UNIVERZA V LJUBLJANI
FAKULTETA ZA MATEMATIKO IN FIZIKO
ODDELEK ZA MATEMATIKO

Matjaž Zaveršnik

RAZČLEMBE OMREŽIJ

Doktorska disertacija

Ljubljana, 2003

Zahvala

Mentorju, prof. dr. Vladimirju Batagelju, bi se rad zahvalil za predlagano temo, dragocene napotke, podporo in potrpežljivost pri nastajanju tega dela. Hvala vsem sodelavcem ter članom sredinega seminarja in seminarja za diskretno matematiko, ki so prisluhnili mojim razlagam in vprašanjem. Zahvaljujem se tudi vsem domačim za vzpodbudo.

Povzetek

Precej znanih postopkov za analizo omrežij je prepočasnih, da bi jih lahko uporabili na velikih omrežjih. Zato si pogosto pomagamo z razčlembo omrežja na manjše in lažje obvladljive dele. V doktorski disertaciji se ukvarjamo z učinkovitimi (podkvadratičnimi) postopki za razčlembe velikih omrežij.

V uvodnem poglavju so podane osnovne definicije in oznake, ki jih potrebujemo v nadaljevanju. Drugo poglavje je pregled najbolj znanih že obstoječih postopkov za določanje razčlemb. Opisani so postopki za določanje razbitij danega omrežja, požrešni postopek za izboljšanje dobljenega razbitja, in nekaj drugih vrst razčlemb. Tretje, četrto in peto poglavje vsebujejo izvirne prispevke disertacije.

V tretjem poglavju definiramo točkovne in povezavne otoke omrežja. Točkovni otok je povezana skupina točk, ki glede na svoje vrednosti izstopa od točk v svoji soseščini. Podobno so definirani tudi povezavni otoki, le da imamo tam vrednosti na povezavah. Opisani so učinkoviti postopki za določanje otokov in dokazanih več njihovih lastnosti. V naslednjih dveh poglavjih sta razvita primera učinkovito izračunljive točkovne in povezavne funkcije.

V četrtem poglavju se ukvarjamo s sredicami. Sredica reda k je maksimalen podgraf, v katerem ima vsaka točka vsaj k sosedov. Sredice določajo gnezdeno razslojitev grafa, povezane komponente teh slojev pa določajo hierarhijo nad množico točk. Opisan je učinkovit postopek za določanje sredic. Pojem sredic je posplošen tudi na omrežja, kjer namesto stopnje v dani točki opazujemo kakšno drugo točkovno funkcijo. Pokazano je, da obstaja učinkovit postopek za cel razred točkovnih funkcij.

V petem poglavju posplošimo pojem povezanosti v grafu. Definiramo različne relacije povezanosti točk (točkovno in povezavno povezanost s kratkimi cikli v neusmerjenih in usmerjenih grafih). Opazimo, da so nekatere od teh relacij ekvivalenčne relacije na množici točk, druge pa določajo ekvivalenčno relacijo na množici povezav, kar nam spet določa razbitje grafa.

Math. Subj. Class. (2000): 05 C 40, 05 C 85, 05 C 90, 68 R 10, 68 W 40, 93 A 15.

Ključne besede: analiza velikih omrežij, vrednosti točk, uteži povezav, prerezi, otoki, sredice, povezanost, kratki cikli

Abstract

Several well known algorithms for network analysis are too slow to be used on large networks. Therefore we often decompose the network to smaller parts that are easier to handle. In doctoral dissertation we study efficient (subquadratic) algorithms for large networks decompositions.

The preamble contains some basic definitions and notations used in the continuation. The second chapter is overview of the already existent algorithms for determining the network decompositions. Algorithms for determining the partition of a given network, a greedy algorithm for improvement of a given partition, and some other types of decompositions are described. Third, fourth and fifth chapter contain original contribution of the dissertation.

In the third chapter the vertex and edge islands of network are defined. Vertex island is connected set of vertices having values greater than the vertices in their neighborhood. Similarly the edge islands for networks with weights on edges are defined. Efficient algorithms for determining the islands were developed and some properties of islands are proved. In the next two chapters examples of efficiently computable vertex and edge functions are developed.

In the fourth chapter we study cores. The core of order k is the maximal subgraph in which every vertex has at least k neighbors. The cores form nested layers of a graph, while the set of connected components of these layers is hierarchy on the set of vertices. Efficient algorithms for determining the cores are presented. The concept of cores is also generalized for networks, where we observe some other vertex function instead of degree. It is shown that there exists an efficient algorithm for the whole class of vertex functions.

In the fifth chapter the concept of graph connectivity is generalized. We define several different connectivity relations on the set of vertices (vertex and edge short cycles connectivity in undirected and directed graphs). We notice, that some of them are equivalence relations on the set of vertices, while the others determine equivalence relations on the set of lines, which give us another decomposition of a given graph.

Math. Subj. Class. (2000): 05 C 40, 05 C 85, 05 C 90, 68 R 10, 68 W 40, 93 A 15.

Key words: large networks analysis, values of vertices, weights of edges, cuts, islands, cores, connectivity, short cycles

Kazalo

Prvo poglavje

Uvod

Pri analizi velikih omrežij se pogosto srečamo s problemom razkrivanja njihove zgradbe. Vpogled v zgradbo omrežja nam omogoča razumevanje njegovega delovanja, lahko pa je tudi osnova za razvoj učinkovitih postopkov za delo z omrežji.

Eden od možnih pristopov k problemu razkrivanja zgradbe omrežja je razčlemba omrežja na več manjših in lažje obvladljivih podomrežij. Pogosto zahtevamo, da so ta podomrežja iz izbranih družin ali pa morajo zadoščati kakšnim drugim pogojem. Te razčlembe so lahko osnova za izgradnjo učinkovitih (približnih) postopkov za posamezne probleme na velikih omrežjih. Pri takih postopkih najprej rešimo problem na manjših omrežjih, nato pa poskušamo iz dobljenih rešitev dobiti približno rešitev problema na celem omrežju.

Znanih je več vrst razčlemb omrežij. Eno od njih so razbitja. Ta so določena z razbitjem množice njegovih točk na več ločenih nepraznih podmnožic (skupin). Običajno iščemo takšna razbitja, da bo število povezav, ki imajo krajišča v različnih skupinah, čim manjše (razdelitev poslov med k procesorjev), včasih pa bi radi imeli takšnih povezav čim več (barvanje točk danega omrežja s k barvami). Večkrat postavimo še dodatno zahtevo, da naj bodo posamezne skupine čim bolj uravnotežene (po moči čim bolj enake). Razbitju danega omrežja na k skupin pravimo tudi k-razbitje.

Druga vrsta razčlemb omrežij so razslojitve. Pri teh razčlembah vsaki točki omrežja določimo njen nivo, sloj na nivoju t pa je potem sestavljen iz točk, ki so na nivojih z vrednostjo t ali več. Ker so sloji na višjih nivojih vsebovani v slojih na nižjih nivojih, govorimo tudi o gnezdenih razslojitvah.

Včasih pa nas ne zanima razbitje ali razslojitev celotnega omrežja, pač pa samo tiste točke ali povezane skupine točk, ki izstopajo glede na neko lastnost. V takem primeru imamo v točkah ali na povezavah podane vrednosti (lahko jih tudi izračunamo iz samega omrežja), ki določajo kako pomembna je posamezna točka ali povezava. Če v takem omrežju poiščemo najpomembnejše točke, lahko iz njih dobimo razbitje omrežja tako, da vsako od teh pomembnih točk proglasimo za predstavnika skupine, vsako od preostalih točk pa postavimo v skupino z najbližjim predstavnikom.

Namesto navadnih okolic pomembnih točk lahko iščemo tudi pomembne skupine. To so povezane skupine točk, katerih pomembnost je večja od pomembnosti točk iz njihove okolice. Takim skupinam rečemo točkovni otoki. Podobno lahko definiramo tudi povezavne otoke.

V naslednjih poglavjih bomo predpostavili, da je bralec seznanjen z osnovnimi pojmi iz teorije grafov [41] in razvrščanja. Vseeno povzemimo nekaj osnovnih definicij in oznak, ki jih bomo potrebovali v nadaljevanju.

1.1 Osnovni pojmi iz teorije grafov

Graf $\mathcal{G}$ je struktura, sestavljena iz množice *točk* $\mathcal{V}$ in seznama *povezav* med njimi (vsaka povezava je par točk). Posamezni točki v paru, ki predstavlja povezavo, rečemo *krajišče* povezave. Če sta krajišči enaki, povezavi rečemo *zanka*. Podmnožici množice točk bomo rekli *skupina* točk. Naj bo n število vseh točk, m pa število vseh povezav v grafu.

Povezava je lahko *neusmerjena* (neurejen par točk) ali *usmerjena* (urejen par točk). Neusmerjeno povezavo med točkama u in v bomo zapisali $(u : v)$, usmerjeno povezavo od točke u do točke v pa (u, v). V splošnem, ko ne bomo želeli predpisati, ali mora biti povezava usmerjena ali neusmerjena, jo bomo zapisali $(u ; v)$.

Graf je *enostaven*, če vsaka povezava v seznamu nastopa natanko enkrat (v tem primeru lahko namesto o seznamu govorimo o množici povezav). V nadaljevanju se bomo ukvarjali samo z enostavnimi grafi. Graf je *neusmerjen*, če so vse povezave neusmerjene, in je *usmerjen*, če so vse povezave usmerjene. V neusmerjenem grafu bomo za množico povezav uporabljali oznako $\mathcal{E}$, v usmerjenem grafu oznako $\mathcal{A}$, v splošnem grafu, kjer lahko imamo obe vrsti povezav, pa oznako $\mathcal{L}$.

$$(u ; v) \in \mathcal{L} \iff (u : v) \in \mathcal{L} \vee (u, v) \in \mathcal{L} \vee (v, u) \in \mathcal{L}$$

Naj bo $v \in \mathcal{V}$ poljubna točka. *Soseščino* točke v lahko definiramo na več načinov, odvisno od tega, kako obravnavamo usmerjene povezave.

$$\begin{aligned} N_{in}(v) &= \{u \in \mathcal{V} : (u : v) \in \mathcal{L} \vee (u, v) \in \mathcal{L}\} \setminus \{v\} \\ N_{out}(v) &= \{u \in \mathcal{V} : (u : v) \in \mathcal{L} \vee (v, u) \in \mathcal{L}\} \setminus \{v\} \\ N(v) &= N_{in}(v) \cup N_{out}(v) \end{aligned}$$

Naj bo $\mathcal{C} \subseteq \mathcal{V}$ poljubna skupina točk. Za $S \in \{N, N_{in}, N_{out}\}$ lahko definiramo še

$$\begin{aligned} S^{+}(v) &= S(v) \cup \{v\} \\ S(v, \mathcal{C}) &= S(v) \cap \mathcal{C} \\ S(\mathcal{C}) &= \bigcup_{v \in \mathcal{C}} S(v) \setminus \mathcal{C} \end{aligned}$$

V teoriji grafov številu povezav s krajiščem v danim točki rečemo *stopnja* točke. V enostavnih grafih je stopnja točke enaka številu njenih sosedov. Spet ločimo med različnimi vrstami stopenj.

$$\begin{aligned}
\deg(v) &= |N(v)| \\
\operatorname{indeg}(v) &= |N_{in}(v)| \\
\operatorname{outdeg}(v) &= |N_{out}(v)| \\
\deg(v,\mathcal{C}) &= |N(v,\mathcal{C})| \\
\operatorname{indeg}(v,\mathcal{C}) &= |N_{in}(v,\mathcal{C})| \\
\operatorname{outdeg}(v,\mathcal{C}) &= |N_{out}(v,\mathcal{C})|
\end{aligned}$$

Največjo stopnjo točke v grafu bomo označili z $\Delta = \max_{v\in\mathcal{V}} \deg(v)$.

Pogosto imamo poleg grafa definirano še funkcijo $p\colon \mathcal{V} \to \mathbb{R}$, ki točkam grafa priredi realne vrednosti, ali pa funkcijo $w\colon \mathcal{L} \to \mathbb{R}$, ki realne vrednosti priredi povezavam (vrednosti povezave rečemo tudi *utež*). V takem primeru namesto o grafu $\mathcal{G} = (\mathcal{V}, \mathcal{L})$ govorimo o omrežju $\mathcal{N} = (\mathcal{V}, \mathcal{L}, p)$ ali $\mathcal{N} = (\mathcal{V}, \mathcal{L}, w)$. Vrednosti so lahko poljubna realna števila, velikokrat pa se pripeti, da so vse nenegativne in/ali cele. Običajno sta v tem primeru analiza omrežja in izvedba nekaterih postopkov veliko preprostejša.

Vrednosti točk ali povezav so pogosto kar dane kot rezultati različnih meritev. V omrežju avtorjev, kjer sta dva avtorja povezana, če sta napisala kakšen skupen članek, je utež povezave lahko število skupnih člankov, vrednost točke pa na primer letnica rojstva. V splošnem so vrednosti lahko tudi drugih vrst. Tako je na primer vrednost točke lahko tudi država rojstva.

Poleg tega pa obstaja še veliko možnosti, kako vrednost točke ali utež povezave izračunati iz samih podatkov o omrežju. Tako je vrednost točke lahko njena stopnja, število različnih poti, ki potekajo skozi točko, stopnja vmesnosti ... Točka je vmesna, če leži na veliko najkrajših poteh med drugimi točkami. Stopnja vmesnosti točke u je definirana z

$$b(u) = \frac{1}{(n-1)(n-2)} \sum_{\substack{v,t\in\mathcal{V}: n(v,t)\neq 0 \\ v\neq t, u\neq v, u\neq t}} \frac{n(v,t;u)}{n(v,t)}$$

kjer je $n(v,t)$ število najkrajših poti od v do t, $n(v,t;u)$ pa število najkrajših poti od v do t, ki gredo skozi u. Stopnja vmesnosti je realno število med 0 in 1.

Tudi uteži posameznih povezav lahko naračunamo iz samega omrežja. Tako je utež povezave lahko število različnih ciklov omejene dolžine, ki vsebujejo to povezavo, ...

1.2 Osnovni pojmi iz teorije razvrščanja

Razvrstitev $\mathcal{C}$ množice A je podmnožica množice njenih nepraznih podmnožic, torej $\mathcal{C} \subseteq \mathcal{P}A \setminus \{\emptyset\}$. To pomeni, da je $\mathcal{C} = \{A_i\}$, $A_i \subseteq A$ in $A_i \neq \emptyset$. Kadar bomo želeli poudariti, da je $\mathcal{C}$ razvrstitev množice A, bomo namesto $\mathcal{C}$ pisali $\mathcal{C}(A)$. Razvrstitev $\mathcal{C}$ je *ploska*, če velja: $i \neq j \Longrightarrow A_i \cap A_j = \emptyset$. *Nosilec* razbitja $\mathcal{C}$ je unija vseh množic v tem razbitju. Na kratko ga označimo z $\bigcup\mathcal{C}$.

Razvrstitev $\mathcal{C}$ je *razbitje*, če je ploska in je njen nosilec enak celi množici A, torej če je $\bigcup\mathcal{C} = A$. Razbitju na k podmnožic bomo rekli tudi razbitje reda k ali k-razbitje. Vsaki od množic v razbitju bomo rekli tudi *skupina*.

Če se omejimo samo na končne množice, potem je tudi vseh različnih razbitij samo končno mnogo. Hitro lahko izračunamo, da je vseh razbitij reda 2 natanko $2^{n-1} - 1$, razbitij višjega reda pa je še precej več. Znano je, da je vseh k-razbitij množice moči n natanko $S(n,k)$, kjer je $S(n,k)$ Stirlingovo število druge vrste.

$$S(n,k) = \frac{1}{k!}\sum_{i=0}^{k-1}(-1)^i \binom{k}{i} (k-i)^n$$

Da bi dobili vtis o velikosti, si poglejmo nekaj takih števil:

$$\begin{aligned} S(24,8) &= 82\,318\,282\,158\,320\,505 \\ S(36,9) &= 54\,294\,340\,536\,065\,700\,496\,358\,447\,625 \\ S(50,10) &= 26\,154\,716\,515\,862\,881\,292\,012\,777\,396\,577\,993\,781\,727\,011 \end{aligned}$$

Običajno nas zanimajo samo razbitja z določeno lastnostjo. Ena od lastnosti, ki nas večkrat zanima, je *uravnoteženost* razbitja. Ta je definirana kot razlika moči največje in najmanjše skupine, torej

$$\mathrm{bal}(\mathcal{C}) := \max\{|A_i| : i = 1,\ldots,k\} - \min\{|A_i| : i = 1,\ldots,k\}$$

Razbitje $\mathcal{C}$ je *uravnoteženo*, če je $\mathrm{bal}(\mathcal{C}) \leq 1$. Izračunajmo, koliko je vseh uravnoteženih k-razbitij množice moči n. Da bo izračun preprostejši, predpostavimo da je n deljiv s k. Vse skupine bodo tako enako velike (v vsaki bo $s = \frac{n}{k}$ točk).

Ker nas zanimajo samo bistveno različna razbitja, najprej izberemo prvi element prve skupine (ta element bo vedno v prvi skupini). Izmed preostalih $n-1$ izberemo poljubnih $s-1$ elementov in jih dodamo v prvo skupino. Preostalih $n-s$ točk pa na podoben način razdelimo v $k-1$ skupin. Tako pridemo do rekurzivne zveze

$$\begin{aligned} S^*(n,1) &= 1 \\ S^*(n,k) &= \binom{n-1}{s-1} S^*(n-s,k-1) \end{aligned}$$

Od tod dobimo

$$S^*(n,k) = \prod_{i=1}^{k} \binom{is-1}{s-1}$$

Poglejmo si še nekaj takih števil:

$$\begin{aligned} S^*(24,8) &= 9\,161\,680\,528\,000 \\ S^*(36,9) &= 388\,035\,036\,597\,427\,985\,390\,625 \\ S^*(50,10) &= 13\,536\,281\,554\,808\,237\,495\,608\,549\,953\,475\,109\,376 \end{aligned}$$

Tudi uravnoteženih razbitij je zelo veliko, a kot vidimo v tabeli 1.1, jih je glede na vsa razbitja razmeroma malo.

n	k	$\frac{S^*(n,k)}{S(n,k)}$
24	8	$1.11296 \cdot 10^{-4}$
36	9	$7.14688 \cdot 10^{-6}$
50	10	$5.17546 \cdot 10^{-7}$

Tabela 1.1: Razmerja med S^* in S

Razbitje grafa $\mathcal{G}$ je razbitje množice njegovih točk $\mathcal{V}$. Ker je vseh možnih (pa tudi samo uravnoteženih) razbitij ogromno, pogosto postavimo dodatne zahteve, ki jih mora iskano razbitje izpolnjevati. Ker je v primeru končnega grafa tudi vseh razbitij samo končno mnogo, bi razbitje, ki ustreza vsem našim zahtevam, načeloma lahko poiskali s pregledom vseh dopustnih razbitij, a si tega ravno zaradi njihovega velikega števila ne moremo privoščiti.

Znano je, da je večina problemov razbitja NP-težkih [2, 22, 13], kar pomeni, da zanje ni učinkovitega (polinomskega) postopka. Preprosto povedano to pomeni, da za te probleme zaenkrat ni znan noben postopek za iskanje optimalnega razbitja, ki bi bil bistveno hitrejši od pregledovanja vseh dopustnih razbitij, pa tudi upanja, da bi tak postopek našli, ni skoraj nobenega. Zato bomo morali namesto z optimalno biti zadovoljni tudi s približno rešitvijo. Preostane nam uporaba približnih postopkov, ki dajejo dobre (skoraj vedno skoraj optimalne) rešitve.

Seveda pa obstajajo tudi polinomsko rešljivi problemi. Tak je na primer problem uravnoteženega razpolavljanja s čim manj povezavami iz ene v drugo skupino. Postopek je opisan v [24].

Glede na to, kakšne točke želimo imeti v isti skupini, ločimo dva osnovna pristopa k določanju razbitij.

- **Povezanostni pristop.** Skupine želimo oblikovati tako, da bodo močno povezane točke v isti skupini, kar z drugimi besedami pomeni, da bo med skupinami čim manj povezav. Primer uporabe tega pristopa je na primer razdelitev n nalog med k procesorjev. Točke grafa so naloge, povezave pa določajo, kje je potrebna izmenjava podatkov. Skupine, ki jih iščemo, bodo določale, kateri procesor bo obdelal katero nalogo, da pa bi bilo izmenjave podatkov med procesorji čim manj, moramo poiskati skupine, med katerimi bo kar se da malo povezav.

Po drugi strani pa včasih želimo, da bi bilo povezav znotraj skupin čim manj, med skupinami pa čim več. Tak primer je barvanje grafa na n točkah s k barvami. V tem primeru bodo točke v isti skupini enako pobarvane, zato mora biti med njimi čim manj povezav, da bo barvanje čim boljše.

- **Podobnostni pristop.** Skupine želimo oblikovati tako, da bodo točke, ki so podobno povezane s preostalimi točkami, v isti skupini. Ta pristop se veliko uporablja predvsem pri analizi družboslovnih omrežij – bločno modeliranje.

Razvrstitev $\mathcal{H}$ množice A je *hierarhija* nad množico A, če je $A_i \cap A_j \in \{\emptyset, A_i, A_j\}$. Hierarhija $\mathcal{H}$ je *polna*, če je $\bigcup \mathcal{H} = A$. Hierarhija $\mathcal{H}$ je *osnovna*, če za vsak $x \in \bigcup \mathcal{H}$ velja $\{x\} \in \mathcal{H}$.

Drugo poglavje

Postopki za določanje razčlemb

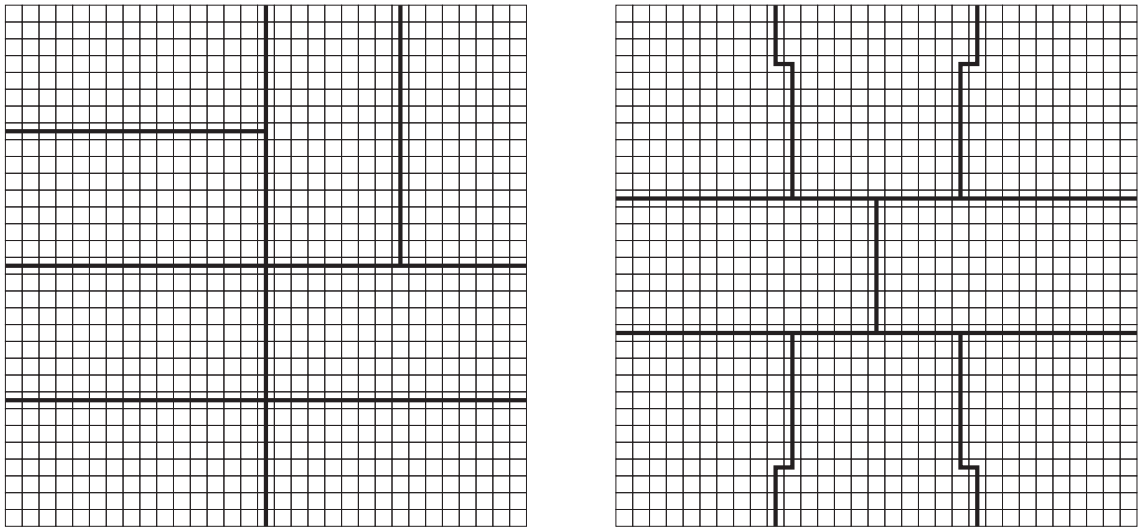
Znanih je precej postopkov za določanje čim boljših razčlemb grafa. Ločimo jih glede na različne lastnosti. Nekateri postopki na primer uporabljajo geometrijske lastnosti grafa (koordinate točk) in so zato uporabni samo za posebne vrste problemov, medtem ko drugi poleg grafa ne potrebujejo dodatnih podatkov. Večina postopkov res poišče iskano razčlembo, poznamo pa tudi take postopke, ki samo poskušajo izboljšati neko dano začetno razčlembo. Strogo deterministični postopki dajo pri istih podatkih vedno enak rezultat, obstajajo pa tudi postopki, ki uporabljajo kar precej naključnih odločitev. Postopki za razčlembo omrežij lahko uporabljajo druge znane postopke iz teorije grafov, ali pa problem obravnavajo samo kot poseben primer nekega drugega problema (kot je nelinearna optimizacija).

V tem poglavju bomo pregledali nekatere znane postopke za razčlembo omrežij. Skoraj vsi postopki bodo predstavljeni za grafe brez vrednosti v točkah ali na povezavah, saj je posplošitev na grafe z vrednostmi v večini primerov zelo preprosta. Prav tako se ne bomo ukvarjali s tehničnimi podrobnostmi, kot so nepovezani grafi, na katere moramo biti pri izvedbi postopka še posebej pozorni.

2.1 Razpolavljanje

Pri določanju k-razbitij grafa se zelo pogosto uporablja razpolavljanje. To je metoda, s katero najprej poiščemo čim boljše (običajno uravnoteženo) 2-razbitje danega grafa, nato pa rekurzivno razbijemo še obe skupini. Ideja izhaja iz dejstva, da je pri mnogih problemih razbitja število skupin enako neki potenci števila 2 (večina vzporednih računalnikov ima 2^p procesorjev).

Pri uporabi te metode moramo biti zelo pazljivi, saj se lahko zgodi, da bo končno k-razbitje daleč od optimalnega, tudi če znamo na vsakem koraku poiskati optimalno 2-razbitje [38]. Na sliki 2.1 je na desni strani prikazano 8-razbitje mreže 32×32, ki ga dobimo z razpolavljanjem (vsak korak je optimalen). Med skupinami tega razbitja je 128 povezav. Na desni strani iste slike pa je prikazano optimalno 8-razbitje istega grafa, kjer je povezav med skupinami samo 116.



Slika 2.1: *Slabo (levo) in optimalno (desno) 8-razbitje mreže* 32×32

Načeloma lahko z razpolavljanjem dobimo zelo slabo razbitje, a izkaže se, da za nekatere pomembne družine grafov (npr. ravninski grafi) dobimo rezultat, ki je kar blizu optimalnemu. Podobno velja tudi, če popustimo pri uravnoteženosti in dopustimo večje razlike v velikosti skupin.

2.2 Določanje razbitij z uporabo koordinat

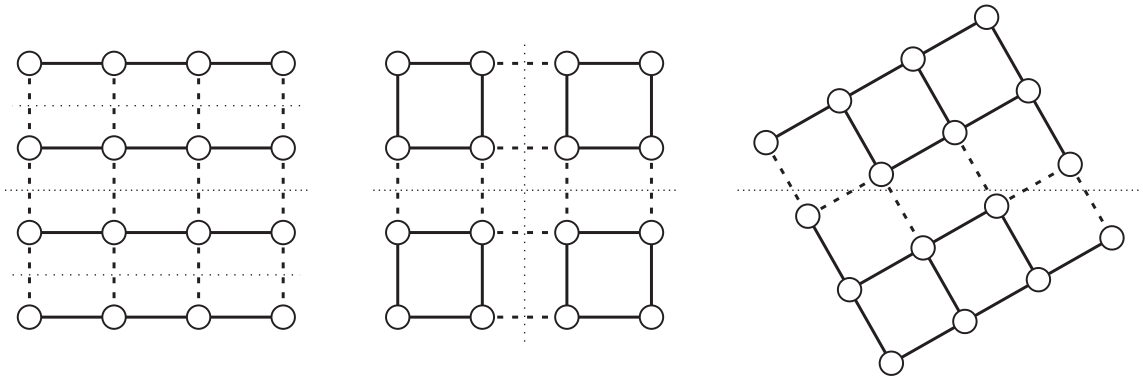
Včasih imamo na razpolago dodatne podatke o grafu, ki jih lahko koristno uporabimo pri določanju razbitja. V veliko primerih (na primer pri reševanju parcialnih diferencialnih enačb) tako poznamo tudi koordinate posameznih točk. Postopki, ki uporabijo te podatke o grafu, predpostavljajo, da sta točki, ki sta si blizu glede na položaj, tudi v grafu povezani s kratko potjo. Pravzaprav podatke o povezavah kar ignorirajo, kar zelo zmanjša njihovo uporabno vrednost v primerih, ko je povezanost točk v grafu pomembna (in znana). To pa tudi pomeni, da teh postopkov ne moremo posplošiti na grafe z uteženimi povezavami.

2.2.1 Koordinatno razpolavljanje

Koordinatno razpolavljanje je najpreprostejši postopek, ki temelji na koordinatah točk. S tem postopkom poiščemo hiperravnino, pravokotno na izbrano koordinatno os, ki točke razdeli v dve enako veliki skupini. Če imamo na primer točke v ravnini in iščemo hiperravnino (premico), pravokotno na os y, moramo poiskati vrednost $\overline{y}$, tako da bo polovica točk imela manjšo, druga polovica točk pa večjo koordinato y. Iskana premica je torej premica $y = \overline{y}$.

Rekurzivno lahko ta poskopek uporabimo na dva načina. Pri prvem izberemo vedno isto koordinatno os, pri čemer dobimo ozke trakove. To ni ravno najbolje, saj imajo trakovi dolge robove, kar ima za posledico veliko povezav med skupinami (glej levi del slike 2.2). Pri drugem pristopu pa koordinatne osi izbiramo ciklično (prvič x, drugič y, ..., potem spet x, ...). Tako dobimo skupine z boljšim razmerjem v dolžini robov, kar pomeni manj povezav med skupinami (glej srednji del slike 2.2).

Ena od težav s koordinatnim razpolavljanjem je v tem, da je odvisna od koordinat. V drugačnem koordinatnem sistemu lahko dobimo drugačno razbitje istega grafa (glej desni del slike 2.2).



Slika 2.2: *Težave pri koordinatnem razpolavljanju*

2.2.2 Stabilno razpolavljanje

Ta postopek (glej postopek 2.1) je podoben koordinatnemu razpolavljanju, razlika je samo v načinu, kako izberemo koordinatno os. Namesto ene od standarnih koordinatnih osi izberemo takšno os, da je vsota kvadratov razdalj posameznih točk do te osi čim manjša.

Ker je večina točk grafa blizu te osi, so si tudi njihove projekcije na pravokotno hiperravnino blizu. To nam daje upanje, da je točk, ki so blizu te hiperravnine, malo, kar pomeni, da bo tudi povezav, ki potekajo iz ene v drugo skupino točk, malo. Poglejmo si, kako poiščemo takšno os.

Naj bo $P_i \in \mathbb{R}^d$ položaj točke v_i, $i = 1, \ldots, n$. Iskana os je premica p, podana s točko T in normalizirano smerjo s, tako da je $p = \{T + \alpha s; \alpha \in \mathbb{R}\}$.

Ker je p taka premica, da minimizira vsoto kvadratov razdalj točk do premice, hitro opazimo, da težišče točk leži na njej. Težišče je namreč točka, ki minimizira vsoto kvadratov razdalj do točke. Zato izberemo

$$T = \frac{1}{n} \sum_{i=1}^{n} P_i$$

Postopek 2.1 *En korak stabilnega razpolavljanja*

Podatki: točke $P_1, P_2, \ldots, P_n$
Rezultat: skupini $\mathcal{V}_1$ in $\mathcal{V}_2$

izračunaj težišče $T = \sum_{i=1}^{n} P_i$
sestavi matriko $A = \sum_{i=1}^{n} (P_i - T)(P_i - T)^T$ (upoštevaj simetričnost)
izračunaj lastni vektor s matrike A, ki pripada njeni največji lastni vrednosti
izračunaj projekcije $\alpha_i = s^T P_i$
poišči mediano $\bar{\alpha}$ projekcij α_i
if $\alpha_i \leq \bar{\alpha}$ **then** točko v_i postavi v skupino $\mathcal{V}_1$
else točko v_i postavi v skupino $\mathcal{V}_2$

Naj bo P_i' ortogonalna projekcija točke P_i na premico p, torej $P_i' = T + \alpha_i s$, kjer je $\alpha_i = s^T(P_i - T)$. Potem je razdalja med točko P_i in premico p enaka $d_i = ||P_i - P_i'||_2$. Ker je vektor $P_i - P_i'$ pravokoten na s, je

$$d_i^2 = ||P_i - T||_2^2 - ||\alpha_i s||_2^2$$

Izbrati moramo takšno smer s, da bo vsota kvadratov razdalj čim manjša.

$$\begin{aligned}
\sum_{i=1}^{n} d_i^2 &= \sum_{i=1}^{n} ||P_i - T||_2^2 - \alpha_i^2 \\
&= \sum_{i=1}^{n} ||P_i - T||_2^2 - \sum_{i=1}^{n} s^T (P_i - T)(P_i - T)^T s \\
&= \sum_{i=1}^{n} ||P_i - T||_2^2 - s^T \left(\sum_{i=1}^{n} (P_i - T)(P_i - T)^T \right) s
\end{aligned}$$

Prvi sumand je neodvisen od s, torej je dovolj maksimizirati $s^T A s$, kjer je

$$A = \sum_{i=1}^{n} (P_i - T)(P_i - T)^T$$

Toda matrika A je simetrična, zato lahko za vektor s izberemo kar normaliziran lastni vektor, ki pripada največji lastni vrednosti matrike A. To sledi iz [23].

Časovna zahtevnost postopka je $\mathcal{O}(d^2 n)$, saj je edini časovno zahteven korak sestavljanje matrike A.

2.3 Določanje razbitij brez uporabe koordinat

Poleg postopkov, ki potrebujejo podatke o koordinatah točk, obstaja tudi skupina postopkov, ki teh koordinat ne potrebujejo. Za svoje delovanje uporabljajo samo podatke o povezanosti med točkami. Ta skupina postopkov je veliko bolj uporabna, saj podatkov o koordinatah velikokrat nimamo.

Postopek 2.2 *Požrešen postopek*

PODATKI: graf $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, število skupin k
REZULTAT: skupine $\mathcal{V}_1$, $\mathcal{V}_2$, ..., $\mathcal{V}_k$

```
for each v ∈ V do used[v] := false;
for i := 1 to k do begin
    izberi začetno točko u
    used[u] := true
    V_i := {u}
    while V_i dovolj majhna do begin
        izberi novo točko w
        used[w] := true
        V_i := V_i ∪ {w}
    end
end
```

2.3.1 Požrešni postopki

Ena od prvih idej, ki jo dobimo, ko želimo poiskati razbitje grafa, je naslednja. Izberimo točko, nato pa na nek način dodajajmo druge točke, dokler skupina ne bo dovolj velika. Vsaka točka, ki jo dodamo, mora biti na nek način najboljša (na primer takšna, da bo število povezav med skupinami čim manjše), ali pa takšna, da se skupina povečuje na določen način (na primer v širino). Pričnemo lahko tudi z več začetnimi točkami (vsaka v svoji skupini) in vzporedno povečujemo skupine. Postopek 2.2 prikazuje najbolj preprost postopek te vrste. Spada v skupino požrešnih postopkov.

Podrobnejši opis potrebujeta samo koraka za izbiro začetne in nove točke. Začetno točko za prvo skupino izberemo kot eno izmed točk, ki sta si na največji oddaljenosti v grafu. Začetno točko za druge skupine pa izberemo med prostimi točkami, ki so sosedne prejšnji skupini. Pri tem vedno izberemo tisto točko, ki ima najmanj prostih sosedov (točka v je prosta, če je used$[v] = false$). Na podoben način izberemo tudi novo točko w (med vsemi prostimi točkami, ki so sosedne trenutni skupini, izberemo tisto, ki ima najmanj prostih sosedov). Če trenutna skupina nima nobene proste sosedne točke, izberemo prosto točko, ki je sosedna katerikoli drugi skupini. Opisani postopek poskuša sestaviti povezane skupine, saj vedno (če je to le mogoče) izbiramo med točkami, ki so sosedne trenutni skupini.

Naslednji dobro poznan postopek, ki ga pogosto imenujemo tudi požrešni, je Farhatov postopek [19]. Podrobnejša analiza postopka pokaže, da je požrešna samo izbira začetnih točk.

Spet moramo podrobneje opisati korak, kjer izberemo začetno skupino točk. Pri prvi skupini to naredimo podobno, kot prej (v začetno skupino damo eno izmed točk, ki sta si na največji oddaljenosti v grafu). Pri določanju drugih začetnih skupin pa

Postopek 2.3 *Farhatov postopek*

Podatki: graf $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, število skupin k
Rezultat: skupine $\mathcal{V}_1$, $\mathcal{V}_2$, ..., $\mathcal{V}_k$

```
for each v ∈ V do used[v] := false
for i := 1 to k do begin
    izberi začetno skupino točk V_i
    for each u ∈ V_i do used[u] := true
    while V_i dovolj majhna do
        for each u ∈ V_i do
            for each w ∈ N(u) do
                if ¬used[w] then begin
                    used[w] := true
                    V_i := V_i ∪ {w}
                end
end
```

v prejšnji skupini poiščemo točko z najmanj prostimi sosedi (a vsaj enim), te sosede pa potem vzamemo za začetno razbitje.

Požrešni postopki so precej hitri. Njihova dobra lastnost je, da graf takoj razbijejo v k skupin, torej brez rekurzivnega razpolavljanja. To je še posebej uporabna lastnost, če željeno število skupin ni potenca števila 2. Tudi časovna zahtevnost je neodvisna od števila skupin.

Po drugi strani pa kvaliteta dobljenega razbitja ni ravno najboljša. Pri večjem številu skupin se tudi pokaže, da zadnje skupine niso povezane. Kvaliteta dobljenega razbitja je pogostokrat odvisna od izbire začetne točke. A ker so ti postopki hitri, si lahko privoščimo in poiščemo več razbitij (pri različnih začetnih točkah), potem pa med njimi izberemo najboljše razbitje.

Postopek Kernigan-Lin

Eden od prvih postopkov za iskanje razbitij je postopek Kernigan-Lin [30]. Ta ne poišče razbitja, pač pa poskuša izboljšati neko začetno razbitje, pri čemer ni potrebno, da bi skupine začetnega razbitja bile enako velike. Postopek je kasneje doživel številne izboljšave, največje sta opravila Fiduccia in Mattheyses. Ta pristop se po avtorjih imenuje KL/FM.

Prvotno je bilo mišljeno, da bi postopek izvedli na skupini naključnih razbitjih, potem pa izbrali najboljše dobljeno razbitje. Pri majhnih grafih dobimo kar smiselne rezultate, pri večjih pa je to precej neučinkovito. Zdaj se postopek uporablja predvsem za izboljšavo razbitij, ki jih dobimo z drugimi postopki.

Postopek zamenjuje po dve sosedni točki, ki sta v različnih skupinah, če pri tem

Postopek 2.4 *En korak postopka Kernigan-Lin*

PODATKI: omrežje $\mathcal{N} = (\mathcal{V}, \mathcal{L}, w)$, začetno razbitje $(\mathcal{V}_A, \mathcal{V}_B)$
REZULTAT: popravljeno razbitje $(\mathcal{V}_A, \mathcal{V}_B)$

```
for each v ∈ V do begin
    used[v] := false
    izračunaj vrednost diff[v]
end
k_0 := velikost prereza začetnega razbitja
for i := 1 to min(|V_A|, |V_B|) do begin
    med vsemi prostimi točkami poišči par (a_i, b_i) z največjo vrednostjo gain
    used[a_i] := true
    used[b_i] := true
    for each v ∈ N(a_i) ∪ N(b_i) do
        popravi vrednost diff[v], kot da bi zamenjali točki a_i in b_i
    k_i := k_i + gain(a_i, b_i)
end
poišči najmanjši j, tako da je k_j = min k_i
zamenjaj prvih j parov točk (a_i, b_i)
```

dobimo boljše razbitje. Torej na grafu deluje lokalno in je primerna dopolnitev postopkov, ki na grafu delujejo globalno (taki so na primer koordinatno razpolavljanje in spektralni postopki). Če želimo dobiti čim bolj uravnoteženo razbitje, dovolimo samo premike točk iz večje v manjšo (ali enako veliko) skupino. Postopek ponavljamo, dokler se razbitje izboljšuje, lahko pa tudi dovolimo, da se začasno poslabša in upamo, da se bo kasneje veliko bolj izboljšalo.

Postopek je lažje opisati za grafe z uteženimi povezavami, zato naj bo $\mathcal{N}$ omrežje $(\mathcal{V}, \mathcal{L}, w)$, množica točk $\mathcal{V}$ pa naj bo razdeljena na dve ločeni skupini $\mathcal{V}_A$ in $\mathcal{V}_B$.

Naj vrednost $\mathrm{diff}(v)$ pove, za koliko se zmanjša velikost prereza (vsota uteži na povezavah med skupinama), če točko v prestavimo v drugo skupino. Za $v \in \mathcal{V}_A$ je torej

$$\mathrm{diff}(v) = \mathrm{diff}(v, \mathcal{V}_A, \mathcal{V}_B) := \sum_{b \in \mathcal{V}_B} w((v\,;b)) - \sum_{a \in \mathcal{V}_A} w((v\,;a))$$

Če točko prestavimo iz ene skupine v drugo, se spremeni samo njena vrednost diff in vrednosti diff njenih sosedov.

Vrednost gain para točk $a \in \mathcal{V}_A$ in $b \in \mathcal{V}_B$ pa naj bo sprememba v velikosti prereza, če obe točki prestavimo v drugo skupino. Očitno je

$$\mathrm{gain}(a, b) = \mathrm{gain}(a, b, \mathcal{V}_A, \mathcal{V}_B) := \mathrm{diff}(a) + \mathrm{diff}(b) - 2w_{ab}$$

En korak postopka Kernigan-Lin prikazuje postopek 2.4.

2.3.2 Spektralno razpolavljanje

Spekralno razpolavljanje je popolnoma drugačen postopek od vseh do sedaj opisanih. Ne uporablja koordinat točk, pa tudi na samem grafu ne deluje, pač pa na matrični predstavitvi grafa. Medtem, ko drugi postopki skoraj ne potrebujejo operacij z realnimi števili, pa pri spektralnem razpolavljanju brez tega ne gre, saj postopek temelji na operacijah z matrikami in vektorji.

Bistvo spektralnega razpolavljanja je določiti razbitje omrežja na osnovi izbranega lastnega vektorja matrike, dobljene iz podatkov o omrežju [12]. Različne spektralne metode se razlikujejo večinoma po matriki, katere lastni vektor iščemo [35, 36].

Najbolj zahteven del teh metod je ravno izračun lastnega vektorja, saj je pri velikih omrežjih tudi matrika, katere lastni vektor računamo, precejšnja, a ponavadi redka. Čeprav so spektralne metode znane že okoli 20 let, se je zanimanje zanje povečalo šele v zadnjih nekaj letih, ko so računalniki postali dovolj zmogljivi in ko so izboljšali metode za iskanje lastnih vektorjev redkih matrik.

Standardni spekter

Matrika, katere lastni vektor iščemo, je matrika sosednosti A z elementi

$$a_{ij} = \begin{cases} 1 & \text{če sta točki } i \text{ in } j \text{ povezani} \\ 0 & \text{sicer} \end{cases}$$

Fiedler je ugotovil, da ima druga najmanjša lastna vrednost λ_2 te matrike zelo zanimivo lastnost. Če je omrežje povezano, c poljubna realna konstanta, x pa lastni vektor, ki pripada lastni vrednosti λ_2, potem je povezano tudi podomrežje, določeno s točkami, ki imajo v lastnem vektorju x pripadajočo komponento manjšo ali enako konstanti c. Isto velja za točke, ki imajo ustrezno komponento v lastnem vektorju večjo ali enako konstanti c. Konstanto c običajno izberemo tako, da bo točke omrežja razdelila v dve približno enako veliki skupini.

Laplaceov spekter

Matrika, katere lastni vektor iščemo, je Laplaceova matrika L z elementi

$$l_{ij} = \begin{cases} -1 & \text{če je } i \neq j \text{ ter sta točki } i \text{ in } j \text{ povezani} \\ \deg(i) & \text{če je } i = j \\ 0 & \text{sicer} \end{cases}$$

Matriko L lahko dobimo kot razliko $L = D - A$, kjer je D diagonalna matrika s stopnjami točk po diagonali, A pa matrika sosednosti, ali pa kot produkt $L = QQ^T$, kjer je Q incidenčna matrika velikosti $|\mathcal{V}| \times |\mathcal{L}|$ z elementi:

$$q_{ij} = \begin{cases} 1 & \text{če je točka } i \text{ začetek povezave } j \\ -1 & \text{če je točka } i \text{ konec povezave } j \\ 0 & \text{sicer} \end{cases}$$

Laplaceova matrika ima precej pomembnih lastnosti. Vse njene vrstične (in stolpčne) vsote so enake nič. Zaradi tega ima vsaj eno ničelno lastno vrednost, pripadajoči lastni vektor pa ima vse komponente enake. Ker je matrika L pozitivno semidefinitna, so vse ostale lastne vrednosti pozitivne, če je omrežje povezano. Če ni, je večkratnost ničelne lastne vrednosti natanko število povezanih komponent omrežja. Iz tega lahko sklepamo, da bo lastni vektor, ki pripada lastni vrednosti blizu ničle, razdelil točke omrežja v dve skoraj nepovezani skupini. V prvo skupino gredo točke, ki imajo v lastnem vektorju negativno komponento, v drugo pa točke s pozitivno komponento. Če želimo imeti bolj uravnoteženi skupini, lahko točke razporejamo tudi glede na srednjo vrednost (mediano) komponent lastnega vektorja namesto glede na ničlo. Z istim postopkom lahko nadaljujemo na manjših podomrežjih in tako prvotno omrežje delimo naprej. Ta metoda se imenuje RSB (rekurzivno spektralno razpolavljanje).

Namesto, da s pomočjo ene majhne lastne vrednosti razdelimo omrežje na dva dela, lahko s pomočjo dveh ali več majhnih lastnih vrednosti razdelimo omrežje na več delov.

Čeprav ima Laplaceov spekter precej zanimivih teoretičnih lastnosti, pomembnih pri razčlenjevanju omrežja, pa ima tudi nekaj slabosti. Ena je ta, da uporablja majhne lastne vrednosti. Lanczoseva metoda za iskanje nekaj (ne vseh) lastnih vektorjev namreč veliko hitreje najde lastni vektor, ki pripada največji lastni vrednosti, kot tistega, ki pripada najmanjši.

Normalni spekter

Matrika, katere lastni vektor iščemo, je normalna matrika N z elementi

$$n_{ij} = \begin{cases} \frac{1}{\sqrt{\deg(i)\deg(j)}} & \text{če sta točki } i \text{ in } j \text{ povezani} \\ 0 & \text{sicer} \end{cases}$$

Normalno matriko N dobimo kot kvadrat matrike X, ki temelji na povprečjih. Matriko X dobimo tako, da vzamemo matriko Q, na kateri temelji Laplaceova matrika L, ter vse elemente -1 zamenjamo z enicami. Vsako vrstico še delimo s kvadratnim korenom iz vrstične vsote, vsak stolpec pa s kvadratnim korenom iz stolpčne vsote (ta vsota je povsod enaka 2). Normalna matrika N je potem kvadratna matrika XX^T.

Podobno kot pri Laplaceovi matriki, lahko tudi tu takoj najdemo eno lastno vrednost in pripadajoči lastni vektor. Vsi elementi matrike N so namreč nenegativni in vse vrstične vsote so enake 1. To pomeni, da je N stohastična matrika, ter da ima lastno vrednost 1, pripadajoči lastni vektor pa ima vse komponente enake. Vse druge lastne vrednosti so po absolutni vrednosti manjše od 1 (če je omrežje povezano). Od tod sklepamo podobno kot v primeru Laplaceove matrike. Večkratnost lastne vrednosti 1 je ravno število povezanih komponent omrežja, iz lastnega vektorja, ki je

po absolutni vrednosti blizu 1, pa dobimo razbitje omrežja na dve med seboj skoraj nepovezani podomrežji.

2.3.3 Večnivojske metode

Z večnivojskimi metodami veliko omrežje poskusimo najprej smiselno zmanjšati, nato pa poiskati razbitje manjšega omrežja, kar je običajno precej hitreje. Razbitje osnovnega omrežja potem dobimo iz razbitja manjšega omrežja in podatkov o tem, kako je bilo to manjše omrežje zgrajeno.

a) **Poenostavitev omrežja (zmanjšamo število točk)**

Točke in povezave osnovnega omrežja najprej utežimo z utežjo 1, nato pa sestavimo zaporedje manjših omrežij $\mathcal{G}_i(\mathcal{V}_i, \mathcal{E}_i)$, sestavljenih iz osnovnega omrežja $\mathcal{G}_0(\mathcal{V}_0, \mathcal{E}_0)$, tako da je $|\mathcal{V}_i| < |\mathcal{V}_{i-1}|$. Manjše omrežje običajno dobimo tako, da na prejšnjem omrežju poiščemo čim večje ujemanje, nato pa združimo krajišči povezav, ki so v ujemanju. Pri tem je utež nove točke vsota uteži prejšnjih točk, podobno pa velja tudi za povezave. S postopkom končamo, ko dobimo dovolj majhno omrežje, oziroma ko se novo omrežje od prejšnjega le še malo razlikuje.

b) **Razbitje manjšega omrežja**

Poenostavljeno omrežje vsebuje dovolj informacije (uteži na točkah in povezavah), da lahko poiščemo uravnoteženo razbitje, ki bo minimiziralo število povezav med posameznimi skupinami.

c) **Projekcija razbitja nazaj na osnovno omrežje**

Vsako točko v manjšem omrežju lahko nadomestimo z množico točk prvotnega omrežja, iz katerih je bila dobljena. Vendar nam to ne prinese najboljšega razbitja. Zato razbitje prvotnega omrežja rajši sestavimo po korakih (obratno od poenostavljanja), na vsakem koraku pa dobljeno razbitje izboljšamo še s kakšno hevristično metodo.

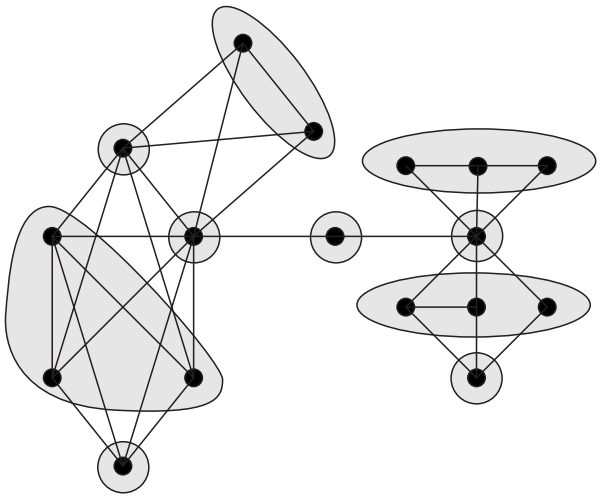
Najbolj znan program za razčlembo velikih grafov, ki deluje po tem postopku [27, 29, 28] je METIS [26]. Avtor programa je George Karypis iz Univerze v Minnesoti. Zadnja različica programskega paketa nosi oznako 4.0.1 (november 1998). Programi so v celoti napisani v jeziku ANSI C. To pomeni, da lahko izvorno kodo, ki jo dobimo na omrežju, prevedemo na kateremkoli operacijskem sistemu, ki ima standarden prevajalnik za jezik C, torej skoraj povsod. Iz izvorne kode si lahko naredimo knjižnice funkcij, ki jih potem uporabljamo v svojih programih. Na omrežju dobimo tudi že prevedene programe za okolje Windows. Sorodna paketu METIS sta še paketa parMETIS in hMETIS. Prvi je vzporedna različica osnovnega paketa, drugi pa je namenjen razčlembi hipergrafov.

2.4 Zaporedne modularne razčlembe

Modul grafa $\mathcal{G} = (\mathcal{V}, \mathcal{L})$ je takšna skupina točk $\mathcal{C} \subseteq \mathcal{V}$, da za vsako točko $v \in \mathcal{V} \setminus \mathcal{C}$ velja: v je povezana z vsemi točkami iz $\mathcal{C}$ ali pa v ni povezana z nobeno točko iz $\mathcal{C}$. *Trivialni moduli* so $\emptyset$, $\mathcal{V}$ in vse množice, ki vsebujejo eno samo točko.

Moduli imajo močne algebraične lastnosti [14]. Če sta $\mathcal{C}$ in $\mathcal{D}$ modula z nepraznim presekom, potem so presek, unija in razlika tudi moduli. Ločena modula sta bodisi povezana (vsaka točka prvega modula je povezana z vsako točko drugega modula), bodisi nepovezana (nobena točka prvega modula ni povezana z nobeno točko drugega modula).

Modularna razčlemba je še ena od možnosti, kako dobiti razbitje danega grafa. Dobimo jo tako, da poiščemo čim bolj grobo razbitje množice točk na vsaj dva modula (tako razbitje je enolično določeno), potem pa rekurzivno razbijemo vsak netrivialen modul posebej. Slika 2.3 prikazuje eno od možnih razbitij grafa na module.



Slika 2.3: *Eno od možnih razbitij grafa na module*

Za določanje modularnih razčlemb obstaja veliko različnih postopkov. Enostavni imajo časovno zahtevnost $\mathcal{O}(n^3)$ ali celo $\mathcal{O}(n^4)$, obstajajo pa tudi postopki linearne časovne zahtevnosti [14, 32], ki pa so precej zapleteni.

2.5 Pajek

Na koncu ne smemo pozabiti omeniti programa Pajek [4] za analizo velikih omrežij. Avtorja sta V. Batagelj (Univerza v Ljubljani, Fakulteta za matematiko in fiziko) in A. Mrvar (Univerza v Ljubljani, Fakulteta za družbene vede). Vsi postopki, ki bodo opisani v nadaljevanju, so že vključeni v ta program.

Poleg običajnih (usmerjenih, neusmerjenih in mešanih) omrežij Pajek podpira tudi delo z dvovrstnimi omrežji, kjer imamo točke dveh vrst (dve ločeni množici točk), povezave pa potekajo samo iz ene v drugo množico točk in delo s časovnimi omrežji, ki se spreminjajo skozi čas. Pajek omogoča tudi delo z drugimi strukturami, kot so razbitja, hierarhije, permutacije in vektorji.

V Pajka je vključeno veliko znanih postopkov za analizo omrežij, postopkov za izračun najrazličnejših vrednosti na točkah in povezavah in postopkov za razčlenjevanje, omogoča pa tudi prikaz omrežja na različne načine.

Tretje poglavje

Prerezi in otoki

3.1 Točkovni otoki

Pogosto nas v omrežju zanimajo skupine povezanih točk, katerih vrednosti so večje od vrednosti točk v okolici. Uveljavljen postopek, kako določiti takšne skupine točk (otoke), je z uporabo točkovnih prerezov. Točkovni prerez danega omrežja na nivoju t dobimo tako, da iz omrežja odstranimo vse točke (in pripadajoče povezave), katerih vrednost je manjša od t. Za lažjo predstavo in razumevanje postopkov bomo vrednosti v dani točki rekli kar *višina* točke. Omrežje s točkami na danih višinah nam torej določa nekakšno pokrajino.

Definicija 3.1: *Točkovni prerez* omrežja $\mathcal{N} = (\mathcal{V}, \mathcal{L}, p)$ na nivoju t je podomrežje $\mathcal{N}(t) = (\mathcal{V}', \mathcal{L}', p)$, določeno z

$$\begin{aligned} \mathcal{V}' &= \{v \in \mathcal{V} : p(v) \geq t\} \\ \mathcal{L}' &= \{(u\,;v) \in \mathcal{L} : u, v \in \mathcal{V}'\} \end{aligned}$$

Definicija 3.2: Neprazna skupina točk $\mathcal{C} \subseteq \mathcal{V}$ je *točkovni otok*, če njene točke določajo povezan podgraf in so višine sosednih točk manjše ali enake višinam točk iz skupine.

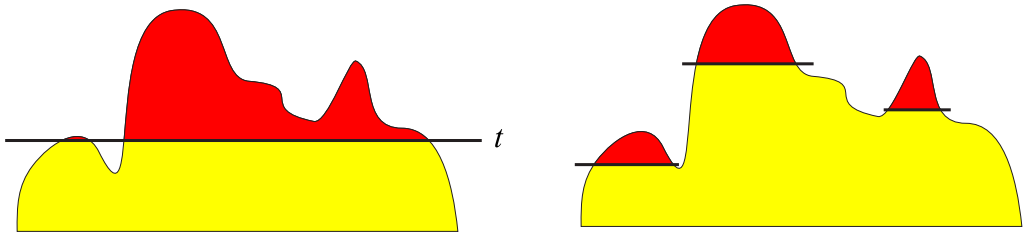
$$\max_{u \in N(\mathcal{C})} p(u) \leq \min_{v \in \mathcal{C}} p(v)$$

Definicija 3.3: Točkovni otok $\mathcal{C} \subseteq \mathcal{V}$ je *pravi točkovni otok*, če so višine sosednih točk strogo manjše od višin točk iz skupine.

$$\max_{u \in N(\mathcal{C})} p(u) < \min_{v \in \mathcal{C}} p(v)$$

Skupine točk, ki ustrezajo povezanim komponentam točkovnega prereza danega omrežja, izpolnjujejo vse zahtevane pogoje za prave otoke. Komponenta je povezana, višine točk v komponenti so vse večje ali enake t, višine točk v okolici pa so vse manjše od t. Komponente točkovnega prereza so torej pravi točkovni otoki.

Od izbire nivoja t je odvisno, na koliko povezanih komponent razpade omrežje in kako velike so te komponente. Dobimo lahko komponente najrazličnejših velikosti – od izoliranih točk do neobvladljivo velikih komponent. To pa ni ravno tisto, kar si želimo. Rajši bi imeli skupine, katerih velikosti bi lahko vnaprej določili. Da pa bi to dosegli, moramo v omrežju narediti več rezov na različnih nivojih (glej sliko 3.1).



Slika 3.1: *Točkovni prerez na enem in več nivojih*

Potrebujemo torej učinkovit postopek za določanje maksimalnih pravih točkovnih otokov omejene velikost. Pri tem bomo velikost otokov omejili navzdol in navzgor – otok ne sme biti niti premajhen, niti prevelik.

3.1.1 Splošni točkovni otoki

Postopek za določanje maksimalnih pravih točkovnih otokov omejene velikosti temelji na točkovnih prerezih omrežja na vedno nižjih nivojih. Delovanje postopka bomo najlažje razumeli, če si predstavljamo, da celotno omrežje potopimo v vodo, nato pa gladino vode postopoma znižujemo in opazujemo, kakšni otoki se pojavljajo. Pri tem nam gladine vode ni treba zniževati zvezno, pač pa samo po višinah točk od najvišje do najnižje.

Ko iz vode pokuka prva točka, je to vrh prvega otoka. Ko zagledamo drugo točko, je ta lahko vrh drugega otoka, ali pa pripada prvemu otoku (če je povezana s prvo točko). V splošnem iz vode že gleda nekaj otokov. Ko zagledamo novo točko v, je ta lahko povezana s katero od že vidnih točk, ali pa ne. Če ni povezana z nobeno vidno točko, je to vrh novega otoka, sicer pa združimo vse otoke, ki imajo kakšno točko povezano s točko v, dodamo točko v in tako dobimo nov, večji otok. Otoki, ki jih združimo v večji otok, postanejo *podotoki* novega otoka, točka v pa njegovo *pristanišče* (tako jo poimenujemo zato, ker je to najnižja točka na otoku).

Postopek 3.1 za določanje maksimalnih pravih točkovnih otokov omejene velikosti je sestavljen iz dveh delov. V prvem (pripravljalnem) delu postopka sestavimo množico *otoki*, ki določa, kako so otoki vsebovani eden v drugem. Za vsak otok izvemo, kaj so njegovi podotoki, katera točka je pristanišče ter ali je otok pravi. V drugem delu postopka pa v tej hierarhiji otokov poiščemo vse maksimalne prave točkovne otoke, katerih velikost je omejena s številoma min in max. Iskane otoke nam postopek vrne v množici L.

Postopek 3.1 *Maksimalni pravi točkovni otoki omejene velikosti*

PODATKI: omrežje $\mathcal{N} = (\mathcal{V}, \mathcal{L}, p)$, števili min in max
REZULTAT: množica L vseh maksimalnih pravih točkovnih otokov velikosti vsaj min in največ max

$otoki := \emptyset$
uredi $\mathcal{V}$ nenaraščajoče glede na p
for each $v \in \mathcal{V}$ (v dobljenem vrstnem redu) **do begin**
  $otok :=$ **new** $Otok()$
  $otok.pristan := v$
  $otok.podotoki := \{o \in otoki : o \cap N(v) \neq \emptyset\}$
  $otoki := otoki \cup \{otok\} \setminus otok.podotoki$
  for each $o \in otok.podotoki$ **do** $o.pravi := p(o.pristan) > p(v)$
end
for each $o \in otoki$ **do** $o.pravi :=$ **true**

$L := \emptyset$
while $otoki \neq \emptyset$ **do begin**
  izberi $otok \in otoki$
  $otoki := otoki \setminus \{otok\}$
  if $|otok| < min$ **then delete** $otok$
  else if $|otok| > max \vee \neg otok.pravi$ **then begin**
    $otoki := otoki \cup otok.podotoki$
    delete $otok$
  end
  else $L := L \cup \{otok\}$
end

Pripravljalni del postopka pričnemo z urejanjem množice točk $\mathcal{V}$ nenaraščajoče po njihovih višinah. Nato spuščamo gladino vode od najvišje do najnižje točke, tako da vsakič iz vode pogleda ena nova točka (točke pregledamo v dobljenem vrstnem redu). Da postopka ne zapletamo po nepotrebnem, se točke, ki so na isti višini, iz vode ne pojavijo vse hkrati, pač pa ena za drugo (vrstni red ni pomemben).

Z vsako novo točko v, ki pogleda iz vode, dobimo natanko en nov otok. Točka v je njegovo pristanišče, podotoki pa so tisti otoki, ki vsebujejo kakšno točko, povezano s točko v. Nov otok dodamo v trenutno množico otokov, iz nje pa odstranimo vse njegove podotoke. Teh otokov s tem ne izgubimo, saj do njih lahko še vedno pridemo kot do podotokov novega otoka.

Ker nas zanimajo samo pravi otoki, si moramo za vsak otok še zabeležiti, ali je pravi. Tega pa ne moremo storiti takoj, ko otok nastane, pač pa šele kasneje, ko otok postane podotok večjega otoka. Otok je pravi, če je njegovo pristanišče višje od pristanišča večjega otoka. Vsi otoki, ki nam ostanejo ob koncu postopka, so pravi.

Ko zmanjka vode, nam ostane toliko otokov, kolikor povezanih komponent ima omrežje. Nas zanimajo samo pravi otoki omejene velikosti, pri čemer je velikost otoka omejena navzdol in navzgor. Te poiščemo v drugem delu postopka. Pregledati moramo vse otoke iz množice *otoki*. Ko izberemo nek otok iz te množice, ga odstranimo iz množice in preverimo njegovo velikost. Če je otok premajhen, ga zavržemo, če je prevelik ali pa ni pravi, ga razbijemo na podotoke (potopimo pristanišče) in te dodamo v množico še nepregledanih otokov, če pa je ustrezne velikosti in je pravi, si ga zapomnemo. Postopek ponavljamo tako dolgo, da pregledamo vse otoke.

3.1.2 Enostavni točkovni otoki

Definicija 3.4: Skupina točk $\mathcal{C} \subseteq \mathcal{V}$ je *lokalni točkovni vrh*, če je pravi točkovni otok in imajo vse njene točke enako višino.

Definicija 3.5: Točkovni otok, ki ima en sam lokalni točkovni vrh, imenujemo *enostaven točkovni otok*.

Poiskati vse maksimalne enostavne prave točkovne otoke omejene velikosti je razmeroma preprosto. Po vrsti moramo pregledati vse točke. Če je točka v lokalnem točkovnem vrhu še neodkritega otoka, smo našli nov enostavni točkovni otok. Poiskati moramo še vse točke, ki jih lahko dodamo temu otoku, tako da bo ostal enostaven. To storimo tako, da po vrsti dodajamo najvišje točke iz soseščine trenutnega otoka, dokler je izpolnjen pogoj za otok (končamo lahko že prej, če dosežemo največjo dovoljeno velikost otoka). Če je v soseščini trenutnega otoka več točk na isti višini, izberemo katerokoli izmed njih. Pred koncem postopka je morda potrebno še odstraniti nekaj točk (v obratnem vrstnem redu, kot smo jih dodajali), da dobimo pravi otok. Če je dobljeni otok premajhen, ga zavržemo, sicer pa je to eden od iskanih otokov.

Izvedba tega postopka pa ni tako enostavna. Najprej moramo ugotoviti, katere točke pripadajo lokalnim točkovnim vrhovom. To storimo tako da po vrsti pregledamo vse še nepregledane točke. Če točka nima višjih sosedov, moramo poiskati vse točke na isti višini, ki so iz nje dosegljive. Če je v soseščini teh točk kakšna višja točka, si točke označimo za pregledane, sicer pa si jih označimo kot lokalni točkovni vrh. Med postopkom moramo vzdrževati tudi množico sosednih točk trenutnega otoka, iz katere vedno odstranjujemo najvišjo točko. Te točke je najbolje hraniti v kopici (najvišja točka bo vedno v korenu kopice), kar pomeni, da potrebujemo še operacije za delo s kopico.

Veliko manj dela pa imamo, če samo dopolnimo postopek 3.1 za določanje splošnih točkovnih otokov, tako da ne bo vrnil vseh maksimalnih pravih točkovnih otokov omejene velikosti, pač pa samo tiste izmed njih, ki so tudi enostavni.

Pripravljalni del postopka, kjer določimo vseh n otokov, dopolnimo tako, da vsakemu otoku določimo, kakšne vrste je. Vrsta otoka je lahko FLAT (vse točke

Postopek 3.2 *Maksimalni enostavni pravi točkovni otoki omejene velikosti*

PODATKI: omrežje $\mathcal{N} = (\mathcal{V}, \mathcal{L}, p)$, števili min in max
REZULTAT: množica L vseh maksimalnih enostavnih pravih točkovnih otokov velikosti vsaj min in največ max

```
otoki := ∅
uredi V nenaraščajoče glede na p
for each v ∈ V (v dobljenem vrstnem redu) do begin
    otok := new Otok()
    otok.pristan := v
    otok.podotoki := {o ∈ otoki : o ∩ N(v) ≠ ∅}
    otoki := otoki ∪ {otok} \ otok.podotoki
    for each o ∈ otok.podotoki do o.pravi := p(o.pristan) > p(v)
    if |otok.podotoki| = 0 then otok.vrsta := FLAT
    else if |otok.podotoki| = 1 then begin
        o := podotok
        if o.vrsta ≠ FLAT then otok.vrsta := o.vrsta
        else if p(o.pristan) = p(v) then otok.vrsta := FLAT
        else otok.vrsta := SINGLE
    end
    else begin
        for each o ∈ otok.podotoki do begin
            ok := o.vrsta = FLAT ∧ p(o.pristan) = p(v)
            if ¬ok then break
        end
        if ok then otok.vrsta := FLAT
        else otok.vrsta := MULTI
    end
end
for each o ∈ otoki do o.pravi := true

L := ∅
while otoki ≠ ∅ do begin
    izberi otok ∈ otoki
    otoki := otoki \ {otok}
    if |otok| < min then delete otok
    else if |otok| > max ∨ ¬otok.pravi ∨ otok.vrsta = MULTI then begin
        otoki := otoki ∪ otok.podotoki
        delete otok
    end
    else L := L ∪ {otok}
end
```

otoka imajo isto višino), SINGLE (otok ima en sam lokalni točkovni vrh) ali MULTI (otok ima več lokalnih točkovnih vrhov). Poglejmo si podrobneje, kako določimo vrsto novega otoka.

- Če otok nima podotokov, je sestavljen iz ene same točke, torej je vrste FLAT.
- Če ima otok samo en podotok, imamo tri možnosti. Če podotok ni vrste FLAT, torej če je vrste SINGLE ali MULTI, potem je tudi nov otok enake vrste. Če pa je podotok vrste FLAT, je vrsta novega otoka odvisna od višine pristanišča podotoka in višine nove točke. Če sta višini enaki, je tudi nov otok vrste FLAT, sicer pa dobimo otok vrste SINGLE.
- Če pa ima otok več podotokov, moramo najprej preveriti, kakšni so ti podotoki. Če so vsi vrste FLAT in so vsa njihova pristanišča na isti višini kot nova točka, potem je tudi nov otok vrste FLAT, sicer pa je vrste MULTI.

Pogoj v drugem delu postopka, ki odloči, ali je otok potrebno razbiti na podotoke in pregledati vsak podotok posebej, dopolnimo tako, da dodatno preverimo še, ali je otok vrste MULTI (tudi v tem primeru ga je potrebno razbiti). Postopek 3.2 prikazuje tako dopolnjen postopek 3.1, kjer je stara koda napisana v svetlejši barvi, da so dopolnitve bolj vidne.

3.1.3 Izvedba

Predpostavimo, da je omrežje predstavljeno s strukturo `Graph`. Ker sama predstavitev omrežja v postopku ni bistvenega pomena, se v podrobnosti o tej strukturi ne bomo spuščali. Točko v omrežju bomo predstavili s strukturo `Vert`. Ta bo vsebovala celoštevilsko lastnost `id` z vrednostjo med 0 in $n-1$, ki bo enolično določala točko. Vsebovala bo tudi realno lastnost `value`, v kateri bomo hranili vrednost točke. Na druge podatke o točki (na primer seznam sosedov) se v izvedbi postopka ne bomo sklicevali, zato so v opisu strukture izpuščeni.

```
struct Vert
{
   int id;                 // enolična oznaka iz [0,n)
   double value;           // vrednost
};
```

Izkaže se, da je najprimernejša podatkovna struktura za predstavitev otoka drevo. Vsak otok namreč nastane z združitvijo točke, ki ji rečemo pristanišče (koren drevesa) z nekaj manjšimi otoki, ki jim rečemo podotoki (poddrevesa). Množica podotokov je lahko tudi prazna, kar pomeni, da ustrezno vozlišče v drevesu ne bo imelo poddreves (list).

Ker vsako vozlišče v drevesu ustreza enemu otoku, ga bomo predstavili s strukturo `VertIsland`. Točka v vozlišču (lastnost `port`) je pristanišče otoka, poddrevesa pa predstavljajo podotoke. Ker ne vemo vnaprej, koliko podotokov bo imel

posamezen otok, bomo poddrevesa vodili v obliki seznama. Tako bomo v vsakem vozlišču imeli kazalec na prvo poddrevo (lastnost `subislands`), ker pa skoraj vsako vozlišče ustreza tudi podotoku nekega večjega otoka, še kazalec na naslednje poddrevo v seznamu (lastnost `next`). Poleg tega bomo v vsakem vozlišču imeli zapisano še velikost otoka (lastnost `size`), da nam kasneje ne bo treba pregledovati celih dreves, da bi se dokopali do tega podatka.

Pri sestavljanju novega otoka bo za dano točko v podotoku potrebno čim hitreje najti koren drevesa, v katerem se ta točka nahaja, da bomo to drevo pripeli kot poddrevo večjemu drevesu. Da bomo koren lahko poiskali, bomo v vsakem vozlišču potrebovali še kazalec na prednika v drevesu (lastnost `parent`). Pozabiti pa ne smemo še na logično lastnost `regular`, ki določa, ali je otok pravi.

```
struct VertIsland
{
   int size;                  // velikost otoka
   Vert *port;                // pristanišče
   VertIsland *subislands;    // kazalec na prvi podotok
   VertIsland *next;          // kazalec na naslednji podotok
   VertIsland *parent;        // kazalec na prednika
   bool regular;              // ali je otok pravi
};
```

Za predstavitev vseh otokov, torej za vsa drevesa skupaj, bomo potrebovali natanko n vozlišč. Namesto da bi pomnilnik rezervirali za vsako vozlišče posebej, lahko ves potreben pomnilnik zanje rezerviramo kar v enem kosu (tabela vozlišč). Vozlišča v tabeli `islands` bomo zasedali po vrsti od začetka proti koncu tabele.

Za dano točko bo treba znati hitro ugotoviti, ali je že vidna, oziroma ali že pripada kateremu od dreves, s katerimi so predstavljeni otoki. Ker imamo lahko več točk na isti višini, si z višinami točk pri odgovoru na to vprašanje ne moremo pomagati. Rešitev je v dodatni tabeli `pos`, ki za vsako točko vsebuje kazalec na koren drevesa, ki predstavlja otok, ki ima to točko za pristanišče. Če točka še ni vidna, ustrezni kazalec ne kaže nikamor.

Pripravljalni del postopka je zdaj preprost. Po vrsti pregledujemo točke iz nenaraščajoče urejene množice $\mathcal{V}$ in vsakič zasedemo eno vozlišče v tabeli. To vozlišče postane koren drevesa, ki predstavlja otok, ki ima trenutno točko za pristanišče. Na začetku je to drevo brez poddreves, torej je njegova velikost enaka 1.

Potem pregledamo vse sosede trenutne točke. Če je sosed že v katerem od dreves (to vidimo v tabeli `pos`), moramo poiskati koren tega drevesa in drevo pripeti kot poddrevo k novemu vozlišču (najlažje ga pripnemo na začetek seznama poddreves). Kazalce na prednike v vseh vozliščih na poti do korena preusmerimo neposredno na koren novega drevesa. S tem si precej skrajšamo morebitne naslednje sprehode proti korenu drevesa (podobno kot pri postopku Union-Find).

Pri iskanju korena drevesa, ki ustreza podotoku, se lahko pripeti, da je to drevo že pripeto kot podotok k novemu otoku. V tem primeru se iskanje korena ustavi pri vozlišču, ki ustreza novemu otoku. Seveda takrat ne smemo narediti ničesar več.

Program 3.1: *Maksimalni pravi točkovni otoki omejene velikosti*

```
int *vertIslands(Graph *g, int min, int max)
{
   Vert *vert = vertices(g);
   int n = size(vert);
   qsort(vert, n, sizeof(*vert), vertDescending);

   VertIsland *islands = new VertIsland[n];
   VertIsland **pos = new VertIsland *[n];
   for (int i = 0; i < n; ++i) pos[i] = NULL;
   for (int i = 0; i < n; ++i) {
      Vert *v = vert[i];
      VertIsland *p = pos[v->id] = &islands[i];
      p->size = 1;
      p->port = v;
      p->subislands = p->next = p->parent = NULL;
      p->regular = true;
      for (Vert *u in neighbors(g, v)) {
         if (pos[u->id] != NULL) {
            VertIsland *q = pos[u->id];
            while (q->parent != NULL) {
               VertIsland *t = q;
               q = q->parent;
               t->parent = p;
            }
            if (p != q) {
               q->parent = p;
               q->next = p->subislands;
               p->subislands = q;
               p->size += q->size;
               q->regular = q->port->value > v->value;
            }
         }
      }
   }

   int c = 0;
   int *part = new int[n];
   while (n--) {
      VertIsland *p = &islands[n];
      if (p->parent != NULL)
         markVert(part, p, part[p->parent->port->id], p);
      else if (p->size < min)
         markVert(part, p, 0, p);
      else if (p->size > max || !p->regular)
         markVert(part, p, 0, NULL);
      else
         markVert(part, p, ++c, p);
   }

   delete [] pos;
   delete [] islands;
   return part;
}
```

Zdaj lahko med vsemi dobljenimi otoki poiščemo maksimalne prave točkovne otoke omejene velikosti. Rezultat bomo vrnili v obliki razbitja – vsak otok bo svoja skupina, točke ki ne pripadajo nobenemu ustreznemu otoku, pa bomo dali v

posebno skupino z oznako 0. Razbitje predstavimo s tabelo celih števil, kjer vsaki točki določimo oznako skupine.

Iskane otoke bi lahko poiskali s preprosto rekurzivno funkcijo, kjer bi pregledali vsa vozlišča dobljenih dreves. Če bi bil ustrezen otok premajhen, bi iskanje v globino prekinili, če bi bil prevelik ali nepravi, bi postopek ponovili na vseh poddrevesih, sicer pa bi ga označili kot novo skupino. A pri velikih omrežjih uporaba rekurzije ni priporočjiva, saj lahko hitro pride do prekoračitve na skladu. Ker pa so vsa vozlišča dreves zapisana v tabeli, se rekurziji lahko izognemo.

Vozlišča dreves so v tabeli urejena tako, da je koren vsakega drevesa vedno za koreni vseh svojih poddreves. Ker iščemo maksimalne otoke ustrezne velikosti, bomo vozlišča pregledovali s konca proti začetku tabele. Tako bomo vsak otok pregledali prej, kot kateregakoli od njegovih podotokov. Poleg tega bomo vrednost lastnosti `parent` v vozlišču uporabili za to, da bomo razlikovali med nepregledanimi točkami in tistimi točkami, za katere že vemo, v katero skupino razbitja jih moramo postaviti. Pri nepregledanih točkah bo vrednost te lastnosti enaka `NULL`, pri drugih pa bo kazalec kazal na neposrednega prednika, torej na točko, ki bo določala oznako skupine. Po izteku pripravljalnega dela postopka je vrednost lastnosti `parent` v vseh vozliščih, ki predstavljajo glavne otoke, enaka `NULL`, pri tistih, ki predstavljajo podotoke, pa kaže na enega od prednikov v drevesu. To pa je ravno tisto, kar potrebujemo.

Pregled točk torej poteka od konca proti začetku tabele. Če je znan prednik v drevesu, bomo točko postavili v isto skupino, v kateri je prednik, sicer pa otok, ki ustreza trenutni točki še ni bil pregledan. Če je ta otok premajhen, prevelik ali pa ni pravi, bomo točko postavili v skupino 0, sicer pa v novo skupino.

Pri pregledu vsake točke moramo točkam v korenih poddreves še nastaviti kazalce na prednika, da bomo kasneje vedeli, kako jih obravnavati. Če moramo vse točke v poddrevesih postaviti v isto skupino kot trenutno točko, bomo kazalce na prednika v korenih poddreves nastavili na trenutno vozlišče. To naredimo v primerih, ko točko postavimo v isto skupino kot njenega prednika, če točko postavimo v novo skupino ali pa če je ustrezen otok premajhen. V vseh drugih primerih (otok je prevelik ali pa ni pravi) kazalce na prednika v korenih poddreves postavimo na `NULL`. S tem si označimo, da ustrezen otok še ni pregledan.

Obe nalogi skupaj (postavljanje točke v skupino in nastavljanje kazalcev na prednika) opravlja pomožna funkcija `markVert`. Točko, ki ustreza pristanišču otoka, ki ga predstavlja vozlišče `p`, postavi v skupino `c`, korenom vseh poddreves pa nastavi prednika na `q`. Oznake skupin za posamezne točke vodimo v tabeli `part`, ki jo funkcija dobi za parameter. Ta tabela bi lahko bila tudi globalna, a to ne spada k pravilom lepega programiranja.

Razširitev opisane izvedbe postopka, tako da bi lahko iskali tudi maksimalne enostavne prave otoke omejene velikosti, je preprosta. Potrebno je samo slediti dopolnitvam v postopku 3.2, zato se s tem tu ne bomo ukvarjali.

```
void markVert(int *part, VertIsland *p, int c, VertIsland *q)
{
   part[p->port->id] = c;
   for (p = p->subislands; p != NULL; p = p->next) p->parent = q;
}
```

3.1.4 Časovna zahtevnost

Poglejmo si še, koliko časa porabi opisani postopek za določanje maksimalnih pravih točkovnih otokov omejene velikosti. Za urejanje množice točk $\mathcal{V}$ potrebujemo $\mathcal{O}(n \log n)$ časa.

V pripravljalnem delu postopka pregledamo vse točke, za vsako točko pa še vse njene sosede. Pogojni stavek, kjer preverjamo, ali je sosed trenutne točke že viden, se tako izvrši $\mathcal{O}(m)$ krat. Znotraj tega pogojnega stavka pa imamo še eno zanko, kjer se od vozlišča, ki predstavlja soseda trenutne točke v podotoku, pomikamo proti korenu. Ker med pomikanjem proti korenu vse povezave preusmerjamo neposredno na koren (podobno kot pri postopku Union-Find), porabimo za vse operacije skupaj $\mathcal{O}(m\,\alpha(n))$ časa, kjer je α obrat Ackermannove funkcije, ki narašča zelo počasi (še pri $n = 2^{65536}$ ima vrednost manjšo od 5).

V zaključnem delu postopka imamo eno samo zanko, s katero pregledamo vsa vozlišča dreves, torej se izvrši natanko n-krat. V vsaki ponovitvi te zanke imamo klic pomožne funkcije, v kateri pa je še ena zanka. Ta teče po vseh poddevesih. Vseh ponovitev te zanke je natanko toliko, kot je vseh poddreves, torej za zaključni del postopka potrebujemo $\mathcal{O}(n)$ časa.

Ker je vrednost funkcije α za smiselne vrednosti n manjša od majhne konstante, je časovna zahtevnost celotnega postopka $\mathcal{O}(\max\{n \log n, m\})$.

3.1.5 Primer

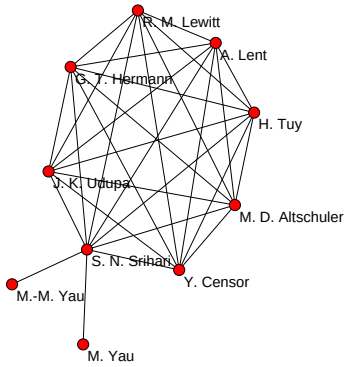
Poglejmo si omrežje avtorjev, ki je bilo dobljeno iz bibliografije [11] iz *Computational Geometry Database* `geombib`, verzija februar 2002 [25].

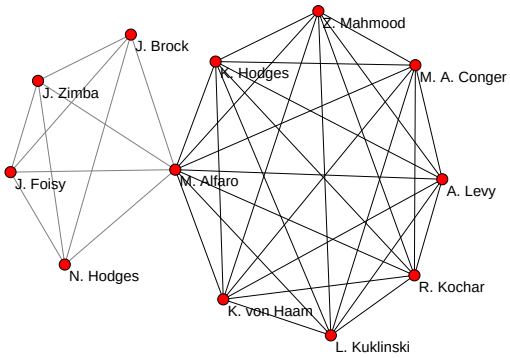
Dva avtorja sta povezana, če sta napisala vsaj en skupen članek. Utež na povezavi določa število skupnih člankov, a teh uteži ne bomo potrebovali. Z uporabo preprostega programa, napisanega v programskem jeziku Python, sem podatke iz oblike BibTEX predelal v obliko, ki jo razume Pajek. Dobljeno omrežje ima 9072 točk (avtorjev) in 22577 povezav (vsaj toliko je skupnih člankov ali knjig), od tega je 13567 povezav z utežjo 1.

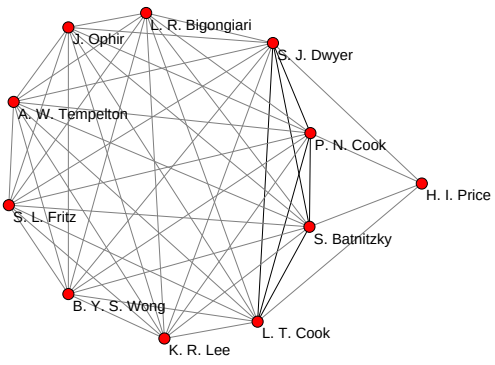
Pri predelavi podatkov iz oblike BibTEX je prišlo do številnih težav zaradi različnih načinov pisanja imen in priimkov avtorjev. Tako dobljeno omrežje vsebuje več točk, ki ustrezajo istemu avtorju. Na primer R.S. Drysdale, Robert L. Drysdale, Robert L. Scot Drysdale, R.L. Drysdale, S. Drysdale, R. Drysdale in R.L.S. Drysdale ali Pankaj K. Agarwal, P. Agarwal, Pankaj Agarwal in P.K. Agarwal. Ti so očitni in

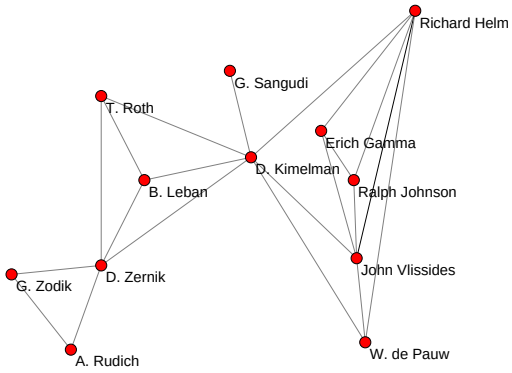
jih hitro razrešimo, ni pa očitno, da sta tudi Otfried Schwarzkopf in Otfried Cheong ista oseba. Veliko dodatnega dela pa bi bilo potrebno, da bi ločili dve osebi z enakim imenom. S tem se nisem ukvarjal. Ročno sem pripravil ekvivalenčne razrede in s Pajkom skrčil omrežje. Dobljeno omrežje vsebuje 7343 točk in 11898 povezav. Je redko omrežje s povprečno stopnjo $2m/n = 3.24$. Dosegljivo je na Pajkovi domači strani med drugimi omrežnimi podatki.

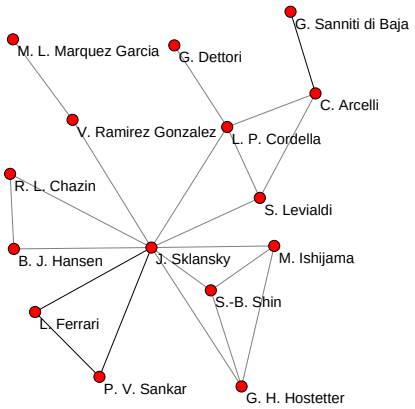
Vsaki točki v tem omrežju izračunamo njeno stopnjo vmesnosti. Veliko stopnjo vmesnosti bodo imele tiste točke, skozi katere poteka veliko število najkrajših poti, torej tisti avtorji, ki velikokrat nastopajo kot posredniki med drugimi avtorji. Če v dobljenem omrežju poiščemo vse točkovne otoke omejene velikosti, bomo dobili skupine avtorjev, ki veliko sodelujejo med sabo, imajo pa tudi veliko povezav z drugimi avtorji. Program nam vrne 15 točkovnih otokov velikosti med 10 in 30, ki vsi skupaj vsebujejo 208 točk. Prikazanih so na slikah, ki sledijo.

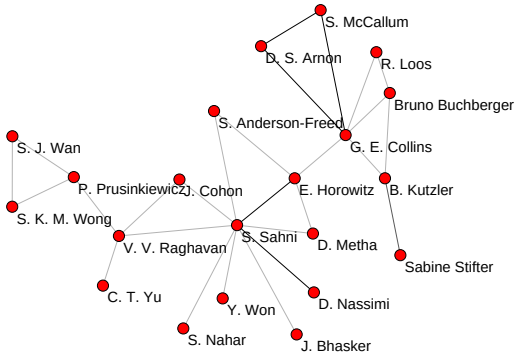


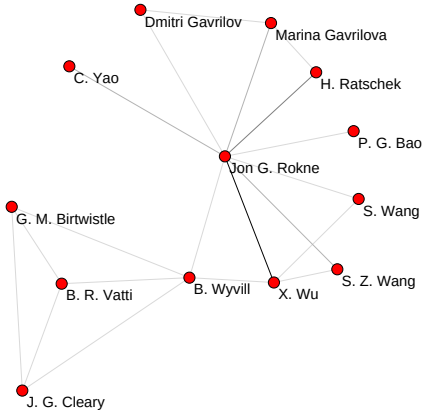


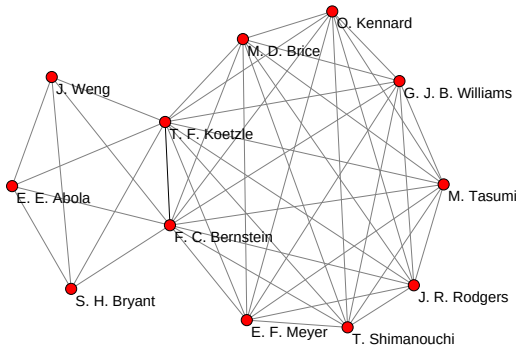


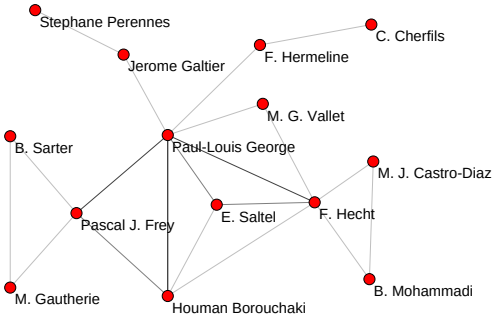


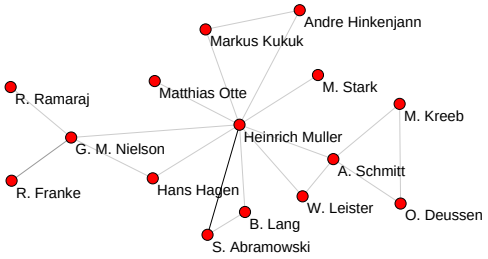

G. Sanniti di Baja
M. L. Marquez Garcia
G. Dettori
C. Arcelli
V. Ramirez Gonzalez
L. P. Cordella
R. L. Chazin
S. Levialdi
B. J. Hansen
J. Sklansky
M. Ishijama
S.-B. Shin
L. Ferrari
P. V. Sankar
G. H. Hostetter

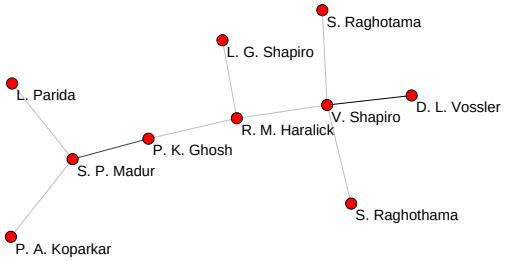

S. McCallum
D. S. Arnon
R. Loos
Bruno Buchberger
S. Anderson-Freed
G. E. Collins
S. J. Wan
P. Prusinkiewicz
J. Cohon
E. Horowitz
B. Kutzler
S. K. M. Wong
S. Sahni
D. Metha
V. V. Raghavan
Sabine Stifter
C. T. Yu
D. Nassimi
Y. Won
S. Nahar
J. Bhasker

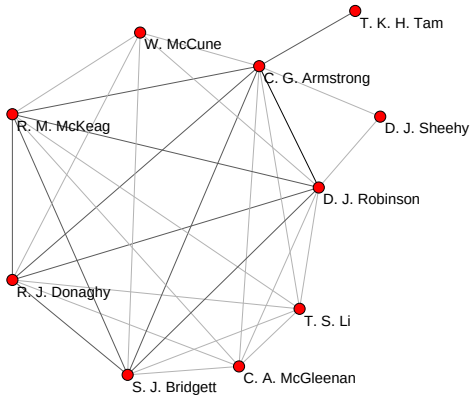

Dmitri Gavrilov
Marina Gavrilova
C. Yao
H. Ratschek
P. G. Bao
Jon G. Rokne
S. Wang
G. M. Birtwistle
S. Z. Wang
B. R. Vatti
B. Wyvill
X. Wu
J. G. Cleary

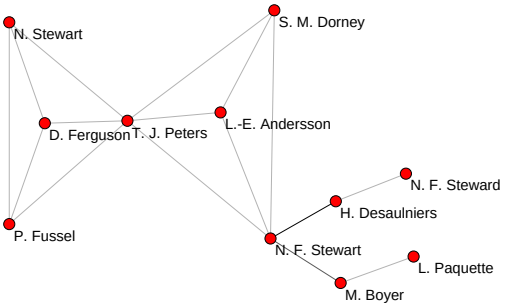

O. Kennard
M. D. Brice
J. Weng
G. J. B. Williams
T. F. Koetzle
E. E. Abola
M. Tasumi
F. C. Bernstein
S. H. Bryant
J. R. Rodgers
E. F. Meyer
T. Shimanouchi

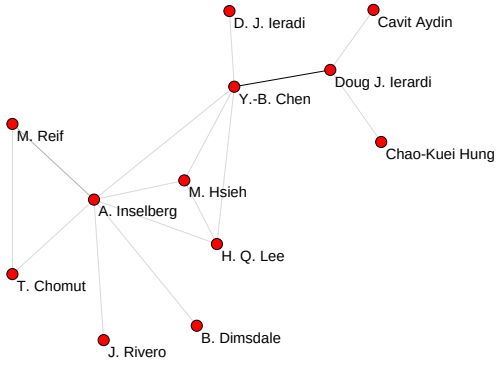

Stephane Perennes
C. Cherfils
Jerome Galtier
F. Hermeline
M. G. Vallet
B. Sarter
Paul-Louis George
M. J. Castro-Diaz
Pascal J. Frey
E. Saltel
F. Hecht
B. Mohammadi
M. Gautherie
Houman Borouchaki

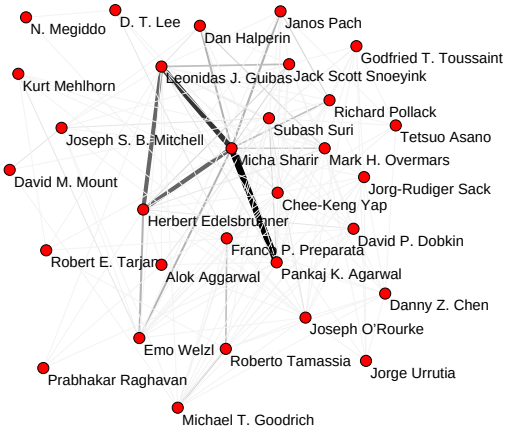

Andre Hinkenjann
Markus Kukuk
R. Ramaraj
Matthias Otte
M. Stark
M. Kreeb
Heinrich Muller
G. M. Nielson
A. Schmitt
R. Franke
Hans Hagen
W. Leister
O. Deussen
B. Lang
S. Abramowski


S. Raghotama
L. G. Shapiro
L. Parida
D. L. Vossler
V. Shapiro
R. M. Haralick
P. K. Ghosh
S. P. Madur
S. Raghothama
P. A. Koparkar


T. K. H. Tam
W. McCune
C. G. Armstrong
R. M. McKeag
D. J. Sheehy
D. J. Robinson
R. J. Donaghy
T. S. Li
C. A. McGleenan
S. J. Bridgett


S. M. Dorney
N. Stewart
D. Ferguson
T. J. Peters
L.-E. Andersson
N. F. Steward
H. Desaulniers
P. Fussel
N. F. Stewart
L. Paquette
M. Boyer


D. J. Ieradi
Cavit Aydin
Doug J. Ierardi
Y.-B. Chen
M. Reif
Chao-Kuei Hung
M. Hsieh
A. Inselberg
H. Q. Lee
T. Chomut
B. Dimsdale
J. Rivero


N. Megiddo
D. T. Lee
Janos Pach
Dan Halperin
Godfried T. Toussaint
Leonidas J. Guibas
Jack Scott Snoeyink
Kurt Mehlhorn
Richard Pollack
Subash Suri
Tetsuo Asano
Joseph S. B. Mitchell
Micha Sharir
Mark H. Overmars
David M. Mount
Jorg-Rudiger Sack
Chee-Keng Yap
Herbert Edelsbrunner
David P. Dobkin
Franco P. Preparata
Robert E. Tarjan
Alok Aggarwal
Pankaj K. Agarwal
Danny Z. Chen
Joseph O'Rourke
Emo Welzl
Roberto Tamassia
Jorge Urrutia
Prabhakar Raghavan
Michael T. Goodrich

3.2 Povezavni otoki

Tudi za omrežja, ki imajo uteži na povezavah, lahko definiramo otoke. Da si bomo lažje predstavljali delovanje postopka, bomo uteži dane povezave rekli kar *višina* povezave.

Podobno, kot pri točkovnem primeru, lahko povezavne otoke iščemo s pomočjo povezavnih prerezov. Povezavni prerez danega omrežja na nivoju t dobimo tako, da iz omrežja odstranimo vse povezave, katerih višina je manjša od t, ter vse točke, ki po odstranitvi teh povezav postanejo izolirane.

Definicija 3.6: *Povezavni prerez* omrežja $\mathcal{N} = (\mathcal{V}, \mathcal{L}, w)$ na nivoju t je podomrežje $\mathcal{N}(t) = (\mathcal{V}', \mathcal{L}', w)$, določeno z

$$\begin{aligned} \mathcal{L}' &= \{e \in \mathcal{L} : w(e) \geq t\} \\ \mathcal{V}' &= \{v \in \mathcal{V} : \exists u \in \mathcal{V} : (u\,;v) \in \mathcal{L}'\} \end{aligned}$$

Definicija 3.7: Neprazna skupina točk $\mathcal{C} \subseteq \mathcal{V}$ je *povezavni otok*, če sta v skupini vsaj dve točki, če točke skupine določajo povezan podgraf in če obstaja vpeto drevo $\mathcal{T}$ na teh točkah, tako da so višine povezav, ki povezujejo točke iz skupine s točkami izven skupine manjše ali enake višinam povezav v drevesu $\mathcal{T}$.

$$\max_{\substack{(u\,;v)\in\mathcal{L}:\\ u\in\mathcal{C}\wedge v\notin\mathcal{C}}} w((u\,;v)) \leq \min_{e\in\mathcal{L}(\mathcal{T})} w(e)$$

Definicija 3.8: Povezavni otok $\mathcal{C} \subseteq \mathcal{V}$ je *pravi povezavni otok*, če obstaja vpeto drevo $\mathcal{T}$, tako da so višine povezav, ki povezujejo točke iz skupine s točkami izven skupine strogo manjše od višin povezav v drevesu $\mathcal{T}$.

$$\max_{\substack{(u\,;v)\in\mathcal{L}:\\ u\in\mathcal{C}\wedge v\notin\mathcal{C}}} w((u\,;v)) < \min_{e\in\mathcal{L}(\mathcal{T})} w(e)$$

Skupine točk, ki ustrezajo povezanim komponentam v povezavnem prerezu danega omrežja, izpolnjujejo vse zahtevane pogoje za prave povezavne otoke. Ker so to skupine točk, ki so po brisanju povezav z višino manjšo od t ostale povezane, obstaja vpeto drevo, ki vsebuje samo povezave na višinah, ki so večje ali enake t. Po drugi strani so višine povezav med skupinami vse manjše od t, sicer bi krajišči take povezave bili obe v isti skupini. Te skupine točk so torej pravi povezavni otoki.

Podobno, kot pri točkovnem primeru, je tudi tu od izbire nivoja t odvisno, na koliko povezanih komponent razpade omrežje ter kako velike so te komponente. Najmanjša možna komponenta je sestavljena iz dveh točk (ker smo pobrisali vse izolirane točke), navzgor pa velikost ni omejena z drugim, kot z velikostjo celega omrežja. Treba bo torej zopet najti postopek, ki bo poiskal prave povezavne otoke omejene velikosti.

3.2.1 Splošni povezavni otoki

Pri določanju povezavnih otokov bomo poskusili uporabiti isto idejo, ki nas je pripeljala do točkovnih otokov. Podobno kot v točkovnem primeru, bomo tudi tu omrežje najprej potopili v vodo, nato pa gladino vode postopoma zniževali in opazovali, kakšni otoki se pojavljajo. Seveda nam gladine vode ni treba zniževati zvezno, pač pa po korakih po višinah povezav od najvišje do najnižje. Tudi tu bomo predpostavili, da povezave, ki so na isti višini, ne pridejo iz vode vse naenkrat, pač pa postopoma ena za drugo.

Postopek 3.3 za določanje maksimalnih pravih povezavnih otokov omejene velikosti je zelo podoben postopku 3.1 za določanje maksimalnih pravih točkovnih otokov omejene velikosti.

Da bo postopek bolj preprost, bomo malo popustili pri definiciji povezavnega otoka in dovolili tudi povezavne otoke, ki vsebujejo eno samo točko. Takemu otoku bomo rekli *trivialen povezavni otok*. Točke najprej razporedimo v n trivialnih povezavnih otokov, potem pa po vrsti pregledamo vsako povezavo od najvišje do najnižje. Če krajišči povezave pripadata istemu otoku, ne storimo ničesar, če pa pripadata dvema različnima otokoma, ta dva otoka združimo (v bistvu počnemo isto, kot če bi iskali maksimalno vpeto drevo s Kruskalovim postopkom).

Tu opazimo tudi edino večjo razliko med točkovnim in povezavnim postopkom. V točkovnem primeru smo tekočo točko vedno dodali neki množici otokov (ki je lahko bila tudi prazna), v povezavnem primeru pa tekočo povezavo ignoriramo ali pa združimo natanko dva otoka (postaneta *podotoka* večjega otoka). Razlika je tudi, kako definiramo *pristanišče*. V točkovnem otoku je to bila najnižja točka na otoku, v povezavnem primeru pa je to povezava, s katero smo združili dva otoka (to ni nujno najnižja povezava med točkami na otoku). Otok, ki vsebuje eno samo točko, nima pristanišča.

Za vsak otok si moramo tudi zabeležiti, ali je pravi. To lahko naredimo za oba podotoka šele takrat, ko ju združimo. Otok je pravi, če nima pristanišča ali pa če je njegovo pristanišče višje kot pristanišče večjega otoka.

Ko zmanjka vode, nam ostane toliko otokov, kolikor povezanih komponent ima omrežje. A nas zanimajo samo pravi otoki omejene velikosti. Te poiščemo podobno, kot v točkovnem primeru.

3.2.2 Enostavni povezavni otoki

Definicija 3.9: Skupina točk $\mathcal{C} \subseteq \mathcal{V}$ je *lokalni povezavni vrh*, če je pravi povezavni otok in obstaja vpeto drevo, v katerem imajo vse povezave enako višino kot najvišja povezava med poljubnima dvema točkama.

Definicija 3.10: Povezavni otok, ki ima en sam lokalni povezavni vrh, imenujemo *enostaven povezavni otok*.

Postopek 3.3 *Maksimalni pravi povezavni otoki omejene velikosti*

PODATKI: omrežje $\mathcal{N} = (\mathcal{V}, \mathcal{L}, w)$, števili min in max
REZULTAT: množica L vseh maksimalnih pravih povezavnih otokov velikosti vsaj min in največ max

$otoki := \{\{v\} : v \in \mathcal{V}\}$
for each $o \in otoki$ **do** $o.pristan :=$ **null**
uredi $\mathcal{L}$ nenaraščajoče glede na w
for each $e(u\,;v) \in \mathcal{L}$ (v dobljenem vrstnem redu) **do begin**
  $o1 := otok \in otoki : u \in otok$
  $o2 := otok \in otoki : v \in otok$
  if $o1 \neq o2$ **then begin**
    $otok :=$ **new** $Otok()$
    $otok.pristan := e$
    $otok.podotok1 := o1$
    $otok.podotok2 := o2$
    $otoki := otoki \cup \{otok\} \setminus \{o1, o2\}$
    $o1.pravi := o1.pristan =$ **null** $\vee\, w(o1.pristan) > w(e)$
    $o2.pravi := o2.pristan =$ **null** $\vee\, w(o2.pristan) > w(e)$
  end
end
for each $o \in otoki$ **do** $o.pravi :=$ **true**

$L := \emptyset$
while $otoki \neq \emptyset$ **do begin**
  izberi $otok \in otoki$
  $otoki := otoki \setminus \{otok\}$
  if $|otok| < min$ **then delete** $otok$
  else if $|otok| > max \vee \neg otok.pravi$ **then begin**
    $otoki := otoki \cup \{otok.podotok1, otok.podotok2\}$
    delete $otok$
  end
  else $L := L \cup \{otok\}$
end

Da bomo izmed vseh pravih povezavnih otokov izločili samo enostavne, moramo otoku predpisati vrsto. Podobno, kot v točkovnem primeru, je vrsta otoka lahko FLAT (obstaja vpeto drevo, v katerem imajo vse povezave enako višino kot najvišja povezava med poljubnima dvema točkama), SINGLE (otok ima en sam lokalni povezavni vrh) ali MULTI (otok ima več lokalnih povezavnih vrhov). Poglejmo si podrobneje, kako določimo vrsto novega otoka.

- Trivialen povezavni otok je vrste FLAT.

- Netrivialen povezavni otok ima natanko dva podotoka. Vrsta otoka je v tem primeru odvisna od vrste teh dveh podotokov in višin njunih pristanišč. Če je kateri od teh podotokov trivialen, nima pristanišča, zato se obnašamo, kot da ima pristanišče na isti višini, kot je višina trenutne povezave.
  - Če sta oba podotoka vrste FLAT in sta višini obeh pristanišč enaki višini trenutne povezave, je tudi nov otok vrste FLAT.
  - Če je samo en podotok vrste FLAT in je višina njegovega pristanišča enaka višini trenutne povezave, je nov otok vrste SINGLE.
  - V vseh ostalih primerih dobimo otok vrste MULTI.

Pogoj v drugem delu postopka, ki odloči, ali je otok potrebno razbiti na podotoke in pregledati vsak podotok posebej, dopolnimo tako, da dodatno preverimo še, ali je otok vrste MULTI (tudi v tem primeru ga je potrebno razbiti). Postopek 3.4 prikazuje tako dopolnjen postopek 3.3, kjer je stara koda napisana v svetlejši barvi, da so dopolnitve bolj vidne.

3.2.3 Izvedba

Podobno, kot pri točkovnem primeru predpostavimo, da je omrežje predstavljeno s strukturo `Graph`, točka v omrežju pa s strukturo `Vert`. Dodatno moramo še opisati, kako je predstavljena povezava med dvema točka. Povezavo opišemo s strukturo `Edge`, ki bo vsebovala kazalca na začetno in končno točko povezave (lastnosti `source` in `target`) ter višino povezave (lastnost `value`).

Najprej se moramo odločiti, kakšno podatkovno strukturo bomo uporabili za predstavitev povezavnega otoka. Ker je povezavni otok ena sama točka, ali pa je sestavljen iz natanko dveh podotokov, bomo za njegovo predstavitev uporabili dvojiško drevo, ki bo imelo dve vrsti vozlišč. Vozlišče opišemo s strukturo `EdgeIsland`. V listih bo drevo imelo zapisane vse točke (lastnost `vert`), ki so na otoku, notranja vozlišča pa bodo hranila povezave (lastnost `port`), s katerimi smo združevali dva otoka v večji otok. Povezava v korenu drevesa bo torej pristanišče ustreznega otoka, poddrevesi (lastnosti `left` in `right`) pa bosta predstavljali njegova podotoka.

Vsako vozlišče v drevesu bo moralo vsebovati podatek o tem, kdo je njegov prednik (lastnost `parent`). Le tako bomo za dano točko lahko ugotovili, kateremu drevesu pripada (poiskati bomo morali koren drevesa). Da pa se ne bomo pri vsakem iskanju korena pomikali vedno po isti poti, bomo po vsakem prehodu poti vsem vozliščem na poti podatek o predniku popravili, tako da bo kazal neposredno na koren (podobno kot pri postopku Union-Find). Poleg tega bomo v vsakem vozlišču imeli še podatek o velikosti otoka (lastnost `size`) ter podatek, ki pove, ali je otok pravi (lastnost `regular`).

Za predstavitev vseh otokov bomo potrebovali n vozlišč za točke (listi) in največ $n-1$ vozlišč za povezave, skupno torej največ $2n-1$ vozlišč. Namesto da bi pomnilnik

Postopek 3.4 *Maksimalni enostavni pravi povezavni otoki omejene velikosti*

Podatki: omrežje $\mathcal{N} = (\mathcal{V}, \mathcal{L}, w)$, števili min in max
Rezultat: množica L vseh maksimalnih enostavnih pravih povezavnih otokov velikosti vsaj min in največ max

```
otoki := {{v} : v ∈ V}
for each o ∈ otoki do o.pristan := null
for each o ∈ otoki do o.vrsta := FLAT
uredi L nenaraščajoče glede na w
for each e(u ; v) ∈ L (v dobljenem vrstnem redu) do begin
    o1 := otok ∈ otoki : u ∈ otok
    o2 := otok ∈ otoki : v ∈ otok
    if o1 ≠ o2 then begin
        otok := new Otok()
        otok.pristan := e
        otok.podotok1 := o1
        otok.podotok2 := o2
        otoki := otoki ∪ {otok} \ {o1, o2}
        o1.pravi := o1.pristan = null ∨ w(o1.pristan) > w(e)
        o2.pravi := o2.pristan = null ∨ w(o2.pristan) > w(e)
        p1 := o1.vrsta = FLAT ∧ (o1.pristan = null ∨ w(o1.pristan) = w(e))
        p2 := o2.vrsta = FLAT ∧ (o2.pristan = null ∨ w(o2.pristan) = w(e))
        if p1 ∧ p2 then otok.vrsta := FLAT
        else if p1 ∨ p2 then otok.vrsta := SINGLE
        else otok.vrsta := MULTI
    end
end
for each o ∈ otoki do o.pravi := true

L := ∅
while otoki ≠ ∅ do begin
    izberi otok ∈ otoki
    otoki := otoki \ {otok}
    if |otok| < min then delete otok
    else if |otok| > max ∨ ¬otok.pravi ∨ otok.vrsta = MULTI then begin
        otoki := otoki ∪ {otok.podotok1, otok.podotok2}
        delete otok
    end
    else L := L ∪ {otok}
end
```

rezervirali za vsako vozlišče posebej, lahko ves potreben pomnilnik zanje rezerviramo kar v enem kosu (tabela vozlišč). Vozlišča v tabeli `islands` bomo zasedali po vrsti

```
struct Edge
{
   Vert *source;          // kazalec na začetno točko
   Vert *target;          // kazalec na končno točko
   double value;          // vrednost
};

struct EdgeIsland
{
   int size;              // velikost otoka
   Vert *vert;            // točka v listu
   Edge *port;            // pristanišče
   EdgeIsland *left;      // kazalec na levo poddrevo
   EdgeIsland *right;     // kazalec na desno poddrevo
   EdgeIsland *parent;    // kazalec na prednika
   bool regular;          // ali je otok pravi
};
```

od začetka proti koncu tabele, pri čemer prvih n vozlišč za liste zasedemo takoj na začetku postopka.

V pripravljalnem delu postopka najprej uredimo množico vseh povezav nenaraščajoče, nato pa pripravimo prvih n vozlišč v tabeli `islands`. V glavni zanki nato po vrsti pregledamo vse povezave. Za trenutno povezavo `e` poiščemo vozlišči `v1` in `v2`, v katerih sta zapisani njeni krajišči. Nato poiščemo korena `p1` in `p2` dreves, v katerih sta ti dve vozlišči vsebovani. Če sta vsebovani v dveh različnih drevesih, moramo ti dve drevesi združiti. To naredimo tako, da pripravimo novo vozlišče v tabeli, in ti dve drevesi postavimo za poddrevesi. Sproti še seštejemo velikosti poddreves, da dobimo velikost novega drevesa (število listov), nato pa se po poti od krajišč do korena sprehodimo še enkrat in vse kazalce na prednike preusmerimo neposredno na koren novega drevesa.

Zaključni del postopka je zelo podoben tistemu pri točkovnem primeru. Edina razlika je v tem, kako točke na otoku postavimo vse v isto skupino. V točkovnem primeru smo točko v korenu postavili v novo skupino, vse druge točke v poddrevesih pa v isto skupino, kot njihovega prednika. V našem primeru pa imamo točke samo v listih drevesa, drugje pa povezave, ki jih ne moremo razporejati po skupinah. Zato pri prvi povezavi naredimo novo skupino in vanjo postavimo krajišči povezave, krajišča drugih povezav v poddrevesih pa v isto skupino, kot je eno od krajišč prednika v drevesu. Nekatere od točk tako večkrat postavimo v isto skupino (namesto k različnih točk postavimo v skupino $2k - 2$ točk, med katerimi je samo k različnih). Pomožna funkcija `markEdge` krajišči povezave, ki ustreza pristanišču otoka, ki ga predstavlja vozlišče `p`, postavi v skupino `c`, korenoma obeh poddreves pa nastavi prednika na `q`.

Namesto lastnosti `vert` in `port` bi bilo dovolj imeti samo eno, saj lastnost `vert` potrebujemo samo v listih (trivialni otoki), lastnost `port` pa samo v notranjih vozliščih, ki predstavljajo poveyave. Ker pa sta to kazalca različne vrste, smo zaradi preglednosti raje ohranili obe lastnosti.

Program 3.2: *Maksimalni pravi povezavni otoki omejene velikosti*

```
int *edgeIslands(Graph *g, int min, int max)
{
   Vert *vert = vertices(g);
   Edge *edge = edges(g);
   int n = size(vert);
   int m = size(edge);
   qsort(edge, m, sizeof(*edge), edgeDescending);

   EdgeIsland *islands = new EdgeIsland[2 * n - 1];
   for (int i = 0; i < n; ++i) {
      islands[i].size = 1;
      islands[i].vert = vert[i];
      islands[i].port = NULL;
      islands[i].left = islands[i].right = islands[i].parent = NULL;
      islands[i].regular = true;
   }
   int nn = n;
   for (int i = 0; i < m; ++i) {
      Edge *e = edge[i];
      EdgeIsland *p1, *v1 = &islands[e->source->id];
      EdgeIsland *p2, *v2 = &islands[e->target->id];
      for (p1 = v1; p1->parent != NULL; p1 = p1->parent);
      for (p2 = v2; p2->parent != NULL; p2 = p2->parent);
      if (p1 != p2) {
         EdgeIsland *p = &islands[nn++];
         p->size = p1->size + p2->size;
         p->vert = NULL;
         p->port = e;
         p->left = p1;
         p->right = p2;
         p->parent = NULL;
         p->regular = true;
         p1->regular = p1->port == NULL || p1->port->value > e->value;
         p2->regular = p2->port == NULL || p2->port->value > e->value;
         for (p1 = v1; p1 != NULL;)
            { EdgeIsland *pp = p1;  p1 = p1->parent;  pp->parent = p; }
         for (p2 = v2; p2 != NULL;)
            { EdgeIsland *pp = p2;  p2 = p2->parent;  pp->parent = p; }
      }
   }
   int c = 0;
   int *part = new int[n];
   for (int i = 0; i < n; ++i) part[i] = 0;
   while (nn > n) {
      EdgeIsland *p = &islands[--nn];
      if (p->parent != NULL)
         markEdge(part, p, part[p->parent->port->source->id], p);
      else if (p->size < min)
         markEdge(part, p, 0, p);
      else if (p->size > max || !p->regular)
         markEdge(part, p, 0, NULL);
      else
         markEdge(part, p, ++c, p);
   }
   delete [] islands;
   return part;
}
```

```
void markEdge(int *part, EdgeIsland *p, int c, EdgeIsland *q)
{
   part[p->port->source->id] = part[p->port->target->id] = c;
   p->left->parent = p->right->parent = q;
}
```

Opisano izvedbo postopka 3.3 bi lahko preprosto dopolnili, tako da bi dobili vse maksimalne enostavne prave povezavne otoke omejene velikosti. Dovolj je slediti navodilom v postopku 3.4.

3.2.4 Časovna zahtevnost

Za urejanje množice povezav potrebujemo $\mathcal{O}(m \log m)$ časa, potem pa še $\mathcal{O}(n)$ časa za pripravo tabele vozlišč.

Pri določanju vseh otokov moramo pregledati vse povezave, torej se glavna zanka izvrši m-krat, v zanki pa moramo poiskati poti od točk, ki predstavljata krajišči povezave, do korenov dreves, ki ti dve točki vsebujeta. Ker poti vsakič skrajšamo podobno kot pri postopku Union-Find, za gradnjo vpetega gozda potrebujemo $\mathcal{O}(m\,\alpha(n))$ časa, kjer je α obrat Ackermannove funkcije.

Za zaključni del postopka, kjer določimo iskane otoke, potrebujemo $\mathcal{O}(n)$ časa, saj moramo pregledati največ $n-1$ vozlišč, vsakič pa pokličemo pomožno funkcijo, ki ima konstantno časovno zahtevnost.

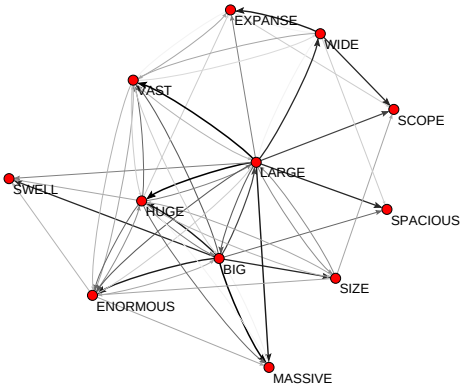
Ker je m omejen s kvadratom n-ja, lahko namesto $\log m$ pišemo kar $\log n$. Ker $\alpha(n)$ narašča precej počasneje, kot $\log n$, pa je skupna časovna zahtevnost postopka enaka $\mathcal{O}(m \log n)$.

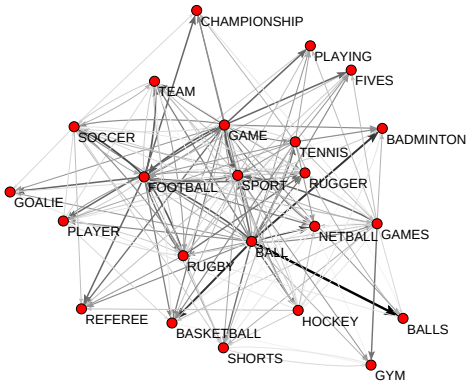
3.2.5 Primer

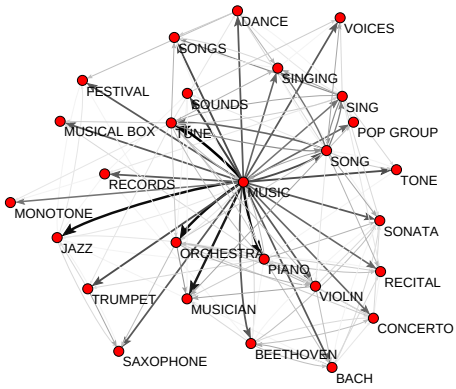
Na britanskih univerzah so od junija 1968 do maja 1971 anketirali tamkajšnje študente. Za vsako od besed iz danega seznama so morali napisati besedo, ki jim pride prva na misel. Tako so dobili veliko omrežje, sestavljeno iz 23219 točk (besede) in 325624 usmerjenih povezav, med katerimi je tudi 564 zank.

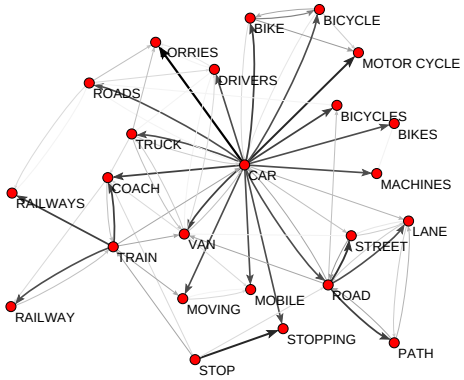
Zanima nas, katere so tiste skupine besed, ki so med seboj najbolj povezane. Uteži na povezavah določimo tako, da za vsako povezavo preštejemo, na koliko tranzitivnih trikotnikih leži. Tranzitivne trikotnike izberemo zato, ker nas poleg neposrednih povezav med besedami zanimajo tudi posredne, preko neke vmesne točke.

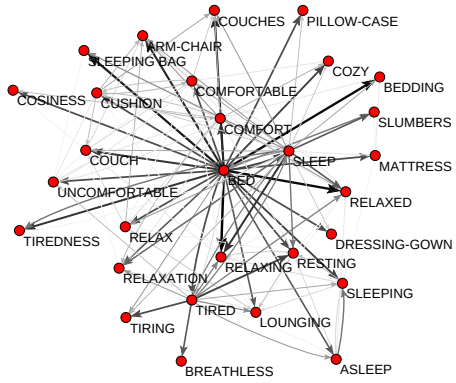
Ko na dobljenem omrežju poiščemo vse povezavne otoke velikosti od 5 do 30, dobimo 53 otokov, ki vsi skupaj vsebujejo 664 točk. Nekateri od bolj zanimivih otokov so prikazani na slikah, ki sledijo.

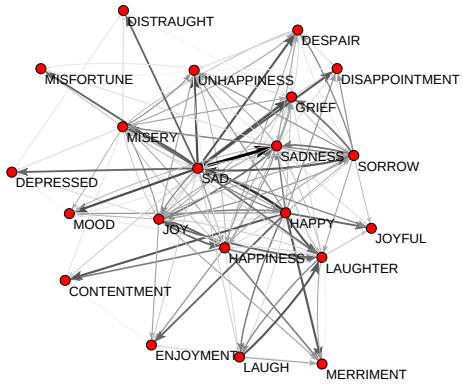

EXPANSE
WIDE
VAST
SCOPE
LARGE
SWELL
HUGE
SPACIOUS
BIG
SIZE
ENORMOUS
MASSIVE

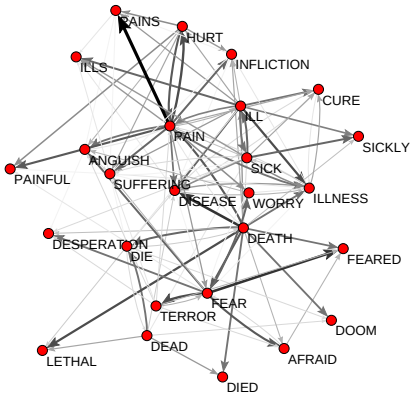

CHAMPIONSHIP
PLAYING
FIVES
TEAM
SOCCER
GAME
BADMINTON
TENNIS
FOOTBALL
SPORT
RUGGER
GOALIE
PLAYER
NETBALL
GAMES
BALL
RUGBY
REFEREE
BASKETBALL
HOCKEY
BALLS
SHORTS
GYM

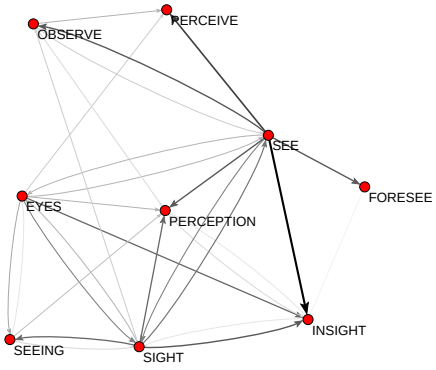

DANCE
VOICES
SONGS
SINGING
FESTIVAL
SOUNDS
SING
MUSICAL BOX
TUNE
POP GROUP
SONG
RECORDS
TONE
MUSIC
MONOTONE
SONATA
JAZZ
ORCHESTRA
PIANO
RECITAL
TRUMPET
MUSICIAN
VIOLIN
CONCERTO
SAXOPHONE
BEETHOVEN
BACH

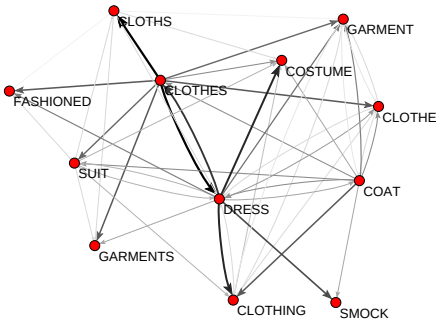

BICYCLE
BIKE
WORRIES
MOTOR CYCLE
DRIVERS
ROADS
BICYCLES
BIKES
TRUCK
CAR
MACHINES
COACH
RAILWAYS
LANE
VAN
STREET
TRAIN
ROAD
MOBILE
MOVING
RAILWAY
STOPPING
PATH
STOP

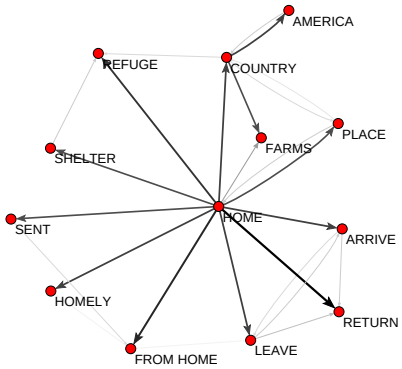

COUCHES
PILLOW-CASE
ARM-CHAIR
SLEEPING BAG
COZY
COMFORTABLE
BEDDING
COSINESS
CUSHION
SLUMBERS
COMFORT
SLEEP
COUCH
MATTRESS
BED
UNCOMFORTABLE
RELAXED
TIREDNESS
RELAX
DRESSING-GOWN
RESTING
RELAXING
RELAXATION
SLEEPING
TIRED
TIRING
LOUNGING
BREATHLESS
ASLEEP

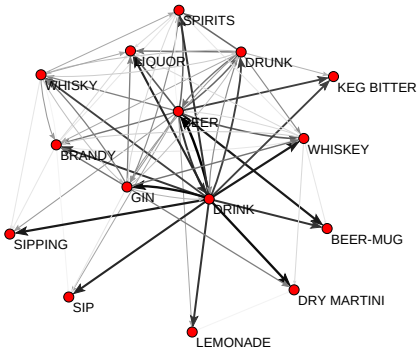

DISTRAUGHT
DESPAIR
MISFORTUNE
UNHAPPINESS
DISAPPOINTMENT
GRIEF
MISERY
SADNESS
SORROW
DEPRESSED
SAD
MOOD
JOY
HAPPY
JOYFUL
HAPPINESS
LAUGHTER
CONTENTMENT
ENJOYMENT
LAUGH
MERRIMENT

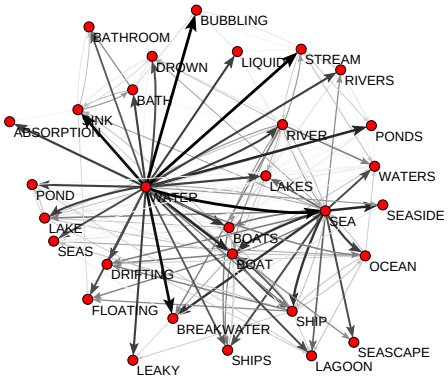

PAINS
HURT
ILLS
INFLICTION
CURE
ILL
PAIN
SICKLY
ANGUISH
SICK
PAINFUL
SUFFERING
ILLNESS
DISEASE
WORRY
DEATH
DESPERATION
DIE
FEARED
FEAR
TERROR
DOOM
DEAD
AFRAID
LETHAL
DIED

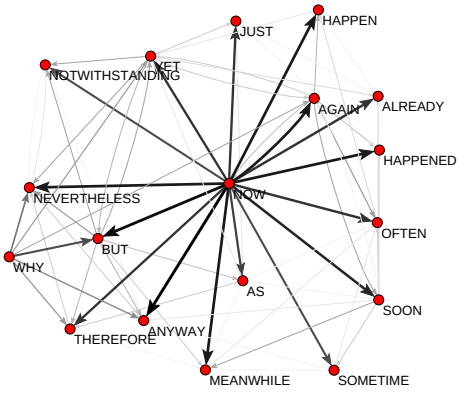

PERCEIVE
OBSERVE
SEE
FORESEE
EYES
PERCEPTION
INSIGHT
SEEING
SIGHT

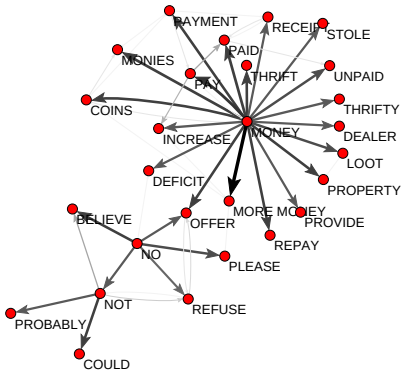

CLOTHS
GARMENT
COSTUME
CLOTHES
FASHIONED
CLOTHE
SUIT
COAT
DRESS
GARMENTS
CLOTHING
SMOCK

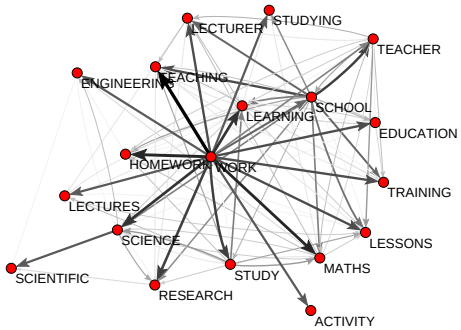

AMERICA
REFUGE
COUNTRY
PLACE
FARMS
SHELTER
HOME
SENT
ARRIVE
HOMELY
RETURN
FROM HOME
LEAVE

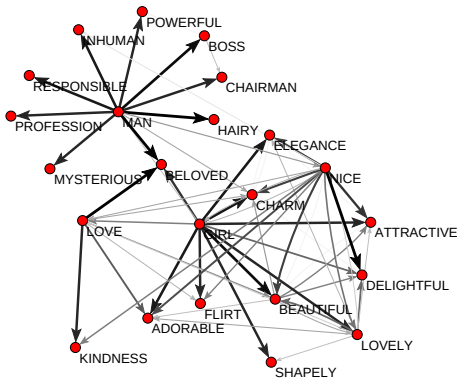

SPIRITS
LIQUOR
DRUNK
WHISKY
KEG BITTER
BEER
BRANDY
WHISKEY
GIN
DRINK
BEER-MUG
SIPPING
SIP
DRY MARTINI
LEMONADE


BUBBLING
BATHROOM
DROWN
LIQUID
STREAM
RIVERS
BATH
SINK
ABSORPTION
RIVER
PONDS
WATERS
LAKES
POND
WATER
SEASIDE
SEA
LAKE
BOATS
SEAS
BOAT
OCEAN
DRIFTING
FLOATING
SHIP
BREAKWATER
SEASCAPE
SHIPS
LAGOON
LEAKY


HAPPEN
JUST
YET
NOTWITHSTANDING
AGAIN
ALREADY
HAPPENED
NEVERTHELESS
NOW
OFTEN
BUT
WHY
AS
SOON
ANYWAY
THEREFORE
MEANWHILE
SOMETIME


PAYMENT
RECEIP
STOLE
MONIES
PAID
PAY
THRIFT
UNPAID
COINS
THRIFTY
INCREASE
MONEY
DEALER
LOOT
DEFICIT
PROPERTY
MORE MONEY
BELIEVE
OFFER
PROVIDE
REPAY
NO
PLEASE
NOT
REFUSE
PROBABLY
COULD


STUDYING
LECTURER
TEACHER
ENGINEERING
TEACHING
SCHOOL
LEARNING
EDUCATION
HOMEWORK
WORK
TRAINING
LECTURES
SCIENCE
LESSONS
SCIENTIFIC
STUDY
MATHS
RESEARCH
ACTIVITY


POWERFUL
INHUMAN
BOSS
RESPONSIBLE
CHAIRMAN
PROFESSION
MAN
HAIRY
ELEGANCE
MYSTERIOUS
BELOVED
NICE
CHARM
LOVE
GIRL
ATTRACTIVE
DELIGHTFUL
FLIRT
BEAUTIFUL
ADORABLE
LOVELY
KINDNESS
SHAPELY

Na eni od slik se lepo vidi, da so bili anketirani študenti. V otoku, ki vsebuje besedo work, so namreč večinoma besede, ki so značilne za študentsko delo (activity, education, engineering, homework, learning, lecturer, lectures, lessons, maths, research, school, science, scientific, study, studying, teacher, teaching, training).

3.3 Lastnosti otokov

Hitro opazimo, da so točkovni in povezavni otoki neodvisni od samih vrednosti točk ali povezav, pomembna je samo njihova medsebojna urejenost glede na višino. Torej lahko vrednosti točk in povezav preslikamo s katerokoli strogo naraščajočo realno funkcijo, ne da bi pri kakorkoli vplivali na otoke.

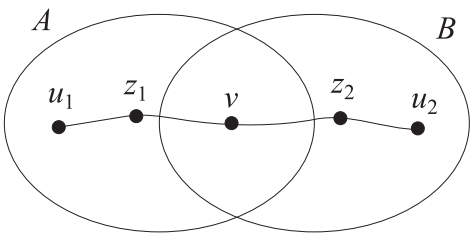
Trditev 3.1: *Sosednji točki ne moreta pripadati dvema ločenima pravima točkovnima otokoma.*

Dokaz: Naj bosta u in v dve sosednji točki. Točka u naj pripada otoku A, točka v pa otoku B. Po definiciji bi za višine teh dveh točk moralo veljati: $p(u) < p(v)$ in $p(v) < p(u)$, kar pa je nemogoče. □

Trditev 3.2: *Naj bo $\mathcal{H}_p(\mathcal{N})$ množica vseh pravih točkovnih otokov omrežja $\mathcal{N}$. Potem je $\mathcal{H}_p$ polna hierarhija nad množico $\mathcal{V}$.*

Dokaz: Pravi točkovni otok je po definiciji neprazna podmnožica množice točk $\mathcal{V}$, torej je $\mathcal{H}_p$ razvrstitev množice $\mathcal{V}$.

Naj bosta A in B poljubna otoka iz $\mathcal{H}_p$. Recimo, da je njun presek neprazen, različen od A in različen od B. To pomeni, da je $A \cap B \neq \emptyset$, $A \not\subseteq B$ in $B \not\subseteq A$, torej obstajajo točke $v \in A \cap B$, $u_1 \in A \setminus B$ in $u_2 \in B \setminus A$. Ker točki v in u_1 obe pripadata otoku A, vsak otok pa določa povezan podgraf, obstaja pot od v do u_1 po točkah iz A. Ker točka v pripada tudi otoku B, moramo nekje na tej poti prestopiti iz množice B v množico A, torej obstaja vsaj ena točka $z_1 \in A \cap N(B)$. Na podoben način se prepričamo, da obstaja tudi pot od v do u_2 po točkah iz B, na njej pa obstaja točka $z_2 \in B \cap N(A)$. Ker je $z_1 \in A$ in $z_2 \in N(A)$, mora biti $p(z_1) > p(z_2)$, ker pa je $z_2 \in B$ in $z_1 \in N(B)$, mora biti tudi $p(z_2) > p(z_1)$. To pa je protislovje, torej mora biti $A \cap B \in \{\emptyset, A, B\}$, kar pomeni, da je $\mathcal{H}_p$ hierarhija.



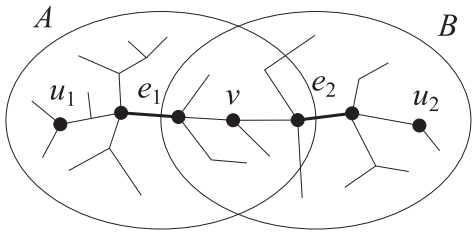
Slika 3.2: *$\mathcal{H}_p$ je polna hierarhija*

Množice točk, ki ustrezajo povezanim komponentam grafa, so tudi pravi točkovni otoki, torej so v $\mathcal{H}_p$. To pomeni, da $\bigcup \mathcal{H}_p = \mathcal{V}$, torej je $\mathcal{H}_p$ polna hierarhija nad množico $\mathcal{V}$. □

Trditev 3.3: *Naj bo $\mathcal{H}_w(\mathcal{N})$ množica vseh pravih povezavnih otokov omrežja $\mathcal{N}$. Potem je $\mathcal{H}_w$ hierarhija nad množico $\mathcal{V}$.*

Dokaz: Pravi povezavni otok je po definiciji neprazna podmnožica množice točk $\mathcal{V}$, torej je $\mathcal{H}_w$ razvrstitev množice $\mathcal{V}$.

Naj bosta A in B poljubna otoka iz $\mathcal{H}_w$. Recimo, da je njun presek neprazen, različen od A in različen od B. To pomeni, da je $A \cap B \neq \emptyset$, $A \not\subseteq B$ in $B \not\subseteq A$, torej obstajajo točke $v \in A \cap B$, $u_1 \in A \setminus B$ in $u_2 \in B \setminus A$. Ker je A povezavni otok, obstaja vpeto drevo, katerega povezave imajo vse večjo utež kot povezave, ki povezujejo točke na otoku s točkami izven otoka. Ker pa imamo v otoku A točko $v \in B$ in točko $u_1 \in A \setminus B$ mora v tem vpetem drevesu obstajati povezava e_1 z enim krajiščem v B in drugim krajiščem v $A \setminus B$. Na podoben način se prepričamo, da v otoku B obstaja vpeto drevo, ki vsebuje povezavo e_2 z enim krajiščem v A in drugim krajiščem v $B \setminus A$. Od tod dobimo, da mora biti $w(e_1) > w(e_2)$ in $w(e_2) > w(e_1)$. To pa je protislovje, torej mora biti $A \cap B \in \{\emptyset, A, B\}$, kar pomeni, da je $\mathcal{H}_w$ hierarhija.



Slika 3.3: *$\mathcal{H}_w$ je hierarhija*

$\mathcal{H}_w$ pa ni nujno polna hierarhija. Če imamo v grafu kakšno izolirano točko, potem ta ne pripada nobenemu povezavnemu otoku. Če pa v grafu ni izoliranih točk, potem so množice točk, ki ustrezajo povezanim komponentam grafa, tudi pravi povezavni otoki, torej so v $\mathcal{H}_w$. To pa pomeni, da je $\mathcal{H}_w$ polna hierarhija nad množico $\mathcal{V}$ natanko tedaj, ko graf ne vsebuje izoliranih točk. □

Naj bo $\mathcal{H}(A; k, K)$ hierarhija nad množico A, ki vsebuje samo tiste množice iz $\mathcal{H}(A)$, katerih moč je med k in K, torej $\mathcal{H}(A; k, K) = \{X \in \mathcal{H}(A) : k \leq |X| \leq K\}$. Očitno je tudi $\mathcal{H}(A; k, K)$ hierarhija in $\mathcal{H}(A; k, K) \subseteq \mathcal{H}(A)$, kjer je $\subseteq$ običajna vsebovanost množic. Pravimo tudi, da je $\mathcal{H}(A; k, K)$ bolj groba hierarhija kot $\mathcal{H}(A)$. Če je $\mathcal{H}(A)$ polna hierarhija, pa $\mathcal{H}(A; k, K)$ ni več nujno polna.

Trditev 3.4: *Za $k_1 \leq k \leq K \leq K_1$ velja $\mathcal{H}(A; k, K) \subseteq \mathcal{H}(A; k_1, K_1)$.*

Dokaz: Naj bo $X \in \mathcal{H}(A; k, K)$. Potem je $k \leq |X| \leq K$. Ker pa je $k_1 \leq k$ in $K \leq K_1$, je tudi $k_1 \leq |X| \leq K_1$, torej je $X \in \mathcal{H}(A; k_1, K_1)$. □

Trditev 3.5: *Za $k_1 \leq k \leq K \leq K_1$ velja $\mathcal{H}(A; k, K) = \mathcal{H}(\bigcup \mathcal{H}(A; k_1, K_1); k, K)$.*

Dokaz: Tudi to je očitno. Če iz $\mathcal{H}$ najprej odstranimo vse množice, katerih velikost je manjša od k_1 ali večja od K_1, nato pa iz dobljene hierarhije še vse tiste množice, katerih velikost je manjša od k ali večja od K, nam ostanejo natanko tiste množice iz $\mathcal{H}$, katerih velikost je med k in K. □

Bolj kot same hierarhije pa nas zanimajo omrežja, ki jih dobimo pri določanju otokov. Naj bo $\mathcal{N}_p(\mathcal{N}; k, K)$ omrežje pravih točkovnih otokov omrežja $\mathcal{N}$, katerih velikost je vsaj k in največ K. To je podomrežje omrežja $\mathcal{N}$, določeno z množico točk $\bigcup \mathcal{H}_p(\mathcal{N}; k, K)$. Zanima nas, ali lahko velikost točkovnih otokov omejujemo postopoma, torej ali lahko točkovne otoke še bolj omejene velikosti določimo kar na dobljenem omrežju točkovnih otokov omejene velikosti, ali pa moramo vse skupaj narediti spet na prvotnem omrežju. Izkaže se, da lahko.

Trditev 3.6: *Za $k_1 \le k \le K \le K_1$ velja $\mathcal{N}_p(\mathcal{N}; k, K) = \mathcal{N}_p(\mathcal{N}_p(\mathcal{N}; k_1, K_1); k, K)$.*

Dokaz: Recimo, da v omrežju $\mathcal{N}$ določimo vse prave točkovne otoke velikosti vsaj k_1 in največ K_1. Iz omrežja potem odstranimo vse točke, ki ne pripadajo nobenemu takemu otoku, skupaj z vsemi pripadajočimi povezavami. Dobljeno omrežje označimo z $\mathcal{N}_1$. Povezane komponente tega omrežja so ravno maksimalni pravi točkovni otoki velikosti med k_1 in K_1, saj dva ločena točkovna otoka med sabo ne moreta biti povezana.

Poglejmo sedaj, kaj se zgodi, če poskušamo velikost otokov še bolj omejiti. Ker je množica vseh pravih točkovnih otokov hierarhija, je vsak pravi točkovni otok velikosti med k in K vsebovan v nekem maksimalnem pravem točkovnem otoku velikosti med k_1 in K_1, torej je vsebovan v eni od povezanih komponent omrežja $\mathcal{N}_1$. Zato je vseeno, ali prave točkovne otoke velikosti med k in K iščemo v omrežju $\mathcal{N}$ ali pa v omrežju $\mathcal{N}_1$. V obeh primerih dobimo iste otoke. □

Podobno definiramo tudi omrežje $\mathcal{N}_w(\mathcal{N}; k, K)$ pravih povezavnih otokov omrežja $\mathcal{N}$, katerih velikost je vsaj k in največ K. To je podomrežje omrežja $\mathcal{N}$, določeno z množico točk $\bigcup \mathcal{H}_w(\mathcal{N}; k, K)$, iz katerega dodatno odstranimo vse povezave med maksimalnimi otoki in vse povezave znotraj maksimalnih otokov, ki imajo manjšo višino kot pristanišče. Tudi za omrežje povezavnih otokov velja podobno, kot za omrežje točkovnih otokov.

Trditev 3.7: *Za $k_1 \le k \le K \le K_1$ velja $\mathcal{N}_w(\mathcal{N}; k, K) = \mathcal{N}_w(\mathcal{N}_w(\mathcal{N}; k_1, K_1); k, K)$.*

Dokaz: Recimo, da v omrežju $\mathcal{N}$ določimo vse prave povezavne otoke velikosti vsaj k_1 in največ K_1. Iz omrežja potem odstranimo vse točke, ki ne pripadajo nobenemu takemu otoku, vse povezave med maksimalnimi dobljenimi otoki in vse povezave znotraj maksimalnih otokov, ki imajo utež manjšo od uteži pristanišča. Dobljeno omrežje označimo z $\mathcal{N}_1$. Povezane komponente tega omrežja so ravno maksimalni pravi točkovni otoki velikosti med k_1 in K_1, saj smo odstranili vse povezave med maksimalnimi otoki, točke znotraj maksimalnih otokov pa so še vedno med seboj povezane, saj mora obstajata vpeto drevo, katerega povezave imajo vse večjo ali enako utež, kot pristanišče.

Poglejmo sedaj, kaj se zgodi, če poskušamo velikost otokov še bolj omejiti. Ker je množica vseh pravih povezavnih otokov hierarhija, je vsak pravi povezavni otok velikosti med k in K vsebovan v nekem maksimalnem pravem povezavnem otoku velikosti med k_1 in K_1, torej je vsebovan v eni od povezanih komponent omrežja $\mathcal{N}_1$. Zato je vseeno, ali prave povezavne otoke velikosti med k in K iščemo v omrežju $\mathcal{N}$ ali pa v omrežju $\mathcal{N}_1$. V obeh primerih dobimo iste otoke. □

Postavlja se tudi vprašanje, ali sta hierarhiji $\mathcal{H}_p$ in $\mathcal{H}_w$ kako povezani med sabo. Poglejmo najprej, v kakšnem odnosu so povezavni otoki na danem omrežju s točkovnimi otoki na pripadajočem povezavnem omrežju.

Definicija 3.11: *Povezavno omrežje* omrežja $\mathcal{N} = (\mathcal{V}, \mathcal{L}, w)$ z utežmi na povezavah je omrežje $\mathcal{N}_L = (\mathcal{L}, \mathcal{L}_L, w)$ z utežmi v točkah, katerega točke so povezave omrežja $\mathcal{N}$, dve točki pa sta povezani, če imata ustrezni povezavi v omrežju $\mathcal{N}$ skupno krajišče.

$$(s\,;t) \in \mathcal{L}_L \iff s,t \in \mathcal{L} \wedge \mathcal{V}(s) \cap \mathcal{V}(t) \neq \emptyset$$

Vrednost izbrane točke v omrežju $\mathcal{N}_L$ je pri tem enaka uteži pripadajoče povezave v omrežju $\mathcal{N}$.

Povezavni otok v omrežju $\mathcal{N}$ je skupina točk v $\mathcal{N}$, točkovni otok v omrežju $\mathcal{N}_L$ pa je skupina točk v $\mathcal{N}_L$, torej skupina povezav v $\mathcal{N}$. Če bomo želeli takšne otoke primerjati med sabo, bomo morali množice povezav nekako predelati v množice točk. To bomo storili tako, da bomo vsako povezavo v množici nadomestili z njenima krajiščema.

Trditev 3.8: *Naj bo $\mathcal{N}$ omrežje, v katerem funkcija w določa uteži na povezavah, in naj bo $\mathcal{N}_L$ pripadajoče povezavno omrežje, v katerem funkcija $p = w$ določa vrednosti točk. Potem je $\mathcal{H}_w(\mathcal{N}) = \mathcal{V}(\mathcal{H}_p(\mathcal{N}_L))$*

Dokaz: Naj bo $A \in \mathcal{H}_w(\mathcal{N})$ poljuben pravi povezavni otok omrežja $\mathcal{N}$. Potem obstaja vpeto drevo $\mathcal{T}$ na A, tako da so uteži njegovih povezav vse večje od uteži povezav, ki imajo eno krajišče na otoku, drugo pa izven otoka A. Naj bo t najmanjša utež povezav drevesa $\mathcal{T}$ in naj bo B množica tistih povezav znotraj otoka A, ki imajo utež vsaj t. Točke omrežja $\mathcal{N}_L$, ki ustrezajo tem povezavam, imajo vse vrednost večjo ali enako t. Ker imajo v omrežju $\mathcal{N}$ vse sosednje povezave utež manjšo od t, imajo tudi v omrežju $\mathcal{N}_L$ vse sosednje točke vrednost manjšo od t. Množica B je torej pravi točkovni otok omrežja $\mathcal{N}_L$. Množica krajišč povezav iz B pa je enaka množici A, saj povezave iz B povezujejo samo točke A, ker pa določajo povezan podgraf, so notri vse točke iz A.

Zdaj pa naj bo $A \in \mathcal{H}_p(\mathcal{N}_L)$ poljuben pravi točkovni otok omrežja $\mathcal{N}_L$. Naj bo t vrednost najnižje točke otoka A. Ta otok je podmnožica množice povezav omrežja $\mathcal{N}$. Vsaka povezava iz A ima utež vsaj t, vse sosedne povezave pa imajo utež manjšo od t, ker imajo v omrežju $\mathcal{N}_L$ vse sosedne točke vrednost manjšo od t. Ker vsak točkovni otok določa povezan podgraf, tudi množica povezav A določa povezan podgraf v $\mathcal{N}$. Množica krajišč povezav iz množice A je torej pravi povezavni otok omrežja $\mathcal{N}$. □

Za konec si še poglejmo, ali obstaja kakšna povezava med pravimi točkovnimi in pravimi povezavnimi otoki na omrežju, kjer vrednost posamezne točke definiramo kot največjo utež njenih sosednih povezav.

$$p(v) = \max_{u \in N(v)} w((v\,;u)) \tag{3.1}$$

Naj bo $A \in \mathcal{H}_p$, torej je A pravi točkovni otok. Naj bo v pristanišče otoka A, $t = p(v)$ pa njegova vrednost. Vsaka točka na otoku A ima potem vrednost večjo ali enako t, vsaka točka iz soseščine $N(A)$ pa vrednost manjšo od t.

Iz definicije vrednosti točk sledi, da mora imeti vsaka točka v omrežju vsaj eno sosedno povezavo z enako utežjo, kot je vrednost točke, hkrati pa je to tudi največja utež njenih sosednih povezav. Ker so vrednosti točk v soseščini $N(A)$ vse manjše od t, so tudi uteži povezav, ki imajo eno krajišče na otoku A, drugo pa izven otoka, vse manjše od t. Za vsako točko na otoku A pa mora obstajati vsaj še ena točka znotraj otoka, do katere obstaja povezava z utežjo vsaj t. Od tod dobimo prvo ugotovitev, da vsi pravi točkovni otoki vsebujejo vsaj dve točki.

Poglejmo si zdaj vse tiste povezave znotraj otoka A, ki imajo utež vsaj t. Če bi te povezave med seboj povezale vse točke otoka A, potem je A tudi pravi povezavni otok, torej $A \in \mathcal{H}_w$. Žal pa to ni vedno res, saj lahko te povezave razbijejo otok A na več povezanih komponent. Vsaka od teh komponent pa je pravi povezavni otok. Od tod sledi, da vsak pravi točkovni otok na nivoju t razpade na nekaj pravih povezavnih otokov na nivojih vsaj t.

Preverimo še v drugo smer, torej ali je vsak pravi povezavni otok na nivoju t vsebovan v katerem od pravih točkovnih otokov na nivoju vsaj t. Hitro vidimo, da so vrednosti vseh točk na takem povezavnem otoku večje ali enake t, saj mora obstajati vpeto drevo, ki vsebuje samo povezave z utežmi vsaj t. Če k otoku dodamo še vse točke iz soseščine, katerih vrednost je vsaj t, pa dobimo iskani pravi točkovni otok na nivoju vsaj t. Povzemimo

Izrek 3.9: *Če v omrežju $\mathcal{N}(\mathcal{V}, \mathcal{L}, w)$ definiramo funkcijo $p\colon \mathcal{V} \to \mathbb{R}$ s predpisom 3.1, veljajo naslednje trditve.*

- *Vsak pravi točkovni otok vsebuje vsaj dve točki.*
- *Vsak pravi točkovni otok na nivoju t je sestavljen iz enega ali več pravih povezavnih otokov na nivoju vsaj t.*
- *Vsak pravi povezavni otok na nivoju t je vsebovan v pravem točkovnem otoku na nivoju vsaj t.*

Četrto poglavje

Sredice

Eno od pomembnih vprašanj v analizi družboslovnih omrežij je prepoznavanje močno povezanih skupin točk. Takšne skupine so podmnožice točk, znotraj katerih so relativno močne povezave. Pri formalnem opisu takih skupin naletimo na različne pojme, kot so k-klika, k-pleme, k-križ, k-spletka, k-sredica [37], množica lambda ... Za večino od njih se izkaže, da so algoritmično zelo zahtevni (NP težki ali vsaj kvadratični), za določanje sredic, ki jih je že leta 1983 predstavil Seidman, pa obstaja zelo učinkovit postopek [9].

Po analizi takih gosto povezanih skupin točk lahko skupine stisnemo v eno samo točko ali pa jih cele odstranimo iz omrežja. Tako dobimo manjše, lažje obvladljivo omrežje, ki ga analiziramo naprej na isti način, ali pa kako drugače.

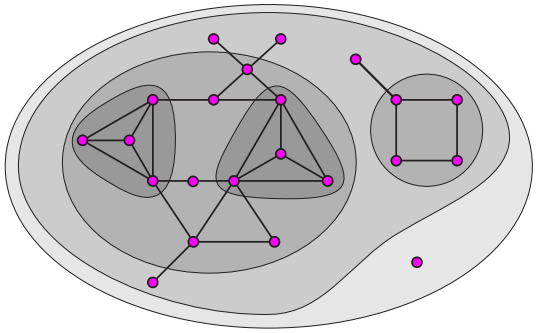
Definicija 4.1: Podgraf $\mathcal{H} = (\mathcal{C}, \mathcal{L}|\mathcal{C})$ grafa $\mathcal{G} = (\mathcal{V}, \mathcal{L})$, določen z množico $\mathcal{C} \subseteq \mathcal{V}$, je *sredica reda k* ali *k-sredica* natanko tedaj, ko za vsak $v \in \mathcal{C}$ velja: $\deg_{\mathcal{H}}(v) \geq k$, in je $\mathcal{H}$ največji podgraf s to lastnostjo. Sredici največjega reda rečemo tudi *glavna sredica*. *Središčno število* točke v je največji red sredice, ki vsebuje to točko. Pogosto tudi podmnožici točk $\mathcal{C}$ rečemo sredica, saj enolično določa podgraf $\mathcal{H}$.

Stopnjo točk lahko definiramo na različne načine. Stopnja je lahko število različnih sosedov, število vhodnih povezav, število izhodnih povezav, vsota števil vhodnih in izhodnih povezav ... Tako dobimo različne vrste sredic, vse pa imajo naslednje lastnosti.

- Sredica reda 0 je vedno cel graf. Vsaka točka grafa ima namreč vsaj 0 sosedov.
- Sredice so gnezdene. To pomeni, da je vsaka sredica vsebovana v vseh sredicah manjšega reda, torej sredice določajo gnezdeno razslojitev grafa.

$$i < j \implies \mathcal{H}_j \subseteq \mathcal{H}_i$$

- Sredice niso nujno povezani podgrafi. Na sliki 4.1 vidimo primer, ko ima sredica reda 2 dve komponenti. To ni nič presenetljivega, saj že osnovni graf ni povezan. Na sliki pa lahko vidimo tudi dve komponenti sredice reda 3, ki pa ležita v isti komponenti prvotnega grafa.



Slika 4.1: *Sredice redov 0, 1, 2 in 3*

Množica vseh povezanih komponent posameznih slojev je hierarhija. Povezani komponenti v istem sloju imata namreč prazen presek, če pa gre za komponenti v različnih slojih, imata prazen presek ali pa je komponenta višjega sloja vsebovana v komponenti nižjega sloja.

4.1 Postopek za določanje hierarhije sredic

V postopku za določanje hierarhije sredic moramo vsaki točki danega grafa izračunati njeno središčno število. Pri tem si pomagamo z naslednjo lastnostjo [6]:

> Če v grafu rekurzivno odstranimo vse točke stopnje manjše od k (in vse pripadajoče povezave), je preostanek grafa ravno sredica reda k.

Postopek torej temelji na odstranjevanju točk. Pričnemo s celim grafom (sredica reda 0), kjer najprej odstranimo vse točke stopnje 0. Središčna števila odstranjenih točk so enaka 0. Iz dobljenega grafa (sredica reda 1) rekurzivno odstranimo vse točke stopenj 0 ali 1. Ostane nam sredica reda 2, odstranjene točke pa imajo središčno število enako 1. Postopek ponavljamo tako dolgo, da odstranimo vse točke. Središčno število točke, ki jo odstranjujemo, je maksimum njene trenutne stopnje in središčnega števila točke, ki smo jo odstranili pred njo.

Točke grafa je potrebno najprej urediti naraščajoče glede na njihove stopnje, nato pa jih po vrsti odstranjevati iz grafa. Vsakič, ko odstranimo neko točko, se stopnja njenih sosedov zmanjša, zato moramo točke ustrezno preurediti.

Opazimo še, da točk ni potrebno odstranjevati iz grafa, dovolj je že, da si samo zapomnemo, kakšne bi bile stopnje preostalih točk, če bi točko odstranili. Če je stopnja sosedne točke manjša od stopnje trenutne točke, potem je sosedna točka že bila obdelana in navidezno odstranjena iz grafa. V tem primeru ne naredimo ničesar. Če naletimo na sosednjo točko, ki ima isto stopnjo kot trenutna točka, imamo dve možnosti. Če je sosednja točka že bila obdelana, ne smemo narediti ničesar, če pa še ni bila obdelana, bo kmalu prišla na vrsto, njeno središčno število pa bo enako

Postopek 4.1 *Določanje sredic*

Podatki: graf $\mathcal{G} = (\mathcal{V}, \mathcal{L})$, predstavljen s seznami sosedov
Rezultat: tabela *core* s središčnimi števili posameznih točk

```
izračunaj stopnje točk (dobimo tabelo degree)
uredi množico točk V nepadajoče glede na njihove stopnje
for each v ∈ V (v dobljenem vrstnem redu) do begin
    core[v] := degree[v]
    for each u ∈ N(v) do
        if degree[u] > degree[v] then begin
            degree[u] := degree[u] − 1
            preuredi V
        end
end
```

središčnemu številu trenutne točke, torej nam ni treba storiti ničesar. Ukrepati moramo samo v primeru, ko naletimo na sosedno točko večje stopnje. Takšna točka še ni bila obdelana, zato moramo njeno stopnjo zmanjšati za 1 in preurediti množico točk. Tako pridemo do postopka 4.1.

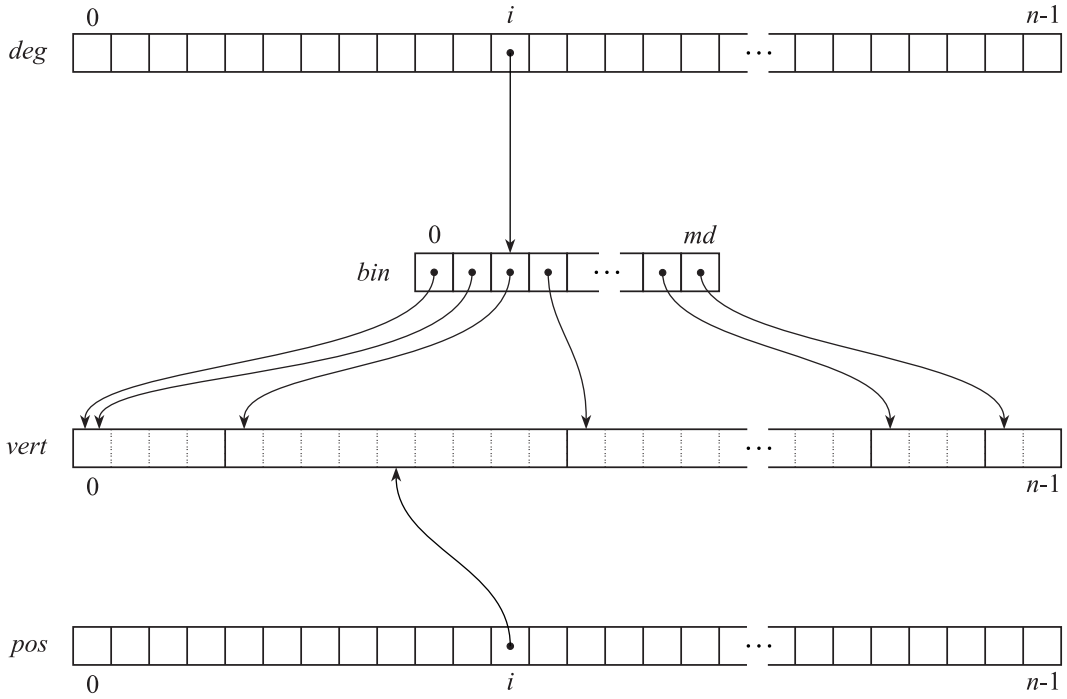
4.2 Izvedba

V začetku postopka moramo čim bolj učinkovito urediti množico točk glede na njihove stopnje. Ker so vse stopnje nenegativne, največja stopnja pa je običajno majhna glede na število točk, lahko uporabimo urejanje s koši. Vse točke z isto stopnjo damo v isti koš, v tabelo pa nato po vrsti zapišemo najprej točke iz prvega koša (točke s stopnjo 0), nato točke iz drugega koša (točke s stopnjo 1) ...

Preurejanje množice točk je malo zahtevnejše. Tabelo je potrebno preurediti vsakič, ko neki točki zmanjšamo stopnjo za 1. Če si predstavljamo, da so točke v tabeli še vedno v koših, to pomeni, da moramo točko iz enega koša prestaviti v prejšnji koš. Ker položaj točk znotraj košev ni pomemben, lahko to naredimo tako, da točko, ki jo moramo prestaviti, zamenjamo s točko, ki je prva v istem košu, nato pa velikost prejšnjega koša povečamo za ena, začetek koša, v katerem je bila točka, pa pomaknemo za eno mesto v desno (velikost tega koša se pri tem zmanjša za ena). Da bi to lahko hitro izvedli, moramo poznati položaje točk v tabeli in začetke vseh košev (kje v tabeli se prične kateri koš).

Če si predstavljamo, da so točke oštevilčene od 0 do $n-1$, bi podatkovna struktura bila videti takšna, kot je prikazana na sliki 4.2.

Izvedba postopka za določanje hierarhije sredic je narejena v programskem jeziku C++. Dani graf $\mathcal{G}$ je predstavljen s strukturo `Graph`. Ker lahko graf predstavimo na več različnih načinov, strukture ne bomo podrobneje opisovali. Predpostavimo



Slika 4.2: *Tabele v postopku določanja hierarhije sredic*

samo, da so točke grafa oštevilčene od 0 do $n-1$. Uporabnik mora priskrbeti še funkcije `size`, `degree` in `in neighbors`, opisane v tabeli 4.1. Z uporabo primerne strukture za opis grafa $\mathcal{G}$ (seznami sosedov) lahko vse omenjene funkcije napišemo tako, da bodo imele konstantno časovno zahtevnost.

Celoten postopek za določanje hierarhije sredic je izveden s funkcijo `cores` v programu 4.1, ki za parameter dobi podatke o grafu $\mathcal{G}$, do katerih pridemo preko spremenljivke `g`. Rezultat funkcije je tabela, ki vsebuje središčna števila posameznih točk grafa $\mathcal{G}$.

Najprej definiramo nekaj pomožnih spremenljivk (03-04) in tabel (05-07). V spremenljivko `n` naračunamo velikost grafa (število točk v grafu). Tabela `vert` bo vsebovala točke grafa, urejene nepadajoče glede na njihove stopnje. Kje v tabeli `vert` se nahaja posamezna točka, bo zapisano v tabeli `pos`. V tabelo `deg` bomo naračunali stopnje posameznih točk, na koncu pa bomo v tej tabeli dobili njihova središčna števila.

`ime(`*parametri*`)`	rezultat
`size(`$\mathcal{G}$`)`	število točk grafa $\mathcal{G}$
`degree(`$\mathcal{G}$`,`v`)`	stopnja točke v v grafu $\mathcal{G}$
`in neighbors(`$\mathcal{G}$`,`v`)`	naslednji še neobiskan sosed točke v v grafu $\mathcal{G}$

Tabela 4.1: Opis funkcij, ki jih mora priskrbeti uporabnik

Program 4.1: *Določanje sredic v enostavnih neusmerjenih grafih*

```
01  int *cores(Graph *g)
02  {
03     int v, d;
04     int n = size(g);
07     int *vert = new int[n];
06     int *pos = new int[n];
05     int *deg = new int[n];
08
09     int md = 0;
10     for (v = 0; v < n; ++v) {
11        deg[v] = degree(g, v);
12        if (deg[v] > md) md = deg[v];
13     }
14
15     int *bin = new int[md + 1];
16     for (d = 0; d <= md; ++d) bin[d] = 0;
17     for (v = 0; v < n; ++v) ++bin[deg[v]];
18
19     int start = 0;
20     for (d = 0; d <= md; ++d) {
21        int num = bin[d];
22        bin[d] = start;
23        start += num;
24     }
25
26     for (v = 0; v < n; ++v) {
27        pos[v] = bin[deg[v]]++;
28        vert[pos[v]] = v;
29     }
30
31     for (d = md; d > 0; --d) bin[d] = bin[d - 1];
32     bin[0] = 0;
33
34     for (int k = 0; k < n; ++k) {
35        v = vert[k];
36        for (int u in neighbors(g, v)) {
37           if (deg[u] > deg[v]) {
38              int du = deg[u];
39              int pu = pos[u];
40              int pw = bin[du];
41              int w = vert[pw];
42              if (u != w) {
43                 pos[u] = pw;  vert[pu] = w;
44                 pos[w] = pu;  vert[pw] = u;
45              }
46              ++bin[du];
47              --deg[u];
48           }
49        }
50     }
51
52     delete [] vert;
53     delete [] pos;
54     delete [] bin;
55     return deg;
56  }
```

Stopnje točk izračunamo v zanki (10-13), tako da za vsako točko pokličemo funkcijo `degree`. Hkrati izračunamo še največjo stopnjo `md`. Ko vemo, kakšna je največja stopnja, lahko ustvarimo še eno pomožno tabelo. To je tabela `bin`, v kateri bomo za vsako možno stopnjo imeli indeks, kje v tabeli `vert` se pričnejo točke te stopnje. Grafičen prikaz uporabljenih tabel je na sliki 4.2.

Nato uredimo točke v naraščajočem vrstnem redu po njihovih stopnjah. Pri tem uporabimo urejanje s koši (15-29). Najprej preštejemo, koliko točk bo v posameznem košu (15-17), pri čemer so v istem košu točke z isto stopnjo. Koši so oštevilčeni od 0 do `md`.

Iz podatkov o velikosti košev lahko določimo (19-24) začetne položaje košev v tabeli `vert`. Koš 0 se prične na mestu 0, drugi koši pa na mestu, ki je enako vsoti začetnega položaja in velikosti prejšnjega koša. Da bi se izognili dodatni tabeli, za shranjevanje začetnih položajev uporabimo kar isto tabelo (`bin`). Nato lahko zapišemo točke grafa `g` v tabelo `vert` (26-29). Za vsako točko vemo, kateremu košu pripada in kakšen je začetni položaj tega koša. Točko zapišemo na ustrezno mesto, v tabelo `pos` zapišemo njen položaj, in povečamo začetni položaj koša, ki smo ga uporabili. Ko se zanka izteče, so točke urejene po stopnjah.

V zadnjem koraku pripravljalne faze moramo povrniti začetne položaje košev na stare vrednosti (31-32). V prejšnjem koraku smo jih večkrat povečali, ko smo zapisali točke v ustrezne koše. Očitno je, da je spremenjen začetni položaj ravno prvotni začetni položaj naslednjega koša. Dovolj je torej samo prepisati vrednosti v tabeli `bin` za eno mesto v desno. Začetni položaj koša 0 postavimo nazaj na 0.

Glavna zanka (34-50) ustreza zanki **for each** v postopku 4.1. Zanka teče po vseh točkah `v` grafa `g` v vrstnem redu, določenem s tabelo `vert`. Središčno število tekoče točke `v` je ravno trenutna stopnja te točke, ki je že zapisana v tabeli `deg`. Vsaki sosedni točki `u` točke `v`, ki ima večjo stopnjo, moramo zmanjšati stopnjo in jo prestaviti za en koš v levo. To je operacija, ki jo opravimo v konstantnem času. Najprej zamenjamo točko `u` in prvo točko v istem košu. V tabeli `pos` zamenjamo tudi njuna položaja. Na koncu povečamo začetni položaj koša (s tem povečamo velikost prejšnjega in zmanjšamo velikost trenutnega koša).

4.2.1 Časovna zahtevnost

Za izračun stopenj točk (10-13) potrebujemo $\mathcal{O}(n)$ časa. Urejanje s koši (15-32) je sestavljeno iz petih zank velikosti največ n. Vsaka se izvede v konstantnem času, torej je celotna časovna zahtevnost urejanja $\mathcal{O}(n)$.

Stavek (35) je konstantne časovne zahtevnosti, izvede se za vsako točko enkrat, torej prispeva $\mathcal{O}(n)$ k časovni zahtevnosti postopka. Pogojni stavek (37-48) je tudi konstantne časovne zahtevnosti. Ker se izvrši za vsako povezavo v grafu največ dvakrat, je prispevek zanke (36-49) v vseh ponovitvah zanke (34-50) ravno $\mathcal{O}(\max\{m, n\})$.

Če seštejemo vse skupaj, je časovna zahtevnost postopka enaka $\mathcal{O}(\max\{m, n\})$. Ker pa za povezana omrežja velja $m \geq n - 1$, je $\mathcal{O}(\max\{m, n\}) = \mathcal{O}(m)$.

4.2.2 Prilagoditev postopka za druge vrste sredic

Da bi opisani postopek deloval tudi na grafih z usmerjenimi povezavami in zankami, so potrebni samo malenkostni popravki, odvisno od tega, kako definiramo stopnjo točke.

V primeru vhodne (izhodne) stopnje mora funkcija `in neighbors` v vrstici 36 vrniti naslednjega še neobiskanega izhodnega (vhodnega) soseda. Če je stopnja definirana kot vsota vhodne in izhodne stopnje pa mora vrniti naslednjega še neobiskanega vhodnega ali izhodnega soseda. Če je neka točka vhodni in izhodni sosed, ga mora funkcija vrniti dvakrat. Maksimalna stopnja točke je v tem primeru lahko največ $2n - 2$, na kar moramo paziti pri rezerviranju pomnilnika za tabelo `bin`.

4.3 Primer

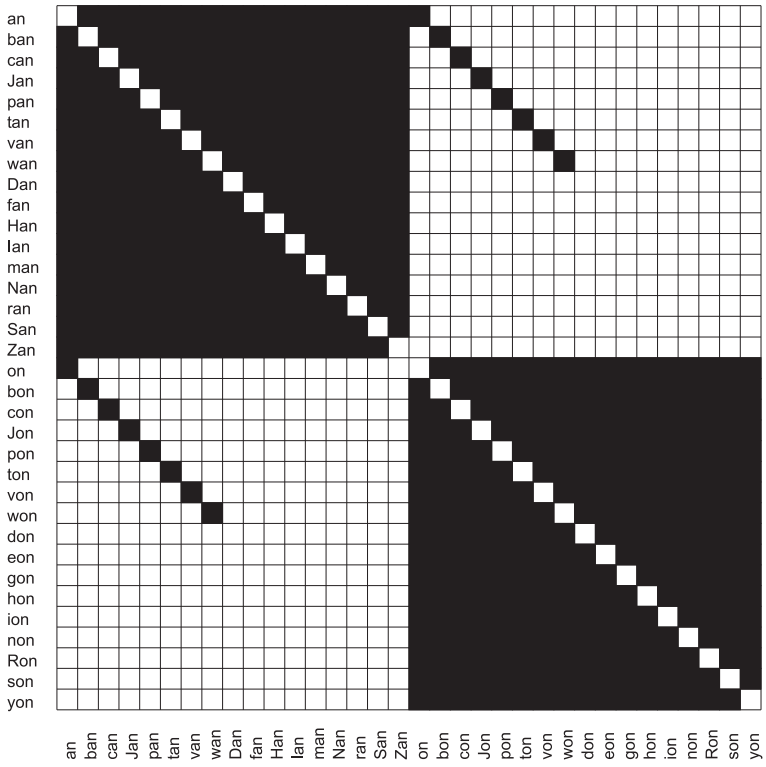
Knuthov slovar angleških besed [31] je omrežje, ki ima 52652 točk (angleške besede dolžine od 2 do 8 znakov) in 89038 povezav. Dve točki (besedi) sta povezani, če lahko eno besedo dobimo iz druge tako, da zamenjamo, odvzamemo ali dodamo eno črko. Dobljeno omrežje je redko, saj je povprečna stopnja komaj 3.382. Rezultati programa so prikazani v tabeli 4.2.

	točke s središčnim številom k		velikost sredice reda k	
k	#	%	#	%
25	26	0.049	26	0.049
16	34	0.065	60	0.114
15	16	0.030	76	0.144
14	59	0.112	135	0.257
13	82	0.156	217	0.412
12	200	0.380	417	0.792
11	202	0.384	619	1.176
10	465	0.883	1084	2.059
9	504	0.957	1588	3.016
8	923	1.753	2511	4.769
7	1114	2.116	3625	6.885
6	1590	3.020	5215	9.905
5	2423	4.602	7638	14.507
4	3859	7.329	11497	21.836
3	5900	11.206	17397	33.042
2	8391	15.937	25788	48.978
1	13539	25.714	39327	74.693
0	13325	25.308	52652	100.000

Tabela 4.2: Sredice v primeru Knuthovega slovarja angleških besed

Točke s središčnim številom 0 so izolirane točke. Točke s središčnim številom 1 so točke na izrastkih (poti, ki štrlijo iz omrežja). Sredica reda 25 (glavna sredica) je sestavljena iz 26 točk, kjer ima vsaka točka vsaj 25 sosedov znotraj sredice (očitno je to klika). Besede, ki ustrezajo tem točkam, so `a's`, `b's`, `c's`, ..., `y's`, `z's`.

Sredica reda 16 ima dodatnih 34 točk (`an`, `on`, `ban`, `bon`, `can`, `con`, `Dan`, `don`, `eon`, `fan`, `gon`, `Han`, `hon`, `Ian`, `ion`, `Jan`, `Jon`, `man`, `Nan`, `non`, `pan`, `pon`, `ran`, `Ron`, `San`, `son`, `tan`, `ton`, `van`, `von`, `wan`, `won`, `yon`, `Zan`). Med sredicama redov 16 in 25 ni nobene povezave. Matrika sosednosti podgrafa, določenega s temi 34 točkami, je prikazana na sliki 4.3. Na njej lahko vidimo dve kliki reda 17 in nekaj dodatnih povezav.



Slika 4.3: *Matrika sosednosti sredice reda 16 brez sredice reda 25*

Sredica reda 15 ima dodatnih 16 točk (`ow`, `bow`, `cow`, `Dow`, `how`, `jow`, `low`, `mow`, `now`, `pow`, `row`, `sow`, `tow`, `vow`, `wow`, `yow`). To je spet klika, saj se besede razlikujejo samo v prvih črkah.

4.4 Primer

Delovanje postopka si poglejmo še na omrežju avtorjev `geombib` [25]. Sumarni rezultati so predstavljeni v tabeli 4.3.

Sredica	Frekvenca	KumFrekv%	Predstavnik
0	1185	16.1378	N. Bourbaki
1	2218	46.3435	S. Kambhampati
2	1714	69.6854	G. Bilardi
3	1023	83.6171	Y. I. Yoon
4	503	90.4671	B. B. Kimia
5	248	93.8445	C. A. Duncan
6	122	95.5059	T. M. Murali
7	126	97.2218	J. M. Kleinberg
8	34	97.6849	F. F. Yao
9	37	98.1888	K. R. Lee
10	20	98.4611	H. Alt
11	52	99.1693	M. Flickner
13	1	99.1829	M. H. Overmars
14	7	99.2782	M. Sharir
15	14	99.4689	N. M. Amato
16	17	99.7004	D. White
21	22	100.0000	L. J. Guibas
Vsota	7343		

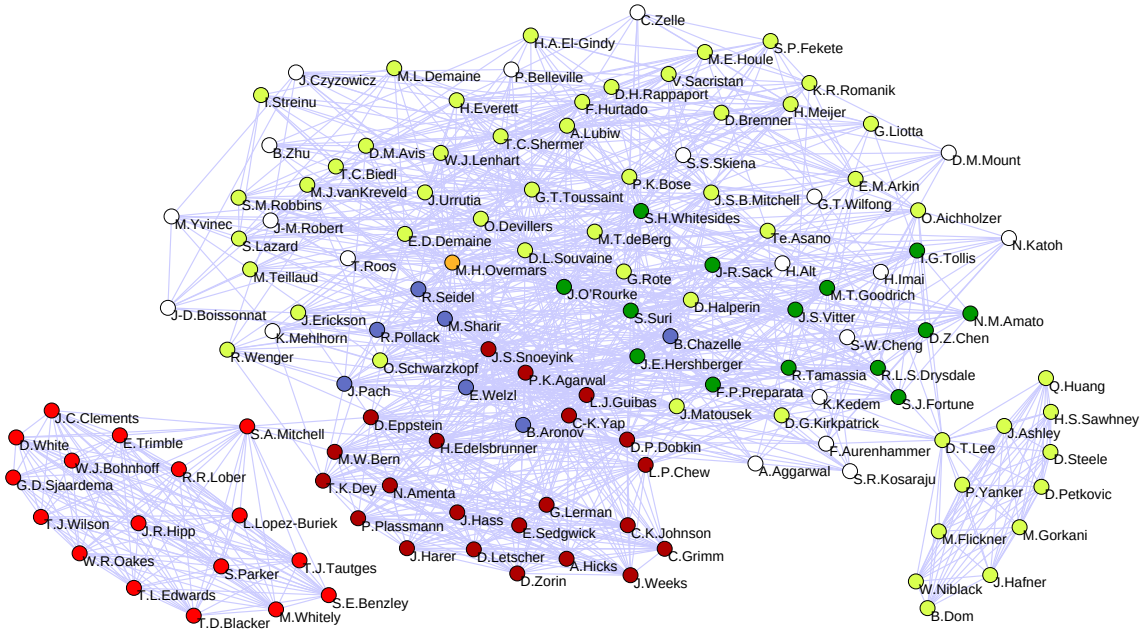
Tabela 4.3: Porazdelitev središčnih števil

Točke stopnje 0 so izolirane točke – avtorji brez soavtorjev. Sredica reda 21 (glavna sredica) je sestavljena iz 22 točk, kjer ima vsaka točka vsaj 21 sosedov znotraj sredice (očitno je to klika).

Na sliki 4.4 je prikazan podgraf, porojen s točkami iz sredice reda 10 (in višjih). Ima 133 točk. Te so obarvane različno (sivi odtenki) glede na središčno število. Slika je bila dobljena z uporabo avtomatskega risanja s postopkom Kamada-Kawai. Nekatere točke so bile dodatno rahlo prestavljene, da so oznake lažje berljive.

Glavna sredica je sestavljena iz najtemnejših točk v sredini spodnjega dela slike. Na njo se povezujejo sredice nižjih stopenj. Podrobnejši pogled v prerez na nivoju 15 razkrije, da tudi točke s središčnim številom 15 tvorijo kliko reda 14. Ta klika je močno povezana s skupino avtorjev (P. K. Agarwal, D. P. Dobkin, L. J. Guibas, C.-K. Yap, H. Edelsbrunner, D. Eppstein, J. S. Snoeyink, L. P. Chew in M. W. Bern) iz glavne sredice. Drugi člani glavne sredice sodelujejo v glavnem samo med seboj. Dve skupini s središčnima številoma 16 in 11 (obe sta spet kliki) sta povezani z 'zveznimi' avtorji (Scott A. Mitchell in D. T. Lee) iz glavne skupine.

Opazimo tudi, da je vrstni red avtorjev glede na središčna števila rahlo drugačen, kot vrstni red glede na njihove stopnje (avtorji in število različnih soavtorjev): L. J. Guibas (102), P. K. Agarwal (98), J. S. Snoeyink (91), H. Edelsbrunner (90), M. H. Overmars (88), M. Sharir (87), J. O'Rourke (85), R. Tamassia (79), J. S. B. Mitchell (76), C.-K. Yap (76), E. Welzl (74), D. P. Dobkin (73), G. T. Toussaint (72), M. T. Goodrich (70), K. Mehlhorn (69), R. E. Tarjan (69).



Slika 4.4: *Računska geometrija: sredica reda 10*

Števila skupnih člankov, ki jih imata dva avtorja, v tem primeru nismo upoštevali. Da bi upoštevali tudi vrednosti na povezavah, bi morali uporabiti posplošen postopek za določanje sredic [8]. Tudi ta je učinkovit (podkvadratičen), njegova časovna zahtevnost je $\mathcal{O}(m \cdot \max\{\Delta, \log n\})$, kjer je Δ maksimalna stopnja v omrežju.

4.5 Posplošitev sredic

Do sedaj je pri določanju sredic edino vlogo igrala stopnja posameznih točk. Namesto stopnje pa bi lahko uporabili tudi kakšno drugo funkcijo. Pri izbiri funkcije imamo veliko možnosti, še posebej, če imamo v grafu podane vrednosti točk ali uteži na povezavah. To nas pripelje do posplošitve pojma sredice.

Definicija 4.2: *Točkovna funkcija* p na omrežju $\mathcal{N}$ ali krajše *p-funkcija* je funkcija $p(v, \mathcal{C})$, $v \in \mathcal{V}$, $\mathcal{C} \subseteq \mathcal{V}$ z realnimi vrednostmi.

Poglejmo si nekaj primerov točkovnih funkcij.

- $p_1(v, \mathcal{C}) = \deg(v, \mathcal{C})$
- $p_2(v, \mathcal{C}) = \text{indeg}(v, \mathcal{C})$
- $p_3(v, \mathcal{C}) = \text{outdeg}(v, \mathcal{C})$
- $p_4(v, \mathcal{C}) = \text{indeg}(v, \mathcal{C}) + \text{outdeg}(v, \mathcal{C})$

- $p_5(v,\mathcal{C}) = \sum_{u\in N(v,\mathcal{C})} w((v\,;u))$, kjer je $w\colon \mathcal{L} \to \mathbb{R}_0^+$
- $p_6(v,\mathcal{C}) = \max_{u\in N(v,\mathcal{C})} w((v\,;u))$, kjer je $w\colon \mathcal{L} \to \mathbb{R}$
- $p_7(v,\mathcal{C}) =$ število ciklov dolžine k po točkah iz $\mathcal{C}$, ki gredo skozi točko v
- $p_a(v,\mathcal{C}) = \dfrac{\deg(v,\mathcal{C})}{\deg(v)}$, če je $\deg(v) > 0$; 0, sicer (težave z listi – $\deg(v) = 1$)
- enovita relativna gostota
  $p_b(v,\mathcal{C}) = \dfrac{\deg(v,\mathcal{C})}{\max_{u\in N(v)} \deg(u)}$, če je $\deg(v) > 0$; 0, sicer
  $p'_b(v,\mathcal{C}) = \dfrac{\deg(v,\mathcal{C})}{\max_{u\in N(v,\mathcal{C})} \deg(u,\mathcal{C})}$, če je $\deg(v,\mathcal{C}) > 0$; 0, sicer
- raznovrstnost
  $p_c(v,\mathcal{C}) = \max_{u\in N(v,\mathcal{C})} \deg(u) - \min_{u\in N(v,\mathcal{C})} \deg(u)$
  $p'_c(v,\mathcal{C}) = \max_{u\in N(v,\mathcal{C})} \deg(u,\mathcal{C}) - \min_{u\in N(v,\mathcal{C})} \deg(u,\mathcal{C})$
- $p_d(v,\mathcal{C}) = \max_{u\in N^+(v,\mathcal{C})} \deg(u) - \min_{u\in N^+(v,\mathcal{C})} \deg(u)$
  $p'_d(v,\mathcal{C}) = \max_{u\in N^+(v,\mathcal{C})} \deg(u,\mathcal{C}) - \min_{u\in N^+(v,\mathcal{C})} \deg(u,\mathcal{C})$
- za primer, ko je $\mathcal{G} = (\mathcal{V},\mathcal{E})$, $K(X)$ naj bo poln graf na množici točk X
  $p_e(v,\mathcal{C}) = \dfrac{|\mathcal{E}(\mathcal{C}) \cap \mathcal{E}(K(N^+(v)))|}{|\mathcal{E}(K(N^+(v)))|}$
  $p'_e(v,\mathcal{C}) = \dfrac{|\mathcal{E}(\mathcal{C}) \cap \mathcal{E}(K(N^+(v,\mathcal{C})))|}{|\mathcal{E}(K(N^+(v,\mathcal{C})))|}$

V definicijah funkcij p_c in p_d lahko funkcijo deg zamenjamo s poljubno funkcijo $q\colon \mathcal{V} \to \mathbb{R}$.

Definicija 4.3: Podgraf $\mathcal{H} = (\mathcal{C}, \mathcal{L}|\mathcal{C})$, porojen z množico točk $\mathcal{C} \subseteq \mathcal{V}$ je *p-sredica na nivoju $t \in \mathbb{R}$* natanko tedaj, ko

- $\forall v \in \mathcal{C} : t \le p(v,\mathcal{C})$
- $\mathcal{C}$ je največja takšna množica.

Definicija 4.4: Točkovna funkcija p je *monotona* natanko tedaj, ko velja

$$\mathcal{C}_1 \subset \mathcal{C}_2 \Longrightarrow \forall v \in \mathcal{V} : p(v,\mathcal{C}_1) \le p(v,\mathcal{C}_2)$$

Točkovne funkcije p_1 – p_7 in p_a – p_e so vse monotone, funkcije p'_a – p'_e pa ne. Za monotono točkovno funkcijo p lahko dobimo p-sredico na nivoju t z zaporednim brisanjem točk, v katerih je vrednost funkcije p manjša od t, glej postopek 4.2.

Postopek 4.2 *Določanje p-sredice na nivoju t*

Podatki: omrežje $\mathcal{N} = (\mathcal{V}, \mathcal{L}, p)$, predstavljeno s seznami sosedov in $t \in \mathbb{R}$
Rezultat: množica $\mathcal{C} \subseteq \mathcal{V}$, $\mathcal{C}$ je p-sredica na nivoju t

$\mathcal{C} := \mathcal{V}$
while $\exists v \in \mathcal{C} : p(v, \mathcal{C}) < t$ **do** $\mathcal{C} := \mathcal{C} \setminus \{v\}$

Trditev 4.1: *Za monotono točkovno funkcijo p postopek 4.2 vrne p-sredico na nivoju t.*

Dokaz: Očitno je, da množica $\mathcal{C}$, ki jo vrne postopek, zadošča prvi lastnosti iz definicije p-sredice. Pokazati moramo še, da je rezultat neodvisen od vrstnega reda točk, ki jih odstranjujemo.

Predpostavimo nasprotno. Naj obstajata dve različni p-sredici na nivoju t. Označimo ju s $\mathcal{C}$ in $\mathcal{D}$. Sredico $\mathcal{C}$ smo dobili z zaporednim odstranjevanjem točk $u_1, u_2, u_3, \ldots, u_r$, sredico $\mathcal{D}$ pa z zaporednim odstranjevanjem točk $v_1, v_2, v_3, \ldots, v_q$. Predpostavimo, da $\mathcal{D} \setminus \mathcal{C} \neq \emptyset$. Pokazali bomo, da nas to pripelje do protislovja.

Izberimo poljubno točko $z \in \mathcal{D} \setminus \mathcal{C}$. Pokazali bomo, da je tudi točko z potrebno odstraniti. Najprej odstranimo točke iz zaporedja $v_1, v_2, v_3, \ldots, v_q$. Tako dobimo množico $\mathcal{D}$. Ker je $z \in \mathcal{D} \setminus \mathcal{C}$, se pojavi v zaporedju $u_1, u_2, u_3, \ldots, u_s = z$. Naj bo $\mathcal{U}_0 = \emptyset$ in $\mathcal{U}_i = \mathcal{U}_{i-1} \cup \{u_i\}$. Ker je $p(u_i, \mathcal{V} \setminus \mathcal{U}_{i-1}) < t$ za vsak $i = 1, \ldots, r$, velja zaradi monotonosti funkcije p tudi $p(u_i, (\mathcal{V} \setminus \mathcal{D}) \setminus \mathcal{U}_{i-1}) < t$ za vsak $i = 1, \ldots, r$. Torej lahko odstranimo tudi vse točke $u_i \in \mathcal{D} \setminus \mathcal{C}$, kar pomeni, da je $\mathcal{D} \setminus \mathcal{C} = \emptyset$, to pa je protislovje, ki smo ga želeli pokazati.

Ker je rezultat postopka enolično določen in ker je vrednost funkcije p na točkah izven $\mathcal{C}$ manjša od t, je izpolnjen tudi drugi pogoj iz definicije p-sredice. Dobljena množica $\mathcal{C}$ je torej p-sredica na nivoju t. □

Posledica 4.2: *Za monotono točkovno funkcijo p so p-sredice gnezdene*

$$t_1 < t_2 \Longrightarrow \mathcal{H}_{t_2} \subseteq \mathcal{H}_{t_1}$$

Dokaz: Dokaz sledi neposredno iz trditve 4.1. Ker je rezultat neodvisen od vrstnega reda brisanj, določimo najprej sredico $\mathcal{H}_{t_1}$. Nato odstranimo še nekaj dodatnih točk in tako dobimo sredico $\mathcal{H}_{t_2}$. Torej je $\mathcal{H}_{t_2} \subseteq \mathcal{H}_{t_1}$. □

Podobno, kot v primeru navadnih sredic, lahko sklepamo, da je za monotono točkovno funkcijo p množica povezanih komponent posameznih slojev p-sredic hierarhija. Povezani komponenti, ki sta obe v istem sloju, sta ločeni, če pa sta v dveh različnih slojih, sta ločeni, ali pa je komponenta višjega sloja vsebovana v komponenti nižjega sloja.

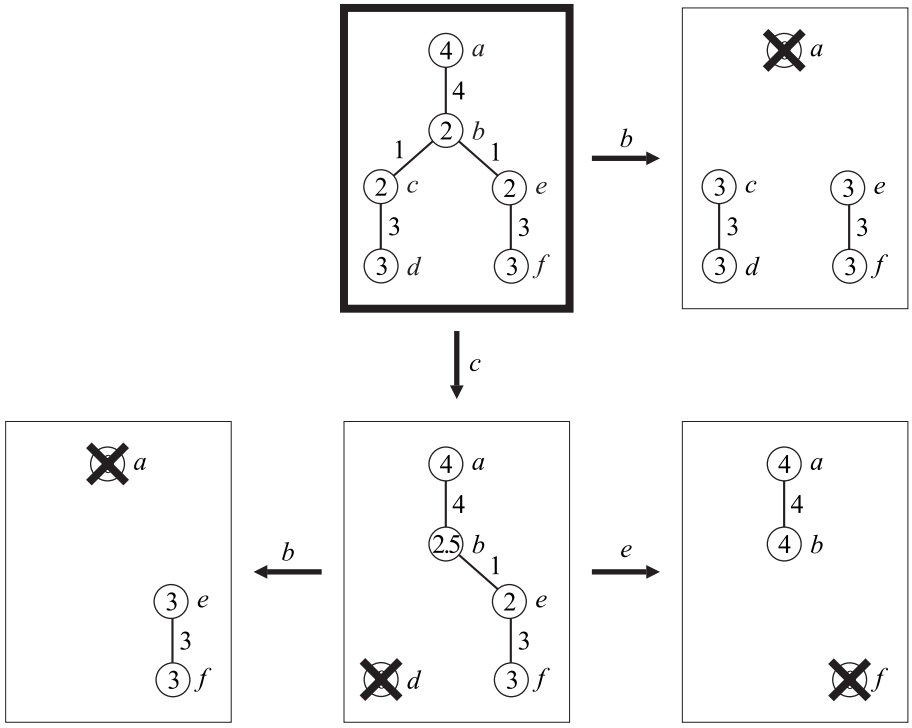
Poglejmo si še primer nemonotone točkovne funkcije. Takšna je na primer funkcija, ki vrača povprečno utež na povezavah do sosednih točk. Definirana je takole:

$$p(v,\mathcal{C}) = \begin{cases} \dfrac{1}{|N(v,\mathcal{C})|} \displaystyle\sum_{u \in N(v,\mathcal{C})} w((v\,;u)), & N(v,\mathcal{C}) \neq \emptyset \\ 0, & \text{sicer} \end{cases}$$

Pri tem je $w\colon \mathcal{L} \to \mathbb{R}_0^+$ funkcija, ki določa uteži na povezavah danega omrežja. Za testno omrežje izberimo omrežje $\mathcal{N} = (\mathcal{V}, \mathcal{L}, w)$ na množici točk $\mathcal{V} = \{a, b, c, d, e, f\}$, kjer so povezave in njihove uteži podane s tabelo:

$\mathcal{L}$	$(a\,;b)$	$(b\,;c)$	$(c\,;d)$	$(b\,;e)$	$(e\,;f)$
w	4	1	3	1	3

Če najprej odstranimo točko b, dobimo drugačen rezultat, kot če bi najprej odstranili točko c (ali točko e). Dobljeni rezultati so prikazani na sliki 4.5.



Slika 4.5: *Nemonotona točkovna funkcija*

Začetno omrežje je p-sredica na nivoju 2. Če želimo dobiti p-sredico na nivoju 3, imamo tri možnosti pri izbiri točke, ki jo bomo pobrisali kot prvo: b, c ali e. Če se odločimo za točko b, dobimo potem, ko odstranimo še izolirano točko a, p-sredico $\mathcal{C}_1 = \{c, d, e, f\}$ na nivoju 3. Opazimo lahko, da je vrednost funkcije p v točkah c in e narasla z 2 na 3.

Postopek 4.3 *Določanje p-sredice na nivoju t*

Podatki: omrežje $\mathcal{N} = (\mathcal{V}, \mathcal{L}, p)$, predstavljeno s seznami sosedov in $t \in \mathbb{R}$
Rezultat: množica $\mathcal{C} \subseteq \mathcal{V}$, $\mathcal{C}$ je p-sredica na nivoju t

```
C := V
for each v ∈ V do p[v] := p(v, N(v))
sestavi min-kopico heap iz točk V glede na funkcijo p
while |heap| > 0 ∧ p[u := heap.top] < t do begin
    C := C \ {u}
    odstrani u iz kopice heap
    for each v ∈ N(u, C) do begin
        p[v] := p(v, N(v, C))
        popravi kopico heap (dvigni element v)
    end
end
```

Če pričnemo z brisanjem točke c, dobimo množico $\{a, b, e, f\}$ na nivoju 2. Pri tem se vrednost točke b poveča na 2.5. Nadaljujemo lahko z brisanjem točke b, kar nam da p-sredico $\mathcal{C}_2 = \{e, f\}$ na nivoju 3, ali pa z brisanjem točke e, pri čemer dobimo p-sredico $\mathcal{C}_3 = \{a, b\}$ na nivoju 4.

Kot vidimo, je rezultat postopka odvisen od vrstnega reda brisanj. Tudi gnezdenosti sredic ni: p-sredica na nivoju 4 ni vsebovana v p-sredici na nivoju 3.

4.6 Postopki za določanje posplošenih sredic

Definicija 4.5: Točkovna funcija p je *lokalna* natanko tedaj, ko

$$p(v, \mathcal{C}) = p(v, N(v, \mathcal{C}))$$

Točkovne funkcije p_1 – p_6 so vse lokalne, funkcija p_7 pa ni lokalna za $k \geq 4$. V nadaljevanju bomo predpostavili, da za točkovno funkcijo p obstaja konstanta p_0, tako da je

$$\forall v \in \mathcal{V} : p(v, \emptyset) = p_0$$

Za lokalno monotono točkovno funkcijo p obstaja postopek za določitev p-sredice na nivoju t, čigar časovna zahtevnost je $\mathcal{O}(m \cdot \max\{\Delta, \log n\})$, glej postopek 4.3. Pri tem smo predpostavili, da lahko vrednost $p(v, N(v, \mathcal{C}))$ izračunamo v času $\mathcal{O}(\deg(v, \mathcal{C}))$.

Stavek, kjer izračunamo novo vrednost $p[v]$ lahko pogosto pospešimo, tako da to vrednost samo dopolnimo, ne da bi jo šli računati od začetka. Opisani postopek lahko preprosto popravimo tako, da nam vrne hierarhijo p-sredic, glej postopek 4.4. Hierarhija je določena s središčnimi števili, ki pripadajo posameznim točkam (nivo najvišje p-sredice, ki še vsebuje izbrano točko).

Postopek 4.4 *Določanje hierarhije p-sredic*

Podatki: omrežje $\mathcal{N} = (\mathcal{V}, \mathcal{L}, p)$, predstavljeno s seznami sosedov
Rezultat: tabela *core* s središčnimi števili posameznih točk

$\mathcal{C} := \mathcal{V}$
for each $v \in \mathcal{V}$ **do** p$[v] := p(v, N(v))$
sestavi min-kopico *heap* iz točk $\mathcal{V}$ glede na funkcijo p
while $|heap| > 0$ **do begin**
  $u := heap.top$
  $\mathcal{C} := \mathcal{C} \setminus \{u\}$
  odstrani u iz kopice *heap*
  core$[u]$:= p$[u]$
  for each $v \in N(u, \mathcal{C})$ **do begin**
    p$[v] := \max\{\text{p}[u], p(v, N(v, \mathcal{C}))\}$
    popravi kopico *heap* (dvigni element v)
  end
end

Predpostavimo, da je P največji čas, ki ga potrebujemo za izračun vrednosti $p(v, \mathcal{C})$, kjer je $v \in \mathcal{V}$ in $\mathcal{C} \subseteq \mathcal{V}$. Potem je časovna zahtevnost prvih treh korakov postopka 4.3 enaka $\mathcal{O}(n) + \mathcal{O}(Pn) + \mathcal{O}(n \log n) = \mathcal{O}(n \cdot \max\{P, \log n\})$. Poglejmo si še telo zanke **while**. Ker se velikost množice $\mathcal{C}$ na vsakem koraku zanke zmanjša za 1, imamo največ n ponovitev zanke. Prva dva stavka v tej zanki lahko izvršimo v konstantnem času, torej je njun prispevek v postopku enak $\mathcal{O}(n)$. V vseh ponovitvah zanke **for** znotraj zanke **while** obravnavamo vsako povezavo grafa največ enkrat. Torej se zanka **for** izvede največ m-krat, za kar potrebujemo $m \cdot (P + \mathcal{O}(\log n))$ časa. Če seštejemo vse skupaj, dobimo časovno zahtevnost celotnega postopka, ki je enaka $\mathcal{O}(m \cdot \max\{P, \log n\})$.

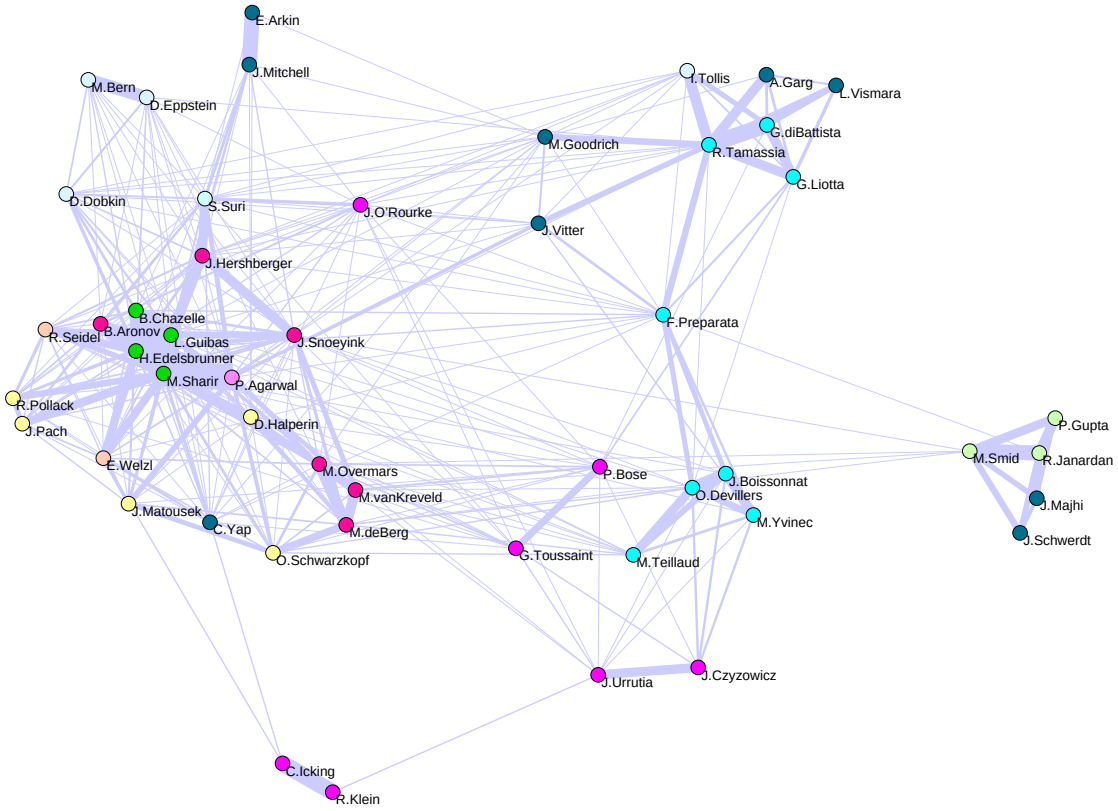
Za lokalno točkovno funkcijo p, katere vrednost $p(v, N(v, \mathcal{C}))$ lahko izračunamo v času $\mathcal{O}(\deg(v, \mathcal{C}))$, je $P = \mathcal{O}(\Delta)$. V primerih, ko pa lahko vrednost funcije popravimo brez ponovnega izračunavanja, pa je $P = \mathcal{O}(1)$.

Za točkovne funkcije p_1 – p_4 postopek vrne isti rezultat kot postopek, opisan na začetku poglavja, a se bolj splača uporabljati prejšnjega, ker je hitrejši (linearen v številu povezav).

4.7 Primer

Tudi delovanje tega postopka si poglejmo na omrežju avtorjev `geombib` [25]. Prej smo vrednosti na povezavah ignorirali, zdaj pa bomo te vrednosti uporabili pri določanju posplošenih sredic. Spomnimo se, da je p_5 funkcija, ki točki predpiše vsote uteži na sosednih povezavah. Utež povezave pove, koliko skupnih člankov

imata dva avtorja. Prej nas je zanimalo število soavtorjev, zdaj pa število člankov v soavtorstvu. Slika 4.6 prikazuje p_5-sredico na nivoju 46.



Slika 4.6: *Računska geometrija: p_5-sredica na nivoju 46*

Peto poglavje

Povezanost s kratkimi cikli

Povezanost s kratkimi cikli je posplošitev običajne povezanosti. Namesto s potjo (zaporedje povezav) morata biti dve točki povezani z zaporedjem kratkih ciklov, v katerem imata dva zaporedna cikla vsaj eno skupno točko. Če imajo vsi sosedni cikli v zaporedju vsaj eno skupno povezavo, govorimo o povezavni povezanosti s kratkimi cikli [10].

5.1 Trikotniška povezanost

5.1.1 Neusmerjeni grafi

Naj bo $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ enostaven neusmerjen graf.

Točka $u \in \mathcal{V}$ je v relaciji $\mathbf{K}$ s točko $v \in \mathcal{V}$, če je $u = v$ ali obstaja pot v $\mathcal{G}$ od u do v. Relaciji $\mathbf{K}$ pravimo *relacija povezanosti*.

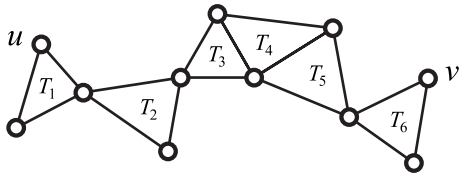
Točka $u \in \mathcal{V}$ je v relaciji $\mathbf{B}$ s točko $v \in \mathcal{V}$, če je $u = v$ ali obstaja cikel v $\mathcal{G}$, ki vsebuje u in v. Relaciji $\mathbf{B}$ pravimo *relacija dvopovezanosti*.

Podgrafu, ki je izomorfen 3-ciklu C_3, bomo rekli *trikotnik*. Podgraf $\mathcal{H}$ grafa $\mathcal{G}$ je *trikotniški*, če vsaka njegova točka in vsaka njegova povezava pripada vsaj enemu trikotniku v $\mathcal{H}$.

Definicija 5.1: Zaporedje $(\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_s)$ trikotnikov v $\mathcal{G}$ (*točkovno*) *trikotniško povezuje* točko $u \in \mathcal{V}$ s točko $v \in \mathcal{V}$, če je

1. $u \in \mathcal{V}(\mathcal{T}_1)$,
2. $v \in \mathcal{V}(\mathcal{T}_s)$ in
3. $\mathcal{V}(\mathcal{T}_{i-1}) \cap \mathcal{V}(\mathcal{T}_i) \neq \emptyset \quad$ za $\; i = 2, \ldots, s$.

Takemu zaporedju pravimo (*točkovna*) *trikotniška veriga*, glej sliko 5.1.



Slika 5.1: *Trikotniška veriga*

Definicija 5.2: Točka $u \in \mathcal{V}$ je (*točkovno*) *trikotniško povezana* s točko $v \in \mathcal{V}$, $u\mathbf{K}_3 v$, če je $u = v$ ali obstaja (točkovna) trikotniška veriga, ki (točkovno) trikotniško povezuje točko u s točko v.

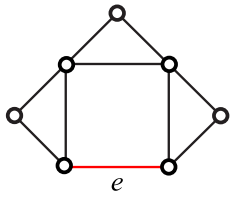
Trditev 5.1: *Relacija* $\mathbf{K}_3$ *je ekvivalenčna relacija na množici točk* $\mathcal{V}$.

Ker bomo kasneje to in večino drugih trditev iz tega razdelka posplošili, jih tukaj navajamo brez dokazov.

Ekvivalenčna relacija razbije množico točk v enega ali več ekvivalenčnih razredov, oziroma *komponent*. Komponenta je *trivialna*, če je sestavljena iz ene same točke.

Trditev 5.2: *Množice točk maksimalnih povezanih trikotniških podgrafov so natanko netrivialne komponente (točkovne) trikotniške povezanosti.*

Toda podgrafi, ki so porojeni z netrivialnimi komponentami (točkovne) trikotniške povezanosti niso nujno trikotniški podgrafi, torej tudi ne maksimalni povezani trikotniški podgrafi. To lahko vidimo na sliki 5.2, kjer vse točke grafa pripadajo isti komponenti trikotniške povezanosti, graf pa ni trikotniški zaradi povezave e, ki ne pripada nobenemu trikotniku.



Slika 5.2: *Graf ni trikotniški*

Postopek za določitev relacije $\mathbf{K}_3$ je preprost, glej postopek 5.1. Množico točk razbije na k skupin (ekvivalenčnih razredov), ki jih označimo s $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_k$.

Najprej izberemo poljubno točko $u \in \mathcal{V}$ in jo damo v novo množico, ki bo na koncu eden od ekvivalenčnih razredov. Potem po vrsti v množico dodajamo vse tiste točke, ki so iz točke u dosegljive po trikotnikih. Postopek ponavljamo tako dolgo, da so vse točke v svojih ekvivalenčnih razredih.

Če so množice sosedov urejene, lahko uporabimo prilagojeno različico zlivanja in tako izračunamo $N(u) \cap N(v)$ v času $\mathcal{O}(\Delta)$. V tem primeru je časovna zahtevnost postopka enaka $\mathcal{O}(\Delta m)$, saj moramo pregledati vse točke, pri obdelavi točke u pa moramo obiskati vse njene sosede in za vsakega soseda v poiskati presek $N(u) \cap N(v)$.

Postopek 5.1 *Ekvivalenčni razredi relacije* $\mathbf{K}_3$

Podatki: graf $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
Rezultat: komponente $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_k$ relacije $\mathbf{K}_3$

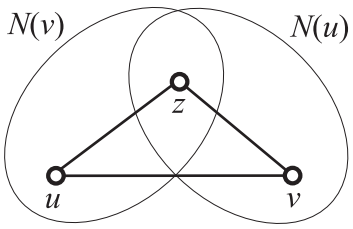
```
k := 0
while V ≠ ∅ do begin
    izberi u ∈ V
    k := k + 1
    C_k := ∅
    L := {u}
    while L ≠ ∅ do begin
        izberi u ∈ L
        C_k := C_k ∪ {u}
        for each v ∈ N(u) do begin
            N := N(u) ∩ N(v)
            if N ≠ ∅ then L := L ∪ N ∪ {v}
        end
        V := V \ {u}
        L := L \ {u}
    end
end
```

Definicija 5.3: *Trikotniško omrežje* $\mathcal{N}_3(\mathcal{G}) = (\mathcal{V}, \mathcal{E}_3, w_3)$, ki ga določa graf $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, je podgraf $\mathcal{G}_3 = (\mathcal{V}, \mathcal{E}_3)$ grafa $\mathcal{G}$, katerega povezave so tiste povezave grafa $\mathcal{G}$, ki pripadajo kakemu trikotniku: $e \in \mathcal{E}_3$, če $e \in \mathcal{E}$ in e pripada trikotniku. Utež $w_3(e)$ povezave $e \in \mathcal{E}_3$ je število različnih trikotnikov v grafu $\mathcal{G}$, na katerih leži povezava e.

Trditev 5.3:

$$\mathbf{K}_3(\mathcal{G}) = \mathbf{K}(\mathcal{G}_3)$$

Postopek za določitev $\mathcal{E}_3$ in w_3 je preprost, glej postopek 5.2 in sliko 5.3. Če so množice sosedov urejene, je računanje uteži $w_3(e)$ časovne zahtevnosti $\mathcal{O}(\Delta)$, ker pa moramo utež naračunati za vsako povezavo posebej, je skupna časovna zahtevnost postopka enaka $\mathcal{O}(\Delta m)$.



Slika 5.3: $w_3(e) := |N(u) \cap N(v)|$

Postopek 5.2 *Trikotniško omrežje*

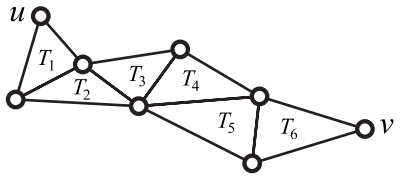
Podatki: graf $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
Rezultat: trikotniško omrežje $\mathcal{N}_3(\mathcal{G}) = (\mathcal{V}, \mathcal{E}_3, w_3)$

$\mathcal{E}_3 := \emptyset$
for each $e(u:v) \in \mathcal{E}$ **do begin**
  $w_3(e) := |N(u) \cap N(v)|$
  if $w_3(e) > 0$ **then** $\mathcal{E}_3 := \mathcal{E}_3 \cup \{e\}$
end

Definicija 5.4: Zaporedje $(\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_s)$ trikotnikov v $\mathcal{G}$ *povezavno trikotniško povezuje* točko $u \in \mathcal{V}$ s točko $v \in \mathcal{V}$, če je

1. $u \in \mathcal{V}(\mathcal{T}_1)$,
2. $v \in \mathcal{V}(\mathcal{T}_s)$ in
3. $\mathcal{E}(\mathcal{T}_{i-1}) \cap \mathcal{E}(\mathcal{T}_i) \neq \emptyset$ za $i = 2, \ldots, s$.

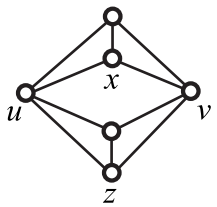
Takemu zaporedju pravimo *povezavna trikotniška veriga*, glej sliko 5.4.



Slika 5.4: *Povezavna trikotniška veriga*

Definicija 5.5: Točka $u \in \mathcal{V}$ je *povezavno trikotniško povezana* s točko $v \in \mathcal{V}$, $u\mathbf{L}_3 v$, če je $u = v$ ali obstaja povezavna trikotniška veriga, ki povezavno trikotniško povezuje točko u s točko v.

Poglejmo si dvopovezan graf na sliki 5.5. Točki u in v sta povezavno trikotniško povezani, medtem ko točki x in z nista. Relacija $\mathbf{L}_3$ ni tranzitivna: $x\mathbf{L}_3 v$, $v\mathbf{L}_3 z$, točka x pa ni v relaciji $\mathbf{L}_3$ s točko z.



Slika 5.5: *Dvopovezan trikotniški graf*

Trditev 5.4: *Relacija* $\mathbf{L}_3$ *določa ekvivalenčno relacijo na množici povezav* $\mathcal{E}$.

Postopek 5.3 *Ekvivalenčni razredi relacije na $\mathcal{E}$, določene z $\mathbf{L}_3$*

Podatki: graf $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
Rezultat: komponente $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_k$ relacije na $\mathcal{E}$

```
k := 0
while E ≠ ∅ do begin
    izberi e ∈ E
    k := k + 1
    C_k := ∅
    L := {e}
    while L ≠ ∅ do begin
        izberi e(u : v) ∈ L
        C_k := C_k ∪ {e}
        E := E \ {e}
        N := N(u) ∩ N(v)
        L := L ∪ {(u : x), x ∈ N} ∪ {(v : x), x ∈ N}
        L := L \ {e}
    end
end
```

Postopek za določitev relacije $\mathbf{L}_3$ je preprost, glej postopek 5.3. Množico povezav razbije na k skupin (ekvivalenčni razredi relacije na $\mathcal{E}$), ki jih označimo s $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_k$. Točka u je v relaciji $\mathbf{L}_3$ s točko v, če sta obe točki krajišči povezav iz iste skupine.

$$u\mathbf{L}_3 v \iff \exists i\ \exists e, f \in \mathcal{C}_i : u \in \mathcal{V}(e) \wedge v \in \mathcal{V}(f)$$

Z $\mathcal{V}(e)$ smo označili množico krajišč povezave e.

V vsaki ponovitvi notranje zanke iz množice $\mathcal{E}$ odstranimo eno povezavo in jo shranimo v množico $\mathcal{C}_k$. Notranja zanka se torej izvrši natanko m-krat. Vse preiditvene stavke in primerjave lahko opravimo v konstantnem času, razen stavka, kjer določamo presek dveh množic sosednih točk. Če so množice sosedov urejene, ima ta stavek časovno zahtevnost $\mathcal{O}(\Delta)$, ker pa se nahaja v notranji zanki, je časovna zahtevnost celotnega postopka enaka $\mathcal{O}(\Delta m)$.

Opozoriti velja, da v notranji zanki odstranimo povezavo e iz množice povezav $\mathcal{E}$, torej tudi iz grafa. Zato so množice sosedov dinamične – odvisne so od trenutne množice povezav $\mathcal{E}$. Od tod sledi, da potem ko odstranimo povezavo e iz množice $\mathcal{E}$ (in tudi iz $\mathcal{L}$), se ta povezava ne more več pojaviti v $\mathcal{L}$.

Definicija 5.6: Naj bo $\mathbf{B}_3 = \mathbf{B} \cap \mathbf{K}_3$.

Trditev 5.5: *V grafu $\mathcal{G}$ velja:*

a. $\mathbf{B} \subseteq \mathbf{K}$
b. $\mathbf{K}_3 \subseteq \mathbf{K}$
c. $\mathbf{L}_3 \subseteq \mathbf{B}_3$
d. $\mathbf{B}_3 \subseteq \mathbf{B}$
e. $\mathbf{B}_3 \subseteq \mathbf{K}_3$

Definicija 5.7: Naj bo $t(v)$ število različnih trikotnikov grafa $\mathcal{G}$, ki vsebujejo točko v.

Trditev 5.6:

$$2t(v) = \sum_{e:e(v:u)} w_3(e)$$

Dokaz: Vsak trikotnik, ki vsebuje točko v, vsebuje tudi natanko dve povezavi s krajiščem v tej točki. Če torej za vsako povezavo s krajiščem v točki v preštejemo, koliko trikotnikom pripada, smo vsak trikotnik šteli dvakrat. □

Definicija 5.8: *Stopnja nasičenosti* v točki v je definirana kot v [40]:

$$C(v) = \frac{2t(v)}{\deg(v)(\deg(v)-1)}$$

Ker v enostavnem grafu velja $t(v) \leq \binom{\deg(v)}{2}$, je $C(v) \in [0,1]$. Problem s tako definirano stopnjo nasičenosti je, da favorizira točke z majhno stopnjo. To lahko popravimo tako, da $C(v)$ pomnožimo z $\frac{\deg(v)}{\Delta}$.

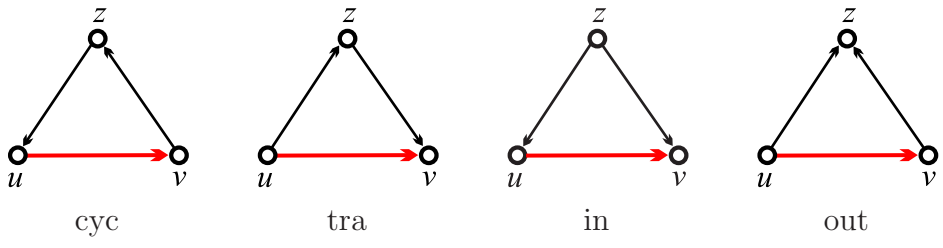
$$C'(v) = \frac{2t(v)}{\Delta(\deg(v)-1)}$$

Še vedno je $0 \leq C'(v) \leq 1$, a samo točke v največji kliki imajo vrednost 1.

5.1.2 Usmerjeni grafi

Če ima graf $\mathcal{G}$ kakšno neusmerjeno povezavo, jo nadomestimo z dvema nasprotno usmerjenima povezava. V nadaljevanju naj bo $\mathcal{G} = (\mathcal{V}, \mathcal{A})$ enostaven usmerjen graf.

Nad izbrano povezavo $a(u,v) \in \mathcal{A}$ obstajajo štirje različne vrste usmerjenih trikotnikov: ciklični (cyc), tranzitivni (tra), vhodni (in) in izhodni (out) trikotniki (glej sliko 5.6). Za vsako vrsto dobimo ustrezno trikotniško omrežje $\mathcal{N}_3^{cyc}$, $\mathcal{N}_3^{tra}$, $\mathcal{N}_3^{in}$ in $\mathcal{N}_3^{out}$.



Slika 5.6: *Vrste usmerjenih trikotnikov nad izbrano povezavo*

V usmerjenih omrežjih razlikujemo med šibko in krepko povezanostjo. Na šibko povezanost lahko gledamo kot na običajno povezanost v skeletu $\mathcal{S} = (\mathcal{V}, \mathcal{E}_S)$ grafa $\mathcal{G}$

$$\mathcal{E}_S = \{(u:v) : u \neq v \wedge (u,v) \in \mathcal{A}\}$$

Definicija 5.9: Podgraf $\mathcal{H}$ grafa $\mathcal{G}$ je *cikl̆no trikotniški*, če vsaka njegova točka in vsaka njegova povezava pripada vsaj enemu cikl̆nemu trikotniku v $\mathcal{H}$.

Trditev 5.7: *Šibko povezan cikl̆no trikotniški graf je tudi krepko povezan.*

Podobno, kot v neusmerjenem primeru, lahko tudi v usmerjenem definiramo več vrst povezanostnih relacij. Če se omejimo na cikl̆ne trikotnike, lahko definiramo (točkovno) cikl̆no trikotniško povezanost $\mathbf{C}_3$ in povezavno cikl̆no trikotniško povezanost $\mathbf{D}_3$.

Definicija 5.10: Zaporedje $(\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_s)$ cikl̆nih trikotnikov v $\mathcal{G}$ (*točkovno*) *cikl̆no trikotniško povezuje* točko $u \in \mathcal{V}$ s točko $v \in \mathcal{V}$, če je

1. $u \in \mathcal{V}(\mathcal{T}_1)$,
2. $v \in \mathcal{V}(\mathcal{T}_s)$ in
3. $\mathcal{V}(\mathcal{T}_{i-1}) \cap \mathcal{V}(\mathcal{T}_i) \neq \emptyset \quad$ za $\; i = 2, \ldots, s$.

Takemu zaporedju pravimo (*točkovna*) *cikl̆na trikotniška veriga*.

Definicija 5.11: Točka $u \in \mathcal{V}$ je (*točkovno*) *cikl̆no trikotniško povezana* s točko $v \in \mathcal{V}$, $u\mathbf{C}_3v$, če je $u = v$ ali obstaja (točkovna) cikl̆na trikotniška veriga, ki (točkovno) cikl̆no trikotniško povezuje točko u s točko v.

Definicija 5.12: Zaporedje $(\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_s)$ cikl̆nih trikotnikov v $\mathcal{G}$ *povezavno cikl̆no trikotniško povezuje* točko $u \in \mathcal{V}$ s točko $v \in \mathcal{V}$, če je

1. $u \in \mathcal{V}(\mathcal{T}_1)$,
2. $v \in \mathcal{V}(\mathcal{T}_s)$ in
3. $\mathcal{A}(\mathcal{T}_{i-1}) \cap \mathcal{A}(\mathcal{T}_i) \neq \emptyset \quad$ za $\; i = 2, \ldots, s$.

Takemu zaporedju pravimo *povezavna cikl̆na trikotniška veriga*.

Definicija 5.13: Točka $u \in \mathcal{V}$ je *povezavno cikl̆no trikotniško povezana* s točko $v \in \mathcal{V}$, $u\mathbf{D}_3v$, če je $u = v$ ali obstaja povezavna cikl̆na trikotniška veriga, ki povezavno cikl̆no trikotniško povezuje točko u s točko v.

Za $\mathbf{C}_3$ in $\mathbf{D}_3$ veljajo podobne lastnosti, kot za $\mathbf{K}_3$ in $\mathbf{L}_3$.

Trditev 5.8: *Relacija $\mathbf{C}_3$ je ekvivalenčna relacija na množici točk $\mathcal{V}$.*

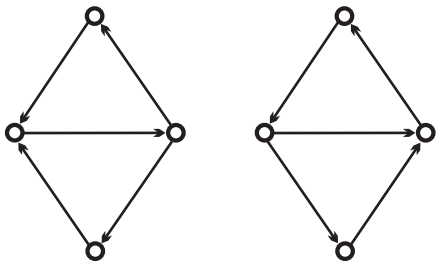
Trditev 5.9: *Relacija $\mathbf{D}_3$ določa ekvivalenčno relacijo na množici povezav $\mathcal{A}$.*

Tudi postopki za določitev relacij in pripadajočih omrežij so podobni, kot v neusmerjenem primeru, torej imajo tudi enako časovno zahtevnost. Na primer, v cikl̆nem trikotniškem omrežju $\mathcal{N}_3^{cyc} = (\mathcal{V}, \mathcal{A}_3^{cyc}, w_3^{cyc})$ za vsako povezavo $a \in \mathcal{A}_3^{cyc}$ velja:

$$w_3^{cyc}(a(u,v)) = |N_{out}(v) \cap N_{in}(u)|$$

Trikotniško povezanost lahko definiramo tudi drugače:

Definicija 5.14: Točka $u \in \mathcal{V}$ je *krepko trikotniško povezana* s točko $v \in \mathcal{V}$, $u\mathbf{S}_3 v$, če je $u = v$ ali obstaja krepko povezan trikotniški podgraf, ki vsebuje točki u in v.



Slika 5.7: *Krepko trikotniško povezana grafa*

Naj bo $\mathbf{R}$ *relacija dosegljivosti* v danem usmerjenem grafu $\mathcal{G}$. Točka $v \in \mathcal{V}$ je dosegljiva iz točke $u \in \mathcal{V}$, $u\mathbf{R}v$, če je $u = v$ ali pa obstaja sprehod od u do v.

Naj bo $\mathbf{S}$ *relacija krepke povezanosti* v danem usmerjenem grafu $\mathcal{G}$. Točki $u \in \mathcal{V}$ in $v \in \mathcal{V}$ sta v relaciji $\mathbf{S}$, če je $u = v$ ali obstajata sprehoda od u do v in od v do u. Očitno je $\mathbf{S} = \mathbf{R} \cap \mathbf{R}^{-1}$.

Trditev 5.10: *V usmerjenem grafu $\mathcal{G}$ velja:*

a. $\mathbf{D}_3 \subseteq \mathbf{C}_3$ *b.* $\mathbf{C}_3 \subseteq \mathbf{S}_3$ *c.* $\mathbf{S}_3 \subseteq \mathbf{S}$

Trditev 5.11:

$$\mathbf{C}_3(\mathcal{G}) = \mathbf{S}(\mathcal{G}_3^{cyc})$$

5.1.3 Tranzitivnost

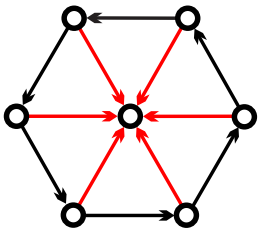
Povezava je *trikotniško tranzitivna*, če je osnova nekega tranzitivnega trikotnika. Tranzitivne povezave so pravzaprav *bližnjice* – če nekatere od njih zbrišemo, lahko podatki še vedno potujejo po *podpornih* povezavah. Uporabimo jih lahko tudi za preverjanje pravilnosti prenosa.

Trditev 5.12: *Če iz grafa $\mathcal{G} = (\mathcal{V}, \mathcal{A})$ zbrišemo nekaj (ali vse) povezave, ki pripadajo trikotniško tranzitivni poti π (vsaka povezava na poti π je trikotniško tranzitivna), se relacija dosegljivosti ne spremeni: $\mathbf{R}(\mathcal{G}) = \mathbf{R}(\mathcal{G} \setminus \mathcal{A}(\pi))$.*

Dokaz: Ker je graf $\mathcal{G} \setminus \mathcal{A}(\pi)$ podgraf grafa $\mathcal{G}$, je očitno $\mathbf{R}(\mathcal{G} \setminus \mathcal{A}(\pi)) \subseteq \mathbf{R}(\mathcal{G})$. Naj bo povezava a poljubna povezava na trikotniško tranzitivni poti π. Ker je trikotniško tranzitivna, lahko od enega do drugega krajišča pridemo tudi po dveh podpornih povezavah, torej lahko povezavo a odstranimo, ne da bi s tem prekinili katerega

od sprehodov po grafu (sprehod lahko kvečjemu postane za eno povezavo daljši). Preveriti moramo še, da podporni povezavi povezave a ne moreta biti del poti π, torej da ju ne bomo zbrisali. Ker imata povezava a in podporna povezava skupno točko, bi ti dve povezavi na poti π lahko bili kvečjemu zaporedni povezavi. To pa se zaradi usmerjenosti teh dveh povezav ne more zgoditi. Če je torej v prvotnem grafu $\mathcal{G}$ obstajal sprehod med dvema točkama, potem tudi v grafu $\mathcal{G} \setminus \mathcal{A}(\pi)$ sprehod med tema dvema točkama obstaja, le da je morda daljši (kvečjemu za toliko povezav, kot smo jih odstranili). Tako smo dokazali še obratno: $\mathbf{R}(\mathcal{G}) \subseteq \mathbf{R}(\mathcal{G} \setminus \mathcal{A}(\pi))$. □

Toda vseh trikotniško tranzitivnih povezav v grafu ne moremo vedno odstraniti. Na sliki 5.8 je prikazan primer grafa, kjer se relacija dosegljivosti spremeni, če odstranimo vse trikotniško tranzitivne povezave. Graf je sestavljen iz usmerjenega 6-cikla, čigar točke so povezane z dodatno točko v sredini. Ta točka v sredini je torej dosegljiva iz katerekoli druge točke. Vse povezave iz točk na robu do točke v sredini so trikotniško tranzitivne. Če jih odstranimo, točka v sredini postane nedosegljiva iz drugih točk.



Slika 5.8: *Vseh trikotniško tranzitivnih povezav ne moremo odstraniti*

5.2 k-kotniška povezanost

Ker je pojem k-ciklična povezanost že uporabljen v teoriji grafov, bomo nadaljevali z 'geometrijsko' terminologijo (trikotnik, k-kotnik).

5.2.1 Neusmerjeni grafi

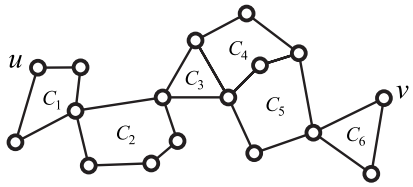
Naj bo $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ enostaven neusmerjen graf.

Podgrafu, ki je izomorfen k-ciklu C_k, bomo rekli *k-kotnik*, podgrafu, ki je izomorfen enemu od s-ciklov C_s, $3 \leq s \leq k$ pa *(k)-kotnik*. Podgraf $\mathcal{H}$ grafa $\mathcal{G}$ je *k-kotniški*, če vsaka njegova točka in vsaka njegova povezava pripada vsaj enemu (k)-kotniku v $\mathcal{H}$.

Definicija 5.15: Zaporedje $(\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_s)$ (k)-kotnikov v $\mathcal{G}$ *(točkovno) k-kotniško povezuje* točko $u \in \mathcal{V}$ s točko $v \in \mathcal{V}$, če je

1. $u \in \mathcal{V}(\mathcal{C}_1)$,
2. $v \in \mathcal{V}(\mathcal{C}_s)$ in
3. $\mathcal{V}(\mathcal{C}_{i-1}) \cap \mathcal{V}(\mathcal{C}_i) \neq \emptyset$ za $i = 2, \ldots, s$.

Takemu zaporedju pravimo *(točkovna) k-kotniška veriga*, glej sliko 5.9.



Slika 5.9: *k-kotniška veriga*

Definicija 5.16: Točka $u \in \mathcal{V}$ je *(točkovno) k-kotniško povezana* s točko $v \in \mathcal{V}$, $u\mathbf{K}_k v$, če je $u = v$ ali obstaja (točkovna) k-kotniška veriga, ki (točkovno) k-kotniško povezuje točko u s točko v.

Trditev 5.13: *Relacija $\mathbf{K}_k$ je ekvivalenčna relacija na množici točk $\mathcal{V}$.*

Dokaz: Refleksivnost sledi iz definicije relacije $\mathbf{K}_k$.

Če k-kotniško verigo od u do v obrnemo, dobimo k-kotniško verigo od v do u, torej je relacija $\mathbf{K}_k$ simetrična.

Še tranzitivnost. Naj bodo u, v in z takšne točke, da je $u\mathbf{K}_k v$ in $v\mathbf{K}_k z$. Če točke niso paroma različne, je pogoj za tranzitivnost trivialno izpolnjen. Predpostavimo torej, da so točke u, v in z paroma različne. Ker je $u\mathbf{K}_k v$, obstaja k-kotniška veriga od u do v, ker pa je $v\mathbf{K}_k z$, obstaja tudi k-kotniška veriga od v do z. Če ti dve verigi staknemo, dobimo k-kotniško verigo od u do z, torej je $u\mathbf{K}_k z$. □

Trditev 5.14: *Množice točk maksimalnih povezanih k-kotniških podgrafov so natanko netrivialne komponente (točkovne) k-kotniške povezanosti.*

Dokaz: Naj bosta u in v poljubni točki, ki pripadata povezanemu k-kotniškemu podgrafu. Če sta točki enaki, je očitno $u\mathbf{K}_k v$. Sicer pa obstaja pot $\pi = u, e_1, z_1, e_2, z_2, e_3, z_3, \ldots, e_s, v$ od u do v. Ker je podgraf k-kotniški, vsaka povezava e_i na tej poti pripada vsaj enemu (k)-kotniku $\mathcal{C}_i$ v tem podgrafu. Za dobljeno k-kotniško verigo $(\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_s)$ velja:

- $e_i \in \mathcal{E}(\mathcal{C}_i)$ za $i = 1, \ldots, s$
- $u \in \mathcal{V}(\mathcal{C}_1)$, $v \in \mathcal{V}(\mathcal{C}_s)$
- $z_{i-1} \in \mathcal{V}(\mathcal{C}_{i-1}) \cap \mathcal{V}(\mathcal{C}_i)$ za $i = 2, \ldots, s$

Zato je $u\mathbf{K}_k v$, torej vse točke poljubnega (tudi maksimalnega) povezanega k-kotniškega podgrafa pripadajo isti komponenti relacije $\mathbf{K}_k$.

Zdaj naj bosta u in v dve različni točki netrivialne $\mathbf{K}_k$-komponente $\mathcal{C} \subseteq \mathcal{V}$. Ker je u v relaciji $\mathbf{K}_k$ s točko v, obstaja k-kotniška veriga od u do v. Očitno je, da vse točke k-kotniške verige pripadajo istemu maksimalnemu povezanemu k-kotniškemu podgrafu, torej tudi točki u in v. Ker pa sta bili točki u in v poljubni točki iz $\mathcal{C}$, vse točke netrivialne komponente k-kotniške povezanosti pripadajo istemu maksimalnemu povezanemu k-kotniškemu podgrafu. □

Toda podgrafi, ki so porojeni z netrivialnimi $\mathbf{K}_k$-komponentami niso nujno k-kotniški podgrafi, torej tudi ne maksimalni povezani k-kotniški podgrafi. Glej primer, prikazan na sliki 5.2.

Definicija 5.17: *k-kotniško omrežje* $\mathcal{N}_k(\mathcal{G}) = (\mathcal{V}, \mathcal{E}_k, w_k)$, ki ga določa graf $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, je podgraf $\mathcal{G}_k = (\mathcal{V}, \mathcal{E}_k)$ grafa $\mathcal{G}$, katerega povezave so tiste povezave grafa $\mathcal{G}$, ki pripadajo vsaj enemu (k)-kotniku: $e \in \mathcal{E}_k$, če $e \in \mathcal{E}$ in e pripada (k)-kotniku. Utež $w_k(e)$ povezave $e \in \mathcal{E}_k$ je število različnih (k)-kotnikov v grafu $\mathcal{G}$, na katerih leži povezava e.

Trditev 5.15:

$$\mathbf{K}_k(\mathcal{G}) = \mathbf{K}(\mathcal{G}_k)$$

DOKAZ: Naj bo $u\mathbf{K}_k v$ v grafu $\mathcal{G}$. Če je $u = v$, potem je tudi $u\mathbf{K}v$ v grafu $\mathcal{G}_k$. Če pa sta točki različni, obstaja (točkovna) k-kotniška veriga v $\mathcal{G}$ od u do v. Vse povezave te verige pripadajo vsaj enemu (k)-kotniku, zato cela veriga leži tudi v grafu $\mathcal{G}_k$. To pomeni, da sta točki u in v v grafu $\mathcal{G}_k$ povezani, torej je $u\mathbf{K}v$ v $\mathcal{G}_k$.

Še obratno. Naj bo $u\mathbf{K}v$ v grafu $\mathcal{G}_k$. To pomeni, da v $\mathcal{G}_k$ obstaja pot od u do v. Ker je $\mathcal{G}_k$ k-kotniški graf, vsaka povezava na tej poti pripada vsaj enemu (k)-kotniku, torej lahko sestavimo k-kotniško verigo od u do v v $\mathcal{G}_k$. Ker je $\mathcal{G}_k$ podgraf grafa $\mathcal{G}$, je ta veriga tudi v $\mathcal{G}$, kar pomeni, da je $u\mathbf{K}_k v$ v grafu $\mathcal{G}$. □

Da bi določili ekvivalenčne razrede relacije $\mathbf{K}_k$, lahko torej najprej določimo pripadajoči k-kotniški podgraf $\mathcal{G}_k$, nato pa v njem poiščemo povezane komponente. Da bi določili tudi uteži w_k posameznih povezav, pa bi za vsako povezavo morali prešteti, koliko (k)-kotnikom pripada. Učinkovit postopek za določanje uteži je še v razvoju.

Uteži w_k lahko uporabimo pri določanju gostih delov danega omrežja. Na primer, za izbrano povezavo e v r-kliki lahko preštejemo, na koliko k-kotnikih leži, $k \leq r$. Krajišči povezave e sta prvi dve točki k-kotnika. Tretjo točko lahko izberemo na $r-2$ načinov, četrto na $r-3$ načinov, ..., zadnjo točko (ki je povezana s prvo) pa na $r-k+1$ načinov. Torej imamo $(r-2)(r-3)\cdots(r-k+1)$ različnih k-kotnikov in $\sum_{i=3}^{k}(r-2)(r-3)\cdots(r-i+1)$ različnih (k)-kotnikov. Od tod dobimo oceno za velikost uteži w_k na povezavi e, ki pripada r-kliki.

$$w_k(e) \geq \sum_{i=3}^{k}(r-2)(r-3)\cdots(r-i+1)$$

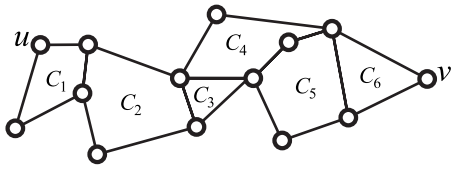
Everett [16] je k-razbitje danega neusmerjenega grafa $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ definiral kot razbitje $\{\mathcal{C}_1, \ldots, \mathcal{C}_p, \mathcal{B}_1, \ldots, \mathcal{B}_q\}$ množice točk $\mathcal{V}$, kjer so $\mathcal{C}_i$ k-kotniške povezane komponente in $\mathcal{B}_j$ mostovi – povezane komponente množice točk $\mathcal{V} \setminus \cup C_i$.

Postopek za določitev Everettovega razbitja je naslednji: najprej določimo k-kotniško omrežje $\mathcal{N}_k(\mathcal{G})$, potem v njem poiščemo povezane komponente $\{\mathcal{C}_i\}$, na koncu pa še povezane komponente $\{\mathcal{B}_i\}$ v preostanku $\mathcal{V} \setminus \cup \mathcal{C}_i$.

Definicija 5.18: Zaporedje $(\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_s)$ (k)-kotnikov v $\mathcal{G}$ *povezavno k-kotniško povezuje* točko $u \in \mathcal{V}$ s točko $v \in \mathcal{V}$, če je

1. $u \in \mathcal{V}(\mathcal{C}_1)$,
2. $v \in \mathcal{V}(\mathcal{C}_s)$ in
3. $\mathcal{E}(\mathcal{C}_{i-1}) \cap \mathcal{E}(\mathcal{C}_i) \neq \emptyset$ za $i = 2, \ldots, s$.

Takemu zaporedju pravimo *povezavna k-kotniška veriga*, glej sliko 5.10.



Slika 5.10: *Povezavna k-kotniška veriga*

Definicija 5.19: Točka $u \in \mathcal{V}$ je *povezavno k-kotniško povezana* s točko $v \in \mathcal{V}$, $u\mathbf{L}_k v$, če je $u = v$ ali obstaja povezavna k-kotniška veriga, ki povezavno k-kotniško povezuje točko u s točko v.

Trditev 5.16: *Relacija $\mathbf{L}_k$ določa ekvivalenčno relacijo na množici povezav $\mathcal{E}$.*

Dokaz: Naj bo $\sim$ relacija na $\mathcal{E}$, ki je definirana takole: $e \sim f$, če je $e = f$ ali pa obstaja povezavna k-kotniška veriga $(\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_s)$, kjer je $e \in \mathcal{E}(\mathcal{C}_1)$ in $f \in \mathcal{E}(\mathcal{C}_s)$.

Refleksivnost relacije $\sim$ sledi iz njene definicije.

Tudi simetričnost je preprosta. Naj bo $e \sim f$. Potem obstaja povezavna k-kotniška veriga 'od' e 'do' f. Če jo obrnemo, dobimo povezavno k-kotniško verigo 'od' f 'do' e, torej je $f \sim e$.

Še tranzitivnost. Naj bodo e, f in g takšne povezave, da je $e \sim f$ in $f \sim g$. Potem obstajata povezavni k-kotniški verigi 'od' e 'do' f in 'od' f 'do' g. Če ti dve verigi staknemo, dobimo povezavno k-kotniško verigo 'od' e 'do' g (oba (k)-kotnika na stiku verig vsebujeta povezavo f, torej njun presek ni prazen). Zato je tudi $e \sim g$. □

Definicija 5.20: Naj bo $\mathbf{B}_k = \mathbf{B} \cap \mathbf{K}_k$.

Trditev 5.17: *V grafu $\mathcal{G}$ velja:*

a. $\mathbf{B} \subseteq \mathbf{K}$
b. $\mathbf{K}_k \subseteq \mathbf{K}$
c. $\mathbf{L}_k \subseteq \mathbf{B}_k$
d. $\mathbf{B}_k \subseteq \mathbf{B}$
e. $\mathbf{B}_k \subseteq \mathbf{K}_k$

za $i < j$ pa velja tudi:

f. $\mathbf{K}_i \subseteq \mathbf{K}_j$
g. $\mathbf{L}_i \subseteq \mathbf{L}_j$
h. $\mathbf{B}_i \subseteq \mathbf{B}_j$

DOKAZ:

a. Sledi iz definicij relacij $\mathbf{B}$ in $\mathbf{K}$.

b. Naj bosta u in v takšni točki, da je $u\mathbf{K}_k v$. Če je $u = v$, potem je po definiciji tudi $u\mathbf{K}v$. Sicer pa obstaja k-kotniška veriga od u do v, torej obstaja tudi pot od u do v, kar pomeni, da je tudi v tem primeru $u\mathbf{K}v$.

c. Naj bosta u in v takšni točki, da je $u\mathbf{L}_k v$. Če je $u = v$, je po definiciji tudi $u\mathbf{B}v$ in $u\mathbf{K}_k v$, od koder sledi, da je $u\mathbf{B}_k v$. Če pa sta točki različni, obstaja povezavna k-kotniška veriga od u do v, vsaka povezavna k-kotniška veriga pa je tudi točkovna k-kotniška veriga (če imata dva k-kotnika skupno povezavo, imata tudi skupno točko). Zato je $u\mathbf{K}_k v$. Podgraf, ki ga določa povezavna k-kotniška veriga, je dvopovezan [15], torej obstaja cikel, ki vsebuje u in v. Zato je tudi $u\mathbf{B}v$. Oboje skupaj pa nam po definiciji da $u\mathbf{B}_k v$.

d. Sledi iz definicije relacije $\mathbf{B}_k$.

e. Sledi iz definicije relacije $\mathbf{B}_k$.

f. Naj bosta u in v taki točki, da je $u\mathbf{K}_i v$. Če je $u = v$, potem je po definiciji tudi $u\mathbf{K}_j v$. Sicer pa obstaja i-kotniška veriga od u do v, kjer noben (i)-kotnik ni daljši od i. Ta veriga je tudi j-kotniška veriga od u do v, torej je $u\mathbf{K}_j v$.

g. Naj bosta u in v taki točki, da je $u\mathbf{L}_i v$. Če je $u = v$, potem je po definiciji tudi $u\mathbf{L}_j v$. Sicer pa obstaja povezavna i-kotniška veriga od u do v, kjer noben (i)-kotnik ni daljši od i. Ta veriga je tudi povezavna j-kotniška veriga od u do v, torej je $u\mathbf{L}_j v$.

h. Sledi iz definicije relacije $\mathbf{B}_k$ in točke f te trditve.

□

Vsebovanosti relacij lahko prikažemo tudi z naslednjim diagramom:

$$\begin{array}{ccccc}
 & & \mathbf{B} & \subseteq & \mathbf{K} \\
 & & \cup\!| & & \cup\!| \\
\vdots & & \vdots & & \vdots \\
\cup\!| & & \cup\!| & & \cup\!| \\
\mathbf{L}_k & \subseteq & \mathbf{B}_k & \subseteq & \mathbf{K}_k \\
\cup\!| & & \cup\!| & & \cup\!| \\
\mathbf{L}_{k-1} & \subseteq & \mathbf{B}_{k-1} & \subseteq & \mathbf{K}_{k-1} \\
\cup\!| & & \cup\!| & & \cup\!| \\
\vdots & & \vdots & & \vdots
\end{array}$$

5.2.2 Usmerjeni grafi

Podgrafu, ki je izomorfen usmerjenemu ciklu dolžine vsaj 2 in največ k, bomo rekli (k)-*cikel*. Podgraf $\mathcal{H}$ grafa $\mathcal{G}$ je *ciklično k-kotniški*, če vsaka njegova točka in vsaka njegova povezava pripada vsaj enemu (k)-ciklu v grafu $\mathcal{H}$.

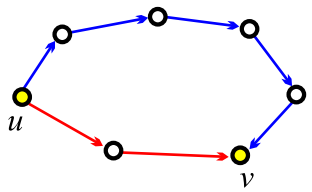
Trditev 5.18: *Šibko povezan ciklično k-kotniški graf je tudi krepko povezan.*

Dokaz: Naj bosta u in v poljubni točki šibko povezanega ciklično k-kotniškega grafa. Zaradi šibke povezanosti obstaja pot od u do v, kjer usmerjenost povezav ni pomembna. Vsaka povezava na tej poti pripada vsaj enemu (k)-ciklu.

Poiskati moramo sprehod od u do v. Ta bo sestavljen iz tistih povezav na poti od u do v, ki so pravilno usmerjene, vsako narobe usmerjeno povezavo pa nadomestimo z drugimi povezavami ustreznega (k)-cikla. Tako namesto po narobe usmerjeni povezavi gremo po več drugih povezavah, ki pa so pravilno usmerjene.

Tako lahko poiščemo sprehod med poljubnima točkama, torej je graf tudi krepko povezan. □

Za enostavne usmerjene grafe brez zank je Everett [17, 18] definiral kot *sprejemljive* samo usmerjene cikle dolžine vsaj 2 in največ k, ter cikle, sestavljene iz dveh ločenih vzporednih poti s skupno dolžino največ k – imenovali jih bomo *Everettovi semicikli*, glej sliko 5.11.

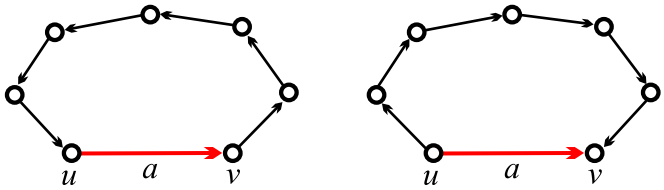


Slika 5.11: *Everettov semicikel - okrepitev z vzporedno potjo*

Definicija 5.21: Povezavo, ki pripada sprejemljivemu semiciklu, imenujemo *Everettova povezava*. Te povezave sestavljajo *Everettov graf* $\mathcal{G}_E = (\mathcal{V}, \mathcal{A}_E)$, kjer je $a(u, v) \in \mathcal{A}_E$, če je $a \in \mathcal{A}$ in je a Everettova povezava v grafu $\mathcal{G}$.

Razbitje usmerjenega grafa lahko definiramo na podoben način, kot smo to storili za neusmerjeni graf.

Posebno pozornost bomo posvetili dvema posebnima vrstama Everettovih semiciklov glede na izbrano povezavo $a \in \mathcal{A}$: *usmerjenim ciklom* (povezava s povratno potjo) in *tranzitivnim semiciklom* (povezava z vzporedno potjo), glej sliko 5.12.



povratna pot vzporedna pot

Slika 5.12: *Semicikli nad izbrano povezavo*

Definicija 5.22: Zaporedje $(\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_s)$ (k)-ciklov v $\mathcal{G}$ *(točkovno) cikl̆ično k-kotniško povezuje* točko $u \in \mathcal{V}$ s točko $v \in \mathcal{V}$, če je

1. $u \in \mathcal{V}(\mathcal{C}_1)$,
2. $v \in \mathcal{V}(\mathcal{C}_s)$ in
3. $\mathcal{V}(\mathcal{C}_{i-1}) \cap \mathcal{V}(\mathcal{C}_i) \neq \emptyset$ za $i = 2, \ldots, s$.

Takemu zaporedju pravimo *(točkovna) cikl̆ična k-kotniška veriga*.

Definicija 5.23: Točka $u \in \mathcal{V}$ je *(točkovno) cikl̆ično k-kotniško povezana* s točko $v \in \mathcal{V}$, $u\mathbf{C}_k v$, če je $u = v$ ali obstaja (točkovna) cikl̆ična k-kotniška veriga, ki (točkovno) cikl̆ično k-kotniško povezuje točko u s točko v.

Definicija 5.24: Zaporedje $(\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_s)$ (k)-ciklov v $\mathcal{G}$ *povezavno cikl̆ično k-kotniško povezuje* točko $u \in \mathcal{V}$ s točko $v \in \mathcal{V}$, če je

1. $u \in \mathcal{V}(\mathcal{C}_1)$,
2. $v \in \mathcal{V}(\mathcal{C}_s)$ in
3. $\mathcal{A}(\mathcal{C}_{i-1}) \cap \mathcal{A}(\mathcal{C}_i) \neq \emptyset$ za $i = 2, \ldots, s$.

Takemu zaporedju pravimo *povezavna cikl̆ična k-kotniška veriga*.

Definicija 5.25: Točka $u \in \mathcal{V}$ je *povezavno cikl̆ično k-kotniško povezana* s točko $v \in \mathcal{V}$, $u\mathbf{D}_k v$, če je $u = v$ ali obstaja povezavna cikl̆ična k-kotniška veriga, ki povezavno cikl̆ično k-kotniško povezuje točko u s točko v.

Za $\mathbf{C}_k$ in $\mathbf{D}_k$ veljajo podobne lastnosti, kot za $\mathbf{K}_k$ in $\mathbf{L}_k$.

Trditev 5.19: *Relacija $\mathbf{C}_k$ je ekvivalenčna relacija na množici točk $\mathcal{V}$.*

Dokaz: Refleksivnost sledi iz definicije relacije $\mathbf{C}_k$.

Če ciklično k-kotniško verigo od u do v obrnemo, dobimo ciklično k-kotniško verigo od v do u, torej je relacija $\mathbf{C}_k$ simetrična.

Še tranzitivnost. Naj bodo u, v in z takšne točke, da je $u\mathbf{C}_k v$ in $v\mathbf{C}_k z$. Če točke niso paroma različne, je pogoj za tranzitivnost trivialno izpolnjen. Predpostavimo torej, da so točke u, v in z paroma različne. Ker je $u\mathbf{C}_k v$, obstaja ciklična k-kotniška veriga od u do v, ker pa je $v\mathbf{C}_k z$, obstaja tudi ciklična k-kotniška veriga od v do z. Če ti dve verigi staknemo, dobimo ciklično k-kotniško verigo od u do z, torej je $u\mathbf{C}_k z$. □

Trditev 5.20: *Relacija $\mathbf{D}_k$ določa ekvivalenčno relacijo na množici povezav $\mathcal{A}$.*

Dokaz: Naj bo $\sim$ relacija na $\mathcal{A}$, ki je definirana takole: $e \sim f$, če je $e = f$ ali pa obstaja povezavna ciklična k-kotniška veriga $(\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_s)$, kjer je $e \in \mathcal{A}(\mathcal{C}_1)$ in $f \in \mathcal{A}(\mathcal{C}_s)$.

Refleksivnost relacije $\sim$ sledi iz njene definicije.

Tudi simetričnost je preprosta. Naj bo $e \sim f$. Potem obstaja povezavna ciklična k-kotniška veriga 'od' e 'do' f. Če jo obrnemo, dobimo povezavno ciklično k-kotniško verigo 'od' f 'do' e, torej je $f \sim e$.

Še tranzitivnost. Naj bodo e, f in g takšne povezave, da je $e \sim f$ in $f \sim g$. Potem obstajata povezavni ciklični k-kotniški verigi 'od' e 'do' f in 'od' f 'do' g. Če ti dve verigi staknemo, dobimo povezavno ciklično k-kotniško verigo 'od' e 'do' g (oba (k)-cikla na stiku verig vsebujeta povezavo f, torej njun presek ni prazen). Zato je tudi $e \sim g$. □

Definicija 5.26: Povezava je *ciklična*, če pripada vsaj enemu usmerjenemu ciklu v grafu $\mathcal{G}$. Dolžina cikla pri tem ni pomembna. Ciklični povezavi, ki ne pripada nobenemu (k)-ciklu, rečemo *k-dolga* povezava.

Trditev 5.21: *Če graf $\mathcal{G} = (\mathcal{V}, \mathcal{A})$ ne vsebuje nobene k-dolge povezave, potem je njegova ciklična k-kotniška redukcija $\mathcal{G}/\mathbf{C}_k = (\mathcal{V}/\mathbf{C}_k, \mathcal{A}^*)$, kjer za $X, Y \in \mathcal{V}/\mathbf{C}_k$ velja $(X, Y) \in \mathcal{A}^* \iff \exists u \in X\ \exists v \in Y : (u, v) \in \mathcal{A}$, acikličen graf.*

Dokaz: Predpostavimo, da ciklična k-kotniška redukcija grafa $\mathcal{G}$ ni aciklična. Potem vsebuje cikel C^*, ki ga lahko razširimo do cikla C v grafu $\mathcal{G}$. Naj bo a^* poljubna povezava cikla C^* in naj bo a ustrezna povezava cikla C. Ker sta krajišči povezave a^* dve različni točki, krajišči povezave a pripadata dvema različnima komponentama relacije $\mathbf{C}_k$. Povezava a torej ne pripada nobenemu cikličnemu (k)-kotniku. Ker pa a pripada ciklu C, je to ciklična povezava, torej je k-dolga. To pa je protislovje s predpostavko, da graf ne vsebuje k-dolgih povezav. Torej mora biti ciklična k-kotniška redukcija grafa $\mathcal{G}$ acikličen graf. □

Iz tega dokaza vidimo, kako v grafu poiskati k-dolge povezave. To so natanko tiste povezave, ki ustrezajo cikličnim povezavam v $\mathcal{G}/\mathbf{C}_k$.

Definicija 5.27: Točka $u \in \mathcal{V}$ je *krepko k-kotniško povezana* s točko $v \in \mathcal{V}$, $u\mathbf{S}_k v$, če je $u = v$ ali obstaja krepko povezan k-kotniški podgraf, ki vsebuje točki u in v.

Trditev 5.22: *V usmerjenem grafu $\mathcal{G}$ velja:*

a. $\mathbf{D}_k \subseteq \mathbf{C}_k$ *b.* $\mathbf{C}_k \subseteq \mathbf{S}_k$ *c.* $\mathbf{S}_k \subseteq \mathbf{S}$

za $i < j$ pa velja tudi:

d. $\mathbf{C}_i \subseteq \mathbf{C}_j$ *e.* $\mathbf{D}_i \subseteq \mathbf{D}_j$ *f.* $\mathbf{S}_i \subseteq \mathbf{S}_j$

DOKAZ:

a. Naj bosta u in v takšni točki, da je $u\mathbf{D}_k v$. Če je $u = v$, je po definiciji tudi $u\mathbf{C}_k v$, če pa sta točki različni, obstaja povezavna ciklična k-kotniška veriga od u do v, vsaka povezavna ciklična k-kotniška veriga pa je tudi točkovna ciklična k-kotniška veriga (če imata dva k-cikla skupno povezavo, imata tudi skupno točko). Zato je $u\mathbf{C}_k v$.

b. Naj bosta u in v takšni točki, da je $u\mathbf{C}_k v$. Če je $u = v$, je po definiciji tudi $u\mathbf{S}_k v$, če pa sta točki različni, obstaja ciklična k-kotniška veriga od u do v. Podgraf, ki ga določa ta veriga, je krepko povezan in k-kotniški, torej je tudi $u\mathbf{S}_k v$.

c. Naj bosta u in v takšni točki, da je $u\mathbf{S}_k v$. Če je $u = v$, je po definiciji tudi $u\mathbf{S}v$, če pa sta točki različni, obstaja krepko povezan k-kotniški podgraf, ki vsebuje u in v. V krepko povezanem grafu pa so vse točke dosegljive ena iz druge, torej je $u\mathbf{S}v$.

d. Naj bosta u in v taki točki, da je $u\mathbf{C}_i v$. Če je $u = v$, potem je po definiciji tudi $u\mathbf{C}_j v$. Sicer pa obstaja ciklična i-kotniška veriga od u do v, kjer noben (i)-cikel ni daljši od i. Ta veriga je tudi ciklična j-kotniška veriga od u do v, torej je $u\mathbf{C}_j v$.

e. Naj bosta u in v taki točki, da je $u\mathbf{D}_i v$. Če je $u = v$, potem je po definiciji tudi $u\mathbf{D}_j v$. Sicer pa obstaja povezavna ciklična i-kotniška veriga od u do v, kjer noben (i)-cikel ni daljši od i. Ta veriga je tudi povezavna ciklična j-kotniška veriga od u do v, torej je $u\mathbf{D}_j v$.

f. Naj bosta u in v taki točki, da je $u\mathbf{S}_i v$. Če je $u = v$, potem je po definiciji tudi $u\mathbf{S}_j v$. Sicer pa obstaja krepko povezan i-kotniški podgraf, ki vsebuje u in v. Ta podgraf je tudi j-kotniški, torej je $u\mathbf{S}_j v$.

□

Vsebovanosti relacij lahko prikažemo tudi z naslednjim diagramom:

$$\begin{array}{ccccc}
 & & & & \mathbf{S} \\
 & & & & \cup\!| \\
\vdots & & \vdots & & \vdots \\
\cup\!| & & \cup\!| & & \cup\!| \\
\mathbf{D}_k & \subseteq & \mathbf{C}_k & \subseteq & \mathbf{S}_k \\
\cup\!| & & \cup\!| & & \cup\!| \\
\mathbf{D}_{k-1} & \subseteq & \mathbf{C}_{k-1} & \subseteq & \mathbf{S}_{k-1} \\
\cup\!| & & \cup\!| & & \cup\!| \\
\vdots & & \vdots & & \vdots
\end{array}$$

Podobno kot v trikotniškem primeru lahko definiramo več vrst omrežij, ki nam omogočijo boljši pregled nad zgradbo omrežja:

- *Povratno omrežje* $\mathcal{N}_k^{cyc} = (\mathcal{V}, \mathcal{A}_k^{cyc}, w_k^{cyc})$, kjer je $w_k^{cyc}(a)$ enako številu različnih (k)-ciklov, ki vsebujejo povezavo a.
- *Tranzitivno omrežje* $\mathcal{N}_k^{T} = (\mathcal{V}, \mathcal{A}_k^{T}, w_k^{T})$, kjer je $w_k^{T}(a)$ enako številu različnih tranzitivnih semiciklov dolžine največ k, ki vsebujejo povezavo a kot tranzitivno povezavo (bližnjico).
- *Podporno omrežje* $\mathcal{N}_k^{S} = (\mathcal{V}, \mathcal{A}_k^{S}, w_k^{S})$, kjer je $w_k^{S}(a)$ enako številu različnih tranzitivnih semiciklov dolžine največ k, ki vsebujejo povezavo a kot netranzitivno povezavo.

Trditev 5.23:

$$\mathbf{C}_k(\mathcal{G}) = \mathbf{S}(\mathcal{G}_k^{cyc})$$

Dokaz: Naj bo $u\mathbf{C}_k v$ v grafu $\mathcal{G}$. Če je $u = v$, je tudi $u\mathbf{S}v$ v grafu $\mathcal{G}_k^{cyc}$. Če pa sta točki različni, obstaja ciklična k-kotniška veriga v $\mathcal{G}$ od u do v. Vsaka povezava na tej verigi pripada vsaj enemu (k)-ciklu, torej je cela veriga v $\mathcal{G}_k^{cyc}$. Točki u in v sta medsebojno dosegljivi po povezavah te verige, torej je $u\mathbf{S}v$ v grafu $\mathcal{G}_k^{cyc}$.

Naj bo $u\mathbf{S}v$ v grafu $\mathcal{G}_k^{cyc}$. Potem v $\mathcal{G}_k^{cyc}$ obstaja sprehod od u do v. Ker je $\mathcal{G}_k^{cyc}$ ciklično k-kotniški, vsaka povezava pripada vsaj enemu (k)-ciklu, torej lahko sestavimo ciklično k-kotniško verigo od u do v, ki v celoti leži v $\mathcal{G}_k^{cyc}$. Ker pa je $\mathcal{G}_k^{cyc}$ podgraf grafa $\mathcal{G}$, leži ta veriga tudi v $\mathcal{G}$, kar pomeni, da velja $u\mathbf{C}_k v$ v grafu $\mathcal{G}$. □

5.2.3 Tranzitivnost

Naj bo $\mathbf{T}_k$ relacija *k-tranzitivne dosegljivosti* v usmerjenem grafu $\mathcal{G}$. Točka v je k-tranzitivno dosegljiva iz točke u, $u\mathbf{T}_k v$, če je $u = v$ ali pa obstaja sprehod od u do v, v katerem je vsaka povezava k-tranzitivna – je osnova tranzitivnega cikla dolžine največ k.

Točki u in v sta medsebojno k-tranzitivno dosegljivi, če je točka u k-tranzitivno dosegljiva iz točke v in obratno. Relacijo medsebojne k-tranzitivne dosegljivosti bomo označili s $\hat{\mathbf{T}}_k$. Torej

$$u\hat{\mathbf{T}}_k v \iff u\mathbf{T}_k v \wedge v\mathbf{T}_k u$$

Trditev 5.24: *Relacija medsebojne tranzitivne dosegljivosti* $\hat{\mathbf{T}}_k = \mathbf{T}_k \cap \mathbf{T}_k^{-1}$ *je ekvivalenčna relacija na množici točk* $\mathcal{V}$.

Dokaz: Znano je, da če je $\mathbf{Q}$ refleksivna in tranzitivna relacija, potem je $\hat{\mathbf{Q}} = \mathbf{Q} \cap \mathbf{Q}^{-1}$ ekvivalenčna relacija. Relacija $\mathbf{T}_k$ je refleksivna po definiciji, torej moramo dokazati samo še tranzitivnost.

Naj bodo u, v in z take točke, da je $u\mathbf{T}_k v$ in $v\mathbf{T}_k z$. Če točke niso paroma različne, je pogoj za tranzitivnost trivialno izpolnjen. Predpostavimo torej, da so točke u, v in z paroma različne. Ker je $u\mathbf{T}_k v$, obstaja sprehod od u do v, v katerem je vsaka povezava k-tranzitivna, ker pa je $v\mathbf{T}z$, obstaja tudi sprehod od v do z, v katerem je vsaka povezava k-tranzitivna. Če ta dva sprehoda staknemo, dobimo sprehod od u do z, v katerem je vsaka povezava k-tranzitivna, torej je $u\mathbf{T}_k z$. □

5.3 Možne posplošitve

V do sedaj prikazanih primerih so bile verige vedno sestavljene iz točno določenih gradnikov (trikotniki, k-kotniki z omejeno dolžino), presek dveh zaporednih gradnikov pa je bila točka ali povezava. To lahko posplošimo tako, da definiramo družino dopustnih gradnikov in družino dopustnih presekov.

Definicija 5.28: Naj bosta $\mathbb{H}$ in $\mathbb{H}_0$ dve družini grafov. Zaporedje $(\mathcal{H}_1, \mathcal{H}_2, \ldots, \mathcal{H}_s)$ podgrafov v $\mathcal{G}$ $(\mathbb{H}, \mathbb{H}_0)$-*povezuje* točko $u \in \mathcal{V}$ s točko $v \in \mathcal{V}$, če je

1. $u \in \mathcal{V}(\mathcal{H}_1)$,
2. $v \in \mathcal{V}(\mathcal{H}_s)$,
3. $\mathcal{H}_i \in \mathbb{H}$ za $i = 1, \ldots, s$ in
4. $\mathcal{H}_{i-1} \cap \mathcal{H}_i \supseteq \mathcal{H} \in \mathbb{H}_0$ za $i = 2, \ldots, s$.

Takemu zaporedju pravimo $(\mathbb{H}, \mathbb{H}_0)$-*veriga*.

Primer: Za $r < k$ lahko definiramo (k, r)-povezanost s klikami:

$$\mathbb{H} = \{K_{r+1}, K_{r+2}, \ldots, K_k\} \qquad \mathbb{H}_0 = \{K_r\}$$

Primer: Nekatere od prej opisanih povezanosti bi lahko definirali tudi takole:

- $\mathbf{K}_3 = (3, 1)$-povezanost s klikami
- $\mathbf{L}_3 = (3, 2)$-povezanost s klikami
- $\mathbf{K}_k = (\{C_3, \ldots, C_k\}, \{K_1\})$-povezanost
- $\mathbf{L}_k = (\{C_3, \ldots, C_k\}, \{K_2\})$-povezanost

kjer C_i označuje cikel dolžine i.

5.4 Primeri

Na spletu je dosegljivih veliko slovarjev, kjer je vsak pojem razložen z drugimi pojmi iz slovarja. Taki so na primer slovarji Online Dictionary of Library and Information Science (ODLIS) [34], Free Online Dictionary of Computing (FOLDOC) [20], Dictionary of Visual Art [1], Online Dictionary of Musical terms [33], Project Gutenberg [39], Connected Thesaurus [21], ...

Na sliki 5.13 je prikazano, kako so definirani posamezni pojmi v slovarju ODLIS. Iz takih definicij pojmov lahko sestavimo usmerjen graf $\mathcal{G} = (\mathcal{V}, \mathcal{A})$, katerega točke so pojmi, ki so definirani v slovarju, dva pojma pa sta povezana z usmerjeno povezavo, če pri definiciji prvega pojma uporabimo drugi pojem. Na sliki 5.13 vidimo, da je $\{\text{note area}, \text{area}, \text{catalog record}, \text{work}, \text{notebook}, \text{loose-leaf}, \text{spiral}\} \subset \mathcal{V}$, $(\text{note area}, \text{contents}) \in \mathcal{A}$ in $(\text{notebook}, \text{cover}) \in \mathcal{A}$.

Graf ODLIS ima 2909 točk (definiranih pojmov) in 18419 usmerjenih povezav, med katerimi je tudi 5 zank (book, database, leaf, paper, subject). Povprečna stopnja je 6.33. Graf vsebuje 11 šibko povezanih komponent, med katerimi je 9 izoliranih točk, ena komponenta vsebuje dve točki (use life, shelf life), vse ostale točke pa so vsebovane v isti šibko povezani komponenti (2898 točk).

Če v grafu ODLIS za vsako povezavo preštejemo, kolikim cikličnim trikotnikom pripada, dobimo uteži na povezavah. Povezavni prerez na nivoju 7 nam da omrežje, prikazano na sliki 5.14. Podobno lahko za vsako povezavo preštejemo, kolikim tranzitivnim trikotnikom pripada. Povezavni prerez dobljenega omrežja na nivoju 11 je prikazan na sliki 5.15.

Slovar FOLDOC je precej večji. Vsebuje 13425 definiranih pojmov, a preden se lotimo analize, moramo še počisti nekaj stvari. Najprej pobrišemo pojma Jargon

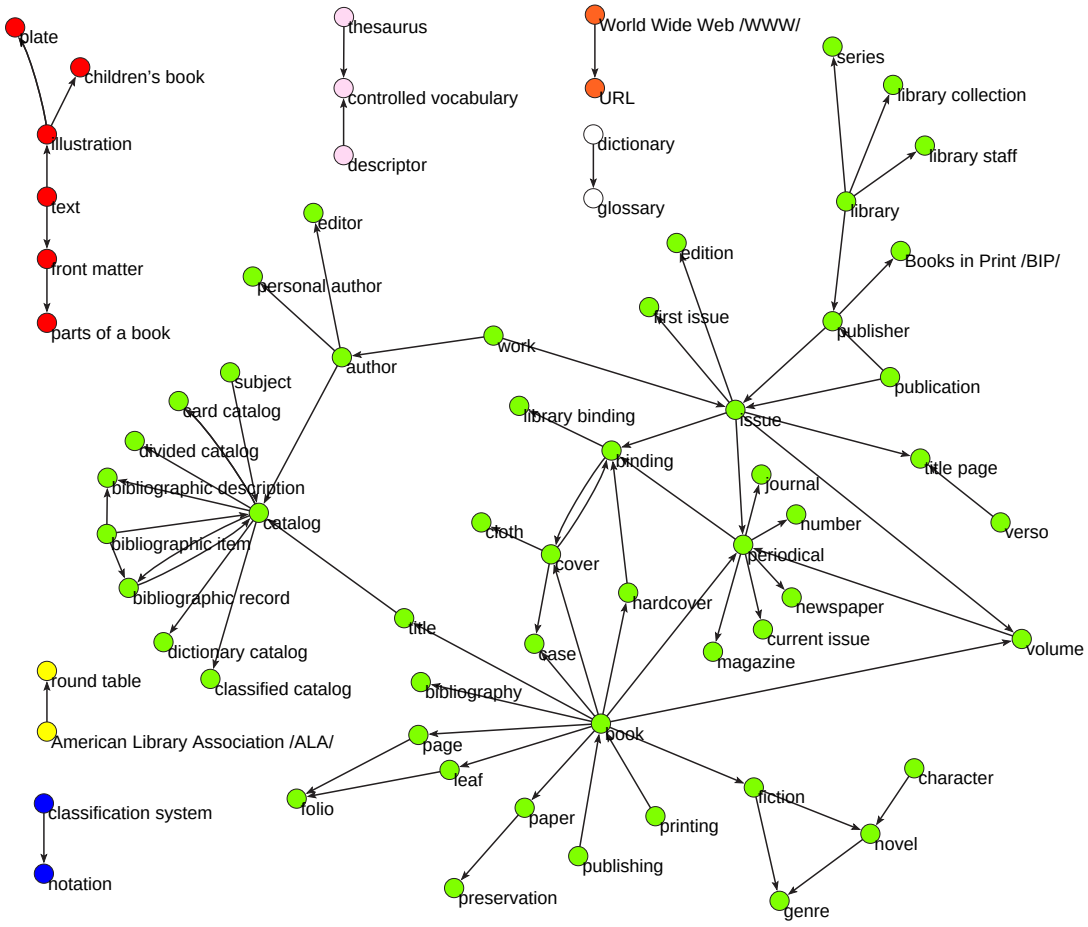
note area
The area of a catalog record following the physical description which gives the contents of the work, its relationship to other works, and any physical characteristics not included in preceding parts of the bibliographic description. Each note is given a separate paragraph.

notebook
A loose-leaf or spiral binder with flexible or inflexible cardboard or plastic covers, usually filled with blank ruled or unruled leaves for taking notes and/or filing syllabi, reading lists, assignments, and other material pertaining to a specific project or course of study. Also synonymous with laptop computer.

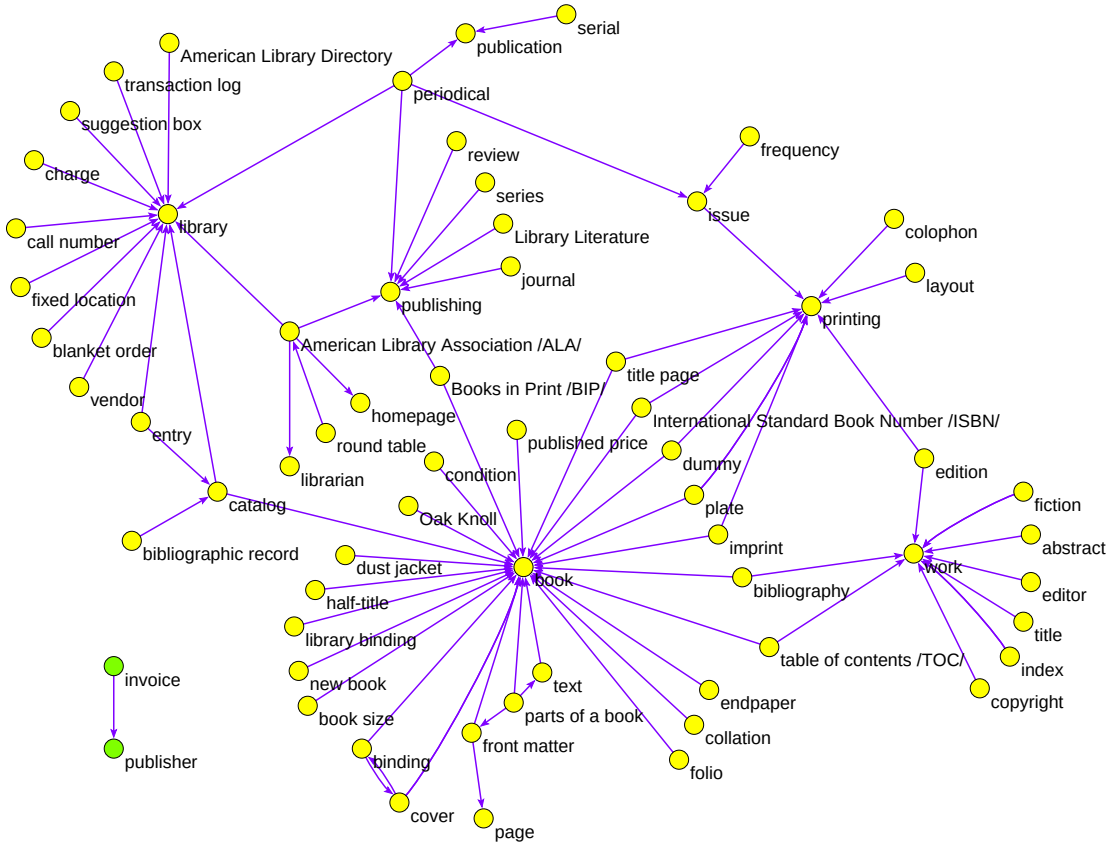
Slika 5.13: Definicija pojmov v slovarju ODLIS

File in TLAs (Three Letter Abbraviations). V slovarju sta zelo pogosta, nikjer pa nista razložena. Večji problem imamo z različnimi pojmi, ki imajo isti pomen, na primer Intel486, 486 in i486. Potrebnega je kar nekaj dela za identifikacijo vseh takih pojmov, ko pa dobimo ekvivalenčne razrede, lahko omrežje skrčimo. Tako dobimo graf na 13356 točkah, ki vsebuje 120238 usmerjenih povezav (nobene zanke). Povprečna stopnja grafa FOLDOC je 9.003.

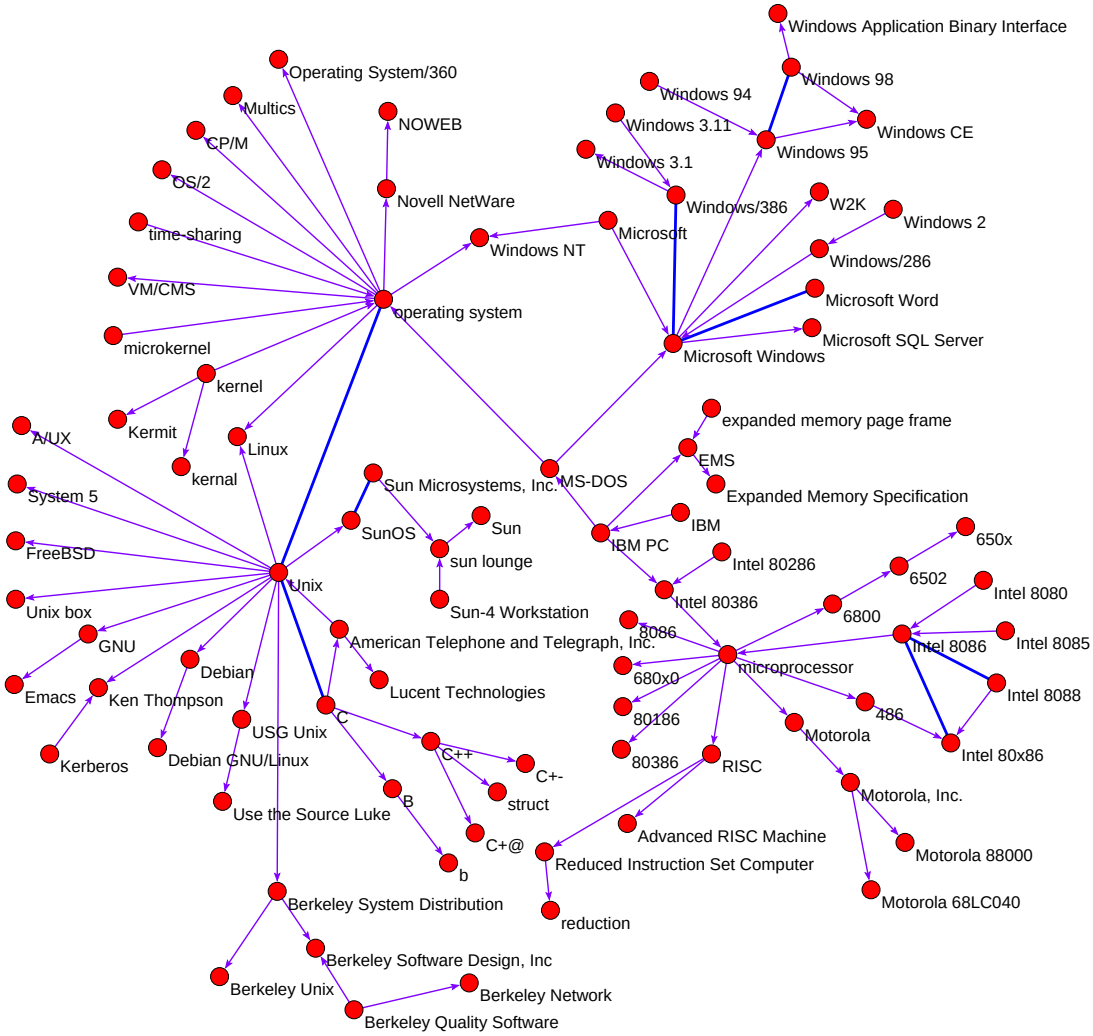
S prerezi v cikličnem trikotniškem omrežju grafa FOLDOC dobimo več manjših komponent. Na nivoju 5 dobimo tudi nekaj večjih komponent. Najpomembnejše so operacijski sistemi in procesorji (92 točk, glej sliko 5.16), internet (19), Commodore (13), Motorola (12), java (10), in socket (10). Podobne rezultate nam da tudi povezavni prerez v tranzitivnem trikotniškem omrežju grafa FOLDOC na nivoju 7. Največje komponente so operacijski sistemi, Unix in sorodni pojmi, programski jeziki (83), procesorji (20), standardi in protokoli (12), procesor 386 (12) in java (12).



Slika 5.14: *Povezave, ki ležijo na vsaj 7 cikličnih trikotnikih*



Slika 5.15: *Povezave, ki ležijo na vsaj 11 tranzitivnih trikotnikih*



Slika 5.16: *Največja komponenta cikličnega trikotniškega omrežja na nivoju 5*

Šesto poglavje

Zaključek

Delo vsebuje najprej kratek pregled že znanih postopkov za določanje razčlemb danega omrežja. Na tem področju je bilo narejenega že precej, a veliko postopkov je prezahtevnih, da bi jih lahko uporabili tudi na zelo velikih omrežjih (več stotisoč točk in povezav). Pri lastnem raziskovalnem delu sem se zato osredotočil na iskanje učinkovitih postopkov, ki dajo v sprejemljivem času (največ nekaj minut) smiselno razčlembo tudi za velika omrežja. Ti rezultati so predstavljeni v poglavjih o otokih, sredicah in povezanosti s kratkimi cikli.

Otoki so (poleg krepke, šibke in dvo-povezanosti) eden redkih konceptov, za katere obstajajo učinkoviti postopki, ki dajo smiselne rezultate. Otoke lahko določamo v omrežjih, ki imajo vrednosti v točkah ali pa na povezavah. Če vrednosti nimamo, jih moramo prej smiselno določiti iz omrežja samega, odvisno od tega, katera lastnost omrežja nas zanima.

Za primer si poglejmo omrežje sklicevanj, kjer so točke različni članki ali knjige, povezave pa opisujejo relacija sklicevanja (citiranja). V takem omrežju želimo določiti skupine člankov, ki obravnavajo podobne teme. Omrežje sklicevanj je običajno brez zank in je skoraj aciklično. Ker so krepke komponente običajno zelo majhne, lahko aciklično omrežje dobimo tako, da vsako krepko komponento stisnemo v eno samo točko in pobrišemo vse zanke.

Eden od možnih načinov, kako določiti uteži povezav v omrežju sklicevanj, je opisan v [3]. Utež na izbrani povezavi definiramo kot število vseh različnih poti, ki gredo skozi to povezavo (Search Path Count). Pomembni citati so tiste povezave, ki ležijo na velikem številu poti, ker pa so v člankih običajno citati na članke na podobno temo, bodo povezavni otoki določali skupino pomembnih člankov, ki vsi obravnavajo podobne teme.

Poleg nekaterih znanih točkovnih in povezavnih funkcij, ki so bile naštete že v uvodnem poglavju, smo vpeljali še nekaj takih funkcij: središčna števila, število ciklov omejene dolžine, ki vsebujejo dano povezavo ali točko ...

Pričakujemo, da se da na osnovi središčnih števil razviti učinkovite pristope k reševanju drugih problemov na velikih omrežij zgraditi na tej osnovi. Na primer,

zaporedje točk pri zaporednem barvanju lahko določimo iz padajočega zaporedja njihovih središčnih števil (kombinirano z njihovimi stopnjami). Na tej osnovi dobimo naslednjo mejo za barvnost danega grafa $\mathcal{G}$:

$$\chi(\mathcal{G}) \leq 1 + \mathrm{core}(\mathcal{G})$$

S pomočjo sredic lahko tudi omejimo iskanje drugih zanimivih podomrežij v velikih omrežjih [7, 5]. Pri tem si pomagamo z naslednjima ugotovitvama.

- Če obstaja, je k-komponenta vsebovana v k-sredici.
- Če obstaja, je k-klika vsebovana v k-sredici: $\omega(\mathcal{G}) \leq \mathrm{core}(\mathcal{G})$.

Sredice lahko povežemo tudi s točkovnimi otoki, tako da središčna števila uporabimo kot vrednosti točk.

Pri povezanosti s kratkimi cikli smo opisali učinkovite postopke za določanje števila trikotnikov različnih vrst, ki vsebujejo dano povezavo. Splošnejši postopek, ki bi za vsako povezavo ali točko preštel, v koliko (k)-kotnikih je vsebovana, ostaja odprt problem. Zahtevnost problema lahko zmanjšamo tako, da se omejimo na 2-povezane komponente grafa. Manjšo zahtevnost postopka pričakujemo tudi v primeru, če nas namesto natančne vrednosti zanima samo, ali je ta vrednost večja od 0. Izračun točnih vrednosti bi lahko potem omejili samo na omrežje, iz katerega odstranimo vse povezave z vrednostjo 0.

Predvidevamo, da so otoki močno orodje za analizo velikih omrežij in da bodo dobro sprejeti med raziskovalci z najrazličnejših področij.

Literatura

[1] ArtLex, *Dictionary of visual art*, 2002.
http://www.artlex.com/

[2] V. Batagelj, *Razvrščanje v skupine – nehierarhični postopki*, magistrsko delo, Fakulteta za elektrotehniko, Ljubljana, 1985.

[3] V. Batagelj, *Efficient algorithms for citation network analysis*, poslano v objavo, 2003.
http://arxiv.org/abs/cs.DS/0309023

[4] V. Batagelj, A. Mrvar, *Pajek – A program for large network analysis*, Connections **21** (1998), št. 2, 47–57.
http://vlado.fmf.uni-lj.si/pub/networks/pajek/

[5] V. Batagelj, A. Mrvar, *Some analyses of Erdős collaboration graph*, Social Networks **22** (2000), št. 2, 173–186.

[6] V. Batagelj, A. Mrvar, *Pajek - Analysis and visualization of large networks*, Graph Drawing Software (M. Jünger, P. Mutzel, ur.), Springer, Berlin, 2003, str. 77–103.

[7] V. Batagelj, A. Mrvar, M. Zaveršnik, *Partitioning approach to visualization of large graphs*, Graph drawing (J. Kratochvíl, ur.), Lecture Notes in Computer Science, št. 1731, Springer, Berlin, 1999, str. 90–97.

[8] V. Batagelj, M. Zaveršnik, *Generalized cores*, poslano v objavo, 2002.
http://arxiv.org/abs/cs.DS/0202039

[9] V. Batagelj, M. Zaveršnik, *An $\mathcal{O}(m)$ algorithm for cores decomposition of networks*, poslano v objavo, 2002.
http://arxiv.org/abs/cs.DS/0310049

[10] V. Batagelj, M. Zaveršnik, *Short cycles connectivity*, poslano v objavo, 2003.
http://arxiv.org/abs/cs.DS/0308011

[11] N. H. F. Beebe, *Nelson H. F. Beebe's Bibliographies Page*, 2002.
http://www.math.utah.edu/~beebe/bibliographies.html

[12] F. R. K. Chung, *Spectral graph theory*, CBMS Lecture Notes, št. 92, AMS Publications, 1997.

[13] P. Crescenzi, V. Kann, *A compendium of NP optimization problems.*
http://www.nada.kth.se/~viggo/wwwcompendium/wwwcompendium.html

[14] E. Dahlhaus, J. Gustedt, R. M. McConnell, *Efficient and practical algorithms for sequential modular decomposition*, Journal of Algorithms **41** (2001), št. 2, 360–387.
http://www.cs.colostate.edu/~rmm/linearDecompJournal.ps

[15] R. Diestel, *Graph theory*, 2. izd., Graduate Texts in Mathematics, št. 173, Springer-Verlag, New York, 2000.
http://www.math.uni-hamburg.de/home/diestel/books/graph.theory/

[16] M. G. Everett, *A graph theoretic blocking procedure for social networks*, Social Networks **4** (1982), 147–167.

[17] M. G. Everett, *EBLOC: A graph theoretic blocking algorithm for social networks*, Social Networks **5** (1983), 323–346.

[18] M. G. Everett, *An extension of EBLOC to valued graphs*, Social Networks **5** (1983), 395–402.

[19] C. Farhat, *A simple and efficient automatic FEM domain decomposer*, Computers and Structures **28** (1988), št. 5, 579–602.

[20] FOLDOC, *Free on-line dictionary of computing*, 2002.
http://wombat.doc.ic.ac.uk/foldoc/

[21] Lexical FreeNet, *Connected Thesaurus*, 2002.
http://www.lexfn.com/

[22] M. R. Garey, D. S. Johnson, *Computers and intractability: A guide to the theory of NP-completness*, W. H. Freeman and Company, San Francisco, 1979.

[23] G. H. Golub, C. F. van Loan, *Matrix computations*, 3. izd., The Johns Hopkins University Press, Baltimore and London, 1996.

[24] D. S. Hochbaum, A. Pathria, *The bottleneck graph partition problem*, Networks **28** (1996), št. 4, 221–225.

[25] B. Jones, *Computational Geometry Database*, februar 2002.
http://compgeom.cs.uiuc.edu/~jeffe/compgeom/biblios.html
ftp://ftp.cs.usask.ca/pub/geometry/

[26] G. Karypis, *METIS.*
http://www-users.cs.umn.edu/~karypis/metis/

[27] G. Karypis, V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM Journal on Scientific Computing **20** (1998), št. 1, 359–392.

[28] G. Karypis, V. Kumar, *Multilevel algorithms for multi-constraint graph partitioning*, Technical Report TR 98-019, Department of Computer Science, University of Minnesota, 1998.

[29] G. Karypis, V. Kumar, *Multilevel k-way partitioning scheme for irregular graphs*, Journal of Parallel and Distributed Computing **48** (1998), št. 1, 96–129.

[30] B. Kernigan, S. Lin, *An efficient heuristic procedure for partitioning graphs*, The Bell System Technical Journal **29** (1970), št. 2, 291–307.

[31] D. E. Knuth, *Dictionaries of English words.*
ftp://labrea.stanford.edu/pub/dict/

[32] R. M. McConnell, J. P. Spinrad, *Linear-time modular decomposition and efficient transitive orientation of comparability graphs*, Proc. 5th ACM-SIAM Symp. Discrete Algorithms, 1994, str. 536–545.

[33] Creative Music, *Online dictionary of musical terms*, 2002.
http://www.creativemusic.com/features/dictionary.html

[34] ODLIS, *Online dictionary of library and information science*, 2002.
http://vax.wcsu.edu/library/odlis.html

[35] W. Richards, A. Seary, *Partitioning Networks by Eigenvectors.*
http://www.sfu.ca/~richards/Pages/london98.pdf

[36] W. Richards, A. Seary, *Spectral methods for analyzing and visualizing networks: an introduction.*
http://www.sfu.ca/~richards/Pages/NAS.AJS-WDR.pdf

[37] S. B. Seidman, *Network structure and minimum degree*, Social Networks **5** (1983), 269–287.

[38] H. D. Simon, S.-H. Teng, *How good is recursive bisection*, SIAM Jounal on Scientific Computing **18** (1997), št. 5, 1436–1445.
http://www-sal.cs.uiuc.edu/~steng/horst.ps

[39] Thesaurus, *Project Gutenberg*, 2002.
ftp://ibiblio.org/pub/docs/books/gutenberg/etext02/mthes10.zip

[40] D. J. Watts, S. H. Strogatz, *Collective dynamics of 'small-world' networks*, Nature **393** (1998), št. 6684, 440–442.

[41] R. J. Wilson, J. J. Watkins, *Uvod v teorijo grafov*, Sigma, št. 63, DMFA Slovenije, Ljubljana, 1997.

Izjava

Izjavljam, da predložena disertacija predstavlja rezultate lastnega znanstveno raziskovalnega dela.

Ljubljana, oktober 2003 Matjaž Zaveršnik