Caminhos mínimos

Algoritmo de Dijkstra

Prof. Edson Alves

2018

Faculdade UnB Gama

Sumário

Algoritmo de Dijkstra

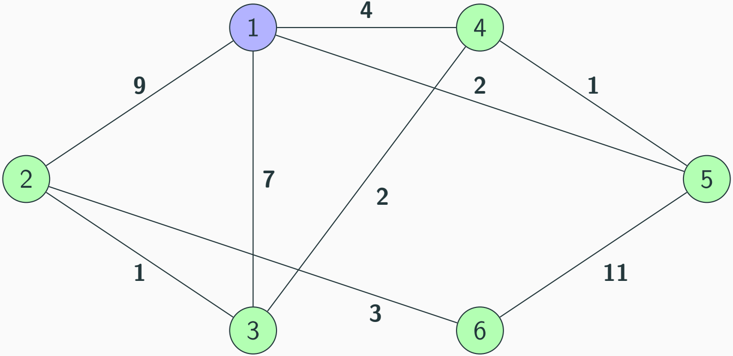
Dijkstra $\times$ Bellman-Ford

- Assim como o algoritmo de Bellman-Ford, o algoritmo de Dijkstra computa as distâncias mínimas de todos os vértices u de um grafo G a um nó s dado
- Por assumir que todas as arestas não tem peso negativo, este algoritmo tem menor complexidade assintótica e pode processar grafos com um maior número de nós em relação ao algortimo de Bellman-Ford
- Contudo, ele não deve ser usado em grafos ponderados com arestas com pesos negativos, pois os resultados produzidos não estarão corretos
- A eficiência do algoritmo provém do fato de que cada aresta do grafo é processada uma única vez
- Isto se dá através de uma escolha inteligente da ordem de processamento dos vértices

Algoritmo de Dijkstra

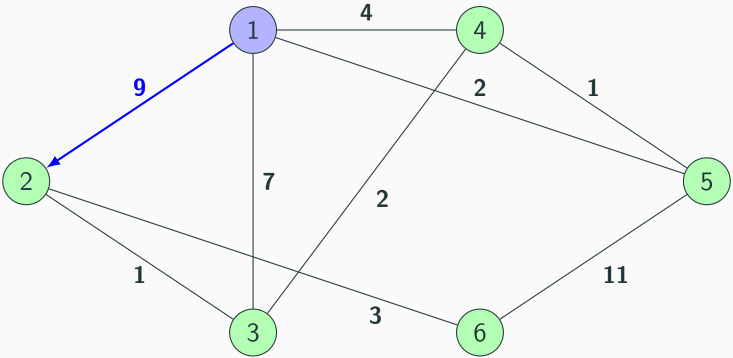
- Inicialmente, a distância de s a s é igual a zero, e todas as demais distâncias são iguais a infinito
- A cada iteração, o algoritmo escolhe o nó u mais próximo de s que ainda não foi processado
- Todas as arestas de partem de u então são processadas, atualizando as distâncias quando possível
- Esta operação de atualização de distância é chamada relaxamento
- Para escolher o próximo nó a ser processado de forma eficiente, é utilizada uma fila com prioridade
- Desta forma, a complexidade do algortimo é $O(V + E \log E)$
- Se o grafo for denso, é possível processar aproximadamente 1.000 vértices
- Se o grafo for esparso, é possível computar até um milhão de vértices em segundos

Visualização do algoritmo de Dijkstra



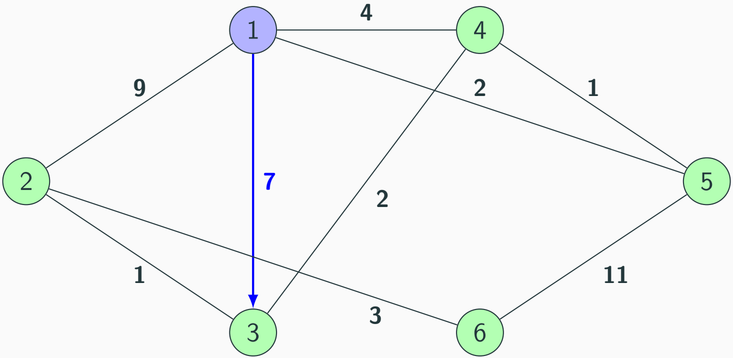
	1	2	3	4	5	6
Distância ao nó 1:	0	∞	∞	∞	∞	∞

Visualização do algoritmo de Dijkstra



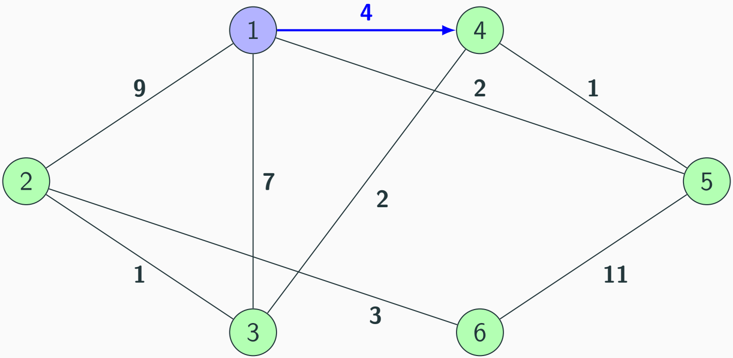
	1	2	3	4	5	6
Distância ao nó 1:	0	9	∞	∞	∞	∞

Visualização do algoritmo de Dijkstra



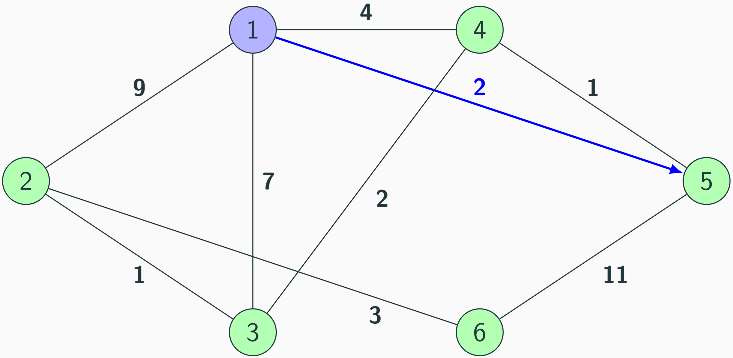
	1	2	3	4	5	6
Distância ao nó 1:	0	9	7	∞	∞	∞

Visualização do algoritmo de Dijkstra



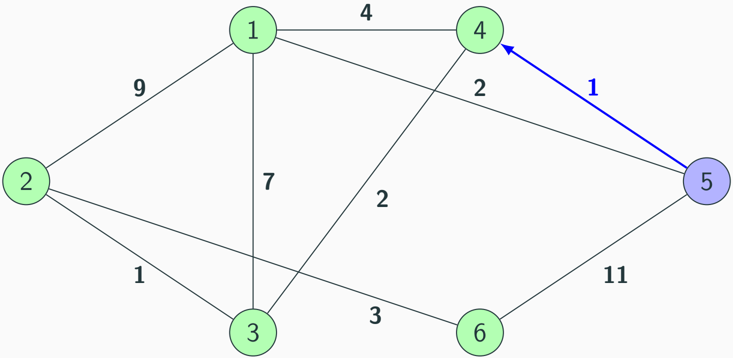
	1	2	3	4	5	6
Distância ao nó 1:	0	9	7	4	∞	∞

Visualização do algoritmo de Dijkstra



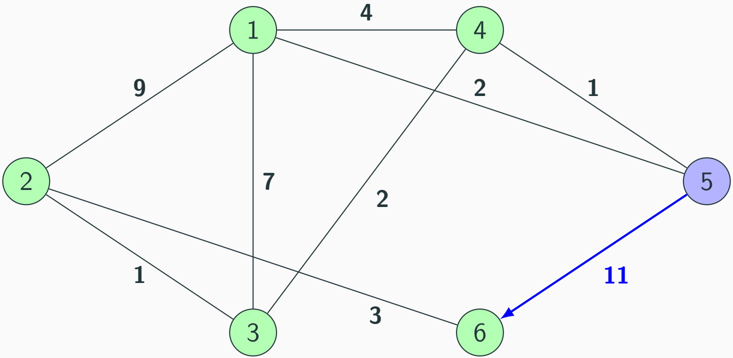
	1	2	3	4	5	6
Distância ao nó 1:	0	9	7	4	2	∞

Visualização do algoritmo de Dijkstra



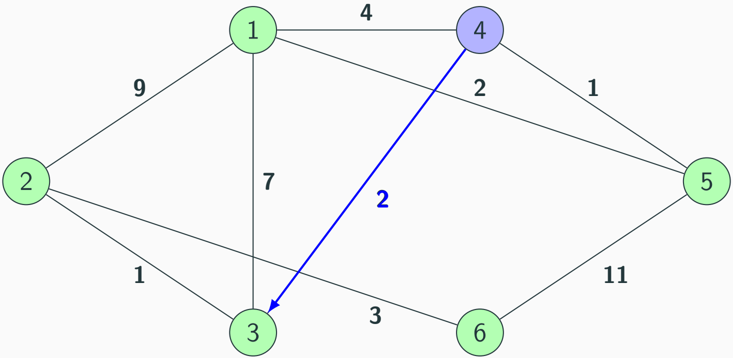
	1	2	3	4	5	6
Distância ao nó 1:	0	9	7	3	2	∞

Visualização do algoritmo de Dijkstra



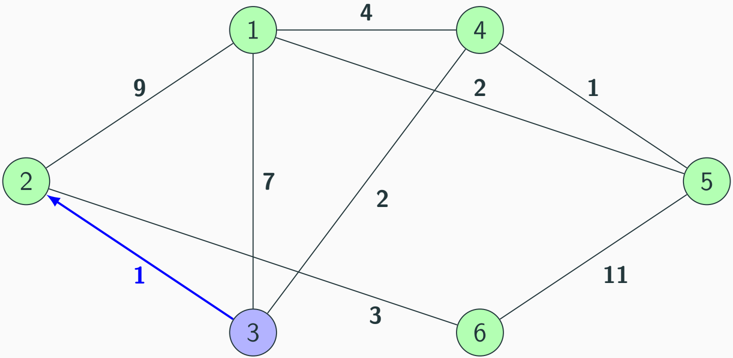
	1	2	3	4	5	6
Distância ao nó 1:	0	9	7	3	2	13

Visualização do algoritmo de Dijkstra



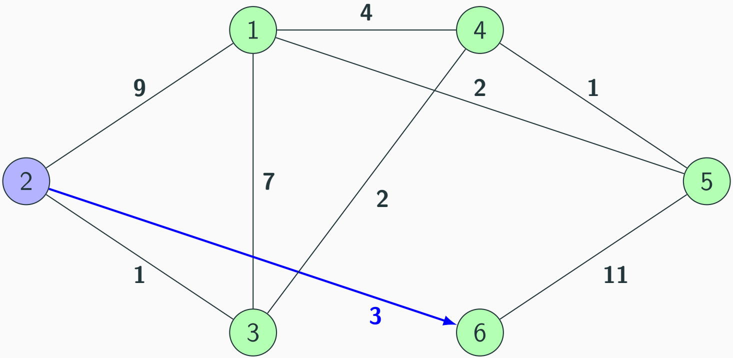
	1	2	3	4	5	6
Distância ao nó 1:	0	9	5	3	2	13

Visualização do algoritmo de Dijkstra



	1	2	3	4	5	6
Distância ao nó 1:	0	6	5	3	2	13

Visualização do algoritmo de Dijkstra



	1	2	3	4	5	6
Distância ao nó 1:	0	6	5	3	2	9

Implementação do algoritmo de Dijkstra 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 };
int dist[MAX];
vector<ii> adj[MAX];
bitset<MAX> processed;

void dijkstra(int s, int N)
{
    for (int i = 1; i <= N; ++i)
        dist[i] = oo;

    dist[s] = 0;
    processed.reset();

    priority_queue<ii, vector<ii>, greater<ii>> pq;
    pq.push(ii(0, s));
```

Implementação do algoritmo de Dijkstra em C++

```cpp

    while (not pq.empty())
    {
        auto [d, u] = pq.top();
        pq.pop();

        if (processed[u])
            continue;

        processed[u] = true;

        for (const auto& [v, w] : adj[u])
        {
            if (dist[v] > d + w) {
                dist[v] = d + w;
                pq.push(ii(dist[v], v));
            }
        }
    }
}

```

Implementação do algoritmo de Dijkstra 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));
    }

    dijkstra(1, 6);

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

    return 0;
}
```

Caminhos mínimos

Identificação do caminho mínimo

- Assim como no algoritmo de Bellman-Ford, é possível recuperar a sequência de arestas que compõem o caminho mínimo
- 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

Recuperação do caminho mínimo

```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 };
int dist[MAX], pred[MAX];
vector<ii> adj[MAX];
bitset<MAX> processed;

void dijkstra(int s, int N)
{
    for (int i = 1; i <= N; ++i) {
        dist[i] = oo;
        pred[i] = -1;
    }

    dist[s] = 0;
    pred[s] = s;
    processed.reset();
```

Recuperação do caminho mínimo

```cpp

    priority_queue<ii, vector<ii>, greater<ii>> pq;
    pq.push(ii(0, s));

    while (not pq.empty()) {
        auto [d, u] = pq.top();
        pq.pop();

        if (processed[u])
            continue;

        processed[u] = true;

        for (const auto& [v, w] : adj[u]) {
            if (dist[v] > d + w) {
                dist[v] = d + w;
                pq.push(ii(dist[v], v));
                pred[v] = u;
            }
        }
    }
```

```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));
    }

    dijkstra(1, 6);

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

        vector<int> path;
```

Recuperação do caminho mínimo

```cpp
        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;
}
```

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.