Aula 10 · Menor Ancestral Comum, Emparelhamento Bipartido e Fluxo Máximo/Corte Mínimo

Desafios de Programação

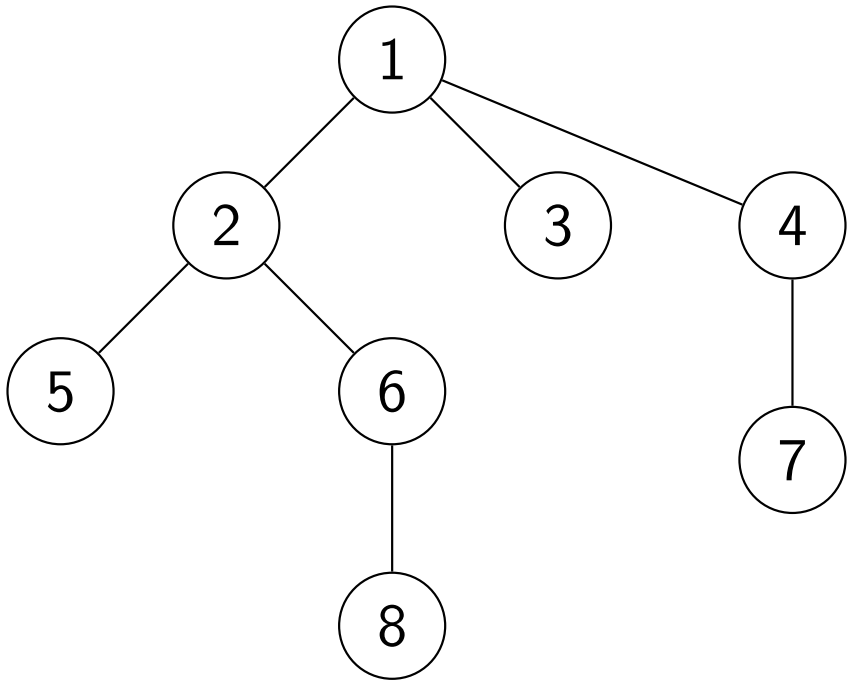
Fernando Kiotheka Victor Alflen

UFPR

10/08/2022

Menor Ancestral Comum

Menor Ancestral Comum



Subida binária

Árvore enraizada.

Código `bl.h`

```
const int L = log2(N);
vector<int> depth (N, 0);
vector<vector<int>> weiop (N, vector<int>(L+1));
vector<vector<int>> up (N, vector<int>(L+1));
void bl_euler_tour(int u, int p, int w) {
    up[u][0] = p; weiop[u][0] = w; depth[u] = depth[p] + 1;
    for (auto [v, w] : g[u]) if (v != p)
        bl_euler_tour(v, u, w);
}
void bl_init(int u, int n) {
    depth[u] = 0; bl_euler_tour(u, u, 0);
    for (int l = 0; l < L; l++)
    for (int u = 0; u < n; u++) {
        int a = up[u][l];
        up[u][l+1] = up[a][l];
        weiop[u][l+1] = OP(weiop[u][l], weiop[a][l]);
    }
}
```

Menor ancestral comum

Código `bllca.h`

```
int bl_lca(int a, int b) {
    if (!(depth[b] < depth[a])) { swap(a, b); }
    int diff = depth[a] - depth[b];
    for (int l = L; l >= 0; l--) if (diff & (1 << l))
        a = up[a][l];
    if (a == b) { return a; }
    for (int l = L; l >= 0; l--) if (up[a][l] != up[b][l])
        a = up[a][l], b = up[b][l];
    return up[a][0];
}
```

Operações no caminho

Código `blop.h`

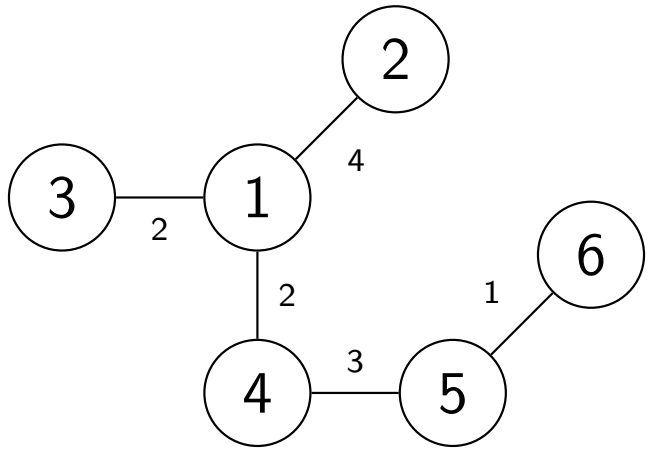
```
int bl_op(int a, int b) {
    if (!(depth[a] > depth[b])) { swap(a, b); }
    int res = NEUTRAL;
    int diff = depth[a] - depth[b];
    for (int l = L; l >= 0; l--) if (diff & (1 << l)) {
        res = OP(res, weiop[a][l]); a = up[a][l]; }
    if (a == b) { return res; }
    for (int l = L; l >= 0; l--)
        if (up[a][l] != up[b][l]) {
            res = OP(res, OP(weiop[a][l], weiop[b][l]));
            a = up[a][l], b = up[b][l];
        }
    return OP(res, OP(weiop[a][0], weiop[b][0]));
}
```

Maior peso entre dois vértices de uma árvore

Código blmax.cpp

```
#include <bits/stdc++.h>
using namespace std; using ii = pair<int, int>;
const int N = 1e5+15;
vector<vector<ii>> g (N);
#define NEUTRAL 0
#define OP(X, Y) max(X, Y)
#include "bl.h"
#include "blop.h"
int main() {
    int n, m, q; cin >> n >> m >> q;
    while (m--) {
        int u, v, w; cin >> u >> v >> w; u--; v--;
        g[u].push_back(ii(v, w)); g[v].push_back(ii(u, w));
    }
    bl_init(0, n);
    while (q--) {
        int u, v; cin >> u >> v; u--; v--;
        cout << bl_op(u, v) << "\n";
    }
}
```

Exemplo de maior peso entre dois vértices de uma árvore



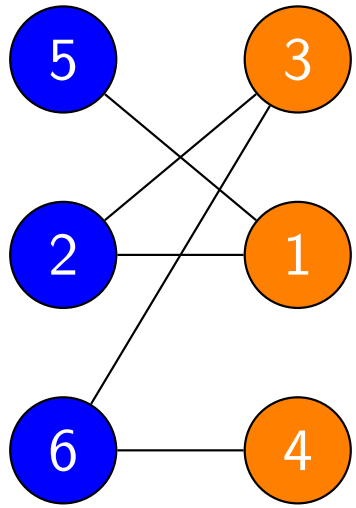
Entrada	Saída
6 5 5 3 1 2 1 4 2 2 1 4 4 5 3 5 6 1 1 6 2 6 3 2 3 4 5 6	3 4 4 2 1

Onde ir daqui

Algoritmo	Construção	Ancestral	Atualização	Memória
Subida Binária	$\mathcal{O}(n \lg n)$	$\mathcal{O}(\lg n)$	–	$\mathcal{O}(n \lg n)$
Tabela Esparsa	$\mathcal{O}(n \lg n)$	$\mathcal{O}(1)$	–	$\mathcal{O}(n \lg n)$
Árvore de Segmentos	$\mathcal{O}(n)$	$\mathcal{O}(\lg n)$	$\mathcal{O}(\lg n)$	$\mathcal{O}(n)$
Decomp. Pesado-Leve	$\mathcal{O}(n)$	$\mathcal{O}(\lg n)$	$\mathcal{O}(\lg n)$	$\mathcal{O}(n)$

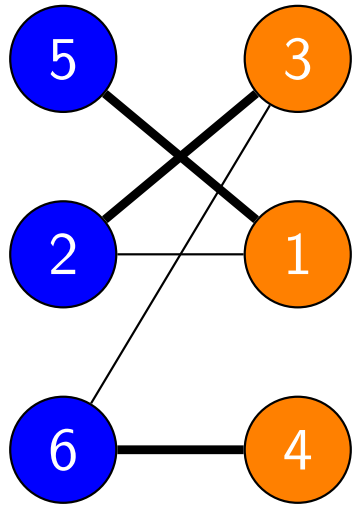
Emparelhamento Bipartido

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.

Emparelhamento de cardinalidade máxima



- Um emparelhamento é um subconjunto das arestas de forma que nenhuma aresta desse subconjunto compartilha vértices.
- O problema do emparelhamento de cardinalidade máxima é encontrar um subconjunto desse com tamanho máximo.
- Para esse grafo temos: $M = \{\{1, 5\}, \{2, 3\}, \{4, 6\}\}$, $|M| = 3$.
- Por teorema, um emparelhamento é máximo se não é possível "expandí-lo" para um emparelhamento maior. Usaremos esse conceito no algoritmo de Kuhn.

Implementação do Kuhn

Código `kuhn.h`

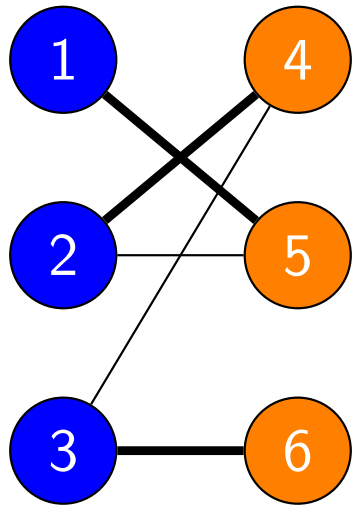
```
// u e v são enumerações distintas (podem repetir)
vector<vector<int>> g (L);
vector<int> matchr (R, -1);
int dfs(int u) {
    if (vis[u] == cts) { return 0; }
    vis[u] = cts;
    for (int v : g[u])
        if (matchr[v] == -1 || dfs(matchr[v])) {
            matchr[v] = u; return 1;
        }
    return 0;
}
int kuhn(int l) {
    int ans = 0;
    for (int u = 0; u < l; u++) { ans += dfs(u); cts++; }
    return ans;
}
```

Utilização do Kuhn

Código kuhn.cpp

```
#include <bits/stdc++.h>
using namespace std;
const int L = 1e4+15, R = 1e4+15;
int cts = 1; vector<int> vis (L);
#include "kuhn.h"
int main() {
    int l, r, m; cin >> l >> r >> m;
    while (m--) {
        int u, v; cin >> u >> v; u--; v--;
        g[u].push_back(v);
    }
    cout << kuhn(l) << "\n";
    for (int u = 0; u < r; u++) if (matchr[u] != -1)
        cout << matchr[u]+1 << " " << u+1 << "\n";
}
```

Utilização do Kuhn (continuado)



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

Dá pra fazer melhor?

- O algoritmo de Kuhn tem complexidade $\mathcal{O}(VE)$, ou seja, $\mathcal{O}(V^3)$ caso as arestas sejam completas.
- Dá. Com o Algoritmo de Hopcroft-Karp-Karzanov que tem complexidade $\mathcal{O}(\sqrt{V}E)$.
- Porém, note que as complexidades enganam: O Algoritmo de Kuhn pode ser bem rápido dependendo do grafo.
- O Algoritmo de Hopcroft é bem mais complexo, diz-se que ele é um caso especial do Algoritmo de Dinic para o Problema de Fluxo Máximo. Veremos com calma o que isso significa mais tarde.

Busca em largura do algoritmo de Hopcroft-Karp-Karzanov

Código `hopcroftbfs.h`

```
vector<int> dist (L);
bool bfs(int l) {
    queue<int> Q;
    for (int l = 0; l < L; l++)
        if (matchl[l] == -1) { dist[l] = 0; Q.push(l); }
        else { dist[l] = -1; }
    bool ans = false;
    while (!Q.empty()) {
        int l = Q.front(); Q.pop();
        for (auto r : g[l])
            if (matchr[r] == -1) {
                ans = true;
            } else if (dist[matchr[r]] == -1) {
                dist[matchr[r]] = dist[l] + 1;
                Q.push(matchr[r]);
            }
    }
    return ans;
}
```

Busca em profundidade do algoritmo de Hopcroft-Karp-Karzanov

Código `hopcroftdfs.h`

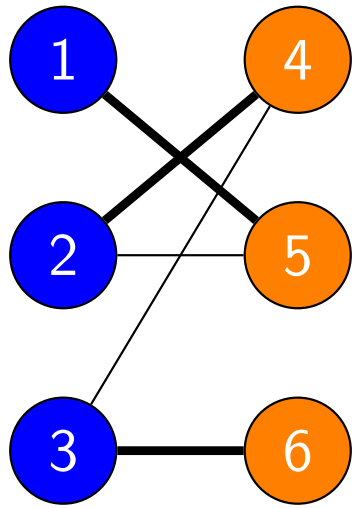
```
bool dfs(int l) {
    if (l == -1) { return true; }
    for (auto r : g[l])
        if (matchr[r] == -1 || dist[matchr[r]] == dist[l] + 1)
            if (dfs(matchr[r])) {
                matchr[r] = l;
                matchl[l] = r;
                return true;
            }
    return false;
}
```

Implementação do algoritmo de Hopcroft-Karp-Karzanov

Código hopcroft.cpp

```
#include <bits/stdc++.h>
using namespace std;
const int L = 1e4+15, R = 1e4+15;
vector<vector<int>> g (L);
vector<int> matchl (L, -1), matchr (R, -1);
#include "hopcroftbfs.h"
#include "hopcroftdfs.h"
int hopcroft(int l) {
    int ans = 0;
    while (bfs(l)) for (int u = 0; u < l; u++)
        if (matchl[u] == -1 && dfs(u)) { ans++; }
    return ans; }
int main() {
    int l, r, m; cin >> l >> r >> m;
    while (m--) { int u, v; cin >> u >> v; u--; v--;
        g[u].push_back(v); }
    cout << hopcroft(l) << "\n";
    for (int u = 0; u < r; u++) if (matchr[u] != -1)
        cout << matchr[u]+1 << " " << u+1 << "\n";
}
```

Implementação do algoritmo de Hopcroft-Karp-Karzanov (continuado)



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

Outros problemas relacionados

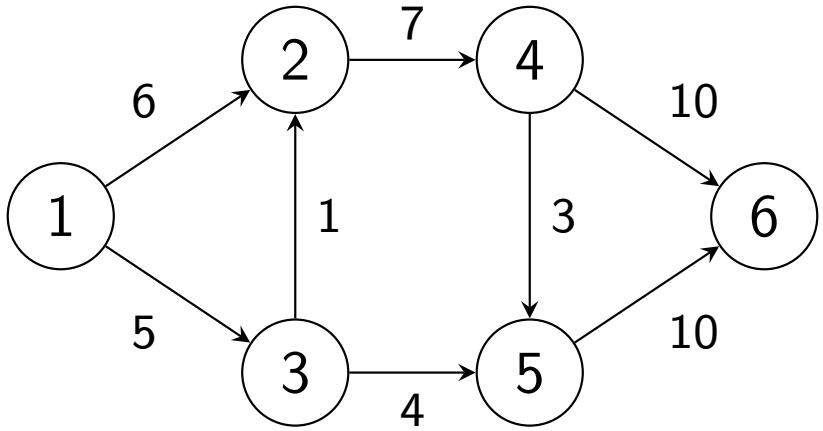
- Vimos o Algoritmo de Kuhn que é uma subrotina de um algoritmo mais complicado.
- O Algoritmo Húngaro/Kuhn-Munkres resolve o problema da atribuição ou emparelhamento de peso mínimo em $\mathcal{O}(n^3)$.
- O problema do emparelhamento é ultimamente um problema de fluxo, que veremos como resolver a seguir.

Fluxo Máximo e Corte Mínimo

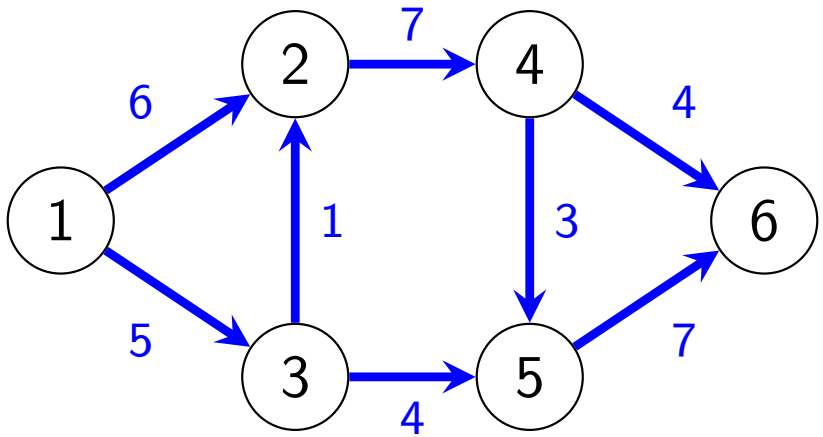
Fluxo máximo

- Cada aresta recebe uma capacidade.
- Queremos saber, dado uma origem (chamada também de fonte) e um destino (chamado também de sumidouro), qual o **fluxo máximo** possível.
- A ideia é que você pode usar todas as arestas até a sua capacidade máxima para passar qualquer coisa que seja.
- Um dos jeitos de pensar é como se fosse um fluxo de água, e a capacidade seria o diâmetro do cano, permitindo a passagem de mais ou menos água.
- Porém, é óbvio que por exemplo se há um cano pequeno no caminho, os canos subsequentes não vão conseguir passar mais água, pois ali acontece um gargalo.

Fluxo máximo ilustrado



Porém, nem todas as arestas são usadas em máxima capacidade.



Acabamos com fluxo máximo de apenas 11. Note que tudo que sai de um vértice é igual ao que entra.

Resolvendo o problema do fluxo máximo

- O algoritmo de Ford-Fulkerson resolve o problema em $\mathcal{O}(Ef_{\text{max}})$ basicamente achando um caminho da origem até o destino.
- As capacidades das arestas vão sendo reduzidas com o tempo, então os caminhos geralmente mudam.
- A redução da capacidade das arestas é feita de um em um, resultando na complexidade com esse fator f_{max} que é o fluxo máximo.
- Porém, como forma de melhorar a complexidade, dois cientistas proporam simultaneamente uma implementação chamada de Edmond-Karp que reduz o tempo para $\mathcal{O}(VE^2)$ por meio da aplicação de uma busca em largura.

Busca em largura do Ford-Fulkerson/Edmons-Karp

Código `ffekbfs.h`

```
vector<int> ix (N), dist (N), par (N);
bool minimum_path(int s, int t) {
    fill(dist.begin(), dist.end(), oo); dist[s] = 0;
    queue<int> q; q.push(s);
    while (!q.empty()) {
        int u = q.front(); q.pop();
        if (u == t) { break; }
        for (int i : res[u]) {
            edge e = edges[i]; int v = e.v;
            if (e.cap && dist[v] == oo) {
                dist[v] = dist[u] + 1;
                par[v] = u; ix[v] = i;
                q.push(v);
            }
        }
    }
    return dist[t] < oo;
}
```

Implementação do Ford-Fulkerson/Edmons-Karp

Código `ffek.h`

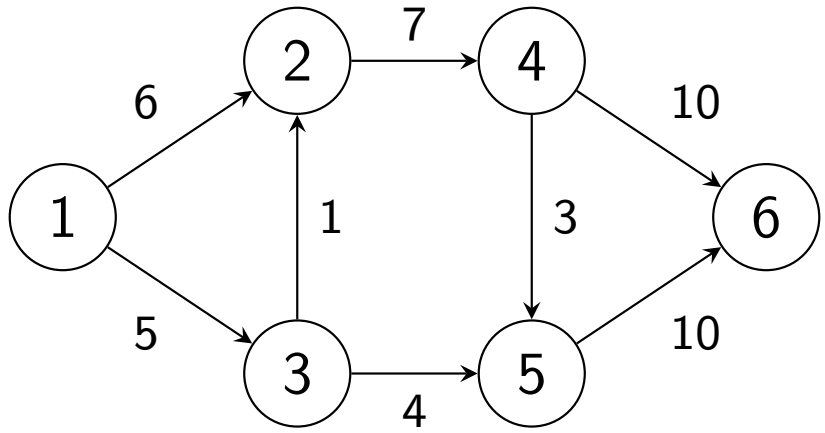
```
pair<int, int> ffek(int s, int t) {
    int min_cost = 0, max_flow = 0;
    while (minimum_path(s, t)) {
        int flow = oo;
        for (int u = t; u != s; u = par[u]) {
            flow = min(flow, edges[ix[u]].cap);
        }
        for (int u = t; u != s; u = par[u]) {
            edges[ix[u] ].cap -= flow;
            edges[ix[u]^1].cap += flow;
        }
        min_cost += flow * dist[t];
        max_flow += flow;
    }
    return { min_cost, max_flow };
}
```

Utilização do Ford-Fulkerson/Edmons-Karp

Código `ffek.cpp`

```
#include <bits/stdc++.h>
using namespace std; const int oo = 987654321;
const int N = 1e4+15; vector<vector<int>> res (N);
struct edge { int u, v, cap; }; vector<edge> edges;
#include "ffekbfs.h"
#include "ffek.h"
#define pb push_back
int main() {
    int n, m; cin >> n >> m;
    for (int i = 0; i < m; i++) {
        int u, v, c; cin >> u >> v >> c; u--; v--;
        res[u].pb(edges.size()); edges.pb({ u, v, c });
        res[v].pb(edges.size()); edges.pb({ v, u, 0 });
    }
    auto [min_cost, max_flow] = ffek(0, n-1);
    cout << min_cost << " " << max_flow << "\n";
    for (int i = 0; i < 2*m; i++) if (i % 2 && edges[i].cap) {
        edge e = edges[i]; cout << e.v+1 << " -> " << e.u+1;
        cout << " used=" << e.cap << "\n"; }
}
```

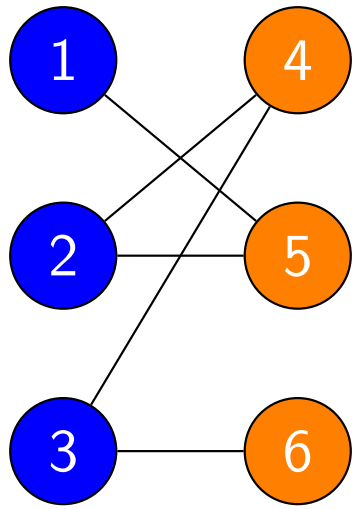
Utilização do Ford-Fulkerson/Edmons-Karp (continuado)



Entrada	Saída
6 8 1 2 6 1 3 5 3 2 1 2 4 7 3 5 4 4 5 3 4 6 10 5 6 10	34 11 1 -> 2 used=6 1 -> 3 used=5 3 -> 2 used=1 2 -> 4 used=7 3 -> 5 used=4 4 -> 6 used=7 5 -> 6 used=4

Modelando o problema do emparelhamento como fluxo

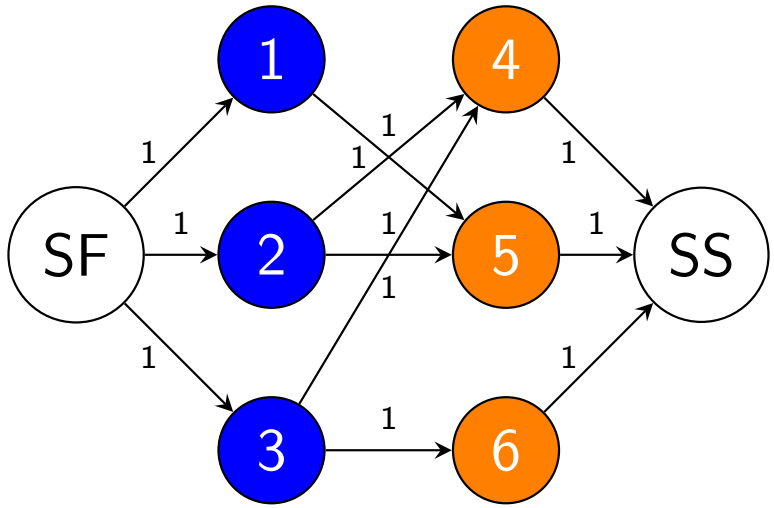
- Lembra que emparelhamento pode ser reduzido a fluxo?
- Tomemos como exemplo esse grafo bipartido que temos:



- É ideia é criar arestas de capacidade 1 da primeira partição para a segunda partição.
- Em seguida, precisamos de uma fonte e um sumidouro para rodar o algoritmo de fluxo, então criamos uma **superfonte** e um **supersumidouro**, e cada um ligará respectivamente a cada umas das partições.

Modelando o problema do emparelhamento como fluxo

- Acabamos com o seguinte grafo ao seguirmos os passos:



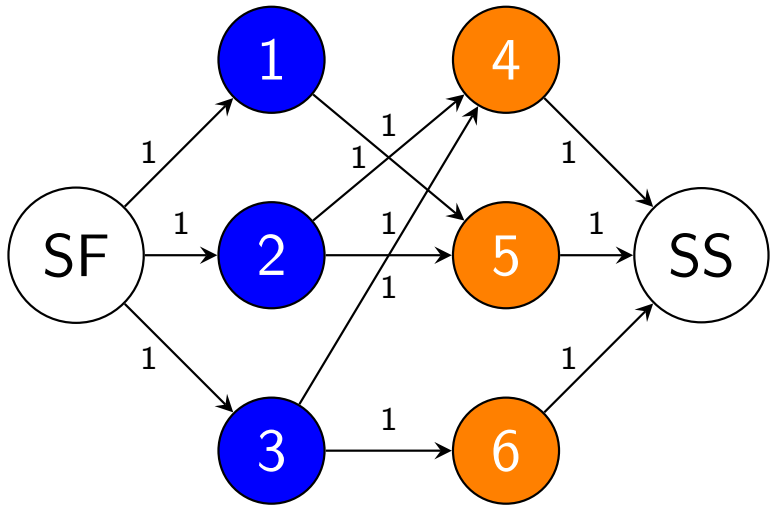
- Vamos fazer um programa que pega aquela entrada e faz isso automaticamente?

Emparelhamento bipartido usando fluxo

Código `flowmatch.cpp`

```
#include <bits/stdc++.h>
using namespace std; const int oo = 987654321;
struct edge { int u, v, cap; }; vector<edge> edges;
const int N = 1e4+15; vector<vector<int>> res (N);
#include "ffekbfs.h"
#include "ffek.h"
void link(int u, int v) {
    res[u].push_back(edges.size()); edges.push_back({ u, v, 1 });
    res[v].push_back(edges.size()); edges.push_back({ v, u, 0 }); }
int main() {
    int l, r, m; cin >> l >> r >> m;
    for (int i = 0; i < m; i++) {
        int u, v; cin >> u >> v; u--; v--; link(u, l+v); }
    int ssrc = l+r, sdst = l+r+1;
    for (int u = 0; u < l; u++) { link(ssrc, u); }
    for (int u = l; u < l+r; u++) { link(u, sdst); }
    cout << ffek(ssrc, sdst).second << "\n";
    for (int i = 0; i < 2*m; i++) if (i % 2 && edges[i].cap) {
        edge e = edges[i]; cout << e.v+1 << " " << e.u+1-l << "\n"; }
}
```

Emparelhamento bipartido usando fluxo (continuado)



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

Corte Mínimo

- O corte mínimo é um conjunto de arestas cujo peso é mínimo que se removidas desconectariam a origem e o destino.
- O peso das arestas é modelado apenas como capacidade das arestas.
- O Teorema do Fluxo Máximo e Corte Mínimo determina que o fluxo máximo é igual ao peso total das arestas do corte mínimo.
- Isso significa que é simples achar o corte mínimo: Após encontrar o fluxo máximo, faça uma busca em profundidade partindo da origem, percorrendo apenas arestas com capacidade residual.
- Assim, você obtém uma das partes do corte mínimo. A outra é o conjunto de vértices oposto.

E que tal colocar um custo?

- Podemos, ao invés de considerar apenas a capacidade das arestas, adicionar também um *custo* pra cada aresta.
- Imagine por exemplo se usar determinada aresta em uma rede custa proporcionalmente a banda de rede que passa nele (que nem a Internet da sua casa, bom, mais ou menos...).
- Cada aresta tem então duas propriedades: capacidade e custo.
- Esse é um problema que é felizmente simples de resolver: Ao invés de usar um BFS que considera tudo com custo 1, a gente usa um algoritmo de caminho mínimo.
- Esse algoritmo de caminho mínimo tem que suportar pesos negativos, então Dijkstra não cola. É ou o Bellman-Ford ou um melhoramento dele que a gente vai usar, o SPFA (que tem no pior caso, a mesma complexidade).

Implementação do SPFA

Código `ffekspfa.h`

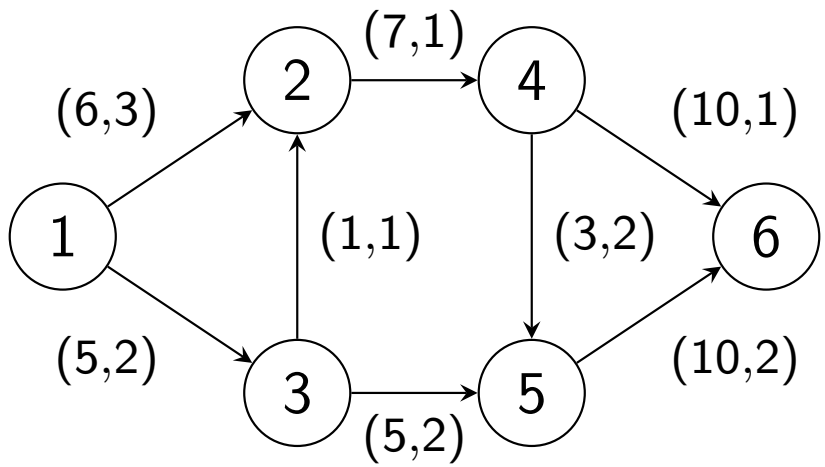
```
vector<int> queu (N), ix (N), dist (N), par (N);
bool minimum_path(int s, int t) {
    fill(dist.begin(), dist.end(), oo); dist[s] = 0;
    queue<int> q; q.push(s);
    while (!q.empty()) {
        int u = q.front(); q.pop();
        queu[u] = 0;
        for (int i : res[u]) {
            edge e = edges[i]; int v = e.v;
            if (e.cap && dist[v] > dist[u] + e.cost) {
                dist[v] = dist[u] + e.cost;
                par[v] = u; ix[v] = i;
                if (!queu[v]) { q.push(v); queu[v] = 1; }
            }
        }
    }
    return dist[t] < oo;
}
```

Implementação do fluxo máximo com custo mínimo

Código `ffekmincost.cpp`

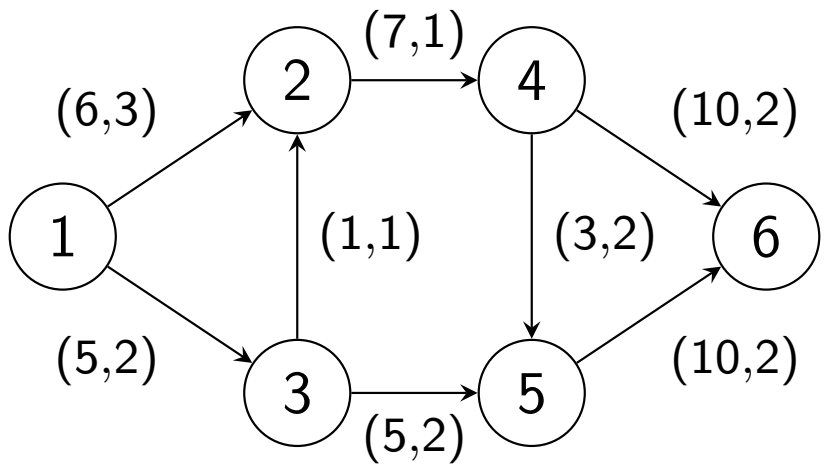
```
#include <bits/stdc++.h>
using namespace std; const int oo = 987654321;
const int N = 1e4+15; vector<vector<int>> res (N);
struct edge { int u, v, cap, cost; }; vector<edge> edges;
#include "ffekspfa.h"
#include "ffek.h"
#define pb push_back
int main() {
    int n, m; cin >> n >> m;
    while (m--) {
        int u, v, c, w; cin >> u >> v >> c >> w; u--; v--;
        res[u].pb(edges.size()); edges.pb({ u, v, c, w });
        res[v].pb(edges.size()); edges.pb({ v, u, 0, -w });
    }
    auto [min_cost, max_flow] = ffek(0, n-1);
    cout << min_cost << " " << max_flow << "\n";
    for (auto e : edges) if (e.cap && e.cost < 0) {
        cout << e.v+1 << " -> " << e.u+1 << " used=" << e.cap;
        cout << " (cost=" << e.cap * -e.cost << ")\n"; }
}
```

Implementação do fluxo máximo com custo mínimo (continuado)



Entrada	Saída
6 8 1 2 6 3 1 3 5 2 3 2 1 1 2 4 7 1 3 5 5 2 4 5 3 2 4 6 10 1 5 6 10 2	59 11 1 -> 2 used=6 (cost=18) 1 -> 3 used=5 (cost=10) 3 -> 2 used=1 (cost=1) 2 -> 4 used=7 (cost=7) 3 -> 5 used=4 (cost=8) 4 -> 6 used=7 (cost=7) 5 -> 6 used=4 (cost=8)

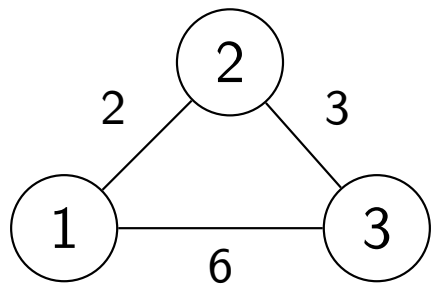
Implementação do fluxo máximo com custo mínimo (continuado)



Entrada	Saída
6 8 1 2 6 3 1 3 5 2 3 2 1 1 2 4 7 1 3 5 5 2 4 5 3 2 4 6 10 2 5 6 10 2	66 11 1 -> 2 used=6 (cost=18) 1 -> 3 used=5 (cost=10) 2 -> 4 used=6 (cost=6) 3 -> 5 used=5 (cost=10) 4 -> 6 used=6 (cost=12) 5 -> 6 used=5 (cost=10)

Mais generalizações do fluxo

- Esse algoritmo agora considerando o custo pode ser usado para resolver aquele problema da atribuição (maior cardinalidade, arestas de menor custo).
- Ultimamente, os problemas de fluxo são problemas de programação linear (bom que a gente tem a matéria de Otimização pra isso!)



$$\begin{aligned} \min : 2|a_{12}| + 3|a_{23}| + 6|a_{13}| \\ a_{12} + a_{13} = 10 \\ a_{23} + a_{13} = 10 \\ a_{12} = a_{23} \end{aligned}$$

E isso é tudo!

- O conteúdo que vocês precisam para fazer a competição que logo começará está todo aqui.
- Existem muitos algoritmos que a gente não viu como o Stoer-Wagner que serve para encontrar o corte mínimo global (sem origem e destino).
- Veremos mais adiante alguns conteúdos mais avançados relacionados como a Decomposição Pesado-Leve.
- O Algoritmo de Dinic é um algoritmo de fluxo máximo com outra complexidade ($\mathcal{O}(V^2E)$) e é um pouco mais complicado (`https://cp-algorithms.com/graph/dinic.html`).
- Bons estudos!