Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

# DIPLOMOVÁ PRÁCE

*Martin Mareš*

## Dynamické grafové algoritmy

Katedra teoretické informatiky

Vedoucí diplomové práce: Mgr. Vladan Majerech, Dr.
Studijní program: informatika

Praha 2000

*Děkuji všem, kteří mne při psaní této práce podpořili, zejména pak Dr. Majerechovi za odborné vedení a povzbuzování a Zuzaně za trpělivost. Mé díky rovněž patří prof. Knuthovi za typografický systém TEX a Otfriedu Schwarzkopfovi za grafický editor IPE.*

*Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.*

*V Praze dne 6. 8. 2000* *Martin Mareš*

# Obsah

# 0. Úvod

Ověřování souvislosti a hranové respektive vrcholové $k$-souvislosti neorientovaných grafů, jakož i rozklad grafů na komponenty souvislosti a bloky patří mezi klasické problémy studované teorií grafových algoritmů již od jejího vzniku. Každý problém tohoto typu je možno řešit dvěma způsoby:

- *staticky,* to jest sestrojit algoritmus, který dostane na vstupu graf a jeho výstupem je řešení příslušného problému pro tento graf, u problému souvislosti tedy například rozklad zadaného grafu na komponenty souvislosti.
- *dynamicky,* to znamená sestrojit dynamickou datovou strukturu, která reprezentuje graf a umožňuje jednak tento graf modifikovat (obvykle přidávat a odebírat jednotlivé hrany) a jednak odpovídat na dotazy na řešení příslušného problému pro právě reprezentovaný graf, tedy například zda dané 2 vrcholy patří do téže komponenty souvislosti či nikoliv.

  Tento přístup je výhodný zejména tehdy, máme-li daný problém řešit pro posloupnost grafů lišících se navzájem pouze v malém počtu hran. Tehdy můžeme nahradit opakované spouštění statického algoritmu postupným modifikováním grafu reprezentovaného algoritmem dynamickým. K tomu nalezneme příležitost jednak v podprogramech složitějších algoritmů, jako jsou například toky v sítích či optimalizace návrhu integrovaných obvodů, a jednak v online aplikacích, ve kterých pracujeme s grafy popisujícími postupně se měnící stav reality, tedy například u dynamického směrování v počítačových sítích.

Pro statický případ souvislosti i 2-souvislosti jsou již mnoho let známy optimální algoritmy pracující v lineárním čase, obecně pro $k$-souvislost je situace obtížnější, ale i zde byla objevena efektivní řešení. Přehled dosažených výsledků lze nalézt v libovolné monografii o grafových algoritmech, za všechny jmenujme například [K83]. Rovněž byly objeveny paralelní algoritmy pro souvislost [SV82] a 2-souvislost [TV85] s polylogaritmickou* časovou složitostí.

Situace na poli dynamických algoritmů byla donedávna mnohem méně příznivá. Obecné techniky dynamizace datových struktur popsané Mehlhornem v [M84b] bohužel není možno použít, jelikož $k$-propojenost není rozložitelnou vlastností. Prvním výsledkem nevyžadujícím lineární čas na každou operaci byl Fredericksonův dynamický algoritmus pro souvislost s časovou složitostí $O(\sqrt{m})$ na operaci publikovaný v roce 1985 v [F85], následovaný v roce 1991 stejně rychlým algoritmem pro hranovou 2-souvislost popsaným v [F97]. Oba algoritmy se posléze Eppsteinovi, Galilovi, Italianovi a Nissenzweigovi zdařilo použitím sparsifikace (viz [EGI92] a [EGI93]) zrychlit na $O(\sqrt{n})$. Na konci této řady „odmocninových“ řešení stojí algoritmus pro dynamickou souvislost pracující v čase $O(\sqrt[3]{n} \cdot \log n)$ objevený Monikou Henzinger a Valerií King v roce 1997 [HK97a].

Ve světě dynamických datových struktur jsou ovšem obvyklé spíše logaritmické, resp. polylogaritmické časové složitosti, většina odborníků se proto shoduje na tom, že odmocninové dynamické algoritmy jsou pouze přechodným vývojovým stupněm na cestě k polylogaritmickému řešení, které je v mnohých pracích uváděno jako otevřený problém. Pro randomizované algoritmy se tohoto cíle podařilo dosáhnout pro souvislost v roce 1995 opět Monice Henzinger a Valerii King [HK95] (jejich algoritmus posléze zrychlil Mikkel Thorup na $O(\log^2 n)$ [HT96]), pro hranovou 2-souvislost těmže autorkám v roce 1997 [HK97b].

* tedy polynomiální v logaritmu velikosti vstupu

Cílem této práce je prozkoumat a popsat deterministické dynamické grafové algoritmy pro souvislost a hranovou 2-souvislost a pokusit se nalézt polylogaritmický algoritmus pro hranovou 2-souvislost. To učiníme tak, že nejprve popíšeme polylogaritmické datové struktury pro udržování stromů a operace na nich, pomocí nich odvodíme některé semi-dynamické algoritmy a posléze tyto algoritmy zobecníme na plně dynamické převodem reprezentace nestromových hran na reprezentaci bodů v rovině. Vzhledem k tomuto cíli některé datové struktury a jejich rozbor zjednodušíme a naopak jiné doplníme a budeme je analyzovat detailněji. Rovněž budeme místo časové složitosti v nejhorším případě obvykle počítat složitost amortizovanou, což ovšem vzhledem k aplikacím našich algoritmů jako podprogramy není nikterak na škodu.

Během tvorby této práce se ukázalo, že hledané polylogaritmické algoritmy byly mezi tím objeveny Mikkelem Thorupem, Jacobem Holmem a Kristianem De Lichtenbergem z university v Kodani. Jejich řešení, doposud publikované pouze jako technické zprávy kodaňské university [HLT97a] a [HLT97b], je uvedeno v poslední kapitole. Tato práce nicméně ukazuje alternativní přístup k problému a demonstruje mnoho technik, které mohou posloužit k budování polylogaritmických algoritmů pro další grafové problémy.

# 1. Definice

V této kapitole zavedeme pojmy souvislosti, násobné souvislosti, minimální kostry a dynamického grafového algoritmu a vyslovíme několik jednoduchých charakterizačních lemmat. Neuvedeme-li jinak, budeme grafem nazývat vždy neorientovaný graf bez smyček a násobných hran.

## 1.1. Souvislost

**Definice 1.1.1:** *Buď $G = (V, E)$ graf. Potom definujeme, že:*

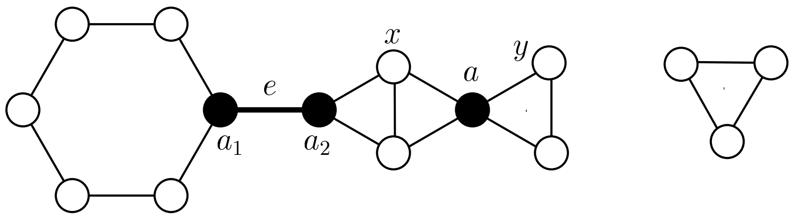
- *Vrcholy $v$ a $w$ jsou* ***propojeny*** $\equiv$ *existuje cesta v $G$ mezi $v$ a $w$.*
- *Vrcholy $v$ a $w$ jsou* ***vrcholově $k$-propojeny*** $\equiv$ *$v$ a $w$ jsou propojeny, odstraníme-li z $G$ libovolných nejvýše $k-1$ vrcholů různých od $v$ a $w$ a s nimi incidentní hrany.*
- *Vrcholy $v$ a $w$ jsou* ***hranově $k$-propojeny*** $\equiv$ *$v$ a $w$ jsou propojeny, odstraníme-li z $G$ libovolných nejvýše $k-1$ hran.*
- *$G$ je* ***souvislý*** $\equiv$ *libovolné dva vrcholy $G$ jsou propojeny.*
- *$G$ je* ***$k$-souvislý*** *(vrcholově resp. hranově)* $\equiv$ *$|V| > k$ a libovolné dva vrcholy jsou (vrcholově resp. hranově) $k$-propojeny.*

**Pozorování:** 1-propojenost je totéž co propojenost, 1-souvislost (pro neprázdné grafy) totéž co souvislost. Jsou-li vrcholy $v$, $w$ (vrcholově | hranově) $k$-propojeny, pak jsou také (vrcholově | hranově) $l$-propojeny pro libovolné $0 < l < k$. Jsou-li vrcholy $v$, $w$ vrcholově $k$-propojeny, jsou i hranově $k$-propojeny (místo každé hrany odebereme jeden z jejích krajních vrcholů). Totéž platí pro $k$-souvislost.

**Úmluva:** Nadále se omezíme pouze na souvislost a 2-souvislost. Neuvedeme-li, o jakou 2-souvislost (resp. 2-propojenost) se jedná, budeme vždy předpokládat hranovou.

Budeme studovat následující relace a jejich vlastnosti:

- $v \leftrightarrow w \equiv$ vrcholy $v$ a $w$ jsou propojeny.
- $v \rightleftarrows w \equiv$ vrcholy $v$ a $w$ jsou hranově 2-propojeny.
- $v \leftrightsquigarrow w \equiv$ vrcholy $v$ a $w$ jsou vrcholově 2-propojeny.




Obr. 1: 2-souvislost, mosty a artikulace

**Lemma 1.1.2:** *Relace $\leftrightarrow$ a $\rightleftarrows$ jsou ekvivalence.*

**Důkaz:** $\leftrightarrow$ je z definice reflexivní a symetrická, stačí tedy ověřit transitivitu. Pokud $u \leftrightarrow v$ a $v \leftrightarrow w$, existují cesty $P = \langle u = p_1, \ldots, p_m = v \rangle$ a $Q = \langle v = q_1, \ldots, q_n = w \rangle$. Buď $x$ poslední vrchol cesty $Q$, který se vyskytuje i v $P$, $x = p_i = q_j$. Potom cesta

$$\big\langle u = p_1, \ldots, p_{i-1}, p_i = q_j, q_{j+1}, \ldots, q_n = w \big\rangle$$

spojuje $u$ a $w$, a proto $u \leftrightarrow w$.

Reflexivita a symetrie $\rightleftarrows$ plyne z reflexivity a symetrie $\leftrightarrow$, zbývá opět transitivita: Nechť $u \rightleftarrows v$ a $v \rightleftarrows w$. Pak zvolíme-li si libovolnou hranu $e$, v grafu $G' = G - e$* bude (z definice 2-propojenosti) stále $u \leftrightarrow v$ i $v \leftrightarrow w$, tím pádem i (díky transitivitě $\leftrightarrow$) $u \leftrightarrow w$. $\square$

**Definice 1.1.3:**

- *Ekvivalence $\leftrightarrow$ a $\rightleftarrows$ určují rozklady vrcholů grafu $G$ na třídy. Těmto třídám budeme říkat* ***komponenty souvislosti*** *(pro $\leftrightarrow$) a* ***komponenty 2-souvislosti*** *(pro $\rightleftarrows$). Pokud bude jasné, že hovoříme o souvislosti, resp. 2-souvislosti, budeme je nazývat krátce* ***komponenty****, resp.* ***2-komponenty****.*
- *Pokud vrcholy $v$ a $w$ z téže komponenty nejsou 2-propojeny, existuje hrana $e$ taková, že v $G - e$ není $v \leftrightarrow w$; takovou hranu nazveme* ***mostem*** *oddělujícím $v$ od $w$.*
- *Analogicky pro vrcholovou 2-propojenost: Pokud $v \leftrightarrow w$ a $v \not\leftrightsquigarrow w$, existuje vrchol $x$, jehož odstraněním přestane být $v \leftrightarrow w$; tento vrchol nazveme* ***artikulací*** *oddělující $v$ od $w$.*

V grafu na obrázku 1 jsou dvě komponenty souvislosti, levá z nich je rozdělena mostem $e$ na dvě komponenty hranové 2-souvislosti. Černé vrcholy $a$, $a_1$ a $a_2$ jsou artikulacemi grafu.

**Pozorování:** Komponenty souvislosti jsou souvislé podgrafy, komponenty hranové 2-souvislosti jsou hranově 2-souvislé podgrafy. Vrcholová 2-souvislost žádný takový rozklad vrcholů na komponenty neurčuje, jelikož relace $\leftrightsquigarrow$ není ekvivalencí (není totiž transitivní – v příkladu na obr. 1 jsou vrcholy $x$ a $a$ 2-propojené, $a$ a $y$ taktéž, ale $x$ a $y$ nikoliv, protože $a$ je artikulací, která je odděluje).

**Definice 1.1.4:** *Podgraf $F$ grafu $G$ nazveme* ***kostrou*** *$G$, pokud je acyklický a relace propojenosti $\leftrightarrow_F$ a $\leftrightarrow_G$ si jsou rovny. Jinými slovy, $F$ je les, jehož komponenty souvislosti obsahují tytéž vrcholy jako komponenty souvislosti grafu $G$. Hrany kostry budeme nazývat* ***stromovými****, ostatní hrany grafu* ***nestromovými****.*

**Lemma 1.1.5:** *Následující tvrzení o grafu $G = (V, E)$ na alespoň dvou vrcholech jsou ekvivalentní:*

*a) $G$ je 2-souvislý.*
*b) $G$ je souvislý a každá hrana $e \in E$ leží na nějaké kružnici.*
*c) Pro každé 2 vrcholy $v, w \in V$ existuje uzavřený tah, který je obsahuje. (analogie Mengerovy věty pro hranovou 2-souvislost)*

**Důkaz:** (a) $\Longrightarrow$(b): Mějme libovolnou hranu $e = \{v, w\}$. Díky 2-souvislosti $G$ musí existovat v $G - e$ cesta $P$ mezi $v$ a $w$. Tato cesta spolu s hranou $e$ tvoří kružnici v $G$.

(b) $\Longrightarrow$(c): Indukcí podle vzdálenosti $d(v, w)$. Pokud $d(v, w) = 1$, vede mezi $v$ a $w$ hrana, ta leží podle (b) na kružnici a každá kružnice je uzavřeným tahem. Indukční krok: Pokud již tvrzení platí pro $d(v, w) < k$ a máme jej dokázat pro $d(v, w) = k$, vezměme předposlední vrchol $x$ na nejkratší cestě z $v$ do $w$. Pak $d(v, x) = k - 1$ a podle indukčního přepokladu existuje uzavřený tah $T$, na kterém leží $v$ i $x$. Hrana $\{x, w\}$ leží podle (b) na nějaké kružnici $\langle x = x_0, w = x_1, x_2, \ldots, x_{n-1}, x_n = x\rangle$. Vezměme nyní $i > 0$ nejmenší takové, že $x_i$ leží na tahu $T$. Pak lze $T$ rozdělit na otevřené tahy $T_0$ (z $x$ do $x_i$) a $T_1$ (z $x_i$ zpět do $x$) a na jednom z nich (BÚNO† na $T_0$) leží $v$. Potom $T_0 \cup \langle x_i, x_{i-1}, \ldots, x_2, w, x\rangle$ je uzavřený tah, který obsahuje jak $v$, tak $w$.

* $G$ po odebrání hrany $e$

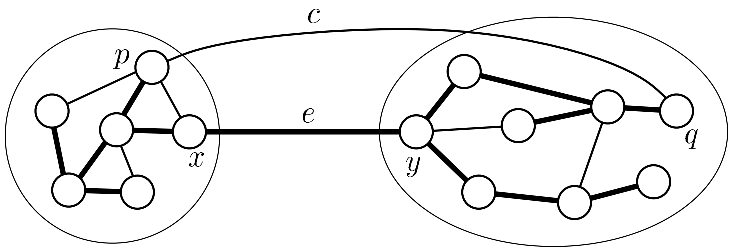
† Bez Újmy Na Obecnosti

(c) $\Longrightarrow$(a): Pokud $v$ a $w$ leží na společném uzavřeném tahu $T$, pak leží i na společné cestě, takže $G$ je souvislý. Odstraníme-li nyní z $G$ libovolnou hranu $e$, pak buďto $e \notin T$ a tím pádem $T$ spojuje $v$ a $w$ i v $G - e$ a nebo $e \in T$, takže $T - e$ je otevřený tah na téže množině vrcholů, tudíž opět spojuje $v$ a $w$. $\square$

**Pozorování:** Zvolíme-li si libovolnou kostru $T$ grafu $G$, musí všechny mosty $G$ ležet v této kostře. To, které stromové hrany jsou mosty a které nikoliv, charakterizuje následující lemma.

**Lemma 1.1.6:** *Buď $G = (V, E)$ graf a $T$ jeho libovolná kostra. Pak hrana $e \in E$ je mostem v $G$ právě tehdy, když $e \in T$ a neexistuje hrana $c \in E - T$ spojující obě komponenty souvislosti grafu $T - e$ (o takové hraně říkáme, že hranu $e$ **pokrývá**). (Jinými slovy: $G$ je 2-souvislý $\Longleftrightarrow$ $G$ je souvislý a každá hrana $T$ je pokryta.)*

**Důkaz:** $\Longrightarrow$: Pokud je $e$ mostem, musí být nutně součástí každé kostry (kdyby nebyl, jsou krajní vrcholy $e$ spojeny v $G - e$ cestou v kostře), tedy i $T$. $G - e$ je nesouvislý a $T - e$ je jeho kostrou, tím pádem nemůže existovat žádná hrana v $G - T \subseteq G - e$, která by spojovala komponenty $T - e$, a tak pokryla $e$.




Obr. 2: Hrana $e$ je pokryta hranou $c$, $\langle x, \dots, p, q, \dots, y, x \rangle$ je kružnice.

$\Longleftarrow$: Každá hrana $e \in E$ leží na kružnici tvořené hranou $c$, která pokrývá $e$, a cestou v $T$ mezi krajními vrcholy $c$. Proto $e$ nemůže být mostem. Situace je vyznačena na obr. 2: hrany kostry jsou vysázeny tučně, hrana $c = \{p, q\}$ spolu s cestou v $T$ mezi $p$ a $q$ dávají kružnici obsahující hranu $e = \{x, y\}$. $\square$

## 1.2. Minimální kostry

**Definice 1.2.1:** *Buď $G = (V, E)$ graf, $c : E \to \mathbb{R}$ jeho hranové ohodnocení a $F$ jeho kostra. **Cenou** kostry $F$ budeme nazývat součet ohodnocení všech v ní obsažených hran, tedy $c(F) = \sum_{e \in E(F)} c(e)$. Kostru $G$ s nejmenší možnou cenou nazveme **minimální**, jinak též **nejlevnější** kostrou.*

**Lemma 1.2.2:** *Kostra $T$ grafu $G$ s hranovým ohodnocením $c : E(G) \to \mathbb{R}$ je minimální právě tehdy, když pro každou hranu $\{v, w\} = e \in E(G) - E(T)$ platí, že $c(e)$ není menší než ceny všech hran na cestě mezi $v$ a $w$ v $T$.*

**Důkaz:** $\Longrightarrow$: Pokud by existovala nějaká hrana $e = \{v, w\}$ mimo kostru a nějaká hrana $e'$ ležící na cestě v kostře mezi $v$ a $w$ a $c(e) < c(e')$, získali bychom nahrazením $e'$ hranou $e$ kostru s menší cenou, což je spor s minimalitou $T$.

$\Longleftarrow$: Nechť kostra $T$ splňuje naši podmínku (tu si označíme $(*)$) a $\check{T}$ je libovolná z minimálních koster. Dokážeme, že $c(T) \leq c(\check{T})$, a to postupným upravováním kostry $\check{T}$ na $T$, při kterém nebude $c(\check{T})$ stoupat: $T$ a $\check{T}$ jsou kostry, takže mají stejný počet hran, a tak je-li $T \neq \check{T}$, existuje nutně hrana $\{v, w\} = e \in E(T) - E(\check{T})$. Pak označíme $\check{e}$ některou z hran na cestě mezi $v$ a $w$ v $\check{T}$, která není v $T$ (kdyby žádná taková hrana neexistovala, tvořila by tato cesta spolu s $e$ kružnici v $T$). Podle $(*)$ je $c(\check{e}) \geq c(e)$, takže nahradíme-li v $\check{T}$ hranu $\check{e}$ hranou $e$, získáme opět kostru a $c(\check{T})$ nevzroste. Navíc se velikost symetrické diference $|E(T) \triangle E(\check{T})|$ zmenší o 2, tudíž po konečném počtu kroků dokončíme kýženou transformaci. □

## 1.3. Dynamické grafové algoritmy

**Definice 1.3.1:** ***Dynamickým grafem*** *nazveme dynamickou datovou strukturu, která reprezentuje neorientovaný graf modifikovaný posloupností některých z následujících operací:*

- *Build*(**graph** $G$) – *inicializuje strukturu tak, aby reprezentovala daný graf $G$.*
- *Insert*(**edge** $e$) – *vloží hranu $e$ do grafu.*
- *Delete*(**edge** $e$) – *odstraní hranu $e$ z grafu.*
- *Backtrack – odstraní hranu, která byla vložena jako poslední a nebyla dosud smazána.*

**Definice 1.3.2:** ***Dynamickým grafovým algoritmem pro $k$-souvislost*** *nazveme dynamický graf, na němž je mezi jednotlivými modifikacemi struktury možno provádět navíc následující operace:*

- **boolean** *Query*(**vertex** $x$, $y$) – *testuje, zda jsou vrcholy $x$ a $y$ v reprezentovaném grafu $k$-propojeny.*
- **edge** *Bridge*(**vertex** $x$, $y$) – *(pouze pro hranovou 2-souvislost) pokud nejsou vrcholy $x$ a $y$ 2-propojeny, vrátí některý z mostů, které je oddělují.*
- **vertex** *Art*(**vertex** $x$, $y$) – *(pouze pro vrcholovou 2-souvislost) pokud nejsou vrcholy $x$ a $y$ vrcholově 2-propojeny, vrátí některou z artikulací, které je oddělují.*

**Definice 1.3.3:** ***Dynamickým grafovým algoritmem pro udržování kostry*** *nazveme dynamický graf, který navíc udržuje svoji kostru a na každou operaci Insert, Delete, resp. Backtrack odpoví posloupností $O(1)$ změn kostry (Insert resp. Delete hrany v kostře).*

**Úmluva:** Při vyjadřování časové složitosti operací grafového algoritmu bude $n$ vždy značit počet vrcholů grafu a $m$ počet jeho hran.

# 2. Dynamické stromy

Pro mnohé grafové problémy na stromech (resp. lesích) jsou známy efektivní dynamické algoritmy, které mnohdy slouží jako základ algoritmů pro obecné grafy, a to jak algoritmů dynamických (jak uvidíme v následujících kapitolách), tak i statických, jako je například algoritmus pro hledání maximálního toku v řídké síti uvedený v [ST83].

Nejznámější dynamickou reprezentací lesů jsou Sleator-Tarjanovy dynamické stromy zavedené v [ST83], které budou popsány v této kapitole. Umožňují udržovat komponenty souvislosti a ceny hran, počítat celkové ceny cest mezi zadanými dvěma vrcholy či vyhledat nejdražší hranu na takové cestě atd., to vše v amortizovaném čase $O(\log n)$.

**Definice 2.1:** *Sleator-Tarjanův* ***dynamický strom*** *je dynamická datová struktura reprezentující les složený s disjunktních zakořeněných stromů, jejichž hrany jsou ohodnoceny reálnými čísly (cenami hran). Na dynamických stromech je možno provádět následující operace:*

- **vertex** *Parent*(**vertex** $v$) – *vrátí otce vrcholu $v$. Pokud otec neexistuje ($v$ je kořenem), vrátí speciální hodnotu* **null**.
- **vertex** *Root*(**vertex** $v$) – *vrátí kořen stromu obsahujícího vrchol $v$.*
- **real** *Cost*(**vertex** $v$) – *pro vrchol $v$, který není kořenem, vrátí cenu hrany vedoucí z $v$ do jeho otce.*
- **vertex** *Mincost*(**vertex** $v$) – *pro vrchol $v$, který není kořenem, vrátí vrchol $w$, který je nejblíže k $Root(v)$ takový, že hrana $(w, Parent(w))$ je nejlevnější na cestě z $v$ do kořene.*
- *Update*(**vertex** $v$, **real** $x$) – *přičte $x$ k cenám všech hran ležících na cestě z vrcholu $v$ do kořene.*
- *Link*(**vertex** $v$, $w$, **real** $x$) – *spojí stromy s kořeny $v$ a $w$ do jediného stromu s kořenem $v$ přidáním hrany $(v, w)$ o ceně $x$.*
- **real** *Cut*(**vertex** $v$) – *rozdělí strom obsahující vrchol $v$, který není kořenem, odstraněním hrany $(v, Parent(v))$ a vrátí cenu této hrany.*
- **vertex** *Evert*(**vertex** $v$) – *učiní $v$ kořenem stromu, a to tak, že otočí orientaci všech hran na cestě mezi vrcholem $v$ a původním kořenem. Vrátí původní kořen.*
- **list** *WalkPath*(**vertex** $v$, $w$) – *vrátí seznam obsahující všechny vrcholy na cestě mezi $v$ a $w$.*

## 2.1. Reprezentace cest

Nejprve vyřešíme jednodušší případ, a to dynamickou reprezentaci grafů, jejichž všechny komponenty souvislosti jsou orientované cesty. Tyto grafy budeme ukládat jako množinu splay-stromů, každý splay-strom bude obsahovat informace o jedné cestě, a to tak, že jeho listy v pořadí zleva doprava budou odpovídat vrcholům cesty seřazeným od začátku do konce cesty a vnitřní vrcholy v inorder pořadí hranám cesty opět seřazeným od první na cestě k poslední.

Všechny vrcholy splay-stromu obsahují položky *parent*, *lson* a *rson* obsahující ukazatel na otce, levého syna a pravého syna vrcholu či speciální hodnotu **null**, pokud příslušný vrchol neexistuje. Každý vnitřní vrchol $v$ reprezentující hranu $e$ obsahuje navíc:

- *bhead*, *btail* – ukazatele na nejlevější a nejpravější list v podstromu určeném vrcholem $v$.
- *grosscost* – cena hrany $e$.

- *grossmin* – minimum z hodnot *grosscost* přiřazeným všem vnitřním vrcholům tohoto podstromu.
- *reversed* – bit udávající, zda je v celém podstromu určeném vrcholem $v$ prohozena levá a pravá strana (toho využíváme k efektivnímu obracení úseků cesty). Položka *lson* tedy odpovídá skutečnému levému synovi a *rson* pravému právě tehdy, je-li na cestě od tohoto vrcholu ke kořeni sudý počet vrcholů s bitem *reversed* nastaveným. Totéž platí pro *bhead* a *btail*.

Hodnoty *grosscost* a *grossmin* nebudeme mít ve vrcholech uloženy přímo, jelikož by bylo obtížné je udržovat. Nahradíme je relativními hodnotami:

- $netcost = grosscost - grossmin$ (nezáporné číslo udávající, o kolik je hrana $e$ dražší než nejlevnější hrana na úseku cesty reprezentovaném podstromem)
- $netmin = \begin{cases} grossmin & \text{pokud } v \text{ je kořen splay-stromu} \\ grossmin - grossmin(w) & \text{pokud } w \text{ je otcem } v \text{ ve splay-stromu} \end{cases}$

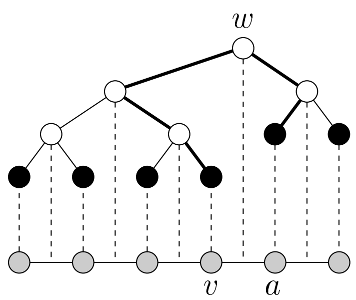
  (to jest o kolik je *grossmin* vyšší než *grossmin* otce, což musí pro každý vrchol kromě kořene být nezáporné číslo)

Hodnotu *grossmin* můžeme nyní rekonstruovat sečtením všech *netmin* na cestě od $v$ ke kořeni, *grosscost* pak sečtením *netcost* a *grossmin*.

Jelikož všechny změny struktury stromu jsou realizovány přidáváním hran, ubíráním hran a vynášením vrcholů do kořene (operace *Splay* tak, jak se standardně pro splay-stromy definuje), stačí, abychom dokázali informace uložené ve vrcholech udržovat při těchto operacích, což je snadné, neboť *bhead*, *btail* a *netmin* můžeme kdykoliv rekonstruovat z hodnot uložených v synech vrcholu, *netcost* nejlépe převedeme na *grosscost* před započetím operace a po jejím ukončení jej spočítáme znovu. Bitu *reverse* není nutno věnovat žádnou zvláštní pozornost mimo toho, že si musíme pamatovat, zda jsou v daném vrcholu směry prohozeny či nikoliv. Všechny odhady časové složitosti operací budou vycházet z následující věty:

**Věta 2.1.1** *(Sleator & Tarjan)***:** *Amortizovaná časová složitost operace Splay ve splay-stromu je $O(\log n)$.*

**Důkaz:** Viz [ST83]. □




Obr. 3: Cesta a její reprezentace splay-stromem

Na této datové struktuře snadno implementujeme následující operace:

- **path** *Path*(**vertex** $v$) – vrátí identifikátor cesty, na níž vrchol $v$ leží (provedeme-li libovolnou operaci, která mění strukturu stromu, nejen váhy hran, identifikátory se mohou změnit). To učiníme tak, že vystoupáme z $v$ do kořene stromu a prohlásíme za identifikátor tento kořen, což zvládneme v čase $O(d(v))$, kde $d(v)$ je hloubka vrcholu $v$.

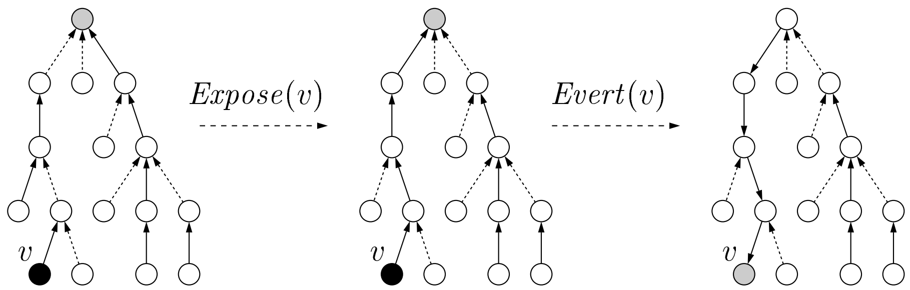
- **vertex** *Head*(**path** $p$) – vrátí počáteční vrchol cesty $p$. Jelikož cestu identifikujeme kořenem splay-stromu, který ji reprezentuje, stačí vrátit buďto $bhead(p)$ nebo $btail(p)$ podle toho, zda je bit $reversed(p)$ nastaven či snulován. Časová složitost: konstantní.
- **vertex** *Tail*(**path** $p$) – vrátí koncový vrchol cesty $p$. Symetrické s *Head*.
- **vertex** *After*(**vertex** $v$) – vrátí vrchol $a$ ležící za $v$ na příslušné cestě (předpokládá, že $v$ není koncový vrchol). Vystoupí po stromu z $v$ do kořene, zapamatuje si všechny vrcholy na této cestě, spočte, v kterých z nich jsou prohozeny směry, a nalezne nejhlubší vrchol $w$ takový, že do něj nalezená cesta přichází zleva. Tento vrchol reprezentuje hranu $(v, a)$ – viz obrázek 3. Nejlevější list v podstromu připojeném k $w$ zprava (ten nalezneme pomocí *bhead* či *btail* v konstantním čase) je hledaný následník vrcholu $v$. Časová složitost této operace je $O(d(v))$.
- **vertex** *Before*(**vertex** $v$) – vrátí vrchol ležící před $v$, není-li $v$ počáteční vrchol cesty. Symetrické s *After*.
- **real** *Pcost*(**vertex** $v$) – vrátí cenu hrany $(v, \mathit{After}(v))$, existuje-li taková hrana. Vystoupí z vrcholu $v$ do kořene a pro všechny vrcholy na této cestě spočte prohození směrů a *grossmin*. Stejně jako u *After* nalezne vrchol $w$ reprezentující hranu $(v, \mathit{After}(v))$. Nyní již stačí vrátit *grosscost* tohoto vrcholu, což je součet *netcost* a vypočteného *grossmin*. Časová složitost: $O(d(v))$.
- **vertex** *Pmincost*(**path** $p$) – vrátí vrchol $v$ nejbližší k $\mathit{Tail}(p)$ takový, že hrana $(v, \mathit{After}(v))$ je nejlevnější na cestě $p$ (předpokládá, že cesta $p$ obsahuje více jak jeden vrchol). Začneme v kořeni $p$, spočteme *grossmin* a *grosscost* pro tento vrchol a oba jeho syny (levého $l$ a pravého $r$; pokud je libovolný ze synů listem, považujeme jeho hodnoty za nekonečně velké). Pokud $grossmin(p) = grossmin(r)$, vyskytuje se některá z nejlevnějších hran v podstromu určeném pravým synem a jelikož nás zajímá na cestě poslední výskyt takové hrany, pokračujeme stejným způsobem v tomto podstromu, pouze případně prohodíme směry. Pokud tomu tak není a $grossmin(p) = grosscost(p)$, je právě hrana reprezentovaná vrcholem $p$ poslední z nejlevnějších hran na cestě $p$. Ve zbývajících případech se hledaná hrana vyskytuje v levém podstromu. Časová složitost: $O(d(v))$.
- *Pupdate*(**path** $p$, **real** $x$) – přičte $x$ k cenám všech hran na cestě $p$. Stačí přičíst $x$ k $netmin(p)$, čímž se implicitně zvýší ceny všech hran na cestě, jakož i všechna vypočtená minima. Časová složitost: konstantní.
- *Reverse*(**path** $p$) – obrátí směr cesty $p$. Stačí znegovat bit $reversed(p)$, a to zvládneme opět v konstantním čase.
- **path** *Concatenate*(**path** $p$, $q$, **real** $x$) – připojí cestu $q$ za cestu $p$ hranou o ceně $x$ a vrátí identifikátor nově vzniklé cesty. Vytvoří nový vrchol, učiní $p$ jeho levým synem a $q$ pravým, nastaví jeho *netmin* na minimum z *netmin* synů a $x$, *netcost* na $x - netmin$, *netmin* synů sníží o nově vypočtený *netmin* otce a vrátí nově vytrořený vrchol. Čas: konstantní.
- **list** *Split*(**vertex** $v$) – odstraní hrany incidentní s vrcholem $v$, a tak rozdělí cestu obsahující $v$ na až tři části. Vrací seznam $[p, q, x, y]$, kde $p$ je podcesta vedoucí do předchůdce $v$, $q$ podcesta pokračující z následníka $v$, $x$ cena hrany, která vedla do $v$ a $y$ cena hrany původně vedoucí z $v$ (pokud libovolná z těchto hodnot nemá smysl, použije se místo ní konstanta **null**). Operace *Split* funguje tak, že stejně jako $\mathit{After}(v)$ nalezne vrchol $w$ odpovídající hraně z $v$ do jeho následníka, tento vrchol pomocí operace $\mathit{Splay}(w)$ vynese do kořene stromu a

odebere jej, čímž se strom rozpadne na levý podstrom, který reprezentuje cestu od počátku až k vrcholu $v$, a pravý podstrom odpovídající cestě $q$; *grosscost* odebraného vrcholu bude přesně hledaná cena $y$. Poté symetricky nalezne hranu z předchůdce $v$ do $v$ a jejím vysplayováním a odebráním získá $p$ a $x$. Časová složitost: $O(d(v) + \log n)$.

- **list** *Enumerate*(**path** $p$) – sestrojí seznam obsahující všechny vrcholy na cestě $p$, a to tak, že projde splay-strom reprezentující tuto cestu do hloubky, což stihne v čase $O(|p|)$, tedy lineárně s velikostí výsledku.

Všechny operace, které mají časovou složitost lineární v hloubce vrcholu, se kterým pracují (tedy $O(d(v))$), je možno upravit přidáním $Splay(parent(v))$ (místo $v$ splayujeme jeho otce, protože $v$ může být listem stromu a my musíme zajistit, aby listy zůstaly listy). Tím se celkový čas zvětší o trvání operace *Splay*, které je též $O(d(v))$, ale jelikož z předchozí věty víme, že *Splay* amortizovaně trvá čas $O(\log n)$, musí i celá operace trvat $O(\log n)$.

## 2.2. Reprezentace stromů




Obr. 4: Rozdělení hran na plné a čárkované, operace *Expose* a *Evert*

Nyní popis cest rozšíříme na popis obecných zakořeněných stromů. Hrany stromu zorientujeme směrem ke kořeni a rozdělíme je na plné a čárkované tak, aby do každého vrcholu vedla nejvýše jedna plná hrana (viz obr. 4). Tím se strom rozpadne na orientované cesty složené z plných hran (těm budeme říkat plné cesty), které jsou navzájem propojeny čárkovanými hranami. Každá čárkovaná hrana vede z koncového vrcholu jedné cesty do nějakého vrcholu cesty jiné. Cesty uložíme, jak jsme již popsali, jako splay-stromy, navíc si u každého koncového vrcholu cesty (což je list splay-stromu) budeme pamatovat, zda z něj vede nějaká čárkovaná hrana a pokud ano, tak kam (*dparent*) a jakou má cenu (*dcost*).

Abychom mohli s touto datovou strukturou pracovat, aniž bychom se museli zajímat o to, které hrany jsme si zvolili jako čárkované a které jako plné, zavedeme nejprve čtyři základní operace pracující s propojením, označením a orientací hran. Tyto operace jsou vyznačeny na obr. 4, šedě je obarven kořen stromu, černě vrchol, se kterým se operace provádí.

- *Expose*(**vertex** $v$) – „odkryje“ vrchol, tj. přeznačí hrany tak, aby se cesta z vrcholu $v$ do kořene stromu stala plnou cestou a všechny hrany incidentní s touto cestou čárkovanými. Funguje takto:

**function** *Expose*(**vertex** $v$)
    **path** $p, q, r$;     *lokální proměnné*
    **vertex** $w$;
    **real** $x, y$;
    $[p, q, x, y] \leftarrow Split(v)$;     *rozděl plnou cestu ve* $v$

```
    if p ≠ null →                              část pod v připoj čárkovaně
        dparent(Tail(p)) ← v;
        dcost(Tail(p)) ← x;
    p ← Path(v);                               p je 1-vrcholová cesta obsahující v
    if q ≠ null →                              obnov případnou plnou hranu z v
        p ← Concatenate(p, q, y);              v nyní leží na plné cestě p
    while dparent(Tail(p)) ≠ null do:          dokud je nad v čárkovaná hrana:
        w ← dparent(Tail(p));                  w je horní konec této hrany
        [r, q, x, y] ← Split(w);               rozděl plnou cestu obsahující w
        if r ≠ null →                          spodní část připoj čárkovaně k w
            dparent(Tail(r)) ← w;
            dcost(Tail(r)) ← x;
        p ← Concatenate(p, q, y);              horní připoj plně k cestě p
end
```

- **vertex** *Evert*(**vertex** $v$) – „překoření“ strom, tj. obrátí orientace hran na cestě z kořene stromu do $v$, čímž se vrchol $v$ stane kořenem. Vrátí původní kořen stromu. Pracuje tak, že provede *Expose*($v$) a poté *Reverse*(*Path*($v$)).
- *Link*(**vertex** $v$, $w$, **real** $x$) – propojí dva stromy tak, že kořen stromu $v$ připojí hranou o ceně $x$ pod vrchol $w$ (přepokládá, že $v$ je kořen a $w$ je vrchol jiného stromu). Funguje tak, že pomocí *Expose*($w$) změní cestu z $w$ do kořene jeho stromu na plnou a poté tuto cestu připojí hranou o ceně $x$ za plnou cestu obsahující vrchol $v$.
- **real** *Cut*(**vertex** $v$) – rozdělí strom tak, že odstraní hranu vedoucí z vrcholu $v$, který není kořenem, do jeho otce a vrátí její cenu. Pomocí *Expose*(*Parent*($v$)) změní tuto hranu na čárkovanou, poté přenastaví *dparent*($v$) ← **null** a vrátí *dcost*($v$).

**Věta 2.2.1** (Sleator & Tarjan)**:** *Libovolná posloupnost $k$ operací Expose, Evert, Link a Cut, která začíná s lesem obsahujícím $n$ vrcholů a žádné hrany, trvá $O(k \cdot \log n)$, tedy amortizovaná složitost každé z těchto operací je $O(\log n)$.*

**Důkaz:** Viz [ST83]. □

Všechny ostatní operace na dynamických stromech lze již převést na *Expose*, *Evert* a operace na cestách:

- **vertex** *Parent*(**vertex** $v$) – Vrátí *After*($v$), pokud to není **null**; v opačném případě (to znamená, že $v$ je na konci nějaké plné cesty), vrátí *dparent*($v$). Celkový čas: $O(\log n)$.
- **vertex** *Root*(**vertex** $v$) – Provede *Expose*($v$) a poté vrátí *Tail*(*Path*($v$)). Časová složitost: $O(\log n)$.
- **real** *Cost*(**vertex** $v$) – Podobně jako *Parent*($v$), tj. pokud *After*($v$) ≠ **null**, vrátí *Pcost*($v$), jinak vrátí *dcost*($v$). Časová složitost: $O(\log n)$.
- **vertex** *Mincost*(**vertex** $v$) – *Expose*($v$) a poté vrátí *Pmincost*($v$), což zvládne v čase $O(\log n)$.
- *Update*(**vertex** $v$, **real** $x$) – *Expose*($v$) a poté *Pupdate*($v$, $x$), to celé opět v čase $O(\log n)$.
- **list** *WalkPath*(**vertex** $v$, $w$) – Pomocí *Evert*($w$) a *Expose*($v$) vytvoří plnou cestu z $v$ do $w$ a poté zavolá *Enumerate*(*Path*($v$)). Nakonec opětovným voláním *Evert* strom překoření do původního stavu. Celkový čas na provedení této ope-

race je $O(\log n + d(v,w))$, kde $d(v,w)$ je vzdálenost $v$ od $w$, tedy velikost vygenerovaného seznamu.

**Věta 2.2.2:** *Všechny operace na dynamických stromech mají amortizovanou časovou složitost $O(\log n + |\mathit{výstup}|)$. Dynamický strom v každém okamžiku zabírá prostor $O(n)$.*

**Důkaz:** Časová složitost plyne z věty 2.2.1 a z uvedené konstrukce zbývajících operací.

Co se prostorové složitosti týče, každá plná cesta $P$ zabírá prostor $O(|P|)$. Dynamický strom sestává se z $O(n)$ plných cest o celkové délce $n$ (čárkované hrany jsou uloženy v koncových vrcholech plných cest), tedy uložených v prostoru $O(n)$. $\square$

# 3. Semidynamické algoritmy

V této kapitole budeme studovat dynamické algoritmy kladoucí omezení na operace, které je s dynamickým grafem možno provádět, a to zejména na mazání hran. Takovým algoritmům se obvykle říká ***semidynamické***. Jsou to jednak ***inkrementální*** algoritmy, jež podporují pouze vkládání nových hran, což je ovšem vyváženo extrémně malou časovou složitostí operací *Insert* a *Query*, dále se budeme zabývat algoritmy umožňujícími *Backtrack* (mazání hran v opačném pořadí, než v jakém byly vkládány) a ***kvazidynamickými*** algoritmy, u nichž je možno mazat libovolné hrany ležící mimo kostru a také mosty.

## 3.1. Souvislost inkrementálně

Inkrementální udržování komponent souvislosti lze snadno redukovat na problém udržování ekvivalence, pro který byl již roku 1975 nalezen R. Tarjanem velice efektivní algoritmus [T75], o němž se později v [TL84] ukázalo, že jeho časovou složitost lze při provedení $k$ operací omezit funkcí $O(k \cdot \alpha(k,n))$, kde $\alpha(m,n)$ je inverzní funkce k Ackermannově funkci*.

**Definice 3.1.1:** ***Problém udržování ekvivalence*** *(zvaný též Union/Find problem) tkví v konstrukci dynamické datové struktury reprezentující ekvivalenci na $n$-prvkové množině a poskytující tyto operace:*

- *Init – inicializuje strukturu na triviální ekvivalenci (každý prvek je ekvivalentní pouze sám se sebou).*
- **class** *Find*(**element** $x$) *– nalezne třídu ekvivalence, v níž se nachází prvek $x$.*
- *Union*(**class** $x$, $y$) *– spojí dvě třídy ekvivalence v jednu.*

**Algoritmus 3.1.2** (*Udržování ekvivalence*)**:** *Každou ekvivalenční třídu budeme reprezentovat stromem orientovaným směrem ke kořeni, jehož vrcholy jsou jednotlivé prvky třídy a kořen slouží jako identifikátor třídy. Jelikož budeme po cestách procházet vždy směrem ke kořeni, stačí si u každého prvku $x$ pamatovat jeho otce $p(x)$ (***null***, je-li $x$ kořenem) a pro kořeny komponent navíc velikosti komponent $s(x)$.*

*Operace Init nastaví $p(x)$ všech prvků na* **null** *a velikost $s(x)$ na 1. Při sjednocování dvou tříd operací Union$(x, y)$ stačí přidat hranu připojující kořen menší komponenty pod kořen komponenty větší, tedy pro $s(x) \leq s(y)$ nastavit $p(x) \leftarrow y$ a zvýšit $s(y)$ o $s(x)$. Klíčovým místem pro dosažení kýžené efektivity je šikovná implementace operace Find$(x)$: nejprve projitím cesty z $x$ do kořene přes $p(x)$, $p(p(x))$ atd. nalezneme identifikátor komponenty a poté tuto cestu kontrahujeme, to znamená nastavíme $p(y)$ všech vrcholů $y$ na této cestě kromě kořene na kořen, čímž zrychlíme všechny následující Findy na dotyčné vrcholy.*

**Věta 3.1.3** (*Tarjan & van Leeuwen*)**:** *Libovolná posloupnost operací tvořená počátečním voláním Init následovaným $k$ operacemi Union a Find v libovolném pořadí je zpracována v čase $O(n + k \cdot \alpha(k,n))$ a prostoru $O(n)$.*

**Důkaz:** Viz [TL84]. □

Navíc můžeme dodefinovat operaci **integer** *Count*(**class** $x$), která vrátí $s(x)$, tedy velikost komponenty $x$, a to v konstantním čase.

---

* Inverze Ackermannovy funkce je pravděpodobně nejpomaleji rostoucí neomezenou funkcí, která se při analýze algoritmů vůbec vyskytuje. Její limita pro $n \to \infty$ je sice $\infty$, ale „pro libovolné $n$, které nějak souvisí s tímto vesmírem, je $\alpha(n,n) \leq 4$“. PVPÚ (pro všechny praktické účely) ji tak lze považovat za konstantu, i když jí samozřejmě není.

**Algoritmus 3.1.4** *(Inkrementální souvislost)*: *Budeme udržovat ekvivalenci $\leftrightarrow$ pomocí předchozí struktury a operace na grafu převedeme na práci s touto strukturou:*

- *Query*(**vertex** $v$, $w$) – *pro zjištění, zda jsou vrcholy $v$ a $w$ propojeny, stačí porovnat Find($v$) a Find($w$).*
- *Insert*(**vertex** $v$, $w$) – *nejprve nalezneme pomocí Find($v$) a Find($w$) třídy ekvivalence $\leftrightarrow$, do nichž vrcholy $v$ a $w$ patří (tedy vlastně příslušné komponenty souvislosti) a pokud se jedná o různé třídy, použitím Union je sloučíme do jedné.*

*Strukturu pro graf bez hran inicializujeme voláním Init, čímž vytvoříme triviální ekvivalenci odpovídající jednovrcholovým komponentám souvislosti.*

**Věta 3.1.5:** *Časová složitost algoritmu 3.1.4 při provedení posloupnosti $k$ operací počínajících grafem bez hran je $O(n + k \cdot \alpha(k, n))$, prostorová $O(n)$.*

**Důkaz:** Každý *Insert* či *Query* způsobí provedení $O(1)$ operací *Find* a *Union* na ekvivalenci $\leftrightarrow$, z čehož podle předchozí věty okamžitě plyne odhad časové i prostorové složitosti. □

## 3.2. 2-souvislost inkrementálně

Inkrementální udržování komponent hranové 2-souvislosti je o něco obtížnější, ale jak ukázali Tarjan a Westbrook v [TW92], existuje pro něj stejně rychlý algoritmus založený opět na udržování ekvivalencí.

O grafu si budeme udržovat následující informace: ekvivalenci 1-propojenosti ($\leftrightarrow$) a 2-propojenosti ($\rightleftarrows$) a les $B$ propojení komponent 2-souvislosti. Vrcholy tohoto lesa odpovídají jednotlivým 2-komponentám, hrany pak mostům mezi nimi.

Operaci *Query* převedeme na *Find* v ekvivalenci $\rightleftarrows$ stejně jako v předchozím případě, inicializace na prázdný graf vytvoří triviální ekvivalence $\leftrightarrow$ i $\rightleftarrows$ a les bez hran v čase $O(n)$.

Při vkládání nové hrany $e = \{v, w\}$ nejprve otestujeme, zda $v \leftrightarrow w$. Pokud ne, stává se $e$ mostem, takže nalezneme 2-komponenty, do nichž patří $v$ a $w$ a spojíme je v $B$ novou hranou. Pokud $v \leftrightarrow w$ i $v \rightleftarrows w$, hrana $e$ leží v jedné 2-komponentě, takže ji je možno ignorovat. Ve zbývajícím případě $v$ a $w$ náleží do různých 2-komponent $C_v$ a $C_w$, a tak v grafu vznikne nová kružnice, na níž leží všechny dřívější mosty na cestě mezi $C_v$ a $C_w$ v $B$. Tím dochází ke spojení všech 2-komponent na této cestě do jedné (jelikož všechny krajní vrcholy zmíněných mostů leží na kružnici, jsou navzájem 2-propojené a díky transitivitě $\rightleftarrows$ jsou tak 2-propojené i všechny ostatní vrcholy jejich 2-komponent). Z této úvahy plyne následující algoritmus pro *Insert*:

| | |
|---|---|
| **function** *Insert*(**vertex** $v$, $w$) | |
|     **if** $\neg$ *Find*$_{\leftrightarrow}(v, w) \rightarrow$ | $v \not\leftrightarrow w$ |
|         *Union*$_{\leftrightarrow}(v, w)$; | *Spojíme 1-komponenty* |
|         *Link*$(v, w)$; | *Přidáme hranu do B* |
|     **elsif** $\neg$ *Find*$_{\rightleftarrows}(v, w) \rightarrow$ | $v \leftrightarrow w$, *ale* $v \not\rightleftarrows w$ |
|         *Condense*$(v, w)$; | *Kontrahujeme v B, tím i Union*$_{\rightleftarrows}$ |
| **end** | |

Potřebujeme tedy vhodně reprezentovat les $B$ tak, abychom do něj byli schopni efektivně přidávat hrany a nalézat a kontrahovat cesty. K tomu použijeme opět stromů orientovaných směrem ke kořeni, přičemž pro snadné kontrahování cest nebudeme informace o vrcholech ukládat přímo, nýbrž do struktury popisující třídy ekvivalence $\rightleftarrows$. Pro každou třídu $C$ (čili pro každý vrchol lesa $B$) si budeme pamatovat libovolného reprezentanta třídy, která je v $B$ otcem $C$. Díky tomu jsme pro každý vrchol lesa $B$ (reprezentovaný libovolným vrcholem původního grafu, který patří do příslušné 2-komponenty), schopni nalézt provedením jediné operace *Find* na relaci $\rightleftarrows$ jeho otce v $B$ (reprezentovaného stejným způsobem).

Operace na $B$ zrealizujeme takto:

- *Condense*$(v, w)$ – nalezneme cestu $P$ mezi vrcholy $v$ a $w$ (to učiníme tak, že budeme vystupovat z $v$ a z $w$ směrem ke kořeni, střídavě po jedné hraně na jedné a na druhé cestě, a budeme si značkovat navštívené vrcholy; jakmile narazíme na již označkovaný vrchol, nalezli jsme nejnižšího společného předka $v$ a $w$ a stačí spojit cesty z $v$ a $w$ do tohoto společného předka, přičemž vrcholy odznačkujeme) a zkontrahujeme ji do jediného vrcholu postupným voláním *Union* na $\rightleftarrows$. Díky použitému triku pro uložení otců vrcholů se tak automaticky vrchol reprezentující nově vzniklou komponentu stane otcem všech synů kontrahovaných vrcholů. To vše jsme zvládli na $O(|P|)$ vyhledání otce a $|P| - 1$ sloučení komponent.
- *Link*$(v, w)$ – budeme vždy připojovat menší strom v $B$ k většímu (ve smyslu počtu vrcholů příslušné komponenty souvislosti, což poznáme pomocí operace $\mathit{Count}_{\leftrightarrow}$; BÚNO $v$ leží v tom větším). Nejprve otočíme směry hran na cestě z $w$ do kořene, čímž se $w$ stane novým kořenem, který poté připojíme pod $v$. To je celkem $d(w)$ vyhledání pointeru.

**Lemma 3.2.1:** *Provedení $O(n)$ operací Condense na lese, který má na počátku $n$ vrcholů, vyžaduje vykonání $O(n)$ operací Union a Find na ekvivalenci $\rightleftarrows$. Pokud navíc provedeme $O(n)$-krát Link, zvýší se počet těchto operací na $O(n \cdot \log n)$.*

**Důkaz:** Vzhledem k tomu, že kontrakce podle cesty $P$ odebere z $B$ $|P| - 1$ vrcholů, musí být celková délka všech cest, podle nichž se kontrahovalo, a tím pádem i celkový počet operací na $\rightleftarrows$, rovny $O(n)$. Co se operace *Link* týče, každý vrchol může ležet na cestě, kterou otáčíme, nejvýše $\log_2 n$ krát (při každém volání *Link* se totiž minimálně zdvojnásobí velikost stromu, v němž vrchol leží), proto celkově provedeme $O(n \cdot \log n)$ operací s těmito vrcholy. $\square$

Nyní již můžeme snadno odvodit časovou složitost operací struktury pro 2-souvislost:

**Věta 3.2.2:** *Provedení posloupnosti $k = \Omega(n \cdot \log n)$ operací Insert a Query v libovolném pořadí na grafu, který zpočátku neobsahoval žádné hrany, trvá čas $O(k \cdot \alpha(k, n))$ a zabere prostor $O(n)$.*

**Důkaz:** Nechť posloupnost obsahuje $k_I$ operací *Insert* a $k_Q$ operací *Query*. Každá taková operace se skládá z rozhodování v konstantním čase a operací s ekvivalencemi $\leftrightarrow$ a $\rightleftarrows$. Spočítejme nyní tyto operace:

Každý dotaz *Query* se přímo přeloží na jednu operaci. Všech $k_I$ operací *Insert* způsobí $O(k_I)$ přímých operací a dále pak $O(n)$ volání *Condense* a *Link*, která podle předchozího lemmatu dohromady učiní dalších $O(n \cdot \log n)$ operací. Celkem tedy $O(k_Q + k_I + n \cdot \log n) = O(k)$ operací.

Z odhadu časové složitosti struktury pro udržování ekvivalence tak získáme slíbený čas $O(k \cdot \alpha(k, n))$. Obě ekvivalence jsou definovány na $n$-prvkové množině a strom $B$ může mít maximálně $n$ vrcholů a $n - 1$ hran, takže zabraný prostor je opravdu lineární. $\square$

**Poznámka:** Pokud bychom nezačínali s grafem bez hran, nýbrž se souvislým grafem, můžeme z odhadu složitosti vynechat operaci *Link*, les $B$ předem zkonstruovat správně zorientovaný během inicializace struktury, a tak získat výše zmíněný časový odhad již pro $k = \Omega(n)$. Tarjan a Westbrook jdou v [TW92] ještě dále a zavedením datové struktury nazývané *Link/Condense* tree (což je poměrně komplikované zobecnění Sleator-Tarjanových dynamických stromů) dosáhnou času $O(k \cdot \alpha(k, n))$ dokonce pro $k$ libovolné.

## 3.3. Souvislost a 2-souvislost s backtrackem

Pro udržování komponent souvislosti s operacemi *Insert* a *Backtrack* navrhl Westbrook ve [W89] strukturu pracující v čase $O(\log n / \log\log n)$ na operaci, nicméně tento čas není podstatným vylepšením oproti níže uvedenému kvazidynamickému algoritmu. Pro udržování 2-komponent dosáhli La Poutré a Westbrook v [LW94] času $O(\log n)$ na operaci, což je opět stejně efektivní jako zde uvedená kvazidynamická struktura. Tyto algoritmy proto neuvedeme.

## 3.4. Souvislost a 2-souvislost kvazidynamicky

Kvazidynamické algoritmy udržují kostru dynamického grafu a umožňují provádět *Delete* libovolných hran mimo tuto kostru a navíc hran kostry, které jsou mosty. Hrana odstraňovaná operací *Backtrack* evidentně tuto podmínku vždy splňuje, takže lze každou kvazidynamickou strukturu používat i jako *Insert/Backtrack* strukturu. Zde uvedeme algoritmus inspirovaný článkem [KR97] a využívající pozorování o pokrývajících hranách z lemmatu 1.1.6.

Kostru grafu budeme udržovat pomocí Sleator-Tarjanových dynamických stromů z kapitoly 2, přičemž ceny hran budou udávat, kolik nestromových hran danou stromovou hranu pokrývá. Jednotlivé operace realizujeme takto:

- **boolean** $\mathit{Query}_1$(**vertex** $v$, $w$) (dotaz na propojenost) – prostřednictvím operací $\mathit{Root}(v)$ a $\mathit{Root}(w)$ zjistíme komponenty kostry, do nichž $v$ a $w$ patří a vrátíme **true**, pokud $v = w$.
- **boolean** $\mathit{Query}_2$(**vertex** $v$, $w$) (dotaz na 2-propojenost) – $v$ a $w$ jsou 2-propojené, právě tehdy, jsou-li propojené (to již umíme zjistit) a všechny hrany na cestě v kostře mezi $v$ a $w$ jsou pokryté alespoň jednou hranou, což zjistíme tak, že spočítáme voláním $\mathit{Evert}(v)$, $\mathit{Mincost}(w)$ a *Cost* minimální cenu hrany na této cestě.
- **edge** *Bridge*(**vertex** $v$, $w$) – most oddělující nalezneme během $\mathit{Query}_2$ – je to totiž libovolná hrana, v níž se minimum nabývalo.
- $\mathit{Insert}(v,\ w)$ – pokud jsou $v$ a $w$ v různých komponentách (to již zjistit umíme), přidáme pomocí $\mathit{Evert}(v)$ a $\mathit{Link}(v,\ w,\ 0)$ novou nepokrytou hranu (most) mezi kostry těchto dvou komponent. Pokud $v$ a $w$ leží v téže komponentě, pak hrana $\{v, w\}$ pokrývá všechny hrany na cestě v kostře mezi $v$ a $w$, takže voláním $\mathit{Evert}(v)$ a $\mathit{Update}(w,\ 1)$ zvýšíme cenu všech hran na této cestě.
- $\mathit{Delete}(v,\ w)$ – zjistíme, zda je hrana $\{v, w\}$ mostem – na to stačí zavolat $\mathit{Evert}(v)$ a $\mathit{Cost}(w)$). Pokud ano, pak se jejím odebráním příslušná komponenta rozpadá, takže hranu vyjmeme z její kostry ($\mathit{Evert}(v)$ a $\mathit{Cut}(w)$); pokud ne, pouze odkrýváme hrany na stromové cestě mezi $v$ a $w$ ($\mathit{Evert}(v)$ a $\mathit{Update}(w,\ -1)$).

**Věta 3.4.1:** *Libovolná posloupnost $k$ operací Insert, Delete, Query$_1$, Query$_2$ a Bridge, která začíná s grafem neobsahujícím žádné hrany, proběhne v čase $O(k \cdot \log n)$ a prostoru $O(n)$.*

**Důkaz:** Uvědomme si, že každá z těchto operací provádí $O(1)$ operací na dynamickém stromu plus konstantní práci, takže aplikováním odhadu složitosti dynamických stromů z věty 2.2.2 získáváme přímo kýžený výsledek. □

Tento algoritmus bohužel nelze triviálně rozšířit na plně dynamický, jelikož v případě smazání hrany kostry nejsme schopni efektivně nalézt její náhradu mezi nestromovými hranami. V příštích kapitolách se pokusíme sestrojit vhodné řešení.

## 3.5. Minimální kostra kvazidynamicky

Pro kvazidynamické udržování minimální kostry použijeme opět dynamických stromů, kterými budeme kostru reprezentovat, přičemž při vkládání hran budeme podle lemmatu 1.2.2 testovat, zda jí máme zařadit do kostry či nikoliv:

- *Insert*(**vertex** $v$, $w$, **real** $c$) – stejně jako u algoritmu pro kvazidynamickou 2-souvislost nejprve ověříme, zda se hrana $\{v, w\}$ stane mostem a pokud ano, zařadíme ji do kostry. Jinak pomocí *Evert*($v$) a *Mincost*($w$) nalezneme nejlevnější hranu $e$ na cestě mezi $v$ a $w$ v kostře. Pokud $c(e) \geq c$, můžeme nově vloženou hranu ignorovat, jinak (pomocí *Evert*, *Cut* a *Link*) hranu $e$ nahradíme hranou $\{v, w\}$. Podle lemmatu 1.2.2 tím získáme minimální kostru nového grafu.
- *Delete*(**vertex** $v$, $w$) – pokud je hrana $\{v, w\}$ součástí kostry, musí být mostem (jinak by totiž toto volání *Delete* bylo ilegální), a tak ji pomocí *Evert*($v$) a *Cut*($w$) můžeme odstranit bez náhrady. Není-li součástí kostry, pak její zrušení můžeme ignorovat.

Obě operace je možno snadno upravit tak, aby mimo volání *Link* a *Cut* ještě generovaly výstup v podobě seznamu $O(1)$ operací *Insert* a *Delete*, který popisuje provedenou úpravu kostry. Navíc je možno doplnit operaci *Query* stejně jako v kvazidynamické struktuře pro souvislost.

**Věta 3.5.1:** *Libovolná posloupnost $k$ operací Insert a Delete, která začíná s grafem neobsahujícím žádné hrany, proběhne v čase $O(k \cdot \log n)$ a zabere prostor $O(n)$.*

**Důkaz:** Uvědomme si, že každá z těchto operací provádí $O(1)$ operací na dynamickém stromu plus konstantní práci, takže aplikováním odhadu složitosti dynamických stromů z věty 2.2.2 získáme přímo kýžený výsledek. □

## 3.6. Souvislost s orákulem

V případě souvislosti lze dokonce slevit i z požadavku na nemazání hran kostry, pokud při vkládání hran předem víme, v jakém pořadí budou mazány:

**Definice 3.6.1:** *Dynamickým grafem s orákulem nazveme dynamický graf, u nějž při každé operaci Insert přiřadíme hraně navíc váhu takovou, že Delete budeme vždy používat pouze na hranu s nejvyšší vahou.*

Na takovém dynamickém grafu můžeme komponenty souvislosti udržovat velice snadno: použijeme předchozí algoritmus pro udržování minimální kostry vzhledem k zadaným vahám a doplníme jej o *Query* tak, jak bylo popsáno. Vzhledem k definici vah bude každý *Delete* legální: Kdybychom jej totiž volali na hranu $\{v, w\}$, která by byla součástí kostry a přitom nebyla mostem, tedy byla pokryta nějakou hranou $e$ mimo kostru, musela by být váha hrany $e$ být větší nebo rovna váze $\{v, w\}$, což je ovšem ve sporu s tím, že $e$ dosud nebyla smazána.

# 4. Topologické stromy

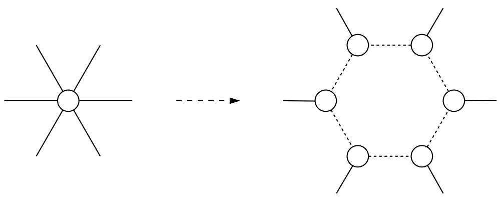
Sleator-Tarjanovy dynamické stromy jsou velice elegantní a efektivní datovou strukturou pro řešení mnohých grafových problémů, nicméně mají jednu velice podstatnou nevýhodu: nezohledňují nikterak topologickou strukturu grafu, tudíž je obtížné (ne-li nemožné) rozšířit je o libovolnou reprezentaci nestromových hran grafu. Výhodnější alternativou jsou v takových případech Fredericksonovy topologické stromy (původně definovány v [F85], zde uvedeny ve zjednodušené verzi) – práce s nimi je sice o něco obtížnější, leč pro všechny základní operace dávají také horní odhad času $O(\log n)$.

## 4.1. 3-omezenost

Topologické stromy jsou založeny na myšlence rekurzivní clusterizace – rozkladu neorientovaného lesa na clustery (vhodně zvolené podgrafy), jejichž kontrakcí získáme „jednodušší" les, který opět clusterujeme atd. Zde uvedený způsob clusterizace funguje pouze pro grafy se stupni všech vrcholů $\leq 3$ (takovým grafům budeme říkat 3-omezené), což ovšem není na škodu, jelikož alespoň pro potřeby zkoumání souvislosti a 2-souvislosti lze každý graf převést na jiný, který je s ním ekvivalentní a tuto podmínku splňuje:

**Věta 4.1.1:** *Ke každému grafu $G$ existuje graf $H$ a zobrazení $f : V(G) \to V(H)$ takové, že platí:*

*(i)* $\forall v \in V(H) : deg(v) \leq 3$.
*(ii)* $|V(H)| = O(|V(G)| + |E(G)|), |E(H)| = O(|V(H)|)$.
*(iii)* *Přidání či odebrání hrany v $G$ způsobí přidání, resp. odebrání $O(1)$ vrcholů a hran v $H$. Překlad těchto operací v $G$ na operace v $H$ lze provést v čase $O(1)$.*
*(iv)* $\forall v, w \in V(G) : v \leftrightarrow_G w \iff f(v) \leftrightarrow_H f(w)$.
*(v)* $\forall v, w \in V(G) : v \rightleftarrows_G w \iff f(v) \rightleftarrows_H f(w)$.



Obr. 5: Zavedení kruhových objezdů

**Důkaz:** Vrcholy stupňů větších než 3 nahradíme jeden po druhém „kruhovými objezdy" – vrchol $v$ stupně $k > 3$ nahradíme kružnicí $C_k$ tak, že každá z hran incidentních s $v$ bude nyní připojena na některý z vrcholů kružnice a žádný vrchol nepoužijeme vícekrát (viz obr. 5). Nově přidané hrany budeme nazývat čárkovanými. Funkce $f$ bude každému vrcholu grafu $G$ se stupněm $\leq 3$ přiřazovat tentýž vrchol v grafu $H$, ostatním vrcholům pak libovolný vrchol jejich kruhových objezdů. Takový graf jistě splňuje podmínku (i).

(ii): Vrcholy grafu $H$ sestávají jednak z vrcholů $G$, jednak z vrcholů vytvořených kruhových objezdů, kterých je ovšem nejvýše tolik, kolik je $\sum_{v \in V(G)} deg_G v = 2|E(G)|$. Jelikož $H$ má omezené stupně, musí mít počet hran nejvýše lineární vzhledem k počtu vrcholů.

(iii): Graf $H$ je pro daný graf $G$ určen až na isomorfismus a pořadí vrcholů na kruhových objezdech jednoznačně a přidání a odebrání hrany v $G$ jsou navzájem inverzní operace, takže stačí (iii) dokázat pouze pro jednu z nich. Přidáme-li hranu do $G$, pak pro každý z jejích koncových vrcholů nastane jedna z následujících situací: buďto má příslušný vrchol stupeň menší než 3 (pak zůstává tak, jak je) nebo již je nahrazen kruhovým objezdem (pak přidáváme jeden vrchol a jednu čárkovanou hranu, abychom kruhový objezd rozšířili), případně z vrcholu musíme vytvořit nový kruhový objezd (čímž přidáme 2 nové vrcholy a 3 čárkované hrany). Celkem tedy v $H$ vytvoříme nejvýše 7 nových hran a 4 nové vrcholy a případně přepojíme hrany od původních vrcholů k novým, což je dohromady $O(1)$ změn.

Pro efektivní převod operací v $G$ na operace v $H$ stačí pamatovat si u vrcholů $G$ jejich stupně (ani není nutné mít uložený graf samotný), funkci $f$, přiřazení vrcholů kruhových objezdů hranám $G$ a graf $H$ ve formě seznamů hran incidentních s jednotlivými vrcholy.

(iv) $\Longrightarrow$: Nechť $v \leftrightarrow_G w$ a $P = \langle v = p_0, \ldots, p_l = w\rangle$ je cesta v $G$ spojující $v$ a $w$. Potom stačí dokázat, že $\forall i : f(p_i) \leftrightarrow_H f(p_{i+1})$ a z transitivity $\leftrightarrow_H$ získáme $f(v) \leftrightarrow_H f(w)$. Ovšem pokud mezi $p_i$ a $p_{i+1}$ vede v $G$ hrana, musí v $H$ vést hrana $e$ mezi nějakým vrcholem $q$ na kruhovém objezdu vzniknuvším z vrcholu $p_i$ (pokud $deg_G(p_i) \leq 3$, považujeme vrchol samotný za triviální kruhový objezd a $q_i = p_i$) a nějakým vrcholem $q'$ na kruhovém objezdu $p_{i+1}$. Proto jsou $f(p_i)$ a $f(p_{i+1})$ propojeny cestou vedoucí nejprve z $f(p_i)$ po kruhovém objezdu vrcholu $p_i$ do $q$, pak po hraně $e$ do $q'$ a konečně po kruhovém objezdu $p_{i+1}$ do $f(p_{i+1})$.

(iv) $\Longleftarrow$: Graf $G$ je kontrakcí grafu $H$ podle všech čarkovaných hran kruhových objezdů (tak každý kruhový objezd nahradíme jedním vrcholem grafu $G$). Pokud jsou $f(v)$ a $f(w)$ spojeny v $H$ nějakou cestou $P$, pak $v$ a $w$ jsou v $G$ spojeny kontrakcí $P$, což sice nemusí být cesta, ale $\leftrightarrow_G$ je transitivní, takže $v \leftrightarrow_G w$.

(v) $\Longrightarrow$: Potřebujeme ukázat, že pokud $v \rightleftarrows_G w$, pak pro každou hranu $e \in E(H)$ platí $f(v) \leftrightarrow_{H-e} f(w)$. Pokud je $e$ čárkovanou hranou nějakého kruhového objezdu, pak lze použít tentýž argument jako v (iv), jelikož kruhové objezdy jsou souvislé i po odebrání libovolné jedné hrany. Je-li $e$ hrana zděděná z $G$, pak $v \leftrightarrow_{G-e} w$ a jelikož $H - e$ je transformací $G - e$ pomocí kruhových objezdů (případně s podrozdělenou jednou hranou, ale to na propojenosti vrcholů nic nemění), plyne z (iv), že $f(v) \leftrightarrow_{H-e} f(w)$.

(v) $\Longleftarrow$: Buď $e \in E(G)$ a $f(v) \rightleftarrows_H f(w)$, tedy i $f(v) \leftrightarrow_{H-e} f(w)$; buď $P$ cesta, která to potvrzuje. Analogicky s důkazem (iv): $G - e$ je kontrakcí $H - e$, takže kontrakce $P$ je tah v $G - e$, který spojuje $v$ s $w$, a proto $v \leftrightarrow_{G-e} w$. [Pro (iv) a (v) ve skutečnosti nebyla struktura kruhových objezdů významná – stačilo, že jsme vrcholy $G$ nahrazovali 2-souvislými grafy.] $\square$

Nevýhodou této konstrukce ale je, že stromy nemusí převádět na stromy. Ve většině aplikací to nebude na škodu, protože budeme pracovat s obecnými grafy, které nejprve transformujeme, poté spočteme kostru, a tu teprve budeme clusterovat. Pro úplnost ovšem vyslovíme analogickou větu i pro stromy.

**Věta 4.1.2:** *Ke každému lesu $T$ existuje les $U$ a zobrazení $f : V(T) \to V(U)$ takové, že platí:*

*(i)* $\forall v \in V(U) : deg(v) \leq 3$.
*(ii)* $|V(U)| = O(|V(T)|), |E(U)| = O(|V(U)|)$.
*(iii) Přidání či odebrání hrany v $T$ způsobí přidání, resp. odebrání $O(1)$ vrcholů a hran v $U$. Překlad těchto operací v $T$ na operace v $U$ lze provést v čase $O(1)$.*
*(iv)* $\forall v, w \in V(T) : v \leftrightarrow_T w \Longleftrightarrow f(v) \leftrightarrow_U f(w)$.

**Důkaz:** Použijeme tutéž konstrukci jako v důkazu předchozí věty, pouze z každého kruhového objezdu vynecháme jednu čárkovanou hranu, čímž zabráníme vzniku cyklů. Vlastnosti (i), (iii) a (iv) budou nadále platit, pro (ii) si stačí povšimnout, že každý les má $O(|V|)$

hran, a tak $O(|V(T)| + |E(T)|) = O(|V(T)|)$. Vlastnost (v) by sice pro tuto konstrukci také platila, nicméně u lesů je bezpředmětné hovořit o 2-souvislosti. □

## 4.2. Clusterizace

Nyní již můžeme přistoupit k definici clusterizace 3-omezených grafů:

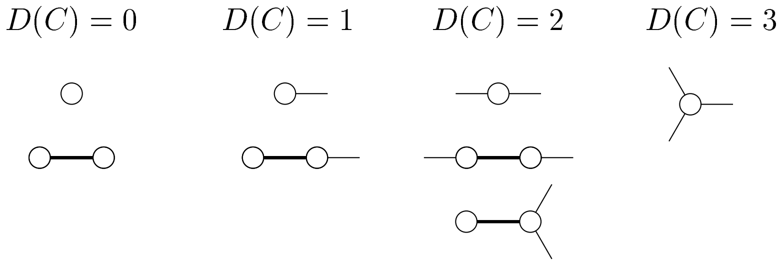
**Definice 4.2.1:** *Buď $T = (V, E)$ les. Množinu $\mathscr{C}$ indukovaných podgrafů lesa $T$ nazveme* ***rozkladem** $T$ **řádu** $k$, právě tehdy, když:*

*(i) Všechny clustery (prvky $\mathscr{C}$) mají maximálně $k$ vrcholů.*
*(ii) Všechny clustery jsou souvislé grafy.*
*(iii) Každý vrchol $T$ je obsažen v právě jednom clusteru.*

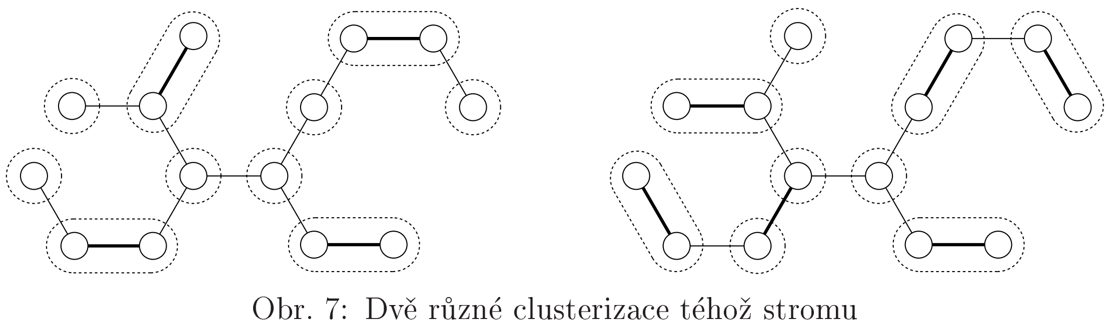
**Definice 4.2.2:** *Buď $\mathscr{C}$ rozklad lesa $T$. Pro každý cluster $C \in \mathscr{C}$ nadefinujeme* ***kardinalitu*** *$|C|$ jako počet vrcholů clusteru a* ***vnější stupeň*** *$D(C)$ jako počet hran $T - E(C)$ incidentních s vrcholy $C$. Dva clustery označíme za* ***sousední****, jsou v $T$ spojeny hranou.*

**Definice 4.2.3:** *$\mathscr{C}$ nazveme* ***clusterizací*** *3-omezeného lesa $T$, je-li $\mathscr{C}$ rozkladem $T$ řádu 2 a pro každý cluster $C \in \mathscr{C}$ jsou splněny následující podmínky:*

*(i) $D(C) \leq 3$ ($C$ má vnější stupeň nejvýše 3)*
*(ii) $D(C) = 3 \Longrightarrow |C| = 1$ (clustery stupně 3 jsou jednovrcholové)*
*(iii) $C$ není možno spojit s žádným sousedním clusterem, aniž by byla porušena některá z předchozích podmínek.*




Obr. 6: Možné typy clusterů

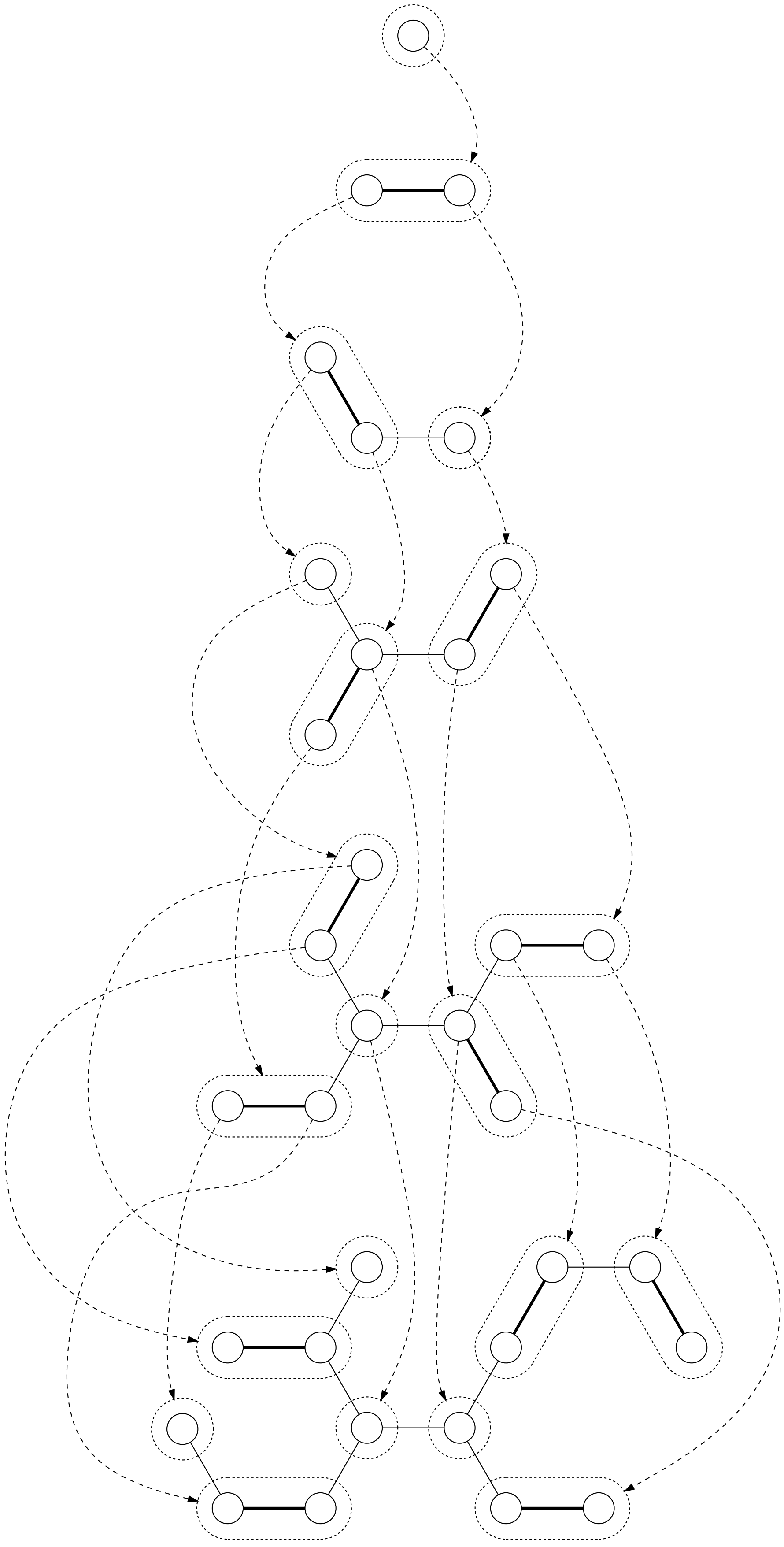


Obr. 7: Dvě různé clusterizace téhož stromu

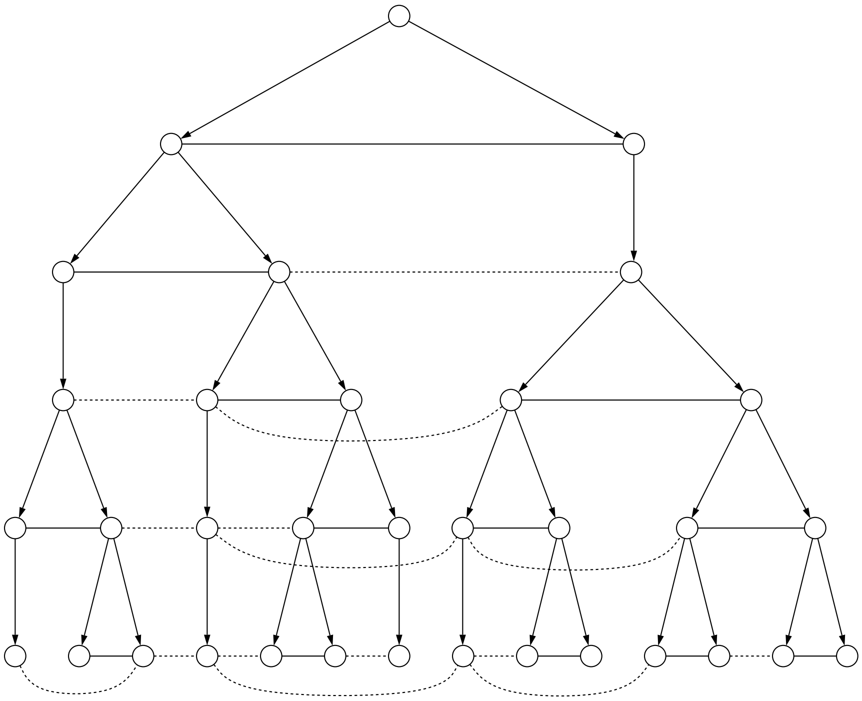
Obrázek 6 ukazuje všech 7 možných typů clusterů, obr. 7 pak dvě různé clusterizace téhož stromu, které obě splňují definici a liší se nejen rozdělením vrcholů do clusterů, ale i počtem clusterů. Z toho plyne, že podmínka (iii) z definice clusterizace nevede nutně k minimálnímu počtu clusterů, pouze k lokálně minimálnímu. Celkový počet clusterů je ale možno omezit alespoň takto:

**Věta 4.2.4** (Frederickson)**:** *Buď $T$ 3-omezený les a $\mathscr{C}$ jeho clusterizace. Potom $|\mathscr{C}| \leq 5/6 \cdot |V(T)|$.*

**Důkaz:** Viz [F85]. □



Obr. 8: Rekurzivní clusterizace



Obr. 9: Topologický strom clusterizace z obr. 8

Clustery můžeme kontrahovat podle jejich vnitřních hran, čímž získáme nový 3-omezený les, jehož vrcholy odpovídají clusterům a hrany meziclusterovým hranám. Tento les je možno znovu clusterovat atd., až po konečném počtu opakování každou komponentu původního lesa zkontrahujeme do jediného vrcholu.

**Definice 4.2.5:** *Posloupnosti* $\mathscr{R} = \left\langle T^0, \mathscr{C}^1, T^1, \mathscr{C}^2, \ldots, \mathscr{C}^k, T^k \right\rangle$ *budeme říkat* ***rekursivní clusterizace*** *3-omezeného lesa* $T$*, pokud* $T^0 = T$*,* $\mathscr{C}^i$ *je clusterizace* $T^{i-1}$*, jejíž kontrakcí vznikne* $T^i$ *a* $T^k$ *neobsahuje žádné hrany. Grafy* $T^i$ *nazveme* ***úrovněmi*** $\mathscr{R}$*, prvky* $\mathscr{C}^i$ *označíme za* ***clustery úrovně*** $i$*.*

Tento proces je naznačen na obr. 8: jednotlivé úrovně jsou zobrazeny pod sebou, clustery jsou ohraničeny tečkovaně, hrany uvnitř clusterů vytaženy tučně, čárkované hrany ukazují, který vrchol na úrovni $k$ vznikl z kterého clusteru úrovně $k-1$.

**Definice 4.2.6:** *Buď* $T$ *3-omezený les,* $\mathscr{R}$ *jeho rekursivní clusterizace s úrovněmi* $\{T^i\}_{i=0}^k$*.* ***Topologickým stromem**** *této clusterizace nazveme graf, jehož vrcholy jsou disjunktním sjednocením vrcholů všech* $T_i$ *a* $(v, w)$ *je hranou právě tehdy, když* $\exists i : v \in T^i, w \in T^{i-1}$ *a vrchol* $v$ *vznikl kontrakcí clusteru obsahujícího vrchol* $w$*. Navíc* $v$ *a* $w$ *na téže úrovni jsou spojeny čárkovanou (pomocnou) hranou, jsou-li kontrakcemi sousedních clusterů.*

Topologický strom clusterizace z obr. 8 je ukázán na obrázku 9.

**Lemma 4.2.7:** *Topologický strom libovolné rekursivní clusterizace 3-omezeného lesa na* $n$ *vrcholech má hloubku* $O(\log n)$ *a velikost (to jest počet vrcholů + počet hran)* $O(n)$*.*

* Zde je Fredericksonova terminologie poněkud zavádějící – topologický strom vůbec nemusí být souvislý a pokud uvažujeme i čárkované hrany, není ani acyklický. My se jí ovšem budeme držet, aby nevznikl ještě větší zmatek. Pokud ovšem bude hrozit nedorozumění, zdůrazníme raději, že se jedná o topologický les.

**Důkaz:** Z věty 4.2.4 víme, že $|V(T^i)| \leq (5/6)^i \cdot n$, tudíž pro $k = \lceil -\log_{5/6} n \rceil$ musí být $V(T^k) \leq 1$, takže kořeny stromů topologického lesa se nacházejí na úrovni nejvýše $k = O(\log n)$. Navíc $|V| = \sum_{i=0}^{k} |V(T^i)| = O(n)$. Z každého vrcholu mohou vycházet maximálně 3 hlavní hrany (k otci a ke dvěma synům) a maximálně 3 hrany pomocné (existují nejvýše 3 sousední clustery), proto i hran je $O(n)$. $\square$

## 4.3. Udržování topologických stromů

Clusterizaci stromu je možno nalézt v lineárním čase prohledáním grafu do hloubky z libovolného listu. Procházíme vrcholy stromu v postorderu a pro každý vrchol sestrojíme clusterizaci podstromu určeného tímto vrcholem, a to takto: je-li $v$ listem, učiníme jej jednovrcholovým clusterem, jinak pro jeho syny $w_1$ a případně $w_2$ (mohou být maximálně dva, jelikož pracujeme s 3-omezeným stromem a buďto je $v$ kořen a pak má stupeň nejvýše 1 nebo není, a pak z něj jedna hrana vede do jeho otce) nejprve spočteme clusterizace podstromů určených $w_1$ a $w_2$ obsahující clustery $C_1, C_2 : w_i \in C_i$ a poté za výslednou clusterizaci prohlásíme jejich sjednocení, ke kterému ovšem musíme přidat nový cluster obsahující vrchol $v$ a případně jej spojit s $C_1$ nebo $C_2$, abychom dodrželi podmínku (iii) z definice clusterizace. Pokud má $v$ oba syny, je možno $v$ přiclusterovat pouze k $C_i$ velikosti 1, ze kterého nevede žádná další hrana (cluster stupně 3 musí mít jediný vrchol). Pokud má $v$ pouze jednoho syna, můžeme $v$ k jeho clusteru připojit jen tehdy, když $|C_1| = 1$ a $D(C_1) \leq 2$.

Iterováním tohoto algoritmu, přičemž po každém průchodu všechny clustery kontrahujeme, zkonstruujeme všechny úrovně rekursivní clusterizace v čase $O(n)$ (velikosti úrovní klesají geometrickou řadou). Při výpočtu kontrakcí je rovněž možno ve stejném čase sestrojit přislušný topologický strom.

Pro naše účely ovšem topologický strom potřebujeme umět udržovat dynamicky, to jest přepočítat ho při přidání nebo odebrání hrany v původním grafu. Takové operace bohužel nejsou lokálními úpravami jedné úrovně clusterizace – odebrání jedné hrany způsobí na každé úrovni buďto zánik jedné meziclusterové hrany nebo rozdělení clusteru, které se musí promítnout o úroveň výše, a konečným důsledkem je pak rozdělení jedné komponenty topologického stromu na dvě, což odpovídá vzrůstu počtu komponent v původním grafu; analogicky pro přidání hrany. Naštěstí je možno přepočítávání úrovní zorganizovat tak, že je počet nutných úprav omezen logaritmicky.

**Algoritmus 4.3.1** *(Update topologického stromu)***:**

*Topologický strom budeme procházet po úrovních, počínaje úrovní nultou (původní graf). V každém kroku si budeme pamatovat seznam $L_D$ zrušených vrcholů, seznam $L_C$ změněných vrcholů (to jsou ty, kterým se změnil buďto některý ze synů nebo k nim příslušné meziclusterové hrany, takže je nutno ověřit, jedná-li se stále o korektní cluster) majících otce a seznam $L_A$ vrcholů, které nemají otce, ať již proto, že byly nově vytvořeny nebo proto, že byl jejich otec zrušen.*

*Na počátku inicializujeme $L_A$ a $L_D$ na prázdné seznamy a $L_C$ obsahuje oba krajní vrcholy přidané, resp. odebrané hrany.*

*Pro každou úroveň odebíráme vrcholy z těchto seznamů, zpracováváme je a zařazujeme nové vrcholy bezprostředně vyšší úrovně do nových seznamů $L'_A$, $L'_D$ a $L'_C$. Přitom přepočítáváme čárkované hrany, aby odpovídaly nové situaci, jak se ji dozvídáme z předchozí úrovně stromu. Nejprve probereme $L_D$: pro každý vrchol $x \in L_D$ odstraníme $x$ z $L_D$, zrušíme jej v reprezentaci topologického stromu včetně hrany spojující jej s jeho otcem $y$*

*(pokud $x$ není kořen) a nemá-li $y$ další syny, vložíme jej do $L'_D$. Má-li $y$ druhého syna $x'$, který dosud není na $L_C$ ani $L_D$, vložíme $x'$ do $L_C$.*

*Poté vyhledáme všechny vrcholy v $L_C$, které mají sourozence – buď $x$ takový vrchol, $y$ jeho otec a $x'$ sourozenec $x$. Pokud cluster odpovídající $y$ je ještě stále korektním clusterem (to znamená, že $x$ a $x'$ jsou spojeny hranou a vnější stupeň $y$ nepřekračuje 2), odstraníme $x$ i $x'$ z $L_C$ (pokud tam jsou) a přidáme $y$ do $L'_C$. V opačném případě odstraníme hrany spojující $x$ a $x'$ s $y$, vrcholy $x$ a $x'$ přesuneme do $L_A$ a $y$ do $L'_D$.*

*Následně zpracujeme seznam $L_A$ a zbylé prvky seznamu $L_C$. Každý prvek $x$ některého z těchto dvou seznamů odebereme a rozlišíme tyto případy (nechť $C_x$ je cluster reprezentovaný vrcholem $x$):*

*(1) $D(C_x) = 3$ a $x$ má nějakého souseda $x'$ takového, že $D(C_{x'}) = 1$. Pak $C_x$ a $C_{x'}$ spojíme do jednoho clusteru a $x'$ odebereme ze seznamu, jehož je případně členem. Neměl-li ani $x$, ani $x'$ otce, založíme nový vrchol o úroveň výše a přidáme jej do $L'_A$. Měl-li otce právě jeden z nich, připojíme k němu i druhý vrchol a otce zařadíme do $L'_C$. Pokud měly otce oba, použijeme otce $y$ vrcholu $x$, připojíme k němu i $x'$, zařadíme $y$ do $L'_C$ a původního otce $y'$ vrcholu $x'$ do $L'_D$.*

*(2) $D(C_x) = 3$ a takový soused neexistuje: $C_x$ bude jednovrcholový cluster. Pokud $x$ měl otce $y$, vložíme $y$ do $L'_C$, jinak mu otce vytvoříme a zařadíme do $L'_A$.*

*(3) $D(C_x) = 2$: Pokud $x$ sousedí s $x'$ takovým, že $D(C_{x'}) \leq 2$ a $x'$ nemá žádného sourozence (buďto proto, že je jediným synem nebo proto, že mu ještě žádný otec nebyl vytvořen), spojíme $C_x$ a $C_{x'}$ do jednoho clusteru stejně jako v případě (1). Neexistuje-li takový vrchol $x'$, $C_x$ bude jednovrcholovým clusterem, který zpracujeme stejně jako v případě (2).*

*(4) $D(C_x) = 1$: analogicky s (3), pouze nevyžadujeme žádné omezení stupně sousedního clusteru.*

*(5) $D(C_x) = 0$: $x$ je nyní kořenem a není mu potřeba věnovat žádnou další pozornost.*

*Nakonec nahradíme vyprázdněné seznamy $L_A$, $L_D$ a $L_C$ nově spočtenými $L'_A$, $L'_D$ a $L'_C$ a pokud je alespoň jeden z nich neprázdný, pokračujeme následující úrovní.*

**Věta 4.3.2** (Frederickson)**:** *Algoritmus 4.3.1 přepočte topologický strom po přidání nebo odebrání hrany původního grafu v čase $O(\log n)$.*

**Důkaz:** Viz [F85]. $\square$

**Důsledek 4.3.3:** *Pomocí topologických stromů je možno v čase $O(\log n)$ na operaci implementovat plně dynamickou strukturu pro udržování komponent souvislosti lesa s operacemi Link, Cut a Query.*

**Důkaz:** Pomocí transformace z věty 4.1.2 převádíme původní les na ekvivalentní 3-omezený les (velikosti $O(n)$), který budeme reprezentovat pomocí topologického lesa. Každou operaci *Link* či *Cut* nahradíme $O(1)$ operacemi na 3-omezeném lese, které provedeme v čase $O(\log n)$. Máme-li odpovědět na dotaz $Query(x, y)$, nejprve zjistíme reprezentanty $f(x)$ a $f(y)$ vrcholů $x$ a $y$ v 3-omezeném lese, načež nalezneme v čase $O(\log n)$ kořeny $r_x$ a $r_y$ stromů topologického lesa, ve kterých tito reprezentanti leží. Pokud $r_x = r_y$, jsou $x$ a $y$ v téže komponentě, jinak nikoliv. $\square$

## 4.4. Rozklad cest

Na rozdíl od Sleator-Tarjanových dynamických stromů topologické stromy přímo neobsahují informace o cestách v grafech. Naštěstí je možno využít struktury clusterů a vhodnou reprezentaci cest zavést:

**Definice 4.4.1:** *Každému clusteru $C$ v rekurzivní clusterizaci přiřadíme* ***expanzi clusteru*** *$G(C)$, což je podgraf původního grafu, jehož postupným kontrahováním vznikl cluster $C$. Vrcholy $G(C)$ nazveme* ***vnitřními vrcholy*** *$C$ a budeme je značit $I(C)$.* ***Hraničními vrcholy*** *$\partial C$ nazveme ty vnitřní vrcholy, ze kterých vedou hrany spojující $C$ se zbytkem grafu na téže úrovni ($|\partial C| \leq D(C)$).*

**Definice 4.4.2:** *Buď $\mathscr{T}$ topologický strom. Každému clusteru $C$ přiřadíme* ***částečnou a úplnou cestu*** *podle následujících pravidel:*

- *(i) Je-li $C$ cluster úrovně 0 (tedy vrchol původního grafu), jeho částečná cesta obsahuje tento jediný vrchol, úplná cesta je prázdná.*
- *(ii) Je-li $C$ cluster, který má v topologickém stromu pouze jednoho syna, je mu přiřazena částečná i úplná cesta tohoto syna.*
- *(iii) Je-li $C$ cluster vzniklý spojením clusterů $A$ a $B$ (BÚNO $D(A) \leq D(B)$) hranou $e$, pak rozlišíme tyto případy:*
  - *(1) $D(A) = 1, D(B) = 1$ ($C$ odpovídá kořeni topologického stromu): částečná cesta je prázdná, úplná cesta je spojením částečných cest $A$ a $B$ hranou $e$.*
  - *(2) $D(A) = 1, D(B) = 2$ nebo $D(A) = 2, D(B) = 2$: částečná cesta je spojením částečných cest $A$ a $B$ hranou $e$, úplná cesta je prázdná.*
  - *(3) $D(A) = 1, D(B) = 3$ (v takovém případě je $|B| = 1$): částečná cesta obsahuje pouze jediný vrchol clusteru $B$, úplná cesta je spojením částečné cesty $A$, hrany $e$ a vrcholu clusteru $B$.*

Podle této definice je částečná cesta v clusteru stupně 2 cestou mezi jeho dvěma hraničními vrcholy, v clusteru stupně 1 cestou „odněkud zevnitř clusteru“ k jeho hraničnímu vrcholu a konečně v clusteru stupně 3 jediný (tím pádem také hraniční) vrchol. Na tuto cestu se případně může napojovat úplná cesta téhož clusteru (to může nastat pouze v případech, kdy spojujeme clustery lichého stupně). Místu jejího napojení budeme říkat ***vršek*** této cesty.

**Lemma 4.4.3:** *Každá hrana je obsažena v právě jedné úplné cestě.*

**Důkaz:** Vezměme cluster $C$ takový, že obsahuje danou hranu $e$ a žádný z jeho podclusterů ji už neobsahuje (takový cluster existuje právě jeden). Jinými slovy, $C$ vznikl spojením dvou clusterů hranou $e$, která se tak stává buďto součástí jeho úplné cesty (to tehdy, jsou-li oba podclustery lichého stupně) nebo jeho částečné cesty, která na další úrovni clusterizace stává součástí další částečné cesty atd., až se dostane do nějaké úplné cesty. Úplné cesty se již nadále slučování cest neúčastní, takže nemohou být součástí jiných úplných cest, čímž je lemma dokázáno. □

**Věta 4.4.4:** *Kteroukoliv cestu mezi dvěma vrcholy $v$ a $w$ lze v čase $O(\log n)$ rozložit na $O(\log n)$ částečných cest a samostatných hran.*

**Důkaz:** Sestrojíme algoritmus, který tento rozklad nalezne. Vyhledejme nejnižšího společného předka vrcholů $v$ a $w$ v topologickém stromu a označme jej $C$. Expanze $G(C)$ je souvislý graf, který obsahuje $v$ i $w$, tudíž i celou cestu $P$ mezi nimi, již hledáme. Navíc $C$ musí být tvořen spojením nějakých dvou clusterů $C_1$ a $C_2$ hranou $\{x_1, x_2\}$ (kde $x_i \in I(C_i)$) a $v$ je vnitřním vrcholem $C_1$, zatímco $w$ vnitřním vrcholem $C_2$ – kdyby tomu tak nebylo, tak jsou $v$ i $w$ vnitřními vrcholy téhož syna, čili $C$ není nejnižším společným předkem.

Proto se hledaný rozklad musí skládat z rozkladu cesty v $G(C_1)$ z $v$ do $x_1$, hrany $\{x_1, x_2\}$ a rozkladu cesty v $G(C_2)$ z $x_2$ do $w$.

Tím jsme úlohu převedli na rozklad cesty v expanzi clusteru $X$ z jeho vnitřního vrcholu $x$ do hraničního vrcholu $y$. Takový rozklad nalezneme jednoduchou rekursivní procedurou:

- Je-li $X$ cluster na úrovni 0, odpovíme cestou skládající se z tohoto clusteru.
- Je-li $X$ triviální cluster ($|X|=1$), hledáme stejnou cestu v jeho synovi.
- Jsou-li $x$ a $y$ hraniční vrcholy $X$, vrátíme jako výsledek částečnou cestu clusteru $X$.
- Skládá-li se $X$ z dvou clusterů $X_1$ a $X_2$ spojených hranou $e = \{x_1, x_2\}$, kde $x_i \in \partial X_i$ a $y \in \partial X_2$, rozlišíme dva případy: Pokud $x \in G(X_2)$ (to zjistíme snadno podle toho, že se $X_2$ vyskytuje v topologickém stromu na cestě z $x$ do $C$), leží celá cesta z $x$ do $y$ v expanzi $X_2$, takže řešíme tentýž problém s $X_2$ místo $X$. Pokud $x \in G(X_1)$, skládá se hledaná cesta z cesty v $G(X_1)$ z $x$ do $x_1$ (tu nalezneme rekursivním aplikováním téže procedury), hrany $e$ a cesty v $G(X_2)$ z $x_2$ do $y$ (to je částečná cesta, takže ji již nemusíme rozkládat).

Tato procedura na každé hladině topologického stromu stráví čas $O(1)$. Vzhledem k tomu, že hloubka topologického stromu je logaritmická a popsanou proceduru voláme během výpočtu dvakrát, bude časová složitost celého algoritmu a tím pádem i velikost jeho výstupu $O(\log n)$. $\square$

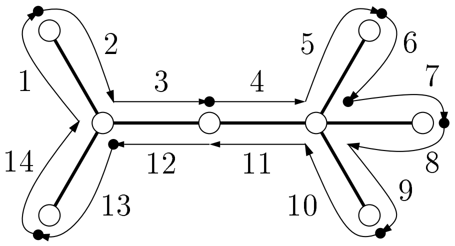
**Důsledek 4.4.5:** *Popsaný rozklad cest můžeme použít například k online výpočtům celkové ceny cesty mezi danými dvěma vrcholy v hranově ohodnoceném stromu. U každé hrany si budeme pamatovat její cenu a stejně tak celkovou cenu každé částečné cesty. Pro každé vrcholy $v$ a $w$ můžeme cestu mezi nimi rozložit na $O(\log n)$ podcest, pro které již celkové ceny známe, a tak odpovídat na dotazy v čase $O(\log n)$. Při úpravách stromu (Link a Cut) využijeme toho, že algoritmus pro aktualizaci rekursivní clusterizace upravuje clustery od nejnižší úrovně k nejvyšší, takže budeme-li přepočítávat v každém clusteru, který leží na seznamu $L_C$ či $L_A$ informace o jeho částečné cestě z analogických údajů uložených pro jeho syny, můžeme si být jisti, že informace v synech jsou již aktuální. Tím pádem i tato aktualizace bude trvat čas $O(\log n)$.*

Podobně nám topologické stromy dají stejně efektivní řešení na všechno, na co Sleator-Tarjanovy dynamické stromy, ovšem za cenu daleko komplikovanějších algoritmů.

# 5. ET-stromy

Třetí klasickou dynamickou reprezentací lesů jsou ET-stromy (Euler Tour trees) zavedené v práci [HK95] a zde zobecněné. Tyto stromy nejsou natolik flexibilní jako Fredericksonovy topologické stromy či Sleator-Tarjanovy dynamické stromy, leč jsou implementačně daleko jednodušší. Proto je lze s výhodou používat u algoritmů, které si vystačí s několika základními stromovými operacemi, konkrétně rozdělováním a spojováním stromů a vyhledáváním informací asociovaných s jejich vrcholy. ET-stromy navíc poskytují eulerovské očíslování vrcholů, na kterém je možno založit elegantní reprezentaci nestromových hran.

## 5.1. Eulerovské tahy



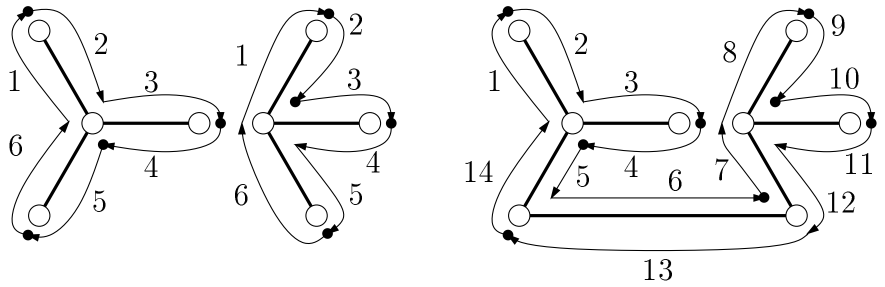

Obr. 10: Eulerovský tah ve stromu

Máme-li reprezentovat neorientovaný les $T$, nejprve nahradíme každou hranu dvojicí orientovaných hran, čímž se každá komponenta stane eulerovskou (vstupní a výstupní stupeň si budou u všech jejích vrcholů rovny). Poté ve všech komponentách zkonstruujeme uzavřený eulerovský tah (ten získáme v lineárním čase tak, že komponentu prohledáme do hloubky a vypisujeme vrcholy jak v případě, kdy do nich vstupujeme, tak když je opouštíme), v libovolném místě jej rozpojíme (při prohledávání do hloubky vyjde přirozené rozpojení tahu v kořeni) a zkonstruujeme $(a, b)$-strom s $b = 2a$, jehož vnější vrcholy budou odpovídat hranám komponenty a inorderové pořadí těchto vrcholů bude udávat, jak tah hrany postupně navštěvoval. Hranu reprezentovanou vrcholem $w$ označíme $e_w$.

***ET-stromem*** (resp. ET-lesem) budeme nazývat vzniklou množinu $(a, b)$-stromů spolu s následujícími pomocnými informacemi: Ke každému vrcholu $v$ původního stromu si zapamatujeme jeho ***význačný výskyt*** *designated*$(v)$ v ET-stromu, což je libovolný vrchol, který reprezentuje hranu z $v$ vycházející nebo **null** pro izolované vrcholy. Na obr. 10 je vyobrazen eulerovský tah v jednoduchém stromu, černě jsou vyznačeny význačné výskyty.

U každého vnějšího vrcholu $w$ ET-stromu uložíme navíc hodnoty *from*$(w)$ a *to*$(w)$, tedy počáteční a koncový vrchol hrany $e_w$ v $T$ a *twin*$(w)$, což je vrchol odpovídající hraně opačné k $e_w$. U vnitřních vrcholů místo toho udržujeme hodnotu *count*$(w)$ obsahující počet vnějších vrcholů v podstromu vrcholem $w$ určeném. Operace s $(a, b)$-stromy jsou všechny definovány prostřednictvím štěpení a slučování vnitřních vrcholů, při nichž lze snadno hodnoty *count* přepočítávat, a tak je udržovat aktuální. Pokud chceme, abychom operaci *Delete* zadávali čísla vrcholů a ne identifikátor hrany (za který můžeme s výhodou prohlásit jeden ze dvou vrcholů v ET-stromu, které této hraně odpovídají), musíme si ještě založit datovou strukturu (například také $(a, b)$-strom), která bude dvojicím čísel vrcholů přiřazovat identifikátory hran.

Nejprve nadefinujeme operaci *Count*(**edge** $e$), která hraně $e$ přiřadí její pořadí na eulerovském tahu. Budeme z vrcholu odpovídajícího hraně $e$ stoupat směrem ke kořeni a v každém vnitřním vrcholu, který navštívíme, přičteme počet vnějších vrcholů v podstromech nalevo od toho, ze kterého jsme přišli (to jsou buďto triviální podstromy nebo podstromy určené vnitřním vrcholem, ve kterém již máme spočtenu hodnotu *count*).




Obr. 11: *Cut* a *Link* v ET-stromu

Při odstraňování hrany operací $Cut(v,\, w)$ nejprve zjistíme její identifikátor, tedy vrchol $e_1$ v ET-stromu odpovídající jedné z orientací této hrany, pomocí $twin(e_1)$ pak získáme párovou hranu $e_2$ s opačnou orientací. Poté zajistíme, aby $e_1$ na eulerovském tahu ležela před $e_2$ (to budeme značit $e_1 \prec e_2$) – pomocí operace *Count* zjistíme, zda tomu tak je, a pokud není, tak $e_1$ a $e_2$ prohodíme. Nyní $(a, b)$-strom rozdělíme na pět částí:

$$T_1 = \langle \text{hrany od počátku do předchůdce } e_1 \rangle$$
$$T_2 = \langle e_1 \rangle$$
$$T_3 = \langle \text{hrany po } e_1\text{, ale před } e_2 \rangle$$
$$T_4 = \langle e_2 \rangle$$
$$T_5 = \langle \text{hrany po } e_2 \text{ až do konce} \rangle$$

a spojíme $T_1$ a $T_5$ do jednoho stromu, který bude obsahovat eulerovský tah v jedné z vzniklých komponent, $T_3$ bude reprezentovat tah v druhé z nich a $T_2$ a $T_4$ zrušíme. Navíc musíme ošetřit, zda jsme neodstranili význačný výskyt počátečního vrcholu $e_1$ resp. $e_2$ a pokud ano, nahradit jej hranou následující po $e_2$, resp. $e_1$ v původním tahu; pokud neexistuje, vrchol se tím stal izolovaným a jeho hodnotu *designated* nastavíme na **null**.

Vkládání hran operací *Link*(**vertex** $v$, $w$) je o něco obtížnější. Označme si $T^v$ ET-strom obsahující $v$ a analogicky $T^w$ pro $w$. Nejprve předpokládejme, že $x$ ani $y$ nejsou izolované vrcholy. Nalezneme význačné výskyty $e_v$ a $e_w$ vrcholů $v$ a $w$. Nyní $T^v$ a $T^w$ rozdělíme následovně:

$$T_1^v = \langle T_v \text{ od začátku do předchůdce } e_v \rangle$$
$$T_2^v = \langle T_v \text{ od } e_v \text{ do konce} \rangle$$
$$T_1^w = \langle T_w \text{ od začátku do předchůdce } e_w \rangle$$
$$T_2^w = \langle T_w \text{ od } e_w \text{ do konce} \rangle$$

a sestrojíme výsledný tah spojením postupně $T_1^v$, nově založeného jednovrcholového stromu, $T_2^w$, $T_1^w$, druhého nového jednovrcholového stromu a $T_2^v$. Pokud byl $v$ nebo $w$ izolovaný (jeho *designated* byl **null**), použijeme místo jeho $T_1^*$ a $T_2^*$ prázdné stromy a nastavíme *designated* na nově vytvořenou hranu.

ET-strom je rovněž možno použít pro zjišťování, zda dva vrcholy leží v téže komponentě souvislosti – operaci *Query*(**vertex** $v$, $w$): nalezneme význačné výskyty vrcholů $v$ a $w$, z každého z nich vystoupíme do kořene příslušného $(a, b)$-stromu a pokud jsou kořeny různé, leží $v$ a $w$ v různých komponentách, jinak ve stejných.

**Věta 5.1.1:** *Operace Count a Query na ET-stromu mají časovou složitost $O(\log_a n)$, operace Link a Cut $O(a \cdot \log_a n)$. Celý ET-strom zabere prostor $O(a + n)$.*

**Důkaz:** *Count* a *Query* procházejí po cestě z daného vrcholu do kořene a hloubka $(a, b)$-stromu je vždy $O(\log_a n)$. *Link* provede $O(1)$ rozdělení a sloučení $(a, b)$ stromů, z nichž každé trvá $O(a \cdot \log_a n)$. Pokud navíc udržujeme přiřazení dvojic vrcholů hranám v $(a, b)$-stromu, spotřebujeme další čas $O(a \cdot \log_a n)$ na jeho aktualizaci. *Cut* nejprve v čase $O(\log_a n)$ nalezne a z pomocné struktury odstraní identifikátor hrany, poté provede $O(1)$ operací *Count*, rozdělení a sloučení $(a, b)$ stromů s časy $O(a \cdot \log_a n)$. Každý $(a, 2a)$-strom zabírá prostor $O(a + n)$, pomocné struktury pak $O(n)$. □

**Důsledek 5.1.2:** *Pokud jsme schopni dynamicky nebo semidynamicky udržovat kostru grafu tak, aby vložení resp. odebrání hrany v původním grafu bylo zpracováni v čase $f(n)$ a způsobilo $O(1)$ změn kostry, pak udržováním kostry v ET-stromech a odpovídáním na propojenost vrcholů podle ní získáme datovou strukturu stejného druhu pro souvislost pracující v čase $O(f(n) + a \cdot \log_a n)$ na Insert a Delete a $O(\log_a n)$ na Query pro libovolné $a$.*

**Důsledek 5.1.3:** *Ze semidynamické struktury pro udržování kostry pracující v čase $O(\log^2 n)$ (jako jsou ty, které jsme popsali v minulé kapitole) tak volbou $a = \log n$ získáme semidynamickou strukturu pro souvislost, která provádí modifikace grafu rovněž v čase $O(\log^2 n)$ a na dotazy odpovídá v čase $O(\log n / \log \log n)$. Pokud má původní struktura časovou složitost $O(n^\varepsilon)$, získáme volbou $a = n^\varepsilon$ stejný čas na modifikace a $O(1)$ na odpovědi.*

## 5.2. Nestromové hrany

ET-stromy je možno rozšířit tak, aby obsahovaly informace o všech hranách grafu. Budeme si udržovat kostru $T$ reprezentovaného grafu $G$ v ET-stromu (rozhodování o tom, které hrany jsou v kostře a jak se má kostra upravovat v případě přidávání či odebírání hran $G$, ponechme na uživateli struktury). Nestromové hrany $G$ připojíme k ET-stromu tak, že všechny význačné výskyty vrcholů $G$ v ET-stromu doplníme o seznamy nestromových hran incidentních s těmito vrcholy (pokud je $G$ neorientovaný, pak každou hranu zařadíme k oběma jejím krajním vrcholům) a do vnitřních vrcholů $(a, b)$-stromu přidáme navíc položku *ncount*$(w)$ udávající počet nestromových hran uložených v daném podstromu (tu budeme udržovat stejně jako *count*$(w)$) a seznam $N_w$ synů s nenulovým *ncount*em.

*InsertNontree*, jakož i *DeleteNontree* nestromové hrany zvládneme v čase $O(\log_a n)$ – stačí totiž vyhledat význačný výskyt obou krajních vrcholů, vložit, resp. odebrat hranu ze seznamu nestromových hran příslušných k tomuto výskytu a nakonec vystoupat do kořene $(a, b)$-stromu a přepočítat hodnoty *ncount*, což může znamenat vkládání a vyjímání vrcholů ze seznamů $N_w$. Pokud se na hranu při *DeleteNontree* neodkazujeme identifikátorem, musíme opět přičíst $O(a \cdot \log_a n)$ na práci s pomocnou strukturou pro vyhledávání hran. (Na rozdíl od „čistých“ ET-stromů musíme mít uloženo až $\Omega(n^2)$ identifikátorů, ale naštěstí $\log_a n^2 = O(\log_a n)$, takže asymptotická složitost zůstane zachována.)

Nadefinujme si nyní operaci **edge** *FindNontree*(**vertex** $v$), která nalezne nestromovou hranu incidentní s komponentou obsahující vrchol $v$. Bude fungovat tak, že nejprve nalezne význačný výskyt $v$, a pak vystoupí do kořene $w$ příslušného $(a, b)$ stromu. Pokud *ncount*$(w) = 0$, neexistuje v této komponentě žádná nestromová hrana a vrátí **null**, jinak nalezne libovolného syna $w'$ takového, že *ncount*$(w') > 0$ a bude v něm pokračovat stejným způsobem, až po $O(\log_a n)$ krocích nalezne list s neprázdným seznamem nestromových hran, jehož první položku vrátí jako výsledek.

Navíc potřebujeme operace *Link* a *Cut* na ET-stromech upravit tak, abychom při změně význačného výskytu vrcholu automaticky přesunuli seznam nestromových hran k novému výskytu, což je ovšem možno provést v konstantním čase.

**Věta 5.2.1:** *Operace InsertNontree a DeleteNontree trvají čas $O(a \cdot \log_a n)$, FindNontree trvá $O(\log_a n)$. Reprezentace $m$ nestromových hran zabere prostor $O(m + n)$.*

**Důkaz:** *InsertNontree* a *DeleteNontree* jsme již analyzovali. U *FindNontree* trvá hledání kořene $O(\log_a n)$ a poté každé nalezení syna s kladným *ncount*em zvládneme v konstantním čase díky seznamům $N_w$, takže celkem hledáním nestromové hrany strávíme čas $O(\log_a n)$.

K vrcholům $(a, b)$-stromu jsme přidali nové položky, kterých je dohromady $O(n)$. Každá nestromová hrana je reprezentována jedním nebo dvěma prvky seznamu, celkem tedy $O(m)$ paměťových buněk. □

## 5.3. Nestromové hrany podruhé

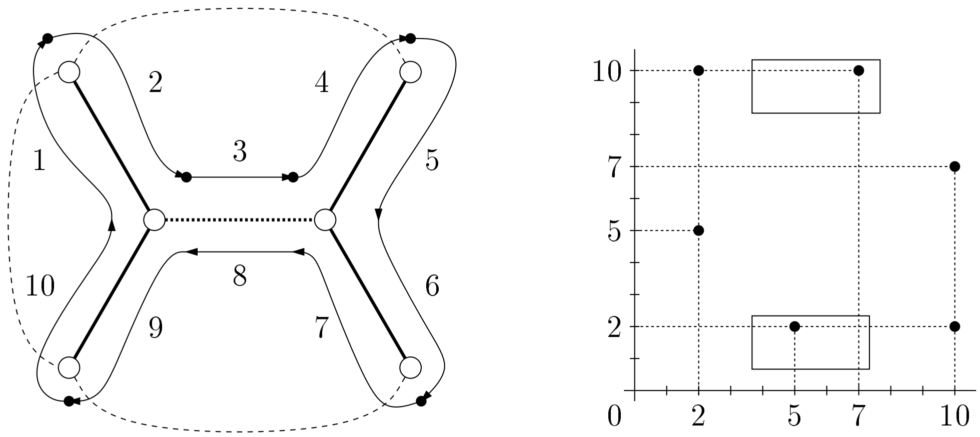
Při mazání stromových hran v našich kvazidynamických algoritmech jsme potřebovali nalézt nestromovou hranu, kterou je možno smazanou hranu nahradit, jinými slovy takovou, která ji pokrývá. V tom nám bohužel právě sestrojená struktura nepomůže, leda že bychom se spokojili s probíráním všech nestromových hran vedoucích z menší z obou komponent a u každé z nich otestovali, vede-li do té větší – to by ovšem mohlo zabrat až čas $\Omega(m)$. Zde se pokusíme tento problém redukovat na problém jiný, konkrétně na nalezení vhodné reprezentace bodů v rovině:

**Definice 5.3.1:** ***Plane shift strukturou*** *nazveme datovou strukturu reprezentující konečnou množinu $M$ mřížových bodů roviny $\mathbb{Z}^2$ ohodnocených prvky nějaké konečné pologrupy $(X, \oplus, \mathbf{0})$ s následujícími operacemi:*

- *InsertPoint*(**integer** $x$, $y$, **value** $v$) – *přidá do $M$ nový bod o souřadnicích $(x, y)$ a ohodnocení $v \in X$.*
- *DeletePoint*(**integer** $x$, $y$) – *odebere z $M$ bod o souřadnicích $(x, y)$.*
- **value** *RectSum*(**integer** $x_1$, $y_1$, $x_2$, $y_2$) – *spočte $\oplus$ z ohodnocení všech bodů z $M$ vyskytujících se v obdélníku s vrcholy $(x_1, y_1)$, $(x_2, y_1)$, $(x_2, y_2)$ a $(x_1, y_2)$, přičemž si může zvolit libovolné pořadí. V případě, že daný obdélník žádné body neobsahuje, výsledek je definitoricky* $\mathbf{0}$.
- **integer** *RectMaxX*(**integer** $x_1$, $y_1$, $x_2$, $y_2$) – *vrátí maximální x-ovou souřadnici bodu z $M$, který se vyskytuje v zadaném obdélníku. Analogicky jsou definovány operace RectMaxY, RectMinX a RectMinY.*
- *ShiftX*(**integer** $x_1$, $x_2$, $x_t$) – *každý bod $(x, y)$ takový, že $x_1 \leq x \leq x_2$ přesune na souřadnice $(x - x_1 + x_t, y)$ a ohodnocení ponechá. Jinými slovy pás roviny vymezený přímkami $x = x_1$ a $x = x_2$ přesune tak, aby byl omezen přímkami $x = x_t$ a $x = x_t + x_2 - x_1$. Přitom předpokládá, že v cílové oblasti se nevyskytují žádné body.*
- *ShiftY*(**integer** $y_1$, $y_2$, $y_t$) – *analogie ShiftX v druhé souřadnici.*

Na efektivní plane shift struktuře můžeme založit srovnatelně efektivní reprezentaci nestromových hran, která splňuje naše požadavky. Každou nestromovou hranu $\{v, w\}$ uložíme do plane shift struktury $\mathcal{R}$ jako dvojici bodů $(c(v), c(w))$ a $(c(w), c(v))$, kde $c(v)$ je pořadové číslo hrany *designated*$(v)$ v tahu.

Výhodou tohoto popisu je, že vyjmeme-li z kostry libovolnou hranu $e$, rozpadne se interval souřadnic přiřazený eulerovskému tahu komponenty na pět podintervalů (možno i prázdných), přičemž sjednocením některých z nich získáme množiny pořadových čísel $N_1$ a $N_2$ hran, které budou patřit do jedné, resp. druhé komponenty – viz diskuse o rozpadu tahu při operaci *Cut*. Body odpovídající nestromovým hranám, které spojují tyto dvě komponenty (jinými slovy pokrývají $e$), musí podle definice náležet do $N_1 \times N_2 \cup N_2 \times N_1$, což je ovšem sjednocení $O(1)$ obdélníků, takže k nalezení nějakého takového bodu můžeme použít *RectSum*, zvolíme-li za pologrupu $(X, \oplus, \mathbf{0})$ množinu všech identifikátorů hran



Obr. 12: Nestromové hrany → body v rovině

spolu s prvkem **0**, který neidentifikuje žádnou hranu, a s operací, jež vrátí svůj první nenulový operand (ta je určitě asociativní). Pokud takovou hranu nalezneme ($\mathit{RectSum} \neq \mathbf{0}$) a přidáme ji do kostry, přesuneme poté pomocí operací *ShiftX* a *ShiftY* naše body tak, aby odpovídaly nové kostře.

Obrázek 12 ukazuje, jak situace typicky vypadá: nakreslené body odpovídají nestromovým hranám vyobrazeného grafu. Pokud odstraníme čárkovanou hranu z kostry, budou hrany vedoucí mezi oběma komponentami odpovídat bodům ležícím uvnitř označených obdélníků (nalezneme tak jen jednu jejich orientaci, ale to stačí).

Tolik intuice, nyní vše nadefinujeme precizně. Jelikož se beztak musíme postarat o správný popis nesouvislých grafů, nebudeme nutně pracovat s kostrou, nýbrž s obecným acyklickým podgrafem (budeme mu říkat pseudokostra) a původní nestromové hrany pro nás nyní budou libovolné hrany, které do tohoto podgrafu nepatří (mohou tak spojovat i různé jeho komponenty).

**Definice 5.3.2:** ***Struktura pro nestromové hrany*** *(zkráceně* ***N-struktura****) reprezentuje graf a jeho pseudokostru, přičemž podporuje následující operace ($(X, \oplus, \mathbf{0})$ opět buďiž libovolná pologrupa):*

- *Link*(**vertex** $v$, $w$), *Cut*($v$, $w$) – *přidání, resp. odebrání hrany v pseudokostře.*
- **boolean** *IsTreeEdge*(**vertex** $v$, $w$) – *zjistí, zda $\{v, w\}$ je hranou pseudokostry či nikoliv.*
- *InsertNontree*(**vertex** $v$, $w$, **value** $x$), *DeleteNontree*(**vertex** $v$, $w$) – *přidání, resp. odebrání nestromové hrany s ohodnocením $x \in X$.*
- **value** *SumNontree*(**vertex** $v$, $w$) – *spočte součet ($\oplus$) ohodnocení všech nestromových hran vedoucích mezi vrcholy komponenty pseudokostry v níž leží $v$ a komponenty, v níž leží $w$, přičemž hrany zpracovává v libovolném pořadí a vrací* **0***, pokud žádná taková hrana neexistuje.*
- **boolean** *Query*(**vertex** $v$, $w$) – *zjistí, zda jsou vrcholy $v$ a $w$ v téže komponentě pseudokostry.*

Jak jsme již naznačili, pseudokostru budeme reprezentovat ET-stromem a nestromové hrany pomocí bodů v rovině uložených v plane shift struktuře. Abychom byli schopni tuto reprezentaci použít i pro nesouvislé grafy, očíslujeme si vrcholy $v_1, \ldots, v_n$ a pro $v_i$ rezervujeme interval souřadnic $[2n \cdot (i-1)+1, 2n \cdot i]$ a pokud bude z $v_i$ vycházet eulerovský tah některé komponenty, přiřadíme hranám tahu čísla z příslušného intervalu. Pro libovolnou komponentu lze tak snadno určit, který interval jí je přiřazen – stačí vyhledat index nejlevějšího vrcholu v ET-stromu komponenty, což zvládneme v čase $O(\log_a n)$. Pro každý vrchol grafu pak snadno spočteme jemu přidělenou souřadnici tak, že vyhledáme komponentu a k začátku jejího intervalu připočteme $Count(designated(v)) - 1$.

Změny kostry operacemi *Link* a *Cut* nicméně způsobují přečíslování souřadnic přidělených jednotlivým vrcholům grafu, takže potřebujeme plane shift strukturu aktualizovat. Místo toho, abychom znovu analyzovali, jak se při těchto operacích změní kostra a složitě u jednotlivých případů rozebírali, jak je třeba body přesunout, využijeme toho, že veškeré změny kostry jsou složeny z rozdělování a spojování příslušných $(a, b)$-stromů a body budeme přesouvat při těchto operacích.

Při spojování zjistíme, jaké intervaly souřadnic jsou přiřazeny vrcholům obou stromů a přesuneme interval druhého stromu těsně za interval prvního (vzhledem k tomu, že výsledný strom bude mít nejvýše $2n - 2$ vrcholů, určitě nepřekročíme hranice přiděleného prostoru):

| | |
|---|---|
| **function** *BeforeLink*(**tree** $A$, $B$); | *spojujeme stromy $A$ a $B$* |
| $\quad i \leftarrow index(First(A))$; | *index prvního vrcholu v $A$* |
| $\quad j \leftarrow index(First(B))$; | *index prvního vrcholu v $B$* |
| $\quad a_1 \leftarrow 2n \cdot (i - 1) + 1$; | *vrcholům $A$ jsou přiřazeny* |
| $\quad a_2 \leftarrow a_1 + Count(A)$; | *souřadnice $[a_1, a_2 - 1]$* |
| $\quad b_1 \leftarrow 2n \cdot (j - 1) + 1$; | *a vrcholům $B$ $[b_1, b_2]$* |
| $\quad b_2 \leftarrow b_1 + Count(B) - 1$; | |
| $\quad ShiftX(b_1, b_2, a_2)$; | *přesuneme v $x$* |
| $\quad ShiftY(b_1, b_2, a_2)$; | *přesuneme v $y$* |
| **end** | |

Analogicky ošetříme rozdělování stromů:

| | |
|---|---|
| **function** *BeforeSplit*(**tree** $A$, **vertex** $v$) | *rozdělujeme $A$ před vrcholem $v$* |
| $\quad i \leftarrow index(First(A))$; | *index prvního vrcholu v $A$* |
| $\quad j \leftarrow index(v)$; | *index vrcholu $v$* |
| $\quad a_1 \leftarrow 2n \cdot (i - 1) + 1$; | *v $A$ zůstanou vrcholy se souřadnicemi* |
| $\quad a_2 \leftarrow a_1 + Count(v) - 2$; | *$[a_1, a_2 - 1]$,* |
| $\quad b_1 \leftarrow 2n \cdot (j - 1) + 1$; | *zbytek přesuneme do $[b_1, \ldots]$* |
| $\quad ShiftX(a_2, a_1 + 2n - 1, b_1)$; | *přesuneme v $x$* |
| $\quad ShiftY(a_2, a_1 + 2n - 1, b_1)$; | *přesuneme v $y$* |
| **end** | |

*Link* či *Cut* způsobí $O(1)$ rozdělování a spojování stromů, a tak $O(1)$ operací *ShiftX* a *ShiftY* na $\mathcal{R}$ a $O(1)$ dotazů na $(a, b)$ strom vyhodnotitelných v čase $O(\log_a n)$.

Funkci *SumNontree*$(v, w)$ stačí spočíst, které intervaly $I_v$ a $I_w$ souřadnic jsou přiřazeny stromům pseudokostry, v nichž leží $v$, resp. $w$ a poté vyhodnotí *RectSum* přes obdélník $I_v \times I_w$. Pro testy *IsTreeEdge*$(v, w)$ si stačí pro každou hranu pamatovat, zda je tato hrana součástí pseudokostry či nikoliv (k tomu se nám opět hodí identifikátory hran).

**Věta 5.3.3:** *Existuje-li plane shift struktura $\mathcal{R}$ pracující s $m$ body v čase $T_I(m)$ na Insert a Delete, $T_S(m)$ na RectSum a $T_M(m)$ na ShiftX a ShiftY a využívající prostor $S(m)$, kde $T_I$, $T_S$, $T_M$ a $S$ jsou neklesající funkce, pak pro každé $a$ existuje struktura pro nestromové hrany reprezentující graf s $n$ vrcholy a $m$ hranami v prostoru $O(n{+}a{+}S(2m))$ a provádějící operace Link a Cut v čase $O(a \cdot \log_a n + T_M(2m))$, InsertNontree a DeleteNontree v čase $O(\log_a n{+}T_I(2m))$, SumNontree v čase $O(T_S(2m))$, IsTreeEdge a Query v čase $O(\log_a n)$.*

**Důkaz:** Prostorovou složitost získáme sečtením prostorové složitosti ET-stromu a plane shift struktury. Operace *Link* a *Cut* se skládají z původních operací na ET-stromech doplněných o $O(1)$ volání *ShiftX* a *ShiftY*. *InsertNontree* a *DeleteNontree* se překládají na dvě volání *Count* v ET-stromu a *InsertPoint* resp. *DeletePoint* v $\mathcal{R}$; *SumNontree* provede konstantní výpočet intervalu a *RectSum*. *IsTreeEdge* pouze vyhledává identifikátor hrany. □

## 5.4. Plně dynamická souvislost a minimální kostra

Právě zavedené struktury pro reprezentaci nestromových hran je možno použít k sestrojení jednoduchého plně dynamického algoritmu pro udržování komponent souvislosti – budeme udržovat kostru grafu a na dotazy odpovídat podle této kostry, pouze musíme při mazání hran kostry pomocí N-struktury vyhledávat pokrývající hranu. Za pologrupu $(X, \oplus, \mathbf{0})$ si zvolíme množinu identifikátorů hran spolu se speciálním prvkem $\mathbf{0}$, který neidentifikuje žádnou hranu, a operací $\oplus$, jež vrátí svůj první nenulový operand.

*Query*(**vertex** $v$, $w$) je zpracováno přímo N-strukturou.

*Insert*(**vertex** $v$, $w$) probíhá tak, že nejprve pomocí *Query*($x$, $y$) zjistí, zda jsou $v$ a $w$ ve stejné komponentě. Pokud ne, je nově přidaná hrana mostem, takže stačí operací *Link*($v$, $w$) spojit kostry obou komponent. Pakliže je již $v \leftrightarrow w$, hranu přidáváme jako nestromovou pomocí *InsertNontree*$(x, y, (x, y))$.

*Delete*(**vertex** $v$, $w$) zprvu otestuje voláním *IsTreeEdge*($v$, $w$), zda mažeme stromovou hranu či nikoliv. Pokud ne, stačí zavolat *DeleteNontree*($v$, $w$). V opačném případě pomocí *Cut*($v$, $w$) hranu vyjmeme z kostry. Poté za ni zkusíme nalézt náhradu voláním operace *SumNontree*($v$, $w$) a pokud náhrada existuje, operací *Link* ji do pseudokostry přidáme a pomocí *DeleteNontree* náhradu vyjmeme z množiny nestromových hran, a tak vznikne opět kostra.

**Věta 5.4.1:** *Pokud každá operace InsertNontree, DeleteNontree, SumNontree, Link a Cut na struktuře pro nestromové hrany trvá čas $O(T_X(m,n))$ a operace Query trvá $O(T_Q(m,n))$, to vše v paměti $O(S(m,n))$, pak dynamický algoritmus pro souvislost pracuje v paměti $O(S(m,n))$ a operace Insert a Delete zpracovává s časovou složitostí $O(T_X(m,n) + T_Q(m,n))$ a operaci Query s $O(T_Q(m,n))$.*

**Důkaz:** Prostým sečtením časů operací na ET-stromu, které používáme. □

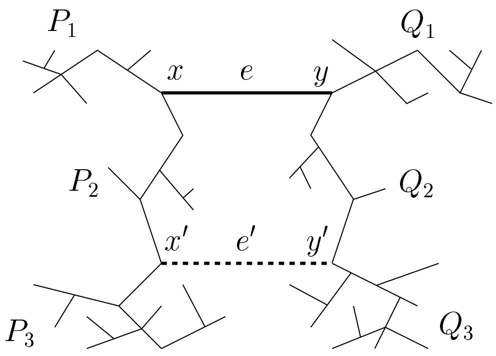
**Důsledek 5.4.2:** *Tento algoritmus je rovněž možno upravit tak, aby udržoval minimální kostru grafu – stačí při vyhledávání náhrady hrany použít pokrývající hranu s nejmenší možnou cenou. Toho docílíme tím, že operaci $\oplus$ předefinujeme tak, aby, dostane-li oba operandy (identifikátory hran) nenulové, vrátila jako výsledek hranu s nižší váhou. Všechny operace na ET-stromu pak snadno doplníme o vytváření seznamu změn kostry, který tvoří výstup algoritmu.*

# 6. Plně dynamická 2-souvislost

Majíce vybudovány struktury pro reprezentaci grafů včetně nestromových hran, přikročíme nyní k algoritmům pro plně dynamickou hranovou 2-souvislost.

## 6.1. Naivní algoritmus

První myšlenkou, která se nabízí, je stejně jako v případě souvislosti jednoduše rozšířit kvazidynamický algoritmus z kapitoly 3.4, to jest ošetřit mazání stromových hran. Vyhledání nahrazující hrany jsme již vyřešili v plně dynamickém algoritmu pro souvislost, pokrytí této hrany jsme schopni zjistit snadno – stačí nechat N-strukturu spočíst všechny nestromové hrany mezi oběma nově vzniklými komponentami pseudokostry (za pologrupu zvolíme přirozená čísla se sčítáním a všem hranám přiřadíme jednotkové ohodnocení). Mimo to se ovšem změní i pokrytí ostatních hran kostry.




Obr. 13: Změna pokrytí při nahrazení stromové hrany

Celá situace je naznačena na obr. 13: nahrazujme stromovou hranu $e = \{x, y\}$ hranou $e' = \{x', y'\}$ a pozorujme, jak se mění, které stromové hrany jsou pokryty kterými nestromovými. Odebereme-li vrcholy $x$, $x'$, $y$ a $y'$, kostra se rozpadne na několik komponent souvislosti: $P_2$ obsahující cestu mezi $x$ a $x'$, $Q_2$ obsahující cestu z $y$ do $y'$, $P_1$ obsahující komponenty všech vrcholů původně incidentních s $x$, které ale nepatří do $P_2$; $P_3$, $Q_1$ a $Q_3$ analogicky pro $x'$, $y$ a $y'$.

Nestromových hran mezi vrcholy cest $P_i$ a $P_j$ či $Q_i$ a $Q_j$ se změna kostry netýká, pokrývají stále tytéž hrany kostry.

Vede-li nějaká nestromová hrana mezi vrcholy $a \in P_1$ a $b \in Q_3$, pokrývala původně cestu $a \ldots x$, pak hranu $e$ a cesty $y \ldots y'$ a $y' \ldots b$. Po výměně $e'$ za $e$ bude pokrývat cesty $a \ldots x$ a $x \ldots x'$, hranu $e'$ a cestu $y' \ldots b$. Pro každou nestromovou hranu tohoto druhu se tedy o 1 zvýší pokrytí cesty $x \ldots x'$ o 1 sníží pokrytí $y \ldots y'$. To je možné snadno ošetřit tím, že si od struktury pro nestromové hrany necháme spočítat, kolik takových hran mezi $P_1$ a $Q_3$ existuje a reprezentujíce pokrytí v dynamickém stromu, v čase $O(\log n)$ opravíme jeho hodnoty. Stejně můžeme ošetřit hrany mezi $P_1$ či $P_3$ a $Q_1$ či $Q_3$.

Problémem ovšem jsou ostatní hrany (incidentní s $P_2$ nebo $Q_2$) – kupříkladu hrana mezi $a \in P_2$ a $b \in Q_2$ původně pokrývala cestu $a \ldots x$, hranu $e$ a cestu $y \ldots b$, nyní bude pokrývat $a \ldots x'$, $e'$ a $y' \ldots b$. Na cestách $x \ldots x'$ a $y \ldots y'$ se tedy změní pokrytí, a to pro každou hranu jinak, což bohužel nejsme schopni počítáním nestromových hran mezi částmi kostry a aktualizací informací o cestách v dynamických stromech efektivně zohlednit. TTNPN.*

* Tak Tady Nám Pšenka Nepokvete, jak by řekl klasik.

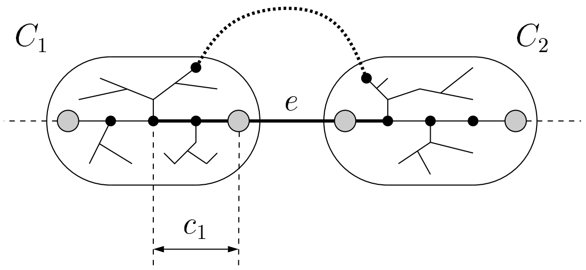
## 6.2. Pokrytí částečných a úplných cest

Vzhledem k uvedeným problémům zvolíme jinou reprezentaci pokrytí stromových hran, a to na základě Fredericksonovy clusterizace. Naše struktura je založena na přístupu z [F97], místo komplikovaných dvourozměrných topologických stromů a ambivalentních datových struktur však použijeme naši strukturu pro nestromové hrany.

Jelikož clusterizace funguje pouze pro 3-omezené grafy, nejprve podle věty 4.1.1 původní graf $G_0$ transformujeme na 3-omezený graf $G$ a každý *Insert*, *Delete* resp. *Query* na $G_0$ budeme překládat na $O(1)$ analogických operací na $G$. Dále tedy budeme předpokládat, že graf, s nímž pracujeme, je 3-omezený, nepříjemným důsledkem ale je, že $G$ má $O(E(G_0))$ vrcholů a řádově stejně hran, což budeme muset zohlednit v odhadech časové složitosti.

Nalezneme kostru $T$ grafu $G$ a budeme ji uchovávat jednak jako ET-strom doplněný o N-strukturu a jednak jako topologický strom. Pro každý cluster $C$ rekursivní clusterizace určené tímto topologickým stromem si budeme pamatovat, které hrany částečné a úplné cesty tohoto clusteru jsou pokryty a které nikoliv – v případě částečných cest pokrýváme pouze nestromovými hranami mezi vrcholy $G(C)$, v případě úplných všemi hranami grafu. Těchto informací o hranách je ovšem mnoho a bylo by obtížné je udržovat pro každý cluster odděleně, budeme se proto snažit využít toho, že částečné a úplné cesty obsahují částečné cesty dílčích clusterů.

Ke každé cestě (ať již částečné nebo úplné) si budeme uchovávat binární strom, jehož listy (ty budeme kreslit jako hranaté vrcholy) odpovídají v inorderovém pořadí hranám cesty a vnitřní vrcholy (kulaté) obsahují pomocné informace pro vyhledávání. Přitom strom odpovídající cestě clusteru na úrovni $k$ může obsahovat jako své podstromy stromy cest clusterů úrovně $k-1$. Musíme si ale dát pozor na to, abychom v případě změn struktury stromů aktualizovali i všechny nadřazené stromy. Ty odpovídají clusterům, které v topologickém stromu vždy leží na cestě z aktualizovaného clusteru do kořene.



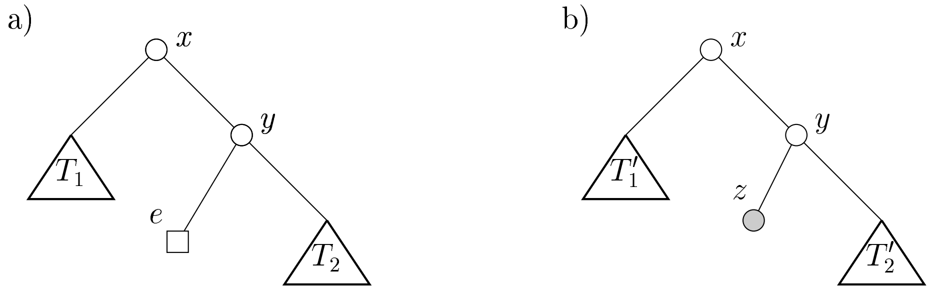

Obr. 14: Pokrývání částečných cest novými hranami

Tyto stromy budeme konstruovat rekursivně, podobně jako jsme definovali částečné a úplné cesty. Pro triviální clustery úrovně 0 zkonstruujeme prázdné stromy, v clusterech velikosti 1 pouze zdědíme informace z nižší úrovně. Pokud spojujeme hranou $e$ dva clustery $C_1$ a $C_2$ do jednoho, potřebujeme spočíst pokrytí hrany $e$, spojit stromy pro částečné cesty v $C_1$ a $C_2$ a nový strom pro hranu $e$ a upravit pokrytí podle nově přibyvších hran: spojujeme-li cluster stupně 3 s clusterem stupně 1 (BÚNO $D(G_1) = 3$ a $D(G_2) = 1$), jsou to nestromové hrany vedoucí mezi vrcholy $G(C_2)$ a zbytkem grafu, v ostatních případech hrany mezi vrcholy $G(C_1)$ a $G(C_2)$. Pakliže jsou $C_1$ i $C_2$ lichého stupně, popisuje vzniklý strom úplnou cestu $C$ a částečná cesta je triviální, v opačném případě popisuje částečnou cestu a úplná cesta je prázdná. Nově pokryté hrany tvoří na vzniklé cestě souvislý úsek – hrany uvnitř $G(C_1)$ musí být pokryty nestromovou hranou z $G(C_1)$ do $G(C_2)$ (či do zbytku grafu v případě „3+1“), a tak stačí nalézt první pokrytou hranu na částečné cestě

$C_1$ a vše od ní až do konce této částečné cesty je pokryto, analogicky pro $C_2$ a poslední pokrytou hranu (viz obr. 14).

Abychom mohli ve stromu efektivně vyhledávat pokrytí libovolné hrany v čase úměrném hloubce stromu, doplníme si ještě do kulatých vrcholů několik pomocných údajů: délku segmentu $\mathit{length}(w)$, bit $\mathit{allcover}(w)$, který říká, zda segment cesty popsaný tímto podstromem je celý pokryt (v takovém případě jsou hodnoty v ostatních vrcholech podstromu irelevantní) a bit $\mathit{reverse}(w)$ indikující, že v celém podstromu prohazujeme levé a pravé syny (stejně jako u Sleator-Tarjanových stromů). U listů stromu předpokládejme $\mathit{length}(w) = 1$ a $\mathit{allcover}(w) = 0$, u prázdných podstromů $\mathit{length}(\emptyset) = 0$ a $\mathit{allcover}(\emptyset) = 1$. V čase $O(\mathit{hloubka\ stromu})$ jsme pomocí těchto informací schopni zjistit, zda je $i$-tá hrana pokryta (ošetřování $\mathit{reverse}(w)$ budeme vynechávat, stačí v případě, že je tento bit nastaven, prohodit levého a pravého syna a *reverse* u tohoto vrcholu a u jeho synů znegovat – to je ekvivalentní úprava):

```
function Covered(treevertex w, integer i);
    if i < 1 ∨ i > length(w) → return 0;          je-li i mimo rozsah, konec
    if allcover(w) → return 1;                    celý segment pokryt
    if i ≤ length(left(w)) →                      levý podsegment
        return Covered(left(w), i);
    else                                          pravý podsegment
        return Covered(right(w), i − length(left(w)));
end
```




Obr. 15: Spojování stromů reprezentujících pokrytí částečných cest

Přepokládejme prozatím, že jsme schopni zjistit, zda je $e$ pokryta a kolik hran na částečných cestách $C_1$ a $C_2$ je pokryto (to si označíme $c_1$ a $c_2$) – detaily vyřešíme později. Sestavování stromů při spojování clusterů tak zařídíme snadno (viz obr. 15):

a) pokud je hrana $e$ nepokryta (tehdy nemohou být nově pokryty ani žádné další hrany), založíme kulaté vrcholy $x$ a $y$ a hranatý vrchol odpovídající hraně $e$. Připojíme již hotové podstromy $T_1$ a $T_2$ částečných cest clusterů $C_1$ a $C_2$ tak, jak je naznačeno na obrázku 15, a přeorientujeme je tak, aby poslední vrchol $T_1$ a první vrchol $T_2$ incidovaly s $e$ – pokud potřebujeme v $T_i$ otočit orientaci, zkopírujeme jeho kořen a kopii znegujeme bit *reverse*. Poté nastavíme: ($r_i$ je kořenem $T_i$)

$$\mathit{allcover}(x) = \mathit{allcover}(y) = 0$$
$$\mathit{length}(y) = \mathit{length}(r_2) + 1$$
$$\mathit{length}(x) = \mathit{length}(r_1) + \mathit{length}(y)$$

b) pokud je hrana $e$ pokryta, může být nově pokryta i část částečných cest clusterů $C_1$ a $C_2$ (jak již víme, jedná se o souvislý úsek délky $c_i$ v $C_i$). Proto sestrojíme stromy $T_i'$ odpovídající $T_i$ bez nově pokrytých $c_i$ hran a založíme nové kulaté vrcholy $x$, $y$ a $z$, přičemž $z$ bude reprezentovat nově pokrytý úsek a $x$ a $y$ budou sloužit ke spojování segmentů. Opět připojíme podstromy podle obr. 15 a nastavíme: ($r_i'$ je kořenem $C_i'$)

$$\begin{aligned} allcover(z) &= 1 \\ allcover(y) &= allcover(r_2') \\ allcover(x) &= allcover(r_1') \wedge allcover(y) \\ length(z) &= c_1 + c_2 + 1 \\ length(y) &= length(z) + length(r_2') \\ length(x) &= length(r_1') + length(y) \end{aligned}$$

$T_1'$ zkonstruujeme z $T_1$ odstraněním $c_1$ hran zprava pomocí následující funkce $CutRight(r_1, c_1)$:

```
function CutRight(treevertex w, integer i)
    if i = 0 → return w;                                 neodstraňujeme nic
    if i ≥ length(w) → return ∅;                         všechny vrcholy pryč
    if i ≥ length(right(w)) →                            celý pravý podstrom pryč
        return CutRight(left(w), i − length(right(w)));
    else                                                 jen část pravého
        z ← NewVertex;                                   nový vnitřní vrchol
        l ← left(z) ← left(w);                           levý podstrom zachován
        r ← right(z) ← CutRight(right(w), i);            pravý ořízneme
        length(z) ← length(l) + length(r);
        allcover(z) ← allcover(l) ∧ allcover(r);
    return z;
end
```

Funkce *CutRight* zpracuje nejvýše jeden vrchol v každém patře stromu $T_1$ a případně jej nahradí jiným vrcholem. Proto proběhne v čase lineárním s hloubkou stromu a tuto hloubku zachová. Strom $T_2'$ sestrojíme pomocí funkce *CutLeft* symetrické ke *CutRight*.

**Lemma 6.2.1:** *Hloubka stromů reprezentujících částečné a úplné cesty je logaritmická. Pro každý cluster jsme schopni stromy jeho cest sestrojit v čase $O(\log n)$, známe-li již stromy částečných cest clusterů, z nichž je složen.*

**Důkaz:** Konstruujeme-li strom nějaké cesty uvedeným způsobem, vzroste jeho hloubka maximálně o dvě patra za každé spojení clusterů. To se ovšem může stát nejvýše $O(\log n)$-krát, z čehož získáme logaritmický odhad hloubky stromu. Konstrukce stromu cesty sestává z konstantní práce a $O(1)$ volání funkcí *CutRight* a *CutLeft*, což dává celkový čas $O(\log n)$. □

## 6.3. Vyhledávání mostů

Z věty 4.4.4 víme, že libovolnou stromovou cestu mezi dvěma vrcholy je možno rozložit na $O(\log n)$ částečných cest a samostatných hran, ale bohužel nestačí prozkoumat stromy těchto částečných cest, jelikož hrany cest mohou být pokryty i nestromovými hranami ležicími mimo daný cluster. Pro úplné cesty se to ovšem stát nemůže – informace o jejich pokrytí, které máme, jsou definitivní. A tak můžeme ke každé částečné cestě i ke každé samostatné hraně vyhledat, které úplné cesty je součástí a určit pokrytí podle stromu této úplné cesty. To provedeme v čase $O(\log n)$, takže celé vyhodnocení dotazu trvá čas $O(\log^2 n)$. Existuje ovšem způsob, jak to provést rychleji, a ten zde popíšeme detailně.

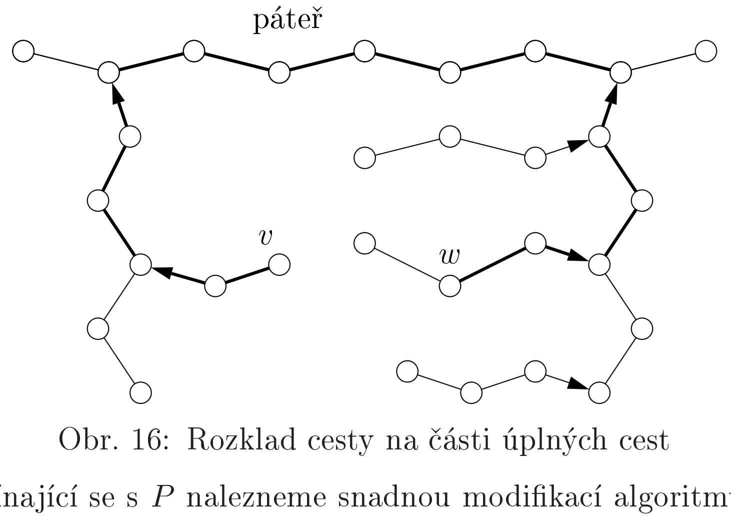
Pro každou úplnou cestu $P$, která nepatří ke clusteru stupně 0 (takový cluster je v topologickém stromu každé komponenty souvislosti právě jeden a je to jeho kořen), si zapamatujeme, kolik hran od vršku cesty je pokryto, než přijde první nepokrytá. Této hodnotě budeme říkat *topcover*. Pokud bude $topcover = 0$, je hned první hrana (incidentní s vrškem cesty) mostem, pokud $topcover = délka\ cesty$, jsou všechny hrany na cestě pokryty.

Hodnotu *topcover* můžeme pro libovolnou úplnou cestu zjistit prohledáním jejího stromu do hloubky. Pokud je vrškem cesty její první vrchol, spočteme pokryté hrany zleva následující funkcí *LeftCover*, v opačném případě funkcí *RightCover* k ní symetrickou.

```
function LeftCover(treevertex w);
    if leaf(w) → return 0;                                    je-li v listem, končíme
    if allcover(w) → return length(w);                        celý segment pokryt
    if allcover(left(w)) →                                    levý podsegment celý pokryt
        return length(left(w)) + LeftCover(right(w));
    else                                                      první nepokrytá hrana vlevo
        return LeftCover(left(w));
end
```

Tato funkce tráví na každé úrovni stromu cesty čas omezený konstantou, dohromady tedy $O(\log n)$.

S pomocí hodnot *topcover* můžeme rozhodnout, zda jsou dané 2 vrcholy $v$ a $w$ 2-propojeny, tedy zda je každá hrana cesty $P$ v kostře mezi $v$ a $w$ pokryta. Nalezneme nejprve nejnižšího společného předka $C$ vrcholů $v$ a $w$ v topologickém stromu. To je nějaký cluster, který vznikl spojením dvou clusterů nižší úrovně hranou obsaženou v cestě $P$ (tento argument jsme použili již jednou v důkazu věty 4.4.4) a celá cesta $P$ je obsažena v expanzi $C$, to znamená v částečné nebo úplné cestě clusteru $C$, čili v úplné cestě některého z nadřazených clusterů (té budeme říkat páteř cesty $P$) a v úplných cestách všech clusterů ležících v topologickém stromu pod $C$ (viz lemma 4.4.3).




Obr. 16: Rozklad cesty na části úplných cest

Úplné cesty protínající se s $P$ nalezneme snadnou modifikací algoritmu pro vyhledávání rozkladu cest (viz věta 4.4.4 a obr. 16). Budeme opět postupovat v topologickém stromu od listů směrem ke společnému předkovi, ale jednotlivé částečné cesty v rámci jedné úplné cesty si budeme akumulovat, využívajíce toho, že na sebe navazují a že vždy tvoří souvislý úsek od vrcholu, kde jsme na úplnou cestu vstoupili (tato místa si zapamatujeme, budeme je potřebovat), až k jejímu vršku. Rovněž si zapamatujeme místa, kde se poslední úplná cesta ve směru od $v$, resp. od $w$ dotýká páteře (pokud vrchol $v$ či $w$ leží přímo na páteři, je sám dotykovým bodem).

Pro páteř pomocí její stromové reprezentace zjistíme, zda je úsek mezi oběma dotykovými body pokryt (přímočarou modifikací funkce *LeftCover*, získáme při tom i případný most na páteři). Pokud pokryt je, ověříme ještě pokrytí průniků $P$ s jednotlivými úplnými cestami. Tyto průniky jsou ovšem souvislé části od dotykového bodu předchozí úplné cesty k vršku cesty, takže stačí porovnat *topcover* cesty s pozicí dotykového bodu. Pokud je *topcover* menší, udává opět polohu mostu.

**Lemma 6.3.1:** *2-propojenost dvou vrcholů lze zjistit v čase $O(\log n)$ pomocí stromů úplných cest a pomocných informací, které je ke každému stromu cesty možno spočíst v čase $O(\log n)$. Pokud zadané vrcholy nejsou 2-propojeny, nalezneme při tom rovněž některý z mostů, které je oddělují.*

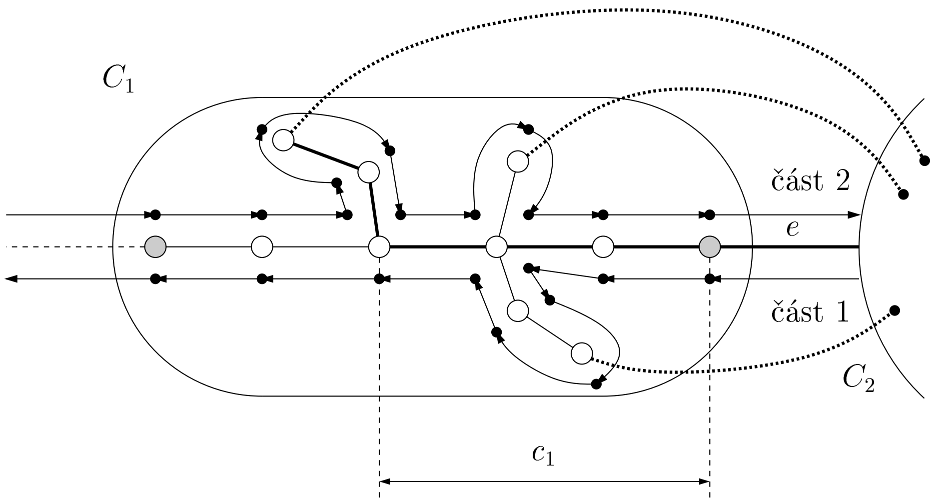
**Důkaz:** Jediné pomocné informace, které potřebujeme, jsou *topcover*y a ty, jak jsme již ukázali, lze v požadovaném čase spočítat. 2-propojenost pak testujeme rozkladem cesty na páteř a části úplných cest ($O(\log n)$), kontrolou páteře ($O(\log n)$) a kontrolou úplných cest ($O(1)$ na každou, celkově $O(\log n)$). Oba typy kontrol navíc v případě negativního výsledku vrací některý z mostů (je ovšem těžké charakterizovat, který). □

## 6.4. Aktualizace struktury

K dynamickému udržování naší 2-souvislostní struktury upravíme plně dynamický algoritmus pro 1-souvislost, který jsme odvodili v kapitole 5.4. Kostru grafu nyní budeme ukládat nejen jako ET-strom, ale také jako Fredericksonův topologický strom.

Nejprve potřebujeme vyřešit, jak v případě změny kostry a následné modifikace topologického stromu, aby odpovídal nové kostře, přepočítat stromy reprezentující všechny dotčené částečné a úplné cesty. Připomeňme si algoritmus pro udržování topologického stromu uvedený v algoritmu 4.3.1: Postupujeme ve stromu zdola nahorů a na každé úrovni nějaké clustery založíme, jiné zrušíme a další aktualizujeme. Pro každý založený či aktualizovaný cluster během toho můžeme přepočíst všechny údaje, které potřebujeme pro 2-souvislost, tedy strom částečné a úplné cesty a připadně *topcover*. Přitom víme, že clustery ležící v topologickém stromu pod právě přepočítávaným clusterem již obsahují aktuální informace, takže můžeme výše popsaným inkrementálním algoritmem pro konstrukci stromů cest tyto stromy pro daný cluster znovu sestrojit a poté podle stromu úplné cesty spočítat *topcover* tak, jak bylo odvozeno v předchozí kapitole. Tak při každém přidání nebo ubrání stromové hrany strávíme čas $O(\log n)$ aktualizací topologického stromu podle věty 4.3.2 a následně přepočteme $O(\log n)$ clusterů.

Pakliže nahrazujeme strom cesty nově zkonstruovaným stromem, můžeme paměť spotřebovanou tím původním ihned uvolnit, pokud si pamatujeme, které vrcholy byly vytvořeny pro tento strom a které zděděny z nižších pater. Nadřazené stromy, které obsahují součásti právě rušeného stromu, se tím sice stanou nekonzistentními, ale to nevadí, protože stejně budou vzápětí přepočítány. Druhou možností je místo toho udržovat u každého vrcholu počet odkazů a automaticky vrchol uvolňovat, kdykoliv tento počet klesne na nulu. Tím si ušetříme komplikace s rozpoznáváním zděděných vrcholů za cenu udržování počítadel. Zmíněný inkrementální algoritmus navíc pro svoji činnost potřebuje vědět, které hrany jsou nově pokryty. Přesněji, spojujeme-li clustery $C_1$ a $C_2$ do clusteru $C$ hranou $e$, chceme znát následující informace:




Obr. 17: Hledání pokrytého úseku pomocí eulerovského číslování

- Pokrytí hrany $e$: to nám N-struktura zodpoví snadno: dočasně odstraníme $e$ a hrany vedoucí z $G(C)$ do zbytku grafu (těch je $D(C) \leq 2$) a položíme struktuře dotaz na existenci hrany mezi komponentami $G(C_1)$ a $G(C_2)$, což je přesně hrana, která by pokrývala $e$. Poté odstraněné hrany vrátíme zpět. (Místo odstraňování a přidávání hran můžeme využít toho, že hrany $G(C_1)$ i $G(C_2)$ tvoří v eulerovském tahu $O(1)$ intervalů a převést tento dotaz na $O(1)$ obdélníkových dotazů v plane shift struktuře.)
- Počet $c_1$ nově pokrytých hran na konci částečné cesty clusteru $C_1$, jinými slovy první vrchol na této cestě, z nějž či z vrcholů připojených k němu hranami mimo částečnou cestu vede nestromová hrana buďto do $G(C_2)$ nebo do celého zbytku grafu (pokud $D(C_2) = 3$ a konstruujeme úplnou cestu), což je v každém případě podgraf $Z$ kostry, jehož čísla hran podle eulerovského tahu tvoří $O(1)$ intervalů (tah může vstoupit do $Z$ a vystoupit z něj pouze na hranách incidentních se $Z$, a ty jsou maximálně tři). Rovněž tak čísla hran $G(C_1)$ tvoří intervaly: jeden pro část tahu od hrany $e$ k vnější hraně vedoucí mimo $G(C)$, druhý pro opačný směr (některá z těchto částí může být ve skutečnosti očíslována dvěma intervaly, pokud v ní leží místo, kde jsme původní uzavřený tah rozpojili, ale to nevadí, pouze se tím může zvýšit celkový počet intervalů).

  Hledaná nestromová hrana musí vést buďto z vrcholu očíslovaného největším možným význačným výskytem hrany z první části tahu nebo naopak nejmenším možným z druhé části (poloha vrcholu, ve kterém se podstrom obsahující krajní vrchol nestromové hrany dotýká částečné cesty, je totiž v každém intervalu monotónní vzhledem k očíslování hran v eulerovském tahu; viz obr. 17). Stačí nám tedy v každém obdélníku kartézského součinu intervalů tvořících čísla hran $G(C_1)$ a intervalů tvořících hrany podgrafu $Z$ (těch je dohromady $O(1)$) určit nejlevější, resp. nejpravější bod, což plane shift struktura zvládne pomocí operace *RectMinX* resp. *RectMaxX*.
- Počet $c_2$ nově pokrytých hran na začátku částečné cesty clusteru $C_2$: symetricky.

Nyní již můžeme vyslovit následující lemma:

**Lemma 6.4.1:** *Informace potřebné pro udržování 2-souvislosti pro libovolný cluster za předpokladu, že pro jeho subclustery je již známe, přepočteme v čase $O(\log n)$ plus $O(1)$ dotazů na strukturu pro nestromové hrany a $O(1)$ obdélníkových dotazů na plane* *shift strukturu. Při každé základní operaci s topologickým stromem těchto přepočtení provedeme $O(\log n)$.*

Nakonec popíšeme, jak zrealizujeme jednotlivé operace (stále předpokládáme, že graf, se kterým pracujeme, je 3-omezený):

- **boolean** $Query_1$(**vertex** $v$, $w$) – dotazy na 1-propojenost vyhodnotíme stejně jako u plně dynamické struktury pro 1-souvislost, tedy dotazem na ET-strom kostry.
- **boolean** $Query_2$(**vertex** $v$, $w$) – dotaz na 2-propojenost vyhodnotíme algoritmem uvedeným v minulé kapitole pomocí stromů cest a hodnot *topcover*, jež si udržujeme.
- **edge** *Bridge*(**vertex** $v$, $w$) – most oddělující $v$ a $w$ nalezneme jako vedlejší produkt operace $Query_2(v, w)$.
- *Insert*(**vertex** $v$, $w$) – při vkládání hrany nejprve pomocí $Query(v, w)$ zjistíme, zda spojuje různé 1-komponenty grafu či nikoliv.
    - spojuje: hrana je novým mostem. Přidáme ji do kostry, stejně jako u algoritmu pro 1-souvislost aktualizujeme ET-strom a strukturu pro nestromové hrany, navíc ještě aktualizujeme topologický strom a podle lemmatu 6.4.1 tak způsobíme přepočtení pomocných dat (stromů cest a *topcover*ů) pro $O(\log n)$ clusterů.
    - nespojuje: hrana může nově pokrývat některé hrany kostry, takže musíme přepočítat pomocná data pro všechny clustery, které vzniknou spojením dvou podclusterů $C_1$ a $C_2$ takových, že hrana $\{v, w\}$ vede z $G(C_i)$ do $G - G(C_i)$. Clustery tohoto druhu se ovšem mohou vyskytovat pouze na cestě v topologickém stromu z $v$ či $w$ do kořene, takže jich je $O(\log n)$.
- *Delete*(**vertex** $v$, $w$) – rozdělíme na 3 případy, opět analogicky s 1-souvislostním algoritmem:
    - mažeme nestromovou hranu: hrana pokrývala část kostry, ošetříme stejně jako při vkládání nestromové hrany.
    - mažeme stromovou hranu a náhrada existuje (to zjistíme dotazem na strukturu pro nestromové hrany): odstraníme hranu $\{v, w\}$ z topologického stromu a přidáme do něj její náhradu. Při tom přepočteme pomocná data v $O(\log n)$ clusterech.
    - mažeme most (stromová hrana a náhrada neexistuje): pouze je odstraníme z topologického stromu, což nás opět stojí přepočet pomocných dat v $O(\log n)$ clusterech.

**Věta 6.4.2:** *Nechť všechny operace InsertNontree, DeleteNontree, SumNontree, Link a Cut na struktuře pro nestromové hrany a operace RectMinX a RectMaxX na v ní použité plane shift struktuře trvají čas $T_X(m,n)$ a operace Query trvá $T_Q(m,n)$, to vše v paměti $S(m,n)$, kde $T_X$, $T_Q$ a $S$ jsou neklesající funkce. Potom plně dynamický algoritmus pro 2-souvislost 3-omezených grafů pracuje v paměti $O(n + S(m,n))$, operace Insert a Delete zpracovává s časovou složitostí $O(\log n \cdot (\log n + T_X(m,n)))$, operaci $Query_1$ s $O(T_Q(m,n))$ a operaci $Query_2$ s $O(\log n)$.*

**Důkaz:** Každá operace *Insert* či *Delete* způsobí $O(1)$ operací s topologickým stromem, které trvají $O(\log n)$, $O(1)$ modifikací struktury pro nestromové hrany ($T_X(m,n)$ každá) a přepočet pomocných údajů o $O(\log n)$ clusterech, přičemž pro každý cluster strávíme čas $O(\log n)$ prací se stromy cest a provedeme $O(1)$ dotazů na nestromové hrany a na plane shift strukturu. Celkem tedy $O(\log n) + O(T_X(m,n)) + O(\log n) \cdot (O(\log n) + O(T_X(m,n))) = O(\log n \cdot (\log n + T_X(m,n)))$.

Prostorové nároky jsou tvořeny velikostí topologického stromu $O(n)$, velikostí všech stromů cest (opět $O(n)$) a prostorem potřebným pro struktury reprezentující nestromové hrany, což celkově dává uvedený odhad. $\square$

# 7. Body v rovině

V předchozích dvou kapitolách jsme převedli plně dynamické udržování komponent souvislosti a 2-souvislosti na udržování bodů v rovině (plane shift strukturu, viz definice 5.3.1). Nyní popíšeme jednu z možných datových struktur pro tento problém a ukážeme, jaké z ní plynou odhady složitosti našich plně dynamických grafových algoritmů.

## 7.1. Plane shift struktura

Připomeňme, že podle definice 5.3.1 má plane shift struktura reprezentovat množinu mřížových bodů roviny ohodnocených prvky nějaké konečné pologrupy a umět jednak do této množiny přidávat a z ní odstraňovat jednotlivé body (operace *InsertPoint* a *DeletePoint*), jednak zodpovídat dotazy na průnik této množiny s obdélníky (*RectSum*, *RectMaxX* apod.) a rovněž přesouvat souřadnice bodů (*ShiftX*, *ShiftY*).

Reprezentované body $(x_1, y_1), \ldots, (x_n, y_n)$ rozdělíme do příhrádek $P_1, \ldots, P_k$ tak, aby platily následující invarianty:

(i) $a < b, (x_i, y_i) \in P_a, (x_j, y_j) \in P_b \Longrightarrow y_i < y_j$
(příhrádky tvoří pásy roviny omezené rovnoběžkami s osou $x$ a jsou uspořádány vzestupně podle $y$-ové souřadnice)

(ii) $(x_i, y_i) \in P_a, (x_j, y_j) \in P_a, y_i \neq y_j \Longrightarrow |P_a| < z$
(příhrádky, které obsahují body s různými $y$-ovými souřadnicemi, mají velikost nejvýše $z$; hodnotu $z$ zvolíme později)

(iii) $|C_a| + |C_{a+1}| > z$
(žádné dvě sousední příhrádky není možno spojit bez porušení předchozí podmínky)

**Lemma 7.1.1:** *Ať zvolíme hodnotu $z$ libovolně, libovolné korektní rozdělení $n$ bodů do příhrádek má $O(n/z)$ příhrádek.*

**Důkaz:** Přihrádky rozdělíme na velké (ty obsahují alespoň $z/2$ bodů) a malé (všechny ostatní). Velkých příhrádek může být nejvýše $2n/z$, jinak by totiž dohromady obsahovaly více než $n$ bodů. Malé příhrádky se mohou vyskytovat před první velkou, po poslední velké, případně mezi velkými, nikdy ovšem dvě za sebou, jelikož tím by porušily podmínku (iii). Malých příhrádek proto je maximálně $2n/z + 1$, a všech příhrádek tudíž $O(n/z)$. $\square$

Pro každou příhrádku $P_a$ si budeme udržovat celkový počet bodů v ní $c(p) = |P_a|$, minimální a maximální $y$-ovou souřadnici bodů z $P_a$ ($miny(p)$ a $maxy(p)$) a binární vyhledávací strom obsahující všechny body uložené v této příhrádce, uspořádané podle $x$-ové souřadnice. V každém vrcholu tohoto stromu si budeme pamatovat součet $\oplus$ ohodnocení všech bodů reprezentovaných vrcholy v podstromu tímto vrcholem určeném; snadno ověříme, že při vkládání, odebírání, rozdělování a slučování stromů můžeme tyto součty přepočítat podle součtů uložených v synech (blíže viz [M84a]).

Podobně jako jsme ve Sleator-Tarjanových stromech udržovali implicitně váhy, zde budeme implicitně udržovat souřadnice bodů: $y$-ové budou relativní vůči $miny$, $x$-ové budou posouvány hodnotami $shiftx(v)$ uloženými ve vnitřních vrcholech stromu. Skutečné souřadnice bodu uloženého jako $(x, y)$ tedy budou $(x + s, y + miny)$, kde $s$ je součet všech $shiftx$ na cestě z vrcholu reprezentujícího tento bod do kořene stromu. To nám umožní snadno přemisťovat celé příhrádky resp. celé podstromy. Při práci se stromy můžeme relativní hodnoty v dotčených vrcholech udržovat například tak, že je před změnou struktury stromu (např. rotací) převedeme na absolutní a po ní zpět.

Nejprve popíšeme tři interní operace, které nám poslouží jako základní kameny pro konstrukci operací ostatních:

- *Build*(**set** $S$) (inicializace struktury tak, aby obsahovala body z množiny $S$) – uspořádáme body podle $y$-ové souřadnice a „hladově" rozdělíme do příhrádek: Začneme s jednou prázdnou příhrádkou. Poté vždy nalezneme všechny dosud nezařazené body s minimální $y$-ovou souřadnicí a pokud by velikost aktuální příhrádky nepřesáhla $z$, umísíme je do této příhrádky. V opačném případě založíme příhrádku novou a přesuneme body do ní. Nakonec setřídíme body v každé příhrádce podle $x$, vytvoříme vyhledávací stromy a spočteme $c$, $miny$ a $maxy$. *Build* vytvoří rozdělení splňující všechny invarianty v čase $O(n \cdot \log n)$.
- *Split*(**integer** $y_0$) – přerozdělí body v příhrádkách tak, aby souřadnice $y_0$ tvořila hranici příhrádek (to jest aby žádná příhrádka neobsahovala body s $y$-ovou souřadnicí větší i menší než $y_0$). Přitom může porušit invariant (iii). *Split* provedeme tak, že projdeme všechny příhrádky (čas $O(k)$), vyhledáme tu, která obsahuje souřadnici $y_0$ „uvnitř" (pokud existuje) a rozdělíme ji na dvě příhrádky (uvědomme si, že taková příhrádka musela obsahovat body s různými $y$-ovými souřadnicemi, takže měla velikost $\leq z$ – proto rozdělení stihneme provést v čase $O(z)$). Celkový čas operace *Split* tedy je $O(k + z)$.
- *Merge* – sloučí některé sousední příhrádky tak, aby invariant (iii) opět platil. Projde všechny příhrádky a jakmile nalezne nějakou dvojici $P_a, P_{a+1}$ takovou, že $|P_a| + |P_{a+1}| \leq z$, sloučí je dohromady (sléváním setříděných posloupností bodů a rekonstrukcí stromu). Časová složitost celkem $O(k)$ na vyhledání příhrádek plus $O(z)$ na každé sloučení.

Standardní operace plane shift struktury implementujeme takto:

- *InsertPoint*(**integer** $x$, $y$, **value** $v$) – nalezne příhrádku, do které patří body s touto $y$-ovou souřadnicí. Pokud není plná (to znamená buďto obsahuje méně než $z$ bodů nebo všechny body v ní mají stejné $y$), bod zatřídíme do jejího stromu a aktualizujeme $c$, $miny$ a $maxy$. Pokud plná je, provedeme *Split*($y$) resp. *Split*($y + 1$) (pokud $y = miny$), abychom ji rozdělili na dvě příhrádky menší a poté již nový bod můžeme přidat bez porušení invariantů pro jeho příhrádku. Mohlo se nám ovšem stát, že druhá část rozdělené příhrádky může být zkombinována se svým druhým sousedem, což ošetříme voláním *Merge*. Vkládáním bodu tak strávíme čas $O(k)$ za hledání příhrádek, $O(\log n)$ za práci se stromy, případně $O(k + z)$ za *Split* a $O(k + z)$ za *Merge* (slučujeme nejvýše jednou), celkem tedy $O(k + z + \log n)$.
- *DeletePoint*(**integer** $x$, $y$) – vyhledáme příhrádku, v níž se bod $(x, y)$ vyskytuje, odebereme jej z jejího stromu, přepočítáme $c$, $miny$ a $maxy$ a zavoláme *Merge*, aby příhrádku sloučil s některou ze sousedních, je-li toho třeba. Časová složitost: $O(k)$ za hledání, $O(\log n)$ za mazání ze stromu, $O(z)$ na přepočítání pomocných hodnot a $O(k + z)$ za *Merge*, celkem $O(k + z + \log n)$.
- **value** *RectSum*(**integer** $x_1$, $y_1$, $x_2$, $y_2$) – voláním *Split*($y_1$) a *Split*($y_2 + 1$) rozdělíme příhrádky tak, že každá příhrádka bude buďto celá obsažena v intervalu $y$-ových souřadnic $[y_1, y_2]$ nebo naopak bude celá mimo tento interval. Poté pro každou příhrádek uvnitř intervalu spočteme pomocí jejího binárního stromu součet ohodnocení bodů mající $x \in [x_1, x_2]$ a sečteme tyto součty. Nakonec zavoláme *Merge* a tím opět obnovíme platnost invariantů. Tím strávíme čas

$O(k+z)$ za *Splity*, $O(k \log n)$ za sčítání v přihrádkách a $O(k+z)$ za *Merge* (slučujeme opět $O(1)$ přihrádek), celkem $O(k \log n + z)$.

- **integer** *RectMinX*(**integer** $x_1$, $y_1$, $x_2$, $y_2$) a *RectMaxX*(**integer** $x_1$, $y_1$, $x_2$, $y_2$) – podobně jako *RectSum*, pouze místo sčítání ohodnocení bodů vyhledáme v každé přihrádce nejlevější, resp. nejpravější bod s $x \in [x_1, x_2]$. Čas rovněž $O(k \log n + z)$.
- **integer** *RectMinY*(**integer** $x_1$, $y_1$, $x_2$, $y_2$) a *RectMaxY*(**integer** $x_1$, $y_1$, $x_2$, $y_2$) – opět provedeme *Split*($y_1$) a *Split*($y_2+1$) a procházíme přihrádky uvnitř $[y_1, y_2]$ podle rostoucí, resp. klesající $y$-ové souřadnice a pro každou přihrádku ověříme, zda obsahuje nějaký bod s $x \in [x_1, x_2]$ a pokud ano, probereme body v této přihrádce jeden po druhém a nalezneme bod s minimální, resp. maximální $y$-ovou souřadnicí. Následně voláním *Merge* obnovíme platnost invariantů. Čas: $O(k+z)$ na *Splity*, pak projdeme $O(k)$ přihrádek a každou v čase $O(\log n)$ testujeme, až uspějeme, strávíme čas $O(z)$ zkoumáním přihrádky [velkými přihrádkami se totiž nemusíme zabývat] – celkem tedy $O(k \log n + z)$.
- *ShiftX*(**integer** $x_1$, $x_2$, $x_t$) – pro každou přihrádku rozdělíme její strom na části odpovídající intervalům $[-\infty, x_1 - 1]$, $[x_1, x_2]$, $[x_2+1, x_t - 1]$, $[x_t, x_t + x_2 - x_1]$ a $[x_t + x_2 - x_1 + 1, +\infty]$ (BÚNO $x_2 < x_t$). Poté interval $[x_1, x_2]$ přičtením $x_t - x_1$ k $x$-ovému offsetu v kořeni příslušného stromu posuneme na správné místo a stromy opět sloučíme. To zvládneme v čase $O(\log n)$ na přihrádku, celkově tedy $O(k \log n)$.
- *ShiftY*(**integer** $y_1$, $y_2$, $y_t$) – rozdělíme pomocí operací *Split*($y_1$), *Split*($y_2+1$) a *Split*($y_t$) přihrádky protnuté dělícími přímkami, poté přesuneme přičtením $y_t - y_1$ k *miny* všechny přihrádky z rozsahu $y$-ových souřadnic $[y_1, y_2]$ na jejich nová místa a taktéž je přemístíme v seznamu přihrádek. Nakonec operací *Merge* sloučíme, co je třeba (konstantou omezený počet přihrádek). Časová složitost: $O(k+z)$ na *Splity*, $O(k)$ na posouvání přihrádek a $O(k+z)$ na *Merge*, tudíž celkem $O(k+z)$.

Všechny standardní operace pracují v čase omezeném $O(k \cdot \log n + z) = O((n/z) \cdot \log n + z)$. Proto $z$ zvolíme tak, abychom dosáhli nejlepší časové složitosti. Jelikož s rostoucím $z$ první člen výrazu klesá a druhý stoupá, optima dosáhneme, když budou vyrovnány, tedy $(n/z) \cdot \log n = z$, což nastane právě pro $z = \sqrt{n \cdot \log n}$. Musíme ale zařídit, aby $z$ mělo stále řádově tuto hodnotu a pokud počet bodů ve struktuře řádově vzroste nebo klesne, budeme muset strukturu reinicializovat; tento čas pak amortizovaně rozložíme mezi jednotlivé operace.

**Věta 7.1.2:** *Všechny operace na této implementaci plane shift struktury pracují v amortizovaném čase $O(\sqrt{n \cdot \log n})$ a prostoru $O(n)$, kde $n$ je počet bodů uložených ve struktuře.*

**Důkaz:** Zapamatujeme si počet bodů $n_0$ ve struktuře v okamžiku posledního volání *Build* (a volby $z = \sqrt{n_0 \cdot \log n_0}$; mezi těmito inicializacemi struktury $z$ neměníme) a kdykoliv $n$ vzroste nad $2n_0$ nebo klesne pod $n_0/2$, strukturu zrekonstruujeme opětovným voláním *Build*. Tak bude vždy $n_0 = \Theta(n)$, $z = \Theta(\sqrt{n \log n})$ a $k = \Theta(n/z) = \Theta(\sqrt{n / \log n})$. Každá operace proto bude trvat $O(k \cdot \log n + z)$, jak jsme ukázali výše, což je rovno $O(\sqrt{n \cdot \log n})$. Navíc musíme připočíst amortizaci reinicializací struktury – ty trvají $O(n_0 \cdot \log n_0)$ a mezi každými dvěma z nich musí proběhnout minimálně $n_0/2$ operací, amortizovaná cena na operaci tedy činí $O(\log n)$.

Co se prostorové složitosti týče, plane shift struktura zabírá prostor $O(k)$ na informace o přihrádkách plus $O(n)$ na jednotlivé stromy reprezentující vnitřky přihrádek. $\square$

## 7.2. Aplikace v dynamických algoritmech

Nyní dosadíme tuto plane shift strukturu do dynamických grafových algoritmů odvozených v předchozích kapitolách (všechny časové složitosti budou amortizované).

Podle věty 5.3.3 s volbou $a = \sqrt{n}$ získáme strukturu pro ukládání nestromových hran v prostoru $O(m + n)$ s časovou složitostí $O(\sqrt{n \cdot \log n})$ na operace *Link*, *InsertNontree*, *Cut*, *DeleteNontree* a *SumNontree* a $O(1)$ na *IsTreeEdge*. Z věty 5.4.1 pak získáme plně dynamický algoritmus pro 1-souvislost a minimální kostru s časem $O(\sqrt{n \cdot \log n})$ na *Insert* a *Delete* a časem $O(1)$ na *Query*, to vše v prostoru $O(n)$. Algoritmus pro plně dynamickou 2-souvislost 3-omezených grafů má pak podle věty 6.4.2 časovou složitost $O(n^{1/2} \cdot \log^{3/2} n)$ na operace *Insert* a *Delete* a $O(\log n)$ na operaci *Query*. Pokud budeme obecné grafy transformovat na 3-omezené podle věty 4.1.1, získáme tak dynamickou 2-souvislost pro obecné grafy v čase $O(m^{1/2} \cdot \log^{3/2} n)$ na modifikace a $O(\log n)$ na dotaz.

Pomocí techniky sparsifikace zavedené Galilem, Eppsteinem a Italianem v [EGI92] a posléze vylepšené v [EGI93] je možno tyto časy redukovat na $O(\sqrt{n \cdot \log n})$ na update a $O(1)$ na dotaz pro 1-souvislost a minimální kostru a $O(n^{1/2} \cdot \log^{3/2} n)$ na update a $O(\log n)$ na dotaz pro 2-souvislost v obecných grafech. Detaily sparsifikace zde vynecháme, jelikož hlavním cílem této práce jsou polylogaritmické převody, u kterých není sparsifikace zapotřebí, jelikož $\log m = O(\log n)$.

Pokud by se podařilo nalézt polylogaritmickou verzi plane shift struktury, získali bychom polylogaritmické algoritmy pro všechny tři v této práci studované grafové problémy:

**Věta 7.2.1:** *Existuje-li plane shift struktura s polylogaritmickým časem na operaci, pak existují dynamické algoritmy pro souvislost, minimální kostru a 2-souvislost mající polylogaritmický čas na modifikaci grafu a čas $O(\log n)$ na dotaz.*

**Důkaz:** Dosazením do vět 5.3.3, 5.4.1, 6.4.2 a 4.1.1. □

Mehlhorn v [M84b] odvozuje několik dvojrozměrných datových struktur, z toho jednu polylogaritmickou (range trees), nicméně žádná z nich nesplňuje naše požadavky. Rovněž ukazuje odmocninový dolní odhad pro obdélníkové dotazy, nicméně pouze pro struktury, které jsou omezeny lineární pamětí (range trees používají paměť $O(n \log n)$ a odpovídají na obdélníkové dotazy v čase $O(\log^2 n)$). To dává naději, že by polylogaritmická plane shift struktura mohla existovat.

# 8. Polylogaritmické algoritmy

Holm, Lichtenberg a Thorup popsali v [HLT97a] plně dynamické algoritmy pro udržování komponent souvislosti a minimální kostry s amortizovaně polylogaritmickým časem na modifikaci struktury při vložení či odstranění hrany, jakož i na dotaz. Později (v [HLT97b]) na podobné myšlence založili i polylogaritmické plně dynamické algoritmy pro udržování hranové i vrcholové 2-propojenosti.

## 8.1. Souvislost

Nejprve popíšeme myšlenku, na níž je algoritmus založen. Budeme udržovat kostru $F$ dynamického grafu $G$. Každé hraně $e \in E(G)$ (ať již to je stromová či nestromová hrana) přiřadíme úroveň $\ell(e) \leq L = \lfloor \log_2 n \rfloor$. Označíme $F_i$ podgraf $F$ indukovaný hranami úrovně alespoň $i$. Bude tedy platit $F = F_0 \supseteq F_1 \supseteq \cdots \supseteq F_L$. Budeme zachovávat následující invarianty:

(i) $F$ je maximální kostra grafu $G$ vzhledem k váhám daným úrovněmi hran. (Jinými slovy pokud $\{v, w\}$ je nestromová hrana, $v \leftrightarrow w$ v grafu $F_{\ell(v,w)}$.)

(ii) Maximální velikost komponenty souvislosti v grafu $F_i$ je nejvýše $n/2^i$. (Proto úrovně větší než $L$ jsou všechny prázdné a nemá je smysl uvažovat.)

Intuitivně řečeno, každá úroveň tvoří rozklad kostry do stromů příslušné velikosti, které jsou tvořeny stromovými hranami této a vyšších úrovní a navzájem propojené hranami nižších úrovní. Nestromové hrany na úrovni $i$ vždy vedou mezi vrcholy téhož stromu rozkladu. V ideálním případě by na nejvyšší úrovni byly všechny stromy jednovrcholové a na každé další úrovni by pak vznikaly spojením dvou stromů bezprostředně vyšší úrovně hranou, tvoříce rozklad grafu podobný Fredericksonově clusterizaci bez omezení stupňů vrcholů. Udržovat tento fixní tvar je ovšem neúnosné, místo toho budeme úrovně hran postupně zvyšovat, jakmile objevíme, že krajní body hrany jsou v $G$ dostatečně blízko a čas strávený neúspěšným zkoumáním hrany při hledání náhrady za smazané stromové hrany vždy amortizujeme tím, že se zvýší její úroveň, takže stačí, aby každý *Insert* zaplatil čas těchto nejvýše $L$ zvýšení.

Nyní popíšeme jednotlivé operace:

- *Init* – na počátku nastavíme všem hranám úroveň 0, a tak jsou oba invarianty splněny.
- *Insert*(**vertex** $v$, $w$) – hraně $e = \{v, w\}$ přidělíme úroveň 0. Pokud $v \leftrightarrow w$, přidáme ji jako nestromovou, jinak jako stromovou do $F = F_0$.
- *Delete*(**vertex** $v$, $w$) – odstraníme hranu $e = \{v, w\}$ z naši reprezentace grafu, pokud byla nestromovou hranou, nic se neděje, pokud byla stromovou, musíme podobně jako v algoritmu 5.4 vyhledat její náhradu, kvůli dodržení invariantu (i) ovšem na nejvyšší možné úrovni. Jelikož $F$ je maximální kostra, musí mít taková hrana (pokud vůbec existuje) úroveň maximálně $\ell(e)$. Začneme proto prohledávat jednotlivé úrovně od $\ell(e)$ k nižším a na každé se pokusíme náhradu nalézt:

  Nechť právě zpracováváme úroveň $i$ a $T_v$ a $T_w$ jsou stromy v $F_i$ obsahující vrcholy $v$ a $w$ (BÚNO $|T_v| \leq |T_w|$). Předtím, než jsme smazali hranu $e$, byl $T = T_v \cup \{\{v, w\}\} \cup T_w$ strom úrovně $i$ a měl alespoň $2 \cdot |T_v|$ vrcholů. Podle invariantu (ii) měl $T$ nejvýše $n/2^i$ vrcholů, tudíž $T_v$ jich nyní má nejvýše $n/2^{i+1}$, a proto můžeme při dodržení obou invariantů přesunout všechny hrany stromu $T_v$, které mají úroveň $i$, na úroveň $i + 1$.

Poté budeme probírat jednu nestromovou hranu úrovně $i$ incidentní s $T_v$ po druhé. Pokud nalezneme hranu, která spojuje $T_v$ s $T_w$, právě jsme našli náhradu za $e$ a můžeme ji zařadit do kostry a operaci *Delete* úspěšně ukončit. Pokud nespojuje, pak to znamená, že vede mezi dvěma vrcholy $T_v$ (z invariantu (i) totiž víme, že původně vedla uvnitř $T$), a proto zvýšíme její úroveň na $i+1$ (invariant (i) neporušíme, jelikož jsme všechny hrany $T_v$ přesunuli o úroveň výše), čímž zaplatíme marné zkoumání této hrany.

Pokud prozkoumáme všechny incidentní hrany na úrovni $i$, budeme pokračovat úrovní $i-1$ atd., pokud nenalezneme náhradu ani na nulté úrovni, byla $e$ mostem a $v$ a $w$ jsou nyní v různých komponentách souvislosti.

Abychom tyto operace implementovali efektivně, musíme zvolit vhodnou reprezentaci lesů $F_i$ a nestromových hran. Použijeme k tomu pro každé $i$ jeden ET-strom (viz 5.2) pro $a = 2$ popisující eulerovský tah v $F_i$ (tedy stromové hrany úrovní $\leq i$) a nestromové hrany úrovně právě $i$. V každém vnitřním vrcholu $w$ si budeme navíc udržovat (podobně jako udržujeme údaje popsané v definici ET-stromu) počet $vc(w)$ vrcholů v příslušném podstromu a počet $tcount(w)$ listů reprezentujících stromové hrany úrovně právě $i$.

*Query* přeložíme na ET-stromové *Query* na $F_0$, které trvá čas $O(\log n)$.

*Insert* bude proveden triviálně – jako *Query* a buďto *Link* nebo *InsertNontree* na ET-stromu reprezentujícím $F_0$. To vše v čase $O(\log n)$.

*Delete* nejprve pomocí *IsTreeEdge* na $F_0$ zjistí, zda mažeme stromovou hranu a provede $O(\log n)$ operací s ET-stromy (odstraňování hrany a vkládání náhrady do až $L$ úrovní). Při hledání náhrady vždy pomocí ET-stromu pro $F_i$ v čase $O(\log n)$ spočteme velikost komponenty (stačí vyhledat kořen a použít jeho $vc$), následně vyhledáme všechny stromové hrany úrovně právě $i$ (to dokážeme v čase $O(\log n)$ na jednu hranu analogicky s *FindNontree* použitím hodnot *tcount*) a každou z nich v čase $O(\log n)$ přidáme do $F_{i+1}$. Poté opakovaně voláme *FindNontree* pro vyhledání nestromových hran ($O(\log n)$ na každou), ty z $F_i$ odstraňujeme ($O(\log n)$ na *DeleteNontree*), testujeme, zda druhý konec padne do $T_w$ (vyhledání kořene v čase $O(\log n)$) a konečně v případě neúspěchu hranu vložíme do ET-stromu úrovně $i+1$ ($O(\log n)$ na *InsertNontree*). Časová složitost operace *Delete* je $O(\log^2 n)$ plus cena za neúspěšná zkoumání hran. Jelikož však jedné hraně zvýšíme úroveň nejvýše $\log_2 n$-krát, trvají za celou dobu existence hrany její neúspěšná zkoumání nejvýše $O(\log^2 n)$, kterýžto čas můžeme připočíst k trvání operace *Insert*, a tak jej amortizovat.

Operaci *Query* lze navíc při zachování asymptotických časů operací *Insert* a *Delete* zrychlit podle důsledku 5.1.3 na $O(\log n / \log\log n)$.

**Věta 8.1.1:** *Popsaný dynamický algoritmus udržuje komponenty souvislosti v čase* $O(\log^2 n)$ *na Insert a Delete a* $O(\log n / \log\log n)$ *na Query, to vše v prostoru* $O(m + n \log n)$.

**Důkaz:** Časová složitost plyne z diskuse výše. Prostorovou složitost získáme sečtením prostorových nároků ET-stromů podle věty 5.2.1. □

## 8.2. 2-souvislost

Polylogaritmický algoritmus pro dynamické udržování 2-propojenosti uvedený v [HLT97b] je poměrně komplikovaný a vymyká se rozsahu této práce. Proto zde uvedeme pouze následující větu:

**Věta 8.2.1** *(Holm, De Lichtenberg, Thorup)***:** *Existuje plně dynamický algoritmus pro udržování vrcholové i hranové 2-propojenosti pracující v čase* $O(\log^4 n)$ *na operaci.*

**Důkaz:** Viz [HLT97b]. □

# 9. Závěr

V předchozích kapitolách jsme popsali dynamické datové struktury s logaritmickou časovou složitostí sloužící k reprezentaci stromů – dynamické stromy (u nich jsme se spokojili s jednodušším logaritmickým odhadem složitosti), topologické stromy (omezili jsme se na 2-rozklady na všech úrovních clusterizace) a ET-stromy (ty jsme rozšířili a doplnili analýzu). Na základě těchto struktur jsme posléze odvodili několik polylogaritmických semidynamických a plně dynamických grafových algoritmů pro problémy souvislosti, hranové 2-souvislosti a udržování minimální kostry neorientovaných grafů.

Rovněž jsme ukázali polylogaritmickou redukci reprezentace nestromových hran grafu, a tím pádem také plně dynamických verzí studovaných grafových problémů, na problém reprezentace bodů v rovině s přesouváním (plane-shift problem). Pro plane-shift problem jsme sestrojili jednoduchou datovou strukturu, která sice není polylogaritmická, nicméně při použití ve zmíněných algoritmech dává výsledky srovnatelné s předchozími Fredericksonovými pracemi (zpomalení o $O(\sqrt{\log n})$ vyvážené značným zjednodušením algoritmů díky tomu, že není zapotřebí udržovat vícerozměrné topologické stromy ani ambivalentní struktury). Z této redukce vyplývá, že pokud je reprezentace bodů v rovině řešitelná v polylogaritmickém čase, pak existují plně dynamické grafové algoritmy pro souvislost, minimální kostru i hranovou 2-souvislost s taktéž polylogaritmickou složitostí.

Na závěr jsme demonstrovali nové Thorupovy, De Lichtenbergovy a Holmovy polylogaritmické výsledky. Jimi je kapitola dynamické souvislosti více méně uzavřena a zbývá již prostor pouze pro logaritmická zrychlení a v případě 2-souvislosti rovněž pro zjednodušování algoritmů.

Tato práce ukazuje alternativní pohled na problematiku dynamických grafových algoritmů a popisuje četné techniky pro jejich konstrukci, které mohou být využity pro efektivní dynamická řešení dalších grafových problémů.

# Literatura


[EGI92] Eppstein David, Galil Zvi, Italiano Giuseppe F., Nissenzweig Ammon: Sparsification – A technique for speeding up dynamic graph algorithms. In *Proc. 33rd Symposium on Foundations of Computer Science*, 60–69, 1992.

[EGI93] Eppstein David, Galil Zvi, Italiano Giuseppe F.: Improved sparsification. Technical Report 93-20, Department of Information and Computer Science, University of California, Irvine, 1993.

[F85] Frederickson Greg N.: Data structures for on-line updating of minimum spanning trees. In *SIAM Jounal on Computing*, **14**:781–798, 1985.

[F97] Frederickson Greg N.: Ambivalent data structures for dynamic 2-edge connectivity and $k$ smallest spanning trees. In *SIAM Journal on Computing*, **26**:484–538, 1997.

[HK95] Henzinger Monika R., King Valerie: Randomized dynamic graph algorithms with polylogarithmic time per operation. In *Proc. 27th Symp. on Theory of Computing*, 519–527, 1995.

[HK97a] Henzinger Monika R., King Valerie: Maintaining minimum spanning tree in dynamic graphs. In *Proc. 24th International Colloquium on Automata, Languages and Programming*, 594–604, 1997.

[HK97b] Henzinger Monika R., King Valerie: Fully dynamic 2-edge connectivity algorithm in polylogarithmic time per operation. Technical report, Digital, 1997.

[HLT97a] Holm Jacob, De Lichtenberg Kristian, Thorup Mikkel: Poly-logarithmic deterministic fully dynamic graph algorithms I: connectivity and minimum spanning tree. Technical Report 97-17, Department of Computer Science, University of Copenhagen.

[HLT97b] Holm Jacob, De Lichtenberg Kristian, Thorup Mikkel: Poly-logarithmic deterministic fully dynamic graph algorithms II: 2-edge and biconnectivity. Technical Report 97-26, Department of Computer Science, University of Copenhagen.

[HT96] Henzinger Monika R., Thorup Mikkel: Improved sampling with applications to dynamic graph algorithms. In *Proc. 23rd International Colloquium on Automata, Languages and Programming*, 290–299, 1996.

[K83] Kučera Luděk: Kombinatorické algoritmy. SNTL, Praha 1983.

[KR97] Korupolu Madhukar R., Ramachandran Vijaya: Quasi-fully dynamic algorithms for two-connectivity, cycle equivalence and related problems. In *Proc. 5th Annual European Symposium on Algorithms*, Springer-Verlag LNCS, 1997.

[LW94] La Poutré Johannes A., Westbrook J.: Dynamic 2-connectivity with backtracking. In *Proc. 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, 204–212, 1994.

[M84a] Mehlhorn Kurt: Data Structures and Efficient Algorithms, Volume 1: Sorting and searching. Springer Verlag, EACTS Monographs, 1984.

[M84b] Mehlhorn Kurt: Data Structures and Efficient Algorithms, Volume 3: Multidimensional searching and computational geometry. Springer Verlag, EACTS Monographs, 1984.

[ST83] Sleator Daniel. D., Tarjan Robert E.: A data structure for dynamic trees. In *Journal of Computer and System Sciences*, vol. 26, no. 3, June 1983.

[SV82] Shiloach Yossi, Vishkin Uzi: An $O(\log n)$ parallel connectivity algorithm. In *Journal of Algorithms*, **3**:57–63, 1982.

[T75] Tarjan Robert E.: Efficiency of a good, but not linear set union algorithm. In *Journal of the Association for Computing Machinery*, **22**:212–225, 1975.

[TL84] Tarjan Robert E., van Leeuwen J.: Worst-case analysis of set union algorithms. In *Journal of the Association for Computing Machinery*, **31**:245–281, 1984.

[TV85] Tarjan Robert E., Vishkin Uzi: An Efficient Parallel Biconnectivity Algorithm. In *SIAM Journal on Computing* **14**:862–874, 1985.

[TW92] Tarjan Robert E., Westbrook J.: Maintaining bridge-connected and biconnected components on-line. In *Algorithmica*, **7**:433–464, 1992.

[W89] Westbrook J.: Algorithms and data structures for dynamic graph problems. PhD thesis, Department of Computer Science, Princeton University, Princeton, NJ, 1989.


*Elektronickou verzi této práce, jakož i většiny citovaných článků je možno nalézt na Internetu na http://atrey.karlin.mff.cuni.cz/~mj/dga/.*