Studijní materiál na BI-GRA15

Martin Holoubek

22. května 2015

1 Definice

Je-li graf souvislý, potom platí, že: $|H| \geq |U| - 1$

$\Rightarrow$ souvislý graf o n uzlech musí mít alespoň $n - 1$ hran (kružnice)

multigraf = graf, který může mít smyčky, rovnoběžné hrany
prostý graf = multigraf bez rovnoběžných hran
obyčejný graf = prostý graf bez smyček

orientovaný graf = graf s orient. a rovnob. hranami a smyčkami
prostý OG = graf bez rovnoběžných hran
obyčejný OG = prostý OG bez smyček

podgraf = podgraf $G'\langle U', H', \rho' \rangle$ grafu $G\langle U, H, \rho \rangle$, takový, že $U' \subseteq U$, $H' \subseteq H$ při zachování incidence.
faktor = podgraf se všemi uzly (hranový podgraf)
izomorfismus = izomof. dvou grafů je vzájemné zobrazení jejich hran a uzlů při zachování incidence.
automorfismus = izomorfismus grafu na sebe

sousední uzly = dva uzly, které incidují se stejnou hranou
stupeň uzlu, $\delta(u)$ = počet hran, které s uzlem incidují[1]

$\Rightarrow$ obecně pro graf $G\langle U, H, \rho \rangle$ platí, že $\sum_{u \in U} \delta(u) = 2\,|H|$

sled = posloupnost uzlů a hran, začíná a končí uzlem
tah = sled, ve kterém se neopakují hrany
cesta = tah, ve kterém se neopakují ani uzly
délka tahu = počet hran v tahu

kružnice = uzavřená cesta délky alespoň 1
cyklus = uzavřená orientovaná cesta

(slabě) souvislý graf = mezi všemi body existuje sled
(slabá) komponenta = maximální slabě souvislý podgraf
silně souvislý graf = mezi všemi body u, v existuje spojení z u do v

[1] Smyčka zvyšuje stupeň uzlu o 2.

silná komponenta = maximální silně souvislý podgraf

kondenzace = silné komponenty se změní na uzly a necháme zbylé hrany
strom = souvislý graf bez kružnic
kostra = faktor grafu, který je stromem

⇒ silně souvislý graf má každou hranu v cyklu

$\delta^+(u)$**, výstupní stupeň** = počet hran vedoucích z uzlu
$\delta^-(u)$**, vstupní stupeň** = počet hran vedoucích do uzlu

⇒ obecně pro OG $G\langle U, H, \rho\rangle$ platí, že $\sum_{u\in U} \delta^+(u) = \sum_{u\in U} \delta^-(u) = |H|$

acyklický graf = orientovaný graf bez cyklů
topologické uspoř. = posloupnost $u_1, u_2 \dots u_n$ tak, že pro $\forall h(u_i, u_j) \in H$ platí, že $i < j$

nalezení topologického uspořádání = postupně odebírám uzly, jejichž stupeň je 1, snižuji stupeň jejich sousedů. Pořadí odebírání je jejich topologické uspořádání.

matice incidence = matice $U \times H$, sloupce obsahují 1 v místě uzlu, se kterým daná hrana inciduje.
matice sousednosti = matice $U \times U$, obsahuje 1 v poli, které značí sousednost dvou uzlů.
spojová reprezentace = indexované pole spojových seznamů na sousedy daného uzlu

pokrytí grafu = rozklad množiny hran do tříd, kde každá třída je tahem.
minimální pokrytí = pokrytí s minimálním počtem tahů.
Eulerův graf (neorient.) = všechny uzly mají sudý stupeň

⇒ aby graf mohl být pokryt jedním tahem musí být (slabě) **souvislý** a **Eulerův**.

Pokud má souvislý graf $2n$ uzlů lichého stupně

⇒ potom k jeho pokrytí stačí n **otevřených** tahů.

Eulerův graf (orient.) = pro $\forall u \in U$ platí, že $\delta^+(u) = \delta^-(u)$

Pokud u souvislého OG rozdělíme uzly do množin $U_1 = \{u : \delta^+(u) = \delta^-(u)\}, U_2 = \{u : \delta^+(u) > \delta^-(u)\}, U_3 = \{u : \delta^+(u) < \delta^-(u)\}$

$\Rightarrow$ potom k jeho pokrytí stačí k orientovaných tahů.

$$k = \sum_{u \in U_2} \delta^+(u) - \delta^-(u) = \sum_{u \in U_3} \delta^-(u) - \delta^+(u)$$

Hamiltonovská cesta = kružnice, která obsahuje všechny uzly grafu.
Hamiltonovský graf = graf, který obsahuje Hamiltonovskou kružnici.

nezávislá podmnožina = taková podmnožina uzlů, pro kterou platí, že žádné dva její uzly nejsou svými sousedy.
nezávislost, $\alpha(G)$ = velikost maximální nezávislé podmnožiny.
klika = maximální úplný podgraf.
klikovost, $\omega(G)$ = velikost kliky.

$\Rightarrow$ obecně platí, že $\omega(G) = \alpha(G')$, kde G' je doplněk grafu G.
dominující podmnožina = taková podmnožina uzlů, pro něž platí, že spolu se svými sousedy tvoří celou množinu uzlů.
dominance, $\beta(G)$ = velikost minimální dominující podmnožiny.

$\Rightarrow$ obecně platí, že $\beta(G) \leq \alpha(G)$, rovnost nastává, pokud je množina maximální nezávislá a zároveň minimální dominující.
barevnost, $\chi(G)$ = minimální počet barev, kterými lze graf obarvit tak, každá hrana vede mezi dvěma různě obarvenými uzly.

$\Rightarrow$ obecně platí, že $\chi(G).\alpha(G) \geq |U|$

$\Rightarrow$ obecně platí, že $\chi(G) \geq \omega(G)$

$\Rightarrow$ obecně platí, že $\chi(G) \leq \delta_{max} + 1$

$\Rightarrow$ obecně platí, že $\chi(G) = 2 \Leftrightarrow G$ neobsahuje kružnici liché délky

vzdálenost = mezi dvěma uzly u a v je délka (počet hran) nejkratší cesty mezi těmito uzly.
průměr grafu, $T(G)$ = největší vzdálenost dvou uzlů v grafu.
excentricita uzlu, $e(u) = max(d(u, v)) : \forall v \in U$.
poloměr grafu, $r(G)$ = minimální excentricita na grafu.

$\Rightarrow$ obecně platí, že $r(G) \leq T(G) \leq 2 * r(g)$

$\Rightarrow$ u stromů platí, že $T(G) = 2 * r(g)$ nebo $T(G) = 2 * r(g) - 1$

tětiva = hrana, které není v kostře.
cyklomatické číslo, $\mu(G)$ = počet lineárně nezávislých kružnic v grafu = počet tětiv.

$\Rightarrow$ obecně platí, že $\mu(G) = |H| - |U| + p$, kde p je počet komponent.
hodnost grafu, $h(G)$ = počet hran kostry, $h(G) = |U| - p$, kde p je počet komponent.
hloubka uzlu, $hl(u)$ = v kořenovém stromu je to vzdálenost od kořene.
hloubka grafu, $hl(G)$ = maximální hloubku uzlu v grafu.

$\Rightarrow$ obecně platí, že **pravidelný strom** stupně r s k vnitřními uzly má $k(r-1)+1$ listů a $k * r$ hran.

vnitřní délka stromu = součet hloubky všech vnitřních uzlů stromu.
vnější délka stromu = součet hloubky všech listů stromu.

w - délka spojení = součet ohodnocení všech hran, které spojení tvoří.
w - vzdálenost, $d_w(u, v)$ = nejmenší délka spojení mezi uzly u a v.

$\Rightarrow$ obecně pro konečné i nekonečné vzdálenosti platí trojúhelníková nerovnost, $d_w(s, v) \leq d_w(s, u) + w(u, v)$, kde (u, v) je libovolná hrana a s je nějaký uzel.

hranový řez = taková minimální podmnožina hran S, že po jejím odebrání se graf rozpadne na dvě komponenty.

$\Rightarrow$ lze formulovat jako $S \subseteq H : h(G - S) = h(G) - 1$.
artikulace = uzel grafu u takový, že po jeho odebrání bude mít graf více komponent.

$\Rightarrow$ množina hran incidujících s uzlem u je hranový řez, **právě když** u **není artikulace**.

$\Rightarrow$ hranový řez obsahuje alespoň jednu hranu kostry, proto se zmenší $h(G)$.

planární graf = graf, který lze zakreslit v $\mathbb{R}^2$ bez křížení hran.

$\Rightarrow$ nechť G je planární graf $\Rightarrow$ platí, že $|H| - |U| + 2 = r$, kde r je počet "stěn"grafu a tzv. **Eulerovo číslo**.

$\Rightarrow$ pro planární grafy platí, že $|H| = k\frac{|U|-2}{k-2}$, to lze použít k důkazu, že graf není planární ne naopak.

$\Rightarrow$ základní graf, který není planární je $K_{3,3}$ a K_5.
homeomorfismus = dva grafy jsou homeomorfní jsou-li izomorfní, nebo se jimi stanou půlením hran.

$\Rightarrow$ graf je planární, **právě když neobsahuje** podgraf homeomorfní s $K_{3,3}$ ani K_5.

síť = síť se skládá z orientovaného grafu, kapacity hran (zobrazení hran na celá čísla), zdroje a spotřebiče, což jsou dva význačné uzly.
přípustný tok = je takový tok, který na žádné hraně nepřekračuje kapacity a pro každý uzel platí Kirchhoffův zákon.
Kirchhoffův zákon = požaduje, aby se suma toho, co do uzlu vtéká rovnala sumě toho, co odtéká.
velikost toku = rozdíl toho, co odteče ze zdroje a toho, to ce do něj vrací. Nebo rozdíl toho, co přiteče do spotřebiče, mínus to, co z něj odtéká.
řez sítě = takový řez, který odděluje zdroj od spotřebiče.
kapacita řezu = suma kapacity hrany, které by vedly z komponenty ze zdrojem do komponenty se spotřebičem.

$\Rightarrow$ velikost maximálního toku je rovna kapacitě minimálního řezu.

cirkulace = tok v síti, kdy pro každý uzel platí, že suma vtékající je rovná sumě vytékající.

$\Rightarrow$ cirkulace existuje, právě když **každý** hranový řez má nezápornou kapacitu.

2 Algoritmy

2.1 Základní algoritmy

2.1.1 BFS

Prohledávání do šířky, používá se fronta. Uzly mají stavy **FRESH**, **OPEN**, **CLOSE**

```
void BFS (Graph G, Node s) {
        for (Node u in U(G)-s){
                stav[u] = FRESH;
        } stav[s] = OPEN;
        Queue.Init(); Queue.Push(s);
        while (!Queue.Empty()) {
                u = Queue.Pop();
                for (v in Adj[u]) {
                        if (stav[v] == FRESH) {
                                stav[v] = OPEN; Queue.Push(v);
                        }
                } stav[u] = CLOSED;
        }
}
```

2.1.2 DFS

Prohledávání do hloubky, používá se zásobník. Uzly mají stavy **FRESH**, **OPEN**, **CLOSE** a také časové značky otevření $d[t]$ a uzavření $f[t]$, platí, že $d[t] < f[t]$

DFS klasifikuje 4 druhy hran: **stromová**, **zpětná** vede k **OPEN** uzlu, **dopředná** $h(u,v), d[u] < d[v]$, **příčná** $h(u,v), d[u] < d[v]$
Složitost je O($|U|+|H|$)

```
void DFS (Graph G)
{
        for (Node u in U(G)) {
                stav[u] = FRESH; p[u] = null;
        } i = 0;
```

```
        for (Node u in U(G))
                if (stav[u] == FRESH)
                        DFS-Projdi(u);
}

void DFS-Projdi(Node u) {
        stav[u] = OPEN; d[u] = ++i;
        for (Node v in Adj[u]) {
                if (stav[v] == FRESH) {
                        p[v] = u; DFS-Projdi(v);
                }
        }
        stav[u] = CLOSED; f[u] = ++i;
}
```

2.1.3 Topo-sort

Řeší topologické uspořádání uzlů. Na graf pustíme DFS a v okamžiku uzavření uzlu jej vložíme na zásobník. Zásobník poté obsahuje uzly topologicky uspořádané.

2.1.4 Silné komponenty pomocí DFS

DFS určí pro všechny uzly časy otevření a uzavření. Vytvoříme nový opačně orientovaný graf. Provedeme na něm DFS tak, že v hlavním cyklu bereme uzly v pořadí klesající příznaku uzavření uzlu. Stromy DFS lesa označují silné komponenty.[2]

2.1.5 Hledání Eulerova tahu

Greedy algoritmus, který zkouší na zásobníku všechny možnosti.

Ověříme, že je graf souvislý a jeho uzly mají sudý stupeň. Vybereme počáteční uzel s a vložíme jej na zásobník. Dokud není zásobník prázdný a uzel u je na vrcholu. Pokud má uzel u nějakou incidující hranu (u, v)

[2]Viz. druhý úkol na progtestu

odstraníme hranu z grafu a přidáme v na zásobník. Pokud nemá incidující hranu odebereme ho ze zásobníku a přidáme do výsledného Eulerova tahu.

```
function find_Euler_Trail(Graph g, Node u):
        stack.push(u)
        result = {}
        while not stack.Empty():
                edge = stack.top().adjacentEdge()
                if edge not None:
                        g.removeEdge(edge)
                        stack.push(edge.to())
                else:
                        result.insert(stack.top())
                        stack.pop()
        return result
```

2.2 Hledání kostry

2.2.1 Borůvkův - Kruskalův algoritmus

Algoritmus hledání **minimální kostry** na ohodnoceném grafu. Hrany si seřadí podle ceny a postupně je přidává do kostry a kontroluje, zda daná hrana nevytvoří kružnici.
Kontrola kružnic funguje na principu rozdělení uzlů do množin, pokud přidáme hranu mezi dva uzly ze stejné množiny, víme, že už mezi nimi existuje cesta a proto jsme právě vytvořili kružnici.
Složitost je $O(|H|+\lg|H|)$, kterou způsobuje řazení, zbytek algoritmu je rychlejší.

```
void BK–MST(Graph G, Weights w) {
        for (Node u in U(G))
                makeSet(u);
        T = Empty; sortEdges(G)
        for (Edge [u,v] in H(G)) {
                if (FIND(u) != FIND(v)) {
                        T.addEdge([u, v])
                        unionSet(u,v);
                }
```

```
        }
        return T;
}
```

2.2.2 Jarníkův - Primův algoritmus

Algoritmus hledání **minimální kostry** na ohodnoceném grafu. Tvoří si množinu uzlů, které má spojené minimální kostrom, postupně ji zvětšuje tak, že přidává nejkratší hranu, které spojuje tuto množinu s nějakým jiným uzlem.
Algoritmus pro svou činnost používá prioritní frontu a jeho zradou je, že **potřebuje měnit priority** již vložených prvků. Složitost je $O(|U|) + O(|U| * \log|U|) + O(|H| * \lg|U|)$.

```
void JP–MST(Graph G, Weights w, Node r) {
        Q = U; T = Empty;
        for (Node u in Q)
                d[u] = inf.;
        d[r] = 0; p[r] = null;
        while (not Q.empty()) {
                u = Q.ExtMin();
                for (Node v in Adj[u]){
                        if ((v in Q) && (w(u,v) < d[v])){
                                p[v] = u;
                                //Here the priority changes
                                d[v] = w(u,v);
                        }
                }
        }
}
```

2.3 Kódování dat

2.2.1 Huffmanův algoritmus

Slouží ke generování jednoznačného optimálního prefixového stromu. Platí, že žádné slovo není prefixem jiného slova, proto můžeme vždy deter-

ministicky určit další postup.
Algoritmus je uveden pro **zobecněný stupeň** uzlu. Pro stupeň uzlu 2 se bere v úvahu pouze x a y.

```
void HuffmanTree(Weights w, int n) {
        Q.Init();
        for (int i = 1; i <= n; i++) {
                u = MakeNode(w[i]);
                Q.Push(u);
        }
        for (int i = 1; i < n; i++) {
                x = Q.ExtMin(); y = Q.ExtMin();
                # z = Q.ExtMin();
                node = MakeNode(w[x] + w[y] ... + w[z]);
                node.setChild(0, x); node.setChild(1, y);
                # node.setChild(2, z);
                Q.Push(node);
        }
        return Q.ExtMin();
}
```

2.4 Určování vzdálenosti od uzlu

2.4.1 Společné funkce pro určení vzdálenosti

Inicializace vzdáleností a počátečního uzlu.

```
void InitPaths(Graph G, Node s) {
        for(Node u in U(G)) {
                d[u] = +inf.;
                p[u] = null;
        } d[s] = 0;
}
```

Princip relaxace (kontrola nalezení lepší cesty)

```
void Relax (Node u, Node v, Weights w) {
        newVal = d[u] + w(u,v);
```

```
        if(d[v] > newVal) {
                d[v] = newVal; p[v] = u;
        }
}
```

2.4.2 Dijkstrův algoritmus

Dijkstrův algoritmus určí vzdálenosti všech uzlů od počátečního uzlu. Funguje na orientovaných i neorientovaných grafech. Ohodnocení hran musí být z $\mathbb{R}_{\geq 0}$. V každém kroku algoritmus vyřeší jeden uzel a přepočítá všechny vzdáledosti k sousedním uzlům. Správnost algoritmu je založena na tom, že pokaždé bereme **uzel s nejmenší vzdáleností** od počátku. Složitost je $O(|H| + |U|^2) = O(|U|^2)$.

```
void Dijkstra(Graph G, Node s, Weights w) {
        InitPaths(G,s);
        S = Empty; Q = U(G);
        while(not Q.empty()) {
                u = Q.pop(); //u with lowest distance
                S = S union {u};
                for(Node v in Adj[u]) {
                        Relax(u, v, w);
                }
        }
}
```

2.4.3 Belman - Fordův algoritmus

Dijkstrův algoritmus určí vzdálenosti všech uzlů z počátečního uzlu. Ohodnocení hran mohou být z $\mathbb{R}$, tzv. funguje správně i s **negativním ohodnocením**. V každém cyklu algoritmus relaxuje pohle všechn hran v grafu. Postupně tak počítá nejkratší cesty určité délky podle čísla iterace.
Další vlastností algoritmu je detekce záporných cyklů. Princip je jednoduchý, po doběhnutí algoritmus už nemůžeme najít kratší cestu, protože jsme všechny již zkusili. Pokud kratší cestu najdeme, znamená to, že graf obsahuje záporný cyklus. Složitost je $O(|U| * |H|)$.

```
bool Bel–Ford(Graph G, Node s, Weights w) {
        InitPaths(G, s);
        for (int i = 1; i < |U|; i++)
                for(Edge (u,v) in H)
                        Relax(u,v,w);
        for(Edge (u,v) in H) {
                if(d[v] > d[u] + w(u,v))
                        return false;
        }
        return true;
}
```

2.4.4 Úprava pro acyklické grafy

Nejdřív provedeme topologické seřazení uzlů. Potom v uzly v hlavním cyklu **bereme podle** jejich topologického uspořádání.
Složitost topologického uspořádání i výsledného algoritmu je O($|U|+|H|$)

```
void DAG–Paths(Graph G, Node s, Weights w) {
        TopoSort(G);
        InitPaths(G, s);
        for(Node u in U(G)) {
                for(Node v in Adj[u])
                        Relax(u,v,w);
        }
}
```

2.5 Určení vzdálenosti mezi všemi uzly

2.5.1 Floyd - Warshall algoritmus

Algoritmus postupně propočítává vzdálenosti mezi všemi uzly, index i určuje momentálně počítanou vzdálenost. Pomocí úpravy algorimu lze snadno počítat matici předchůdců, tranzitivně reflexivní uzávěr (relace) a nebo dostupnost (boolovská matice). Počítáme s maticí vzdáleností W a maticí předchůdců P.
Složitost algoritmu je O($|U|^3$)

$$W_{ij}^0 = \begin{cases} 0 & \text{if } i = j \\ w(u_i, u_j) & \text{if } i \neq j \wedge (u_i, u_J) \in H \\ \infty & \text{else} \end{cases}$$

$$P_{ij}^0 = \begin{cases} null & \text{if } i = j \vee \text{path not exists} \\ u_k & \text{if } u_k \text{ is predecessor } u_j \text{ in path}(u_i, u_j) \end{cases}$$

$$P_{ij} = \begin{cases} P_{ij} & \text{if } d_{ij} \leq d_{ik} + d_{kj} \\ P_{kj} & \text{if } d_{ij} > d_{ik} + d_{kj} \end{cases}$$

```
float** Floyd-Warshall(Weights W) {
        D = copy(W);
        for (k = 1; k <= n; k++) {
                for (i = 1; i <= n; i++) {
                        for (j = 1; j <= n; j++) {
                                newDist = D[i][k] + d[k][j];
                                D[i][j] = min(D[i][j], newDist);
                        }
                }
        }
        return D;
}
```

2.5.2 Johnsonův algoritmus

Zlepšený algoritmus pro nalezení všechn vzdáleností pro záporně ohodnocené grafy. Pridáme nový uzel x a hrany z něj ke všem původním uzlům. Tyto hrany mají cenu 0. Pomocí Bellman-Forda spočítáme vzdálenosti $d(x, u) : \forall u \in U$. Přehodnotíme původní hrany. $w'(u, v) = w(u, v) + d(x, u) - d(x, v) : \forall h(u, v) \in H$.

Nyní pustíme pro každý uzel Dijkstrův algoritmus s novým ohodnocením a po jeho doběhnutí přepočítáme vzdálenosti podle původních cen.

```
bool Johnson(Graph G, Weight W, Node x) {
        new_G = G.addNode(x);
        if (!Bellman-Ford(new_G, W, x))
```

```
            return false;
        for (Node u in U(G))
            h(u) = d(x,u);
        for (Edge (u,v) in H(G))
            new_W(u,v) = W(u,v) + h(u) - h(v);
        for (Node u in U(G)) {
            Dijkstra(G, new_W, u);
            for (Node v in U(G))
                d(u,v) = new_d(u,v) - (u) + h(v);
        } return true;
}
```

2.6 Hledání maximálního toku

Algoritmus funguje na principu iterace. Pro konvergenci algoritmu musí být hrany oceněny celými čísly. Hledáme zlepšující cestu ze zdroje ke spotřebiči takovou, která nám umožní přenést větší kapacitu materiálu. Pokud algoritmus nenajde zlepšující cestu zmanemá to, že již máme řešení.

Další variantou může být síť se zápornými kapacitami, ty nahradíme opačne orientovanými hranami.

Případné neorientované hrany zorientujeme tak, že přidáme dvě obousměrné orientované hrany se stejnou kapacitou jako původní.

2.6.1 Algoritmus Forda Fulkersona

```
int Ford-Fulkerson(Graph g, Pricing p) {
        for(Edge (u,v) in H(G))
            f(u,v) = 0;
        while((S = improvingFlow(G, f, p)) not null)
            increaseFlow(G, S, f);
        return f;

}
```

2.7 Hledání cirkulace

Každý hranový řez sítí musí mít nezápornou kapacitu, to je nutnou podmínkou pro to, aby to co přiteče mohlo také odtéct. Algoritmus hledá nepřípustné toky a opravuje je tak, aby byla cirkulace přípustná. Pokud je každý tok přístupný algoritmus končí.

Hledáme zlepšující cestu tak, aby nová cirkulace splňovala minimální tok i kapacitu.

Obsah