

Implementação 3 - Grafos

Daniel Salgado, Arthur Martinho

10/10/2024

1 Algoritmo de Dijkstra em C++

```
1 #include <iostream>
2 #include <vector>
3 #include <climits> // Para usar INT_MAX como infinito
4 #include <algorithm> // Para usar reverse
5
6 using namespace std;
7
8 struct Aresta {
9     int destino, peso;
10 };
11
12 struct Vertice {
13     int distancia = INT_MAX; // Dist ncia inicial infinito
14     int antecessor = -1;     // Antecessor inicial indefinido
15     bool visitado = false;   // Indica se o v rtice j foi visitado
16 };
17
18 // Fun o que mostra o caminho e a dist ncia de origem at destino
19 void mostrarCaminho(int destino, int origem, const vector<Vertice>& vertices) {
20     if (vertices[destino].distancia == INT_MAX) {
21         cout << "N o h caminho de " << origem << " para " << destino << endl;
22     } else {
23         vector<int> caminho;
24         for (int v = destino; v != -1; v = vertices[v].antecessor) {
25             caminho.push_back(v);
26         }
27         reverse(caminho.begin(), caminho.end());
28
29         cout << "Caminho de " << origem << " para " << destino << ": ";
30         for (size_t i = 0; i < caminho.size(); i++) {
31             cout << caminho[i];
32             if (i < caminho.size() - 1) cout << " -> ";
33         }
34         cout << " | Dist ncia: " << vertices[destino].distancia << endl;
35     }
36 }
37
38 void dijkstra(int n, int origem, vector<vector<Aresta>>& grafo, vector<Vertice>&
vertices) {
39
40     vertices[origem].distancia = 0; // A dist ncia do v rtice de origem para ele
mesmo 0
41
42     for (int k = 0; k < n; k++) {
43         // Encontra o v rtice n o visitado com a menor dist ncia
44         int y1 = -1;
45         for (int i = 0; i < n; i++) {
46             if (!vertices[i].visitado && (y1 == -1 || vertices[i].distancia < vertices
[y1].distancia)) {
47                 y1 = i;
48             }
49         }
50
51         if (y1 == -1) {
```

```

52         break; // Se n o houver mais v rtices para visitar, termina
53     }
54
55     vertices[y1].visitado = true; // Marca o v rtice como visitado
56
57     // Atualiza as dist ncias dos vizinhos de y1
58     for (const Aresta& vizinho : grafo[y1]) {
59         if (!vertices[vizinho.destino].visitado) {
60             int novaDistancia = vertices[y1].distancia + vizinho.peso;
61             if (novaDistancia < vertices[vizinho.destino].distancia) {
62                 vertices[vizinho.destino].distancia = novaDistancia;
63                 vertices[vizinho.destino].antecessor = y1; // Atualiza o
64                 antecessor
65             }
66         }
67     }
68
69
70     // Mostra as dist ncias e os caminhos para todos os v rtices
71     for (int i = 0; i < n; i++) {
72         mostrarCaminho(i, origem, vertices);
73     }
74 }
75
76 int main() {
77     int n = 5; // N mero de v rtices
78
79     vector<Vertice> vertices(n); // Vetor que armazena informa es de cada v rtice
80
81     vector<vector<Aresta>> grafo(n); // o grafo um vetor de inteiros(vertices),
82     onde cada vertice possui um vetor de arestas
83
84     // Adiciona as arestas ao grafo (Exemplo de grafo)
85     grafo[0].push_back({1, 4});
86     grafo[0].push_back({2, 3});
87     grafo[1].push_back({2, 5});
88     grafo[1].push_back({3, 2});
89     grafo[2].push_back({4, 3});
90     grafo[2].push_back({3, 1});
91     grafo[3].push_back({4, 4});
92
93     int origem = 0; // V rtice inicial (origem)
94
95     dijkstra(n, origem, grafo, vertices);
96
97     return 0;
98 }

```

Listing 1: Exemplo C++

1.1 Como Funciona

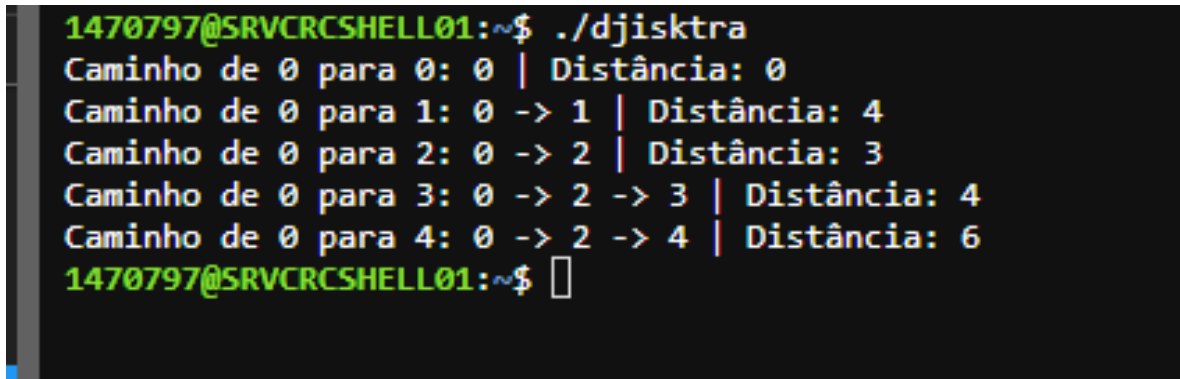
Função mostrarCaminho: mostra o caminho e a distância de origem até destino. A função começa verificando se existe um caminho até o destino, se houver um caminho válido, a função percorre os antecessores dos vértices, começando pelo destino, até chegar na origem, construindo um vetor chamado caminho que contém os vértices nessa ordem. Por fim, a função exibe o caminho completo da origem até o destino, separando cada vértice com -, e, em seguida, imprime a distância total do caminho.

Função dijkstra: define a distância do vértice de origem como 0, pois a distância dele para ele mesmo é zero. Na iteração principal, para cada vértice, busca o vértice não visitado que tenha a menor distância atual. Se não houver mais vértices para visitar (todos foram processados ou são inacessíveis), interrompe a execução. Cada iteração marcará o vértice com a menor distância como visitado e para cada vizinho desse vértice, calcula a nova distância potencial. Se essa nova distância for menor do que a distância armazenada

atualmente no vizinho, atualiza a distância e define o vértice antecessor. Após processar todos os vértices, a função chama a função mostrarCaminho para exibir os caminhos e as distâncias mínimas de todos os vértices a partir da origem.

Função main: Informa o número de vértices e cria um vetor para armazenar os vértices e arestas. A função main adiciona arestas ao grafo e chama a função de djikstra, passando como parâmetros o número de vértices, o vértice de origem, o vetor de inteiro vértices e o vetor que armazena as informações de cada vértice.

1.2 Teste do Código



```
1470797@SRVCRCHELL01:~$ ./djisktra
Caminho de 0 para 0: 0 | Distância: 0
Caminho de 0 para 1: 0 -> 1 | Distância: 4
Caminho de 0 para 2: 0 -> 2 | Distância: 3
Caminho de 0 para 3: 0 -> 2 -> 3 | Distância: 4
Caminho de 0 para 4: 0 -> 2 -> 4 | Distância: 6
1470797@SRVCRCHELL01:~$
```

Figure 1: Teste Dijkstra

2 Algoritmo MinMax em C++

```
1 #include <iostream>
2 #include <vector>
3 #include <climits> // Para usar INT_MAX como infinito
4 #include <algorithm> // Para usar reverse
5 #include <queue> // Para usar fila de prioridade
6
7 using namespace std;
8
9 struct Aresta {
10     int destino, peso;
11 };
12
13 struct Vertice {
14     int dist = INT_MAX; // Inicialmente a menor distância infinito
15     int antecessor = -1; // Antecessor inicial indefinido
16 };
17
18 void mostrarCaminho(int destino, int origem, const vector<Vertice>& vertices) {
19     if (vertices[destino].dist == INT_MAX) {
20         cout << "N o h caminho de " << origem << " para " << destino << endl;
21     } else {
22         vector<int> caminho;
23         for (int v = destino; v != -1; v = vertices[v].antecessor) {
24             caminho.push_back(v);
25         }
26         reverse(caminho.begin(), caminho.end());
27
28         cout << "Caminho de " << origem << " para " << destino << ": ";
29         for (size_t i = 0; i < caminho.size(); i++) {
30             cout << caminho[i];
31             if (i < caminho.size() - 1) cout << " -> ";
```

```

32     }
33     cout << " | Peso da maior aresta: " << vertices[destino].dist << endl;
34 }
35 }
36
37 void minMax(int n, int origem, vector<vector<Aresta>>& grafo, vector<Vertice>&
vertices) {
38     // Inicializa a distancia do vertice de origem
39     vertices[origem].dist = 0; // A distancia inicial é 0
40
41     // Usamos uma fila de prioridade para selecionar a aresta com o menor peso máximo
42     priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq
; // par (peso, vertice)
43     pq.push({0, origem}); // Começamos com a origem
44
45     while (!pq.empty()) {
46         auto [peso, u] = pq.top(); // Pega o menor peso máximo
47         pq.pop();
48
49         // Se a distancia do vertice atual é menor que a já registrada, ignoramos
50         if (vertices[u].dist < peso) continue;
51
52         // Atualiza as distancias acumuladas dos vizinhos de u
53         for (const Aresta& vizinho : grafo[u]) {
54             int novaDist = max(vertices[u].dist, vizinho.peso); // O peso máximo do
caminho
55             // Se encontramos um caminho com um peso máximo menor
56             if (novaDist < vertices[vizinho.destino].dist){
57                 vertices[vizinho.destino].dist = novaDist;
58                 vertices[vizinho.destino].antecessor = u; // Atualiza o antecessor
59                 pq.push({novaDist, vizinho.destino}); // Adiciona fila de
prioridade
60             }
61         }
62     }
63
64     // Mostra os caminhos para todos os vertices
65     for (int i = 0; i < n; i++) {
66         if (i == origem)
67             continue;
68         mostrarCaminho(i, origem, vertices);
69     }
70 }
71
72 int main() {
73     int n = 5; // Número de vertices
74
75     vector<Vertice> vertices(n); // Vetor que armazena informações de cada vertice
76
77     vector<vector<Aresta>> grafo(n); // o grafo é um vetor de inteiros (vertices),
onde cada vertice possui um vetor de arestas
78
79     // Adiciona as arestas ao grafo (Exemplo de grafo)
80     grafo[0].push_back({1, 4});
81     grafo[0].push_back({2, 3});
82     grafo[1].push_back({2, 5});
83     grafo[1].push_back({3, 2});
84     grafo[2].push_back({4, 3});
85     grafo[2].push_back({3, 1});
86     grafo[3].push_back({4, 4});
87
88     int origem = 0; // Vertice inicial (origem)
89
90     minMax(n, origem, grafo, vertices);
91
92     return 0;
93 }

```

Listing 2: Exemplo C++

2.1 Como Funciona

Função `mostrarCaminho`: mostra o caminho e a distância de origem até destino. A função começa verificando se existe um caminho até o destino, se houver um caminho válido, a função percorre os antecessores dos vértices, começando pelo destino, até chegar na origem, construindo um vetor chamado `caminho` que contém os vértices nessa ordem. Por fim, a função exibe o caminho completo da origem até o destino, separando cada vértice com `->`, e, em seguida, imprime a distância total do caminho.

A função `minMax` implementa um algoritmo para encontrar o caminho que minimiza o peso máximo das arestas em um grafo, ou seja, busca maximizar a largura de banda do caminho de origem até todos os outros vértices.

Função `minMax`: define a distância do vértice de origem como 0, pois a distância dele para ele mesmo é zero. Em seguida, utiliza-se uma fila de prioridade (`priority-queue`) configurada para retornar o menor peso máximo, iniciando com a origem e um peso de 0. Durante o processamento, enquanto a fila não estiver vazia, o algoritmo seleciona e remove o vértice com o menor peso máximo; se a distância registrada desse vértice for menor do que o peso, ele é ignorado. Para cada vizinho, calcula-se a nova distância como o máximo entre a distância acumulada atual e o peso da aresta, e, se essa nova distância for menor do que a registrada para o vizinho, a distância é atualizada e o antecessor é ajustado. O vizinho é então adicionado à fila para processamento posterior. Ao final, a função exibe os caminhos e as distâncias minimizadas máximas da origem para todos os outros vértices, utilizando `mostrarCaminho`. Em relação às variáveis, utilizamos `vertices[u].dist` para armazenar o peso máximo do caminho minimizado, `vertices[u].antecessor` para indicar o vértice anterior, `grafo[u]` que representa os vizinhos conectados e a fila de prioridade que assegura a minimização do peso máximo.

Função `main`: Informa o número de vértices e cria um vetor para armazenar os vértices e arestas. A função `main` adiciona arestas ao grafo e chama a função de `minMax`, passando como parâmetros o número de vértices, o vértice de origem, o vetor de inteiro vértices e o vetor que armazena as informações de cada vértice.

2.2 Teste do Código

```
1470797@SRVCRCHELL01:~$ g++ -o MinMax MinMax.cpp
1470797@SRVCRCHELL01:~$ ./MinMax
Caminho de 0 para 0: 0 | Distância total: 0
Caminho de 0 para 1: 0 -> 1 | Distância total: 4
Caminho de 0 para 2: 0 -> 2 | Distância total: 3
Caminho de 0 para 3: 0 -> 2 -> 3 | Distância total: 4
Caminho de 0 para 4: 0 -> 2 -> 4 | Distância total: 6
1470797@SRVCRCHELL01:~$
```

Figure 2: Teste MinMax

3 Algoritmo MaxMin em C++

```

1 #include <iostream>
2 #include <vector>
3 #include <climits> // Para usar INT_MAX como infinito
4 #include <algorithm> // Para usar reverse
5 #include <queue> // Para usar fila de prioridade
6
7 using namespace std;
8
9 struct Aresta {
10     int destino, peso;
11 };
12
13 struct Vertice {
14     int dist = INT_MIN; // Inicialmente a menor distancia menos infinito
15     int antecessor = -1; // Antecessor inicial indefinido
16 };
17
18 void mostrarCaminho(int destino, int origem, const vector<Vertice>& vertices) {
19     if (vertices[destino].dist == INT_MIN) {
20         cout << "N o h caminho de " << origem << " para " << destino << endl;
21     } else {
22         vector<int> caminho;
23         for (int v = destino; v != -1; v = vertices[v].antecessor) {
24             caminho.push_back(v);
25         }
26         reverse(caminho.begin(), caminho.end());
27
28         cout << "Caminho de " << origem << " para " << destino << ": ";
29         for (size_t i = 0; i < caminho.size(); i++) {
30             cout << caminho[i];
31             if (i < caminho.size() - 1) cout << " -> ";
32         }
33         cout << " | Peso da menor aresta: " << vertices[destino].dist << endl;
34     }
35 }
36
37 void maxMin(int n, int origem, vector<vector<Aresta>>& grafo, vector<Vertice>&
vertices) {
38     // Inicializa a distancia do vertice de origem
39     vertices[origem].dist = INT_MAX; // Mesmo inicialmente
40
41     // Usamos uma fila de prioridade para selecionar a aresta com o maior peso
42     priority_queue<pair<int, int>> pq; // par (peso, vertice)
43     pq.push({INT_MAX, origem}); // Começamos com a origem
44
45     while (!pq.empty()) {
46         auto [peso, u] = pq.top(); // Pega o maior peso
47         pq.pop();
48
49         // Se a distancia do vertice atual menor que a j registrada, ignoramos
50         if (vertices[u].dist > peso) continue;
51
52         // Atualiza as distancias acumuladas dos vizinhos de u
53         for (const Aresta& vizinho : grafo[u]) {
54             int novaDist = min(vertices[u].dist, vizinho.peso);
55             // Se encontramos um caminho com um peso maior
56             if (novaDist > vertices[vizinho.destino].dist) {
57                 vertices[vizinho.destino].dist = novaDist;
58                 vertices[vizinho.destino].antecessor = u; // Atualiza o antecessor
59                 pq.push({novaDist, vizinho.destino}); // Adiciona fila de
prioridade
60             }
61         }
62     }
63
64     // Mostra os caminhos para todos os vertices
65     for (int i = 0; i < n; i++) {
66         if (i == origem)
67             continue;
68         mostrarCaminho(i, origem, vertices);
69     }

```

```

70 }
71
72 int main() {
73     int n = 5; // Número de vértices
74
75     vector<Vertice> vertices(n); // Vetor que armazena informações de cada vértice
76
77     vector<vector<Aresta>> grafo(n); // o grafo é um vetor de vetores (vértices),
78     // onde cada vértice possui um vetor de arestas
79
80     // Adiciona as arestas ao grafo (Exemplo de grafo)
81     grafo[0].push_back({1, 4});
82     grafo[0].push_back({2, 3});
83     grafo[1].push_back({2, 5});
84     grafo[1].push_back({3, 2});
85     grafo[2].push_back({4, 3});
86     grafo[2].push_back({3, 1});
87     grafo[3].push_back({4, 4});
88
89     int origem = 0; // Vértice inicial (origem)
90
91     maxMin(n, origem, grafo, vertices);
92
93     return 0;
94 }

```

Listing 3: Exemplo C++

3.1 Como Funciona

Função mostrarCaminho: mostra o caminho e a distância de origem até destino. A função começa verificando se existe um caminho até o destino, se houver um caminho válido, a função percorre os antecessores dos vértices, começando pelo destino, até chegar na origem, construindo um vetor chamado caminho que contém os vértices nessa ordem. Por fim, a função exibe o caminho completo da origem até o destino, separando cada vértice com -, e, em seguida, imprime a distância total do caminho.

A função maxMin é uma variação de um algoritmo de caminhos mínimos que, em vez de buscar o caminho de menor custo, encontra o caminho que maximiza o menor peso ao longo das arestas do grafo. O objetivo é encontrar um caminho em que o menor peso de todas as arestas no caminho seja o maior possível.

Função maxMin: define a distância do vértice de origem como 0, pois a distância dele para ele mesmo é zero. Em seguida, utiliza-se uma fila de prioridade (priority-queue) para selecionar a aresta com o maior peso disponível, inicializando-a com a origem e o valor INT-MAX. Enquanto a fila não estiver vazia, o algoritmo processa o vértice com a maior distância acumulada, removendo-o da fila. Se a distância do vértice atual for menor que a registrada, ele é ignorado. Para cada vizinho do vértice atual, calcula-se uma nova distância acumulada como o menor valor entre a distância atual e o peso da aresta para o vizinho. Se esse valor for maior que a distância registrada do vizinho, ela é atualizada, e o antecessor é definido como o vértice atual. O vizinho é então adicionado à fila para processamento. Após analisar todos os vértices, a função exibe os caminhos e as distâncias máximas mínimas da origem para os outros vértices usando mostrarCaminho. Em relação às variáveis, utilizamos vertices[u].dist para armazenar a maior distância mínima, vertices[u].antecessor para indicar o vértice anterior no caminho, grafo[u] para listar os vizinhos, e priority-queue para garantir o processamento dos vértices que maximizam o menor peso possível.

Função main: Informa o número de vértices e cria um vetor para armazenar os vértices e arestas. A função main adiciona arestas ao grafo e chama a função de maxMin, passando como parâmetros o número de vértices, o vértice de origem, o vetor de inteiro vértices e o vetor que armazena as informações de cada vértice.

3.2 Teste do Código

```
1470797@SRVCRCHELL01:~$ g++ -o MaxMin MaxMin.cpp
1470797@SRVCRCHELL01:~$ ./MaxMin
Caminho de 0 para 1: 0 -> 1 | Peso da menor aresta: 4
Caminho de 0 para 2: 0 -> 1 -> 2 | Peso da menor aresta: 4
Caminho de 0 para 3: 0 -> 1 -> 3 | Peso da menor aresta: 2
Caminho de 0 para 4: 0 -> 1 -> 2 -> 4 | Peso da menor aresta: 3
1470797@SRVCRCHELL01:~$
```

Figure 3: Teste MaxMin