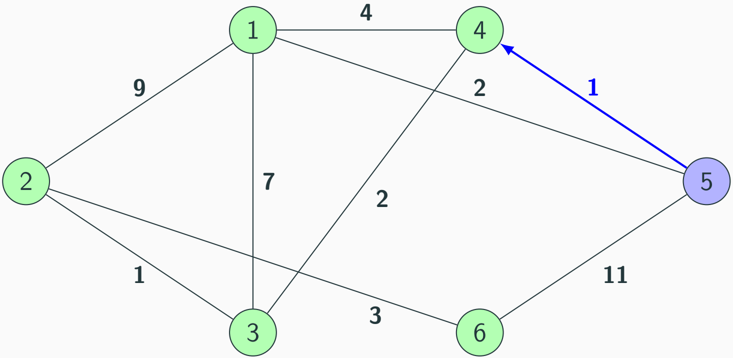
Visualização do algoritmo SPFA



Fila: 5 6 2 3

	1	2	3	4	5	6
Distâncias:	0	8	6	4	2	12

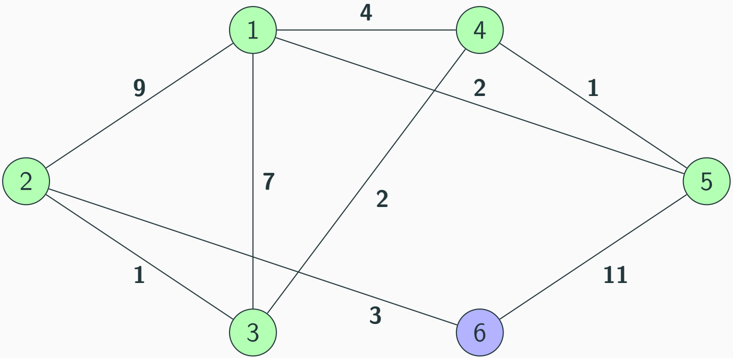
Visualização do algoritmo SPFA



Fila: 6 2 3 4

	1	2	3	4	5	6
Distâncias:	0	8	6	3	2	12

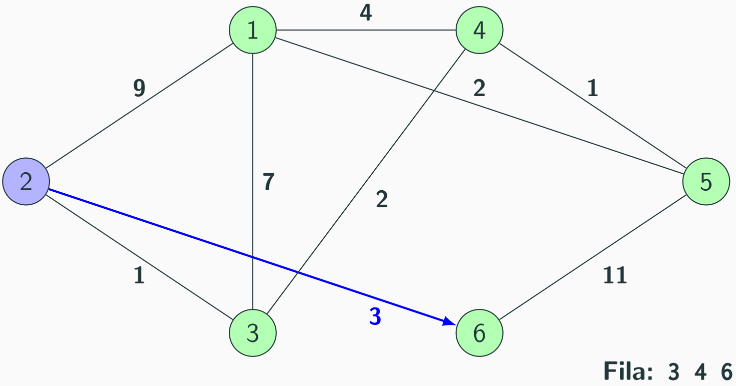
Visualização do algoritmo SPFA



Fila: 2 3 4

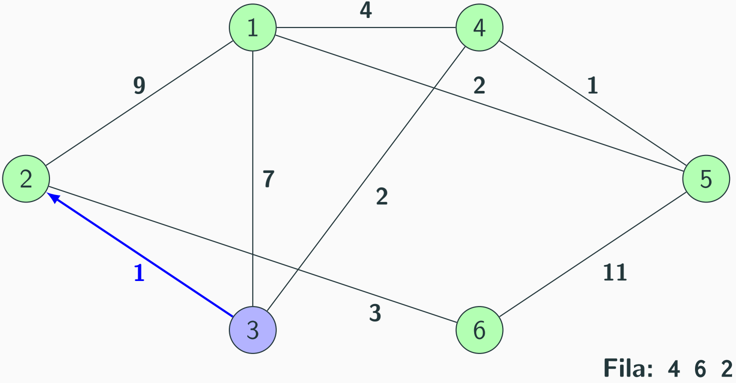
	1	2	3	4	5	6
Distâncias:	0	8	6	3	2	12

Visualização do algoritmo SPFA



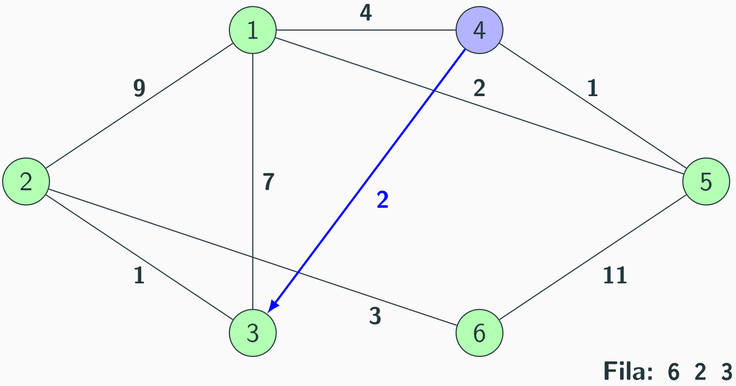
	1	2	3	4	5	6
Distâncias:	0	8	6	3	2	11

Visualização do algoritmo SPFA



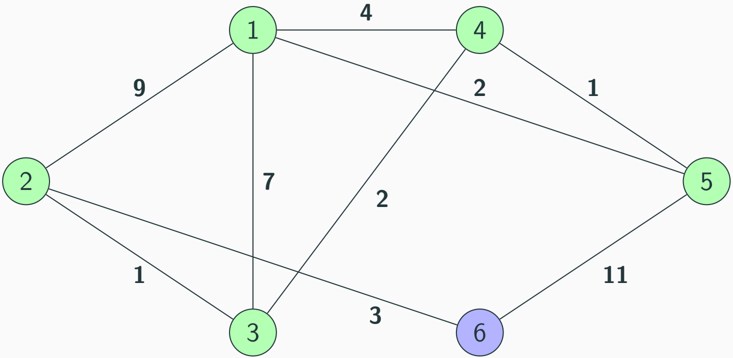
	1	2	3	4	5	6
Distâncias:	0	7	6	3	2	11

Visualização do algoritmo SPFA



	1	2	3	4	5	6
Distâncias:	0	7	5	3	2	11

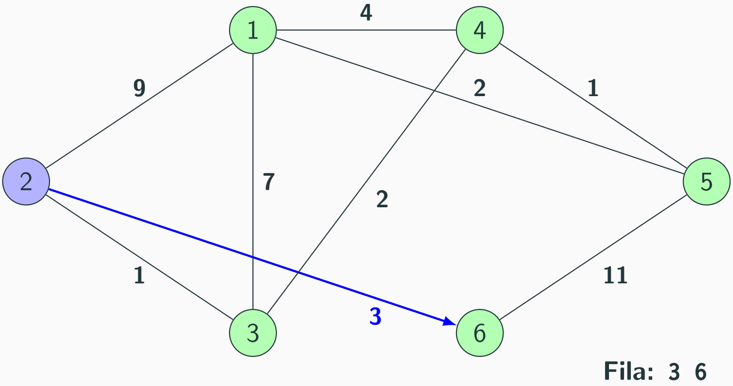
Visualização do algoritmo SPFA



Fila: 2 3

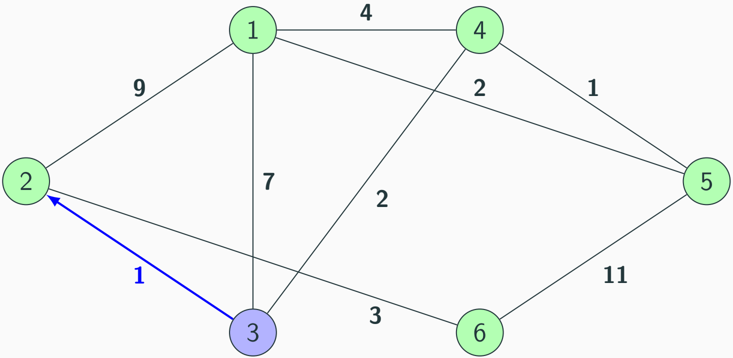
	1	2	3	4	5	6
Distâncias:	0	7	5	3	2	11

Visualização do algoritmo SPFA



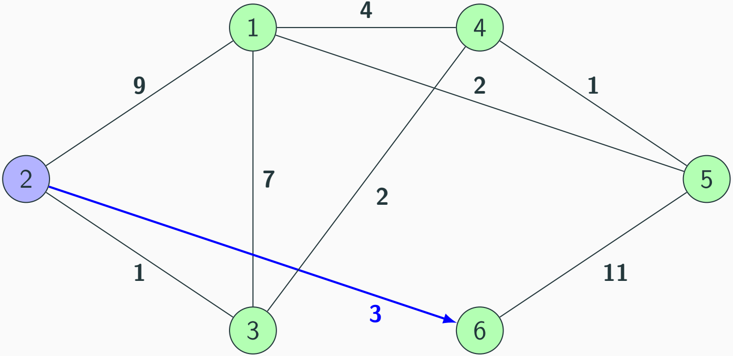
	1	2	3	4	5	6
Distâncias:	0	7	5	3	2	10

Visualização do algoritmo SPFA



	1	2	3	4	5	6
Distâncias:	0	6	5	3	2	10

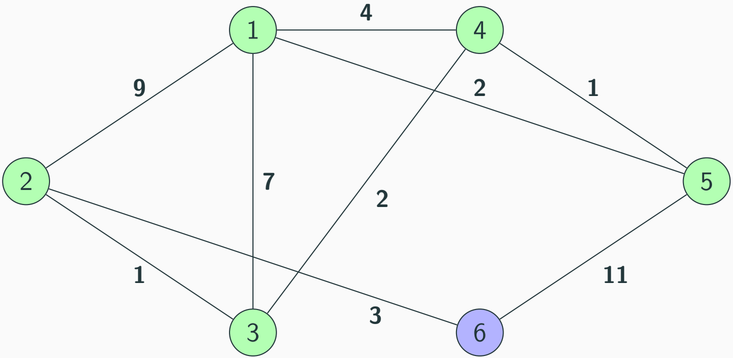
Visualização do algoritmo SPFA



Fila: 6

	1	2	3	4	5	6
Distâncias:	0	6	5	3	2	9

Visualização do algoritmo SPFA



Fila:

	1	2	3	4	5	6
Distâncias:	0	6	5	3	2	9

Implementação do SPFA em C++

```cpp
#include <bits/stdc++.h>

using namespace std;
using ii = pair<int, int>;
using edge = tuple<int, int, int>;

const int MAX { 100010 }, oo { 1000000010 };
vector<ii> adj[MAX];
int dist[MAX];

void spfa(int s, int N) {
    bitset<MAX> in_queue;

    for (int i = 1; i <= N; ++i)
        dist[i] = oo;

    dist[s] = 0;

    queue<int> q;
    q.push(s);
    in_queue[s] = true;
```

Implementação do SPFA em C++

```cpp
    while (not q.empty())
    {
        auto u = q.front();
        q.pop();
        in_queue[u] = false;

        for (const auto& [v, w] : adj[u])
        {
            if (dist[v] > dist[u] + w)
            {
                dist[v] = dist[u] + w;

                if (not in_queue[v])
                {
                    q.push(v);
                    in_queue[v] = true;
                }
            }
        }
    }
}
```

Implementação do SPFA em C++

```cpp
int main()
{
    vector<edge> edges { edge(1, 2, 9), edge(1, 3, 7), edge(1, 4, 4),
        edge(1, 5, 2), edge(2, 3, 1), edge(2, 6, 3), edge(3, 4, 2),
        edge(4, 5, 1), edge(5, 6, 11) };

    for (const auto& [u, v, w] : edges)
    {
        adj[u].push_back(ii(v, w));
        adj[v].push_back(ii(u, w));
    }

    spfa(1, 6);

    for (int u = 1; u <= 6; ++u)
        cout << "Distância minima de 1 a " << u << ": " << dist[u] << '\n';

    return 0;
}
```

SPFA e ciclos negativos

- A presença de ciclos negativos pode levar o SPFA a um laço infinito, pois a fila nunca ficaria vazia neste caso
- Para evitar tal situação, é preciso manter um registro do número de vezes que um nó entrou na fila
- Se um mesmo nó tiver entrado V vezes na fila, o grafo tem um ciclo negativo
- Com este cuidado adicional, a implementação do SPFA é mais longa do que a do algoritmo de Bellman-Ford, mas produz um tempo de execução menor
- Para grafos sem a presença de arestas negativas, contudo, há um algoritmo mais eficiente para o mesmo problema: o algoritmo de Djikstra

Referências

1. **HALIM**, Felix; **HALIM**, Steve. *Competitive Programming 3*, 2010.
2. **LAAKSONEN**, Antti. *Competitive Programmer's Handbook*, 2018.
3. **SKIENA**, Steven S.; **REVILLA**, Miguel A. *Programming Challenges*, 2003.