Aula 6 · Introdução e busca em Grafos e Caminho Mínimo

Desafios de Programação

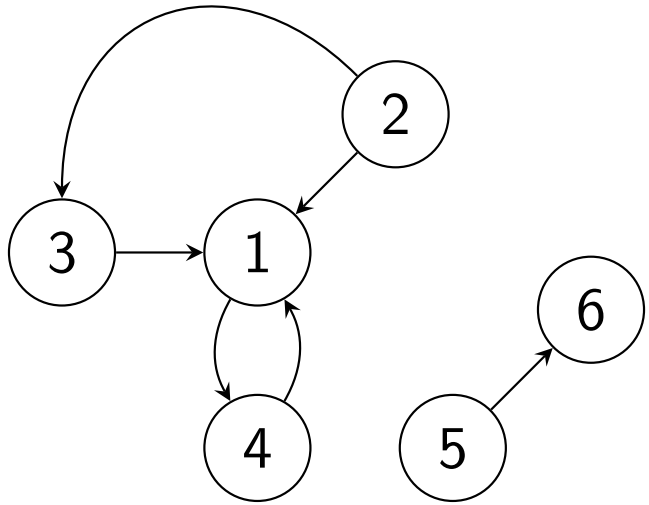
Fernando Kiotheka Victor Alflen

UFPR

13/07/2022

Introdução a Grafos

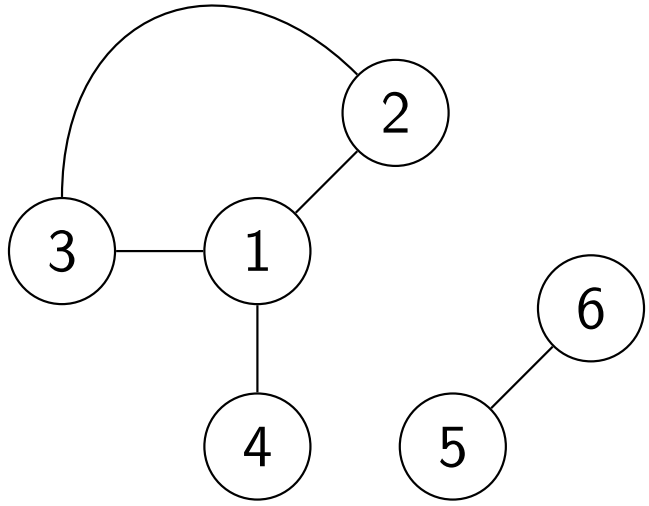
Grafos direcionados



$$G = (V, E)$$
$$V = \{1, 2, 3, 4, 5, 6\}$$
$$E = \{(3, 1), (2, 1), (2, 3), (1, 4), (4, 1), (5, 6)\}$$

Um grafo direcionado é definido como um par ordenado (V, E), onde V é um conjunto de vértices e E um conjunto de arestas (ou arcos) de pares ordenados (u, v) onde $u, v \in V$.

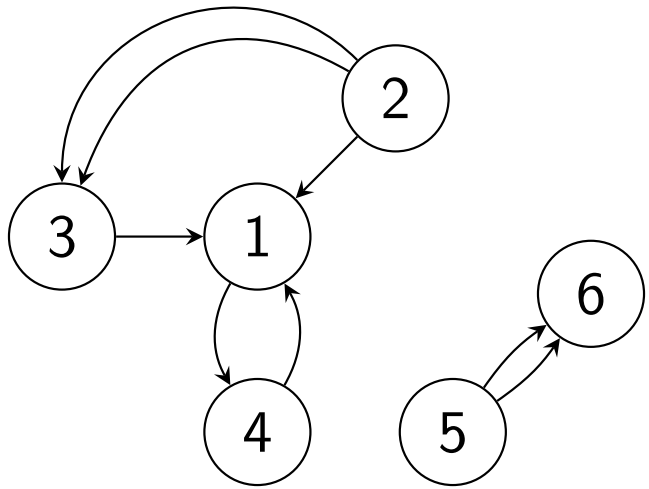
Grafos não direcionados



$$G = (V, E)$$
$$V = \{1, 2, 3, 4, 5, 6\}$$
$$E = \{\{3, 1\}, \{2, 1\}, \{2, 3\}, \{1, 4\}, \{5, 6\}\}$$

Um grafo não direcionado (que é o mais "simples") é a mesma coisa, mas a direção das arestas não importa, então o conjunto de arestas contém conjuntos de dois elementos $\{u, v\}$ onde $u, v \in V$. Podemos definí-los como grafos direcionados onde para cada aresta (u, v), existe uma aresta (v, u) correspondente.

Multigrafos



$$G = (V, E)$$
$$V = \{1, 2, 3, 4, 5, 6\}$$
$$E = \{(3, 1), (2, 1), (2, 3), (2, 3), (1, 4), (4, 1), (5, 6), (5, 6)\}$$

Pode ser o caso onde E é um conjunto com elementos repetidos (multiconjunto), esse grafo é caracterizado como multigrafo. Multigrafos podem ser direcionados ou não direcionados.

Mas por que classificar grafos?

Saber a natureza de um grafo é o primeiro passo para a resolução de um problema com grafos. Podemos com isso, por exemplo, pensar casos de borda possíveis:

- Podemos ir e voltar?
- Quais algoritmos são adequados?
- Ciclos?
- Arestas repetidas na entrada?

Veremos mais classificações de grafos a seguir, mas primeiro, vamos olhar para jeitos de se representar grafos no nosso programa.

Representações de grafos

Há pelo menos três jeitos de se representar grafos:

- Lista de adjacência. Cada vértice u contém uma lista de seus vizinhos. Exemplo: `vector<vector<edge>> adj (N)`.
- Matriz de adjacência. Cada célula da matriz denota a informação sobre a aresta (u, v). Exemplo: `vector<vector<edge>> mat (N, vector<edge>(N))`.
- Vetor de arestas. Cada entrada do vetor representa uma aresta. Exemplo: `vector<edge> edges (M)`. Pode ser bem útil por exemplo para conseguir a aresta oposta rápido: `edges[e]` e `edges[e^1]` (assumindo inserção nesta ordem).

Cada jeito de representar será mais benéfico ao uso de certos algoritmos. Pode ser útil também ficar convertendo entre representações diferentes.

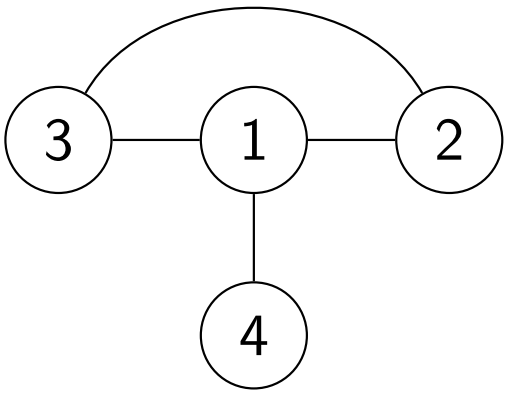
Informações associadas aos vértices, ou as arestas

- Colocamos de um jeito genérico o `edge` porque as informações associadas a arestas variam.
- Exemplos comuns incluem: O peso da aresta, o custo da aresta, a capacidade de fluxo da aresta, a probabilidade da aresta, etc.
- Geralmente podemos usar um `pair<int, T>` numa lista de adjacências, ou criar uma estrutura de dados na lista de arestas com todos os atributos necessários.
- Frequentemente o grafo é criado implicitamente no problema. Em um espaço 2D, podemos olhar para os quadrados vizinhos, e isso forma implicitamente uma aresta entre cada quadrado e seus vizinhos, porém não é necessário antes de resolver o problema criar essas arestas explicitamente.
- Podem existir informações associadas aos próprios vértices. Saber se uma informação é referente a uma aresta ou a um vértice é de suma importância.

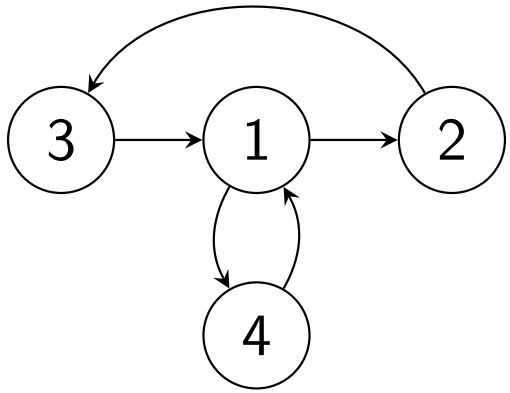
Famílias de grafos

- Já classificamos os grafos de acordo com as suas arestas.
- Agora veremos outra classificação útil: Famílias de grafos.
- Não veremos todas, existem muitas e muitas, por exemplo:
  - Grafos *split*
  - Grafos planares
  - etc.
- Mas veremos os mais importantes *por enquanto*.

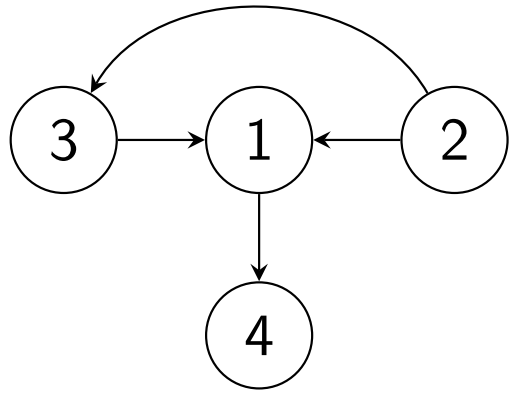
Grafos conectados



(a)



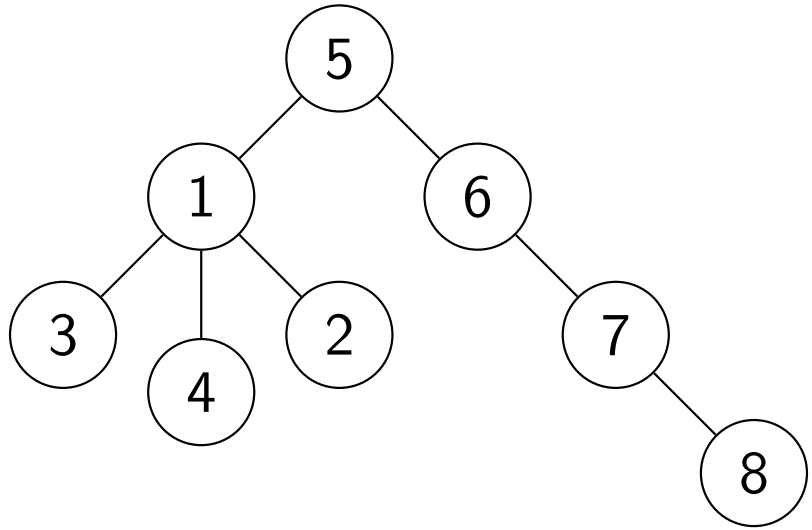
(b)



(c)

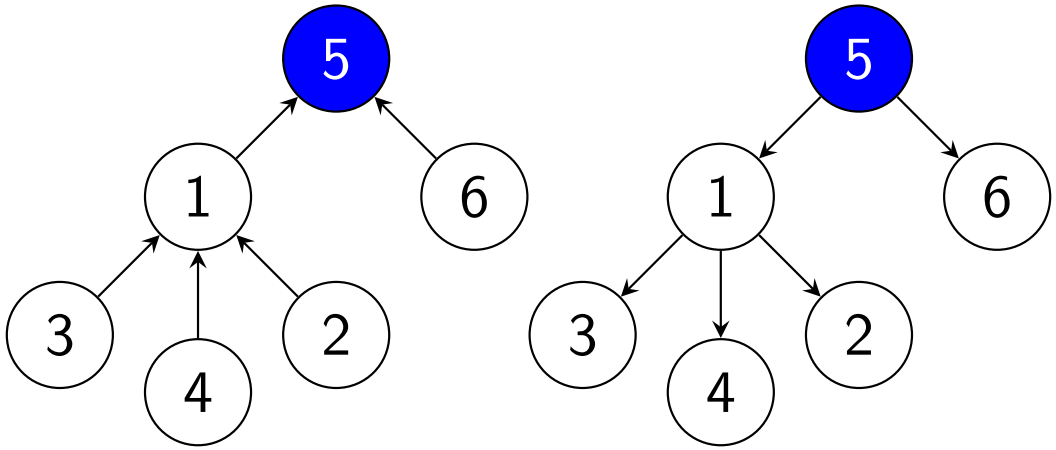
- Um grafo não direcionado é **conectado** (a) se todos os vértices são conectados por um caminho.
- Um grafo direcionado é **fortemente conectado** (b) se todos os vértices são conectados por um caminho.
- Um grafo direcionado é **fracamente conectado** (c) se todos os vértices são conectados ignorando a direção das arestas.

Árvores



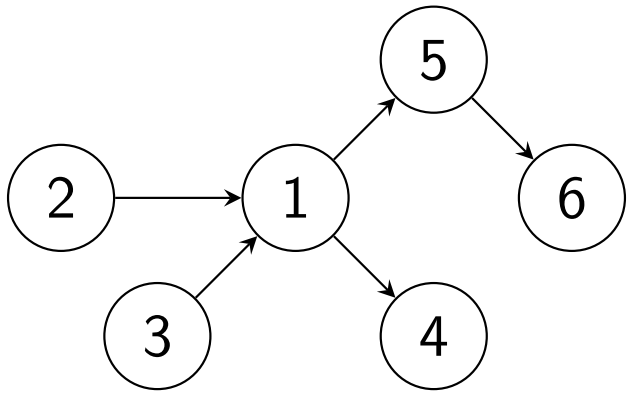
- Árvores são grafos não direcionados onde todos os vértices tem apenas **um** caminho para qualquer outro vértice.
- Também são chamados de grafos não direcionados acíclicos (sem ciclos).
- Quando um grafo representa várias árvores desconexas, ele é chamado de floresta.

Arborescências ou árvores enraizadas



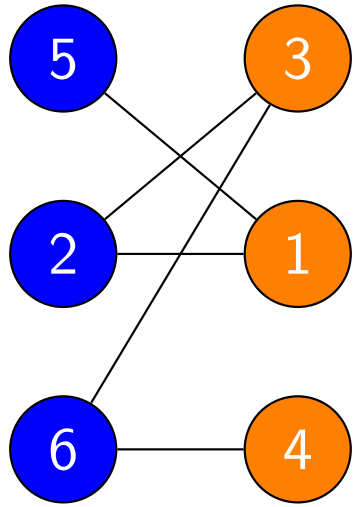
- Designamos um vértice da árvore como raiz (todos servem).
- Criamos um grafo direcionado a partir disso, sendo que as arestas ficam na mesma direção — arborescência (a) — ou em direção oposta a raiz — anti-arborescência (b).
- É essencial para a resolução de muitos problemas (e é parecido por exemplo, com o padrão de busca em profundidade que veremos a seguir).
- Toda arborescência é um DAG (grafo acíclico direcionado).

Acíclicos direcionados (DAGs)



- Todos os grafos direcionados sem ciclos.
- Modelagem de dependências (dependência cíclica é problema).
- Podem ser topologicamente ordenados (isto é, existe uma ordem que permite que todas as dependências sejam resolvidas antes da própria tarefa).
- Todo grafo direcionado com ciclos pode ser reduzido a uma versão "compactada" sem ciclos, veremos esse processo na próxima vez que revisitarmos grafos.

Bipartidos



- Geralmente é um grafo não direcionado.
- Vértices de um lado se ligam apenas a vértices do outro lado.
- Toda árvore é um grafo bipartido, o inverso não é verdade.
- Um grafo é bipartido se e somente se não tem ciclo ímpar.
- Associado ao problema do emparelhamento máximo.
- Existem variações: Grafos k-partidos.

Percorrendo Grafos

Como busco um vértice?

A busca em grafos é iniciado por um vértice inicial. O vértice inicial adequado depende da operação que se vai fazer, sendo que muitas vezes qualquer vértice do grafo serve.

Em profundidade (*dfs*, *depth-first search*) Cada nó é explorado o mais fundo possível antes de voltar (backtracking). Controle feito por uma pilha.

Em largura (*bfs*, *breadth-first search*) "Camada por camada", visitando todos os vértices que estão a distância 1, então 2, e assim por diante do vértice inicial. Controle feito por uma fila.

Ambos os algoritmos são $O(V + E)$, pois consideramos todos os vértices e todas as arestas (mesmo se não as visitamos).

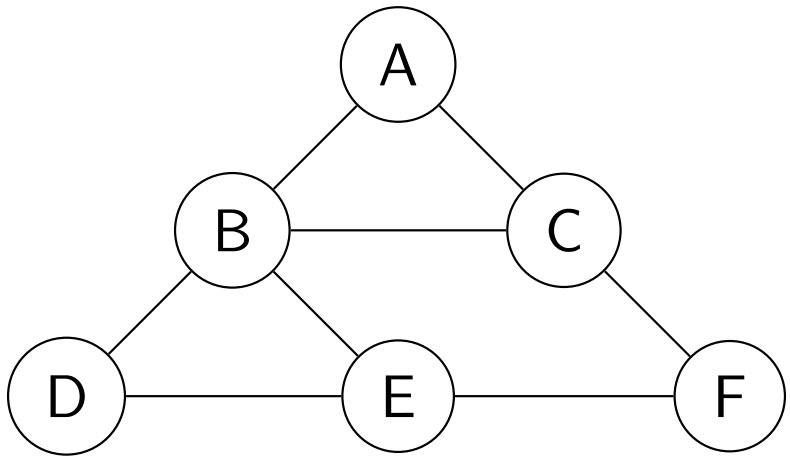
Problema da contagem de vértices

Vamos formular um problema simples para aplicar estas buscas.

- Temos um grafo de entrada não direcionado e várias consultas.
- A consulta quer saber quantos vértices são *alcançáveis* de um determinado vértice u (existe um caminho de u para eles).

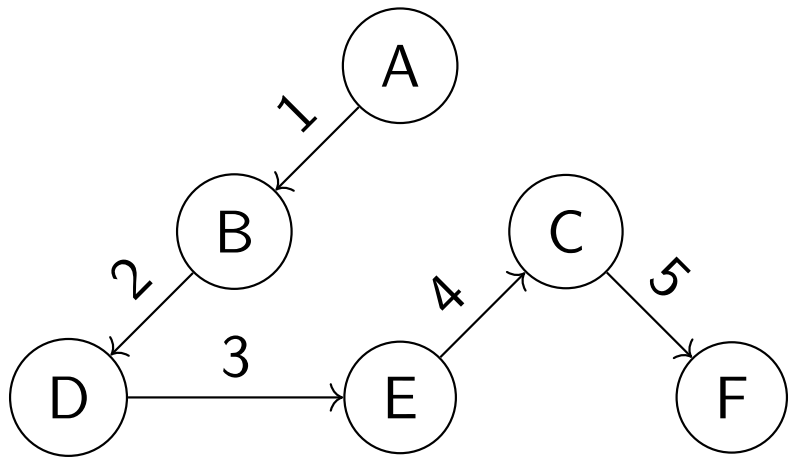
Como resolver?

Exemplo em um grafo



Note que a ordem de caminhamento no grafo é **arbitrária**. Mostraremos um exemplo, mas dependendo da ordem das arestas, a ordem da visita pode ser completamente diferente. Vamos supor que queremos contar a partir do vértice A.

DFS



Busca em profundidade recursiva

Utilizamos a pilha implícita da chamadas recursivas.

Código `dfsrec.h`

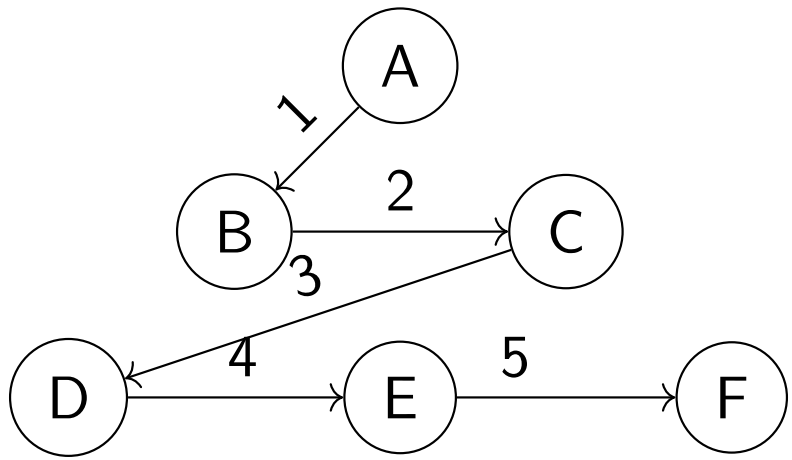
```
int explore(int u) {
    visited[u] = true;
    int c = 1;
    for (int v : g[u]) if (!visited[v])
        c += explore(v);
    return c;
}
```

Busca em profundidade com pilha

Código `dfsstack.h`

```
int explore(int u) {
    int c = 0;
    stack<int> s;
    s.push(u);
    while (!s.empty()) {
        int u = s.top(); s.pop();
        for (int v : g[u]) if (!visited[v]) {
            c++;
            visited[v] = true;
            s.push(v);
        }
    }
    return c;
}
```

BFS



Busca em largura

Código `bfs.h`

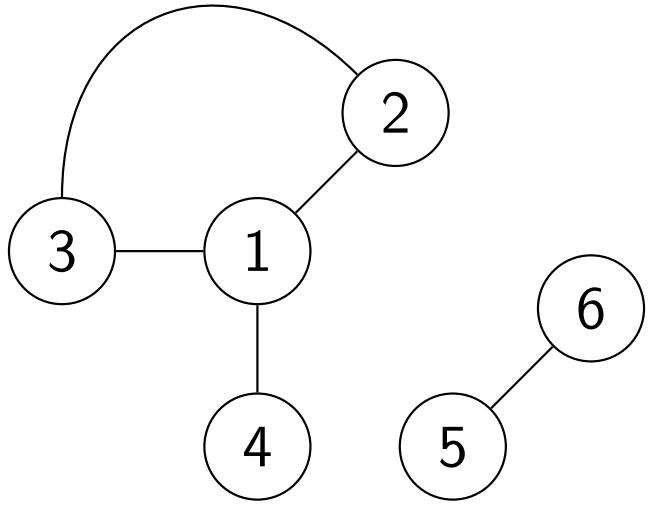
```
int explore(int u) {
    int c = 0;
    queue<int> q;
    q.push(u);
    while (!q.empty()) {
        int u = q.front(); q.pop();
        for (int v : g[u]) if (!visited[v]) {
            c++;
            visited[v] = true;
            q.push(v);
        }
    }
    return c;
}
```

Resolvendo o problema da contagem de vértices

Código `count.cpp`

```
#include <bits/stdc++.h>
using namespace std;
vector<vector<int>> g (1e5+15);
vector<bool> visited (1e5+15);
#include "bfs.h"
int main() {
    int n, m; cin >> n >> m;
    while (m--) {
        int u, v; cin >> u >> v;
        g[u].push_back(v);
        g[v].push_back(u);
    }
    int q; cin >> q; while (q--) {
        int u; cin >> u;
        fill(visited.begin(), visited.end(), false);
        cout << explore(u) << "\n";
    }
}
```

Exemplo de contagem de vértices



Entrada	Saída
6 4 3 1 1 4 2 3 5 6 6 1 2 3 4 5 6	4 4 4 4 2 2

Outros algoritmos de busca em grafos

Na área da Inteligência Artificial existem algumas outras buscas relevantes como

- Best-first search (pega o melhor caminho sempre)
- A*
- IDS
- IDA*

Para alguns problemas esses outros tipos de busca são necessários (geralmente envolvendo resolver jogos), mas não vamos entrar nesses detalhes.

Busca de caminho mínimo

E o caminho mínimo?

Dados dois vértices v e u, qual o caminho mínimo (menor soma de pesos em arestas) que começa em v e termina em u?

- Com DFS ou BFS: Testar todas as possibilidades de caminho
- Dijkstra: Expandir os caminhos mais curtos até achar o primeiro que chega ao destino
- Bellman-Ford: *Relaxar* (termo técnico) os caminhos o máximo possível
- Floyd-Warshall: Programação dinâmica

SSSP e APSP

A partir de um grafo G, definimos os seguintes problemas:

- **Caminhos mínimos de uma origem (SSSP):** Menor distância entre um vértice $u \in V$ e todos os outros vértices $v \in G$.
- **Caminhos mínimos de todos os pares (APSP):** Menor distância entre cada par de vértices (u, v) onde $u, v \in G$.

Resolvemos o primeiro com Dijkstra e Bellman-Ford, e o segundo com Floyd-Warshall (mas os três algoritmos podem ser aplicados aos dois problemas).

Dijkstra

- É um algoritmo de programação dinâmica que utiliza de uma fila de prioridade.
- Permite obter o caminho mínimo de uma origem em $O((V + E) \lg V)$ (com heap).
- *Uniform-cost search* é o Dijkstra conhecido no campo da Inteligência Artificial onde a gente não explora todos os caminhos, para quando acha um destino.
- Não aceita pesos negativos (importante!)
- Nesta implementação, não vamos tirar as coisas já visitadas da fila de prioridade, isso não é problema em questões assintóticas.

Implementação do Dijkstra

Código `dijkstra.h`

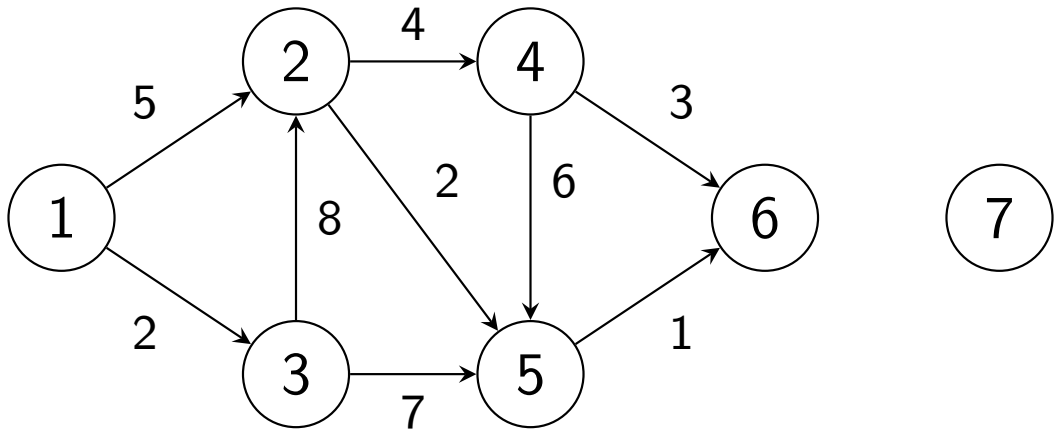
```
vector<int> d (N, oo), vis (N);
void dijkstra(int src) {
    priority_queue<pair<ll, int>,
        vector<pair<ll, int>>, greater<pair<ll, int>>> Q;
    d[src] = 0;
    Q.push({0, src});
    while (!Q.empty()) {
        auto [c, u] = Q.top(); Q.pop();
        if (vis[u]) { continue; }
        vis[u] = true;
        for (auto [v, w] : g[u])
            if (d[v] > d[u] + w) {
                d[v] = d[u] + w;
                Q.push({d[v], v});
            }
    }
}
```

Implementação do Dijkstra (continuado)

Código dijkstra.cpp

```
#include <bits/stdc++.h>
using namespace std; using ll = long long;
const int N = 1e5+15; const int oo = 987654321;
using edge = pair<int, int>;
vector<vector<edge>> g (N);
#include "dijkstra.h"
int main() {
    int n, m; cin >> n >> m;
    while (m--) {
        int u, v, w; cin >> u >> v >> w; u--; v--;
        g[u].push_back(edge(v, w));
    }
    dijkstra(0);
    for (int u = 0; u < n; u++)
        cout << d[u] << "\n";
}
```

Exemplo do Dijkstra



Entrada	Saída
7 9 1 2 5 1 3 2 3 2 8 2 5 2 3 5 7 2 4 4 4 5 6 4 6 3 5 6 1	0 5 2 9 7 8 987654321

Floyd-Warshall

- **Ideia**: Programação dinâmica para resolver o caminho mínimo em grafo direcionado para todos os pares de vértices.
- Para cada vértice intermediário m, verificamos se o caminho formado pelos vértices (u, v) seria menor se passasse pelo vértice m.
- Roda em $\mathcal{O}(n^3)$
- O caminho entre vértices (u, v) que não estão diretamente ligados é infinito.
- O caminho formado pelo mesmo vértice, isto é, (u, u) é zero.
- Cuidado com "infinito", pois é comum somar dois "infinitos".
- Não funciona caso existam ciclos de pesos negativos.
- É possível reconstruir o menor caminho facilmente, fazendo a recuperação da programação dinâmica.

Bellman-Ford

- Utiliza o conceito de relaxar os pesos.
- Pior caso: $\mathcal{O}(VE)$
- Aceita pesos negativos.

Implementação do Bellman-Ford

Código bellman-ford.cpp

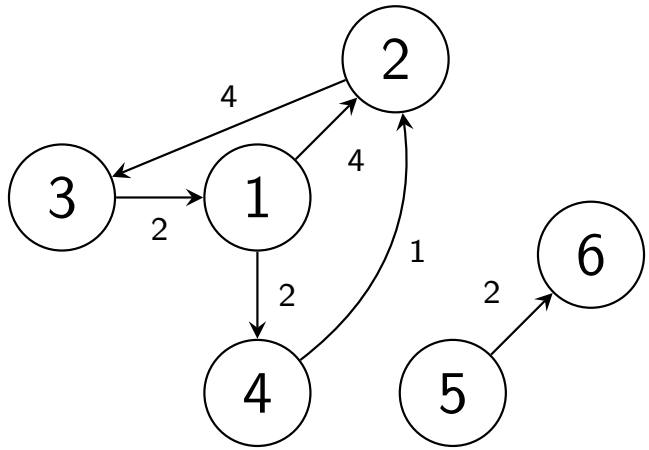
```
struct edge { int u, v, w; };
vector<edge> edges;
vector<int> d (N, oo);
int bellman_ford(int src, int dest, int n) {
    d[src] = 0;
    for (int i = 0; i < n - 1; i++)
    for (auto e : edges)
        if (d[e.u] != oo && d[e.v] > d[e.u] + e.w)
            d[e.v] = d[e.u] + e.w;
    // Verificação de ciclos negativos
    for (auto e : edges)
        if (d[e.u] != oo && d[e.v] > d[e.u] + e.w)
            return -oo;
    return d[dest];
}
```

Implementação do Floyd-Warshall

Código `floyd-warshall.cpp`

```
#include <bits/stdc++.h>
using namespace std; using ll = long long;
const ll oo = 1987654321987654321;
int main() {
    int n, m, q; cin >> n >> m >> q;
    vector<vector<ll>> d (n, vector<ll>(n, oo));
    while (m--) { int u, v, w; cin >> u >> v >> w;
        d[--u][--v] = w; }
    for (int u = 0; u < n; u++)
        d[u][u] = 0;
    for (int m = 0; m < n; m++)
    for (int u = 0; u < n; u++)
    for (int v = 0; v < n; v++)
        d[u][v] = min(d[u][v], d[u][m] + d[m][v]);
    while (q--) { int u, v; cin >> u >> v;
        cout << d[--u][--v] << "\n"; }
}
```

Exemplo do Floyd-Warshall



Entrada	Saída
6 6 4 3 1 2 1 2 4 1 4 2 2 3 4 5 6 2 4 2 1 1 3 2 1 4 5 5 6	7 6 19876543219876543 21 2