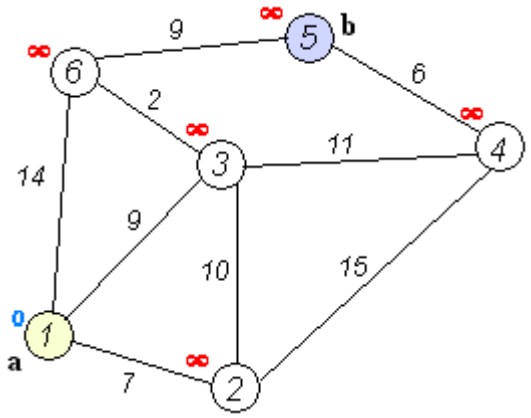
# Algoritmo de Dijkstra

Origem: Wikipédia, a enciclopédia livre.

O **algoritmo de Dijkstra**, concebido pelo cientista da computação holandês Edsger Dijkstra em 1956 e publicado em 1959[1] [2] , soluciona o problema do caminho mais curto num grafo dirigido ou não dirigido com arestas de peso não negativo, em tempo computacional O([m+n]log n) onde m é o número de arestas e n é o número de vértices. O algoritmo que serve para resolver o mesmo problema em um grafo com pesos negativos é o algoritmo de Bellman-Ford, que possui maior tempo de execução que o Dijkstra.

O **algoritmo de Dijkstra** assemelha-se ao BFS, mas é um algoritmo guloso, ou seja, toma a decisão que parece ótima no momento. Para a teoria dos grafos uma "estratégia gulosa" é conveniente já que sendo P um menor caminho entre 2 vértices U e V, todo sub-caminho de P é um menor caminho entre 2 vértices pertencentes ao caminho P, desta forma construímos os melhores caminhos dos vértices alcançáveis pelo vértice inicial determinando todos os melhores caminhos intermediários. Nota: diz-se 'um menor caminho' pois caso existam 2 'menores caminhos' apenas um será descoberto.

**Algoritmo de Dijkstra**




| | |
|---|---|
| classe | Algoritmo de busca |
| estrutura de dados | Grafo |
| complexidade pior caso | $O(\|E\| + \|V\| \log \|V\|)$ |

**Algoritmos**

O algoritmo considera um conjunto S de menores caminhos, iniciado com um vértice inicial I. A cada passo do algoritmo busca-se nas adjacências dos vértices pertencentes a S aquele vértice com menor distância relativa a I e adiciona-o a S e, então, repetindo os passos até que todos os vértices alcançáveis por I estejam em S. Arestas que ligam vértices já pertencentes a S são desconsideradas.

Um exemplo prático do problema que pode ser resolvido pelo algoritmo de Dijkstra é: alguém precisa se deslocar de uma cidade para outra. Para isso, ela dispõe de várias estradas, que passam por diversas cidades. Qual delas oferece uma trajetória de menor caminho?

## Índice

# Algoritmo de Dijkstra

- **1º passo:** iniciam-se os valores:

```
para todo v ∈ V[G]
     d[v]← ∞
     π[v] ← nulo
d[s] ← 0
```

V[G] é o conjunto de vértices(v) que formam o Grafo G. d[v] é o vetor de distâncias de s até cada v. Admitindo-se a pior estimativa possível, o caminho infinito. π[v] identifica o vértice de onde se origina uma conexão até v de maneira a formar um caminho mínimo.

- **2º passo:** temos que usar o conjunto *Q*, cujos vértices ainda não contém o custo do menor caminho d[v] determinado.

```
Q ← V[G]
```

- **3º passo:** realizamos uma série de **relaxamentos** das arestas, de acordo com o código:

```
enquanto Q ≠ ø
         u ← extrair-mín(Q)                    //Q ← Q - {u}
         para cada v adjacente a u
              se d[v] > d[u] + w(u, v)         //relaxe (u, v)
                 então d[v] ← d[u] + w(u, v)
                       π[v] ← u
                       Q ← Q ∪ {v}
```

w(u, v) é o peso(weight) da aresta que vai de u a v.

u e v são vértices quaisquer e s é o vértice inicial.

extrair-mín(Q), pode usar um heap de mínimo ou uma lista de vértices onde se extrai o elemento u com menor valor d[u].

No final do algoritmo teremos o menor caminho entre *s* e qualquer outro vértice de G. O algoritmo leva tempo O(m + n log n) caso seja usado um heap de Fibonacci, O(m log n) caso seja usado um heap binário e $O(n^2)$ caso seja usado um vetor para armazenar Q.

# Implementações

## Implementação em C

A implementação a seguir utiliza uma matriz de adjacência para representar o grafo e tem complexidade de tempo $O(E+V^2)$. Dependendo da quantidade limite de vértices (MAXV) pode se tornar ineficiente quanto ao uso de memória, sendo recomendado usar uma lista de adjacência, que pode ser facilmente implementada como um array de listas encadeadas.

```
// Implementação do algoritmo de Dijkstra em C utilizando uma matriz de adjacência
```

```
#include <string.h> //memset

// MAXV é uma constante que define a quantidade máxima de vértices
#define MAXV 100

// Matriz de adjacências
// Se G[i][j] > 0, então há aresta que liga 'i' a 'j' com custo G[i][j].
int G[MAXV][MAXV];
int V; // quantidade de vértices no grafo G

// Armazena a distância mínima partindo de um vértice 'Vi' até todos os outros vértices
// dis[i] representa a menor distância de 'Vi' a 'i'.
int dis[MAXV];

void dijkstra (int Vi) // calcula dis[]
{
        char vis[MAXV];
        memset (vis, 0, sizeof (vis));
        memset (dis, 0x7f, sizeof (dis));
        dis[Vi] = 0;

        int t, i;
        for (t = 0; t < V; t++)
        {
                int v = -1;
                for (i = 0; i < V; i++)
                        if (!vis[i] && (v < 0 || dis[i] < dis[v]))
                                v = i;
                vis[v] = 1;

                for (i = 0; i < V; i++)
                        if (G[v][i] && dis[i] > dis[v] + G[v][i])
                                dis[i] = dis[v] + G[v][i];
        }
}
```

## Implementação em C++

A implementação a seguir guarda o caminho utilizado para se ir de um vértice inicial a cada um dos outros vértices e considera que o grafo é representado por uma lista de adjacência.

```
// Implementação do Algoritmo de Dijkstra O(E+V²) em C++
// Caso não tenha conhecimento do funcionamento do algoritmo, os comentários podem auxiliar.
// Se estiver apenas em busca de uma implementação, use http://codepad.org/AmMuyTl3

#include <vector>
using namespace std;

// MAXV é uma constante que define a quantidade máxima de vértices no grafo
#define MAXV 100

// Lista de adjacências
// Para inserir uma aresta partindo do vértice 'a' ao vértice 'b', com custo/peso 'c', podemos usar:
// G[a].push_back(make_pair(b,c));
vector< pair<int,int> > G[MAXV];
int V; // quantidade de vértices no grafo G

// Armazenará a distância mínima de um vértice Vi até todos os outros vértices após execução do Dijkstra
// dis[i] representa a distância mínima para se ir de Vi a i.
int dis[MAXV];

// (Extra/opcional) Armazenará o caminho percorrido para se ir de um vértice a outro
// Para se chegar no vértice i, foi usado o vértice path[i]; para se chegar no vértice path[i], usou-se pa
// Usamos path[Vi] = -1, para indicar o fim do caminho, sendo Vi o vértice inicial
int path[MAXV];

// Algoritmo de dijkstra: calcula a menor distância para se ir de Vi a qualquer outro vértice do grafo G
// Armazenará as distâncias mínimas em dis[], e os caminhos em path[]
void dijkstra(int Vi)
{
        // vis[i] = true apenas se o vértice i já foi visitado (analisado) pelo algoritmo
        bool vis[MAXV];
```

```
        for (int i = 0; i < V; i++)
                vis[i] = false;

        // Inicialmente, consideramos que a distância de Vi para qualquer vértice é infinita, exceto de Vi
        // Use algum valor alto o suficiente para poder ser considerado infinito; se não existe tal valor
        // use algum número mágico e trate esse caso no algoritmo, quando comparar distâncias (por exemplo
        #define inf 1000000000
        for (int i = 0; i < V; i++)
                dis[i] = inf;
        dis[Vi] = 0;

        path[Vi] = -1;

        while (true)
        {
                int v = -1; // armazenará o vértice que será analisado/visitado nesta etapa

                // Seleciona o vértice v não visitado (analisado) cuja distância Vi->v seja mínima; por es
                // distância dentre todos os outros, garante-se que não existe nenhum caminho que nos perm
                // distância menor (a não ser que hajam arestas negativas, caso em que se recomenda o algo
                // Portanto, garante-se que a distância para este vértice já está completamente atualizada
                for (int i = 0; i < V; i++)
                        if (!vis[i] && (v < 0 || dis[i] < dis[v]))
                                v = i;

                // Se não selecionamos nenhum vértice ou a distância para o vértice selecionado for infini
                // Nesta etapa, a distância para v ser infinita significa que não há caminho de Vi->v.
                if (v < 0 || dis[v] == inf) break;
                // Visitamos o vértice v, para que não seja selecionado novamente.
                vis[v] = true;

                // Tentamos usar o vértice v como parte do caminho de Vi para os outros vértices.
                // Para um vértice u, apenas faremos isso caso a dis[u] > dis[v] + pesoDaAresta[v][u].
                // Percorremos a lista de adjacências (arestas) de v, verificando essa condição para cada
                for (int i = 0; i < G[v].size(); i++)
                {
                        int u = G[v][i].first; // vértice ligado a v (aresta v->u)
                        int w = G[v][i].second; // peso da aresta entre v e u
                        if (dis[u] > dis[v] + w)
                        {
                                // Atualizamos a distância de Vi para u, considerando que passamos pelo vé
                                dis[u] = dis[v] + w;

                                // (Extra/opcional) Marcamos para o nosso caminho que chegamos a u usando
                                path[u] = v;
                        }
                }
        }
}

// (Extra/opcional) Obtém o percurso usado para se chegar de Vi até v (necessário executar dijkstra(Vi) an
// caminho.front() = Vi e caminho.back() = v
void obterCaminho(int v, vector<int>& caminho)
{
        if (v == -1) return;
        obterCaminho(path[v], caminho);
        caminho.push_back(v);
}
vector<int> obterCaminho(int v)
{
        vector<int> caminho;
        obterCaminho(v, caminho);
        return caminho;
}
```

### Análise do algoritmo

Notamos que o algoritmo tem complexidade de tempo $O(V^2+E)$, sendo V a quantidade de vértices e E a quantidade de arestas do grafo. Fazendo uma análise detalhada:

```
for (int i = 0; i < V; i++)
        if (!vis[i] && (v < 0 || dis[i] < dis[v]))
```

```
                v = i;
```

Percorremos os V vértices em busca de um vértice não visitado cuja distância é a menor dentre todos os vértices ainda não analisados. O custo é O(V). Como este algoritmo é repetido até V vezes (em caso de o grafo estar totalmente conectado, deveremos selecionar um vértice [ainda não analisado] V vezes), temos uma complexidade total de $O(V^2)$.

```
for (int i = 0; i < G[v].size(); i++)
{
        int u = G[v][i].first;
        int w = G[v][i].second;
        if (dis[u] > dis[v] + w)
        {
                dis[u] = dis[v] + w;
                path[u] = v;
        }
}
```

Percorremos as arestas de um vértice v, tendo um custo de O(G[v].size()) = O(quantidadeDeArestas(v)). No caso de o grafo estar totalmente conectado, executamos este algoritmo para cada vértice, tendo como complexidade a soma da quantidade de arestas de cada vértice, resultando em O(E).

As inicializações de variáveis (dis[] e vis[]) têm um custo O(V), sendo desprezíveis em meio à complexidade final $O(V^2+E)$.

Na primeira parte do algoritmo, nossa tarefa é encontrar o vértice não visitado cuja distância seja a menor dentre todos os outros vértices não visitados. Se utilizarmos uma estrutura de dados que nos forneça essa informação em O(log N), podemos reduzir o custo total dessa parte para O(V log V). A segunda parte do algoritmo deverá inserir nessa estrutura os vértices cujas distâncias foram atualizadas (alteradas); para isso, o ideal é utilizar uma estrutura com inserção constante, O(1), mantendo a complexidade em O(E). Uma estrutura com inserção em O(1) e remoção (e leitura) em O(log N) é o Heap de Fibonacci, sendo usado para implementar o algoritmo de Dijkstra em O(E + V log V).

Quando podemos garantir que, no nosso grafo, a quantidade de arestas se aproxima muito da quantidade de vértices, como em uma árvore, O(E + V log V) se aproxima de O(V log V). Nesse caso, uma estrutura com inserção em O(log N) resultaria em uma complexidade O(E log V + V log V) = O((V+E) log V), próxima de O(V log V). Heaps simples são bastante utilizados para esse tipo de situação, por serem fáceis de implementar. Além disso, várias linguagens trazem implementações prontas de heap. O código a seguir usa a priority queue da STL do C++ (uma implementação de max-heap) para implementar o algoritmo de Dijkstra em O((V+E) log E). Já que a quantidade de arestas de um grafo completo é $V*(V-1)/2 \sim= V^2$, então $\log(E) \sim= \log(V^2) = 2*\log(V) = O(\log V)$. Desse modo, O((V+E) log E) = O((V+E) log V).

### Implementação utilizando um heap

```
// Implementação do Algoritmo de Dijkstra O((V+E) log V) em C++
// Se estiver apenas em busca de uma implementação, use http://codepad.org/NgxirXBz

#include <vector>
#include <queue>//priority_queue

using namespace std;

#define MAXV 100

vector< pair<int,int> > G[MAXV];
```

```
int V;

int dis[MAXV];
int path[MAXV];
void dijkstra(int Vi)
{
        // Heap que armazenará os nós que tiveram suas distâncias atualizadas. Como queremos ordenar pela
        // distância, utilizamos um par <-dis[v], v>. Usamos -dis[v], em vez de dis[v], porque a priority_
        // ordena do maior para o menor (max-heap), e nós queremos o contrário (vértice com a menor distâ
        priority_queue< pair<int,int> > pq;

        bool vis[MAXV];
        for (int i = 0; i < V; i++)
                vis[i] = false;

        #define inf 1000000000
        for (int i = 0; i < V; i++)
                dis[i] = inf;
        dis[Vi] = 0;

        // Inicialmente, o único vértice cuja distância já foi atualizada é o Vi (de infinito para 0).
        pq.push(make_pair(0, Vi));

        path[Vi] = -1;

        while (true)
        {
                int v = -1;

                // Selecionamos um vértice v não visitado cuja distância Vi->v é a menor dentre todos os v
                // Como um mesmo vértice pode ter tido sua distância atualizada mais de uma vez, é possíve
                // vértices já visitados ainda estejam no heap, portanto devemos procurar até que seja enc
                // um não visitado ou o heap fique vazio.
                while (!pq.empty() && (v < 0 || vis[v]))
                        v = pq.top().second, pq.pop();

                if (v < 0 || dis[v] == inf) break;
                vis[v] = true;

                for (int i = 0; i < G[v].size(); i++)
                {
                        int u = G[v][i].first;
                        int w = G[v][i].second;
                        if (dis[u] > dis[v] + w)
                        {
                                dis[u] = dis[v] + w;
                                path[u] = v;
                                // Inserimos a "atualização" no heap; como priority_queue é um max-heap, e
                                // min-heap, usamos -dis[u], em vez de dis[u].
                                pq.push(make_pair(-dis[u], u));
                        }
                }
        }
}
```

## Implementação em Python

A implementação representa o grafo como uma lista de adjacência, e retorna uma lista dis[] das distâncias, sendo dis[i] a distância do vértice inicial Vi para um vértice i do grafo. Por utilizar um heap, tem complexidade de tempo O((E+V) log V).

```
from heapq import heappop, heappush

MAXV = 1000 # numero de vertices no grafo
graph = [[] for x in xrange(MAXV)]

def add_edge(v, u, w):
    graph[v].append((u,w))

def dijkstra(graph, Vi):
    dis = [-1] * len(graph)
    vis = [False] * len(graph)
```

```
    heap = [(0,Vi)]
    dis[Vi] = 0

    while True:
        v = -1
        while len(heap) > 0 and (v < 0 or vis[v]):
            v = heappop(heap)[1]

        if v < 0 or dis[v] < 0:
            break
        vis[v] = True

        for (u, w) in graph[v]:
            if dis[u] < 0 or dis[u] > dis[v] + w:
                dis[u] = dis[v] + w
                heappush(heap, (dis[u],u))

    return dis
```

# Problemas relacionados

O algoritmo de Dijkstra não consegue encontrar o menor caminho em um grafo com pesos negativos. Para esse propósito, pode-se usar o algoritmo de Floyd-Warshall, que consegue descobrir a menor distância entre todos os pares de vértices de qualquer grafo sem ciclos com peso negativo em uma complexidade de tempo $O(V^3)$. Se o problema não exigir o cálculo da distância entre todos os pares de vértices ou se existirem ciclos com peso negativo, pode-se aplicar o algoritmo de Bellman-Ford, com complexidade de tempo O(V*E). Em uma árvore, é possível encontrar a distância entre um vértice inicial e todos os outros vértices em tempo O(V+E), utilizando busca em profundidade (também conhecida como DFS). Em um grafo cujas arestas têm todas o mesmo peso, pode-se encontrar a distância entre um vértice inicial e todos os outros vértices, para um grafo qualquer, em O(V+E), utilizando busca em largura (também conhecida como BFS). O processo utilizado no algoritmo de Dijkstra é bastante similar ao processo usado no algoritmo de Prim. O propósito deste último, entretanto, é encontrar a árvore geradora mínima que conecta todos os nós de um grafo.

# Ver também

- Algoritmo de Floyd-Warshall
- Algoritmo de Bellman-Ford
- Algoritmo A*
- Busca em profundidade
- Busca em largura

# Ligações externas

- Artigo explicativo sobre o algoritmo (http://www.inf.ufsc.br/grafos/temas/custo-minimo/dijkstra.html)
- Artigo e Implementação do Algoritmo de Dijkstra em C (http://www.vivaolinux.com.br/script/Algoritmo-de-Dijkstra)
- (em inglês) Algoritmo de Dijkstra em C# (http://www.codeproject.com/useritems/Shortest_Path_Problem.asp)
- (em inglês) Applet do algoritmo de Dijkstra (http://www-b2.is.tokushima-u.ac.jp/~ikeda/suuri/dijkstra/Dijkstra.shtml)
- (em inglês) Simulação do algoritmo de Dijkstra (http://optlab-server.sce.carleton.ca/POAnimations2007/DijkstrasAlgo.html)

# Referências

1. ↑ DIJKSTRA, Edsger; Thomas J. Misa, Editor. (2010-08). "An Interview with Edsger W. Dijkstra". *Communications of the ACM* **53** (8): 41–47. DOI:10.1145/1787234.1787249 (http://dx.doi.org/10.1145/1787234.1787249).
2. ↑ Dijkstra 1959

Obtida de "http://pt.wikipedia.org/w/index.php?title=Algoritmo_de_Dijkstra&oldid=36754847"
Categoria: Algoritmos de grafos

---