Aula 9 · Caminho Mínimo, Árvore Geradora Mínima e União-Busca

Desafios de Programação

Fernando Kiotheka Vinícius Tikara Date

UFPR

17/05/2023

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

Propriedades importante de caminhos mínimos

- Podem existir mais de um caminho mínimo entre uma origem e um destino.
- Todas as arestas de u para v com peso w que fazem parte de um caminho mínimo satisfazem:

$$dist[src][u] + w + dist[v][dst] = dist[src][dst]$$

- E todos vértice u que faz parte de um caminho mínimo satisfaz:

$$dist[src][u] + dist[u][dst] = dist[src][dst]$$

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`

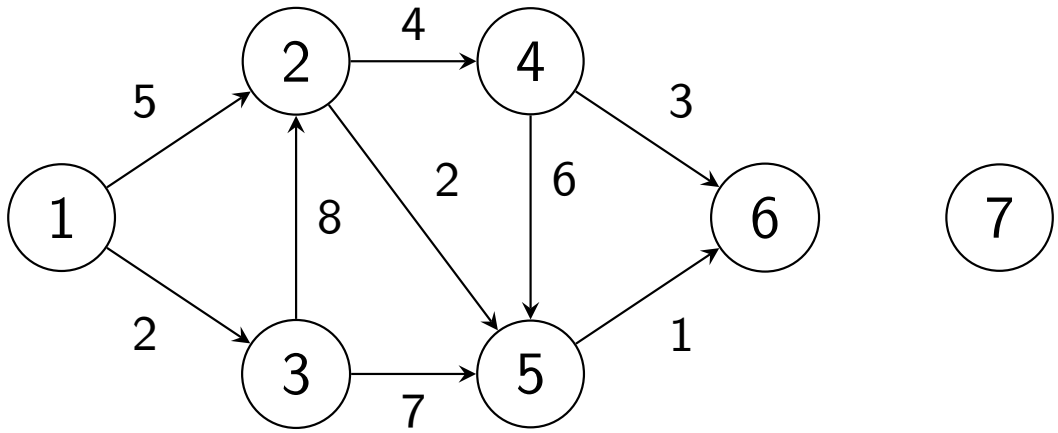
```
vector<int> d (N, oo), vis (N);
void dijkstra(int src) {
    priority_queue<pair<ll, int>,
        vector<pair<ll, int>>, greater<pair<ll, int>>> Q;
    d[src] = 0;
    Q.push({0, src});
    while (!Q.empty()) {
        auto [c, u] = Q.top(); Q.pop();
        if (vis[u]) { continue; }
        vis[u]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

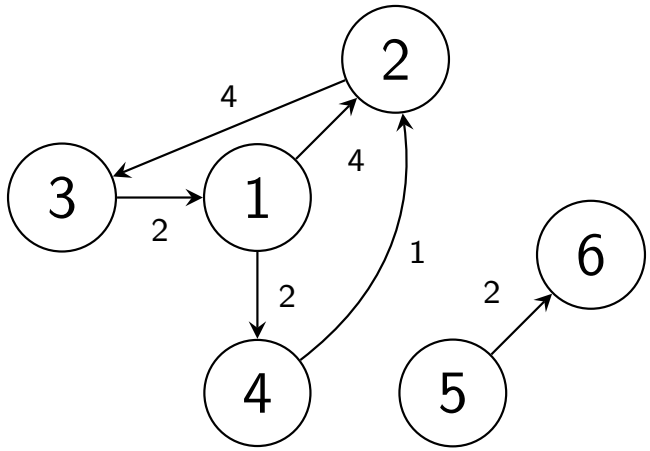
```
struct edge { int u, v, w; };
vector<edge> edges;
vector<int> d (N, oo);
int bellman_ford(int src, int dest, int n) {
    d[src] = 0;
    for (int i = 0; i < n - 1; i++)
    for (auto e : edges)
        if (d[e.u] != oo && d[e.v] > d[e.u] + e.w)
            d[e.v] = d[e.u] + e.w;
    // Verificação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

```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

Outros algoritmos de busca em grafos

Na área da Inteligência Artificial existem algumas outras buscas relevantes como

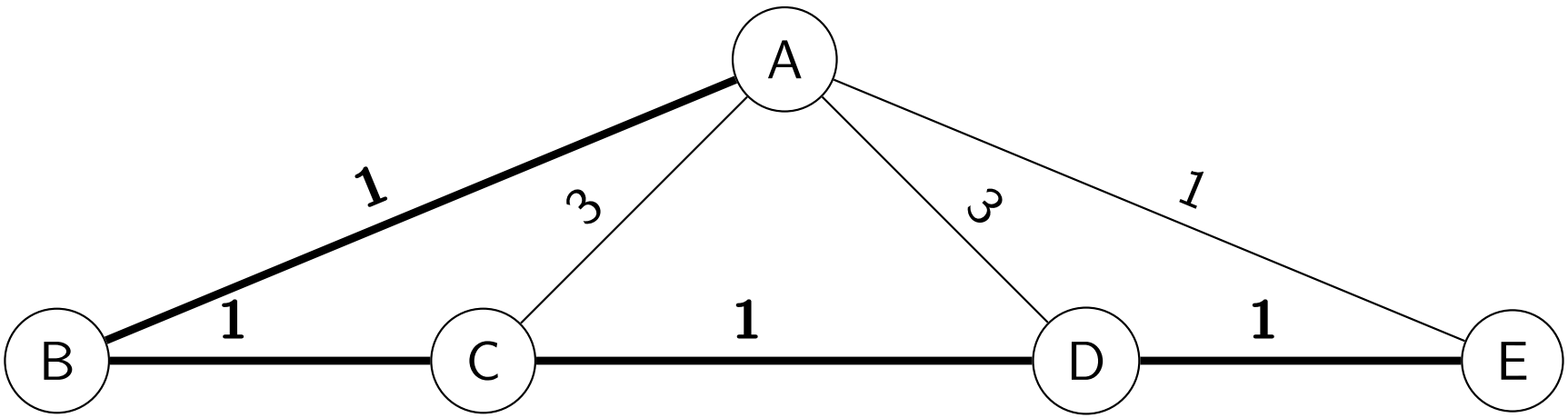
- Best-first search (pega o melhor caminho sempre)
- A*
- IDS
- IDA*

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

Árvore Geradora Mínima

Árvore Geradora Mínima

Dado um grafo G, qual o subgrafo com menor soma de pesos de arestas que inclui todos os vértices em G?



Algoritmo de Prim

Código `prim.h`

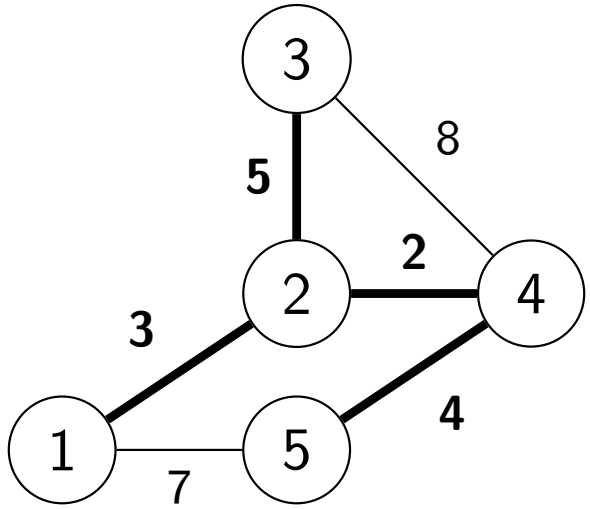
```
vector<int> d (N, oo), vis (N);
int prim(int src) {
    ll sum = 0;
    priority_queue<pair<int, int>,
        vector<pair<int, int>>, greater<pair<int, int>>> Q;
    Q.push(make_pair(d[src] = 0, src));
    while (!Q.empty()) {
        auto [c, u] = Q.top(); Q.pop();
        if (vis[u]) { continue;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

Exemplo do Prim

Código `prim.cpp`

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

Exemplo do Prim (continuado)



Entrada	Saída
5 6 1 2 3 2 3 5 2 4 2 3 4 8 5 1 7 5 4 4	14

Algoritmo de Kruskal

O algoritmo de Kruskal obtém a árvore geradora mínima escolhendo sempre a aresta mais barata que adiciona um vértice à árvore de resposta (usando união-busca)

- Porém precisamos saber quando que **podemos** adicionar uma aresta.
- Isto é, precisamos saber se a aresta cria ou não, um ciclo na árvore que está sendo criada.
- Esta estrutura é chamada de União-Busca.

União-Busca

União-busca

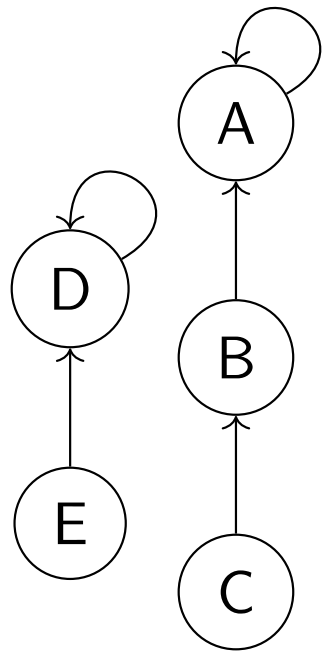
Dado um conjunto de vértices V, inicialmente cada um em um grupo que só o contém, vamos realizar dois tipos de operações:

- Unir os grupos de dois vértices v e u
- Decidir se dois vértices v e u estão no mesmo grupo

Solução trivial: para cada vértice, guardar uma lista dos vértices que pertencem ao seu grupo (requer atualizar o vetor de cada vértice no grupo)
Solução elegante: Merge Union-Find (MUF), Union-Find (UF), Disjoint Set Union (DSU), e outros nomes

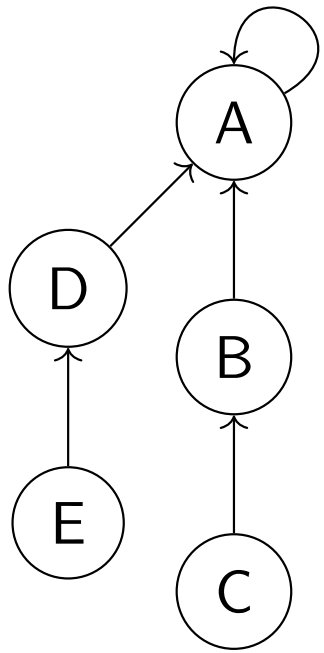
Consulta

Vamos atribuir um representante a cada grupo. Numa operação de **consulta**, percorremos a árvore de representantes até que o vértice seja seu próprio representante (raiz).



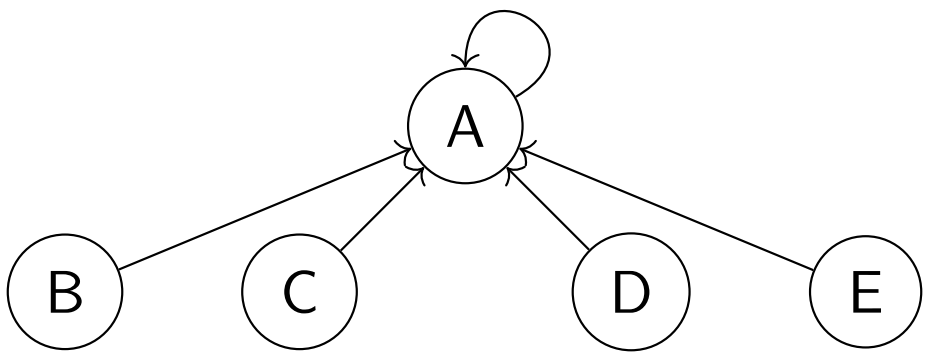
União

Numa operação de união, unimos as raízes da consulta dos dois vértices unidos.



Compressão de Caminho

Note que, já que não separamos grupos, consultas futuras sempre fazem o mesmo caminho (e, se houver um aumento no grupo, o caminho é maior). Podemos, então, salvar o resultado para facilitar consultas futuras.



Implementação do união-busca

Código `disjoint.h`

```
vector<int> rep (N);
vector<int> rnk (N);
vector<int> siz (N, 1);

int ds_find(int u) {
    if (rep[u] != u) { rep[u] = ds_find(rep[u]); }
    return rep[u];
}

void ds_union(int u, int v) {
    u = ds_find(u); v = ds_find(v);
    assert(u != v);
    if (!(rnk[u] > rnk[v])) { swap(u, v); }
    if (rnk[u] == rnk[v]) { rnk[u]++; }
    rep[v] = u;
    siz[u] += siz[v];
}
```

Implementação do Kruskal

Código `kruskal.h`

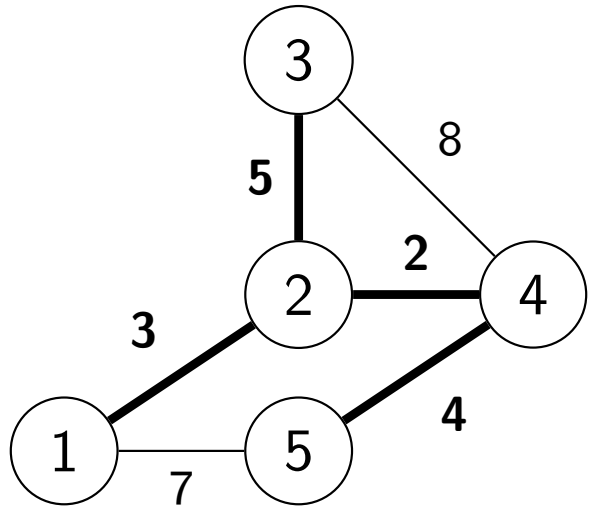
```
#include "disjoint.h"
int kruskal(int n) {
    sort(edges.begin(), edges.end());
    for (int u = 0; u < n; u++) {
        rep[u] = u; rnk[u] = 0; }
    int components = n;
    ll sum = 0;
    for (auto [u, v, w] : edges) {
        if (components == 1) { break; }
        if (ds_find(u) != ds_find(v)) {
            ds_union(u, v);
            components--;
            sum += w;
        }
    }
    return sum;
}
```

Exemplo do Kruskal

Código `kruskal.cpp`

```
#include <bits/stdc++.h>
using namespace std; using ll = long long;
struct edge {
    int u, v, w;
    bool operator<(struct edge &o) { return w < o.w; }
};
const int N = 1e5+15;
vector<edge> edges;
#include "kruskal.h"
int main() {
    int n, m; cin >> n >> m;
    while (m--) {
        int u, v, w; cin >> u >> v >> w; u--; v--;
        edges.push_back({ .u = u, .v = v, .w = w });
    }
    cout << kruskal(n) << "\n";
}
```

Exemplo do Kruskal (continuado)



Entrada	Saída
5 6 1 2 3 2 3 5 2 4 2 3 4 8 5 1 7 5 4 4	14